Cognitive Complexity of Comprehending Computer Programs

Rodrigo Duran
Cognitive Complexity of Comprehending Computer Programs

Rodrigo Duran

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2 of the school on 29th of June 2020 at 12:00.

Aalto University
School of Science
Computer Science
Learning + Technology Group (LeTech)
Supervising professor
Professor Lauri Malmi

Thesis advisor
D.Sc. (Tech) Juha Sorva

Preliminary examiners
Assistant Professor Briana Morrison, University of Nebraska, USA
Assistant Professor Lauren Margulieux, Georgia State University, USA

Opponent
Professor Matti Tedre, University of Eastern Finland, Finland

Aalto University publication series
DOCTORAL DISSERTATIONS 100/2020

© 2020 Rodrigo Duran

ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Unigrafia Oy
Helsinki 2020

Finland
### Abstract

**Author**
Rodrigo Duran

**Name of the doctoral dissertation**
Cognitive Complexity of Comprehending Computer Programs

**Publisher**
School of Science

**Unit**
Computer Science

**Series**
Aalto University publication series DOCTORAL DISSERTATIONS 100/2020

**Field of research**
Computing Education

**Manuscript submitted** 1 April 2020  
**Date of the defence** 29 June 2020

**Permission for public defence granted (date)** 1 June 2020  
**Language** English

- [ ] Monograph  
- [x] Article dissertation  
- [ ] Essay dissertation

**Abstract**

Instructional designers must consider learners’ learning trajectories and design tasks that are neither too hard nor too easy for them, sequencing tasks from less to more complex ones. Most efforts in programming assessment have been directed to code writing. However, programming is a multi-faceted skill, including precursor skills such as the comprehension of programs, which recent studies suggest having many interacting elements.

An essential part of assessment is characterization of what makes a program unique and how to estimate learner’s previous knowledge. When programs are different enough, instructors can intuitively compare the effort demanded of the learners. However, when the difference is subtle, instructors struggle to evaluate program’s cognitive demands.

Complexity is the metric used to describe the cognitive demands of a task. While research of computational and software engineering metrics of complexity are well-established, little research has been devoted to evaluating the complexity of comprehending programs from learners’ perspective. In general, the taxonomies used by Computing Education to evaluate complexity do not evaluate the core content used in programming tasks. While subjective evaluations of difficulty proved to be useful in empirical evaluations, researchers have advocated for a complementary \emph{(a priori)} analytical metric of complexity to support instructional design.

To alleviate such gaps, we seek to develop and present the necessary toolset to define and evaluate the cognitive complexity of comprehending computer programs. We introduce a novel conceptualization, the Rules of Program Behavior, which augment previous notional machine research and offer guidelines to communicate semantics instruction among practitioners and set the expectations of possible learners’ mental models. We designed and partially validated a self-evaluation instrument to assess prior programming knowledge, inspired by frameworks successfully used in linguistics. These tools serve as input to our theoretical framework of cognitive complexity, which is based on educational psychology theories. The framework analyzes the cognitive elements present in a given program, and the way these elements are intertwined, extracting measurable aspects of complexity. Finally, we investigated instructors’ perspectives of program comprehension and presented activities to foster program comprehension and how such activities could be sequenced to create learning trajectories.

Our framework and tools laid the foundation to evaluate the cognitive complexity of program comprehension. We expect that the results of this thesis could support the design of assessment instruments, curricula, and programming languages. Our work particularly fits frameworks that holistically integrate skills and knowledge using authentic tasks while keeping learners’ cognitive load in check. We believe that our results can be adapted to other aspects of programming and could help researchers to generate and test hypotheses related to program comprehension.

**Keywords** Complexity, plans, self-evaluation, notional machines, program comprehension


**ISSN (printed)** 1799-4934  
**ISSN (pdf)** 1799-4942

**Location of publisher** Helsinki  
**Location of printing** Helsinki  
**Year** 2020

**Pages** 266  
Abstract

Instructional designers must consider learners’ learning trajectories and design tasks that are neither too hard nor too easy for them, sequencing tasks from less to more complex ones. Most efforts in programming assessment have been directed to code writing. However, programming is a multi-faceted skill, including precursory skills such as program comprehension. At the core of assessing the latter one are concrete programs, which recent studies suggest having many interacting elements.

An essential part of assessment is characterization of what makes a program unique and how to estimate learner’s previous knowledge. When programs are different enough, instructors can intuitively compare the effort demanded of the learners. However, when the difference is subtle, instructors struggle to compare the cognitive demands imposed by programming language constructs and the code structure.

Complexity is the metric used to describe the cognitive demands of a task. While research of computational complexity and software engineering metrics of complexity is well-established, little research has been devoted to evaluating the complexity of comprehending programs from learners’ perspective. In general, the taxonomies used by Computing Education to evaluate complexity do not evaluate the core content used in programming tasks. While subjective evaluations of difficulty proved to be useful in empirical evaluations, researchers have advocated for a complementary a priori analytical metric of complexity to support instructional design.

To alleviate such gaps, we seek to develop and present the necessary toolset to define and evaluate the cognitive complexity of comprehending computer programs. We introduce a novel conceptualization, the Rules of Program Behavior, which augment previous notional machine research and offer guidelines to communicate semantics instruction among practitioners and set the expectations of possible learners’ mental models. We designed and partially validated a self-evaluation instrument to assess prior programming knowledge, inspired by frameworks successfully used in linguistics. These tools serve as input to our theoretical framework of cognitive complexity, which is based on educational psychology theories. The framework analyzes the cognitive elements present in a given program, and the way these elements are intertwined, extracting measurable aspects of complexity. Finally, we investigated instructors’ perspectives of program comprehension and presented activities to foster program comprehension and how such
activities could be sequenced to create learning trajectories.

Our framework and the tools to operationalize it laid the foundation to evaluate the cognitive complexity of comprehending programs. We expect that the results of this thesis could support the design of assessment instruments, curricula, and programming languages. Our work particularly fits for frameworks that holistically integrate skills and knowledge using authentic tasks while keeping learners’ cognitive load under control. We believe that our results can be adapted to other aspects of programming and could help researchers to generate and test hypotheses related to program comprehension.
To my family, friends, and my loved wife Aline.

To my students that inspired me to learn and be better.
I did not expect that this preface would be so challenging to write. Since it is both the end of my doctoral studies and my stay in Finland, I feel a mix of feelings. It is great to finish such a long journey, but I will miss every moment. I am indebted to so many great people that made my journey possible. I can only hope I can do justice to all of them here.

First and foremost, I would like to thank Professor Lauri Malmi. After almost six years, it seems incredible that this journey started with a very unpretentious e-mail that I thought would never be answered. Not only did Lauri answer the e-mail and kindly received for a full afternoon at Aalto, but also offered the possibility of joining his research group in the future. I sincerely thank professor Lauri for allowing me to pursue my interests so freely and always trusting me, even with so many changes in my research plan.

I great friend once (accurately) described me as someone that enjoys being challenged (and sometimes needs to be challenged). It is safe to say that no one challenged me more than Juha Sorva in the last few years. I am deeply indebted to Juha, and I cannot describe how pleasant it was to conduct research with you. I certainly learned a lot, you changed how I see and think about research. Thanks for all the “out-of-nowhere” discussions in your room and the careful tutoring of my work.

I would like to thank my many co-authors. Lassi Haaranen certainly made not only this work possible, but life in Finland itself easier. We share similar interests in games and programming, and while this dissertation deviated from this theme, I am deeply thankful for his collaboration. Lassi kindly supported me through so many issues, research-wise or not, and was always a supporting voice. Arto Hellas was a late addition to our research group. Arto has this kind of “let’s do it” mentality that always encouraged me to pursue my goals, even when the deadlines were particularly tight.
Thank you, Arto, for your light mood and perseverance. Jan-Mikael Rybicki brought so many insightful perspectives from Language education to this work. Also, I am deeply thankful to Jan-Mikael for improving my writing in this thesis and our papers. I also thank Sanna Suoranta for her comments on our articles and in this dissertation. My colleagues from the ITiCSE working groups are too many to be named and acknowledged individually. Still, I am undoubtedly indebted to them and will be forever grateful for the knowledge and experience I have acquired while intensively working with them.

Many others contributed in many different ways to my doctoral studies. Thanks, Otto Seppälä, for all the long talks and suggestions. Your ability to think “outside-the-box” is unparalleled. Thank you, Kerttu Pollari-Malmi, for kindly allowing me to disturb her courses so often to collect data and learn from her students. I thank Ari Korhonen for all the great football games we went, they definitely helped me remember home. All LeTech members helped me a lot. Thanks to Abby Zavgorodniaia, Sercan Erer, Teemu Lehtinen, Tapio Auvinen, Sami Sarsa, and others for the excellent talks. Room A135 was definitely a fantastic place to be at Aalto!

I cannot thank enough my family and friends. They gave me strength and the calm needed to face the most strenuous situations. Thanks, mom, dad, and Tati. I know it is hard to be always away from the family, but your frequent visits here ever made me feel relaxed and loved. I cannot thank enough my “Finnish” family and friends (hello Carolina Carvalho, Vitória Pacela), particularly Seija Mahlamäki-Kultanen. To be honest, none of this would be possible if it was not for her. I will be forever grateful. Finally, my loved wife, Aline, endured many days of work where she barely saw me for weeks. Thank you for all your support and caress. This thesis is an accomplishment for both of us.

Furthermore, I thank my pre-examiners, Assistant Professors Briana Morrison and Lauren Margulieux, for their insightful and detailed feedback of my thesis. I am honored to have Professor Matti Tedre as my opponent. Thanks for the CNPq and IFMS for their financial support.

Espoo, Finland, June 4, 2020,

Rodrigo Duran
Contents

Preface

Contents

List of Publications

Author's Contribution

1. Introduction
   1.1 Gaps in Current Research
   1.2 Objectives and Scope
   1.3 Research Questions
   1.4 Research Process
   1.5 Thesis Structure

2. Theoretical Foundation
   2.1 Notional Machines
   2.2 Mental Representations of Knowledge: Schemata & Plans
      2.2.1 Human Memory Apparatus & Expertise
      2.2.2 Plans
      2.2.3 Plan-Composition Strategies
   2.3 Complexity & Difficulty
      2.3.1 Definitions and Models Complexity and Difficulty
      2.3.2 The Model of Hierarchical Complexity
      2.3.3 Neo-Piagetian Theory and Working Memory
      2.3.4 The Cognitive Load Theory
      2.3.5 Evaluating Complexity
   2.4 Assessing Program Comprehension Knowledge
      2.4.1 Tests
      2.4.2 Questionnaires
2.4.3 Self-Evaluation Instruments .................... 67

2.5 Program Comprehension Activities .................. 72
2.5.1 Models of Program Comprehension ............... 73
2.5.2 Program Comprehension Activities ............... 74
2.5.3 Towards a Comprehension First Pedagogy in CER 76

2.6 Instructional Design for Complex Learning .......... 78
2.6.1 Design Learning Tasks .......................... 80
2.6.2 Design Performance Assessment ................. 81
2.6.3 Sequence Learning Tasks ....................... 82
2.6.4 Design Supportive Information .................. 83
2.6.5 Analyze Cognitive Strategies ................... 84
2.6.6 Analyze Mental Models ......................... 84
2.6.7 Design Procedural Information .................. 84
2.6.8 Analyze Cognitive Rules ......................... 85
2.6.9 Analyze Prerequisite Knowledge ................ 85
2.6.10 Design Part-task practice ...................... 85

3. Results 87
3.1 RQ 1.1: How to Design Representations of Rules of Program Behavior? ..................... 87
3.2 RQ 1.2: How to define and evaluate the cognitive complexity of comprehending concrete programs? .................. 96
3.2.1 Hierarchical Analysis .......................... 96
3.2.2 Interactivity Analysis ........................... 99
3.3 RQ 1.3: How to evaluate students’ prior-knowledge of program comprehension? .................. 103
3.4 RQ 1.4: What kind of activities can foster program comprehension and how such activities can be organized? 115
3.4.1 Activities to Foster Program Comprehension 116
3.4.2 Learning Trajectories for Program Comprehension 121

4. Discussion and Conclusion 127
4.1 How to design explicit representations of the rules of program behavior? ..................... 128
4.2 How to define and evaluate the cognitive complexity of comprehending concrete programs? 130
4.3 How to evaluate students’ prior-knowledge of program comprehension? 131
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>What kind of activities can foster program comprehension and how such activities can be organized?</td>
</tr>
<tr>
<td>4.5</td>
<td>Revisiting Thesis Main Research Question</td>
</tr>
<tr>
<td>4.6</td>
<td>Implications for Research</td>
</tr>
<tr>
<td>4.6.1</td>
<td>CER Theories of Complexity</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Notional Machines Research</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Assessment Research</td>
</tr>
<tr>
<td>4.7</td>
<td>Implications for Practitioners</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Course Design</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Choosing Programming Languages</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Learning Trajectories</td>
</tr>
<tr>
<td>4.7.4</td>
<td>Writing “Good Code”</td>
</tr>
<tr>
<td>4.7.5</td>
<td>Communicating Instruction</td>
</tr>
<tr>
<td>4.8</td>
<td>Limitations, validity and reliability</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Cognitive Complexity Analysis</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Rules of Programming Behavior</td>
</tr>
<tr>
<td>4.8.3</td>
<td>Self-Evaluation Instrument</td>
</tr>
<tr>
<td>4.8.4</td>
<td>Program Comprehension Learning Trajectories</td>
</tr>
<tr>
<td>4.9</td>
<td>Computing Education Research Through the Lens of the Four Components/Instructional Design</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Learning Tasks, Fading Support, and Variability</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Sequencing Learning Tasks</td>
</tr>
<tr>
<td>4.9.3</td>
<td>Supportive Information</td>
</tr>
<tr>
<td>4.9.4</td>
<td>Procedural Information</td>
</tr>
<tr>
<td>4.9.5</td>
<td>A Road Map to Programming Instruction</td>
</tr>
<tr>
<td>4.10</td>
<td>Recommendations for Future Research</td>
</tr>
</tbody>
</table>

References 157

Publications 193
This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.


V Izu, Cruz and Schulte, Carsten and Aggarwal, Ashish and Cutts, Quintin and Duran, Rodrigo and Gutica, Mirela and Heinemann, Birte and Kraemer, Eileen and Lonati, Violetta and Mirolo, Claudio and Weeda,
Author’s Contribution

Publication I: “Design of Rules of Program Behavior for Teaching”

I am the single author. I performed the conceptualization, developed the example case study and wrote the report.

Publication II: “Towards an Analysis of Program Complexity From a Cognitive Perspective”

I conducted the literature review and theoretical formulation of the model, with the second and third authors acting in a consultant role through its development. I designed, with support of the second author, the case studies presented in the paper. The second author contributed equally to the writing of the report.


I led the development of the instrument with the support of the second author. I performed the quantitative analysis reported in the results of the paper, except the factor analysis performed by the fourth author. All authors contributed equally to the writing of the literature review and discussion sections of the reporting.
Publication IV: “Exploring the Value of Student Self-Evaluation in Introductory Programming”

I performed the data analysis in the results section. All authors contributed equally to the writing of the literature review and discussion sections.

Publication V: “Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories”

In an ITiCSE Working Group (WG), researchers apply to join a group interested in a specific topic. When accepted, research is conducted before the conference. At the conference, the results are analyzed, and a preliminary draft is submitted. If accepted, this draft is later extended and revised to be submitted for peer review. A WG has leaders coordinating the research and assigning tasks to group members, that are nevertheless participating in every step of the research and later writing of the report. I contributed with the interview data collection and was one of the three members in charge of creating the learning trajectories for program comprehension. I also contributed to the writing of the theoretical foundations of program comprehension presented in the report.
One of the major goals in curriculum development is to establish a theoretically motivated, empirically sustainable, and pedagogically feasible methodology for sequencing topics [297]. Fields such as mathematics [13], professional development [309], and medicine [380, 354] have benefited from instructional design frameworks intended to teach complex subjects, such as the Four Components/Instructional Design (4C/ID) [375] and guidelines aimed at instruction the Ten Steps to Complex Learning (Ten Steps) [374]. At the core of the Ten Steps are whole authentic tasks that “integrate many skills, knowledge, and attitudes, stimulate learners to learn to coordinate constituent skills and facilitate transfer to new problem situations” [374]. However, while authentic tasks can bring many benefits to instruction, they also possibly combine several interacting elements. Such interaction between elements can make the task too complex for learners, particularly to beginners.

Therefore, for instructional design to be successful, designers must consider learners’ learning trajectories. The conceptualization of the Zone of Proximal Development (ZPD), famously captured by Vygotsky [49], shows that, ideally, learners engage in activities that are neither too hard nor too easy for them. With growing expertise, each learner can eventually tackle increasingly complex tasks.

Customarily, teachers assess the complexity of the tasks they present to learners and attempt to sequence those tasks, from less to more complex tasks. While certain fields such as Physics, Chemistry, or Mathematics have established curricula with explicit guidelines based on empirical evidence and instructors’ expertise in how to sequence tasks, Computing Education Research (CER) is still at its infancy, and such methods are still in development.

At the heart of introductory programming courses are tasks with concrete
programs, which students should comprehend or write by themselves. While most efforts to develop assessments have been directed to teach the ability to write programs, programming education itself is multi-faceted, encompassing essential (and some say precursory) skills such as design, test, and comprehension, as illustrated in Figure 1.1. In this thesis, we focus on the program comprehension sub-skill of programming education. To assess learner’s knowledge in more detail and provide individualized feedback, more fine-grained assessment methods [364, 204, 207, 322, 254] have received considerable attention in recent years by CER.

Figure 1.1. The many facets of programming.

At the essence of such fine-grained assessment methods is the characterization of what makes each program unique and how it estimates learner’s knowledge. Using their expertise, instructors “know” that some programs are different, and some are more complex than others. When the goals, structure, and size of programs are different enough, instructors can intuitively evaluate and compare the effort demanded by learners to comprehend such programs. For example, the archetypal most straightforward program (e.g., a “hello world”) is undoubtedly “easier” to understand than a program that implements a sorting algorithm (e.g., Quicksort).

However, when the difference between programs is not so obvious, or programs are written in different paradigms/behaviors/programming languages, how could instructors compare the demands required from learners to comprehend them? Not uncommonly, researchers claim to have presented tasks demanding similar effort from learners [195, 325, 240]. However, frequently the methodology used to sequence such tasks is not clear or even absent. This comparison of programs should be made at a granular-level, improving the design of learning trajectories and assessment instruments. Simon et al. [325] ask:

“Is an assignment statement easier or harder to read and understand than a print statement? Is a nested loop easier or harder than an if-then-else? Is the difficulty of a piece of code simply the linear sum of the difficulties of its
constituent parts? [...] If we are to have a reliable measure of students’ abilities at reading and writing code, we would need to consider a minute analysis of the difficulty levels of code-reading and code-writing questions at the micro-level.”

Earlier works in CER by Soloway, Spohrer, Rist, and others [336, 296, 343] presented plans and schemata as the primary units of analysis when evaluating the cognitive demands of programming tasks. Soloway [335, 344] describes plans as “chunks” of knowledge incorporated in a canned-solution, and how learners compose a solution to a given problem, a plan-composition strategy, is one of the primary skills in programming and source of bugs and other difficulties faced by learners. Recent studies have shown that certain behaviors of the programming language and pedagogical approaches related to the program’s plan-composition could affect code comprehension and performance [99, 102].

If plans are the primary unit of analysis, the complexity is the metric (often loosely defined) used to evaluate the necessary cognitive demands required to successfully complete a task [60, 198]. While research regarding machine execution complexity is vast and well-established [168], little research has been devoted to conceptualize and evaluate the cognitive complexity of comprehending programs from the learners’ perspective.

1.1 Gaps in Current Research

In general, CER resorted to adaptations of taxonomies such as Bloom’s [218] (emphasizing the general types of activities that programmers engage), and SOLO [29] (outlining the general degree of structuredness in learning outcomes) to categorize tasks or skills [45, 219, 155, 364, 82, 105, 47]. However, these taxonomies are not particularly concerned with the complexity of the tasks, nor evaluate the content used in programming tasks, the concrete programs employed in such activities. Furthermore, these taxonomies are not granular enough to differentiate similar activities manipulating dissimilar programs.

Software Engineering (SE) has attempted for decades to produce metrics of complexity from concrete programs that can express the “readability”, “comprehensibility” or “understandability” of code by analyzing control and data flow, lines of code, and other features of programs. Metrics such as Cyclomatic Complexity [222], Halstead [138], lines of code (LoC), and block depth have been applied to in a variety of programming tasks [163, 87, 308].
While such metrics provide one perspective on software complexity, they neglect the cognitive effort demanded, particularly from beginners, to comprehend the code. These metrics cannot show how different constructs and goals embedded in parts of the code affect complexity [2]; have a low correlation with perceived complexity [164]; have a poor or no theorization of comprehension to support them [399], and disregard the role of prior-knowledge when comprehending programs.

* A posteriori subjective perceptions of mental effort have been widely used in engineering studies to investigate elements of difficulty in a task and provide reliable scales of cognitive load using the NASA TLX instrument [142], recently adapted to Computer Science (CS) activities by Morrison et al. [241]. While such a metric could prove to be useful in empirical evaluations, educational psychology has advocated for a complementary *a priori* analytical metric of complexity to support instructional design.

For example, research on Cognitive Load Theory (CLT), one of the most frequently used frameworks in CER, has been a focus of tension regarding its ability to predict the results of interventions precisely due to limitations in current methods to estimate *a priori* its effects [255, 268]. At the core of CLT is the ability to differentiate what is unavoidably required (complexity) to be processed by learners, and what is extraneous to learning. Recent re-conceptualizations of CLT try to accommodate such tensions, although recent studies show that intrinsic and extraneous elements of learning could be highly interconnected. Furthermore, some effects can only be observed when learners engage in difficult tasks [51]. Paas et al. [268] state that:

To determine the utility of CLT for generating instructional designs, it is important that the specification of what is intrinsic, [and] extraneous [...] can be done *a priori*. We argue that only the combination of analytical and empirical measures will enable us to determine intrinsic, [and] extraneous [...] cognitive load. If we can analytically identify the interactive elements of a task, the aspects of the task that interfere with schema construction and automation, and the aspects of the task that are beneficial to these processes, and if we can determine their cognitive consequences empirically, then it will be possible to determine empirically intrinsic,[and] extraneous [...] load, respectively."

---

1The quote has been adapted to suppress germane load, which is no longer present in current characterizations of the CLT.
Previous CER studies attempted to model the cognitive complexity of program comprehension (e.g., the Block Model [316]) by evaluating textual features of the code and its relationship with its goals. Cant et al. presented a framework, the Cognitive Complexity Model (CCM) [45], later applied to program comprehension tasks in eye-tracking experiments and cognitive simulations [140]. While the CCM shares with CLT similar conceptualizations of human cognition, its operationalization on concrete programs has proved to be difficult, and some metrics are still only theoretical approximations with little connection to the actual structures of the code. Our current research was inspired by such previous works but aimed to provide a more concise and clear conceptualization that could, potentially, be more easily applied by instructors and researchers.

Previous works describe the semantics of a given programming language in terms of a Notional Machine [84, 337], which presents a simplified model of program execution to learners. Such models are usually presented by instructors using diverse pedagogical approaches, such as visualizations, animations, or even oral explanations. Since Notional Machines are approximations of reality, they can be used to convey, or at least set instructors’ expectations of, learners’ mental representations of the rules of program behavior. For example, instructors could present a given loop simply as “repeat ten times”, while the same loop could be presented as “the variable x is initialized with the first value (0) in a sequence of numbers. At each step, the program increments x to the next integer value until it reaches the upper limit of the sequence”. While both Notional Machines describe the same semantics, they have different levels of detail. Learners whose instruction employed the later representation will have different cognitive demands since they need to coordinate semantic elements. Of course, they will also have a broader grasp of program behavior.

While previous works shed light on the program’s complexity from many different perspectives and had distinct limitations, to the best of our knowledge, the vast majority neglected the impact of prior-knowledge on complexity. Programs can be different, and each program might have its complexity analytically extracted. Still, the same program can be comprehended in different ways by different learners. In this thesis, we differentiate complexity, the analytically extracted characteristics of a program, from difficulty, the actual observable construct when applied to a given learner, and later discuss the impacts of such characterizations to assessment.
This research identified as gaps in previous works the lack of concrete, detailed formalizations system’s semantics instruction process that could elicit learners’ program comprehension process and set expectations of learners’ mental models. We identified the absence of an analytical model of cognitive complexity that can be operationalized on concrete programs comprehended by learners. Furthermore, since prior-knowledge is intimately related to the difficulty in comprehending programs, this research identified a lack of assessment instruments that could provide quick and granular quantification of learners’ prior-knowledge regarding program comprehension. While previous studies acknowledge the importance of program comprehension when learning to program, we identified a gap in how to operationalize program comprehension learning trajectories informed by program’s cognitive complexity and how to use tasks that could foster program comprehension.

Our analytical model of cognitive complexity derives mainly from two main theoretical frameworks. The first is schema theory and the related Cognitive Load Theory [355, 53, 23], which are concerned with the limitations of working memory and the growth of expertise as schemata in long-term memory. Our other primary influence is the Model of Hierarchical Complexity (MHC) [60], a neo-Piagetian theory concerned with the relative complexities of tasks.

1.2 Objectives and Scope

Traditionally, introductory programming courses favored problem-solving, create-first tasks usually associated with code writing activities. Studies have shown that code writing might be a complex task, with students failing to produce even simple programs at the end of their introductory courses [224, 232, 372]. One of the reasons for such disappointing performance from students, at least from the instructors’ perspective, is that such code writing activities presented from the onset of introductory courses might overwhelm learners’ cognitive resources. Soloway and colleagues have shown that even problems that appear to be simple, in fact, require the integration of many distinct elements simultaneously [335, 336, 344].

However, even at a beginner level, programming cannot be understood solely as writing code. It encompasses many facets, such as program comprehension, testing, and debugging, among possibly many other activities. Some authors provide evidence of a hierarchy of such skills, where learners progress from reading programs to tracing and finally being able to write
Introduction

code [360, 63, 193, 201, 115, 194, 361], similar to learners’ developmental stages of cognitive development. Inspired by such hierarchy, studies have presented novel pedagogical approaches emphasizing program comprehension abilities with promising results [232, 396, 256, 255].

In this research, we focus on the initial steps of programming instruction investigating the complexity of comprehending programs from the learner’s perspective. We use a broader definition of program comprehension, i.e., a process in which an individual constructs his or her mental model of a program using tasks where the learner is asked to interact with an artifact representing the program, which is broad enough to encompass a variety of tasks. While we acknowledge that other tasks (e.g., code writing) are equally important and perhaps even more widespread in introductory programming courses, this work is limited to the task of comprehending concrete programs. Figure 1.2 shows what kind of skills are possibly present in program comprehension tasks and their relationship with other programming skills. We believe that some aspects of our analysis and results could be extrapolated to other tasks, given the close interaction of programming skills, but it is beyond the scope of our work to discuss them in detail.

Figure 1.2. The many facets of programming. This research focus on a narrow set of skills used in program comprehension tasks.

![Diagram showing the many facets of programming](image)

The first contribution of this research is to present a novel conceptualization, the *Rules of Program Behavior*, which augment previous Notional Machines definitions and works, offering design guidelines for staged, explicit, and textual instruments to set expectations of learners’ mental models and communicate semantics instruction among practitioners.

The second contribution of this research is to propose a *theoretical framework for reasoning about the complexity of computer programs*. The framework analyzes the cognitive elements present in the design of a given program and the way those elements are intertwined.

The third contribution is the design and partial validation of a self-
evaluation instrument to assess program comprehension prior knowledge, inspired by the cognitive complexity of programs and theoretical frameworks successfully adopted by linguistics.

The fourth contribution of this research is an evaluation of activities used to foster program comprehension and how such activities could be sequenced based on their cognitive complexity to create program comprehension learning trajectories.

As a statement of this thesis:

“The rules of program behavior set the expected, based on learners’ instruction, cognitive actions that should be employed by learners when comprehending programs. It will be valuable to analytically extract metrics from concrete programs to describe how syntactical elements of the language (described by the rules of program behavior) and code-composition strategies, mediated by learner’s prior-knowledge, are sufficient to explain the cognitive effort demanded from learners in a code comprehension task. The cognitive complexity of programs can be used to design learning trajectories to foster program comprehension and inform the instructional design of programming courses.”

1.3 Research Questions

To define and evaluate the cognitive complexity of comprehending programs, its relationship with prior-knowledge, and how it could be used in instructional design, we investigated the following research questions:

- **RQ 1**: How to operationalize an a priori evaluation of the cognitive complexity of comprehending programs, and what are the implications of a model of cognitive complexity on the instructional design of introductory programming courses?
  - **RQ 1.1**: How to design representations of Rules of Program Behavior?
  - **RQ 1.2**: How to define and evaluate the cognitive complexity of comprehending concrete programs?
  - **RQ 1.3**: How to evaluate students’ prior-knowledge of program comprehension?
– **RQ 1.4**: What kind of activities can foster program comprehension and how such activities can be organized?

Each publication provides a partial solution to **RQ 1**, and the publications presented as part of this thesis answer the remaining research questions, as summarized by Table 1.1. This compiling part of the thesis combines the contributions of these publications.

**Table 1.1.** Research questions and corresponding publications

<table>
<thead>
<tr>
<th>Research Question</th>
<th>Publication I</th>
<th>Publication II</th>
<th>Publication III</th>
<th>Publication IV</th>
<th>Publication V</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ 1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RQ 1.1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RQ 1.2</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RQ 1.3</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>RQ 1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

In the discussion section of this thesis, we provide future directions that could lead to extensions of our work and possible applications to other tasks and skills. We offer a proposal on how a model of cognitive complexity of comprehending programs could inform instructional design and how it could fit frameworks such as the 4C/ID and the Ten Steps blueprint. In this proposal, we present an overview of CER through the lens of the 4C/ID, analyzing possible applications of current evidence and existing gaps in the field.

### 1.4 Research Process

CER has a tradition of being an interdisciplinary field. Being in the intersection of Education, Psychology, and Computer Science (see Figure 1.3), it often employs methods from these areas to provide rich and rigorous evidence. This thesis has its domain in the locus of Computing Education and cognitive psychology, with a particular focus on programming education, cognitive development, cognitive learning, and cognitive informed instructional design. The research was conducted mainly in higher-education contexts, although in the future, some of its conceptualizations could be extrapolated to other contexts.
**Figure 1.3.** CER is interdisciplinary by nature. This research falls in the intersection of cognitive development, comprehension of human cognition apparatus, and how programming activities interact with such apparatus.

Each publication part of this thesis used a different set of research methods to present evidence for our research questions. More specifically:

1. **Publication I** Adapts theories and conceptualizations from CER and other fields to a new context. This work presents a novel conceptualization for the Rules of Program Behavior, inspired in previous Notional Machines works. The Rules of Program Behavior can be used to set expectations of learners’ mental models based on their instruction and effectively communicate contextual decisions embedded in instructional design. We presented a case study to illustrate how an explicit, textual rule of machine behavior could be developed in incremental stages of completeness.

2. **Publication II** resorted to the tradition of building theories by elaborating conceptualizations, adapting previous theories, and using pieces of evidence from different fields applied to a new context. This work presents the theoretical basis for the development of a new model of cognitive complexity and presents a series of case studies to exemplify
how it could be applied to extract meaningful metrics of complexity from short programs.

3. **Publication III** uses quantitative methods to present the initial steps towards the validation of a Self-Evaluation Instrument (SEI) to assess students’ prior knowledge of program comprehension. We investigated SEI internal consistency metrics and performed a Factor Analysis (FA) and a Component Factor Analysis (CFA) to extract knowledge groups from the instrument. Next, we investigate if monotonically increasing or decreasing patterns could be extracted from SEI’s responses to provide evidence for a hierarchy of concepts and finally examine the correlation between SEI’s concepts and background variables.

4. In **Publication IV**, we provide the next steps towards the validation of the SEI by providing a longitudinal evaluation of students’ responses in a MOOC, collecting data at three distinct moments of the course. To investigate external validity, we collect students’ responses on a validated test instrument (the SCS1 [270]) and final course exam grade. To investigate students’ behavior when responding to each of the instruments, we examine the completion rates of those instruments at the end of the course. To investigate if the SEI could capture students’ learning over the course, we test SEI’s score differences at each data collection point and present the effect size of knowledge growth for each SEI concept. We performed a Confirmatory Factor Analysis to check if the SEI was able to capture the distinct nature of the programming concepts taught at the course and if those factors change over the weeks depending on learners’ experience. We conduct a replication of previous SCS1 studies and provide an in-depth analysis of our SCS1 results by conducting an Item Response Theory (IRT) analysis, comparing our results with previous studies. Finally, we investigate the correlation between the scores of the SEI, exam grades, and SCS1.

5. In **Publication V**, we conducted semi-structured interviews to analyze teacher’s perceptions of the usefulness of program comprehension activities, how they define program comprehension, and what kind of activities they use in the classroom. We used the results of such interviews to present a new conceptualization of program comprehension, and classify the tasks presented by teachers using the Block-Model (BM) framework.
We sequence those tasks using the Cognitive Complexity of Computer Programs Model (Publication II) to provide cues into how to break a complex program into smaller sub-tasks to fit into activities on each of the Block-Model cells.

1.5 Thesis Structure

In Section 2, we present previous works with Notional Machines, how expertise develops in other fields and computer science, and explore the theoretical conceptualizations of complexity in psychology and computer science. We present previous studies using different instruments of assessment of previous knowledge and how CER developed activities that focus on the program comprehension aspect of programming.

Section 3.1 presents a novel conceptualization of the Rules of Program Behavior, shows the guidelines of its design, and explores an incremental instrument through a case study. Section 3.2 presents our conceptualization of cognitive complexity of comprehending programs from the learners’ perspective and our formulation of the Cognitive Complexity of Computer Programs (CCCP) model to explore facets of cognitive complexity. Section 3.3 outlines the development and initial steps towards the validation of a quick Self-Evaluation Instrument to assess prior programming knowledge. In Section 3.4, we present how instructors use program comprehension tasks in the classroom and how those tasks could be decomposed and sequenced to create tentative learning trajectories.

In Section 4, we present the theoretical and practical implications of a cognitive model of program comprehension of Computer Science Education. We further explore those implications and discuss how such a model fits into a broader perspective of a review and mapping of CER through the lenses of the 4C/ID.
2. Theoretical Foundation

In this section, we present the theoretical frameworks and conceptualizations related to the cognitive complexity of programs. Section 2.1 presents how simplified models of program execution, Notional Machines, have been described and used by CER.

Section 2.2 presents the relationship between the human cognitive apparatus and how expertise develops through the lenses of schema and chunking theory, which provide models of cognitive structures used to represent knowledge. We introduce the conceptualization of plans, schema representation in a program, and how such conceptualization can be used to describe certain aspects of programming learning. Also, Section 2.2 discuss how programs’ plans can be organized into distinct structures using different plan-composition strategies.

Section 2.3 presents previous definitions and models of complexity from different contexts and how previous works have assessed the complexity of tasks. We outline the importance of analytical models of complexity and models of complexity presented by previous works. In this section, we also describe the relationship between expertise, complexity, and difficulty.

In Section 2.4, we present previous efforts to evaluate program comprehension knowledge using tests, questionnaires, and self-evaluation instruments. Section 2.5 discusses the relationship between programming skills and new pedagogical approaches advocating a program-comprehension-first strategy to teach programming. We also introduce possible activities usually used to foster program comprehension.

In Section 2.6, we present a framework designed to support complex learning, the Four Components/Instructional Design, and how it was successfully used in different fields to promote learning with authentic, complex tasks.

Figure 2.1 presents how the elements discussed before are related to
complexity and its implications in instructional design.

**Figure 2.1.** Relationship between different aspects of program comprehension cognitive complexity aspects.

2.1 Notional Machines

Computer Science devoted decades of research and development to provide layers of abstraction over sophisticated computational systems, freeing users from comprehending the intricate details of those systems and focusing on its usage through reduced interfaces. For some audiences, particularly novices, a complete grasp even of such reduced interfaces might prove to be a daunting task. Therefore, instructors routinely present simplifications of the system’s rules, its semantics, in a format that should be palatable for novices.

In CER we describe such simplification in terms of a Notional Machine (NM), a term introduced by Benedict du Boulay [85] where he describes the “difficulties associated with understanding the general properties of the machine that one is learning to control, the notional machine, and...
realizing how the behavior of the physical machine relates to this notional machine.”

While most previous CER works refer to du Boulay’s definition of NM (e.g., [367, 257, 71, 145, 158]), some recent works have presented similar but simplified versions of it. For example, Berry and Kölling [27] define an NM as “an abstraction designed to provide a model to aid in understanding of a particular language construct or program execution”, while Kohn and Komm describe the NM as “a model of the machine that conceptually executes the program code” [170]. Krishnamurthi and Fisler [175] bring the NM closer to the semantics of programming languages, presenting it as “[...] human-accessible operational semantics; A notional machine is a crisp, human-friendly abstraction that explains how programs execute in a given language or family of closely-related languages—i.e., a model of computation.”

Juha Sorva’s work [337] extends du Boulay’s definition by defining an NM as an “idealized construct to convey a simplified model of execution for computer programs; [...] a characterization of the computer in its role as executor of programs in a particular language or a set of related languages. A notional machine encompasses capabilities and behaviors of hardware and software that are abstract but sufficiently detailed, for a certain context, to explain how programs get executed and what the relationship of programming language commands is to such executions.”

Different actors in a system define and use different types of models. Conceptual models usually explain how a given system works, or a “explanation of a system deliberately created by a system designer, a teacher, or someone else” [338]. Conversely, a mental model refers to “a mental structure that represents some aspect of one’s environment” [337], a user’s representation of his or her subjective interpretation of the system. Some previous works present a different perspective of NM, closer to the conceptualization of a mental model. Bower and Falkner define an NM as a “a mental model that enables its user to make predictions about how a machine will perform” [36], while Lowe [202] presents an NM as “an individual’s mental model, representing how a programming language executes on a real device.”.

While undoubtedly related, most authors (e.g., [337, 175, 84]) agree that a mental model and NM are distinct concepts. Ideally, instruction will help learners to construct an accurate mental model of the NM [175], but often such mental models are simplifications, incomplete, inconsistent,
Theoretical Foundation

imprecise, and they sometimes can be based on guesswork and naïve assumptions and beliefs [337, 175].

Some aspects of NM are not precisely defined in the literature. Most authors agree that there is no single NM that satisfies a spectrum of programming languages or a computer in general. However, some authors (e.g., [145]) state that each programming language provides a single NM. Others (e.g., [175]) argue that “there can be many notional machines for a given language, reflecting different goals, degrees of sophistication, levels of abstraction, and so forth.” Krishnamurthi and Fisler go further, arguing that the “similarity between two languages is the extent to which a notional machine for one gives an accurate account of the behavior of the other.”

Some aspects of the NM conceptualization remain unclear, partly because very few studies present an explicit, concrete NM alongside teaching programming constructs. The majority of the papers use an implicit NM in visualizations and animations (e.g., [158, 15, 304, 27]) or it is implicit in the pedagogical approach. Even when the NM abstraction level is low, closer to interpreter instructions, such as in PLTutor [256], it is not immediately apparent what kind of characteristics such NM possesses. For example, Python Tutor [129] and JSVEE [332] implicitly use different NM when animating expressions. While Python Tutor implicitly evaluates the whole expression and presents its output, JSVEE evaluates the expression step-by-step.

Such differences in the NM used by JSVEE and Python Tutor reflect not only the level of detail and distinct presentation choices, but most of all values and pedagogical decisions embedded in such tools. Such differences are particularly relevant when instructors desire to investigate what kind of mental model representation learners possess of specific aspects of language semantics. Possibly, users of JSVEE would possess a more detailed mental model of expression semantics than those who learn with Python Tutor. This level of detail has a direct impact on how many cognitive actions, or steps, learners need to perform to comprehend a given concept.
2.2 Mental Representations of Knowledge: Schemata & Plans

Notional Machines (NM) describe a simplified model of a computational system semantics to a particular audience and context. Learners need to internalize NM elements using their expertise and previous mental representations of concepts, possibly creating new mental models. Ideally, instructors would have a reasonably good estimate of learners' prior knowledge and their existing mental models. In practice, investigating learners' mental models and misconceptions is a time-consuming and challenging task [33, 307]. NM representations, particularly concrete ones, can be useful by setting expectations for mental representations of a particular system, even not being perfect representations of each mental models. Given the current state of research regarding NM, it is not clear how mental models could be extracted from NMs. Could a given NM spawn several distinct mental models, depending on the context? Does a programming language possess multiple NMs?

In the following subsections, we explore how such mental representations are formed, stored, and retrieved in human memory. We present a model of human memory and how each of its components process information. Humans, in general, need to process and act on very complex information to create and retrieve knowledge. Usually learning to program requires that learners process complex information and learners are frequently asked to manipulate several aspects of programming simultaneously: retrieve keywords from a programming language syntax; remember the specific semantics used by different programming constructs; and at a higher level of abstraction retrieve known sub-steps for a problem solution and compose them into a coherent program.

With no expertise, this amount of information would undoubtedly over-
whelm us. In this section, we present mechanisms, such as chunking, that allow us to process and store increasing complex information, using structures such as schemata and templates. We show how experiments in a particular domain, chess players, have presented evidence on how expertise develops and how differences between novices and experts can be mostly attributed to the acquisition and retrieval of large chunks of information using such complex mental representations.

Next, we present studies that investigated how expertise in programming develops and how plans, schemata in the programming domain, can be used as language-independent idioms to represent programs. By using plans as the essential building blocks in programming, we finally discuss how articulating and composing those plans is a significant programming skill and how we can trace some programming difficulties to the plan-composition strategies used by learners.

2.2.1 **Human Memory Apparatus & Expertise**

Most conceptualizations of human memory in educational psychology and cognitive learning theories are based on a multiple-component model initially formulated by Atkinson and Shiffrin [12], later revised by many authors. Figure 2.2 illustrates a model of memory proposed by Gobet and colleagues (see [117, 119, 334]) consisting of four main components:

- A Sensory Memory (SM) system collects information from the sensory channels (e.g., visual, auditory) and is capable of storing information for a minimal amount of time (the exact estimate is not clear, but no more than a second [65]) and extracting features that serve as stimuli for other memory systems.

- A Short-Term Memory (STM, for practical purposes, in this thesis used in a similar meaning of Working Memory), allow humans to retain limited amounts of information for brief periods (seconds). The STM supports the acquisition of new knowledge, by processing present stimuli, or the retrieval of existing knowledge from the Long Term Memory (LTM). The most well-known conceptualization of the Working Memory from Baddeley [16] defines two subsystems, the phonological loop, and the visuospatial sketchpad, which are specialized for the processing and temporary maintenance of material within a particular domain (i.e., verbally coded information and visual and/or spatial information, respectively).
• A Long-Term Memory (LTM) stores large amounts of information using complex mental representations, possibly with no time limits. It acts as a discriminating network storing representations of knowledge and its relationships. The LTM is capable of contextualizing stimuli and retrieving knowledge for processing.

• The Central Executive (CE) controls and regulates the STM system. The CE performs various executive functions, such as coordinating the two subsidiary systems, focusing and switching attention (Attention Control System (ACS) from Gobet’s model, in Figure 2.2), and activating representations within LTM.

**Figure 2.2.** Human memory organization in a short-term memory (STM), a long-term memory (LTM) and mechanisms of sensory perception (adapted from [117, 119, 334]).

**Representing Complex Information**

The relationship between the LTM and STM and their distinctions in terms of capacity and time limits have been investigated for decades to present evidence for differences in performance and expertise. In his seminal work, Miller [234] presents a series of studies on information processing where he introduces a limit of seven (with a variation of two) elements to the STM capacity when stimuli are not intrinsically related to each other. Cowan [65] later revisited the channel capacity experiments to argue that the bottleneck that limits memory capacity is, in fact, located in our attention control system, regulated by the *focus of attention* [69, 136].

While the STM could potentially hold several stimuli from its subsystems (e.g., a detailed portrait of an image when immediately recalled, possibly from the visuospatial sketchpad), only a handful, roughly four elements, could be actively processed by the focus of attention in the STM. Other
elements could remain dormant in a region of direct access of the STM where they could be quickly retrieved if necessary, but do not leave a trace in the LTM. For example, one might be executing a task (e.g., performing calculations where numbers need to be active in the focus of attention) and get this task suddenly interrupted (e.g., a phone call). As long as the task (call) is not too long, it is possible that after its completion, some elements of the previous task could still be retrieved from the STM. However, if the call is too long, since no trace is formed in the LTM, this information will probably disappear.

With such a diminished processing capacity, what enables us to process large amounts of information is the ability to create and retrieve complex representations of information, or what Miller describes as chunks. The term itself has conveyed several meanings and interpretations in different fields and contexts (see Gobet’s discussion). Cowan defines chunks as “a collection of concepts that have strong associations to one another and much weaker associations to other chunks concurrently in use”. Given the limited capacity of the STM, forming increasingly complex chunks is what allows the retrieval of vast amounts of information for processing. With growing expertise, learners can combine similar information into larger chunks, using a recursive method to add other connected chunks.

**Chunking Information**

The ability to create chunks, chunking, and retrieve those chunks play a crucial role in expertise and provide explanations on differences in performance between novices and experts. Differences in expertise among chess players were primarily investigated by De Groot and later by Chance and Simon. Their main contribution (often misrepresented, see [30]) was to show the existence of apparent differences between players with distinct levels of mastery in a memory task involving familiar patterns, as in positions taken from a chess game. Typically, while master players can recall entire boards very accurately and weaker players (still experts) perform poorly, Chase and Simon found no difference when novices, experts, and masters were asked to recall random positions [119]. The central hypothesis in Chase and Simon’s work is that chess players perceive a board arrangement as chunks, and by chunking information into groups of similar features, master players increase the amount of information that can be actively processed. The STM, limited in capacity, holds pointers
to these chunks, possibly very large (see Figure 2.2), in the LTM, and recalling board positions mainly consists of unpacking those chunks.

Gobet et al. [120] describe two distinct chunking processes: deliberate and automatic. According to Gobet et al., deliberate chunking is “conscious, explicit, and intermittent with a specific goal to structure information to be processed”, and “can be based on distinct characteristics of the stimuli and at different levels of abstraction”. For example, learners can group items to create a chunk (e.g., the input 00011100 can be chunked as [000], [111], [000]); categorize items (e.g., the input [blue, yellow, Paris, black, Rome, Toronto] can be chunked as [blue, yellow, black] and [Paris, Rome, Toronto]); they recode information into a more meaningful representation (e.g., transform the value bit code 0010000100000100 to 83). Automatic chunking, contrarily, happens in the LTM and is unconscious, implicit, and continuous and happens when learners are acquiring expertise in a domain [120] or when acquiring biologically primary skills (e.g., facial recognition or first-language learning [108]).

**Schemata & Templates**

Chunking allows us to group large amounts of information that otherwise would overwhelm our STM. However, simply grouping similar information would not suffice to explain some expertise effects. Gobet and Simon [121, 122, 124] found that experts process information on a more abstract level than non-experts [116], which allowed them to retrieve patterns much faster than was previously considered. Also, chess masters still show a small but robust increasing ability to recall random positions when compared to less experienced players [116]. Gobet [116] argues that those findings suggest that a mental representation, often described as Retrieval Structures (RE), is used to store cues that could help us retrieve similar knowledge from the LTM.

The most famous conceptualization of RE is a schema (plural, schemata), and its studies fall under the umbrella of the Schema Theory [302]. Recently, comparable formulations of schemata have introduced the representation of knowledge as templates [123, 119, 118, 117, 116, 334]. The Template Theory [123] combines the concepts of chunks and retrieval structures, sharing several features with schemata, and given their similar characteristics, these terms are used interchangeably in this thesis, otherwise noted. For more details regarding specific differences between schemata and templates, see Gobet et al. [118].
Schemata are the basic units where knowledge and information on how to use this knowledge are stored in the memory. Schemata represent not only factual knowledge but also store (hierarchical) interconnections among related concepts and other schemata and meta information that allow us to contextualize such knowledge. A schema is represented as a green circle, and meta-information on the relationship between chunks is represented by dotted red lines in Figure 2.2.

According to Rumelhart [302], schemata possess as major characteristics:

1. Schemata have variables or attributes that serve as parameters to contextualize knowledge.

2. Schemata can embed, one within another. Schemata are a hierarchy of subschemata, each of which carries out its assigned task of evaluating its goodness of fit whenever activated. The hierarchical nature of schemata connections allows them to encapsulate new knowledge into existing schemata using a process of chunking.

3. Schemata can represent knowledge in all levels of abstraction. As expertise increases, more complete schemata are chunked to represent representations of knowledge.

4. Schemata represent knowledge rather than definitions. Schemata encapsulate factual information and contextual connections that allow such knowledge to be used.

5. Schemata are active processes, subject to changes whenever necessary.

6. Schemata act as recognition devices, which depending on the environment characteristics, evaluate their best fit.

The chunking, schema, and template theories have spawned a series of computational models such as ACT-R [6, 7], EPAM [96], and CHREST [118, 119, 117, 334] that offer validation for memory models using available empirical data. In the template theory, implemented by CHREST, chunks in the LTM are represented as a discrimination network (see Figure 2.2) and acquisition of knowledge is represented as the growth of such network, which develops both as a function of the current state of the system and
the input from the environment [117]. Retrieving information consists of a series of decisions in a hierarchical structure until a matching node (chunk) is found, activated, and later linked to the focus of attention in the STM [65].

Figure 2.3 shows how templates can represent multiple chunks in the LTM. Gobet [116] describes templates as a large chunk in their essence and parameters that could be loaded with different inputs. These parameters can have a default state based on more frequent stimuli and contain strategic and semantic information. Templates can be referenced within the STM by using pointers, therefore “occupying” a single memory slot. Gobet (ibid.) argues that templates are formed automatically during pattern recognition. When schemata are created or updated based on stimuli that match existing schemata in the LTM, the chunks can have an arbitrarily large size. However, if no matching pattern can be found in the LTM when acquiring new information, the newly formed chunk size is limited to four elements that should be held in the focus of attention simultaneously [65].

**Figure 2.3.** Template formation from related chunks in the LTM (adapted from [119]).

### 2.2.2 Plans

Computing Education has devoted a great deal of research to understand how expertise develops in programming, locate the source of bugs in the code, and comprehend how students create programs. Studies in the 1980 decade, contrary to the most accepted hypothesis at the time, found that a significant source of bugs in the code could not be attributed to poor understanding of syntax and semantics of the language. However, instead,
bugs were related to problems with limits in data structures, problem interpretation, and, most of all, due to difficulties in program composition [342].

Soloway, Spohrer, and others later showed that most of the errors in novices’ code originated in difficulties in structuring interconnecting parts of the program, or “putting the pieces together” i.e., composing and coordinating components of a program [335, 342, 345, 336]. Inspired by the Schema Theory, Soloway and colleagues proposed a novel perspective to investigate difficulties in programming instruction: they argued that “learning to program amounts to learning how to construct mechanisms and how to construct explanations” [335]. The building blocks of programming expertise are plans, which Soloway describes as “stereotypical solutions to problems as well as strategies for coordinating and composing them” [335]. What Soloway described as plans are essentially schemata in the programming domain.

Similar to the chess studies from De Groot and Chance and Simon, the difference in expertise in programming could be attributed to experts having built up large libraries of stereotypical solutions to problems as well as strategies to use them. Rist [296] showed that when schemata are available, both novices and experts resort to higher-level strategies, moving from abstract plans to concrete code (top-down). However, much like chess players recalling random positions, when existing schemata is not available, both novices and experts resort to a bottom-up strategy, expanding code around a known goal (or part of the code) in a process of focal expansion. Rist argued that programmers frequently change between bottom-up and top-down strategies based on their familiarity with the problem and existing plans.

To illustrate how plans are organized and later realized in the code, Soloway [335] proposed a problem, averaging the rainfall (The Rainfall problem), now a classic benchmark for CER as:

Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.

The Rainfall problem analysis illustrates how plans and goals could be used as a programming language independent idiom to describe a solution to a given problem. One of the exciting features of the Rainfall Problem is that its description is concise and straightforward, but its solution embeds
many interacting plans. For example, Figure 2.4 (a) shows (in gray) a possible set of required goals and plans to solve the Rainfall problem and how these plans are related to each other. One of the main difficulties students face when programming is to correctly interpret the problem, retrieve the necessary plans to solve it, and later be able to compose such plans in a coherent solution using a programming language.

**Figure 2.4.** Isomorphic plan structure from different problems, with different programs.

```plaintext
Goal
Write Program

Goal
Output Average

Goal
Stopping Condition

Goal
Protection Divide by 0

Goal
Counter

Goal
Division

Goal
Print

a)

Goal
Write Program

Goal
Output Average

Goal
Stopping Condition

Goal
Protection Divide by 0

Goal
Counter

Goal
Division

Goal
Print

b)

Goal
Write Program

Goal
Output Average

Goal
Stopping Condition

Goal
Protection Divide by 0

Goal
Counter

Goal
Division

Goal
Print
```

Plan and goals analysis decoupled the strategies used to solve a problem from its concrete code representation, allowing the comparison of different solutions across programming languages and freeing researchers from the minutiae and specifics of each programming language. Theoretically, plans would allow comparing different problems based on the similarity of their plan-goal structure. To illustrate how different problems could be compared, Soloway proposed another problem:

*Write a program that will output 'T' if all the inputs are 'T', but output 'F' if there is just one 'F' in the input sequence. Stop reading when a '#' is input.*
Both problems have distinct inputs, achieve different goals, and their description appears dissimilar. However, their plan structure (strategies) used to solve both problems are almost isomorphic (see Figure 2.4 (a) and (b)), even if the code used to implement those plans are different.

### 2.2.3 Plan-Composition Strategies

Plans and goals present an idiom to describe a set of strategies to decompose a problem. However, there are different ways to “glue together” these plans, a plan-composition strategy, that will have a direct impact on the actual code produced [335]. In Figure 2.4, the right side of the image presents the code for the plans of the left side of the image. While the plans itself have distinct goals, some parts of the code required to implement these goals (color-coded in the program) have overlapping elements or even plans that are spread in several parts of the code. In general, research has identified three main plan-composition strategies [335, 345]:

- **Abutted or sequenced plan-composition**, where plans are presented in sequence, one after the other.

- **Nested plan-composition**, where one plan completely surrounds another plan.

- **Merged plan-composition**, where at least two plans are interleaved.

Using different plan-composition strategies could yield very distinct programs. For example, listing 2.1 presents a merged solution to the Rainfall problem in Java. While some plans are nested (e.g., the print plan surrounded by the sentinel plan), overall, the data flow, control flow,
and shared variables make most of the plans interleaved. Listing 2.2 presents a program using a sequenced plan-composition for the same Rainfall problem, in Java. The extensive usage of functions allows the data and control flow to be compartmentalized, and plans can, in general, be evaluated separately in a sequence.

**Listing 2.1.** A merged plan-composition solution for the Rainfall problem.

```java
public class Main {

    public static void main(String[] args) {

        int[] rain = new int[]{};
        final int sentinel = 99999;
        int count = 0;
        float sum = (float) 0.0;
        float average = 0;
        int index = 0;
        int input = 0;

        while (true) {
            input = rain[index];
            if (input > 0) {
                if (input == sentinel) {
                    if (count == 0) {
                        System.out.println("No data!");
                        break;
                    }
                    System.out.println("Average is "+ average);
                    break;
                }
                count += 1;
                sum = sum + input;
                average = sum / count;
            }
            index += 1;
        }
    }
}
```

Theoretical Foundation

```java
private static List<Integer> myTakeWhile(List<Integer> numbers) {
    List<Integer> tmp = new ArrayList<Integer>();
    int i = 0;
    int value = numbers.get(i);
    while ((value != 99999) && (i < numbers.size())){
        tmp.add(value);
        ++i;
        value = numbers.get(i);
    }
    return tmp;
}

private static float avg(List<Integer> numbers) {
    if (numbers.size() > 0){
        float sum = numbers.stream()
            .reduce(0, (a, b) -> a + b);
        return sum / numbers.size();
    } else {
        return -1;
    }
}

private static float rainfall(List<Integer> numbers) {
    List<Integer> clean = myTakeWhile(numbers);
    List<Integer> filt = clean.stream()
        .filter(n -> n > 0)
        .collect(Collectors.toList());
    System.out.print(filt);
    return avg(filt);
}
```

Plan-composition strategies play a crucial role in learning to program. Some studies suggested that students find merged or interleaved programs hard to write or comprehend [335, 345, 110]. There is also evidence that students wrongly concatenate plans instead of merging them and frequently struggle with plans which interact in complex ways [111]. Stu-
students sometimes read merged plans as if they were composed sequentially [127], and there is evidence that students’ plan composition strategy may lead them to write defective code [100]. Also, there is some evidence favoring a sequenced-plans approach, which may lead to better outcomes when writing programs [336].

Although the Rainfall problem has been usually presented as the benchmark for plan-composition studies, and other problems were used in the literature, there is an ongoing discussion of its fit to modern programming language features. New problems that emphasize data, diverse methods of data input, extensive use of higher-order functions, and libraries could support providing better evidence of the impact of plan-composition strategies in performance [102].

The results from plans and plan-composition strategies have been influential in CER. Although several important questions regarding how students decompose code and how different plan-composition strategies have an impact on students’ comprehension of code remain, their results have spawned several pedagogical approaches that emphasize the teaching of higher-level abstractions over low-level syntax and semantics with promising results [303, 306, 176, 75, 395].

### 2.3 Complexity & Difficulty

When designing learning materials, or assessment instruments, instructors must consider the path learners take while learning. Routinely, teachers must decide the order in which materials are presented to learners, if the activities used in the classroom are not too demanding nor too easy, and if the assessment instruments match instruction levels of complexity.
It is fundamental, according to some learning hypothesis (e.g., the Cognition Hypothesis for second language learning [298], the Ten Steps for Complex Learning [374]), that pedagogic tasks should be designed, and then sequenced for learners based on increasing levels of cognitive complexity, gradually approximating classroom settings to the full complexity of real-world target task demands [298]. Sequencing tasks requires systematic methods of classifying the features of tasks, and what makes them complex. Those methods should account for the principles necessary to sequence these features, and combinations of them, in an order which approximates target-task demands [114, 298]. Preferably, such a method should be reliable and generalizable in different contexts.

Traditionally, CER has employed Bloom’s taxonomy [40, 114, 218] to classify the activities in which students engage during instruction or the SOLO taxonomy [29] to characterize the level of interconnectedness of learners’ outcomes. However, classifying tasks using Bloom’s taxonomy can be challenging even to an experienced group of programming educators [387]. Simon et al. [323] reported low inter-rater reliability rates (27%) using the Fleiss-Davies kappa when classifying tasks according to their intellectual complexity, a rate similar to those found by Gluga et al. [114]. Simon et al. [324] emphasize that some items had different interpretations among raters, and some items had a substantial mismatch between instructor expectations and learners’ performance.

Even when recent works provided clarified level descriptors, examples closer to CS context (e.g.,[364, 219]), and offered training in how to use Bloom’s taxonomy [112, 113], disagreement between instructors regarding how to assign a task to a particular level still exists. For example, Meerbaum-Salant et al. [229] argue that:

“...Creating is considered to be much more complex than Understanding, but can we really say that creating a simple project whose goal is to move one sprite from one point to another is cognitively complex than fully understanding the concept of concurrency?”

Difficulties in rating tasks might not only be restricted to Bloom’s taxonomy. Lister and Leaney [196] (when classifying tasks using Fuller et al. [105] taxonomy to CS) argue that:
“We felt that putting tasks of the same kind into different categories because of magnitude differences is not the proper solution and that the issue of magnitude (and therefore of complexity) should be addressed explicitly.”

Part of the difficulty of assessing CS activities may also be related to the number of interconnected concepts in the programs used in such activities. Some authors claim that learning to program itself can be considered difficult due to the high number of interacting elements in programs [220], including those used in introductory courses assessment instruments [204]. Attempts to measure the programs’ interacting elements and their complexity (e.g., [276]) resorted mainly to counting programming language’s syntactical elements without considering whether different concepts might have different intrinsic levels of difficulty.

Of course, it would be feasible to present an activity (e.g., comprehend a program) to learners and subsequently ask their subjective ratings of difficulty (e.g., using Morrison’s instrument [241]). Such an approach is used in many cases, and subjective ratings have proved to be a reliable measure of difficulty. However, a posteriori measures are not always adequate, and complementary methods to estimate the difficulty of a task need to be available for instructors.

For example, Mason et al. [220] argue that using a posteriori ratings are “simply not feasible when choosing a programming language and IDE for a proposed future version of a course”. Mason et al., instead, advocate for an a priori estimation of a given intervention complexity. Such estimation would help to distinguish the effects that contribute to cognitive load and would provide more robust evidence in support of the efficacy of interventions [220]. Some educational psychology theories, such as the Cognitive Load Theory (CLT) [355, 315, 53] have been the focus of debate for similar reasons. Predictability and falsifiability of CLT could be significantly improved by providing methods to distinguish, a priori, what elements are necessary to perform a task, from extraneous factors to learning. We will discuss in detail the CLT and the rationale for an a priori analytical evaluation of complexity later in this section.

In general, CER lacks a more precise definition of complexity and what are the constituents of the cognitive demands required when learning to program, in particular when comprehending programs. Overall, approaches based on taxonomies lack granularity and are difficult to generalize. Furthermore, previous works using taxonomies were mainly focused
Theoretical Foundation

on investigating activities related to programming, whereas a framework to investigate the complexity of comprehending programs is mostly absent in the literature. In the next section, we explore how other fields have defined complexity and difficulty and presented models to evaluate these concepts.

2.3.1 Definitions and Models Complexity and Difficulty

Liu and Li [198] present a comprehensive review of previous works in different fields that attempted to define complexity. Unfortunately, there is no consensus regarding how to define complexity, and, frequently, authors modeled factors that affect the complexity of a task instead. In their review, Liu and Li (ibid.) identified three distinct perspectives of complexity in tasks: the structuralist, resource requirement, and the interaction viewpoints. In the structuralist viewpoint, task complexity is understood from the structure of a task [198]. Proponents of structuralist models of complexity, in general, define complexity in terms of the number elements in the task and how these elements interact with each other. In the resource requirement viewpoint, according to Liu and Li, task complexity is defined as resource requirements. The more complex the task, the more resources are required from learners to perform the task [198]. In the interaction viewpoint, task complexity is defined as a product of the interaction between task and task performer characteristics (e.g., personal needs, prior knowledge, and experience) [198].

Several models have been described using one or more aspects of these viewpoints. Wood's [393] task model defines complexity as a function of task components, their interaction, and the effect of external factors on the task [198]. Campbell [44] models complexity based on the task's characteristics: multiple paths, multiple outcomes, conflicting interdependence among paths, and uncertain or probabilistic linkages [198]. Bonner's [32] complexity model describes three types of task complexity (input, processing, and output complexity), with each type consisting of two dimensions (amount and clarity of information) [198]. Robinson [297] argues that the cognitive factors contributing to complexity are a consequence of the structure of the task, which imposes resource demands. Therefore, task complexity is “the result of the attentional, memory, reasoning, and other information demands imposed by the structure of the task on the learner” [297].

In the Relation Complexity (RC) [137, 8] theory, complexity can be defined
by the number of slots or the “arity” of a relation required to perform a specific cognitive process [136]. Table 2.1 presents the arity in some examples of relationships according to the RC theory and at which age pupils, in general, should be able to perform tasks with a particular arity. According to Halford et al. [136], the relations in the RC theory constitute a coordinate system, so RC corresponds to the number of bindings to a coordinate system, consistent with Working Memory (WM) capacity constraints (see Section 2.2.1). To overcome WM limitations, chunking reduces complexity by decreasing the number of relations. Halford et al. (ibid.) use as an example the concept of speed, a ternary relation (speed, distance, time), which can be reduced to a unary relation (a given speed, 100 Km/h, as in a speed dial). However, when the factors in a relation need to be evaluated separately, the relation needs to be again unpacked, increasing its complexity. The segmentation process allows learners to break tasks into less complex steps, which can be processed serially. Strategies, algorithms, and hierarchical representations, where learners can process one level of the hierarchy at a time, are effective ways to segment a process and reduce complexity [136].

Table 2.1. Arity of relations and examples of task arity in the Relation Complexity (RC) theory [136].

<table>
<thead>
<tr>
<th>Arity of relation</th>
<th>Slots or variables</th>
<th>Example</th>
<th>Median age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>One</td>
<td>Class membership, e.g. cat(Smoke)</td>
<td>One year</td>
</tr>
<tr>
<td>Binary</td>
<td>Two</td>
<td>Larger(elephant, mouse)</td>
<td>1.5 years</td>
</tr>
<tr>
<td>Ternary</td>
<td>Three</td>
<td>Addition(2,3,5)</td>
<td>5 years</td>
</tr>
<tr>
<td>Quaternary</td>
<td>Four</td>
<td>Proportion(2,3,6,9)</td>
<td>11 years</td>
</tr>
</tbody>
</table>

Liu and Li [198] define task complexity as “the aggregation of any intrinsic task characteristic that influences the performance of a task”. In this definition, the term *aggregation* means that task complexity is an “integrative, multi-dimensional, global task characteristic that composed of other task characteristics”. The term *intrinsic* means that “task complexity does not depend on task performers and the environment”. Liu and Li (ibid.) argue that the essential criterion for judging whether or not a specific task characteristic is a part of task complexity is that it should influence the performance of a task (resource requirement viewpoint). Liu and Li used such definition to describe a Task-component-factor-dimension framework, where a task model with five components and twenty-seven factors (see Table 2.2) is used as inputs to a task-complexity model with ten components (see Table 2.3) and the same twenty-seven factors (rearranged
Theoretical Foundation

to fit into the ten complexity components).

**Table 2.2. Complexity contributory factors (CCFs)** [198].

<table>
<thead>
<tr>
<th>Task Component</th>
<th>CCFs</th>
<th>Relationship with Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>Clarity</td>
<td>Negative</td>
</tr>
<tr>
<td></td>
<td>Quantity</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Conflict</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Redundancy</td>
<td>Negative</td>
</tr>
<tr>
<td></td>
<td>Change</td>
<td>Positive</td>
</tr>
<tr>
<td>Input</td>
<td>Clarity</td>
<td>Negative</td>
</tr>
<tr>
<td></td>
<td>Quantity</td>
<td>U-Shaped</td>
</tr>
<tr>
<td></td>
<td>Diversity</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Inaccuracy</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Rate of change</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Redundancy</td>
<td>Negative</td>
</tr>
<tr>
<td></td>
<td>Conflict</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Guidance</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Mismatch</td>
<td>Negative</td>
</tr>
<tr>
<td></td>
<td>Non-routine events</td>
<td>Positive</td>
</tr>
<tr>
<td>Process</td>
<td>Clarity</td>
<td>Negative</td>
</tr>
<tr>
<td></td>
<td>Quantity of paths</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Quantity of actos/Steps</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Conflict</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Repetitiveness</td>
<td>Negative</td>
</tr>
<tr>
<td></td>
<td>Cognitive requirements by an action</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Physical requirements by an action</td>
<td>Positive</td>
</tr>
<tr>
<td>Time</td>
<td>Concurrency</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Pressure</td>
<td>Positive</td>
</tr>
<tr>
<td>Presentation</td>
<td>Format</td>
<td>Dependent</td>
</tr>
<tr>
<td></td>
<td>Heterogeneity</td>
<td>Positive</td>
</tr>
<tr>
<td></td>
<td>Compatibility</td>
<td>Negative</td>
</tr>
</tbody>
</table>
Table 2.3. Complexity dimensions [198].

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Number of task components</td>
</tr>
<tr>
<td>Variety</td>
<td>Diversity in terms of the number of distinguishable and dissimilar task components</td>
</tr>
<tr>
<td>Ambiguity</td>
<td>Degree of unclear, incomplete, or non-specific task components</td>
</tr>
<tr>
<td>Relationship</td>
<td>Interdependency between task components</td>
</tr>
<tr>
<td>Variability</td>
<td>Changes or unstable characteristics of task components</td>
</tr>
<tr>
<td>Unreliability</td>
<td>Inaccurate and misleading information</td>
</tr>
<tr>
<td>Novelty</td>
<td>Appearance of novel, irregular and non-routine events (e.g., interruption) or tasks that are not performed with regularity</td>
</tr>
<tr>
<td>Incongruity</td>
<td>Inconsistency, mismatch, incompatibility, and heterogeneity of task components</td>
</tr>
<tr>
<td>Action Complexity</td>
<td>Cognitive and physical requirements inherent in human actions during the performance of a task</td>
</tr>
<tr>
<td>Temporal Demand</td>
<td>Task requirement caused by time pressure, concurrency between tasks and between presentations, or other time-related constraints</td>
</tr>
</tbody>
</table>

**Difficulty**

So far, in this thesis, the terms complexity and difficulty have been loosely used to describe similar concepts. Often, this is also the case in the literature, as some authors do not clarify the distinction between complexity and difficulty in their definitions and models. However, Beckmann [23] argues that those terms should not be used interchangeably. Liu and Li [198] present a distinction between these terms, where task complexity “involves the objective characteristics of a task, whereas task difficulty involves the interaction among task, task performer, and context characteristics”.

Sweller [357] argues that a complex task can be effortless for one person, whereas a low complex task could be challenging for another. The schemata and template theory explain such a difficulty effect, since multiple interacting elements for one learner with low knowledge levels may constitute a single element for a learner with a higher level of knowledge. Robinson [297] argues that complexity and difficulty cannot be assumed to be in a fixed relationship to each other. Learners differing in intelligence or aptitude may experience different difficulty levels when performing a task.
due to their inherent ability differentials, and such difference may have an impact on performance [297]. Moreover, temporary limiting factors such as motivation can lead to more cognitive resources allocated to meet the demands of a task. According to Robinson [297], learners with relatively less aptitude or intelligence might have higher performance compared to learners with more ability due to temporary cognitive resource allocation.

Overall, Robinson [297] argues that “the task difficulty will help explain between learners’ variance in task performance due to learners’ available cognitive resources and previous experience. Task complexity will help explain within learner variance when performing any two tasks”.

Complexity and difficulty evaluation can be operationalized (e.g., using psychometric evaluations) using similar but distinct constructs that define cognitive load: mental load, mental effort, and performance. While intimately related, they are not a direct translation of complexity nor difficulty. Mental load represents the “estimated cognitive load a particular task imposes on a particular learner’s cognitive system” [23]. Task characteristics and expertise in the task domain are used to estimate mental load through task analyses and mathematical modeling. Mental effort is “the aspect of cognitive load that refers to the cognitive capacity that is actually allocated by a learner to perform a task” [268]. Mental effort can be measured while participants are working on a task (e.g., using physiological methods) or after task performance (e.g., using subjective ratings [241, 191]). Performance can be defined in terms of learner’s achievements, error rates, and time on task [268].

All aspects of cognitive load are essential to instructional design and offer complementary perspectives of a given task. For example, Paas et al. [268] argue that some instructional manipulations to alter mental load will only be effective if learners are motivated and invest mental effort to perform a task. Also, learners with different abilities can achieve similar performance levels by employing different levels of mental effort.

In summary, the models of complexity previously presented in this section provide a broad perspective of complexity, the factors that affect complexity, and its relationship with difficulty. However, these models only vaguely described complexity factors, making them difficult to operationalize in program comprehension tasks. While some dimensions from Liu and Li [198] model could be extracted from concrete programs (e.g., size), others are not easily observable (e.g., ambiguity, incongruity). We agree that the dimensions and factors described in these models are an essential part
of task complexity and should be incorporated to account for complexity factors in tasks such as code writing. However, in this thesis, we have a narrower goal to investigate not activities, but the concrete programs that are the core of program comprehension tasks. We aim to more precisely define constructs that would allow instructors to distinguish programs’ complexity and the cognitive demands required to comprehend them. Therefore, we are focused on models and theories that provide metrics that allow us to explore aspects we consider essential in the cognitive complexity of comprehending programs: the ability to evaluate the complexity of different programming elements present in computer programs and the role of expertise and the limitations of working memory when comprehending programs.

In the next sections, we explore the Model of Hierarchical Complexity (MHC), which will provide a learner independent metric of complexity, deeply connected with theories of cognitive development, and the Cognitive Load Theory (CLT), which presents a simplified model for learning within working memory constraints.

2.3.2 The Model of Hierarchical Complexity

The Model of Hierarchical Complexity (MHC) [58] is a neo-Piagetian, quantitative-behavioral, and developmental model that sets forth the measurement system by which actions are put into a hierarchical order [57]. The MHC offers a standard method of examining the universal patterns of evolution and development [57], and seeks to characterize the domain-specific stages of development that a learner goes through as they gain expertise in the domain and become capable of successful performance on increasingly complex actions.

Central to the MHC is the notion of hierarchical complexity as opposed to non-hierarchical (horizontal) complexity. Horizontal complexity is associated with bits of information, and it is often associated with difficulty. Commons [58] argues that performing twenty addition problems is of the same order of complexity as doing one addition problem, but most subjects would agree that a task that includes twenty problems is more difficult than one which includes only two, because it requires more work. More difficult tasks may demand more memory, be more time-consuming, could be affected by familiarity with the elements of a problem, among other factors.
Hierarchical complexity concerns the structural relationships between actions, in particular, the recursive relationships between a more complex action and its less complex sub-actions. The MHC defines actions as “behavioral events that produce outcomes” [58]. Actions may be combined to produce new, more complex actions. A task in the MHC can be defined as “a set of required actions that obtain an objective” [58]. An action is at a given stage of complexity when it successfully completes a task of a given hierarchical order of complexity. Roughly, hierarchical complexity refers to the number of non-repeating recursions that the coordinating actions must perform on a set of primary elements. Recursion refers to the process by which the output of the lower-order actions forms the input of the higher-order actions [60]. According to Commons [60], actions at a higher order of hierarchical complexity: a) are defined in terms of the actions at the next lower order of hierarchical complexity; b) organize and transform the lower-order actions; c) produce organizations of lower-order actions that are new and not arbitrary, and cannot be accomplished by those lower-order actions alone.

A simple task has the lowest level of complexity (see Table 2.4), and all other tasks have complexity, one greater than the complexity of their highest task demand [60]. The MHC defines three distinct organization rules (R) between actions:

- The **prerequisite rule** applies where succeeding at action A requires successful performance of exactly one other action at the same level of complexity as A. However, this does not mean that A is more complex than its prerequisite, only that successful performance on A is preceded by successful performance on it. The prerequisite rule defines *subtasks*. Commons et al. [58] exemplify that counting is one subtask action that is a prerequisite for addition. The addition is another subtask action and is a prerequisite for multiplication. They do not coordinate two or more actions but coordinate one action from the same order and one or more from lower orders. Such coordination does not result in an increment of order [58]. In Figure 2.5, prerequisite rules are illustrated as dotted arrows.

- The **chain rule** applies where a higher-level action A requires the organization of two or more lower-level actions in an arbitrary way: the lower-level actions are parts of A but can be carried out in any order,
and the whole is no greater than the sum of its chained parts. Commons et al. [58] present the case of \((a + b) + c\), where the organization of two actions of addition is arbitrary and no more hierarchically complex than addition in the evaluation of \((a + b) + c\) or \(a + (b + c)\), because addition is associative.

- **Coordination** (or concatenation) rule applies where a higher-level action \(A\) organizes two or more actions at a lower level of complexity in a non-arbitrary way. Coordination implies that the lower-level actions must serve distinct roles within the higher-level action; they cannot be simply swapped for each other or performed in an arbitrary order [58]. The distributive law is an example: computing \(2 \times (3 + 4) = (2 \times 3) + (2 \times 4)\) displays more complex behavior by giving addition and multiplication distinct roles rather than just performing the sub-actions separately. Coordination rules are illustrated as arrows in Figure 2.5.

The axioms of the MHC use these rules to create a hierarchy of complexity and define an order of hierarchical complexity \((h)\), defined as the number of recursions that the coordinating actions must perform on a set of primary elements [60]. For an action to be more complex than another, it must coordinate a minimum of two lower-level actions. Every primary (lowest-level) action \(A_0\) within a domain has the complexity level \(h(A_0) = 0\). Every more complex action \(A_k\) coordinates at least two lower-level actions \(A_i...j\) and has a higher level of complexity than any of them: \(h(A_k) = \max(h(A_i), ..., h(A_j)) + 1\). The recursive nature of the MHC postulates that someone who can perform at level \(n\) is also able to perform at level \(n - 1\). Therefore, learning occurs from less complex actions to more complex ones. Stage of performance of a given learner, \(\text{stage}(S, A)\), is defined as the highest order of the actions in \(A\) completed successfully by a learner \(S\). Table 2.4 illustrates the orders of complexity and a general description of the tasks performed at order of complexity.

The MHC has been empirically validated using Rasch analysis methods showing a positive correlation between the predicted complexity of equations in physics (the pendulum test) and measured student performance [59]. Dawson [73] compared an MHC based metric of learner development to other developmental scoring systems and found that it measured the same latent variables and was more internally consistent. The MHC has also shown to measure latent variables similar to SOLO taxonomy [349].
### Table 2.4. Orders of Hierarchical Complexity and Structures of Tasks [57].

<table>
<thead>
<tr>
<th>Order</th>
<th>Name</th>
<th>General Descriptions of Tasks Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Calculatory</td>
<td>Exact without generalization. Task: simple machine arithmetic on 0s, 1s</td>
</tr>
<tr>
<td>1</td>
<td>Sensory or motor</td>
<td>Discriminate in a rote fashion, stimuli generalization. Discriminative and conditioned stimuli. Task: Either see circles, squares, etc.</td>
</tr>
<tr>
<td>2</td>
<td>Circular sensory-motor</td>
<td>Form open-ended classes. Task: Reach and grasp a circle or square</td>
</tr>
<tr>
<td>3</td>
<td>Sensory-motor</td>
<td>Form concepts; respond to stimuli in a class successfully. Task: A class of open squares may be recognised</td>
</tr>
<tr>
<td>4</td>
<td>Nominal</td>
<td>Find relations among concepts. Use names; use names and other words as successful commands. Task: That class may be named, “Squares”</td>
</tr>
<tr>
<td>5</td>
<td>Sentential</td>
<td>Imitate and acquire sequences; follow short sequential acts; generalize match-dependent task actions; Task: The numbers, 1, 2, 3, 4, 5 may be said in order.</td>
</tr>
<tr>
<td>6</td>
<td>Pre-operational</td>
<td>Make simple deductions; follow lists of sequential acts; Use connectives: as, when, then, why, before; Task: The objects in a row of 5 may be counted; last count called 5, five, cinco, etc.</td>
</tr>
<tr>
<td>7</td>
<td>Primary</td>
<td>Simple logical deduction and empirical rules involving time sequence. Task: Perform simple arithmetic operations: (1 + 3 = 4; 5 + 15 = 20; 5 \times (4) = 20; 5 \times (3) = 15)</td>
</tr>
<tr>
<td>8</td>
<td>Concrete</td>
<td>Carry out full arithmetic, design, plan. Follow complex social rules, take and coordinate perspective of other and self. Task: There are behaviors that order the simple arithmetic behaviors: (5 \times (1 + 3) = 5 \times (1) + 5 \times (3) = 5 + 15 = 20)</td>
</tr>
<tr>
<td>9</td>
<td>Abstract</td>
<td>Discriminate variables such as stereotypes; use logical quantification; form variables out of finite classes based on an abstract feature. Make and quantify propositions; uses quantifiers (all, none, some); Task: All the forms of five are equivalent in value, (x = 5).</td>
</tr>
<tr>
<td>10</td>
<td>Formal</td>
<td>Argue using empirical or logical evidence; logic is linear, one-dimensional; use Boolean logic’s connectives (not, and, or, if, if and only if); form relationships out of variables; use terms such as if . . . then, therefore; Task: The general left hand distributive relation is (x \times (y + z) = (x \times y) + (x \times z))</td>
</tr>
<tr>
<td>11</td>
<td>Systematic</td>
<td>Construct multivariate systems, coordinate more than one variable as input; situate events and ideas in a larger context; Task: The right hand distribution law is not true for numbers but is true for proportions and sets.</td>
</tr>
<tr>
<td>12</td>
<td>Metasystematic</td>
<td>Integrate systems to construct multisystems or metasystems out of disparate systems; compare systems and perspectives in a systematic way (across multiple domains). Task: The propositional logic system and elementary set theory are isomorphic</td>
</tr>
<tr>
<td>13</td>
<td>Paradigmatic</td>
<td>Discriminate how to fit, and fit metasystems together to form new paradigms.</td>
</tr>
<tr>
<td>14</td>
<td>Cross-paradigmatic</td>
<td>Fit paradigms together to form new fields. Only by crossing paradigms can the new fields be conceived and formed; it requires the coordination of multiple paradigms to form genuinely new fields.</td>
</tr>
</tbody>
</table>
Theoretical Foundation

Figure 2.5. Distributive law stages of complexity [60].

1 + 2
x (y + z)

D = \{(+, \cdot), R_x\}
\text{h}(D) = \max(\text{h}(\cdot), \text{h}(\cdot)) + 1

C = \{[+, \cdot], R_y\}
\text{h}(C) = \max(\text{h}(\cdot), \text{h}(\cdot))

B = \{[+, \cdot], R_x\}
\text{h}(B) = \max(\text{h}(\cdot), \text{h}(\cdot)) + 1

A = \{[+, \cdot], R_x\}
\text{h}(A) = \max(\text{h}(\cdot), \text{h}(\cdot))

+ and x denote addition and multiplication operations on real numbers.
* and * denote addition and multiplication operations on variables.

when evaluating student outcomes. The MHC has been used for complexity analysis in diverse domains such as physics [346], chemistry [26], and student competence in graduate courses [235].

The MHC adopts the notion of the ideal task performance [60], whereas an ideal learner performs a task using an optimal set of actions using all available cognitive resources and is sufficiently motivated. Therefore, the measures obtained analytically by the MHC serve as a theoretical estimation of complexity. No claims are made as to cognitive structures of the brain, or about some overall stage of competence in the subject. Order of performance on one task is not necessarily generalizable to other tasks, even where the tasks share the same order of complexity and are found in the same domain. Performance can, and usually is, affected by aspects of horizontal complexity [60].

2.3.3 Neo-Piagetian Theory and Working Memory

Recent research shows that the growth of Working Memory (WM) capacity and the stages of cognitive development, such as those postulated by the MHC, might be intrinsically related [272, 76]. For example, the neo-Piagetian theories advocate that expansion in working memory capacity accounts for intellectual growth. The pioneering work of Jean Piaget [277] described developmental stages and how ability growth was related to specific age intervals, without providing robust evidence for such assumptions. Later, proponents of a novel interpretation of Piaget’s theory
(e.g., [273, 137, 8]) provided evidence of the relationship between developmental growth and an increase in the number of element relationships that can be kept simultaneously in the WM [67].

The relationship between stages of cognitive development and working memory capacity in the neo-Piagetian theory may have significant impacts on instruction. For example, Cowan [67] argues that “if the growth of capacity results only from the growth of knowledge, then it should be possible to teach any concept at any age if the concept can be made familiar enough”. However, some studies presented evidence that growth and maturation of WM capacity may be related to a primary (biologically) system development, therefore correlated with age (e.g., [70, 109, 66, 68]). Growth and maturation of WM would allow learners to focus and process more elements simultaneously, consequently allowing learners to cope with more complex tasks. Cowan (ibid.) concludes that while such interpretation is not yet definitive, knowledge differences alone cannot account for the age difference in working memory capacity. In practical terms, several aspects of working memory are likely to develop with age (e.g., capacity, speed, knowledge, and the use of strategies), and instruction should account for such aspects of development, primary or not [67].

The MHC presented a formalization of complexity based on an ideal performer using an ideal set of actions. However, often that is not the case in classroom settings (and probably elsewhere). The MHC can be valuable to set a theoretical hierarchy of concepts, but it does not provide a mentalist model of learning and its interaction with expertise. To explore how to evaluate the difficulty of comprehending concrete programs, we now set our sights in the Cognitive Load Theory (CLT), discussed in the next section.

2.3.4 The Cognitive Load Theory

Cognitive Load Theory (CLT) is an instructional theory inspired by the human cognitive architecture. Fundamental to the CLT is the relationship between working memory (WM), long-term memory (LTM), and their distinct characteristics. Given the limitations of the WM, both in capacity and duration (see Chapter 2.2.1), the CLT is primarily concerned with how to promote schema creation and automation, allowing learners to overcome WM limits by chunking increasingly complex schemata and storing them in the LTM. CLT aims to generate innovative instructional methods that avoid learners’ cognitive overload [52, 23, 268].
According to CLT, cognitive load can be analytically separated into two components: intrinsic and extraneous [53]. Sweller [357] defines intrinsic cognitive load as “the natural complexity of information that must be understood and material that must be learned, unencumbered by instructional issues such as how the information should be presented”. For a given task and given learner knowledge levels, it is fixed and cannot be altered other than by either changing the primary task or changing knowledge levels [357].

Kalyuga [162] defines extraneous cognitive load as cognitive processes that are not necessary for learning and are invoked by suboptimal instructional designs. While previous conceptualizations of the CLT described the third component of cognitive load, the germane load, current conceptualizations of the CLT [53] now refers only to germane resources, a function of the WM resources actually devoted to performing a task [357]. CLT contends that learning is facilitated when the instructional design eliminates extraneous load, optimizes intrinsic load (by matching learning task demands and learner expertise to avoid cognitive overload), hence maximizing the number of germane resources dedicated to learning [135].

However, in some cases, the distinction between intrinsic and extraneous cognitive load is not immediately apparent. First, what constitutes intrinsic or extraneous is mainly goal dependent [315, 357]. For example, in an introductory programming course, if the goal is to present the syntax of a programming language to beginners, syntax can be considered intrinsic. In contrast, if the goal is to teach strategies to solve a given problem using a programming language, syntax can be considered extraneous to learning.

Second, recent research has shown that intrinsic and extraneous elements of a task may be deeply interconnected since element interactivity is present in both intrinsic and extraneous cognitive load [51, 357]. Element interactivity refers to the number of cognitive actions that must be considered simultaneously in working memory when performing a task and is the major source of difficulty [139, 23, 357, 172]. Beckmann [23] defines the complexity of a task as “a result of the combination of the physical properties of the stimuli represented by the task per se and the requirements for a particular cognitive behavior”. More complex tasks will require not only more elements to be processed simultaneously, but the degree of interactivity between those elements is also a source of difficulty. Beckmann [23] argues that learners’ available schemata mediate difficulty by helping the learner deal with it in larger chunks, thereby reducing the
element interactivity - and, by extension, the difficulty - of a complex task. Some authors suggested that it is possible to reduce the complexity of a task by splitting it into subtasks. The isolated-interacting elements effect depends on artificially altering the element interactivity associated with the intrinsic cognitive load of a task. The intrinsic cognitive load of a task cannot be changed, but the task itself can be changed to a different task \cite{Beckmann2014}. Beckmann \cite{Beckmann2014} describes an experiment where tasks were split into subtasks, and instruction allowed the storage of intermediate results of each subtask, later recombined to achieve the complete goal of the task. The results of this work showed that such arrangement reduced the difficulty of the task, and learners performed better in the sequenced-tasks approach.

**The Demand for an a priori Evaluation of Complexity**

Part of the criticism of the CLT can be attributed to its lack of falsifiability \cite{Beckmann2014}. Many authors acknowledged it, and in recent years advocated for an analytical a priori metric of complexity (in terms of element interactivity) that would allow investigations to distinguish at design-time what is beneficial to learning and what hinders instruction.

Sweller \cite{Sweller2014} acknowledges that “by defining both intrinsic and extraneous cognitive load in terms of element interactivity, it may be possible to analyze element interactivity prior to an experiment and so more easily predict experimental outcomes”. Sweller argues that psychometric measures are not likely to improve interventions since the source of complexity would be unknown, and the effects of interventions could be attributed to either intrinsic or extraneous sources. An analytical a priori estimation of complexity could support CLT research by empirically distinguishing intrinsic and extraneous load and altering one form of cognitive load while keeping the other constant \cite{Sweller2014}. A priori complexity analysis can be used to provide hypotheses to be tested using conventional, randomized, controlled experiments.

Beckmann \cite{Beckmann2014} also advocates for an a priori investigation of the cognitive requirements of a task, and that “the differentiation between task and situation has to be based on an a priori task/situation analysis” \cite{Beckmann2014}. Some authors \cite{Beckmann2014, Sweller2014, Beckmann2014} argue that current subjective ratings may not be sufficient to determine the success of interventions, and a combination of analytical and empirical measures can improve the efficacy of CLT based interventions.
Furthermore, complexity is not always undesirable. Ideally, instructors must manage complexity to match learners’ abilities and knowledge [162, 315]. Only an analytical a priori metric of complexity is capable of achieving such a goal in instructional design. It is essential not only to know that something is difficult but what and why makes it difficult [162].

Besides, some effects predicted and described by CLT studies such as the worked example [356], split-attention, or redundancy effects [397] could only be observed under high-complexity conditions [162]. Being able to predict the difficulty of an intervention or task to a particular group of learners could better support instructors to select the appropriate pedagogical tools.

2.3.5 Evaluating Complexity

Previous works suggested several methods to evaluate complexity and difficulty in CS and other fields. Mostly, these methods can be distinguished by “when” (a priori, during task performance, a posteriori) or “how” (objective or subjective measures) cognitive load was evaluated. It is worth noting that the definitions of complexity and difficulty were loosely used in those methods, therefore in many occasions, authors referred to complexity, difficulty, and cognitive load interchangeably. Based on Liu and Li’s [198] distinction of these terms, in the next subsections, we adapted methods described in the literature to fit into such distinction. CS has adapted methods from other fields to evaluate complexity and difficulty, but also developed its particular methods to evaluate the program’s complexity, often using definitions that are dissimilar to those presented in this section. Next, we explore some of these methods and their benefits and shortcomings to evaluate complexity and difficulty.

Analytical, a priori Measures of Complexity

Analytical measures provide a priori objective methods to analyze the complexity of tasks and are usually performed before instruction to predict intervention effects. Most of the analytical measures resort to mathematical or analytical models to define and extract metrics of complexity from a given task. Analytical measures were one of the original evaluation methods in the CLT. Sweller [356] described a production system to construct computational models to count differences in the way the models functioned under different instructional conditions to indicate different levels of load [401][pg. 4].
Most CLT studies that attempted to analytically evaluate complexity a priori used approximations of element interactivity (e.g., [259, 51]). Such estimations of element interactivity were derived from a procedure proposed by Sweller and Chandler [358], which aims to establish what constitutes a learning element for a particular audience and then counting the number of interacting elements required to perform the task. An interacting element is one that the learner cannot process without simultaneously processing other elements. The number of such elements provides the element interactivity count [139]. Retnowati et al. [290] described complexity in terms of steps required to complete a task, but also acknowledged that the level of conceptual knowledge should also affect complexity. As discussed before, some studies conflate the notion of complexity and difficulty, even in analytical measures, and often the methods used to operationalize complexity are, at best, loosely described.

While models of complexity such as the RC [136] or the Task-component-factor-dimension framework [198] could be used to predict the complexity of a task, in practice, their dimensions are not easily operationalizable. In general, such models have been used to provide rule-of-thumb or approximations to complexity measurements to support instructional design. Mathematical models, such as the MHC, on the other hand, are a direct measure of complexity, although not of difficulty.

**Physiological Measures of Difficulty**

Previous studies used physiological measures as proxies for cognitive load, assuming that when cognitive load increases, significant physiological changes could also be observed [400]. Since such measurements can be done during task performance, physiological methods could, in theory, provide a more fine-grained evaluation of the cognitive load, including attributes of cognitive load such as peak, average, and overall loads [268]. Those physiological measures used a variety of methods such as spectral analysis of heart rate [269], cognitive papillary response [373], electroencephalography (EEG) [9, 180], functional magnetic resonance imaging (fMRI) [388, 81], among others [400].

While such physiological measures have provided sufficient evidence of their reliability in measuring cognitive load, they cannot mostly differentiate the types of cognitive load during task performance [241].
Subjective Posteriori Measures of Difficulty

One of the most used methods to evaluate the difficulty of activities are questionnaires of subjective rating of difficulty (mostly described as mental effort in the literature). In general, subjective ratings are an easy, quick, and reliable method to evaluate learners’ mental effort in a given task [14]. While some instruments used a single item to describe difficulty, most instruments were inspired by the NASA-TLX [142] scale. Leppink et al. [191] developed a ten-item instrument based on previous cognitive load instruments to evaluate the (at the time) three components of cognitive load, according to the CLT. Items evaluating intrinsic load (IL) correlate (partially) with task complexity. Overall, subjective rating instruments have shown to have good reliability.

However, rating scales may subject to criticism regarding its validity, particularly content validity [172]. Korbach [172] argues that subjective ratings may suffer from timing effects and provide no continuous information about the actual cognitive load. Moreover, attempts to improve accuracy by continuous application of subjective ratings may interrupt a task and interfere with learning.

Complexity Evaluation in Computer Science

In general, computer scientists associate the term complexity to the long-established field of computational complexity (e.g., using asymptotic notation methods). Initial attempts to quantify complexity by Software Engineering (SE) resorted to methods inspired by computational complexity, where surface-level metrics attempted to model users’ difficulty in managing programs by weighting programming language constructs. Metrics such as Cyclomatic Complexity [222], Halstead metrics [138], lines of code, and block depth have been applied to a variety of programming tasks (e.g., code comprehension [163], code-writing [87]). A myriad of other metrics explored similar methods of weighting constructs have been proposed (e.g., [398, 182, 239]) in an attempt to improve the accuracy to evaluate complexity. While such metrics provided a perspective of program’s complexity, they have a low correlation with perceived complexity [164], do not indicate how different constructs affect complexity [2], disregard the role of prior knowledge and are not grounded on any meaningful theory of learning.

Morrison [241] adapted Leppink’s [191] cognitive load instrument into the CS context. Complexity, through intrinsic load, is evaluated by a three
Theoretical Foundation

item component:

1. The topics covered in the activity were very complex.

2. The activity covered program code that I perceived as very complex.

3. The activity covered concepts and definitions that I perceived as very complex.

Using such an instrument, Morrison [241] showed that subjective ratings were able to capture differences in intrinsic load in two different contexts (sound processing and processing lists) and, as predicted, the sound processing activity was less cognitive demanding than the list processing activity. However, Morrison (ibid.) argues that even if the subjective rating scales are practical and straightforward to use and have proven reliable through repeated use, they only deliver a one point post hoc assessment of the cognitive load imposed by the learning task. Also, subjective ratings cannot clarify which of the specific aspects of the learning situation caused the level of cognitive load reported by students [241].

Other approaches evaluated complexity from a more concrete perspective, analyzing individual programs. Luxton-Reilly et al. [204] used the Java abstract syntax tree (AST) to explore the relationship among concepts and extract hierarchical representations of concrete programs. Luxton-Reilly and Petersen [207] defined the Co-Coverage component metric (Co-C) to represent the average number of concept components that appear in a question involving a specific component. The Co-C metric can facilitate the analysis of questions at the syntax level, indicating the complexity of the code that includes a given component. Similar work was presented by Mead et al. [228], which links concepts in intuitively-constructed “anchor graphs”. Anchor graphs also describe a form of hierarchy, where the learning an earlier concept ought to carry some of the cognitive load to later concepts.

Sheard et al. [322] propose a framework to evaluate the complexity and difficulty of exam questions based on multiple dimensions, as presented by Table 2.5. Complexity itself was evaluated using six dimensions, while difficulty is one-dimensional. Their results show a reasonable agreement between raters and a good correlation among dimensions. However, it is an open question if, given the subjective nature of the categorization, such a
framework would yield similar evidence in other contexts, and the results, would be in fact, be comparable across contexts.

**Table 2.5. Complexity measures and level of difficulty [322].**

<table>
<thead>
<tr>
<th>Measure</th>
<th>Focus</th>
<th>Classification values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>External domain references</td>
<td>Question</td>
<td>low, medium, high</td>
<td>Reference to a domain beyond what one would reasonably expect introductory programming students to know</td>
</tr>
<tr>
<td>Explicitness</td>
<td>Question</td>
<td>high, medium, low</td>
<td>Extent of prescriptiveness as to how to answer the question</td>
</tr>
<tr>
<td>Linguistic complexity</td>
<td>Question</td>
<td>low, medium, high</td>
<td>Length and sophistication of the natural language used to specify the question</td>
</tr>
<tr>
<td>Conceptual complexity</td>
<td>Question &amp; Answer</td>
<td>low, medium, high</td>
<td>Classification of the individual programming concepts required to answer the question</td>
</tr>
<tr>
<td>Code length</td>
<td>Question &amp; Answer</td>
<td>low, medium, high, NA</td>
<td>Whether code is up to half a dozen lines long, up to two dozen lines long, or longer</td>
</tr>
<tr>
<td>Intellectual complexity</td>
<td>Question &amp; Answer</td>
<td>knowledge, comprehension, application, analysis, evaluation, synthesis</td>
<td>Bloom’s taxonomy as applied to programming</td>
</tr>
<tr>
<td>Level of difficulty</td>
<td>Question &amp; Answer</td>
<td>low, medium, high</td>
<td>Subjective assessment of difficulty of question</td>
</tr>
</tbody>
</table>

Cant et al. [45] proposed a set of cognitive complexity metrics (CCM) to quantify the cognitive processes involved in the development, modification, and debugging of computer programs. While Cant et al. provide definitions for the factors that influence each process, some factors were not defined and are just placeholders for future empirical studies. Hansen et al. [141] discuss each factor from the CCM and how they could be potentially operationalized and simulated in a simulation framework for psychological models such as the ACT-R [7].
Difficulty can not be evaluated without a deeper understanding of learners’ prior-knowledge and mental models. Therefore, learners with different backgrounds will experience programming courses in different ways. In general, researchers agree that prior programming experience is an essential factor that influences learners’ performance in introductory programming courses. Research shows that at the beginning of the course, learners with some prior knowledge in CS outperform learners with no experience [146, 391], and prior knowledge can be used (with moderate effects) to reduce the gap between experienced/inexperienced learners by introducing intermediate level CS1 courses [167].

Students increasingly come to introductory programming courses with different programming expertise levels. With growing, but unequal, access to programming education in elementary and high schools, it becomes essential to develop prior-knowledge assessment instruments [134, 321, 359, 382, 389, 392] and use them to attend a cohort’s prior knowledge.

However, specific assessment instruments have distinct qualities. Sometimes, a detailed and accurate instrument is necessary; at other times, just an overall grade is enough. In some contexts, it will be acceptable for learners to self-assess their abilities; in others, an external evaluator must be required. While specific assessments must be mandatory, others should remain voluntary. In some cases, it is acceptable that assessment takes time, while in others, the assessment must be quick even if less accurate.

Assessment instruments of different nature can be used to achieve several goals. For example, program comprehension assessment instruments can be used to provide feedback to learners, provide tailored activities to individual learners [46], promote self-regulation and self-reflection oppor-
tunities [402], and identify learners at-risk [383]. Some assessment instruments in CER were used to compare learners’ performance in program comprehension [196] across institutions. Some validated instruments are more flexible, providing a multilingual context to evaluate comprehension skills in introductory courses [363, 270].

In some fields, such as linguistics, validated evaluation instruments such as the TOEFL [88] and IELTS [54] have been successfully used for decades to compare and certify learners' proficiency. Other instruments, such as the CEFR [265] can be used as a quick rubric of learners’ performance and even provide comparable levels of proficiency across skills and languages. CEFR instruments have both a writing and a comprehension component. Furthermore, while validated tests can be potentially more accurate, on average, the CEFR can be used to develop self-evaluation instruments and present to learners an opportunity to reflect on their learning skills. In this thesis, we focus on instruments mainly aimed at comprehension skills.

In the next subsections, we explore the different assessment instruments used in CER and elsewhere. We present in more detail the relevant work connected to self-evaluation instruments in CER and how language skills assessment, particularly the Common European Framework of Reference, evaluates learners’ levels of ability.

### Tests

Perhaps the most well-known and frequent kind of assessment instrument used in introductory courses is a test. Recent work points out that perceptions of difficulty in learning to program may be due to poorly designed CS tests, which require the integration of too many elements [203]. More reasonable instruments aimed at beginners should employ more granular assessment questions [204, 254]. Therefore, more detailed information regarding learners’ knowledge could be used to set more realistic expectations of performance and to design assessment instruments matching learner’s abilities. While we do not present an extensive list of relevant tests in this thesis, Margulieux et al. [214] provide a comprehensive compilation of validated test instruments.

Recently, CER presented test instruments aimed at program comprehension and code extension, such as the FCS1 [363] and SCS1 [270]. Both instruments cover concepts usually presented at introductory courses such as variables, assignment, input and output, conditional statements, loops, data collections, functions and methods, and classes and objects. The FCS1
and SCS1 were validated in different contexts, providing accurate and reliable results. However, these instruments still have some limitations. As other validated test instruments (e.g., [88]), they are not generally available due to validity concerns, therefore making them difficult to be used in repeated-measurement study designs. In general, tests are time-consuming and difficult to scale. Although the FCS1 and SCS1 use pseudocode to not be bound to a particular programming language syntax, these cannot be considered language independent. Students with different backgrounds (e.g., visual blocks or functional programming language users) could be disproportionately affected.

### 2.4.2 Questionnaires

Questionnaires can be flexible, easy to administer, and quick assessment instruments frequently used to evaluate prior knowledge in CS courses. Students have been asked, for example, if they have any prior programming experience [1]; how many programming languages they have used [134]; and what is the largest program they have written [246].

Feigenspan et al. [97] developed a comprehensive questionnaire that includes several of the background variables used in previous works (see Table 2.6). Feigenspan et al. results support that self-evaluation questionnaires can be useful for measuring experience when compared with a program-comprehension test instrument. However, in general, programming experience is treated as a single construct [147, 166]. Even if the metrics used in the questionnaire have shown to be correlated with grades, they poorly describe specific skills or knowledge that learners possess. They can help instructors to have a rough understanding of experience (e.g., group learners in “beginners” or “experts” groups), but offer little support to tackle specific needs of each learner regarding their instruction.
Table 2.6. Programming experience questionnaire from Feigenspan et al. [97].

<table>
<thead>
<tr>
<th>Question</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>On a scale from 1 to 10, how do you estimate your programming experience?</td>
<td>1 - very inexperienced 10 - very experienced</td>
</tr>
<tr>
<td>How do you estimate your programming experience compared to experts</td>
<td>1 - very inexperienced 5 - very experienced</td>
</tr>
<tr>
<td>with 20 years of experience?</td>
<td></td>
</tr>
<tr>
<td>How do you estimate your programming experience compared to your classmates?</td>
<td>1 - very inexperienced 5 - very experienced</td>
</tr>
<tr>
<td>How experienced are you with the following languages: Java/C/Haskell/Prolog</td>
<td>1 - very inexperienced 5 - very experienced</td>
</tr>
<tr>
<td>How many additional languages do you know (medium experience or better)?</td>
<td>Number</td>
</tr>
<tr>
<td>How experienced are you with the following programming paradigms:</td>
<td>1 - very inexperienced 5 - very experienced</td>
</tr>
<tr>
<td>functional/imperative/logical/object-oriented programming?</td>
<td></td>
</tr>
<tr>
<td>For how many years have you been programming?</td>
<td>Number</td>
</tr>
<tr>
<td>For how many years have you been programming for larger software projects,</td>
<td></td>
</tr>
<tr>
<td>e.g., in a company?</td>
<td></td>
</tr>
<tr>
<td>What year did you enroll at university?</td>
<td>Number</td>
</tr>
<tr>
<td>How many courses did you take in which you had to implement source code?</td>
<td>Number</td>
</tr>
<tr>
<td>How large were the professional projects typically?</td>
<td>NA, &lt;900, 900-40000, &gt;40000</td>
</tr>
<tr>
<td>How old are you?</td>
<td>Number</td>
</tr>
</tbody>
</table>

2.4.3 Self-Evaluation Instruments

Although both questionnaires and self-evaluation instruments rely on learners’ self-reported data, in this thesis, we distinguish questionnaires, which reflect one’s broad estimation of expertise, from self-evaluation instruments, which reflect one’s estimation of ability in a particular aspect of the task or skill. Since in literature the terms self-assessment and self-evaluation are often used interchangeably, in this thesis, we adopted the term self-evaluation.

Although self-evaluation has been researched in numerous contexts, including health professions [125], classroom teaching [300], language teaching [265] and MOOCs [177], there are still concerns regarding the validity and reliability of its measures. Ross [300] presents supporting evidence for the reliability of self-evaluation in terms of internal consistency, consistency across tasks and items, and over short periods. However, factors such as age, experience, grading expectancy, and prior achievement affect learners’ self-evaluation [300]. Therefore, often teacher’s assessments may only correlate moderately with self-evaluation measures. Furthermore, learners with little experience in self-evaluation may over or underestimate their abilities.

Nevertheless, self-evaluation can contribute to higher student achievement, mainly when students are guided in this process. Miller [238]
Theoretical Foundation

shows that ceiling effects in self-evaluation studies are related to overly vague or confusing scoring criteria. Rubrics, for example, could provide a criterion-based self-evaluation, which helps to understand the goals and requirements of a course [300]. Instrument sensitivity can be increased by adding measurement levels and precisely defined criteria. Furthermore, self-evaluation is a skill. Research has shown that the validity, accuracy, and effectiveness of self-evaluation can be improved with repeated measures, providing qualitative formative feedback to learners [300], and allowing learners to compare their self-evaluation with teacher's and peer's measures [125, 177].

Self-assessment can be critical to learners’ self-regulated learning [173] and closely related to self-efficacy. Danielsiek et al. [72], for example, presented a validated instrument to indirectly assess learners’ computing skills in algorithms courses through the lens of self-efficacy: the students’ belief in their ability to perform specific tasks. Literature shows that self-efficacy correlates strongly with successful learning [17, 192, 318] and can be assessed rapidly. However, self-efficacy measures are not a direct measure of students’ conceptual understanding.

In this thesis, we distinguish self-evaluation from self-efficacy measures. Self-evaluation is the evaluation or judgment of “the worth” of one’s performance and the identification of strengths and weaknesses to improve one’s learning outcomes [300]. In short, while both measures are related and overlap, self-evaluation is particularly suited for assessing past or current work, and self-efficacy attempts to estimate the confidence and extrapolate future performance, as presented in Figure 2.6.

Figure 2.6. Self-evaluation and Self-efficacy interactions over time.

![Figure 2.6. Self-evaluation and Self-efficacy interactions over time.](image-url)
CER explored self-evaluation instruments in introductory and advanced courses. Murphy and Tenenberg [251] showed that learners’ self-evaluation of performance has a moderate correlation with their actual test score in a data structures test, and learners with higher scores were more accurate. Kallia and Sentance [161] also investigated the accuracy of self-evaluation questions, showing that learners who have few misconceptions tend to overestimate their performance in the self-evaluation.

Ngai et al. [258] presented supporting evidence that self-evaluation could reduce learners' anxiety. Their work evaluated three distinct skills (debugging, coding, and programming) with a provided rubric (see Table 2.7) and found a strong correlation between learners’ self-evaluated grades and instructor-given grades.

Table 2.7. Ngai et al. rubric for self-evaluation [258].

<table>
<thead>
<tr>
<th></th>
<th>Debugging</th>
<th>Coding</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>It is hard for me to figure out what is wrong with other people's code.</td>
<td>I don't think that I could code in a problem, or my solution is not likely to compile.</td>
<td>I don't think that I could write a program to solve a problem.</td>
</tr>
<tr>
<td></td>
<td>I can identify syntax errors if I compile the code and track down the errors using the compiler.</td>
<td>I can code in a problem from a step by step description and get it to compile, but it may not give the correct answer.</td>
<td>I can write a program to solve a problem, if parts of it look similar to things I have seen before.</td>
</tr>
<tr>
<td></td>
<td>I can read other people's code and identify syntax errors without needing to compile it.</td>
<td>I can read other people's code and guess what it was intended to do and correct syntax and logical errors.</td>
<td>I can write a program to solve a problem.</td>
</tr>
</tbody>
</table>

Alaoutinen and Smolander [5] were inspired by Bloom’s Taxonomy [31] levels to design a self-evaluation instrument aimed for an introductory programming course. For each one of the concepts usually covered in a CS1 course (data types, operators, data input and output, conditional statements, loops, functions, recursion, library calls, arrays, strings, files, pointers, dynamic memory, program design, implementation, testing and
debugging, version control) learners were asked to self-assign into one of
the levels described by Table 2.8. The students’ self-evaluations correlated
strongly with exam grades, and the topics considered more difficult by
teachers were also rated more difficult by the students.

Table 2.8. Alaoutinen and Smolander [5] levels of programming ability.

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I can list related commands/concepts.</td>
</tr>
<tr>
<td>2</td>
<td>I can explain what the command/concept means. I can apply an example to a similar problem.</td>
</tr>
<tr>
<td>3</td>
<td>I can list cases when the command/concept can be used. I can apply an example to a different problem.</td>
</tr>
<tr>
<td>4</td>
<td>I can explain the meaning of the command/concept in its context, why it is there.</td>
</tr>
<tr>
<td>5</td>
<td>I can ensure the correct use of the command/concept.</td>
</tr>
<tr>
<td>6</td>
<td>I can (use) the command/concept in problem-solving without an example.</td>
</tr>
</tbody>
</table>

The Common European Framework of Reference

Given the range of languages and educational contexts across European educational institutions, the Education Department of the Council of Europe developed a general framework of language ability, the Common European Framework of Reference (CEFR) [265] so that instructors could more reliably evaluate learners’ ability across distinct contexts.

The CEFR proposes a cumulative range of language skills levels from A0 (unfamiliar) to C2 (mastery), as shown by Figure 2.7. Learners that master a given level should also be able to operate in all its precedent levels. Skills are categorized into separate areas, such as listening, speaking, reading, or writing. While the descriptors used by each skill and area are highly contextualized (e.g., reading for leisure, writing academic texts, write prose), skill level's definitions and ranges remain somewhat similar, as illustrated in Table 2.9.

The common framework allows both students to estimate their own skills and teachers to gain a quick understanding of each student’s approximate skills. When offering standardized language tests, such as TOEFL [88] and IELTS [54], the test results can be linked with the CEFR levels, allowing instructors to understand the overall skills levels of students in different
Theoretical Foundation

skills.

**Figure 2.7.** CEFR cumulative levels of ability.

![CEFR Levels Diagram](image)

**Table 2.9.** Generic level descriptors extracted from the CEFR guidelines [264].

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>The learner is unfamiliar with the skill/concept in any context.</td>
</tr>
<tr>
<td>A1</td>
<td>The learner can transfer surface features from similar skills/concepts in other contexts into the current context.</td>
</tr>
<tr>
<td>A2</td>
<td>The learner can recognize surface and structural features of the skill/concept in the current context.</td>
</tr>
<tr>
<td>B1</td>
<td>The learner can comprehend the meaning of a concept (or skill), in generic terms, when it uses a small number of features.</td>
</tr>
<tr>
<td>B2</td>
<td>The learner can comprehend the meaning of the concept (skill) that uses a large number of features when there is familiarity in the context and the structure used to present it.</td>
</tr>
<tr>
<td>C1</td>
<td>The learner can summarize a concept (or skill) when there is familiarity with the context and the structure used to present it, or aids are provided in case of no familiarity in the context.</td>
</tr>
<tr>
<td>C2</td>
<td>The learner can summarize the meaning of a concept or skill and understand its underlying structure. The learner can transfer the acquired knowledge to different contexts.</td>
</tr>
</tbody>
</table>
When McCracken et al. [224] presented evidence that introductory programming students performed below teachers’ expectations when writing code\(^1\), it raised concerns regarding students’ abilities to write code and how much was being asked from students in introductory courses.

Mainly, CER explored alternative methods of instruction that introduce other skills before writing code. Some authors provide evidence that programming skills are organized as a hierarchy, and instruction should account for this progression [360, 63, 193, 201, 115, 194, 361]. The findings of the BRACELet project [55] support a hierarchy of skills, from reading to tracing, summarizing, and finally writing code. Their findings also suggest that beginners tend not to abstract beyond the concrete code, as opposed to experts that can summarize the goal of the program; the ability to trace code indicates that learners are ready to comprehend the relationships between parts of the code; the ability to summarize the goal of the program in short sentences correlates positively with code writing skills (also supported by [250]).

In general, tasks such as reading [43], tracing [256], summarizing [250], or manipulating code [154, 361, 271] have been described under the program comprehension term, i.e., any activity that supports an individual to construct his or her mental model of a program by interacting with an artifact representing the code. Since those skills can be considered precursory to writing code and could produce a more efficient learning trajectory in introductory programming courses, it is crucial to understand the relationship between program comprehension and other programming

\(^1\)Such findings may have been affected by lack of support or excessive expectations from teachers [372].
Theoretical Foundation

2.5.1 Models of Program Comprehension

Multiple models have been proposed to analyze program comprehension. Wiedenbeck and Ramalingam [390] describe Pennington’s model of program comprehension [275, 274] as a mental model approach, inspired by models of textual comprehension. Pennington’s model introduces a verbatim level, representing the code itself, and two mental representations levels above it: the program model and the domain model. The program model contains information on programming language structures and relationships immediately available from the verbatim level. The program model uses knowledge of basic elements of the code, usually corresponding to one or a few lines of the program, and knowledge of control flow, which direct which parts of the program will be executed at a given point. The domain model refers to the context in which the program is executed, described in terms of data flow, i.e., changes on data through program execution; and function information, i.e., program goals in a problem. Usually, domain model knowledge requires more expertise and practice from learners than the program model.

Wiedenbeck and Ramalingam [390] used Pennington’s model of program comprehension to investigate learners’ program comprehension strategies in procedural and object-oriented paradigms. Wiedenbeck and Ramalingam show that novices tend to develop a strong mental representation of function-related knowledge in object-oriented programs, but weaker in terms of data flow and program-related knowledge. By contrast, novices’ mental representations of procedural programs were more robust in program-related knowledge.

The Block Model [316] is an educational framework that supports the analysis of aspects of program comprehension. The Block Model (illustrated in Table 2.10) describes a program from two perspectives: a quantitative and a functional. The quantitative dimension describes elements in the code and their relationship, from simplest instructions (atoms) to blocks, related blocks, and finally, the macro-structure of the program. The functional dimension has an orthogonal standpoint, reminiscent of the Structure Behavior Function (SBF) model [371], where programs are broken into a text surface, the static program code; the program execution, a dynamic entity; and a function/purpose dimension, where the program...
act as an artifact with an extrinsic purpose. Sanders et al. have used the Block Model \[308\] to classify programming tasks, and has shown to be more accurate than Bloom’s \[31\] and SOLO \[29\] taxonomies to categorize programs \[386\]. Schulte et al. \[317\] compared and contrasted some of the previous program comprehension models with the Block Model, providing some insight into program comprehension to CS educators.

Table 2.10. The Block Model \[316\].

<table>
<thead>
<tr>
<th>Macrostructure</th>
<th>Understanding the overall structure of the program text</th>
<th>Understanding the algorithm underlying a program</th>
<th>Understanding the goal/ purpose of the program (in the context at hand)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relationships</td>
<td>Relations and references between blocks (e.g. method calls)</td>
<td>Sequence of method calls, object sequence diagrams.</td>
<td>Understanding how subgoals are related to goals, how a function is achieved by subfunctions</td>
</tr>
<tr>
<td>Blocks</td>
<td>Regions of Interest (ROI) that syntactically or semantically build a unit</td>
<td>Operations of a block, a method, or a ROI (chunk from a set of statements)</td>
<td>Understanding the function of a block, seen as a subgoal.</td>
</tr>
<tr>
<td>Atoms</td>
<td>Language elements</td>
<td>Operation of a statement</td>
<td>The function of a statement: its purpose can only be understood in a context</td>
</tr>
<tr>
<td>Text Surface</td>
<td>Program Execution</td>
<td>Function / Purpose</td>
<td></td>
</tr>
<tr>
<td>Architecture / Structure Dimensions</td>
<td>Relevance Dimension</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.5.2 Program Comprehension Activities

Novice understanding of programs has been explored from a variety of perspectives, among them: interpreting students’ ways of classifying code fragments based on perceived similarities and differences \[36\]; categorizing novices’ mental models of the notional machine underlying imperative \[33\] and recursive \[307\] computations, as well as comparing students’ mastery of recursion vs. iteration \[237\]; assessing the understanding of conditionals, loops and nested loops \[155, 48\]; analyzing the relations between students’ performance and their annotations in the exam papers \[223\], and finally comparing block versus textual representations of programs \[385\].

Such understandings of program comprehension originated a series of activities that emphasized program comprehension. Next, we present a few examples of how CER explored such activities in previous works.
**Tracing Tasks**

We define tracing as an activity where learners follow the execution of a program. With little expertise, learners read the program element by element, line by line, following the control flow. With growing expertise, learners can construct more robust mental models chunking elements of the code that represent higher-level strategies, or plans, until they can comprehend the whole program. Learners can use supporting materials to keep track of the program state.

Hazzan [143] describes some variations of tracing tasks. Learners can follow the code execution according to a given input; choose a given input and then follow code execution; mentally execute code using a series of supplied inputs, which scaffold the comprehension process of the code; find different sets of inputs that represent a different flow by which the code is executed, and find a set of inputs that yields a specific output.

While initial studies showed that learners might have issues to comprehend code [195], recent studies presented some evidence that introducing tracing tasks could improve syntactic [178] and semantic [179] aspects of writing code. Learners value tracing activities, and such activities can improve their grades and reduced drop out and failure from programming courses [144]. Moreover, tracing activities can be used as meaningful learning events [352]. When tracing, learners who use supportive materials, such as sketches, perform better than those who do not [71], and clear strategies to support tracing and sketching provide even better results [396]. Evidence shows that tracing is highly correlated with code writing skills [381, 201].

**Summarizing Tasks**

Similar to tracing tasks, summarizing tasks ask learners to understand a piece of code and follow program execution. However, in summarizing tasks, learners should achieve a higher level of abstraction, being able to move from element to element tracing strategies to comprehension of the program as a whole. Some authors described such abstraction level in terms of the SOLO taxonomy relational level [197, 387], or “explain in plain English” [250, 62, 201]. Teague et al. [362] use a different approach, between summarizing in short sentences and tracing (sometimes referred to as “concrete tracing”), to define ‘abstract tracing’ as reading the code without relying on concrete inputs.

While such approaches differ in terms of outcomes, they induce the
same learning process where learners need to abstract from the line-by-line execution and “be able to see the forest, not the trees” [197]. While proficiency in programming can be associated with the ability to summarize code [248], it remains challenging to create rubrics to categorize learners’ summaries, and some previous studies show that only 30% of learners can summarize code [387].

**Interacting with Existing Code: Parsons Problems**

In the previous activities, a completely functional program was provided to learners so they could trace or summarize it. In Parsons problems [271], the task is in between reading and writing activities [78]. Learners are provided the correct code, but with an incorrect ordering of its lines of code. Parsons problems free learners from creating a program solution to a given problem, but still require a deep understanding of program behavior and its goal to reorder the code correctly.

Parsons problems can be an efficient method to teach program comprehension [91]. Previous works used Parsons problems showed it could be more efficient and equally effective as writing code [91] and can have their difficulty easily adapted [92]. Adaptative Parsons problems [90, 89], which change characteristics of the task based on learner performance, have shown to have superior performance to regular Parsons problems in some contexts. Providing subgoal labels in Parsons problems could also improve learners’ performance [242]. Ericson at al. [92] found that solving two-dimensional Parsons problems with distractors took significantly less time than fixing code with errors or than writing the equivalent code. Denny et al. [78] noted a direct correlation between Parsons problems and code writing scores. For a more comprehensive review of Parsons problems in CER, see Du et al. [83].

### 2.5.3 Towards a Comprehension First Pedagogy in CER

Given the potential benefits of program comprehension activities and a hierarchy of skills where program comprehension precedes writing code, some recent pedagogical approaches follow an instructional design that emphasizes comprehension-fist strategies.

Recently, some pedagogical approaches introduced frameworks presenting a sequence of skills for programming instruction. Predict-Run-Investigate-Modify-Make (PRIMM) [319] and Use-Modify-Create (UMC) [188] share similar pedagogical approaches. Learners first need to observe a con-
crete program from others and comprehend its behavior (the Use in UMC, Predict-Run-Investigate in PRIMM), modify its contents to alter an existing functionality or implement a new one (M on both approaches) and finally create their own programs (M on PRIMM and C on UMC). Experience reports with teachers’ usage of PRIMM show that teachers value the collaborative approach of PRIMM, the structure given to lessons, and the way that resources can be differentiated\[320\]. In CS POGIL (Process Oriented Guided Inquiry Learning) [183], groups of students construct their understanding about code through critical thinking questions that include reading, analyzing, adjusting code, and finally reflecting on what has been learned.

PLTutor [256] uses online learning materials and animations to introduce program comprehension and semantics before learners engage in writing programs. PLTutor explicitly presents program execution to learners at a very detailed level, close to the language interpreter, where learners are exposed to a Notional Machine through observation of different aspects of the programming language during program execution.

Xie et al. [395] designed guidelines for instructional design supported by previous CER works and theories for novice instruction. Their theoretical framework distinguishes between four learning steps focused on different programming skills. First, learners should gain knowledge of the operational semantics, demonstrated by being able to trace code. Second, learners gain knowledge of the syntactic structures of the programming language, demonstrated correctly by translating instructions in natural language to code. Third, learners are introduced to reusable abstractions, or program templates, demonstrated by the ability to identify the components of such abstractions as well as their purpose in a given program. Finally, learners use problem-solving skills, demonstrated by being able to apply or combine program templates to solve a problem. Their initial results show an increase in completion rate on practice exercises, decreased error rates, and improved understanding of programming concepts.
Some authors claim that programming is inherently complex, therefore making it very difficult for students in general to master [224, 195]. In contrast, others argue that while programming tasks do have several interacting elements, the unrealistic expectations, and poorly designed assessment practices and instruction might be more relevant factors than the inherent complexity of programming itself [203].

Perhaps, both points of view can be valid. Since educational psychology defines that complex learning “involves integrating knowledge, skills, and attitudes, coordinating qualitatively different constituent skills” [374], often focusing on authentic real-life tasks, facilitating the transfer of what is learned to new problem situations, programming can, in fact, be considered a complex skill. Programming instruction can also benefit from mature educational psychology frameworks, particularly those aimed explicitly at complex learning such as the Four Components/Instructional Design (4C/ID) [376] and its instructional blueprint, the Ten Steps to Complex Learning (Ten Steps) [374]. The Ten Steps and the 4C/ID were successfully adapted, with varying degrees of conformity, to many domains such as Information and Communication Technology (ICT) [310], health professionals, [354], electrical circuits [231], app development [211], law [252] and physics [230].

To facilitate the transfer of learning to new contexts, the Ten Steps use a holistic approach to instruction, as opposed to an atomistic one. Merriënboer and Kirschner [374] describe atomistic design approaches, where a complex task is broken into smaller and less complex parts until it can be reduced to the presentation of facts and practice of skills, usually practiced in isolation. Such an instructional approach can be acceptable as
long as the element interactivity of those parts is low. However, when a task has closely interrelated parts, the parts cannot be presented and practiced in isolation, since the relationship between its parts is fundamental for mastering the task.

Conversely, Merriënboer and Kirschner [374] argue that a holistic approach attempts to deal with complexity without losing sight of the interrelationship of the elements being taught, exposing learners to all constituent skills of a whole authentic task since the beginning of instruction. A holistic design approach can mitigate three significant problems described in the educational literature: compartmentalization, fragmentation, and the transfer paradox.

According to Merriënboer and Kirschner [374], a holistic approach aims to reduce compartmentalization, where instruction focuses on only one domain of learning (cognitive, affective, or psychomotor, for example), by integrating declarative learning (conceptual knowledge), procedural learning (skills) and affective learning (disposition). A holistic approach can reduce fragmentation, where tasks are broken into small, incomplete parts, reduced to learning objectives, which are incrementally being added to instruction and practiced in isolation, by presenting whole authentic tasks to learners from the onset. Holistic approaches deal with the transfer paradox, where the repeated practice of each learning objective in isolation is very efficient (but leads to less transfer at the end of instruction) by adopting an interleaved practice of tasks, aiming for maximum variability.

To achieve such goals, the Ten Steps defines a blueprint for instructional design, as presented in Table 2.11, by using the main components of the 4C/ID illustrated in Figure 2.8 and adding clear steps to design for complex learning. In the next subsections, we summarize how to approach each one of those steps described in Ten Steps for Complexes Learning handbook [374].
Table 2.11. The Ten Steps of Complex Learning blueprint and the 4C/ID components.

<table>
<thead>
<tr>
<th>Blueprint components of the 4C/ID</th>
<th>Ten Steps to Complex Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning Tasks</td>
<td>1. Design Learning Tasks</td>
</tr>
<tr>
<td></td>
<td>2. Design Performance Assessments</td>
</tr>
<tr>
<td></td>
<td>3. Sequence Learning Tasks</td>
</tr>
<tr>
<td>Supportive Information</td>
<td>4. Design Supportive Information</td>
</tr>
<tr>
<td></td>
<td>5. Analyze Cognitive Strategies</td>
</tr>
<tr>
<td></td>
<td>6. Analyze Mental Models</td>
</tr>
<tr>
<td>Procedural Information</td>
<td>7. Design Procedural Information</td>
</tr>
<tr>
<td></td>
<td>8. Analyze Cognitive Rules</td>
</tr>
<tr>
<td></td>
<td>9. Analyze Prerequisite Information</td>
</tr>
<tr>
<td>Part-Task Practice</td>
<td>10. Design Part-Task Practice</td>
</tr>
</tbody>
</table>

Figure 2.8. The Four Components/Instructional Design Framework (4C/ID) [375].

2.6.1 Design Learning Tasks

Learning Tasks (circles in Figure 2.8) are authentic, whole tasks, often inspired by professional tasks or situations. According to Merriënboer and Kirschner [374], since learning tasks aim to integrate knowledge, skills, and attitudes, those tasks are ill-structured and cannot be performed using simple rules or procedures. Learning tasks require from learners knowledge of the domain (mental models) and knowledge about how to approach
problems in the domain (cognitive strategies) or heuristics [374]. Learning tasks use a process of inductive learning from concrete experiences and skills, but always presenting “the big picture” to learners.

What constitutes an authentic task is highly context-dependent. Depending on learners’ prior knowledge and goals, an authentic task can be a case study, a project to be executed in groups or individually, or even professional tasks. When the task itself is too complex, particularly when introduced to beginners, parts of such task can be used as representative of an authentic task, as long as it still integrates knowledge, skills, and attitudes in more than a single constituent aspect of the task.

Ideally, learners should operate close to their Zone of Proximal Development. To allow beginners to perform authentic tasks, instruction should embed support and guidance to learners (semi-filled areas inside the learning task circles in Figure 2.8). Van Merriënboer and Kirschner [374] argue that support “focuses on providing learners with the assistance with the task elements involved in the training, the steps in a solution that get them from the gives to the goals (product-oriented)”. On the other hand, guidance “focuses on providing learners with assistance with the process inherent to finding a solution (process-oriented)”. Both support and guidance should fade during task execution, using a process of scaffolding. When presenting a task to learners with little prior-knowledge, instructors could make use of worked examples, case studies, and other strategies that provide extensive support to learners. With increasing expertise, the level of support should decrease, keeping the task at the same difficulty level, using completion tasks, for example. Finally, with enough expertise, learners should be able to perform a task without support, using, for example, full problem-solving strategies.

To facilitate transfer, learning tasks should provide maximum variability (black triangles in each learning task in Figure 2.8). Each learning task should differ in terms of context, surface, and structural features, helping learners to abstract from the details of the task and form more generic mental models.

2.6.2 Design Performance Assessment

Complex tasks can be decomposed in many different aspects, with the complex skill at the top of the hierarchy. Each one of those aspects can be characterized in a particular learning domain, which requires a different learning strategy. Merriënboer and Kirschner [374] argue that
the aspects of the hierarchy can be further categorized as non-routine, routine, or automated. Merriënboer and Kirschner (ibid.) describe non-routine aspects of the task as aspects using problem-solving strategies or reasoning and require presentation of supportive information before and during task performance. Routine aspects of the task are consistent across learning tasks and require procedural information, rules, or manuals to be performed. Finally, tasks needing to be automated are routine tasks that require high levels of automation or be fully automated, requiring procedural information and additional practice.

According to Merriënboer and Kirschner [374], having established the hierarchy of constituent aspects of the task and their respective categories, the instructor could define performance objectives for all learning aspects. Performance objectives should clearly state what learners should be able to perform after the training, under which conditions the task is performed, and which tools and materials are used in the task. The objectives should assert standards for acceptable performance, including criteria for speed, accuracy, rules applicable to the task at hand, and attitudes expected from the learner. Those standards could be formalized as rubrics or other assessment benchmarks.

### 2.6.3 Sequence Learning Tasks

To provide efficient instruction, instructors should create a distinctive range of learning tasks and order them according to their complexity, from less to more complex tasks. Here, the Ten Steps specifically distinguish tasks by their complexity, not difficulty. Difficulty refers to the interaction of learners’ prior-knowledge and support with the task’s complexity, moderated by learners’ attitudes and dispositions. Therefore, complexity must be defined analytically and independently of the learner.

For each learning task, it is possible to define a task class (dotted rectangles in Figure 2.8) where each learning task has similar complexities and can be performed with the same body of knowledge [374]. For each task class, learners initially receive high levels of support that fades over the next learning tasks in sequence. Only after learners can perform a task, according to the desired standards, with no support, they can move to the next task class.

To provide a holistic approach to learners using authentic tasks, even the most straightforward learning task must be representative of the authentic task in all its aspects [374]. Since beginners have fewer mental models that
would allow them to perform in a completely authentic task, instructors could use a “simplified reality” approach to sequence the learning tasks. For example, Merriënboer and Kirschner [374] describe a “simplifying conditions method”, presenting a few steps that can be used to sequence learning tasks:

1. Identify the different conditions under which a task might be performed;

2. Identify the conditions that affect the complexity of the task;

3. Provide concrete instances to the conditions that affect the complexity of the task;

4. Define the first task class with the simplest conditions and the last task class with the most complex condition;

5. Add task classes in between, incrementally increasing complexity.

### 2.6.4 Design Supportive Information

The Ten Steps use an elaboration learning process, i.e., a process of enhancement of current mental models using activities or information that specifies the relationship between learners’ prior-knowledge and the task to be performed, to present supportive information. Supportive information (L shaped bars in Figure 2.8) should be presented before learners engage and through the task class activities. The supportive information will help them perform non-recurrent aspects of the task, helping learners to create the necessary schemata so and perform adequately in the task.

The non-recurrent aspects of a task can be presented using cognitive, problem-solving, and reasoning strategies. Such activities can be presented to learners using cognitive feedback strategies, where learners critically compare their own mental models and strategies with mental models of experts and peers. Instructors could also present conceptual models to demonstrate how elements are defined in a domain, structural models to show how elements are constructed in a domain, or causal models to present the rules of behavior in a domain to learners. Finally, instructors could introduce Systematic Approaches to Problem-solving (SAPs), which describe how a proficient performer would organize his/her actions in
a domain. In general, SAPs describe the heuristic and steps executed towards a given solution.

### 2.6.5 Analyze Cognitive Strategies

Usually, textbooks, manuals, or documentation pertinent to the domain describe cognitive strategies and systematic approaches to problem-solving (SAPs). However, if no such materials are available, the instructor can extract and define the cognitive strategies by investigating actions performed by experts. For example, interviews can be used to define specific phases or steps to solve a given problem, or the heuristics habitually used by experts to approach the task.

### 2.6.6 Analyze Mental Models

Domain models represent how knowledge is organized in a given domain. Domain models are usually outlined by theories in the domain and can be presented, graphically, or in other formats as conceptual, structural, or causal model representations.

### 2.6.7 Design Procedural Information

Procedural information should support learners in routine aspects of the task (represented as arrows in Figure 2.8). Those aspects are consistent across all tasks in a task class and should support schema automation. The information included in procedural information is strongly connected to each learning task. Therefore, it should be presented only when necessary, preferably just-in-time, and fading over instruction. Procedural information resorts to cognitive rules an algorithmic materials that are better represented by how-to instructions. By presenting procedural information, the Ten Steps use a process of knowledge compilation to support the automation of routine aspects of the task.

The most common forms of procedural information are presented by how-to instructions, quick reference guides or manuals, step-by-step procedures, or algorithmic flow charts. Corrective feedback can also be used as procedural information since it helps learners recover from an error by pointing to the specific procedure that generated the error and provides hints on how to fix such an error.
2.6.8 Analyze Cognitive Rules

Just-in-time information is usually documented as quick reference guides, manuals, or other methods of documentation. If no such information is available, the instructor can refer to experts in order to document their algorithmic knowledge when solving routine aspects of the task.

2.6.9 Analyze Prerequisite Knowledge

Prerequisite knowledge refers to the information necessary for learners to perform routine aspects of the task. This sort of knowledge should not be confused with theory or non-routine aspects of the task that are usually presented before and during task performance. Instructors should analyze which kind of concepts, information, or skills are not familiar to students at the moment of task execution and present them just-in-time to learners.

2.6.10 Design Part-task practice

Part-task practice is only necessary for critical routine aspects that need to be performed at a very high level of automaticity. The goal of part-task practice is to enforce schema automation using “drill and kill” practices, employing a process of strengthening. This part-task practice should not be presented in isolation of the actual learning task but presented when high-levels of automation are required during task execution.
In this chapter we present the results that supports answering our main research question:

**RQ 1:** *How to operationalize an a priori evaluation of the cognitive complexity of comprehending programs, and what are the implications of a model of cognitive complexity on the instructional design of introductory programming courses?*

To answer research **RQ 1**, we explored four sub research questions:

1. **RQ 1.1:** How to design representations of Rules of Program Behavior?
2. **RQ 1.2:** How to define and evaluate the cognitive complexity of comprehending concrete programs?
3. **RQ 1.3:** How to evaluate students’ prior-knowledge of program comprehension?
4. **RQ 1.4:** What kind of activities can foster program comprehension and how such activities can be organized?

Next, we summarize how the results presented in the publications of this thesis presented evidence to support each one of our research questions.

### 3.1 RQ 1.1: How to Design Representations of Rules of Program Behavior?

The previous definitions of Notional Machines (NM) presented in Section 3.1 do not adequately capture some essential aspects of a system behavior necessary to the evaluation of programs’ cognitive complexity. Therefore,
to answer RQ 1.1, we defined a new pedagogical instrument, the Rules of Program Behavior (RPB), in Publication I, inspired by NM works. RPBs aim to define more clearly contextual and representational aspects of the program behavior communication.

Figure 3.1 illustrates the learning process of a complex system behavior using a representation of the RPB. Instructors make use of their experience, intuition, pedagogical content knowledge (PCK) [238], and knowledge of the system to create a concrete instrument. Such an instrument can be adopted by other instructors or serve as inspiration for the development of learning materials such as visualizations or oral explanations. Depending on the vocabulary set by the instrument, learners may learn directly from the instrument itself.

In the setting presented by Figure 3.1, our goal is twofold: First, to present a simplification of a system to a particular audience. RPBs can be used to communicate rules to learners, but mainly to communicate intention and goals to instructors; Second, RPBs can be used to set expectations of learners’ mental models. By inspecting the RPBs of previous instructors, teachers might be able to extrapolate how much learners know and how their comprehension of system behavior evolved.

**Figure 3.1.** Actors and relationships during instruction of system’s rules.

For this conceptualization of the RPB (in red on Figure 3.1), we adapted previous NM definitions to account for such goals and settings. We define
the Rules of Program Behavior as:

*The Rules of Program Behavior is an explicit, concrete instrument *intentionally designed* to serve the **pedagogical purpose** of representing and explaining the **behavior** of a computational system. The Rules of Program Behavior uses **terminology and abstraction levels** aimed at a **particular audience** to support their practices in a particular context. It is often a simplification and can be communicated in different **formats**. The Rules of Program Behavior are presented in **stages** of increasing completeness.*

The main goal of an RPB is still to present the rules of a system. RPB definition clarifies what makes a representation of behavior (or semantics) an RPB or not. Is a metaphor alone an RPB? Is a simple oral explanation an RPB? If we observe only a perspective of explaining the behavior of a system, metaphors and explanations accomplish such a goal. However, they do not provide context rationale for instructors. Therefore, we do not consider such instruments RPB.

An RPB instrument should serve the **pedagogical purpose** of communicating a complex system to an audience, and RPBs should be intentionally designed for this goal. The context and the goal of an RPB instrument dictate the **granularity of the abstraction and detail** and later influence the presentation format in choosing an appropriate **vocabulary** for such an audience. Finally, the RPB could have distinct **presentation formats**. It can be presented textually (e.g., formal semantics, textual representations), graphically (e.g., code visualization), or even orally using a variety of tools: examples, metaphors (hopefully, good ones), images, physical experiences, among others.
Results

**Figure 3.2.** The design factors of the Rules of Program Behavior.

In an RPB, the semantics are tied to a specific system. The completeness of the semantics will define how much of this particular system will be covered by the RPB. Semantics soundness reflects if every part or step in representing a rule is the ground-truth [367]. An RPB should have a design goal to be as sound as possible to avoid introducing misconceptions.

Intimately connected to learners’ prior knowledge, the vocabulary used to describe an RPB should serve as a contextual element to activate specific mental models. Different stages of cognitive development, age, social and cultural aspects, and educational level can affect how to communicate an RPB. For example, a lecturer could design an RPB to represent a single aspect of Python (e.g., assignment) to three distinct audiences: elementary school, high school, and university level introductory programming students (CS1). At the elementary level, simple analogies that connect with physical experiences using terminology and phenomena common to that particular age group (e.g., “labels”, “values”, “store”) could suffice to convey the rules of the assignment construct.

High school students, on the other hand, might already possess a rudimentary knowledge of how a computer works and its inner components such as RAM, processor, and Input/Output devices. Students might even have an embryonic mental model for data storage, e.g., “the computer stores data in a digital format in a computer’s memory”. This mental model allows the instructor to use descriptions of the assignment construct
to connect with this piece of prior knowledge, e.g., “variables are sections in a computer’s memory that store a single value and are manipulated using an identifier to this value, the variable name. Assigning a new value to variables changes the data stored in it.” To a more specialized audience, such as (CS1) students, vocabulary can be closer to textbook materials.

A particular set of semantic rules (e.g., Python semantics), aimed at a particular audience (e.g., CS1 students), can have entirely different designs of RPB by changing its goals, and manipulating the abstraction level and detail of the RPB. For example, the piece of code on Listing 3.1 could have distinct RPB rules:

**Listing 3.1. “Loop over a range of numbers in Python”**

```python
for x in range(10):
    print(x)
```

1. “It prints a sequence of numbers from 0 to 9.”

2. “The variable x gets values from 0 to 9, and at each step, it prints its value.”

3. “The variable x is initialized with the first value (0) in the sequence of numbers defined by the range function. At each step, the program prints the current value stored in x and increments it to the next integer value until it reaches the upper limit of the sequence, ending one element shorter of the value specified by range (9), printing ten values in total.”

4. “The range function creates a list of numbers from the lower bound (0) to the upper bound (10) minus 1, with ten elements in total. At each step of the repetition, the variable x is assigned to the current element in the list. At the end of each step in the repetition, the current element in the list is updated to the next one in the sequence.”

5. “The range function creates an iterator where a formula calculates each value of the sequence. At each step of the repetition, the iterator gets the next value in the sequence (from the lower bound to the upper bound minus one) and assigns it to the variable x. The variable x, containing the current value in the sequence, is then printed at each step of the
6. “The range function, whose formal syntax is class *range*(start, stop[, step]), represents an immutable sequence of numbers. The arguments of the range constructor are integer numbers (positive or negative). If the *step* argument is omitted, it defaults to 1. If the *start* argument is omitted, it defaults to 0. For a positive step, the contents of a range *r* are determined by the formula *r*[i] = *start* + *step* *i* where *i* ≥ 0 and *r*[i] < *stop*. For a negative step, the contents of the range are still determined by the formula *r*[i] = *start* + *step* *i*, but the constraints are *i* ≥ 0 and *r*[i] > *stop*. Each value of the sequence is determined in a new step of the repetition. The new value is then assigned to the variable *x*, which is printed.”

While the previously presented six RPB rules are valid representations of the language constructs (as presented on Listing 3.1) and Python semantics, some are more suitable than others for a specific goal. Eventually, instructors may have to choose different levels of semantics soundness and balance them with the goal levels of detail and audience vocabulary. For example, RPB rule (5) uses fewer concepts to present the language constructs and uses a less demanding vocabulary, compared to RPB rule (6). However, RPB rule (5) is sound for Python 2.x, while RPB rule (6) is sound for Python 3.x.

The level of detail described by each one of those rules might have different mental model representations. While it is not possible to precisely predict which kind of representation students will possess during instruction, it is possible to set the expectations of the number of concepts and the relationships between those concepts, as described by a rule. For example, Figure 3.3 presents possible representations of mental models of rules 1, 2, 3 and 6.
In this novel conceptualization, the RPB is entirely independent of its presentation. More frequently, program visualizations or animations embeds NM implicitly. There are not many concrete textual representations of NM. While formal representations of programming languages semantics (e.g., [279]) could be an elegant, concise, complete, and sound NM, they use a very particular notation and jargon in its representation.

Here we present a case study exploring the design process of a tentative textual representation of a Python RPB. To create an explicit, concrete, textual RPB, we first convey the intention and process embedded in its design:

- The RPB follows a formal semantics description inspired by formal semantics of Python (e.g., [279]). The design process follows, as much as possible, the formal definition of the language semantics, from the simplest RPB that represents the simplest element in the language, and later introducing new elements to increase its completeness, much like the semantics introduce new production rules.

- The RPB stipulates an audience, CS1 students, and a goal for such an audience, proficiency in Python. Striving for representation simplicity,
the RPB cover literals, variables, expressions, and assignment constructs. We also introduce some simplifications in this RPB regarding types, values, and other characteristics of the language.

- The concrete textual format should communicate the RPB explicitly to the audience. There is an open question if using explicit textual representations would be a more efficient or effective way to communicate an RPB, compared to visualizations, for example. In the design of this RPB, we aimed to use phrasal descriptions of elements and actions over these elements where possible.

Figure 3.4 shows the rules present on each iteration of the RPB (in columns). Each subsequent iteration of an RPB carries all rules from the previous RPBs, added the necessary rules to represent new behaviors. Such new behaviors will allow learners to comprehend increasingly complex programs, as exemplified at the bottom of Figure 3.4.
Programs can store and manipulate integer values.

Variables are labels for sections of the program’s memory. Variables can hold only one value at a time.

A given identifier can be associated with only one variable. Variable identifiers should begin with a letter or an underscore, followed by any number of letters, numbers, or underscores. Some names cannot be used as identifiers since they are reserved for language functionalities.

The assignment statement is the mechanism by which variables can have their value changed. We use the equal (=) operator to tell the program that a variable on the left of the operator will have its value changed to the value on the right side of the operator. The assignment statement first evaluates its right-hand side and then assigns this value to the variable indicated in the left-hand side.

A variable is created when we assign a value to it, as long as its name is valid.

A Python program consists of zero or more lines of code. Each line constitutes a statement. If the statement is valid, it will instruct the program to execute an action. Next, the program looks to the next instruction in the next line until no more lines are left to check. Statements are executed from top to bottom.

An expression is a section of the program that can be evaluated to a value.

Example Program

<table>
<thead>
<tr>
<th>a = 2</th>
<th>a = 2</th>
<th>a = 2</th>
<th>a = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>b = 3</td>
<td>b = a</td>
<td>b = a</td>
<td>b = a + 1</td>
</tr>
</tbody>
</table>
3.2 RQ 1.2: How to define and evaluate the cognitive complexity of comprehending concrete programs?

To answer RQ 1.2 we developed the Cognitive Complexity of Computer Programs (CCCP) framework, presented in Publication II. The CCCP is a theoretical model for reasoning about the complexity of computer programs and generating metrics that summarize aspects of complexity and difficulty.

The primary unit of analysis of the CCCP are plans (see Section 2.2.2), concrete realizations of schemata in a program. Since plans are cognitive representations of knowledge and strategies, we consider them cognitive actions, therefore subject to the same rules of the Model of Hierarchical Complexity (MHC) (see Section 2.3.2). We examine not only abstract, language-agnostic plans but also the lower-level plans closer to the programming language structures.

We define the cognitive complexity of a program in terms of the hierarchical structure of plans in the program. The CCCP is concerned with two aspects of complexity: First, the complexity level of each plan in the hierarchy; Second, any interactions between plans that demand simultaneous processing of the plans in Working Memory (WM), thus contributing to higher intrinsic load. Correspondingly, the CCCP can be used for two types of analysis, which we term hierarchical analysis and interactivity analysis, respectively. A CCCP analysis of a program starts with the concrete code and produces a plan hierarchy as well as the associated metrics of plan depth and maximal plan interactivity.

3.2.1 Hierarchical Analysis

A hierarchical analysis of a program produces a description of the program as a tree of plans. Each plan appears in the tree at a particular level of complexity, with the primary plans at the lowest level (zero) and more complex plans at higher levels. The plan that corresponds to the whole program is at the top (root) of the hierarchy, matching the highest order of complexity in the analysis.

The level of a plan is known as its plan depth (PD), our programming-domain equivalent of what is termed as action’s “order” by the MHC. Programs can be compared in terms of their plan depth, which also provides a comprehensive examination of the plans involved in comprehending the program.
The hierarchical analysis is performed in bottom-up order. We start from the simplest elements in the program, closer to the primitive operations of a notional machine (NM) [84]. While traditionally NMs do not provide precise specifications of the elements used in the program, a Rule of Program Behavior (RPB) can better provide a detailed account of the relationship and number of concepts in a program (see Publication I and Figure 3.4). We start with system capabilities for which we could identify no coordination of other plans without introducing low-level concepts outside of the target RPB implied by the program. To build the upper levels of the hierarchy, we considered how the plans depend on each other using coordination or prerequisite rules.

**Case Study: Summing Program**

We analyze a program that sums a fixed sequence of numbers using an explicit loop control variable, as shown by the code Figure 3.5. The Hierarchical tree, a product of the hierarchical analysis, is shown in Figure 3.6.

**Figure 3.5.** JavaScript code snippet of a summing program.

```javascript
int i, input, sum;
sum = 0;
for (i = 1; i <= 10; i++) {
    read(input);
    sum = sum + input;
}
```

The summing program features four primary-plans (P1–P4) at the lowest levels:

- The *define literals* plan (P1) represents the use of numerical data in the program.
- P1 is a prerequisite for the *declare variable* plan (P2), an abstraction of the linking of names to storage in computer memory.
- The *arithmetic operator* plan (P3) signifies the use of arithmetic operations such as addition.
- The *jump to code* plan (P4) represents the unconditional transition of control to a different part of the code, as at the end of the loop.
Next, the subsequent plans coordinate the previous primary-plans or establish prerequisites rules. As we move towards the top of the hierarchy, more complex plans coordinate the plans previously defined in the lower levels of the hierarchy:

- The initialize variable plan (P5) represents the assignment of a literal value to a variable. It organizes two lower-level plans (P1 and P2) using a coordination rule, therefore $PD(P5) = \max(PD(P1), PD(P2)) + 1 = \max(0, 0) + 1 = 1$.

- The evaluate an expression plan (P6) organizes P2 and P3, so expressions are evaluated over variables. Thus, $PD(P6) = \max(PD(P2), PD(P3)) + 1 = 1$.

- The accumulate in a variable (P8) plan coordinates the assignment of a literal to a variable (P5) and the evaluation of the expression to be assigned (P6). Therefore, $PD(P8) = \max(PD(P5), PD(P6)) + 1 = 2$.

- The test for termination plan (P9) is a selection plan that coordinates P6
Results

and P4, evaluating an expression and branching to the appropriate part of the code. \( PD(P9) = \max(PD(P4), PD(P6)) + 1 = 2 \).

- The read input plan (P7) represents a library function call. To apply P7, it is only necessary to comprehend the purpose of the function and consider its input, a variable (P2); P7 coordinates P2 and the return value that is assigned to a variable with P5. (While the assignment of the return value and an initialization plan have different goals, they coordinate the same plans and have the same plan structure.) Thus, \( PD(P7) = \max(PD(P5), PD(P2)) + 1 = 2 \).

- Loop over a range (P10) coordinates the initialization (P5), test for termination (P9) and increment (P8) plans in a non-arbitrary order defined by the desired control flow. Thus, \( PD(P10) = \max(PD(P5), PD(P8), PD(P9)) + 1 = 3 \).

- Finally, the entire summing program (P11) coordinates the loop (P10), input (P7), and accumulation (P8). \( PD(P11) = \max(PD(P10), PD(P7), PD(P8)) + 1 = 4 \).

3.2.2 Interactivity Analysis

Hierarchical analysis (HA) and its Plan Depth (PD) metric describe the complexity of a program. It suggests how learning to understand a particular aspect of the program (plan) is predicated on first learning to understand other aspects. HA does not, however, attend to simultaneous processing in working memory, which is central to element interactivity and cognitive load (see Section 2.3.4). For this purpose, the CCCP extends HA with an Interactivity Analysis (IA).

Whereas HA was concerned with complexity alone, IA deals with difficulty. We define **difficulty as the interaction between complexity and learners’ prior-knowledge; moderated by learners’ disposition.** Complexity analysis assumes an ideal learner with no prior knowledge in the domain. Such an assumption is, in most cases, unrealistic, since most learners would not be able to cope with such a large number of interacting plans (as shown in Figure 3.6). Disposition dictates how many cognitive resources are allocated to deal with the task. To be able to measure the observable facet of complexity, difficulty, we must consider
the programmer's prior knowledge and the chunking of multiple subplans into larger plans, which are processed as a single element.

As a starting point for IA, we take the program code and the HA tree and state our assumptions about the prior knowledge of the programmer (e.g., using Publication III instrument): we evaluate which plans we expect the learner will be able to deal with as single elements, i.e., “collapsed” through chunking in the HA tree. Next, we estimate which plans must be kept in mind simultaneously as the programmer mentally manipulates the (higher-level) plans of the program.

We build on the concept of focus of attention (FoA) [69, 263], which we have adapted to a programming context. At any given time, the FoA is a single plan that has been activated in WM for immediate processing (see Figure 3.7, (a)). The plan in the FoA is linked to a subset of other plans in the program that need to be considered simultaneously (Figure 3.7, (b)); these other plans form a region of direct access (RDA) [263] that must also be stored in WM. As a programmer processes a plan, their FoA will shift from one plan to another, and rearrange the RDA in the WM as required. When a given plan is processed, its result is stored in the RDA for later manipulation, no longer as multiple interacting plans but as a single value (Figure 3.7, (c)). When all plans are processed, the final output of the interacting plans is kept in the RDA for immediate access when needed (Figure 3.7, (d)). We define such a pattern of processing interacting plans as switch-process-store-output (SPSO) pattern.
To conduct IA, we examine the control and data flow of the program. We trace the execution of the program step by step, considering how the program flows and the FoA shift from one plan to another. At each FoA shift, we compute a Plan Interactivity (PI) metric that equals the number of plans inside the RDA; PI is essentially a programming-domain estimate of element interactivity as defined by CLT [23]. Maximal plan interactivity (MPI) is the highest PI value at any FoA in the program.

Which plans fall within the RDA of a particular FoA depend on the way plans are merged, sequenced, and nested. We consider a plan A to be in the RDA of another plan B if A is directly nested within B or vice versa, or if the execution of A interleaves with that of B. If A’s execution is done before B’s starts, the plans can be considered non-simultaneously and are not in each other’s RDA.

**Case Study: Averaging Rainfall**

In this case study, we consider distinct implementations of the Rainfall Problem [335] using different plan-composition strategies: a merged-plans solution (Figure 3.8) and a sequenced-plans solution (Figure 3.9), both adapted from Sorva and Vihavainen [341]. Each color corresponds to one of the main high-level plans identified in previous Rainfall works [101]:

---

**Figure 3.7.** Focus of Attention switch, process and output pattern.
iteration, a sum, read, average, count, guard against negative, guard against division by zero, and sentinel. We assume the programmer can deal with each highlighted plan as a single chunk.

**Figure 3.8.** A color-coded merged-plans solution for Rainfall.

```scala
var count = 0
var sum = 0
var average = 0
while (true) {
  val input = readInt()
  if (input >= 0) {
    if (input >= 999999) {
      if (count == 0) { println("No data!"); break }
      println(average);
      break
    }
    count += 1
    sum += input
    average = sum / count
  }
}
```

Analyzing the merged-plans program in Figure 3.8, we observe that all plans share the same control flow through the loop plan (using the while (true) loop) and some plans share the same data flow. For instance, the variable input is shared among the input, negative, sentinel, and sum plans, while count is shared by the count and guard plans. This interleaving of control and data flow forces the activation of all these subplans in working memory for every FoA shift. In order to be able to process the program and extract its meaning (or write it), the programmer needs to evaluate the impact of each plan on the data and control flow. Therefore, for the merged-plans program in Figure 3.8, all plans must interact in the RDA, yielding an MPI of 8.

**Figure 3.9.** A sequenced-plans solution for Rainfall.

```scala
def isNotSentinel (input : Int) = input < 999999
def isValid (input : Int) = input >= 0
def average (numbers : List[Int]) =
  if (numbers.nonEmpty)
    numbers.sum / numbers.size
  else 0
val validInputs = inputs.takeWhile(isNotSentinel).filter(isValid)
val averageRainfall = average(validInputs)
```

The sequenced-plans approach in Figure 3.9 uses, where possible, a
compartmentalization strategy by making extensive use of functions. By
switching the FoA at each function call (or function evaluation), sequenced
plans composition induces an SPSO processing pattern, reducing the num-
ber of simultaneously activated plans in the RDA.

To process the reading plan (line 7), the FoA switches to the sentinel
plan, composed of the loop plan, with both simultaneously in the RDA (PI
2). At the end of the input plan (sentinel found), the plans collapse to a
single result (a collection), stored in the RDA. The FoA switches to the
negative plan (also line 7), which is processed using (only) the result of
the previous step as input and which outputs a collection to the reading
plan (PI 1, processing just the negative plan). The reading plan is then
processed with its inputs and stored for later composition.

At line 8, a function call switches the FoA to the averaging plan of line 3.
The averaging plan activates the guard-against-zero plan (lines 4 and 6)
and the computation of the average itself. To compute the average (with
guard and average active in the RDA), the FoA switches to the sum plan
(using an SPSO pattern), which makes the sum plan the only active plan
in the RDA for now. After another switch to the count plan (SPSO again),
we are back to the first RDA in order to compute the averaging expression.
Having the average and guard plans activated yields a PI of 2. Overall,
the sequenced-plans Rainfall program of Figure 3.9 has an MPI of 2.

3.3 RQ 1.3: How to evaluate students’ prior-knowledge of program
comprehension?

To answer RQ 1.3, we developed a self-evaluation instrument (SEI) in-
spired by the self-assessment grid for common reference levels in CEFR
for languages [265, p. 26-27]. The SEI aims to provide a fast, easy to apply,
and widely available instrument to evaluate prior knowledge in program
comprehension that covers much of the common introductory programming
course content. The SEI can be used in repeated measurements without
losing validity. Although the instrument can use support information (e.g.,
guidelines, examples) that can be bound to a specific language or paradigm,
the instrument itself is language-agnostic.

The SEI presented in Figure 3.10 has two dimensions: concepts and
levels. The vertical axis contains the concepts related to programming in a
CS1 course, as listed by the FCS1 [363]: variables (var), Input and Output
(io), expressions (exp), conditional statements (sel), loops, lists, functions,
and classes and objects (obj). Although the FCS1 does not explicitly order the concepts in increasing levels of complexity, our instrument uses an order consistent with concepts’ order of complexity of a program written in an imperative paradigm, as described by the CCCP model in Publication II. The horizontal axis contains the estimated proficiency levels of programming concepts, inspired by the CEFR scale of proficiency in language skills [265] adapted to programming contexts.

The levels were designed using the principles of the stage of cognitive development of skills in the programming context [360, 193, 201]. Every level matches the description of a stage of cognitive development or familiarity with the concept. We defined complexity in terms of the number of interacting elements, using the CCCP model in Publication II to evaluate learner’s expertise level.

Each level should describe how significant are the chunks representing programming structures or even entire plans in learners’ long-term memory. Overall, the levels describe the path the learner takes to be fully proficient in a concept. Learners move from total unfamiliarity with the concept to familiarity with the concept drawn from other contexts. In the next level, learners can recognize and comprehend the syntax of the concept in a concrete program. Next, learners can trace a program step-by-step using concrete values with meaningful naming conventions (with a distinct number of interacting elements). Later, learners can comprehend a program using abstract inputs to finally being able to summarize code using a concept in a short sentence. Each level is described by a letter to define its stage and a number for a degree. Each level has a statement to be evaluated by the learner, who self-assigns to a level.

**The SEI and Programming Background Variables**

The first step towards validation of the SEI was performed by evaluating the instrument’s metrics of internal consistency and assessing the correlation of its metrics with other background variables described in the literature, as presented by Publication III. Students of a Massive Online Open Course (MOOC) from the University of Helsinki were asked to voluntarily fill an online form in the first week of the course containing the eight (8) variables from the SEI, personal information data, and thirteen (13) programming background questions. Overall, 2196 students granted research consent.

The internal consistency of the instrument was high for all concepts
### Table 3.10: The Self-Evaluation Instrument (SEI)

<table>
<thead>
<tr>
<th>Function</th>
<th>Higher-Order</th>
<th>Concept</th>
<th>Concept with this concept in mind</th>
<th>In general</th>
<th>Comments with code examples and explanations</th>
</tr>
</thead>
<tbody>
<tr>
<td>methods</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>class</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>function</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>loop</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>if/else</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>variables and assignments</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>control structures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>logic operators</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>arithmetic</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>basic input and output</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive naming</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive commenting</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive documentation</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive reasoning</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive practices</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive habits</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive procedures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive standards</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive conventions</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive protocols</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive policies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>descriptive guidelines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
(inter-item mean of 0.86, Cronbach’s alpha mean of 0.98). To investigate the theoretical latent factors in the data, a Factor Analysis (FA) was performed, yielding a two-factor solution. The first factor contains ten items: all concepts in the instrument, plus the background variables of other courses taken in high school (otherC), and languages that students consider themselves advanced users (advancL) or experts (expL) accounting for 65% of the variance. The second factor contains four items: otherC, advancL, expL, and total hours programmed (HPT) accounting for 12% of the variance. The first factor can be labeled as “concepts” and second as “prior knowledge”.

We investigated how the variables in the instrument were grouped using a Principal Component Analysis (PCA). The PCA indicates three distinct groups, which we interpreted as: basic (var, io, exp, sel, loops), advanced (lists, functions) and expert concepts (objects). This grouping was possibly influenced by the subject’s instruction (which we conjecture to be closer to imperative programming), and other contexts could yield different groupings and levels.

Since the PCA and FCA suggest the existence of three distinct groups of concepts in the instrument, we investigated if such concepts or groups constituted a hierarchy by analyzing patterns arising from responses. We evaluated for each respondent if the sequence of answers, analyzed by concept (labeled as single in Table 3.1), could be classified as strictly increasingly monotonic, increasingly monotonic, strictly decreasingly monotonic, decreasingly monotonic, constant or other. We hypothesized that patterns of constant, strictly decreasingly monotonic and decreasingly monotonic responses could indicate a hierarchy of concepts (basic concepts with high scores, gradually decreasing to low scores in objects). Of the 538 respondents classified as constant, 269 (50%) evaluated themselves as A0 in all concepts, 98 (18.21%) as A1, 24 (4.46%) as A2, 36 (6.50%) as B1, 24 (4.46%) as B2, 31 (5.76%) as C1, and 56 (9.85%) as C2.
Table 3.1. Patterns in students responses, analyzed by concepts (single) or by groups of concepts (beginner, advanced, mastery). N = 2190.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Single</th>
<th>Ratio</th>
<th>Groups</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monotonically increasing</td>
<td>29</td>
<td>1.32%</td>
<td>119</td>
<td>7.2%</td>
</tr>
<tr>
<td>Strictly monotonically increasing</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Monotonically decreasing</td>
<td>389</td>
<td>43.28%</td>
<td>490</td>
<td>64.53%</td>
</tr>
<tr>
<td>Strictly monotonically decreasing</td>
<td>0</td>
<td>43.28%</td>
<td>576</td>
<td>64.53%</td>
</tr>
<tr>
<td>Constant</td>
<td>538</td>
<td>538</td>
<td>538</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>1234</td>
<td>56.34%</td>
<td>469</td>
<td>28.39%</td>
</tr>
</tbody>
</table>

Given the high amount (56.34%) of non-classified responses (see Table 3.1), we investigated if the components described in the PCA could improve the classification methods. For each respondent, we calculated the mean of variables within each of the three groups (basic, advanced, and expert) and observed if such mean followed an increasing or decreasing pattern. Respondents previously classified as constant were excluded from this analysis. In Table 3.1, the “group” column represents this second analysis. The results show a decrease in the number of non-classified responses (56.34% - 28.39%) and an increase in the ratio of responses that can be classified as part of a hierarchy (43.28% - 64.53%).

To evaluate how the instrument compares with traditionally used ad-hoc measures of programming background, we calculated the correlations between the concepts instrument and traditional background variables. Figure 3.11 shows the correlations among all variables, with non-significant correlations (at $p = 0.05$) marked as crosses. All p-values were adjusted using the Bonferroni correction method.
The SEI and Performance Scores

In Publication IV, we investigated how students answered the SEI compared to a course exam and the SCS1 [270]. Data were collected at the same MOOC of Publication III. However, since students were still enrolling in the course, we have more respondents to the SEI. A total of 3807 respondents voluntarily answered the SEI at the end of the first week, 1379 at the end of week 4, and 867 at the end of week 7. The SCS1 was presented online as an optional task at the end of week 7 (simultaneously with the SEI). From the 640 students who answered the SCS1, 477 permitted to research with their data. We only considered test-takers those who answered at least ten questions [394], yielding a total of 440 respondents. At the end of week 7, 740 students took an online exam to determine the final course grade.

To investigate how students behaved in answering the SEI, SCS1, and
the course exam, we compared the completion rates of those instruments in Week 7. Since the participation was voluntary, students completed the instruments in different ways.

To investigate student answers to the SEI and SCS1, we compared the completion rates of the SEI in Week 7 with those of the SCS1 and course exam. Of the 867 respondents who attempted the SEI in Week 7:

- 867 (100%) completed the SEI.
- 498 (57.4%) completed the exam.
- 376 (43.4%) answered > 10 questions of SCS1 (test-takers).
- 319 (36.8%) answered at least half of SCS1 questions.
- 84 (9.7%) answered all questions of SCS1.

Of the 740 respondents who attempted the course exam:

- 740 (100%) completed the course exam.
- 498 (67.3%) completed the SEI.
- 254 (34.3%) answered > 10 questions of SCS1 (test-takers).
- 216 (29.2%) answered at least half of SCS1 questions.
- 55 (7.4%) answered all questions of SCS1.

Of the 477 respondents who attempted at least one question of SCS1:

- 422 (88.5%) completed the SEI.
- 283 (59.3%) completed the exam.
- 440 (99.2%) answered > 10 questions of SCS1 (test-takers).
- 365 (76.5%) answered at least half of SCS1 questions.
92 (19.3%) answered all questions of SCS1.

To investigate if the SEI could capture changes in students’ confidence, we explored the differences between each week’s means by concept ($\Delta$). All differences shown in Table 3.2 are significant ($p < .001$, Mann-Whitney U-Test). However, the effect size (Cohen’s d) of changes from week 1 to week 4 and from week 1 to week 7 is large, while the effect size from week 4 to week 7 is negligible or small, except for classes and objects with a large effect size.

Table 3.2. $\Delta$ of means by SEI concept and effect size of $\Delta$.

† = large, † = small and * = negligible effect size.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Week 4 - Week 1</th>
<th>Week 7 - Week 4</th>
<th>Week 7 - Week 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Delta$</td>
<td>Cohen’s d</td>
<td>$\Delta$</td>
</tr>
<tr>
<td>Var.</td>
<td>2.75</td>
<td>1.47†</td>
<td>.23</td>
</tr>
<tr>
<td>I/O</td>
<td>3.04</td>
<td>1.67†</td>
<td>.13</td>
</tr>
<tr>
<td>Exp.</td>
<td>2.7</td>
<td>1.48†</td>
<td>.22</td>
</tr>
<tr>
<td>Sel.</td>
<td>2.61</td>
<td>1.45†</td>
<td>.33</td>
</tr>
<tr>
<td>Loops</td>
<td>2.71</td>
<td>1.51†</td>
<td>.31</td>
</tr>
<tr>
<td>D.C.</td>
<td>2.4</td>
<td>1.41†</td>
<td>.35</td>
</tr>
<tr>
<td>Func.</td>
<td>2.5</td>
<td>1.46†</td>
<td>.44</td>
</tr>
<tr>
<td>Obj.</td>
<td>1.49</td>
<td>.83†</td>
<td>1.68</td>
</tr>
</tbody>
</table>

Figure 3.12 shows that such effect size differences match the timeline of course contents, particularly the small effect size of the change in all concepts from week 4 to 7, except for classes and objects, introduced in week 4, which could still be considered a “new content” for students.
To investigate if the instrument can distinguish between concept-specific differences in students’ evaluations of their ability, we examined if students’ answers could be better understood as single or multiple factor models performing a Confirmatory Factor Analysis (CFA). Week 1 analysis determined that the best fit for the data is an one-factor model, as shown in Figure 3.13 (a)(i). Week 7 analysis showed that a two-factor model was the best fit for the data, as presented in Figure 3.13 (a)(ii).

Figure 3.13. Factor loadings on weeks 1 and 7 for overall data (a) and experienced students (b).

<table>
<thead>
<tr>
<th>Week 1</th>
<th>Week 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i)</td>
<td>ii)</td>
</tr>
<tr>
<td>VARS</td>
<td>0.93</td>
</tr>
<tr>
<td>I/O</td>
<td>0.94</td>
</tr>
<tr>
<td>EXP</td>
<td>0.94</td>
</tr>
<tr>
<td>SEL</td>
<td>0.97</td>
</tr>
<tr>
<td>LOOP</td>
<td>0.96</td>
</tr>
<tr>
<td>DC</td>
<td>0.91</td>
</tr>
<tr>
<td>FUNC</td>
<td>0.86</td>
</tr>
<tr>
<td>OBJ</td>
<td>0.92</td>
</tr>
<tr>
<td>F1</td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 1</th>
<th>Week 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i)</td>
<td>ii)</td>
</tr>
<tr>
<td>VARS</td>
<td>0.92</td>
</tr>
<tr>
<td>I/O</td>
<td>0.93</td>
</tr>
<tr>
<td>EXP</td>
<td>0.93</td>
</tr>
<tr>
<td>SEL</td>
<td>0.98</td>
</tr>
<tr>
<td>LOOP</td>
<td>0.98</td>
</tr>
<tr>
<td>DC</td>
<td>0.94</td>
</tr>
<tr>
<td>FUNC</td>
<td>0.92</td>
</tr>
<tr>
<td>OBJ</td>
<td>0.92</td>
</tr>
<tr>
<td>F1</td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td></td>
</tr>
</tbody>
</table>
We investigated if the previously described factors and loadings would remain when focusing on expert students (measured by students’ self-reported total hours programmed). For experienced students, the best model fit for week 1 is a three-factor model. Analysis of week 7 data also suggested a three-factor model as best fit, but with different factor loadings for each concept, as presented by Figure 3.13 (b).

Many studies have investigated students’ performance using the FCS1/SCS1 (e.g.,[394, 256, 270, 363]). One main advantage of using a validated instrument is to produce comparable and generalizable results, providing meaningful information about the investigated context. To investigate the external validity of the SEI, we examined its reliability metrics and compared its scores with a course final exam and the SCS1 scores. Since we are replicating SCS1 results, we also present a detailed comparison between our results and previous SCS1 studies.

The metrics of internal consistency and construct validity show that the SEI is very reliable. Cronbach’s α reliability test is very high: \( \alpha = 0.98 \) on week 1, \( \alpha = 0.92 \) on week 4 and \( \alpha = 0.96 \) on week 7. The composite reliability of the instrument is also very high: 0.98 for week 1, 0.94 for week 4, and 0.95 for week 7.

The scaled final score (min = 0, max = 100) of the 440 SCS1 test-takers showed a mean of 36.98 points and a standard deviation \( \sigma = 22.15 \), similar to Parker et al. [270] results (M =35.85, \( \sigma = 13.1 \)). Cronbach’s α of our SCS1 data (α = .87) was higher than Xie’s et al. [394] (α = .7) and Parker’s et al. [270] (α = .59) values. Examining α changes when removing items, question 6 was the only one considered problematic with a .01 increase in α (Xie et al. found that items 20, 24 and 27 were problematic). By analyzing the factor loadings, we found that questions 6 (.08) and 20 (.18) had a poor loading while Xie et al. reported questions 20, 24 and 27 with poor loadings. Figure 3.14 shows the Spearman’s correlations (Bonferroni’s corrected) of all 27 items of SCS1 and its mean score.
Results

Figure 3.14. Correlations between SCS1 questions (1-27) and the test mean (M). Several questions showed non-significant correlations and could be considered problematic (e.g. 6, 20).

The final course exam test-takers achieved a mean of 76.16 points (min = 3.33, max = 100) and a standard deviation $\sigma = 23.26$. The Cronbach’s $\alpha$ reliability test showed that the course exam was reliable ($\alpha = .79$).

Figure 3.15 presents Spearman’s correlation between the course exam, SCS1, and SEI on week 7. The SEI on week 7 and SCS1 had a very similar correlation with course exam grades while being moderately correlated with each other (.31). Whereas Parker et al. [270] do not provide a correlation of SCS1 with course grades, Tew and Guzdial [363] reported the same correlation (.51) between the FSC1 and a final exam.
Figure 3.15. Distribution and Spearman’s correlations between the SCS1, course exam, and SEI (** p < .001).

We found similar student performance on the SCS1 and similar metrics of the SCS1’s internal reliability as earlier studies. We now seek to provide a more nuanced analysis to investigate if our data suggest difficulties in the same questions by conducting an Item Response Theory (IRT) analysis using previous research, mainly Xie et al. [394] and Parker et al. [270], as benchmarks for methods and results.

As the test was presented online, one question at a time, we argue that the conditional independence of items, one of the IRT requirements, is accepted. To conform to the unidimensionality requirement, a Confirmatory Factor Analysis (CFA) suggested that the data could be fitted into a single latent factor model. An ANOVA test confirmed that a three-parameter (3PL) model was a better fit than restricted models, with all items fitting our data. The SCS1 can be considered moderately difficult with a mean difficulty $\delta_j = .81$ (Table 3.3). Students’ ability ($\theta$) ranged from $\theta = [-1.74, 2.76]$ with a mean value of .018. All discrimination values ($\alpha_j$) can be considered meaningful (> .4). Since our analysis used a 3PL (as opposed to a 2PL by Xie et al. [394]) some questions showed a positive value in the third parameter, which can be interpreted as a pseudo-guessing value where students at lower levels of ability no longer show a 0 probability of having a correct answer, but instead a ($\gamma_j$) probability.

Table 3.3 shows that questions 6 and 20 were the most difficult ($\delta_j$), but
no question was considered too difficult ($\delta_j > 3$), as opposed to Xie et al. [394] results ($\delta_k$) where questions 5, 13, 15 and 18 were considered too difficult. Using Parker’s et al. [270] methodology (% of correct answers), our data suggest that questions 3, 11, 12, 19, and 23 could be considered moderately difficult and all other items hard. Parker et al. found that questions 1, 2, 3, 19, and 23 could be considered moderately difficult, and all other questions hard.

<table>
<thead>
<tr>
<th>$j$</th>
<th>$\delta_j$ (SE)</th>
<th>$\delta_k$</th>
<th>$\alpha_j$ (SE)</th>
<th>$\gamma_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.89 (.23)</td>
<td>.64 (.13)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>.62 (.16)</td>
<td>.89</td>
<td>2.03 (.52)</td>
<td>.15</td>
</tr>
<tr>
<td>3</td>
<td>-.17 (.58)</td>
<td>.27</td>
<td>.92 (.18)</td>
<td>.01</td>
</tr>
<tr>
<td>4</td>
<td>1.76 (.37)</td>
<td>2.26</td>
<td>.87 (.49)</td>
<td>.16</td>
</tr>
<tr>
<td>5</td>
<td>1.25 (.14)</td>
<td>3.91</td>
<td>2.16 (.58)</td>
<td>.18</td>
</tr>
<tr>
<td>6</td>
<td>2.93 (1.16)</td>
<td>.99</td>
<td>1.29 (1.17)</td>
<td>.31</td>
</tr>
<tr>
<td>7</td>
<td>.83 (.11)</td>
<td>2.33</td>
<td>2.1 (.44)</td>
<td>.06</td>
</tr>
<tr>
<td>8</td>
<td>1.02 (.13)</td>
<td>1.6</td>
<td>2.24 (.7)</td>
<td>.19</td>
</tr>
<tr>
<td>9</td>
<td>.78 (.32)</td>
<td>1.16</td>
<td>1.04 (.34)</td>
<td>.1</td>
</tr>
<tr>
<td>10</td>
<td>.71 (.21)</td>
<td>1.27</td>
<td>3.24 (2.16)</td>
<td>.23</td>
</tr>
<tr>
<td>11</td>
<td>.41 (.15)</td>
<td>1.9</td>
<td>2.12 (.49)</td>
<td>.2</td>
</tr>
<tr>
<td>12</td>
<td>-.016 (.09)</td>
<td>0.63</td>
<td>1.58 (.19)</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>1.46 (.19)</td>
<td>3.99</td>
<td>1.18 (.36)</td>
<td>.01</td>
</tr>
<tr>
<td>14</td>
<td>.52 (.12)</td>
<td>1.13</td>
<td>2.02 (.38)</td>
<td>.05</td>
</tr>
<tr>
<td>15</td>
<td>.98 (.09)</td>
<td>3.11</td>
<td>2.81 (.52)</td>
<td>.05</td>
</tr>
<tr>
<td>16</td>
<td>.86 (.16)</td>
<td>1.28</td>
<td>1.37 (.33)</td>
<td>.03</td>
</tr>
<tr>
<td>17</td>
<td>.7 (.1)</td>
<td>1.68</td>
<td>2.33 (.44)</td>
<td>.05</td>
</tr>
<tr>
<td>18</td>
<td>1.36 (.15)</td>
<td>5.07</td>
<td>1.39 (.2)</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>-0.33 (.08)</td>
<td>.74</td>
<td>2.51 (.33)</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>2 (.19)</td>
<td>NA</td>
<td>4.81 (4.06)</td>
<td>.09</td>
</tr>
<tr>
<td>21</td>
<td>.77 (.12)</td>
<td>1.58</td>
<td>1.66 (.31)</td>
<td>.03</td>
</tr>
<tr>
<td>22</td>
<td>.45 (.09)</td>
<td>1.36</td>
<td>1.57 (.19)</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>-0.55 (.11)</td>
<td>.08</td>
<td>1.28 (.17)</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>1.31 (.11)</td>
<td>NA</td>
<td>2.2 (.42)</td>
<td>.02</td>
</tr>
<tr>
<td>25</td>
<td>.49 (.09)</td>
<td>.78</td>
<td>1.56 (.19)</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>.84 (.1)</td>
<td>1.77</td>
<td>2.55 (.52)</td>
<td>.05</td>
</tr>
<tr>
<td>27</td>
<td>1.6 (.16)</td>
<td>NA</td>
<td>2.06 (6.1)</td>
<td>.04</td>
</tr>
</tbody>
</table>

### 3.4 RQ 1.4: What kind of activities can foster program comprehension and how such activities can be organized?

As suggested by previous CER works (e.g., [301, 55]), novice programmers should develop program comprehension skills as they learn to code so that they are able both to reason about code created by others, and to reflect on their code when writing, debugging or extending it. Publication V presents a comprehensive review of program comprehension studies and previous models of model comprehension. In this section, we will summarize the findings of our investigation. To answer RQ 1.4, we identified two main goals: First, to collect and define learning activities that explicitly address critical components of program comprehension; Second, to define tentative theoretical learning trajectories that will guide teachers as they select and sequence those learning activities in their introductory programming courses. We focus on how different tasks develop the skills of learners and
how tasks should be ordered to support effective learning progressions.

To achieve these goals, we interviewed instructors at various institutions in their classroom activities to foster Program Comprehension. We used the output of such interviews and other activities identified in the literature to describe Program Comprehension in the context of novice programmers and create a catalog of learning activities that could be used to foster program comprehension. Next, we used the Block Model [316] to create a map of learning activities and explored a tentative learning trajectory to foster program comprehension.

3.4.1 Activities to Foster Program Comprehension

While previous studies presented distinct definitions of program comprehension, in this work, we use a more comprehensive conceptualization that distinguishes between the process of Program Comprehension and tasks to achieve it:

**Program Comprehension (ProgComp)** – usually conceptualized as a process in which an individual constructs his or her mental model of a program. The mental model is expected to include features such as the elements and structure of the program (starting from the basic coding constructs), the execution behavior, and the purpose of the program.

**ProgComp task** – a task through which the learner encounters an artifact that represents the program. The artifact is typically source code, but might also be another form of specification such as the nodes in Parsons problems. The task asks the learner to engage with the artifact in some way. Through this interaction with the artifact, the learner is stimulated to build, elaborate on, and refine their mental model. As the learner interacts with the artifact, she may choose to create external representations such as notes, traces, sketches, or diagrams to help overcome the limitations of working memory and further support the development of her mental model.

Although what pertains to ProgComp is the ability to read, interpret and explain code, other related code editing activities require reading and writing skills, such as debugging, refactoring, or extending the functionality of existing code. Since these tasks require a code comprehension phase from learners, we consider them part of the ProgComp spectrum. Thus, we
will consider *ProgComp* tasks on this broader context, with a focus on the comprehension facet — be it the final goal or an *explicit* subgoal of the task at hand.

Learners can engage with *ProgComp* tasks by *explaining* programs, providing *annotations* to programming tasks, or even *modifying* code. In explaining or articulating tasks, the learner reads the code and then explains (to themselves or others) what they think the code (or parts of the code) is doing. In annotation tasks, students might be asked to add comments or highlights or to create secondary external representations such as trace tables or sketches. Modification tasks require adjusting or reworking the original code to, for example, correct a bug, make the program more readable or add a feature.

**Data collection** We interviewed 31 instructors from institutions in 10 countries (Canada, Finland, Germany, Italy, Peru, Spain, The Netherlands, Turkey, UK, and the USA). The interviewees are secondary school (9) or post-secondary teachers (22), some having taught at different instruction levels (4), including a primary school. Eight interviewees have been teaching for 20 years or more, 12 between 10 and 19 years, 7 at least for 5 years. Most of them have a background in CS, the others either in Mathematics or in Engineering fields. The majority of their students are learning CS or computing-related subjects. The interviews were conducted in person or by e-mail using a short structured interview.

**Interview coding and analysis** We used a team-coding method with several researchers participating in interview coding and analysis. We focused our analysis on three main themes: (1) definition of *ProgComp*, (2) what concepts and skills are the most important for students’ learning. Based on our findings, we evaluated which teaching aspects described by our participants matched cells of the Block Model.

Our approach to coding the first two main themes was inductive [221]: based on the interview data, we performed an initial coding and proposed the codes presented in Table 3.4 and Table 3.5. We used a deductive approach to match the activities reported by instructors into the Block Model.

**Teachers’ Views of Program Comprehension**
In Publication V, we asked our interviewees to “explain in a few words what the term ‘program comprehension’ means to you” and collected their responses. Except for one interviewee (“I do not think I explicitly teach
program comprehension, but rather writing code"), most teachers are aware of one or multiple aspects of ProgComp as summarized in Table 3.4. The range of coverage varied: 60% of instructors provided only one view, while 13% of them gave very comprehensive definitions that included many categories. Next, we present a summary of interviewees’ perceptions of program comprehension in each category.

Table 3.4. Practitioners’ views of Program Comprehension.

<table>
<thead>
<tr>
<th>ProgComp description</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProgComp as code reading ability</td>
<td>23</td>
</tr>
<tr>
<td>ProgComp as mental model of the NM</td>
<td>9</td>
</tr>
<tr>
<td>ProgComp as writing code</td>
<td>5</td>
</tr>
<tr>
<td>ProgComp as knowledge of prog. constructs</td>
<td>4</td>
</tr>
<tr>
<td>Other views of ProgComp</td>
<td>7</td>
</tr>
</tbody>
</table>

**ProgComp as code reading ability** Most teachers think of ProgComp as being able to read and explain (to themselves or others) code. Some interviewees linked reading skills to the ability to predict the outcome of executing the code, while one teacher argues that such understanding is more profound than merely being able to trace code.

**ProgComp as a mental model of the Notional Machine** Some interviewees presented the perspective of ProgComp as the process of developing a mental model of the notional machine ("what is happening beneath the hood").

**ProgComp as Writing code** Some interviewees presented programming as a skill progression where reading precedes writing code. Others discuss ProgComp in terms of a development cycle, including activities such as editing, compiling, debugging, and teamwork.

**ProgComp as Understanding Basic Constructs** Some interviewees associated ProgComp to the understanding of the basic imperative constructs, possibly including language syntax features. One of our interviewees explicitly expressed the need to distinguish between different levels of understanding in connection with "the progression of constructs".

**Learning objectives linked to Program Comprehension**

We asked our interviewees, "what concepts and skills do you want your students to learn in connection with program comprehension?"
Table 3.5 provides a summary of our analysis. Most instructors indicated either one (38%) or two (48%) learning objectives (LO), whereas three teachers indicate three LOs.

**Table 3.5. Learning objectives linked to ProgComp.**

<table>
<thead>
<tr>
<th>Learning Objective</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Develop a model of the notional machine</td>
<td>14</td>
</tr>
<tr>
<td>Being able to chunk and explain code</td>
<td>14</td>
</tr>
<tr>
<td>High level thinking and abstraction</td>
<td>12</td>
</tr>
<tr>
<td>ProgComp as knowledge of prog. constructs</td>
<td>4</td>
</tr>
<tr>
<td>Being able to write/modify/debug code</td>
<td>11</td>
</tr>
</tbody>
</table>

**Developing a Mental Model of the Notional Machine**  Nearly half of the interviewees mentioned the development of a mental model of the notional machine either explicitly or indirectly by referring to tracing tasks.

**Being Able to Chunk and Explain Programs**  Interviewees pointed as LO of ProgComp the ability to chunk code when reading. More often, however, their statements mention the ability to explain programs.

**High-level thinking and abstraction**  Several instructors explicitly addressed the role of abstraction both in ProgComp. Some teachers linked tracing activities to the ability of abstract and “imagine” the structure of the program before writing code.

**Being Able to Write/Modify/Debug code**  Concerning the ProgComp LO, several teachers mostly refer to code writing, modifying and debugging abilities, maybe also implying problem-solving skills. The underlying idea is that ProgComp can be developed by practicing programming, by writing code, somehow as a by-product outcome of this practice.

**Teachers views mapped into the Block Model**

Based on interviewees’ reported ProgComp activities and their perspectives of ProgComp, we mapped interviews’ content into cells of the Block Model matrix. Interestingly, although in the participants’ perceptions of ProgComp “reading ability” was discussed more frequently than “the notional machine”, the Programming execution (P) domain was the most frequent theme overall (74), followed by Function (F) as shown in Table 3.6.
Table 3.6. Block Model mapping of amount of interview references.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>(A)tom</th>
<th>(B)lock</th>
<th>(R)elationship</th>
<th>(M)acrostructure</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T)ext Surface</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>(P)rogram Execution</td>
<td>20</td>
<td>17</td>
<td>19</td>
<td>18</td>
<td>74</td>
</tr>
<tr>
<td>(F)unction</td>
<td>6</td>
<td>14</td>
<td>17</td>
<td>24</td>
<td>61</td>
</tr>
</tbody>
</table>

At most levels, the Text Surface (T) dimension was mentioned the least. Based on the interview data, we believe that most interviewees do not associate ProgComp with programming language syntax minutiae, or instructors were more focused on describing areas that students struggle with. Consequently, it makes sense that the T dimension, which is considered to be straightforward, is cited less frequently. The more challenging topics, program execution at the atom level (A-P) and strategic knowledge of relating goals to plans (R-P and R-F, M-P, and M-F), are well-represented.

**Collection and Categorization of ProgComp Activities**

Based on literature analysis, and examples of activities provided by our interviewees, we collected and categorized several types of activities to foster ProgComp. We analyzed each of the available tasks considering the level of complexity, task structure (atoms, blocks, relational, and macro) and what dimension of the program it concerns (Text Surface, Program Execution, and Functional) and categorized such tasks using the Block Model framework. Since the analysis of interviewees’ data did not yield any task to fill individual cells in the Block Model, we suggested new tasks with the potential to fill those gaps.

**Text Surface Tasks**  The text dimension of the Block Model is based on the concrete representation of a program. Learners should identify and discriminate between atomic elements in the text, then recognizing their organization into language structures of growing complexity, culminating in the overall program structure. The types of tasks in this category (illustrated in Publication V) are then focused on *statically detectable properties*, i.e., syntax as well as static typing.

**Program Execution Tasks**  In order to deal with the dynamic aspects of execution, the information provided by the program text is not sufficient,
and learners must deal with the operational semantics aspects of a program. At the heart of any characterization of the program dimension lies the construction of a viable mental model of the Notional Machine.

**Function or Purpose Tasks**

Relative to the function dimension of the Block Model, introducing properties *extrinsic* to the program at hand, comes into play. Drawing a borderline between (abstraction on) code execution features and purpose-driven features is not always straightforward, and it is likely to depend to a large extent on the knowledge assumed at a particular learning stage.

### 3.4.2 Learning Trajectories for Program Comprehension

Learning trajectories (LT) have garnered attention in other fields given their ability to model the path learners take while learning, which supports research-based curriculum development [315]. However, empirical knowledge about LT is largely absent in computer science education. One reason is that there is no established methodology to systematize and define learners’ progression in CS disciplines. Some recent studies attempted to extract data from the literature to create learning trajectories for sequence, conditionals, and repetition [294]; abstraction [291] and debugging [293]. These LTs provide a path for particular aspects for programming and comprehension, but to the best of our knowledge, there is no learning trajectory for *ProgComp* as an integrative skill.

Complex tasks such as program comprehension favor the holistic integration of knowledge, skills, and attitudes to avoid fragmentation, promote meaningful schema creation, and facilitate near transfer [374]. To achieve such goals, frameworks such as the Four Components Instructional Design (4C/ID [375]) provide a blueprint to sequence learning tasks in a trajectory from less to more complex tasks, promoting schema creation while not exceeding learner’s cognitive load capacity. It also presents a guide of how to provide scaffolding to learners, decreasing support so that learners can gradually accommodate increasingly complex schemas and, at the end of instruction, be able to perform in an authentic, complex environment.

The methodology presented here can support instructors in two ways: First, it provides practical examples for instructors on how to decompose a task that fosters *ProgComp* into less complex sub-tasks (suitable for beginners), and later move to more complex tasks aimed to advanced learners, working on different aspects of *ProgComp* as presented by the
cells of the Block Model. Second, it provides a guideline that could help instructors identify where a specific task fits into the Block Model and what particular aspects of program comprehension are being fostered.

Publication II defines the atomic elements in a program as plans and sub-plans that can be extracted from concrete programs by analyzing the relationship of syntactic and semantic elements in the code and the respective cognitive actions learners need to perform to comprehend the program. We use Publication II model to provide cues on how to decompose a ProgComp task into sub-tasks and fit them in the Block Model. What becomes evident is that comprehending code too can be decomposed into multiple facets, each of which can be practiced independently.

The LT for ProgComp defines a spectrum of activities that will foster program comprehension using as many cells in the Block Model as desired by the instructor. Different cells usually will use different activities that are better suited to achieve the desired learning outcome. Creating an LT is an iterative process where the instructor evaluates learner’s prior-knowledge in a particular context (e.g., using tests or self-evaluation instruments described by Publication III and Publication IV) to identify a sub-learning-outcome appropriate to learners’ needs, match this learning outcome to a given Block Model cell and use an appropriate activity to foster ProgComp at that cell. The process is repeated until the instructor is satisfied with the granularity of the LT (the number of activities in different cells) or if the LT reaches the lowest level of complexity in the Block Model (the sub-task is already simple enough, e.g., uses an Atomic-Text-Surface activity) and no further refinements are required.

LT could be used by instructors in two different ways, depending on their goal. In the first one, the instructor iterates through the task’s learning outcomes and evaluates if students’ needs, prior knowledge, and granularity will be addressed with a proposed spectrum of activities. If tasks are too easy or too difficult, the instructor can further decompose the tasks until a saturation point is reached. In a second way, the instructor follows an already defined learning trajectory, using the planned tasks to identify gaps in the existing set of activities and integrate new activities where needed. The LT could also work in tandem with diagnosing tools, where learners’ difficulties in a particular cell of the Block Model could be mitigated by using the appropriate activities.
**A Tentative LT for ProgComp**

As an example of LT, we take a typical comprehension task, to summarise the goal of a program in a short sentence, and show how it can be decomposed into subtasks, each fitting into one of the cells in the Block Model matrix. We consider the following problem: *Given an array \( A \) of temperature measurements (in degrees Celsius), summarize (in words) the goal of the Java program presented in Listing 3.2.*

**Listing 3.2.** Activity example: summarise the goal of the following program.

```java
int[] A = {20, 24, 23, 35, 30, 35};
int c=1;
int b=A[0];
for( int i=1; i<A.length; i++ ){
    if( A[i]>b ){
        b=A[i];
        c=1;
    } else {
        if( A[i]==b ){
            c++;  
        }
    }
}
System.out.println(c);
```

A correct solution for this task would be a sentence similar to “*print the frequency of the highest temperature in the dataset*”. However, for learners to achieve the correct solution, many different sub-tasks of program comprehension have to be performed: comprehend syntactic elements of the programming language, comprehend the behavior (semantics) of these elements in the code, and comprehend the goal of each particular element in the code. These subtasks are merged to create increasingly complex plans, moving up in the Block Model until the highest level plan, which is the overall goal of the program itself.

As discussed in previous sections, the summarizing activity (Macro-Function level) might overwhelm learners without sufficient practice or prior knowledge. Therefore, we decomposed the task according to the 12 cells of the Block Model table. For each cell, we used our PCK knowledge and the ProgComp activities and experiences from the teacher interviews to identify an activity that would most appropriately practice the corre-
sponding aspect of *ProgComp*. We worked out an entire example, finding an activity for every cell. Our tentative LT aims to show how a more complex task can be decomposed into more straightforward activities, rather than to suggest that one should be able to find a simpler activity for all Block Model cells.

Our decomposition of the program in Listing 3.2 is given in Figure 3.16. Considering the Block row in Figure 3.16, the Text Surface level now asks the learner to identify a section of code, consisting of a few lines, and with a higher-level naming — *the code belonging to the else statement* (akin to subgoal labeling [242, 217, 243]). While a student may do this quite mechanically, initially, by reading through line by line from else to the appropriate closing brace, with practice, they will be able to abstract over the detail, and see the block of code as a unit.

An instructor can assign any of the *ProgComp* sub-tasks in the Block Model to a learner. If a learner is unable to complete the task, this can indicate that their knowledge gap or fragile knowledge resides in this or a prerequisite block, so the instructor could use activities in the previous levels of the Block Model to direct the learner to activities that could improve *ProgComp* and close their knowledge gap. A learner who can complete a task should be challenged with a task residing above or next to the currently accomplished one. In each case, there are many types of *ProgComp* tasks from which an instructor can select. We made a template to describe these tasks as well as what can be done to adjust them to make them more/less difficult.
Figure 3.16. Exercise decomposition of the Listing 3.2 program. In each of the Block Model cells, the title (blue) describes the goal of the activity. Below the title there is an example statement of the activity, followed by the answer (A) of the activity.

**MACRO (M)**

**Indicate overall program structure**

Draw nested boxes to indicate the overall program block structure.

```
int[] I = {20, 25, 30, 35, 40};
int a;
int b=0;
for (int i=0; i<I.length; i++)
  if (I[i]==b)
    a=1;
else
  if (I[i]==b)
    a++;
else
  if (I[i]==b)
    a--;

System.out.println(a);
```

**Determine redundant code**

Identify and check all potential execution flows. Does each statement get executed at least once?

A: The code block below will not be executed if all elements in the array have the same value or if they are all smaller than the first element.

```
b=I[1];
a=1;
```

**Summarize purpose**

Summarize the goal of the program using a short sentence.

A: Prints the frequency of the highest temperature in the array.

**RELATIONAL (R)**

**Identify scope**

Indicate the scope of variable b.

```
int[] I = {20, 25, 30, 35, 40};
int a;
int b=0;
for (int i=0; i<I.length; i++)
  if (I[i]==b)
    a=1;
else
  if (I[i]==b)
    a++;
else
  if (I[i]==b)
    a--;

System.out.println(a);
```

**Complete the code and diagram**

The code and diagram below represent the same program. Complete both so they have a correct behavior.

```
int[] I = {20, 25, 30, 35, 40};
int a;
int b=0;
for (int i=0; i<I.length; i++)
  if (I[i]==b)
    a=1;
else
  if (I[i]==b)
    a++;
else
  if (I[i]==b)
    a--;

System.out.println(a);
```

**Parson’s puzzles**

The following program segment should print the highest value in the array. Rearrange the blocks into the for loop in the correct order to complete the program.

```
int[] I = {20, 25, 30, 35, 40};
int a;
int b=0;
for (int i=0; i<I.length; i++)
  if (I[i]==b)
    a=1;
else
  if (I[i]==b)
    a++;
else
  if (I[i]==b)
    a--;

System.out.println(a);
```

**BLOCK (B)**

**Identify blocks**

Draw a box around the code that belongs to the else statement.

```
int[] I = {20, 25, 30, 35, 40};
int a;
int b=0;
for (int i=0; i<I.length; i++)
  if (I[i]==b)
    a=1;
else
  if (I[i]==b)
    a++;
else
  if (I[i]==b)
    a--;

System.out.println(a);
```

**Parson’s puzzles**

The following program segment should print the highest value in the array. Rearrange the blocks into the for loop in the correct order to complete the program.

```
int[] I = {20, 25, 30, 35, 40};
int a;
int b=0;
for (int i=0; i<I.length; i++)
  if (I[i]==b)
    a=1;
else
  if (I[i]==b)
    a++;
else
  if (I[i]==b)
    a--;

System.out.println(a);
```

**ATOMIC (A)**

**Identify statements**

Draw a box around each assignment statement.

```
int[] I = {20, 25, 30, 35, 40};
int a;
int b=0;
for (int i=0; i<I.length; i++)
  if (I[i]==b)
    a=1;
else
  if (I[i]==b)
    a++;
else
  if (I[i]==b)
    a--;

System.out.println(a);
```

**Trace values**

Determine the value of a after execution.

A: a has the value 2.

**TEXT SURFACE (T)**

**PROGRAM EXECUTION (P)**

**FUNCTION/INTENTION (F)**

**Reflect on code**

For the code below, propose a more appropriate initialization than int b = 0.

```
int[] I = {20, 25, 30, 35, 40};
int a;
int b;
for (int i=0; i<I.length; i++)
  if (I[i]==b)
    a=1;
else
  if (I[i]==b)
    a++;
else
  if (I[i]==b)
    a--;

System.out.println(a);
```

A: b, which is the maximum value in the temperature array could be negative. A better alternative could be int b = I[0];

**Explain purpose of a block of code**

Describe the purpose of this block of code.

```
int[] I = {20, 25, 30, 35, 40};
int a;
int b=0;
for (int i=0; i<I.length; i++)
  if (I[i]==b)
    a=1;
else
  if (I[i]==b)
    a++;
else
  if (I[i]==b)
    a--;

System.out.println(a);
```

A: The block determines, stores in variable a, and prints the maximum temperature in a given array.

**Explain the goal of an element**

Given that array I represents daily temperature measurements and b is the maximum temperature measured before day i, what is the purpose of test I[i] = b? In terms of the problem?

A: Tests if the temperature on day i is higher than b, hence hotter than any previous day.
Results
4. Discussion and Conclusion

The main goal of this thesis is to provide tools for evaluating the cognitive complexity of computer programs. In the formalization of the Cognitive Complexity of Computer Programs (CCCP) model, presented in Publication II, we identified two components necessary to our analysis of complexity and difficulty. First, our analysis requires the ability to extrapolate “what learners should know” of the program semantics, possibly acquired through instruction in programming courses, that allow them to comprehend a program. Distinct pedagogical approaches and instruction might yield different numbers of cognitive actions present in learners’ mental models used to comprehend the elements of the program. Second, our analysis requires an assessment of “how much do they actually know” of program semantics, represented by measures of learners’ knowledge.

The first component and the structure of the code will describe how many elements are necessary to comprehend the program and how they are interconnected. The second element describes how many plans in the hierarchy learners can “collapse” and chunk, enabling learners to comprehend a program within the limits of their working memory. Available CER tools and methods do not specifically address the evaluation of these two components. Therefore, in Publication I, we developed an instrument to explicitly describe “what learners should know” in terms of Rules of Program Behavior (RPB). In Publication III and Publication IV, we developed and partially validate an instrument to investigate “how much do they actually know”.

By establishing the necessary toolset to evaluate the cognitive complexity of computer programs, we investigated how such toolset could be used to organize program comprehension tasks in tentative learning trajectories using activities to foster program comprehension in Publication V. Figure 4.1 presents how each work contributes to our primary goal of this thesis.
In the next sections we discuss several aspects and implications of our works and findings. Sections 4.1, 4.2, 4.3, and 4.4 discuss how our findings in each research question compared to previous studies. In Section 4.5 we revisit our main research question in light of the findings of our research questions. In Section 4.6 and 4.7 we discuss what are the implications of our work to CER and practitioners. Section 4.8 presents the limitations of our work. In Section 4.9 we present a broader perspective of our work: we organize current CER studies through the lens of the 4C/ID and the Ten Steps and how our toolset connects to instructional design frameworks. Finally, in Section 4.10 we present prospects for future research.

4.1 How to design explicit representations of the rules of program behavior?

Albeit CER has presented many works that use the Notional Machines (NM) conceptualization, there is still no consensus regarding its definition. Studies often claim that “this work uses a different NM”, but no clear account for such NM is presented. Moreover, while there is value in discussing what makes an NM or not, we would like to instead propose an instrument with similar characteristics, but with more clear definitions to serve more specific goals.

To answer RQ 1.1, in Publication I, we conceptualized the Rules of Program Behavior (RPB). RPBs are staged, detailed, concrete, and intentionally designed instruments that serve an explicit pedagogical goal. RPBs are essentially teacher-facing instruments used to communicate a model of instruction and teachers’ intentions on instructional design. An
RPB emphasize context and learning goals and cannot be dissociated from it. Much like an NM, RPBs are used to present the semantics of program execution in a particular programming language. Contrary to previous NM works, we enforce, by definition, a concrete representation of the system's rules. RPBs explicitly state that instructors should consider their context, audience, and goals when designing an RPB and explicitly present such considerations.

There are very few concrete NM examples in the literature. The closest instruments that resemble an RPB (if we consider them an NM) are formal semantics representations. However, while aimed at a particular audience, formal semantics were not designed to serve the pedagogical goal of teaching program behavior nor intentionally define a learning context and goal. It is possible to argue that visualizations (if they are NM or have NM embedded in it), while positively communicating the behavior of a system, are not RPBs since they do not explicitly communicate their pedagogical choices and goals.

Investigating learners’ mental models is a time-consuming and challenging task. Previous works employed instruments such as personal interviews to elicit learners’ mental models [299]. Concept inventories [282] can reveal learners’ misconceptions and indicate that learners’ possess incorrect mental models, but do not describe what sort of mental model learners’ possess. Tests can only determine, to a limited extent, what has been learned, but do not represent the cognitive actions learners possess when comprehending programs. Moreover, tests and interviews are difficult to scale to large courses.

We expect instructors and researchers, overall, to create RPBs distinct from the one presented in Publication I. We expect that new examples of RPBs will make use of the same framework to transmit their intention and clarify their goals to an audience. By providing clear guidelines and a concrete end product, we expect that RPBs allow a more transparent comparison between different pedagogical instruments. RPBs can also be used to support instructors to set expectations of learners’ mental models by examining RPBs used in previous instruction.
4.2 How to define and evaluate the cognitive complexity of comprehending concrete programs?

To answer RQ 1.2, we conceptualized a new model of cognitive complexity for program comprehension, the Cognitive Complexity of Computer Programs (CCCP) model, presented in Publication II.

Our analytical, a priori, model of cognitive complexity, differs from previous works using taxonomies in that we seek to characterize the content of the programming activities at a relatively fine level of detail. Since taxonomies such as Bloom’s characterize activities, not programs, it is possible to assign tasks using very distinct programs to the same level (e.g., assigning every code writing activity “create”). Lister and Leaney [196] used Bloom’s and SOLO taxonomies to characterize programming activities. They describe discrepancies in their results, where achievements in the category of “Relational Applying” were much higher than in the lower category of “Multistructural Creating”. Lister and Leaney argue that tracking code (“Applying”) is traditionally considered easier for students than creating code. However, according to the rationale of their taxonomy, the complexity of the code, expressed in the SOLO abilities it requires (Multistructural or Relational), is a more significant factor.

By adopting the axioms of complexity from the MHC, the CCCP provides a theoretical grounding for claims about the relative complexity of different programming constructs; by further considering the shifting focus of attention and cognitive load, the CCCP suggests that difficulty is not simply a linear sum of the entire code.

We are not the first to propose a set of cognitive complexity metrics for programming. Our present work overlaps that of Cant et al. [45]. However, we focus on a narrower set of metrics, operationalize them, and bring them to bear on actual programs.

The CCCP suggests a progression from the concepts required at the lowest levels of the plan hierarchy towards the higher-level roots. In this respect, our work shares some goals with earlier work that has proposed learning trajectories for introductory programming such as Mead et al. [228]. However, Mead at al. work focused on generic programming concepts, whereas we analyze plans in individual programs. Similarly, Izu et al. [155] suggested an intuitive learning trajectory in which the learner abstracts increasingly complex “building blocks” (language constructs and plan-like “templates”) in order to tackle the next concepts; the CCCP differs
Discussion and Conclusion

from this work in its use of MHC to structure the plan hierarchy.

Previous CER works described surface metrics to evaluate programs. Luxton-Reilly et al. and colleagues, like ourselves, analyzed individual programs in order to characterize their structure [207, 204]. Their analysis and metrics are fundamentally different from ours in that they focused exclusively on syntactic features of programs, whereas we have focused on cognitive plans. Contrary to surface-level metrics from Software Engineering, the CCCP is able to differentiate how constructs and plans affect complexity. The CCCP, through its interactivity analysis, can describe how the same program could have different difficulties and be comprehended in different ways by different learners.

Finally, a different sort of metric was validated by Morrison et al. [241], who estimated learners’ mental effort by asking students for their subjective perceptions of difficulty while engaging in various CS activities. We certainly see such post-activity surveys as being valuable, too, but we hope that the CCCP will mature into a complementary tool for the analytical a priori assessment of individual programs during instructional design.

4.3 How to evaluate students’ prior-knowledge of program comprehension?

To answer RQ 1.3, we designed and evaluated a Self-Evaluation Instrument (SEI), presented in Publication III. The SEI is a language-independent instrument that aims to evaluate prior programming knowledge of program comprehension for introductory programming courses. Unlike traditional instruments measuring prior programming knowledge, our instrument focuses on core areas relevant to learning programming and allows students to evaluate and reflect on their current level of knowledge.

Most of the previous works using questionnaires or self-evaluation instruments are not particularly focused on program comprehension skills. Previous questionnaires, such as the work presented by Feigenspan et al. [97] (see Table 2.6), treated programming ability as a single construct, not allowing the investigation of learners’ ability in particular concepts. Ngai et al. [258] (see Table 2.7) presented a self-evaluation instrument similar to ours, where students self-evaluate their expertise and programming skills by self-assigning to levels of ability using descriptors. However, their work did not differentiate learners’ expertise in programming concepts.
Aloutinen and Smolander [5] (see Table 2.8) work resembles ours in presenting a list of concepts and descriptors for levels of ability which learners could self-assign. However, their work used Bloom’s taxonomy inspired levels of ability, while ours is inspired by the CEFR levels, adapted to the programming context using previous CER that supports a hierarchy of skills and the CCCP to define the complexity of comprehending programs within each skill.

Our aim in designing the SEI is not to replace tests but to offer an additional supportive instrument. Current test instruments as SCS1 [363] and FCS1 [276] can provide a general score of ability, but given the number of interconnected elements in the code, such instruments are not well suited to point specific expertise in learners’ expertise using a particular concept. Overall, to the best of our knowledge, no other instrument provided the opportunity to evaluate learners’ ability in tracing, recognition of templates, and programming summarizing skills.

The analysis presented in Publication III and Publication IV suggests that the instrument is internally consistent, its reliability does not decrease with repeated measures, and its internal consistency remains higher than those of the tests. Principal component analysis of the responses identified three main components: (1) basic programming concepts, (2) advanced concepts, and (3) object-oriented programming concepts. Traditional background metrics, such as those presented by Feigenspan et al. [97], are still relevant and correlate moderately well with our instrument. The SEI can capture learners’ growing confidence, and the SEI scores match the course structure in which it was evaluated.

The SEI has the potential to be quickly applied (including in online settings), it can be easy to scale, and it can be applied as many times as necessary, potentially without losing its reliability, and it is generally available. Given such characteristics, we suggest that students are more likely to volunteer to answer the SEI than take a test (which can be strenuous and time-consuming). We suggest that the SEI could allow students from diverse backgrounds to be assessed with relative accuracy; the SEI may reduce test anxiety and provide more opportunities for students to self-reflect. We suggest the SEI improve learners’ meta-cognition skills as well as plan their learning. Given the simplified nature of the SEI, based on the extremely high completion rates and comparable correlations with course grades presented in our analysis, we suggest that the SEI can be used as an efficient additional tool in the assessment process of learning.
Furthermore, it potentially provides students an opportunity to self-reflect without the burden of feeling tested.

However, the instrument and our results have some limitations. The validation of the instrument shows that the prototype SEI achieved some of these goals partially. Measures of divergent validity (e.g., comparing the SEI with self-efficacy instruments) could provide further support of a distinguishable construct. There is certainly an overlap between self-efficacy and self-evaluation. Our results show that, like Danielsiek et al. [72], there is an increase in students’ perception of programming related ability and distinguishable factors in the instrument. Whereas Danielsiek et al. (ibid.) evaluate skills such as algorithm design, runtime analysis, and pseudocode writing and tracing, our instrument evaluates the ability to comprehend programming concepts.

4.4 What kind of activities can foster program comprehension and how such activities can be organized?

To answer RQ 1.4, Publication V established two main goals: first, to collect and define learning activities that explicitly address critical components of program comprehension; second, to define learning trajectories that will guide teachers as they select and sequence program comprehension activities in their programming courses.

To address the first goal, we investigated program comprehension from instructors’ perspectives and collected more than 60 different learning tasks/activities for Program Comprehension. We later presented examples of such tasks and mapped them into the Block Model matrix. Using the methodology presented in Publication V, we expect that practitioners can more easily tackle the task of categorizing program comprehension activities and support the creation of future program comprehension task repositories.

To address the second goal, we seek to present a concrete example of how to decompose a task into a learning trajectory aimed for program comprehension. We put ourselves in the role of practitioners facing an important question: “What does it mean, if a learner is not able to solve the Program Comprehension task at hand?”. Usually, the answer is “provide easier tasks to learners”. We support practitioners by presenting a list of activities and introducing methods to develop learning trajectories where a given activity can be decomposed into less complex tasks and fit into a
Discussion and Conclusion

Block Model cell. If learners’ are able to complete a task successfully, they should move to a more complex task in the trajectory. If not, learners’ can keep practicing tasks at the same level of complexity or even step back and attempt less complex tasks. Our methodology supports practitioners in choosing and adapting tasks that are appropriate to learners’ level of expertise, based on learners’ prior achievements and the task complexity. We describe task complexity in our learning trajectory in terms of the CCCP, where we analyze the content of programs, and in terms of the activity, using the Block Model.

Our work differs from previous learning trajectories (e.g., [294, 291, 293]) by using a program comprehension framework, the Block Model, and a model of program comprehension cognitive complexity, the CCCP, to create our learning trajectories. We expect that the design methods presented in Publication V could better support practitioners and researchers developing pedagogy that emphasize program comprehension as a precursory skill (e.g., [319, 395]) to create and adapt a diverse set of tasks aimed to foster program comprehension.

4.5 Revisiting Thesis Main Research Question

We proposed as main research question of this thesis to answer how to operationalize an a priori evaluation of the cognitive complexity of comprehending programs, and what are the implications of a model of cognitive complexity on the instructional design of introductory programming courses?

We presented a statement of this thesis:

“The rules of program behavior set the expected, based on learners’ instruction, cognitive actions that should be employed by learners when comprehending programs. It will be valuable to analytically extract metrics from concrete programs to describe how syntactical elements of the language (described by the rules of program behavior) and code-composition strategies, mediated by learner’s prior-knowledge, are sufficient to explain the cognitive effort demanded from learners in a code comprehension task. The cognitive complexity of programs can be used to design learning trajectories to foster program comprehension and inform the instructional design of programming courses.”
The main research question is partially answered by the findings presented in this thesis. We presented in this thesis the foundations that allow the evaluation of the cognitive complexity of comprehending programs by establishing a theoretical model of complexity aimed at program comprehension and the necessary supporting tools to operationalize such model in concrete programs. **RQ 1.1 and RQ 1.2** are partially answered by presenting the theoretical foundations and design guidelines of the Rules of Program behavior (RPB), the Cognitive Complexity of Computer Programs (CCCP) model. Our work is grounded on strong theoretical models from educational psychology, and while we do not provide empirical validation of RPBs and the CCCP, we present detailed case studies to support researchers and practitioners in performing their own analysis or designing their instruments.

**RQ 1.3** is answered by presenting the design and partial validation of our Self-Evaluation Instrument (SEI), which allows the detailed investigation of learners’ ability when comprehending programs using concepts usually covered by introductory programming courses. We provide a detailed analysis of the internal, construct, and convergent validity of the SEI. **RQ 1.4** is partially answered by collecting and categorizing tasks to foster program comprehension. We present a method to decompose and sequence those tasks to create learning trajectories, but no empirical validation of this method is provided in our work. We expect that our findings support answering fundamental questions posed by different groups of stakeholders in Computing Education. Next, we explore how this thesis can support researchers and practitioners.

### 4.6 Implications for Research

#### 4.6.1 CER Theories of Complexity

We suggest that our work can advance the research of neo-Piagetian models of programming and the investigations of hierarchies of programming skills. First and foremost, this research establishes that *it is not possible to dissociate learners’ performance in a task from their prior-knowledge and task complexity*. In our conceptualization, difficulty is the interaction between task requirements, expressed by its complexity, and learners’ expertise. Difficulty will be the observable outcome of task performance.
Disposition moderates the cognitive resources devoted by learners to perform a task. Disposition depends on factors that were not discussed in this thesis, but positively impacts performance such as engagement, motivation, environment variables, among others.

Therefore, our model suggests that there is no general reading or tracing ability. Such tasks can only be assessed in light of the the concrete programs used in a given task and who is performing such task. Our conceptualization suggests more generalizable interpretations of learners’ behavior according to the neo-Piagetian theory. While previous research described stages of development in terms of “learners who can only trace with 50% accuracy” [63], we offer a different perspective on such classifications. While the MHC also describes Piagetian stages of development, we do not characterize such stages in terms of generic behaviors or hard thresholds. Instead, we present the rules that contextually organize plans and cognitive actions yielding more complex behaviors when comprehending programs.

Teague et al. [362] describe different behaviors of learners when performing “concrete tracing” as opposed to “abstract tracing”. We support the notion of a more complex task when tracing abstract inputs, compared to tracing concrete inputs. Different abstractions organize different precursory actions, therefore yielding a different order of complexity. Our conceptualization also agrees with the empirical evidence supporting a hierarchy of skills. Increasing expertise allows learners to collapse plans (creating more complex schemata) and deal with more complex tasks. By chunking plans that encompass other lower-level plans (which are closer to programming language semantics), learners can move from step-by-step comprehension of programming constructs to be able to summarize the goal of higher-level plans. Finally, with enough expertise, learners can summarize the goal of the whole program (the root plan in our hierarchy).

Our findings may help interpret some earlier results that attempted to measure constructs’ demands. Our findings support Mühling et al. psychometric results [245], which revealed that sequences were easy for the participants, loops with a fixed number of iterations were easier than all items involving conditionals, and (as authors suggest, surprisingly) nested control structures were easier than a loop with an exit condition. Findings from Ajami et al. [2] suggest that conditionals were less complex than for loops, that the size of the expressions used as inputs for constructs had an impact on performance, and that flat structures are slightly easier
than nested ones; these results, seemingly contradictory, can be explained by our plan depth metric.

It has been suggested that students might find merged, interleaved plans difficult and that sequential plan composition is more natural to deal with [335, 341], and that a student’s plan composition strategy interacts with the likelihood they will write defective code [336, 99]. Our findings offer a partial explanation for these findings: merged plan composition demands more plans to be simultaneously processed in the region of direct access, which may result in cognitive overload. We provide a theoretical model to explain the impact of plan-composition strategies and code structure in complexity and difficulty.

Our work can also support researchers to design more “learnable” programming languages. Amy Ko [169] discusses how programming language designers “intuitively” define the “simplicity” of certain language constructs. We suggest that our work could assist programming language designers by distinguishing programming concepts complexity and support the design of less complex constructs aimed at a particular audience. For example, our framework supports, theoretically, “consume before produce” strategies, where learners’ are first introduced to readily-available functions, which abstract away details from learners using elements such as library functions, teacher-created functions, or even some simple higher-order functions. Later, learners can move to more complex constructs that achieve similar goals, such as plans using loops and conditional statements.

4.6.2 Notional Machines Research

Krishnamurthi and Fisler [175] argue that “understanding how different Notional Machines (NM) for the same language impact the formation of mental models of program execution is a significant open problem”. We suggest that the Rules of Program Behavior (RPB) could support answering such demand by providing explicit guidelines to design concrete representations of semantics that more accurately reflect the instruction process and serve as a possible standard tool for communication among researchers. Researchers can compare their RPBs and more accurately and explicitly provide an account of how their semantics are distinct.

Krishnamurthi and Fisler (ibid.) discuss potential implications of distinct NM for instruction: “if some notional machines prove easier or more robust for students to learn at first, should that affect the order in which we introduce language features, even if it means moving away from the models
that currently underlie most introductory curricula?”. While we do not provide empirical evaluations of RPBs, it is possible to analytically evaluate the efficacy of different RPBs by investigating how learners comprehend a program using distinct RPBs. Each RPB can be used to estimate the expected set of cognitive actions used by learners when comprehending a given program. These actions can then be used as input to the CCCP and analytically evaluate the complexity of a given program. Since the same program may yield different complexity metrics by using distinct RPBs as input, our analysis partially supports the comparison of language features by inspecting their yielded order of complexity.

We suggest that large scale, multi-institutional studies that sought to compare student performance using code of “similar complexity” [195, 325] could benefit from our work in different aspects. Usually, such studies reach the conclusion that introductory-level students commonly perform at a level below their teachers’ expectations [224, 195, 372]. In order for measurable progress to be made and reasonable expectations set, the complexity of the programs in such assessment instruments should be evaluated. Simon et al. [325] argue: “If we are to have a reliable measure of students’ abilities at reading and writing code, we would need to consider a minute analysis of the difficulty levels of code-reading and code-writing questions at the micro-level.” We expect that a tool like the CCCP could contribute towards such a “micro-level” analysis.

We suggest that RPBs can support researchers to more clearly describe (and present) the instruction process used at each participant institution and provide a concrete instrument to support claims that courses are, in fact, comparable. Currently, when providing contextual information, researchers often claim that courses are “similar”, usually presenting a list of covered concepts by each course. However, little information is provided regarding the timeline of instruction and the level of detail in which each topic is covered. We suggest that the required cognitive demands of different methods could be more accurately informed by inspecting the RPB used in instruction.

4.6.3 Assessment Research

Our SEI can provide a quick and detailed investigation of learners’ abilities when comprehending programs, and be used to replace or complement formal tests. Our catalog of program comprehension activities can support researchers by providing more diverse program comprehension tasks. We
expect that by using such a methodology, researchers could design and present better instruments to be used in their investigations. By having an a priori analysis of tasks’ complexity, researchers could predict learners’ performance. Where performance predictions are not corroborated, our toolset can better equip researchers to generate hypotheses and explanations for such phenomena.

In Publication IV, we show that the SCS1 performance in our results matches previous research. Nevertheless, the same could not be observed when analyzing question-by-question responses. Our Item Response Theory (IRT) analysis showed that not only the difficulties of the questions were ranked differently, but also some questions showed very dissimilar difficulties and performance when compared to previous studies (see Table 3.3 in Section 3.3). Some questions, for example, can be considered very easy in our results, but moderately difficult when compared to Xie et al. results [394]. Conversely, we observed the opposite phenomenon in other questions, which were considered difficult in our results but very difficult for students in Xie et al. analysis. We suggest that a complete analysis of the complexity, difficulty, and method of instruction could support researchers to better comprehend such inconsistencies in performance.

### 4.7 Implications for Practitioners

#### 4.7.1 Course Design

Similarly to researchers, practitioners can benefit from our work to design assessment instruments. Given the usual heterogeneity of students’ backgrounds, practitioners can benefit from quick and scalable methods to evaluate learners’ prior knowledge. Moreover, contrary to previous methods, the SEI can assess with moderate accuracy learners’ level of expertise in concepts usually covered in introductory courses, giving a fair estimation of where each learner will require additional support. Since complexity can, potentially, be analytically extracted a priori, practitioners can create repositories of granular assessment items (which evaluate fewer concepts simultaneously) to be used in exams and routine assessment, and match learners’ expertise with the program’s complexity, allowing learners to operate within the limits of their Zone of Proximal Development. Our analysis can assist teachers in setting more realistic expectations of learn-
ers’ performance by adequately adjusting the difficulty of their assessment instruments. However, in this thesis, we did not investigate how easily practitioners can make use of the CCCP and RPBs to evaluate the complexity of tasks, nor evaluated methods to match learners’ expertise with the cognitive demands of a task.

4.7.2 Choosing Programming Languages

When designing their course, teachers dwell with the choice of appropriate language/paradigm. We can partially support practitioners in this choice by analytically evaluating the complexity of programming constructs in a given program. Practitioners can express their instruction method using an RPB and compare the complexity of a given program in two different programming languages and make appropriate choices. However, we acknowledge that our analysis does not evaluate aspects of comprehension related to syntax and how the presentation could affect comprehension. Margulieux [212] presents an interesting discussion of her personal learning process using two similar representations of for loops. While, in theory, both constructs should achieve the same goal, the cognitive actions they require to be comprehended may be different, depending on the RPB presented to the learner. However, we conjecture that some presentation formats can support some external cognition and cause some form of syntax to be easier to comprehend than another. We did not explore such issues in the thesis, hope that future work can help elucidate such conjecture. Moreover, we acknowledge that the choice of programming languages in a course is more nuanced, potentially driven by factors other than pedagogical choices, such as perceived authenticity, supporting tools, a community of practice, and accessible learning materials, among others.

4.7.3 Learning Trajectories

We suggest that our work is a useful complement to the existing work that examines the complexity of different programming activities and the relationships between them. We expect that our findings can support establishing distinct learning trajectories for programming courses. In a micro-perspective, practitioners can evaluate which programming language constructs and concepts are more complex (in a context) and sequence them. From a macro-perspective, we presented a tentative effort to design a concrete progression of activities that help practitioners
decompose and sequence program comprehension activities.

4.7.4 Writing “Good Code”

Programming teachers hope their students produce “good code”. What constitutes good code is still an open question, but recent studies present evidence that students, teachers, and professional developers usually associate good code to “readable” or “comprehensible” code [34]. Usually, one crucial component of readability is the structure of the code or its plan-composition strategy. It has been argued that the same plan implemented with different structures could have different complexities, that plan-composition strategies can change the complexity, or that merged plans are more complex than sequence plans [341]. The CCCP interactivity analysis supports that merged plans are more complex than sequenced plans, and the hierarchical analysis supports that different structures implementing the same plan might have different complexities. We suggest that practitioners can benefit from our work in more clearly defining what less complex code means (in terms of comprehension) and support the usage of more sequenced plan-composition strategies. Our findings suggest that readily-available functions can reduce the complexity of programs by using less cognitive demanding structures and inducing more sequenced code.

4.7.5 Communicating Instruction

We propose that RPBs might be useful for practitioners in distinct scenarios. For example, given the current expansion of programming instruction at different educational levels (e.g., elementary, secondary, or high school), or programming courses directed at non-majors in higher education, instructors might not possess the necessary PCK to design courses or teach them in those settings. Instructors might be unsure of what is the correct detail level to be presented to learners in those settings and how to communicate program execution. In Section 3.1, we presented distinct alternatives to communicate the same program execution. What should be the appropriate choice based on cohorts’ characteristics and learning goals?

For example, an experienced instructor might have good evidence (anecdotal or empirical) that his course design is good and she could support others by communicating his method of instruction. We propose that RPBs
can support the communication of instruction methods by providing not only a concrete end product but also by presenting the context and goals in each RBP. This same scenario applies to tools that use RPBs as inspiration, such as visualizations and animations. By having an RPB as a supporting apparatus, visualizations could make explicit their pedagogical choices and allow instructors to make better-informed decisions regarding their tool of choice.

4.8 Limitations, validity and reliability

In this thesis, we have laid the theoretical foundations that allow the analysis of the cognitive complexity of concrete programs and presented some empirical evidence of the validity and reliability of our self-evaluation instrument. However, in this thesis, we have not provided empirical evidence that our toolset is able to accurately and reliably evaluate the cognitive effort demanded from learners when comprehending programs. We aimed to anchor our instruments in solid theoretical basis and present the necessary tools to be used by the CER community in the future. In the next sections, we discuss the limitations of each one of our findings.

4.8.1 Cognitive Complexity Analysis

Our methodological exploration of the CCCP is tentative. We have a goal, a framework, and examples of plausible analyses of programs in terms of the framework. Our analysis process has been used mostly in restricted research settings. Before our analysis of cognitive complexity can be applied more efficiently and transparently, we must further refine the steps that an analyst must take in order to elicit cognitive plans from an RPB and produce hierarchical and interactivity analysis using our model.

The CCCP is built on general theoretical models for which there is empirical support. Nevertheless, it needs to be directly evaluated and refined based on empirical findings. Each of our case studies reflects one possible breakdown of an example program in terms of the CCCP, and while we have provided a rationale for this analysis in theoretical terms, it remains to be seen whether it aligns with student performance, for instance. We believe that the foundations presented by this thesis will allow such evaluation.
Violating programmers’ expectations of code structure leads to imperfect comprehension, as existing schemata fail to apply. In our analysis, so far, we have only considered programs that are familiar and unsurprising. We did not investigate how “breaking patterns”, where learners are presented to similar, but different, plans may affect their comprehension of a program. The examples used in our analysis programs cover only a handful of basic plans. Additional work is required in order to extend the present work to other content (e.g., recursion, objects) and more complex plans.

4.8.2 Rules of Programming Behavior

The work with RPBs is still tentative. We provide a theoretical framework to support our new conceptualization and present how our work is related to previous Notional Machines studies and how the distinctions presented in this thesis set us aside from previous NM definitions and establish more clear learning objectives. It still requires empirical validation, and it is still unclear if the RPB is, in fact, able to mold learners’ mental models and how much learners deviate from the RPB instruction. There is no validation, besides the examples presented here, that RPBs can accurately and reliably elicit learners’ mental models or how easily this task is performed. The RPB should be tested in a real context and evaluated as a valuable tool for communication among instructors. We hope to have set a solid base for future discussions in the CER community.

4.8.3 Self-Evaluation Instrument

We acknowledge that self-evaluation can suffer, in general, from poor reliability and accuracy, and that can impact our SEI evaluations. It is not clear how truthful students are in their responses, how well they comprehended statements in the instrument, or how much effort they devoted to answering the questions. It is also possible that learners with distinct backgrounds have different standards and self-evaluate in different ways. Learners with high competency may underestimate, and learners with low competency overestimate their knowledge. Moreover, in our SEI analysis, we only analyzed a single course in a MOOC context, which reduces generalization claims. The SEI still requires more substantial evidence of external validation. While our results were comparable in this respect to those of the SCS1, we believe that both the SEI and tests could be improved to provide more reliable and accurate information.
used code-writing questions while the SEI evaluates code comprehension ability, which could explain the correlation levels observed in our results. Future research should also aim to compare the SEI and SCS1 with code comprehension course exams.

4.8.4 Program Comprehension Learning Trajectories

We presented a comprehensive list of activities to foster program comprehension, but by no means, it is a complete one. We hope that other activities might be added to our inventory in the future. The learning trajectory aimed for program comprehension is grounded in strong theoretical foundations, but distinct learning trajectories still need to be empirically validated in future work.

4.9 Computing Education Research Through the Lens of the Four Components/Instructional Design

We presented in this thesis a toolset that can potentially support researchers and practitioners to evaluate prior-knowledge, communicate their perspective of program behavior, evaluate the cognitive complexity of comprehending computer programs, and decompose tasks that foster program comprehension. Although we believe that our findings may prove to be a useful tool for researchers and practitioners, they are still limited to a particular niche of CER theories and practices. In this section, we would like to use our “teacher hat” and support the CER community by presenting a broader picture of how to apply our toolset, and how instructional frameworks can be used to navigate the CER literature.

We propose that frameworks such as the 4C/ID could support CS teachers to design their programming courses by holistically integrating authentic and engaging whole tasks while managing the learners’ cognitive load. Different from CER pedagogical approaches such as PRIMM, UMC, and POGIL, the Ten Steps could provide a more comprehensive blueprint to the instructional design. The 4C/ID has been used by previous CER works, with different degrees of conformity, to teach programming skills [128], to design instruction using microworlds [64], and in application development courses [211].

Our goal was to review the CER literature through the lens of the 4C/ID and present examples of how to map existing practices into the 4C/ID
components. This review had no intention to be comprehensive. We mainly focused on some CER conferences and journals (e.g., SIGCSE Symposium (2008-2019), ICER (2008-2019), ITiCSE, TOCE, Computer Science Education, IEEE Transactions on Education and others) and selected works we judged to be relevant to our review. We aimed to map existing research into the components of the 4C/ID to support teachers by providing instruments for “what works” while showing how the 4C/ID could organize CER practices into a holistic instructional design method. We suggest that teachers can later integrate the following steps described by the Ten Steps (see Section 2.6), which use the 4C/ID as the backbone to support their instructional design. In the next sections, we explore how existing CER practices fit each component of the 4C/ID.

4.9.1 Learning Tasks, Fading Support, and Variability

Learning Tasks

Contexts What constitutes an authentic programming task is still an open question. While some consider any code writing activity as authentic (after all, it produces a concrete result, a program that can be executed and manipulated), the Ten Steps states that authentic tasks should integrate skills, knowledge, and disposition from learners. CER has presented several works where a context is used to engage students, present an “authentic” setting, and provide some variability of practice to support distinct programming activities. CER has explored contexts such as electronic textiles [159, 160], game development [205, 19, 289, 18, 190, 56], robotics [353, 227, 184, 226], music remixing [208, 104, 384] and microworlds such as Alice [344].

Other approaches explored the integration of programming skills with other contexts such as Science [185], or Biology [45]. Bootstrap¹ has successfully integrated programming with Algebra [311, 312, 313] and game design [314].

Media Computation Perhaps the more prominent research regarding contexts is related to Media Computation [130], where students explore programming by engaging with rich media. There is evidence that Media Computation can be an engaging context and improve retention in programming courses [131]. Media Computation was successfully explored

¹https://www.bootstrapworld.org/
in MOOC environments [186] and with majors [329], and non-majors students [186].

However, not all aspects of Media Computation can be considered authentically aligned with a community of practice. Guzdial and Tew suggest [132] that a perspective of authenticity can be generated within a context (in this case, Media Computation) and influence learner perception of authenticity. While we agree with such conclusions, we conjecture that other scenarios could better integrate other programming skills. As we presented before in this thesis, programming cannot be solely reduced to code writing. Instead, we recommend that programming could be holistically explored through its many facets, including skills such as testing, designing, and comprehension. Such a viewpoint, where programming activities resemble Software Engineering (or studio-based) practices, was explored by previous studies in CS1 [150] and advanced courses [262].

Peer Review We previously presented in this thesis examples of tasks to foster program comprehension, including tasks aimed at sub-skills such as tracing (see Publication V). Peer-review (or code-review) share similar methods but have distinct goals of the program comprehension tasks presented in this thesis. CER has investigated peer-review in introductory courses [153], CS2 courses [369], and used peer review to investigated object-oriented design misunderstandings [368]. Furthermore, code-reviews have the potential to be scalable and used in online environments [152, 151].

Testing Another constituent skill of programming, testing, has more extensively been used with code writing skills. Approaches such as How to Design Programs (HtDP) [98] introduce a design process where learners’ first analyze of a problem statement to extract a rigorous description of program’s input and output data, encouraging learners to write test sets before coding. Denny et al. [79] show evidence that learners’ using a similar approach produced fewer errors when coding. Janzen and Saiedian [157, 156] propose a Test-Driven Learning (TDL) pedagogical approach where learners are introduced and explore new concepts through automated unit tests. Subsequent investigations using similar methods in CS1/CS2 courses [80] increased learners’ workload but allowed learners to develop unit tests while learning to program. Buffardi and Edwards [41] show that learners display different behaviors when testing using different tasks, and learners exhibit different testing behaviors between each other. Buffardi and Edwards argue that higher coverage in early development
is associated with higher quality code and with completing work earlier. A few other venues explored the integration of testing and peer-review \cite{333,278}. Overall, CER offers some evidence that testing is beneficial to learning and explored some pedagogical approaches to introduce tests in programming courses.

**Debugging** Debugging is a programming skill closely related to testing. Previous CER studies presented purposely bugged code to learners to assist them in developing their debugging skills \cite{10}; investigated the nature of debugging skills \cite{267}; and what strategies novices’ \cite{249,103}, end-users \cite{126} and different genders \cite{351} use to debug code. A few previous works presented tools aimed at learning to debug \cite{93,206}. Some studies explicitly presented debugging strategies to learners \cite{350}, with positive impacts on learners’ self-efficacy expectations and debugging performance. Overall, CER presents practices that support the instruction of programming constituent skills. However, with few exceptions, those skills are taught in isolation or integrating a few skills only. We suggest that authentic programming tasks could be achieved by integrating several constituent skills and using the presented contexts to engage students and promote the variability of tasks.

**Fading Support** Authentic tasks should integrate many constituent skills. Given the amount of element interactivity, introducing authentic tasks to beginners from the onset might overwhelm learners, particularly those less experienced. Therefore, The 4C/ID attempts to reduce the difficulty of tasks by providing scaffolding to learners using faded support. Fading support in programming tasks can be operationalized by presenting learners to tasks that follow a hierarchy of skills, moving from code comprehension tasks to code completion tasks, and finally being able to write code.

**Hierarchy of Skills** As learners gain expertise, support fades. Therefore, while learners can tackle increasingly complex activities, difficulty remains constant (challenging learners but not overwhelming them). After being able to comprehend programs (e.g., using tasks presented in Publication V), learners can manipulate readily-available programs using Parsons Problems, for example. Next, learners can create functional programs by completing or extending existing code. Code completion strategies have been successfully used in previous studies \cite{181,340,366,260,133,107}, and have shown the potential to increase learners' performance when
compared to code writing only strategies [232]. Ultimately, learners can explore code writing activities with no support.

We are not the first ones to propose a similar progression of activities. Merriënboer and Krammer [378] presented a very similar progression in programming courses aimed at high school students. Merriënboer [377] later presented evidence to support code completion strategies, where learners’ improved their ability to write code, and course retention was improved as well.

4.9.2 Sequencing Learning Tasks

Task classes define a group of activities that have the same complexity but are still distinct from each other. Task classes then should be sequenced to promote efficient learning trajectories. Evaluating complexity is our raison d’être and the driving question of this thesis. The ability to quantify and compare programs is crucial to operationalize instructional design using the 4C/ID.

As presented before in this thesis, previous methods did not allow a priori evaluation of complexity, making it challenging to design tasks classes. While our model, presented in Publication II, can potentially support the evaluation of the complexity of comprehending programs, we hope that future research could augment our work and expand it, allowing the evaluation of complexity of other tasks and skills.

4.9.3 Supportive Information

Supportive information should be presented to learners before and during task performance, assisting learners to construct schemata that will allow them to perform a task efficiently. Since supportive information should be connected to a task class, it is crucial first to comprehend learners’ prior knowledge. We presented several instruments to evaluate programming knowledge (see Section 2.4 and the SEI in Publication III and Publication IV). Based on learners’ expertise, instructors should adequately suggest materials that induce the creation of the necessary new schemata.

**Before Task Performance**

**Notional Machines** To efficiently comprehend and write programs, learners’ should be introduced to a model of computer [378]. We presented in Section 2.1 how CER describes a simplified model of the machine in terms
Discussion and Conclusion

of a Notional Machine. We suggested in this thesis a novel instrument, of the Rules of Program Behavior (see Publication I) that can support instructors to present explicit models of program execution to learners.

**Abstraction**  Bucci et al. [39] argue that strategies to foster abstraction, a crucial programming skill, usually are not explicitly introduced in programming courses, which affect learners’ performance. Koppelman and Dijk [171] show that beginners perceive abstraction as a new and challenging concept, which conflicts with their natural problem-solving strategies. Some CER previous works, such as Cook’s et al. [61] systematic approach to teaching abstraction and mathematical modeling, suggest that explicitly presenting abstraction to learners could improve their programming performance. Armoni [11] introduced a framework for teaching abstraction in the context of algorithmic problem-solving to novices, which was highly effective in developing CS abstraction skills and improve students’ general CS performance [348], particularly for girls [347]. Some CER works explored a similar vein: Rich et al. [295] present a framework of decomposition to Computational Thinking while Rich et al. [292] introduce a learning trajectory for decomposition in K-8 settings.

**Problem-Solving**  Another crucial CS skill, problem-solving, is rarely explicitly introduced in programming courses. Falkner and Palmer [95] included problem-solving classes to their programming courses, increasing class attendance rates, and improving students’ exam results. Their results show that problem-solving lectures also have an affective outcome: students’ surveys show that they have greater motivation for learning and believe they have a better understanding of concepts. While previous tutoring methods provided positive results when teaching problem-solving skills in programming courses (e.g., [200, 199]), Prather et al. [283] suggest a scalable, automated assessment tool to support learners’ problem-solving processes and developing metacognitive awareness. Their results show that students who received the intervention showed a higher degree of understanding of problems and were more likely to complete the programming task successfully.

**Plans and Goals**  In Section 2.2.2 we introduce plans, the stereotypical solution to programming problems. By explicitly presenting plans to learners, instructors can foster schema creation to be used by learners to compose programs. Krammer et al. [174] propose an intelligent tutoring system that explicitly presents plans and goals and their connection with code
and the problem at hand to learners; an approach similarly explored by Merriënboer and Paas [377]. Other similar approaches introduced algorithmic patterns in teacher preparation courses [285]; pattern-oriented instructional approaches [247]; and plan-based instruction using visual programming languages [148, 149].

**Roles of Variables** While plans are usually concerned with broader and generic goals, the Roles of Variables (RoV) [303] describe a concise set of stereotypical uses of variables that occur in programs covered in introductory courses. The RoV can be used in learning materials or presented through visualizations and animations [305, 306]. Previous RoV studies showed that it could improve learners’ debugging skills [3]; that students can understand the role of concepts and apply it in new situations [306]; and that the RoV can be applied in various programming paradigms [304].

**Live-Coding** Live-coding is an approach to teaching programming by writing actual code during class as part of the lectures [287]. Learners can construct new mental models by observing the heuristics and strategies used by an expert while coding. Potential benefits of live-coding include supporting learners to improve their program comprehension and debug skills and expose them to good programming practices [287]. Regarding learning improvements, results are somehow mixed: Rubin’s [301] results show that live-coding can have as good as or better outcomes than methods using static coding, while Raj et al. [286] experiments failed to show a difference between live-coding and static practices.

**During Task Performance**

**Peer Instruction** The previously presented practices support schema creation by presenting high-level strategies used in programming to learners. During task performance, it is possible to support schema acquisition or accommodation by encouraging learners to compare their mental models to peers using peer instruction [327, 326]. Peer instruction methods ask students to engage in meaningful discussions regarding their comprehension of a topic by comparing their answers to tests. Peer instruction has been successfully used in several studies [330, 28, 328], and it results show that it can halve course failure rates [280], and students perceive peer instruction as a valid instruction method [281]. Previous results show that peer instruction can improve students’ final exam grades when compared to traditional lectures [331] and that it can be successfully used in advanced
Discussion and Conclusion

CS courses, with results similar of those in introductory courses [187]; Peer instruction also has the potential to increase students’ self-efficacy [403].

**Pair Programming** Students can further test their comprehension of programming concepts and compare their mental models using less structured pedagogical approaches such as pair programming, which, in general, showed positive outcomes [37, 284, 379, 225, 370]. Braught et al. [38] compared individual and pair programming practices and provide evidence that pair programming improves individual programming skills (particularly of lower entrance examination score students), and students confidence in their work and ability to complete a course. Nagappan et al. [353] results show that pair programming can help retaining more students in introductory programming courses, and students have a more positive attitude toward working in collaborative environments.

**Visualizations** Visualizations and animations are popular tools to present program execution to learners and can be used to support mental model construction and accommodation. Sorva et al. [339] and recently by Al-Sakkaf et al. [4] present a comprehensive list of visualization and animation tools and evidence to support their usage.

### 4.9.4 Procedural Information

Procedural information should support learners in routine aspects of learning that are consistent across tasks. Procedural information help learners to automate existing schemata, usually by presenting rules, how-to instructions, step-by-step procedures, or corrective feedback. Procedural information is best presented when learners require it and should fade over instruction.

**IDEs** IDE usage is a usually overlooked aspect of programming courses, particularly when learners use IDEs aimed at professional usage. CER, however, has produced IDEs aimed at classroom settings such as BlueJ [24], Pythy [86], or in-between adaptations that bridge educational and professional IDEs [288]. However, there is little investigation on how to train students to use IDE in programming courses, particularly in introductory ones.

**Compiler Error Messages** Perhaps one of the more widespread forms of corrective feedback in CS are compiler error messages [21, 22, 209, 210, 189]. CER has dedicated a great deal of research to investigate how learners read error messages and produce methods to improve them. Presenting
more detailed error messages, particularly to novices, does not necessarily simplify learners’ understanding of errors [261]. Recently, Becker et al. [20] investigated a large body of research regarding compiler error messages. Their conclusions indicate that compiler error messages, particularly in educational settings, still require research and improvement: while error messages are pedagogically important, only a small percentage of studies are directly linked to pedagogy. Becker et al. argue that programming error messages are still problematic, regardless of language, and progress in the field has been slow.

**Automated Feedback** Given the size of some programming courses in some settings, CER has developed tools to support teachers to assess programming activities at scale, mostly using automated feedback or grading tools [189, 106], which have been extensively used to provide immediate feedback to learners. While automated feedback, particularly in grading systems, could help students identify bugs in their code, the systems may inadvertently discourage students from thinking critically and testing thoroughly [42]. Keuning et al. [165] review of automated feedback in CS shows that the feedback that tools generate is not very diverse and mainly focused on identifying learners’ mistakes. Keuning et al. argue that, in general, test-based feedback tools are most widespread, and very few those tools provide feedback with “knowledge on how to proceed”. Moreover, their review observed that teachers could not quickly adapt tools to their own needs, except for providing test data as input for a tool. While the literature shows that currently available tools still lack methods to present more constructive feedback, Ott et al. [266] present a road map to consider effective feedback practices in the CS1, possibly setting guidelines to future research. Their road map presents guidelines to evaluate communication, goals, performance, and activities for improvements at task, process, and self-regulation levels.

**Worked Examples, and Subgoal Labels** Worked examples [356, 233, 378] are an efficient practice to teach problem-solving procedures for well-structured problems, commonly used to demonstrate how to apply an abstract procedure to a concrete problem before learners can solve problems independently [217]. Worked examples have the potential to reduce learners’ cognitive load while learning to program [401]. When paired with subgoal labels, worked examples have the potential to produce a more efficient method of instruction and improve learners’ performance [216, 215], increase persistence in courses and have enhanced benefits to
lower-achieving students [217]. However, using subgoal labels and worked examples is not always straightforward. For example, Morrison et al. [243] results show that subgoal labels in worked examples can improve learning, but results are nuanced. In some contexts, asking students to generate their labels is more beneficial, while in others providing labels to learners presented better results. Margulieux and Catrambone [213] showed that learner-constructed labels used to scaffold problem solving improved learners’ outcomes when compared to expert construct labels.

4.9.5 A Road Map to Programming Instruction

The review of the CER literature through the lens of the 4C/ID and the Ten Steps suggests that some aspects of programming instruction received more attention and have strong supporting evidence, while others still require further investigation. Notably, our review shows strong support for practices such as Media Computation; peer instruction and pair programming; Parsons problems; worked examples with subgoal labels; and automated feedback.

Some of the practices presented in our review overlap and could be used in different components of the 4C/ID, other than those presented here. In general, most of such practices are used in isolation or integrate only a few other practices during instruction. While the pioneering work of Merriënboer and colleagues [233, 174, 378, 232] outlined guidelines for programming instruction that encompassed some of the practices presented in our review, and the Ten Steps framework [340] inspired some programming courses, there is little research regarding how to design, use and evaluate an integrative and holistic instructional design of programming courses.

We suggest that the works and findings in this thesis can contribute to the future instructional design of programming courses inspired by the Ten Steps. In Figure 4.2, we present a tentative road map to programming instructional design that integrates several existing CER practices. Furthermore, we suggest that some answers to questions that researchers and non-researchers in the CER community deemed relevant (extracted from Denny et al. [77], presented in Table 4.1) can be explored through this road map by future research.
Table 4.1. Questions deemed relevant by researchers and non-researchers [77].

<table>
<thead>
<tr>
<th>Researchers</th>
<th>Non-Researchers</th>
</tr>
</thead>
<tbody>
<tr>
<td>What fundamental programming concepts are the most challenging for students?</td>
<td>How and when is it best to give students feedback on their code to improve learning?</td>
</tr>
<tr>
<td>What teaching strategies are most effective when dealing with a wide range of prior experience in introductory programming classes?</td>
<td>What kinds of programming exercises are most effective when teaching students CS?</td>
</tr>
<tr>
<td>What affects students’ ability to generalize from simple programming examples?</td>
<td>What are the relative merits of project-based learning, lecturing, and active learning for students learning computing?</td>
</tr>
<tr>
<td>What teaching practices are most effective for teaching computing to children?</td>
<td>What is the most effective way to provide feedback to students in programming classes?</td>
</tr>
<tr>
<td>What kinds of problems do students in programming classes find most engaging?</td>
<td>What do people find most difficult when breaking problems down into smaller tasks while programming?</td>
</tr>
<tr>
<td>What are the most effective ways to teach programming to various groups?</td>
<td>What are the key concepts that students need to understand in introductory computing classes?</td>
</tr>
<tr>
<td>What are the most effective ways to scale computing education to reach the general student population?</td>
<td>What are the most effective ways to develop computing competency among students in non-computing disciplines?</td>
</tr>
<tr>
<td></td>
<td>What is the best order in which to teach basic computing concepts and skills?</td>
</tr>
</tbody>
</table>
4.10 Recommendations for Future Research

In the future, we intend to adopt a mixed-methods approach to evaluating the CCCP and developing the analysis process. In this thesis, we developed the necessary tools to operationalize such empirical investigation. The next steps in the evaluation may incorporate elements such as expert validation, correlation of predicted complexity with task performance through Rasch analysis (cf. how the MHC has been validated in other domains [59]), and triangulation against mental effort ratings [241] or physiological measures of cognitive load.

Since the interactivity analysis predicts that merged plans are more difficult to comprehend than sequenced plans, future investigations might investigate if such plan-compositions strategies, in fact, yield different performances from learners. Our interactivity analysis proposes that differences might be explained by distinct strategies that learners employ when comprehending programs using different plan-composition strategies. We suggest that more nuanced experiments that investigate learners’ strategy while reading and comprehending programs, for example, using eye-tracking devices, could provide empirical evidence to our theory.

To the present moment, our analysis has been restricted to program comprehension, but we hope that the CCCP can be augmented to evaluate code writing and other programming skills. We suggest that a complete model of task complexity for programming could significantly improve our field. We hope to have set the first seed to this endeavor. A more comprehensive task complexity model could allow researchers to compare not only programming tasks but perhaps compare tasks across fields, such as mathematics and science.

Since we adopt neo-Piagetian theories and models of cognitive development, in the future, it might be possible to refine the predictions for classroom learning and make some specific predictions about how much cognitive capacity is present at a particular age and match with task complexity. What is the appropriate programming task for children? Can they efficiently learn machine learning and other advanced topics? How to adapt instruction based on learners’ cognitive capacity? We suggest that the capability of matching task complexity and cognitive development could improve programming language design and assessment instruments aimed at specific audiences.

This study of the prototype SEI shows its potential to outline a context
independent framework that instructors can use to create comparable SEIs tailored for specific needs. In future validations, we aim to improve the clarity of descriptors by collecting student feedback, conduct qualitative investigations of student reasoning while answering the instrument, present contextualized examples, and replicate this work in other contexts.

As future work, we would like to explore how the community develops their own RPBs, possibly setting repositories for successfully utilized RPBs. Such repositories could be a valuable tool for practitioners, particularly in contexts little explored. For example, inexperienced (or in training) teachers in a given context (e.g., K-12) could benefit from more experienced instructors by using and adapting a successful strategy described by the RPB. Since such instruments are concrete and contextualized, we expect that teachers could better adapt such tools and incur fewer misconceptions by learners.


[10] Pasquale Ardimento, Mario Luca Bernardi, Marta Cimitile, and Giuseppe De Ruvo. Reusing bugged source code to support novice program-


[47] Francisco Enrique Vicente Castro and Kathi Fisler. Designing a multi-faceted solo taxonomy to track program design skills through an entire course. In Proceedings of the 17th Koli Calling International Conference on
References


References


References


[154] Cruz Izu, Claudio Mirolo, and Amali Weerasinghe. Novice programmers’ reasoning about reversing conditional statements. In Proceedings of the
References


170


[201] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory


[221] Philipp Mayring. Qualitative content analysis: theoretical foundation, basic procedures and software solution. 2014.


[223] Robert McCartney, Jan Erik Moström, Kate Sanders, and Otto Seppälä. Questions, annotations, and institutions: observations from a study of novice programmers. In the Fourth Finnish/Baltic Sea Conference on
References


References


References


References


[387] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. An australasian study of
reading and comprehension skills in novice programmers, using the bloom
and solo taxonomies. In *Proceedings of the 8th Australasian Conference
Australian Computer Society, Inc.

[388] Robert R. Whelan. Neuroimaging of cognitive load in instructional multi-

[389] Susan Wiedenbeck, Deborah Labelle, and Vennila NR Kain. Factors af-
fecting course outcomes in introductory programming. In *PPIG*, page 11,
2004.

[390] Susan Wiedenbeck and Vennila Ramalingam. Novice comprehension of

[391] Chris Wilcox and Albert Lionelle. Quantifying the benefits of prior pro-

[392] Brenda Cantwell Wilson and Sharon Shrock. Contributing to success
in an introductory computer science course: A study of twelve factors.
In *Proceedings of the Thirty-Second SIGCSE Technical Symposium on
Computer Science Education*, SIGCSE ’01, page 184–188, New York, NY,


[394] Benjamin Xie, Matthew J. Davidson, Min Li, and Amy J. Ko. An item re-
sponse theory evaluation of a language-independent cs1 knowledge assess-
ment. In *Proceedings of the 50th ACM Technical Symposium on Computer
ACM.

[395] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J. Davidsson, Dong-
sheng Dong, Harrison Kwik, Alex Tan Hui, Leanne Hwa, Min Li, and
Amy Ko Ko. A theory of instruction for introductory programming skills.

[396] Benjamin Xie, Greg L. Nelson, and Amy J. Ko. An explicit strategy to
scaffold novice program tracing. In *Proceedings of the 49th ACM Technical
Symposium on Computer Science Education*, SIGCSE ’18, pages 344–349,
New York, NY, USA, 2018. ACM.

and learner expertise: Split-attention and redundancy effects in reading
with explanatory notes. *Contemporary Educational Psychology*, 23(1):1 –

complexity of software. In *The Second IEEE International Conference on


Cognitive Complexity of Comprehending Computer Programs

Rodrigo Duran