Max Kesselbacher

Supporting the Acquisition of Programming Skills with Program Construction Patterns

DOCTORAL THESIS

submitted in fulfillment of the requirements for the degree of

Doktor der Naturwissenschaften

Universität Klagenfurt
Fakultät für Technische Wissenschaften

Supervisor:
Univ.-Prof. Dipl.-Ing. Dr. Andreas Bollin
Universität Klagenfurt
Institut für Informatikdidaktik

Evaluator:
Prof. Dr. Marc Berges
Friedrich-Alexander Universität Erlangen-Nürnberg
Department Informatik

Klagenfurt, August/2021

AFFIDAVIT

I hereby declare in lieu of an oath that

- the submitted academic paper is entirely my own work and that no auxiliary materials have been used other than those indicated,
- I have fully disclosed all assistance received from third parties during the process of writing the thesis, including any significant advice from supervisors,
- any contents taken from the works of third parties or my own works that have been included either literally or in spirit have been appropriately marked and the respective source of the information has been clearly identified with precise bibliographical references (e.g. in footnotes),
- to date, I have not submitted this paper to an examining authority either in Austria or abroad and that
- when passing on copies of the academic thesis (e.g. in bound, printed or digital form), I will ensure that each copy is fully consistent with the submitted digital version.

I am aware that a declaration contrary to the facts will have legal consequences.

Max Kesselbacher m.p. Klagenfurt, August 2021

Dedication

I use this space to express my gratitude to people that were important to me during the completion of this thesis. I want to thank my colleagues at the Department of Informatics Didactics. First and foremost, I want to thank Andreas Bollin as head of department and also Barbara Sabitzer for believing in me, providing me the opportunities to begin working at the department, and, finally, making it possible for me to contribute to research with this thesis. I also want to thank Andreas as my supervisor for all the support in research and personal matters, for all challenging comments and nudges while still letting me go down my own small rabbit holes, and generally for all opportunities to grow as a researcher. Next, I want to thank my Ph.D. and post-doc colleagues for all open ears, thoughtful discussions, and support throughout the years in a mutual polishing of our research. My biggest thanks in this regard go to Stefan for his continuous support and also to Elisa and Corinna for their support and involvement in the beginning drafts of my research. Lastly, I want to thank all other colleagues of the department for their full support on numerous occasions: Lukas and Christian for technical support, the Informatik-Werkstatt team of Alexandra, Katharina, Markus, Nina, and Philipp for their support in my research studies, and also many thanks to Annette, Marlene, and Tatjana for taking clerical work off my desk. I would also like to thank all researchers that I have met at conferences and workshops, with many encounters contributing to my growth as a researcher.

Without personal support from my family and friends, I might not have had the strength to complete this thesis. I want to thank my parents, Doris and Heimo, for providing me with countless possibilities for my education, for giving me perpetual support in me and my abilities, and for being the best parents I could have ever wished for – thank you, and I love you! Thanks go to my friends and sports colleagues for taking my mind off of the thesis for some periods so that I could be productive when it counted. At last, my thanks for personal support go to my beloved Sabine, who served as a catalyst of important ideas during my research and writing, endured quite some days and nights of my mind being not at work but not at home either, and still always supported me in focusing on finishing my thesis. I know there was a lot of borrowed time, and I will make up for it. I love you!

ABSTRACT

Digital technologies are spreading wide and fast, encompassing many aspects of our daily life. This also affects education, with programming becoming incorporated in many K–12 educational standards worldwide. This early and diverse programming education imposes a challenge on traditional programming education. It necessitates the development of theory, methods, and tools to support the acquisition of programming skills on an individual basis to best facilitate learning in heterogeneous groups of programmers.

Advancements in programming education can be initiated from two perspectives. The research field of computer science education offers a rich body to support novice programmers in learning to program. The research field of software engineering has a great collection of results regarding the differences between novice programmers and expert programmers, culminating in the effort to characterize expert programming skills. Notably, there is a gap between these two perspectives, which is the adequate and individual support of programmers that have overcome the first struggles of novice programmers but have not developed into experts either.

My research work, presented in this thesis, draws inspiration from two sources to work towards closing the identified gap. First is the idea that architectural design patterns of programs constitute *distilled* expert programming skills and that patterns of program construction could provide the same. Second is the rising field of learning analytics that provides the technical and methodological means to research programming data on a new scale.

Combining these inspirations, my research encompasses a detailed investigation of the sequential process of program construction of individual programmers. The assumption that drives my research is that expert programmers follow ***specific patterns*** in the ***sequential construction of their programs*** and that such patterns have regular, discernible forms that can be elicited with learning analytics approaches and subsequently utilized to improve programming education.

To study this assumption, I developed a learning analytics system that consists of IDE-based instrumentation tools on client-side and a uniform data collection server. With the help of this system, it is possible to record and store snapshots of block-based (Scratch 2, Scratch 3) and text-based (IntelliJ, Java) programs during their sequential creation on the granularity of key strokes, and later analyze them on the granularity of key strokes or compilations.

I define specific variable-based program construction patterns as *a typed sequence of compilable program changes that affect related program statements of a variable* to assess whether and how expert programmers construct their programs differently. I conduct a mixed methods study to identify patterns used by student programmers and professional programmers and conduct a comparative study to report on significant and substantial differences in the use of patterns between programmers of different skill levels.

My results provide evidence that program construction patterns can indeed capture expert programming skills. Specifically, my comparative results show with significance that, while the use of individual patterns is more dependent on the specific algorithmic problem to be solved, professional programmers have a favored **order** of program construction that is measurable with my learning analytics approaches. For my example problems, they construct the `for` loop header **before** initializing their data variable. This is notably different from the order of program construction favored by student programmers, which is the construction of the `for` loop header **after** initializing the data variable.

Knowledge of this evidence can give rise to instructional methodologies for programming education that are supported by theoretical and, with my results, empirical findings. My results suggest that the acquisition of expert programming skills can be facilitated by introducing the experts' program construction patterns to learning programmers – not for them to memorize the order of construction, but to let them break out of reasoning associated with novice programmers and let them engage in abstract reasoning towards becoming an expert. Moreover, my results provide impulses for learning analytics methods and tools that can facilitate the real-time assessment and feedback of program construction, thereby enabling the support of individual skill acquisition.

ZUSAMMENFASSUNG

Die digitalen Technologien verbreiten sich schnell und weit und erfassen viele Aspekte unseres täglichen Lebens. Dies wirkt sich auch auf die Bildung aus, denn das Programmieren wird weltweit in vielen Bildungsstandards vom Kindergarten bis zum Abschluss der Schule aufgenommen. Dieser frühe und vielfältige Programmierunterricht stellt eine Herausforderung für die traditionelle Programmierausbildung dar. Sie erfordert die Entwicklung von Theorien, Methoden und Werkzeugen, die den Erwerb von Programmierfähigkeiten auf individueller Basis unterstützen, um damit das Lernen in heterogenen Gruppen zu erleichtern.

Fortschritte in der Programmierausbildung können aus zwei Perspektiven initiiert werden. Das Forschungsgebiet der Informatikdidaktik bietet einen reichen Fundus zur Unterstützung von Programmieranfänger*innen beim Erlernen des Programmierens. Das Forschungsgebiet des Software Engineering verfügt über eine große Sammlung von Ergebnissen zu den Unterschieden zwischen Programmieranfänger*innen und -expert*innen, die in dem Bemühen gipfelt, die Programmierfähigkeiten von Expert*innen zu charakterisieren. Zwischen diesen beiden Perspektiven ist eine Lücke, nämlich die adäquate und individuelle Unterstützung von Programmierer*innen, die die ersten Schwierigkeiten überwunden haben, sich aber auch noch nicht zu Expert*innen entwickelt haben.

Meine Forschungsarbeit, die in dieser Dissertation zusammengefasst ist, nimmt Inspiration aus zwei Quellen, um auf die Schließung dieser Lücke hinzuarbeiten. Die erste ist die Idee, dass Entwurfsmuster von Software Teile der Programmierfähigkeiten von Expert*innen darstellen und dass Muster der Programmkonstruktion dasselbe bieten könnten. Die zweite Quelle ist das aufstrebende Feld der Learning Analytics, das die technischen und methodischen Mittel zur Erforschung von Programmierdaten in einem neuen Maßstab bereitstellt.

Durch die Kombination dieser Inspirationen umfasst meine Forschung eine detaillierte Untersuchung des sequenziellen Prozesses der Programmerstellung einzelner Programmierer*innen. Die Annahme, die meine Forschung antreibt, ist, dass erfahrene Programmierer*innen ***bestimmten Mustern*** in der ***sequenziellen Konstruktion ihrer Programme*** folgen und dass solche Muster regelmäßige, erkennbare Formen haben, die mit Ansätzen der Learning Analytics identifiziert und anschließend zur Verbesserung der Programmierausbildung genutzt werden können.

Um diese Annahme zu untersuchen, habe ich ein Learning-Analytics-System entwickelt, das aus IDE-basierten Instrumentierungswerkzeugen auf der Client-Seite und einem einheitlichen Datenerfassungsserver besteht. Mit Hilfe dieses Systems ist es möglich, Schnappschüsse von blockbasierten (Scratch 2, Scratch 3) und textbasierten (IntelliJ, Java) Programmen während ihrer sequenziellen Erstellung auf der Granularität von Tastenanschlägen aufzuzeichnen und zu speichern, um sie später auf der Granularität von Tastenanschlägen oder Kompilierungen zu analysieren.

Ich definiere spezifische variablenbasierte Programmkonstruktionsmuster als *eine typisierte Sequenz kompilierbarer Programmänderungen, die sich auf zusammenhängende Programmanweisungen einer Variable auswirken*, um zu beurteilen, ob und wie erfahrene Programmierer*innen ihre Programme unterschiedlich konstruieren. Ich führe eine Mixed-Methods-Studie durch, um Muster zu identifizieren, die von Student*innen und professionellen Programmierer*innen verwendet werden, und führe eine vergleichende Studie durch, um über signifikante und wesentliche Unterschiede in der Verwendung von Mustern zwischen Programmierer*innen mit verschiedenen Fähigkeiten zu berichten.

Meine Ergebnisse belegen, dass Programmkonstruktionsmuster in der Tat Teile der Programmierfähigkeiten von Expert*innen erfassen. Insbesondere zeigen meine vergleichenden Ergebnisse mit Signifikanz, dass, während die Verwendung einzelner Muster mehr von dem spezifischen algorithmischen Problem abhängt, das gelöst werden soll, professionelle Programmierer*innen eine bevorzugte Reihenfolge der Programmkonstruktion haben, die mit meinen Ansätzen messbar ist. Bei meinen Beispielproblemen konstruieren sie den `for`-Schleifenkopf, **bevor** sie ihre Datenvariable initialisieren. Dies unterscheidet sich deutlich von der von den Student*innen bevorzugten Reihenfolge der Programmkonstruktion, nämlich der Konstruktion des `for`-Schleifenkopfes **nach** der Initialisierung der Datenvariablen.

Diese Erkenntnis kann zu Lehrmethoden für die Programmierausbildung führen, die durch theoretische und – mit meinen Ergebnissen – empirische Erkenntnisse gestützt werden. Meine Ergebnisse deuten darauf hin, dass der Erwerb von Programmierkenntnissen durch die Verwendung von Programmkonstruktionsmustern von Expert*innen für Programmieranfänger*innen erleichtert werden kann – nicht, damit sie die Reihenfolge der Konstruktion auswendig lernen, sondern damit sie aus der bisher gelernten Denkweise ausbrechen und abstrakte Überlegungen anstellen können, um sich zu Expert*innen zu entwickeln. Darüber hinaus geben meine Ergebnisse Impulse für Methoden und Werkzeuge, basierend auf Learning Analytics, die die Echtzeitbewertung und das Feedback zur Programmkonstruktion erleichtern und so den individuellen Kompetenzerwerb unterstützen können.

CONTENTS

1. INTRODUCTION

There has been progress in mechanisms for gathering data, but there is a great deal of scope for [...] ***understanding the differences between novice and expert programmers*** *[emphasis added].* Luxton-Reilly et al. [LRSA$^+$18, p. 61]

1.1 Using Data of Program Construction Processes in Education

Digital technologies are spreading wide and fast, encompassing many aspects of our daily life. This comes with a growing demand on educational systems worldwide to properly support and prepare society to take an active role in the technology-driven world of today and tomorrow. Many see the establishment of *computational thinking*, the notion of a set of cognitive abilities and skills related to computer science but applicable in many other areas [Win06, SW13, GP13], as one of the main driving factors for the observed educational change. While one can argue about the general-educational character of *computational thinking*, it did spark change in educational systems to cater to these rising demands, albeit in different ways [Pas20]. The result is that competencies related to digital technologies, computer science, and programming are incorporated in many K–12 educational standards.

This incorporation into education leads to additional points of contact with programming. These points are *horizontal* (more and more areas make use of computational ideas, techniques, and tools [Win06]) and also *vertical* (the first contact with programming is now earlier than ever, in part thanks to block-based programming languages [Pap80, vFP96, CDP03, MPK$^+$08]). In the context of programming education, these additional points of contact provide a twofold challenge. On the one hand, it makes for heterogeneous groups of learners that have different previous experiences with programming. On the other hand, also the aims and goals of learners engaging in programming education are different: not every school pupil or university student who is learning to program wants to become a software architect or professional programmer. Therefore, it is especially important to offer a diversified educational program to cater to the learners' individual needs and goals.

A model of how novice programmers can be educated and supported is established by Lister [Lis16] in his research catalog. He adapts the neo-Piagetian model of cognitive stages that occur in *overlapping waves* to build an understanding of the cognitive processes and transitions during learning to program. However, there is an open issue regarding the stage of expert programmers (*formal operational* reasoning [CTAL12]). The transitions to becoming an expert programmer and, consequently, the support necessary to facilitate a learner's skill acquisition towards expert programming skills are not mapped out. This gap is relevant to all learners: those who learn to program to become tech-versed in the sense of general-educational *computational thinking* and those who learn to program to become expert programmers can both benefit from acquiring specific programming skills earlier and faster.

Some facets of expert programming skills have already been formalized since the early days of computing education (e.g., see [Sol86, Win96]). Related to these facets are models of what constitutes good software design and program implementation, with a prominent example being the (object-oriented) patterns of architectural design [GHJV95]. In short, models of expert programming skills, *distilled* in these patterns, have been captured from finished program code.

One way to complement the research of such models and thus to work towards conveying expert programming skills to learning programmers is to investigate the process of program construction. As expert programmers tend to implement specific patterns that have proven to perform well and provide a highly functional and maintainable software design, perhaps they also use specific patterns during the process of program construction to facilitate the construction of correct and well-designed programs. That is, maybe expert programmers follow ***specific patterns*** in the ***sequential construction of their programs***?

This hypothesis is the central assumption that drives the research of this thesis. My research approach is motivated by the introduction citation that differences between novice and expert programmers still need to be researched [LRSA$^+$18] and is supported by the rising field of *learning analytics* in programming education [IBE$^+$15, BASK18] that provides the technical and methodological means to research not only single, finished program versions but evolving program versions.

In the context of this thesis, I set out to analyze the process of program construction on a detailed level, utilizing the means of learning analytics to record, process, and analyze how programmers of different skill levels implement their program code. The objective is to assess whether and how expert programmers construct their programs differently. The aim is to derive recommendations that can be applied directly by educators and can be incorporated in assessment and feedback systems with the help of learning analytics, ultimately improving programming education. The research questions and methods of this thesis are described in Section 1.2.

1.2 Research Aims and Research Questions

The main research aim of the thesis is to support the acquisition of programming skills by examining a less extensively researched facet of learning to program, the sequence of program construction. Specifically, with the use of learning analytics, I work towards supporting educators in providing timely and individual feedback to learning programmers and students in individually acquiring programming skills with the use of automatically processed assessments.

The underlying hypothesis that drives the research of program construction sequences is that expert programmers use regular sequences (i.e., *patterns*), maybe unconsciously, to implement programs efficiently and correctly. These patterns represent expert programming skills, which are demonstrated in a planned manner or unbeknownst to the experts as part of latent expert programming skills. In any case, the hypothesis is that expert programmers create programs in a measurably different way. Deriving, I conjecture that evidence of such expert patterns and the patterns themselves can be used to educate programmers of different skill levels. Thus, the main research question is:

MRQ. To what extent can patterns of program construction sequences be indicative, whether positive or not, of the acquisition of expert programming skills?

Two parts of this question need to be further clarified. First is the notion that there are no *correct* patterns and that my research aims not only to identify expert patterns but also to identify patterns of learning programmers. Latter patterns might not be *positive* in the sense that they might hinder the construction of a correct program or impede the future acquisition of program skills. Similarly, expert patterns also might not be *positive* and only hold true for specific program constructions. However, all those patterns provide knowledge about the process of program construction and can help towards improving education and training.

Second is the objective of the thesis that is represented in the main research question. This thesis aims to uncover whether patterns of program construction sequences are suitable for eliciting differences in the program construction that are attributable to a difference in programming skills. To this end, I record the program construction of different skill groups of programmers (school pupils, university students, professional programmers) and develop tools and techniques to assess recorded sequences of program construction towards this goal. In this thesis, I do not assess the potential effects of the use of my tools, techniques, and findings in education and training, but I provide the basis for doing so.

The main question is divided into four sequential research questions to work towards answering the main question.

RQ1. What distinguishes novice from expert programmers?

The objective of the first research question is to define the notions of programming skills for this thesis. Towards this goal, I employ a **literature study** to identify the established dimensions and hierarchies of programming skills from related works in the area of computer science education and software engineering. Furthermore, this research question serves to solidify the notions of **novice programmers** and **expert programmers** for this thesis, based on the dimensions of programming skills and the differences between programmers of varying skill levels that have been reported in related works.

In the context of the main research question, answers to this research question serve to illuminate how expert programming skills are formed, interpreted, and represented in program construction.

RQ2. How can program construction patterns be defined?

The objective of the second research question is to provide a definition of **program construction patterns** for this thesis. I follow two methodological approaches to arrive at answers to this research question. The first approach is a **literature study** to elicit similar notions of sequential regularities in program construction processes, with the goal of developing a notion of **program construction patterns**. The second approach is an **exploratory study** to record and analyze exemplary program construction sequences of varying programmers (school pupils, university students, professional programmers) with the help of learning analytics. The goal is to assess different features that can be measured in and derived from program construction sequences to arrive at a set of defining features and a corresponding definition of **program construction patterns**.

In the context of the main research question, answers to this research question provide a solid foundation of the notion of program construction patterns and make it possible to analyze recorded sequences of program construction to identify specific patterns.

RQ3. What program construction patterns are used by novice and expert programmers during program construction?

Building on the definition and the set of measurable features of program construction patterns established above, the objective of the third research question is to identify specific patterns that are used by programmers of different skill levels during program construction. I conduct a sequential **mixed methods study** on recorded program construction sequences of university students and professional programmers to work on answers to this research question. I first employ the qualitative method of **Grounded Theory** to identify a basic catalog of patterns based on the measurable features of program construction patterns and conditional on a set of used example problems. Second, I employ the quantitative method of **descriptive statistics** to report on the usage of the identified specific patterns by the recorded programmers.

In the context of the main research question, answers to this research question do not aim to provide a differentiation between programmers of different skill levels but to build a catalog of used program construction patterns.

RQ4. What are the differences between novices and experts concerning the use of program construction patterns?

The objective of the fourth research question is to evaluate whether differences in program construction sequences, measurable with program construction patterns, can be attributed to a difference in programming skills. To work towards answers to this research question, I conduct a **comparative study** on the recorded program construction sequences and corresponding patterns of university students and professional programmers. I employ quantitative analysis methods, including **inferential statistics** and **machine learning**, to evaluate significant and substantial differences between programmers of different skill levels in the use of program construction patterns. The evaluation is structured with three null hypotheses to evaluate differences on three levels of granularity. Findings to acknowledge or refute these null hypotheses and interpretations towards the respective indication of programming skills constitute the answers to this research question.

In the context of the main research question, answers to this research question provide the final piece to answering the main research question by differentiating programmers of different skill levels based on their used program construction patterns.

Answers to the main research question include evaluating how and why the catalog of program construction patterns can be used to measure features in program construction sequences that indicate expert programming skills. I formulate answers to how and why specific patterns of program construction can be attributed to a higher level of programming skill.

1.3 Structure of the Thesis

The chapters of this thesis all contribute to deriving answers to the main research question and are sequentially structured as follows.

Theoretical Background and Context

The context of this thesis is introduced in two chapters. In Chapter 2, I introduce related works of computer science education, focusing on programming education in text-based and block-based programming environments, and I introduce the psychological frameworks of neo-Piagetian theory and cognitive load theory that feature widespread use in programming education. The centerpiece of this chapter is the identification of dimensions and hierarchies of programming skills, which culminates in answering the first research question `RQ1`.

In Chapter 3, I introduce the specific context of this thesis: the use of learning analytics (LA) in programming to collect learners' data during programming, process this data, derive meaningful interventions, and deliver these interventions to improve education. Besides related works, I introduce my instrumentation and data collection approach to support IDE-based LA for this thesis[1].

Towards a Definition of Program Construction Patterns

In Chapter 4, I first report on an exploratory study that makes use of my IDE-based LA approach to investigate different types of syntactic and semantic features extractable from recorded program construction sequences. I conduct an exploratory analysis of block-based and text-based program construction sequences. The result of this exploratory study gives answers to the second research question `RQ2`: a general definition of *program construction patterns* in terms of syntactic and semantic features of program construction sequences, and a specific definition of *variable construction patterns*, focusing on construction steps that change a variable.

Secondly, I describe a mixed methods study in Chapter 4 to elicit the types of *variable construction patterns* that are used by student programmers and professional programmers. The result of this mixed methods study gives answers to the third research question `RQ3` in the form of i) a catalog of used patterns (conditional on the collected example problems), and ii) usage statistics of the different programmer cohorts.

Relating Program Construction Patterns and Programming Skills

I build on these answers and describe a comparative study in Chapter 5 to evaluate significant and substantial differences between student programmers and professional programmers that can be found based on collected program construction sequences and the used *variable construction patterns*. The study is designed with three null hypotheses, and the results of acknowledging or refuting these null hypotheses provide the foundation to answer the fourth research question `RQ4`.

Contextualization and Conclusion

In Chapter 6, I provide theoretical and practical contextualizations of the approaches and results of my research work. Finally, a comprehensive summary of my research work and potential future work is given in Chapter 7.

[1] Link to the developed open source projects: `https://gitlab-iid.aau.at/seqtrex`

Appendix

In the appendix, Chapter A includes technical information regarding the formalization of slice-based cohesion metrics on the level of variables. These metrics are utilized in Chapter 4 to discern the features and qualities of program construction sequences. Appendix Chapter B provides example visualizations of *variable construction patterns*, represented in the form of variable-specific *construction flowlines*, that are the result of Chapter 4.

2. THEORETICAL BACKGROUND

In treating the theoretical background of the thesis, the purpose of this chapter is twofold. On the surface level, the aim of the chapter is to introduce the context of computer science education this work is situated in. The context is established in three separate sections:

1. In Section 2.1, the research area of computer science education is sketched, focusing on programming education.
2. In Section 2.2, dimensions and hierarchies of programming skills are distilled by reporting a literature study.
3. In Section 2.3, two psychological frameworks often used in programming education are introduced: the *(neo)-Piagetian* theory of cognitive development, and the theory of *cognitive load*.

A summary of the sections outlined above and a derived answer to the first research question is given in Section 2.4. This constitutes the second level of this chapter, answering the first research question of the thesis:

RQ1. What distinguishes novice from expert programmers?

The overall structure of the literature review is visualized in Figure 2.1.

2.1 Programming Education as Part of Computer Science Education

In the early days of programming education, a great deal of research was concerned with establishing an understanding of the cognitive process of programming, often grounded on a comparison of programming performances of novice programmers (typically computer science students) and expert programmers (typically ranging from graduate students to professional programmers). See, for example, the publications by Soloway and Ehrlich [SE84, Sol86], in which they empirically validate how expert programmers structure their programming knowledge (in generic program fragments called *programming plans*, and strategies of how to compose the

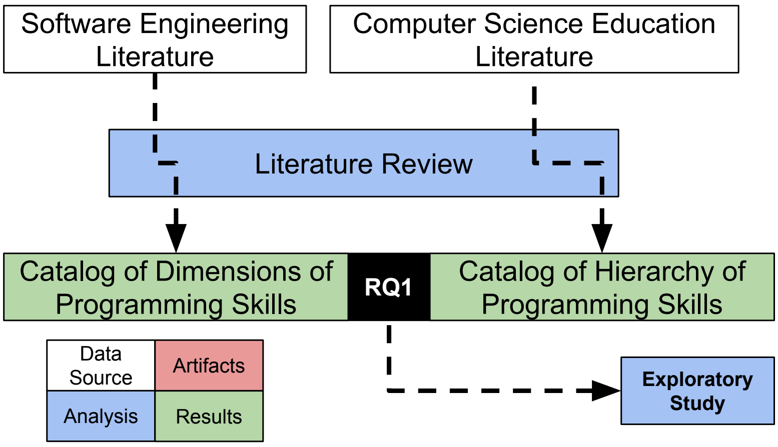

Software Engineering Literature
Computer Science Education Literature
Literature Review
Catalog of Dimensions of Programming Skills
RQ1
Catalog of Hierarchy of Programming Skills
Data Source
Artifacts
Analysis
Results
Exploratory Study

Fig. 2.1: Structure of the literature review.

fragments into programs called *rules of programming discourse*) and explore how introductory programming could benefit from these structures of programming knowledge. In another early publication, Détienne [Dét90] introduces the *schema-based approach* to programming (programming with an emphasis on semantic structures), which is contrasted with a *control-flow approach* to programming (programming with an emphasis on syntactic structures). According to Détienne [Dét90], expert programmers most often following the *schema-based approach* when organizing their programming knowledge.

The goal of this research was to derive a '*programming pedagogy*', as described by Winslow [Win96], with the deflating answer that, concluding from psychological studies in various fields, novice programmers cannot *turn into* expert programmers through the course of a four-year undergraduate study program. The reason is that experts are characterized by qualities such as appropriate structures of mental models and of domain knowledge and by the ability of strategical problem solving approaches – qualities developed beyond the mastery of '*basic facts, features and rules*'[Win96, p. 21].

The findings of Winslow have later been supported by a multi-national assessment of programming skills of first-year computer science students, conducted by McCracken et al. [MAD$^+$01]. The students participating in the trial assessment scored an average 22.89 out of 110 points ($n = 216$), illustrating that students are **not** able to program after attending introductory programming courses. The results are framed with a process students should learn, consisting of the following steps: i) abstract the problem from its description, ii) generate sub-problems, iii) transform sub-problems into sub-solutions, iv) re-compose the sub-solutions into a working problem, and v) evaluate and iterate. The authors map and explain the students' low scores to failure in the early stages of this process. The suggestion is to direct pedagogy towards planning, designing, and testing in addition to implementation activities.

Lister et al. [LAF$^+$04] conducted a follow-up study to investigate alternative explanations on why novice programmers fail in the way reported by McCracken et al. [MAD$^+$01]. Their study uses two task types, both touching on programming skills related to reading and (mentally) executing program code. The first is to predict the outcome of executing a short program. The second is to select correct completion fragments when given a short near-complete code and its desired functionality. The authors conclude that students perform poorly on these tasks of basic programming knowledge and programming skills, and therefore lack the abilities that are precursors to problem-solving. Following this explanation, programming education needs to focus on these basic abilities before considering software design and implementation.

With a study of subjective difficulties of novice programmers, Lahtinen et al. [LAMJ05] show that students rate their difficulties in '*Understanding programming structures*' (2.92 ± 1.02) not much lower compared to difficulties in software design, like '*Designing a program to solve a certain task*' (3.12 ± 0.98) and '*Dividing functionality into procedures*' (3.10 ± 1.09). Educators, however, rate the students' difficulties in the latter two areas much higher. The authors do not report on whether there are groups within the student responses. That is, it cannot be said whether students that struggle with reading-related programming skills (as described by Lister et al. [LAF$^+$04]) have a different perception of their difficulties.

In 2006, Jeanette Wing published an article that sparked international change to computer science education, including a more thorough commitment to programming education. In her article, she explored the idea of *computational thinking*[1] as a necessary skill in a society more and more ubiquitously filled with computing systems [Win06]. Subsequently, the definition of *computational thinking* is refined: Selby and Woollard [SW13] establish that decisive skills of *computational thinking* are the cognitive abilities of thinking in abstractions, in terms of decomposition, algorithmically, in terms of evaluations, and in generalizations. These skills can be transferred to domains other than computer science, as intended by Wing. Furthermore, Grover and Pea [GP13] note that the skills of *computational thinking* can best be acquired by training them in their main domain, computer science, and specifically activities related to modeling and programming. This viewpoint would solidify the importance of computer science education for all if the goal is to prepare students for a world in which computing skills are very much needed – at least from the view of computer science educators.

However, the true educational situation is often clouded by forces pulling into

[1] Wing was not the first to use the term (see Papert [Pap80] in 1980), but managed to mediate an educational-political message.

different directions. Although many countries strive to include computer science education in their educational programs as best they can, it cannot always be easily introduced as a compulsory subject. Hubwieser et al. [HAB^{+}11] report a qualitative case study to model and represent the situation of computer science education in different countries (Austria, the federal state of Bavaria of Germany, Greece, Israel, and Lithuania), resulting in the *Darmstadt model* to capture regulated computer science education in three dimensions. A notable finding is the topic of learning objectives, categorized with *ACM Computing Classification Scheme* from 1998[2]. More than a third of all learning objectives are categorized to the knowledge part of *Information Systems*. Only about 11% of the learning objectives touch directly on programming education ('*Computing Methodologies\...\I.1.2 Algorithms*'). While the countries support the strand of *computational thinking* in 73% of prominent objectives, on average, the lack of learning objectives in programming education reflects its respective importance in the studied curricula.

A recent international review of the situation of educational systems concerning computer science education is given by Pasterk [Pas20]. He again demonstrates a high variability of computer science education along the dimensions of compulsory education, age groups, corresponding curricula/standards/frameworks, and their modeled levels of learning objectives. Concluding, the standardization of computer science education is far from complete, and this also affects programming education.

In recent years, there have been different angles to drive (introductory) programming education forward. Watson and Li [WL14] tackle the myth of low pass rates in programming education by replicating a study on failure rates in introductory programming education. They report a worldwide pass rate of 67.7%. While the failure rate is significantly moderated by grade level, country, and class size, the authors regard these not as substantial differences. More important is that pass rates did not significantly change since the 1980s. Based on their interpretation, introductory programming education suffers from a troubled reputation much more than from alarmingly low pass rates – which is in of itself a motivation to increase the pass rates, in turn potentially increasing its reputation.

McCartney et al. [MBE^{+}13] perform an adjustment on the instrument introduced by McCracken et al. [MAD^{+}01], namely providing partially implemented project code which simplifies input/output and data structure tasks and report on significantly higher student scores compared to McCracken et al. [MAD^{+}01]. The adjusted instrument *lessen[s] cognitive load and remove[s] troublesome knowledge* and presents the students with a programming task they can solve.

In a similar vein, Luxton-Reilly [LR16] provides the argument that learning to program is not difficult, but educators might hold unrealistic expectations of what

[2] The link in the paper redirects to the updated classification scheme of 2012: `https://www.acm.org/publications/class-2012`

students are capable of at a particular stage or in specific courses. He suggests to letting go of '*disciplinary norms*' and starting to teach to '*achievable outcomes*' [LR16, p. 288]. Future work should uncover what novice programmers can achieve at certain stages and how basic programming skills are mastered. Later, Luxton-Reilly directly contributes to finding out what makes programming assessments hard for novice programmers, with a major factor being the compound nature of the assessment problems [LRP17, LRBC$^+$18]. According to these authors, assessment questions involve a number of different concepts and facts at the same time, demanding a student's mastery in all of the required areas in order to produce a solution. Therefore, students may fail while potentially knowing some or most of the material, and educators cannot elicit the areas the students are struggling with.

Lister contributes to the notion of not holding unrealistic expectations by providing a developmental epistemology of programming, grounded in neo-Piagetian psychological stage theory [Lis16]. In his view, the overlapping stages of reasoning should be applied to programming education, with each novice programmer traversing through them. Lister identified archetypical tasks for three of four stages, but the transition between stages is not fully mapped pedagogically.

In the last decade, there was a noticeable shift in programming education: moving away from the premise of easily raising expert programmers by training technical and cognitive skills, to acknowledging the difficulties of novice programmers and tailoring pedagogical approaches to their needs. Additionally, the widespread use of computing technology led to the development of programming environments designed for a new clientele, including introducing programming to a broader and younger audience with adjusted text-based programming environments like `Logo` and its turtle [vFP96], and block-based programming environments like `Alice` [CDP03] and `Scratch` [MPK$^+$08]. There is no comprehensive approach that ties together block-based and text-based programming for an integrated path to learn programming. In the following sections, findings of these types of programming environments are given: Section 2.1.1 summarizes findings specific to text-based programming education, and Section 2.1.2 summarizes findings specific to block-based programming education.

For a more comprehensive review of existing literature, there is a review of students' difficulties and misconceptions in introductory programming [QL17], and a systematic literature review with a focus on categorized groups: student, teaching, curriculum, assessment [LRSA$^+$18].

2.1.1 Text-based Programming Education

In this section, a chronological summary of findings in text-based (introductory) programming education of the last two decades is given. The focus is not on studies that train novice programmers to become expert programmers, but on studies that identify specific aspects and difficulties in novice programming skill acquisition, and that describe frameworks and approaches to improve skill acquisition. In this context, novice programmers are typically undergraduate students, not only including introductory programming courses.

Most notable is the research program of Lister, culminating in the description of his findings towards a developmental epistemology of programming education [Lis16]. The interpretation of neo-Piagetian developmental stages has been documented in the thesis of Teague [Tea15]. In this section, selective findings are given; the hierarchy of novice programming skills in the frame of neo-Piagetian developmental stages is described in Section 2.2.2.

In an early study, Lister and Leaney [LL03] report on the application of a criterion-referencing approach to programming skill assessment tasks and differentiate the tasks based on Bloom's taxonomy. This way, different programming tasks can be solved by students of different programming skill levels. The authors differentiate three fundamental assessment tasks: reading and understanding programs (demonstrating knowledge and comprehension); traditional, familiar code writing tasks (demonstrating the ability to apply and analyze); and open-ended project tasks (demonstrating the ability to synthesize and evaluate). This approach highlights that programming education often focuses on higher levels of Bloom's taxonomy and neglects lower levels, putting students in need of support at a disadvantage.

In a follow-up study, Lister et al. [LST^{+}06] apply the SOLO (*Structure of Observed Learning Outcomes*) educational taxonomy on written and verbal responses to programming tasks and find a profound difference between novices (students) who form a multistructural response, and experts (educators) who form a relational response. The authors interpret that a sole focus on writing and hand-executing programs does not elicit relational responses and does not provide novice programmers with opportunities to develop the required skills. Moreover, the authors advocate for increased importance to teach students to read and relationally understand program code.

The previous study led to several reports regarding the relationship between the programming skills reading, mentally executing (tracing), and writing program code [LWRL08, LFT09, VTL09]. The results of these three studies support a hierarchical development of programming skills. Students first learn to read and trace program code. Next, with the ability of reliable tracing program code, the students develop the ability to explain code (including giving natural language summaries of what a piece of code does). Lastly, the students' ability to systematically write program code emerges. The hierarchy is not believed to be strict, but rather in such a way that the different skills support each other and develop in parallel, but with threshold effects capping development (i.e., writing ability is poor and underdeveloped with a very low tracing score, although writing and tracing code is only weakly correlated for high tracing scores [VTL09]).

These findings set the context for the interpretation of neo-Piagetian developmental stages in introductory programming, first given by Lister [Lis11b]. The stages describe the cognitive abilities of novice programmers and highlight a discrepancy between the educator who reasons and teaches in the highest form of reasoning (*formal operational*) and the students following at lower forms of reasoning. According to Lister, students most often reason on the *preoperational* stage – when mental execution is consistently possible; or on the *concrete operational* stage – when abstract reasoning on specific code is possible). Lister concludes that students do not improve in their reasoning '*simply by being exposed to learning materials couched in formal operational terms*' [Lis11b, p. 17] – a didactical approach fitting to the students' abilities is essential. For example, writing code is not an appropriate type of learning experience for students not able to trace consistently or reason in an abstract way.

In the following studies, Lister and his collaborators focused on transitions between developmental stages:

1. Corney et al. describe empirical results of novice programmers' reasoning in the frame of neo-Piagetian developmental stages [CTAL12]. They provide evidence that seemingly *simple* questions cannot be solved by students with insufficient reasoning abilities (with a focus on the transition between *preoperational* and *concrete operational* stages). With the neo-Piagetian perspective, the authors interpret the results as normal behaviours during the cognitive development of novice programmers.

2. Teague and Lister investigate the ability to trace program code [TL14a]. With think-aloud sessions, the tracing strategies of students between the *preoperational* and *concrete operational* stages are differentiated. Most importantly, the process of how students go about tracing and how they use results of individual traces to reason about the whole code characterizes the stage. Students at the *preoperational* stage are reliant on specific values, are preoccupied with the process of tracing, and can only infer from the actual traces. Students at the *concrete operational* stage are able to reason about the code during tracing and during reading it and thereby understand abstract relations in the code.

3. Teague and Lister investigate the ability to reason about reversibility as a defining characteristic of the *concrete operational* stage [TL14c]. They provide evidence that advanced programming tasks require reasoning abilities not readily available for a majority of students at the introductory programming education level, which makes them unable to produce a coherent response to such tasks. Again, the implication is to confront students with programming tasks in their respective '*reality*', and not overburden their reasoning abilities.

4. Teague and Lister report on a longitudinal think-aloud case study of one novice programmer, following them through their developmental stages [TL14b]. The case study includes direct observational evidence on the manifestation of reasoning on the succession of developmental stages. An unsolved question is how to generalize the successive development through these stages from this single case study.

Altogether, the foundational approach of Lister frames novice difficulties in an understandable way and makes their interpretation possible. Moreover, it can direct research focus on different didactical points of programming education, including questions such as: How do interventions work on students of different developmental stages? Which support is best suited for students of particular developmental stages?

In the context of this thesis, the most important question unanswered in the research program of Lister is: How can students be supported to reach the highest stage of programming (*formal operational* reasoning)? This question gives a frame to all approaches and results of this thesis.

Besides the findings of Lister and his colleagues, there have been other approaches to support the acquisition of programming skills of novice programmers. These approaches are also summarized in chronological order.

Tying back to the early days of programming pedagogics, De Raadt, Watson, and Toleman performed a pair of studies regarding the explicit teaching and assessing of problem solving strategies in order to educate programming novices [dRWT06, dRWT09], inspired by Soloway's notion of plans and schemata as part of expert programming skills [Sol86]. They report the following evidence: experts indeed exhibit plan-like structures in their programming solutions, the explicit inclusion of named strategies as solutions to specific programming sub-problems has an effect on novice programmers' use of those strategies – they generally use them more frequently. However, the effects on the development of novices have not been studied.

A number of studies investigated the mental model of novice programmers regarding key programming concepts. Both Bornat et al. (investigation of the consistency of mental models of value assignment, and a description of wrong mental models) [BDS08] and Reges (investigation of a boolean value assignment of the form `b:=(b=false)`, and the students' ability to accurately predict the outcome) [Reg08] frame the formation of correct mental models as a question of aptitude, but they essentially measure whether students can accurately execute the program code (i.e., whether they can accurately trace program code). The model of Lister much rather suggests that programming skills can be acquired by any student, only the transition between stages could be different [Lis16].

Ma et al. assess the programming concepts of value and reference assignment, distinguishing between viable mental models (*consistently appropriate*) and non-viable mental models (further distinguished between *inconsistent*, and *consistently* *inappropriate*) [MFRW07, MFRW11]. The results include a catalog of non-viable mental models regarding the studied programming concepts, the finding that some students hold a viable mental model regarding value, but not regarding reference assignment (representing the need for further support), the finding that students with viable mental models perform significantly better in programming tasks and exams. Based on these findings, the authors develop a learning model and a visualization tool in order to facilitate the students' development of viable mental models, with a significant improvement of the developed mental models.

With a step away from program code, Hanks and Brandt identify a number of successful (use external resources like books, use the debugger, use print statements for troubleshooting) and unsuccessful (inefficient solution approach with regard to planning sub-programs, infrequent compilation, inadequate testing of their own code, not reading API documentation, commenting out code, making erroneously generalized assumptions from the problem statement, patching instead of understanding errors) problem solving approaches when dealing with a programming task in Java [HB09]. Interpreting these findings with neo-Piagetian stages, the students could be working at the *concrete operational* stage and could develop further with proper support for the identified approaches.

Accompanying the results of Lister, Simon and Snowdon show further evidence that code explaining questions present a serious challenge to students with low programming skills [SS11]. Even transforming the code explaining questions to multiple-choice questions (giving one correct and three incorrect potential choices), students are not significantly better at correctly solving the problem compared to students with written answers. The authors conclude that the students lack the skill to read and understand the code (reason about it).

Another study on the relationship between code tracing and code writing is reported by Kumar [Kum15]. Students had been administered a problem-solving session utilizing code tracing as an activity, which led to a significant increase in programming performance, comparing pre- and post-programming tasks. The increase was not found in control problems regarding syntactic and semantic programming constructs. On detailed inspection, the authors found out that this increase in performance was profoundly exhibited in the high-performers of the problem solving session ($\geq 90\%$). The authors conclude that code tracing is potentially essential for novice programmers to learn code writing.

With tracing being an important ability in the development of programming skills, the question is how it could be incorporated in programming education. A detailed analysis of different sketch types used by students and educators when tracing and solving code reading tasks, and their relation to correct answers, is carried out by Cunningham et al. [CBEG17]. They show that the sketch type *trace*, a systematic listing of all variables and their current and previous values, is the most successful sketch type and that students who draw any kind of sketch are more successful compared to those who do not draw any sketch. They also find out that educators (and students with prior programming experience) trace differently (which is most likely related to their higher level of reasoning). Following, Xie et al. [XNK18] report that explicitly teaching a tracing strategy to students encourages them to trace systematically, which leads to significantly higher scores in an experimental situation compared to students who have not been taught the tracing strategy explicitly. Explicitly teaching a tracing strategy even translates to higher average scores on the course midterm exams.

There is still little empirical evidence on the order in which novice programmers should learn programming concepts. On this topic, Hofuku et al. [HCNK13] describe an approach to support learning programming in small steps by analyzing introductory programming textbooks, finding the learning order of programming concepts, and, most importantly, identifying gaps in the learning order that need to be bridged by suitable didactical teaching units. They identify variability in the order of fundamental programming concepts of `C` textbooks and advocate to measure the *height* of the gap when introducing new concepts so as not to overwhelm students.

Rich et al. [RSB$^+$18] report on learning trajectories, derived from literature, for the programming concepts *sequence*, *repetition*, and *conditionals*. Each trajectory consists of three successive levels and multiple learning goals. The learning goals for the three programming concepts include cognitive abilities as well as programming abilities. The learning trajectories are designed to capture K–8 learning but can also be transferred to novice programmers at student levels (albeit students will be proficient in cognitive abilities of earlier levels). The authors state that several more trajectories have to be created in order to properly map out programming education. Moreover, the relative difficulty of different concepts has to be evaluated empirically.

Lastly, there are studies that tackle the question of which pedagogical and didactical approaches are more effective when teaching programming. Koulouri et al. [KLM14] report an experiment to compare three factors when teaching programming: the programming language (`Java` and `Python`), teaching problem solving explicitly, and using formative assessment. The authors report that two of the three factors lead to a significant improvement in student learning: the use of a syntactically simpler programming language (`Python`), and the approach to teach problem solving explicitly. The third investigated factor, the use of formative assessment, is critically interpreted by the authors. Students may need to be externally motivated, and the feedback has to be carefully applied in accordance with the tasks and the students. These results provide evidence that novice programmers' programming skill acquisition is dependent on the teaching approach. While teaching introductory programming still remains a *multidimensional problem*, the authors contribute two explicit findings on how to improve introductory programming education.

Ericson studies the use of *Parson's Problems*, a programming task where the correct code is broken into mixed-up blocks and has to be reordered to solve it, to teach novice programmers [EMR17, EFR18]. In a comparative study, they found that *Parson's Problems* provide an efficient way to acquire programming skills, compared to fixing or writing the equivalent code. Learning with *Parson's Problems* took significantly less time with no impact on learning performance or students' knowledge retention [EMR17]. Moreover, through the use of adaptive *Parson's Problems*, an introduced dynamical adaption of the programming task (either intra-problem by merging blocks on incorrect student attempts, or inter-problem by modifying the difficulty of the next problem given a specific performance), students can acquire similar programming skills compared to writing equivalent code in less time [EFR18].

To summarize, research in introductory programming education is conducted with different viewpoints. Some findings and approaches aim to solve specific open questions of programming education ([dRWT06, dRWT09, HCNK13, RSB^{+}18]), aim to investigate, understand and mitigate specific problems that are faced by novice programmers ([BDS08, Reg08, HB09, MFRW07, MFRW11]), or aim to improve quantifiable effects of programming education [KLM14, EMR17, EFR18]. Other authors seek to develop holistic approaches to improve programming education ([SS11, Kum15, CBEG17, XNK18, Lis16]). The framework of Lister, built to understand different stages in the cognitive development of novice programmers, makes explicit that programming is a highly cognitive skill and cannot be developed by rote learning of basic programming knowledge, syntax rules, and programming strategies. Much rather, sensitive instruments need to be developed to identify the individual needs of novice programmers in order to support them acquire the cognitive abilities and programming skills according to their current stage of reasoning. Such an approach is not only applicable to novice programmers but also to those experienced programmers that seek to become experts.

2.1.2 Block-based Programming Education

Not all researchers and educators have focused on training software engineers during programming education. Quite on the contrary, the influential book *Mindstorms* written by Seymour Papert [Pap80] and published in 1980 first introduces the notion of *computational thinking* as a valuable skill for children, which is best cultivated by learning how to program[3].

In this context, Papert also introduced the concept of turtle graphics to the text-based programming language `Logo`, which offers one or more virtual turtle agents

[3] At this stage, *computational thinking* has been introduced with a lesser pronounced political message compared to the article of Wing [Win06].

that can be programmatically controlled to draw shapes on the screen [vFP96]. Education-based use and research of the `Logo` programming language and its turtle graphics is still ongoing. For example, Forster et al. [FHSS18] report on an adaption of the programming environment for primary education as well as a reporting methodology of syntactic mistakes which facilitate the learners' autonomous recovery. Turtle graphics have been made available in many later programming environments. Nowadays, `Python` is one of the most well known programming languages that offerelem a turtle graphics module to support learning programmers.

In the following decades, the approach of early programming education to cultivate *computational thinking*, outlined by Papert, was refined in different ways. A refinement that opened programming education to a broad audience was to simplify the input interface used to interact with the programmable agents and construct programs. In such visual programming environments, programs are created not by textual manipulations of program elements but by different forms of graphical manipulations. The automata-based programming environment `Kara` [HNR01] is an example of the shift from text-based programming environments to alternate forms of interaction elements. Another example is the programming environment `Alice` [CDP03], offering a drag-and-drop interface while still supporting the object-oriented programming paradigm.

A subgroup of visual programming environments are block-based programming environments, which offer program elements in the form of graphical blocks that can be combined in pre-defined ways to construct programs. One of the most widespread block-based programming environments is `Scratch` [MPK$^+$08], with a pronounced body of research starting in 2008.

`Scratch` builds on Papert's legacy of `Logo`, with the key goal of introducing programming to those with no previous experience while encouraging self-directed learning. Maloney et al. [MRR$^+$10] summarize the initial design principles in their article regarding the user interface and the programming language.

Features and design considerations of the user interface of `Scratch` include [MRR$^+$10]: i) the single-window user interface that contains multiple panes to ensure that key components are always visible, ii) its liveness (there is no distinction between edit/run mode) and tinkerability (user can experiment with commands and code snippets regardless of their connections), iii) visual feedback on affected blocks during execution, iv) eliminate syntax errors (with only allowing certain blocks to fit together) and runtime errors (by making the blocks be *failsoft*), v) making data concrete by including visual monitors for variables and lists that show values and changes, and vi) striving to minimize the command set (e.g., by including different commands as single blocks with drop-down menus).

Features and design considerations of the programming language of `Scratch` include [MRR$^+$10]: i) the syntax is composed of blocks with different shapes, with the shapes dictating which blocks can be snapped together, ii) three first-class data types are supported (boolean, number, string) and variables are untyped and can contain either numbers or strings; with implicit conversions between types, iii) supporting the object-based paradigm with sprites as objects that encapsulate states (their variables) and behaviour (their scripts), with stamping and cloning as mechanisms to achieve class-like behaviour (without inheritance support), iv) the support for messaging between sprite objects in the form of broadcasting (supporting a one-to-many, loosely-coupled, and asynchronous model), v) future support of procedures (in the article, currently implemented as *user-defined blocks*), and vi) implicit multithreading enabled by the object and broadcasting model.

Kesselbacher and Bollin provide a thematic overview of research findings regarding the block-based programming language `Scratch` [KB19a][p. 2], presented in the next three paragraphs.

The group of Armoni, Mordechai, and Meerbaum-Salant has published a number of articles on this topic. Chronologically, first Meerbaum-Salant et al. [MSABA11] describe certain programming *habits* that are fostered by learning programming in Scratch and are contradictory with accepted software engineering practices: a bottom-up development process, and *extremely fine-grained programming* where solutions are programmed in a fine-grained, not a general way. Later, Meerbaum-Salant et al. [MSABA13] describe a taxonomy, incorporating both SOLO [BC82] and Bloom's [AKA+01] taxonomies, to measure the learning of computer science concepts with environments like Scratch. Armoni et al. [AMSBA15] conducted an experiment in which they investigate the effects of transitioning from Scratch programming to textual programming (C# or Java) in secondary schools, with regards to the acquired programming knowledge. They found that students familiar with Scratch needed less time to learn new topics, had fewer learning difficulties and achieved a higher cognitive level of understanding. At the end of the teaching process, there were no significant differences in the achievements, regardless of Scratch familiarity.

These results make a point in favor of teaching programming with learning environments like Scratch. But the articles also show that care must be taken to prepare the correct instructions for learning how to program. Other articles investigated specific learning interactions with the Scratch environment. Swidan et al. [SSH17] examined how Scratch programmers name their variables and procedures and found that they prefer longer identifier names compared to names used in textual programming languages. More than one third of the analyzed projects use identifier names that contain spaces, a feature unique to Scratch as opposed to textual programming languages. The authors argue that this can hamper the transition from Scratch to a textual programming language. Grover and Basu [GB17] describe an experiment made with middle school students and report on misconceptions of loops, variables and boolean logic used in Scratch programs. The authors show that, even after completing an introductory programming course with Scratch, the students are unfamiliar with the use of variables, and have misconceptions with how loops and boolean logic operators work.

Also the presence of code smells in block-based program scripts has been examined. Hermans et al. [HSH16] report that more than 88% of the analyzed projects contain code smells, most frequently *lazy class*, *duplication*, and *dead code* smells. Hermans and Aivaloglou [HA16] conducted a controlled experiment with novice Scratch programmers to investigate the effects of code smells when comprehending Scratch programs. They found that *long method* code smells lead to a decreased system understanding, while *duplication* code smells lead to a decreased ease of program modification. These results imply that code smells already have an effect on Scratch program scripts. When transitioning students to a textual programming language, care must be taken to prevent the transfer of program construction strategies that lead to more *smelly* program code.

There are other articles not included in the summary of Kesselbacher and Bollin [KB19a]. An early account by Adams and Webster [AW12] of the use of `Alice` and `Scratch` to learn programming through games, music videos, and storytelling projects, covering programming camps in the years 2003–2011, reports that game projects best facilitate the use of programming concepts. Statter and Armoni [SA16] report on the use of `Scratch` to teach abstract thinking in secondary education, measured with pre- and post-tests and an examination of the final projects. They highlight that students often neglect higher levels in the abstraction hierarchy and tend to rush to the programming environment, which can be alleviated by incorporating the step of writing verbal descriptions of problems and of programs that are to be constructed.

Weintrop et al. [WW17, WKF18] performed a number of comparative experiments to assess the effects of the programming environment modality (text-based or block-based) on student learning, making use of the programming environment `Pencil Code` that supports seamless switching between block and text representation of program scripts. Their results include a greater increase in programming assessment scores for students using block-based programming environment, and they show that students using the block-based programming environment during learning also perform better on text-based representations of program scripts across all assessed programming concepts, find the concepts easier to use and are more confident in their abilities, compared to students learning with the text-based programming environment. A stark difference is the result that students learning with the block-based programming environment rate their further interest in programming courses significantly higher. Altogether, Weintrop et al. make a case for block-based programming environments to be the superior alternative in learning to program.

Price and Barnes [BP14] also investigated the specific effect of a block-based programming interface on novice programmers' learning compared to a text-based programming interface. To do so, the authors utilize the web-environment `Tiled Grace`, which supports block-based and text-based programming input and lets programmer switch the interface. The authors do not classify their work as learning analytics – yet in their approach, each student action in the programming environment yields a timestamped log entry containing the action as well as the current program code. The authors conduct an analysis for goal completion regarding a target program and evaluate a survey of difficulty ratings elicited from the novice programmers (understanding instructions, making a plan, constructing the program to compile and run, implement a solution, and debugging to find out what went wrong). The authors find out that the interface did not have an effect on the students' perceived difficulties. Regarding goal completion, students using the block interface outperformed students using the text interface, having a higher percentage of solved goals and a (significantly) lower time to complete most goals.

Swidan et al. [SHS18] report on a multiple-choice questionnaire with Scratch programming exercises designed to elicit programming misconceptions. The most common programming misconceptions of participating school students are regarding the sequential nature of executed code and regarding how many (and which) values a variable can hold at a time. They also report on children's open answers to better elicit misconceptions. For older children (study participants have been in the span from 7–17 years), the tendency to hold misconceptions decreases significantly. Moreover, previous experience in `Scratch` and other programming languages also decreases the tendency to hold misconceptions. Swidan et al. [SHS18] also report on `Scratch`-specific issues, namely that the lack of understanding of variables was also apparent in children's answers of how variable values are displayed (they identify the problem with the *lingual imperative sense* of the block `Say`), and that the block to set a variable to a value has a potential ambiguity regarding the direction of the assignment. Knowledge of the children's misconceptions and of stumbling blocks when learning `Scratch` provides a great didactical tool.

Moreno-León et al. [MR14, MLRRG15, MLRGHR17] report on a publicly available tool, *Dr. Scratch*[4], which supports automatic analysis of `Scratch` projects based on *computational thinking* concepts. With static block analysis, the projects are evaluated in the following concepts, with each being assigned a level between 0 (competence not demonstrated) and 3 (proficient competence level): abstraction and problem decomposition, parallelism, logical thinking (use of conditional blocks), synchronization, flow control (use of sequential and/or loop blocks), user interactivity, and data representation. They also report that the repeated use of *Dr. Scratch* leads to improvements in the assessed concepts, especially for students at the intermediate *developing* level (competence level of 2 in most concepts).

Dr. Scratch was used in other research, for example, in the study of Troiano et

[4] `http://www.drscratch.org/`

al. [TSA+19], reporting on an iterative assessment of the evolving code of students' game projects, and thereby facilitate their acquisition of programming skills. With a critical thought, they also report that the metrics of *Dr. Scratch* may promote code practices that are not recommended (*code smells*), achieve a state of saturation in some concepts (data representation, user interactivity), and develop without intermediate levels (logic). This depicts *Dr. Scratch* as a tool to promote the early development of programming skills but not as a tool to assess mastery.

Francisco et al. [GSH+18] report on a programming workshop that is designed to teach software engineering practices through `Scratch` to 10–12 year old students with no prior experience. They evaluate the students' final projects on a rubric that assesses the following parts of the programming project: the fraction of sprites that control themselves, the fraction of blocks that have a single purpose, the documentation, the fraction of reachable blocks, the fraction of items with appropriate names, the changes from sample projects, and the project novelty in the context of their workshop. The authors compare their rubric scores with *Dr. Scratch* scores, which yields a weak positive correlation, showing that their rubric scores measure slightly different concepts. They map their rubrics to software engineering practices and conclude that most of the practices can adequately be presented to their target audience. Notably, they conclude that the following practices have not been sufficiently assimilated by the students: the adequate use of names, documentation, and pattern-based design and implementation.

Besides `Scratch`, another block-based programming environment that is actively researched is `Blockly`, which is a framework to create block-based programming environments. In a format of lessons learned without providing supporting data, Fraser [Fra15] presents a number of design considerations that should be applicable to all block-based programming environments, visualized in Table 2.1.

Some of these considerations directly translate into didactical considerations when using block-based programming environments in education and training (1–3, 5, 7, 10), and can easily serve as the basis of comparative studies of programming interfaces.

To summarize, different research groups have conducted studies to investigate the benefits and drawbacks of the use of block-based programming environments in order to learn programming [AMSBA15, BP14, WW17, WKF18]. Armoni et al. state that '*if experience with Scratch improves learning "only" of difficult concepts, that would more than justify its use*' [AMSBA15, p. 12]. Fraser puts it this way: '*How long this block-based programming period should last for students is hotly debated, but that it is temporary is not debated*' [Fra15, p. 50]. There is a consensus that the use of block-based programming environments, at some point in education, is beneficial. However, studies have also shown that block-based programming environments potentially have unintended side effects in the form of bad habits and code

Tab. 2.1: 10 lessons learned regarding the design of block-based programming environments [Fra15].

10 Design Points for Block-Based Programming Environments

1. Blocks representing conditionals and loops should be distinct by color and category so as to not confuse learners.	2. Visual decorations are necessary to distinguish connected blocks of the same colour.
3. Improve visual placement of overlapping blocks that are not connected (to further prevent syntax errors.	4. Transitive connections of large stacks of blocks should be implemented carefully.
5. Blocks that allow nested sub-stacks (so called `C` shaped blocks like loops) should visually inform a new user that any number of blocks can be fit inside.	6. Documented block images should be solved with *readonly* executions of the block-based programming environment to resolve colour, language and version differences.
7. Arrows representing the direction of turns in a circular fashion are most often understood correctly by learners.	8. Instructions have been programmed contextually-aware to guide new learners when first using the block-based programming environment.
9. A sense of code ownership is important for learners, for example facilitated by providing their solution for an exercise as the starting point for the next.	10. Block-based programming is only temporary, and should facilitate moving to text-based programming.

smells [MSABA11, MR14, AH16, HA16, HSH16, SSH17, SHS18], which may transfer to text-based programming. Specific research areas to move the field forward are how to identify bad habits in block-based programming and to prevent students from forming them, and how to easily transition from block-based to text-based programming.

2.2 Dimensions and Hierarchies of Programming Skills

Skills in a specific domain represent one of the three factors that affect task performance, with the other two factors being task-related motivation (the willingness to perform) and domain knowledge (knowing facts about how to perform) [BSD14]. The domain of programming includes many different skills which novices should learn in order to perform programming-related tasks, such as writing, comprehending/understanding, and debugging program code. As there is seldom an inherent scale associated with a specific skill, frequently the performance of novices and experts is measured and subsequently compared. In this context, Weinert [Wei99] offers two definitions that help to substantiate the terms of `expert` and of `skill`.

Definition 2.1 (Expert)**.** An *expert* is one possessed of particular (usually learned) proficiency in some branch of science, sport, art, or industry.

Definition 2.2 (Proficiency)**.** *Proficiency* is an ability of a certain degree, usually of a high degree, that is necessary for the performance of a task or the involvement in a vocation.

Definition 2.3 (Skill)**.** *Skill* (b) is an ability to perform complex motor and/or cognitive acts with ease, precision, and adaptability to changing conditions.

Following these definitions, expert programmers are individuals that have learned the (cognitive, domain-specific) skills necessary to perform programming tasks. They have learned them to such a high degree that they are capable of performing them with ease, precision, and in changing conditions. Conversely, novice programmers usually lack these qualities: they struggle with applying programming skills with high precision and in changing conditions and see performing programming skills like hard work [Lis16]. These definitions help in characterizing the features of novice and expert programmers. Still, they do not help in measuring or comparing programming skills or pave the way for individual support of skill acquisition.

A standard procedure to assess programming skills is to use proxy variables such as experience [BSD14]. In this regard, Siegmund et al. performed a number of experiments and literature surveys [FKH$^+$12, SKL$^+$14]. Additionally, Siegmund et al. [SS15] identify a total of 39 confounding parameters that affect the results of measuring program comprehension – it can be argued that the identified confounding parameters also apply to measuring other specific programming skills. An alternative procedure is to measure task-based performance and model the level of programming skills based on performance. See Bergersen et al. [BSD14] for an example instrument to measure programming skills (skills in this study being writing, modifying, and debugging Java program code).

The goal of this section is to outline explanations for differences in performance between novices and experts, which cannot be given with the frequently used dimensions of experience and task performance. Such explanations are key to enabling individual support of skill acquisition. This section consists of two subsections with the following content. First, dimensions (other than experience and task performance) that can be used to distinguish novice and expert programmers are explored. Second, a hierarchy of programming skill development, based on the psychological framework of neo-Piagetian stages of development, is presented.

Tab. 2.2: Comparison between novice and expert programmers regarding their mental model of the program structure.

Mental Model of Program Structure	
Novices	***Experts***
Lack mental model [Win96, SMD08]	Apply mental models as needed [Win96, SMD08]

2.2.1 Dimensions to Distinguish Novice and Expert Programmers

Studies that investigate differences between novice and expert programmers quantify those differences in various specific programming skills and infer educational implications have already been performed since the early 1980s [SE84, Sol86, Dét90, Win96]. A review of the first two decades of literature regarding the learning and teaching of programming has been conducted by Robins et al. [Rob03], focusing on aspects of novice teaching and learning.

In the latest broad literature review of introductory programming, Luxton-Reilly et al. [LRSA$^+$18] report that about 30% of the investigated 1666 publications primarily focus on the students in the process of introductory programming. It can be said that supporting novice programmers has always been one of the main aspects of the field of computing education.

As described in the introduction to this section, it is important to look at dimensions in which novice programmers lack proficiency to perform in programming tasks (i.e., they are distinguishable from expert programmers). Explaining and interpreting differences in such dimensions is a step towards providing individual support for novice programmers.

In the following, dimensions on which such an assessment is possible are explored. The dimensions have been selected based on the differences between novices and experts described by Winslow [Win96], and additional findings of other sources have been filled into the fitting dimensions. This literature review is by no means exhaustive and much rather servers to develop an informed view of observable effects when discriminating novice and expert programmers. These observable effects, when combined with the theoretical frameworks of Section 2.3, will serve to contextualize the studies, findings, and implications described in this thesis.

The first considered dimension is how programmers construct and access a mental model of the program structure, with an overview visualized in Table 2.2. There is one axis on which experts and novices can be distinguished. While novice programmers generally lack mental models and do not have stable strategies on how to form a mental model of an unknown software system, expert programmers apply mental models as needed and can more easily form mental models during program comprehension [Win96, SMD08]. While not directly comparing participants with

Tab. 2.3: Comparison between novice and expert programmers regarding their programming knowledge.

Application of Programming Knowledge

Novices	***Experts***
Fragile knowledge, struggle to apply knowledge [Win96, LAMJ05, SMD08]	Hierarchical knowledge, can apply knowledge [Win96, SMD08]
	Better syntactical and semantical knowledge [Wie86, Win96, SKL$^+$14]

Tab. 2.4: Comparison between novice and expert programmers regarding their programming strategies.

Problem Solving Strategies

Novices	***Experts***
Neglect specific strategies [Win96]	Hierarchical development of subgoals, better tactical and strategical skills [Win96]
Use general problem solving strategies [Win96, SMD08]	Recognize problems with known solutions, or use general problem solving [SE84, Wie86, Win96, SMD08]
Struggle to divide functionality in programming parts [LAMJ05, dRWT06]	Application of programming plans [SE84, dRWT06]

different programming experience (measured in years), Sillito et al. [SMD08] investigate how participants explore an unknown software system and form a mental model of the system. They summarize their findings with a categorized catalog of questions that can guide system comprehension.

The next considered dimension is how programmers access and apply programming knowledge, with an overview visualized in Table 2.3. In this dimension, there are two axes on which differentiation is possible. The first one is that novice programmers generally have fragile knowledge and also struggle to apply knowledge. At the same time, expert programmers can apply their knowledge in a systematic, hierarchical form [Win96, SMD08]. The lack of knowledge of novice programmers was also verified by a survey eliciting novice programmer difficulties as perceived by students and educators study, conducted by Lahtinen et al. [LAMJ05].

The second axis is that expert programmers generally have better syntactical and semantical knowledge [Wie86, Win96, SKL$^+$14]. This axis is in line with the expected high degree of proficiency as described in the definition of experts. This axis

Tab. 2.5: Comparison between novice and expert programmers regarding their programming approaches and comprehension approaches.

Programming & Comprehension Approach	
Novices	***Experts***
Approach through syntax and control structures [WS83, CS90, Win96, BT06, SMD08]	Approach through data structures and objects [WS83, CS90, Win96, BDW98, BT06, SMD08]
Line-by-line bottom-up approach [WS83, Win96]	Approach with algorithms [WS83, Win96] and top-down [BDW98]
Struggle to understand larger entities of a program [LAMJ05, BT06, MSG+18]	Faster acquisition of detailed systematic view of software system [CW99, CW01, BDW02, LMH+16, MSG+18]

Tab. 2.6: Comparison between novice and expert programmers regarding their programming efficiency.

Programming Efficiency	
Novices	***Experts***
Higher effort during software maintenance [RSS00]	Faster and more accurate in a variety of tasks [Wie86, Win96, MSG+18]
Can highly benefit from support of tools [RDT+07, YKM07] and techniques [AGDS07, HAES10]	

is confirmed by Wiedenbeck [Wie86] in a study on the organization of programming knowledge of novices and experts and by Siegmund et al. when exploring how to model programming experience, which they define as '*the amount of acquired knowledge regarding the development of programs, so that the ability to analyze and create programs is improved*' [SKL+14, p. 1303].

The next considered dimension is how programmers apply strategies during programming, with an overview visualized in Table 2.4. Three axes have been identified in this dimension. The first one is that novices neglect specific strategies when solving domain-specific problems, while experts generally have better tactical and strategical skills while also being able to hierarchically develop subgoals for domain-specific problems [Win96].

The next axis is concerned with the problem solving strategies used by programmers. While novice programmers use general problem solving strategies [Win96, SMD08], experts can recognize problems with known solutions or use general problem solving as applicable [SE84, Wie86, Win96, SMD08]. In this context, Soloway and Ehrlich introduce the concept of programming plans, which are program fragments that represent stereotypical action sequences in programming internalized by experts [SE84].

The last axis is concerned with the decomposition of program parts. While novice programmers struggle to divide a functionality to be implemented in programming parts [LAMJ05, dRWT06], expert programmers are capable of applying their internalized programming plans and thereby, almost naturally, decompose their programs in functional parts [SE84, dRWT06].

The next considered dimension includes differences in the approaches to programming in general and differences in the task of program comprehension in detail. An overview is visualized in Table 2.5. Three axes have been identified in this dimension. The first one captures that novice programmers approach different programming tasks through syntax and control structures, verified in a variety of contexts, including program design [WS83], writing program code [Win96], code maintenance [SMD08], and code comprehension [CS90, BT06]. Expert programmers, on the other hand, approach similar tasks through data structures and objects [WS83, CS90, Win96, BDW98, BT06, SMD08]. One could argue that novice programmers cling to singular, specific elements without regarding the relations between those elements, while experts can incorporate element relationships to work with relational understandings (interpretable with the SOLO taxonomy [BC82]).

The next axis encompasses the direction of programming and comprehension approaches. Novice programmers tend to approach programming tasks line-by-line and with a general bottom-up approach, again first focusing on singular, specific elements [WS83, Win96]. Expert programmers approach programming tasks with known algorithms [WS83, Win96] and a general top-down approach, first focusing on high-level understandings of entities and their relations [BDW98].

The last axis captures how software systems are comprehended. Novice programmers struggle to understand larger entities of a program, as verified with surveys [LAMJ05] and in comprehension and debugging studies employing eye-tracking methodologies [BT06, MSG+18]. Experts are capable of acquiring a detailed, systematic view of a software system and do so faster and more correct compared to lesser experienced programmers, as has been shown in debugging and program comprehension tasks [CW99, CW01, BDW02, LMH+16, MSG+18].

The last dimension concerns the efficiency of programmers in relation to different programming tasks, with an overview visualized in Table 2.6. A basic evaluation of this dimension is fruitless: as identified in the section introduction, novice programmers lack the ability to solve programming tasks with high precision. Because of this, the two identified axes focus on a specific aspect of efficiency. The first axis is concerned with efficiency in the sense of time and effort. Ramanujan et al. have shown that individual programmer characteristics have an effect on software maintenance effort: students with lower programming skills (in this study, measured as semantic knowledge and programming experience) have a significantly higher effort in software maintenance tasks [RSS00]. On the other hand, a number of studies already described in this section have shown that experts are faster (and more accurate) in various programming tasks like program comprehension and debugging [Wie86, Win96, MSG$^+$18].

The second axis only features a characterization of novice programmers, but an essential one at that. Different studies have shown that novice programmers can highly benefit from support in the form of tools (as an example, the tool-supported comprehension of UML diagrams has been investigated by Ricca et al. [RDT$^+$07] and Yusuf et al. [YKM07]), and in the form of techniques (as an example, effects of pair programming with regard to programming experience have been studied by Arisholm et al. [AGDS07] and Hannay et al. [HAES10]). Of note is that the inverse characterization is not apparent: expert programmers can also benefit from the support of appropriate tools and techniques. Much rather, the interpretation should be to support a novice programmer's skill acquisition with all possible tools and techniques, which could be different ones compared to those that support experts during task performance.

2.2.2 A Neo-Piagetian Hierarchy of Programming Skill Development

The dimensions on which the programming skills of novice programmers and expert programmers can be gauged, presented in Section 2.2.1, feature a number of blind spots important for programming education. First, they resemble a collection of baseline studies that report on the effects of increased knowledge, skills, and experience in programming, but do not explain the resulting gap between novices and experts. Second, there is no indication on how to bridge those gaps in the identified dimensions, apart from an increase in training and experience. Third, and most importantly, there are no identifiable stages or steps of skill development between novice programmers and expert programmers, impeding the formalization of evaluation and feedback of individual learners.

Within a neo-Piagetian theoretical framework of domain-specific cognitive development, adopted by Lister [Lis16], the issues mentioned above can be explained more specifically, and aspects of those issues have already been solved as well. In this section, the neo-Piagetian stages of cognitive development in the context of programming education, as established by Lister [Lis16], are described. A summary of theoretical, psychological approaches in a neo-Piagetian framework is given in Section 2.3.1.1.

Lister first formulated his research program with a neo-Piagetian framework of developmental stages in 2011 [Lis11b]. The framework is motivated by the model of *overlapping waves* formulated by Siegler [Sie96], meaning that in the domain of programming, a student can make use of different, gradually more advanced ways of reasoning depending on the problem, their ability, and the context of the situation [Lis16]. Lister also transfers the notion of microdomains to programming: novice programmers separately learn programming concepts (e.g., variables or loops) as they are introduced and do so in a cyclical process of assimilation and accommodation while regressing to lower stages of reasoning as needed [Lis16].

A key skill to describe stage differences in the framework of Lister is the ability of *code tracing*, which is the ability to '*systematically, manually execute a piece of code*' line by line [LAF$^+$04, p. 120]. For novice programmers, *code tracing* is a challenging cognitive ability, as it requires them to systematically and correctly apply basic programming concepts as would be carried out by the executing computer[5] while keeping track of variable values, loop iterations, and more. Code tracing is often associated with graphical sketches created by students, which visualize the process of tracing. Cunningham et al. [CBEG17] provide a detailed analysis of different sketch types and their relation to a student's ability to correctly answer code reading problems.

Since the introduction of neo-Piagetian developmental stages of programming, considerable evidence of the transitions between *sensorimotor*, *preoperational*, and *concrete operational* stages of cognitive development in the domain of programming has been reported. One of Lister's key points is that, by interpreting the collective findings, programming educators should keep in mind that their students have to transition through these phases as a normal process of development [Lis16], and that it cannot be assumed that all students perform on *concrete operational* levels. For each stage, the relevant studies in Lister's research program are given. Studies that predate the formulation of neo-Piagetian developmental stages of programming but which follow comparable ideas of a cognitive process of development are referenced separately. Table 2.7 provides an overview of the core features of each of the developmental stages.

The first stage of *sensorimotor* reasoning is the least mature stage. Novice programmers in this stage have trouble tracing (mentally executing) program code line by line, which is why Lister also calls this stage *pre-tracing* [Lis16]. This stage represents students who just learn to program: they have misconceptions of basic programming concepts, cannot reason about program code (as Lister puts it, '*they tend to see static text not a dynamic program*' [Lis16, p. 8]), and, as a result, cannot trace program code consistently and correctly. A longitudinal case study investigating a novice programmer's development through the neo-Piagetian stages is reported by Teague and Lister [TL14b]. In the think aloud interviews reported in this study, the fragile nature of understanding of programming concepts (often vari-

[5] In another framework, this process is called to step through the process of the *notional machine*, which lays bare the students' mental models [Sor13]

Tab. 2.7: Features of neo-Piagetian development stages of programming (see also [Lis16]).

Neo-Piagetian Developmental Stages of Programming

Sensorimotor (pre-tracing)	***Preoperational (tracing)***
Misconceptions of basic concepts Code reasoning not possible No consistent, correct code tracing	Code reasoning by induction Non-iterative inference not possible Reliable code tracing
Summary: Novices in this stage struggle with code tracing, and basic programming concepts in general	Summary: Novices in this stage are capable of successful code tracing and inductive reasoning

Concrete operational (post-tracing)	***Formal operational***
Code reasoning by code reading Abstract, transitive reasoning possible Writing code for purpose possible	Hypothetico-deductive reasoning Abstract reasoning and reflection Reasoning about unfamiliar problems
Summary: Programmers in this stage are capable of abstract reasoning about familiar situations	Summary: This stage covers expert reasoning on unfamiliar problems, which is not covered in research so far

able, ridden with misconceptions, and manifested in incorrect tracing of sequential program code) can be seen. There is a number of studies carried out by Lister or co-authored by Lister that can be ascribed to investigating the sensorimotor stage: [LAF$^+$04, VS07, LWRL08, LFT09, VTL09].

The next stage of *preoperational* reasoning is already a considerable developmental step. At this stage, novice programmers consolidate and stabilize their conceptions about how programs work, resulting in their capability to reliably and correctly trace program code – this stage is also called *tracing* [Lis16]. However, novice programmers at this stage are only capable of a single type of reasoning about program code: reason by induction. This is because their reasoning is tied to code tracing with explicit values. In order to infer the purpose of a program, they typically trace the code with multiple sets of input values (assigned to variables), and infer the purpose based on relations of inputs and observed outputs. As novice programmers also employ this type of reasoning when writing their own code, traditional scaffolding intended to aid them is often not effective (as they cannot reason about code the same way compared to instructors), and Lister states that preoperational programmers should be closely supervised during writing program code [Lis16]. A number of studies have been devoted to this stage, the following *concrete operational* stage, and what constitutes the gap between those two stages [Lis11b, CTAL12, TL14a, TL14c, TL14b, Lis16]. Again, there is a number of studies carried out by Lister or co-authored by him that can be ascribed to investigations regarding the preoperational stage: [LAF$^+$04, VS07, LWRL08, LFT09, VTL09].

The third stage of *concrete operational* reasoning is finally the stage when programmers can reason about abstractions of code, also called *post-tracing* or *abstract tracing* stage by Lister [Lis16]. This abstract reasoning is restricted to familiar and real situations, which can be improved by letting the students write code and thereby develop their programming skills on the way to the next stage. Abstract tracing of a *concrete operational* programmer is the ability to reason about code while reading it and maintaining a set of '*algebraic-like constraints on possible values in each variable*' [Lis16, p. 14]. This defining quality sets this stage apart from the previous stage: manual traces with sets of input values are not required for microdomains for which *concrete operational* reasoning is already possible. Besides this quality, *concrete operational* reasoning is traditionally defined with the ability to reason about quantities that are conserved and processes that are reversible [Lis11b]. In the context of programming, Lister identifies the writing of program code that reverses the effect of given code as a prime example to assess *concrete operational* reasoning [Lis16]. For additional descriptions, example tasks, and findings related to the transition from *preoperational* to *concrete operational* reasoning, see the following studies: [Lis11b, CTAL12, TL14a, TL14c, TL14b, Lis16]. Also, there is a number of preliminary studies (in the context of the neo-Piagetian reasoning framework) carried out by Lister or co-authored by him that can be ascribed to investigating the concrete operational stage: [LWRL08, LFT09, VTL09].

The last stage of *formal operational* reasoning represents the level of cognitive abilities on which an expert programmer performs [Lis16]. Casually described by Corney et al., this type of reasoning is '*what competent programmers do, and what we'd like our students to do*' [CTAL12, p. 79]. Compared to the former stage, formal operational reasoning includes hypothetico-deductive reasoning (which Lister identifies as an important reasoning skill during debugging [Lis16]), abstract and reflective reasoning (moving from abstract to concrete forms of thought, as put by Corney et al. [CTAL12]), and reasoning about unfamiliar situations. This stage is not covered in the research program of Lister. As such, this neo-Piagetian framework does not help in bridging the gap between somewhat experienced programming students capable of *concrete operational* reasoning and expert programmers on the stage of *formal operational* reasoning.

2.3 Psychological Frameworks in Programming Education Research

The approaches of this thesis, the interpretation of the findings, and the contextualization of theoretical and practical applications are embedded in two psychological frameworks: the *neo-Piagetian* theory of cognitive development and the theory of *cognitive load.* This section includes a general description of both psychological frameworks as well as their use in computer science education and programming education in particular.

2.3.1 Neo-Piagetian Theory of Cognitive Development

A widespread epistemology of cognitive development is the theory of Jean Piaget and the work building on his legacy called neo-Piagetian research. In this section, the basic concepts of Paiget's theory of cognitive development are discussed before turning to neo-Piagetian theoretical developments that advance research on cognitive development. The section is concluded with applications of (neo)-Piagetian theory in programming education.

2.3.1.1 The Roots: Jean Piaget's Theory of Cognitive Development

The research program and encompassing theory of Jean Piaget lie at a unique intersection between psychological empirical research and epistemology: there is a heavily developed theoretical framework of assumptions in which his studies of the psychology of human development is grounded [MCS09]. In this section, the focus is on the theoretical framework, which provides a foundation for the following neo-Piagetian developments.

The theory of Piaget is based on the constructivist view that knowledge is neither pre-existent in the world or the subjects nor is it copied from external models, it is rather constructed by each subject through the process of acting on objects. The construction of knowledge and the accompanying development of cognitive structures is tied to a logical system of reasoning, one example of which is the algebraic Klein's four-group [Inh92].

The cognitive development happens in phases of transformations of cognitive structures, which at some developmental point transforms the subject's cognitive process to the next stage. According to the Piagetian theory, the criteria of development stages are as follows [Inh92]:

- Each stage consists of a period of formation and attainment, the former understood as genesis and the latter as progressive organization of a composite structure of mental operations.
- Each structure constitutes the attainment of one stage as well as the beginning of the next stage.
- While the age of attainment can vary depending on different factors, the order of succession of stages is constant.

- The transition from one stage to another follows a process of integration; preceding structures become a part of later structures.

Based on these criteria of development stages, Inhelder describes three successive operational structures of cognitive development, each with substages. Those structures are: *sensory-motor* operations, *concrete thinking* operations (with the *preoperational* and *concrete operational* stages of development as substages [Lis16]), and *formal thinking* operations. In the following description, I use the stage names established by Lister [Lis16] to help facilitate the contextualization of my findings in (neo)-Piagetian terms.

The first *sensorimotor* stage is concerned with the development of children's own motor senses and is typically ascribed to children until the age of two years. In this stage, an understanding of object permanence and of the inversion of motor displacements is developed.

The second *preoperational* stage forms the first major substage of *concrete thinking* operations, ascribed to children of ages two to seven. This substage is characterized by a process of elaboration of mental operations, during which concrete thought processes are still irreversible.

The third *concrete operational* stage forms the second major substage of *concrete thinking* operations, ascribed to children of ages seven to twelve. This substage is characterized by a process of structuration of the previously elaborated mental operations, during which concrete thought processes become reversible and form a system of concrete operations.

The fourth and last *formal operational* stage encompasses the cognitive development of formal, abstract thought operations. This stage is characterized by a hypothetico-deductive level of thought and reasoning.

In Piaget's view, the developmental process happens through a child's interactions with the world, characterized by the functional processes of *assimilation* and *accommodation* [MCS09]. The process of *assimilation* captures the incorporation of new elements into existing schemes, while the process of *accommodation* accounts for the necessary modification of existing schemes to incorporate new observed features of objects or situations. The process that guides and encompasses all development is *equilibration*, consisting of assimilation and accommodation as well as cyclical formalizations (from abstraction until reflection) and organization/coordination of cognitive structures.

2.3.1.2 Advancing on Piaget's Legacy: Neo-Piagetian Theories of Cognitive Development

Piaget's theory of cognitive development, consisting of a dialectic view of development as well as the definition of stages in terms of logical competence, is not short on being criticized (e.g., a critical take on the Piagetian developmental stages as explanatory constructs [Bra92]), and has spawned a number of observations which do not conform the theoretical framework and need to be further explained. These observations include the problem of explaining the temporal displacements in the acquisition of logically equivalent concepts (*horizontal décalages*), the problem of adults' performance in logic tasks not adhering to the final stage of cognitive development, and the problem that concepts develop at different paces in different cultures [MGMS08, p. 13ff].

These observations led to the development of theories that build on the legacy of Piaget's theory of cognitive development while adapting the theoretical construct. Neo-Piagetian research shares a couple of similarities with the Piagetian theory [MGMS08, p. 38f], [RF09]:

- Cognitive development is framed in a constructivist approach.
- Cognitive development is divided into periods or stages that have qualitatively different characteristics.
- The learner's thinking increases in complexity from one stage to another. However, unlike Piaget tying the increase to logic, the increase in cognitive capacity corresponds to features of the information processing system.

Morra et al. provide a summary of starting points that gave birth to neo-Piagetian theories, incorporating various aspects of cognitive development [MGMS08, p. 22ff]. The first point, stemming from Piaget's idea of the *field of attention* of learners and interpreted in a new light, is the idea that cognitive development increases the capacity to process information, and therefore permits the solution of problems that have a higher information load. This idea is also present in the theory of cognitive load, described in Section 2.3.2. The second point is summarized in the key idea that, during cognitive development, the representation and organization of knowledge in learners' minds are developed in new and more abstract forms. The last point is concerned with learners' use and development of strategic and heuristic processes, namely a '*cognitive operation aimed at achieving a predetermined goal*', and meta-cognitive abilities to control their own strategic processes. In the rest of this section, five neo-Piagentian theories are briefly presented in order to illustrate the variety of approaches following the legacy of Piaget's theory of cognitive development. Note that most theories have dominantly been evaluated on children, but especially domain-specific skill development is very relevant to any age group. Because of this, the following descriptions are given within the context of an arbitrary *learner* without age connotation, knowingly using this term, albeit the Piagetian difference between *learning* and *development*.

An example of a structuralist approach to neo-Piagetian theoretical development, an approach that seeks out formalizable structures in order to deductively interpret the facts of observations is the concept of *cognitive systems* by Halford [Hal82, HWP98] [MGMS08, p. 91ff]. In the theory of Halford, *cognitive systems* consist of a problem, its symbolic representation (systems of symbols), and a set of correspondences between the problem and its representation. The complexity of systems of symbols is measured in terms of the complexity of its relations, with the main factor being the `n-ary` dimensionality of the relations. Regarding cognitive development, the theory is centered on the idea that, through developmental stages, learners are able to learn '*concepts, representations and mental abilities that require cognitive systems of increasing complexity*' [MGMS08, p. 93].

The skill theory of Fischer, called *constructive dynamic web* [FB06], is a neo-Piagetian theory that supports both synchronous as well as sequential developments within task domains and attributes development to a process of knowledge construction [MGMS08, p. 149ff]. In this theory, a skill is a '*capacity to act in an organized way in a specific context*'. The related cognitive structures are dynamic, actual organizations of systems of activity, in contrast to preexisting structures as described by Halford. Skills are hierarchical; lower level skills become subsystems when integrated into higher level skills. The dynamic nature of skills is given by the reciprocal influence upon each other in accordance with internal learner conditions and external situations. Because of this dynamic nature, skill acquisition is described as a nonlinear, dynamic growth in a constructive web. Variation in skill performance is tied to skill support, which differentiates a learner's performance on *functional level* (highest possible performance without support) to the performance on *optimal level* (highest possible performance with expert support for which a learner can manage the complexity). This concept is closely tied to Vygotsky's zone of proximal development [Vyg78]. Development is divided into tiers of qualitative changes (in turn: actions, representations, and abstractions), with cyclical levels of reorganization of cognitive structures (in turn: single actions/representations/abstractions, mappings, systems, and principles).

Other approaches of neo-Piagetian theoretical developments are centered on the idea of cognitive development as successive changes in (mental) representations. Mounoud specifies four types of knowledge organization that correspond to levels of development, which are characterized by the type of internal organization of contents [Mou93] [MGMS08, p. 230ff]: sensorial, perceptual, conceptual, and formal. Mounoud structures cognitive development as cyclical and recursive phases of adaption, in which two alternate processes are a dominant part of learning: first the decomposing of representations and components into elementary or partial actions, and second the integration or consolidation of knowledge by coordinating the single actions into larger units.

In contrast to Mounoud, Karmillof-Smith devises a theory of development by changes in representations applicable in specific domains [KS92] [MGMS08, p. 239ff]. In her theory, the learners' representations undergo a transformation from implicit to explicit, which is called *representational redescription*. In this frame, development is a process by which representations of representations are formed. These processes take place in each specific domain, also called in microdomains. This process of development can be described as passing from knowing how to do something while not perfectly understanding how the action is done to being able to consciously access most parts of the internal representations. This development makes flexible and creative behaviors possible. However, lower procedural levels never disappear and can always be accessed, depending on the problem.

Another theory of domain-specific cognitive development is proposed by Siegler, characterized by the metaphor of *overlapping waves* which represents that, at any point in development, cognitive processes are composed of multiple potential approaches [Sie96] [MGMS08, p. 125]. Various representations and strategies are available for use, and through cognitive development, the learners come to use them in an increasingly adaptive manner. According to Siegler, this adaptive choice of strategy is related to the conceptual structure and is formed during discovery, adaption, and consolidation of the used strategies.

2.3.1.3 Applying Neo-Piagetian Theory to Programming Education

Programming is a technique of formal and abstract problem solving. Unsurprisingly, Piagetian theory has already been applied to programming education over the course of many decades, with the goal of developing a predictor of programming ability.

Lister provides an overview of earlier literature that connects the Piagetian theory to programming education [Lis11b]. An example of an early account of the use of Piagetian theory to investigate the performance in programming education is given by Fischer [Fis86]. She employs a test instrument to distinguish between concrete and formal operational thought and reports a strong discrimination of high-performing students (grades of `B+` and higher, dominantly formal operational) and other students (lower grades, all are concrete operational). However, Cafolla reports another study that correlates students' performance in an introductory programming exam with their performance in a formal operational reasoning test, with a comparatively R^2 of 0.35 [Caf87].

Learning from these example studies, there seems to an unmeasured factor in the students' programming performance – only a small part of the performance variance can be explained with the use of Piagetian measurement instruments. Lister summarizes his reviewed studies, suggesting that the mixed findings result from the fact that Piagetian theory does not account for domain-specific cognitive development. He proposes a neo-Piagetian view of task-specific skill development: '*there is no reason to expect that a person's ability on a non-programming test of abstract* *reasoning should correlate with that person's ability at programming*' [Lis11b, p. 11].

Lister proceeds to ground his research in neo-Piagetian theory, developing an understanding and a body of evidence regarding the (overlapping) *sensorimotor*, *preoperational*, and *concrete operational* developmental stages in programming [Lis11b, CTAL12, TL14a, TL14c, TL14b, Tea15, Lis16]. An overview of the developmental stages in the context of programming education, as developed by Lister and Teague, is given in Section 2.2.2.

The research group of Lister is not the only one that recently applies neo-Piagetian theory to programming education. Aggarwal reports on a case study, assessing the code reasoning abilities (mental simulation, i.e., code tracing) of elementary schools children programming in the tile-based visual programming language Microsoft Kodu Game Lab [Agg17]. The neo-Piagetian developmental stages established by Lister and Teague are used to characterize student performance. This study marks the use of neo-Piagetian theory to assess visual, rules-based programming environments.

While still not generally used in programming education, neo-Piagetian theory provides a promising framework for research that seeks to explain programming and related skill acquisition as cognitive processes. In the context of this thesis, the idea of domain-specific, overlapping hierarchies of increasing cognitive capabilities to explain programming skills is the most important feature of the neo-Piagetian developmental stages, as established by Lister [Lis16] in a model of overlapping waves (related to Siegler [Sie96]).

2.3.2 Cognitive Load Theory

Cognitive Load Theory (CLT) is a theory that describes the load on (working) memory in relation to the structure of information that should be processed by learners. CLT identifies three areas of load that contribute to the overall load imposed on learners. It was first described in 1988 by Sweller [Swe88] and then updated in 1998 by Sweller et al. [SVP98]. While this thesis does not directly deal with instructional design, the findings are put in a learning and teaching context with key concepts of CLT in mind. In this subsection, I first summarize CLT and its relation to design principles for learning instructions before turning to applications in programming education.

2.3.2.1 Cognitive Load Theory and Instructional Design

The opening to the first full description of CLT sets the scene for a theory of cognitive processes that describes how instructional design can be improved to support learning, regardless of the domain: '*Considerable evidence indicates that domain specific knowledge in the form of schemas is the primary factor distinguishing experts* *from novices in problem-solving skills*' [Swe88, p. 257]. Sweller already identifies key elements that play a role during the acquisition of new skills and knowledge, namely the distinction between cognitive processes for learning and solving problems and the limited cognitive capacities that need to be managed [Swe88].

In a more refined description of CLT, Sweller et al. [SVP98] coin the terms now commonly associated with CLT:

- The different categories of load: *intrinsic*, *extraneous*, and *germane load* (see below)
- The learners' limited *working memory* and the need for instructional design to balance the categories of cognitive load so that learning can occur
- The cognitive structures of *schemas* which subsume single elements and other schemas and act as new, single elements for the working memory, thereby representing a key aspect of expertise
- The factor of *element interactivity* to assess learning elements in their cognitive load

As described by Paas et al. [PRS03], CLT provides support to consider both the structure of information and its interactivity, as well as the cognitive architecture that allows learners to process that information in order to develop instructional designs fit for a variety of learning situations.

CLT distinguishes two main categories of cognitive load that contribute to the overall load in a learning situation. The first category is *intrinsic load*, which is intrinsic to the material being learned. The *intrinsic load* can only be changed by lowering the interactivity of the elements of the task, therefore changing the task itself, such that less advanced schemas (containing a smaller number of single elements) are needed to solve the task [PRS03].

The second category is *extraneous load*, which is not intrinsic to the material that should be learned but is rather unnecessarily imposed on learners in the form of information and learning activities. An important point is that *intrinsic load* and *extraneous load* are understood as additive quantities, which means that an increase in the former (a higher element interactivity) is only possible by reducing the latter [PRS03]. Sweller also describes *extraneous load* as '*element interactivity that can be reduced without altering what is learned*' [Swe10, p. 125].

Working memory (the total capacity that can be used to solve tasks, acquire more refined schemas, and learn material) is affected by the intrinsic nature of the material (represented in the *intrinsic load*) and by the manner of presented material and required student activities (represented in the *extraneous load*) [SVP98]. For instructional design, the goal is to decrease the load on working memory by adapting the *intrinsic load* with regard to the learners' available schemas (e.g., limiting the element interactivity of tasks according to the learners' skills and knowledge) and to reduce *extraneous load* by adapting the instructional procedure (omitting materials and activities that hinder what should be learned [Swe10]) [PRS03].

The result of effective instructional design is to spare learners' cognitive resources, which can be used to refine their schemas. Previously, this would have been called '*an increase in germane cognitive load*' [PRS03, p. 2], which has been falling out of favor [Kal11] and is currently described as '*purely a function of the working memory resources devoted to the interacting elements that determine intrinsic cognitive load*' [Swe10, p. 126].

Since the inception of CLT, numerous effects of cognitive load in instructional design have been reported and researched. A comprehensive summary of cognitive load effects is given by Sweller et al. [SvMP19].

2.3.2.2 Applying Cognitive Load Theory to Programming Education

Cognitive load theory (CLT) has handily found its way into programming education, an area in which learners struggle because of high element interactivity and where compound tasks are the norm and not the exception [LRP17, LRBC$^+$18]. Selected publications reporting on the use of CLT in programming education or embedding their argumentation in CLT are highlighted in this section.

Lister provides a short but concise argument of the use of the theoretical framework of cognitive load in programming education, identifying the task of programming as '*an intrinsically cognitively demanding task*' [Lis11a, p. 22]. While extrinsic load can be alleviated by tools like block-based programming languages (which transform the need for textual manipulation to a graphical form and prevent syntax errors during programming), intrinsic cognitive load can only be lowered by didactical and pedagogical adaptations. This important distinction of the benefits and shortcomings of such tools sets the scene for computer science education research.

Hundhausen et al. embed different learning theories in their process model of learning analytics research for computing education, one of which is CLT [HOC17]. The authors are concerned with an IDE-based process model, with the key question of '*How can an intervention designer make choices along these dimensions [content, presentation, and timing] so as to design effective interventions—that is, interventions that effect positive changes in the student learning processes, outcomes, and attitudes?*' [HOC17, p. 15] While embedded in learning analytics research, their key question is actually applicable to the design of any learning intervention. The authors state that learning theories need to be included when designing interventions, not only IDE-based interventions. With regard to CLT, they derive three key design principles that should shape the design of in-IDE interventions:

1. Information relevant to task completion should be placed in the immediate context of the task (CLT1 in the article).

2. Redundancy of presented information during programming tasks should be removed (CLT2 in the article).

3. Use multiple channels (like visual and auditory channels) to present complementary information (CLT3 in the article).

With these design principles, the authors show that CLT motivates meaningful changes in intervention design.

Two studies report on a questionnaire to measure the categories of cognitive load in introductory programming education lessons [MDG14, ZDH^{+}20], with reliable (replicated) findings of measuring the factors *intrinsic load* and *extraneous load*. Applying the authors' questionnaire, the amount of load imposed on the learners by a specific programming learning intervention can be measured. Both authors state that the measurement of another engagement-related construct is necessary to provide a richer picture.

The demand is on instructional design in programming education to minimize *extraneous load*, control *intrinsic load* and make room for schema acquisition and learning, which can be realized in different ways. With the theoretical framework of CLT, domain-specific education research can be situated in a context of learning psychology that promotes avenues for further understanding of the learning process to guide research and, ultimately, education. In the context of this thesis, the most important feature of CLT is the idea of *schema* acquisition and the resulting decrease in a problem's *intrinsic load* as a key aspect of expertise.

2.4 Summary of Theoretical Background

The previous sections provide a theoretical fundament upon which the approaches and findings of this thesis are built, including the contemporary state of programming education as a part of computer science education (Section 2.1), specific dimensions and hierarchies with which programming skills can be quantified (Section 2.2), and a summary of psychological frameworks important for this thesis and their applications in programming education (Section 2.3). Summarizing the theoretical background, this section includes the identified gaps in programming education research relevant to this thesis (Section 2.4.1) as well as an answer to the first research question (Section 2.4.2).

2.4.1 Gaps in Research

In this section, I summarize the state of programming education by highlighting two gaps in research that prove of importance to this thesis. The first gap is that researchers in programming education focus on novice programmers and how to support their programming skill acquisition, while software engineering research focuses on expert programmers and their mastery and application of industrial-level programming skills. There is a literal gap to be bridged – how can novice programmers be supported to improve their programming skill at any level, especially at a level at the end of introductory programming courses (when they might reach the peak of novice programming)?.

To clarify this gap, the terms of **novice programmers** and **expert programmers** need to be defined more clearly. In the course of this thesis, they are interpreted within the framework of Lister [Lis16]. **Novice programmers** include students of the reasoning stages *sensorimotor* and *preoperational*, and also (resulting from the overlapping waves interpretation) students beginning the reasoning stage *concrete operational*. In short, **novice programmers** are those who struggle to reason about code deductively (not by tracing) and write purposefully code on their own. On the other end of the spectrum are **expert programmers**, who are firmly at the stage of *formal operational* reasoning.

This exemplifies the gap and leaves a portion of the developmental stages uncovered: students firmly in the stage of *concrete operational* reasoning who can deductively reason about code and write purposeful code but cannot do so in an abstract manner for unfamiliar situations (the *formal operational* stage is still out of reach for them). In this thesis, such students are called **experienced programmers**, as they have developed beyond novice programming skills but still have to develop into *expert programmers*.

The second gap is caused by the ubiquitous use of computing systems and the rise of easily accessible programming environments dedicated to teaching programming to K–12 students, as described in Section 2.1.2. While novice programmers have mostly been undergraduate students in previous decades, educational systems worldwide gear towards an early inclusion of computing education and thereby also programming education. A resulting effect is that many school pupils come in contact with programming before the usually anticipated entry point of undergraduate studies, leading to highly heterogeneous groups of 'novice programmers' with varying previous experiences in programming.

The gap can be formulated as two questions. How can novice programmers of varying levels of programming skills be supported in introductory programming courses? How do the varying previous experiences in programming (computational thinking, block-based and/or text-based programming) affect a novice programmer's acquisition of programming skills? This gap again highlights the need in programming education for individual assessment and feedback of students' learning progress and stresses the need for automatic evaluation approaches.

The approaches of this thesis are designed to support the programming skill acquisition of **novice programmers** (in line with programming education research) while also potentially supporting **experienced programmers** in their individual developmental path towards *formal operational* reasoning. This is achieved by embedding the approaches in a learning analytics framework to facilitate individual assessment, feedback, and support in an automated way, potentially even during programming tasks. This thesis thereby addresses some parts of the identified gaps in the research of programming skill acquisition. The learning analytics framework is described in Chapter 3.

2.4.2 Distinguishing Novice and Expert Programmers

After an overview of the state of research, an answer to the first research question of this thesis can be given to conclude the theoretical background. The first research question is concerned with documenting the distinguishing characteristics between programmers of varying levels of programming skills, with facilitates a grounded approach to improving the individual acquisition of programming skills. The first research question is:

RQ1. What distinguishes novice from expert programmers?

Section 2.2 embeds the literature research that yields an answer to this research question of the thesis – a summary is given here. The answer consists of two steps, comprising dimensions to distinguish novice and expert programmers on the one hand while also arranging the novice programmer's development in an (overlapping) sequential hierarchy on the other hand.

First, performance in domain tasks can be measured by three factors (domain skills, domain knowledge, and task motivation) [BSD14], with the first two being relevant to this thesis within the domain of programming. Experience is the most prominently used proxy variable when assessing programming skills [BSD14], but it does not help in individual assessment, feedback, and support of skill acquisition. In this thesis, *skill* is understood as '*an ability to perform complex [...] cognitive acts with ease, precision, and adaptability to changing conditions*' [Wei99, p. 35]. The research question, put into this context, now asks: how can novice and expert programmers be distinguished based on cognitive, domain-specific abilities?

Starting with dimensions of differences compiled from psychological studies in various domains [Win96], the following dimensions to distinguish novice and expert programmers based on cognitive, domain-specific abilities have been identified:

- **Mental model of program structure:** Novice programmers lack a mental model of the program structure, while expert programmers can acquire and apply mental models as needed.

- **Application of programming knowledge:** Novice programmers have fragile domain knowledge and struggle to apply it, while expert programmers have better hierarchical, syntactical, and semantical domain knowledge and can apply it.

- **Problem solving strategies:** Novice programmers can only use general problem solving strategies, neglect specific strategies and struggle to divide functionality into programming parts. Expert programmers have better tactical and strategical skills, can hierarchically develop subgoals and apply programming plans, and can recognize problems with known solutions.

- **Programming & comprehension approaches:** Novice programmers approach through syntax and control structures in a line-by-line, bottom-up approach and struggle to understand larger programs. Expert programmers approach through data structures and objects, in an algorithmic, top-down way, and can acquire a detailed, systematic view of a software system faster.

- **Programming efficiency:** Novices can highly benefit from support in the form of tools and techniques and have higher effort during software maintenance tasks. Experts are generally faster and more accurate in a variety of tasks.

Concluding the first step in answering `RQ1`, profound differences have been identified in the cognitive abilities of programmers of varying skill levels. A question that remains is how programmers can be assessed in a quantifiable manner along those dimensions with the goal to measure their programming skill acquisition. The approaches of this thesis are designed to shed light on the space between novice and expert programmers.

Second, Lister established a developmental epistemology based on neo-Piagetian stages of development that captures the development of novice programmers in a sequential way [Lis16]. The sequential stages describe how a programmer can reason about program code. Stemming from studies that provide evidence towards a (loose) hierarchy in the development of fundamental programming skills (first reading and manually executing program code, second explaining code by giving natural language summaries of what it does, and third systematically writing program code [VTL09]), the transitions between the sequential stages have been empirically investigated to uncover how novice programmers develop in their programming skills. The four stages described in the developmental epistemology by Lister [Lis16] are:

- The **sensorimotor** reasoning stage comprises an incoherent understanding of programming concepts in general and of the program execution in particular. Programmers in this reasoning stage are unable to manually execute code in a reliable way.

- The **preoperational** reasoning stage comprises the ability to manually execute code in a reliable way, which opens the possibility for programmers in this reasoning stage to inductively reason about program code. The reasoning is closely tied to specific input and output pairs of manual code executions.

- The **concrete operational** reasoning stage comprises the abilities to reason about program code deductively by reading it and to write purposeful program code. Instead of input and output pairs of manual code executions, programmers in this reasoning stage are capable of reasoning about abstract classes of relationships within the program code.

- The **formal operational** reasoning stage comprises the abilities of reflective reasoning, abstract reasoning in unfamiliar situations, and hypothetico-deductive reasoning. Programmers in this reasoning stage are capable of reasoning logically, consistently, and systematically.

This formed hierarchy of reasoning stages, effectively a hierarchy of cognitive, domain-specific abilities, makes it possible to distinguish programmers of varying skill levels using differences in qualitative characteristics. A notable absence is the formulation of developmental steps in the space between **concrete operational** and **formal operational** reasoning stages – representing experienced programmers capable of purposefully writing program code on their way to acquire programming skills towards being an expert. Utilizing the neo-Piagetian developmental stages and the developmental epistemology described by Lister [Lis16], programmers of different levels of reasoning are thereby denominated as follows in this thesis:

- **Novice programmers** are programmers that are firmly in the stages of *sensorimotor* or *preoperational* reasoning, or beginning programmers in the stage of *concrete operational reasoning* without the ability of abstract tracing.

- **Experienced programmers** are programmers firmly at the stage of *concrete operational reasoning*, capable of abstract tracing and purposeful code writing.

- **Expert programmers** are programmers firmly at the stage of *formal operational* reasoning.

Concluding the second step in answering `RQ1`, novice programmers' differences in cognitive abilities have been put into a sequence of developmental stages, with empirically verified transitions for learning programming. Novice programmers acquire programming skills in the following order: i) fundamental programming concepts, ii) reliable manual code execution, and iii) reasoning about code by code reading and purposeful code writing. From there, it is still a long road to mastering programming. In this thesis, I aim to carve out characteristics of programming skills along this road.

3. LEARNING ANALYTICS IN PROGRAMMING

As a by-product of the use of digital learning environments in education, inadvertently data about the educational process (the learning and teaching process) is generated. Starting in the mid-90s, the field of education data mining (EDM) established, mostly with statistics, visualizations as well as the mining of web-based data to make use of the data to support learning and teaching [RV07]. The review by Romero and Ventura [RV07] incorporates all early research work in this field of study. Not much later, Baker and Yacef [BY09] performed another review in order to evaluate the impact and growth of the developing field of EDM, collecting additional viewpoints on the approaches and topics of the field.

As the field of EDM was growing, the related research community of *Learning Analytics and Knowledge* (LAK) was formed with its first conference in 2011, starting a debate on the relation between the research areas of EDM and learning analytics (LA). Researchers of both communities engaged in a dialogue in 2012, pinpointing the combining as well as differentiating features of both research areas [SB12, BDS$^+$12]. While both communities reflect the emerging benefits of dealing with data-intensive approaches in education, there are differences in the methods as well as goals employed in those fields. EDM is seen as the more general approach, striving to employ data mining and analytics to uncover generalizable findings, finding out more about the process of learning, and building '*predictive models to explain and detect aspects of learning*' [BDS$^+$12, p. 2]. LA, on the other hand, is concerned with more specialized and advanced methods, focusing on the learner, exploring how an individual learner interacts with technology and finding out how the learning is thereby changed. Moreover, LA also focuses on visualizing the inadvertently generated traces of learners to facilitate educational improvements for learners and educators [BDS$^+$12].

The community of LAK defines LA as '*the measurement, collection, analysis and reporting of data about learners and their contexts, for purposes of understanding and optimizing learning and the environments in which it occurs*' [SB12, p. 252f]. In this thesis, I adopt the viewpoint of LA focusing on individual learners their context, with the goal of improving individual learning and teaching.

This chapter includes three sections that develop the foundation of the experiments and analysis methods introduced in the later chapters of the thesis:

1. In Section 3.1, categorizations and research frameworks for LA approaches in

programming are described. Most importantly, this includes a general categorization (Ihantola et al. [IBE$^+$15]) and a framework for IDE-based LA research (Hundhausen et al. [HOC17]).

2. In Section 3.2, the related work of LA in text-based and block-based programming is reviewed. Moreover, gaps in the commonly employed methods, and thereby potential points of future work, are identified.

3. In Section 3.3, the IDE-based LA approach employed in this thesis is described. This includes a description of the IDE instrumentations (`Scratch 2` and `Scratch 3` for block-based programming, `IntelliJ` plug-in for text-based programming) and the uniform data collection server.

A summary of the chapter is given in Section 3.4.

3.1 Research Frameworks for Learning Analytics

Since the literature reviews of educational datamining by by Romero and Ventura [RV07] and Baker and Yacef [BY09], the applications of data-intensive approaches to education have become increasingly more elaborated in their methods and have also been applied in more focused fields of study. Computing education and specifically programming education is one of the fields of study that, one might argue, quite naturally supports the automatic collection of students' learning processes and therefore provides a fruitful ground for learning analytics approaches to improve education.

In this section, I review existing categorization and research frameworks for learning analytics (LA), specifically LA in programming. A discussion of these frameworks is important to properly locate related work in the area of LA in programming and contextualize this thesis within this area. This section consists of two subsections, respectively containing the description of a general literature review and categorization framework for LA approaches in programing (Section 3.1.1), and the description of two specific frameworks for LA in programming, which focus on combining hypothesis and data-driven approaches on the one hand [GBB$^+$17] and IDE-based approaches on the other hand [HOC17] (Section 3.1.2).

3.1.1 General Categorization of Learning Analytics in Programming

A comprehensive literature review of the state of LA in programming has been conducted by Ihantola et al. [IBE$^+$15]. The authors analyzed 76 papers from ten years (2005–2015) along eight dimensions and corresponding categories.

The dimensions are summarized in Table 3.1 and are shortly described now. The *research goal* dimension differentiates publications regarding their main concern to use LA in programming: the students' abilities, knowledge, behaviour, and more;

Tab. 3.1: The eight dimensions to discriminate approaches to learning analytics in programming, and subcategories of each dimension [IBE+15].

Dimensions of LA in Programming

1. Research Goals

Student	Programming	Learning Environment

2. Approach

Case Study	Constructive research	Experiment
Study	Survey research	Other

3. Context

Formal course	No formal course	Other
Programming Language	Development Environment	

4. Subjects

Number of experimental subjects	Programming experience of experimental subjects	

5. Task

Number of tasks	Nature of tasks	Task assessment
Task feedback		

6. Data and Collection

Granularity of data	Data collection process	Supplemental data

7. Analysis Methods

Descriptive statistics	Inferential statistics	Exploratory statistics
Interpretative analysis	Automated classification	Other

8. Quality

Based on quality criteria for research [DB98, RH09]

Tab. 3.2: Common levels of granularity of learning analytics data collection in programing [IBE+15]. Note that higher-numbered levels of granularity can fully contain all lower-numbered levels of granularity.

Granularity of Data Collection

1 Key strokes	2 Line-level edits	3 File saves
4 Compilations	5 Executions	6 Submissions

the programming process including errors, patterns, strategies, and more; and lastly the use of the programming environment and (automated) feedback and grading.

The *approach* dimension is divided into six categories (case study, constructive research, experiment, study, survey research, and other). The authors remark that most of the reviewed publications (81%) only report work conducted in a single institution. Comparably, only a small number of publications presented longitudinal data or data from multiple courses.

The *context* dimension is split in educational context (type of course: formal, not formal, or other) and programming context (type of programming language and programming environment) reported in the reviewed publications. Nearly half of all publications worked with the `Java` programming language and its associated programming environments `BlueJ` and `Eclipse`.

The *subjects* dimension differentiates publications based on the number of experimental subjects reported on (with most publications reporting on 100−500 students) and the programming experience of experimental subjects (with most publications investigating first-year or CS1 students).

The *task* dimension includes differentiation on the number of tasks, the nature of the tasks (open-ended or not), whether the task was assessed based on some criteria, and whether some process of automated task feedback was afforded.

The *data and collection* dimension sports three categories to differentiate upon. First is the granularity of the data, visualized in Table 3.2. The authors identify the following, ascendingly inclusive, levels of granularity for which LA data in programming tasks can be collected: key strokes, line-level edits, file saves, compilations, executions, submissions. Second is the data collection process, which has multiple qualitative differentiations, ranging from key logging to specific IDE instrumentation for automatic data collection, and from revisions in version control systems to specific task submissions for student-triggered data. Third is the collection of supplemental data like questionnaires, manual assessments, observations, and others.

The *analysis methods* dimension differentiates between categories of (statistical) analysis methods on the collected data: descriptive, inferential, exploratory statistics, interpretative analysis, automated classification, and others. The number of publications using each of those analysis methods is quickly decreasing: while most publications use descriptive statistics (83%), inferential statistics is only used by 29% of the publications – and the other methods are used even less. This signifies that many publications only scratch the surface of possible analysis methods, which is not surprising for the field of LA in programming as it is not even 20 years deep.

The *quality* dimension was assessed based on quality criteria for research ([DB98, RH09]). The main shortcomings of the reviewed publications are: a lack of acknowledging and discussing confounding factors, a discussion of threats to validity and of utilized countermeasures, and a discussion of ethical issues. This again signifies the relative immatureness of LA in programming.

Besides the categorization along the described dimensions, Ihantola et al. discuss three topics to improve LA research in programming [IBE+15]. First is the necessity to verify experimental findings on multiple axes: re-analysis (the same data is analysed again), replication (the same method is employed to verify observed results), and reproduction (the same hypotheses are tested with varying methods and data). The authors introduce three criteria that can vary in reproduction experiments: the researchers, the data analysis, and the production. These three criteria make it possible to categorize verification approaches and identify their contribution.

The second topic is the necessary process of sharing experimental data (for the process of verification). The authors identify a number of relevant points that can hinder sharing experimental data, including varying context (programming language, teaching/course environment, assignment metadata, student levels) as well as data collection granularity (comparable units of program code).

The last topic is the lack of reports on student and teacher privacy as well as ethical concerns regarding data collection and publishing. These topics, emerging from the meta-analysis during literature review, showcase the shortcomings of the emerging field of LA in programming.

Two aspects of the general framework to categorize approaches of LA in programming [IBE+15] are directly relevant to this thesis. First is the granularity of data collection: different levels of granularity make different analysis types possible – therefore, it has to be carefully considered which levels of granularity to support. The instrumentations constructed to collect programming traces for this thesis have been designed with the reported levels of granularity in mind (Table 3.2). Second are considerations regarding privacy and ethics. Both are covered in Section 3.3.

3.1.2 Specific Frameworks for Learning Analytics in Programming

In this subsection, two specific frameworks for LA in programming are highlighted. The first is a process framework by Grover et al. [GBB+17] that combines benefits of hypothesis-driven and data-driven approaches, proposed for use in the context of block-based programming environments but applicable to any programming context (Section 3.1.2.1). The second is a process model and taxonomy for IDE-based LA by Hundhausen et al. [HOC17], which also provides categorizations of data and information obtainable in the context of programming IDEs (Section 3.1.2.2).

3.1.2.1 Process Framework for Learning Analytics in Programming

At face value, LA in programming is all about instrumenting programming environments to collect data of programming processes and quantitatively analyze the data to uncover regularities and irregularities in students' programming endeavours with the goal to improve education. But, as illustrated in a literature review [IBE^{+}15], only basic forms of statistical analysis (like descriptive and inferential statistics) are used to analyze the data. While the data mining aspect of LA is a backbone of the field, LA is also characterized by proper contextualization of analysis approaches, methods, and findings.

Grover et al. [GBB^{+}17] also start with the need to understand and contextually interpret categorized interaction sequences of students solving a given programming task with the `Alice` programming environment. The authors describe that '*interpreting these sequences is nontrivial, especially in the absence of any means of ground-truthing*' [GBB^{+}17, p. 8].

They proceed to introduce a mixed-methods process framework to combine hypothesis-driven and data-driven approaches for LA in programming, with the goal of providing a qualitative and grounded context to the analysis of programming processes. The authors apply the framework to block-based programming environments but there is no aspect which renders the use for text-based programming impossible. The process framework is visualized in Table 3.3 and described in detail in the following paragraphs.

In the first step of the process framework, *domain modeling*, the programming skills and programming practices that should be elicited by the LA approach are modeled. The outcome of this step is a set of focal concepts and practices that constitute the domain of the approach.

The second step represents the *task modeling*, including choosing and designing tasks and corresponding assessment rubrics based on the domain model established in the first step. In particular, the outcome of this step is a set of tasks that can elicit the identified focal concepts and practices of the domain model and corresponding assessment rubrics that make it possible to observe and/or measure the students' use and mastery of the concepts and practices.

The third step is *programming task piloting*, comprising a pilot study to use the designed set of tasks and assessment rubrics. The outcome of this step are data sets, which can include program code of various stages in the programming process (e.g., after each task, final program code, every compilation, etc.), additional process data from observations (e.g., screen observations, programmer observations through video, etc.), and log data from the programming environment.

Tab. 3.3: The process framework proposed by Grover et al. [GBB^{+}17] combines hypothesis-driven and data-driven approaches to LA in programming.

Framework of Hypothesis– and Data-Driven Learning Analytics

	Activity	***Outcome***
1	Domain Modeling: Programming Skills and Practices	Focal Concepts and Practices
2	Task Modeling	Programming Tasks and Assessment Rubrics
3	Programming Task Piloting	Multiple Data Sets
4	Analysis of Program Code, Process Observations, Log data	Initial Understanding, Task Refinement
5	Qualitative and Quantitative Analysis	Qualitative Rubrics and (Anti)-Patterns of Programming Practices/Processes
6	Combined Analysis of Code Sequences and Patterns	Rich Interpretation of Student Actions

The fourth step consists of an *initial analysis* of the data obtained from the pilot study, in order to build an initial understanding of variations in student approaches. Most importantly, this data analysis facilitates a critical view of the designed tasks and assessment rubrics and on the domain model as a whole and can highlight changes to both of them.

In the fifth step of *qualitative and quantitative analysis*, the log data is quantitatively analyzed and enriched by qualitatively analyzing the program code and process data observations. The outcome consists of qualitative rubrics on the one hand and detectable patterns and anti-patterns on the other hand. The authors note that this approach can reveal '*students' actions that are unseen in the solution programs*' [GBB^{+}17, p. 10].

The last step of the process framework is a *combined analysis* of code sequences and patterns, which makes it possible to understand student behaviour in a way not possible with either half of the approach. The relationship between programming actions and focal concepts and practices is explored so that those can be detected in data logs. The overall outcome of the process framework is a richer interpretation of student actions by postulating a domain of target skills and practices, iteratively designing fitting tasks and assessment rubrics, and analyzing and interpreting students' programming data through a qualitative lens (thereby contextualizing them).

Tab. 3.4: Iterative process model for IDE-based LA proposed by Hundhausen et al. [HOC17].

Process Model for IDE-based LA

`1.` Collect data	`2.` Analyze / process data
`3.` Design intervention	`4.` Deliver intervention

The process framework outlined by Grover et al. [GBB^{+}17], and applied by them on a block-based programming task in the `Alice` programming environment, enriches a sole data mining approach. As such, it is capable of covering weaknesses of both qualitative and quantitative approaches in educational data mining. Qualitative approaches lack the quantitative verifiability, while quantitative approaches lack the contextualized interpretation of findings. With the outlined process framework, the strengths of both types of approaches can be combined.

3.1.2.2 Process Model and Taxonomy for IDE-based Learning Analytics

Students learning to program inevitably encounter one or more integrated programming environments (IDEs), and they represent a central interface of programming education. Moreover, many approaches to LA in programming are tied to an instrumented IDE to facilitate automated data collection of programming actions. To incorporate the IDE as a more prominent aspect of LA research, Hundhausen et al. [HOC17] propose a process model of IDE-based LA research with the focus on designing and improving interventions (which, specific to programming education, take place in the context of IDEs).

The process model is visualized in Table 3.4 and described hereafter. The cyclical process model of powering IDE-based interventions with LA in programming education consists of four steps:

1. **Collect data:** Data is collected in an automated and unobtrusive way through instrumenting the IDE the students are naturally working in.

2. **Analyze / process data:** Collected data is processed into metrics and analyzed to understand students' programming processes, programming behaviour, and effects of previous interventions.

3. **Design intervention:** Findings from data analysis are used to design in-IDE interventions. An intervention is '*an event in which some combination of information, guidance, and feedback is shared with learners for the purpose of positively influencing the learner's behavior, attitudes, or physiological state*' [HOC17, p. 12].

Tab. 3.5: Top-level data categories and sub-categories of learning analytics data that can be automatically collected in augmented IDEs [HOC17]. Data collectible through IDE plugins are specified with ***(A)*** for augmented. Data categories from the lowest level are omitted for brevity and can be found in the source.

(Augmented) IDE Data Categories

Programming		
Editing	Compilation	Execution
Debugging	***(A)*** Assignment	
(A) Social		
(A) Activity Feed	***(A)*** Q&A Forum	***(A)*** Personal Messaging
(A) Testing	***(A) Survey and Quiz***	***(A) Physiological***

4. **Deliver intervention:** The designed intervention is delivered to learners through the IDE. Effected changes can be measured and analyzed, and the cyclical process model leads to the start.

For the first step in their process model, Hundhausen et al. [HOC17] establish data categories that can be collected by augmented IDEs to facilitate LA research, and devise a taxonomy of metrics that can be derived from collected data in IDE-based LA settings. The authors describe augmentation as extending IDE functionality '*through a plug-in architecture*' [HOC17, p. 6], for example, to incorporate the collection of unit testing data or social data (class activity and posts, internet platform forum questions and answers, and personal messaging tools). The first two levels of data categories are visualized in Table 3.5, and the first two levels of the taxonomy of metrics are visualized in Table 3.6. Both of these categorizations are important in the context of this thesis, as they describe the realm of data that can be captured and analyzed in IDE-based LA settings and therefore make it possible to situate the design considerations of the approaches and the contributions of this thesis. Because of this, the different categories are described in the following paragraphs.

First, the data categories of Table 3.5 are described. The first top-level category of *programming* encompasses the basic data categories that can be collected in instrumented IDEs, with sub-categories naming the specific events upon which data can be collected: program code editing (sub-categories: editing actions, the clipboard operations, file and program project actions, and file snapshots), compilations (sub-categories: file-snapshots, compile attempts and compiler errors), program execution (sub-categories: execution attempts, invocations, run-time errors, and the

Tab. 3.6: Taxonomy of learning analytics metrics that can be derived from information collected in augmented IDEs [HOC17]. Top-level categories, as well as sub-categories, are shown. Specific metrics are omitted for brevity and can be found in the source.

Taxonomy of Information Derivable from IDE Data	
Programming Behaviour	
Time Management	Programming Process
Program Content, Correctness, Efficiency	
Social Behaviour	
Participation Level	Participation Content and Quality
Knowledge	***Attitudes***
Eye Movement	***Physiological Response***

execution state), debugging (sub-categories: breaks and watches, stepping and running code, variable inspections), and assignments corresponding to programming tasks (sub-categories: viewing assignments, submitting solutions).

The other top-level categories can be supported by IDE plug-ins and are augmented data categories. The top-level category of *social* data includes data categories from: activity feeds and Q&A forums (sub-categories: audience, posts and replies, likes and other marks, badges and reputation), personal messaging (sub-categories: recipients, message content). The top-level category of *testing* includes the sub-categories of test executions and test results. The top-level category of *survey and quiz* includes the sub-categories of questions and responses. The top-level category of *physiological* data includes the sub-categories of eye tracking data, mouse and keyboard pressure, electro-dermal data collection, and heart rate.

Concluding, the authors propose the use of process data not directly tied to the program code to enrich IDE-based LA research. In their context, the proposal of using all kinds of data collectible from interactions with an IDE makes sense as they aim to improve IDE interventions. Also, this approach very much ties into the general understanding of LA as the research field of (educational) data mining focusing on individual learners and their processes and interactions.

For the second step in their process model, Hundhausen et al. state that a fundamental task in IDE-based learning analytics is to ’*transform the data into useful information [...], and therefore can serve as a suitable foundation for educationally-effective interventions*’ [HOC17, p. 10]. They devise a taxonomy of seven top-level

Tab. 3.7: Design dimensions of IDE-based interventions identified by Hundhausen et al. [HOC17]. IDE-based interventions along these dimensions can be investigated and improved by IDE-based LA approaches.

Taxonomy of Design Dimensions for IDE-based Interventions

Content

Data	Information	Critique
Suggestion	Encouragement	

Presentation

Visualizations	Notifications	Constraints

Timing

Persistent	Event-based	State-based
On request		

categories of metrics that can be derived from data collected through augmented IDEs (Table 3.5) and the analysis techniques used to derive them (counting, computing with mathematical formulas, computing with algorithms, visualizations, and machine learning approaches). The taxonomy is visualized in Table 3.6.

The first metrics category of *programming behaviour* encompasses metrics of time management during programming and metrics of the programming process. The next metrics category of *program content, correctness and efficiency* encompasses metrics relating to the program code (e.g., in relation to program size, in relation to specific programming keywords or constructs, and others).

Starting with the augmented data categories, the metrics category of *social behaviour* encompasses metrics of (social) participation level (e.g., number of posts, questions and answers made in forums, platforms, messaging tools), and of participation content and quality. The data category of *survey and quiz* hosts two metrics categories: *knowledge* (number of correct and incorrect answers, and to which extent the target knowledge is covered) and *attitudes* (self-efficacy, motivation, and other psychological measurements depending on the used survey).

From the *physiological* data category, two metrics categories can be derived. The metrics category of *eye movement* encompasses metrics of eye gaze data collection (e.g., heat map generations, screen fixations and saccades over time, gaze times in target locations, etc.). Finally, the metrics category of *physiological response* encompasses metrics of mood and personal state.

Altogether, the data categories and taxonomy proposed by Hundhausen et al. are a great tool to categorize IDE-based LA approaches and make them comparable by assessing the types of data and metrics used [HOC17]. However, the augmented data categories and derived metrics have to be carefully engineered and explained to experimental subjects with regards to privacy and ethical concerns.

For the third step in their process model, Hundhausen et al. [HOC17] introduce a taxonomy of design dimensions to be considered when designing educational interventions that take place in IDEs. The full taxonomy is given in Table 3.7. The authors identify three top-level design dimensions to gauge interventions: *content* (*'What information will the intervention contain?'*), *presentation* (*'How will the intervention be presented to the learner?'*), and *timing* (*'When will the intervention be delivered'*) [HOC17, p. 12–15].

The process model, at its heart, aims to help educators design *'interventions that effect positive changes in student learning processes, outcomes, and attitudes'* [HOC17, p. 15]. The authors propose to do so by tying the intervention design not only to LA research but also to learning theories. The authors identify possible implications on intervention design from the perspective of cognitive load theory, locus of control, situated learning theory, and social cognitive theory.

Concluding, Hundhausen et al. [HOC17] offer two contributions to shape research on LA in programming. On the one hand, the authors establish a pronounced focus on one of the natural contexts in programming education, the IDE, and a cyclical process model to design and deliver in-IDE intervention events to improve student learning. On the other hand, the authors also establish a set of taxonomies to categorize different aspects of LA approaches in the context of IDEs, namely the data to be collected, the metrics to be derived from the data, and the dimension to be considered during IDE-based intervention design.

For the approaches described in this thesis, the categories and taxonomies proposed by Hundhausen et al. [HOC17] have provided a guiding frame for LA settings in the context of IDEs. The authors depict the relation between data categories that can be collected from IDE interactions and resulting LA metrics that can be derived from the data categories to investigate programming in an IDE. This made it possible to focus on the data categories, and derived LA metrics, that are needed to investigate the research questions of this thesis: *programming behaviour* and *program content*. Also resulting from the influence of the Hundhausen et al. [HOC17], cognitive load theory is adopted for this thesis as a fundamental school of thought, especially the notions of limited working memory and the cognitive structures of schemata and their relation to expertise.

3.2 Related Work of Learning Analytics in Programming

In this section, the related work of learning analytics (LA) in programming is summarized. The section is organized in two subsections, collecting related work in text-based programming education (Section 3.2.1) and block-based programming education (Section 3.2.2). This organization has the following reason: i) they differ in the analysis approaches (text-based programming pursues syntactic analysis approaches, while block-based programming – due to the entities of programming behaviour being more interpretable semantically – pursues semantic analysis approaches), and ii) they differ in programming skills of experimental subjects (text-based programming most often investigates undergraduate programmers, while block-based programming most often investigates (upper) secondary school students).

In both subsections, two categorization frameworks are applied: first the categorization of granularity of collected data (Ihantola et al. [IBE$^+$15]), and second the categorization of LA metrics used for analysis (Hundhausen et al. [HOC17]).

3.2.1 Learning Analytics in Text-based Programming Education

This subsection provides a chronological overview, per group of authors, of related work of learning analytics (LA) approaches in text-based programming education.

One of the first applications of LA approaches on text-based programming education was published by Jadud [Jad06]. He investigated the interactions of novice student programmers with the *edit* and *compile* actions when editing `Java` code. Snapshots of the whole program were collected whenever the students compiled their code (*compilation* granularity [IBE$^+$15]). First, Jadud reports that the time between compilations is dependent on the program error state and its transition: students are more likely to take less than 30 seconds to edit and compile their program in a faulty state, but take more than two minutes of time to edit and compile programs in a correct state. Next, Jadud reports on visualization possibilities to evaluate different programming behaviours: the error state of the program snapshots and relative measures (code churn, time of edit-compile cycle, location of changes in code file). Last, Jadud introduces the measure of *error quotient*, an algorithmic metric to capture how much students struggle with compilation errors. The measure penalizes sequences of program snapshots that are successively faulty, and even more so when they have the same compilation error. The metric can be categorized as a LA metric of *Programming Behaviour/Programming Process* [HOC17].

Norris et al. [NBF$^+$08] introduce the toolset `ClockIT` to log and monitor novice programmers using the `BlueJ Java` development environment. The toolset monitors 10 types of events regarding project and package management, compilation and invocation in `BlueJ` as well as file changes. The data is collected on at least *compilation* granularity [IBE$^+$15] – file changes are also monitored, but the specific procedure is not detailed. The authors showcase the usage of their tool to monitor students and report on different descriptive statistics of the students' programming process (percentage of specific events like failed compilations). The related LA metrics can be categorized as *Programming Behaviour* and *Programming Content* [HOC17].

LA approaches are often applied to open-ended tasks in order to elicit student's unconstrained programming behaviour. Blikstein [Bli11] reports on a data collection in the `NetLogo` programming environment, collecting students' programming snapshots on *key stroke* granularity [IBE$^+$15] before condensing the log into LA metrics (evolution of code size, time between compilations, error state). The used LA metrics can be categorized as *Programming Behaviour* and *Programming Content* [HOC17]. Blikstein reports on a small-scale data analysis to showcase how semantic events during the programming process can be detected, including: *stripping down* existing programs, no coding activity while students *browse other code*, *jumps in program size* when students paste code, and *final submission phase* when students fix formatting, indentation, variable names and more. The author states that difficulties during the programming process can be detected by various measures: successive compilation errors with a low amount of program changes, atypical error rate curves, high code size changes for a long period of time.

Later Blikstein et al. [BWP$^+$14] perform data mining to analyze students' programming trajectories on several and single programming assignments in the `Karel the Robot` programming language (`Java`-like). The data collection consists of program code snapshots whenever characters or lines of code change (at least *line-level edit* granularity [IBE$^+$15]), and the supported LA metrics include *Programming Behaviour* and *Programming Content* [HOC17]. Notable, the authors use a measure of *program distance* to evaluate how far away a given student program snapshot (measured with, among others, needed changes to the abstract syntax tree) is from other students' program snapshots. This made it possible to apply a Hidden Markov Model on students' sequences of program snapshots, with the result of defining clusters of common program snapshots for the students' pathways to an assignment's solution. The authors define three clusters, distinguished by the trajectories between the common program snapshots. Investigating the grades of the students for each cluster, the authors report on the clusters' predictive quality regarding the students' midterm grades.

Fernandez-Medina et al. [FMPPAGPR13] showcase a model to collect students' `Java` program sequences, categorize compiler errors and warnings based on their conceptual family (syntax, semantic, structural) and compiler error family (field, syntax, type, import, method, constructor), and visualize the results by reporting percentages of specific errors during the collected six programming sessions. The employed `Eclipse` plug-in collects data on *compilation* granularity [IBE$^+$15], and the resulting LA metrics can be categorized as *Programming Behaviour* and *Program Content* [HOC17]. They show that syntax errors decrease along programming sessions, internal errors always account for at least 20% of all errors, and other errors can be attributed to the topic of the specific programming sessions. The authors also suggest incorporating this feedback in IDEs for teachers as well as students in order to improve the teaching and learning process, with the caveat that '*the level* *of analysis relies on the expertise of teachers to determine the theoretical programming concepts associated to the errors made by students*' [FMPPAGPR13, p. 241], thereby only providing the teachers the opportunity to more easily identify errors.

Watson et al. [WLG13] introduce an alternative metric to predict students' programming performance based on compilation errors, called the Watwin score. The authors collect data on *compilation* granularity [IBE$^+$15] through a `BlueJ` plugin, and compute a score for each student based on their successive compilation errors and the relative time (measured against all other considered students) to fix each respective type of error. The score is normalized between 0 and 1, with 0 representing a programming sequence without any compilation errors. The resulting LA metric can be categorized as *Programming Behaviour/Programming Process* [HOC17]. The authors establish that the Watwin score can explain more than 40% of variance in a linear regression and overall course mark as the dependent variable, and achieves a classification accuracy of 75%. The authors also evaluate their metric against the predictive power of the error quotient by Jadud [Jad06] over the duration of the course, showing that the Watwin score already achieves an explanation of 30% halfway through the course, outperforming the error quotient. The authors also note methodological shortcomings of the error quotient with the way sequences of compilation pairs are constructed: the switching of files is not taken into account, which showed to be a proven way of students' programming process in their data.

Ihantola et al. [ISV14] use a LA approach to investigate indicators of the difficulty of programming assignments with basic syntactic features. The authors collected student program snapshots on *key stroke* granularity [IBE$^+$15] and aggregated the data to utilize LA metrics of *Programming Behaviour*, *Programming Content*, and *Attitudes* (perceived task difficulty) [HOC17]. The authors differentiate students with and without prior programming experience and conclude with the following findings: the perceived difficulty does not correlate with the number of states that fail to compile or the length a student's program is in a non-compiling state; the time spent and the amount of programming events (number of key strokes) does correlate with the perceived difficulty – more so compared to syntactic features of the programs like lines of code and the number of control-flow elements. While the correlations are not very high the authors note that the benefit of their utilized metrics is that they can be easily obtained from every stream of programming events in an automated fashion.

Carter et al. [CHA15] identify shortcomings of the error quotient proposed by Jadud [Jad06] and the Watwin score proposed by Watson et al. [WLG13]: they only focus on students' compilation activities and predict performance based on the ability to quickly and accurately fix syntax errors. To enable a more holistic view of students' programming processes, Carter et al. [CHA15] introduce two notions that can be approximated from the stream of `C++` programming log data. First is the program correctness, which can be assessed on the dimension of syntactic correctness (measured with syntax errors during compilation) and semantic correctness (measured with runtime exceptions in the last execution), forming a two by two table of correctness states. Second is a programming state model that includes execution, debugging, and editing states of different kinds. With these two notions of programming states, the transitions between the states can be studied. The authors analyze recorded programming sessions with the normalized programming state model (NPSM), in which the time of each student in every programming state is normalized against the total programming time. The NPSM metrics outperform the error quotient and the Watwin score, using a linear regression model to predict the students' average and final assignment grades, producing a significant explanation of variance ($R^2 = 0.39/0.36$ respectively). With their study, the authors show that the metrics of error quotient and Watwin score are not easily generalizable to different programming languages. The granularity of data collection is not given in detail but can be assumed to be at least *line-level edits* [IBE$^+$15], in addition to logging programming-related activities in the IDE. The LA metrics fall into the category of *Programming Behaviour* [HOC17].

In a follow-up work, Carter et al. [CHA17] incorporate measures of social behaviour in addition to NPSM in order to create a predictive model of students' programming achievement. In addition to the data collection outlined above, the authors also collected posts and replies in a learning management system, thereby incorporating LA metrics of the category *Social Behaviour* [HOC17]. They model the level of participation in 2–week intervals on a scale of four levels, starting with no posts and replies at level one, with the maximum of two or more posts and replies at level four. The authors show that incorporating the social behaviour metrics improves the explained variance of NPSM (up to 49% of the variance in final grades can be explained in the combined model). Moreover, 28% of the variance can be explained by the combined model with data of a single assignment, which exemplifies the potential use of the predictive model as an instrument of early warning.

Rivers et al. [RHK16] employ learning curve analysis to investigate students' learning curves of specific programming concepts (measured as token types in the abstract syntax tree) in `Python`. The authors measure correct use of specific concepts by comparing students' programs with the correct program version, computed by an automated hinting tool and the ITAP algorithm. The data collection was done on *submission* granularity [IBE$^+$15]. The LA metrics used in the analysis fall into the category of *Program Content* [HOC17]. The authors collect data of 89 students, collecting a total of 3287 program states. In the resulting learning curves for each programming concept, the trend of the error rate can be investigated. The authors identify different kinds of learning curves: good learning curves, where the error rates decrease to zero, still-learning curves where the error rate is only slowly decreasing and remains higher – students' mastery is yet to be shown, already-learned curves that start with a low error rate and do not increase, and no-learning curves when the error rate does not decrease. The authors exemplify that a closer look at specific concepts and the students' learning of those concepts is possible. Future work includes the grouping of programming concepts (like comparison operators) to describe programming concepts and their learning more accurately.

The `Blackbox` data collection project of the `BlueJ` development environment ([BASK18]) supports a heap of novice programmers' compilation data. Altadmri et al. [AB15] perform an extensive analysis of programming mistakes, categorizing 18 common syntax and semantic errors (ranging from misunderstandings in syntax use to type errors and semantic errors when calling methods). The data was analyzed on *compilation* granularity [IBE+15], and the LA metrics can be categorized as *Programming Content* [HOC17]. The results include an analysis of the frequency of all errors (with unbalanced brackets being the most frequent syntax error), the median time to fix those errors, the average number of times a specific error is repeated, and the evolution of error types during the academic year (showing increases in certain error types at different stages). The data is only investigated in a descriptive way the programming process of specific students is hardly taken into account.

Fu et al. [FSO+17] introduce a LA system for programming education that facilitates the learning and teaching of `C` programming, providing the means to capture students' programming behaviour and identify their difficulties. Students can use the system to locate their syntax errors and browse educational material that can help them fix their errors. The system collects data on *compilation* granularity [IBE+15], and the authors categorize the compiler errors into 36 types. The error types are used to automatically guide students to educational material that was used by their peers to resolve the same error types. Therefore, the used LA metrics are of the categories *Programming Behaviour* [HOC17]. The authors categorize the students based on the finished programs, the errors in the final programs, and the time used by the students. Based on this categorization, the authors propose a number of visualizations to use the compiler error data: an absolute error distribution per student, a protocol of student activities (in 5–minute buckets), a detailed visualization of the progress of single students containing sequential compilation events for each program they work on as well as the presence and absence of compiler errors. A promising result of this research is the automatic feedback in the form of educational resources likely related to a student's source of errors.

Sharma et al. [SMT+18] use syntactic measures of students' `Java` coding to identify different programming profiles, in order to identify early low performers. They collect data through an Eclipse plug-in on *compilation/execution* granularity [IBE+15], and use metrics of unit test runs (number of tests run, improvement in fraction of correct tests, first test score, time between run and edit actions) and of program content (size differences between program versions, compilation errors and warnings) for their analysis. The resulting LA metrics can be categorized as *Programming Behaviour* and *Program Content* [HOC17]. The authors identify three profiles of students: intellects (skilled and confident, less frequent test runs), thinkers (want to receive early feedback, run tests more frequently), and probers (experience difficulty, most frequent test runs). With these results, the authors highlight the possibility of providing feedback in a timely and appropriate manner, depending on a student's profile.

Djelil et al. [DMS19] report on learner's behaviour in a custom programming microworld designed to teach object-oriented programming (in `C++`) with the help of 3D mechanical structures, called `PrOgO`. Programming data is collected at least on the granularity of *line-level edits* [IBE+15], additionally also interactions with the interface to modify the 3D structures are collected. The LA metrics can be categorized as *Programming Behaviour* [HOC17]. By employing hierarchical clustering, the authors identify four clusters of students based on their interactions with the programming environment. The clusters differentiate students who have little interaction with the studied interface (i), students who focus on the 3D structures (ii), students who focus on both the 3D structures and the code editor (iii), and students who focus on the code editor solely (iv).

Lu and Hsiao [LH19] propose a method to identify example code based on the programming language parse labels present in a method. The authors employ deep learning models (among them, neural networks) on extracted textbook examples, consisting of `Java` program code and corresponding inline and method comments, with the goal of identifying method concepts based on a textual description. In the results, neural networks outperform the other models for most concepts and types of documentation. The authors state that the results are a next step to facilitate automatic assistants in programming learning '*such as example code recommendation and auto coder*' [LH19, p. 159].

To summarize, LA approaches in text-based programming education have devised different ways to utilize the LA metrics categories of *Programming Behaviour* and *Program Content* with the goal of modeling student performance based on **syntactic** programming data automatically obtainable from programming sequences. Approaches frequently employ compiler error messages, a modeled state of student's solutions, and used programming concepts in the model of student performance. The goal of modeling the student performance is either i) to validate a predictive model based on average grades, final grades, or specific assignment grades, with the future goal of monitoring the performance of struggling students and alerting educators and learners, ii) to identify profiles of programming behaviour, which are again related to predicting student performance, or iii) to identify difficulties of specific students in the form of concepts they struggle with, and means of feedback for educators and students to assist in handling these struggles (e.g., visualizations and appropriate educational material). Publications that only report descriptive statistics (e.g., rates of specific errors) can still provide a picture of student performance.

Other than the two mentioned LA metrics categories, other augmented IDE data is hardly considered in the reviewed related work. A factor in this matter could be the availability of technical solutions corresponding to the teaching environment in which the data collection is situated – augmented LA metrics categories need specific plug-ins and additional technologies (e.g., learning management systems), which might be incompatible. In this regard, the collection of *Programming Behaviour* and/or *Program Content* could be seen as a baseline to qualify research as LA in programming. Still, as proposed by Hundhausen et al. [HOC17], the use of augmented LA metrics can be an important factor in the design and evaluation of IDE interventions.

The review of related work also highlights a gap in research: the analysis of text-based programming sequences based on **semantics** with interpretable programming actions. There are some publications that work in this direction (e.g., interpreting students' programming actions [Bli11], identifying behavioural patterns [SMT^{+}18]). However, sets of interpretable programming actions based not only on syntactical features of program content but on semantic features of program content (e.g., relations between variables and their semantic in the program code) have not been investigated so far. This is the point where the LA analysis approach of this thesis is situated.

3.2.2 Learning Analytics in Block-based Programming Education

This subsection again provides a chronological overview, per group of authors, of related work of learning analytics (LA) approaches in block-based programming education. Of note is that approaches in block-based programming education differ from text-based programming education in a number of ways. First, the experimental subjects investigated in block-based programming education are most often (upper) secondary school students compared to the undergraduate students investigated in text-based programming education. Next, in the previous subsection, it was outlined that text-based programming education often focuses on syntactic features of programming sequences, and specifically on (compiler) errors – this is not possible in block-based programming education by design, as the programming environment prevents syntax errors. Moreover, the possible programming actions in block-based programming education are more directly interpretable as they, again by design, form semantic units (e.g., most blocks in `Scratch` have a semantic in the context of the programmed sprite-object). This makes semantic analysis approaches feasible. These differences showcase that the context of LA research in block-based programming education is quite different – this is important to objectively assess and compare them to the approaches summarized above.

From a LA point of view, Meerbaum-Salant et al. [MSABA11] provide an initial investigation of learners' characteristics in the form of *programming habits* in block-based programming environments which should not be adopted as they could be '*detrimental to more advanced study*' [MSABA11, p. 168] (identifying two habits, *bottom-up programming* and *extremely fine-grained programming*). They qualitatively analyze exam solutions and submitted Scratch projects – as such the identification of *programming habits* for arbitrary projects is not possible.

With the analysis tool Hairball, introduced by Boe et al. [BHL+13], static analysis of Scratch 2 programs is possible. Hairball is designed in a modular fashion and can be extended with plug-ins to capture different facets of block-based programs. Each plug-in labels the investigated facet of the program as correct, semantically incorrect, incorrect, or incomplete. The authors evaluate their analysis tool with four plug-ins: the initialization plug-in to evaluate correct initialization of modified sprite attributes, the say/sound synchronization plug-in to evaluate the synchronized use of message and sound blocks in junction, the broadcast and receive plug-in to evaluate the existence of broadcast blocks and corresponding receive blocks, and the complex animation plug-in to evaluate more complex animation including rotations or costume changes inside a loop. The authors show the benefits of using the static analysis tool Hairball: with a very low false positive rate of 0.4% the *correct* labels of Hairball can be trusted and it can therefore be a valuable tool in assessing Scratch programs. With these results, the authors showcase the use of an analysis tool to automatically assess correct programs, but their analysis tool lacks in identifying incorrect programs.

Another step closer to evaluating large amounts of data, and in line with the LA goal of investigating learners' characteristics foremost, Berland et al. [BMB+13] report on an experiment with female high school students interacting with the visual programming environment IPRO to program virtual soccer AI bots. The authors aim to investigate the process of *tinkering* (defined as explorative, playful activities involving just-in-time planning while producing feedback from the environment) during learning to program. During programming activities, the program states of the students were saved on each program change. The states were clustered in similar program states and paths between them, resulting in three main interaction phases and transitions between those: exploration (clustered program states are named re/start, minimal, active), tinkering (clustered program states are named active, balanced, testbed, compact), and refinement (clustered program states are named balanced, logical). The authors investigate the clustered program states based on the program contents (number of logic elements, number of action elements, number of unique elements, sensoric roboter action coverage, program length). The authors end with proposing the EXTIRE framework, with the notion of aggregating students' programming edits and resulting state transitions in order to pinpoint their phase transitions (between exploration, tinkering, and refinement). Using this framework, the authors identify a significant increase in program quality with each phase transition. Discussing their results, the authors note that each student went through all phases, but on their own terms, referring to Piaget's '*epistemological pluralism*' [BMB$^+$13, p. 587].

Sherman and Martin [SM15] illustrate a research design to investigate students' programming behaviour in the block-based programming environment `App Inventor`. They highlight the difference between ex-post-facto (capturing programming interactions and analyzing them after programming is finished) and real-time assessment instruments (providing assessment feedback during programming) and design their research to incorporate ex-post-facto instruments. Their descriptions align with parts of the IDE-based LA process model introduced by Hundhausen et al. [HOC17]. The research design has led to the development of a toolkit to facilitate research and analysis of program code developed with `App Inventor`: the persistent collection of project snapshots after every student change, audio and video capture (which has to be ethically justified), a real-time event server to facilitate instrumentation applications, and project analysis tools to construct declarative queries over the blocks present in a programming project [PST18]. This work again shows the comparatively simple means needed to identify specific actions in a stream of programming events, represented by the semantic nature of programming blocks.

For block-based programming environments, there is no open data collection initiative like the `Blackbox` data collection which stores novice programmers learning to program `Java` in `BlueJ` [BASK18]. To move the LA community in block-based programming forward, Aivaloglou [AHMLR17] published a prepared data set of about 250k scraped `Scratch` projects, including the following information: project information, remix information to link different projects together, programming information (scripts, blocks, procedures, block types), and '*programming mastery scores*' measured with *Dr. Scratch* [MLRGHR17]. The data set enables static analysis of finished projects but leaves some aspects to be desired: i) only the end version of a project is stored – the analysis of program construction is not possible, ii) no information is available regarding the learning and teaching context, or the programming experience of the students. Both lacking aspects form the backbone of holistic LA approaches to best support the learning on an individual basis and need to be considered as important future work in the context of data sets of (block-based) programming.

Amanullah and Bell [AB19] investigate the use of elementary patterns, similar in their ideas to programming strategies, in teaching novice programmers in `Scratch` as a means to prevent bad habits and problematic programming patterns from forming. The authors select a subset of elementary patterns for programs with loops and with selections, and analyze a sample of about 212k projects regarding the use of those elementary patterns. The authors show that, in the sampled project data set, most of the elementary patterns are under-used. The authors argue that a more extensive educational use of elementary patterns should form the basis to develop problem solving and logical thinking skills in computing. In an IDE-based LA setting, interventions could be designed to make use of elementary patterns in order to help students who struggle with a specific, well-defined task.

Filvà et al. [FFGP^{+}19] perform clickstream analysis to assess students' behaviour when interacting with the Scratch 2 programming interface while programming predefined tasks. They instrumented the Scratch 2 programming environment to collect context and click information and store them in a '*learning record store*'. The authors used the clickstream data to extract two types of behavioural patterns from an experimental cohort based on clustered click concentration. The first type of patterns, based on click focus on the execution buttons, are categorized as such: blocked development (students behind on their coding), normal development (students with a balanced focus between graphics interface and coding interface), rapid trial-error development (students make rapid changes and check the execution results). The second type of patterns includes coding trends, condensed in the following taxonomy: students more prone to development (click focus on the scripts space and the block palette), students more prone to organize (click focus on moving rather than adding/modifying/deleting objects, blocks or sprites), students more prone to design (click focus on sprites and scenarios), and students more prone to create multimedia (click focus on sounds and drawing tools). The authors incorporate the feedback of the analysis results in the virtual learning environment Moodle, making it accessible to educators for improved assessment. The authors show the analysis possibilities without examining the program code.

Frädrich et al. [FOK^{+}20] use a static analysis framework on Scratch 3 programs in order to automatically identify bug patterns as they occur in student-written program code. They investigate statically identifiable bug patterns of three categories: i) syntax errors (as they would be detected by compilers in a text-based programming language), ii) general bugs (which can occur regardless of the programming environment), and iii) bugs specific to Scratch (and, by extension, applicable to comparable block-based programming environments). They perform their analysis on a fresh data set of new Scratch 3 programs, capturing about 135k programs. With their automatic means of identification, the authors show that their proposed bug patterns occur frequently and in similar structures and often yield failures upon execution. The authors demonstrate a valuable contribution regarding the analysis means of block-based program code, opening up possibilities for future LA approaches with a detailed analysis of program content. This analysis of block-based program content can even be enriched by the semantics of the underlying block-based programming actions, something not easily possible in text-based programming languages.

Körber et al. [KGSF21] improve on the static analysis framework described above and perform *anomaly detection* on Scratch 3 programs to identify anomalies in students' programs. The authors follow the assumption that common behaviour in students' solutions is more likely correct behaviour and that rare deviations, *anomalies*, are likely wrong. Within this assumption, the authors implement an automated approach that can identify and flag anomalies. With a set of student-created solutions to six example problems, they authors report that there is a clear relationship between the number of anomalies found in a solution and its correctness. Furthermore, most of their classified anomalies (50 out of 60) hint at defective or smelly code, while the others hint at distinguished student work. Altogether, the automated approach supports educators in evaluating their students' solutions and providing feedback.

To summarize, LA approaches in block-based programming most often rely on the static analysis of program content or on the analysis of the interaction behaviour with the programming environment (incorporating the IDE-based LA metrics *Programming Behaviour* and *Program Content* [HOC17]). This is not surprising due to two aspects. First, the graphical interface novice programmers interact with and its implications for program construction is one of the defining features of block-based programming environments compared to text-based programming. Therefore evaluations regarding students' programming behaviour in relation to the programming environment is of particular interest when aiming to improve programming education. Second, the program content is pre-categorized in semantic entities, based on the block structure of the programs, which makes its analysis potentially more meaningful and interpretable.

LA approaches in block-based programming still have the potential for improvement to facilitate comparable analysis approaches and results. While the investigation of programming and interaction behaviour is important, it is also dependent on the specific (block-based) programming interface. A potential avenue of future work is to introduce a research framework with the aim to standardize potential interaction parameters, making it possible to evaluate different programming interfaces regarding novice programmers' learning. Regarding the analysis of program content, the nature of the block structure makes semantic analysis possible more easily, but again hinders comparability between different block-based programming environments. In this regard, Talbot et al. [TGSH20] offer a possible solution: the representation of block-based program code in a common (graph) format to make static analysis possible. Lastly, an open data collection project to share block-based programming snapshots of program construction could facilitate the comparison of findings across different block-based programming environments.

Moreover, LA approaches in block-based programming can benefit from the other IDE-based LA metric categories established by Hundhausen et al. [HOC17], following the same argument as above: the interface that makes programming interactions possible is one of the defining features of block-based programming, and naturally gives affordance to IDE interventions and design improvements. As an example without the LA context, Papavlasopoulou et al. [PSGJ17, PSG18] incorporate eye-tracking metrics in their investigation of students' attitudes towards block-based programming. Such measures can yield additional insights into students' programming behaviour, when coupled with the other IDE-based LA metric categories.

In this thesis, programming environment instrumentations, data collection sharing, and analysis methods for text-based and block-based programming are introduced. The goal is to use the best of both halves, combining the semantic nature of block-based programming and the strength of syntactic analysis of text-based programming. I set out to conduct semantic and interpretative analysis, using the IDE-based LA metrics of *Program Content* and *Programming Behaviour*, on recorded program construction sequences. This entails the use of mixed methods by first analyzing program construction sequences in a qualitative way to uncover and interpret patterns with regard to their meaning in the program construction and then quantitatively analyzing the use of the patterns based on static program analysis.

3.3 Instrumentation for IDE-based Learning Analytics

In this section, the IDE instrumentations used to collect program construction data in learning analytics (LA) settings, which form the basis of the scientific investigations of analysis methods for this thesis, are technically described. Three IDE instrumentations have been implemented to facilitate data collection and analysis over the course of this thesis: a locally running click logger to record `Scratch 2` programming, an instrumented online programming environment to record `Scratch 3` programming, and an instrumented, locally installed plug-in to record `IntelliJ Java` programming. Moreover, a uniform data collection server has been implemented to serve as a single data repository for all IDE instrumentations.

The categorization of logical services made by Ihantola et al. [IBE$^+$15] is used to situate these implementations in the context of LA research. Ihantola et al. [IBE$^+$15] identify the typical logical services on which learning analytics instrumentation and programming data collection are performed. They are depicted in Figure 3.1. They identify five main logical services, two of which are dominantly used by students: *IDEs* on the one hand and *assessment systems* on the other hand. Those two logical services are connected to the third logical service, *data storage*, which in turn makes the logical services of *export* and *visualization* for educators possible.

The instrumentations described in this section are all situated in the *IDE* service (Figure 3.1 `(a)`), accessible to students; the uniform data collection server represents a *data storage* service (Figure 3.1 `(c)`). In Chapter 6, I extend the idea of the uniform data collection server to incorporate elements of the logical services of *export* (Figure 3.1 `(d)`) and *visualization* (Figure 3.1 `(e)`).

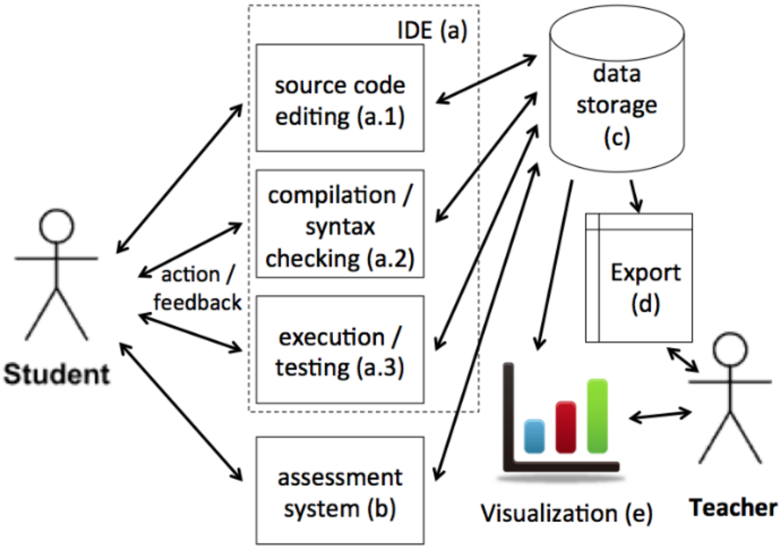


Fig. 3.1: Logical services on which learning analytics instrumentation and programming data collection are typically performed (Ihantola et al. [IBE$^+$15]).

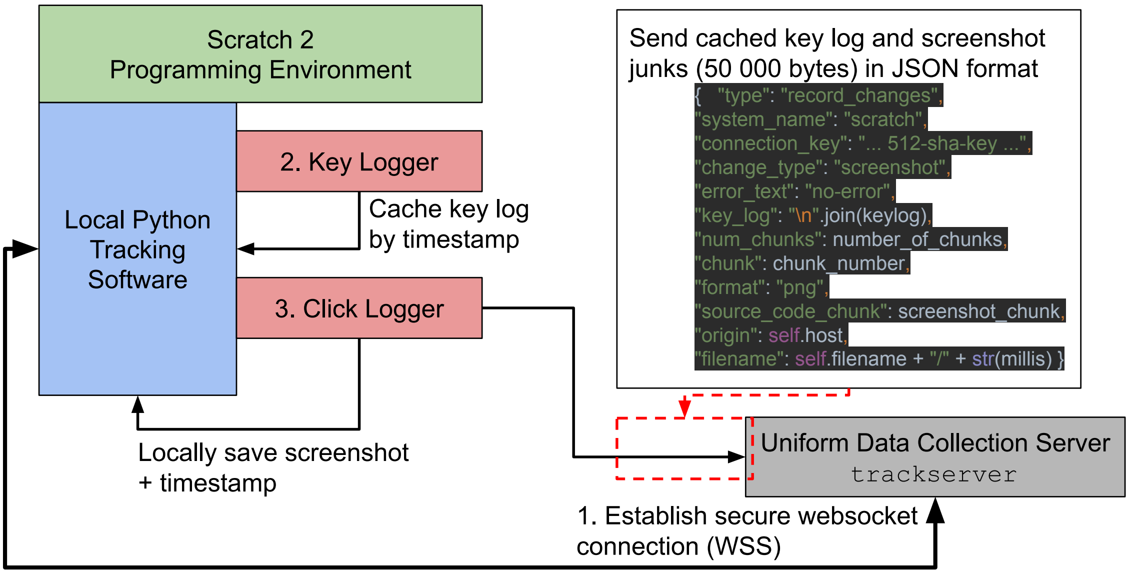
3.3.1 Block-based IDE Recording: Scratch 2

In the present day, recording the program construction of the outdated block-based programming environment `Scratch 2` seems out of place. However, the following recording implementation originates from a preliminary study in the context of this thesis, evaluating the potential of learning analytics (LA) data collection and analysis in block-based programming environments. Subsequently, this preliminary study led to the instrumentation of a custom online programming environment for `Scratch 3`, described in Section 3.3.2.

During the preliminary study, `Scratch 3` was still in development – therefore, the preliminary study was carried out on the programming environment `Scratch 2` (which, in contrast to `Scratch 3`, is based on `Flash`[1]). The goal was not to fully instrument the IDE, as a transition to `Scratch 3` was planned for future studies. Rather, the goal was to get more experience designing IDE instrumentations, designing tasks and problems suitable for LA, and analyzing program construction data.

These goals influenced the design of this IDE recording. Rather than modifying the source code of the programming environment (instrumentation), the recording is relocated to the basic actions students do to interact with the programming environment: mouse and keyboard interactions. For this recording procedure, a locally running background measurement script, written in `Python`, is used to record mouse

[1] `https://en.scratch-wiki.info/wiki/Scratch_2.0`



```
{   "type": "record_changes",
"system_name": "scratch",
"connection_key": "... 512-sha-key ...",
"change_type": "screenshot",
"error_text": "no-error",
"key_log": "\n".join(keylog),
"num_chunks": number_of_chunks,
"chunk": chunk_number,
"format": "png",
"source_code_chunk": screenshot_chunk,
"origin": self.host,
"filename": self.filename + "/" + str(millis) }
```

Fig. 3.2: Structure of `Scratch 2` IDE recording procedure.

and keyboard events and corresponding timestamps during programming, and to record screenshots on each mouse click. The rationale is that students need to click, drag and drop blocks to construct their program and need to interact with the keyboard to fine-tune their program. By collecting all available mouse and keyboard events as well as the program state for each programming action, as a screenshot, the program construction can be reproduced and investigated.

The technical structure of the `Scratch 2` IDE recording procedure is given in Figure 3.2. In the following, the elements of the recording procedure are described.

The `Scratch 2` programming environment (online or offline) is not changed and is only indirectly recorded by screenshots. The `Python` tracking software has to be locally executed and does not interfere with the programming environment.

`1. WSS connection:` On startup, the `Python` tracking software establishes a secure websocket connection to the uniform data collection server `trackserver`. When successful, a bidirectional communication channel. On this channel, program construction data is sent to the server in `JSON` format. In any case, the local tracking software also prepares a local folder for offline recording.

`2. Key Logger:` The `Python` tracking software registers a key logger for all keyboard interactions on the machine while it is executed. Note that participants need to be informed on this key logging beforehand, even when using it on workshop computers. The key log (pressed keys) and corresponding timestamps are written locally and are cached to be sent with the next click interaction.

`3. Click Logger:` The `Python` tracking software registers a click logger for all click interactions on the machine while it is executed. Again, note that participants need to be informed on this click logging beforehand. Click coordinates, mouse button states, and mouse wheel interactions are locally logged by timestamp for mouse events. Additionally, on each mouse click, a screenshot is made and saved locally.

Each mouse click triggers the transmission of program construction data to the data collection server, consisting of: i) the cached key log since the last mouse click, extended by the triggering mouse event, ii) chunks with a size of 50 000 bytes of screenshot data (which are assembled to a single file on server side) and the corresponding file format (`png` for this tracking software), iii) identification information of the tracking software (type of recording message, pre-registered system name of tracking software, pre-established SHA-512 connection key for safety), and iv) coarse user information (host origin). Note that the supported change type for this tracking software is only `screenshot` and that no errors are sent (this is reserved for compilation errors of other IDE instrumentations).

There are a number of positive features of using this recording procedure: i) it can be used for offline programming, ii) recording works in an unobtrusive way, iii) recording does not interfere with the programming environment (and, theoretically, also works for the `Scratch 3` programming environment).

However, there are also drawbacks of this recording procedure: i) consent and great trust are required from the participants, as their every interaction with the two main input devices is logged, ii) the approach of screenshots is dependent on the screen resolution. Therefore, the best analysis results are achieved by using workshop or lab computers with matching screen resolutions, iii) the program construction actions have to be re-interpreted, as the programming actions need to be triangulated from the click and key logs and the screenshots of pre and post states. While this works for most basic programming actions, a full instrumentation can circumvent this drawback by directly logging the interface interactions which lead to programming actions, iv) for detailed program analysis, a model of the constructed program in each step has to be constructed from the triangulated data (key and click logs and screenshot). This could enable an insightful analysis of the recorded data but has not been done so far. However, the `Scratch 3` IDE instrumentation also circumvents this drawback by directly logging the full program state on each programming interaction.

Following the classification of Ihantola et al. [IBE^{+}15], this measurement procedure records programming data by *key logging*. Regarding the granularity, this measurement procedure also records data on *key stroke* and *compilation* granularity. Following the categorization of LA metrics of Hundhausen et al. [HOC17], no data category is directly recorded by the `Scratch 2` recording procedure. However, from the recorded interactions with the programming environment, the following LA metric categories can be computed: *Programming Behaviour* can be re-constructed from the recorded programming actions, and *Program Content* could be re-constructed by examining the programming actions and the blocks present in the program script.

The source code of the `Scratch 2` IDE recording procedure is publicly available: `https://gitlab-iid.aau.at/seqtrex/blockchange`.

3.3.2 Block-based IDE Instrumentation: Scratch 3

In contrast to `Scratch 2`, the current `Scratch 3` block-based programming environment is based on `JavaScript`. The modular source code of `Scratch 3`[2] is open source and can easily be extended for client-side and/or server-side functionality. This enables the approach of IDE instrumentation by selectively modifying parts of the programming environment, with the goal of recording users' programming actions and resulting programming states.

The rationale of the block-based IDE instrumentation is based on the underlying block programming framework `scratch-blocks`[3], which is based on the `Blockly`[4] `JavaScript` library for visual programming editors. In this framework, each programming action is a block modification, captured as an emitted block event. This means that, by instrumenting the IDE to listen on all block events, all program construction actions and the resulting program states can be logged.

In this instrumentation, two parts of the `Scratch 3` ecosystem have been modified: the `scratch-gui`[5], which renders the graphical user interface, and the `scratch-vm`[6] which manages the internal program states of the sprites, given by the programming blocks, and executes the program code.

The technical structure of the `Scratch 3` IDE instrumentation is given in Figure 3.3. In the following, the elements of the IDE instrumentation are described.

The `Scratch 3` custom server is run to server the IDE instrumentation[7]. The instrumentation is automatically run on the web client in an unobtrusive way.

`1. WSS connection:` On every new client connection, a secure websocket connection to the uniform data collection server `trackserver` is established. When successful, a bidirectional communication channel is established which is used to send program construction data to the server in `JSON` format.

`2. Nickname:` In the `scratch-gui`, the instrumentation enables the use of an URL parameter `username` to set a nickname identifier. The final nickname consists of the user-chosen part and a random identifier assigned by the data collection server

[2] https://github.com/LLK/

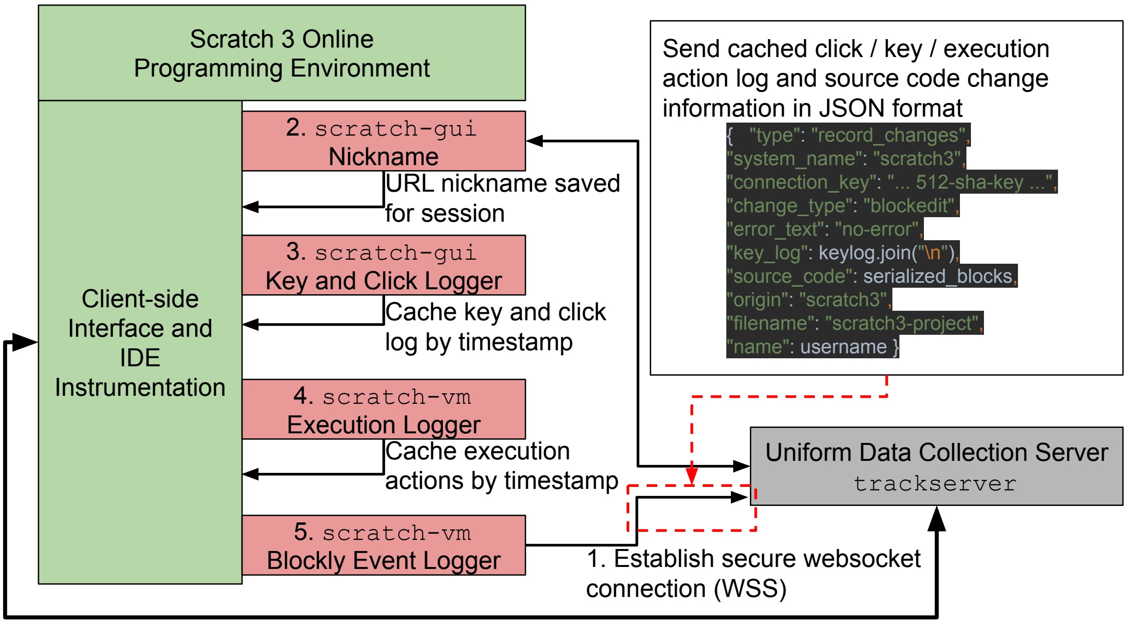
[3] `https://github.com/LLK/scratch-blocks`

[4] `https://developers.google.com/blockly`

[5] `https://github.com/LLK/scratch-gui`

[6] `https://github.com/LLK/scratch-vm`

[7] Custom `Scratch 3` server: `http://seqtrex.aau.at/`


Scratch 3 Online
Programming Environment
Client-side Interface and IDE Instrumentation
2. scratch-gui
Nickname
URL nickname saved for session
3. scratch-gui
Key and Click Logger
Cache key and click log by timestamp
4. scratch-vm
Execution Logger
Cache execution actions by timestamp
5. scratch-vm
Blockly Event Logger
Send cached click / key / execution action log and source code change information in JSON format
Uniform Data Collection Server
trackserver
1. Establish secure websocket connection (WSS)

```
{   "type": "record_changes",
"system_name": "scratch3",
"connection_key": "... 512-sha-key ...",
"change_type": "blockedit",
"error_text": "no-error",
"key_log": keylog.join("\n"),
"source_code": serialized_blocks,
"origin": "scratch3",
"filename": "scratch3-project",
"name": username }
```

Fig. 3.3: Structure of `Scratch 3` IDE instrumentation.

when sending the first data. The final nickname is sent in each message and is saved on the data collection server for the current session. Sharing this nickname across multiple sessions is currently not implemented.

`3. Key and Click Logger:` Also in the `scratch-gui`, a key and click listener is registered that listens on the whole programming interface and caches them with timestamp and with context information (information of the affected HTML node). Key and click events do not trigger sending a new program version.

`4. Execution Logger:` In the `scratch-vm`, an execution listener is registered that listens on interactions with two programming interface buttons: the *greenflag* button for program execution and the *stop all* button for program termination. These events are cached with timestamp. These events do not trigger sending a new program version.

`5. Blockly Event Logger:` In the `scratch-vm`, the main instrumentation in the form of an event logger for all `Blockly` events is found. After executing each block modification event (which represents a program modification initiated by the programmer), the modification action and the resulting new program state of the modified object is sent to the server as a new program version (in the form of a serialized JSON string). See Section 4.2.2.1 and Kesselbacher and Bollin [KB19a] for a detailed overview of block listen events and derived program change events.

The websocket session is used to send new program versions to the server, initiated by block modification events. The `JSON` message consists of: i) the cached log of key, click, and execution actions (each with timestamp) in CSV format, ii) the serialized blocks in JSON format as text, representing the full text source code of the currently modified sprite, iii) identification information of the plug-in (type of recording message, pre-registered system name of tracking software, pre-established SHA-512 connection key for safety), iv) coarse user information (user nickname if set). Note that the supported change type for instrumentation is `blockedit`, as only block modification events trigger sending a new message. No errors are sent in this instrumentation.

The `Scratch 3` IDE instrumentation is independent of any client specifications and works with all browsers that support `Scratch 3`. The instrumentation can therefore be used in a variety of use cases to record block-based program construction. Notably, the IDE instrumentation can be adapted to other block-based programming environments that are based on the `Blockly` framework, as the instrumentation uses emitted block events as the smallest units of program construction.

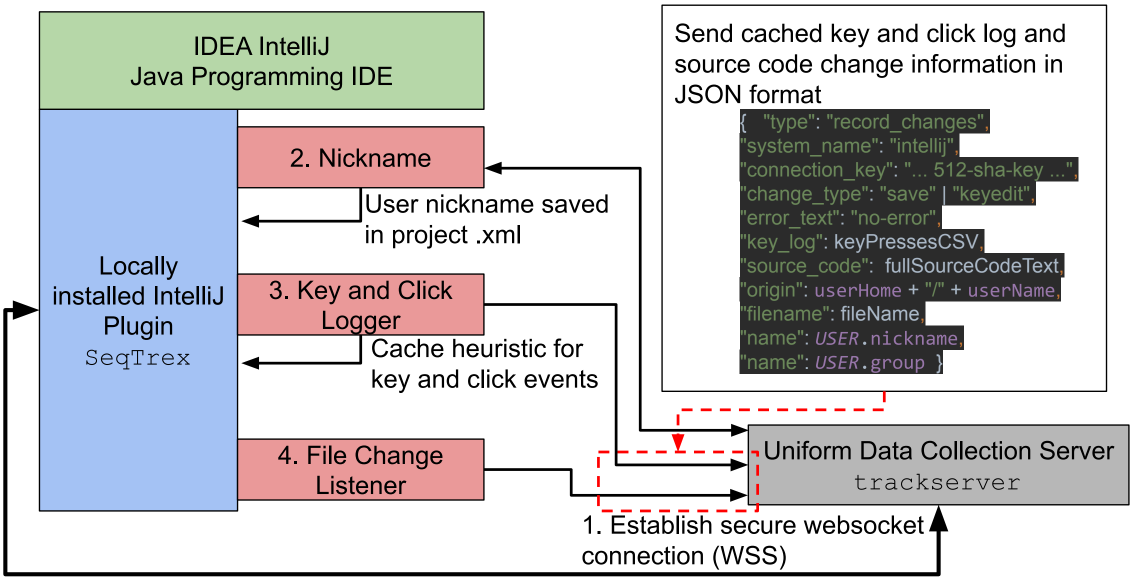
Following the classification of data collection for learning analytics in programming ([IBE$^+$15]), this measurement framework records programming data by means of *IDE instrumentation* and *key logging*. The instrumentation procedure records data on *key stroke* granularity. However, as each programming interaction in `Scratch 3` produces a compilable and executable state in the underlying virtual machine, the data is actually recorded on *compilation* granularity. Following the categorization of LA metrics of Hundhausen et al. [HOC17], the subcategories of *program editing* and *program execution* are recorded. The LA metric categories of *Programming Behaviour* and *Program Content* can be derived from data recorded with the `Scratch 3` instrumentation.

The source code of the `Scratch 3` IDE instrumentation is publicly available: `https://gitlab-iid.aau.at/scratch3-instrumented`.

3.3.3 Text-based IDE Instrumentation: IntelliJ Plug-in

Besides block-based programming, the instrumentation of a text-based programming environment was necessary to record the program construction of programming experts, usually professional programmers. As a programming environment, the IDE `IntelliJ`[8] for `Java` programming was chosen. Three main factors led to this decision: i) `Java` is the first programming language taught at our university and is also a common programming language in the local software development industry, making it easier to acquire potential experiment subjects on different skill levels (novice programmers: school students as well as (under)-graduate students, experienced programmers: employees of software development companies), ii) the IDE `IntelliJ` is used in all introductory `Java` programming courses at our university – its

[8] `https://www.jetbrains.com/idea/`


IDEA IntelliJ
Java Programming IDE
Locally installed IntelliJ Plugin
SeqTrex
2. Nickname
User nickname saved in project .xml
3. Key and Click Logger
Cache heuristic for key and click events
4. File Change Listener
Send cached key and click log and source code change information in JSON format
Uniform Data Collection Server
trackserver
1. Establish secure websocket connection (WSS)

```
{   "type": "record_changes",
"system_name": "intellij",
"connection_key": "... 512-sha-key ...",
"change_type": "save" | "keyedit",
"error_text": "no-error",
"key_log": keyPressesCSV,
"source_code":  fullSourceCodeText,
"origin": userHome + "/" + userName,
"filename": fileName,
"name": USER.nickname,
"name": USER.group  }
```

Fig. 3.4: Structure of `IntelliJ` plug-in IDE instrumentation.

use for instrumentation facilitates recording students in their familiar environment. Moreover, the IDE `IntelliJ` is part of an IDE ecosystem by `JetBrains`, which facilitates the instrumentation of other IDEs of the same ecosystem in the future (e.g., `PyCharm` for `Python` programming, and `AndroidStudio` for specific, Android-based `Java` programming), and iii) for the analysis of the LA metric of `Program Content`, many analysis tools are available for `Java`.

The instrumentation for the IDE `IntelliJ` consists of a plug-in, developed in `Java`, which has to be locally installed by experimental subjects. This way, unobtrusive recording of program construction in a familiar software engineering environment is possible, in contrast to web-interfaces which may not offer the same support in programming (e.g., bulk renaming of variables, loop completion, and hints and completion for object methods). This aligns with the goal of the thesis, uncovering how program construction is performed and what patterns can be identified in it.

The locally installed plug-in captures project file actions and key and click interactions in source files with specified file endings (`.java` for `IntelliJ`) and sends the program construction actions (key and click logs) as well as the program source code to the uniform data collection server.

The technical structure of the `IntelliJ` plug-in IDE instrumentation is given in Figure 3.4. In the following, the elements of the IDE instrumentation are described.

The core `IntelliJ` IDE is not changed by the instrumentation plug-in. The instrumentation plug-in `SeqTrex` has to be locally installed. After installation, recording of program modifications in all files with the file ending `.java` is supported.

1. WSS connection: On startup, the plug-in establishes a secure websocket connection to the uniform data collection server trackserver. When successful, a bidirectional communication channel is established which is used to send program construction data to the server in JSON format.

2. Nickname: The plug-in saves a user-chosen nickname in an .xml file for the current working project to facilitate a long-term collection of program construction data. The final nickname consists of the user-chosen part and a random identifier assigned by the data collection server when sending the first data. The final nickname is sent in each message and is saved on the data collection server to link different programming sessions to the same participant. An optional group name can also be set in the .xml plug-in configuration file.

3. Key and Click Logger: The plug-in registers a listener on keyboard and click actions inside files with the file ending .java. Not every single keyboard and click action is sent to the server – two heuristics are used to limit the number of messages, and thereby data, on the server. First, clicks set a flag of potential change, which only results in sending a new program version to the server if the program code is indeed changed; otherwise, the click action is logged and cached. Second, for keyboard events, alpha-numeric characters are logged and cached – with the rationale of recording full typings of names; all other keyboard events result in sending a new program version to the server.

4. File Change Listener: The plug-in also registers a file change listener that listens on various file management hooks, for example, to detect file creations, deletions, copying, and more. These actions always result in sending a new program version to the server.

The websocket session is used to send new program versions to the server, initiated by keyboard and file change events. The JSON message consists of: i) the cached log of key and click actions (each with timestamp) in CSV format, ii) the filename and the full text source code of the currently modified program file, iii) identification information of the plug-in (type of recording message, pre-registered system name of tracking software, pre-established SHA-512 connection key for safety), iv) coarse user information (user nickname, user group, user.home and user.name retrieved by the JVM). Note that the supported change types for the plug-in are save for messages triggered by the file change listener (as, underlying, a new program version is saved) and keyedit for messages triggered by the keyboard listener. No errors are sent – compilation and analysis are done on server side or during analysis.

For ethical reasons, the plug-in also supports the functionality of disabling click and key logging and listening on file actions – thereby turning off the recording of program construction sequences. This way, long-term collections with the plug-in follow an opt-in model, with participants always in the control of when they will be recorded and when not.

Following the classification of Ihantola et al. [IBE$^+$15], the plug-in records program construction data on *key stroke* and *file save* granularity, with the means of *IDE instrumentation*. Following the categorization of LA metrics of Hundhausen et al. [HOC17], the *program editing* data category is recorded. From the recorded programming actions, the following LA metric categories can be computed: *Programming Behaviour* and *Program Content*. The data collection supported by the plug-in results in fine-grained *key stroke* data, which can be aggregated into *compilation* data by identifying the compilable program versions. The *key stroke* data is retained to enable detailed analysis.

The source code of the `IntelliJ` plug-in IDE instrumentation is publicly available: `https://gitlab-iid.aau.at/seqtrex/intellij-plugin`.

3.3.4 Uniform Data Collection Server

Already beginning with the recording procedure for `Scratch 2` programming, a uniform data collection server was envisioned to support future data collection and analysis approaches. For this data collection server, *uniform* is understood in the following sense. Regardless of the interaction possibilities with the programming environment, the means to modify the program code, and the structure of how the program code is (persistently) presented and saved, there are two main factors in program construction of interest in this thesis, and therefore the target of data collection as well as analysis. First is that program code is **modified by users through specific, well-planned actions**, regardless of these actions being dragging and dropping blocks in a graphical programming interface to construct a stack of blocks, or typing program code, which implies that these actions can be recorded in a sequential way. Second is that program code is modified with **incremental programming actions**, as each individual programming action only affects a usually low number of syntactic programming elements, which again implies that the incremental program versions can be recorded in a sequential way. Summarizing, these two uniform aspects of program construction made it possible to design a data collection server that can be used to store program construction data from a variety of programming environments.

The uniform data collection server is implemented in `NodeJS`[9], which enables lightweight server frameworks for this headless data collection application and enables native `JSON` handling for easy transmission on the secure websocket communication.

The technical structure of the uniform data collection server `trackserver` is given in Figure 3.5. The data collection server consists of two software parts: the websocket endpoints and the database of source code changes.

[9] `https://nodejs.org/`

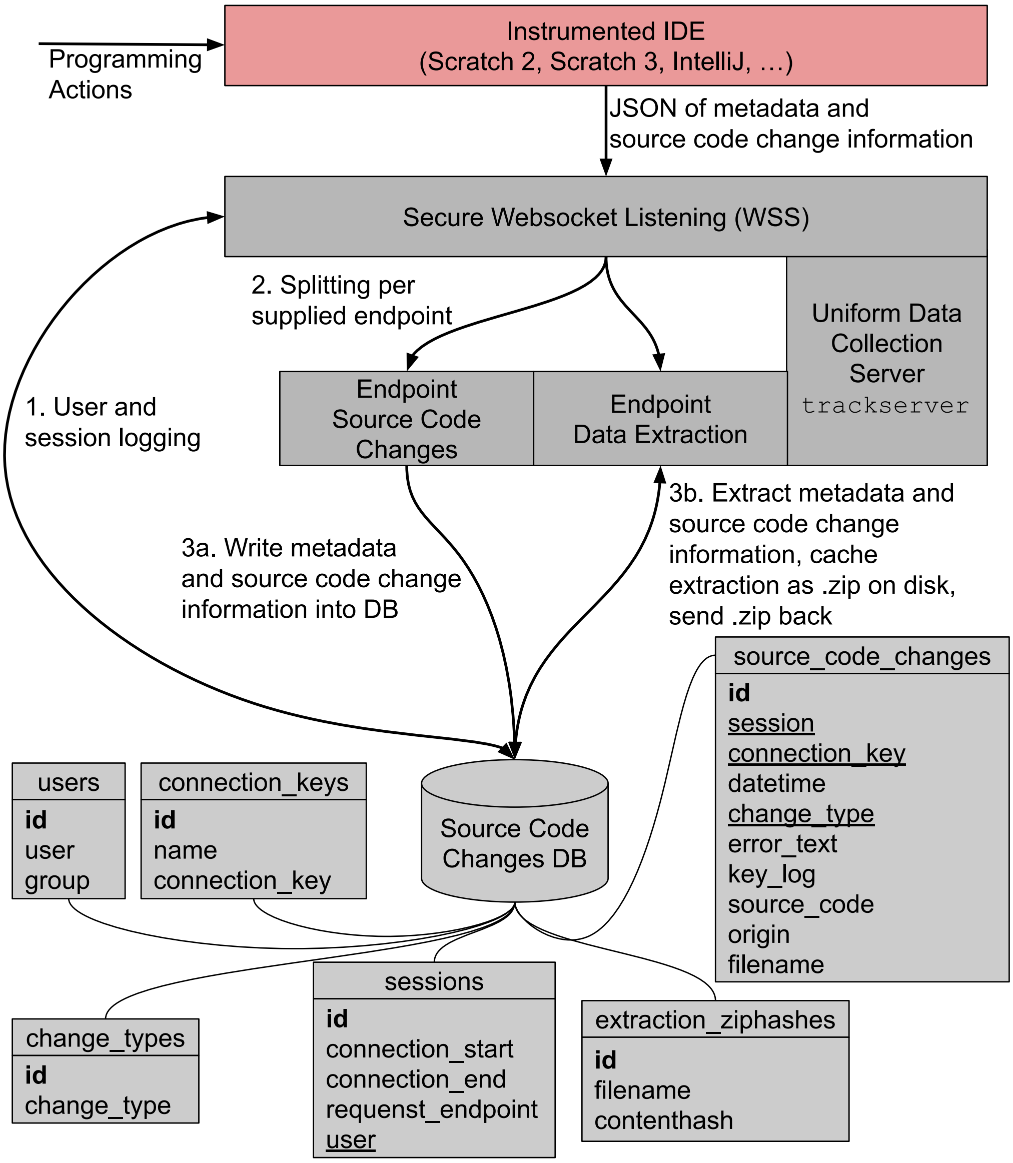

Programming Actions
Instrumented IDE (Scratch 2, Scratch 3, IntelliJ, …)
JSON of metadata and source code change information
Secure Websocket Listening (WSS)
1. User and session logging
2. Splitting per supplied endpoint
Endpoint Source Code Changes
Endpoint Data Extraction
Uniform Data Collection Server trackserver
3a. Write metadata and source code change information into DB
3b. Extract metadata and source code change information, cache extraction as .zip on disk, send .zip back
Source Code Changes DB
users
id
user
group
connection_keys
id
name
connection_key
source_code_changes
id
session
connection_key
datetime
change_type
error_text
key_log
source_code
origin
filename
change_types
id
change_type
sessions
id
connection_start
connection_end
requenst_endpoint
user
extraction_ziphashes
id
filename
contenthash

Fig. 3.5: Internal structure of uniform data collection server `trackserver`.

The uniform data collection server trackserver is headless, does not listen to HTTP or HTTPS connections, and does not offer a graphical user interface. Only secure websocket connections are supported.

1. User and session logging: On each new websocket connection, the data collection session is logged to the database. Moreover, the supplied user name is queried and connected to the session if it exists. Otherwise, the supplied user name is assigned a random alphanumeric suffix to generate the final user name, which is sent back to the instrumented client. There is a single anonymous user that is used to connect anonymous sessions.

2. Splitting per endpoint: The server offers two types of functionality, invoked with the correctly supplied URL endpoints in the JSON message. The first is a service to record the sent source code changes to the database, and the second is a service to extract and download source code changes with specified parameters.

3a. Record source code changes: For the service to record source code changes to the database, the server listens on different websocket URL endpoints in order to invoke the correct method of saving the source code changes. Currently, there are two endpoints for recording source code changes – one for saving Scratch 2 changes, as the picture chunks have to be assembled on server side, and one for saving all other source code changes. All source code changes are saved to the database in the same manner.

3b. Download source code changes: For the service to extract and download source code changes, the server listens on one websocket URL endpoint. These sessions are currently implemented as anonymous sessions. The following specified parameters can be submitted to extract source code changes: start and end date of the recording session, list of strings including the instrumented platforms to be extracted, list of strings including the change types to be extracted, source file name, user name, and error text that is wildcard-searched in the form %text%. The extracted data consists of included sessions, and for each session, all recorded source code changes and associated information that match the extraction parameters are included. The extracted data is cached on the server file system as a .zip that contains sessions as folders and source code changes as text files. The zip hash is recorded in the database for future use. The .zip is sent to the client.

In the database tables, primary keys are shown in **bold** and foreign keys are shown in underline. Some tables are pre-filled and enumerate all currently supported configurations: the connection_keys table stores the name and SHA-512 keys of the registered connections, the change_types table stores all supported source code change types. The users and sessions tables are populated as experimental subjects connect to the server. The extraction_ziphashes table caches the ziphashes of downloaded data. The source_code_changes table, finally, is populated with each recorded program construction action.

Looking back at the categorization of LA metrics of Hundhausen et al. [HOC17], the design of the uniform data collection server currently does not permit the collection of augmented LA metrics categories (like *Social Behaviour*, *Attitudes*, or *Eye Movement Information*). Connecting augmented LA metrics to source code changes is possible via timestamps on the one hand (in order to correctly attribute augmented data to the source code changes), and via user nicknames and their sessions on the other hand. However, this is subject of future work and not currently supported.

The source code of the uniform data collection server `trackserver` is publicly available: `https://gitlab-iid.aau.at/seqtrex/trackserver`.

3.4 Summary of Learning Analytics in Programming

In this chapter, methodologies of learning analytics (LA) in programming have been introduced. Two research frameworks have been identified which help in structuring research approaches for LA in programming:

- The process framework by Grover et al. [GBB$^+$17], blending hypothesis-driven and data-driven approaches, with the potential of improving LA data analysis of (block-based) programming with semantic dimensions. This process framework bears important implications on LA research text-based programming, which often neglects semantic programming components.

- The process model by Hundhausen et al. [HOC17], stating an iterative process for IDE-based LA research. The important notion of this process model is that LA findings can and should be incorporated into the integrated programming environment. Specific to programming education is that learning is happening in the same environments in which the data is generated, making it worthwhile to improve students' interactions with and learning in these environments. This process model shaped many ideas during this thesis' genesis.

Summarizing related work of LA research in block-based and text-based programming, the IDE-based LA metrics most often used are *Programming Behaviour* and *Program Content*. While LA research approaches in text-based programming excel in the static analysis of program construction and using their findings with predictive power, LA research approaches in block-based programming are a step ahead in interpreting their respective findings due to the nature of block-based programming yielding more semantic information of the process of program construction. In this thesis, I aim to combine the best of both worlds by using the power of static analysis while enriching program construction sequences with more semantics.

The IDE instrumentations and the data collection server make it possible to record, store, process, and later analyze program construction sequences. For block-based programming, `Scratch 2` and `Scratch 3` have been instrumented to enable the recording of programming actions and sequential program states. For text-based programming, an `IntelliJ` plug-in has been implemented to instrument the construction of `Java` programs, again recording programming actions and sequential program states. At heart, the basic actions of programming are not dependent on the type of programming environment. Programs are modified by the programmer with **specific, well-planned actions**, and programs are constructed by **incremental programming actions**. By recording these specific actions as well as the resulting program state after each incremental action, the process of program construction can be analyzed in detail regardless of the programming environment.

The LA-based methodology introduced in this chapter facilitates the experiments and analysis methods to answer the remaining research questions. The assumption then is that semantic patterns can be identified in program construction sequences of programmers of varying programming skills.

4. INVESTIGATION AND DEFINITION OF PROGRAM CONSTRUCTION PATTERNS

Patterns in programming, understood as capturing a best practice to solve a programming problem [Ber99], have been a focus of programming research since the early 1990s. This research traditioanlly aims to identify parts of program code that solve a general problem and could be transferred to other instances of a related problem. The book of object-oriented software design patterns by Gamma et al. [GHJV95] is a well-known source of patterns that are regarded as best-practice patterns when designing object-oriented software systems. These patterns can act as a model or template to writing reusable, object-oriented code depending on the desired functionality – they are a canned '*solution to a problem in a context*' [GHJV95, p. 3].

In computer science education, the community of *elementary patterns*, organized by Wallingford [Wal03], work on identifying patterns of different types to improve instructions for novice programmers: Astrachan and Wallingford [AW98] identify loop patterns for searching, processing, and traversing item collections with loops and iterators, and Bergin [Ber99] identifies selection patterns for selecting cases dependent on boolean outcomes, and for writing easily readable `if-else` constructs. Proulx [Pro00] uses the concepts of these *elementary patterns* to create a framework for introductory computer science courses, including elementary programming patterns and design patterns.

While programming patterns have been used by many computer science research and professional communities, there is no precise definition of what constitutes a `pattern`. General definitions of `pattern` can be found in English dictionaries[1], including:

1 A model or design used as a guide in [...] crafts

2 A regular and intelligible form or sequence discernible in certain actions or situations

The patterns discussed so far fit the first definition – they are models used as a guide to solving general problems in the craft of programming. As an example of a

[1] `https://www.lexico.com/en/definition/pattern`

creational design pattern, the *Factory method* pattern [GHJV95, p. 107] describes how to implement a software interface that decouples the construction of an object from the object themselves (constructing them in a factory), thereby letting subclasses govern the specifics of object instantiation. On the other hand, the *selection* pattern of *Sequential Choice* [Ber99] describes the programming style and formatting of how the selection of exactly one out of several possible actions should be programmed (which is, by sequential `if` and `else if` constructs).

While the first design pattern is concerned with the design of object-oriented interfaces at a high abstraction level, the latter elementary pattern is at a lower abstraction level and could also be seen as a style guide for readable code. Following the summary of Winslow [Win96], these elementary patterns are close to the program syntax and are therefore suited for novice programmers. What both pattern types have in common is that they originate from a programmer's semantic understanding of what a program should accomplish (which is partially a general problem, for which the pattern is a best practice solution) – these patterns focus on the strategical planning aspects of writing program code and designing software systems.

In this thesis, I take a different look at programming patterns related to the second definition of patterns. In the learning analytics context, with sequential program construction steps being recordable and analyzable, patterns might be observable in the sequential, temporal construction of program code. This approach is based on the rationale that there are different ways to write even the smallest programs – and again following Winslow [Win96], novices and experts have been proven to focus on different parts of the program (syntactic and control flow elements, respectively semantic and data flow elements). The goal is to identify whether there are patterns in the program construction that distinguish novice and expert programmers, to improve education by teaching the best possible patterns.

I call these *program construction patterns* to indicate the focus on the steps of program construction. The notion of these patterns, a definition, and practical examples are developed in this chapter. This chapter is structured as follows:

1. In Section 4.1, fundamental considerations regarding the notion of program construction patterns adopted in this thesis are discussed. Motivations include programming plans [Sol86], concepts in program construction [RW02], slicing [Wei81, HRB90] and chunking [BRS^{+}97].

2. Section 4.2 covers an exploratory study of syntactic and semantic features that can be extracted from program construction sequences and how those features relate to the adopted notion of program construction sequences.

3. In Section 4.3, the findings from the exploratory study are fleshed out to arrive at a definition of the term *program construction pattern*, which answers the second research question.

4. In Section 4.4, a qualitative and quantitative study is described that focuses on a specific type of pattern, the *variable construction pattern.* In this section, answers to the third research question are developed.

A summary of this chapter is given in Section 4.5, including answers to the second and third research questions of this thesis:

RQ2. How can program construction patterns be defined?

RQ3. What program construction patterns are used by novice and expert programmers during program construction?

4.1 Fundamental Considerations regarding Program Construction Patterns

In this section, a fundamental notion towards *program construction patterns* is developed. I first present background motivations of building a semantic understanding of program construction (Section 4.1.1), followed by approaches to program analysis that build on the semantic understanding of program construction (Section 4.1.2), before finally compounding the notion of *program construction patterns* adopted in this thesis (Section 4.1.3).

4.1.1 Plans, Strategies and Concepts in Program Construction

In this thesis, I strive to uncover additional explanations of how programmers of different skill levels perform their program construction, culminating in the definition, identification, and comparison of *program construction patterns* in sequences of program construction steps. A scientific goal is to understand more about the process of programming, including how to interpret and explain specific steps in a program construction sequence, and what to learn from those explanations for education and training.

Two fundamental considerations have shaped the goal of the thesis and the targeted notion of *program construction patterns*: first the works of Soloway and Ehrlich regarding programming plans and strategies to '*glue together*' the plans [SE84, Sol86], and second the work of Rajlich and Wilde regarding concepts in program comprehension, which are high-level understandings of code parts and their mappings to syntactic elements [RW02]. Although the authors specifically work in the area of program comprehension, by extension, the notion of concepts can also be applied to the subject at hand. With this notion, program construction is pre-comprehension and continuous comprehension of the program code under construction. These works, influential for this thesis, are described in this section.

Soloway and Ehrlich identify a part of programming knowledge that sets apart experts from novice programmers: programming plans, known program fragments that represent '*stereotypical methods for achieving goals*' [Sol86, p. 855]. The notion of programming plans corresponds to the notion of *schemas* in text comprehension, with *schemas* being '*[...] generic knowledge structures that guide the comprehender's interpretations, inferences, expectations, and attention when passages are comprehended*' [SE84, p. 855]. These programming plans are rather fine-grained, e.g., the *Sentinel-Controlled Counter-Loop* plan for a program that counts the number of processed elements in a loop until a sentinel value is processed, and these plans necessitate strategies of effective plan composition to construct whole programs. Four plan composition strategies are presented by Soloway: abutment (gluing plans together), nesting (a plan is completely nested in another plan), merging (interleaving plans as appropriate), and tailoring (no fitting plan is known, adaption is necessary) [Sol86].

With Soloway's understanding, these programming plans play an important, strategic role in program construction. When breaking down a problem into smaller subproblems, ideally those subproblems should be accounted for with programming plans. According to Soloway, this is an important aspect in programming education as novice programmers should be made aware that knowledge of these '*canned solutions*' and the strategies to compose them together is paramount to achieve a suitable macrostrategy for a given programming goal [Sol86, p. 855].

For expert programmers, the notion of programming plans and composition strategies result in the following steps of program construction: i) identifying applicable programming plans, ii) identifying necessary composition strategies, and iii) executing those plans and strategies in the target programming language to arrive at the finished implementation. While the first two points may happen on a sheet of paper, in a design tool, or in the programmer's head, the last point is easily recordable in a learning analytics setting and documents the specific way of how a programmer goes about constructing a program according to the plans and composition strategies. Key questions that can be answered by analyzing such recorded program construction sequences include:

- Are there regularities of how the program construction is carried out that can be ascribed to programming skills? An aim of approaching this question could be to improve the assessment of programming skills.
- Are there *best* ways to implement specific programming plans, adopted and, maybe unconsciously, used by expert programmers? An aim of approaching this question could be to extracting those ways and incorporating them in programming education and training.

For this thesis, the notion is adopted that expert programmers can access generic knowledge structures of partial solutions to construct their programs. However the focus is on the sequential process of program construction itself, with the key questions raised above as guiding anchors in developing the notion of program construction patterns.

Another viewpoint on how programmers build a semantic understanding of program code comes in concepts, presented by Rajlich and Wilde [RW02]. In their work, concepts are defined as '*[...] units of human knowledge that can be processed by the human mind (short-term memory) in one instance*' [RW02, p. 2]. With this definition, the authors include domain concepts (like payment) but also high-level software design concepts (like an iterator pattern) and lower-level concepts related to programming issues (like relevant error conditions). The authors use the notion of concepts in the domain of program comprehension, specifically to describe the problem of concept location: the location of parts of the program code that implement the desired concept.

At heart, concepts represent the mapping of syntactic elements (the program code) to a semantic understanding. By applying cognitive load theory on the notion of concepts, it can be deduced that smaller concepts can be part of larger concepts. It can be supposed that this hierarchical consolidation of concepts to higher-level concepts is subject to the programmers' expertise – specifically the programmers' ability to abstract a more general semantic understanding from larger parts of a software system. In my understanding, it is important that lower-level concepts remain accessible at will (e.g., being unpacked from their consolidated higher-level concepts).

In this thesis, the focus is on program construction instead of program comprehension. However, the notion of concepts is still relevant for program construction, as the role of concepts can be seen as necessarily bidirectional. In addition to the direction relevant to program comprehension (the mapping of syntactic elements to a semantic understanding), the second direction is the mapping of a pre-formed semantic understanding (of the target functionality to be implemented) to the (potentially partial) combinations of syntactic elements that are necessary to express the functionality within program code. Precisely this direction can again be investigated by recording and analyzing specific sequences of program construction, which constitutes the approach demonstrated in this thesis.

Summarizing, both the works of Soloway and Ehrlich [SE84, Sol86] and of Rajlich and Wilde [RW02] pinpoint the craft of programming as an interwoven process of semantic understanding and syntactic mastery. With the notion of program construction patterns developed in this chapter, I strive to contribute to the analysis and explanation of this interwoven process by analyzing and categorizing the sequential steps of actual program construction.

4.1.2 Program Analysis with Slicing and Chunking

After discussing the theoretical approaches and notions to semantic understandings of program code, in this section, two approaches to program analysis are covered that also strive to interweave a syntactic and semantic understanding of program code: program slicing and chunking.

The technique of program slicing was originally devised by Weiser [Wei81] as a formalization of how expert programmers determine the parts of a program that affect the program behaviour at a specified program point. These (sliced) parts of a program are subsequently of interest in certain programming tasks (like debugging and program maintenance tasks). Program slices are usually smaller than the whole program – therefore, the technique of program slicing is a valuable tool to reduce the cognitive load in programming tasks. In the words of Weiser, a program slice is a reduced and independent program [Wei81, p. 439] – a coherent program piece that is not contiguous [Wei82, p. 446].

In this thesis, I build on combined definitions of Weiser [Wei81] and Horwitz et al. [HRB90] and use the following adapted definitions of backward and forward program slices in the context of dependence graphs.

Definition 4.1 (Backward slice)**.** A *backward slice* of a program, with respect to the program statement `p` and a variable `x` defined and/or used at statement `p`, is the set of all statements (and predicates) of the program that might affect the value of `x` at statement `p`. Variable `x` at program statement `p` is the *backward slicing criterion.*

Definition 4.2 (Forward slice)**.** A *forward slice* of a program, with respect to a program statement `p` and a variable `x` defined and/or used at statement `p`, is the set of all statements (and predicates) that might be affected by the value of `x` at statement `p`. Variable `x` at program statement `p` is the *forward slicing criterion.*

When applied on program dependence graphs [OO84], an abstract program representation in which nodes represent the source code statements, variable references and definitions and edges represent the data and control dependencies, the computation of backward and forward slices is reduced to a graph reachability problem.

Horwitz et al. state that program slices can be used to '*isolate individual computation threads within a program*' [HRB90, p. 27]. In this sense, program slicing is a technique applied on syntactic elements (the program code) which yields an approximation of what constitutes the semantically related parts of a program. Expanding this idea to the learning analytics context of the thesis, the computation of program slices at different points of the sequential program construction can make it possible to identify and comprehend strategies in constructing and interweaving those *computation threads* – which might correspond to *trains of thought* [Bol13].

In contrast to program slicing, which is based on the syntactic division of a program, chunking is a program division that focuses on semantic units of a program.

Program chunks emerged from the works of Burnstein et al. [BRS$^+$97] in the context of program recognition, a research area that is concerned with identifying and classifying programs based on their contained semantic units. There have been different approaches to pinpoint those semantic units, such as programming plans or clichés. Burnstein et al. [BRS$^+$97] offer an intuitive as well as a technical definition of program chunks as a way to describe semantic units of programs. The intuitive definition is the following [BRS$^+$97]:

Definition 4.3 (Program chunk intuitive)**.** A chunk is a sequence of software instructions that achieves a coherent purpose and that can be understood outside of the context in which it is used.

Burnstein et al. [BRS$^+$97] also give a technical, programming language-oriented definition of program chunks, based on the fundamental units of structured programs, *programming primes*. For ease of reading and understanding, I alter the definition and substitute *programming prime* with statement, including the following fundamental units: sequential statements, iterative statements (loops), and conditional statements (conditional branching). With this change, the technical definition is the following:

Definition 4.4 (Program chunk technical)**.** A chunk is either a statement (with all containing statements), or a sequence of statements that exist within the same programming scope, where for each pair of statements either one statement is data dependent on the other, or both statements are data dependent on a third statement within the sequence.

With this technical definition, chunks can be found by grouping statements based on shared data elements (variables) and data dependencies. Chunk boundaries correspond to program parts where few data elements are shared. Chunks can be hierarchically arranged based on control dependencies.

To summarize, both program slicing and program chunks combine a syntactic and semantic view on program code, albeit at different focal points. Program slicing focuses on the syntactic elements in its first step, while program chunks as a notion are inherently semantic. In this thesis, the syntactic elements obtained from sequential program construction sequences (the program code) can be considered main objects for analysis – because of this, the approach of program slicing is better suited. The technique of program slicing to uncover syntactic and semantic program features is further investigated in Section 4.2.

4.1.3 Towards the Notion of Program Construction Patterns

Approaches to reason about and analyze the relationship of syntactic and semantic elements of program code have been summarized. Programming plans and combination strategies on the one hand [Sol86] and program slicing on the other hand [HRB90] have been identified as key pieces in the development of the notion and definition of program construction patterns. The next step is to focus on the process of program construction and its sequential nature to determine fundamental units of program construction, which can then be part of patterns. The goal is to establish basic terminology that facilitates an exploratory analysis of features of program construction sequences (described in Section 4.2).

The approaches and techniques described so far are usually applied on single versions of finished programs – programs that are committed to version control systems or handed in as assignment or experiment solutions. In contrast, recording the program construction with an IDE-based learning analytics approach on *keystroke* granularity produces numerous program versions, which sequentially differ by the last recorded set of keystrokes and approach the final, last recorded program version.

To determine the fundamental units of program construction relevant for this thesis, I revisit the notion of how programs are modified and constructed established in Section 3.4. I established that programs are constructed with **specific, well-planned actions** that are carried out **sequentially**. This places weight on the single actions of program constructions carried out during the sequential generation of program versions. Regardless of the programming environment (block-based or text-based), these single actions can include changes of three fundamental types:

- **Addition** of syntactic elements, which comprises the addition of new syntactic elements
- **Modification** of syntactic elements, which comprises the modification of existing syntactic elements
- **Deletion** of syntactic elements, which comprises the deletion of existing syntactic elements

It is important to note that a single action can include multiple fundamental change types. As an example, an **addition** or a **deletion** of a variable reference on the right hand side of a variable assignment entails a **modification** of the affected variable definition. All single actions, applied sequentially from the start program state, represent the program construction as performed by the programmer and generate the final version of the program code.

The next step is to identify measurable units in those sequential, single actions, possibly containing multiple change types. There are two problems when recording on *keystroke* granularity: i) single actions might touch a programming element several times in succession before arriving at the desired outcome (e.g., constructing a multi-digit integer can be broken down in multiple single modification actions of typing the digits, but is most likely a single **well-planned** action in the programmer's intention), ii) single actions might generate program code that is not compilable simply because the **well-planned** action is not completed by a single action (e.g., constructing a new boolean comparison includes the addition of left-hand and right-hand expressions and a comparison operator).

The first problem is tackled in the `IntelliJ` instrumentation plug-in by caching all alpha-numeric key strokes, so that typed text like variable names and literals are grouped to a single change. In the block-based instrumentation, this caching is not implemented as each key stroke produces a block change event that is logged.

The second problem is does not exist in block-based programming, as the programs are always compilable. To arrive at measurable units of program construction that are comparable between block-based and text-based programming, I introduce the notion of a `program increment`. The idea is to group together the program change actions between two compilable program versions, which represent an *increment* from the starting program version towards the final program version (regardless of the included change types). `Program increments` are defined as follows:

Definition 4.5 (Program increment)**.** A sequence of single program change actions and the included changes (addition, modification, and deletion of syntactic elements) that produces a compilable program version, starting from a compilable program version.

Note that the sequence of single actions is retained – the order of the single program changes matters and can be analyzed.

The sequence of `program increments` generates the sequence of compilable program versions, also called `intermediate programs`, between the start version (`start program`) and the end version (`end program`) of the program construction. The `intermediate programs` all consist of well-defined and well-ordered sets of syntactic elements (programming statements or sets of linked blocks) that support different types of analysis. Additionally, the `end program` represents, both syntactically and semantically, the programming plans employed by the programmer. Therefore the relation of single program change actions and corresponding `intermediate programs` to the `end program` is of particular interest.

The notion of sequential `intermediate programs` as program states reached during programming, and the notion of `program increments` as transitions between program states (captured by the **well-planned, sequential actions** executed by programmers), facilitate a rich analysis in multiple dimensions. This concludes the fundamental considerations towards *program construction patterns* and establishes the necessary fundamental, measurable units of program construction sequences that can be further analyzed.

4.2 Syntactic and Semantic Features of Program Construction Sequences

In this section, I report on an exploratory study of syntactic and semantic features that can be extracted from program construction sequences, based on the notions of fundamental program change action, *program increments*, and the resulting sequence of *intermediate programs* introduced in Section 4.1. The purpose of this exploratory study is to narrow down the possible ways to analyze data of program construction sequences, with the goal of developing a notion of *program construction patterns*. First, the example problems used to record block-based and text-based program construction sequences are introduced. Next, my employed methods to extract syntactic features and semantic features of program construction sequences are explained, respectively. Lastly, I discuss the analysis and visualization of program construction sequences and extracted features within my introduced notions.

The analysis of features in block-based program construction sequences only serves an exploratory purpose, to inform and guide the definition of `program construction patterns`, which is related to answering the second research question of the thesis. For the subsequent research questions, I focus on text-based program construction sequences.

The findings of the exploratory study are summarized at the end of this section and provide a foundation to develop a definition of *program construction patterns* in Section 4.3. The structure of the exploratory study is visualized in Figure 4.1.

4.2.1 Example Problems for the Exploratory Study

For the exploratory study, two example problems have been devised to facilitate the extraction and analysis of features in program construction sequences – features that might be indicative of program construction patterns. These problems have been crafted to allow for different syntactic solutions (different programming constructs can be used to arrive at a functional solution) and, in the case of the text-based problem, different semantic solutions (different strategies can be employed to solve the problem). These are described alongside the problem, as applicable. This openness in the example problems facilitates a rich potential for data analysis.

First, I describe the cohort and the example problem used to record program construction sequences of block-based programming in `Scratch 2` and `Scratch 3`. Studies with this example problem have already been published [KB19b, KB19a], and I base the description of the example problem on these publications.

Table 4.1 shows the student cohorts participating in the exploratory block-based study, and the solution type counts for the example problem. The first two cohorts consist of upper secondary school students that worked as interns at our department. The students had varied programming skills (ranging from no programming skills to

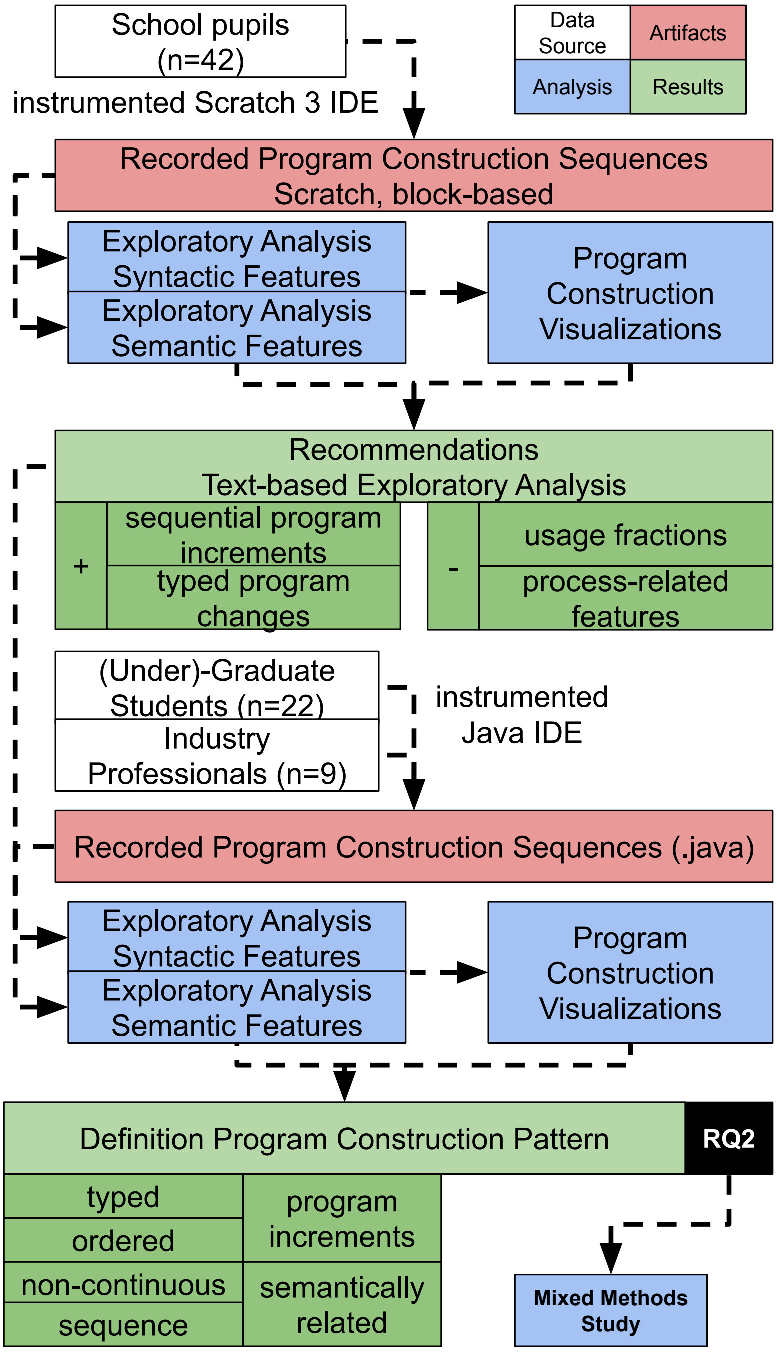

School pupils (n=42)
instrumented Scratch 3 IDE
Data Source
Artifacts
Analysis
Results
Recorded Program Construction Sequences Scratch, block-based
Exploratory Analysis Syntactic Features
Exploratory Analysis Semantic Features
Program Construction Visualizations
Recommendations Text-based Exploratory Analysis
+
sequential program increments
typed program changes
-
usage fractions
process-related features
(Under)-Graduate Students (n=22)
Industry Professionals (n=9)
instrumented Java IDE
Recorded Program Construction Sequences (.java)
Exploratory Analysis Syntactic Features
Exploratory Analysis Semantic Features
Program Construction Visualizations
Definition Program Construction Pattern
RQ2
typed
ordered
non-continuous
sequence
program increments
semantically related
Mixed Methods Study

Fig. 4.1: Structure of the exploratory study.

Tab. 4.1: Cohorts of school students that implemented the block-based example problem for the exploratory study [KB19a].

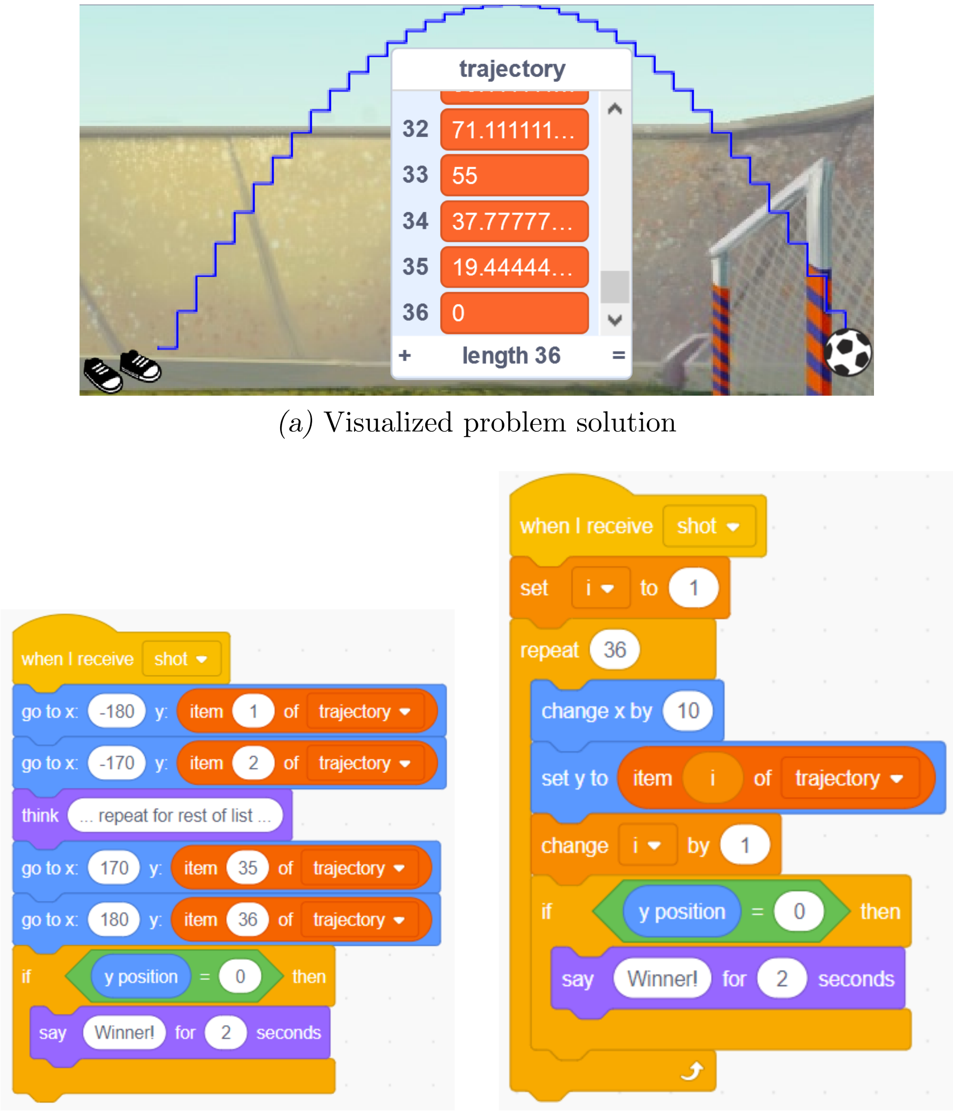
			Solution	*Solution*	*Solution*
Cohort	***Age***	***N***	***None***	***Linear***	***Loop***
Interns 2018	15–18	8	1	0	7
Interns 2019	15–18	6	3	1	2
School Upper	14–15	8	6	1	1
School Lower	13–14	20	14	6	0
Total		42	24	8	10

programming skills in multiple text-based programming languages), and worked on the example individually. The second two cohorts represent two computer science classes in different settings. In both classes, block-based programming in `Scratch` had been covered by instructions. The upper secondary school students (cohort *School Upper*) were measured during a compulsory computer science class. The lower secondary school students (cohort *School Lower*) were measured during an elective computer science class.

One cohort, *Interns 2018*, has been recorded with the `Scratch 2` recording procedure. Their program construction sequences have been manually replayed in the instrumented `Scratch 3` IDE. All other cohorts have been recorded in the instrumented `Scratch 3` IDE.

In the example problem of the block-based study, students had to move a soccer ball `Scratch` object on a pre-computed trajectory path and check whether a fixed, marked point in the goal was hit with the trajectory. The trajectory is stored in a `Scratch` list variable with 36 elements, denoting the flight heights (absolute coordinates on the y-axis) that are given in 10–point distances (coordinate change on the x-axis). The students received an information sheet that specified the programming task, received a prepared `.sb3` Scratch file to open in the instrumented Scratch 3 environment, and had a maximum of 20 minutes to complete the task.

Figure 4.2 `(a)` shows a visualization of a correct solution (the painted trajectory), and also shows two student-developed end programs exemplifying two different syntactic solution approaches. A `(b)` *linear solution* to the problem consists of accessing each value in the trajectory list one by one to extract the sequence of y-coordinates. The movement of the ball object can be achieved with different blocks of the type *move*: either hard-coding the x-coordinates or implementing a sequential change of the x-coordinate by 10 units (as specified in the task). A `(c)` *loop solution* to the problem consists of a fixed-size loop (as specified in the task), an iteration variable to access the values in the trajectory list, and corresponding blocks of the type *move*



```
trajectory
32  71.111111...
33  55
34  37.77777...
35  19.44444...
36  0
+   length 36   =
```

(a) Visualized problem solution

```
when I receive shot
go to x: -180 y: item 1 of trajectory
go to x: -170 y: item 2 of trajectory
think ... repeat for rest of list ...
go to x: 170 y: item 35 of trajectory
go to x: 180 y: item 36 of trajectory
if y position = 0 then
    say Winner! for 2 seconds
```

```
when I receive shot
set i to 1
repeat 36
    change x by 10
    set y to item i of trajectory
    change i by 1
    if y position = 0 then
        say Winner! for 2 seconds
```

(b) Exemplary linear solution

(c) Exemplary loop solution

Fig. 4.2: `(a)`: Visualization of the example problem used to record program construction sequences of students programming in `Scratch 2` and `Scratch 3`. `(b+c)`: Two exemplary end programs generated by students. The left program shows a snippet of a linear solution. The right program shows a looping solution, correctly applying the programming concepts of loops, conditionals and variables.

to set x– and y-coordinates in each loop iteration. End programs of students that did not solve the problem are very diverse.

Second I describe the cohort and the example problem used to record program construction sequences of text-based programming in `Java` [KB21].

Tab. 4.2: Cohorts of students and professional programmers that implemented the text-based example problem for the exploratory study.

		Solution convertBinaryArray			
Cohort	***N***	***None***	***Decimal***	***Hexadec.***	***Both***
(Under)-Graduate Students	17	10	0	0	7
Industry Professionals	9	3	1	0	5
Total	26	13	1	0	12

```
public class ConvertBinary {
/* The method converts the input array of 8 bits,
* and prints corresponding decimal and hexadecimal numbers.
* Example input: convertBinArr({1,1,0,0,0,0,1,1})
* Example output: decimal: 195 / hexadecimal: C3 */
    void convertBinaryArray(int[] bN){ /* TODO: implement */ } }
```

Fig. 4.3: Example problem used to record program construction sequences in `Java`.

Table 4.2 shows the cohorts participating in the exploratory text-based study, and the solution type counts for the example problem. Participants from two populations have been recruited. The first cohort is composed of undergraduate and graduate students from a Software Engineering project management course at the University of Klagenfurt in the winter term 2018/2019. All students attending this course have successfully completed at least two basic programming courses with *Java*. On a scale from 1 to 10, the students rated their own programming skills at 6.15 ± 1.59.

The second cohort is composed of professional programmers from an Austrian software development company, recruited in Summer 2019. The recruited professional programmers reported a mean programming experience of 17.29 ± 9.62 years and rated their own programming skills at 7.86 ± 1.73 out of 10.

The difference in the self-rated programming skills is not significant (an independent two-sample t-test yields a p-value of .0585). However the years of programming experience could also have an effect on the self-perceived programming skills (meaning that professional programmers might rate themselves lower compared to confident student programmers).

The example problem used in the text-based study is shown in Figure 4.3. The participants had to implement the method `convertBinArr(...)` in *Java*, and their program construction was recorded with the instrumented `IntelliJ` plug-in. The data of the different cohorts was separated so that a comparative analysis is possible.

The example problem was specifically crafted to allow for multiple ways to solve the problem, both syntactically and semantically. In relation to syntactic solutions, participants may elect to use loops and conditional branching, but both concepts are not strictly required to solve the problem. Only mastery of the programming concepts of variables, of array accesses, and of basic mathematical transformations are required. In relation to semantic solutions, participants may elect to compute the decimal and hexadecimal number directly from the input binary array or elect to compute one number from the binary array and compute the second number from the first number.

Concluding, the program construction sequences of school students, (under)-graduate students, and professional programmers implementing the presented example problems have been recorded with the recording and instrumentation measures introduced in Section 3.3. These program construction sequences constitute a repository for an exploratory extraction and analysis of features of program construction sequences. This way, developed programs can be analyzed as opposed to devised programs – a detail important in the context of this thesis with a focus on differences in the program construction of novices and expert programmers.

4.2.2 Block by Block – Line by Line: Extraction of Syntactic Features

In this subsection, the methods used to extract the syntactic features of program construction sequences are introduced. For the extraction of syntactic features of block-based and text-based program construction sequences, separate categorizations of the *program increments* that represent the sequences of program changes are introduced. Syntactic features are extracted along these categorizations, analysis and visualization is carried out in Section 4.2.4.

4.2.2.1 Syntactic Features in Block-based Program Construction Sequences

From the cohorts described in Table 4.1, a mean number of 326 ± 194 programming interactions with the IDE have been recorded. To further analyze these sequences with regards to syntactic features, these programming interactions are first categorized according to: the interacted **(T) block types**, the underlying type of **(E) block listen event**, and the resulting type of **(P) program change event**. This categorization is also described by Kesselbacher and Bollin [KB19a] and is summarized here.

Table 4.3 shows the three categories. Each programming interaction is a **(E) block listen event**. A meaningful sequence of such block listen events is categorized into **(P) program change events**, as described below.

The **(E) block listen events** are directly derived from the block change events observed by the `Scratch 3` virtual machine when a user interacts with the programming environment and makes changes to the program blocks.

Tab. 4.3: Categories to classify programming actions of program construction sequences recorded with the `Scratch 3` instrumented IDE [KB19b].

(T) Block Type

T1 motion	**T6** event	**T11** sound
T2 looks	**T7** control	**T12** sensing
T3 pen	**T8** loop	**T13** user-defined
T4 var	**T9** conditional	**T14** extensions
T5 lists	**T10** operator	

(E) Block Listen Events

E1 create	**E8** create-var-local	**E15** move-comment
E2 change	**E9** create-var-global	**E16** delete-comment
E3 move	**E10** rename-var-local	**E17** stackclick
E4 delete	**E11** rename-var-global	**E18** greenflag
E5 end-drag	**E12** delete-var	**E19** stopall
E6 outside-drag	**E13** create-comment	
E7 end-drag-onto	**E14** change-comment	

(P) Program Change Events

P1 add-program	**P6** detach-nonprogram	**P11** block-change
P2 add-nonprogram	**P7** reorder-program	**P12** immediate-delete
P3 attach-program	**P8** reorder-nonprogram	**P13** block-move
P4 attach-nonprogram	**P9** delete-program	**P14** block-click
P5 detach-program	**P10** delete-nonprogram	

E1 *create* events capture block creations, **E2** *change* events capture changes to block fields, **E3** *move* events capture block movements after block drags, **E4** *delete* events capture block deletions, **E5** *end-drag* events capture block drags. **E6** *outside-drag* and **E7** *end-drag-onto* events capture block drags that leave the program area and can result in copying blocks to other objects. The events **E8**–**E12** handle variable and list management, the events **E13**–**E16** handle comment management.

Interactions that start and stop the program execution are additionally instrumented and captured. **E17** *stackclick* events represent a user's click on a stack of blocks to execute or evaluate them, **E18** *greenflag* events represent a user's click on the greenflag symbol that starts general execution, and **E19** *stopall* events represent a user's click on the stopall symbol that stops general execution.

The block listen events **E1**–**E5** represent single actions and have to be bundled to form meaningful **(P) program change events**. Most of the program change events (**P1**–**P10**) are divided in two modes. Change events with the suffix *–program* indicate that the change affects blocks connected to a main executable block[2]). Change events with the suffix *–nonprogram* indicate that the change affects blocks in the non-executable part of the program.

The addition of new blocks to the program (**P1**, **P2**) is classified from the following sequence: **E1** the creation of the new block from the `Scratch 3` category, **E5** the end of the drag event, and **E3** the movement of the block to the new destination.

The block listen event **E3** *move* leads to different program change events, depending on the context of the block movement. Attaching a block to another block (**P3**, **P4**) is classified from a move event that adds a new parent block to the moved block and adds a new next block to the block attached to. Detaching a block from another block (**P5**, **P6**) is classified from a move event that removes the parent block from the moved block and removes the next block from the block detached from. Reordering of a block stack (**P7**, **P8**) is classified from a move event that specifies old and new parent blocks in the moved block. Attach and reordering are often preceded by detach events, when blocks are moved from one program part to another. A simple block move (**P13**) is classified from a move event that does not change block order.

The deletion of blocks (**P9**, **P10**) is classified from the following sequence: **E5** the end of the drag event, **E3** the movement of the block to the new destination, and **E4** deletion of the block.

Detailed changes to block field parameters (**P11**) are classified from **E2** change events. Program change events of **E3** *move* changes can also change block fields (e.g., moving a variable block into a block field), but are then classified as the respective move event (attach, detach or reorder).

Immediate deletion of blocks (**P12**) is the creation and deletion of blocks from

[2] (Scratch wiki regarding Hat blocks: `https://en.scratch-wiki.info/wiki/Hat_Block`

the Scratch 3 block categories with the same drag event, and is similar to block deletion with a preceding create event. It is classified from the following sequence: **E1** the creation of the new block, **E5** the end of the drag event, **E3** the movement of the block to the new destination, and **E4** deletion of the block. The difference to normal deletion events (**P9**, **P10**) is that immediate deletion does not cause any program block changes.

Block clicks that execute the stack of blocks (**P14**) are classified from **E17** *stackclick* events.

For each program change, the block types of interacted program blocks are categorized. Most **(T) block types** correspond to the Scratch 3 block type categories: **T1** *motion*, **T2** *looks*, **T6** *event*, **T10** *operator*, **T11** *sound*, **T12** *sensing*, **T13** *user-defined*, **T14** *extensions*. The following block types are categorized more fine-grained. The Scratch 3 category *Variables* (internally called data) is split into two categories: **T4** *var* for blocks that deal with variables, and **T5** *lists* for blocks that deal with lists. The Scratch 3 category *Control* is split into three categories: **T8** *loop* for loop blocks, **T9** *conditional* for conditional branching blocks, and **T7** *control* for all other Scratch 3 blocks of this category. The block type **T3** *pen* is a Scratch 3 extension, but is included as a separate block type to ensure backward comparability with the study of Kesselbacher and Bollin [Kes19].

For analysis and visualization, the overall usage fraction of each category type, and the sequence of use for each category type are considered. Moreover, two derived measures employed by Kesselbacher and Bollin [Kes19] are considered: the maximum number of used block types in each intermediate program and the geometric mean change rate of used block types.

4.2.2.2 Syntactic Features in Text-based Program Construction Sequences

Inspecting the recorded data of students and professional programmers described in Table 4.2, three students and one professional programmer did not produce any intermediate programs for the example problem. The remaining students generated a mean number of 309±228 bundled *key stroke* actions that result in a mean number of 19±14 *intermediate programs* ($n = 14$). The professional programmers generated a mean number of 290±168 bundled *key stroke* actions that result in a mean number of 15 ± 6 *intermediate programs* ($n = 8$). Following, I introduce the categorization used to extract syntactic features of text-based program construction sequences.

The program construction sequences, the sequences of *intermediate programs*, are each made of one or more (potentially small) changes to single source code statements. The key challenge in this kind of data is to identify features that make it possible to associate program changes to development with intent, while only depending on syntactic information, at best automatically extractable.

To tackle this challenge, inspiration is drawn from tree-differencing approaches employed in static program analysis. These approaches aim to identify and classify changes between two different versions of a program (e.g., to locate changes) by comparing the program in the representation of an abstract syntax tree (AST). A prominent example of a tree-differencing approach that supports the classification of fine-grained source code change types is `ChangeDistiller`[3] [FWPG07, GPF09]. With `ChangeDistiller`, the AST differences between two `.java` source files can be computed. The result is a '*minimum edit script*' to transform the first AST into the second AST. The individual tree changes are classified according to the change type and a significance level [FG06].

From this classification scheme, a two-dimensional change taxonomy could be constructed for use in this thesis, consisting of the changed AST entity and the classified change type (omitting the significance level as it is not designed for this use). However, the fine-grained change categorization of `ChangeDistiller` [FWPG07, GPF09], stemming from the AST comparison approach, is very detailed – potentially too detailed to extract meaning from differently categorized changes in the data that is recorded for this thesis. A factor is that the sequential *intermediate programs* extracted from the program construction are expected to have a small number of changes between them – a hypothesis stemming from the *key stroke* nature of the data. These changes need to be categorized along dimensions not tied to syntactic program construction (i.e., the viewpoint of a static program analysis tool) but along dimensions tied to semantic program construction (i.e., the viewpoint of expert programmers).

Still drawing inspiration from the change categorization, I now introduce my framework to categorize changes of (text-based) program construction sequences. The basic idea is that *program increments* represent a single (incremental) change unit, containing one or more program change actions that each change a single program statement. I introduce four dimensions along which each program change action can be categorized: **fundamental program change type**, **variable usage type**, **changed program statement type**, and **control context**. The dimensions are shown in Table 4.4.

The first dimension is the type of **fundamental program change**, introduced in Section 4.1. This dimension distinguishes three types: `addition` of syntactic elements, `modification` of syntactic elements, and `deletion` of syntactic elements.

The second dimension encompasses the type of **variable usage** that is affected by the corresponding fundamental program change. This dimension distinguishes two types of **variable use**: changes to the `definition` of a variable (most frequently as left-hand side in assignment statements) and changes to the `use` of a variable (either as right-hand side in assignment statements, as parameter in non-returning

[3] Open source: `https://bitbucket.org/sealuzh/tools-changedistiller/wiki/Home`

Tab. 4.4: Dimensions to categorize *program increments* and included program change actions of text-based program construction sequences.

Fundamental Program Change Type		
addition	modification	deletion

Variable Usage Type	
definition	use

Changed Program Statement Type		
data	control	other

Control Context	
iteration	conditional
method	class

method calls, or as variable in control statements like conditional branching statements).

The third dimension is the type of **program statement** that is affected by the corresponding fundamental program change. This dimension distinguishes three types of **program statements** and respective sub-types. The first type, `control` program statements, encompasses statements that exert control dependency, with the sub-types of *iteration* (specifically *for*, *while*, and *do/while* iteration statements), and *conditional* (specifically *if/else*, *switch*, and *case* conditional statements). Conditional error handling, e.g., in the form of *try/catch*, is not considered at the moment. The second type, `data` program statements, encompasses statements that make use of data (i.e., variables) and are part of a program's data flow, excluding `control` program statements. The third type, `other` program statements, encompasses all other statements that do not directly interact with a method's dependence graph. This includes statements without data use or exerted control dependency (e.g., printing literals to the console) and declaration of classes and methods.

The fourth dimension is the type of control structure encompassing the corresponding fundamental program change, called the **control context**. There are four main types and respective sub-types. The `iteration` context has the sub-categories *for*, *while*, and *do/while*. The `conditional` context has the sub-categories *if*, *else*, *switch*, and *case* (*try/catch* error handling is not considered at the moment). The contexts of `method` (**control context** for method statements with no other control dependency) and `class` (**control context** for method and field declarations) do not sport any sub-categories.

These four dimensions are designed in a way to capture, for each program change action, the effectively possible changes for a statement and possible change effects (along the data and control dependencies) while still relying on syntactic information. As each *program increment* consists of one or more program change actions, the change types of a *program increment* can be interpreted as the sequence or the union of change types of the included program change actions.

Besides the categorization of changes in the introduced four dimensions, I employ a second approach to analyze program construction sequences and extract syntactic features. While the categorization of each individual *program increment* handles syntactic features in a small context (sequential, step-wise convergence to the intended *end program*), another approach is needed to observe syntactic features in a larger context – the whole *end program*. For this approach, I propose to match each *changed* program statement in a *program increment* to a program statement in the *end program*, if possible. To match changed program statements in the *intermediate programs* to the *end program*, the `Gumtree`[4] AST matching algorithm is used [FMB+14].

The output of this approach is, for each changed program statement in a *program increment*, the matched program statement in the *end program*. This way it can be observed what parts of the *end program* are changed, and in what sequence. Moreover, unmatched program statements can provide additional insights on the program construction sequence, especially when paired with their categorized dimensions (e.g., parts of a program that have been deleted before the *end program*).

To summarize, two approaches are combined and used for the analysis and visualization of text-based program construction sequences. First is the classification of *program increments* along the four dimensions (**fundamental program change type**, **variable usage type**, **changed program statement type**, **control context**). Second is the relation of *program increments* to program statements of the *end program* and the respective sequence of changes in relation to the *end program*.

4.2.3 Beyond Blocks and Lines: Extraction of Semantic Features

In this subsection, the methods used to extract the semantic features of program construction sequences are introduced. The categorization of *program increments* that is used to analyze syntactic features is based only on syntactic program change information. Semantic features, on the other hand, encompass characteristics of the program construction sequence that hold meaning as one unit. For block-based program construction sequences, the employed method to identify semantic features of program construction sequences is *association rules mining* to combine syntactic features into semantic units. For text-based program construction sequences, a method to compute slice-based cohesion metrics on the granularity level of variables is employed to identify semantic program blocks during program construction. Analysis and visualization of these semantic features are carried out in Section 4.2.4.

[4] Open source: `https://github.com/GumTreeDiff/gumtree`

4.2.3.1 Semantic Features in Block-based Program Construction Sequences

To recap, a mean number of 326 ± 194 programming interactions with the block-based IDE `Scratch 3` have been recorded from the exploratory cohort of school students (Table 4.1). These programming interactions are categorized with the three categories introduced above: **block listen events**, bundled **program change events**, and interacted **block types** (where applicable).

No specific data preparation is employed to extract semantic features of block-based program construction from these categorized sequences. However, I employ clustering with k-means as a syntactic means to partition different program construction sequences. This makes it possible to subsequently interpret the characteristic differences, thereby generating sets of syntactic features that could represent semantic features of the program construction sequence.

4.2.3.2 Semantic Features in Text-based Program Construction Sequences

To recap, from the program construction sequences to solve the text-based example problem, a mean number of 19 ± 14 *intermediate programs* have been recorded from participating students, and a mean number of 15 ± 6 *intermediate programs* have been recorded from participating professional programmers. In addition to the syntactic dimensions that are used to categorize the *program increments* to extract syntactic features from the construction sequences, slice-based cohesion metrics are used to identify semantic units during text-based program construction.

[KB21]

The underlying assumption is that programmers follow a specific strategy when constructing their program, in accordance with the notion of programming plans and strategies introduced by Soloway and Ehrlich [SE84, Sol86]. The challenge now is to identify whether specific program change actions (captured in *program increments*) contribute to a single strategical goal and belong together as a semantic unit or whether there are multiple *trains of thought* a programmer works on in quasi-parallel succession. This information can be used to qualify the categorized *program increments.*

The notion of *trains of thought* during programming has already been investigated by Bollin [Bol13]. He shows that, in the realm of `Z` specifications, a high slice intersection corresponds to single trains of thought. This result indicates that measuring cohesion is a reasonable way to investigate a common design principle of software systems: software units (regardless of them being specification units or implementation units) should only have a single responsibility. Extending this notion to the program construction sequence, the evolution of slice-based cohesion metrics can provide additional insight into how related programming parts are constructed.

In this thesis, the computation of slice-based cohesion metrics is based on *slice* *profiles* of methods introduced by Ott and Thuss [OT93]. Informally, a *slice profile* is a collection of static slices for method variables that, together, make it possible to measure the cohesion of the method (as employed by Ott and Thuss [OT93]), or the cohesion of single statements (as employed by Krinke [Kri07]). In this section, I introduce an adaption to the computation of *slice profiles* to include variable cohesion and cohesive method parts for each variable.

In this context, Ott and Thuss originally worked on unions of forward and backward slices, calling them *metric slices* [OT93]. For this thesis, the focus is to identify semantic units during program construction. To alleviate the emphasis on the metrics aspect, I simply call the union of forward and backward slices *union slice*. I introduce an adaption to the computation of *union slices* that makes it possible to compute multiple *union slices*, each capturing a different semantic unit (if there are multiple ones in an *intermediate program*).

The basic computation as a union of backward and forward slices in a method remains the same: the backward slice is computed from the last **reference** of the slicing variable, the forward slice is computed from the **definitions** of the slicing variable that are included in the backward slice. However, I propose to only compute the forward slice from the first definition, and only compute *union slices* from backward and forward slicing criteria not covered in another *union slice* for the same variable. This way, multiple *union slices* can be computed for a slicing variable: iteratively computing pairs of backward and forward slices for references and definitions of the slicing variable not covered so far, until all of them are included in at least one *union slice*. This computation necessarily terminates, as the smallest possible *union slices* are those that only include a single or both slicing criteria – *union slices* are never empty. A step-by-step description of the algorithm to compute the slice profile of a method PDG can be found in Section A.1.

Following are the definitions of *slice profiles* and *union slices*.

Definition 4.6 (Slice Profile)**.** The *slice profile* of a method contains, for all local variables and input parameters, a list of *union slices* that each form a cohesive part of the method.

Definition 4.7 (Union Slice)**.** A(n) (iterative) *union slice* is the union of a backward slice (for the last remaining variable reference) and a forward slice (for the first remaining variable definition included in the backward slice) for a local variable or input parameter of a method.

With these definitions settled, next is the adaption of slice-based cohesion metrics to include cohesion measures on the granularity level of variables. Here I only introduce the considerations for the adaption. The formalized metrics can be found in Section A.2.

Previous work by Weiser [Wei81], Longworth [Lon85], and Ott & Thuss [OT93] introduced slice-based cohesion metrics to measure the cohesion of methods based on slices, namely *Coverage*, *Overlap*, *Tightness*, *Parallelism*, and *Clustering*.

Krinke [Kri07] introduced additional slice-based cohesion metrics to compute the cohesiveness of single source code statements based on slice profiles and transitive dependence between input and output variables.

In these previous works, the computation of cohesion metrics differs in multiple elements: the slicing criteria, the variables to be included in backward or forward slices, and the statements to be included in the slices. For the computation employed in this thesis, the computation is based on an input method program dependence graph (PDG). The detailed computational decisions are:

1. The computation is based on the *slice profile* of the PDG, containing all *union slices* for all local variables and input parameters of the method.

2. There is no distinction between input and output variables when computing the cohesion metrics.

3. The *union slices* only contain PDG nodes that correspond to method statements. Formal nodes (input parameter, output nodes) are not part of the *union slices*. Single lines of source code could correspond to multiple nodes in the PDG (e.g., for-loops correspond to three nodes: initialization, loop test, post-iteration statement).

With the introduced notion of *slice profiles* and *union slices*, I employ the following three slice-based cohesion metrics to identify semantic units in sequentially generated *intermediate programs* on the granularity level of variables.

The first metric is *Coverage*, which computes the mean length of the *union slices* for a variable relative to the length of the method. Interpreted, this cohesion metric captures how much of the method is covered, on average, by the *union slices* for a variable – in other words, how much the semantic units of a variable contribute to the method.

The second metric is *Overlap*, which computes the mean fraction of program statements common to all *union slices* for a variable relative to these union slices. Interpreted, this cohesion metric captures how much program code is shared between a variable's *union slices*, or respectively how big the difference between the specific *union slices* is.

The third metric is *Tightness*, which computes the fraction of program statements included in all *union slices* for a variable relative to the length of the method. Interpreted, this cohesion metric captures how large the shared program code between a variable's *union slices* is compared to the whole method.

The slice-based cohesion metrics introduced above only provide a picture of cohesion for each individual variable. They do not help in assessing the relation between two variables and the respective semantic units. Consider two variables, each with a *Coverage* of 0.5. Possible relations of these variables are: a) they cover the same parts of the method, b) they cover disjoint parts of the method, or c) they have any intersection in method parts. To improve the interpretative power of the metrics, I also introduce **pairwise** slice-based cohesion metrics on variable level. Again, the formalized metrics can be found in Section A.2.

Pairwise cohesion metrics are based on the idea to consider different sets of program statements of the *union slices* and the method and to adapt the metrics computation for a variable pair (v, t) (of slice variable v and target variable t), including statements of different types (with respect to target variable t) in the union slices of v. This way, the cohesion of two variables with regard to the method statements can be interpreted. The considered **statement inclusion types** are: all references to the target variable t, all definitions to the target variable t, and references and definitions to the target variable t.

The same three metrics are considered for pairwise computation (*PairCoverage*, *PairOverlap*, *PairTightness*). For a given pairwise metric and statement inclusion type, a matrix of pairwise cohesion metrics can be computed by iterating through all variables as slice variables, and respectively target variables. The resulting *method cohesion matrix* provides an overview of the pairwise cohesiveness of method variables and parameters. In this matrix, slice variables constitute the rows, while target variables constitute the columns.

The values of these metrics range from 0 to 1, with higher values indicating higher cohesion (of a variable, or a pair of variables, in the context of a method). As these metrics can be computed on each *intermediate program*, the resulting data for identifying semantic features consists of the *slice profile* for each method in the sequential *intermediate programs*, a set of cohesion measures for each variable, and corresponding *method cohesion matrices* for all cohesion metrics, statement inclusion types and pairs of variables.

The introduced slice-based cohesion metrics help to identify semantic units during program construction in two ways. First, the *trains of thought* during program construction can be identified, with the assumption that single *trains of thought* manifest in high cohesion for all variables, while multiple and possibly divergent *trains of thought* manifest in low cohesion for a set of variables. Second, the semantic relation between variables can be measured by considering the pairwise cohesion, with the assumption that variables contributing to the same semantic unit have high pairwise cohesion, while variables of different semantic units have low pairwise cohesion. Verifications of these assumption are explored in Section 4.2.4.

4.2.4 Analysis and Visualization of Extracted Features

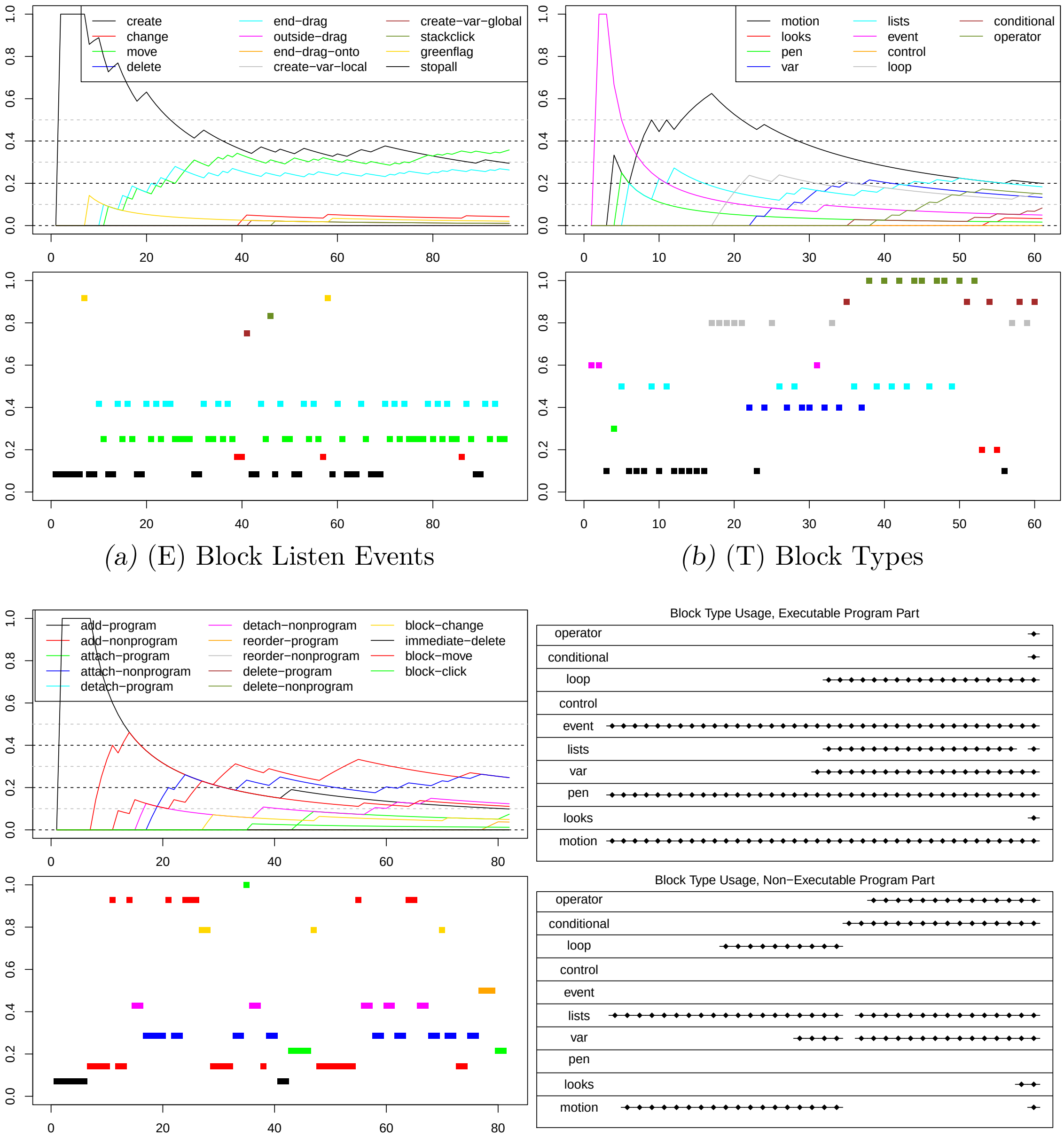
For both block-based and text-based program construction sequences, the categorization of syntactic features and means to extract semantic features have been introduced. Next is a two-step exploratory analysis of the program construction data recorded for the example problems. First is the analysis of block-based construction sequences, serving as a precursor for the analysis of text-based construction sequences. The aim is to investigate how potential *program construction patterns* can be identified in program construction sequences.

4.2.4.1 Exploratory Analysis of Block-based Construction Sequences

The syntactic data for each sequential program construction of the block-based example problem consists of sequential, categorized programming interactions in the three categories **(E) block listen events**, the bundled **(P) program change events**, and corresponding interacted **(T) block types**. This data can be viewed through a cumulative lens of fractions of programming interactions per category and through a sequential lens.

Figure 4.4 `(a)`–`(c)` provides an example visualization of the considered metrics, including cumulative frequencies (top) and sequential use (bottom) for the three categories along which block-based programming interactions in `Scratch 3` are classified. The x-axis represents the sequence of program changes in all figures. The y-axis represents the fractions of categorized programming interactions per program change in the respective figures on the top but represents the type, offset for visual clarity, in the respective figures on the bottom. Figure 4.4 `(d)` additionally shows the change of used block types present in the executable and non-executable program parts, showcasing the program construction sequence of a loop solution.

Every figure alone only provides a single point of view of the program construction sequence – making it possible to identify single aspects but not overarching features of program construction. Regarding fine-grained **(E) block listen events** (Figure 4.4 `(a)`), of note is that the block listen events *create*, *move*, and *end-drag* alone do not carry much meaning – they have to be interpreted with the type of program change event. The block listen events *change* and *delete* represent actions that immediately affect the program. The block listen events *create-var-local* and *create-var-global* do not affect the program immediately but serve an important role in the program construction sequence (especially for this example problem). The block listen event *greenflag* represents a main program execution, which was employed by the student two times, and *stackclick* represents an execution of a stack of blocks. The usage fraction converges to a distribution of block listen events concerned with creating and ordering the program – execution does not take a great fraction of this program construction sequence.



(a) (E) Block Listen Events

(b) (T) Block Types

(c) (P) Program Change Events

(d) Block Type Usage

Fig. 4.4: Four visualizations of one program construction sequence (cohort *Interns 2018*, loop solution), aiding the analysis of block-based program construction data. The visualizations in `(a)`–`(c)` show cumulative frequencies (top, y-axis shows cumulative frequency) and sequential use (bottom, y-axis shows type) of all occurring program interactions of the respective types. The visualization in `(d)` shows the change of used block types in (non)-executable program parts (y-axis shows type). The x-axis always represents the sequence of program changes.

Regarding the **(P) program change events** bundled from the block listen events (Figure 4.4 `(c)`), the program construction sequence offers an organized sequence. The student mostly worked in the non-executable program part (*add*, *attach*, *detach* in the non-executable program), and only performed a small number of *add* and *attach* actions in the executable program part. There are a number of inconsequential *block-move* events. The usage fraction also showcases this type of programming in the non-executable program part, with high converging usage fractions of *add* and *attach* actions in the non-executable program part.

Regarding the **(T) block types** (Figure 4.4 `(c)`), interpretation of this category heavily depends on the example problem. As such, this can be a useful diagnostics of whether students display mastery of or struggle with a programming concept. Moreover, it can be a simple diagnostics of interaction sequences with block types. For this example problem, a balanced use of many different block types was needed – therefore, successful program construction sequences of this example converge to a usage fraction that is balanced in all necessary block types (*motion*, *var*, *lists*, *control*, *loop*, *conditional*, *operator*). For the exemplary program construction sequence, there is a shift in focus from the first half of the program construction (focusing on *motion*, *lists*, *loop*, and *variable* blocks) and the second half of the program construction (focusing on *var*, *lists*, *conditional* and *operator* blocks). This coincides with the two task parts of the example problem.

A richer interpretation and analysis is possible by considering the data categories as dimensions and qualifying the sequence of program changes along those dimensions simultaneously. This approach is considered for the analysis of text-based program construction sequences.

The information of type usage in executable and non-executable program parts (Figure 4.4 `(d)`) can be utilized in different ways. A change rate of used block types can be computed by dividing the number of changes to used block types by the number of program change actions, individually for executable and non-executable program parts. With the data of the example problem, Kesselbacher and Bollin have shown that a compound measure of geometric mean type change rates of both program parts is a strong indicator of the level of programming mastery required for this problem [KB19a, KB19b]. This finding stems from the fact that, in order to solve the problem with a loop solution, many different block types have to be used. For an optimal solution, students only need a small number of interactions with each block type, resulting in a high geometric mean type change rate, as reported in the publications. The utility of this data depends on the example problem and possible solutions.

Moving from a single program construction sequence, I now investigate how different syntactic features of the program construction sequences relate to success in the programming trial, with success being encoded in the following way: 0 for

Tab. 4.5: All significant Spearman correlations ($p < 0.05$, corrected with the Benjamini-Hochberg procedure to control the false discovery rate) between fractions of category types and success in the example problem. Success in the example problem is numerically encoded: 0 for no solution, 0.5 for linear solution, 1 for loop solution [KB19a].

n=42	***Success***
T2 looks	**0.55**
T4 var	**0.44**
T6 event	**-0.40**
T8 loop	**0.43**
T9 conditional	**0.58**
T10 operator	**0.47**

n=42	***Success***
E9 create-var-global	**0.72**
E18 greenflag	**-0.43**
P7 reorder-program	**0.48**
MaxBlockTypes	**0.55**

no solution, 0.5 for linear solutions, and 1 for loop solutions. These results have first been presented in Kesselbacher and Bollin [KB19a]. The syntactic features are the fractions of categorized programming interactions at the *end program*. See Table 4.5 for all significant correlations of usage fractions to the encoded success in the block-based programming cohort. For correlation analysis, I employ Spearman correlation ($p < 0.05$), corrected with the Benjamini-Hochberg procedure to control the false discovery rate.

The left table shows all correlations to usage fractions of **(T) block types**. There are three groups of measures, interpreted in the context of the example problem. First are the block types representing programming concepts (*var*, *loop*, *conditional*, *operator*), which are strictly required to solve the problem with a loop solution and exhibit a moderate, positive correlation to success. Next is the *looks* category, which is needed in the second part of the example problem and also exhibits a moderate, positive correlation to success. The last is the *event* category with a moderate, negative correlation to success – students interacting with this category, which was not needed for a solution, were less likely to solve the problem.

The right table shows all correlations to other fractions, and to the maximum number of block types present in the student solutions. The high correlation to the *create-var-global* block event and to a high maximum number of block types again highlights that multiple different block types, and variables in specific, are required to solve the problem with a loop solution. The *greenflag* block event, representing the main program execution, exhibits a moderate negative correlation to success. Interpreted, students that solve the example problem tend to execute the problem significantly less frequently. For program change events, only the *reorder-program* action is significantly correlated to success.

Tab. 4.6: Results of clustering students' programming sequence data with k-means, $n = 42$. Cluster 2 represents the linear solution, cluster 3 represents the loop solution. Clusters 1 and 4 capture students who did not solve the example problem [KB19a].

Cluster	*Solution None*	*Solution Linear*	*Solution Loop*	*Total*
1	14	1		15
2	4	7		11
3	1		10	11
4	5			5
Total	24	8	10	42

Concluding the correlation results, little features of the program construction sequences have emerged, apart from block types usage (which is tied to the example problem). A notable feature is the fraction of execution actions, which could be interpreted in neo-Piagetian terms: as a novice programmer is not capable of tracing the program, the need of execution as verification arises. Similar process-related features of program construction have already been employed by Carter et al. [CHA15] and will not be employed for the analysis of text-based program construction sequences.

Altogether, while this analysis incorporates data from the whole program construction sequence (the converged usage fraction), specific features during program construction are not considered. As such, this approach is not retained for the analysis of text-based program construction sequences.

The correlation analysis of usage fractions did not provide much guidance for patterns of program construction. The next analysis step is to identify syntactic feature sets that can discriminate different program construction sequences. These results have first been described in Kesselbacher and Bollin [KB19a]. Clustering with k-means is employed with usage fractions, block type change rate measures, and maximum block type measures as input data. Note that success in the trial is not part of the clustering data. 4 clusters are found to be the appropriate number of clusters for the data, employing the elbow technique. Table 4.6 provides an overview of the distribution of solution types for each cluster, and Figure 4.5 shows category types with at least one significant difference between any of the clusters, measured with the Mann-Whitney-U test ($p < 0.05$) .

There are two main types of clusters. The first type, encompassing cluster 1 (left-most boxes in Figure 4.5) and cluster 4 (right-most boxes in Figure 4.5), pre-

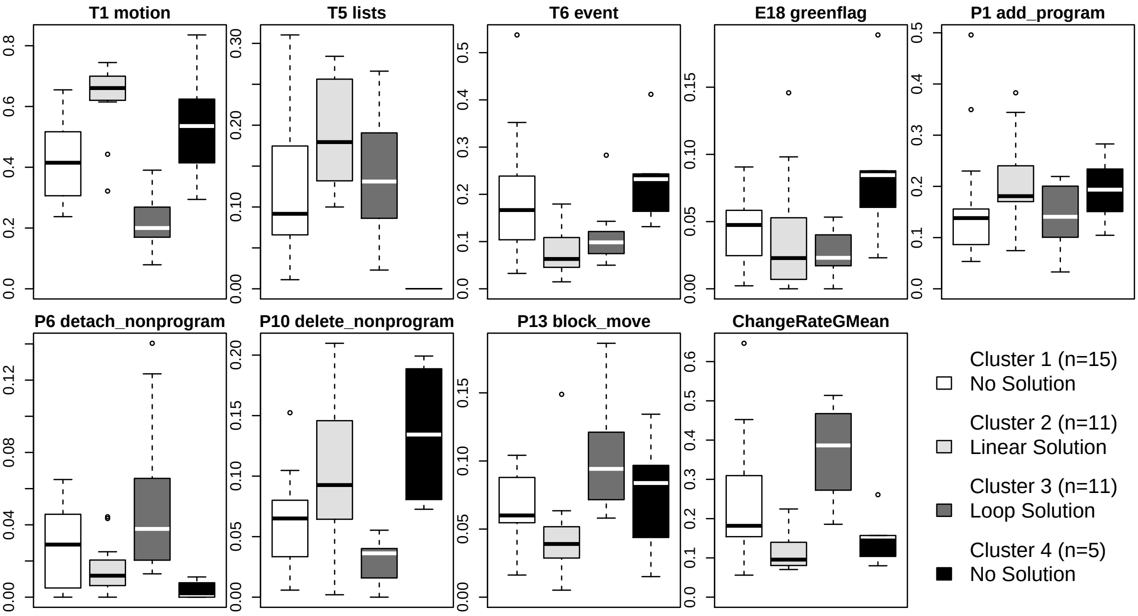


Fig. 4.5: Distribution of students' usage fractions for selected category types, divided into the four clusters. All presented category types exhibit at least one significant difference between any of the clusters [KB19a].

dominantly captures program construction sequences with no solution to the example problem. These clusters execute the program most often, further amplifying the interpretation of the novice programmers' need to execute because of their inability to trace. The execution is done either through *greenflag* main execution (cluster 4) or, not shown, through local block execution with *stackclick* actions (cluster 1). Interpreted, this could mean that cluster 1 novices have a better understanding of the program code, and only need specific parts executed. Further analysis with the significant differences reveals that the clusters indeed capture different ways to attempt the program construction. While cluster 1 students try to utilize different block types, cluster 2 students do not employ any blocks that represent programming constructs (not shown).

The second type of cluster, encompassing cluster 2 (second to left box in Figure 4.5) and cluster 3 (second to right box in Figure 4.5), captures program construction with a solution to the example problem. Cluster 2 distinctively captures program construction sequences with a linear solution, with the characteristic that dominantly *motion* and *lists* blocks are used in the construction sequence. Moreover, these students tend to execute the program less often compared to cluster 1 and cluster 4, which could mean that their tracing abilities are higher compared to the novices – they are likely to have developed tracing skills for some programming concepts but need specific support for other concepts (such as variables and loops).

Cluster 3 distinctively captures program construction sequences with a loop solution. In line with cluster 2 representing students of higher programming skills, this cluster also features a low fraction of program execution actions. In contrast to cluster 2, the students of cluster 3 demonstrate mastery of all necessary block types, which is captured in the balanced use of them (showcased with the highest geometric mean change rate, while not all block types are shown). Students of cluster 2 have likely developed sufficient tracing skills with all basic programming concepts, and have to be supported on their path to post-tracing skills.

Summarizing sets of usage fractions are well-suited to differentiate program construction sequences, and even allow for, albeit careful and speculative, interpretation of programming skills based on program construction data. However, this type of data does not produce tangible features of program construction sequences that can be understood as semantically motivated *program construction patterns*. Therefore, this approach is not retained for the analysis of text-based program construction sequences.

Concluding the analysis of features of block-based program construction sequences, the exhibited analysis approaches are summarized with regard to their use for text-based program construction sequences. To repeat, this analysis serves as a precursor to investigate how features of program construction sequences can be used to capture *program construction patterns*.

The following analysis approaches are **not retained** for the upcoming analysis of text-based program construction. First is data in the form of usage fractions of syntactically categorized *program increments*. While these measures demonstrated power in differentiating successful and unsuccessful solution attempts and can even discriminate between different types of solution attempts, the underlying nature of those solution attempts and the program construction remains shrouded. Specific, tangible features of program construction sequences cannot be uncovered with this analysis approach. Second are process-related features of program construction, for examplerecords of program execution. While these measures are important for assessing program construction as a whole, they are of lesser importance in the current effort of this thesis to identify semantically motivated features in program construction sequences.

The following analysis approaches are **considered** for the upcoming analysis of text-based program construction. Program changes, captured in *intermediate programs* and *program increments*, are considered in a sequential fashion. Moreover, data categories are not considered in isolation of one another but are considered simultaneously by qualifying single program changes along the categorical dimensions. The hypothesis is that this yields more context-dependent program construction data that can be used to capture *program construction patterns*.

```
private int convertBinaryArray(int[] binaryNumber){
  int sum = 0;
  int multiplier = 1;
  for(int index = 0; index < binaryNumber.length; index++){
    sum += multiplier * binaryNumber[index];
     multiplier *= 2;   }
  System.out.println(”decimal:” + decimal);
  System.out.println(”hexadecimal:”
    + convertIntToStr(decimal / 16)
    + convertIntToStr(decimal % 16));   }
```

Fig. 4.6: Exemplary solution of a professional programmer to the text-based example problem, implemented in *Java*.

4.2.4.2 Exploratory Syntactic Analysis of Text-based Construction Sequences

For the analysis of syntactic features of text-based program construction sequences, four dimensions along which individual *program increments*, and thereby the changes resulting in the sequential *intermediate programs*, are classified have been introduced: the **fundamental program change type** of each program change, the type of **variable usage**, the type of the **changed program statement**, and the **control context** of the changes.

In this section, I conduct a step-wise analysis of exemplary program construction sequences by incorporating one or more of those dimensions in an incremental way until all available syntactic features are used to provide the maximum amount of information. This step-wise analysis results in **five layers** of incremental information.

I use two exemplary program construction sequences recorded for the example problem to showcase different aspects that can be investigated with the syntactic and semantic features at hand. The first exemplary solution is implemented by a professional programmer, with the *end program* given in Figure 4.6 and Figure 4.10. The professional's program construction sequence is used to establish the first four layers of incremental information. The professional's implementation consists of a loop to compute the integer number from the binary number. Computation of the two hexadecimal digits is outsourced to another implemented method, depicted in Figure 4.10.

The second exemplary solution is implemented by an undergraduate programmer, with an *intermediate program* and the *end program* given in Figure 4.9. The student's program construction sequence is used to establish the fifths layer of incremental information. Both implementations are used to investigate the capabilities of the semantic features. The undergraduate's implementation consists of multiple control constructs, incorporating looping and conditional branching to compute both the integer number and the two hexadecimal digits directly from the binary number.

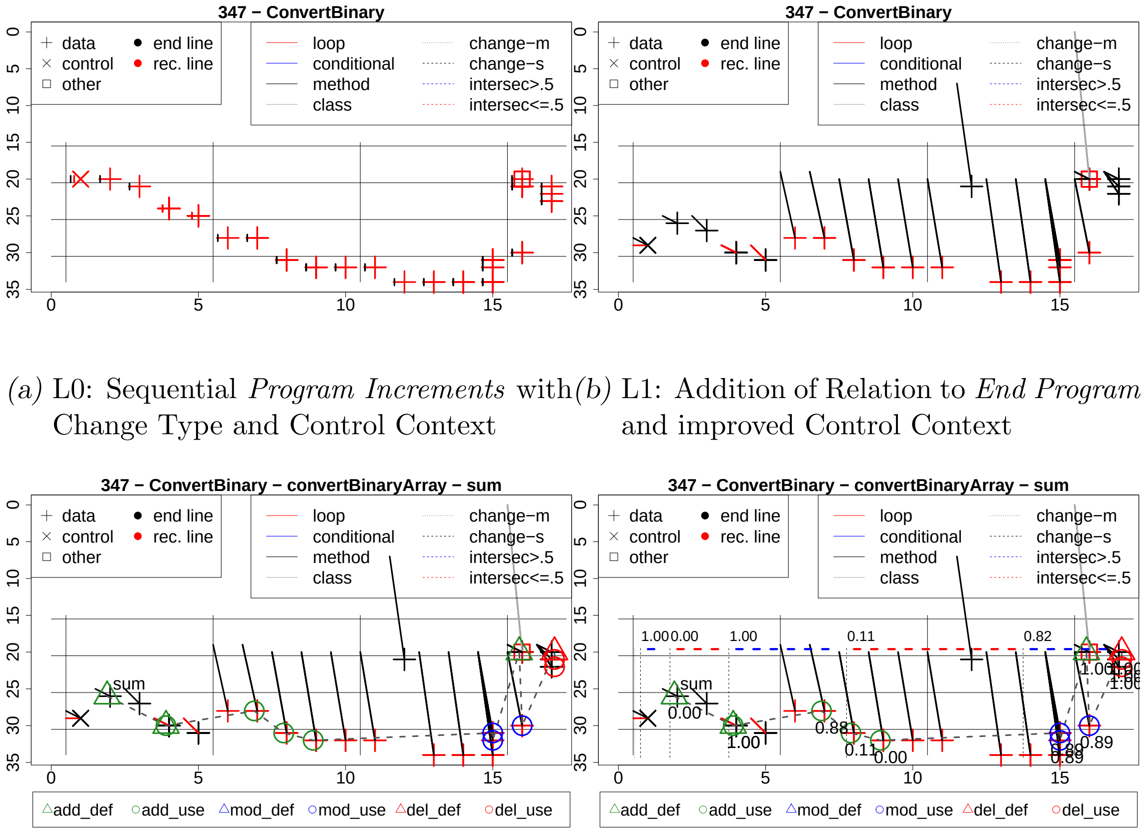
The selection of these two implementations is for the purpose of exploratory showcase investigation and follows no direct methodology. All program construction sequences of student programmers and professional programmers are investigated with a proper methodology in Section 4.4.

I now establish the first four layers of incremental information of program construction sequences based on syntactic features. For each layer, I introduce the included dimensions of syntactic features and potential use of this information. As a reference, see Figure 4.6 for the *end program* and Figure 4.7 for a depiction of the layered information regarding the construction of the *end program.* In this context, the depictions in Figure 4.7 include, regardless of the information layer, the sequential *program increments* and thereby represent the *flow* of program construction. I call these depictions *construction flowlines.*

The basic layer `L0` depicted in Figure 4.7 (`a`) displays the sequential *program increments* with the relative, recorded change source code line number, and includes two dimensions of syntactic features on top of the sequence. First is the type of the **changed program statements**, differentiating between **control** (statements that exert control influence, foremost iteration and conditional statements), **data** (non-control statements that include any use of variables), and **other** (not included in the other two types; e.g., static print statements). This dimension dictates the shape of the depicted *program increment* in the figure. The second dimension is the **control context** of *program increment*, with the four main categories of **iteration**, **conditional**, **method**, and **class**. In this layer, control context information cannot be attributed to another line number and can only be given for each *program increment* as a visual marker.

Layer `L0` is a minimal depiction of a *construction flowline*, missing many pieces to aid in investigating and qualifying steps in the program construction. Extractable information of a program construction sequence include: the fraction and sequence of types of **changed program statements**, contrasting data and control statements, as well as the fraction and sequence of changes relative to the **control context**. Infering from the pre-study on features of block-based program construction sequences, fractions and sequences do not provide enough information to investigate and qualify pattern structures in program construction sequences – additional layers of information are needed.

At the next layer `L1`, depicted in Figure 4.7 (`b`), two missing pieces of information are added to the previous layer to enrich the *construction flowline.* The first addition is, in contrast to `L0`, the relation of each *program increment* to the corresponding



(*a*) L0: Sequential *Program Increments* with Change Type and Control Context

(*b*) L1: Addition of Relation to *End Program* and improved Control Context

(*c*) L2: Addition of Typed Variable Changes

(*d*) L3: Addition of Slice Intersection Information

Fig. 4.7: Stepwise enrichment of a program construction sequence (carried out by a professional programmer for the example problem). The final program version is shown in Figure 4.6. The figures show sequential *program increments* on the x-axis, and the respective program line number on the y-axis. The stepwise enrichment contains: (a) sequential program increments and their relative change line, with the type of changed program statement and control context, (b) enrichment with relations to lines of the *end program* (black) and enriched control dependency, (c) enrichment of typed variable changes for an exemplary variable, and (d) enrichment with slice intersection information and transition between cohesive (> 0.5) and non-cohesive ($<= 0.5$) *intermediate programs*.

program statement in the *end program*, made visible with colour and the respective source code line number. The second addition is a more detailed **control context** resulting from *end program* lines. In this layer `L1`, it is now possible to differentiate the specific control context of each *program increment* by indicating the source code line number of the program statement that exerts the control influence. The specific control context is indicated by colour.

Compared to layer `L0`, the additional information in layer `L1` makes it possible to assess the sequential *program increments* with regard to their respective position in the *end program*. Of note is that program changes that only affect temporary statements (for example statements that are removed before the *end program*) cannot be mapped to a respective end source code line. This provides insight on a valuable aspect of the program construction sequence: which changes *survive* the program construction and contribute to the *end program*, and which do not. However, information on the individual *program increments* is still missing.

Following, the layer `L2` depicted in Figure 4.7 (`c`) improves on previous layers by adding qualitative information for each individual *program increment*, covering the remaining two dimensions of structural features. First is the **fundamental program change type** of each program change, of the types *addition* (*add*), *modification* (*mod*), and *deletion* (*del*), which is depicted by colour for each change in a *program increment* and further differentiates those changes. Second is the type of **variable usage**, of the types *definition* (*def*) and *use*, which is depicted by shape for each change in a *program increment*.

Two considerations are fundamental to the addition of these two dimensions. The first consideration is that the two dimensions are of particular interest as a *view* on the construction of a specific variable, which leads to the variable-specific *construction flowline*, as depicted in Figure 4.7 (`c`). It is thereby possible to track a qualified construction sequence for each variable in a method. In this context of specific variables, the second consideration is that each *program increment* that changes variable usage is necessarily qualified by both dimensions simultaneously. This leads to the combined depiction of these dimensions as shape and colour for each *program increment*. Take note that multiple types of **fundamental program changes** and of **variable usage** can be present in a single *program increment*.

With the additional information introduced in layer `L2`, all four introduced dimensions of structural features are incorporated in *construction flowlines*. The program construction sequence can thereby be investigated based on the qualified sequence of *program increment*. However, the information of this layer cannot differentiate semantic structures in the program construction, which can be very relevant when considering intertwined variable-specific *construction flowlines* that might work on the same, but also on different semantic program structures. Additional layers are needed to introduce this information.

For the next layer `L3`, depicted in Figure 4.7 (`d`), information on the semantic program structure is incorporated based on the relative method slice intersection (the intersection of all union slices in the method slice profile). This information is incorporated in two forms. First by denoting the state of method cohesion, differentiating between cohesive *intermediate programs* with an intersection > 0.5 and non-cohesive *intermediate programs* with an intersection $<= 0.5$, depicted by colour stripes on top of the program construction sequence. The cohesion threshold is selected with the rationale that, above this threshold, all union slices should share sets of semantically meaningful method statements.

Second by denoting, for each change in the *program increments*, the relative method slice intersection of the *intermediate program* to provide additional information on the specific evolution of cohesion.

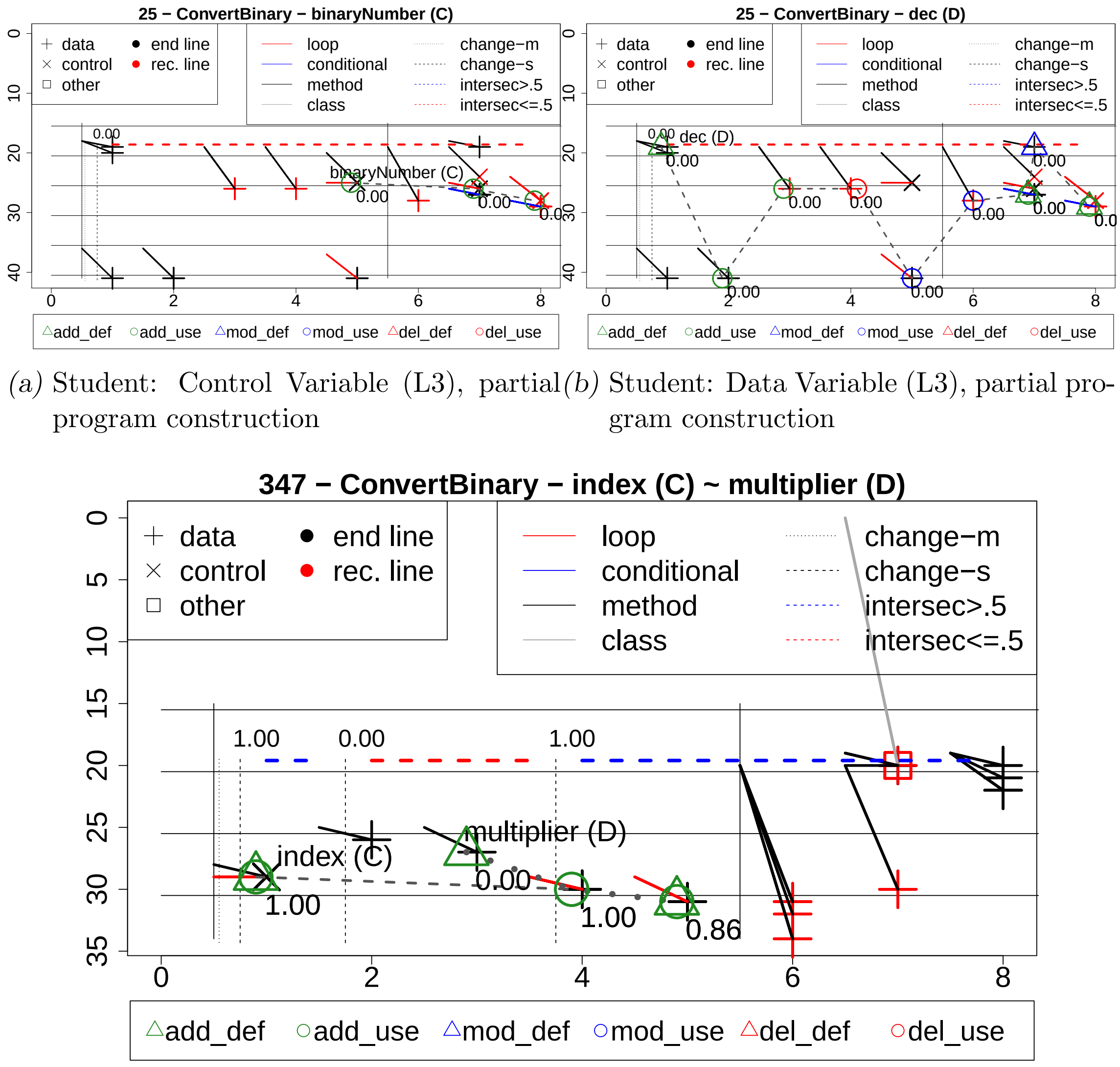
Incorporating information on the evolution of the semantic program structure during program construction provides valuable insight on how the sequence of *program increments* for a variable, typed by the four introduced dimensions, is related to the method cohesion, and thereby to the construction of semantic program structures. This information can be utilized to assess how the program changes in each variable-specific *construction flowline* affect the construction of semantic program structures. Additional semantic features are incorporated with the help of cohesion metrics on (pair-wise) variable-level, discussed below.

So far, sequences of changes to single variables are considered, but single variables seldom characterize an algorithm – programs much rather contain multiple variables and are characterized by their interactions. To add a last layer `L4` to the *construction flowlines*, pairwise views of variables are considered based on the types of involved variables. Depending on the types of the `L0` **changed program statements** that include the variables, two types of variables are differentiated: **control variables** (C) with at least one program statement of type **control**, and **data variables** (D) only with program statements of type **data**.

Based on these two types of variables, three types of pairwise views are possible:

- $C+D$: A **control variable** that exerts control influence on a **data variable**. An example of the expert's *end program*, Figure 4.6: the loop variable `index` governs the control flow and exerts control influence on both **data variables** `sum` and `multiplier`. This view is an archetypical part of program construction, considering the semantic use of control constructs in concert with data.

- $C+C$: Two **control variables** that are related in the program's control flow. This can happen either by the use of both **control variables** in the same control construct or by the use of nested control structures and can easily be differentiated based on the **control contexts**. An example of the undergraduate's *intermediate program*, Figure 4.9 (`b`): both **control variables** `i` and



(a) Student: Control Variable (L3), partial program construction

(b) Student: Data Variable (L3), partial program construction

(c) Professional: Interaction between Control and Data Variable (L4), partial program construction

Fig. 4.8: Showcase of partial program construction sequences with differently typed variables. (a-b) show the construction of two variables in the program construction sequence of an undergraduate student (Figure 4.9 b). (a) shows a control variable with exerted control influence of loop and conditional, (b) shows a data variable that is, at the last shown *intermediate program*, under the inner conditional control context. (c) shows the interaction between a control and data variable for the program construction sequence of the professional programmer (Figure 4.6).

binaryNumber are used in a loop and a conditional control structure, with the latter being nested inside the loop.

- $D+D$: Two **data variables** that are related in their data dependencies, either by common input or by common computational result data. An example of the expert's *end program*, Figure 4.6: the **data variables** sum and multiplier are part of the same loop and contribute to the same computation.

The pairwise views $C+D$ and $C+C$ can be identified based on syntactic features, mostly based on the control context. Figure 4.8 depicts two examples of the view $C+D$. In (a--b), two separate views of the undergraduate's program construction are depicted on layer L3. Take note that only a partial program construction sequence is shown for brevity and readability. Of particular interest are: i) the sequence of the respective program changes for each variable, notable the definition of the data variable prior to the construction of the control constructs (*loop* at change 5, *if* at change 7), and ii) the interaction at the next-to-last change 7, which shows the *add/use* of binaryNumber in a control statement (the conditional if statement, with loop control context) as well as the *add/def+add/use* of dec in a data statement with conditional control context.

Incorporating both variable-specific *construction flowlines* at the same time constitutes layer L4, depicted as an example of view $C+D$ in Figure 4.8 (c). This figure showcases the interaction between the loop control variable index and one of the data variables, multiplier, that is defined and used inside the loop control structure. Again, of interest are: i) the sequence of the respective program changes, this time observed as a construction of the control structure before preparing data variables, and finally, the use of data (and control) variables in data program statements with loop control context, and ii) the interaction at the points of contact, specifically through the control statement that is constructed at change 1 and that exerts control on data statements in the future, and through the data statement at change 4 which includes the use of both the control and data variable.

Altogether, pair-wise views provide many pieces to the puzzle of assessing high-level and overarching features of program construction sequences, which could be turned into signature hallmarks of program construction patterns. However, in the current form, the syntactic features cannot indicate which variables are in relation to each other on the level of semantic program structures. As an example, the pairwise view $D+D$ cannot be identified solely based on syntactic features. The next step is the incorporation of slice based cohesion metrics, which makes it possible to identify semantic blocks in program construction and identify missing relationships. The incorporation of slice based cohesion metrics also sharpens the identifying lens for the other pairwise views, enabling the identification of semantic relations in the constructed program structures.

4.2.4.3 Exploratory Semantic Analysis of Text-based Construction Sequences

For the analysis of semantic features of text-based program construction sequences, three approaches are considered and described in this section. These are, in order and applied on the exemplary program construction sequences introduced above: i) investigating slice profiles, slice-based cohesion metrics on variable-level, and pairwise cohesion metrics to uncover (semantic) relations between variables in exemplary *intermediate programs*, ii) investigating the evolution of slice-based cohesion metrics (method intersection, cohesion metrics on variable level) for all *program increments* of the exemplary program constructions, and iii) investigating the overall cohesiveness for sequences of *program increments.*

For the first analysis approach, two *intermediate programs* of the undergraduate programmer (one *intermediate program* and the *end program*), and two *intermediate programs* of the professional programmer (two *end programs* of different methods) are considered to investigate the analysis capabilities of slice profiles and the slice-based cohesion metric *Coverage* for each variable, and for each pair of variables in the respective methods.

Figure 4.9 contains program code, slice profiles, and method cohesion matrices for the two *intermediate programs* of the undergraduate programmer. Figure 4.10 contains program code, slice profiles, and method cohesion matrices for the two *end programs* of the professional programmer. Take note that, for the sake of space, variable names are abbreviated in the figures. These program construction sequences have already been described by Kesselbacher and Bollin [KB21].

Let me first consider the undergraduate's program construction. The undergraduate programmer starts implementing the method by constructing the loop for the computation of the decimal value. Notably, they also prepare the variable and output for the hexadecimal value, even before constructing the loop (Figure 4.9 `a--b`). The union slices for each variable show that the main computation is overlapping for all involved variables (`binaryNumber`, `dec`, `i`), save for `hex`. *Coverage* for all involved variables is reasonably high ($0.67 - 0.78$), and the pairwise metrics show a clear separation of all involved variables and `hex` (pairwise *Coverage* is 0).

The undergraduate's efforts to finish the method culminate in the introduction of two variables that sum up the two halves of the byte, which are joined to the output String `hex` by invocating `Integer.toHexString((int) ...)` (shown in pseudocode to preserve space, Figure 4.9 `c--d`). In the slice profile of the method, the overall notion is unchanged. There are union slices with high *Coverage* (0.75 for `binaryNumber`, 0.81 for `i`), which form the base computation and its control flow, and union slices with moderate to high *Coverage* (0.44 for `dec`, 0.41 for `hex`, 0.75 for `hD1` and `hD2`) representing the decimal / hexadecimal computation. Notably, the computation of the hexadecimal value now spans nearly the whole program, as indicated by the high *Coverage* of `hD1` / `hD2`. The union slice for `dec` still includes

bN	***dec***	***hex***	***i***
F			
	F		|
		F	
| | |	| | |		F | |
B	|		|
|	|		B
|	B		|
		B	
.67	.78	.22	.78

(a) Student: Slice profile

```
void convertBinaryArray(int[] binaryNumber){
 double dec = 0;
 String hex = "";
 for(int i = 0; i < binaryNumber.length; i++){
  if(binaryNumber[i] == 1){
   dec = dec + Math.pow(2,i); } }
 System.out.println(dec);
 System.out.println(hex); }
```

(b) Student: Method source code

bN	***dec***	***hex***	***hD 1/2***	***i***
F				
	F			
		F		
		|	F / |	|
		|	| / F	
| | |	| | |	| | |	| | |	F | |
B	|	|	|	|
|	|			|
|		|	|	|
|		|	|	|
|		|	|	|
|		|	|	B
|		F	B	|
|	B			|
|		B	|	|
.75	.44	.41	.75	.81

(c) Student: Slice profile

```
void convertBinaryArray(int[] binaryNumber){
 double dec = 0;
 String hex = "";
 double hD1 = 0;
 double hD2 = 0;
 for(int i = 0; i < binaryNumber.length; i++){
  if(binaryNumber[i] == 1){
   dec = dec + Math.pow(2, i);
   if(i <= 3)
    hD2 = hD2 + Math.pow(2,i);
   if(i > 3)
    hD1 = hD1 + Math.pow(2,i-4); } }
 hex = hexString(hD1) + hexString(hD2);
 System.out.println(dec);
 System.out.println(hex); }
```

(d) Student: Method source code

PairCoverage (REF+DEF)

	bN	***dec***	***hex***	***i***
bN	1	.67	0	1
dec	1	1	0	1
hex	0	0	1	0
i	1	1	0	1

(e) Method cohesion matrix for PairCoverage (a–b)

PairCoverage (REF+DEF)

	bN	***dec***	***hex***	***hD 1/2***	***i***
bN	1	.67	.67	.67	1
dec	1	1	0	0	.56
hex	.50	0	.50	.50	.44
hD1	1	0	.67	1	.89
hD2	1	0	.67	1	.89
i	1	.67	.67	1/.67	1

(f) Method cohesion matrix for PairCoverage (c–d)

Fig. 4.9: Slice profiles with statements included in the union slice (|), forward (`F`) and backward (`B`) slicing criteria and overall coverage values per variable (bottom), the corresponding method source codes, and method cohesion matrices for two *intermediate programs* of the method `convertBinaryArray` implemented by an undergraduate student. Darker green means higher cohesion.

<table>
<tr><th>bN</th><th>i</th><th>m</th><th>sum</th><th></th><th></th><th></th></tr>
<tr><td>F</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>|</td><td>|</td><td></td><td>F</td><td></td><td></td><td></td></tr>
<tr><td>|</td><td>|</td><td>F</td><td>|</td><td></td><td></td><td></td></tr>
<tr><td>| | |</td><td>F | |</td><td>| | |</td><td>| | |</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>B</td><td>B</td><td>|</td><td>|</td><td></td><td></td><td></td></tr>
<tr><td>|</td><td>|</td><td>B</td><td>|</td><td></td><td></td><td></td></tr>
<tr><td>|</td><td>|</td><td>|</td><td>|</td><td></td><td></td><td></td></tr>
<tr><td>|</td><td>|</td><td>|</td><td>B</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>inp</td><td>res</td><td>strs</td></tr>
<tr><td></td><td></td><td></td><td></td><td>F</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td>F</td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>|</td><td>|</td><td>|</td></tr>
<tr><td></td><td></td><td></td><td></td><td>|</td><td>|</td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>| |</td><td>| |</td><td>| |</td></tr>
<tr><td></td><td></td><td></td><td></td><td>|</td><td>|</td><td>F</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>B</td><td>|</td><td>B</td></tr>
<tr><td></td><td></td><td></td><td></td><td>|</td><td>B</td><td>|</td></tr>
<tr><td>1</td><td>1</td><td>.89</td><td>1</td><td>.87</td><td>1</td><td>.75</td></tr>
</table>

(a) Professional: Slice profile

```
void convertBinaryArray(
  int[] binaryNumber){
 int sum = 0;
 int multiplier = 1;
 for(int index = 0;
   index < binaryNumber.length; i++){
  sum += multiplier * binaryNumber[i];
  multiplier *= 2; }
 System.out.println(sum);
 System.out.println(intToStr(sum/16)
  +intToStr(sum%16)); }

 String convertIntToStr(int input) {
 String res = "fail";
 if(input > 0 && input < 10){
  res = "" + input;
 } else if(input >9 && input < 16) {
  String[] strs =
   new String[]{"A",...,"F"};
  res = strs[input - 10]; }
 return res; }
```

(b) Professional: Method source code

PairCoverage (REF+DEF)

	bN	*i*	*m*	*sum*
binaryNumber	1	1	1	1
index	1	1	1	1
multiplier	1	1	1	.75
sum	1	1	1	1

(c) Method cohesion matrix for PairCoverage (convertBinaryArray)

PairCoverage (REF)

	input	*res*	*strs*
input	1	1	1
res	1	1	1
strs	.75	1	1

PairCoverage (DEF)

	input	*res*	*strs*
input	0	.67	1
res	0	1	1
strs	0	.34	1

(d) Method cohesion matrix for PairCoverage (convertIntToStr)

Fig. 4.10: Slice profiles with statements included in the union slice (|), forward (`F`) and backward (`B`) slicing criteria and overall coverage values per variable (bottom), the corresponding method source codes, and method cohesion matrices for the two methods `convertBinaryArray` and `convertIntToStr` implemented by a professional programmer. Darker green means higher cohesion.

the same statements, but its *Coverage* heavily decreased, showing the effects of implementing multiple trains of thought in a single method. The pairwise metrics, including references (`REF`) and definitions (`DEF`), support this interpretation of the slice profile: both `binaryNumber` and `i` cover some of the references and definitions of the computed variables but miss the initial definitions (pairwise *Coverage* of 0.67). There is no cohesion between the variables for decimal and hexadecimal values (0). The empty slice intersection of union slices for `hex` results in moderate pairwise *Coverage* values ($0.44 - 0.50$ compared to the large union slice).

To summarize, during the program construction, the method evolves from implementing a single train of thought (computing the `dec` variable) with a stray variable (`hex`) to implementing two concurrent trains of thought. Concurrency can be identified by the control variables (`binaryNumber`, `i`) covering large parts of the method (> 0.5) and the different computational parts not covering each other. The latter is apparent in the pairwise *Coverage* metrics, which show 0 *Coverage* for references and definitions between the two different computational parts.

Next, let me consider the professional's program construction. The professional programmer implements the method `convertBinaryArray` by constructing a highly cohesive computation of the decimal value (Figure 4.10 `a--b`, top part). The slice profile shows that the *Coverage* for each variable is high ($0.89 - 1.00$), as the union slices cover most or all of the method. The pairwise metrics reflect this, but also uncover the single non-covering statement (definition of `sum`) in the union slice of `multiplier` for a pairwise *Coverage* value of 0.75.

The method `intToStr` (Figure 4.10 `a--b`, bottom part) computes the hexadecimal digit conditional on the input integer, and circumvents the construction of a method with concurrent trains of thought. This method is also highly cohesive, with variable *Coverage* values ranging from $0.75 - 1.00$. Notably, the return value is set in the mutually exclusive if-branches and returned at method end, which results in a single slice for the return variable `res`. The pairwise metrics, separately including references (`REF`) and definitions (`DEF`), for `intToStr` show a lack of *Coverage* of the definitions of `res` that is not easily visible in the measures on variable level: 0.67 for `input`, 0.34 for `strs`. Interpreting these metrics, there are parts or branches of the method that are not covered in the union slices of `input` and `strs`. The low *Coverage* (0) of definitions of `input` comes from the fact that this is an input parameter and is only referenced in the method.

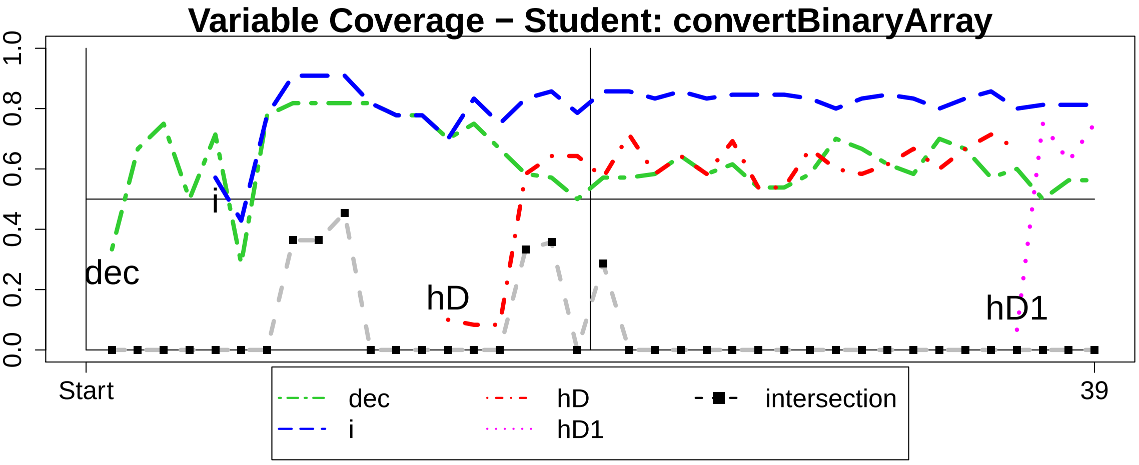
To summarize, the professional programmer resolves the conflict of potentially concurrent trains of thought by introducing a second method that covers the conversion of an integer to a hexadecimal digit as a String. Both methods are highly cohesive, measured by the cohesion for each variable as well as the pair-wise cohesion, and therefore implement a single train of thought. In a post-interview of the program construction, conducted with all professional programmers that have been recorded, the programmer stated that they '*[...] returned back to that binary-to-array [method] to improve that code*', referring to refactoring the method `convertBinaryArray` after introducing the method `intToStr`. In fact, the end result of this approach to program construction can be measured as a more cohesive method and could be indicative of the proper use of cohesion measures to identify worthwhile *practices* in program construction.

Altogether, the slice-based cohesion measures on variable-level make it possible to quantify the semantic relations between variables in a method and identify semantic structures of a program that are related, utilizing individual and pair-wise cohesion metrics.

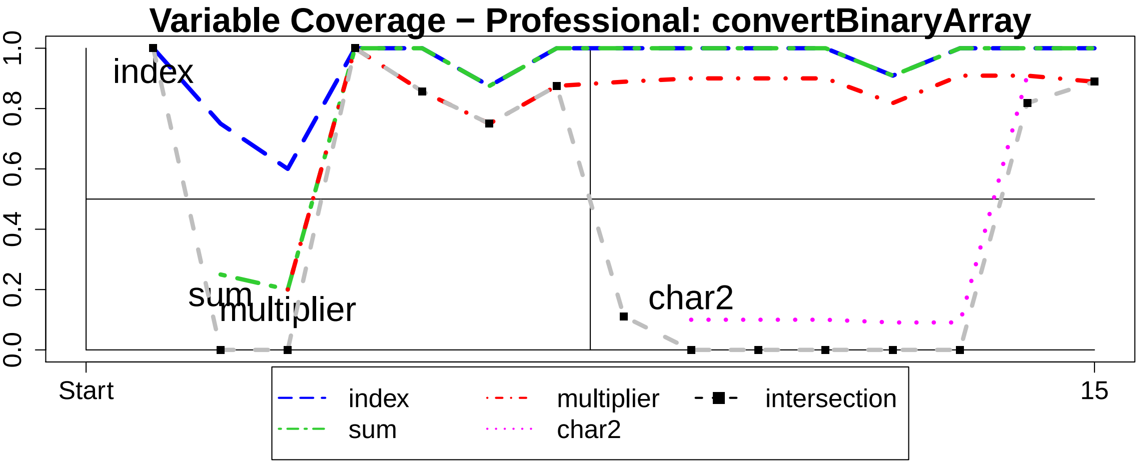
For the second analysis approach, the evolution of method intersections and the evolution of *Coverage* on variable-level is investigated for the sequence of all *intermediate programs* in the two exemplary program construction sequences. This investigation has been described previously by Kesselbacher and Bollin [KB21].

The evolution is depicted in Figure 4.11, showing the *Coverage* for selected variables with different line shapes and colours and showing the relative method slice intersection of the whole method by line and shape. Each point on the x-axis denotes an *intermediate program*. The y-axis denotes the values of *Coverage* and relative intersection, ranging from $0 - 1$. The *intermediate programs* are divided based on their relative intersection: those with a relative intersection of > 0.5 are *cohesive* method versions, while those with a relative intersection of $<= 0.5$ are *non-cohesive* method versions. The rationale for this split is that, above this threshold, union slices are much more likely to share semantically important sets of method nodes. This rationale is grounded in the exemplary slice profiles presented above, where methods are either highly cohesive with a dominant computational train of thought (for example, *Coverage* close to 1) or feature concurrent computational trains of thought and are less cohesive (*Coverage* $<= 0.5$).

Let me first consider the undergraduate's program construction, shown in Figure 4.11 (`a`). The separated method concerns, previously identified with cohesion metrics and slice profiles, can also be observed in the evolution of *Coverage* on variable-level. With the implementation of the second computation part (the addition of the variable *hD*), the *Coverage* of the control variable i remains high while the *Coverage* of the data variables *dec* and *hD* is only moderately high. Towards the end, the undergraduate replaces *hD* with *hD1/hD2*, which further decreases the *Coverage* of *dec*. Both data variables have a *Coverage* greater than 0.5, but the relative method slice intersection is low during the whole method construction and even 0 for most program versions. This means that, for most program versions, there is no method statement that is included in all *union slices*. Even when removing the unnecessary first declaration of `hex`, which causes the empty intersection, the relative method slice intersection is only 0.27.



(a) Evolution of cohesion for method intersection and selected variables (Student)



(b) Evolution of cohesion for method intersection and selected variables (Professional)

Fig. 4.11: The evolution of *Coverage* for selected variables and the relative method slice intersection for all consecutive, compilable program versions, implemented by an undergraduate student (a) and a professional programmer (b). Each point on the x-axis represents a compilable program version during program construction, starting with the first compilable program change (e.g., variable declaration) and ending with the respective end program version. The final number on the x-axis represents the number of consecutive, compilable program versions [KB21].

Next, let me consider the professional's program construction, shown in Figure 4.11 (`b`). The evolution of *Coverage* on variable-level (Figure 4.11 `b`) shows that the professional programmer eventually achieves a high *Coverage* for all variables, in contrast to the undergraduate's program construction sequence. During program construction, the relative method slice intersection decreases whenever a new variable is introduced (*sum / mul / char2*). Subsequently, they are integrated or refactored (in the case of *char2*, which was a temporary variable to hold half of the hexadecimal result), which results in a high coverage for all variables as well as a high relative method slice intersection. The relative method slice intersection of the end version is $\frac{8}{9} = 0.\dot{8}$ – all method nodes other than the definition of *sum* in line `2` are included in every union slice. Note that the loop corresponds to three nodes in the method PDG.

Summarizing this analysis approach, the combined focus on relative method slice intersections and variable-level *Coverage* measures provides a view on the evolution of cohesion during program construction. The relative slice intersection shows the fraction of statements common to all union slices, and therefore provides a strong measure on focused trains of thought. The individual variable-level cohesion measures show, with proper variable selection, the evolution of *Coverage* conditional on changes in different variables. This provides additional insight, combined with the identification of semantic relations with pair-wise cohesion metrics introduced above.

To conclude the exploration of semantic features, the third analysis approach incorporates the cohesion of sequences of *program increments* to assess the process of program construction. This approach does not focus on specific variables but rather on the overall cohesion and continues with the introduced notion of *cohesive intermediate programs* having a relative method slice intersection > 0.5 and *non-cohesive intermediate programs* having a relative method slice intersection $<= 0.5$.

For the investigation of cohesion during the process of program construction, I compute three measures for each method of the recorded program construction sequences of student programmers and professional programmers:

1. Fraction of *cohesive intermediate programs*, compared to all *intermediate programs*

2. Relative mean length of *cohesive intermediate program* sequences

3. Relative mean length of *non-cohesive intermediate program* sequences

All these measures are naturally normalized between $0 - 1$. A value of 0.5 indicates that half of the *intermediate programs* have a relative method slice intersection > 0.5, with higher values indicating more *cohesive*, and lower values indicating

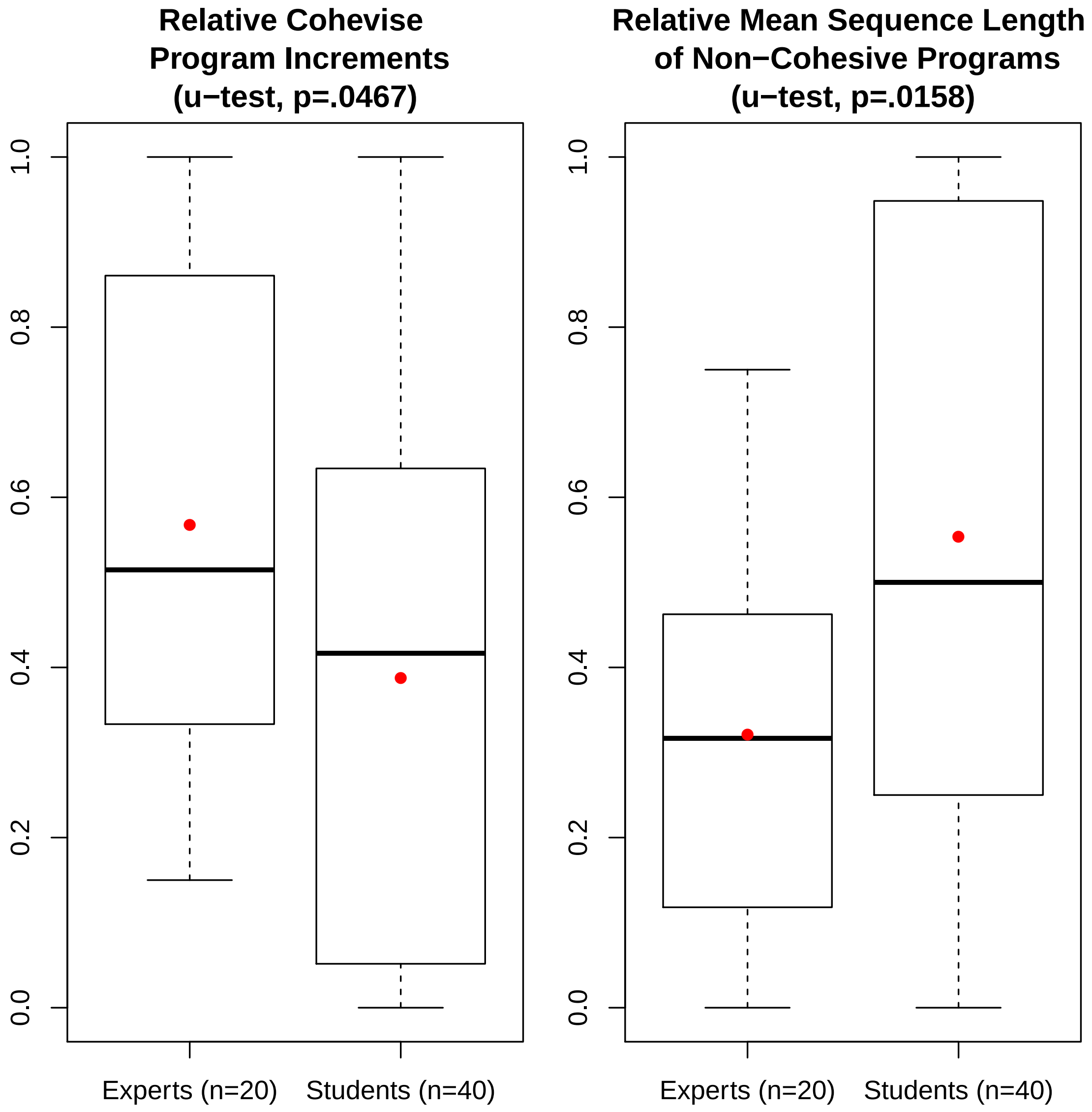


Fig. 4.12: Comparison of evolution of cohesion in program construction sequences. The left boxplot shows the relative cohesive *program increments* (with a method slice intersection greater than 0.5), with a noticeable but not significant difference between professional and student programmers. The right boxplot shows the relative mean length of sequential non-cohesive *program increments* (with a method slice intersection smaller or equal to 0.5), with a significantly lower mean sequence length of professional programmers.

more *non-cohesive intermediate programs.* The sequence measures capture whether programmers produce *intermediate programs* of comparable cohesiveness (either *cohesive*, or *non-cohesive*) in succession. Higher mean lengths indicate the fraction of *intermediate programs* of the specific cohesiveness, averaging over the whole respective program construction sequence.

The rationale behind investigating the sequences of *(non)-cohesive intermediate programs* is to uncover whether there is a difference in the program construction of student and professional programmers that is measurable with cohesion metrics, in the albeit small sample of experimental subjects ($n = 40$ student-produced methods, $n = 20$ professional-produced methods, including the warm-up method *MoveElementsLeft*, not reported above but based on a *Reversing* problem by Lister [Lis11b, p. 15]).

I employ Shapiro-Wilk tests with $p < .05$ to test for normality [Tho11]. The null hypothesis could not be rejected for the measures of professionals in (1) and (3). Therefore, I used Mann-Whitney-U tests with $p < .05$ [Neu11] to test for differences. There are significant differences in two measures: (1) fraction of cohesive method versions (students: $.39 \pm .10$, professionals: $.57 \pm .13$; $p = .0467$) and (3) the relative mean length of non-cohesive method version sequences (students: $.55 \pm .11$, professionals: $.32 \pm .11$; $p = .0158$). Figure 4.12 provides box plots for the significant differences.

I interpret these differences in the following way. The professional programmers create cohesive program versions **more often** over the course of program construction (1), and converge to cohesive program versions **faster** (3) compared to the students. Unconsciously or not, the professional programmers seem to strive to implement cohesive program code. These findings highlight the importance of the use of (slice-based) cohesion metrics as a unique viewpoint on the process of program construction, with great potential for individual assessment and feedback powered in a learning analytics context.

4.2.5 Summary of Exploratory Study of Program Features

Before resuming with the definition of *program construction patterns* in the context of sequences of *intermediate programs* recorded with learning analytics approaches, I conclude this section with a summary of the exploratory study of syntactic and semantic program features. I focus on the exploratory findings of the analysis of text-based features.

I introduced three cornerstones in the exploratory analysis of syntactic features. First is the classification of program changes along four dimensions simultaneously: the **fundamental program change type** of each program change, the type of **variable usage**, the type of the **changed program statement**, and the **control context** of the changes. Second is the notion of highlighting variable-specific views of program construction on the one hand and pairwise views of variables, with the types of **control** and **data** variables, on the other hand. Third is the incorporation of cohesion information on method-level to assess program construction on another dimension.

Put together, these three cornerstones yield the visualization of *construction flowlines*, presenting a detailed view of the program construction sequence. These visualizations form the basis of assessing and classifying different variable-specific programming approaches to form a catalog of *program construction patterns.*

I also introduced three cornerstones in the exploratory analysis of semantic features. First is the use of slice profiles to measure cohesion on the level of individual variables and for pairs of variables, with the goal of identifying semantic relations between the variables present in the program construction. Second is the analysis of the evolution of cohesion, carried out with both method-level cohesion (evolution of relative method slice intersections) and variable-level cohesion (evolution of variable-level *Coverage*) to investigate the evolution of the semantic relations identified above. Third is the incorporation of a measure of sequential cohesiveness to assess the evolution of cohesion from a strategical point of view.

Together, these analysis approaches of semantic features provide the means to assess the program construction from a semantic point of view, which could be a valuable opportunity for improved individual assessment and feedback during education and training. Moreover, expanding on the information accessible in *construction flowlines*, the semantic structure of each *intermediate* program can be assessed, which is important when analyzing the effects of each program change.

With regard to the related work on learning analytics in programming, there is, to the best of my knowledge, no other work that incorporates slice-based cohesion metrics in their investigation of (text-based) programming. Furthermore, the cornerstones of syntactic features make it possible to interpret program changes in a semantic frame, which is more detailed (but also less applicable to large amounts of data) compared to the related work. This is all in light of the notion of *program construction pattern* I aim towards, focusing on *regular forms* in the sequences of program changes and interpreting them on a semantic level.

4.3 Definition of Program Construction Patterns

In this section, the developed notions and findings of the previous two sections are synthesized to arrive at a general definition of *program construction patterns.* Moreover, a specific definition focusing on the construction of variable use is introduced. These definitions of *program construction patterns* entail an answer to the second research question of the thesis:

RQ2. How can program construction patterns be defined?

4.3.1 General Definition of Program Construction Pattern

From the exploratory analysis of program features, syntactic and semantic features and their inter-relations have been observed that qualify the frame of program changes into specific *regular forms* of program construction. Comparable to design patterns, the goal in defining and later identifying construction patterns is to distill planned, widely applicable solution approaches and quantify them according to the introduced features of program construction sequences. But, unlike design patterns, my introduced notion of *program construction patterns* work on a different level of granularity: not on the abstraction level of architectural design, but on the abstraction level of algorithmic design in the context of single methods.

Condensing my findings and observations from the exploratory analysis of program features, I arrive at the following definition of *program construction patterns*:

Definition 4.8 (Program construction pattern)**.** A typed, ordered, non-continuous sequence of program increments that are semantically related.

Each part of this definition has a specific meaning for the general definition of patterns of program construction, refined as follows:

- `Typed`: The *program increments* are typed with regard to the contained fundamental program change actions. There are different possible ways to assign types to the program increments. In this thesis, I consider typing the program increments based on automatically extractable syntactic and semantic features of the program increments and intermediate programs. A concrete way of typing is introduced in the following section, along with a specific definition of *variable construction patterns.*

- `Ordered`: The order of the *program increments* in the sequence qualifies and discriminates patterns with equally typed *program increments.*

- `Non-continuous`: The considered *program increments* need not be in a continuous sequence. Patterns can span non-continuous sequences.

- `Sequence`: Patterns are made of a sequence of multiple, more than one, *program increments.*

- `Program increment`: A sequence of fundamental program change actions that produces a compilable program version. Fundamental program change actions are changes that add, modify or delete program statements.

- `Semantically related`: To be considered part of a pattern, program increments have to be in the same union slice, which constitutes semantically related parts of the program.

At this point, a general answer to the second research question can be given. *Program construction patterns* can be defined in terms of **syntactic** features (typing and order of sequential program changes in the program construction) and **semantic** features (slice-based cohesion metrics to identify relations between semantic structures in sequential program versions) of program construction sequences. These features can be **extracted** from *intermediate programs*, **recorded** from ordinary program construction sequences of programmers solving a given programming task. An investigation of general *program construction patterns* is possible in any learning analytics setting in programming, given the recording granularity of at least *compilation* level. Note that this definition only includes programmer actions that result in code changes, which consequently generate *program increments*. Notably, the definition neither includes interactions with the programming environment nor features of the programming process (executions, tests, debugging, and other process states) and solely focuses on algorithmic design.

Here I want to stress that the introduced definition of *program construction patterns* is universal for any program construction sequences. It needs to be instantiated, specifically in the typing of the sequential program changes, to provide specific patterns of program construction. A specific definition is given in the next section.

4.3.2 Specific Definition of Variable Construction Pattern

In this thesis, I introduce one specific instantiation of the general definition of *program construction patterns* given above. Extending the notion of variable-specific *construction flowlines*, considering a variable-specific *view* of program construction, leads to the definition of variable-specific *program construction patterns*, which are called *variable construction patterns*.

Definition 4.9 (Variable construction pattern)**.** A typed, ordered, non-continuous sequence of program increments with changes to a variable that are semantically related.

To fully arrive at a workable definition of *variable construction patterns*, all parts of the definition need to the put into concrete, measurable terms, following the general definition on the one hand and the variable-specific context on the other hand. The parts of the definition that need refinement are described as follows:

- `Typed`: The fundamental program change actions are typed by the four syntactic dimensions: the **fundamental program change type** (addition, modification, deletion), the type of **variable usage** (definition, use), the type of the **changed program statement** (control statement, data statement, other), and the **control context** (loop, conditional, method, class).

- `Program increments with changes to a variable`: For a *variable construction pattern*, only those *program increments* are considered that affect the usage of the specific variable, regardless of them affecting the definition or the use of the variable. This can be detected by the type of the change actions. By virtue of the `non-continuous sequence` of the *program increments*, no other change in the definition is necessary.

- `Semantically related`: Though unchanged from the general definition, the identification of semantic structures and of semantic relations during program construction are of great interest when considering changes to a specific variable. By measuring method-level and variable-level cohesion, the intricate strategical decisions during programming that lead to one (or more) implemented trains of thought can be observed.

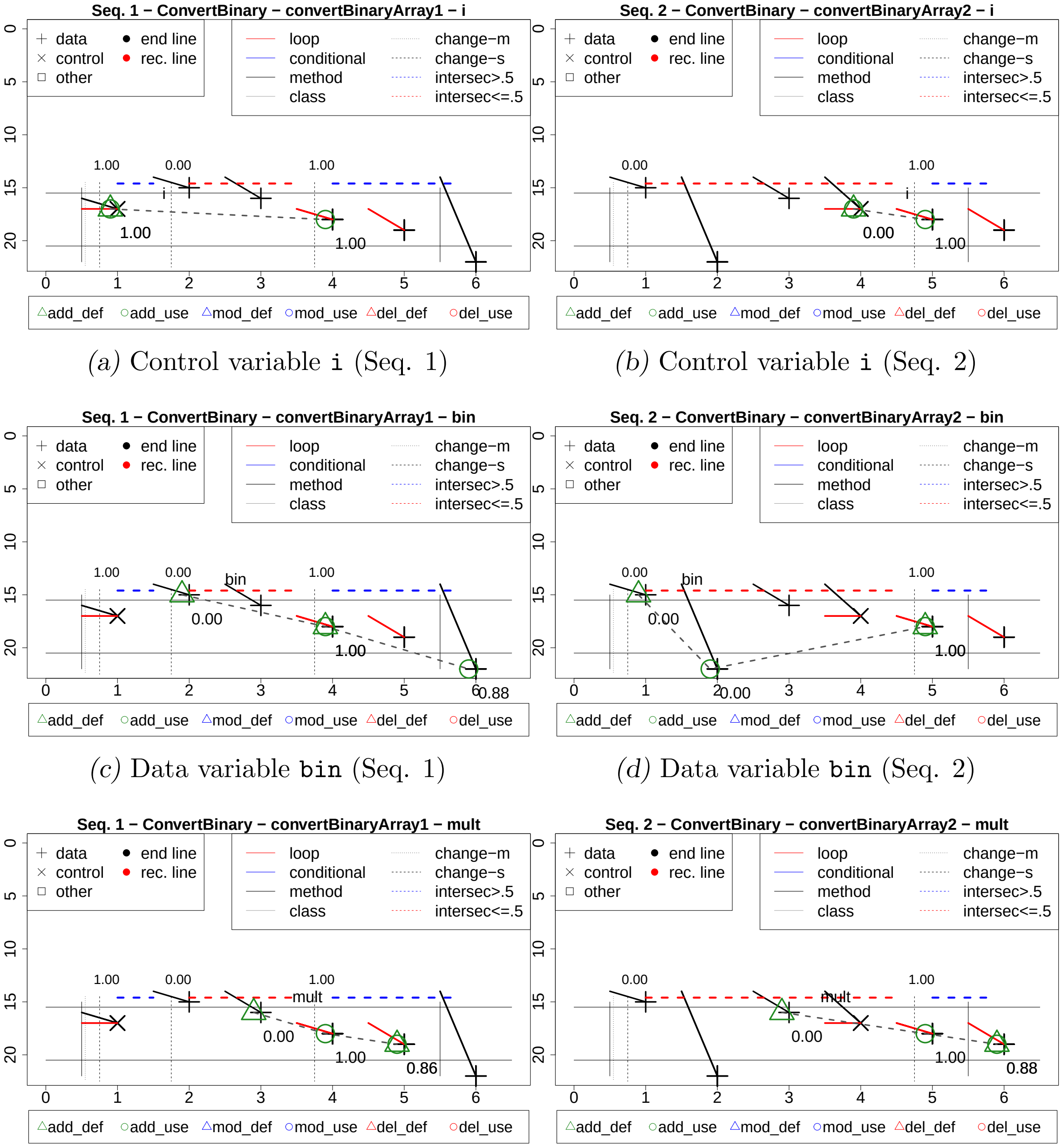
To round out the specific definition of *variable construction patterns*, I now showcase a part of a program construction sequence, distilled from the professional programmer's sequence shown before, to introduce exemplary *variable construction patterns* of data and control variables. A formal study is described in Section 4.4.

In Table 4.7, the exemplary *end program* for this showcase is represented. The program statements are numbered to differentiate two differently ordered program construction sequences: one actually constructed by the professional programmer in the context of the whole example problem (Figure 4.13 (`a/c/e`) right column), and one devised as a variation by myself (Figure 4.13 (`b/d/f`) left column). Note that the second control variable `binNumber` is used in the same program changes as the control variable `i` and is not shown for the sake of space.

Tab. 4.7: Exemplary *end program* version to showcase *variable construction patterns.*

```
  | public void convertBinaryArray2(int[] binNumber) {
1 |  int bin = 0;
2 |  int mult = 1;
3 |  for(int i = 0; i < binNumber.length; i++) {
4 |   bin += binNumber[i] * mult;
5 |   mult *= 2; }
6 |  System.out.println(bin); }
```

First, let me examine the left program construction sequence, originally implemented by the professional programmer. In this sequence, the control structure necessary to process the data structure (the input array) is prepared, observed with definition and use of the control variables `i` and `binNumber` to construct the `for` loop (line `3`). Next, the two data variables `bin` and `mult` are defined and thereby prepared for the following computation (lines `1--2`). Next follow the two construction



(a) Control variable `i` (Seq. 1)

(b) Control variable `i` (Seq. 2)

(c) Data variable `bin` (Seq. 1)

(d) Data variable `bin` (Seq. 2)

(e) Data variable `mult` (Seq. 1)

(f) Data variable `mult` (Seq. 2)

Fig. 4.13: Two exemplary program construction sequences and variable-specific *construction flowlines*, constructing the program code in Table 4.7. The left sequence (`a/c/e`, implemented by a professional) represents a *program construction sequence* with the following sequence of changed program statements: `3-1-2-4-5-6`. The right sequence (`b/d/f`) represents a *program construction sequence* with the following sequence of changed program statements: `1-6-2-3-4-5`.

steps to update the values of the data variables: a self-defining program statement for `bin` (line `4`), in which all variables are used, and another self-defining program statement for `mult` (line `5`), to properly update the multiplier for the binary computation. Lastly, the printout statement for the computation result, the data variable `bin`, is constructed after the loop (line `6`).

Next, let me examine the right program construction sequence. In this sequence, the data variable `bin` is defined (line `1`), and then its printout is prepared (line `6`). Next, the second data variable `mult` is defined (line `2`). Altogether, both data variables are prepared before constructing the control structure of the method, defining and using the control variables `i` and `binNumber` to construct the `for` loop (line `3`). The last two construction steps include a self-defining program statement for `bin` (line `4`), in which all variables are used, and another self-defining program statement for `mult` (line `5`) to properly update the multiplier for the binary computation.

Looking at the variable-specific *construction flowlines*, the typed and ordered sequences are the same for most variables, with only the printout use of `bin` being constructed in different orders. However, a difference can be observed in the methods' cohesion during program construction. The professional's program construction sequence reaches a cohesive *intermediate program* faster and only deviates from a cohesive program state for a total of 2 *program increments*. The second devised program construction sequence is in a non-cohesive program state for the majority of program construction and only arrives at a cohesive *intermediate program* at the next-to-last program change.

While this is only a very small example program, such an analysis could be very important in individual assessment and feedback of program construction sequences. According to Winslow's summary of programming approaches of novice and expert programmers [Win96], there are pronounced differences in how the implementation of problems is approached. Expert programmers '*tend to approach a program through its data structures or objects*', while novice programmers '*tend to approach programming through control structures, and use a line-by-line, bottom up approach to problem solution*' [Win96, p. 18].

Interpreted this way, the professional's program construction sequence follows an expert's approach as they first begin to prepare the means to process the data structure (the array) and then proceed to introduce the data elements necessary for the algorithmic solution.

On the other hand, the second program construction sequence could represent a naive way to implement the example program by first defining and preparing all necessary variables (`bin`, `mult`) in a bottom-up approach (even preparing the printout beforehand) before turning to the actual computation that is dependent on the data structure (the array).

Before turning to concrete implementations and actual *variable construction patterns*, I need to explain the context of validity for those patterns. Because of the fine-grained level of abstraction of *program construction patterns*, focusing on algorithmic design in the context of single methods, different algorithmic problems will necessarily draw from different families of *program construction patterns*.

In this thesis, I examine *variable construction patterns* specific to common control structures (conditional branching and iteration) and their respective data flows. The resulting *variable construction patterns* are dependent on the types of algorithms used in the recorded programs. I do not claim to catalog all available patterns for the recorded example problems. I instead aim to establish a basic set of *variable construction patterns*, with the aim of discerning those patterns used by expert programmers. The hypothesis is that such patterns *incarnate* a level of expert programming skills and are candidates for inclusion in education and training.

In this thesis, I assign *variable construction patterns* to variable-specific *construction flowlines* based on the observed syntactic and semantic features. In Section 4.4, I introduce separate categories of labels for control variables and data variables, and following a multi-label approach, each *variable construction* can be assigned more than one label with the set of labels determining the *variable construction pattern*. As a precursor to the next section, I describe the labeled *variable construction patterns* for the variables in the examined program construction sequence.

The control variable `i` is assigned the control variable labels: i) **loop** as it is defined and used in a loop control statement, and ii) **computation** as it is used in data statements albeit being a control variable. Altogether, the control variable `i` is assigned the *variable construction pattern* loop computation control variable.

The data variable `bin` is assigned the data variable labels: i) **loop-dependent** as it is used and defined in a loop control context, and ii) **self-defining** as it is defined and used in the same statement. Altogether, the data variable `bin` is assigned the *variable construction pattern* loop-dependent self-defining data variable. Take note that not all *program increments* with variable usage contribute to the assigned *variable construction pattern*: in this case, the printout statement does not contribute a discriminating program change for this *variable construction pattern*.

The data variable `mult` is assigned the data variable labels: i) **loop-dependent** as it is used and defined in a loop control context, ii) **self-defining** as it is defined and used in the same statement, and iii) **single-scope-use** as all observed uses are in a single control context. Altogether, the data variable `mult` is assigned the *variable construction pattern* loop-dependent self-defining single-scope-use data variable.

The order in the sequence of program changes is different in the two examples. However, they are assigned the same *variable construction pattern* as the program changes that contribute to the pattern labels are in the same relative sequence. Often, this is given from syntactic necessities. For example, in order to construct compilable programs in most text-based programming languages, variables need to be declared and initialized before their first use. This influences what kind of ordered sequences will be observed in program construction sequences.

For the first-order assignment of *variable construction pattern*, the sequence of program changes for each variable is not taken into account. However, for the analysis of pairs of variables, the order of program changes becomes an important factor – as showcased with the exemplary program construction sequences above, that differ greatly in the cohesion during program construction. The related research hypothesis is that there might be measurable differences in the order of program changes in program construction sequences, which could be related to different levels of programming skills.

To summarize, a specific answer to the second research question can be given. *Variable construction patterns* are defined in terms of **syntactic** features (typing, based on the dimensions of **fundamental program change type**, **variable usage type**, **changed program statement type**, and **control context**, and order of sequential program changes in the program construction) and **semantic** features (slice-based cohesion metrics to identify relations between semantic structures in sequential program versions) of program construction sequences, considering program changes that affect the usage of a specific variable.

Variable construction patterns are assigned to variable-specific *construction flowlines* with a multi-label approach, divided into a catalog of labels for control variables and data variables. The catalog of labels that is obtained from a mixed methods study is introduced in Section 4.4.

4.4 Specific Variable Construction Patterns in the Wild

In this section, I describe a mixed methods study, incorporating qualitative methods (grounded theory) and quantitative methods (descriptive statistics evaluation) on recorded program construction sequences to establish a catalog of *variable construction patterns*. Since these patterns are not devised from example programs but derived from recorded program construction patterns, they represent specific patterns that are found *in the wild*, i.e., used by student and professional programmers to implement a desired functionality in program code (with regard to the used example problems). These patterns do not cover all possible algorithmic solutions but form a basic catalog of *variable construction patterns*.

Of note is that this study focuses on text-based program construction sequences, and therefore the usage of variables in text-based programming. While variables are not widely used in openly published block-based programs (Aivaloglou et al. report that, in a sample of 250k projects, only a third of them used variables, with the mean number of 2.06 variables per project [AH16]), they are certainly supported in block-based programming environments, and can be used with similar success, shown in the exploratory study of block-based programming described in Section 4.2.2.1. However, I opt to establish the catalog of *variable construction* *patterns* based on text-based variable usage, following the hypothesis that variable usage in text-based programming will feature more plan-like, strategical program construction sequences, given the proportionally higher level of programming skills of text-based programmers.

Nevertheless, the findings can be applied to programming education in block-based programming environments as well, considering that no parts of the definition of *variable construction patterns* are specific to text-based programming, but rather based on syntactic and semantic features that can be extracted from program dependence graphs and from program representations such as abstract syntax trees.

This section consists of a description of the mixed methods study methodology (Section 4.4.1), followed by the sequentially applied qualitative methods (Section 4.4.2) and quantitative methods (Section 4.4.3). The section is concluded with a discussion of the threats to validity (Section 4.4.4), and a summary of the study (Section 4.4.5), including an answer to the third research question of this thesis:

RQ3. What program construction patterns are used by novice and expert programmers during program construction?

4.4.1 Description of Mixed Methods Study Methodology

The research question of the mixed methods study coincides with the third research question of this thesis, given above.

This research question builds on the definition of *program construction patterns* introduced in the previous section, considering the specialized definition of *variable construction patterns* (which are typed, ordered, non-continuous sequences of program increments with changes to a variable that are semantically related). The IDE-based learning analytics context introduced so far provides the means to uncover answers to this research question. An underlying hypothesis is that novice and expert programmers use different *variable construction patterns*, but this is not evaluated in this study. The goal is rather to construct a base catalog of different *variable construction patterns*, which form the basis of the following comparative study in Chapter 5.

The description of the mixed methods study methodology consists of the following parts: i) a description of the qualitative and quantitative methods employed, the sequence of these methods, and the respective inputs and outputs of each method, ii) the IDE-based learning analytics methodology used to record program construction sequences, and the specific example problems for which implementations have been recorded, and iii) the cohorts of programmers participating in the study.

I first describe the overall mixed methods study methodology that is visualized in Figure 4.14. There are two factors of the study design that need to be explained;

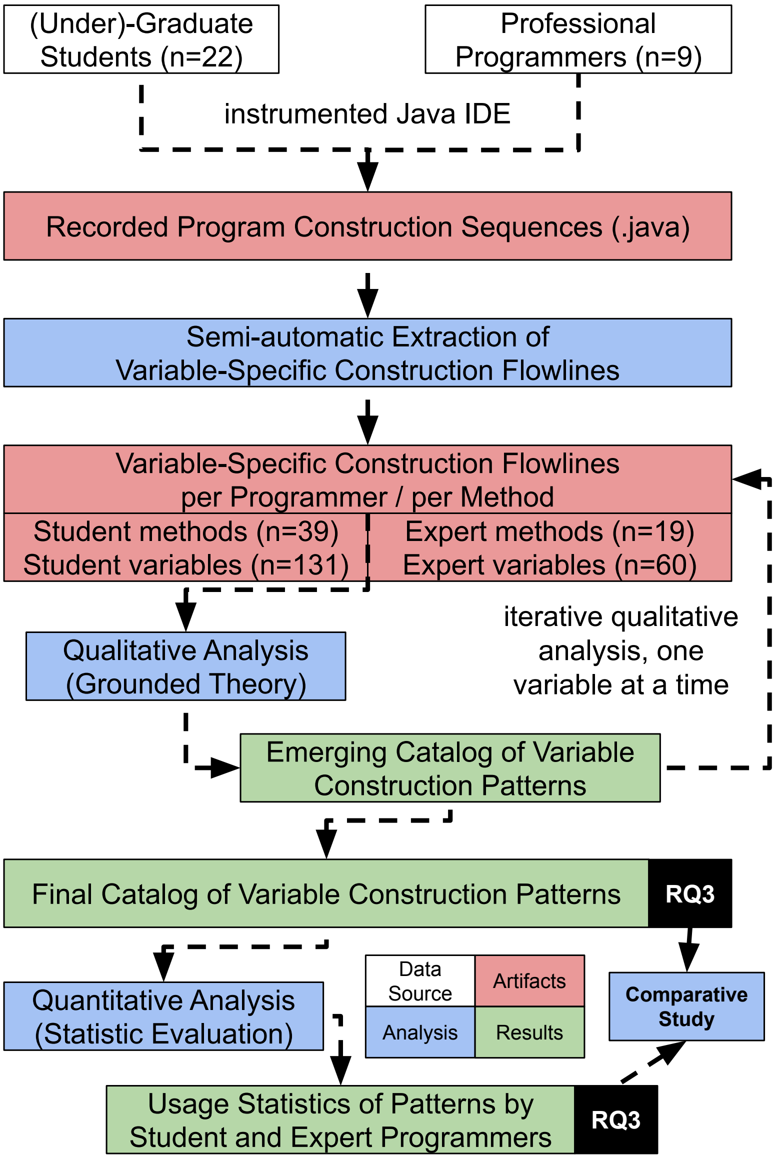

(Under)-Graduate Students (n=22)
Professional Programmers (n=9)
instrumented Java IDE
Recorded Program Construction Sequences (.java)
Semi-automatic Extraction of Variable-Specific Construction Flowlines
Variable-Specific Construction Flowlines per Programmer / per Method
Student methods (n=39)
Student variables (n=131)
Expert methods (n=19)
Expert variables (n=60)
Qualitative Analysis (Grounded Theory)
iterative qualitative analysis, one variable at a time
Emerging Catalog of Variable Construction Patterns
Final Catalog of Variable Construction Patterns
RQ3
Quantitative Analysis (Statistic Evaluation)
Data Source
Artifacts
Analysis
Results
Comparative Study
Usage Statistics of Patterns by Student and Expert Programmers
RQ3

Fig. 4.14: Visualization of the mixed methods study methodology.

also see Cohen et al. [CMM11, p. 21ff] for a section regarding mixed methods studies in education research.

The first factor is the relation of qualitative and quantitative methods and their types. The study design follows a sequential structure, represented in Figure 4.14, with the following methodological steps: i) a quantitative data preparation step comprising the semi-automatic extraction of variable-specific *construction flowlines* from the recorded program construction sequences, ii) a qualitative analysis step, following the methodology of Grounded Theory, to uncover **variable construction patterns** from the variable-specific *construction flowlines*, relative to the programmers and the example problems, and iii) a quantitative analysis step, incorporating descriptive statistics, to investigate relations between **variable construction patterns** and to interpret actual usage, inferred from program construction sequences.

The second factor is the respective inputs and outputs of each methodological step, and the employed data transformations: i) the input to the first quantitative data preparation step consists of recorded program construction sequences. These are sequentially evolving *intermediate programs* on *keystroke* granularity. The output consists of tabulated data that represents, for each changed program file, the sequential program change actions and respective change dimensions regarding the introduced syntactic and semantic features. This output is converted into qualitative input data for the following analysis step by transforming the tabulated data into a figurative representation of the program construction sequence, which is the variable-specific *construction flowline*, ii) the input to the second qualitative analysis step consists of the figurative representations of fundamental program changes, with one variable-specific *construction flowline* on layer `L3` for each variable. The output consists of a catalog of micropattern labels that have emerged during employing Grounded Theory, and of fully coded variable-specific *construction flowlines* for all input variables. iii) input to the third, quantitative analysis step are the catalog of micropattern labels and the fully coded variable-specific *construction flowlines.* The data of the qualitative analysis is converted to quantitative data to incorporate descriptive statistics into the analysis. The output is, finally, a set of usage statistics and relations between **variable construction patterns**.

Summarizing, I employ a **sequential mixed methods** study design that incorporates **data conversion**, first from quantitative to qualitative to apply Grounded Theory to figurative representations, and second from qualitative to quantitative to apply descriptive statistics to tabulated data.

The rationale for incorporating qualitative research methods is to generate a theory of **program construction patterns** while withholding *a priori* assumptions. Such an approach is valuable for this study as the combination of the medial form of the data, its type and granularity, and the specific focus point are unique. As such, there is the potential to generate new kinds of hypotheses for the field of computer science education and specifically for programming education. This is supported by the notion of Kinnunen and Simon, advocating for the use of qualitative methods, specifically grounded theory, in computer science education research to build theories grounded on data, and to generate hypotheses for future studies [KS10]. With regard to the combination of employed study methodology and analysis data, there is no related work to the best of my knowledge that applies qualitative analysis to sequential program construction data.

Second, let me introduce the learning analytics methodology and the example problems used in the mixed methods study. To record the study participants' program construction sequences, the IDE-based learning analytics methodology introduced in Section 3.3 was used. Specifically, the instrumented `IntelliJ` plugin was used to capture `Java` program construction sequences, with the uniform data collection server `trackserver` as backend to store the data. The data of this study is openly available through the use of a data collection repository [KWB20].

Regarding processing the data from captured program construction sequences on *key-stroke* granularity, a software package called `pattern-browser`, consisting of `Java`, `Python`, and `R` code, was developed by me. This software package is openly available[5] and includes the following features noteworthy for this section:

1. A textual **program increment browser** to load and sequentially switch between *program increments* of a programmer

2. Computation of **semantic and syntactic features** of program construction sequences, as introduced in the previous sections

3. Code to **visualize** program construction sequences, founded on the feature computation

Processing the data for this study to obtain the annotated, variable-specific *construction flowlines* was done in a semi-automatic way. First, the features were automatically computed by using `pattern-browser`, obtaining raw data of the *construction flowlines*. Next, I manually edited the raw data to correctly reflect the fundamental program changes in the sequential *program increments*. The main shortcoming of the automatic computation is that the `Gumtree` algorithm incorrectly maps changed parts in some circumstances, which is used to locate the changed lines that represent fundamental program changes. This edited data was then visualized as variable-specific *construction flowlines*, yielding the input artifacts of the mixed methods study (Figure 4.14).

Two example problems have been recorded with the described learning analytics methodology. The first example problem can be seen as a warm-up method and is inspired by a *Reversing* problem showcased by Lister [Lis11b, p. 15], given

[5] Link to the repository: `https://gitlab-iid.aau.at/seqtrex/pattern-browser`

```
public class MoveElements {
  /* The purpose of this code is to move all elements of the
  * input array one place to the right.
  * The rightmost element is moved to the leftmost position. */
  public static void moveElementsRight(int[] arr){
    int temp = arr[arr.length - 1];
    for(int i = arr.length - 2; i >= 0; --i){
      arr[i+1] = arr[i]; }
    arr[0] = temp; }

  /* Implement code that undoes the effect of moveElementsRight.
  * Write code to move all elements of the input array one place to the
  * left. The leftmost element is moved to the rightmost position. */
  public static void moveElementsLeft(int[] arr){
    /* TODO: implement */ }}
```

Fig. 4.15: Warm-up example problem used to record program construction sequences in `Java`. The problem is based on the *Reversing* problem by Lister [Lis11b, p. 15].

in Figure 4.15. This problem requires the study participants to write the method `moveElementsLeft`, reversing the effects of the given method `moveElementsRight`. As a programmer needs to reason about the given method and its effect on the data structure as a whole to be able to reverse the effects, this is a *concrete-operational* task. In practice, most programmers started by copying the given method, and sequentially worked on the lines to modify the method's effect. Consequently, this method did not generate program construction sequences utilizable to identify *variable construction patterns*.

The second example problem requires the study participants to implement the method `convertBinaryArray`, as shown previously in Figure 4.3. To iterate, the study participants were required to convert the input array with binary data to decimal and hexadecimal numbers. Because of the variety of possible solution strategies, this method is suited to identify *variable construction patterns*.

Third, let me introduce the cohorts of programmers participating in the study, presented in Table 4.8. The **students** cohort of the exploratory study was extended, including: i) 17 (under)-graduate students recruited from a course on software project management, held at the University of Klagenfurt in the winter term 2018, and ii) 5 undergraduate students recruited from a course on object-oriented modeling and implementation with `Java`, held at the University of Klagenfurt in the summer term 2019. These groups of participants represent the **students** cohort of the mixed methods study, with a total of 39 implemented methods, and a total of 131 control and data variables that have been semi-automatically processed to variable-specific *construction flowlines*. The students estimated their `Java` programming skills at a mean 5.70 ± 1.93 out of 10 ($n = 20$, not all students completed the demographic survey).

The cohort of industry professionals is identical to the cohort of the exploratory study. 9 professional programmers have been recruited from a software company in Austria, representing the **professionals** cohort of the mixed methods study, with a total of 19 implemented methods, and a total of 60 control and data variables. The professionals estimated their programming experience at a mean 17.29 ± 9.62 years and estimated their Java programming skills at a mean 6.71 ± 1.67 out of 10 ($n = 7$, again not all professionals completed the demographic survey).

Tab. 4.8: Overview of the two experimental cohorts, (under)-graduate students and industry professional programmers, and resulting basic data of the implemented program construction sequences for the two example problems shown in Figure 4.15 and Figure 4.3. This represents the data used in the mixed methods study to identify *variable construction patterns.*

	Student	***Professional***
Cohort Size	22	9
Prog. Experience		(n=7) 17.29±9.62
Estimated Java Skills	(n=20) 5.70±1.93	(n=7) 6.71±1.67
Number of Methods	39	19
Average Methods	1.77±0.60	2.11±0.74
Success moveElementsLeft	9/22 | 41%	5/9 | 55%
Success convertBinaryArray	7/22 | 32%	5/9 | 55%
Number of Variables	131	60
Variables / Method	3.36±2.03	3.16±1.66
Data Vars / Method	1.05±1.55	1.16±1.31
Control Vars / Method	2.31±0.85	2.00±0.92

4.4.2 Qualitative Identification of Variable Construction Patterns

The description of the qualitative analysis is structured in two sections. First a detailed description of the methodological approach following Grounded Theory is given (Section 4.4.2.1, see Cohen et al. [CMM11, p. 598ff] regarding Grounded Theory, and Cohen et al. [CMM11, p. 559ff] regarding coding and content analysis in qualitative research), and second the analysis of results by organizing the findings chronologically and based on logical themes is carried out (Section 4.4.2.2, see Cohen et al. [CMM11, p. 551f] for recommendations on result presentation – I follow way number *seven*).

4.4.2.1 Methodological Approach following Grounded Theory

The methodological approach of Grounded Theory builds on the following set of tools: theoretical sampling, coding (open coding, axial coding, and selective coding), constant comparison, identification of core variables, and saturation [CMM11, p. 598ff].

Theoretical sampling describes the ongoing, iterative data collection and analysis, which influences itself in an iterative, self-reflexive way: data collection is influenced by the emerging theory and its categories (the analysis) and vice versa. A key criterion for sampling is theoretical relevance, how data contributes to the emerging theory.

The **coding** of data, in this case, the visual representations of program construction steps, is a cornerstone of different qualitative research methods. In Grounded Theory, three types of **coding** are used: **open coding** is used to explore the data and to ascribe (categorical) labels, i.e., codes, to chunks of the data, **axial coding** is used to create links between existing codes and categories to arrive at interconnected relationships of codes and categories, and **selective coding** is used to identify core codes and to assess such codes and their respective place in the (axial) coding scheme in accordance with (pre-existing) theory. Lastly, the goal is to generate story-like narratives, integrating categories in the emerging coding model to satisfy and saturate the theoretical demands.

All **coding** and analysis facilitate the strategy of **constant comparison**, where new data is iteratively compared with existing (coded) data and categories, which can lead to modifications to the existing categories (and the underlying theoretical assumptions) until categories achieve a proper fit with the data. This **constant comparison** affords itself to theoretical saturation, i.e., the goal of the emerging theory and the accompanying categories should be a perfect fit with the data. Two methods are suggested for **constant comparison**: *unitizing* by dividing existing, coded narratives into smallest pieces of information, and *categorizing* by assessing the relation of unitized pieces, devising strong categories and their properties.

During **coding** and **constant comparison**, some variables and/or categories naturally emerge to the core of the theory – this encompasses the tool of **identifying core variables**. These variables or categories integrate the greatest number of codes, concepts and, categories, and have the greatest explanatory power regarding the phenomena under study.

The concept of **saturation** describes how and when to finish with Grounded Theory, which is when '*no new insights, properties, dimensions, relationships, codes or categories are produced even when new data are added, when all of the data are accounted for [...]*' [CMM11, p. 601]. Together with **theoretical sampling**, it provides assessable means to reach *theoretical completeness*, which is reached when data can be explained by a theory.

Altogether, in light of the mixed methods study methodology given in Figure 4.14, the Grounded Theory process can be contextualized as follows:

1. **Theoretical sampling** occurs in two forms. First by collecting data of programmer cohorts to supply variable-specific *construction flowlines*. This was carried out in three waves: first the cohort of undergraduate and graduate students described in the exploratory study, second the professional programmers, and third another cohort of undergraduate students to include additional low-level students. Second by selecting the iteratively coded variable-specific *construction flowlines* in a way to maximize the projected theoretical gain regarding open **coding**.

2. Following are iterative cycles of open **coding** of new variable-specific *construction flowlines*, revisiting *construction flowlines* and established codes during axial and selective **coding**, constantly **comparing** emerging relations, combining and structuring codes along their respective axes, and establishing relations between axes and categories.

3. An important part of the iterative cycles is the identification of **core variables** that emerge during the **coding** and **comparison**. In context, the core variables emerged as separating control variables and data variables, as thus with control context being a very important feature. The dimensions along these core variables are captured with a multilabel *micropattern* approach, as explained in the next section.

4. Successively, the process converges to a point of **saturation** when both newly added variable-specific *construction flowlines* and **coding** and **comparison** do not provide additional knowledge. The outcome of the *micropattern labels* for control variables and data variables are summarized in Table 4.9.

This process is put in a structured narrative in the following section.

4.4.2.2 Structured Narrative of Results based on Logical Themes

True to the methodology of Grounded Theory, I did not produce categories prior to open coding. As the combination of topic under study (patterns of program construction on the level of variables) and data representation (variable-specific *construction flowlines*) is unique, I did not enter analysis with pre-constructed theories or relations, letting myself be guided by the emerging findings in the data. The only notion that lingers in the back of my mind during analysis is the notion of *plans* in the work of Soloway [Sol86], comprising the relation of mechanisms (the syntactic-structural programming parts that make up parts of an algorithm) and explanations (the semantic understanding of the programming parts). In my context, I ascribe sufficiently trained programmers the ability to execute programming steps in a strategical, plan-like manner.

After **theoretically sampling** the first two cohort groups, cohorts of different levels of programming skills with 17 (under)-graduate students and 9 professional programmers, I started to get an overall impression of the data and its differentiating aspects by reviewing different program construction sequences regarding the sequential changes to the textual `.java` program and the representations these changes have in the variable-specific *construction flowline*. I reviewed the complete data in this way before starting with **coding**.

Next, I moved to **open coding**, beginning with distinct program construction sequences (those representing a comparatively straightforward implementation) but interleaving the process of **open coding** with program construction sequences that include (potentially) contradictory aspects. Therefore, this selection was based on **theoretical sampling** by, iteratively, coding and comparing program construction sequences that can actively add to the emerging constructs of codes and categories.

During coding, I assigned codes, i.e., semantic labels, to a sequence of non-continuous *program increments* (usually two to four), which are represented in the *construction flowlines*. Initially, these codes are very detailed and specific, taking into account different sequences and dimensions of change types. The names of the codes, the semantic labels, contain an inferential interpretation on what the coded sequence could mean in the context of the specific construction sequence. Example codes include: i) 'construction of *bounded var-dependent loop*' for a construction sequence that adds variables to loop definition, condition, and update, that also **adds** a **use** of a loop variable inside the **loop control context**, or ii) 'construction of *loop accumulation variable*' for a variable that is **defined** and **used** inside a **loop control context**.

For the first program construction sequences, the number of codes quickly grew (converging up to a total amount of 40), considering variations regarding local variables and parameters, the control contexts, the sequence of *program increments*, and the inclusion of modification and deletion actions. Sequences including modification and deletion of program statements are only sparsely coded – given the specificity they represent (they are unique in a programmer's *struggle* to implement an algorithm), they proved difficult to be condensed in an overarching, comparable phenomenon.

In tandem with **open coding**, iterative steps of **axial coding** were inserted in the working cycle to merge the specific, sequence-related codes to those with similar concepts, for example including all loop-conditional related codes regardless of their sequence. The result of **axial coding** is to iteratively capture codes '*along the axes of central categories*' [CMM11, p. 600] and form categories. During **axial coding**, I aimed at finding generalities in the specific codes that relate to one and the same concept. The result of **axial coding** includes the combination of codes, regardless of specific variations (variables/parameters, sequence, modification/deletion actions), along the two central, emerging category types: the **control context** (and therefore, the **scope**) in which variables are **defined** and **used**, the **control purpose** fulfilled by variables.

During **selective coding**, I iteratively worked on the inclusion of the variations still present in the specific codes to flesh out the general properties and dimensions of the categories, distilled from the specific properties and relations found in the program construction sequences. The goal was to describe the relations of different concepts inter– as well as intra-categorical. This resulted in relations between category types of the following nature: single variables fulfilling multiple **control purposes**, variables of different **control purposes** providing the **control context** to other (data) variables, single variables being used in multiple **control contexts** or only in specific **control contexts**, relations of **definition** and **use** for variables (tied to the control flow and data flow).

For a last round of **theoretical sampling**, an additional cohort of students was captured (programmers at the end of being novices: 5 undergraduate students) to work towards **saturation**. This cohort did not contribute additional codes, categories, or insights. Therefore, I came to the conclusion that, for the specific situation given by the example problems, **saturation** was reached.

By **constant comparison**, codes emerge, merge, are interlinked and categorized, and finally **core variables** (**core categories** in order to not confuse them with programming variables) and the criteria how to categorize the codes have emerged. My **core categories** include the basic distinction between **control variables** (variables that fulfill syntactic **control purposes**) and **data variables** (variables that are **control-dependent**). I call this distinction *basic* as certain variables could fulfill **control purposes** in one algorithmic part but are still pure **data variables** in other algorithmic parts. In the investigated data set, such variables are usually for two different computational parts that could be refactored into separate methods, separating the variable categories again. In those cases, the variable is assigned the category of **control variables**.

With the category types and the relations between those category types introduced above, I arrive at the following (grounded) theory to identify and assign **variable construction patterns** in variable-specific *construction flowlines*. Within the **core categories**, the rationale emerged that variables exhibit multiple different parametric variations along the axes of the **core categories**, bound by the *algorithmic affordance* given by the strategical plans chosen by the programmer. This means that different, qualitatively important aspects for the semantic meaning of a variable construction can be present at the same time.

Tab. 4.9: All micropattern labels for *variable construction patterns* of control / data variables.

Control Label	***Explanation***
loop-condition	**Used** in loop condition statements
loop	**Defined** and **used** in loop initialization, condition, and update statements
conditional	**Used** in conditional statements
computation	**Used** in control-dependent data statements

Data Label	***Explanation***
loop-dependent	**Defined** and/or **used** in loop control context
conditionally-dependent	**Defined** and/or **used** in conditional control context
redefined	Data flow includes no **use** between two **definitions**
self-defining	**Used** in the same statement the variable is **defined**
single-scope-use	All **uses** in single control context and scope

To best capture those variations in a catalog of assignable labels, with the goal of assigning **variable construction patterns**, I now introduce my devised multi-label approach that forms the result of my methodological step of Grounded Theory. For each of the **core categories**, I introduce a set of **micropattern labels** that cover the variations discovered in the data. Those labels can be assigned to variables independently – and collectively, as a set, represent the **variable construction pattern** assigned to a specific variable (and its *construction flowline*). Table 4.9 contains an overview and an explanation for each micropattern label of the multi-label assignment approach for *variable construction patterns*. Following, I summarize the catalog of multi-label micropatterns.

Regarding micropattern labels for **control variables**, there are two labels for variables with **loop control purpose**: the label *loop-condition* that is assigned to variables that are **used** in loop condition statements, and the label *loop* that encompasses the previous label and is assigned to variables that are **defined** and **used** in loop initialization, condition and update statements. The next label is assigned to variables with **conditional control purpose**: the label *conditional* is assigned to variables that are **used** in conditional statements. The last label applies to all control purposes: the label *computation* is assigned to variables that are **used** in control-dependent data statements to contribute to computations.

Regarding micropattern labels for **data variables**, there are two labels regarding the **control context**: the label *loop-dependent* is assigned to variables with **defini-**

tions and/or **uses** in a loop control context, and the label *conditionally-dependent* is assigned to variables with **definitions** and/or **uses** in a conditional control context. Given these labels, relations between control variables and data variables can be explored – but this is not currently included.

The remaining labels capture different relations of **definition** and **use** of the same variable. The label *redefined* is assigned to variables with a possible data flow that has no **use** between two **definitions**. The label *self-defining* is assigned to variables that are **used** in the same statement where they are also **defined**. The label *single-scope-use* is assigned to variables for which all **uses** are in a single control context *and* scope. Together, these micropattern labels represent generalized, semantic applications of the labeled variables that have been extracted from the example problems.

I conclude the section with closing remarks regarding the resulting catalog of micropattern labels. Of note is that the sequential order of *program increments* is currently not captured in the labels. But for each micropattern label, there is a finite number of denominating *program increments*. Aware of this finding, the sequence of these different denominating *program increments*, which can be automatically identified in the program construction sequence, can be evaluated in a future study.

Currently, the label assignment is based on the last applicable assignment in the sequential program construction. This means that, for example, a variable could be assigned one set of labels *during* program construction, but another set of labels *at the end* of program construction. This is relevant for an LA-powered live evaluation of program construction to report currently applicable **variable construction patterns**.

Example program construction sequences and detailed explanations for each label, with regard to the automatic discovery, are given in Chapter B. While the identified micropatterns do not provide the same semantic meaning compared to the *plans* proposed by Soloway [Sol86], they are more generic and can be automatically identified in program construction sequences based on the introduced syntactic and semantic features.

4.4.3 Quantitative Distribution of Variable Construction Patterns

With the catalog of micropattern labels of control variables and data variables (Table 4.9) identified for the example problems, I next investigate the programmers' variable use in a quantitative manner to fully arrive at an answer for the third research question of this thesis. Over the course of the qualitative analysis, I manually labeled all variable-specific *construction flowlines*, which results in a set of assigned control labels or data labels for each variable used by the programmers, i.e., for 131 variables in students' program construction sequences, and 60 variables in professionals' program construction sequences.

Tab. 4.10: Distribution of assigned micropatterns for control variables.

Control Micropatterns	***Student Variables (n=90)***	***Professional Variables (n=38)***
loop-condition `LC`	43 | 48%	17 | 45%
loop `L`	41 | 46%	17 | 45%
conditional `C`	19 | 21%	8 | 21%
computation `CP`	78 | 87%	33 | 87%
Mean Labels per Var	2.01 ± 0.46	1.97 ± 0.49

Tab. 4.11: Distribution of assigned micropatterns for data variables.

Data Micropatterns	***Student Variables (n=41)***	***Professional Variables (n=22)***
loop-dependent `LD`	23 | 56%	18 | 82%
conditionally-dependent `CD`	14 | 34%	10 | 45%
redefined `R`	3 | 07%	1 | 05%
self-defining `SD`	23 | 56%	17 | 77%
single-scope-use `SSU`	16 | 39%	4 | 18%
Mean Labels per Var	1.93 ± 0.56	2.27 ± 0.54

I now employ a three-step investigation to assess the quantitative distribution of the use of variable construction patterns: first by assessing the quantitative use of each micropattern label individually, second by assessing sets of micropattern labels, used in conjunction, per variable, and third by assessing all sets of micropattern labels per method. Due to the low sample size of programmers, I opt to use descriptive statistics to complement the qualitative results. This statistical analysis bears no inferential strength regarding the generalization to different cohorts. The following descriptive statistics concern the measured cohorts of students and professional programmers.

4.4.3.1 Distribution of Individual Micropatterns

First comes the use of individual micropattern labels. Keep in mind that each variable is potentially assigned multiple micropatterns labels, with the mean number of labels around 2 per variable. The mean number of labels and individual distributions are given in Table 4.10 for control micropattern labels, and in Table 4.11 for data micropattern labels.

Interpreting the relative fraction of use for each control micropattern label, there is little difference between student variables and professional variables. The predominant control label is *computation*, signifying that most of the control variables are also used in data statements to compute a value. The two control labels *loop-condition* and *loop* are each assigned to nearly half of the variables. These two labels are mutually exclusive in my catalog; that means that most variables are loop-related, which can be attributed to the underlying algorithmic affordance of the example problems. About a fifth of variables were constructed as *conditional* variables, showing that only a portion of programmers opted to use conditional branching in their solution.

Turning to data micropattern labels, differences between student variables and professional variables can be observed. Two labels are predominantly used by both groups, albeit at different relative fractions of use: the labels of *loop-dependent* and *self-defining*. About half of all students' variables are assigned to each of the two labels, but professionals construct variables of those two labels with even higher use (82% and 77%, respectively). This can again be attributed to the example problems: the problems require a computation in a loop (*loop-dependent*) and can be solved by cumulatively computing the value in a *self-defining* variable. The variables of the label *conditionally-dependent* are constructed less often compared to their loop variant, related to the construction of conditional branching. They are constructed more often by professionals (45%) compared to students (34%). On the other hand, students more often construct variables with *single-scope-use* (39%) compared to professionals (18%). The label *redefined* is rarely assigned to constructed variables, with only four total assignments.

4.4.3.2 Distribution of Sets of Micropatterns

The next analysis step is to assess sets of micropattern labels that are simultaneously assigned a single variable. Barring the order of different *program increments* in the program construction sequences, these sets of micropattern labels represent the different **variable construction patterns** that are attainable with the current catalog of micropatterns. The distribution of different sets of labels, ordered by the number of labels, are shown in Table 4.12 for control micropattern labels and in Table 4.13 for data micropattern labels. These distributions are now mutually exclusive; the percentages are rounded but the absolute number of assigned variables adds up to the respective n.

Regarding one-label sets of control micropattern labels, there is a balanced but small construction in both groups. Only the label *computation* is not assigned to any professional variable. This shows that most of the control variables are constructed for more than a single control purpose.

Regarding two-label sets of control micropattern labels, for both students and professionals, two specific sets make up a large fraction of assigned sets: the label

Tab. 4.12: Distribution of assigned *variable construction patterns* (sets of micropatterns) for control variables. Percentages are rounded.

Sets	***Student***		***Professional***	
Control Micropatterns	***Variables (n=90)***		***Variables (n=38)***	
loop-condition LC	2	02%	3	08%
loop L	3	03%	1	02%
conditional C	3	03%	1	02%
computation CP	1	01%	0	00%
LC - C	3	03%	0	00%
LC - CP	35	39%	13	34%
L - C	1	01%	0	00%
L - CP	30	33%	13	34%
C - CP	2	02%	3	08%
LC - C - CP	3	03%	1	02%
L - C - CP	7	08%	3	08%

sets {*loop-condition, computation*} and {*loop, computation*}. Together, these two label sets amount to 72% of all variables for students and 68% of all variables for professionals. The other two-label sets are only constructed in small fractions: professionals only constructed two-label set variables of {*conditional, computation*} while students constructed variables of all reported two-label set types.

Regarding three-label sets of control micropattern labels, only two specific sets are found in the data, with similar fractions for both groups summing to about 10%: the label sets {*loop-condition, conditional, computation*} and {*loop, conditional, computation*}, differing in the label of the loop component. Variables of these sets serve both control purposes of looping and conditional branching and also serve a computational purpose, making them dependent on the specific algorithmic solution.

For data micropattern labels, one-label sets are hardly constructed by professionals (a single *loop-dependent* variable is assigned), but most single labels are constructed by students at a small fraction. Only the label *self-defining* is not constructed in solutions to the example problem.

Regarding two-label sets of data micropattern labels, there are some sets with a high fraction of assigned variables. One set applies to both students and professionals: {*loop-dependent, self-defining*} is the set most often assigned to data variables (students: 22%, professionals: 36%), which can again be attributed to the example problems. For the remaining two-label sets, professional programmers construct them in low fraction. For students, two other sets sport a high fraction: {*loop-dependent, single-scope-use*} (17%) and {*conditionally-dependent,* *Tab. 4.13:* Distribution of assigned *variable construction patterns* (sets of micropatterns) for data variables. Percentages are rounded.

Sets	***Student***		***Professional***	
Data Micropatterns	***Variables (n=41)***		***Variables (n=22)***	
loop-dependent `LD`	3	07%	1	04%
conditionally-dependent `CD`	1	02%	0	00%
redefined `R`	1	02%	0	00%
self-defining `SD`	0	00%	0	00%
single-scope-use `SSU`	3	07%	0	00%
`LD - CD`	0	00%	1	04%
`LD - SD`	9	22%	8	36%
`LD - SSU`	7	17%	1	04%
`CD - R`	1	02%	1	04%
`CD - SD`	9	22%	2	09%
`CD - SSU`	2	05%	1	04%
`LD - SD - SSU`	4	10%	2	09%
`LD - CD - SD`	0	00%	5	23%
`CD - R - SD`	1	02%	0	00%

self-defining} (22%). Students seem to construct different algorithmic solutions as inferred from the data variables in different control contexts.

Regarding three-label sets of data micropattern labels, the second-highest fraction of assigned variables can be found for professionals: the label set {*loop-dependent, conditionally-dependent, self-defining*} (23%), which is not constructed by students, but comparable to the two-label set mentioned above. Professional programmers seem to construct algorithmic solutions where data variables are used in multiple control contexts. The label set {*loop-dependent, conditionally-dependent, single-scope-use*} is constructed by both groups in a similar fraction.

When looking at the label sets with highest fractions, four label sets already include 71% of all student variables (sorted: {*loop-dependent, self-defining*}, {*conditionally-dependent, self-defining*}, {*loop-dependent, single-scope-use*}, {*loop-dependent, self-defining, single-scope-use*}).

Four other label sets include 77% of all professional variables (sorted: {*loop-dependent, self-defining*}, {*loop-dependent, conditionally-dependent, self-defining*}, {*conditionally-dependent, self-defining*}, {*loop-dependent, self-defining, single-scope-use*}). As can be seen, most of the variables are covered when considering five different data micropattern label sets (out of $2^5 - 1$ possible sets excluding the empty set), albeit at different relative fractions of construction between the groups.

4.4.3.3 Distribution of Variable Construction Patterns per Method

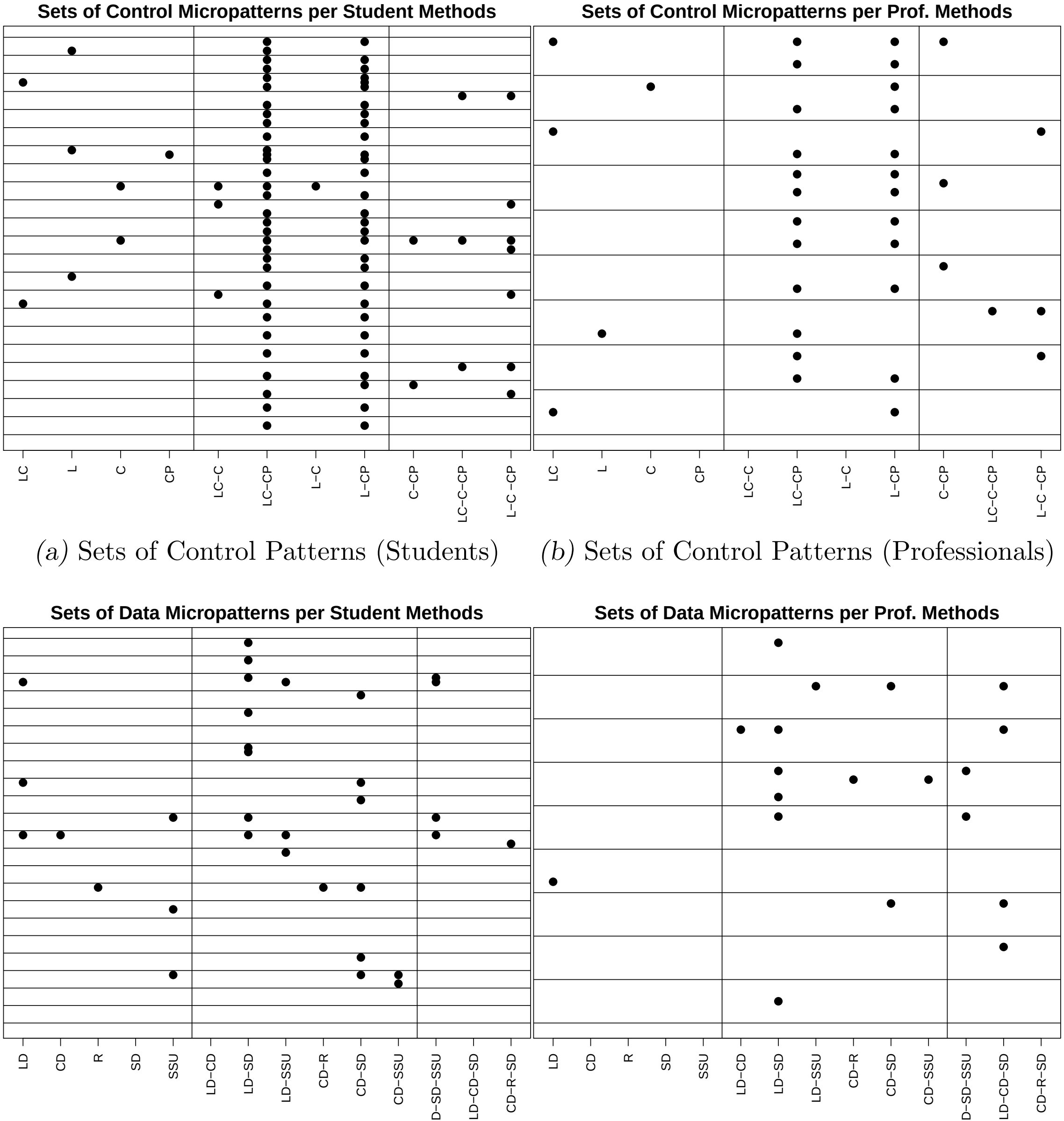
The last analysis step is to assess all sets of micropattern labels per method to investigate the relation of **variable construction patterns** in a method. Figure 4.16 shows, for all programmers and their respective constructed methods, the sets of assigned **variable construction patterns**. The figure indicates the presence or absence of a set in a method, not the number of variables constructed for each set.

Again, let me first focus on the control **variable construction patterns**, shown in the top row of Figure 4.16 (left are student methods, right are professional methods). Recalled from the previous analysis step, the two most commonly constructed control label sets are: {*loop-condition, computation*} ({*LC-CP*}), and {*loop, computation*} ({*L-CP*}). Visible from the figure, these two sets are most commonly used together – which can be traced back to the type of loops of the example problems, all requiring a loop variable as well as a loop condition variable for a *standard* solution.

Next, let me take a look at the less common sets. For one-label control sets, there is no discernible relation to other label sets, which could be attributed to the low fraction of variables assigned a one-label set. For three-label sets, a discernible relation in student patterns is that most (four out of six) students constructing variables that are assigned three-label sets construct multiple such variables with different labels. A possible interpretation is that the algorithmic solutions constructed by those students are *bloated* in terms of their algorithmic affordance, resulting in control variables that need to cover multiple control purposes. This reasoning could also apply to the additional two-label variables constructed by the students – absent in professional program construction sequences. Although the absolute number of professionals constructing three-label set variables is the same, only one out of six professionals constructed variables with more than one different three-label set.

Next, I focus on the data **variable construction patterns**, shown in the bottom row of Figure 4.16 (left are student methods, right are professional methods). The two-label set {*loop-dependent, self-defining*} ({*LD-SD*}) is constructed by the most number of students and professionals alike. A second two-label set with a high fraction, {*conditionally-dependent, self-defining*} ({*CD-SD*}), provides a mutually exclusive partition of student methods. Interpreted, this represents different algorithmic approaches to computing the solutions of the example problems.

Next, I take a look at the less common sets. For one-label data sets, students construct specialized variables (with single labels) in addition to variables with multiple labels. Professionals hardly construct those specialized variables, with only a single method containing one-label data variables. For three-label data sets, while students do not construct any variable that incorporates multiple control contexts, professionals construct such variables – evident in the three-label set {*LD-CD-SD*} as well as the two-label set {*LD-CD*}.



Sets of Control Micropatterns per Student Methods

(a) Sets of Control Patterns (Students)

Sets of Control Micropatterns per Prof. Methods

(b) Sets of Control Patterns (Professionals)

Sets of Data Micropatterns per Student Methods

(c) Sets of Data Patterns (Students)

Sets of Data Micropatterns per Prof. Methods

(d) Sets of Data Patterns (Professionals)

Fig. 4.16: Visualization of the use of sets of micropatterns, which are the variable construction patterns, per method. Student patterns are given on the right, professional pattern are given on the left. Control patterns are given on the top, data patterns are given on the bottom. Each separate row represents one programmer, and *inline-rows* represent single methods. Each dot represents that at least one variable of the method is assigned the specific set of micropatterns of Table 4.12 and Table 4.13.

Assessed per row, i.e., per method, students tend to use a higher number of different data **variable construction patterns** during program construction compared to professionals. Only two methods constructed by professionals contain variables with three different data **variable construction patterns**, while the maximum number of different data **variable construction patterns** assigned in student methods is five. Interpreted, students use multiple, specialized data variables, whereas professionals tend to use data variables that are used in different control contexts. This was already hinted at with the mean number of labels per data variable, shown in Table 4.11, and it becomes evident in this analysis step.

An interesting note is that the label *redefined* is mostly constructed in conjunction with the label *conditionally-dependent*, representing that the variable value is conditionally re-set. This applies to both student and professional variables.

4.4.3.4 Summary of Distribution

In summary, starting with analysis step two, I introduced discerning capabilities of the assigned **variable construction patterns**. At face value, a discerning capability is that algorithmic affordance governs the possible sets of labels, and especially the frequent labels, assigned in program construction sequences. Most variables can be covered by considering only a small number of different patterns; two sets of control patterns to cover more than 70% of all control variables, and five sets of data patterns to cover more than 70% of all data variables. Considering analysis step three, cautious inference towards the algorithmic approach of programmers is possible: discerning between **variable construction patterns** with single labels (highly specialized purpose for variables, but the algorithmic approach is potentially *bloated*) and those with multiple labels (combined purpose for variables, e.g., control context or control dependency). This discerning capability is applicable to control as well as data variables.

4.4.4 Discussion of Threats to Validity

The same example problems and participant cohorts have already been investigated by Kesselbacher and Bollin [KB21], focusing solely on the use of slice-based cohesion metrics in the described learning analytics setting to assess the construction of semantic structures and the identification of trains of thought during programming. The findings of the publication, how to didactically and practically incorporate IDE-based learning analytics in programming education, are further discussed in Chapter 6. A number of threats to validity have already been discussed in the publication – in this thesis, I extend the discussion for the mixed methods study.

Two sources of threats need to be addressed regarding the internal validity, respectively the credibility, of the study: the application of the methods for the mixed methods study and the application of the developed software tools to process the recorded data.

Regarding the application of the qualitative research methods, I follow the intertwined methodological approach of Grounded Theory, consisting of: theoretical sampling (multiple rounds of data collection; being led by the data), coding (open coding, axial coding, and selective coding is applied in an iterative fashion), constant comparison by revisiting codes and documents (variable-specific *construction flowlines*), developing core variables, and reaching saturation with respect to the example problems [CMM11, p. 598ff]. Moreover, I strive to adhere to important kinds of validities for qualitative methods: descriptive validity (by credible descriptions of accounts of the phenomenon under study), interpretative validity (by reproducing the meaning the data has from the participant's point of view as accurate and still as objectively as I could), theoretical validity (by arriving at explanations only from accounted data), generalizability (discussed below), evaluative validity (by evaluating and judging the phenomenon with the accounted data), and transparency (by providing a chronological, logical story of research findings and organizing the results by theme). Specific employed steps include: theoretical sampling, the specific use of outliers and extreme cases, analysing negative cases (in this specific context, including variables that include micropatterns of control and data variables), contrasting and comparing groups, and checking for representativeness (within the group of participants) [CMM11, p. 181ff].

Next, I describe threats specific to the qualitative analysis approach, with the following notion: while qualitative research is concerned with '*processes rather than outcomes*', internal validity (*credibility*) and external validity (*transferability*) are still important measures of validity that need to be addressed [CMM11, p. 180ff]. This is especially true in the mixed methods research design at hand, with data transformations in both directions, and therefore an intertwining of possible interpretative and inferential strengths, but also of possible threats.

The very specific medial form of the data that is subject to qualitative analysis is a threat to its credibility. The figurative representations of the participants' products are different from typical text media (e.g., interview texts or self-written text answers), but are also different from typical audio-visual media. While the figurative representations accurately depict the participants' products, there is a potential analytic-logical bias introduced by the types of information included in the representation: the types and dimensions of syntactic and semantic features. I mitigated this threat in two ways: first by performing an exploratory pre-study (described in Section 4.2) that informed which features are to be included in a representation of program construction sequences, and including all syntactic and semantic features that can automatically be extracted from sequential program changes, and second I followed a recommended approach to apply Grounded Theory research on audio-visual data: at first discerning the overall picture and '*global impressions*' before focusing on detailed parts – a notion that is described as turning the paradigm of Grounded Theory on its head [CMM11, p. 591].

Another threat to the credibility is the single data source. There is no triangulation of different data sources, e.g., participants' accounts of their own program construction sequences. To mitigate this threat for future studies, think-aloud protocols with participants verbalizing their thoughts during programming can be incorporated. Given the unique nature of this study, no briefing of participants regarding what they should verbalize was possible – theory regarding the **variable construction patterns** formed during analysis.

Regarding the application of the quantitative research methods, I only employ descriptive statistics at this stage as the goal was to investigate relations between the use of **variable construction patterns** while still respecting the potential uniqueness of each program construction sequence that is facilitated with the use of the qualitative analysis method. As such, the (cautiously) inferential interpretations foremost apply to the groups of participants under study. In order to generalize the findings, two changes to the research design are possible: i) increase the number of participants in each group to be able to generalize, for different levels of programming skills, the use of **variable construction patterns**, and ii) diversify the experimental problems to expand and validate the catalog of micropatterns. In short, to arrive at a study capable of generalization and transferability of the findings, an experimental setup is needed.

In the mixed methods study methodology, I used a number of software tools developed by me to process the data. This includes: i) recording and storing the program construction sequences on *key-stroke* granularity, ii) computing semantic and syntactic features of individual *program increments*, including an adaptation to the *Java* slicer *JRazor* [Ram13] to compute the slice-based cohesion metrics, and iii) processing the computed features into variable-specific *construction flowlines*.

Regarding `i)` I examined all recorded program construction sequences to assure that they constitute full program construction sequences, that all *program increments* are in the correct order, and that no changes are missing. Regarding the semantic features of `ii)` extracted with the *Java* slicer, I validated the correct computation of slice profiles of the *Java* slicer with small, devised programs regarding typical looping and conditional branching structures. In the variable-specific *construction flowlines* used in the mixed methods study, only the method intersection, computed from the slice profiles, was used. Regarding `ii+iii)` a threat has been identified in the use of `Gumtree`, as changed parts in pairs of *intermediate programs* are incorrectly mapped in some circumstances. This mainly affects: pairs of *intermediate programs* with a great number of changed program statements (when, for example, compile errors are only fixed after many changes), and small changes that do not greatly affect the abstract syntax tree (when, for example, the comparison operator in a conditional branching is changed). To eliminate this threat, I manually edited all automatically extracted variable-specific *construction flowlines* and double-checked the reported syntactic features by hand.

Three sources of threats need to be addressed regarding the external validity and the transferability of the study and the results: the used example problems, the used programming language, and the cohorts of study participants.

Regarding the example problems, the first problem `moveElementsLeft` is inspired by neo-Piagetian findings of Lister (see [Lis11b, Lis16]), and represents '*one of the defining qualities of a concrete operational programmer*' [Lis16, p. 14]. In the context of this study, I planned for the use of this problem for the sake of comparability with previous research. As it turned out, while this problem is a good fit to differentiate programmers based on their level of neo-Piagetian programming skills, it provides less insight on strategical approaches that can be distilled into *variable construction patterns*. Consequently, this problem did not contribute to the generalizability of the study results.

The second problem was devised by me and my supervisor, and its uncontrolled openness can be seen as a threat to external validity. There are various ways to implement a solution, both semantically and syntactically, that are not inherently tied to the level of programming skills. However, for this mixed methods study, the uncontrolled openness is a strength to collect different strategical approaches towards the implementation. With regard to generalizing the results, I already noted that different algorithmic structures afforded by the semantic goals of a program implementation will result in different patterns of program construction. Consequently, additional research with different example problems is necessary, with the contribution of this thesis being an initial catalog of *variable construction patterns* for common structures of constructing program code with loops and conditional branching.

A second threat to the external validity is the use of a single text-based programming language `Java`. This programming language was chosen as it is the primary programming language taught at the University of Klagenfurt, where the students of the study have been recruited. Moreover, it is also a common programming language in the local software development industry, where the professional programmers of the study have been recruited. This is evident in the programmers' self-estimated `Java` programming skills. Therefore it can be concluded that the participants' proficiency in `Java` is sufficient for solving the example problems. Regarding generalizing the study results to other (text-based) programming languages, while programming skills include language-specific programming constructs, no `Java`-specific constructs have been used in the implemented methods. Therefore the catalog of *variable construction patterns* for common structures of loop and conditional branching program code can be generalized to other text-based programming languages.

Third, the recruitment of participants can be seen as a threat to external validity. Undergraduate and graduate students of the curricula of *Applied Computer Science* and *Management of Information Systems* have participated in the study, leading to uncontrolled variance in the students' programming experience. An uncontrolled variance can also be seen in the cohort of professional programmers, having a mean programming experience of 17.29±9.62 years. I mitigate this threat by using a mixed methods study, utilizing the diversity of programming approaches in qualitative analysis methods, and only using quantitative methods as support.

Another potential threat regarding the study cohorts comes from their relatively small size. To mitigate this threat, I opted to use qualitative analysis methods and descriptive statistics instead of inferential statistics. This, however, translates to a lacking strength regarding the generalizability of the results – I cannot claim that the reported usage statistics hold for different student and professional cohorts.

A final note regarding the study cohorts is that gender is disparately represented in the cohort of professional programmers. While 9 students (out of 22, 41%) reported themselves as female, only a single professional programmer (out of 9, 11%) reported themselves as female. This threat cannot be mitigated, but it has to be noted that the resulting *variable construction patterns* of professional programmers can be predominantly described as *male* patterns of program construction. Future research needs to address female professional programmers to investigate potential differences.

4.4.5 Summary of the Use of Variable Construction Patterns

Concluding the mixed methods study, I sequentially combined the qualitative research method of Grounded Theory (applied on figurative data, the variable-specific *construction flowlines*) and the quantitative research method of applying descriptive statistics to identify **variable construction patterns** and evaluate the use of those by the experimental cohorts, consisting of (under)-graduate students and professional programmers.

Following from the findings of the Grounded Theory method, variables are assigned **variable construction patterns** based on two **core categories**: control variables and data variables. For each of those categories, the resulting **variable construction pattern** is assigned based on a multi-label approach, with individual **micropattern** labels for different characteristics.

The **micropattern** labels for control variables are: *loop-condition*, *loop*, *conditional*, *computation*.

The **micropattern** labels for data variables are: *loop-dependent*, *conditionally-dependent*, *redefined*, *self-defining*, *single-scope-use*.

After fully assigning the **micropattern** labels to all program construction sequences and investigating the relative use of the **variable construction patterns**, which are sets of **micropatterns**, an answer to the third research question can be given.

RQ3. What program construction patterns are used by novice and expert programmers during program construction?

Novice programmers, comprising the cohorts of high-semester undergraduate students and graduate students, and the cohort of low-semester undergraduate students in the mixed methods study, use the following **variable construction patterns** when construction implementations for the example problems:

1. **Control variables**: The two most frequently assigned label sets are: {*loop-condition, computation*} and {*loop, computation*}, assigned to respectively 39% and 33% of control variables. Additionally {*loop, conditional, computation*} is assigned to 8% of control variables. All other label sets are assigned to less than four variables.

2. **Data variables**: The four most frequently assigned label sets are: {*loop-dependent, self-defining*} (22%), {*conditionally-dependent, self-defining*} (22%), {*loop-dependent, single-scope-use*} (17%), {*loop-dependent, self-defining, single-scope-use*} (10%), capturing 71% of all students' data variables. All other label sets are assigned to less than four variables.

Expert programmers, comprising the cohort of professional programmers in the mixed methods study, use the following **variable construction patterns** when construction implementations for the example problems:

1. **Control variables**: The two most frequently assigned label sets are: {*loop-condition, computation*} and {*loop, computation*}, both assigned to 34% of control variables. Additionally, {*loop-condition*} and {*loop, conditional, computation*} are both assigned to 8% of control variables. All other label sets are assigned to less than two variables.

2. **Data variables**: The four most frequently assigned label sets are: {*loop-dependent, self-defining*} (36%), {*loop-dependent, conditionally-dependent, self-defining*} (23%), {*conditionally-dependent, self-defining*} (9%), {*loop-dependent, self-defining, single-scope-use*} (9%), capturing 77% of all professionals' data variables. All other label sets are assigned to less than two variables.

As can be seen, the **variable construction patterns** for control variables are very similar (owing to the example problems). Regarding data variables, there is overlap but also differences between the cohorts.

4.5 Summary of Program Construction Patterns

In this section I give a summary to the chapter and conclude with answers to the second and third research questions of the thesis.

Starting with fundamental considerations regarding a semantic understanding of program construction sequences, I first established the notion of **program construction patterns** adopted for this thesis. The notion consists of typed, sequential changes extracted from program construction sequences that I call *program increments*. With these *program increments*, I choose to evaluate program construction sequences on *compilation* granularity [IBE+15].

Building on the established notions, I performed an exploratory study, to evaluate different syntactic and semantic features with the goal of identifying a set of features to be included as dimensions for typed *program increments*. To this end, I assessed the block-based program construction sequences of school students ($n = 42$) programming in `Scratch 2` and `Scratch 3` and the text-based program construction sequences of (under)-graduate students ($n = 17$) and professional programmers ($n = 9$) programming in `Java`. The result includes four syntactic dimensions to categorize *program increments*: **fundamental program change type** (addition, modification, deletion), **variable usage type** (definition, use), **changed program statement type** (data, control, other), **control context** (iteration, conditional, method, class). Additionally, a semantic dimension is considered based on assessing the **union slices** iteratively built unions of forward and backward slices for individual variables that are affected by *program increments*.

The semantic features, building on slice-based cohesion metrics, prove to be a powerful discriminator for the cohorts of the exploratory study. For the text-based cohorts, sequence-based measures of the frequency of cohesive program versions (with a method slice intersection greater than 0.5) and of the relative mean sequence length of non-cohesive program versions (with a method slice intersection smaller or equal to 0.5) reveal that professional programmers construct **significantly more** cohesive program versions (U-test, $p = .0467$), and have **significantly shorter** relative mean sequences of non-cohesive programs during program construction (u-test, $p = .0158$), both compared to the students cohort.

With the findings of the exploratory study, I can answer the second research question of the thesis:

RQ2. How can program construction patterns be defined?

The final definition of **program construction pattern** is: '*A typed, ordered, non-continuous sequence of program increments that are semantically related*'. Explained in other words, **program construction patterns** are defined in terms of **syntactic** (typed with the dimensions introduced above; the ordered, non-continuous sequence of *program increments* extracted from the program construction sequence) and **semantic** features (semantic relation is given for *program increments* affecting the same **union slice**).

Acknowledging that the above definition is generic and can be instantiated with different types of patterns of program construction in mind, I introduce a specific type of **program construction pattern**. This **variable construction pattern** is defined as follows: '*A typed, ordered, non-continuous sequence of program increments with changes to a variable that are semantically related*'. This definition lays the focal lens through which patterns are assigned to specific variables.

With regard to easier understandability, I also introduce a more colloquial definition that goes without the intricacies of my learning analytics context while still retaining the same notion. The colloquial definition of **variable construction patterns** is: '*A typed sequence of compilable program changes that affect related program statements of a variable*'.

With the definition of **variable construction patterns** established, I performed a mixed methods study to identify and assign **variable construction patterns** to variables in program construction sequences, and also to evaluate their actual use during program construction. This study entails an answer to the third research question of the thesis:

RQ3. What program construction patterns are used by novice and expert programmers during program construction?

In the first analysis phase of the mixed methods study, I employed the qualitative research method Grounded Theory to identify **variable construction patterns** in variable-specific *construction flowlines*, a figurative representation of program construction sequences for a specific variable. The result of this analysis phase consists of a catalog of **micropatterns** for control variables (four different micropattern labels) and data variables (five different micropattern labels) that can be independently assigned following a multi-label approach.

In the second analysis phase of the mixed methods study, I evaluated the distributions of the **micropatterns** with descriptive statistics. The results of this analysis phase are the notion that **variable construction patterns** are described by sets of **micropattern** labels. Table 4.14 presents the most prevalent patterns used by student programmers and professional programmers.

Giving an answer to the third research question, novice and professional programmers alike use control variables not only for *control purposes*, but also for *computation* (80% and 76% of control variables, respectively, include the **micropattern** *computation*). Moreover, patterns related to the use of control variables are dependent on the *algorithmic affordance* – in this case, loop-related control variables are most prevalent.

Regarding data variables, distinct patterns related to the control contexts the variables are used in have been identified, separating variables that are used in a single control context (loop **or** conditional) or in multiple contexts (loop **and** conditional). While these patterns are dependent on the *algorithmic affordance* just as control patterns are, the algorithmic strategy / the *plan* [Sol86] pursued by the programmer plays a bigger role in the construction of data variables. This is evident in the non-overlapping use of some patterns (novices: {*loop-dependent, single-scope-use*}, experts: {*loop-dependent, conditionally-dependent, self-defining*}), and in the varying frequency of overlapping patterns.

Lastly, as can be seen, most variables (> 70%) can be covered by a small number of **variable construction patterns**.

Tab. 4.14: Summary of the distribution of **variable construction patterns**, the sets of micropattern labels, used by programmers in the mixed methods study.

		Set of Micropatterns	***Freq.***	***Sum***
Novices	***Control Patterns***	{*loop-condition, computation*}	39%	80%
		{*loop, computation*}	33%	
		{*loop, conditional, computation*}	8%	
	Data Patterns	{*loop-dependent, self-defining*}	22%	71%
		{*conditionally-dependent, self-defining*}	22%	
		{*loop-dependent, single-scope-use*}	17%	
		{*loop-dependent, self-defining, single-scope-use*}	10%	
Experts	***Control Patterns***	{*loop-condition, computation*}	34%	84%
		{*loop, computation*}	34%	
		{*loop-condition*}	8%	
		{*loop, conditional, computation*}	8%	
	Data Patterns	{*loop-dependent, self-defining*}	36%	77%
		{*loop-dependent, conditionally-dependent, self-defining*}	23%	
		{*conditionally-dependent, self-defining*}	9%	
		{*loop-dependent, self-defining, single-scope-use*}	9%	

5. RELATION BETWEEN PROGRAM CONSTRUCTION PATTERNS AND PROGRAMMING SKILLS

In the previous chapter, I evaluated the use of **variable construction patterns** of students and professional programmers, each composed by a set of micropattern labels that have been identified with Grounded Theory, by using descriptive statistics. The next step in my endeavour to evaluate the relation of **variable construction patterns** to programming skills is to assess whether a programmer's use of identified patterns is substantially different when considering their respective level of programming skills. I aim to find out whether there are differences in the construction of program code that could be attributed to a difference in the level of programming skills, measured with my framework of **variable construction patterns** in the LA setting. Towards this aim, I follow a comparative study design to investigate differences between (under)-graduate student programmers (*novices*) and professional programmers (*experts*).

In this comparative study, I investigate specific perspectives of differences, with the goal of uncovering differences attributable to programming skills that can be used to improve programming education. These angles are explained in detail in Section 5.1, and comprise the other sections. This chapter is thus structured as follows:

1. In Section 5.1, I explain the methodological setup of the comparative study, including: the specific perspectives that are investigated and corresponding hypotheses, the means of data collection and data analysis, and the experimental cohort.

2. Section 5.2 covers the comparison of the usage of variable construction patterns, broken down to individual variables.

3. Section 5.3 covers the comparison of the usage of variable construction patterns, aggregated to methods by the construct of *method pattern profiles.*

4. Section 5.4 covers the comparison of the usage of variable construction patterns, analyzed through the lens of the sequence of program construction steps.

5. In Section 5.5, I discuss threats to the validity of the comparative study.

A summary of this chapter is given in Section 5.6, including answers to the fourth research question of this thesis:

RQ4. What are the differences between novices and experts concerning the use of program construction patterns?

5.1 Setup for Comparative Study

In this section, I first describe the methodology of the comparative study and give an overview of the null hypotheses for which differences are evaluated. Next is a summary of the overarching learning analytics methodology to collect and analyze the data, common to all approaches to assess differences. The section is concluded with a description of the participant cohorts.

5.1.1 Study Methodology and Hypotheses

The research question of the comparative study coincides with the fourth research question of this thesis, given above.

This research question utilizes the definition of *variable construction patterns* and the identified micropatterns of control variables and data variables that make up the patterns, aiming to assess differences between groups of programmers differentiated by their level of programming skill. The goal is to use the base catalog of *variable construction patterns*, which is the output of the previous mixed methods study and the answer to the research question `RQ3` covered in Chapter 4, and evaluate whether the identified micropatterns and resulting *variable construction patterns* hold discerning qualities with regards to the level of programming skills. In other words, with the comparative study I seek to evaluate whether the analytical use of identified micropatterns of control variables and data variables can detect differences in program construction sequences that can be attributed to different levels of programming skills.

To work towards an answer to this research question, I follow a sequential quantitative study methodology, represented in Figure 5.1. Following the recruitment of study participants and the semi-automatic preprocessing of the recorded program construction data (described below), I evaluate the programmers' use of patterns and the discerning capabilities of *variable construction patterns* on three sequential analysis levels. Each level is characterized by a null hypothesis ($H0_1$–$H0_3$) that describes the basic assumption of the identified micropatterns not holding discerning capabilities at the specific level of analysis. In order, these analysis levels are: i) differences in micropatterns and variable construction patterns for individual variables, ii) differences in micropatterns, aggregated on the level of methods, and iii) differ-

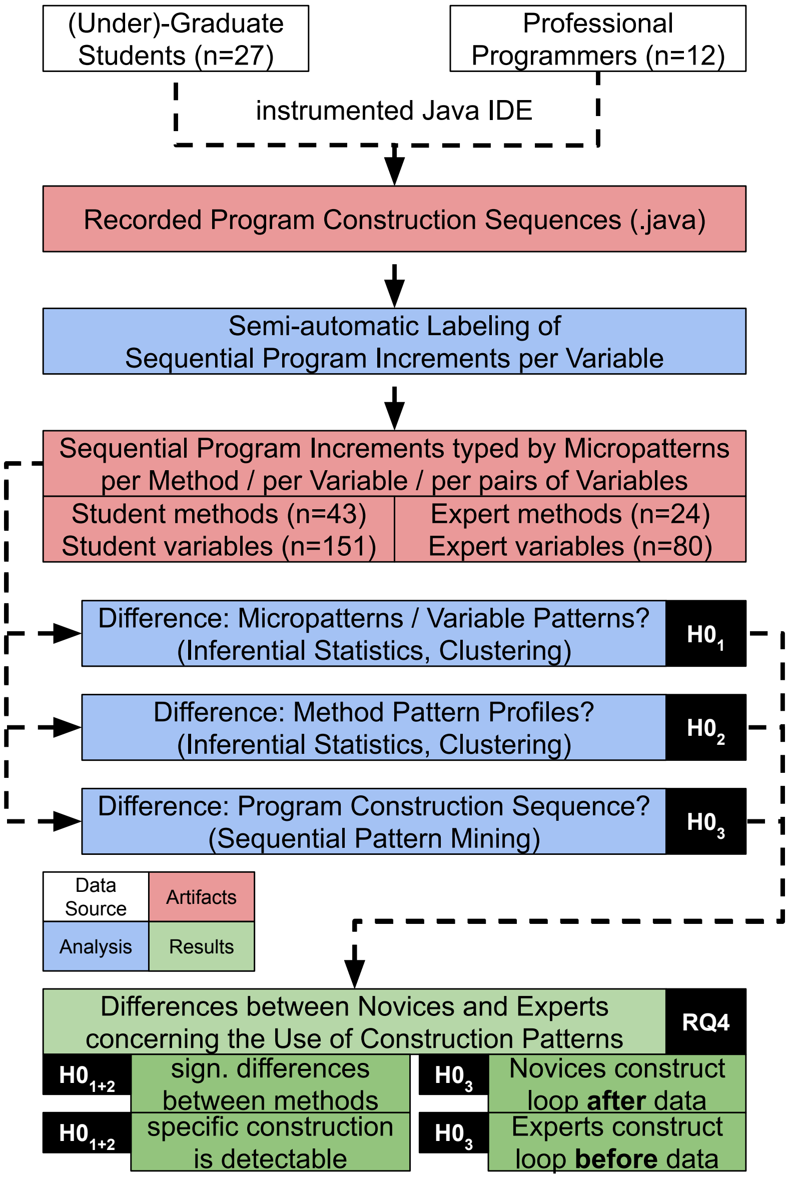

(Under)-Graduate Students (n=27)
Professional Programmers (n=12)
instrumented Java IDE
Recorded Program Construction Sequences (.java)
Semi-automatic Labeling of Sequential Program Increments per Variable
Sequential Program Increments typed by Micropatterns per Method / per Variable / per pairs of Variables
Student methods (n=43) Student variables (n=151)
Expert methods (n=24) Expert variables (n=80)
Difference: Micropatterns / Variable Patterns? (Inferential Statistics, Clustering)
H0₁
Difference: Method Pattern Profiles? (Inferential Statistics, Clustering)
H0₂
Difference: Program Construction Sequence? (Sequential Pattern Mining)
H0₃
Data Source
Artifacts
Analysis
Results
Differences between Novices and Experts concerning the Use of Construction Patterns
RQ4
H0₁₊₂
sign. differences between methods
H0₃
Novices construct loop after data
H0₁₊₂
specific construction is detectable
H0₃
Experts construct loop before data

Fig. 5.1: Visualization of the comparison study methodology.

ences in the order of program construction sequences that are typed and labeled by micropatterns. Following, I describe the three null hypotheses.

$H0_1$. Comparing novice programmers and expert programmers, there is no significant difference in micropattern usage and variable construction pattern usage **for individual variables**.

In the context of the first null hypothesis, the underlying question that I aim to evaluate is: *How well can variable construction pattern usage of single variables be used to explain differences in programming skills?* This null hypothesis is evaluated by using inferential statistics and machine learning to assess the significance of differences.

$H0_2$. Comparing novice programmers and expert programmers, there is no significant difference in micropattern usage **on the level of methods**.

In the context of the second null hypothesis, the underlying question that I aim to evaluate is: *How well can variable construction pattern usage, aggregated on methods, be used to explain differences in programming skills?* This null hypothesis is also evaluated by using inferential statistics and machine learning to assess the significance of differences.

$H0_3$. Comparing novice programmers and expert programmers, there is no substantial difference in micropattern usage regarding the **program construction sequence**.

In the context of the third null hypothesis, the underlying question that I aim to evaluate is: *How well can recorded sequences of program increments, typed by syntactic and semantic features, be used to explain differences in programming skills?* This null hypothesis is evaluated by using frequent sequences and interestingness measures. I thereby assess differences in the construction order of control variables (constituting loop constructs and conditional constructs) and data variables dependent on the respective control structure. However, this assessment approach also necessitates forgoing the assessment of significant differences – and instead assessing substantial differences in frequent construction orders.

The results of the comparative study encompass refuting statements to the null hypotheses, proven either by inferential statistics or by substantially different frequent sequences and interestingness measures, or acknowledging statements to the null hypotheses. Figure 5.1 provides a glimpse of the results. A summary of all results is given in Section 5.6. The results are combined to provide a qualified answer to research question RQ4. Following on the mixed methods study and respective results and answers to research question RQ3, this comparative study provides quantitatively proven results in addition to the previous qualitative results, completing the means of argumentation in order to answer the main research question of this thesis.

5.1.2 Methodology to Collect and Analyze Data

The learning analytics methodology and the example problems used in this comparative study are both identical to the mixed methods study, initially described in Section 4.4.1. I give a brief re-introduction in this subsection.

To record the program construction sequences, the instrumented `IntelliJ` plugin was used to capture `Java` program construction sequences, which are stored at the data collection server `trackserver`. The raw data of this study is openly available [KWB20].

I utilize my software package `pattern-browser`[1] to semi-automatically process the captured program construction data on *key-stroke* granularity. A fully automated process is not possible because of the used `Gumtree` algorithm that incorrectly maps changed parts in some circumstances, which is needed to locate the changed lines in subsequent program versions. The raw data is of the same form compared to the data used in the mixed methods study (that was used to visualize variable-specific *construction flowlines*). For this comparative study, additional `Python` and `R` code was developed to preprocess the data, with the main feature being the extraction of typed and labeled sequential *program increments* on the level of variables used to assess differences in the construction order (Section 5.4).

The example problems again encompass the warm-up method inspired by a *Reversing* problem showcased by Lister [Lis11b, p. 15] (Figure 4.15) and the method `convertBinaryArray` to convert an input array with binary data to decimal and hexadecimal numbers (Figure 4.3).

5.1.3 Participant Cohorts

Due to the *COVID-19* pandemic taking place during the last 15 months of research for this thesis, in-person recruitment of study participants was hardly possible in Austria for this comparative study. This applies to both students and professional programmers. I therefore followed two strategies to gather data for this comparative study.

The first strategy was to use the cohort of the mixed methods study, as this data was not previously used to assess differences with inferential statistics. This strategy provides a basic cohort size of 22 (under)-graduate students of two courses held at the University of Klagenfurt, recruited from courses on software project management

[1] Link to the repository: `https://gitlab-iid.aau.at/seqtrex/pattern-browser`

Tab. 5.1: Overview of data used to assess differences in the usage of micropatterns and variable construction patterns.

	Student	***Professional***	***Sum***
Cohort Size	27	12	39
Number of Methods	43	24	67
MoveElements: Methods	21	9	30
Control Variables	46	19	65
Data Variables	3	2	5
ConvertBinary: Methods	22	15	37
Control Variables	54	29	83
Data Variables	48	30	78

and on object-oriented modeling and implementation with `Java`, and 9 professional programmers recruited from a software company in Austria (with about 50 software developers at the time of recruitment).

The second strategy was to pursue online recruitment by sending out instructional mails to eligible groups of programmers. As an incentive to participate in the study, I offered an online voucher upon successful completion to potential participants, regardless of the correctness of the programs. I distributed similar instructional mails to undergraduate students of all practical courses that deal with programming in the study program of *Applied Informatics* at the University of Klagenfurt. In total, I distributed mails to 16 distinct courses, with a total of 497 students registered in all those courses (which could be overlapping). Unfortunately, I only received implementations from 5 students, increasing the cohort size of (under)-graduate students to 27.

I also distributed similar instructional mails to professional programmers of another Austrian software company (with about 20 professional programmers at the time of recruitment) and to two research labs (also with about 20 professional programmers). In total I received implementations from 3 professionals, increasing the cohort size of professional programmers to 12.

Table 5.1 provides an overview of the resulting cohort of students and professional programmers and the resulting raw data of methods and variables. With the students forming the group of *novice programmers* and the professionals forming the group of *expert programmers*, significant and substantial differences in the program construction sequences of these two groups are assessed and compared.

5.2 Comparison of Pattern Usage – Individual Variables

In this section I describe the data, analysis, and results of my investigation to uncover whether **variable construction pattern** usage is indicative of a certain level of programming skills, specifically focusing on the level of individual variables. I investigate whether there are substantial differences between the pattern usage of novices and expert programmers during program construction regarding individual variables. The corresponding null hypothesis $H0_1$ is:

$H0_1$. Comparing novice programmers and expert programmers, there is no significant difference in micropattern usage and variable construction pattern usage **for individual variables**.

I describe the data in Section 5.2.1 and present two analysis types and results in Section 5.2.2 (inferential statistics to uncover significant differences) and in Section 5.2.3 (k-means clustering to uncover groups of pattern usage). A summary of all results and a contextualization with the research question `RQ4` of the thesis is carried out in Section 5.6.

5.2.1 Patterns: Data Preparation

For the comparisons covered in this section, the recorded program construction sequences are semi-automatically transformed into a sequence of compilable *intermediate programs* and corresponding *program increments* that represent the sequential changes between the compilable programs. An overview of the number of methods and variables of each type are given in Table 5.1.

Next, to generate the data for the comparisons on the level of individual variables, the variables are manually labeled with micropatterns according to the last compilable program state, the `end program`. In other words, variables are labeled with all micropatterns of the respective type (control or data labels) that characterize the variable construction regarding the `end program`. The set of labeled micropatterns dictates the single variable construction pattern that characterizes the variable construction. The resulting data represents a *multilabel* classification for micropatterns and a *multiclass* classification for variable construction patterns. Both are represented with binary values: 1 for all applicable micropatterns and the single variable construction pattern, respectively, and 0 for all others.

5.2.2 Patterns: Comparison with Inferential Statistics

The first comparison to evaluate differences in pattern usage between novice and expert programmers, on the level of individual variables, is to assess differences with

Tab. 5.2: Significant differences (U-test, $p < .05$) regarding micropattern usage, based on individual variables.

Micropatterns	***CCP***	***CLC***	***CL***	***CC***	***DCD***	***DLD***	***DSSU***
All Methods						**.04**	
All Methods Control					—	—	—
All Methods Data	—	—	—	—		**.04**	
MoveElements						**.03**	
MoveElements Control					—	—	—
MoveElements Data	—	—	—	—			
ConvertBinary							
ConvertBinary Control					—	—	—
ConvertBinary Data	—	—	—	—			
Between Methods	**< .01**	**< .01**	**< .01**	**< .01**	**< .01**	**< .01**	**< .01**
Between Methods Control	**.03**			**< .01**	—	—	—
Between Methods Data	—	—	—	—			

inferential statistics. Due to the discrete, binary nature of the data and the distribution not following a normal distribution, I choose to employ the nonparametric *Mann-Whitney U test* to assess significant differences of micropattern and variable construction pattern usage.

I employ two test strategies to assess differences. The first strategy is to compare the micropattern usage between novice and expert programmers. The second strategy, acting as a control test regarding the usefulness of micropatterns, is to compare the micropattern usage between the two implemented classes (`MoveElements` vs. `ConvertBinary`), regardless of programming skill level. The first strategy is additionally multiplexed to assess differences in variables for specific classes, testing both classes together and separately. For both strategies, I assess differences for: i) all variables, ii) only control variables, and iii) only data variables.

Starting with the micropattern usage, I report all significant differences ($p < .05$) in Table 5.2. Regarding control micropatterns, there are no significant differences between novice and expert programmers. Regarding data micropatterns, there are significant differences between novice and expert programmers in the use of *loop-dependent* data variables (`DLD`). For expert programmers, a higher fraction of data variables are assigned to this micropattern label (a mean of 0.36 compared to the mean of 0.23 for novices). This difference is measurable considering both methods, and specifically in implementations of the class `MoveElements`.

Next is the evaluation of the control test strategy. Significant differences between variables of the two implemented classes are found for all micropatterns other than the data micropatterns *redefined* (`DR`) and *self-defining* (`DSD`).

Concerning the variable construction pattern usage, I report all significant dif-

Tab. 5.3: Significant differences (U-test, $p < .05$) regarding variable construction pattern usage, based on individual variables. All control patterns with significant differences are reported.

	CCP	***CCP***	***CCP***	***CCP***	***CCP***
		CLC			***CLC***
	CL			***CL***	
Patterns	***CC***	***CC***	***CC***		
All Methods					
All Methods Control					
MoveElements		—	—		
MoveElements Control		—	—		
ConvertBinary					
ConvertBinary Control					
Between Methods				**< .01**	**< .01**
Between Methods Control	**.04**	**.02**	**.02**	**.03**	**.01**

Tab. 5.4: Significant differences (U-test, $p < .05$) regarding variable construction pattern usage, based on individual variables. All data patterns with significant differences are reported.

	DCD	***DCD***			***DCD***	
		DLD	***DLD***	***DLD***		
	DR					
	DSD	***DSD***		***DSD***	***DSD***	
Patterns			***DSSU***			***DSSU***
All Methods		**.01**				
All Methods Data		**.02**				**.04**
MoveElements		—	—		—	
MoveElements Data		—	—	—	—	
ConvertBinary	—	**.02**				
ConvertBinary Data	—	**.02**				
Between Methods			**.01**	**< .01**	**.02**	
Between Methods Data	**< .01**					

ferences ($p < .05$) in Table 5.3 for control patterns and Table 5.4 for data patterns. Regarding control patterns, there are again no significant differences between novice and expert programmers. Regarding data patterns, there are significant differences for two sets of data micropatterns: {*conditionally-dependent*, *loop-dependent*, *self-defining*} (experts have a higher mean usage of 6% compared to the mean of 0.7% for novices) and {*single-scope-use*} (novices have a higher mean usage of 4% compared to no usage by experts). The first difference shows that experts use significantly more variables in a combination of loop dependencies and conditional dependencies. The second difference shows that some novices construct variables that are only used in a simple method control context (`DSSU`).

Next is the evaluation of the control test strategy. There are significant differences between the control patterns of the two implemented classes. Three of those apply to control patterns, including the micropattern *conditional* (`CC`) – a control structure commonly used by the programmers to implement the `ConvertBinary` methods, but not for the `MoveElements` methods. The other two significant differences apply to control patterns representing loop variables with micropatterns *loop* (`CL`) or *loop-condition* (`CLC`) that are also used as computational variables with the micropattern *computational* (`CCP`). For the implementation of `MoveElements` methods, these two control patterns constitute about 86% of all control patterns, while they only constitute about 29% of all control patterns for the implementation of `ConvertBinary` methods. The diverse nature of used control structures of `ConvertBinary` implementations is hereby captured.

There are also significant differences between data patterns of the two implemented classes. Again, two of those apply to data patterns, including the micropattern *conditionally-dependent* (`DCD`), which characterizes the use of a conditional construct and dependent data variables – more frequently used in `ConvertBinary` method implementations. The other two significant differences apply to data patterns representing loop-dependent (`DLD`) variables, additionally either single-scope-use (`DSSU`) or self-defining (`DSD`). The former is only used in `ConvertBinary` method implementations. The latter has a significantly higher usage fraction for `ConvertBinary` method implementations (with a mean of about 15%, compared to about 1% for `MoveElements` methods). While not as pronounced as with control structures, differences in the required algorithmic structures of these two classes are detectable with variable construction patterns.

Summarizing, these results suggest that differences between varying algorithms are more pronounced compared to differences that could be attributed to varying levels of programming skills. Paraphrased, expert programmers do, for the most part, not use significantly different micropatterns when constructing their solutions. Rather, the *algorithmic affordance* of the implementation to be constructed dictates the mixture of micropatterns required for the variables. However, the null hypothesis

Tab. 5.5: Results of k-means clustering with all variables and micropatterns, tabulated by programmer experience and per method.

	Clusters All Variables							
	Control Vars				***Data Vars***			
Group	***1***	***2***	***3***	***4***	***1***	***2***	***3***	***4***
Student	3	48	47	2	35	0	0	16
Professional	2	24	21	1	21	0	0	11
Method	***1***	***2***	***3***	***4***	***1***	***2***	***3***	***4***
MoveElements	0	33	32	0	3	0	0	2
ConvertBinary	5	39	36	3	53	0	0	25

Tab. 5.6: Results of k-means clustering with specific types of variables construction patterns, tabulated by programmer experience and per method. Tables on the left include clusters with control patterns only, tables on the right include clusters with data patterns only.

	Control Vars		
Group	***1***	***2***	***3***
Student	36	26	38
Professional	16	16	16
Method	***1***	***2***	***3***
MoveElements	29	5	31
ConvertBinary	23	37	23

	Data Vars				
Group	***1***	***2***	***3***	***4***	***5***
Student	22	13	6	9	1
Professional	11	12	0	4	5
Method	***1***	***2***	***3***	***4***	***5***
MoveElements	3	1	1	0	0
ConvertBinary	30	24	5	13	6

$H0_1$ can be refuted in one point: expert programmers tend to have a significantly higher portion of data variables as *loop-dependent* variables. Moreover, specific program constructions are discernible between novice and expert programmers – for example, the construction of data variables in a combination of loop dependencies and conditional dependencies.

5.2.3 Patterns: Comparison with k-means Clustering

The second comparison to evaluate differences in pattern usage between novice programmers and expert programmers, on the level of individual variables, is to assess whether there are groups of programmers that mostly use variables with the same characteristics. To this end, I employ k-means clustering with the elbow and silhouette methods to assess the appropriate number of clusters. I employ two clustering approaches: i) clustering of all variables using micropatterns, ii) clustering of variables per type using variable construction patterns.

For the first clustering approach using the nine micropatterns, the appropriate number of clusters without too much segmentation is *four*. The results of the first clustering approach are presented in Table 5.5. The results are tabulated by the type of variable (control and data variables) and either the programmer group (student vs. professional, top table) or the class (`MoveElements` vs. `ConvertBinary`, bottom table).

Looking at the results of clustering with the nine micropatterns, no distinction between novice and expert programmers can be made. However, two clusters capture control variables (clusters `2` and `3`), while two clusters capture data variables (clusters `1` and `4`). Such a clustering was expected as the different types of variables are non-overlapping.

For the second clustering approach, I investigate clusters of pattern usage specifically for each type of variable. I employ k-means clustering with the variable construction patterns for each type. In total, there are 10 distinct control patterns and 13 distinct data patterns that serve as the dimensions for clustering. The appropriate number of clusters without too much segmentation is *three* for control variables and *five* for data variables. The results of the second clustering approach are presented in Table 5.6. The results are tabulated by the type of variable (clusters for control variables on the left and clusters for data variables on the right) and either the programmer group (student vs. professional, top table) or the class (`MoveElements` vs. `ConvertBinary`, bottom table).

The clusters of control variables provide no distinction between the level of programming skills or between the implemented methods. The clusters of data variables produce two clusters that differentiate between the level of programming skills. Cluster 3 captures six student-constructed data variables that are characterized by the sole use of the variable in a simple method control context (`DSSU`) – a characteristic already uncovered by the inferential statistics. Cluster 5 captures six data variables, five of which have been constructed by experts, that are characterized by the micropattern set {*conditionally-dependent*, *loop-dependent*, *self-defining*}. This characteristic was also uncovered by the inferential statistics.

To summarize, k-means clustering with both micropatterns and variable construction patterns did not provide additional insights regarding the differences of novices and experts in pattern usage during program construction. Moreover, the resulting clusters did not differentiate between the *algorithmic affordance* of the two implemented classes.

5.3 Comparison of Pattern Usage – Methods

In this section, I describe the data, analysis, and results of my investigation to uncover whether **variable construction pattern** usage is indicative of a certain level of programming skills, specifically focusing on the level of methods. I investigate whether there are substantial differences between the pattern usage of novices and expert programmers during program construction regarding all variables of each method. The corresponding null hypothesis $H0_2$ is:

$H0_2$. Comparing novice programmers and expert programmers, there is no significant difference in micropattern usage **on the level of methods**.

I describe the data in Section 5.3.1 and present two analysis types and results in Section 5.3.2 (inferential statistics to uncover significant differences) and in Section 5.3.3 (k-means clustering to uncover groups of pattern usage). A summary of all results and a contextualization with the research question `RQ4` of the thesis is carried out in Section 5.6.

5.3.1 Method Profiles: Data Preparation

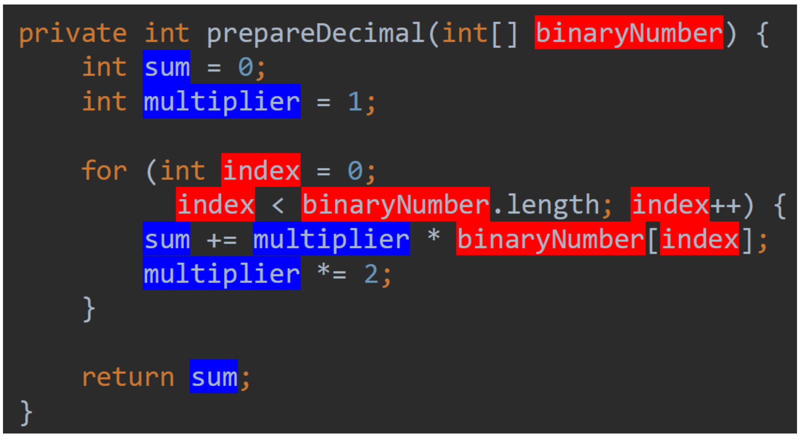
For the comparisons covered in this section, the *multilabel* classification of micropatterns for each variable, introduced in Section 5.2.1, is aggregated on the level of methods. Table 5.1 of the previous section shows the basic data of the number of methods and variables of each type.

The automated process of aggregation has been applied in the following way. Starting from the variables that are manually labeled with micropatterns according to the respective `end program`, all variables for each method are collected. Next, the absolute number of occurrences of each micropattern is computed. Finally, the average value of occurrences of each micropattern is computed, taking into account the overall number of variables in the method. This process yields a number in the range of $[0, 1]$ for each of the nine micropatterns, characterizing the profile of the pattern usage for each method. I call this set of micropattern usage fractions the *method pattern profile*. An example of the aggregation process to create a *method pattern profile* is shown in Figure 5.2.

5.3.2 Method Profiles: Comparison with Inferential Statistics

The first comparison to evaluate differences in pattern usage between novice and expert programmers, aggregated on the level of methods, is to assess differences with inferential statistics. While the *method pattern profiles* consist of continuous data, an investigation into the density distributions of the average usage of the nine micropattern reveals that they do not follow a normal distribution. Because of this, I employ the nonparametric *Mann-Whitney U test* to assess significant differences in the average usage of the micropattern aggregated on the level of methods.

I follow two test strategies to assess differences. The first strategy is to compare the average micropattern usage between novice and expert programmers. The second strategy, acting as a control test, is to compare the average micropattern



```
private int prepareDecimal(int[] binaryNumber) {
    int sum = 0;
    int multiplier = 1;

    for (int index = 0;
          index < binaryNumber.length; index++) {
        sum += multiplier * binaryNumber[index];
        multiplier *= 2;
    }

    return sum;
}
```

(a) Example method with two control and two data variables

Control Micropatterns	***index binaryN***
CLC 1/4	CC 0/4
CL 1/4	CCP 2/4

Method Pattern Profile			
CLC	***CL***	***CC***	***CCP***
0.25	0.25	0	0.5

Data Micropatterns	***sum mult***
DLD 2/4	DSD 2/4
DCD 0/4	DSSU 1/4
DR 0/4	

DLD	***DCD***	***DR***	***DSD***	***DSSU***
0.5	0	0	0.5	0.25

(b) Micropatterns of variables

(c) Resulting method pattern profile

Fig. 5.2: Example aggregation process to create a *method pattern profile*. Starting from an example method (a), first the absolute number of variables for each micropattern are computed (b). Finally, the average value is computed, taking into account the overall number of variables in the method (c).

Tab. 5.7: Significant differences (U-test, $p < .05$) regarding average method profiles, based on all variables for each method.

Method Profiles	***CCP***	***CLC***	***CL***	***CC***	***DCD***	***DLD***	***DSD***	***DSSU***
All Methods		**.04**	**.04**					
MoveElements						**.03**	**< .01**	
ConvertBinary			**.01**					
Between Methods	**< .01**	**< .01**	**< .01**	**< .01**	**< .01**	**< .01**		**< .01**

usage between the two implemented classes (`MoveElements` vs. `ConvertBinary`), regardless of programming skill level. The first strategy is additionally multiplexed to assess differences in variables for specific classes, testing both classes together and separately.

Reporting the results for the first testing strategy, comparing between novice and expert programmers, I report all significant differences ($p < .05$) in Table 5.7. Regarding control micropatterns, two significant differences are found. For both loop-related control micropatterns, novice programmers have a significantly higher average usage compared to expert programmers (0.36 vs. 0.28 for *loop-condition*, 0.38 vs. 0.29 for *loop*). These differences are measurable considering both methods, and the latter is also found for `ConvertBinary` implementations.

Regarding data micropatterns, two significant differences are found when analyzing `MoveElements` implementations. However, as there are only five data variables in total for these implementations, these results are hardly generalizable. For the micropattern *loop-dependent*, experts have a significantly higher average usage (0.07 vs. 0). Students have a significantly higher average usage for the micropattern *self-defining* (0.41 vs. 0.19).

Turning to the control test strategy, the comparison between method implementations of the two classes, again highly significant differences are found for all micropatterns other than the two data micropatterns *redefined* (`DR`) and *self-defining* (`DSD`).

Summarizing, the null hypothesis $H0_2$ can be refuted in one point: there are significant differences in the average usage of loop-related control micropatterns on method level. However, these differences do not constitute substantially different ways of program construction but characterize variants of how to construct similar algorithms. These results confirm the interpretation given in Section 5.2.2 that expert programmers do, for the most part, not use significantly different micropatterns when constructing their solutions. The significant differences between implementations of the two different classes can be attributed to the *algorithmic affordance*, dictating the basic number of variables and their types – which in turn influences the basic structure of the *method pattern profiles* used to assess the differences.

Tab. 5.8: Results of k-means clustering with all method pattern profiles, tabulated by programmer experience and per method.

Clusters All Methods

Group	***1***	***2***	***3***	***4***	***5***
Student	0	18	9	9	8
Professional	2	6	4	4	8

Method	***1***	***2***	***3***	***4***	***5***
MoveElements	0	24	2	5	0
ConvertBinary	2	0	11	8	16

Tab. 5.9: Results of k-means clustering with method pattern profiles of specific classes, tabulated by programmer experience. Table on the left includes clusters of the class `ConvertBinary`, table on the right includes clusters of the class `MoveElements`.

Clusters ConvertBinary

Group	***1***	***2***	***3***
Student	0	7	15
Professional	2	4	9

Clusters MoveElements

Group	***1***	***2***	***3***	***4***	***5***	***6***
Student	2	1	0	1	2	16
Professional	2	2	1	1	0	3

5.3.3 Method Profiles: Comparison with k-means Clustering

The second comparison to evaluate differences in pattern usage between novice programmers and expert programmers, aggregated on the level of methods with *method pattern profiles*, is to assess whether groups of programmers construct methods with the same characteristics. To this end, I employ k-means clustering with the elbow and silhouette methods to assess the appropriate number of clusters. I employ two clustering approaches: i) clustering of all methods combined, ii) clustering of methods for each specific class implementation.

For the first clustering approach of all combined methods, the appropriate number of clusters without too much segmentation is *five*. The results of the first clustering approach are presented in Table 5.8. The results are tabulated by the programmer group (student vs. professional, left table) and by the class (`MoveElements` vs. `ConvertBinary`, right table).

Looking at the clustering results with the average method profiles, only cluster `1` seems to capture unique micropattern usage profiles of expert programmers. However, upon inspection, these method implementations are non-functional and exclusively use control variables with the micropatterns *computation* (`CCP`) and *conditional* (`CC`). The other clusters provide no distinction between novice and expert programmers.

The clusters do, however, provide a distinction between method implementations. Cluster `2` exclusively captures average method profiles of `MoveElements` implementations, characterized by high fractions of control micropatterns (means of 1.00 for *computation*, 0.49 for *loop-condition*, and 0.51 for *loop*). All variables in this cluster are control variables that are used for *computation*. Cluster 5 exclusively captures average method profiles for ConvertBinary implementations, characterized by lower control micropatterns (means of 0.38 for *computation*, 0.21 for *loop-condition*, and 0.19 for *loop*) and a high fraction of *loop-dependent* data variables (with a mean of 0.57). Half of all variables in this cluster are loop-dependent data variables, accompanied by control variables of different micropatterns.

For the second clustering approach, only using methods for each specific class implementation, the appropriate number of clusters is *three* for ConvertBinary implementations and *six* for MoveElements implementations. The results of the second clustering approach are presented in Table 5.9. The results are tabulated by the programmer group (students vs. professional), with the clusters for ConvertBinary implementations on the left, and clusters for the MoveElements implementations on the right.

Looking at the results for the class ConvertBinary, cluster 1 again captures the non-functional method implementations explained above. The other two clusters do not provide any distinction between the programmer groups. The results for the class MoveElements provide many small clusters, each capturing specific profiles. Cluster 3 includes a data variable with the micropattern *redefined* and cluster 5 includes variables of all control micropatterns (suggesting a solution that includes a conditional control variable and unique variables for loop initialization/update and loop condition). Altogether, no meaningful distinction between novice and expert programmers can be made with these clusters.

To summarize, k-means clustering with *method pattern profiles* did not provide additional evidence to refute the null hypothesis $H0_2$. There are no clusters of specific micropattern usage, aggregated on the level of methods, that are attributable to different levels of programming skills. Compared to the clustering approach of individual variables, the clustering approach described in this section differentiates between the varying *algorithmic affordance* of the two implemented classes.

5.4 Comparison of Pattern Usage – Construction Sequences

In this section, I describe the data, analysis, and results of my investigation to uncover whether **variable construction pattern** usage is indicative of a certain level of programming skills, specifically focusing on properties of sequential program construction. I investigate whether there are substantial differences between the pattern usage of novices and expert programmers during program construction regarding the typed sequences of pairs of variables and of all variables of each method. The key interactions I investigate are sequential program changes with a focus on the order of control variables (the construction of loop structures and conditional structures) and data variables dependent on these control structures. The corresponding null hypothesis $H0_3$ is:

$H0_3$. Comparing novice programmers and expert programmers, there is no substantial difference in micropattern usage regarding the **program construction sequence**.

I describe the data in Section 5.4.1 and present two analysis types and results in Section 5.4.2 (frequent sequential rules regarding the construction of pairs of variables) and in Section 5.4.3 (frequent sequential rules regarding the construction of all variables of each method). A summary of all results and a contextualization with the research question `RQ4` of the thesis is carried out in Section 5.6.

5.4.1 Sequences: Data Preparation

I employ the SPADE algorithm [Zak01] to mine frequent sequences that capture the order of variable changes in program construction sequences. My rationale is that frequent and strong sequential association rules can assess whether there are substantial differences in the order of program construction. Whether a particular rule is *strong* is evaluated with *interestingness measures.* Le and Lo have proven that the measure **odds ratio** [TKS04] performs best for the task of evaluating the *strongest* rules [LL15]. While the `R` package `arules`[2] provides a framework to calculate additional *interestingness measures*, they are not available for the `R` package `arulesSequences`[3] that I use to mine frequent sequences with SPADE.

However, Le and Lo also report that the standard measures of **support** and **confidence** are stronger than the measure of **lift** and are, for most of their tested configurations, not much worse compared to the best configuration of *interestingness measures* (**support** and **odds ratio**) [LL15]. Following this reasoning, I evaluate the *strength* of rules with the pair of **support** and **confidence**. Moreover, I apply the following heuristic when reporting frequent rules: for each evaluation, I report those rules that represent the maximum number of implemented methods per programmer group (**frequent** rules), interpreting the **support** as relative frequency. My approach to evaluating results and comparing frequent sequences between novice and expert programmers is to assess *exclusive* frequent sequences for each group. That is, I report all **strong** and **frequent** rules that are not in the intersection of rules, attributed to the respective group.

To transform the sequences of compilable *intermediate programs* and corresponding *program increments* into data used to mine frequent sequences, I developed a set of `Python` scripts (part of the `pattern-browser` package) that takes the raw data

[2] `https://rdrr.io/cran/arules/`

[3] `https://rdrr.io/cran/arulesSequences/`

Tab. 5.10: Overview of data used for sequential rule mining.

	ConvertBinary	***Students***	***Professionals***	***Sum***
(a)	***Cohort***	27	12	39
	Methods	22	15	37
(b)	***Variables***	83	78	161
	Control Vars	54	29	83
	Data Vars	48	30	78
(c)	***Variable Pairs***	237	119	356
	Control × Control	54	22	76
	Control × Data	129	67	196
	Data × Data	54	30	84

of the variable-specific *construction flowlines* as input and produces a sequence of typed, labeled changes for each variable of a method. Each *program increment* is transformed into a set of variables changes. I then manually re-labeled the changes to accurately capture the detailed micropatterns, outlined below. At last, these finished individual variable construction sequences are automatically combined into the following types of program construction sequences: a) pair-wise variable construction sequences for all non-overlapping pairs of a method, and b) method construction sequences that aggregate all variable construction sequences for a method. The sequences are further preprocessed, **combining** multiple *program increments* that consist of the same set of changes to a single *program increment*.

In terms of transaction datasets, each (individual or pair-wise) variable construction sequence and each method construction sequence represents a single **session**. The sequential *program increments* represent the **events** in that session. The set of typed, labeled changes to variables represents the **items** for each event.

Table 5.10 provides an overview of the number of *method* construction sequences (`a`), individual variable construction sequences that represent the raw data for pairs of variable construction sequences (`b`), and the pairs of variable construction sequences that are computed from non-overlapping pairs of variable construction sequences for each method (`c`). Of note is that only implementations for the class `ConvertBinary` are included in this analysis. Implementations for the class `MoveElements` only have a small number of data variables (five in total), invalidating the analysis of the order of changes between control and data variables.

The individual variable changes are semi-automatically typed and labeled according to two dimensions that are shown in Figure 5.3. The first dimension (`a`) is a typed, hierarchical granularity level to assign a single type to each specific change. The granularity levels are represented as a textual label with the following hierarchy:

Level	***Types***		
(Prefix) L0	`C` control	`D` data	
(Suffix) L1	`a` add	`m` modify	`d` delete
(Suffix) L2	`d` definition	`u` use	
(Suffix)	`ad` add-def	`md` mod-def	`dd` del-def
L3 (L1+L2)	`au` add-use	`mu` mod-use	`du` del-use

(a) Granularity levels for typed changes

Detailed Control Micropatterns	
loop-condition	*loop-condition* `CLC`
loop	*loop-initialization* `CLI` *loop-condition* `CLC` *loop-update* `CLU`
conditional computation	*conditional* `CC` *computation* `CCP`

(b) Detailed control micropatterns

Detailed Data Micropatterns	
loop-dependent conditionally-dependent	*loop-dependent* `DLD` *conditionally-dependent* `DCD`
redefined	*redefined-def1* `DR1` *redefined-def2* `DR2`
self-defining single-scope-use	*self-defining* `DSD` *single-scope-use* `DSSU`

(c) Detailed data micropatterns

Fig. 5.3: Type and label categories to annotate the program construction sequences. (`a`) shows the hierarchical granularity levels for typed changes. (`b`) and (`c`) show detailed micropatterns used to label changes in the sequential *program increments*.

`L0`: The prefix label part, distinguishing between **C**ontrol and **D**ata variable changes.

`L1`: A suffix to distinguish between changes that **a**dd, **m**odify, or **d**elete variables.

`L2`: A suffix to distinguish between changes to **d**efinitions or **u**ses of variables.

`L3`: A suffix, combining the previous granularity levels `L1` and `L2`.

Note that, in this hierarchy, the suffix levels `L1`–`L3` are mutually exclusive, i.e., only one suffix level can be present.

The second dimension (Figure 5.3 `b`, `c`) comprises the detailed micropatterns of variable changes. Two micropatterns have been refined to more accurately capture the program construction. The first one is the control micropattern *loop*, now split into three detailed micropatterns (*loop-initialization*, *loop-condition*, *loop-update*) to attribute changes to specific parts of loop constructs. The second one is the data micropattern *redefined*, now split into two detailed micropatterns (*redefined-def1* and *redefined-def2*) to track the construction of this specific micropattern from the first definition to the second (re)-definition.

Combining the type and label categories, a single variable change can be categorized by multiple categorizations, e.g., when a variable is defined and used in the same statement. Every single categorization is made up of `'prefix + micropattern + suffix'`.

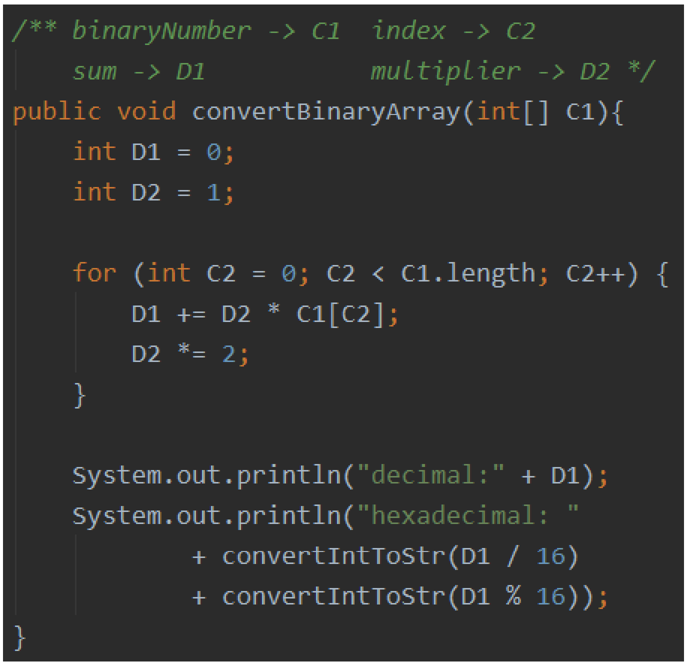
Finally, Figure 5.4 provides an example of a typed, labeled method construction sequence that includes all changes for each variable construction sequence. This sequence is typed according to granularity level `L3`, which is the default granularity level. The other lower granularity levels are produced by stripping away the suffix information of level `L2` (second affix letter: **d**efinition, **u**se) to arrive at level `L1`, stripping away the suffix information of `L1` (first letter: **a**dd, **m**odify, **d**elete) to arrive at level `L2`, or stripping away suffix information of levels `L1` and `L2` to arrive at level `L0`.

5.4.2 Sequences: Comparison of Pairwise Variable Construction

I first investigate frequent sequences of the pairwise variable construction sequences. The transaction data for this evaluation consists of 237 non-overlapping pairwise variable construction sequences of novice programmers with 1765 **events** and 119 sequences of expert programmers with 912 **events** (Table 5.10 `c`). Two evaluations of increasing detail are made: first simple typed pairwise construction sequences, and second pairwise construction sequences typed and labeled with detailed micropatterns. For each level of granularity, frequent sequences are mined separately with SPADE.

For the first evaluation, I consider typed pairwise variable construction sequences that are not labeled with the detailed micropatterns. I call this typing *simple* as it only consists of the prefix and the suffix of granularity levels. The **strongest** and most **frequent** rules of this evaluation are reported in Table 5.11 – only rules for granularity levels `L0` and `L1` offer meaningful sequences to evaluate the order of control and data changes. All **frequent** rules of this evaluation offer a low fraction of **support**, only accounting for three to four student-written methods and two to three professional-written methods.

Interpreting the frequent sequences of granularity level `L0`, students frequently focus on changes related to control structures while professionals have a balanced focus on changes related to control structures and data variables. The frequent sequences of granularity level `L1` could hint at students struggling with logical errors – as they frequently modify and delete control structures. A frequent sequence rule of professionals' program construction shows modifications of data variables after constructing control structures. Altogether, this evaluation hardly provides any insight in the order of construction of control structures and dependent data variables.



```
/** binaryNumber -> C1  index -> C2
    sum -> D1            multiplier -> D2 */
public void convertBinaryArray(int[] C1){
    int D1 = 0;
    int D2 = 1;

    for (int C2 = 0; C2 < C1.length; C2++) {
        D1 += D2 * C1[C2];
        D2 *= 2;
    }

    System.out.println("decimal:" + D1);
    System.out.println("hexadecimal: "
            + convertIntToStr(D1 / 16)
            + convertIntToStr(D1 % 16));
}
```

(a) Example implementation by an expert

Example Method Construction Sequence

		C1	C2	D1	D2		
1	[	C1LCau,	C2LIad, C2LCau, C2LUad, C2LUau			]	1
2	[			D1ad		]	2
3	[				D2ad	]	3
4	[	C1CPau,	C2CPau,	D1LDad, D1LDau, D1SDad, D1SDau,	D2LDau	]	4
5	[				D2LDad, D2LDau, D2SDad, D2SDau, D2SSUau	]	5
6	[			D1au		]	6
7	[			D1au		]	7
8	[			D1au		]	8
9	[			D1mu		]	9
10	[	C1CPau,		D1ad, D1mu		]	10
11	[			D1dd, D1du		]	11

(b) Corresponding method construction sequence typed with granularity level L3

Fig. 5.4: Example method construction sequence that is the input for mining frequent sequences with SPADE [Zak01]. (a) shows an expert implementation, with variable names numbered by occurrence and type. (b) shows the corresponding construction sequence, typed and labeled with granularity level L3.

Tab. 5.11: Strongest association rules of frequent sequences, mined with a maximum gap of 1 from full method construction sequences with numbered variables (Table 5.10 (a,b)) and for changes labeled with simple change labels (Control, Data).

	Pairs of Variable Sequences (Simple)			
Level 0	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{C}] ⇒ [{C, D}]	0.16	0.21	Focus on changes related to control structures
	[{C} → {C}] ⇒ [{D}]	0.15	0.82	
	[{D} → {C}] ⇒ [{C}]	0.15	0.42	
Prof.	[{D} → {C, D}] ⇒ [D]	0.18	0.72	Balanced focus of changes related control structures and data
	[{C} → {D} → {C, D}] ⇒ [{D}]	0.15	0.69	
	[{C} → {D} → {D}] ⇒ [{D}]	0.15	0.62	
	[{D} → {D} → {C}] ⇒ [{D}]	0.13	0.73	

Level 1	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{Cm}] ⇒ [{Cm}]	0.16	0.38	Frequent control structure modifications could hint at logical errors
	[{Cm}] ⇒ [{Cd}]	0.16	0.37	
	[{Ca} → {Cm}] ⇒ [{Cd}]	0.14	0.36	
	[{Ca} → {Ca}] ⇒ [{Cm}]	0.14	0.54	
	[{Ca} → {Cm}] ⇒ [{Cm}]	0.14	0.35	
Prof.	[{Ca} → {Da}] ⇒ [{Dm}]	0.20	0.44	Modification of data statements after constructing control structure

Tab. 5.12: Strongest association rules of frequent sequences, mined with a maximum gap of 1 from pairs of variable construction sequences (Table 5.10 (c)) and for changes labeled with micropatterns.

	Pairs of Variable Sequences (Micropatterns)			
Level 0	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{CLC}] ⇒ [{CC}]	0.16	0.25	Loop constructed **before** conditional constructs
	[{CC}] ⇒ [{CCP}]	0.15	0.42	
	[{CLC}] ⇒ [{C}]	0.15	0.22	
	[{D}] ⇒ [{CLC}]	0.14	0.21	Loop condition **after** generic data changes
Prof.	[{DLD}] ⇒ [{DCD}]	0.19	0.29	Different orders of data change
	[{DCD}] ⇒ [{DLD}]	0.17	0.49	

Level 1	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student				No strong rules
Prof.	[{CLCa}] ⇒ [{Da}]	0.18	0.28	Loop condition **before** generic data addition

For the second evaluation, I consider typed and labeled pairwise variable construction sequences. The **strongest** and most **frequent** rules of this evaluation are reported in Table 5.12. Again, only rules for granularity levels `L0` and `L1` offer meaningful sequences to evaluate the order of control and data changes. These rules offer similarly low fractions of **support**.

Interpreting the frequent sequences of granularity level `L0`, there are two sets of frequent rules that capture the order of variable construction in student-written methods. The first set of rules shows that students implement loop constructs **before** implementing other control structures (like conditional structures or the use of control variables in computations). The second set is a single rule that represents the order of constructing control structures and data variables. Students frequently construct a loop condition (`CLC`) **after** a generic data change (`D`, which could be a data variable definition). The set of frequent rules of professional programmers captures different orders of changes to data variables (specifically sequential changes to loop-dependent and conditionally-dependent data variables), showing that both orders are commonly used. Interpreting the frequent sequences of granularity level `L1`, there are no **strong** rules for students. One rule captures the order of variable construction in professional-written methods, showing that loop conditions are added (`CLCa`) **before** generic data additions (`Da`, which could be data variable definitions).

To summarize, the pairwise variable construction sequences, typed and labeled with detailed micropatterns, hint at a difference in the order of constructing loop control structures and data variables, comparing novice and expert programmers. While novices frequently construct the loop (specifically the loop conditional) **after** changes to data variables, experts frequently construct the loop (again specifically the loop conditional) **before** changes to data variables. No frequent sequences regarding the order of conditional structures and data variables are found, also due to the comparatively low usage of conditional structures in the example problem `ConvertBinary`. Didactical and practical implications of these findings are presented in Chapter 6.

5.4.3 Sequences: Comparison of Method Construction

Next, I investigate frequent sequences of the method construction sequences, aggregating all variable construction sequences for each method. The transaction data for this evaluation consists of 22 method construction sequences of novice programmers with 267 **events** and 15 sequences of expert programmers with 141 **events** (Table 5.10 `c`). I again report two evaluations of increasing specificity: first unnumbered method construction sequences that do not differentiate between changes to different variables in each *program increment*, and second numbered method construction sequences that do differentiate between changes to different variables. For each level of granularity, frequent sequences are mined separately with SPADE.

For the first evaluation, I consider aggregated method construction sequences, typed and labeled by detailed micropatterns, that contain all variable construction sequences of each respective method. The **strongest** and most **frequent** rules of this evaluation are reported in Table 5.13 – rules for each granularity level are reported. The **support** of these **frequent** rules is higher compared to the rules of pairwise sequences, accounting for a quarter to half of each group of programmers.

Interpreting the frequent sequences of granularity level `L0`, there is a set of rules for each group of programmers that captures the order of program construction with regard to loop constructs and dependent data variables. Students frequently, and with a **support** ≥ 0.50, construct their loop conditions (`CLC`), initializations (`CLI`), and updates (`CLU`) **after** generic data variable changes (`D`). For professional programmers, sequences of frequent rules with a **support** of 0.27 are found that capture the opposite order of program construction: the construction of loop initializations (`CLI`) **before** generic data changes (`D`), which lead to loop-dependent changes to data variables (`DLD`).

Interpreting the frequent sequences of granularity level `L1`, the relations identified for level `L0` can be further specified. Students frequently (with a **support** up to 0.50) add loop constructs (`CLCa`, `CLIa`) **after** adding data variable changes, while professional programmers add loop constructs (`CLIa`, `CLUa`, `CLCa`) **before** adding data variable changes (`Da`) and loop-dependent data changes (`DLDa`).

Interpreting the frequent sequences of granularity level `L2`, the relations identified for level `L0` can again be further specified. Students frequently construct loop constructs (`CLCu`, `CLId`) **after** data definitions (`Dd`), while professional programmers construct loop constructs (`CLCu`, `CLId`, `CLUd`, `CLUu`) **before** data definitions (`Dd`) and before loop dependent data definitions and data uses (`DLDd`, `DLDu`). Moreover, at level `L2`, a trivial construction order was captured in the students' construction sequences: the use of a control variable in conditional constructs (`CCu`, like *if* statements) **before** the construction of conditionally-dependent data definitions (`DCDd`).

Interpreting the frequent sequences of granularity level `L3`, the relations identified for level `L0` can be further specified for students. They frequently construct loop constructs (`CLCau`, `CLIad`) **after** adding data definitions (`Dad`). There are no captured **strong** rules for professional programmers.

Giving a summary of the findings so far, the frequent sequences found in aggregated method construction sequences prove that there are substantial differences in the sequential order of program construction of novice and expert programmers. While novices tend to implement the loop construct **after** a data definition (with a **support** of about 0.5 and a **confidence** of up to 0.6 for similar rules), experts tend to implement the loop construct **before** a data definition (with a **support** of about 0.5 and a **confidence** of up to 0.88 for similar rules).

For the second evaluation, I consider aggregated method construction sequences, *Tab. 5.13:* Strongest association rules of frequent sequences, mined with a maximum gap of 1 from full method construction sequences (Table 5.10 `(a,b)`) and for changes labeled with micropatterns.

	Method Construction Sequence (Micropatterns)			
Level 0	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{CLC}] ⇒ [{CCP}]	0.59	0.59	Loop construction **after** data change
	[{D}] ⇒ [{CLC,CLI}]	0.55	0.60	
	[{D}] ⇒ [{CLI}]	0.55	0.60	
	[{D}] ⇒ [{CLC,CLI,CLU}]	0.50	0.55	
	[{D} → {CLC}] ⇒ [{CCP}]	0.41	0.60	
Prof.	[{CLI} → {D} → {DSD,DLD} ... → {D}] ⇒ [{CCP}]	0.27	0.80	Loop construction **before** data change and loop-dependent change
	[{CLI} → {D} → {DSD} → {D}] ... ⇒ [{CCP}]	0.27	0.80	
	[{CLI} → {D} → {DLD} → {D}] ... ⇒ [{CCP}]	0.27	0.80	
Level 1	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{Da}] ⇒ [{CLCa}]	0.50	0.55	Loop construction **after** data change
	[{Da}] ⇒ [{CLIa,CLCa}]	0.45	0.50	
	[{Da}] ⇒ [{CLIa}]	0.45	0.50	
Prof.	[{CLIa,CLUa,CLCa} → {Da} ... → {DLDa,DSDa}] ⇒ [{Da}]	0.27	0.67	Loop construction **before** data change and loop-dependent change
Level 2	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{Dd}] ⇒ [{CLCu}]	0.55	0.60	Loop construction **after** data definition
	[{Dd}] ⇒ [{CLId}]	0.45	0.50	
	[{Dd}] ⇒ [{CLCu,CLId}]	0.45	0.50	
	[{DSDd,CCPu} → {CCu}] ... ⇒ [{DCDd,DSDd}]	0.27	1.00	Trivial construction order
	[{DSDd} → {CCu}] ⇒ [{DCDd}]	0.27	1.00	
Prof.	[{CLCu,CLId,CLUd,CLUu}] [... ⇒ [{Dd}]	0.53	0.73	Loop construction **before** data definition and loop-dependent change
	[{CLCu,CLId,CLUu} → {Dd}] ... ⇒ [{DLDd,DSDd}]	0.47	0.88	
	[{CLCu,CLId,CLUu} → {Dd}] ... ⇒ [{CCPu,DLDu}]	0.40	0.75	
Level 3	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{Dad}] ⇒ [{CLCau}]	0.45	0.50	Addition of data definition **before** loop construction
	[{Dad}] ⇒ [{CLCau,CLIad}]	0.41	0.45	
	[{Dad}] ⇒ [{CLIad}]	0.41	0.45	
Prof.				No strong rules

Tab. 5.14: Strongest association rules of frequent sequences, mined with a maximum gap of 1 from full method construction sequences with numbered variables (Table 5.10 `(a,b)`) and for changes labeled with micropatterns.

	Method Construction Sequence (Micropatterns, Numbered)			
Level 0	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{D1}] ⇒ [{C1LC}]	0.55	0.63	Loop construction **after** data change
	[{D1}] ⇒ [{C2LC}]	0.41	0.47	
	[{D1}] ⇒ [{C2LI}]	0.41	0.47	
	[{C1C,C2C}] ⇒ [{D1CD}]	0.27	1.00	Trivial construction order
	[{C2C}] ⇒ [{D1CD}]	0.27	0.86	
Prof.				No strong rules
Level 1	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{D1a}] ⇒ [{C1LCa}]	0.41	0.47	Loop construction **after** data change
	[{C1LCa}] ⇒ [{C2CPa,D1LDa}]	0.27	0.29	
	[{C1LCa}] ⇒ [{D1LDa}]	0.27	0.29	
Prof.	[{C2LCa,C2LIa,C2LUa} → {D1a} ... → {D2a} → {D1SDa}] ... ⇒ [{D2LDa,D2SDa}]	0.13	1.00	Loop construction **before** data change
	[{D1a} → {C1Ca}] ⇒ [{D1CDa}]	0.13	1.00	Conditional construction **after** data change
	[{C3LCa}] ⇒ [{D1LDa}]	0.13	1.00	Trivial construction order
Level 2	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{D1d}] ⇒ [{C1LCu}]	0.41	0.47	Loop construction **after** data definition
	[{C2LCu}] ⇒ [{D1LDd}]	0.36	0.44	Trivial construction order
	[{C2Cu,C1Cu}] ⇒ [{D1CDd}]	0.27	1.00	Trivial construction order
	[{C2Cu}] ⇒ [{D1CDd}]	0.27	0.86	
Prof.				No strong rules

Tab. 5.15: Strongest association rules of frequent sequences, mined with a maximum gap of 5 with from full method construction sequences with numbered variables (Table 5.10 (`a,b`)) and for granularity levels [0 − 1], for changes labeled with micropatterns.

Method Construction Sequence (Micropatterns, Numbered, Gap=5)

Level 0	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{D1}] ⇒ [{C2LU}]	0.41	0.47	Loop construction **after** data change
	[{D1} → {C2LC,C1LC}]	0.32	0.78	
	... [⇒ [{C1CP,D1LD,D1SD}]			
	[{C2LU} → {D1LD}] ⇒ [{C1LC}]	0.32	0.58	Change to loop condition **after** loop construction
	[{C2LC,C2LI} → {D1LD}] ⇒ [{C1LC}]	0.32	0.58	
Prof.	[{C2LC} → {D2}] ⇒ [{D2LD,D2SD}]	0.33	0.83	Loop construction **before** data change
	[{C2LC} → {D1}] ⇒ [{D1LD,C1CP}]	0.33	0.83	
	[{C2LC,C1LC} → {D1}]	0.27	1.00	
	... ⇒ [{D1LD,D1SD,C1CP,C2CP}]			

Level 1	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{D1a}] ⇒ [{C1LCa}]	0.45	0.53	Loop construction **after** data change
	[{C2LCa,C2LIa,C1LCa}]	0.36	0.67	
	... ⇒ [{C2CPa,C1CPa,D1LDa}]			
Prof.	[{C2LIa} → {C2Ca}]	0.20	1.00	Construction of nested conditional structure
	... ⇒ [{D1CDa,D1SDa}]			
	[{C2LCa} → {D2a}] ⇒ [{D2LDa,D2SDa}]	0.33	0.83	Loop construction **before** data change
	[{C2LCa,C1LCa} → {D2a}] ⇒ [{D1LDa}]	0.27	1.00	
	[{C1LCa} → {D2a}] ⇒ [{D1LDa}]	0.27	1.00	
	[{C2LIa,C2LUa} → {D2a}]	0.27	0.80	
	... ⇒ [{D2LDa,D2SDa}]			
	[{C2LCa,C2LIa} → {D2a}	0.20	1.00	
	... → {D1SDa}] ⇒ [{D2LDa,D2SDa}]			

Tab. 5.16: Strongest association rules of frequent sequences, mined with a maximum gap of 5 with from full method construction sequences with numbered variables (Table 5.10 `(a,b)`) and for granularity levels [0 − 1], for changes labeled with micropatterns.

Method Construction Sequence (Micropatterns, Numbered, Gap=5)

Level 2	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student	[{D1d} → {C2Cu,C1Cu}] ⇒ [{D1CDd}]	0.27	1.00	Conditional construction **after** data definition
	[{C1LCu} → {C1Cu}] ⇒ [{D1CDd}]	0.27	1.00	Construction of nested conditional structure (loop **after** data)
	[{D1d} → {D1LDu} → {C2Cu,C1Cu}] ... ⇒ [{D1CDd}]	0.23	1.00	
	[{C1LCu} → {D1d} → {C2Cu}] ... ⇒ [{D1CDd}]	0.23	1.00	Variant of nested conditional structure (loop **before** data)
	[{D1d} → {C2LCu,C1LCu}] ... ⇒ [{C1CPu,C2CPu,D1LDd,D1SDd}]	0.32	0.70	Loop construction **after** data definition
Prof.	[{C2LCu} → {D2d}] ⇒ [{D2LDd,D2LDu}]	0.33	0.83	Loop construction **after** data definition
	[{C2LCu} → {D1d}] ⇒ [{C1CPu,D1LDu}]	0.33	0.83	
	[{C2LCu,C1LCu} → {D1d}] ... ⇒ [{C1CPu,D1LDu,D1SDd}]	0.27	1.00	
	[{C2LCu} → {D1d} ... → {D1LDd,D1LDu,D1SDd,D1SDu}] ... ⇒ [{C1CPu}]	0.27	0.80	
	[{C2LCu,C2LUu,C1LCu} → {D1d}] ... ⇒ [{C1CPu,C2CPu,D1LDu,D1SDd}]	0.20	1.00	
	[{C2LCu} → {C2Cu} → {D1CDu}] ... ⇒ [{D1u}]	0.20	1.00	Construction of nested conditional structure
	[{C2LCu} → {C2Cu}] ... ⇒ [{D1CDd,D1CDu,D1SDd,D1SDu}]	0.20	1.00	
Level 3	***Sequence***	***Sup.***	***Conf.***	***Interpretation***
Student				No strong rules
Prof.	[{C2LCau} → {D2ad}] ⇒ [{D2LDau}]	0.33	0.83	Loop construction **before** data change
	[{C2LCau} → {D2ad}] ⇒ [{D2LDad}]	0.33	0.83	

typed and labeled by detailed micropatterns, that additionally contain numbered variable construction sequences for all variables of each respective method. The variables are numbered according to type (control or data) and occurrence. The resulting data corresponds to the example sequence of Figure 5.4. The **strongest** and most **frequent** rules of this evaluation are reported in Table 5.14 – rules for the granularity levels L0–L2 are reported. The **support** of these **frequent** rules is high for student-written methods, accounting for up to 0.55 of these methods, but is low for professional-written methods, accounting for only two method implementations (**support** of 0.13).

Interpreting the frequent sequences of granularity level L0, there are two sets of **strong** rules for student-written methods. The first set again shows that students implement loop constructs (CLC, CLI) **after** generic data changes (D). The second set captures the trivial construction order of implementing conditional control variables (CC) before implementing conditionally-dependent data variables (DCD). There are no captured **strong** rules for professional programmers.

Interpreting the frequent sequences of granularity level L1, sets of **strong** rules provide the same details to the construction order of loop constructs, compared to unnumbered sequences. Notably, the construction order of professional programmers is captured as additions to loop constructs (CLCa, CLIa, CLUa) **before** the addition of data changes (Da), which leads to the addition of data variables in a loop-dependent scope (DLDa). Moreover, the professionals' implementation of conditional constructs has a different order: the conditional control variable is added (CCa) **after** a generic data addition (Da). However, the construction of conditional constructs cannot be easily retraced for the example problem.

Interpreting the frequent sequences of granularity level L2, there are three sets of **strong** rules for student-written methods. The first set gives the same details to the construction order of loop constructs compared to unnumbered sequences. The second and third set captures trivial construction orders of implementing uses of control variables for specific structures (CLCu, CCu) **before** constructing data variables dependent on the specific control structure (DLDd, DCDd). There are no captured **strong** rules for professional programmers.

Even when numbering the variables and therefore introducing specificity in the sequences, which leads to a lower **support** of rules, the same sequential order of program construction emerges: novice programmers tend to implement loop constructs **after** data definitions, while expert programmers tend to implement loop constructs **before** data definitions.

Next, I also evaluated the numbered method construction sequences with the SPADE parameter gap set to five. This allows for frequent rules to be found with a gap of five other *program increments* in between. The **strongest** and most **frequent** rules of this evaluation are reported in Table 5.15 and Table 5.16. The highest

support of these **frequent** rules is 0.41 for student methods and 0.33 for rules of professional methods. With the allowed gap in sequential *program increments*, many specific frequent sequences are found for all granularity levels. I now highlight additional results not covered so far.

Interpreting the frequent sequences of granularity level `L0`, two rules capture that students make changes to loop conditions (`CLC`) after constructing the loop (`CLU`, `CLC`, `CLI`) and the loop-dependent data variable (`DLD`). These rules, with a **support** of 0.32, could again hint at logical errors which are fixed by the students by making changes to the loop construct.

Interpreting the frequent sequences of granularity level `L1`, two rules capture the construction order of professionals when constructing nested conditional structures (with a **support** of 0.20): they first construct the loop construct (`CLCu`) before constructing the conditional construct (`CCu`) and finish with the construction of conditionally-dependent data variables (`DCDd`, `DCDu`). The same order can be seen in the frequent sequences of level `L2`.

Interpreting the frequent sequences of granularity level `L2`, the construction order of conditional constructs can be explained in more detail. Professional programmers are described above. Students implement conditional constructs (`CCu`) **after** data definitions (`Dd`), with a **support** of 0.27. Moreover, for nested conditional structures of students, two different construction orders are captured. Some students, with a **support** of 0.23, implement nested control structures in the same order as loop-only structures: they implement the loop construct (`CLCu`, `CLDu`) **after** data definitions (`Dd`) before finally constructing the conditional construct (`CCu`) and conditionally-dependent data variables (`DCDd`). Other students, also with a **support** of 0.23, implement the loop construct (`CLCu`) **before** data definitions (`Dd`) before finally constructing the rest of the nested conditional structure. In all other cases, besides this nested conditional structure, do the **frequent** rules suggest that students implement loop constructs **after** data definitions.

Concluding the findings of this section, hypothesis $H0_3$ can be refuted: there are substantial differences in the order of program construction sequences of novice and expert programmers, given that these sequences are typed and labeled with micropatterns. Novice programmers tend to implement loop constructs **after** allocating and initializing data variables that will be needed in the loop scope, while expert programmers tend to implement loop constructs **before** allocating and initializing these data variables.

This is an essential finding in the context of the example problem `ConvertBinary`. The majority of implementations for this example problem feature a loop construct with one or two control variables and at least one data variable that holds the integer sum of the binary input array. The presented difference in construction order relates to the initial definition of the data variable (which is a variable declaration and initialization before the loop-dependent scope), the loop construct (establishing the loop-dependent scope), and the definition and/or use of the data variable in the loop-dependent scope.

Interpreted with the dimensions of differences established in Chapter 2 and with the specific differences outlined by Winslow [Win96], I frame these substantial differences in construction order in the following way. At first glance, it may seem that the experts in this study '*tend to approach programming through control structures*', which is a novice trait, by starting their implementation with the loop construct. However, in this case, the experts actually approach this program '*through its data structures and objects*' (the input array) and first care about traversing this data structure before dealing with how to compute and store the quantity in demand – thereby demonstrating expert programming skills [Win96, p. 18].

The novices, on the other hand, clearly demonstrate their ascribed behaviour in their construction order. They '*work backwards from the goal to determine the solution*', which could be attributed to a '*bottom-up approach to problem solution*' by initializing the data variable before dealing with how to traverse the data structure (and thereby accessing all the data needed for the problem) [Win96, p. 18].

Differences in the construction order of other control constructs (for example, conditional constructs like *if-else* structures or nested structured specifically) have not been captured. A possible explanation is that the *algorithmic affordance* of the example problem did not necessitate such a structure – making the mining of frequent sequences with other structures not possible. In order to evaluate differences in the construction order of other control structures, future work could incorporate varying example problems that dominantly necessitate those control structures.

5.5 Threats to Validity

There are a number of sources of threats to the validity of this study. I describe these threats and counter-measures by dividing them into threats to the internal and external validity of this quantitative study (also see Cohen et al. [CMM11, p. 183ff]).

Three factors of the comparative study represent potential threats to the internal validity. The first threat to the internal validity is the relatively small size of the study cohorts (individual programmers) that produce the raw data used for inferential statistics. While this threat might not appear as pronounced with a kind of *multiplicative* effect in place (each programmer produces multiple methods with multiple variables), the results derived from inferential statistics still need to relate back to individual programmers. I tried to pre-emptively mitigate this threat by recruiting a greater number of programmers with online recruitment to no avail: only 1% of the contacted students and 7% of the contacted professional programmers participated in the study. Another mitigation to this threat is that the results of the frequent sequence mining are specifically evaluated with the maximum *support* in mind, translating to the maximum number of individual programmers for which the strongest frequent sequences apply.

The second threat to the internal validity is the set of uncontrolled factors in the recruitment of programmers. The uncontrolled factors include: previous experience in programming and programming in `Java` in particular (i.e., actual level of programming skills), previous experience in working with the supported IDE, and gender. While experience in programming has been elicited for a part of the cohort (that was already used in the mixed methods study), the other part of the cohort was not administered any survey to elicit experience in programming. The rationale was not to burden the participants with additional surveys, considering that they are assigned the comparison groups based on the cohort. This, however, includes the possibility of experienced student programmers (who might already be working in industry) being practically assigned to the *wrong* comparison group. This threat is mitigated in two ways. First, the comparative study is not presented as an experiment, lacking the control of confounding factors. Second, the correctness of the implemented methods is not evaluated – thereby not weighing in actual programming experience. Altogether, the design of the comparative study is not suited in an experimental setting but depicts the true setting in programming courses.

The third threat to the internal validity is the manual part of the data preparation. This comprises: i) the semi-automatic processing of the raw data and the manual labeling of all variables for the newly recorded programs and ii) the semi-automatic processing and re-labeling of all changes in the sequential *program increments* for all variables. To mitigate the threat `i)`, I iteratively passed over the program construction and labeling until the variable-specific *construction flowlines* accurately depict the sequential program construction, manually checked with my **program increment browser**. To mitigate the threat `ii)`, I used a step-wise strategy, incorporating automatic processing at every possible step. In the first step, the raw data of the variable-specific *construction flowlines* is automatically processed to produce typed and labeled change sequences for each variable on granularity level `L3`. In the second step, I did two passes over each variable to ensure correct manual re-labeling. In the third step, the sequences on granularity level `L3` are automatically transformed to other granularity levels, and are automatically aggregated to pairs of variables and to method construction sequences. This way, manual labeling was only necessary for a single sequence for each variable.

The sources of threats to the external validity of the comparative study, which influence its potential for generalizability, are similar to the sources of threats discussed for the mixed methods study in Section 4.4.4. I give a brief summary of similar arguments and also discuss the threats specific to this study. There are four factors of the comparative study that represent potential threats to the external validity.

Regarding the first factor of example problems, this study again highlights that the problem `MoveElements`, inspired by neo-Piagetian findings of Lister (see [Lis11b, Lis16]), provides hardly any insight on strategical approaches that can be distilled into *variable construction patterns*. This can be attributed to the comparatively low number of data variables used to solve this problem. The array variable of this problem is utilized as control and data variable, with my categorization framework defaulting to control variable in this case. Future work on the categorization framework is needed to uncover additional information in the case of such variable uses.

The second example problem, `ConvertBinary`, yields a specific *algorithmic affordance* with regards to implementation strategies – with loop solutions being dominantly used and conditional structures being seldom used. Additional example problems with a varied *algorithmic affordance* need to be evaluated to improve the generalizability of the results of loop construction order for general loop structures. For example, `for` loops and `while` loops might be constructed differently. Moreover, additional example problems need to be evaluated to investigate the construction order of conditional control structures and of compound and nested control structures.

The second threat is the use of a single text-based programming language, `Java`. This programming language is the primary one taught at the University of Klagenfurt (covering the student cohort) and is a commonly used programming language in the local software development industry (covering most of the professional cohort). It can therefore be concluded that the programmers' programming skills in `Java` are sufficient for solving the example problem. Regarding the generalizability of the comparative study, no `Java`-specific constructs have been used in the implemented methods. Therefore the findings regarding the construction order of loops can be generalized to similar procedural text-based programming languages.

The recruited cohorts of programmers constitute the third threat to the external validity. The first part of this threat is the uncontrolled level of programming skills of the programmers, especially the students. While most of the professional programmers have a mean programming experience of about 17 years, students have been recruited from varying undergraduate courses (including the first, second, and third year of studies) and from one graduate course. This translates to a lacking strength of generalizability, as the students cannot be pinpointed to homogenous groups of programming skill levels. This threat is mitigated by the fact that the students are not compared and that they can still be separated from the professional programmers by their respective years of programming experience. However, the generalizability of findings could be improved by recruiting whole cohorts of students for which the homogeneity is known (e.g., entry-level courses or courses with specified programming skill prerequisites). The second part of this threat is the relatively small size of the cohorts and the imbalance of the groups of students and professional programmers, which translates to a lacking strength of generalizability. I mitigate this threat by theoretical means, interpreting the findings regarding the construction order of loops in light of previously published results (see Winslow [Win96]). However, I cannot ensure that the findings hold in the same proportion for other cohorts of programmers.

The fourth threat is that gender is disparately represented in the cohort of professional programmers. Again, it has to be noted that the loop construction order obtained from the frequent patterns can be predominantly described as *male* construction order. Future research needs to address female professional programmers to investigate potential differences.

5.6 Summary of Comparative Study

I first summarize the findings of the comparative study in terms of the three null hypotheses. I thereby cover the arguments developed in the respective sections of this study to refute the individual hypotheses or depict why they cannot be refuted for this study.

$H0_1$. Comparing novice programmers and expert programmers, there is no significant difference in micropattern usage and variable construction pattern usage **for individual variables**.

The underlying question for the null hypothesis $H0_1$ is how well micropattern and *variable construction pattern* usage are suited for explaining differences in programming skills. Using the findings of inferential statistics described in Section 5.2, the null hypothesis $H0_1$ can be refuted in two points. First, expert programmers of the measured cohort have a significantly higher usage fraction of *loop-dependent* data variables, compared to students of the measured cohort. Second, expert programmers have a significantly higher usage fraction for the data *variable construction pattern* {*conditionally-dependent*, *loop-dependent*, *self-defining*}. This pattern only occurs in `ConvertBinary` implementations and characterizes a specific type of data variable that is used in loop and conditional control contexts. For other patterns, there are no significant differences between the two cohort groups.

A control test shows that there are pronounced differences in the pattern usage between the two example problems, with significant differences in the usage of multiple micropatterns and *variable construction patterns*. These findings suggest that the pattern usage alone is not suited for explaining differences in programming skills. However, they might be suited for assessing the *algorithmic affordance* of different problems, potentially discriminating types or classes of algorithms. Such a discrimination could be a valuable tool for education and training.

$H0_2$. Comparing novice programmers and expert programmers, there is no significant difference in micropattern usage **on the level of methods**.

The underlying question for the null hypothesis $H0_2$ is how well the micropattern usage, aggregated on the level of methods, is suited for explaining differences in programming skills. Using the findings of the inferential statistics described in Section 5.3, the null hypothesis $H0_2$ can be refuted in one point. Expert programmers of the measured cohort have a significantly higher aggregated usage for loop-related control micropatterns (*loop-condition* and *loop*), compared to students of the measured cohort. This finding could be attributed to experts managing their variables and not using extraneous ones, thereby increasing the aggregated usage of necessary loop-related control variables.

Again, the control test shows that there are significant differences in the aggregated micropattern usage for all but two micropatterns when comparing implementations of the two example problems. Altogether, these findings suggest that the aggregated micropattern usage alone is not suited for explaining differences in programming skills. However, it might be suited for discriminating types or classes of algorithms, as explained above.

$H0_3$. Comparing novice programmers and expert programmers, there is no substantial difference in micropattern usage regarding the **program construction sequence**.

The underlying question for the null hypothesis $H0_3$ is how well suitably typed sequences of program changes are suited for explaining differences in programming skills. Using the findings of the frequent pattern mining described in Section 5.4, a substantial difference between the construction order of expert programmers and novice programmers has been found with multiple granularity levels for typing and with micropatterns to label individual changes. The substantial difference found in the mined frequent rules, proven with a maximum **support** of 0.5 and a maximum **confidence** of 0.6 for novices and of 0.88 for experts, concerns the construction order of loop control variables and of related loop-dependent data variables.

For novice programmers, the dominant construction order consists of first constructing the **initial definition** of the data variable, then constructing the **loop header** (which is, in the implementations of the example problem `ConvertBinary`, a `for` loop with initialization, condition, and update parts), and lastly constructing **loop-dependent changes** to the data variable. To summarize, novice programmers tend to implement the loop construct **after** initializing the needed data variable.

For expert programmers, the dominant construction order consists of first constructing the (`for`) **loop header**, then constructing the **initial definition** of the data variable, and lastly constructing **loop-dependent changes** to the data variable. To summarize, expert programmers tend to implement the loop construct **before** initializing the needed data variable.

Altogether, the null hypothesis $H0_3$ can be refuted: with frequent sequence rules, qualified by the interestingness measures **support** and **confidence**, a substantial difference in micropattern usage can be found in the construction orders of novice programmers and expert programmers.

Now I can summarize the findings to give an answer to the fourth research question of the thesis:

RQ4. What are the differences between novices and experts concerning the use of program construction patterns?

The first key finding of the comparative study is that the use of *variable construction patterns* is closely tied to the *algorithmic affordance* of the problem to be implemented, meaning that different target implementations will necessarily feature differing distributions of the use of construction patterns. In this context, while experts show a nuanced use of specific micropatterns, proven with significant differences for loop-related micropatterns on the level of individual variables and of aggregated method profiles, novice programmers also seem to have a grasp on the basic (*semantic*) capabilities of different control and data variables, dependent on the requirements of the problems. In order to support programmers in improving their nuanced use of control and data variables (and their roles as labeled with the micropatterns), specific instructional strategies can be devised to analyze and model classes of problems with regard to the needed types of variables.

The second key finding of the comparative study is the difference in the construction order of loop headers and data-dependent variables. Expert programmers demonstrate their level of proficiency by, interpreted with Winslow [Win96], approaching the program construction through the respective data structures. For the `ConvertBinary` example problem, experts are first constructing the array traversal before constructing the data variable needed for the computation. Novice programmers go the other way around; their approach could be described as a bottom-up approach, working backwards from the goal to determine the solution. For the `ConvertBinary` example problem, novices first construct the data variable needed for the computation before constructing the array traversal.

These findings prove that, for the specific example problem and similar loop problems, there is a construction order preferred by expert programmers – and it is different from the construction order most often implemented by novice programmers. Deducing from these findings, it can be beneficial for the education and training of novice and experienced programmers to again focus on analyzing and modeling classes of problems with the goal of internalizing the average expert's construction order. This, in turn, constitutes becoming an *expert*.

For the specific example problem and similar loop problems, the recommendation is to let learners experience construction orders typically not found in the training of experienced programmers. This encompasses not approaching example problems with a *step-by-step* bottom-up approach, but instead working on learning programming skills on the level of experts: approaching the problems through data structures and objects, with specific algorithms, and with tactical strategy [Win96]. Such an approach could potentially jump-start a programmer's skill acquisition to become an expert. Such an instructional strategy needs to be investigated with experimental studies.

To conclude this chapter, I uncovered two differences between novices and expert programmers. First, **experts have a more nuanced use of micropatterns**, measured with the usage fractions of individual variables and with aggregated usage fractions for all variables of a method. This nuanced use could be dependent on the *algorithmic affordance* of the target implementation – both example problems required the use of loop control variables, which coincides with the micropatterns for which a significant difference was found.

Second, experts construct loop headers and loop-dependent data variables in a different order compared to students. The **experts first construct the loop** and thereby focus on the data structures, while the **students first construct the data variable** and thereby pursue a bottom-up approach to problem-solving.

6. CONTEXTUALIZATION OF RESEARCH RESULTS

In the previous chapters, I defined program construction patterns and specifically *variable construction patterns* and proved that, for the cohorts participating in my studies, there are significant differences in the sequence of program construction, detectable by my typed patterns. After the methodological approaches and research findings have been summarized, what remains is to properly contextualize the findings to give an answer to the main research question of this thesis:

MRQ. To what extent can patterns of program construction sequences be indicative, whether positive or not, of the acquisition of expert programming skills?

While compiling answers to the main research question, the overarching aim of contextualization is to identify the possible gains from my methodological approaches and from my research findings, both in theoretical and practical nature. This chapter is thus structured in three sections.

Section 6.1 covers the contextualization of the research results within the theoretical frameworks of the neo-Piagetian hierarchy of programming skills established by Lister [Lis16] and the cognitive load theory established by Sweller et al. [SVP98]. This section aims to summarize my approaches and findings towards their theoretical contributions to the research field of computer science education.

Section 6.2 covers the contextualization of the research results within the practical realms of computer science education, including potential applications of learning analytics dashboards, learning analytics task design, and recommendations regarding instructional strategies. This section aims to summarize my approaches and findings towards their practical contributions to the community of educators and instructors of computer science (and programming in particular).

Concluding this chapter, Section 6.3 summarizes the contextualization in order to arrive at answers to the main research question of this thesis in Section 6.4.

6.1 Theoretical Contextualization

In this section, I contextualize the approaches and findings of my research within the two theoretical frameworks introduced in Chapter 2. More specifically, I investigate how my findings of nuanced variable use and of the construction order of loops, which both provide evidence of expert programming skills, relate to the two psychological frameworks of the neo-Piagetian hierarchy of programming skills (Section 6.1.1) and the cognitive load theory (Section 6.1.2). Altogether, this section summarizes my theoretical contributions to the research field of computer science education.

6.1.1 Expert Skills in the Neo-Piagetian Hierarchy

When introducing the neo-Piagetian hierarchy of programming skills, as established through the research catalog of Lister [Lis16], I noted an unanswered question in his research program that is highly relevant to the context of this thesis. The question is: How can students be supported to reach the highest stage of programming (*formal operational* reasoning)?

Lister provides a hierarchy of programming skill development, framed with the skill of mentally executing (*tracing*) program code, that captures a novice programmer's skill acquisition until they are capable of *abstract tracing* and purposeful code writing (the *concrete operational* stage). Nevertheless, using the mental execution as a guiding frame falls short when investigating the transition to the *formal operational* stage. This final stage of reasoning is not only concerned with mentally executing a single piece of program code but comprises skills such as hypothetico-deductive reasoning, systematic abstract reasoning, reflective capacities, and reasoning and problem solving in unfamiliar situations (Corney et al. [CTAL12]).

However, the neo-Piagetian approach can still provide a framework to guide skill acquisition from *concrete operational* to *formal operational* reasoning. Lister identifies that it is essential for a didactical approach to fit the students' abilities [Lis11b]. I use this guideline to put my findings on expert programming skills into a hierarchical context.

To derive theoretical implications from my findings on nuanced variable use and on the loop construction order, I first describe the neo-Piagetian context in which the acquisition of expert programming skills takes place. This is important as the learning experience needs to be coordinated with the learners' level of reasoning. As an example, as Lister puts it, writing code is not an appropriate task for students that cannot trace consistently [Lis11b].

The acquisition of expert programming skills is most relevant for programmers capable of *concrete operational* reasoning at the *post-tracing* or *abstract tracing* stage. In this stage, programmers have developed the necessary skills to mentally execute program code with abstraction. They do not evaluate program code with specific variable assignments in mind and '*instead mentally maintain[s] algebraic-like constraints on the possible values in each variable*' [Lis16, p. 14]. Programmers in this stage effectively arrive at the ability to *read* program code, mentally executing it as they move through the code, and understanding the overall purpose of the code without explicit traces. This ability to mentally store all constraints closely relates to the notion of schemas in cognitive load theory, described below. Moreover, writing code with a purpose is now possible at this stage of reasoning.

With this context of *concrete operational* programming skills, I can contextualize my findings. Lister summarizes that '*the concrete operational programmer may further develop their programming skills using the approach used for decades — by having them write lots of code*' [Lis16, p. 14]. Now, following Lister's own argumentation that the type of learning experience needs to be coordinated with the learner, a *concrete operational* programmer looking to improve their level of programming skills should not only be concerned with learning how to *write* program code but also how to develop expert programming skills along the qualities that define *formal operational* reasoning.

To have programmers develop the necessary reasoning skills, one way is to let them *write lots of code* until they figure out optimal approaches and develop abstract reasoning through experience – which constitutes a largely uncontrolled process. I propose that this process can be improved by appropriate instructional design. An appropriate instruction design can incorporate my findings that expert programmers use specific variables in a nuanced way and apply abstract reasoning before and during program construction (which manifests itself in the construction order, e.g., of loops). Such a design needs to offer learning experiences that not only account for writing program code, but also for reasoning about and designing problem solutions the way an expert would do (as described previously, see Winslow [Win96]). Consequently, such a design could be tailored to triggering a process of assimilation and accommodation, challenging the way of reasoning about implementing program code that has formed in the *concrete operational* programmer.

In this frame, it could be possible to even go a step further. The prime goal of supporting learning programmers in the *post-tracing* stage, finally capable of writing program code on their own, could be to let them experience code writing problems through an *expert lens*, presenting as many opportunities as possible to develop abstract code reasoning skills and not just having them write code for the sake of experience.

Essentially, my argument is to let programmers learn the *expert* way of implementing program code to reduce the time investment needed to develop specific expert programming skills (towards the *formal operational* stage). However, I am aware that not all developmental steps can be similarly simplified – experience with different problems is, for example, helpful to improve reasoning about unfamiliar situations.

There have been previous recommendations to make different programming approaches or strategies explicit to the students. Examples include the approach presented by Rubin [Rub13] where an '*expert*', the educator, provides a live-coding experience and thereby exemplifies their implementation strategies, and the approach of de Raadt et al. [dRWT09] where the students study programming with compiled work sheets of strategies and plans. In contrast, my recommendations are directly derived from my findings that relate the specific construction order and the underlying reasoning to evidence found in expert programmers' construction sequences.

Avenues for future work in this area, founded on my theoretical claims, are conceivable in two directions. The first one is the development and evaluation of types of programming problems capable of detecting the transition in skill levels between *concrete operational* and *formal operational*. The second one is the design of experiments that make use of such programming problems, evaluating instructional strategies with regard to the skill development of programmers. A problematic factor in this context is that fully developing expert programming skills might take a long time. This, in turn, necessitates that the detection problems accurately differentiate among levels of expert programming skills, potentially on specific subscales. Moreover, longitudinal studies to observe the development of expert programming skills could provide valuable insight into sequential factors.

6.1.2 Expert Skills in the Context of Cognitive Load Theory

The argument presented above, providing programmers with learning opportunities to experience and practice expert programming skills during program construction, can be extended with the notions of cognitive load theory (CLT) [SVP98]. When introducing CLT, I noted that its most important feature, in the context of this thesis, is the notion of *schema* acquisition and the resulting decrease in a problem's *intrinsic load* as a key aspect of expertise.

Schemas are an important part of CLT and model the process of developing expertise, with learners gradually forming schemas as a collection of single cognitive elements and other schemas. Schemas make it possible for learners to hold additional information in their limited working memory, as schemas are regarded as new single elements. Summarizing, more experienced programmers can consider and reason with much more syntactic and compound elements during program construction compared to novice programmers. Essentially, this interpretation and the neo-Piagetian theory operate on the similar idea that learners need to develop the appropriate cognitive structures before advanced reasoning in the subject at hand is possible.

I abstract two key observations from this interpretation while working towards integrating learning opportunities of expert programming skills into programming education. The first observation is that, akin to neo-Piagetian developmental theory, CLT suggests that novice programmers have to master basic programming concepts in order to develop the cognitive structures necessary for engaging in expert reasoning. Mastering a concept, in this context, means combining the occurring syntactic elements into comprehensive schemas and, ideally, combining these comprehensive (syntactic) schemas with a basic idea of the semantic purpose of the occurring elements. In total, learning programmers cannot engage in developing expert programming skills (see Winslow [Win96], Corney et al. [CTAL12]) before they have formed schemas of suitable granularity so that they have freed up working memory to devote to learning. This need to form schemas of syntactic elements has to be reflected in the task design for novice programmers.

The second observation, following from the first one, is that task design again needs to be adapted once learning programmers have formed comprehensive schemas of syntactic elements and basic semantic programming structures. In other words, there is little gain in having learning programmers simply write program code that makes use of the acquired schemas – specifically when targeting the development of expert reasoning skills (that have manifested in nuanced variable use and a specific loop construction order in my studies). They instead need fitting learning experiences that facilitate and support the next step in forming schemas – combining their syntactic and basic semantic knowledge with abstract reasoning and plan-like strategical approaches.

A possible avenue to achieve an appropriate task design and give appropriate support for learners is to combine the theoretical notions of the neo-Piagetian hierarchy of programming skills with the implications CLT brings to the table of programming education. I firmly believe that the development of specific expert programming skills, like abstract reasoning, can be streamlined with an appropriate design of programming tasks. My findings provide evidence that expert programming skills manifest themselves in different ways during program construction. Avenues for future work in this direction include devising learning experiences that specifically facilitate the development of expert programming skills, and that can be measured with instruments based on learning analytics. Practical ways to support task design, dependent on a learner's programming skills, are discussed in in the context of learning analytics in Section 6.2.

6.2 Practical Contextualization

In this section, I contextualize the approaches and findings of my research within prospective practical approaches that aim to improve programming skill acquisition. More specifically, I introduce my vision of how learning analytics (LA) approaches can contribute to improving programming skill acquisition. I first present related work on LA dashboards (Section 6.2.1). I then focus on two prospective means to support educators and instructors. The first is to facilitate easy use of LA technology through a LA dashboard (Section 6.2.2). The second is to combine my approaches and research findings to give recommendations regarding instructional strategies that adhere to my findings and can be supported by LA. Moreover, I describe prospective LA interventions and respective task designs that offer a new way of supporting learners when learning to program (Section 6.2.4). Altogether, this section summarizes my practical contributions to the community of educators and instructors that are concerned with improving programming education.

6.2.1 Related Work on Learning Analytics Dashboards

The practical use of LA data by different groups of educators (teachers, instructional designers, curricular developers, among others) is an area that is heavily researched by multiple communities and is not limited to STEM-related fields (which, at first glance, might lend themselves to easier integration of LA data due to their technical nature). I give a brief overview of recent studies towards the research and practical use of LA data (most often in LA dashboards).

Bodily et al. [BKA$^+$18] provide a systematic literature review of Open Learner Model (OLM) research. OLM research aims to model each student's learning state to provide feedback and individual recommendations for improvement, while LA dashboards are geared towards supporting '*data-driven decision making*' [BKA$^+$18, p. 42]. The authors code research publications in the area of Open Learner Models (OLMs) regarding their data usage, their modeling approaches, and their experimental and practical evaluation. They proceed to compare their evaluation with literature reviews of LA dashboard research, noting that LA dashboards lack rigorous evaluations and the inclusion of students' assessment data, among others. The authors call for unified research approaches that draw from the strengths of both individual research strands to improve education.

Konstantinos et al. [MLHLPD20] report on a cross-comparison of two case studies that incorporate educators into the design of new LA applications, including teachers with a variety of main subjects. The authors distill common teacher-identified problems and solutions in the context of LA designs and conclude with a number of lessons learned for future designs of applications that facilitate the use of LA data for educators. Two important lessons are, first, that the educators' practices and priorities should provide the focus to designing LA data interpretations, and, second, that the educators need reflective cycles to be able to make use of LA data for real time activity management and potential re-design of learning activities. Relevant to my contextualization, the authors note that actionable information from LA data can be divided into two types: *design-time* information that enables the customization of future learning designs, e.g., task designs, and *run-time* information that enables the management of learner groups.

Brown [Bro20] conduct a qualitative study to investigate (physics) lecturers' use of LA dashboard data. The author observes in-class sessions, faculty meetings, and interview sessions, collects classroom artifacts, and codes the data for emergent themes. There are a number of organizational and ethical themes that I skip on reporting. Regarding the pedagogical use of LA data that is accessible at all times through a dashboard, the observed participants state that '*dashboards complicated pedagogy*' [Bro20, p. 391] as the tool needs to align with teaching practices and beliefs. A major topic in this regard, as stated by the participants, is that there is a lack of clarity in the data presentation of the observed tool. Participants had a hard time uncovering the meaning of the presented data, making it untrustworthy and not usable. Moreover, the participants were unable to draw actionable insights from the data in the presented way – thereby defeating most of the purpose of the practical use of LA data. Brown concludes that LA dashboards should not only track tally counts (*when*) of student actions, but also show (processed) data along the dimensions of *what*, *how*, and *why*. This again translates to inferring meaning from the collected and processed data.

In a meta-context, Brown also discusses that LA dashboards, as technical tools, are socio-technical systems that shape the pedagogical and didactical space in a classroom. Ideally, these tools should support educators and students to improve education in an unobtrusive way. However, taken to the extremes, they dictate education by stipulating tasks, assessments, and feedback along their terms.

Muljana and Luo [ML21] report on a qualitative study to elicit the voices from instructional designers in higher education towards using LA data to shape and inform course design. They report on four factors, from which I again omit the organizational ones. For the factor of individual differences, the instructional designers state that LA needs to be aligned with, and dictated by, pedagogical beliefs. For the factor of system characteristics, LA data needs to be available on the correct granularity, needs to be legible (for the intended means), and corresponding analytical tools need to be user-friendly to best facilitate data-informed pedagogical and didactical decision-making. Altogether, while LA is seen as a technological-pedagogical tool by some instructional designers and only as measurement means by others, the tenor is that pedagogy should drive potential technological use.

Relevant for my contextualization, the authors distinguish between LA approaches that provide *checkpoint analytics* (that collect access and usage information of learning resources) and *process analytics* (that collect information about the learning process and student engagement). This denotes by approach to be a *process analytics* approach.

The qualitative studies that aim to uncover design needs and practices of usable LA dashboards provide evidence that the communities concerned with improving education with processing learners' data and making those processed data accessible to educators is, as Konstantinos et al. put it, '*at a similar level of infancy globally*' [MLHLPD20, p. 98]. Adding to this sentiment, Bodily et al. state that '*[...] there is still a need to assess the effect of LADs [LA dashboards] in real-life* *settings [...]*' [BKA$^+$18, p. 48]. Moreover, Muljana and Luo euphemistically note that '*[t]he process of translating data into actionable interventions to help students perform better is a non-trivial one*' [ML21, p. 210]. Concluding, there is still no *best* way to design and implement a LA dashboard, or even collect and process learners' data.

However, there are two common findings in the publications described above. The first one is that LA data use has to be tailored towards **specific educational needs** and needs to be adaptable to them. The second one is that the goal of any practical use of LA data should be to support educators in their decision making, enriching it to **data-informed decision making**.

6.2.2 Considerations towards a Dashboard Implementation

For the first practical contextualization, I take a look at how my LA approaches can be turned into practical use for educators in the context of programming education. This contextualization culminates in describing my implementation of a data repository for (block-based) programming and summarizing my ideas to design and implement a LA dashboard for programming education.

At present, there are no LA dashboards that make use of *process analytics* during programming to support teaching and learning in real time. While performing my research for this thesis, we also began to work towards a LA dashboard, which is founded on my LA approaches, at the Department of Informatics Didactics at the University of Klagenfurt.

Two factors need to be considered when implementing such a dashboard: the design of a data repository and the design of a dashboard for educators (and potentially students) that makes use of the collected data. Working towards implementing a LA dashboard for programming, we briefly described the implementation of a data repository for (block-based) programming data and our ideas to implement an educator dashboard in a conference publication [KWB20]. Here I give a comprehensive description.

Regarding the first factor of designing an appropriate data repository, we aimed at supporting easy data collection, which affects researchers and educators that want to contribute to the data repository by collecting data, and we also aimed at supporting easy access to the data, which affects researchers and educators that want to analyze the data.

The data collection is part of my LA approach described in Section 3.3, divided into IDE instrumentation plugins (currently supported are `Scratch 2` and `Scratch 3` as block-based IDEs and the `IDEA` family, like `IntelliJ`, as text-based IDEs) and a uniform data collection server that stores all program changes in similar units (as *program increments*).

To support easy access to the data, we developed a web-based user interface that currently offers the features of user registration and data download[1]. Users can register by entering login data (mail address and password), their current occupations (checkbox selection of teacher / student / researcher / other, with text fields to enter information as needed), their motivation to access the data, and one or more roles they want to register for. Currently, three user roles are implemented in the system.

Two data user roles are provided to give access to all saved programming sessions of block-based and text-based programming, respectively. The third role is to access the educator dashboard, which is currently not implemented and described below. User activation and role assignment is curated by administrator users, curated by the Department of Informatics Didactics at the University of Klagenfurt. Demographic data and roles are stored to evaluate the use of the dashboard and, thereby, research interest in the different types of data. The source code for the web-based user interface is also open-source[2].

To easily access the collected data, we implemented filter options to refine and limit the data when downloading programming sessions. Access to the download form is only possible for registered users. The filter options specify which programming sessions and specific programming changes should be included. The filter options are: i) start and end data of programming sessions, ii) data collection platforms and corresponding change types (access to the data collection platforms is tied to the registered user's roles), and iii) full-text search fields for filtering source file names, user names of participants, and error text messages. This way, we envision that access to both one's own collected data and to data collected by other researchers is easily possible.

The programming data is downloaded as a `.zip` file from my implemented `trackserver` that contains a directory and corresponding incremental program snapshots and change information for each individual programming session.

For the second factor of designing an appropriate dashboard for educators, our ideas are fueled by an evaluation carried out by Brown et al. [BASK18] regarding the use of the programming data collected in the `Blackbox` project. These authors identify that a major shortcoming of current programming data repositories is the lack of information regarding the learning and teaching context.

To accommodate this shortcoming, an educator dashboard has to maintain interfaces to data collection, storage, and access and provide an administrative interface that facilitates planning LA-powered programming lessons. We propose the following key factors when designing a LA educator dashboard for programming data:

1. **Context Information:** Educators should provide context information to the programming lessons so that the resulting collected data can be qualified by

[1] Link to the dashboard: `https://seqtrex-dashboard.aau.at/`

[2] Link to the repository: `https://gitlab-iid.aau.at/seqtrex/sequence-dashboard`

the learning and teaching context. This information can include demographic information about the learner cohort, information about the planned lesson, competencies and related educational standards, and more.

2. **Supplemental Material:** Educators should add the material that is used to realize the programming lesson. This information can include teaching and learning material, example problems, and task specifications.

3. **Data Collection:** Educators should be supported in managing the data collection. This can include creating data collection groups to link programming sessions to programming lessons.

4. **Feedback & Grading:** When the collected programming data is processed into representations, charts, and feedback, the LA educator dashboard should provide the possibility to add notes to evaluations of students that facilitate feedback and grading. Such assessments can provide insightful research data.

In total, the educator dashboard should enable educators to plan lessons and programming assignments, add context information to these lessons to improve future research, and receive representations, charts, and processed feedback from an underlying LA system – supporting them in improving their teaching with *data-informed decision making* (thereby improving their students' learning) [ML21]. Moreover, the dashboard should provide a focal point to facilitate the data collection, for example, by providing a learner interface so that they can register for the correct LA-powered lesson plans and get access to the instrumented IDEs and respective plugins.

We also envision a future research strand that can directly incorporate educators and students into LA research, facilitated by an appropriately designed and implemented LA dashboard for programming data that reports *process analytics* [KWB20].

Incorporating educators can help develop an understanding of the use and benefit of LA for educators. We identified potential template research questions that can be tackled in the described context:

RQ_{e1} Which depictions and evaluations of program sessions help educators with student assessment?

RQ_{e2} Which depictions and evaluations are best suited to identify struggling students?

An example in line with RQ_{e2}, taken from the exploratory block-based study described in Section 4.2, is to notify educators of learners not using blocks or syntactic constructs that are explicitly required (in accordance with a sample solution).

Similarly, incorporating students can help assess the merit of LA interventions, which goes hand in hand with evaluating appropriate task design as discussed in Section 6.2.4. We identified potential template research questions that can be tackled in the described context:

RQ_{s1} Which depictions and evaluations of program sessions help students with programming skill acquisition?

RQ_{s2} How is the usefulness of different depictions and evaluations related to students' programming skills?

An example in line with RQ_{s1} is to show learners the evolution of coverage of their variables during program construction and evaluate, through qualitative or quantitative means, the effects on their (perceived) skill acquisition.

To summarize, my LA approaches of IDE instrumentation and data collection need to be made accessible for educators to be put to greater use. Towards this goal, we implemented a web interface to the uniform data collection server `trackserver` that provides download access to all collected data for registered users [KWB20]. This is a first step towards providing educators access to *process analytics* of students' programming process, which should supply them with processed data to derive '*actionable insights*' from the data [Bro20, p. 392].

The next step, covered only by discussion and future visions, is the design and implementation of an educator dashboard that actually provides educators the technological means to easily plan LA-powered programming lessons, collect their students' data, and use the resulting processed data for feedback and grading. However, as Muljana and Luo found out in their qualitative analysis of instructional designers using LA systems, pedagogy and teaching needs should drive technological use [ML21]. This notion motivates future research to incorporate educators (and potentially students) into LA research to evaluate the direct use and benefit of processed LA data for programming education. As such, LA dashboards can represent an interface between researchers and educators, both contributing towards improving programming education.

6.2.3 Learning Analytics and Instructional Strategies

For the second practical contextualization, I take a look at how my LA approaches and the results of my studies can be used in instructional strategies for programming education. I understand the collection of all teacher-planned materials and activities that support students in learning programming as an instructional strategy. To this end, I derive cases of practical use that can be applied by programming educators with and without LA support, based on my approaches and results.

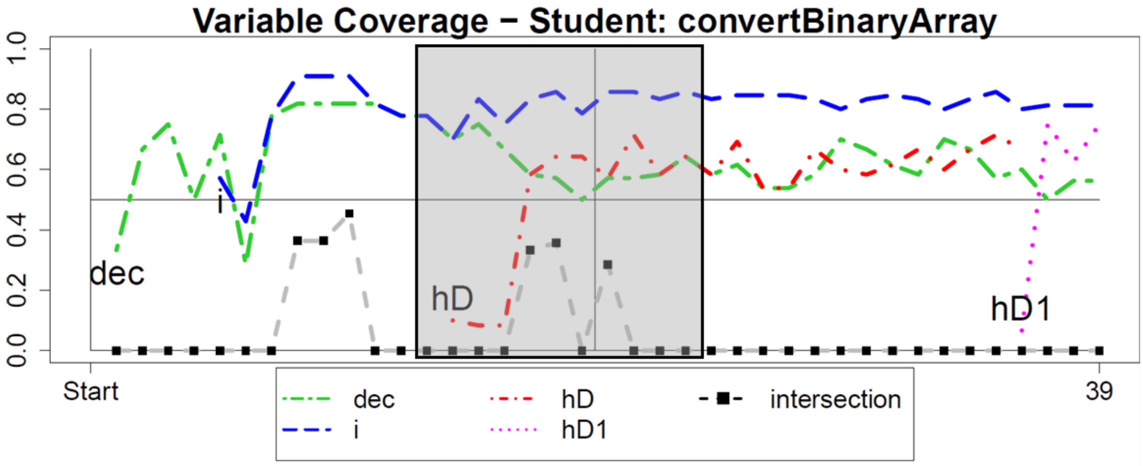


Fig. 6.1: Example use case of live LA metrics to assess the process of program construction. For example, educators can be notified when students introduce variables that result in non-cohesive programs with non-overlapping computations between variables. This example construction results in the program shown in Figure 6.3.

The first focus point is to derive benefits for instructional strategies from my LA approaches, particularly focusing on the potential use of LA data in the context of live monitoring and feedback. The goal is to support educators through processed LA data in order to help them facilitate students' skill acquisition, at best towards expert programming skills. Described previously in a workshop publication [KB21], I envision the productive use of processed LA data (and resulting representations) to facilitate real-time assessment and feedback of students' program construction process. Such an approach enables new types of corrective and supportive instructions.

To illustrate the benefits of live LA monitoring to improve educator feedback, consider an augmented representation of the evolution of a student's cohesion on the level of individual variables (measured with variable-level *Coverage*) shown in Figure 6.1. In the shaded area, the student introduces an additional data variable `hD`, which causes the *Coverage* of the data variable `dec` to decrease, while the *Coverage* of the control variable `i` remains rather high. The pair-wise *Coverage* between the variables (not shown here) additionally reveals that there is no cohesion between these data variables, suggesting multiple trains of thought or, respectively, method concerns.

In a classroom setting, educators can make use of this presented data in real-time to identify struggling students. In the cohesion example, evidence of struggling students could be non-cohesive methods or pairs of variables. More general, evidence of struggling students could be spans of non-compilable programs, the use of many variables relative to a reference implementation, or the use of different syntactic constructs relative to a reference implementation. Moreover, with the described real-time usage of data of students' program construction, a more timely and in-depth evaluation of their program construction process is possible.

Naturally, educators do not know or recognize all factors they need to be aware of to best facilitate students' skill acquisition. Because of this, a set of automatically processed assessments and guidelines on how to evaluate the assessments and resulting representations are needed to support educators in the use of LA data. A key question towards facilitating individual assessment and feedback that can be solved by automatically processed assessments is to help educators identify *where to look at* in the process of program construction. Proper implementation of the use of LA data can bring forth data-informed decision making during programming instructions, focusing on individual assessment and feedback.

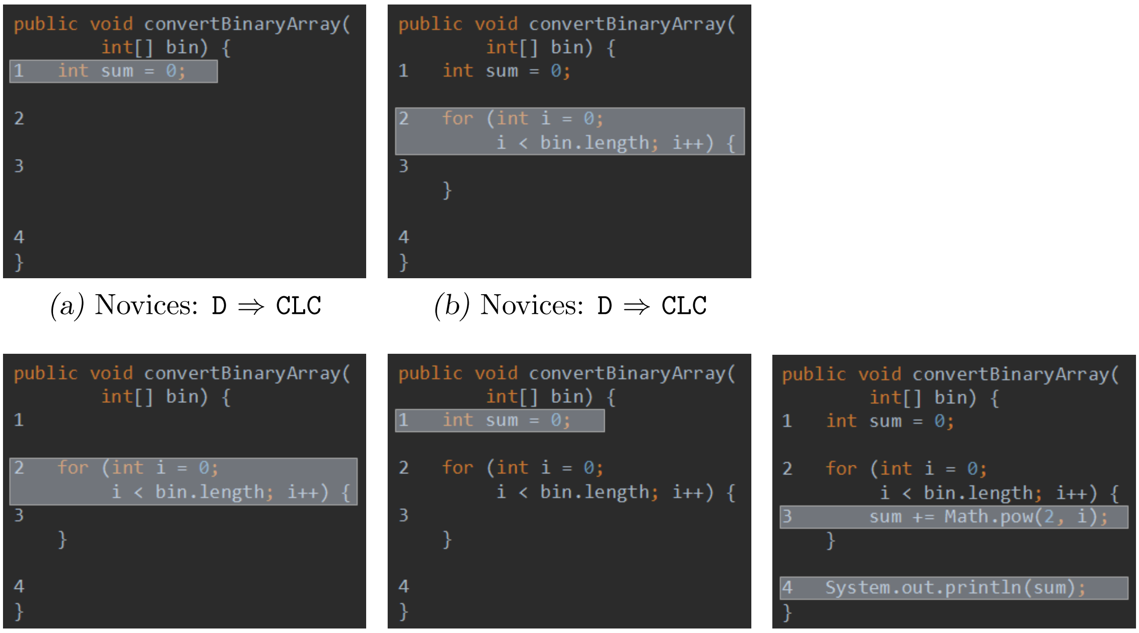
It has to be noted, however, that such an in-depth feedback regarding the program construction process necessitates that the students already possess a reasonably high level of programming skills. We suggest that students can best benefit from this feedback when operating on a firm *concrete operational* stage and can thereby be supported in developing towards *formal operational* reasoning [KB21]. This observation is true for all points derived in this section and the next one.

The second focus point is to derive benefits for instructional strategies from the results of my LA studies, particularly focusing on the potential of adjusting programming instructions to adhere to my findings regarding the specific construction order of loop constructs employed by professional programmers.

To illustrate my findings before deriving recommendations for instructional strategies, I first showcase the results of my LA studies regarding the order of program construction. Two different orders of program construction are shown in Figure 6.2, which represent a solution to the decimal conversion of the problem.

The program construction steps associated with novice programming skills, shown in (`a--b`), represent an ad-hoc approach to constructing the solution. A variation observed in the students' program construction, but not verified during the sequential analysis, is to also construct the change of line 4 before constructing the loop. This way, the data variable is fully prepared (initialization in 1 and use of computed value in 4) before constructing the actual computation. Interpreted, the students move from the concrete and easy-to-implement variable, for which they are sure to need it, to the abstract syntactic construct of looping through the data structure.

The program construction steps associated with expert programming skills, shown in (`c--d`), represent a planned approach to constructing the solution. They prepare the syntactic construct of looping through the data structure before taking care of the concrete individual variables to stuff out the algorithmic skeleton of the solution draft. In the post-interview, several professional programmers stated that they were immediately aware of the need to implement a loop to access the data



```
public void convertBinaryArray(
        int[] bin) {
1   int sum = 0;

2

3


4
}
```

(a) Novices: `D` ⇒ `CLC`

```
public void convertBinaryArray(
        int[] bin) {
1   int sum = 0;

2   for (int i = 0;
           i < bin.length; i++) {
3
    }

4
}
```

(b) Novices: `D` ⇒ `CLC`

```
public void convertBinaryArray(
        int[] bin) {
1

2   for (int i = 0;
           i < bin.length; i++) {
3
    }

4
}
```

(c) Experts: `CLC` ⇒ `D`

```
public void convertBinaryArray(
        int[] bin) {
1   int sum = 0;

2   for (int i = 0;
           i < bin.length; i++) {
3
    }

4
}
```

(d) Experts: `CLC` ⇒ `D`

```
public void convertBinaryArray(
        int[] bin) {
1   int sum = 0;

2   for (int i = 0;
           i < bin.length; i++) {
3       sum += Math.pow(2, i);
    }

4   System.out.println(sum);
}
```

(e) Common next changes

Fig. 6.2: Simplified solution to the `ConvertBinary` problem that showcases the different construction orders found in my studies. (`a--b`) show the construction order of novices: loop construction **after** data definition. (`c--d`) show the construction order of experts: loop construction **before** data definition. (`e`) shows common subsequent changes, in no fixed order, to solve the decimal part of the problem.

structure, demonstrating their ability to '*recognize problems*' in a fast and accurate way [Win96, p. 18]. Interpreted, the problem's *algorithmic affordance* influenced the experts' tactical and strategical plans to approach the program construction.

Turning to a recommendation for instructional strategies in programming education, the evidence that expert programmers construct program code in a specific order can provide an incentive to structure instructions in a new way: program construction that focuses on the view of accessing data structures and focuses on implementing syntactic constructs necessary for the target algorithm. To put it bluntly, I recommend to include programming instructions that have learning programmers deliberately implement abstract constructs (e.g., control structures, accesses to data structures) before concrete constructs (e.g., variables, print statements).

Take note, however, that this recommendation does not aim to mimicking expert programmer behaviour or enforcing a specific order of program construction just for the sake of it. Much rather, learning programmers should experience different ways to approach a problem solution strategically. This recommendation likely is different compared to novice programmers' first programming attempts: starting with concrete variable introductions before tiptoeing towards more abstract constructs. Moreover, this recommendation lends itself to instructions that facilitate the development of programmers' abstract reasoning, as they are urged to plan a strategical approach to the problem solution.

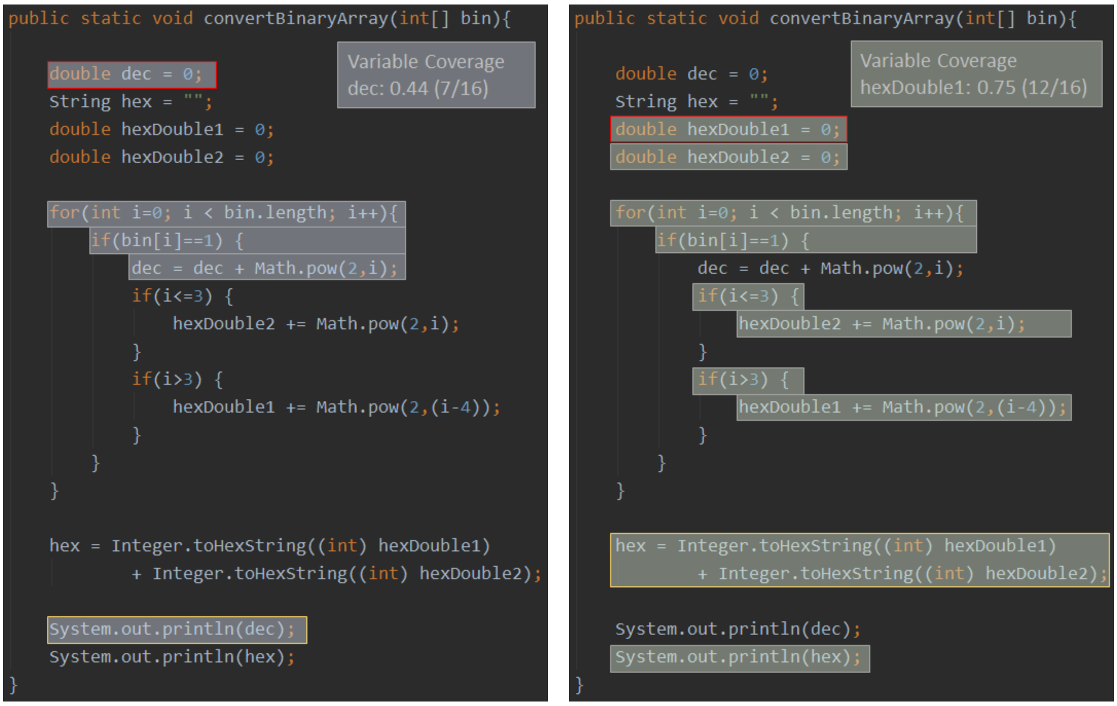
Concluding, my approaches and findings can provide a benefit for two facets of instructional strategies in programming education. The first facet is to facilitate individual assessment and feedback during students' task completion by providing educators with live LA monitoring, including automatically processed assessments and representations of the students' program construction. Proper guidelines and representations help educators to identify *where to look at*, which in turn can more easily support students during skill acquisition. The second facet is to design instructions based on the findings of my LA studies that expert programmers employ a specific order of program construction. The key is to let students experience different ways to approach program construction, which should open them up to new cognitive processes of knowledge construction.

6.2.4 Learning Analytics Interventions and Task Design

For the third practical contextualization, I take a look at how my LA approaches and the results of my studies can be used in task design for programming education. I specifically focus on task design tied to the programming IDE, as this context best facilitates data collection for use in tasks and data collection to design potential future tasks.

Hundhausen et al. provide a detailed process model and a number of taxonomies to guide the process of designing IDE-based *interventions* [HOC17]. In their frame, interventions are events that share '*some combination of information, guidance, and feedback*' [HOC17, p. 12] with learners to (hopefully) positively influence future learning. In this section, I frame my contextualization in the three dimensions established by these authors: *content* (**what** is presented to the learner?), *presentation* (**how** is the intervention presented to the learners?), and *timing* (**when** is the intervention presented to the learner?) [HOC17]. I describe two related concepts for IDE-based interventions that are enabled by the use of processed LA data. These concepts have previously been described in a workshop publication [KB21].

The first concept is the use of live LA metrics in IDE-based interventions to augment students' information during programming. As an illustrative example, consider the showcases of cohesion information shown in Figure 6.3. The two program snippets show an overlay of variable-specific union slices for two variables (`a`: `dec`, `b`: `hexDouble1`) on top of the included source code statements, show the forward criterion with a red border and the backward criterion with a yellow border, and also show the overall variable *Coverage* measure. At a glance, such a representation can provide visual feedback on the cohesion of a method and can also support



```
public static void convertBinaryArray(int[] bin){

    double dec = 0;
    String hex = "";
    double hexDouble1 = 0;
    double hexDouble2 = 0;

    for(int i=0; i < bin.length; i++){
        if(bin[i]==1) {
            dec = dec + Math.pow(2,i);
            if(i<=3) {
                hexDouble2 += Math.pow(2,i);
            }
            if(i>3) {
                hexDouble1 += Math.pow(2,(i-4));
            }
        }
    }

    hex = Integer.toHexString((int) hexDouble1)
            + Integer.toHexString((int) hexDouble2);

    System.out.println(dec);
    System.out.println(hex);
}
```

Variable Coverage
dec: 0.44 (7/16)

```
public static void convertBinaryArray(int[] bin){

    double dec = 0;
    String hex = "";
    double hexDouble1 = 0;
    double hexDouble2 = 0;

    for(int i=0; i < bin.length; i++){
        if(bin[i]==1) {
            dec = dec + Math.pow(2,i);
            if(i<=3) {
                hexDouble2 += Math.pow(2,i);
            }
            if(i>3) {
                hexDouble1 += Math.pow(2,(i-4));
            }
        }
    }

    hex = Integer.toHexString((int) hexDouble1)
            + Integer.toHexString((int) hexDouble2);

    System.out.println(dec);
    System.out.println(hex);
}
```

Variable Coverage
hexDouble1: 0.75 (12/16)

(a) Union slice & coverage for `dec`

(b) Union slice & coverage for `hexDouble1`

Fig. 6.3: Two showcases of augmenting program code with variable-specific union slices and variable coverage information during construction. Slicing criteria (red border: forward criterion, yellow border: backward criterion), included statements (grey border), and coverage information can be displayed as an overlay to facilitate a programmer's reasoning and design during program construction.

pinpointing parts of the program code that need refactoring to improve cohesion. Besides overlay information, graphical representations can also be used to show students diagrams, evaluations, and evolutions of processed metrics (e.g., *Coverage* of each individual variable), supporting them to improve their program code.

The first concept now encompasses providing such a representation of processed LA data to students during program construction to facilitate their development of expert programming skills. In the three dimensions of Hundhausen et al. [HOC17], this IDE intervention can be categorized as follows:

- **Content**: In the basic sense, the processed LA data provides **information** on a program's metrics that are not easily available. To make productive use of this information, learners should additionally be supplied with automatically processed **critique**, hinting at program parts that cause the low cohesion.
- **Presentation**: The processed data can be presented as an overlay **visual-**

ization in the coding window, as shown in Figure 6.3. Additional graphical representations can be presented unobtrusively in add-on windows, supported by most modern IDEs. **Critique** can be shown unobtrusively (for example, as sidebar hints that can be viewed at will) or obtrusively (for example, as a pop-up).

- **Timing**: I envision this type of IDE intervention to be presented at the student's **request**, which signifies the student's cognitive act of being ready to engage in program design with the help of the cohesion information.

The second concept is to support the design of specific tasks with targeted metric values that can be computed from programming LA data. In the cohesion example showcased in Figure 6.3, a task design following this concept is to establish target values for the *Coverage* of specific variables (e.g., ≥ 0.80 for `dec`) and for the method slice intersection (e.g., > 0.50). Such a task design necessitates a restructuring or refactoring of the program code, potentially introducing additional methods to handle parts of the computation so that the target cohesion values can be met. When designing such tasks, care must be taken not to present them as *puzzle tasks* but to properly motivate them, for example, with the principle of *single responsibilities*. This task design provides unique opportunities for students to plan, experience, and practice the effects of their program changes. It is possible to facilitate the students' reasoning and planning of program construction and to support them to learn design principles in a quantifiable way. In the three dimensions of Hundhausen et al. [HOC17], this IDE intervention can be categorized as follows:

- **Content**: **Information** derived from processed LA data is shown to the programmer.
- **Presentation**: The processed data can be presented as **notification** of whether the target values are reached or not. For an effective learning task, this intervention should be coupled with an intervention that makes it possible to explore the values, like the one above.
- **Timing**: This type of IDE intervention can be presented at the student's **request** or in an **event-based** manner when turning in a solution attempt.

To conclude, there are two types of IDE interventions that can make use of my LA approaches to design novel and potentially helpful tasks for learning programmers. The first is to augment students' information during program construction to enable the use of processed LA data during program construction, aiming at supporting planning and reasoning processes not easily possible without access to this kind of data. The second is to make use of processed LA data to design and formulate specific tasks with targeted metric values, which provide a kind of stipulation under which program design, and subsequently, program construction, has to be executed.

These summarized concepts can complement each other, and both give learning programmers access to LA data, empowering them to use data from their own processes to engage in reflective activities and reasoning, which are key skills of expert programmers on the *formal operational* stage [CTAL12].

6.3 Summary of Contextualization

Here I summarize the two presented viewpoints of contextualizing my research approaches and results. Regarding the theoretical contextualization, my findings on the specific order of program construction employed by expert programmers can be embedded in both neo-Piagetian and cognitive load theory to motivate didactical approaches. The goal is to support *experienced programmers* in developing towards expert programming skills. The evidence that expert programmers construct program code differently should be put to use – but only after *novice programmers* develop the necessary skills to purposefully write program code.

Interpreted with the theoretical approaches, the neo-Piagetian notion states that learning programmers need to develop their skills to mentally execute (i.e., *trace*) program code before being able to write program code [Lis16]. Cognitive load theory also supports a notion of the development of cognitive structures, stating that learning programmings need to combine single thought elements into *schemas* before being able to approach more complex tasks [SVP98].

Following from these interpretations, there is a point in the development of programming skills when the now *experienced programmers* can solve code writing problems of higher complexity. This is precisely the point where my results can be put to use. These *experienced programmers* should not be **stuck** in solving programming problems intended for *novice programmers*, as this provides minimal support in developing expert programming skills like abstract reasoning. Instead, they should be confronted with the *expert* way of implementing and reasoning about program code: focusing on data structures and algorithms, a tactical and strategical approach to problem solving, and reasoning from the abstract to the concrete [Win96, CTAL12].

To utilize this recommendation, programming instructions can specifically showcase the program construction in the order exercised by professional programmers. The goal is not to let *experienced programmers* memorize the order of program construction just for the sake of it, but to nudge them towards developing programming skills beyond their current domain. In neo-Piagetian terms, this approach could *shatter* their ways of thinking to facilitate new cycles of accommodation [Lis16].

Regarding the practical contextualization, I identify three areas of programming education that can benefit from my approaches and findings. The first is designing and implementing an LA dashboard that enables educators to plan programming lessons, manage the LA data collection, and access processed LA assessments and representations during task completion. The mentioned factors support future research in LA (while planning, educators can add contextual information to the programming lessons) and educators' ability to provide in-depth feedback in a timely manner. My LA approaches already provide the basis for data collection and storage, and considerations regarding the implementation of an educator LA dashboard have been published [KWB20].

My approaches and findings aim to improve instructional strategies in programming education in two ways. The first one is to specifically help educators identify *where to look at* in students' program construction by supporting them with automatically processed assessments and visualizations of LA data. Benefits include early identification of struggling students based on their program construction or the possibility of individual feedback towards improvement. The second way to improve instructional strategies is to let learning programmers deliberately implement abstract constructs before concrete constructs (e.g., implement loop constructs and data structure access before individual variables and print statements), which is in accordance with the summary of theoretical contextualization and aims to facilitate the programmers' development of expert programming skills.

The third area of practical contextualization is the design of specific tasks, supported by IDE-based LA interventions. This area is grounded on the idea to show learning programmers the processed representations and assessments of their own program construction, empowering them to engage in (reflective) reasoning about their program construction and program design. This reasoning can be fostered through specific tasks that aim at specific target values of program construction that can be measured with LA approaches. An example is to restructure a given piece of program code to meet target values of cohesion metrics, which necessitates specific refactoring and can be used to teach the principle of *single responsibilities* in a quantifiable manner [KB21].

6.4 Deriving Programming Skills from Program Construction Sequences

After contextualizing my approaches and results, I can now tie together all findings to arrive at answers to the main research question `MRQ` of this thesis.

MRQ. To what extent can patterns of program construction sequences be indicative, whether positive or not, of the acquisition of expert programming skills?

A key assumption that has driven my research is that there are factors in program construction that characterize how expert programmers write program code, which have not been elicited so far. In other words, my assumption was that there are discernible regularities, *patterns*, in the sequential construction of program code, executed by expert programmers, that can be put to use in improving programming education for learning programmers to become experts faster.

Another assumption that has driven my approaches of methodological design to answer this research question is that the *patterns* I intend to uncover might be **latent** in the sense that they might not be apparent to the experts making use of them. This is grounded in the fact that programming strategies [dRWT09], elementary patterns of program design [Wal03], and software design patterns [GHJV95] are well known and part of programming education in different capacities, but there are no accounts of *patterns* in sequential program code changes indicating expert programming skills. Therefore expert programmers may not be aware of their specific approach to program construction and cannot verbalize it.

In order to tackle the main research question, I implemented learning analytics approaches for data collection (IDE-based instrumentations) and data storage (uniform storage of sequential program code changes) to record the process of program construction in an unobtrusive way, focusing on sequential changes to the program code. Notably, the resulting data is more fine-grained compared to sequential program versions found in version control systems. My approaches record the program versions not on the basis of committed versions but on keystroke granularity, which is later aggregated to form a sequence of compilable program versions.

With this methodological approach, it was possible to record the program construction of programmers of different skill levels and analyze it with regard to *patterns*. In this research, I specifically focus on variables as centerpieces of program construction. Following an exploratory study of block-based and text-based program construction, I define a *variable construction pattern* in the following way: '*A typed, ordered, non-continuous sequence of program increments with changes to a variable that are semantically related*'.

With this definition, I conducted a mixed methods study to derive a catalog of *patterns* of variable construction that are used by student and professional programmers in two specific example problems. The result of this mixed methods study is a multilabel classification approach that makes it possible to classify each control variable and data variable with multiple *micropatterns* according to their role in the constructed program code. Furthermore, this approach makes it possible to classify each specific program change in the sequential program construction with one or more labels denoting the roles that are affected by the change.

With these methodological foundations, I conducted a comparative study to assess the use of *variable construction patterns* of novice programmers (undergraduate and graduate students) and expert programmers (professional programmers from industry and research).

The results of significant and substantial differences, evaluated with three null hypotheses, are twofold and discussed below.

1. Usage statistics of patterns are **NOT** indicative of expert skills

Two of the three null hypotheses relate to evaluating the use of *variable construction patterns*. I thereby assess whether the use of specific patterns for individual variables or for the set of all variables, aggregated on the level of the implemented method, are indicative of expert programming skills. In other words, I assess whether expert programmers use significantly different patterns in the construction of program code.

The results show that the related null hypotheses can only be refuted in specific points. Expert programmers do have significantly different usage for data variables that are dependent on multiple control contexts and also show significant differences in the overall use of loop-related control *micropatterns* on the level of methods. I attribute these differences to the experts' skills to planning and managing variable usage in a more nuanced way and not using extraneous variables.

While there are significant differences between novices and experts, a control test reveals that the differences in patterns usage is much more pronounced when comparing the two specific example problems. For nearly all of the involved *micropatterns*, there are significant differences in the construction of the two different methods. I relate this result to differences in the *algorithmic affordance* of these different problems. This means that the problems are substantially different on the level of the required algorithmic solutions, which in turn influences the syntactic constructs needed to implement a problem solution and the semantic roles individual variables need to be constructed for. In other words, the different problems *need* their syntactic constructs and variables to be differently constructed. This result puts the findings of significant differences between novices and experts into context and is evidence that usage statistics of patterns alone are **NOT** indicative of expert programming skills.

I believe, however, that there is still untapped potential in eliciting expert programming skills that are directly related to program construction but cannot be easily obtained from recorded sequences of program changes. Such expert programming skills that influence program construction, following Winslow [Win96], are:

- The fast and accurate identification of the *algorithmic affordance* of a program to be implemented. That is, recognizing the data structures, their related algorithmic approaches, and accessing knowledge structures for similar solutions.

- The formation of an implementation plan with a strategical approach. Such a plan might be formed on paper, in design documents, or in the programmer's mind. Too often, they are not elicited in related research.

- Considerations of non-functional requirements that can influence and alter how a specific problem is solved with regard to the used syntactic constructs and variables. This can include the pursued levels of memory and time consumption, influencing the syntactic constructs selected for implementation. Also, more abstract design decisions like architectural design patterns can influence the algorithmic design of specific methods.

Future work should include features of these programming skills in their measurement with the goal of making them accessible in education and training of learning programmers, facilitating the overall acquisition of expert programming skills.

2. The sequential order of used patterns **IS** indicative of expert skills

The third and last investigated null hypothesis relates to the sequential use of *variable construction patterns* in the program construction sequence. Within my devised framework of *multilabel* classification, it is possible to assign one or more typed labels to each sequential variable change. From these classifications, frequent sequential rules can be obtained through data mining to uncover substantial differences between novices and experts in the order of program construction. The frequent rules are measured based on support and confidence.

The frequent rules provide evidence of a substantial difference in the order of program construction of loop constructs and dependent data variables. For novice programmers, the strongest rules with a maximum support of 0.5 and a maximum confidence of 0.6 show that these programmers implement the loop construct (which was most often a `for` loop in the example problem) **after** declaring and initializing a data variable that is later used as a dependent variable in the loop control context.

For expert programmers, the strongest rules with a maximum support of 0.5 and a maximum confidence of 0.88 provide evidence for a different order of program construction order. These programmers implement the loop construct **before** declaring and initializing the data variable above the loop header.

The difference in order can be interpreted with theoretical findings of related work. In the context of expert programming skills, given by Winslow [Win96], beginning with the loop construction demonstrates expert skills by approaching a problem through the data structures (in this case, looping over the input array). Beginning with the concrete data variable demonstrates novice programming skills by approaching the problem with a bottom-up strategy and working backward from the goal to determine concrete solutions.

In the context of the neo-Piagetian hierarchy of programming skills, given by Corney et al. [CTAL12], the experts' order of program construction demonstrates reasoning on the most abstract stage of cognitive development, the *formal operational* stage. This stage encompasses abstract reasoning and beginning with the abstract before moving to the concrete – in this case, implementing the abstract, syntactic construct to loop over the data structure before implementing a concrete data variable. On the other hand, novice programmers do not demonstrate *formal operational* reasoning as they begin with implementing the concrete data variable.

Interpreted with these theoretical notions, the substantial differences between the frequent rules of program construction of novices and experts provide evidence that the sequential order of used patterns **IS** indicative of expert programming skills.

A caveat of this second result is that only the program construction sequences from a single example problem have been used to identify the differences in the order of program construction. Whether this specific order of program construction can be generalized to other problems or to other combinations of syntactic constructs has to be assessed with future work. Moreover, future work can assess whether there are unelicited factors that influence the experts' program construction – essentially asking: *What influences expert programmers to favor a specific order of program construction?* Answers to such questions can help inform instructional design for future learning programmers.

Even with unanswered facets, my results provide evidence that expert programmers approach program construction differently – measurable in the sequence of program construction with patterns of variable use. Knowledge about the favored order of program construction of expert programmers for a class of algorithms (my results show evidence for a summation loop) can be used in education and training to help experienced programmers develop faster towards expert programming skills. Educators can prepare programming instructions that showcase the experts' program construction order, not to have the learners memorize the specific order of construction but to provide them with stimulus to approach problems differently and develop towards (expert) abstract reasoning. Learners, however, need to be receptive to this kind of reasoning and likely need to operate on the level of *concrete operational* reasoning already – being capable of writing code for a purpose [Lis16].

7. CONCLUSION

In this thesis, I present an approach to measure differences in programming skills with a detailed investigation of program construction sequences. I establish the notion of *program construction patterns* as regular, discernible sequences of typed and labeled program changes. My work provides evidence that expert programming skills manifest in the order of individual, sequential changes to program code. My results and accompanying methodological and technical approaches provide a direction for future work in computer science education to support the acquisition of programming skills based on the actual program construction. Especially, my work furthers the understanding of how experienced programmers can be supported to develop towards expert programming skills.

In this last chapter, my research is summarized in Section 7.1, including a description of my research aims, the defining features of my approaches, and summarized answers to the main research question and the four research questions. Lastly, I describe avenues for future work that build on the approaches and results of my thesis in Section 7.2.

7.1 Research Summary

My research is grounded in the aim to support the acquisition of programming skills for learning programmers on an individual level. Specifically, the aim is not only to support novice programmers in learning fundamental skills of programming until they can write purposeful program code [Lis16], but also to provide support for experienced programmers to individually develop their skills toward becoming an expert. As such, my research works on the gap between computer science education research (e.g., supporting novice programmers in introductory programming courses [LRSA+18]) and software engineering research (measuring and quantifying programming skills [BSD14, SKL+14]).

Towards this aim, a question is how the level of programming skills is manifested in activities related to programming – that is, what are measurable, defining features of expert programmers, and how can the related evidence of expert programming skills be uncovered?

With the rise of learning analytics, the means to collect and process large amounts of data is becoming more widespread in educational settings [IBE+15].

Within this methodological context, I approach the questions given above by giving close attention to the program construction process to measure and uncover evidence of developed programming skills. My approach encompasses a detailed investigation of how programmers of different skill levels construct their programs.

The underlying hypothesis is that expert programmers (who have already acquired expert programming skills) follow regular, discernible sequences (i.e., *patterns*) during program construction in order to implement a program efficiently and correctly. These sequences are dependent on different factors subject to expert reasoning (e.g., constraints on time or memory consumption, or on decisions related to software system architecture), but also heavily depend on the *algorithmic affordance* of the problem to be implemented – that is, the collective requirements on the sets of syntactic constructs and related semantic constructs needed to implement the problem.

Summarized, my underlying hypothesis is that programmers of varying skill levels create programs in a measurably different way. And if there are indeed *construction patterns* that are favored by expert programmers, perhaps because these patterns result in faster construction or more correct programs, it can be beneficial to elicit these patterns and use them to improve education and training. As described below, my research provides evidence that this hypothesis holds true, proven for the construction of simple loops.

Following the described aims and the underlying hypothesis, my main research question is:

MRQ. To what extent can patterns of program construction sequences be indicative, whether positive or not, of the acquisition of expert programming skills?

An answer to this main research question provides an answer to whether the analysis of *program construction patterns* provides a meaningful differentiation between programmers of different programming skill levels. To arrive at an answer to the main research question, I sequentially worked on four research questions.

The first research question is related to uncovering the nature of programming skills and what generally distinguishes novice programmers from expert programmers:

RQ1. What distinguishes novice from expert programmers?

I employed a literature study presented in Chapter 2 and arrive at two focal points to answer the first research question. I interpret programming skills as cognitive, domain-specific abilities following Weinert [Wei99] for the first focal point. Expert programmers are distinguished by improved abilities in the following five dimensions: i) experts possess, apply, and easily acquire **mental models of program**

structures, ii) experts possess and apply **better hierarchical, syntactical, and semantical knowledge**, iii) experts can tactically apply **problem solving strategies**, iv) experts approach problems through **data structures and objects in an algorithmic, top-down way**, and v) experts are **faster** and **more accurate** when performing programming tasks.

For the second focal point, I interpret the cognitive abilities in programming with the neo-Piagetian model of overlapping waves established by Lister [Lis16]. The process of skill acquisition in learning programmers is modeled in four stages.

I regard the first two stages as actual **novice programmers**: learners at the *sensorimotor* stage struggle with basic programming concepts, learners at the *preoperational* have learned the basic programming concepts and can mentally execute (i.e., *trace*) programs with specific inputs to inductively reason about them.

I regard learners who are firmly at the third stage as **experienced programmers**: learners at the *concrete operational* have developed the skills necessary for *abstract tracing*, being capable of reasoning deductively about code, and writing code for a purpose.

Finally, learners at the fourth stage are regarded as **expert programmers**: learners at the *formal operational* can reason reflectively and abstractly in unfamiliar situations.

Concluding, expert programming skills are represented in a variety of cognitive, domain-specific abilities, with expert programmers not only being faster, more accurate, and more knowledgeable, but also having qualitatively different cognitive abilities. Of note is that while the skill acquisition of **novice programmers** is widely researched and readily mapped out with empirically verified transitions (see the research catalog of Lister [Lis16]), the support of **experienced programmers** to develop expert programming skills is not widely researched and summarized by Lister as '*having them write lots of code*' [Lis16, p. 14]. With my research, I carve out some characteristics of programming skills along the road to becoming an expert.

The second research question covers the definition of patterns:

RQ2. How can program construction patterns be defined?

Towards answering this research question, I first described my learning analytics approach in Chapter 3, comprising i) the development of recording and instrumentation tools to support IDE-based learning analytics research [HOC17] for block-based (`Scratch 2`, `Scratch 3`) as well as text-based (`IntelliJ`) programming environments and ii) the development of a uniform data collection server to facilitate the collection and distribution of program construction data.

With this IDE-based learning analytics approach, it is possible to record and analyze sequentially evolving versions of a program under construction, which are program construction sequences. I conducted an exploratory study, recording the program construction sequences of school pupils, university students, and professional programmers, to analyze different features of program construction that can be measured in program construction sequences described in Chapter 4.

The first answer developed in Chapter 4 is the definition of program construction patterns based on **syntactic** and **semantic** features of program construction. The generic definition of program construction patterns is: *a typed, ordered, non-continuous sequence of program increments that are semantically related.*

For the context of my thesis, I instantiate this definition to pay closer attention to program construction through the lens of individual variables. I thus define variable construction patterns as *a typed, ordered, non-continuous sequence of program increments* ***with changes to a variable*** *that are semantically related.* This definition is shaped in the context of my learning analytics approaches. I also give a colloquial definition of variable construction patterns: *a typed sequence of compilable program changes that affect related program statements of a variable.*

The third research question builds on the definition given above and aims to identify specific patterns used by programmers during program construction:

RQ3. What program construction patterns are used by novice and expert programmers during program construction?

Following the exploratory study, I conducted a mixed methods study with recorded program construction sequences of university students and professional programmers with two analysis phases, also described in Chapter 4. I first employed a qualitative research method (Grounded Theory) to identify defining features of specific variable construction patterns. I employed the qualitative analysis method in a novel way by analyzing figurative representations of program construction sequences (see Appendix B for such representations, called variable-specific *construction flowlines*).

The results are a catalog of micropatterns for control variables and data variables, with each micropattern capturing one role a variable fulfills in an implemented method. The control micropatterns include: *loop-condition*, *loop*, *conditional*, *computation.* The data micropatterns include: *loop-dependent*, *conditionally-dependent*, *redefined*, *self-defining*, *single-scope-use.*

In the second analysis phase, I quantitatively evaluated the distribution of different variable construction patterns, which are sets of assigned micropatterns per variable, for novices and expert programmers. This evaluation provides answers to the third research question.

Both groups of programmers routinely use control variables for data processing (micropattern *computation).* Besides this finding, patterns related to the use of control variables are highly dependent on the *algorithmic affordance* of the problem. In my example problems, mostly loop-related control variables have been used.

While data variables are also dependent on the *algorithmic affordance*, the algorithmic strategy [Sol86] pursued by the programmers plays a bigger role in the construction of these variables. Expert programmers use data variables in multiple control scopes (micropatterns *loop-dependent* **and** *conditionally-dependent*); in contrast, novice programmers do not. Besides this finding, both groups of programmers employ *self-defining* variables to solve the example problems.

The fourth research question builds on the catalog of control and data micropatterns and aims to investigate them with regard to their discerning qualities towards programming skills:

RQ4. What are the differences between novices and experts concerning the use of program construction patterns?

To answer this research question, I employed a comparative study on the recorded program construction sequences of university students and professional programmers using quantitative methods (inferential statistics, machine learning, association rules mining), described in Chapter 5. I state three null hypotheses to compare the use of micropatterns and variable construction patterns: i) on the level of **individual variables**, ii) aggregated on the level of **methods**, and iii) pattern usage in the **program construction sequence**.

Regarding pattern usage for **individual variables**, there are hardly any significant differences between novices and experts. The significant differences, measured with a U-test and $p < .05$, show that expert programmers have a more nuanced use of variables. This nuanced use is evident in a higher usage fraction of *loop-dependent* data variables and in variable construction patterns that capture data variables in multiple control contexts.

Regarding pattern usage aggregated on the **level of methods**, there is only one set of significant differences. Expert programmers have a significantly higher aggregated use of loop-related control micropatterns, measured with a U-test and $p < .05$, which can be interpreted as better management of variables and not using extraneous ones.

Regarding pattern usage in the **program construction sequence**, there is a substantial difference in the program construction order, found in frequent association rules mined with SPADE. The strongest rules feature a **support** of 0.5 and a confidence of 0.6 for novices and of 0.88 for experts. These strongest rules provide evidence that there is a difference in the favored construction order of loop constructs and loop-dependent data variables. Novice programmers tend to construct the loop construct (in the example problems, mostly `for` loops) **after** constructing the initial definition of the data variable. Experts tend to construct it the other way around:

the loop construct **before** the initial definition of the data variable. This difference can be interpreted as expert programmers first focusing on data structures and abstract reasoning, both evidence of expert programming skills [Win96, CTAL12].

Summarizing an answer to the fourth research question, micropatterns and variable construction patterns provide measurable evidence of differences in programming skills. While the use of variables, measurable with micropatterns, is related to the *algorithmic affordance* of the problems, experts feature a more **nuanced use of micropatterns** compared to novices.

Furthermore, both novices and experts favor a specific construction order when constructing loop headers and loop-dependent data variables, which give information about their respective skill levels. Experts first construct the loop header, focusing on the abstract data structure. Students first construct the data variable, focusing on a concrete bottom-up approach to problem solving.

With all research questions answered, I turn back to the main research question:

MRQ. To what extent can patterns of program construction sequences be indicative, whether positive or not, of the acquisition of expert programming skills?

I provide a contextualization of my approaches and results in Chapter 6, representing a theoretical and practical interpretation of my results and potential applications in the future. With proper contextualization of my findings, I arrive at two answers to the main research question.

The first answer is that usage statistics of variable construction patterns alone are **NOT** indicative of expert programming skills. While there are some significant differences in pattern usage between novices and experts, the difference in pattern usage is much more pronounced between different example problems, with significant differences in most micropatterns both on the level of individual variables and aggregated on methods. I, therefore, regard the differences measurable with variable construction patterns not as a sole indicator of expert programming skills. However, measurable differences that hint at expert programming skills are better overall management of control variables (not using extraneous ones) and more nuanced strategies of data variable use (using them in multiple control contexts as needed).

However, the second answer is that the sequential order of used patterns **IS** indicative of expert programming skills. This answer stems from the fact that specific different construction orders for both novice programmers and expert programmers have been found that can be attributed to the respective level of programming skills. This answer at least holds true for the problem class of the example problem (summation loop) and provides evidence that: i) variable construction patterns provide a means to assess program construction with regard to programming skills, and ii) expert programmers tend to follow a regular, discernible sequence (*pattern*) during program construction, planned or unbeknownst to them.

Finally, I draw the conclusion back to my initial aims. In the context of individual support, my findings provide a novel basis for the support of programming skill acquisition. Educators can use the knowledge of the favored order of program construction of expert programmers, thereby specifically facilitating the development of the type of reasoning about program construction that is associated with expert programming skills. This could make the road to mastering programming faster and more accessible.

My findings also illustrate that, while most of my learning analytics approaches are experimental and not production-ready, there is the potential towards an improved programming education within the context of learning analytics and with data processed from program construction sequences. Potential avenues for future work are described in the following Section 7.2.

7.2 Future Work

In this section, I present aspects of my work that provide an avenue for future work to improve programming education.

7.2.1 Evidence of Programming Mastery

My results provide evidence that expert programmers favor a specific order of program construction when constructing a loop with one or more control variables and one loop-dependent data variable. Their order of program construction, first implementing the loop construct before the data variables, represents expert reasoning: approaching the program through data structures [Win96] and beginning with the abstract [CTAL12]. Future work can develop in two directions to expand and make use of these findings.

The first direction is to design experiments to further investigate expert programmers' use of *program construction patterns* and of specific construction orders on different algorithmic classes of problems. On the one hand, such experiments can include problems that afford specific algorithmic solution strategies, for example nested control structures. On the other hand, such experiments can also elicit the experts' reasoning process of program design, which can, in turn, uncover factors that influence the experts' reasoning during program construction. I expect that making these factors explicit can contribute to improving programming education.

The second direction is to make use of the findings of expert programming skills in adapted instructional strategies for learning programmers. As established in Chapter 6 by interpreting my findings within the theoretical notions of neo-Piagetian theory and cognitive load theory, educators can design programming instructions and tasks that focus on the reasoning and construction order of expert programmers: constructing data structures and abstract program parts before concrete ones. This recommendation does not aim to suggest memorizing the construction order but to demonstrate the process of program construction to learning programmers in a way that is often drastically different compared to novice programming, thereby making them challenge their programming habits and reasoning. This notion of instructional design for programming education represents a novel way to support experienced programmers in developing programming skills on the level of experts.

Researchers can design experiments to evaluate whether instructional strategies following the notions introduced above actually provide a benefit in the long-term acquisition of programming skills towards the level of experts.

In the context of block-based programming education, such an instructional design could also be used. There are, however, limitations stemming from the structure of block-based programming that impede the direct transfer of my results, such as the consolidation of individual syntactic constructs to a single semantic unit, a single block. Nevertheless, an instructional design can still focus on letting learners engage with approaches attributed to expert reasoning, for example, by including design and modeling activities and relating them to block-based program code with abstract reasoning in mind.

7.2.2 Towards Automatic Labeling of Programming Sequences

Regarding the generalization of my learning analytics approaches, a limitation in my toolset that was developed to process and analyze program construction sequences[1] is the lack of a fully automated system to provide typing and labeling of the individual program changes for each variable. An automatic analysis approach provides the potential for generalizing to a higher number of program construction sequences to improve the strength of future experiments.

There are two key points that currently prevent full automation, both related to the syntax tree differencing algorithm `GumTree` [FMB+14] that is used to identify changes between two consecutive compilable source code versions. With this algorithm, change parts are incorrectly mapped in some circumstances when multiple similar syntactic constructs are present, and one of them is changed (e.g., multiple uses of a variable). Moreover, small changes to the AST are not consistently recognized (e.g., changes in the arithmetic operator in computations or changes to boolean evaluations in conditional statements). Both shortcomings contribute to unreliable automatic processing of program construction sequences with the need to manually correct the identified changes.

[1] Link to the repository: `https://gitlab-iid.aau.at/seqtrex/pattern-browser`

The second point that prevents full automation is the lack of semantic information to automatically assign micropatterns to variable changes. While `GumTree` can provide information on which source code parts have changed, another analysis pass is needed to semi-automatically assign most of the micropatterns. For some structures, like `for` loops, it is not possible with the information of `GumTree` to reliably identify the syntactic parts that have changed as well as their semantic purpose.

Summarizing, to enable an automatic analysis of program construction sequences with the notions introduced in my thesis, a combined approach is needed that reliably identifies fine-grained changes of sequential source code versions while also providing semantic information. A possible step in the right direction is the approach of Nguyen et al. [NND$^+$19], utilizing the structure of augmented program dependence graphs that they call fine-grained PDGs (`fgPDG`) to combine change mapping with structural and semantic information of the program under construction.

7.2.3 Learning Analytics Dashboard and Interventions

My approaches and findings can be put to use in Learning Analytics (LA) research in two directions, introduced in Chapter 6. The first direction encompasses the use of processed LA data of the program construction process in IDE-based LA interventions [HOC17, KWB20, KB21]. Programming education is at a unique point in that the main working environment of IDEs can be augmented with LA data to provide additional opportunities for learning experiences. This direction of research makes it possible to augment the data accessible to learners during programming, providing them with fast and individual feedback on the one hand and empowering them to use data from their own processes to engage in reflective activities on the other hand. The latter is specifically a part of expert programming skills [CTAL12].

To provide concrete options for future work, plugins for IDEs can be developed to display IDE-based LA interventions to the programmers during program construction. These interventions can take the form of information overlays, task assessments (e.g., constructing a program with specific, monitored static metrics like high method slice intersections), or visualizations of different metrics of the program construction process. This research avenue provides educators with additional tools to improve programming education and researchers with new experimental setups to evaluate the effects of interventions on the acquisition of programming skill.

The second direction is implementing a LA dashboard that provides educators the possibility to plan programming lessons enriched with LA data. A key factor that is currently lacking in LA data in programming is contextual information [BASK18]. We outlined defining features of such an *educator dashboard* in a publication [KWB20], and I expanded on them in Chapter 6.

An *educator dashboard* provides unique opportunities to help educators improve programming education with the use of processed LA data. A key improving factor is that LA data provides the opportunity for educators to give real-time and in-depth feedback on an individual basis. Moreover, such a dashboard facilitates the collection of contextualized program construction data, and provides access to this data to research – thereby providing an interface between educators and researchers.

I envision that contextualized and shared program construction data is a key to assessing the generalizability of findings in programming education. Hypotheses could be evaluated on multiple sets of contextualized data to pinpoint the influence of different factors. In the long run, such an approach can prove essential to further the research field of computer science education.

7.2.4 Learning Analytics based on Cohesion Metrics

I worked towards developing the notions of *variable construction patterns* as regular, discernible sequences of program changes to variable statements. Aside from this notion, patterns in program construction can also be mapped to other dimensions of change types, potentially uncovering new aspects of understanding programming mastery. A possible dimension that is known to provide semantic information of program code encompasses slice-based cohesion metrics.

Bollin [Bol13] already showed that slice-based cohesion metrics for `Z` specifications can be used to elicit whether there are different trains of thought in a specification, which reduces cohesiveness. We, therefore, incorporated slice-based cohesion metrics on the level of individual variables to assess semantic features of the program construction, focusing on the evolution of variable-specific coverage measures and of the method slice intersection [KB21]. My preliminary results show that expert programmers converge to cohesive program versions faster, thereby consolidating their trains of thought during program construction faster.

This research approach can be extended, designing an experiment to investigate the evolution of trains of thought during program construction. Key factors to control are programming experience and a careful selection of programming examples (making it possible to observe multiple trains of thought during program construction). Example problems could feature fixed starting points with multiple trains of thought with the task of refactoring those into cohesive program parts (methods).

I envision that there is a difference in how experienced programmers and expert programmers approach and solve such a task, measurable with cohesion metrics. Comparable to my findings, a result could be derivable recommendations on how to facilitate faster skill acquisition in experienced programmers.

7.2.5 Incorporating Cognitive Metrics into Learning Analytics

The learning analytics (LA) approaches described in my thesis only include metrics related to the program content and program construction. There are, however, additional dimensions of metrics that can provide important insights into the program construction process to elicit how expert programmers work and reason during program construction. Hundhausen et al. [HOC17] already included cognitive metrics in their taxonomy of LA approaches.

In this thesis, I developed the notions of *program construction patterns* as regular, discernible sequences of program changes and therefore focused on metrics of the program content and construction. But I envision that additional evidence of expert programming skills can be found using cognitive metrics and that behavioural patterns can be derived to improve programming education for learning programmers. A dimension of cognitive metrics that gains increased popularity in SE research is the measurement of eye gaze metrics. The recently introduced tool `gazel` [FRP+21] could provide the technical means to incorporate eye tracking as a cognitive metric into the research of sequential program construction.

7.2.6 Summary

To conclude, I give a perspective on these avenues of future work in light of the neo-Piagetian hierarchy of programming skill levels [Lis16]. I differentiate between *novice programmers* that are not capable of consistent reasoning on the *concrete operational* level (not consistently capable of abstract tracing) and should be supported in developing this kind of reasoning and *experienced programmers* that are capable of doing so and should be supported in developing expert reasoning skills towards the *formal operational* level.

For *novice programmers*, many of my recommendations and future work suggestions might seem out of place. They do not (yet) have the cognitive capacities in the domain of programming to take advantage of processed learning analytics (LA) data to reflectively reason or improve in program construction. However, by providing them with additional data displays, timely and individual feedback, and instructional strategies that might challenge their developing cognitive structures, there might be a positive effect in their skill acquisition that should be researched.

Experienced programmers can be empowered in their skill acquisition with the factors mentioned above. These factors cannot take away the need to design and implement software systems, and to learn the technical knowledge. However, these factors could again positively affect the skill acquisition (of developing expert programming skills), which should also be researched.

Lastly, for educators and researchers, my approaches and findings provide a basis to apply new instructional strategies and research approaches: i) focusing on programming skills that facilitate expert reasoning during program construction, ii) using processed LA data and visualizations to provide timely and individual assessment and feedback, and iii) contributing towards generalizable research by collecting and sharing contextualized programming construction data.

APPENDIX

A. SLICE PROFILE ALGORITHM AND COHESION METRICS FOR VARIABLES

This appendix chapter contains technical information that has been migrated from the main text so as to not break the flow of text. The sections contain technical descriptions of the computation of slice profiles (Section A.1), and of slice-based cohesion metrics for variables (Section A.2). These technical descriptions complement the extraction of semantic features of program construction sequences described in Section 4.2.

A.1 Computation of Slice Profiles

The computation of slice profiles is based on a graph representation of code modules, called the program dependence graph (PDG) [OO84]. Source code statements are represented by nodes with variable references and definitions. Control and data flow dependencies are represented by edges in the graph. This representation reduces the computation of static slices to a graph reachability problem.

The algorithm to compute the *slice profile* of a method is shown in Algorithm 1. The algorithm operates on a method PDG as input, and returns the *slice profile* (a list of *union slices* for each variable) as output. Individual steps of the algorithm are described below, each with a reference to the corresponding statement lines.

`Line 2:` Union slices are computed for all local variables and input parameters in the method.

`Lines 4-6:` The following sets of nodes are initialized: `(4)` all nodes referencing the variable, `(5)` all nodes defining the variable, `(6)` all parameter formal-in nodes (none or one).

`Line 7:` Union slices are generated until all nodes of the previous step are included in at least one union slice.

`Lines 8-9:` A backward slice is computed with the last reference from the set of references. The last node is found by the highest statement line number.

`Line 10-15:` A forward slice is computed with the first definition from the set of definitions and parameters that is in the computed backward slice. If the backward slice is empty (e.g. when the set of references was empty), all definitions and parameters are considered. The first definition is found by the lowest statement line number, with parameters taking precedence.

`Line 16-20:` The union slice is the union of nodes of the forward and backward slices. All nodes of the union slice are removed from the respective sets. This ensures that subsequent union slices contain at least one reference or definition that is not in the previous union slices. For the slice profile, only nodes representing actual method statements are retained.

`Line 22:` When all references, definitions and parameters of the current variable are included in union slices, then the list of union slices is added to the method slice profile.

```
Input: Method $PDG$
Output: Slice Profile Map<Var, List<UnionSlice>>
 1: initialize slice profile $SP$
 2: for $v \in (PDG.vars \cup PDG.params)$ do
 3:    initialize list of union slices $SP_v$
 4:    $REF_v \leftarrow \{node \in PDG.stmt \mid v \in node.REFs\}$
 5:    $DEF_v \leftarrow \{node \in PDG.stmt \mid v \in node.DEFs\}$
 6:    $PAR_v \leftarrow \{node \in PDG.param \mid v \in node.EXP\}$
 7:    while $\neg(REF_v = \emptyset \wedge DEF_v = \emptyset \wedge PAR_v = \emptyset)$ do
 8:       $BW\_Crit \leftarrow last(REF_v)$
 9:       $BW\_Slice \leftarrow Backwards\_Slice(BW\_Crit)$
10:       if $BW\_Slice = \emptyset$ then
11:          $FW\_Crit \leftarrow first(DEF_v \cup PAR_v)$
12:       else
13:          $FW\_Crit \leftarrow first((DEF_v \cup PAR_v) \cap BW\_Slice)$
14:       end if
15:       $FW\_Slice \leftarrow Forward\_Slice(FW\_Crit)$
16:       $UnionSlice_v \leftarrow BW\_Slice \cup FW\_Slice$
17:       $REF_v \leftarrow REF_v \setminus UnionSlice_v$
18:       $DEF_v \leftarrow DEF_v \setminus UnionSlice_v$
19:       $PAR_v \leftarrow PAR_v \setminus UnionSlice_v$
20:       $SP_v.add(UnionSlice_v.stmt)$
21:    end while
22:    $SP(v) \leftarrow SP_v$
23: end for
24: return $SP$
```

Algorithm 1: Slice Profile Algorithm

The output *slice profile* can be used to compute slice-based cohesion metrics, typically for each method and with the adaptions described below for each variable and pairs of variables.

A `Java` slicer (called *JRazor* [Ram13]) was adapted to automatically compute the slice profile of compiled `Java` programs. The source code is publicly available[1].

A.2 Adaption of Slice-based Cohesion Metrics for Variables

A.2.1 Basic Definitions

Let M be the set of statement nodes of the method.

Let V be the set of local variables and input parameters of M. Let v be a variable or parameter in V.

Let SP_v be the list of all *union slices* for v, from slice profile SP of M (computed with Algorithm 1).

Let $UnionSlice(v)_i$ be the ith element of SP_v, that is, the *union slice* after the ith iteration of Algorithm 1 for v.

Let the intersection of *union slices* for v be:

$$UnionSlice(v)_{int} = \bigcap_{j=1}^{|SP_v|} UnionSlice(v)_j$$

Let the method slice intersection, the intersection of all union slices for M, be:

$$MethodSlice_{int} = \bigcap_{v \in V} UnionSlice(v)_{int}$$

Let the relative method slice intersection be:

$$RelMethodSlice_{int} = \frac{|MethodSlice_{int}|}{|M|}$$

A.2.2 Slice-based Cohesion Metrics for Single Variables

I adapt the slice-based cohesion metrics *Coverage*, *Overlap* and *Tightness* [OT93] to measure the cohesion for single local variables and input parameters of methods.

The first metric is *Coverage*, which compares the length of the union slices for a variable to the length of the method. *MinCoverage* and *MaxCoverage* compare the length of the shortest and longest union slice for a variable, respectively, to the length of the method.

Definition A.1 (Coverage(M,v))**.** Coverage for method M and variable v is the average of the variable union slice length divided by the method length.

$$Coverage(M, v) = \frac{1}{|SP_v|} \sum_{i=1}^{|SP_v|} \frac{|UnionSlice(v)_i|}{|M|}$$

[1] Source code available at: https://gitlab-iid.aau.at/seqtrex/jrazor

Definition A.2 (MinCoverage(M,v))**.** MinCoverage for method M and variable v is the ratio of the smallest variable union slice to the method length.

$$MinCoverage(M, v) = \frac{1}{|M|} \min_{i=1}^{|SP_v|} |UnionSlice(v)_i|$$

Definition A.3 (MaxCoverage(M,v))**.** MaxCoverage for method M and variable v is the ratio of the largest variable union slice to the method length.

$$MaxCoverage(M, v) = \frac{1}{|M|} \max_{i=1}^{|SP_v|} |UnionSlice(v)_i|$$

The second metric is *Overlap*, which compares the number of statements common to all union slices for a variable to the length of the method.

Definition A.4 (Overlap(M,v))**.** Overlap for method M and variable v is the average ratio of the length of the intersection of variable union slices to the length of each slice.

$$Overlap(M, v) = \frac{1}{|SP_v|} \sum_{i=1}^{|SP_v|} \frac{|UnionSlice(v)_{int}|}{|UnionSlice(v)_i|}$$

The third metric is *Tightness*, which is based on the number of statements included in every union slice for a variable.

Definition A.5 (Tightness(M,v))**.** Tightness for method M and variable v is the ratio of the length of the intersection of variable union slices to the method length.

$$Tightness(M, v) = \frac{|UnionSlice(v)_{int}|}{|M|}$$

Values of the three metrics *Coverage*, *Overlap*, and *Tightness* range from 0 to 1, with lower values indicating smaller cohesion in a method with respect to the union slices of a specific variable. Note that the following relations hold between the measures for variables with only one union slice:

$$|SP_v| = 1 \implies Overlap(M, v) = 1$$
$$|SP_v| = 1 \implies Coverage(M, v) = Tightness(M, v)$$

Because of these relations, the measure of *Coverage* is most universally expressive and is used to extract semantic features in this thesis.

A.2.3 Slice-based Cohesion Metrics for Pairs of Variables

The slice-based cohesion metrics on variable level provide a picture of cohesion for each individual variable of a method. Yet, possible interpretations for two variables with the *Coverage* measure of 0.5 are that they: a) cover the same parts of the method, b) cover disjoint parts of the method, or c) have any intersection in method parts. To improve the interpretative power of the metrics, I also introduce **pairwise** slice-based cohesion metrics on variable level.

The computation of the list of union slices for each local variable and input parameter of a method remains the same. But for the computation of pairwise metrics, different sets of statement nodes of the union slices and the method are considered. The idea is to adapt the metrics computation for a variable pair (v, t) of slice variable v and target variable t to account for the references and/or definitions of t in the union slices of v. This way, the cohesion of two variables with regard to the method statements can be interpreted.

I introduce the following statement inclusion types when computing the pairwise metrics: $\Phi = \{REF, DEF, REF + DEF\}$

$\phi = REF$: consider all references to t

$\phi = DEF$: consider all definitions of t

$\phi = REF + DEF$: consider all references and definitions of t

Now let v and t be variables in the set of local variables and input parameters of M.

Let $M_{\phi,t}$ be the subset of statement nodes of M of the given type ϕ, conditional on the target variable t.

Let $UnionSlice_{\phi,t}(v)_i$ be the subset of statement nodes of $UnionSlice(v)_i$ of the given type ϕ, conditional on the target variable t.

The intersection of these subsets of union slices of the given type ϕ, conditional on the target variable t, is:

$$UnionSlice_{\phi,t}(v)_{int} = \bigcap_{j=1}^{|SP_v|} UnionSlice_{\phi,t}(v)_j$$

The adapted pairwise slice-based cohesion metrics, namely *PairCoverage*, *PairOverlap*, and *PairTightness*, are similar to the cohesion metrics for single variables. But notably, the computation is based on the subsets of the method statement nodes and union slice statement nodes introduced above. Below, I introduce the definitions of the pairwise metrics.

Definition A.6 (PairCoverage$_\phi$(M,v,t))**.** PairCoverage for method M, slice variable v and target variable t, computed for statement inclusion type ϕ, is the average of the variable union slice length divided by the method length, only considering statement nodes of type ϕ regarding t.

$$PairCoverage_\phi(M, v, t) = \frac{1}{|SP_v|} \sum_{i=1}^{|SP_v|} \frac{|UnionSlice_{\phi,t}(v)_i|}{|M_{\phi,t}|}$$

Definition A.7 (PairMinCoverage$_\phi$(M,v,t))**.** PairMinCoverage for method M, slice variable v and target variable t, computed for statement inclusion type ϕ, is the ratio of the smallest variable union slice to the method length, only considering statement nodes of type ϕ regarding t.

$$PairMinCoverage_\phi(M, v, t) = \\ \frac{1}{|M_{\phi,t}|} \min_{i=1}^{|SP_v|} |UnionSlice_{\phi,t}(v)_i|$$

Definition A.8 (PairMaxCoverage$_\phi$(M,v,t))**.** PairMaxCoverage for method M, slice variable v and target variable t, computed for statement inclusion type ϕ, is the ratio of the largest variable union slice to the method length, only considering statement nodes of type ϕ regarding t.

$$PairMaxCoverage_\phi(M, v, t) = \\ \frac{1}{|M_{\phi,t}|} \max_{i=1}^{|SP_v|} |UnionSlice_{\phi,t}(v)_i$$

Definition A.9 (PairOverlap$_\phi$(M,v,t))**.** PairOverlap for method M, slice variable v and target variable t, computed for statement inclusion type ϕ, is the average ratio of the length of the intersection of variable union slices to the length of each slice, only considering statement nodes of type ϕ regarding t.

$$PairOverlap_\phi(M, v, t) = \frac{1}{|SP_v|} \sum_{i=1}^{|SP_v|} \frac{|UnionSlice_{\phi,t}(v)_{int}|}{|UnionSlice_{\phi,t}(v)_i|}$$

Definition A.10 (PairTightness$_\phi$(M,v,t))**.** PairTightness for method M, slice variable v and target variable t, computed for statement inclusion type ϕ, is the ratio of the length of the intersection of variable union slices to the method length, only considering statement nodes of type ϕ regarding t.

$$PairTightness_\phi(M, v, t) = \frac{|UnionSlice_{\phi,t}(v)_{int}|}{|M_{\phi,t}|}$$

Values of the pairwise metrics *PairCoverage*, *PairOverlap*, and *PairTightness* also range from 0 to 1, with lower values indicating smaller cohesion in a method with respect to the union slices of the specified slice variable and target variable, for the specified statement inclusion type. For each of the pairwise metrics, it is possible to compute a matrix of pairwise cohesion metrics by iterating through all local variables and input parameters of the method. The resulting *method cohesion matrix* provides an overview of the pairwise cohesiveness of method variables and parameters. In this matrix, slice variables constitute the rows, while target variables constitute the columns.

B. EXEMPLARY VARIABLE CONSTRUCTION PATTERNS

In this chapter exemplary visualizations of all *variable construction patterns* identified in Section 4.4 are presented. The visualizations are given in variable-specific *construction flowlines*. *Variable construction patterns* are assigned based on a multilabel appraoch. For each of the different micropattern labels of control variables and data variables, the following is shown: i) an exemplary resulting method implementation, ii) the corresponding exemplary program construction sequence, distilled from multiple observed program construction sequences, visualization, and iii) the means to identify the respective label in an automated way from *program increments* on *compile-level* granularity.

B.1 Micropattern Labels of Control Variable Construction Patterns

Tab. B.1: All micropattern labels for *variable construction patterns* of control variables, identified in the mixed methods study in Section 4.4.

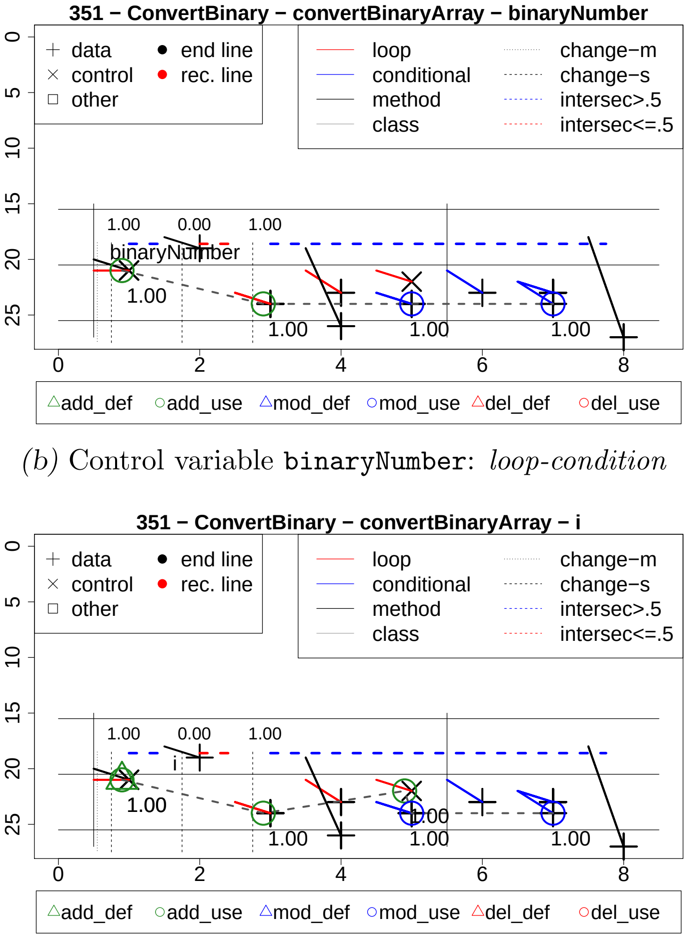
Control Label	***Explanation***
loop-condition	**Used** in loop condition statements
loop	**Defined** and **used** in loop initialization, condition, and update statements
conditional	**Used** in conditional statements
computation	**Used** in control-dependent data statements

Table B.1 provides an overview of all labels for *variable construction patterns* of control variables. Following are exemplary program construction sequences which showcase one label of control variables at a time.

In Figure B.1 (`b`), the variable-specific *construction flowline* shows the construction of the control variable `binaryNumber` for the program in Figure B.1 (`a`). The control variable is labeled a *loop-condition computation* variable. The showcased label is ***loop-condition***, identified with the **added use** of the variable in the change to the first control statement. On close inspection, my approach automatically identifies three different changes in the same line, one for each part of the for-loop - this leads to the different control contexts in the change to the first control

```
public void convertBinaryArray(int[] binaryNumber){
  int result = 0;
  for (int i = 0; i<binaryNumber.length;i++){
    if (i != 0)
      result <<= 1;
    result = result + binaryNumber[i];}
  System.out.println(result);
  System.out.printf("%X", result);}
```

(a) Example Method Implementation (Professional 351)



(b) Control variable `binaryNumber`: *loop-condition*

(c) Control variable `i`: *loop & computation*

Fig. B.1: Example program construction sequence, distilled from a professional programmer, for the *end program* shown in (a). (b) shows the variable-specific *construction flowline* of the control variable `binaryNumber`, which is labeled a *loop-condition computation* variable. (c) shows the variable-specific *construction flowline* of the control variable `i`, which is labeled a *loop computation* variable.

statement. The first part of the for-loop (loop variable initialization) is actually control-dependent on the method, while the other two parts (loop condition, and loop update) are control-dependent on the for-loop. This way, the label ***loop-condition*** signifies a change with loop control context. Of note is that, for while loops, these three parts of a loop are divided in different statement lines - in this case the ***loop-condition*** is control-dependent on the method, but is still a **use** in a changed control statement, that includes a looping construct.

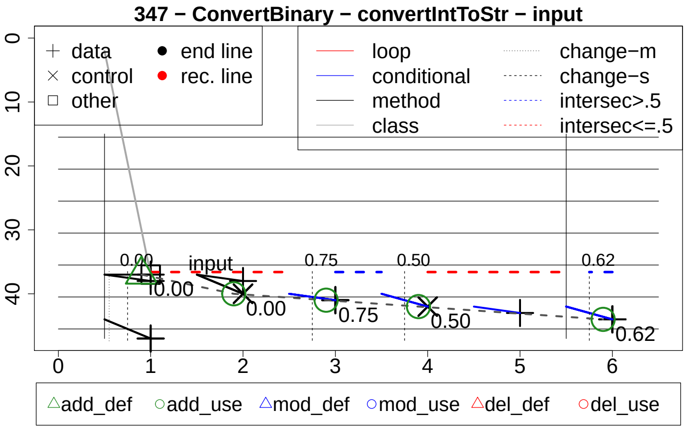
In Figure B.1 (`c`), the variable-specific *construction flowline* shows the construction of the control variable `i` for the program in Figure B.1 (`a`). The control variable is labeled a *condition loop computation* variable. The first showcased label is ***loop***, identified with the **added definition** and the **added use** of the variable in the change to the first control statement. To be labeled with ***loop***, the variable needs to be present in changes to all three loop parts (control statements), being **defined** in the loop initialization, **used** in the loop conditioned, and **defined** in the loop update. This label is therefor easily transferable to while loops.

The second showcased label is ***computation***, signifying that a control variable is **used** in data computations in its respective control context. In this showcase, an example change that leads to the label ***computation*** is the third change, with **added use** of the control variable `i` in the data statement (the computation of the data variable `result`).

In Figure B.2 (`b`), the variable-specific *construction flowline* shows the construction of the control variable `input` for the program in Figure B.2 (`a`). The control variable is labeled a *conditional computation* variable. The showcased label is ***conditional***, identified with **added uses** of the variable in the changes number two and four to the control statements, which is the construction of the if conditional branches. This label is easily transferable to different conditional constructs, for example switch-case. Note that, similar to the **loop computation** variable, this variable is also a *computation* variable as it is **used** in data statements in its respective control context.

```
private String convertIntToStr(int input) {
  String res = "fail";
  if (input > 0 && input < 10) {
    res = "" + input;
  } else if (input > 9 && input < 16) {
    String[] strs = new String[]{"A", "B", "C", "D", "E", "F"};
    res = strs[input − 10];}
  return res;}
```

(a) Example Method Implementation (Professional 347)



(b) Control variable `input`: *conditional*

Fig. B.2: Example program construction sequence, distilled from a professional programmer, for the *end program* shown in (a). (b) shows the variable-specific *construction flowline* of the control variable `input`, which is labeled a *conditional computation* variable.

B.2 Micropattern Labels of Data Variable Construction Patterns

Tab. B.2: All micropattern labels for *variable construction patterns* of data variables, identified in the mixed methods study in Section 4.4.

Data Label	***Explanation***
loop-dependent	**Defined** and/or **used** in loop control context
conditionally-dependent	**Defined** and/or **used** in conditional control context
redefined	Data flow includes no **use** between two **definitions**
self-defining	**Used** in the same statement the variable is **defined**
single-scope-use	All **uses** in single control context and scope

Table B.2 provides an overview of all labels for *variable construction patterns* of data variables. Following are exemplary program construction sequences which showcase one label of control variables at a time.

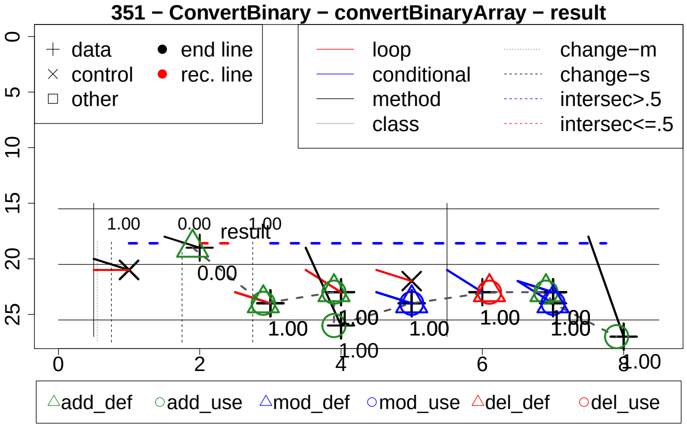
In Figure B.3 (`b`), the variable-specific *construction flowline* shows the construction of the data variable `result` for the program in Figure B.3 (`a`). The data variable is labeled a *conditionally-dependent loop-dependent self-defining* variable. The showcased label is ***conditionally-dependent***, identified with the **modified definition** and **modified use** of the variable in change number 5, which is a change to a data statement in conditional control context. Either change to **definition** or **use** is sufficient for a data variable to be labeled ***conditionally-dependent***. In this case, the variable is also ***loop-dependent***, as another **definition** or **use** in a loop control context is present in the program construction sequence (changes number two and three).

In Figure B.4 (`b`), the variable-specific *construction flowline* shows the construction of the data variable `multiplier` for the program in Figure B.4 (`a`). The data variable is labeled a *conditionally-dependent loop-dependent self-defining* variable. The showcased label is ***loop-dependent***, identified with the **added definition** and **added use** in change number two, a change to a data statement in loop control context. Note that the immediate control context needs to be a loop control context - the change number four, residing in a conditional control context, would not exert this label even though the change is, transitively, in a loop control context. As with the previous label, either change to **definition** or **use** is sufficient for a data variable to be labeled ***loop-dependent***.

In Figure B.4 (`c`), the variable-specific *construction flowline* shows the construction of the data variable `number` for the program in Figure B.4 (`a`). The data variable is labeled a *conditionally-dependent self-defining* variable. The showcased label is ***self-defining***, identified with the **added definition** and **added use** in

```
public void convertBinaryArray(int[] binaryNumber){
  int result = 0;
  for (int i = 0; i<binaryNumber.length;i++){
    if (i != 0)
      result <<= 1;
    result = result + binaryNumber[i];}
  System.out.println(result);
  System.out.printf(”%X”, result);}
```

(a) Example Method Implementation (Professional 351)

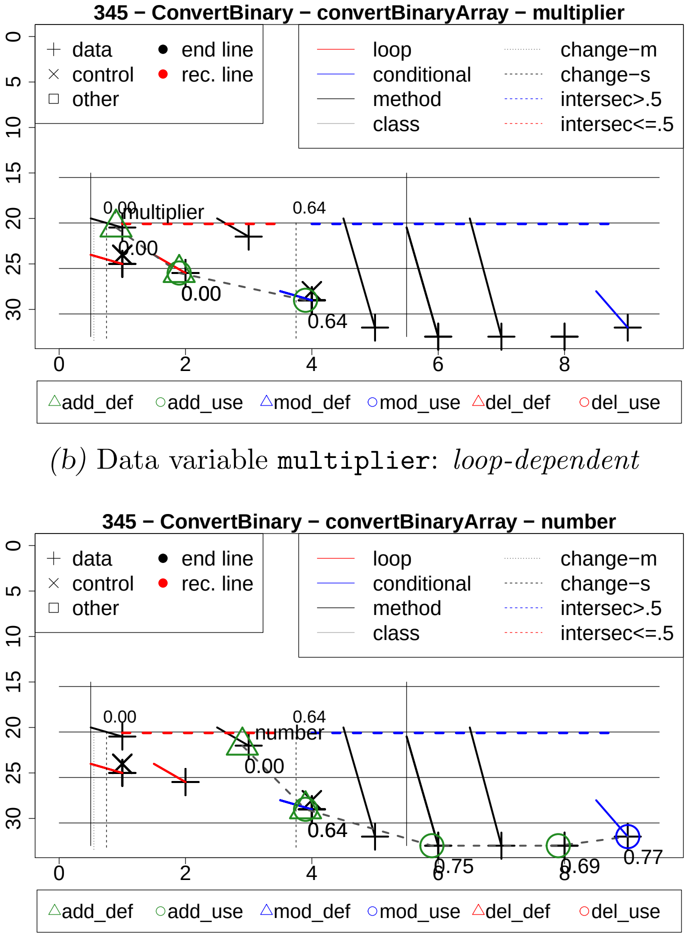


(b) Data variable `result`: *conditionally-dependent*

Fig. B.3: Example program construction sequence, distilled from a professional programmer, for the *end program* shown in (a). (b) shows the variable-specific *construction flowline* of the data variable `result`, which is labeled a *loop-dependent conditionally-dependent self-defining* variable.

```
public void convertBinaryArray(int[] binaryNumber){
  int multiplier = 1;
  int number = 0;
  for(int i = 0; i < 8; i++) {
    boolean isOn = binaryNumber[i] == 1;
    multiplier *= 2;
    if (isOn)
      number += multiplier; }
  System.out.println("decimal: " + number);
  System.out.println("hexadecimal: " + Integer.toHexString(number));}
```

(a) Example Method Implementation (Professional 345)



(b) Data variable `multiplier`: *loop-dependent*

(c) Data variable `number`: *self-defining*

Fig. B.4: Example program construction sequence, distilled from a professional programmer, for the *end program* shown in (a). (b) shows the variable-specific *construction flowline* of the data variable `multiplier`, which is labeled a *loop-dependent conditionally-dependent self-defining* variable. (c) shows the variable-specific *construction flowline* of the data variable `number`, which is labeled a *conditionally-dependent self-defining* variable.

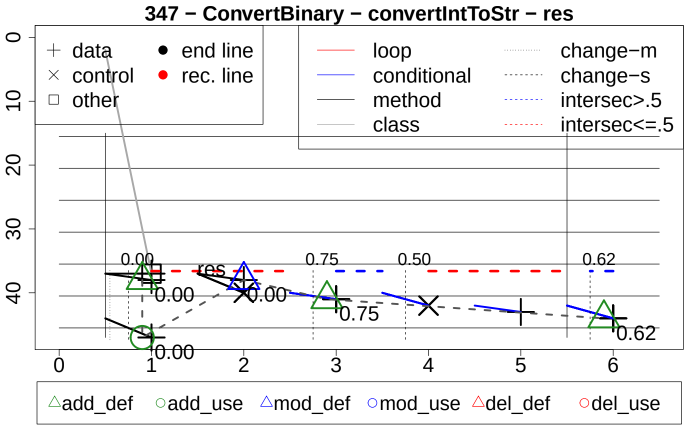
change number four. Regardless of the control context, this label is assigned to data variables that are **used** in the same statement they are also **defined** - effectively depending on their own value. For this label, both **definition** and **use** need to be present.

In Figure B.5 (`b`), the variable-specific *construction flowline* shows the construction of the data variable `res` for the program in Figure B.5 (`a`). The data variable is labeled a *conditionally-dependent redefined* variable. The showcased label is ***redefined***, identified with multiple **definitions** of the variable (changes number one, three, and six) with no **use** in between. Effectively, there is no def-use chain between two **definitions**. This label can be identified with the help of the program dependence graph.

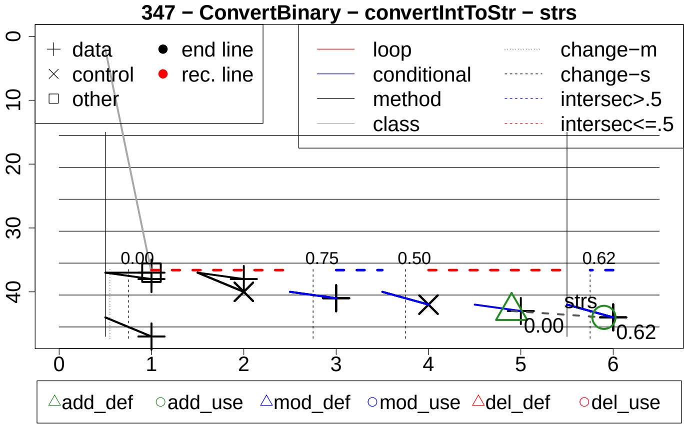
In Figure B.5 (`c`), the variable-specific *construction flowline* shows the construction of the data variable `strs` for the program in Figure B.5 (`a`). The data variable is labeled a *conditionally-dependent single-scope-use* variable. The showcased label is ***single-scope-use***, identified with all variable **uses** in a single control context and scope (change number 6). Take note that only the **uses** need to be in a single control context and scope for this label to apply. The control-context is determined by the respective label. This label is also the default label for data variables, and applies to variables with no other label (for example, with no **definition** or **use** in control contexts).

```
private String convertIntToStr(int input) {
  String res = ”fail”;
  if (input > 0 && input < 10) {
    res = ”” + input;
  } else if (input > 9 && input < 16) {
    String[] strs = new String[]{”A”, ”B”, ”C”, ”D”, ”E”, ”F”};
    res = strs[input − 10];}
  return res;}
```

(a) Example Method Implementation (Professional 347)



(b) Data variable `res`: *redefined*



(c) Data variable `strs`: *single-scope-use*

Fig. B.5: Example program construction sequence, distilled from a professional programmer, for the *end program* shown in (a). (b) shows the variable-specific *construction flowline* of the data variable `res`, which is labeled a *conditionally-dependent redefined* variable. (c) shows the variable-specific *construction flowline* of the data variable `strs`, which is labeled a *conditionally-dependent single-scope-use* variable.

BIBLIOGRAPHY

[AB15] Amjad Altadmri and Neil C.C. Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, page 522–527, New York, NY, USA, 2015. Association for Computing Machinery.

[AB19] Kashif Amanullah and Tim Bell. Analysing students' scratch programs and addressing issues using elementary patterns. *Proceedings - Frontiers in Education Conference, FIE*, 2018-October, 2019.

[AGDS07] Erik Arisholm, Hans Gallis, Tore Dybå, and Dag I.K. Sjøberg. Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, 33(2):65–86, 2007.

[Agg17] Ashish Aggarwal. Neo-piagetian classification of reasoning ability and mental simulation in microsoft's kodu game lab. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 745–746, New York, NY, USA, 2017. Association for Computing Machinery.

[AH16] Efthimia Aivaloglou and Felienne Hermans. How kids code and how we know: An exploratory study on the scratch repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, page 53–61, New York, NY, USA, 2016. Association for Computing Machinery.

[AHMLR17] Efthimia Aivaloglou, Felienne Hermans, Jesus Moreno-Leon, and Gregorio Robles. A dataset of scratch programs: Scraped, shaped and scored. *IEEE International Working Conference on Mining Software Repositories*, pages 511–514, 2017.

[AKA$^+$01] L. W. Anderson, D. R. Krathwohl, P. W. Airasian, K. A. Cruikshank, R. E. Mayer, Pintrich P. R., J. D. Raths, and M. C. Wittrock. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives.* New York: Longman, 1st edition, 2001.

[AMSBA15] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. From Scratch to "Real" Programming. *ACM Transactions on Computing Education*, 14(4):1–15, 2015.

[AW98] Owen Astrachan and Eugene Wallingford. Loop patterns, 1998.

[AW12] Joel C. Adams and Andrew R. Webster. What do students learn about programming from game, music video, and storytelling projects? *Proceedings of the 43rd ACM technical symposium on Computer Science Education - SIGCSE '12*, page 643, 2012.

[BASK18] Neil C.C. Brown, Amjad Altadmri, Sue Sentance, and Michael Kölling. Blackbox, five years on: An evaluation of a large-scale programming data collection project. *ICER 2018 - Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 196–204, 2018.

[BC82] J. Biggs and K. Collis. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome).* NY Academic Press, 1st edition, 1982.

[BDS08] Richard Bornat, Saeed Dehnadi, and Simon. Mental models, consistency and programming aptitude. *Conferences in Research and Practice in Information Technology Series*, 78(1986):53–61, 2008.

[BDS+12] Ryan S. J. d. Baker, Erik Duval, John Stamper, David Wiley, and Simon Buckingham Shum. Educational data mining meets learning analytics. In *Proceedings of the 2nd International Conference on Learning Analytics and Knowledge*, LAK '12, page 20, New York, NY, USA, 2012. Association for Computing Machinery.

[BDW98] J.-M. Burkhardt, F. Detienne, and S. Wiedenbeck. The effect of object-oriented programming expertise in several dimensions of comprehension strategies. *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pages 82–89, 1998.

[BDW02] Jean Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, 7(2):115–156, 2002.

[Ber99] Joseph Bergin. Patterns for selection, 1999.

[BHL+13] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. Hairball: Lint-inspired Static Analysis of Scratch Projects. *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13*, page 215, 2013.

[BKA+18] Robert Bodily, Judy Kay, Vincent Aleven, Ioana Jivet, Dan Davis, Franceska Xhakaj, and Katrien Verbert. Open learner models and learning analytics dashboards: A systematic review. In *Proceedings of the 8th International Conference on Learning Analytics and Knowledge*, LAK '18, page 41–50, New York, NY, USA, 2018. Association for Computing Machinery.

[Bli11] Paulo Blikstein. Using learning analytics to assess students' behavior in open-ended programming tasks. *Proceedings of the 1st International Conference on Learning Analytics and Knowledge - LAK '11*, page 110, 2011.

[BMB+13] Matthew Berland, Taylor Martin, Tom Benton, Carmen Petrick Smith, and Don Davis. Using Learning Analytics to Understand the Learning Pathways of Novice Programmers. *Journal of the Learning Sciences*, 22(4):564–599, 2013.

[Bol13] Andreas Bollin. Metrics for quantifying evolutionary changes in Z specifications. *Journal of Software: Evolution and Process*, 25(9):1027–1059, September 2013.

[BP14] Tiffany Barnes and Thomas W. Price. Comparing Textual and Block Interfaces in a Novice Programming Environment. *ICER 2015 - Proceedings of the 2015 ACM Conference on International Computing Education Research*, 912-914:251–254, 2014.

[Bra92] Charles J. Brainerd. *The Stage Question in Cognitive-developmental Theory*, pages 65–91. Routledge, 1992.

[Bro20] Michael Brown. Seeing students at scale: how faculty in large lecture courses act upon learning analytics dashboard data. *Teaching in Higher Education*, 25(4):384–400, 2020.

[BRS+97] Ilene Burnstein, Katherine Roberson, Floyd Saner, Abdul Mirza, and Abdallah Tubaishat. A Role for Chunking and Fuzzy Reasoning in a Program Comprehension and Debugging Tool. *Artificial Intelligence*, pages 102–109, 1997.

[BSD14] Gunnar R Bergersen, Dag I.K. Sjøberg, and Tore Dyba. Construction and validation of an instrument for measuring programming skill. *IEEE Transactions on Software Engineering*, 40(12):1163–1184, 2014.

[BT06] Roman Bednarik and Markku Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. *Proceedings of the 2006 symposium on Eye tracking research & applications - ETRA '06*, page 125, 2006.

[BWP$^+$14] Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming. *Journal of the Learning Sciences*, 23(4):561–599, 2014.

[BY09] Ryan S J D Baker and Kalina Yacef. The State of Educational Data Mining in 2009: A Review and Future Visions. *Journal of Educational Data Mining*, 1(1):3–16, 2009.

[Caf87] Ralph Cafolla. Piagetian formal operations and other cognitive correlates of achievement in computer programming. *Journal of Educational Technology Systems*, 16(1):45–55, 1987.

[CBEG17] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. Using Tracing and Sketching to Solve Programming Problems. *Proceedings of the 2017 ACM Conference on International Computing Education Research - ICER '17*, pages 164–172, 2017.

[CDP03] Stephen Cooper, Wanda Dann, and Randy Pausch. Teaching objects-first in introductory computer science. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, page 191–195, New York, NY, USA, 2003. Association for Computing Machinery.

[CHA15] Adam S Carter, Christopher D Hundhausen, and Olusola Adesope. The Normalized Programming State Model : Predicting Student Performance in Computing Courses Based on Programming Behavior. *Icer '15*, pages 141–149, 2015.

[CHA17] Adam S Carter, Christopher D. Hundhausen, and Olusola Adesope. Blending Measures of Programming and Social Behavior into

Predictive Models of Student Achievement in Early Computing Courses. *ACM Transactions on Computing Education*, 17(3):1–20, 2017.

[CMM11] Louis Cohen, Lawrence Manion, and Keith Morrison. *Research Methods in Education.* Routledge, 7th edition, 2011.

[CS90] Martha E. Crosby and Jan Stelovsky. How Do We Read Algorithms?: A Case Study. *Computer*, 23(1):25–35, 1990.

[CTAL12] Malcolm Corney, Donna Teague, Alireza Ahadi, and Raymond Lister. Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions. *14th Australasian Computing Education Conference*, 123(2):77–86, 2012.

[CW99] Cynthia L. Corritore and Susan Wiedenbeck. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human Computer Studies*, 50(1):61–83, 1999.

[CW01] Cynthia L. Corritore and Susan Wiedenbeck. Exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human Computer Studies*, 54(1):1–23, 2001.

[DB98] S. H. Downs and N. Black. The feasibility of creating a checklist for the assessment of the methodological quality both of randomised and non-randomised studies of health care interventions. *J Epidemiol Community Health*, 52(6):377–384, June 1998.

[Dét90] Françoise Détienne. Expert Programming Knowledge: A Schema-based Approach. *Psychology of Programming*, pages 205–222, 1990.

[DMS19] Fahima Djelil, Pierre-Alain Muller, and Eric Sanchez. Investigating learners' behaviours when interacting with a programming microworld. In Don Passey, Rosa Bottino, Cathy Lewin, and Eric Sanchez, editors, *Empowering Learners for Life in the Digital Age*, pages 67–76, Cham, 2019. Springer International Publishing.

[dRWT06] Michael de Raadt, Richard Watson, and Mark Toleman. Chick sexing and novice programmers: Explicit instruction of problem solving strategies. *Conferences in Research and Practice in Information Technology Series*, 52:55–62, 2006.

[dRWT09] Michael de Raadt, Richard Watson, and Mark Toleman. Teaching and assessing programming strategies explicitly. *Conferences in Research and Practice in Information Technology Series*, 95(1):45–54, 2009.

[EFR18] Barbara J. Ericson, James D. Foley, and Jochen Rick. Evaluating the efficiency and effectiveness of adaptive parsons problems. *ICER 2018 - Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 60–68, 2018.

[EMR17] Barbara J. Ericson, Lauren E. Margulieux, and Jochen Rick. Solving parsons problems versus fixing and writing code. *Proceedings of the 17th Koli Calling Conference on Computing Education Research - Koli Calling '17*, pages 20–29, 2017.

[FB06] Kurt W. Fischer and Thomas R. Bidell. *Dynamic Development of Action, Thought and Emotion*, pages 313–399. Wiley, 2006.

[FFGP$^+$19] Daniel Amo Filvà, Marc Alier Forment, Francisco José García-Peñalvo, David Fonseca Escudero, and María José Casañ. Clickstream for learning analytics to assess students' behavior with Scratch. *Future Generation Computer Systems*, 93:673–686, 2019.

[FG06] Beat Fluri and Harald Gall. Classifying change types for qualifying change couplings. In *Proceedings of the 9th International Conference on Program Comprehension*, pages 35–45. IEEE Computer Society, January 2006.

[FHSS18] Martina Forster, Urs Hauser, Giovanni Serafini, and Jacqueline Staub. Autonomous recovery from programming errors made by primary school children. In Sergei N. Pozdniakov and Valentina Dagienė, editors, *Informatics in Schools. Fundamentals of Computer Science and Software Engineering*, pages 17–29, Cham, 2018. Springer International Publishing.

[Fis86] Gwen B. Fischer. Computer programming: A formal operational task. In *16th Annual Symposium of the Piaget Society*, page 16, 1986.

[FKH$^+$12] Janet Feigenspan, Christian Kastner, Stefan Hanenberg, Jörg Liebig, and Sven Apel. Measuring Programming Experience. *20th*

International Conference on Program Comprehension, 2005:73–82, 2012.

[FMB⁺14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[FMPPAGPR13] Carlos Fernandez-Medina, Juan Ramón Pérez-Pérez, Víctor M. Álvarez García, and M. del Puerto Paule-Ruiz. Assistance in computer programming learning using educational data mining and learning analytics. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, page 237–242, New York, NY, USA, 2013. Association for Computing Machinery.

[FOK⁺20] Christoph Frädrich, Florian Obermüller, Nina Körber, Ute Heuer, and Gordon Fraser. Common bugs in scratch programs. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, page 89–95, New York, NY, USA, 2020. Association for Computing Machinery.

[Fra15] N. Fraser. Ten things we've learned from blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 49–50, 2015.

[FRP⁺21] Sarah Fakhoury, Devjeet Roy, Harry Pines, Tyler Cleveland, Cole S. Peterson, Venera Arnaoudova, Bonita Sharif, and Jonathan Maletic. gazel: Supporting source code edits in eye-tracking studies. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 69–72, 2021.

[FSO⁺17] Xinyu Fu, Atsushi Shimada, Hiroaki Ogata, Yuta Taniguchi, and Daiki Suehiro. Real-time learning analytics for c programming language courses. In *Proceedings of the Seventh International Learning Analytics and Knowledge Conference*, LAK '17, page 280–288, New York, NY, USA, 2017. Association for Computing Machinery.

[FWPG07] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, November 2007.

[GB17] Shuchi Grover and Satabdi Basu. Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 267–272, 2017.

[GBB$^+$17] Shuchi Grover, Satabdi Basu, Marie Bienkowski, Michael Eagle, Nicholas Diana, and John Stamper. A framework for using hypothesis-driven approaches to support data-driven learning analytics in measuring computational thinking in block-based programming environments. *ACM Trans. Comput. Educ.*, 17(3), August 2017.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[GP13] Shuchi Grover and Roy Pea. Computational thinking in k–12: A review of the state of the field. *Educational Researcher*, 42(1):38–43, 2013.

[GPF09] Harald C. Gall, Martin Pinzger, and Beat Fluri. Change analysis with evolizer and changedistiller. *IEEE Software*, 26(1):26–33, January 2009.

[GSH$^+$18] Francisco J. Gutierrez, Jocelyn Simmonds, Nancy Hitschfeld, Cecilia Casanova, Cecilia Sotomayor, and Vanessa Peña Araya. Assessing software development skills among k-6 learners in a project-based workshop with scratch. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '18, page 98–107, New York, NY, USA, 2018. Association for Computing Machinery.

[HA16] Felienne Hermans and Efthimia Aivaloglou. Do code smells hamper novice programming? A controlled experiment on Scratch programs. *IEEE International Conference on Program Comprehension*, 2016-July(7):1–10, 2016.

[HAB$^+$11] Peter Hubwieser, Michal Armoni, Torsten Brinda, Valentina Dagiene, Ira Diethelm, Michail N. Giannakos, Maria Knobelsdorf, Johannes Magenheim, Roland Mittermeir, and Sigrid Schubert. Computer science/informatics in secondary education. In *Proceedings of the 16th Annual Conference Reports on Innovation and Technology in Computer Science Education - Working Group Reports*, ITiCSE-WGR '11, page 19–38, New York, NY, USA, 2011. Association for Computing Machinery.

[HAES10] Jo E. Hannay, Erik Arisholm, Harald Engvik, and Dag I.K. Sjoberg. Effects of personality on pair programming. *IEEE Transactions on Software Engineering*, 36(1):61–80, 2010.

[Hal82] Graeme S. Halford. *The development of thought.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1982.

[HB09] Brian Hanks and Matt Brandt. Successful and unsuccessful problem solving approaches of novice programmers. *Proceedings of the 40th ACM technical symposium on Computer science education - SIGCSE '09*, page 24, 2009.

[HCNK13] Yoyoi Hofuku, Shinya Cho, Tomohiro Nishida, and Susumu Kanemune. Why is programming difficult?: Proposal for learning programming in "small steps" and a prototype tool for detecting "gaps". *Diethelm, I., Arndt, J., Mittermeir, R. T.*, pages 13–24, 2013.

[HNR01] Werner Hartmann, Jürg Nievergelt, and Raimond Reichert. Kara, finite state machines, and the case for programming as part of general education. In *Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments (Cat. No.01TH8587)*, pages 135–141, 2001.

[HOC17] Christopher D. Hundhausen, Daniel M. Olivares, and Adam S. Carter. IDE-Based Learning Analytics for Computing Education: A Process Model, Critical Review, and Research Agenda. *Critical Review, and Research Agenda. ACM Trans. Comput. Educ*, 17(26):1–26, 2017.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM SIGPLAN Notices*, 39(4):229, 1990.

[HSH16] Felienne Hermans, Kathryn T. Stolee, and David Hoepelman. Smells in block-based programming languages. *Proceedings of*

IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC, 2016-November:68–72, 2016.

[HWP98] Graeme S. Halford, William H. Wilson, and Steven Phillips. Processing capacity defined by relational complexity: Implications for comparative, developmental, and cognitive psychology. *Behavioral and Brain Sciences*, 21(6):803–831, 1998.

[IBE$^+$15] Petri Ihantola, Matthew Butler, Stephen H Edwards, Ari Tech, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Jaime Spacco, Claudia Szabo, DVirginia Toll, and Aniel Korhonen. Educational Data Mining and Learning Analytics in Programming : Literature Review and Case Studies. *ITiCSE WGR'16*, pages 41–63, 2015.

[Inh92] Bärbel Inhelder. *Some Aspects of Piaget's Genetic Approach to Cognition*, pages 92–114. Routledge, 1992.

[ISV14] Petri Ihantola, Juha Sorva, and Arto Vihavainen. Automatically detectable indicators of programming assignment difficulty. In *Proceedings of the 15th Annual Conference on Information technology education - SIGITE '14*, pages 33–38, 2014.

[Jad06] Matthew C. Jadud. Methods and tools for exploring novice compilation behaviour. In *ICER 2006 - Proceedings of the 2nd International Computing Education Research Workshop*, volume 2006, pages 73–84, 2006.

[Kal11] Slava Kalyuga. Cognitive Load Theory: How Many Types of Load Does It Really Need? *Educational Psychology Review*, 23(1):1–19, 2011.

[KB19a] Max Kesselbacher and Andreas Bollin. Discriminating programming strategies in scratch: Making the difference between novice and experienced programmers. In *Proceedings of the 14th Workshop in Primary and Secondary Computing Education*, WiPSCE'19, New York, NY, USA, 2019. Association for Computing Machinery.

[KB19b] Max Kesselbacher and Andreas Bollin. Quantifying patterns and programming strategies in block-based programming environments. *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion, ICSE-Companion 2019*, pages 254–255, 2019.

[KB21] Max Kesselbacher and Andreas Bollin. Towards the use of slice-based cohesion metrics with learning analytics to assess programming skills. In *2021 Third International Workshop on Software Engineering Education for the Next Generation (SEENG)*, pages 6–10, 2021.

[Kes19] Max Kesselbacher. Supporting the Acquisition of Programming Skills with Program Construction Patterns. *ICSE 19*, pages 0–1, 2019.

[KGSF21] Nina Körber, Katharina Geldreich, Andreas Stahlbauer, and Gordon Fraser. Finding anomalies in scratch assignments. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 171–182, 2021.

[KLM14] Theodora Koulouri, Stanislao Lauria, and Robert D. Macredie. Teaching Introductory Programming. *ACM Transactions on Computing Education*, 14(4):1–28, 2014.

[Kri07] Jens Krinke. Statement-level cohesion metrics and their visualization. *SCAM 2007 - Proceedings 7th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 37–46, 2007.

[KS92] Annette Karmiloff-Smith. *Beyond Modularity: A Developmental Perspective on Cognitive Science.* MIT Press, Cambridge, MA, 1992.

[KS10] Päivi Kinnunen and Beth Simon. Building theory about computing education phenomena. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*, pages 37–42, 2010.

[Kum15] Amruth N. Kumar. Solving Code-tracing Problems and its Effect on Code-writing Skills Pertaining to Program Semantics. *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '15*, pages 314–319, 2015.

[KWB20] Max Kesselbacher, Kevin Wiltschnig, and Andreas Bollin. Block-based learning analytics repository and dashboard: Towards an interface between researcher and educator. In *Proceedings of the*

15th Workshop on Primary and Secondary Computing Education, WiPSCE '20, New York, NY, USA, 2020. Association for Computing Machinery.

[LAF$^+$04] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '04, page 119–150, New York, NY, USA, 2004. Association for Computing Machinery.

[LAMJ05] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin*, 37(3):14, 2005.

[LFT09] Raymond Lister, Colin Fidge, and Donna Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education - ITiCSE '09*, page 161, 2009.

[LH19] Yihan Lu and I-Han Hsiao. Exploring programming semantic analytics with deep learning models. In *Proceedings of the 9th International Conference on Learning Analytics and Knowledge*, LAK19, page 155–159, New York, NY, USA, 2019. Association for Computing Machinery.

[Lis11a] Raymond Lister. COMPUTING EDUCATION RESEARCH: Programming, syntax and cognitive load. *ACM Inroads*, 2(2):21–22, 2011.

[Lis11b] Raymond Lister. Concrete and other neo-piagetian forms of reasoning in the novice programmer. *Conferences in Research and Practice in Information Technology Series*, 114(1):9–18, 2011.

[Lis16] Raymond Lister. Toward a Developmental Epistemology of Computer Programming. *Proceedings of the 11th Workshop in Primary and Secondary Computing Education - WiPSCE '16*, pages 5–16, 2016.

[LL03] Raymond Lister and John Leaney. Introductory programming, criterion-referencing, and bloom. *ACM SIGCSE Bulletin*, 35(1):143, 2003.

[LL15] Tien-Duy B. Le and David Lo. Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 331–340, 2015.

[LMH$^+$16] SeolHwa Lee, Andrew Matteson, Danial Hooshyar, SongHyun Kim, JaeBum Jung, GiChun Nam, and HeuiSeok Lim. Comparing Programming Language Comprehension between Novice and Expert Programmers Using EEG Analysis. *2016 IEEE 16th International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 350–355, 2016.

[Lon85] Herbert D. Longworth. Slice based program metrics. Master's thesis, Michigan Technological University, 1985.

[LR16] Andrew Luxton-Reilly. Learning to program is easy. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 11-13-July(7):284–289, 2016.

[LRBC$^+$18] Andrew Luxton-Reilly, Brett A. Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, Simon, and Jacqueline Whalley. Developing assessments to determine mastery of programming fundamentals. *ITiCSE-WGR 2017 - Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, 2018-January:47–69, 2018.

[LRP17] Andrew Luxton-Reilly and Andrew Petersen. The compound nature of novice programming assessments. *ACM International Conference Proceeding Series*, pages 26–35, 2017.

[LRSA$^+$18] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. Introductory programming: A systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2018 Companion, page 55–106, New York, NY, USA, 2018. Association for Computing Machinery.

[LST+06] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. Not seeing the forest for the trees. *ACM SIGCSE Bulletin*, 38(3):118, 2006.

[LWRL08] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory programming. *Proceeding of the fourth international workshop on Computing education research - ICER '08*, pages 101–112, 2008.

[MAD+01] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, 33(4):125–180, 2001.

[MBE+13] Robert McCartney, Jonas Boustedt, Anna Eckerdal, Kate Sanders, and Carol Zander. Can first-year students program yet? *Proceedings of the ninth annual international ACM conference on International computing education research - ICER '13*, page 91, 2013.

[MCS09] Ulrich Müller, Jeremy I. M. Carpendale, and Leslie Smith. *Introduction I: The Context of Piaget's Theory*, page 1–44. Cambridge Companions to Philosophy. Cambridge University Press, 2009.

[MDG14] Briana B. Morrison, Brian Dorn, and Mark Guzdial. Measuring cognitive load in introductory cs: Adaptation of an instrument. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, page 131–138, New York, NY, USA, 2014. Association for Computing Machinery.

[MFRW07] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. Investigating the viability of mental models held by novice programmers. *ACM SIGCSE Bulletin*, 39(1):499, 2007.

[MFRW11] L. Ma, J. Ferguson, M. Roper, and M. Wood. Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*, 21(1):57–80, 2011.

[MGMS08] Sergio Morra, Camilla Gobbo, Zopito Marini, and Ronald Shesse. *Cognitive Development. Neo-Piagetian Perspectives.* Taylor & Francis Group, New York, NY, USA, 2008.

[ML21] Pauline Muljana and Tian Luo. Utilizing learning analytics in course design: Voices from instructional designers in higher education. *Journal of Computing in Higher Education*, April 2021.

[MLHLPD20] Konstantinos Michos, Charles Lang, Davinia Hernández-Leo, and Detra Price-Dennis. Involving teachers in learning analytics design: Lessons learned from two case studies. In *Proceedings of the Tenth International Conference on Learning Analytics and Knowledge*, LAK '20, page 94–99, New York, NY, USA, 2020. Association for Computing Machinery.

[MLRGHR17] Jesús Moreno-León, Marcos Román-González, Casper Harteveld, and Gregorio Robles. On the automatic assessment of computational thinking skills: A comparison with human experts. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '17, page 2788–2795, New York, NY, USA, 2017. Association for Computing Machinery.

[MLRRG15] Jesús Moreno-León, Gregorio Robles, and Marcos Román-González. Dr. Scratch : Automatic Analysis of Scratch Projects to Assess and Foster Computational Thinking. *RED-Revista de Educación a Distancia*, 46(9), 2015.

[Mou93] Pierre Mounoud. Chapter 2 the emergence of new skills: Dialectic relations between knowledge systems. In Geert J.P. Savelsbergh, editor, *The Development of Coordination in Infancy*, volume 97 of *Advances in Psychology*, pages 13 – 46. North-Holland, 1993.

[MPK⁺08] John Maloney, Kylie Peppler, Yasmin B. Kafai, Mitchel Resnick, and Natalie Rusk. Programming by choice: urban youth learning programming with scratch. *SIGCSE '08 Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 367–371, 2008.

[MR14] J. Moreno and G. Robles. Automatic detection of bad programming habits in scratch: A preliminary study. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–4, 2014.

[MRR⁺10] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Trans. Comput. Educ.*, 10(4), November 2010.

[MSABA11] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Habits of programming in scratch. *Proceedings of the 16th annual* *joint conference on Innovation and technology in computer science education - ITiCSE '11*, page 168, 2011.

[MSABA13] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. Learning computer science concepts with Scratch. *Computer Science Education*, 23(3):239–264, 2013.

[MSG^{+}18] Katerina Mangaroska, Kshitij Sharma, Michail Giannakos, Hallvard Trætteberg, and Pierre Dillenbourg. Gaze insights into debugging behavior using learner-centred analysis. In *Proceedings of the 8th International Conference on Learning Analytics and Knowledge*, LAK '18, page 350–359, New York, NY, USA, 2018. Association for Computing Machinery.

[NBF^{+}08] Cindy Norris, Frank Barry, James B. Fenwick, Kathryn Reid, and Josh Rountree. ClockIt: collecting quantitative data on how beginning software developers really work. *ACM SIGCSE Bulletin*, 40(3):37–41, 2008.

[Neu11] Markus Neuhäuser. *Wilcoxon–Mann–Whitney Test*, pages 1656–1658. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[NND^{+}19] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 819–830. IEEE Press, 2019.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5):177–184, 1984.

[OT93] Linda M. Ott and Jeffrey J. Thuss. Slice based metrics for estimating cohesion. *Proceedings - 1st International Software Metrics Symposium, METRIC 1993*, pages 71–81, 1993.

[Pap80] Seymour Papert. *Mindstorms.* Basic Books, 1980.

[Pas20] Stefan Pasterk. *Competence-based Computer Science Education in Austrian's Primary and Lower Secondary Education.* PhD thesis, Universität Klagenfurt, April 2020.

[Pro00] Viera K. Proulx. Programming patterns and design patterns in the introductory computer science course. *ACM SIGCSE Bulletin*, 32(1):80–84, 2000.

[PRS03] Fred Paas, Alexander Renkl, and John Sweller. Cognitive load theory and instructional design: Recent developments. *Educational Psychologist*, 38(1):1–4, 2003.

[PSG18] Sofia Papavlasopoulou, Kshitij Sharma, and Michail N. Giannakos. How do you feel about learning to code? Investigating the effect of children's attitudes towards coding using eye-tracking. *International Journal of Child-Computer Interaction*, 17:50–60, 2018.

[PSGJ17] Sofia Papavlasopoulou, Kshitij Sharma, Michail Giannakos, and Letizia Jaccheri. Using eye-tracking to unveil differences between kids and teens in coding activities. In *Proceedings of the 2017 Conference on Interaction Design and Children*, IDC '17, page 171–181, New York, NY, USA, 2017. Association for Computing Machinery.

[PST18] Evan W. Patton, Mark Sherman, and Michael Tissenbaum. Research tools for mit app inventor. In *Proceedings of the BLOCKS+ Workshop, co-located with SPLASH 2018*, 2018.

[QL17] Yizhou Qian and James Lehman. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.*, 18(1):1:1—-1:24, 2017.

[Ram13] Agim Rama. Slicing von objektorientierten java programmen. Master's thesis, University of Klagenfurt, 2013.

[RDT$^+$07] Filippo Ricca, Massimiliano Di Penta, Marco Torchiano, Paolo Tonella, and Mariano Ceccato. The Role of Experience and Ability in Comprehension Tasks Supported by UML Stereotypes. *Software Engineering 2007 ICSE 2007 29th International Conference on*, pages 375–384, 2007.

[Reg08] Stuart Reges. The Mystery of 'b := (b = false)'. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, page 21–25, New York, NY, USA, 2008. Association for Computing Machinery.

[RF09] L. Todd Rose and Kurt W. Fischer. *Dynamic Development: A Neo-Piagetian Approach*, page 400–422. Cambridge Companions to Philosophy. Cambridge University Press, 2009.

[RH09] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, April 2009.

[RHK16] Kelly Rivers, Erik Harpstead, and Ken Koedinger. Learning Curve Analysis for Programming. In *Proceedings of the 2016 ACM Conference on International Computing Education Research - ICER '16*, pages 143–151, 2016.

[Rob03] N. Robins, A., Rountree, J. and Rountree. Learning and Teaching Programming : A Review and Discussion. *Computer Science Education*, 13(2):137–172, 2003.

[RSB$^+$18] Kathryn M. Rich, Carla Strickland, T. Andrew Binkowski, Cheryl Moran, and Diana Franklin. K–8 learning trajectories derived from research literature. *ACM Inroads*, 9(1):46–55, 2018.

[RSS00] Sam Ramanujan, Richard W. Scamell, and Jaymeen R. Shah. Experimental investigation of the impact of individual, program, and organizational characteristics on software maintenance effort. *Journal of Systems and Software*, 54(2):137–157, 2000.

[Rub13] Marc J. Rubin. The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 651–656, New York, NY, USA, 2013. Association for Computing Machinery.

[RV07] C. Romero and S. Ventura. Educational data mining: A survey from 1995 to 2005. *Expert Systems with Applications*, 33(1):135 – 146, 2007.

[RW02] V. Rajlich and N. Wilde. The role of concepts in program comprehension. *Proceedings - IEEE Workshop on Program Comprehension*, 2002-January:271–278, 2002.

[SA16] David Statter and Michal Armoni. Teaching abstract thinking in introduction to computer science for 7th graders. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*, WiPSCE '16, page 80–83, New York, NY, USA, 2016. Association for Computing Machinery.

[SB12] George Siemens and Ryan S. J. d. Baker. Learning analytics and educational data mining: Towards communication and collaboration. In *Proceedings of the 2nd International Conference on Learning Analytics and Knowledge*, LAK '12, page 252–254, New York, NY, USA, 2012. Association for Computing Machinery.

[SE84] Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on software engineering*, SE-10(5):595–609, 1984.

[SHS18] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. Programming misconceptions for school students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, page 151–159, New York, NY, USA, 2018. Association for Computing Machinery.

[Sie96] Robert S. Siegler. *Emerging minds: The process of change in children's thinking.* Oxford University Press, Oxford, 1996.

[SKL$^+$14] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[SM15] Mark Sherman and Fred Martin. Learning analytics for the assessment of interaction with app inventor. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 13–14, 2015.

[SMD08] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[SMT$^+$18] Kshitij Sharma, Katerina Mangaroska, Hallvard Traeteberga, Serena Lee-Cultura, and Michail Giannakos. *Evidence for Programming Strategies in University Coding Exercises*, volume 11082. Springer International Publishing, 2018.

[Sol86] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[Sor13] Juha Sorva. Notional machines and introductory programming education. *ACM Trans. Comput. Educ.*, 13(2), July 2013.

[SS11] Simon and Susan Snowdon. Explaining Program Code: Giving Students the Answer Helps – But Only Just. *Proceedings of the 7th Annual International ACM Conference on International Computing Education Research (ICER '11)*, pages 93–99, 2011.

[SS15] Janet Siegmund and Jana Schumann. Confounding parameters on program comprehension: a literature survey. *Empirical Software Engineering*, 20(4):1159–1192, 2015.

[SSH17] Alaaeddin Swidan, Alexander Serebrenik, and Felienne Hermans. How do Scratch Programmers Name Variables and Procedures? *Proceedings - 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation, SCAM 2017*, 2017-October:51–60, 2017.

[SvMP19] John Sweller, Jeroen J.G. van Merriënboer, and Fred Paas. Cognitive Architecture and Instructional Design: 20 Years Later. *Educational Psychology Review*, 31(2):261–292, 2019.

[SVP98] John Sweller, Jeroen J.G. Van Merrienboer, and Fred G.W.C. Paas. Cognitive Architecture and Instructional Design. *Educational Psychology Review*, 10(3):251–296, 1998.

[SW13] Cynthia Selby and John Woollard. Computational thinking: the developing definition. Project report, University of Southampton, 2013.

[Swe88] John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2):257–285, 1988.

[Swe10] John Sweller. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review*, 22(2):123–138, 2010.

[Tea15] Donna Teague. Neo-Piagetian Theory and the Novice Programmer. *Thesis*, pages 1–370, 2015.

[TGSH20] Mike Talbot, Katharina Geldreich, Julia Sommer, and Peter Hubwieser. Re-use of programming patterns or problem solving? representation of scratch programs by tgraphs to support static code analysis. In *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*, WiPSCE '20, New York, NY, USA, 2020. Association for Computing Machinery.

[Tho11] Henry C. Thode. *Normality Tests*, pages 999–1000. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[TKS04] Pang-Ning Tan, Vipin Kumar, and Jaideep Srivastava. Selecting the right objective measure for association analysis. *Information Systems*, 29(4):293–313, 2004. Knowledge Discovery and Data Mining (KDD 2002).

[TL14a] Donna Teague and Raymond Lister. Blinded by their plight: Tracing and the preoperational programmer. *Psychology of Programming Interest Group (Ppig)*, 25(6), 2014.

[TL14b] Donna Teague and Raymond Lister. Longitudinal think aloud study of a novice programmer. *Conferences in Research and Practice in Information Technology Series*, 148:41–50, 2014.

[TL14c] Donna Teague and Raymond Lister. Programming: reading, writing and reversing. *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*, pages 285–290, 2014.

[TSA$^+$19] Giovanni Maria Troiano, Sam Snodgrass, Erinç Argımak, Gregorio Robles, Gillian Smith, Michael Cassidy, Eli Tucker-Raymond, Gillian Puttick, and Casper Harteveld. Is my game ok dr. scratch? exploring programming and computational thinking development via metrics in student-designed serious games for stem. In *Proceedings of the 18th ACM International Conference on Interaction Design and Children*, IDC '19, page 208–219, New York, NY, USA, 2019. Association for Computing Machinery.

[vFP96] Thomas von Fintel and Glen Pate, 1996.

[VS07] Vesa Vainio and Jorma Sajaniemi. Factors in novice programmers' poor tracing skills. *ACM SIGCSE Bulletin*, 39(3):236, 2007.

[VTL09] Anne Venables, Grace Tan, and Raymond Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, page 117–128, New York, NY, USA, 2009. Association for Computing Machinery.

[Vyg78] Lev S. Vygotsky. *Interaction between Learning and Development*, pages 79–91. Harvard University Press, 1978.

[Wal03] Eugene Wallingford. The elementary patterns home pagen, 2003.

[Wei81] Mark Weiser. Program slicing. *Proceedings - International Conference on Software Engineering*, pages 439–449, 1981.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[Wei99] Franz E. Weinert. Concepts of competence. *Definition and Selection of Competencies: Theoretical and Conceptual Foundations (DeSeCo)*, 1999.

[Wie86] Susan Wiedenbeck. Organization of programming knowledge of novices and experts. *Journal of the American Society for Information Science*, 37(5):294–299, 1986.

[Win96] Leon E. Winslow. Programming Pedagogy –A Psychological Overview. *ACM SIGCSE Bulletin*, 28(3):17–22, 1996.

[Win06] Jeannette M. Wing. Computational thinking. *Commun. ACM*, 49(3):33–35, March 2006.

[WKF18] David Weintrop, Heather Killen, and Baker Franke. Blocks or text? How programming language modality makes a difference in assessing underrepresented populations. *Proceedings of International Conference of the Learning Sciences, ICLS*, 1(6):328–335, 2018.

[WL14] Christopher Watson and Frederick W.B. Li. Failure rates in introductory programming revisited. *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*, pages 39–44, 2014.

[WLG13] C. Watson, F. W. B. Li, and J. L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*, pages 319–323, 2013.

[WS83] M Weiser and J Shertz. Programming problems representation in novice and expert programmers. *International Journal of Man-Machine Studies*, 19:391–398, 1983.

[WW17] David Weintrop and Uri Wilensky. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Trans. Comput. Educ.*, 18(1), October 2017.

[XNK18] Benjamin Xie, Greg L. Nelson, and Andrew J. Ko. An explicit strategy to scaffold novice program tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, page 344–349, New York, NY, USA, 2018. Association for Computing Machinery.

[YKM07] S. Yusuf, H. Kagdi, and J. I. Maletic. Assessing the comprehension of uml class diagrams via eye tracking. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 113–122, 2007.

[Zak01] Mohammed J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1-2):31–60, 2001.

[ZDH$^+$20] Albina Zavgorodniaia, Rodrigo Duran, Arto Hellas, Otto Seppala, and Juha Sorva. Measuring the cognitive load of learning to program: A replication study. In *United Kingdom & Ireland Computing Education Research Conference.*, UKICER '20, page 3–9, New York, NY, USA, 2020. Association for Computing Machinery.

LIST OF FIGURES

LIST OF TABLES

LIST OF DEFINITIONS