Concepts and Tools for Teaching Programming

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
MICHELA CLAUDIA PEDRONI
Dipl. Informatik-Ing. ETH, ETH Zurich

June 17th, 1977

citizen of
Bellinzona, TI

accepted on the recommendation of
Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Hans Hinterberger, co-examiner
Prof. Dr. Juraj Hromkovič, co-examiner
Prof. Dr. Ulrik Schroeder, co-examiner

2009
Without the support, patience, and guidance of my advisors, colleagues, friends, and my family, this work would have been impossible. It is to all of them that I owe my deepest gratitude.

First of all, I would like to thank my Ph.D. advisor Bertrand Meyer, whose ideas and visions have shaped and guided my work. His commitment to improving the teaching of programming has made it possible for me to work in this fascinating and challenging area.

My deepest thanks also go to Manuel Oriol, a former member of the group, who has been a great mentor and contributed many ideas to this work. His drive, support, and friendship have helped me through the good and the rough Ph.D. times.

I am also very grateful to Juraj Hromkovič, Hans Hinterberger, and Ulrik Schroeder for being the co-examiners of my dissertation. A special thanks goes to Ulrik Schroeder for many interesting and influential discussions, hints, and the guidance that he provided during the work on my thesis.

TrucStudio was developed in collaboration with several students: Enrico Albonico, Lukas Angerer, Michele Croci, Florian Geldmacher, Pascal Goffin, Patrick Huber, Gerry Kammerer, Adrian Müller, Peter von Rohr, Jona Schoch, and Leo Widmer. Daniel Herding from RWTH Aachen acted as an alpha tester of the software and provided a model of object-oriented programming in Java. Their contributions have made TrucStudio a comprehensive framework.

This thesis would not be the same without the proof-readers Manuel Oriol, Sebastian Nanz, and Bernd Schoeller. Also the other past and present members, visitors, and friends of the Chair of Software Engineering have all, in their own way, influenced my work: Ruedi Arnold, Stephanie Balzer, Till Bay, Markus Brändle, Ilinca Ciupa, Susanne Cech Previtali, Carlo Furia, Claudia Günthart, Andreas Leitner, Benjamin Morandi, Marie-Hélène Nienaltowski, Piotr Nienaltowski, Martin Nordio, Max (Yu) Pei, Marco Piccioni, Nadia Polikarpova, Wolfgang Schwedler,
Stephan van Staden, Marco Trudel, Julian Tschannen, Jason (Yi) Wei, and Scott West. A special thanks goes to Till Bay, my friend and former office-mate. We have traveled a long way together.

Finally, I want to thank my friends and my family, in particular, Evi Pedroni, Eros Pedroni, and Andreas Pedroni. They always gave me their love, support, and confidence; without them I could not have carried out this work.
Teaching and learning to program are challenging activities. Some of the foremost issues for educators of introductory programming courses are the diversity of the student body and the complexity and interrelatedness of programming concepts, in particular for object-oriented programming. The student diversity makes it difficult to adapt instruction to students’ prior experience in programming. The complexity of object-orientation leads to challenges when structuring course material and developing examples. The contributions of this thesis provide tools, concepts, and data to address these issues.

The first part of this thesis focuses on student diversity. It presents a study carried out over six years with over 750 students of the Introduction to Programming course at ETH and analyzes students’ prior programming knowledge when entering the CS program.

The results suggest that the students are at one of three levels: roughly one third of the students have little or no programming expertise, another third know one or two programming languages well or very well, and the remaining students have experience with three or more programming languages.

The comparison of these numbers between the six course iterations shows no trend changes and indicates that instructors cannot ignore this issue. Furthermore, the phenomenon seems to be an international issue as evidenced by a data comparison between the ETH students and students from a second institution in another country.

A second set of contributions addresses the high interrelatedness of object-oriented programming concepts. The contributions include a modeling approach for domains of teaching and associated courses, a supporting tool called TrucStudio, and their application to object-oriented programming and the course Introduction to Programming at ETH.

The proposed modeling approach contains two parts: a domain model and a course model. The domain model describes a subject area based on knowledge entities of three levels of granularity (in increasing order of
granularity: *notion*, *Truc* or Testable Reusable Unit of Cognition [85], and *cluster*). It also defines several types of links, which capture the prerequisite structure between the knowledge entities. The course model defines the lectures of a course as paths through the domain entities.

The TrucStudio tool supports the generation and maintenance of the domain and course models. It provides an output generation mechanism to distribute the descriptions to instructors and students, and offers tools for the analysis of the prerequisite structure.

The application of the approach to object-oriented programming resulted in a set of Trucs and a course model for the ETH Introduction to Programming course. The Trucs contain concise descriptions of various technical, pedagogical, and structural aspects connected to the topics captured by each Truc. The pedagogical sections present a reusable collection of common misconceptions, which help improve examples and teaching materials. The structural analysis of dependencies between topics verifies the claim of high interrelatedness for object-oriented concepts and identifies clusters of highly related topics.
ZUSAMMENFASSUNG


Die Analyse der Daten zeigt, dass die Studenten anfangs einer von drei Kenntnisstufen zugeordnet werden können: Etwa ein Drittel aller Studenten hat keine oder nur sehr wenig Programmiererfahrung, ein weiteres Drittel hat gute oder sehr gute Kenntnisse von einer oder zwei Programiersprachen, und die restlichen Studierenden haben eingehende Erfahrung mit drei oder mehr Programiersprachen.

Der zweite Teil der Dissertation untersucht die Komplexität von objektorientierten Konzepten. Hier leistet die Dissertation die folgenden Beiträge: eine Methode zur Modellierung von Fachgebieten und Vorlesungen; die TrucStudio Software, die die Erzeugung und Verwendung der Modelle unterstützt; und die Anwendung der Methode auf die objektorientierte Programmierung und die ETH-Vorlesung “Einführung in die Programmierung”.


Das Programm TrucStudio unterstützt die Erstellung und Verwaltung der Modelle. Es erlaubt die automatische Generierung von Dokumenten, um die Modelle für Lehrer oder Studenten zu publizieren. Ausserdem unterstützt TrucStudio die Analyse der Konzeptabhängigkeiten.

CONTENTS

1 Overview and contributions  
  1.1 The challenges of teaching and learning programming . . . 1
  1.2 Student diversity . . . . . . . . . . . . . . . . . . . . . . 2
  1.3 Object-oriented introductory programming . . . . . . . . . 3
  1.4 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . 5

2 The ETH Introduction to Programming course  
  2.1 Related approaches . . . . . . . . . . . . . . . . . . . . . . . 10
  2.2 The Inverted Curriculum . . . . . . . . . . . . . . . . . . . . 12
  2.3 Course setup . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
    2.3.1 Schedule . . . . . . . . . . . . . . . . . . . . . . . . . . 13
    2.3.2 Covered knowledge areas and general learning goals 14
    2.3.3 Traffic and teaching material . . . . . . . . . . . . . . 14
  2.4 Feedback on the course . . . . . . . . . . . . . . . . . . . . . 17

3 Student backgrounds  
  3.1 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
  3.2 Study setup . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
    3.2.1 Questionnaire on student backgrounds . . . . . . . . . 23
    3.2.2 Participants from ETH . . . . . . . . . . . . . . . . . . 26
    3.2.3 Participants from University of York . . . . . . . . . . 26
  3.3 Threats to validity and limitations . . . . . . . . . . . . . . 27
  3.4 Statistical tests . . . . . . . . . . . . . . . . . . . . . . . . . 28
  3.5 Computer literacy . . . . . . . . . . . . . . . . . . . . . . . 29
    3.5.1 Interpretation . . . . . . . . . . . . . . . . . . . . . . . 33
    3.5.2 Teaching implications . . . . . . . . . . . . . . . . . . 33
  3.6 Programming experience . . . . . . . . . . . . . . . . . . . . 33
    3.6.1 Interpretation . . . . . . . . . . . . . . . . . . . . . . . 36
    3.6.2 Teaching implications . . . . . . . . . . . . . . . . . . 37
  3.7 Programming languages . . . . . . . . . . . . . . . . . . . . 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.17 Instruction</td>
<td>110</td>
</tr>
<tr>
<td>5.2.18 Linked list</td>
<td>112</td>
</tr>
<tr>
<td>5.2.19 Loop</td>
<td>114</td>
</tr>
<tr>
<td>5.2.20 Multiple inheritance</td>
<td>117</td>
</tr>
<tr>
<td>5.2.21 Object</td>
<td>119</td>
</tr>
<tr>
<td>5.2.22 Object creation</td>
<td>122</td>
</tr>
<tr>
<td>5.2.23 Polymorphism</td>
<td>125</td>
</tr>
<tr>
<td>5.2.24 Primitive type</td>
<td>128</td>
</tr>
<tr>
<td>5.2.25 Recursion</td>
<td>130</td>
</tr>
<tr>
<td>5.2.26 Reference</td>
<td>133</td>
</tr>
<tr>
<td>5.2.27 Stack</td>
<td>135</td>
</tr>
<tr>
<td>5.2.28 System execution</td>
<td>137</td>
</tr>
<tr>
<td>5.3 Analysis of the dependency graph</td>
<td>139</td>
</tr>
<tr>
<td>5.3.1 Transitive dependencies</td>
<td>139</td>
</tr>
<tr>
<td>5.3.2 Cycles</td>
<td>141</td>
</tr>
<tr>
<td>5.3.3 Comparison to other models</td>
<td>142</td>
</tr>
<tr>
<td>5.4 Application to Introduction to Programming</td>
<td>145</td>
</tr>
<tr>
<td>5.4.1 Possibly missing notions</td>
<td>148</td>
</tr>
<tr>
<td>5.4.2 Topic sequence</td>
<td>149</td>
</tr>
<tr>
<td>5.4.3 Common confusions</td>
<td>150</td>
</tr>
<tr>
<td>5.5 Implications on teaching</td>
<td>151</td>
</tr>
<tr>
<td>6 TrucStudio and its architecture</td>
<td>155</td>
</tr>
<tr>
<td>6.1 Related work</td>
<td>155</td>
</tr>
<tr>
<td>6.2 Overview</td>
<td>156</td>
</tr>
<tr>
<td>6.3 Domain modeling</td>
<td>157</td>
</tr>
<tr>
<td>6.4 Course modeling</td>
<td>159</td>
</tr>
<tr>
<td>6.5 Output generation</td>
<td>161</td>
</tr>
<tr>
<td>6.6 Structure report</td>
<td>163</td>
</tr>
<tr>
<td>6.7 Semi-automatic extraction of notions</td>
<td>164</td>
</tr>
<tr>
<td>6.7.1 Application to teaching material</td>
<td>167</td>
</tr>
<tr>
<td>6.7.2 Results</td>
<td>168</td>
</tr>
<tr>
<td>6.7.3 Summary</td>
<td>171</td>
</tr>
<tr>
<td>6.8 Architecture of TrucStudio</td>
<td>172</td>
</tr>
<tr>
<td>6.8.1 Overall system architecture</td>
<td>173</td>
</tr>
<tr>
<td>6.8.2 Entity model</td>
<td>174</td>
</tr>
<tr>
<td>6.8.3 Graphical interfaces</td>
<td>176</td>
</tr>
<tr>
<td>6.8.4 TrucStudio XML (TXM)</td>
<td>177</td>
</tr>
<tr>
<td>6.8.5 Generic output generation</td>
<td>178</td>
</tr>
<tr>
<td>6.8.6 Structure report</td>
<td>181</td>
</tr>
<tr>
<td>6.9 Summary</td>
<td>183</td>
</tr>
</tbody>
</table>
7 Future work
7.1 Student backgrounds 185
7.2 Trucs for object-oriented programming 186
7.3 TrucStudio 187

8 Conclusions 189
CHAPTER 1

OVERVIEW AND CONTRIBUTIONS

Learning to program and teaching programming are difficult tasks. On the side of the learner, programming is a complex activity that involves skills and mental models that many novices struggle to develop during programming courses. On the side of the instructor, teaching programming presents considerable challenges in particular with respect to the complexity of the subject matter and the diversity of the student body of introductory programming courses.

This chapter describes two challenges in more detail and summarizes the contributions of the thesis in these areas.

1.1 The challenges of teaching and learning programming

University teachers and CS education researchers acknowledge programming to be a challenging and complex activity and agree that learning to program “involves a variety of cognitive activities, and mental representations related to program design, program understanding, modifying, debugging (and documenting)” [114]. Researchers estimate that it takes approximately ten years to turn a novice into an expert [138]. In particular, novices have problems with understanding the abstract concepts [71] and with grasping the syntax and semantics of a particular programming language [127]. They have misconceptions on numerous programming language constructs [127] and they do not know how to combine programming structures into valid programs [138].

These difficulties are a global phenomenon as evidenced by the study of the McCracken group [79], conducted with students of several univer-
sities across multiple countries. The study assessed the programming and problem solving ability of students taking an introductory programming course. It shows that the majority of the students perform much more poorly than their instructors expected. More concretely, many students lacked the basic technical skills needed for getting a program ready to run – skills that are a prerequisite to problem solving. Another study [74] tested students on their ability to predict the outcome of short code fragments and to complete code when given a description of the desired functionality and a small set of possibilities. This study confirms the observations of the McCracken group.

Programming being a complex activity makes the teaching of programming challenging as well. In fact, a conference on grand challenges in computing education identified teaching programming as one of them [80].

Various issues contribute to the challenges of teaching programming. These issues can be categorized as being intrinsic or accidental complexities [10]. Accidental complexities include the use of inadequate tools, languages, or teaching strategies. They have the property that they can be changed. The intrinsic complexities are inherent to the nature of the subject matter and cannot be removed. Because they are immutable, advancing knowledge on them and developing tools and processes to handle the resulting challenges is important.

This thesis focuses on two complexities. One of them is the diversity of the student body of introductory programming courses in terms of prior programming experience. This complexity is not intrinsic, but it is a consequence of environmental factors related to the spread of computing in society and secondary schools. These factors are mostly beyond the control of a university teacher, so that the student diversity exhibits similar properties as intrinsic complexities. The second challenge on which we focus is the complexity and interrelatedness of object-oriented concepts. It results in a set of difficulties for teachers; for example, it makes it difficult to ensure a sound sequence of topics and it makes it necessary to address misconceptions of students.

1.2 Student diversity

To “know your audience” is one of the fundamental rules of mass communication [123]. This particularly applies to an educational setting: understanding the backgrounds of students is essential for providing quality educational programs tailored to their interests and needs.

Instructors of advanced courses for computer science majors are gen-
erally able to rely on basic knowledge of computing and programming. This is rarely the case for introductory programming courses where students start with a variety of backgrounds; “students are diverse in terms of their prior experiences, their pre-existing skills, their expectations and their motivations” [64].

The diversity with respect to students’ prior computing and programming experience seems to be a rather recent phenomenon. According to Jenkins and Davy [64], in the 1980s, most students approaching degree level courses had some sort of background with using and programming microcomputers. This has changed with the rapid expansion of computing: although a substantial part of the student body has significant experience with programming – some even as professional programmers in industry – another large part starts the study without any strong background in computing and programming.

The diverse prior programming experience of entering CS majors presents considerable challenges to the instructors of introductory programming courses. To keep all students motivated, instructors need to create an environment in which students feel interested and sufficiently challenged while their estimate of being able to succeed is high. The student diversity makes it difficult to design instruction such that it is beneficial to everyone – on one end of the range to the substantial percentage of novices and on the other end to the students who begin the course already with significant programming experience.

For the development of strategies that handle the student diversity, it is important to know how the student body is composed and to investigate whether there are changes in this composition over the years. To answer these questions, we have developed a questionnaire that entering CS undergraduate students fill in during the first weeks of their first semester of study. The data collected over the last six years allow us to highlight trend changes and to gain insights into students’ abilities on entering the course.

### 1.3 Object-oriented introductory programming

Teaching programming to novices is also difficult due to the inherent complexity of the subject matter. Linn [73] proposed a “chain of cognitive accomplishments from computer programming instruction”. It contains three main links of increasing cognitive difficulty: (a) understanding single programming language features that allow students to read and slightly change existing code, but not to compose more complex programs, (b) developing design skills including templates (schemata or plans) and
procedural skills that enable students to solve a problem by combining language features into programs, and (c) developing general problem-solving skills that make it possible to apply the gained knowledge to new formal systems. As compellingly confirmed by most studies (e.g. [73, 79, 74]), many novices do not even pass the first stage of this chain.

If courses should follow the chain of Linn, then it seems natural to first expose students to “single programming language features” [73]. Isolating single features may generally seem easy to do, but it presents considerable challenges for object-oriented programming.

Since the mid 1990s, object-orientation has entered the classrooms of introductory programming courses. Many schools have since then adopted an “objects-first” or “objects-early” approach for their CS1 courses and researchers as well as educators have proposed numerous tools, approaches, and strategies (e.g. [10, 124, 25, 58]). One of the claims made on object-orientation is that “the basic concepts are tightly interrelated and cannot be easily taught and learned in isolation” [10].

The tight interrelatedness of object-oriented concepts has several implications on teaching introductory programming: (1) it makes finding an initial sequence of single language features difficult (a phenomenon known as “big bang problem”), (2) it complicates ensuring that prerequisites are met at all times in the teaching sequence, and (3) it results in a larger quantity of feature facets due to features appearing in combination, which in turn makes it more difficult to ensure that all these combinations are covered in a course.

The present work uses an inductive approach to verify the observation of the high interrelatedness of object-orientated concepts. As part of this ambition, we have developed the TrucStudio tool. TrucStudio supports modeling arbitrary teaching domains, such as object-oriented programming, with its main concepts and relations between them. It also supports modeling courses based on these domain models; it visualizes the domain and course models and their dependency structure, and it provides a set of features that help analyze them. This addresses some of the difficulties associated to the complexity of the domain of object-oriented programming.

The TrucStudio domain models rely on the refined idea of Truc (Testable, Reusable Unit of Cognition) [85]. A Truc contains a set of descriptive elements such as a summary, role in the field, benefits and drawbacks, examples and common misconceptions. The set of Trucs on object-oriented programming developed as part of this thesis thus forms a pedagogical and technical knowledge base.

A fourth difficulty resulting from the subject’s complexity is that novices develop misconceptions about learned concepts. “It is well known
that students have such pervasive conceptual misunderstandings as novice programmers that correct programs early in the learning process come as pleasant surprises" [100]. Although a substantial amount of research has gone into developing an understanding of programming novices’ misconceptions, teaching material only rarely reflects this knowledge in practice. The Trucs on object-oriented programming summarize the findings on a per concept basis to ease lookup and complement it with derived recommendations for teaching material.

1.4 Contributions

The overall goal of this thesis is to advance the knowledge on the student diversity in introductory programming courses and on the interrelatedness of object-oriented concepts. This thesis analyzes the complexities and proposes techniques and tools that help handle them. Its main contributions are the analysis of a questionnaire tracking students’ backgrounds, the development of a method for modeling teaching domains and courses, a model of object-oriented programming containing its central concepts and relations, reusable descriptions of the concepts for educators, and the TrucStudio tool supporting the domain and course modeling process.

Analysis of student backgrounds. Since 2003, we have used a questionnaire in the ETH Introduction to Programming course to track to what extent the first-year undergraduate CS students are familiar with computers and how proficient they are at programming and programming languages. The data collected over the six years show that the characteristics of the student backgrounds are stable with respect to the computing and programming expertise, except for changes in the specific programming languages that students know. The results suggest that teachers can actually rely on certain assumptions regarding computer literacy and programming experience. To allow generalization of the results from ETH, students at the University of York filled in the same questionnaire in 2008. The data indicate that the student backgrounds with respect to prior programming expertise are similar at both institutions. Chapter 3 describes the questionnaire, reports on the outcomes, and suggests implications on instruction.

Method for educational domain modeling. As part of this thesis, we have developed a method for generating domain models based on the idea of Truc. The main intention of the model is to help analyze the structural properties of a teaching domain such as object-oriented programming and
to collect pedagogical knowledge for future use by other educators. Chapter 4 describes the Truc approach.

**Descriptions of the central object-oriented concepts including collected common misconceptions.** Design patterns capture expert knowledge of solutions to recurring problems in software architecture design. Trucs follow this line of thought and collect pedagogical and technical knowledge on concepts of a teaching domain. In particular, they capture common misunderstandings and misconceptions that novices may have when they encounter a certain topic for the first time. Section 5.2 provides a body of Trucs for object-oriented programming.

**Map of object-oriented programming.** To investigate the claim of high interrelatedness of object-oriented programming concepts, this thesis provides a map of the central concepts and their relations. The analysis of the map presented in Section 5.3 provides insights into circular dependencies among concepts and identifies a set of core Trucs. Furthermore, the map serves as the basis for modeling courses that teach the subject matter.

**Application to the ETH Introduction to Programming course.** The developed domain model serves as a basis to model the ETH Introduction to Programming course. Section 5.4 describes the results, identifies a set of notions that are missing in the course, and summarizes how well the teaching material addresses possible misconceptions of students.

**TrucStudio.** The TrucStudio tool supports the Truc methodology. It views the generation of Trucs as an engineering process similar to software development and provides visualization methods easing the analysis of domain and course models. One of its advanced features is the output generation mechanism that supports various output variants such as webpages, hypergraphs, and calendar files. Chapter 6 describes TrucStudio and its architecture.

**Roadmap**

The rest of this thesis is organized as follows: Chapter 2 describes the Introduction to Programming course held at ETH. This course and its participants were in the focus of our efforts towards improving knowledge on student backgrounds and the application of the Trucs. Chapter 3 describes the study on entering CS students’ prior knowledge of computing and
programming. Chapter 4 explains the basic concepts and relations of the TrucStudio domain and course modeling approach and Chapter 5 shows the application of the approach to object-oriented programming and the ETH Introduction to Programming course. Chapter 6 describes TrucStudio and details on the tool’s architecture. Finally, Chapter 7 lists directions for future work and Chapter 8 draws conclusions.
Chapter 2

The ETH Introduction to Programming course

This chapter describes the implementation of the Inverted Curriculum approach [82, 84, 103] in the Introduction to Programming course at ETH. This approach for teaching introductory programming emphasizes the reuse of existing components in an example domain involving graphics and multimedia; it is based on object-oriented, component-based technology with Design by Contract; it relies on the reuse of a large software framework built specifically for the course; and it follows an “outside-in” strategy where software construction depends, right from the start, on components.

The study on computing backgrounds described in Chapter 3 focuses on the students of the Introduction to Programming course and the case study for assessing the various components of TrucStudio and its underlying principles (see Chapter 5) is based on the course material of Introduction to Programming. This chapter thus sets the stage for the research presented in the following chapters. It describes the Inverted Curriculum and details on its principles. The rest of the chapter contains information on several aspects of the course: student numbers, its place in the curriculum, course schedule, covered knowledge units and learning goals, and the supporting software framework. It concludes with a description of feedback from students on the course. Parts of this chapter have already been presented in articles [103, 102].
2.1 Related approaches

Teaching introductory programming presents considerable challenges. Its discussion and the development of effective strategies addressing them has been ongoing since the 1970s [28, 54]. It is still a topic of lively discussions today [79, 39, 113, 133] and substantial research efforts go into the teaching of introductory programming (for example searching for articles that include “CS1” or “introductory programming” in ACM’s digital library produces 2,288 hits in total; 146 of these articles were published alone in 2009 [search done in July 2009]).

Many of the articles describe approaches to teaching introductory programming with a focus on the specific topics that an introductory course covers. The curricula recommendations, proposed by the ACM (American Association for Computing Machinery) and IEEE (Institute of Electrical and Electronics Engineers, Inc.) [131], distinguish between six implementation strategies for the introductory courses. Four of them focus on programming: algorithms-first, functional-first, imperative-first, and objects-first.

Courses that implement the algorithms-first approach introduce algorithmic concepts using pseudocode. This removes the need to handle the peculiarities of a programming syntax (thus they are also called syntax-free [39]). Although certain systems are available to compile and run fragments of pseudocode, students generally do not produce running programs, but they learn to reason about, explain, and trace algorithms by hand.

Implementations of the functional-first approach [13, 60, 2] use a functional language such as Scheme, Lisp, or Haskell and present recursion, linked data structures, and functions early on.

The imperative-first approach focuses on imperative programming aspects such as expressions, control structures, procedures, and functions. Many of the traditional programming courses follow this approach. Most of them use procedural programming languages, but it can also be implemented using an object-oriented programming language. When the introduction of object-oriented concepts is deferred to the end of the introductory sequence, then it is called an “objects-late” approach [15].

The objects-first or objects-early approach presents the notions of object, class, and inheritance immediately at the beginning of the introductory sequence. Since the mid 1990s, teaching object-orientation in CS1 has become mainstream [10] and many instructors have adopted the objects-first approach. In connection with the objects-first approach, the integration of object-oriented modeling techniques into the introductory programming course has received additional attention [3, 20, 9, 88].
2.1. RELATED APPROACHES

All four approaches have their benefits and drawbacks (for a discussion see the curriculum recommendations [131, pp. 29–33]), but the most controversial debate is on the superiority of the objects-first over the imperative-first approach and vice versa [128, 6]. “There is a fairly strong consensus that programming is hard both to teach and to learn, but the case that objects-early is harder (or easier) than objects-late has not yet been made conclusively” [75]. In particular, empirical studies supporting the arguments of either of the two positions are lacking [75] such that the choice of approach is mostly based on the individual position and experience of an instructor. The Introduction to Programming course for CS majors at ETH implements the Inverted Curriculum, an objects-first approach that illustrates the powers of the object-oriented method.

For all the approaches described above the choice of notation (pseudocode or programming language) is important. Many variants of notations are available. They range from programming languages specifically developed for teaching (such as Pascal [65] for structured programming or Blue [68] for object-oriented programming) to commercial programming languages like Java, C++, VisualBasic, Scheme, etc. While Pascal used to be the most widely spread programming language for instruction, the increase in popularity of object-orientation has led many instructors to adopt Java as programming language [49]. With the increase of courses on introductory programming with Java, the use of languages specifically developed for teaching has become rare. At ETH, we use Eiffel, a commercial language. Eiffel is particularly well suited for teaching because it is a purely object-orientated language with a clean syntax and concise constructs.

Besides variations in the focus and notation of an introductory computing course, courses also vary in their use of supporting software. Using programming microworlds started with the development of Logo [94] in the mid 1960s and Turtle graphics [1]. Since then, Turtle graphics have implementations in many languages and other programming microworlds such as Karel the Robot [99], Alice [20] and Kara [112] are widely used in introductory programming courses. Some of the benefits of using a microworld based on a physical metaphor are that it “focuses students’ attention, offers the opportunity to solve interesting problems even from the first lessons and contributes greatly to decreasing the ‘distance’ between the mental models or descriptions of algorithms in a natural language and their description in a programming language” [142]. Our implementation of the Inverted Curriculum approach uses Traffic, a microworld that models public transportation in a city.
2.2 The Inverted Curriculum

The Inverted Curriculum [103] is an objects-first, component-based approach for teaching introductory programming. It relies on a large software framework with a strong visual aspect. Using abstract interfaces, students initially take the role of clients of library components, before they progressively discover the implementations.

It is of course essential that students master all the traditional building blocks of programming: variables, assignment, and control structures. Where the Inverted Curriculum differs from most existing curricula is in the order of exposition. The use of components results in a sequence of topics that is outside-in, starting with what Wilkes called [136] the “outer” structure of the programming language: class interfaces, objects, features. Students then progressively move to the “inner” structure and learn about control structures, local variables, and assignment.

Using components has numerous advantages. By giving students access to high-quality software libraries, novices may take advantage of the functionality through their abstract interfaces, without needing to fully understand what is inside. The more advanced and intellectually curious ones can look inside the components, understand how they work, use them as guidance for their own goals, and potentially modify them.

Relying on the power of libraries also enables students to produce impressive applications from the start – even if initially these applications are really 10-line programs calling existing mechanisms. This serves to catch and retain students’ interest and to “make CS courses fun” [77] thus leading to success in learning [57, 35, 7].

While the Inverted Curriculum is an objects-first approach (rather than “functional-first” or “imperative-first”), it goes beyond “experimenting with [the notions of object and inheritance] in the context of simple interactive programs” [131]. The implementation at ETH uses Traffic [101], a software library that models public transportation of a city. Using Traffic students face, from the start, a large amount of software – much larger at least than anything that is commonly used in introductory courses. The Traffic library consists of over 130 classes with approximately 25,000 lines of code; including the supporting libraries EiffelBase, Gobo, Time, and EiffelVision2, it approaches the 250,000-line, 900-classes threshold.

Without the proper apparatus and method, beginning students (and also advanced ones) would drown in such an abundance of software. Modern techniques for reuse, such as information hiding, data abstraction and Design by Contract are essential here. The early exposition of students to large software frameworks enables us to ingrain key principles of reuse
and abstraction into students’ minds right from the beginning, teaching them that it is good to rely on solid existing solutions.

2.3 Course setup

The Introduction to Programming course for ETH computer science students has applied the Inverted Curriculum approach for six years. The number of students participating in the courses is approximately 1130 (250 in 2003, 180 in 2004, 170 in 2005, 160 in 2006, 170 in 2007, and 200 in 2008).

Introduction to Programming is the first computer science course at ETH and a key step for future computer science graduates on their way to a bachelor’s and a master’s degree. Offered in the very first semester as the only computer science course, it builds foundations for the rest of the studies. In the ETH tradition of providing a strong general science and engineering education to all students, the other courses are on linear algebra, analysis, and discrete mathematics.

Like many other introductory programming courses, Introduction to Programming faces particular challenges: it is very selective and it addresses students with very different backgrounds, some already quite experienced with programming, others having never programmed before. In a semester, it must bring all those who pass it to a level where they can cope with a variety of computer science subjects all relying on close familiarity with the concepts and practical skills of programming. For many students this course is a real challenge.

2.3.1 Schedule

The weekly schedule includes four (two times two) plenary lectures, held by the professor, and two exercise lessons, taught by graduate and doctoral student to groups of approximately 25 students. The duration of each lecture or lesson is 45 minutes.

The students receive weekly assignments from week 1 to week 9 including two sit-in assignments (simulating the exam). The sit-in assignments help students assess their current status and get a feeling on how they are performing compared to their classmates. For the years 2003-2007, students worked from week 10 to week 14 (semester end) in teams of two or three on a programming project. In 2008, due to changes in the semester start and end dates, additional weekly assignments replaced the project assignment. Tutors correct (but do not grade) all the weekly assignments, mock exams, and the project.
There is no grading during the course, only the requirement of doing the homework, sit-in assignments, and project – not necessarily successfully, but showing effort – to get a certificate allowing participation in the exam, held after the year. The exam then determines whether students are allowed to move into their second year of study. Tutors correct assignments and provide constant feedback to the students; but this has no effect on their final grades. This procedure enables the teachers to do their best teaching job without constant student obsession on grades, and students can take a reasoned, long-term approach to learning programming.

2.3.2 Covered knowledge areas and general learning goals

The course mostly covers programming fundamentals (PF) as described by the ACM Computing Curricula [131]. In addition, it introduces declarations and types (PL4) and teaches many object-oriented programming concepts (PL6) like abstraction, encapsulation, inheritance, polymorphism, and object-oriented design.

Besides this, the use of an existing software system allows a glimpse into software engineering concepts such as data abstraction, functional abstraction, information hiding, inheritance, and recursive reasoning.

The main goal of the course is that students obtain the knowledge of fundamental concepts of modern programming and operational skills for developing high-quality programs and working with large programs as in industry. This includes mastering the programming language and software engineering techniques mentioned above and developing problem-solving skills and a notion of the mode of algorithmic reasoning that makes it possible to solve problems through a computer. Since they are working with a large software system, they should also be able to understand and modify large systems, and be prepared for learning the techniques of building such systems.

2.3.3 Traffic and teaching material

One of the key points of the Inverted Curriculum is the use of high quality object-oriented software, made available to the students as a source of examples, models, problems, and challenges. Since the software must be tailored to the needs of this special form of teaching and since no adequate software existed, the team at the Chair of Software Engineering of ETH developed a first version, called Traffic, prior to the realization of the course in Fall 2003.

The application area of Traffic is public transportation in a city, which
provides many sources of examples and programming problems (such as planning a trip or moving passengers), it offers graphical and multimedia possibilities, and illustrates several algorithms (for example shortest path algorithms). Eiffel is used both as the programming language to be taught to students as well as for developing the course software.

The requirements on the design of such a system used by first semester students are very challenging to meet: The design must be easy to understand and immediately familiar to students so that they can extend and modify its components, but at the same time it must support fancy, advanced features like graphical user interfaces. It must provide a rich base for complex algorithms and data structures, and an open-ended source of examples and exercises. It must be well designed and implemented, with non-cryptic interfaces (GUI and API) and documentation.

The first version of Traffic consisted of a basic set of features and enabled the modeling of the metro line system of cities. It provided the basis for many exercises, but was very limited with respect to its application range. After having gained experience with a first set of students, we incorporated further extensions and improvements in the software. In particular, Traffic includes more transportation modes today (bus lines, railway lines), but also passengers, busses, trams, and buildings to realize a lively city.

Figure 2.1: Screenshot of Traffic
The Traffic software\(^1\) and the free EiffelStudio environment\(^2\) are available for download. An introductory programming textbook, “Touch of Class” [86], directly supports the course. Most of the lectures are close to some of the material from one of its chapters. Most of the examples from the book rely on the Traffic software. In the first lecture, for instance, we introduce the mechanism of feature call and argument passing on the following code sample [86]:

```eiffel
class PREVIEW
inherit TOURISM
feature explore_on_click
    -- Show city, highlight items, and animate a route.
    do
        Paris.display
        Louvre.spotlight
        Line8.highlight
        Route1.animate
        Console.show (Route1.origin)
    end
end
```

The first four calls result in the display of the Paris city on the screen (as in Figure 2.1), in marking the Louvre and Metro line 8, and in the animation of a figure, which moves along the predefined Route1. The last call outputs the name of the starting station of Route1 in a text widget.

All the assignments include tasks that use Traffic, but later in the course, we complement the Traffic exercises with tasks that result in small stand-alone applications. For instance, the assignment distributed after students learned to read class interfaces requires the modification of the metro system of Paris by making one of the lines circular and adapting other lines. In a later assignment on inheritance, one task extends the city with a new type of moving object. Another task in the same assignment asks students to implement a class `FRACTION`, which inherits from class `NUMERIC` and handles calculations with fractional numbers. This is an example of a stand-alone application.

All slides used in class are available on the Web, as well as exercises

\(^1\)http://se.ethz.ch/traffic
\(^2\)http://www.eiffel.com/download
and other material\(^3\). In addition, all lectures are recorded on video and put on the Web shortly after being presented.

### 2.4 Feedback on the course

Official end-of-semester evaluations conducted by the ETH administration and the Department of Computer Science collect students’ feedback on the course. The questionnaires cover general student satisfaction with the course, difficulty of the course, and any cross-cuttings with other courses. Table 2.1 shows the average grades of selected questionnaire items over the six years.

Table 2.1: Average grades (ranging from 1: totally disagree to 5: totally agree, unless indicated differently) of selected items from the department evaluation (n/a: not available)

<table>
<thead>
<tr>
<th>Questionnaire item</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall impression of the course (1: -, 2: -, 3: +, 4: +, 5: ++).</td>
<td>4.0</td>
<td>4.1</td>
<td>3.9</td>
<td>3.8</td>
<td>3.6</td>
<td>3.5</td>
</tr>
<tr>
<td>Lecture has no cross-cuttings with other classes.</td>
<td>4.5</td>
<td>4.7</td>
<td>4.6</td>
<td>4.6</td>
<td>4.4</td>
<td>4.2</td>
</tr>
<tr>
<td>Lecture had a clear concept.</td>
<td>3.8</td>
<td>4.1</td>
<td>3.6</td>
<td>3.8</td>
<td>3.6</td>
<td>3.5</td>
</tr>
<tr>
<td>Knowledge transfer resulted from lectures.</td>
<td>3.5</td>
<td>3.4</td>
<td>3.4</td>
<td>3.4</td>
<td>3.2</td>
<td>2.9</td>
</tr>
<tr>
<td>Knowledge transfer resulted from assignments.</td>
<td>3.8</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td>4.2</td>
</tr>
<tr>
<td>Assignments increased understanding of lectures and supported learning.</td>
<td>3.7</td>
<td>4.1</td>
<td>4.0</td>
<td>3.8</td>
<td>3.6</td>
<td>4.0</td>
</tr>
<tr>
<td>Assignments were well synchronized with lectures.</td>
<td>3.4</td>
<td>3.8</td>
<td>3.8</td>
<td>3.8</td>
<td>3.2</td>
<td>3.8</td>
</tr>
<tr>
<td>Liked working with predefined software.</td>
<td>2.7</td>
<td>2.9</td>
<td>2.5</td>
<td>2.6</td>
<td>2.9</td>
<td>3.5</td>
</tr>
<tr>
<td>Traffic was easy to understand.</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>2.6</td>
<td>2.6</td>
<td>3.4</td>
</tr>
</tbody>
</table>

\(^3\)http://se.ethz.ch/openaccess/eprog
The data indicates both positive and negative trends. On the positive side, several items referring to the practical assignments got increasing appreciation:

- Many students felt that knowledge transfer largely took place in connection with the assignments.
- They mostly agreed that the assignments augmented their understanding of the lectures and supported their learning.
- They found the assignments to be well synchronized with the lectures in terms of time and content.
- Additionally, students seem to have accepted working with a predefined software system and find Traffic understandable as indicated by satisfactory grades for the two items covering these aspects.

On the negative side, the average grade for the overall satisfaction with the course, which started out on a very high level with 4.0 (out of 5) points in 2003 and 4.1 in 2004, continuously dropped in the subsequent years to 3.5 in 2008. Together with the lower overall grade, grades for several other items, mostly relating to the lectures, also decreased:

- Students felt that the course had more cross-cuttings with other courses.
- They found that the course had a less clearly recognizable concept.
- They felt that less knowledge transfer took place in connection with the lectures.

The increase in grades for items relating to the practical assignments is probably due to a redesign of the assignments undertaken in 2008. The main goals of the redesign were: (1) to present Traffic in the assignments in a structured way so that students gradually discover the underlying class model and learn how to use and extend it, and (2) to improve synchronization of the assignments with the learning goals of the lectures. The increased grades referring to the items on the assignments are encouraging, since it shows that the students appreciate and value such efforts.

The software used for the course initially got a significantly lower appreciation than the course as a whole. In 2008, the average grades on using a predefined software system in general and Traffic in particular have
reached a satisfactory level. Since the Traffic software was not changed between the iterations of 2007 and 2008, it seems that the redesign of the assignments has influenced students’ appreciation of Traffic. This outcome indicates that the redesign of the assignments has reached its goals.

It is difficult to identify the concrete causes for the lower appreciation of the lectures in 2008. A possible explanation is that in the first two years the enthusiasm of the instructor about developing a new approach to teaching introductory programming influenced the students’ view of the course. Now, six years later, the enthusiasm and the novelty of the approach may have faded to a certain extent. It is also possible that students are more forgiving toward the deficiencies of a course in its first iterations than in later ones and that the improvements to the course did not keep up with students’ expectations. In any case, the course evaluation shows that we should focus our efforts on improving the lectures of the course in the coming semesters. The TrucStudio approach described in the following chapters is part of this intent.
CHAPTER 3

STUDENT BACKGROUNDS

It is difficult to think of another field than computer science in which, at the outset of the studies, instructors face heterogeneity in prior experience similar to what CS1 teachers handle – from students who have already worked as programmers in industry all the way to those who have no programming experience. The diversity of students’ prior exposure to programming constitutes one of the challenges of teaching introductory programming.

This chapter presents a study that investigates entering CS students’ prior experience in programming and computing. The study relies on a questionnaire that was issued in the Introduction to Programming course at ETH for the past six years in the first weeks of class. The collected data make it possible to track changes over the years and to explore the stability of the results. To investigate whether the results are transferrable to other institutions, entering students of the University of York filled in the same questionnaire. A comparison with ETH data identifies similarities and differences between the two universities.

The following sections show related work, describe the setup of the study, and discuss the participant groups. The chapter also contains a discussion of the limitations of the study and presents comparisons of the questionnaire results for the groups dissected by topic (computer experience, programming experience, and programming languages). It proposes several measures to adapt to the student diversity and summarizes the results. Partial results were already presented in [103, 104, 105].
3.1 Related work

A number of studies have provided information on students’ prior computing and programming knowledge. They yield some important insights for the present chapter, but in most cases, the issue of prior experience is subsidiary to the authors’ main interest rather than focus of attention as in this thesis.

Sometimes the main emphasis is on gender differences. For example, Madigan et. al. [76] describe differences in computer literacy. Sakrowitz and Parelius [117] and Fisher et. al. [40] show that women are less likely to have taken a programming course before starting to study CS and that their familiarity with a set of programming languages is lower than for their male colleagues.

In other cases the concentration is on computer literacy of general entering university students as in the study of Hoffmann and Vance [61], which gives an overview of the “technology tasks” that university students are able to handle and whom they learned it from.

Most studies report on prior programming knowledge as a possible predictor of success in a CS1 course. The study by Hagan and Markham [59] in 2000, for example, confirms prior programming experience to be at least initially beneficial for the students. The number of programming languages that the students previously encountered is systematically related to this outcome, but not the specific languages themselves. The most common programming languages that the participants encountered in previous courses are C/C++ and Pascal/Delphi.

Wilson’s study [137] in 2002 indicates that having attended a formal class in programming has a positive correlation with success in a course.

Another study in 2003 [115] reports Pascal as the most widely known programming language among students. In the setting of this study, experience with Pascal, C, and Assembler is a predictor for success, but not knowledge of other languages such as Prolog, C++, or Java.

Gomes and Mendes [50] performed a study in 2008 with participants from two Informatics Engineering courses. It shows the familiarity of students with the programming languages C, Pascal, Java, and Python. From these data, they find that previous programming knowledge is a predictor of success in their (objects-first) course.

Ventura and Ramamurthy’s study [135] in 2004 concludes that for their course (design-centered, objects-first, graphical) prior programming experience is no predictor of success.

The mentioned studies generally confirm prior programming knowledge as a predictor of success, but they differ in the specific aspects con-
considered relevant (such as number of languages or specific programming languages). Most of these studies also report on the particular programming languages that students know, but they do not provide data to investigate changes over time or differences between institutions in terms of prior computing and programming knowledge.

3.2 Study setup

Since Fall 2003, the first semester Computer Science majors at ETH attending the mandatory “Introduction to Programming” course have filled in a questionnaire in the first weeks of the semester. In the first iteration, the questionnaire was handed out on paper in class; in the following years it was available online.

The entering CS majors from University of York form the second group of participants. For these students, the data have only been collected in 2008 with a voluntary online questionnaire.

The data from ETH make it possible to track changes over the years and help assess the stability of the situation concerning their prior programming and computing expertise. The data from York provide the means to compare the results of two institutions in two different countries and thus constitute a first step towards generalization of the multi-year results from ETH.

3.2.1 Questionnaire on student backgrounds

The questionnaire changed only slightly over time to include items that better capture students’ answers. It contains several sections:

1. A personal code generated from a student’s name, birth date, place of birth, and his parents’ names, which makes it possible to relate other questionnaires issued during the course to this one.

2. General information including age and gender, focus of study and origin.

3. Prior education (high school or university degrees, focus subjects).

4. Computer literacy \(^1\) (access to computers at home, usage of computers).

\(^1\)In this thesis, we use the term “computer literacy” to denote students’ familiarity with hard- and software on a level such that they may do basic word processing, enter data into spreadsheets, and use electronic communication channels. This use of the term focuses on “competences”, rather than on “awareness” [78].
5. Prior programming experience.

6. Knowledge of specific programming languages.

Figure 3.1 shows the items on computer literacy asking students whether they have a computer at home and similarly whether they have a laptop. Other items requested them to rate for how long they have been using the computer and for what they use it. Questions on their familiarity with the operating systems Windows, Linux, Mac OS, and BSD complement the computer literacy section of the questionnaire. These items use a four-point scale where students state whether they know the operating system not at all, a little, well or very well.

Figure 3.1: Screenshot of items on computer literacy

The questionnaire section on programming experience contains items on programming expertise (general and with object-oriented programming), where they have learned it, and the size of their largest projects (general and object-oriented) and development team. Figure 3.2 shows these items.

The last part of the questionnaire collects data on the level of familiarity with various programming languages using a four-point scale (knows it not at all, a little, well, and very well).
Did you have programming experience before starting this course? (if no, skip the rest of the questions)

☐ Yes  ☐ No

Where did you learn your first programming language?

☐ I learnt it on my own
☐ At high school
☐ At university
☐ On the job
☐ Other: ____________________________

How many lines of code did the largest project have that you ever worked on?

☐ dozens
☐ hundreds
☐ thousands
☐ ten thousands and more

How large was the largest developer team that you ever worked for?

☐ I only worked alone
☐ 2-3 people
☐ 4-10 people
☐ 10-20 people
☐ more than 20 people

Have you programmed with an object-oriented language before starting this course? (if no, skip the next question)

☐ Yes  ☐ No

How many classes had the largest object-oriented project you ever worked on?

☐ dozens
☐ hundreds
☐ thousands
☐ ten thousands and more

Have you already had a job where you had to program?

☐ Yes  ☐ No

Figure 3.2: Screenshot of items on programming experience
3.2.2 Participants from ETH

At ETH, the students of six iterations of the course Introduction to Programming answered the questionnaire. Out of the approximately 1130 students who took the course over the years, 753 handed back the questionnaire. The percentage of female students in the course is between 5% and 15%.

Introduction to Programming is offered in the very first semester as the only computer science course and a required step for future computer science graduates on their way to a bachelor’s and possibly a master’s degree. The participants from ETH are CS majors (with a few exceptions) with a median age of 19, either new to the computer science study or repeating the first year because they have failed the final examination and retake Introduction to Programming as a preparation for their second try at the exams.

Most of the students who start a CS program at ETH come from one of the Swiss high schools where they graduated with the so-called “Maturity” degree. The Swiss high school system is decentralized: while a federal regulatory instrument sets general standards for the Maturity, each of the 26 cantons implements it with its own school laws. The Maturity system is selective – in 2008 less than 32% of all young permanent residents of Switzerland got a degree that allows them to study at a Swiss university [31]. In the computing area, most high schools offer introductory courses on computer applications (text processing, spreadsheet programming, web surfing), but very few teach computer science, or programming using a higher-level language. Until 2007, the Swiss high school regulations did not mention computer science; it recently became an optional supplementary subject, and implementation started in Fall 2008. It will be interesting to see how this affects the backgrounds of CS students.

3.2.3 Participants from University of York

All entering CS students at University of York (86 CS majors and 25 CS and Math majors) received the invitation to the questionnaire at the beginning of the semester. Out of the 101 students, 77 answered the call for participation.

Like at ETH, most of the CS students at University of York enter directly from high school with a median age of 18. This is one year earlier than for ETH where the students go through a longer secondary education.

The British high school system has similar characteristics as the Swiss
system concerning computing and computer science at high schools: there are no national regulations that require the schools to offer programming courses. The University of York has a selective admission process, but does not take prior computing and programming experience into account. Having taken a course on programming is therefore not a requirement for these students.

The percentage of female students at York is 9% – the same as for the ETH students in 2008.

### 3.3 Threats to validity and limitations

While we believe that many of the conclusions apply to the teaching of computer science anywhere, a number of specifics may limit generalization.

- The Swiss practice of selective high schools and the admission process at University of York, which screen the incoming students, may bias the sample of surveyed students towards higher competence.
- The absence of computer science in Swiss and British high schools (as opposed to many other countries) may bias the results in the reverse direction.
- Another threat to validity of the survey is that it does not measure students’ prior experience objectively, but through their own self-appraisal. We do not know if this introduces a bias, and if so in what direction.
- Finally, the switch from a paper questionnaire, filled out in class at ETH in 2003, to a voluntary online form caused a decrease in participation. This introduces the risk of self-selection, another possible limitation. This threat to validity also applies to the data coming from the York participants.

To minimize the risk of having an unrepresentative sample of students as participants to the questionnaire, we ask ETH students to rate their prior programming expertise again in the official end-of-semester course evaluation questionnaire required and administered by the university and handed out on paper during class. The results of that second test essentially coincide with the initial results, with the exception of a punctual discrepancy (23% of novices from the university questionnaire vs. 13%) for one single year, 2007.
An obvious potential limitation of this work is that it is mainly based on results from one institution. Although we cannot authoritatively claim generalization to other universities and countries, the results from the student group at University of York are very similar to the results at ETH and provide a first indication that there are institutions with similar characteristics.

3.4 Statistical tests

The collected questionnaires come from seven populations – six classes of students taking Introduction to Programming at ETH in the years of 2003 until 2008 and one class from York in 2008. Based on these data, we can identify differences between the York and the ETH population and we can analyze changes over the years at ETH.

For the comparison of the ETH students to those of York, we apply Mann-Whitney-U tests [38, pp. 540–551] to the data of 2008. The Mann-Whitney test is a non-parametric test that makes it possible to identify significant differences between two independent samples (equivalent to the parametric t-test for independent samples, but also applicable to ordinal variables). To report the test results we use the scheme $U = ..., z = ..., r = ..., p < 0.05$, if the difference is significant at the 0.05-level. This means that a difference between the data of York and of ETH students is a chance finding with a probability of less than 5%. For non-significant results the scheme is $U = ..., z = ..., ns$. $U$ is the test statistic of the Mann-Whitney test, $z$ gives the $z$-score, and $r$ reports the effect size. The effect size is an “objective and (usually) standardized measure of the magnitude of an observed effect” [38, p. 785]. This study uses Pearson’s correlation coefficient $r$ as measure for the effect size. We consider $r = 0.10$ a small effect, $r = 0.30$ a medium effect, and $r = 0.50$ a large effect.

To identify significant changes over the years at ETH, the analysis uses the Kruskall-Wallis-H test [38, pp. 559–571]. This non-parametric test makes it possible to identify significant differences between multiple independent samples. To report the Kruskall-Wallis test, we give the test statistic $H$, its degrees of freedom $df$ and its significance $p$, $H(df) = ..., p < 0.05$, or $H(df) = ..., ns$, if it is non-significant.

The Kruskall-Wallis test only determines whether there are differences between the groups, but it does not indicate which groups are responsible for a significant outcome. To identify more precisely, which populations exhibit differences, we run Mann-Whitney-U tests as post hoc procedures on each pair of data sets. Using six groups, this results in 15 Mann-
Whitney tests as post hoc tests. We apply the Bonferroni-Holm correction [116, pp. 337–339] to keep the Type I error rate (false positives) down at the 5% level. The post hoc tests use the scheme of the Mann-Whitney tests for reporting.

3.5 Computer literacy

Without knowing how to use a computer it is extremely difficult even to consider learning how to program. This is usually the first concern of a CS1 educator. In our setup, Figure 3.3 shows that this concern is no longer justified.

![Figure 3.3: Time during which students have used computers](image)

The figure confirms that students entering a computer science study are computer-literate. In fact, the majority has used computers for ten years or more. The data of 2008 from ETH and York do not differ significantly in this respect, $U = 4670, z = -0.30, ns$. With a median age of around 19 for ETH and 18 for York students, the computer has been part of their life for at least half of it.

The application of the Kruskall-Wallis test indicates that there exists a significant difference between the six classes of Introduction to Programming for the time spent with computers, $H(5) = 36.89, p < 0.05$. The post hoc tests show that this difference is mostly due to the 2003 population, which had significantly less exposure to the computer than did the students of the following years 2004-2008. This outcome repeats for most of
the items of the questionnaire: if a significant difference exists between all the groups according to the Kruskall-Wallis test, then it is due to differences between the class of 2003 and the later classes. In the rest of the discussion, we will only report significant differences if they involve other years than 2003.

Consistently with the finding that the class of 2003 has less experience with the computer, the percentage of ETH students who have a desktop computer at home has risen from 87% in 2003 to percentages between 95% and 98% in the following years (see Table 3.1). Similarly, the percentages of students who own a laptop increased from 56% in 2003 to 75%-92% in the next years. The differences between the years are significant for desktop computers, $H(5) = 25.97, p < 0.05$, and for laptops, $H(5) = 69.91, p < 0.05$. The class of 2003, compared to the later classes, had significantly less access to desktop computers at home and a smaller portion of its students owned a laptop. Additionally, the number of students who own a laptop is significantly lower for the class of 2007 than for the students of the previous year, $U = 3501, z = -3.12, r = -0.23, p < 0.05$.

For York students, the situation is similar as for the ETH students in 2008: almost 99% of them have access to a computer at home and 90% own a laptop. These numbers indicate that students have similar access to technology at both institutions with respect to desktop computer, $U = 4566, z = -1.34, ns$, and laptops, $U = 4496, z = -1.02, ns$.

Table 3.1: Access to home computer and laptop

<table>
<thead>
<tr>
<th>institution &amp; year</th>
<th>computer</th>
<th>laptop</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>87%</td>
<td>56%</td>
</tr>
<tr>
<td>2004</td>
<td>98%</td>
<td>83%</td>
</tr>
<tr>
<td>2005</td>
<td>98%</td>
<td>81%</td>
</tr>
<tr>
<td>2006</td>
<td>96%</td>
<td>92%</td>
</tr>
<tr>
<td>2007</td>
<td>97%</td>
<td>75%</td>
</tr>
<tr>
<td>2008</td>
<td>95%</td>
<td>85%</td>
</tr>
<tr>
<td>York</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>99%</td>
<td>90%</td>
</tr>
</tbody>
</table>

Figure 3.4 shows students’ familiarity with the Windows, Linux, Mac, and BSD operating systems. Note that the data for BSD have only been collected in the years 2005-2008. Windows is the operating system that
### 3.5. COMPUTER LITERACY

#### Figure 3.4: Students’ familiarity with operating systems

<table>
<thead>
<tr>
<th>Year</th>
<th>FreeBSD (BSD)</th>
<th>Mac OS</th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>9%</td>
<td>31%</td>
<td>21%</td>
<td>68%</td>
</tr>
<tr>
<td>2007</td>
<td>9%</td>
<td>31%</td>
<td>21%</td>
<td>68%</td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>FreeBSD (BSD)</th>
<th>Mac OS</th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>16%</td>
<td>35%</td>
<td>20%</td>
<td>62%</td>
</tr>
<tr>
<td>2007</td>
<td>17%</td>
<td>35%</td>
<td>20%</td>
<td>62%</td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>FreeBSD (BSD)</th>
<th>Mac OS</th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>12%</td>
<td>37%</td>
<td>20%</td>
<td>64%</td>
</tr>
<tr>
<td>2007</td>
<td>12%</td>
<td>37%</td>
<td>20%</td>
<td>64%</td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>FreeBSD (BSD)</th>
<th>Mac OS</th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2007</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>FreeBSD (BSD)</th>
<th>Mac OS</th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2007</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>FreeBSD (BSD)</th>
<th>Mac OS</th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2007</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>FreeBSD (BSD)</th>
<th>Mac OS</th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2007</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>FreeBSD (BSD)</th>
<th>Mac OS</th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2007</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>FreeBSD (BSD)</th>
<th>Mac OS</th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2007</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>FreeBSD (BSD)</th>
<th>Mac OS</th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2007</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>FreeBSD (BSD)</th>
<th>Mac OS</th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2007</td>
<td>4%</td>
<td>40%</td>
<td>37%</td>
<td>55%</td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
most students know (around 97%-100%), followed by the Linux variants (60%-80%) and Mac OS (45%-60%). The percentage of students who state very high familiarity with Windows has been stable at around 60% over all the surveyed years, $H(5) = 2.41, ns$. The students at York have significantly less experience with the operating system BSD than the ETH students, $U = 4131, z = -2.16, r = -0.15, p < 0.05$, but the effect size is only small.

Concerning the specific tasks that students use the computer for, their main focus is on web-related tasks (see Table 3.2). All of them surf the web and write e-mails. Almost all of them also use the computer for text processing. About three quarters of all students play games and program, while about half of them design web pages, produce graphics, and do system administration and maintenance. There are no significant differences between the years 2005-2008 at ETH except for significantly less students doing web design in 2008 than in 2006, $U = 4832, z = -2.22, r = -0.15, p < 0.05$. ETH students use the computer significantly more for text processing, $U = 4409, z = -2.07, r = -0.15, p < 0.05$, and less for playing games, $U = 4055, z = -2.40, r = -0.17, p < 0.05$, than the York students. Note that the data for 2004 are missing.

Table 3.2: Technology tasks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>surf the web</td>
<td>98%</td>
<td>100%</td>
<td>100%</td>
<td>99%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>write e-mails</td>
<td>98%</td>
<td>99%</td>
<td>100%</td>
<td>97%</td>
<td>99%</td>
<td>100%</td>
</tr>
<tr>
<td>text processing</td>
<td>88%</td>
<td>96%</td>
<td>98%</td>
<td>94%</td>
<td>96%</td>
<td>88%</td>
</tr>
<tr>
<td>play games</td>
<td>75%</td>
<td>74%</td>
<td>70%</td>
<td>73%</td>
<td>69%</td>
<td>84%</td>
</tr>
<tr>
<td>graphics</td>
<td>44%</td>
<td>57%</td>
<td>68%</td>
<td>58%</td>
<td>53%</td>
<td>46%</td>
</tr>
<tr>
<td>web design</td>
<td>47%</td>
<td>61%</td>
<td>68%</td>
<td>63%</td>
<td>48%</td>
<td>42%</td>
</tr>
<tr>
<td>programming</td>
<td>65%</td>
<td>78%</td>
<td>79%</td>
<td>84%</td>
<td>77%</td>
<td>68%</td>
</tr>
<tr>
<td>system administration</td>
<td>43%</td>
<td>48%</td>
<td>53%</td>
<td>43%</td>
<td>50%</td>
<td>39%</td>
</tr>
</tbody>
</table>

Generally, it seems that today’s entering CS majors are computer-literate and use the computer regularly. The ETH and York students show only marginal differences with respect to their familiarity with the BSD operating system and their usage of the computer to play games and do text processing.
3.5.1 Interpretation

The significant differences between the class of 2003 and later classes indicate that the students of 2003 have been less exposed to computing and programming than later ones. This could be related to the increased spread of technology, which has reached the population of all ages. Today, most people use computers daily to read e-mails, surf the web and build up communities. Moreover, people attracted to computer science programs usually exhibit a strong interest in new media and technology.

Another explanation for the differences between 2003 and later years is a change in the Swiss high school regulations that shortened the duration of secondary education in certain cantons of Switzerland in the years of 2000-2003. This may have had an effect on the student body of 2003.

3.5.2 Teaching implications

The surveys do not indicate how deeply students understand the concepts behind computers and computer architecture. But the immediate lesson for CS1 instructors is that they do not need to fret about “computer literacy”. Students are familiar with computers, and instructors can go straight into programming if this is the goal of the course.

3.6 Programming experience

Table 3.3 shows the programming experience of students, broken down into the categories “no programming” (never programmed before), “no OO” (programmed, but never with an object-oriented language), “small project” (worked on object-oriented projects consisting of less than a hundred classes) and “large project” (worked on object-oriented projects with hundreds of classes – a sizable experience for supposed novices).

The number of female students participating in the questionnaire varied between 5% and 15%, reflecting the actual ratio of female first semester CS students. The table indicates that the figures do not differ markedly between the genders, except for the higher number of total beginners among females; however, the low number of females in the samples prevents any statistically significant conclusions on gender differences.

Figure 3.5 visualizes the results of Table 3.3 for students of both genders. It indicates that an increasing subset of the students starts with experience in object-oriented programming, while the percentage of those with no object-oriented programming experience has dropped. These changes over the years are only significant when taking into account the students.
Table 3.3: Previous programming experience of students (*: ≥ 100 classes)

<table>
<thead>
<tr>
<th>institution, year, gender</th>
<th>student numbers</th>
<th>no programming</th>
<th>no OO</th>
<th>some OO</th>
<th>some experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>222</td>
<td>22%</td>
<td>39%</td>
<td>34%</td>
<td>5%</td>
</tr>
<tr>
<td>male</td>
<td>203 (91%)</td>
<td>19%</td>
<td>39%</td>
<td>37%</td>
<td>5%</td>
</tr>
<tr>
<td>female</td>
<td>19 (9%)</td>
<td>53%</td>
<td>42%</td>
<td>5%</td>
<td>0%</td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>127</td>
<td>14%</td>
<td>33%</td>
<td>43%</td>
<td>10%</td>
</tr>
<tr>
<td>male</td>
<td>117 (92%)</td>
<td>11%</td>
<td>34%</td>
<td>44%</td>
<td>11%</td>
</tr>
<tr>
<td>female</td>
<td>10 (8%)</td>
<td>50%</td>
<td>20%</td>
<td>30%</td>
<td>0%</td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>95</td>
<td>18%</td>
<td>25%</td>
<td>42%</td>
<td>15%</td>
</tr>
<tr>
<td>male</td>
<td>81 (85%)</td>
<td>12%</td>
<td>26%</td>
<td>46%</td>
<td>16%</td>
</tr>
<tr>
<td>female</td>
<td>14 (15%)</td>
<td>50%</td>
<td>22%</td>
<td>21%</td>
<td>7%</td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>97</td>
<td>19%</td>
<td>27%</td>
<td>43%</td>
<td>11%</td>
</tr>
<tr>
<td>male</td>
<td>84 (87%)</td>
<td>18%</td>
<td>25%</td>
<td>44%</td>
<td>13%</td>
</tr>
<tr>
<td>female</td>
<td>13 (13%)</td>
<td>23%</td>
<td>39%</td>
<td>38%</td>
<td>0%</td>
</tr>
<tr>
<td>2007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>88</td>
<td>13%</td>
<td>20%</td>
<td>59%</td>
<td>8%</td>
</tr>
<tr>
<td>male</td>
<td>84 (95%)</td>
<td>13%</td>
<td>19%</td>
<td>60%</td>
<td>8%</td>
</tr>
<tr>
<td>female</td>
<td>4 (5%)</td>
<td>0%</td>
<td>50%</td>
<td>50%</td>
<td>0%</td>
</tr>
<tr>
<td>2008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>124</td>
<td>18%</td>
<td>22%</td>
<td>43%</td>
<td>17%</td>
</tr>
<tr>
<td>male</td>
<td>113 (91%)</td>
<td>16%</td>
<td>22%</td>
<td>45%</td>
<td>17%</td>
</tr>
<tr>
<td>female</td>
<td>11 (9%)</td>
<td>46%</td>
<td>18%</td>
<td>18%</td>
<td>18%</td>
</tr>
<tr>
<td>York</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>77</td>
<td>21%</td>
<td>27%</td>
<td>47%</td>
<td>5%</td>
</tr>
<tr>
<td>male</td>
<td>70 (91%)</td>
<td>19%</td>
<td>30%</td>
<td>46%</td>
<td>6%</td>
</tr>
<tr>
<td>female</td>
<td>7 (9%)</td>
<td>43%</td>
<td>0%</td>
<td>57%</td>
<td>0%</td>
</tr>
</tbody>
</table>
of 2003, $H(5) = 24.56, p < 0.05$. The differences between the students of 2004 to 2008 are non-significant, $H(4) = 2.52, ns$.

![Figure 3.5: Programming experience](image)

There is no identifiable change of trend over the years in the number of novices, which hovers around one-sixth to one-fifth of each class.

The distribution of prior programming expertise of the York students is consistent with the ETH data. There seems to exist a general scheme of around 20% novices, between 20-30% of non-OOP programmers, and around 40%-50% with experience on small object-oriented programs. For York, the number of students who have programmed large object-oriented applications is smaller than for ETH, but the Mann-Whitney test shows no significant differences for the two groups, $U = 4159, z = -1.63, ns$.

Another item on the questionnaires asked students where they have learned to program (see Figure 3.6). On average over the six years, 56% of the ETH students stated that they learned programming by themselves; 18% are novices; only 18% took a programming course at high school and the remaining 8% learned it at university, at work or on another occasion (such as courses at an evening school).

Similarly to the ETH students, most students at York learned programming in self-study (45%); 26% took a programming course at high school, and 8% learned it at another occasion. Compared to the ETH students, 8% more students have studied programming at high school, but, as for the ETH population, the most frequent case is having it learned in self-study. Statistical tests did not find a significant difference between the two populations, $U = 2755, z = -1.51, ns$. 
A possible reason for the lower exposure to programming of the class surveyed in 2003 could be that this was still just “after the Internet bubble burst”, after which more students have been attracted to computer science by genuine interest. Observations that would seem to support such a hypothesis include: the highest percentage of novice programmers (22% including both genders) for the year of 2003; the above-average numbers of CS enrollments in that year (although an alternative explanation for that particular phenomenon could be a change that occurred in the Swiss high school system); and our own informal observation that students in subsequent years seemed more genuinely interested in CS.

The growth in object-oriented language experience is probably due to the increasing spread of object-oriented languages such as Java (see also Section 3.7).

Many students have prior programming experience when entering university. Interestingly, most of them learned it in self-study; only one fourth to one sixth of them have taken a programming course at high school. Although partly influenced by the absence of mandatory programming courses at high schools, this result shows how broadly part of computer science has reached some of the world at large, particularly the younger segments of the population.
3.7. PROGRAMMING LANGUAGES

3.6.2 Teaching implications

The evidence on prior object-oriented language use indicates that today object-orientation is a given and needs no particular apology or justification. Tempering this lesson coming from the questionnaire data is a more subjective observation from our informal interactions with students: many do not fully grasp the more sophisticated properties of object technology, such as polymorphism, dynamic binding and other architectural techniques. It seems more useful to explain these concepts in depth than to take pains to justify the use of objects.

Another important conclusion arises from studying the other end of the data: the persistence of the “no prior programming” 15%-20% minority. It raises significant challenges for teachers, especially when assessed against the only slightly lower percentage of those who have programmed fairly large object-oriented systems. The variety of prior programming expertise is, in our experience, one of the largest obstacles facing introductory programming teaching today.

3.7 Programming languages

As part of the questionnaire, students rated 15 programming languages (ranging from Java, PHP and C++ to Fortran, Eiffel and Python; for a full list see Figure 3.8) whether they know it not at all, a little, well or very well. The answers to these questions (Table 3.4) reveal that a typical student of any of the two institutions knows – in his or her self-evaluation – two to three of the languages a little and at least one of the languages well.

As to the number of programming languages students know well or very well, Figure 3.7 confirms that almost half of the current students have sound proficiency in two or more languages and that at least one third of all students have not really mastered any of the languages (the numbers include the students who stated being novice programmers). These figures did not change significantly for ETH students in 2004 to 2008. There are more students who claim to know only one language well or very well at York than at ETH; at ETH, the percentage of students who claim to know three or more languages well or very well is higher than at York. These differences, however, are not statistically significant, $U = .92, z = -.10, ns$.

Questionnaire items on the level of familiarity of the 15 programming languages help answer additional questions: (1) What are the most widely known languages among them? (2) Are there languages with growing or dropping popularity with this particular population?
Table 3.4: Average (and median) number of languages known

<table>
<thead>
<tr>
<th>institution &amp; year</th>
<th>a little</th>
<th>well</th>
<th>very well</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>1.8 (2)</td>
<td>1.0 (1)</td>
<td>0.2 (0)</td>
</tr>
<tr>
<td>2004</td>
<td>3.2 (2)</td>
<td>1.1 (1)</td>
<td>0.6 (0)</td>
</tr>
<tr>
<td>2005</td>
<td>3.2 (3)</td>
<td>1.4 (1)</td>
<td>0.6 (0)</td>
</tr>
<tr>
<td>2006</td>
<td>2.8 (3)</td>
<td>1.2 (1)</td>
<td>0.7 (0)</td>
</tr>
<tr>
<td>2007</td>
<td>2.6 (2)</td>
<td>1.3 (1)</td>
<td>0.6 (0)</td>
</tr>
<tr>
<td>2008</td>
<td>2.4 (2)</td>
<td>1.2 (1)</td>
<td>0.5 (0)</td>
</tr>
<tr>
<td>York</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>2.0 (2)</td>
<td>1.0 (1)</td>
<td>0.5 (0)</td>
</tr>
</tbody>
</table>

Figure 3.7: Number of languages known well or very well
Figure 3.8 shows the 15 programming languages and the percentages of students with the four levels of familiarity (knowing the language in question not at all, a little, well and very well). Some of the languages, marked * and **, were only included in the survey after the first iterations. The analysis takes into consideration the answers from all students (including programming novices).

Taken over all years for ETH students, the web scripting language PHP is the most popular, having both the highest number of students who state they know it very well and the fewest students who do not know it at all. Other popular languages are C/C++, Java/JavaScript, and Basic/VisualBasic.

The top three languages (i.e. the languages where the least ETH students state that they don’t know it at all), separated by year, include most of the languages rated as most known totaled over the years (see Table 3.5 and Figure 3.8). C++ is an evergreen – it appears almost every year in the list of the three top languages. Since 2005, Java, JavaScript and PHP also strengthened their position and for the last two years around 50% of all students have worked with PHP, JavaScript, and/or Java before starting to study CS.

VisualBasic is the top language at York followed by the scripting languages JavaScript and PHP (see Table 3.5). Indeed, while ETH and York students do not exhibit significant differences in their general programming backgrounds, they differ significantly in some of the specific programming languages that they worked with before entering university.

The programming languages in Figure 3.8 marked with ˆ differ significantly for the two populations. Examples of such languages include VisualBasic, $U = 3590, z = -2.84, r = -0.20, p < 0.05$, and Java, $U = 3610, z = -2.61, r = -0.19, p < 0.05$.

The data of ETH collected over multiple years make it possible to analyze changes in the popularity of programming languages. The most popular programming languages such as PHP and JavaScript belong to the list of programming languages with increasing popularity amongst the students. Figure 3.9 shows the most popular programming languages that have a rising tendency in the percentage of ETH students who state to know it a little, well, or very well, i.e. the percentage of ETH students who do not know the programming language at all has decreased since 2003. The programming languages Basic, Pascal, and VisualBasic exhibit a decreasing trend in ETH students’ level of familiarity. However, most of these results are not significant. The Kruskall-Wallis test applied to the ETH populations of 2003-2008 exhibits only two languages with significant differences involving other years than 2003. In 2008, significantly less
Figure 3.8: Familiarity of students with programming languages
* marks missing data of 2003
** marks missing data of 2003-2005
^ marks significant differences between York and ETH students
3.7. PROGRAMMING LANGUAGES

Table 3.5: Most popular programming languages

<table>
<thead>
<tr>
<th>institution &amp; year</th>
<th>1st place</th>
<th>2nd place</th>
<th>3rd place</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETH</td>
<td>Basic</td>
<td>Pascal</td>
<td>C++</td>
</tr>
<tr>
<td>2003</td>
<td>Basic</td>
<td>Pascal</td>
<td>C++</td>
</tr>
<tr>
<td>2004</td>
<td>Eiffel</td>
<td>C++</td>
<td>JavaScript</td>
</tr>
<tr>
<td>2005</td>
<td>C++</td>
<td>Java</td>
<td>PHP</td>
</tr>
<tr>
<td>2006</td>
<td>PHP</td>
<td>JavaScript</td>
<td>Java</td>
</tr>
<tr>
<td>2007</td>
<td>PHP</td>
<td>Java</td>
<td>JavaScript/C++</td>
</tr>
<tr>
<td>2008</td>
<td>PHP</td>
<td>C++</td>
<td>C</td>
</tr>
<tr>
<td>York</td>
<td>VisualBasic</td>
<td>JavaScript</td>
<td>PHP</td>
</tr>
<tr>
<td>2008</td>
<td>VisualBasic</td>
<td>JavaScript</td>
<td>PHP</td>
</tr>
</tbody>
</table>

students knew Basic than in 2005, \( U = 4302, z = -3.21, r = -0.22, p < 0.05 \). The students of 2004 and 2005 generally state to have significantly more experience with Eiffel than the students of all the other years, \( H(5) = 211.46, p < 0.05 \).

3.7.1 Interpretation

The popularity of languages such as JavaScript and PHP most likely reflects that many students’ prior experience has been with web applications. This fits into the observation that today’s students are very web-oriented using the computer for writing e-mails and surfing on the web.

Student backgrounds of the two populations at York and ETH only
differ with respect to the specific programming languages that are best known (such as Java and VisualBasic). This could indicate that the popularity of programming languages is location-specific and possibly due to the varying spread of the programming languages at the two countries’ high schools. But at this point, the collected data are too sparse to verify this conjecture.

Note that our results are limited to the 15 languages itemized in the questionnaire: a student may know additional languages.

3.7.2 Teaching implications

These results show that, when teaching introductory programming, we need to take into account that the number of students who need to learn programming almost from scratch is higher than the 10% to 20% who have never programmed before.

In particular, it may well be that students whose programming has mostly been with Web applications in PHP or JavaScript are adept at writing user interface operations, but only have superficial experience with loops, recursion, data structures and other standard computer science techniques. While the questionnaire does not test this conjecture, it is definitely supported by informal observations. If it is correct, we should not consider that proficiency at GUI and Web programming implies proficiency at concepts and skills of professional software development, meaning that we need to take extra care with the teaching of fundamental topics.

3.8 Effect on teaching

The analysis of the questionnaire on student backgrounds illustrates the full spectrum of student backgrounds that CS1 instructors face. This section presents measures proposed to adapt to such a student body.

Adapting the course material. As a first and simple option, if we want students with prior knowledge to understand courses better, we must connect to that knowledge. This can help adapt the course to students’ needs; when introducing a concept, for example, instructors can provide references to its counterpart in the most known programming languages. They may consider going further and organizing special exercise groups for students with in-depth experience with a specific programming language.
Adapting the teaching methodology. Because the majority of CS1 students already know a programming language, it seems more natural to offer access to the whole libraries and to a complex development environment, thus letting the more curious students explore a richer environment. This is the technique used in the Inverted Curriculum approach [103] (see Section 2.2). While more novice students content themselves with the library’s APIs, their advanced colleagues may explore the library’s internals, discover the more advanced aspects, and enhance their competence through imitation and inspiration.

Another possible measure to tackle differences in prior programming experience is the use of a programming language that only few students have previously encountered in CS1. Interestingly, both institutions use such a language to “level students out” and not favor one student over another. While this is not the full solution to the problem of student diversity, it seems to be a reasonable way of ensuring that all students have a fair treatment.

Offering extra lessons. Students who had learned a programming language prior to the CS1 course are, overall, more successful than novices [59, 115]. It is likely that the extra experience with other programming languages provides intellectual preparation for mastering the intricacies of software development, for which novices enjoy no counterpart. To redress this imbalance, it may be interesting to allow novice students to take extra lessons on programming either before the semester starts (such as in a CS0 course [27]) or during the semester.

Making student groups. The differences in students’ prior programming expertise justify the discussion of offering two or more courses targeted to the various competence levels. This would require the development of competence models and associated programming language independent tests, which guarantee objective assessment of prior knowledge. The competence model for object interaction developed by Bennedsen and Schulte [11] could serve as a starting point. It describes a taxonomy consisting of four levels. The levels range from understanding simple object interactions (such as feature calls and object creation) between a few objects on the lowest level to understanding the effects of inheritance and dynamic binding on dynamic polymorphic object structures on the highest level. The course for advanced programmers could then build upon existing previous knowledge and cover ancillary material while the course for novices could take a slower approach in introducing new concepts. One of the main issues with such a solution is probably scarceness of resources.
and increased costs. A reduced solution could split up the course partially (for example only for the lab or tutoring sessions such that certain parts can be adapted to previous knowledge).

**Individualized instruction.** Going one step further, individualized instruction seems well suited to handle the student diversity. One of the main characteristics of individualized instruction is that students proceed at their own pace, so that the differences in prior knowledge can be handled. Individualized instruction is related to other teaching methods such as programmed instruction, mastery learning, self-controlled learning, and computer-assisted instruction. Several variants of individualized instruction find applications in teaching introductory programming [32, 33, 53].

### 3.9 Summary

The data presented in this chapter show that the entering CS students at ETH and those at University of York start with very similar backgrounds, in particular concerning prior programming experience, and the number of programming languages that they know. They only differ in the specific programming languages that they know and single items concerning computer literacy.

There are several possible factors for this similarity. First, courses on computer science are optional in the high school systems of both countries. In addition, ETH and University of York are both competitive and recruit nationwide. Third, neither ETH nor University of York requires computer science or programming expertise for admission. While the University of York has a selective admission process based on other criteria such as Math grades, ETH accepts anybody with a Swiss Maturity degree. The selection at ETH happens through the selective Swiss high schools, ETH’s reputation as a top university, and the first year exams.

The outcome suggests that generalization of the results is possible, but may be limited to other universities of countries with similar regulations for admission and for high school courses on computer science.

The statistical tests used to identify changes over the years found that the student body of 2003 significantly differs from the later classes. If the data set of 2003 is ignored, then the situation appears stable with the exception of single pointed changes in the percentage of students owning a laptop, the portion of students using the computer for web design, and their knowledge of Basic. The only unstable item over the years is on the knowledge of Eiffel. This confirms that the introductory programming
course at the two institutions has been and – given the mostly stable situation – will be faced with a very diverse student body.

At one end, a considerable fraction of students have no prior programming experience at all (between 13% and 22%) or only moderate knowledge of some of the cited languages (around 30%). The evidence so far does not suggest a decrease in either of these phenomena.

At the other end, the course faces a large portion of students with expertise in multiple programming languages (around 30% know more than three languages in depth). In fact, many have worked in a job where programming was a substantial part (24% in 2003, 30% in 2004, 26% in 2005, 35% in 2006, 31% in 2007 and 2008).

An increasing percentage of students who have programming experience used an object-oriented language; correspondingly, fewer students take the course without prior O-O exposure.

If we try to picture the typical entering CS student at any of the two institutions, he (being typical, the student is most likely a “he”) is between 18 and 20 years old and knows one programming language in depth and another two to three languages slightly. He has a computer with the Windows operating system at home and most likely a laptop. He has a long experience with computers and uses it mostly for web surfing, writing e-mails and text processing. He has learned programming by himself and uses VisualBasic, Java, C++, C, or a web-related programming language such as PHP or JavaScript.
Teaching is a highly personal endeavor; the human touch is essential. A course is not an engineering product, and will never be specified as precisely and rigorously as, for example, a computer program. Still, applying modeling techniques partly imitated from software and other engineering disciplines can help meet some of the challenges of course design, in particular for object-oriented programming, a domain that exhibits a high interrelatedness between its concepts.

Such a modeling approach makes it possible to identify structural dependencies between concepts and to control their effects on the teaching sequence. This includes checking that course material exhibits a sound sequence of topics and is at all times building on prior knowledge. Additionally, a systematic approach to course modeling can help with comparing courses and textbooks, assessing students’ background, and evaluating textbook coverage or how a textbook satisfies a curriculum recommendation or a standard (such as the ACM/IEEE standards [131] for computing education). An adequate modeling technique for courses also provides the means to investigate the interrelatedness of object-oriented concepts and makes it possible to identify concepts or concept groups that are difficult to learn and teach because of a high interrelatedness.

The methodology described in this chapter extends the idea of Truc (Testable, Reusable Unit of Cognition) [85] and provides a modeling approach that helps analyze the concept structure of object-oriented programming and define courses based on the domain model. Parts of this chapter have already been published [106].
4.1 Related work

Constructivism is today’s dominant theory of learning [8]. It asserts that learners actively construct knowledge through experiences rather than passively storing knowledge they receive from a teacher. In this view, knowledge construction happens iteratively as learners adapt their existing cognitive structures to capture a new experience. One of the techniques for knowledge representation that has its roots in constructivism is concept mapping.

“Concept maps are graphical tools for organizing and representing knowledge” [90]. They are graphs whose nodes visualize concepts and whose (usually directed) edges indicate relationships between concepts. The edges contain labels that specify the relationships between their node concepts. The concept mapping technique has a wide range of applications: it can serve as a vehicle for high-level cognitive exam and tutorial questions, it can be used to detect misconceptions, and it can capture the “tacit” knowledge of experts.

A fundamental characteristic of concept maps is that the edges have labels consisting of arbitrary words. But according to Ferguson [37], these relationships “frequently are some type of IS_A relation in which one concept is a type of another concept or the HAS_A type in which one concept has another concept, which is a component of the first concept.” The present approach limits the relationships between concepts to two types: refines and requires links. The refines relation corresponds to Ferguson’s IS_A relation. When modeling concepts for pedagogical applications, it is interesting to capture external requirements additionally to the internal components of a concept (the HAS_A relations). The requires relation in our model comprises both external and internal requirements. Restricting the set of relational types to refines and requires results in a loss of expressiveness as compared to concept maps, but it makes it possible to automatically analyze the structures for example to check concept sequences for their soundness.

A similar idea to concept mapping is mind mapping [16]. Mind maps have a central governing concept or idea with other related items that are arranged radially around it. The principle for the placement of items is that those intuitively considered more important appear closer to the center. Mind maps use colors, images, or other techniques to additionally annotate branches, groups, and areas. The edges of the tree structure are usually unlabeled. Their restriction to tree structures and their focus on a central concept makes them difficult to use as domain models for course contents.
4.1. RELATED WORK

Other approaches to educational knowledge modeling provide techniques tailored, in terms of expressiveness and formality, to the needs of specific applications.

One of them is the MOT ("Modélisation par objets typés") [96, 95] model. The MOT model’s main area of application is the construction of tele-education systems. It includes graphical knowledge representation by defining knowledge graphs containing “knowledge objects” as nodes. MOT proposes six types of knowledge objects and six types of links accounting for the edges of the graph. Exact rules define for each link type what knowledge object types it may connect. This approach features high formality and expressiveness, but requires significant training until novices can use it efficiently. The complexity of the MOT may be needed for building an online education system, but is an obstacle for the application areas targeted by TrucStudio. The extended Trucs take a simpler approach to modeling both knowledge and dependencies.

Another approach is based on the idea of Anchoring Concepts (AC) and Anchoring Graphs developed by Mead et. al. [81]. An AC is defined as a concept that is either foundational, or both integrative and transformative. The definition of AC in connection with Cognitive Load Theory supports the construction of directed graphs (Anchoring Graphs) where an edge from one AC to another shows that the source node carries part of the cognitive load to learn the target node. Although the concept of Trucs and AC appear very similar, they differ significantly in their intents. The Anchoring Graph approach identifies the foundational concepts that should be taught before another AC to reduce the cognitive load. Therefore, to build AC graphs requires taking teaching-related decisions at design time to produce a partially fixed teaching sequence. A Truc-notion graph is only concerned with the structural properties of concepts and does not enforce a particular sequence for teaching them. A conjecture is that Truc-notion graphs could help construct Anchoring Graphs.

In the context of computer and information sciences, an ontology defines a set of primitives that make it possible to model knowledge of a domain [56]. The primitives usually are classes, attributes, instances, and relationships. They usually exhibit properties similar to object-oriented modeling. Many researchers in e-learning advocate the use of ontologies to produce metadata that classify learning objects and support their reuse. Several ontology storage formats such as OWL (Web Ontology Language) [140] support the specification of a domain model. Ontologies are a powerful, general, expressive way of describing arbitrary domains and the Truc-notion domain models can be translated to ontology specifications using an output generation mechanism provided as part of TrucStudio.
CHAPTER 4. MODELING COURSES WITH TRUCS

The Curriculum Initiative CC2001 [131] defines the body of knowledge for Computer Science by listing the core concepts belonging to a specific area. While these efforts have a great impact on curricular planning, they do not specify how compliance to the course definitions (as reported in CC2001) can be assessed and they do not identify dependencies between the units of knowledge.

4.2 Development of domain models

We have noted a number of applications for Trucs: checking the soundness of the order chosen for topics in a course; comparing the contents of courses, textbooks, and curriculum standards; and assessing students’ backgrounds with respect to their conceptual knowledge. Achieving these goals requires two tasks: (1) extracting the knowledge taught in courses/textbooks/curricula as a list of units of knowledge and comparing it to the intended results or knowledge acquired from another source, and (2) defining the dependencies that occur between units of knowledge and verifying that they are met.

The model targets the definition of knowledge units and their dependencies. The original work on Trucs [85] proposes knowledge units at two levels of granularity: “Trucs” and “clusters”. We have extended the original model with a third type of knowledge unit, “notion”. The final model then provides knowledge units at three levels of granularity (in increasing level of granularity): “notions”, “Trucs”, and “clusters”. At the highest level, a “cluster” is a loose collection of Trucs representing a particular knowledge area. Clusters help keep large numbers of Trucs manageable. At the medium level, a “Truc” is a set of concepts and operational skills emerging from one central idea. And at the lowest level, a “notion” is a single concept, or facet of a concept.

The model also defines relations between clusters, between Trucs, and between notions. These relations capture the prerequisite structure of the knowledge units and help improve and assess the sequence of topics of a course. They are also the basis for investigating the “tight interrelatedness of basic concepts” [10] in object-oriented programming.

4.2.1 Trucs

A Truc (Testable, Reusable Unit of Cognition) is “a collection of concepts, operational skills and assessment criteria” [85]. It satisfies the following properties: (1) all components of the Truc follow from a central, clearly
identified idea; (2) the Truc possesses a clear, unambiguous description; (3) the Truc includes assessment criteria to judge whether a student has mastered the concepts and skills; (4) included topics span a scope of general interest; and (5) the scope is small enough to be covered in one or a small number of lectures or in a textbook chapter or a few sections.

A Truc description follows a standardized scheme containing sections on the Truc’s name, any alternative names, its dependencies, and a summary. The standard scheme also includes sections on the role of the Truc in the field, when and where it can be applied, and its benefits; it gives examples, states the common confusions that might occur when mastering the concepts represented by the Truc, and describes the disadvantages of applying the concepts. The Truc’s specification concludes with a set of typical tests for assessing understanding of the Truc’s concepts.

This thesis extends the common confusions section of Trucs with checklists containing recommendations applicable to teaching material such as slides. The recommendations help instructors check that their teaching material addresses the common misconceptions of students and thus are a means to improve the teaching material.

One intended application of the Truc models is to compare courses. To assess how Trucs can help with comparing courses, consider Trucs covering a subset of the concepts for object-oriented programming provided in the ACM/IEEE Curriculum standard [131] “object”, “argument passing”, “class”, and “feature call” (method in the syllabus). An additional Truc is “Design by contract” (covering pre-, postconditions and class invariants). Obviously, a course that teaches introductory programming with Eiffel will cover all of the above Trucs. A similar course taught with Java probably will not teach the Truc “Design by contract” (see Figure 4.1). A comparison of the covered Trucs captures this difference.

Figure 4.1: Example of Trucs
To capture more subtle differences, for example to adapt the course to different audiences, Trucs are too coarse-grained. For example, Java supports overloading of methods; this concept is unknown in Eiffel.

These detailed differences are particularly interesting for courses that build on prior knowledge acquired in other courses. As an example, the instructor of a course for Eiffel programmers on design patterns that uses Java as a support will be able to spend very little time on language details. Thus, he needs to consider what exactly the students have already learned in the Eiffel courses and where this knowledge differs from the required Java knowledge. In most cases the differences are very small, but still need to be understood to comprehend the course.

Checking whether a textbook or a course covers all the required topics constitutes another task where details are of great importance. Coverage verification entails comparing existing educational material to some knowledge to teach. It is easy to verify that the big topics (the Trucs) are covered; instructors or textbook writers rarely forget to treat a whole Truc. But the devil is in the details and they may forget some notions. As an example, the case study in Section 5.4.1 identifies several small parts of otherwise covered Trucs that are often missing in slides, including ours.

These observations suggest that Trucs need some refinement to address the demands of course/textbook comparison.

### 4.2.2 Notions

While Trucs capture the central teaching units of a course, smaller-sized units of knowledge – notions – help refine the specification and address the need for a more fine-grained kind of knowledge unit. A notion “represents a single concept or operational skill or facet of a concept” [106]. Every notion is associated to exactly one Truc. A Truc may have a central notion, which then bears the same name. Examples of notions within the Truc “feature call” are: the central notion “feature call” (capturing the general idea of a method call instruction), “multi dot feature call” (calls of the form o1.o2.o3.f), and “unqualified feature call” (feature calls that do not specify their target).

### 4.2.3 Clusters

A cluster is a collection of Trucs and other clusters or both. A Truc belongs to exactly one cluster in the domain model; the set of all clusters forms a directed acyclic graph. Clusters represent knowledge areas and help keep large numbers of Trucs manageable.
4.2.4 Relations

The framework defines two types of relations between notions: \textit{refines} links and \textit{requires} links. They make it possible to check the soundness of a teaching sequence and to detect prerequisite violations.

“\textit{a requires b}” defines that understanding notion \textit{a} requires knowing notion \textit{b}. This relation is comparable to the client relationship between classes in object-oriented systems. The \textit{requires} relation may appear between any two notions in a model, including notions that belong to two different Trucs. This is the case, if notion \textit{a} belongs to Truc \textit{A} while notion \textit{b} belongs to Truc \textit{B} and \textit{A} \neq \textit{B}. In this case, we consider a link between \textit{a} and \textit{b} a “link across Truc boundaries”. The links across Truc boundaries define the Trucs’ dependencies.

“\textit{a refines b}” expresses that notion \textit{b} is a specialization of notion \textit{a}; it is comparable to the inheritance mechanism in object-oriented systems. A refined notion implicitly inherits all the \textit{requires} links from its ancestor, but may also introduce additional ones. For simplicity, the methodology prohibits \textit{refines} links across Truc boundaries.

The \textit{requires} and \textit{refines} links between notions define a graph with notions as nodes and links as edges. We call this graph a \textbf{notions graph}.

Since each Truc contains a set of notions, the Trucs can be seen as clusters that group notions in the notion graph. The Trucs and notions define a two-layered graph, which provides a domain model for organizing courses and curricula with TrucStudio. Figure 4.2 shows an example of such a \textbf{Truc-notion graph}\footnote{In an earlier article [106], the Truc-notion graph was called \textit{clustered notions graph}. To prevent misunderstandings in connection with the third entity type “cluster”, we use the name \textit{Truc-notion graph} in this thesis.} containing the notions of the Trucs “feature call”, “object”, “class”, “argument passing”, and “Design by contract”. As an example, the notions for the Truc “class” are “class”, “type”, “expanded type”, “reference type”, and “generating class” (shown as grey boxes in the figure). The solid black arrowed lines represent \textit{requires} links while the grey lines mark \textit{refines} links.

A student is said to have mastered a particular Truc if she has mastered all the notions belonging to its Truc-notion graph. Dependencies at the notion level contribute to dependencies at the Truc level: a Truc \textit{A} \textit{depends} on another Truc \textit{B} if any of its notions \textit{requires} a notion of \textit{B}. As an example of links between notions, “unqualified feature call” is an \textit{refines} specialization of “feature call” and “feature call” \textit{requires} the “argument passing”. As a consequence, the Truc “feature call” depends on the Truc “argument passing”.
Figure 4.2: Truc-notion graph with Trucs Argument passing, Feature call, Class, Object, and Design by Contract
The dependency links between Trucs can directly be used in the Trucs graph (see Figure 4.1), which can be abstracted from the Truc-notion graph (see Figure 4.2). The inter-Truc links in Figure 4.2 show the following Trucs dependencies: “feature call” requires “argument passing”, “argument passing” depends on “feature call” and “class”, “class” requires “object” and conversely, and “Design by contract” requires “class”. Such graphs provide the domain model for organizing courses with TrucStudio.

The original work on Trucs [85] prohibits both cycles and transient subgraphs in the dependency structure. The refined methodology presented here refrains from putting such restrictions on the requires relations between notions and the resulting deduced Truc dependencies. Relaxing the requirement of non-transient subgraphs enables a higher expressiveness for Trucs and notions. Identification of cycles in the dependency structure of Trucs or notions is an interesting task since it shows concepts that are probably difficult to teach in isolation and require an iterative teaching approach.

4.3 Development of course models

The course model is a two-level hierarchy: On the lower level, lectures are a sequence of notions taught one after another. Courses – on the higher level – are a set of lectures mapped onto available timeslots.

4.3.1 Lecture

A lecture is a logical building block of a course on a given topic, comparable to a chapter of a textbook or a set of slides on a certain topic. A lecture in this sense is not constrained by time; it could be a short block not even filling a whole lesson, but it could also be a block stretching over several lessons of a course. A lecture contains an estimate of the time needed for teaching the corresponding material and a sequence of notions that represent the order in which they are covered. Notions may appear multiple times in a lecture indicating an iterative visiting structure (see also Bergin’s Spiral pattern [12]).

4.3.2 Course

A course possesses a list of available time slots (e.g. every Monday 2 p.m. to 4 p.m. in the semester) and a list of lectures. The lectures mapped onto
the time slots produce the course schedule. The set of all the notions in a course is called the set of underlying notions of the course.

The course model provides the basis for comparing two courses, textbooks, or standards. Such a comparison takes into account their underlying notions: one is compliant with another if its set of underlying notions includes the other’s. Besides compliance verification, it may also be interesting to compare two sets of underlying notions and find both their intersection (describing the common knowledge they cover) and their relative complements (describing what has to be taught to complete knowledge or what has to be mentioned as a difference from previous knowledge).

4.4 Applications to the teaching of programming

Developing a domain model (with its Trucs, notions, and links) and a course model (with its lectures and underlying notions) entails various subtasks: (1) define the concepts and skills that the Trucs should represent and identify the associated notions; (2) construct the links between notions resulting in a Truc-notion graph, create concise Truc descriptions by producing the contents of technical sections such as summary, role, applicability, benefits, and pitfalls, and collect common confusions and sample questions; (3) create course models as sequences of lectures each covering a series of notions.

Every subtask results in resources that can be applied to aspects of teaching programming. In the following paragraphs, we report on the possible implications of the selection of Trucs and notions, we describe how the descriptive and structural parts of a Truc could help improve teaching of introductory programming, and we detail on the possible applications of the course models to programming courses.

4.4.1 Set of Trucs and notions

To define the Trucs and notions, the instructor or model developer needs to identify the central concepts of a teaching domain (the Trucs) and their facets and variants (the notions). Since according to the original Truc article [85] “many technical topics are in constant evolution” and “even if the field itself is stable, its interpretation may be controversial”, the resulting Trucs and notions will most likely represent the instructor’s understanding and personal view of the field. The communication of this view is important for the discussion and comparison of different approaches.
to teaching and for synchronizing introductory courses with advanced courses relying on the concepts taught.

Comparing approaches and views of the domain. The field of introductory programming has no standard set of concepts as indicated by the four general approaches described in the computing curricula recommendations [131] (see Section 2.1). In fact, students taught with one of the approaches (for example an objects-first approach) will master a mostly different set of concepts than those taught with another approach (such as the imperative-first approach). Codifying the domain as a set of Trucs provides a rational basis for discussing differences between the approaches and the views of the domain.

Furthermore, the specific programming language with which the instructor teaches his course will most likely influence the development of the Trucs and notions of, for example, object-oriented programming. A comparison of domain models developed by teachers with a focus on different programming languages could help analyze conceptual differences between the languages, which also influence the views that students develop when learning a certain programming language.

Synchronizing with advanced courses. Follow-up courses to an introductory programming course often rely on the programming skills and concepts covered in these courses. Since there is no default set of concepts available for programming and the approaches will most likely result in different covered concepts, a list of Trucs and notions can help instructors of later courses to identify the concepts that they can rely on in their teaching. The list of Trucs and notions also gives an indication of the mindset that students develop throughout an introductory course. As such, the list of Trucs supports adapting the contents of later courses to the concepts covered in an introductory programming course or vice versa, adapting the introductory programming course to the expectations of those teaching advanced courses.

4.4.2 Truc descriptions and dependencies

Every Truc contains nine descriptive sections detailing on various technical and pedagogical aspects of the related concepts. The Truc descriptions offer several possibilities for applications.

Handling common misconceptions. The study on students’ backgrounds presented in Chapter 3 indicates that a substantial portion (about 30%) of the students have had no or only limited exposure to program-
ming when entering the introductory programming sequence. These nov-
ices usually struggle when learning to program and it is necessary to adapt
the teaching of introductory programming to their needs.

The Truc section on common confusions provides a useful resource for
identifying conceptual difficulties of novice programmers. Many of the
collected misconceptions are independent of the programming language
used for teaching and reappear across teaching approaches. By extending
the contents of this section with a checklist of proposed measures to better
cope with the misconceptions, we provide a tool that is reusable by other
instructors. The application of these checklists to the slides of Introduction
to Programming at ETH is one of the measures that we take to address the
decreasing satisfaction of students with the Introduction to Programming
lectures (see Section 2.4).

Relying on examples and sample questions. Other uses of the de-
scriptive sections of a Truc include integrating the examples as illustra-
tions of concepts in teaching material such as slides and using the sample
questions in tests to assess whether a student has mastered a concept.

Distributing the Trucs to students. Making the Trucs available to
students, for example as a webpage, would allow them to use it as a
course compendium. The distribution can be adapted to the diverse stu-
dent backgrounds: novices, for example, can use single Trucs as a sum-
mary whenever the according concepts have been introduced in a lecture
of the course. The more experienced students can explore the entire set of
Trucs from the start and identify the concepts they already know, discover
where their views and experiences differ from those taught in the course
and where they correspond. This helps connect to their prior knowledge,
one of the measures we propose for adapting the teaching of introductory
programming to the diverse student body (see Section 3.8). One important
descriptive section in this respect is the list of alternative names provided
for every Truc. Offering a comprehensive lexicon that lists all Truc names
and alternative names, each referencing the according Truc descriptions,
can help students who are used to a different terminology identify the cor-
responding concepts within the course.

Analyzing the dependency structure of a domain. The high inter-
relatedness of object-oriented concepts is the cause for some of the chal-
lenges of teaching introductory programming. Modeling the domain and
analyzing the resulting dependency structure can help identify the con-
structs that are responsible for the high interrelatedness. It can also pro-
vide a basis to develop solutions to the Big bang problem [10] (the problem
of starting with a certain topic without violating its dependencies).

4.4.3 Course models

With the extended Truc approach, once the concept model of a certain domain is available, courses, textbooks, or curriculum standards can be defined as sequences through the notions. The course models offer a set of further possible applications.

Comparing courses, textbooks, and standards. In the US, accreditation organizations (for example ABET) and interest groups provide curriculum recommendations or standards (such as the ACM/IEEE standards for computing education [131, 132]) defining a set of concepts that need to be covered in a course. Verifying compliance of a course or a textbook to a given standard on the basis of sound, objective evidence entails comparing the list of concepts covered in the course to the set of required topics. This mechanism also may help identify variations in different teaching approaches to introductory object-oriented programming (see Section 2.1), in particular those affecting the contents and topic sequence (such as the Inverted Curriculum approach described in Section 2.2).

Detecting prerequisite violations and missing topics. Prerequisite violations and possibly missing notions are difficult to detect for instructors, because the majority of students start with prior programming knowledge and are usually able to fall back on knowledge they have acquired before the course. Novice programmers often are helpless when confronted with a violated prerequisite or a missing topic. If we want to adapt our teaching to novices, then we have to make sure that no prerequisite violations appear and that all topics needed to understand the lectures and to complete the assignments are covered. Modeling courses with the extended Truc approach can help in this endeavor.

4.4.4 Tool support

Effectively developing and maintaining the domain and course models requires tool support that satisfies a set of properties: it supports constructing instances of the five entity types (cluster, Truc, notion, course and lecture) and entering associated data (such as descriptions and dependencies); and it deduces Truc dependencies from the requires relations between notions and cluster dependencies from the Truc dependencies.

The TrucStudio tool described in Chapter 6 provides this functionality. To help develop the domain models including their dependency structure and the associated course models, TrucStudio provides a graphical rep-
representation of the domain model and several visual representations for courses.

The applications just sketched add requirements for TrucStudio. They include, for example, providing several output variants containing the domain model. TrucStudio addresses this need through a flexible and extensible output generation mechanism (see Section 6.5). It supports producing customized webpages, calendar files, and ontology documents. In particular, it can produce a browsable lexicon for advanced students structured by alternative names containing the entire domain model, but also webpages for smaller parts of the domain model for novices.

For comparing two courses, TrucStudio’s course diagram tool visualizes the concepts covered in a course and the sequence in which they are introduced (see Section 6.4). This permits an interpretation of what topics the course focuses on.

The course diagram can also be used to detect missing topics, both on Truc and notion level. To do this, the instructor needs to check that all the notions appear in at least one lecture.

To help identify violated prerequisites, TrucStudio displays the dependencies of notions covered in a lecture. Additionally, it marks possibly violated prerequisites and notions that have already been covered in a previous lecture. This reporting is particularly useful if the sequence of lectures needs to be rearranged.

The structure report of TrucStudio (see Section 6.6) lists direct and indirect requirements of Trucs and notions and provides a complete picture of the dependencies. It also detects cycles in the dependency structure and reports them to the user. This makes it possible to discover concepts or concept groups that are difficult to place in a course because they have many prerequisites or circular dependencies.

4.5 Summary

This chapter has presented a modeling approach for courses and knowledge. It has extended the ideas of Trucs and clusters with finer-grained knowledge units (notions), which help determine dependencies. The central point is to allow analysis of dependency structures, prerequisite violations, and comparisons of similar domains and teaching. This brings both normative and descriptive support to the definition of courses, and yields clear views of previous knowledge and course prerequisites.

The domain and course models help communicate and compare different teaching approaches with respect to the covered topics and the se-
4.5. SUMMARY

quence in which they appear. They are a resource for students with programming experience, which supports them in relating preexisting knowledge to the course contents. They provide a course summary to novice programmers. This allows them to look up definitions and examples if they encounter unknown concepts. They support teachers in designing their course by helping check prerequisite violations, identify missing topics, and adapt the teaching to common misconceptions of novice programmers.

For most of the above uses, the development of Trucs including the notions and links is a premise. Chapter 5 provides a model for object-oriented programming. It consists of 28 Trucs and the course model of Introduction to Programming. The chapter includes an analysis of the domain’s dependency structure and a report on the detection of possibly missing notions, violated prerequisites, and conformance of the course slides with the recommendations addressing novices’ misconceptions.

To develop the domain and course model we have used TrucStudio, the tool supporting the presented approach (see Chapter 6). It has particularly helped in the analysis of the dependency structure and the detection of missing notions and violated prerequisites. Additionally to this, TrucStudio’s output generation mechanism makes it possible to distribute the domain and course models adapted to the readers (for example other instructors, advanced students, or novice programmers).
This chapter shows the TrucStudio domain modeling approach applied to object-oriented programming and the ETH Introduction to Programming course. The result is a set of Trucs with descriptions of common confusions, summary and test of understanding.

Modeling the domain of a course, such as the programming language constructs and the other taught notions of an introductory programming course, helps detect connections between concepts and makes it possible to systematically analyze the sequence in which they are introduced in an actual course. Furthermore, it allows for an analysis of the domain structure and provides interesting insights into the following questions: Which are the central concepts of the field (i.e. concepts with an above-average number of dependencies)? Are there concepts with cyclic dependencies indicating that they must be covered together or repeatedly or both? Are there concepts with no dependencies allowing flexible placement in the teaching sequence? This chapter answers these questions based on the developed domain model and analyzes the course model of the ETH Introduction to Programming course.
CHAPTER 5. TRUCS FOR OBJECT-ORIENTATION APPLIED TO AN INTRODUCTORY PROGRAMMING COURSE

5.1 Related work

Several articles and standards have guided the work of selecting the concepts/skills that should be captured as Trucs.

The article on “the quarks of object-oriented development” [4] has proven very useful. It describes a literature review of 88 sources, which mention 39 concepts that characterize the OO approach. Of these 39 concepts inheritance, object, class, encapsulation, method, message passing, polymorphism, and abstraction appear in over 50% of the sources and thus are elected as “quarks”. Except for abstraction, all of the quarks appear as Trucs (encapsulation under the name information hiding and message passing as feature call). The work on the quarks of object-oriented development also influenced the summary section of the Trucs by providing definitions of the eight quarks.

An experiment by Sanders et. al. [121] contrasts the expert view of the quarks with the view of students who recently had studied object-oriented programming. They asked them to draw concept maps [90] that summarize their knowledge of OOP. The most commonly mentioned concepts are class, method, instance, variable, and object. Other commonly found concepts (implicitly or explicitly) are data/attribute/instance variable, inheritance, and encapsulation. The developed Trucs contain all of these concepts; instance is integrated in the Truc object and data/attribute/instance variable in Truc feature.

Schulte and Bennedsen [124] carried out a study in 2006 where they asked computer science teachers from high schools, colleges, and universities in various countries to rate the difficulty, relevance, and cognitive level of 28 programming topics. They refer to a set of other studies [87, 22, 131] that helped develop the list of topics. The topics with highest relevance are selection and iteration, simple data structures, parameters, scope, object and class and syntax. The Trucs described in the following section cover these topics, except for syntax.

The ACM curricular initiative CC2001 [131] defines the body of knowledge of computer science by specifying 14 knowledge areas ranging from Discrete Structures, Programming Fundamentals, to Social and Professional Issues. Each knowledge area contains a set of units, which hold a set of topics. It also defines six curricular models for introductory courses and proposes a syllabus and set of units for each variant. The syllabi and description of knowledge units have also guided the selection of concepts covered by Trucs.

Various books, articles, and other resources have influenced the descriptions of the Trucs. In particular, Touch of Class [86], which is used
5.2 Modeling object-oriented programming

The model for object-oriented programming contains 28 Trucs ranging from imperative programming concepts such as *Conditional* and *Loop* to object-oriented concepts such as *Inheritance* and simple data structures as for example *Linked list*.

The Truc descriptions follow a fixed scheme. Every Truc starts with a list of alternative names, a list of direct dependencies, and a list of notions (the underlined notion is the central notion) shown in a box. As a second part, it includes the descriptions of summary, role, applicability, benefits, pitfalls, examples, common confusions, and sample questions. To improve readability and reusability of the common confusions section, we add checklists that propose measures to adapt teaching material to the collected misconceptions. The Truc description closes with structural in-
formation provided through an extract of the Truc-notion graph showing the described Trucs and the Trucs it directly depends on with their notions and links. Figure 5.1 shows the legend for the Truc-notion graphs in Sections 5.2 and 5.3.

Figure 5.1: Legend for Truc-notion graphs
5.2.1 Algorithm

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Algorithmic scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Instruction</td>
</tr>
<tr>
<td>Notions</td>
<td>Algorithm, Complexity, Big-O notation, Worst case</td>
</tr>
</tbody>
</table>

Summary
An algorithm is a description of a computational process that solves a certain problem acting on a possibly empty set of data. It specifies precisely to which data sets it may be applied, which elementary steps it takes, and in what order(s) they can be executed. It follows such conventions that it is executable on automata without human intervention. If executed with the same data set on two automata with the same conventions, it should produce the same results. [86] A central concept in connection with algorithms is their efficiency. Efficiency describes the consumption of various types of resources, in particular its execution time and its memory usage.

Role
Today’s software systems are large and involve many algorithms. The choice of algorithm for solving a certain problem makes it possible to increase performance of programs or decrease memory usage. Knowledge of available algorithms and their advantages and drawbacks is fundamental for system development.

Applicability
Need to solve a task.

Benefits
- Many resources provide descriptions of algorithms that solve recurring tasks (such as sorting or searching). A programmer can thus rely on the experience of others.

Pitfalls
Choosing an efficient algorithm requires knowledge of the characteristics of the processed data. If these characteristics change during the execution of a system, an algorithm may become inefficient.

Examples
Consider an unsorted list of numbers that you want to sort in ascending order. There are various strategies to solve this task. One of them, Bubble-sort, starts at the beginning of the list and compares the first two elements. If the first is greater than the second, then the two elements are swapped.
It continues to do this for every pair of adjacent elements until it reaches the end of the list. At the end of the pass, the largest element is at the last position of the list. The algorithm then starts again with the first two elements and continues until no swaps occur in a pass any more. The Bubblesort algorithm is inefficient. Many more efficient sorting strategies are available.

**Common confusions**

- **Short algorithms.** A common misconception is that, of two algorithms, the one with fewer lines of code, fewer variables, fewer instructions, or less syntax is more efficient. [92, 43]
- **Same instructions.** Another common misconception is based on the idea that two algorithms containing the same instructions (although in different order) are similar with respect to their efficiency. [43]
- **Same goal.** Another erroneous conception states that two algorithms solving the same task generally have the same run-time efficiency. [43]

Examples compare algorithms where the shorter (in terms of lines of code, number of variables, and/or number of instructions) is less efficient.

Examples compare two algorithms consisting of identical instructions in differing order that have different execution times.

Examples include algorithms that solve the same problem, but differ in their efficiency.

**Sample questions**

1. What is the difference between a program and an algorithm?
2. Devise two different algorithms for searching an entry in the phone book based on last and first name. Discuss their advantages and disadvantages.

**Truc-notion graph**

![Figure 5.2: Dependencies of Truc Algorithm](image)
5.2.2 Argument passing

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Class, Expression, Feature call, Feature, Reference</td>
</tr>
<tr>
<td>Notions</td>
<td>Argument passing, Actual argument, Argument declaration, Formal argument</td>
</tr>
</tbody>
</table>

Summary
An argument is a value that is passed to a routine using the mechanism of argument passing. Argument passing attaches values or references to the formal arguments of a routine when it is called. The names and types of the formal arguments are defined in the argument declaration, which is part of the feature signature. A client then must make sure that the types of the actual arguments conform to the declaration. There are multiple evaluation strategies for argument passing. The most typical ones are call-by-value (where a local copy of the object is used in the feature) and call-by-reference (where the object is directly modifiable in the feature).

Role
Passing arguments is a fundamental technique of programming. It broadens the scope of routines by letting the client adapt certain of its parameters.

Applicability
On the side of the called feature: Need to get access to information provided by the caller of the feature. On the side of the caller: Need to adapt certain parameters of the called feature.

Benefits
- Favors reuse.

Pitfalls
In the case of call-by-reference, the called routine may change the attributes of the passed object, leading to side effects.

Examples
Consider a class `BANK_ACCOUNT`, which offers a routine `transfer` with two formal arguments `other`: `BANK_ACCOUNT` and `amount`: `INTEGER`.

Assume that the entities `account1` and `account2` denote objects of type `BANK_ACCOUNT`. An instruction calling the `transfer` feature could look as follows: `account1.transfer(account2, 200)`. The expressions `account2` and `200` are called actual arguments while `other` and `amount`
are called *formal arguments* of the routine. When the feature call is executed, the formal arguments are associated to the actual arguments by attachment. If the type of the argument is expanded, attachment means copying the referenced object (call-by-value evaluation strategy); if it is a reference type, the attachment results in a copy of the reference (call-by-reference evaluation strategy). In the example, `your_account` is therefore passed by reference, while 200 is passed by value.

**Common confusions**

- **Passing on arguments.** Formal arguments of a feature may be passed on as actual arguments to another feature. This may generate confusions. [42]

- **Matching the specification.** Novice programmers commonly have difficulties in matching the actual arguments to the specification. In particular, they tend to specify a wrong number of arguments or to choose actual arguments of types that are not matching the formal argument list. [89, 63]

- **Type declaration.** For many novices, it is unclear when they have to state the types of the arguments; they tend to confuse argument declarations with the argument passing in a feature call. [63]

□ Examples include a situation where formal arguments of a feature are passed on as actual arguments to another feature.

□ Examples show feature calls with invalid and valid actual arguments (in terms of type and number of arguments).

□ Examples distinguish between argument declaration (including a type specification) and feature calls with arguments.

**Sample questions**

Produce an example of a feature call with multiple arguments where at least one is passed by value and one is passed by reference. Explain how the two evaluation strategies may affect the objects that are passed as arguments.
Truc-notion graph

Figure 5.3: Dependencies of Truc Argument passing
5.2.3 Array

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Vector, Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Algorithm, Genericity, Primitive type</td>
</tr>
<tr>
<td>Notions</td>
<td>Array, Array performance, Array resizing, Index</td>
</tr>
</tbody>
</table>

**Summary**
An array is a linear container data structure. It is mostly used for frequent index-based access to its elements. Arrays generally occupy a contiguous area of storage, increasing their ability for fast random access, but decreasing the ease of moving elements. There are two main categories of arrays: fixed-sized (or static) arrays do not support resizing after initialization, while dynamic arrays can be resized.

**Role**
Array is one of the fundamental container data structures and can be used to implement other container data structures.

**Applicability**
Need to represent a set of elements identified by indices in a range, in particular mathematical vectors and matrices.

**Benefits**
- Fast random access to an element based on its index (constant time).
- Fast linear traversal.
- Fast replacement of elements.

**Pitfalls**
Moving an element from one array position to another usually requires moving several other elements. This makes arrays less suited for dynamic collections with frequent element insertion or removal.

**Examples**
Consider a program that analyzes an English text with respect to the frequency of each alphabetical letter appearing in it (without distinguishing between upper- and lowercase). Since the English alphabet has a fixed set of 26 letters, the appropriate data structure is an array of integers with size 26. To process a `STRING` as input, the program treats each letter separately and increments the number stored at the according index of the array. This example shows a typical use of arrays: element access based on an index occurs frequently and resizing of the array is unlikely to happen. (Example taken from [23]).
Common confusions

- **Array indices.** In the case of integer arrays, problems may arise because students confuse the array index with the stored value. [29]
- **Special cases.** Depending on the implementation of arrays (for example dynamic arrays), handling of special cases such as empty arrays or full arrays (if there is a maximum capacity) is often not considered. [86]
- **Array limits.** In some implementations, array indices start at 0 and in others they start at 1. These differences also influence the index of the last element, which is then either \( \text{count} \) or \( \text{count} - 1 \) (given that \( \text{count} \) is the size of the array). This may lead to out of bound errors and difficulties with array traversal.
- **Container creation and initialization.** It is a common error to forget to create and initialize container data structures (see also Truc 5.2.22).
- **Two-dimensional arrays.** When dealing with two- and more-dimensional arrays, a common problem appears in connection with the indices for rows and columns. Very often, it is unclear or counter-intuitive which index should be varied and which should stay constant in order to traverse a row or column. [29]

Examples include a case with an integer array that shows the difference between array index and stored value.
Examples include an algorithm that handles special cases of arrays (e.g. full dynamic array that requires resizing).
Examples include correct and incorrect traversals of arrays with respect to the exit condition of the traversing loop.
Examples stress the need to create and initialize containers.
Examples include two- or more-dimensional arrays and show how their elements are accessed.

Sample questions

1. Give an example of when an array outperforms a linked list and conversely.
2. Write a program that stores ten integers in an array and then finds the index of the maximum number.
Truc-notion graph

![Diagram showing dependencies of Truc Array]

Figure 5.4: Dependencies of Truc Array
5.2.4 Assignment

**Alternative names**
Association

**Dependencies**
Class, Expression, Reference

**Notions**
Assignment, Reference assignment, Value assignment

**Summary**
Assignment is the mechanism by which an entity may be attached to an object. The assignment instruction consists of the target entity on the left-hand side of the assignment operator and an expression on the right-hand side, \( x := e \). The target \( x \) needs to be a writable entity. Writable entities are the non-constant attributes of the enclosing class and the local entities of the enclosing routine. The expression \( e \) needs to evaluate to an object of compatible type [83]. In general, we distinguish between reference assignment for reference types (where the assignment results in the attachment to an object or Void) and value assignment for expanded types (where the assignment results in a copy of the evaluated right-hand side expression).

**Role**
Assignment and object creation are the two main mechanisms for manipulating the object structure. Object creation allows references to be attached to newly created objects. Assignment allows the manipulation of the object structure using the existing objects and Void. Assignment is the sole mechanism by which a reference may be reattached to Void (after being attached to a non-Void entity).

**Applicability**
Need to reattach the target reference to the object to which the source expression evaluates.

**Benefits**
- Makes it possible to access an object through different paths in the object structure (aliasing).
- Allows references to become Void.

**Pitfalls**
Aliasing can make programs difficult to understand and difficult to debug.

**Examples**
*Reference assignment:* Consider a class PERSON with a field **loved_one** of type PERSON and a field **name** of type STRING. Given are three variables \( a \), \( b \), and \( c \) of type PERSON referencing three existing PERSON objects as shown
CHAPTER 5. TRUCS FOR OBJECT-ORIENTATION APPLIED TO AN INTRODUCTORY PROGRAMMING COURSE

in Figure 5.5(a). Assume that the following four assignment instructions are individually applied to the initial object structure.

1. \( b := a \) The entity \( b \) is attached to the object attached to \( a \) (see Figure 5.5(b)).
2. \( b := a\.loved\_one \) The expression \( a\.loved\_one \) evaluates to the object with name “Ben”. Since \( b \) is already attached to this object, no changes in the object structure are visible (see Figure 5.5(a)).
3. \( b := c\.loved\_one \) The expression \( c\.loved\_one \) evaluates to \texttt{void}, so \( b \) is attached to \texttt{void}. The object with name “Ben” is now only reachable by going through \( a\.loved\_one \) (see Figure 5.5(c)).
4. \( a := a\.loved\_one\.loved\_one \) The source expression evaluates to the object “Cleo”, so \( a \) is attached to it. The object with name “Anna” is not reachable any more in the object structure (see Figure 5.5(d)).

Value assignment: Given are three variables \( i, j, \) and \( k \) of expanded type \texttt{INTEGER}. The following code extract shows how a sequence of assignments changes the values of the three variables.

\begin{verbatim}
i := 7 -- i has value 7
j := 3 -- j has value 3
j := i -- j has also value 7
i := i*2 -- i has value 14, j still has value 7
k := j + i -- k has value 21
\end{verbatim}

Common confusions

- **Assignment asymmetry.** The assignment operation is an asymmetric operation: it is possible to write \( x := 2 \), but not \( 2 := x \) [29]. In connection with the asymmetry of assignment, a common mistake is to confuse the target and the source expression [109, 89]. This is especially the case in C-like languages that use a graphically symmetric operator (=) for assignment.

- **Accumulation assignment.** Assignments that use an entity both as the target and as part of the source expression (as in \( x := x + 7 \)) require to consider the state of the entity for its execution: the assignment instruction evaluates the right-hand side expression based on the current value of \( x \) and then assigns this result to the entity. The startling issue is that the \( x \) on the left side of the assignment operator stands for an object location while the \( x \) on the right-hand side denotes a value. [120, 29]

- **History of values.** It is important to note that an assignment overwrites the existing value and that the overwritten value cannot be retrieved any more afterwards. [29]
• **Aliasing.** While for reference assignment it is true that an assignment of the form $o := p$ will link $o$ and $p$ together (by referencing the same object), this is not true for value assignment. [29]
• **Moving.** Another misconception is that an assignment of the form $x := y$ will “physically” move the object or value denoted by $y$ to $x$, so that $y$ is “empty” while $x$ now contains the object or value. [29]
• **Swapping.** Assignments, such as $x := y$, are mistakenly interpreted
as swapping the values in the variables such that \( x \) holds the former value of \( y \) while \( y \) holds the former value of \( x \). [126]

- **Unevaluated definition.** Assignments containing an expression on the right-hand side may generate the misunderstanding that the target afterwards holds the unevaluated expression as a definition of its value. For example, \( x := y + 2 \) may appear to result in \( x \) holding the expression \( y + 2 \) such that followed by \( y := 7, x \) evaluates to 9. [29]

- **Type incompatibility.** A common mistake of novice programmers is that they try to write assignments where the result of the source expression is incompatible by type with the target. [89]

- **Read-only entities.** Certain programming languages disallow assignment to attributes of other objects and/or to the arguments of the enclosing routine. Novices have difficulties with understanding which entities are available for assignments and which are not.

- Examples show invalid assignment (source := target).
- Examples emphasize difference between source and target.
- Examples include assignments where the target needs to be evaluated in the source expression.
- Examples include overwritten entities that show that the history of values is lost.
- Examples show that \( x := y \) results neither in swapping of elements nor in moving the attached object or value.
- Examples include a case where an unevaluated definition results in wrong outcomes.
- Examples include valid and invalid assignments with respect to (in)compatible types of the source and targets.
- Examples show valid and invalid assignments with respect to writable entities as targets.

**Sample questions**
Consider the same object structure as described in the example section. Assume that the following assignment instructions are individually applied to the initial object structure. State how each affects the object structure.

\[
c := a
a := \text{Void}
c := a.\text{loved\_one}
b := b.\text{loved\_one}.\text{loved\_one}
c := a.\text{loved\_one}
\]
Truc-notion graph

Figure 5.6: Dependencies of Truc Assignment
5.2.5 Class

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Feature, Object, Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td></td>
</tr>
<tr>
<td>Notions</td>
<td>Class, Expanded type, Generating class, Type, Reference type</td>
</tr>
</tbody>
</table>

Summary
“A class is a description of a set of possible run-time objects to which the same features are applicable” [86]. A class has a dual nature: as a module it groups a set of related services, it enforces information hiding and offers a restricted interface to its clients, and it has clients and suppliers; as a type it denotes possible run-time objects and references, which are the instances of the type, and it can be used for the declaration of entities. The feature definitions specified in a class text determine the run-time behavior of its instances.

Role
A class can be seen as a building plan for an individual component of a complex system. It serves as the fundamental underlying idea of object-oriented programming and design.

Applicability
Need to define the system components or reuse existing components.

Benefits
- Classes help model a closed system by using object-oriented analysis and design.
- Classes allow system designers to break down large complex systems into smaller components based on the decomposition criterion of actors (“who acts in the system?”) leading to a stable design.
- Enables the design of modular architectures.
- Favors reuse and extendibility.

Pitfalls
One of the central aspects of object-oriented development is finding the right abstractions. This task requires experience and challenges most programmers. Choosing the wrong classes for a system results in code that is difficult to maintain, difficult to read and trace, and not reusable.

Examples
Consider an application that models a public library with a class LIBRARY containing a list of rentable items and a list of accounts, ACCOUNT storing the rented items, and BORROWABLE representing the rentable items. These
classes define the properties and behavior of its instances (objects of the respective type) at run time.

Common confusions

- **Class versus object.** Novices have troubles distinguishing the concept of class from the concept of object and instance. This is particularly the case if the series of examples presented in the early stages of instruction only show a single instance of each class. [62]

- **Class as record.** A prevailing misconception is to see only the data aspect of classes and miss the behavioral aspects (as in records or structures of non object-oriented languages or entity relationship diagrams). Examples that focus on data and resemble database records foster this misconception. [62]

- **Class as attribute wrapper.** If classes only contain one single attribute then the class may be seen as a wrapper for this attribute instead of a more complex data type. [62]

- **Class as type container.** If classes only contain attributes of a single type, then students may think that attributes of the class are limited to be of the recurring type. [62]

- **Class as a set of objects.** It is a common misconception to view a class as a mere set of objects, i.e. to attribute set properties to the class concept. A typical error is then to define a procedure that processes a set of objects of type \(X\) in class \(X\) instead of creating a class \(Y\) for handling a set of objects of type \(X\) that contains the procedure. [26]

- **Set versus subset.** Novices tend to confuse the relationship between class and object with the relationship between set and subset (stating, for example, that a certain class \(ANIMALS\) contains objects such as \(BIRDS\) and \(MAMMALS\)). The term “class” seems to provoke this confusion because in natural language a class usually is a set of individual objects (e.g. a certain French class contains all its students). [130]

- **Class design in declarative versus procedural aspects.** Students often start the design of a solution by identifying the domain objects, and then factor out common classes based on their domain knowledge. They defer the procedural aspects of a solution to when they have to produce a running program. This results in solutions where the procedural aspects appear in a single main feature only loosely coupled with the objects of the system. [26]
Examples show classes with more than one instance. [62]
Examples show classes with a strong behavioral aspect that relies substantially on the state of the object. [62]
Examples show classes with more than one attribute. [62]
Examples show classes that have several attributes of various types. [62]
Examples show classes that process a set of instances of a certain type.
Examples use classes named with singular nouns only.
Examples emphasize that the relationship between class and instances is not a set-subset relation.
Examples show object-oriented design that takes into account procedural aspects from the start.

Sample questions
1. Explain the difference between a class and an instance. [21]
2. Must all classes have more than one instance? Give examples of classes with one instance and one with more than one instance. [21]
3. Assume that you have to design the classes for a simple music player application with a library of albums and songs. What could the classes be?

Truc-notion graph

Figure 5.7: Dependencies of Truc Class
5.2.6 Conditional

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Choice, Selection, Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Instruction, Primitive type</td>
</tr>
<tr>
<td>Notions</td>
<td>Conditional, If-statement, If-then-else statement, If-then-elseif-else statement, Nested conditional</td>
</tr>
</tbody>
</table>

Summary
A conditional is a control structure that supports performing different instructions depending on whether a Boolean condition evaluates to true or false. It supports the strategy of solving a problem by partitioning the set of possible initial situations into two or more disjoint domains, so that it is easier to solve the problem separately on each of these domains [86].

Role
The conditional control structure is one of the fundamental control flow statements and needed for the implementation of many algorithms.

Applicability
Need to execute a set of instructions only if a certain condition holds.

Benefits
• Makes it possible to execute a set of instructions only if the condition evaluates to true and another set if it evaluates to false.

Pitfalls
Over-extensive use of conditionals may indicate bad object-oriented design. If the case distinctions reoccur at multiple places in the code, adding a new case requires updating the code in all these locations.

Examples
Consider the function \( \text{max} \), which takes three \( \text{INTEGER} \) arguments \( i, j, \) and \( k \) and returns the highest number of the three. A possible implementation using a simple conditional looks as follows:

```plaintext
max (i, j, k: INTEGER): INTEGER is
   -- Maximum number of 'i', 'j', and 'k'
   do
      if i > j and k > i then
         Result := k
      elseif i > j and k < i then
         Result := i
      elseif j > i and k > j then
         Result := k
```
CHAPTER 5. TRUCS FOR OBJECT-ORIENTATION APPLIED TO AN INTRODUCTORY PROGRAMMING COURSE

```plaintext
elseif j > i and k < j then
    Result := j
end
end
```

Common confusions

- **Branch execution.** There are several misconceptions concerning the parts of the conditional (if-branch and else-branch) that novices expect to be executed. In particular, novices may expect both the if-branch and the else-branch to execute; expect the if-branch to execute whether or not the condition is true; treat a statement after a conditional without else-branch as though it is the else-branch. [126]

- **Unfulfilled condition.** A common misunderstanding is that an unfulfilled condition of an if-statement results in the program stopping its execution or returning to the beginning of the program. [109]

- **Parallelism.** Another common misconception originates from the analogy of programming with natural conversation. This may lead novices to believe that the temporal restrictions of programming conditionals are similar to “if” statements in natural language, which stay active longer than at the instant when they are voiced. [100]

- **Dangling else.** Certain programming languages like C, C++, and Java support omitting delimiters that mark the scope of conditionals. In these languages, two nested conditionals with only one else part in combination with wrong indentation may lead to a lack of clarity to which conditional the else part belongs.

Examples show a step-by-step execution of a conditional addressing misconceptions concerning what parts are executed under each possible condition.

Examples discuss temporal limits of a conditional.

Examples include a case of dangling else (if necessary).

Sample questions

Write a feature min that takes three arguments \( i, j, \) and \( k \) of type INTEGER and returns the smallest of the three.
Truc-notion graph

Figure 5.8: Dependencies of Truc Conditional
5.2.7  Deferred class

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Abstract class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Class, Feature, Inheritance</td>
</tr>
<tr>
<td>Notions</td>
<td>Deferred class, Deferred feature, Interface class</td>
</tr>
</tbody>
</table>

**Summary**

A deferred class is a class that has at least one deferred feature possibly inherited from a deferred ancestor. A deferred feature provides a feature specification, but no implementation [83]. This means that the feature is available for the class where it is declared, but only implemented in its proper descendants. Deferred classes cannot be instantiated since part of their behavior is undefined.

**Role**

Deferred classes mainly act as ancestors to classes that provide implementations for its deferred features. Defining a feature as deferred ensures that effective descendants of the enclosing class provide an implementation. Deferred classes are particularly useful for analysis and design where either no or only certain aspects of the implementation are needed and desired [83]. Deferred classes serve to classify groups of related types of objects, provide high-level reusable modules, and capture common behavior among a set of variants.

**Applicability**

Need to describe intermediate nodes in a classification, need to capture commonalities between implementations, need to specify classes with no or only partial implementation, and need to master complexity.

**Benefits**

- Helps keep software architecture decentralized and extensible.
- Makes it possible to focus on planning and design.

**Pitfalls**

Inclusion of too many intermediate classes may result in overly complicated hierarchies.

**Examples**

Consider classes representing various shapes with common properties and behavior such as a *bounding_box* function, which returns the smallest rectangle holding the shape, and a *draw* routine to display it on the screen. A class *FIGURE* defines the deferred features *draw* and *bounding_box*; effective descendants (such as *ELLIPSE*) provide concrete implementations.
5.2. MODELING OBJECT-ORIENTED PROGRAMMING

Common confusions
- **Levels of abstraction.** For certain tasks, novice students operate on one of three levels of abstractions for deferred classes. The first level consists of writing a deferred class that only includes attributes, the second level adds effective routines, and on the third level deferred features complete the class. Beginners often remain on level one. [91]
- **Taxomania.** Novices to object-orientation sometimes use inheritance and deferred classes over-enthusiastically resulting in complicated hierarchies. For example, they produce classes MAN and WOMAN as subclasses of PERSON instead of adding an attribute gender. [83]

Example classes include deferred and effective attributes and routines.

Examples discuss taxomania.

Sample questions
Design a class hierarchy that models a public library. The class hierarchy should include a class representing general borrowable items called BORROWABLE, a class for PRINT_MEDIA and effective classes such as BOOK and MAGAZINE.

Truc-notion graph

![Figure 5.9: Dependencies of Truc Deferred class](image)

5.2.8 Design by Contract

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Programming by contract, DbC, Contract programming, Contract-first development</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Class, Feature, Inheritance, Object creation, Primitive type</td>
</tr>
<tr>
<td>Notions</td>
<td>Design by Contract, Assertion, Assertion redeclaration, Class invariant, Creation principle, Invariant inheritance, Precondition, Postcondition</td>
</tr>
</tbody>
</table>

Summary
Design by Contract is a software design approach based on the specification of precise interfaces for classes. Classes define preconditions as obligations for clients, and postconditions and class invariants as benefits.

Role
Design by Contract is a design approach. It is based on the idea that programmers should specify contracts before the actual implementation and that the contracts are an integral part of the software elements. Nowadays, many languages integrate mechanisms that support the specification of contracts as part of the programming language.

Applicability
Need to define client and supplier obligations in software elements.

Benefits
- Fosters reusability.
- Helps debug applications by identifying the locations of errors.

Pitfalls
Run-time assertion monitoring entails a significant performance overhead.

Examples
Consider a class `BANK_ACCOUNT` with a class invariant stating that the balance should never be below 0. `withdraw` could then look like this:

```plaintext
withdraw (x: INTEGER)
require
    x_valid: x >= 0 and x <= balance
ensure
    x_subtracted: balance = old balance - x
```
Common confusions
- **Contracts and inheritance.** The rules for contract inheritance state that preconditions can only be weakened; postconditions and class invariants can only be strengthened. A common mistake is to try strengthening preconditions resulting in assertion violations.
- **Redefined preconditions.** The syntax for redefined preconditions fosters the misconception that a given precondition can be partially weakened without restating the clauses that should still hold.

Examples show what happens for strengthened preconditions.
Examples show partially weakened preconditions.

Sample questions
Given the class `BANK_ACCOUNT`, add the contract interface for a feature `deposit`. Ensure that the balance of the account cannot be below 0.

Truc-notion graph

![Truc Design by Contract](image)

**Figure 5.10: Dependencies of Truc Design by Contract**
5.2.9 Dynamic binding

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Late binding, Override polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Feature call, Inheritance, Polymorphism</td>
</tr>
<tr>
<td>Notions</td>
<td>Dynamic binding</td>
</tr>
</tbody>
</table>

Summary
Dynamic binding governs the application to an object of an operation that has more than one variant. It is the policy that will always, at run time, automatically select the variant that best applies to the object’s type.

Role
To improve software architectures by letting modules ignore the implementation variants of the concepts on which they rely from other modules, hence diminishing dependencies between modules and dissemination of information in a system, to facilitate extension and reuse.

Applicability
Need to call an operation that has several variants depending on the type of object to which it is applied.

Benefits
- Architecture is more decentralized; each module is responsible for operations on a specific type of object, but need to know only the minimum on other types.
- Favors extendibility.
- Favors reuse.
- Clearer program text.
- Shorter program text.

Pitfalls
Dynamic binding may imply a performance overhead (need to find an operation at run time).

Examples
Consider a set of classes representing vehicles, with an underlying inheritance structure: CAR, BICYCLE, MOTOR_BICYCLE, BOAT, SAILBOAT and so on, all inheriting directly or indirectly from VEHICLE. Each may have a different version of a procedure stop that stops a vehicle.

A client class may ask a VEHICLE object represented by v to stop through a single instruction v.stop. Dynamic binding guarantees that any such

---

1Truc description taken from original Truc paper [85]. Additions were done to Sections Alternative names, Dependencies, and Common confusions.
call will trigger the appropriate version, depending on the exact "vehicle" variant of the object denoted by $v$.

Without dynamic binding, each attempt to perform the operation would have explicitly to discriminate between the various vehicle types.

**Common confusions**

- **Dynamic typing vs. dynamic binding.** Dynamic typing involves deferring until run time the check that an operation will be available. This is often confused with dynamic binding; but dynamic binding goes well with static typing, which checks at compile time that at least one operation will be available, and only leaves to run time the choice of such an operation if there’s more than one candidate.

- **Polymorphism.** Since dynamic binding heavily relies on polymorphism, it exhibits similar potential for confusions. Dynamic binding requires an understanding of the difference between dynamic and static type of an entity; a concept that is difficult to grasp for novices.

- **Renaming.** Redefinition in connection with renaming might lead to confusions. As an example, consider a class $A$ that has a feature $f$. Its descendant class $B$ redefines $f$ and renames it to $g$. $B$ additionally contains a new feature $f$. A possible confusion in this scenario might be that in a call to $f$ of class $A$, dynamic binding leads to the execution of $B$’s feature $f$ (instead of the correct variant $g$).

☐ Examples emphasize the differences between dynamic and static type and their connection to dynamic binding.

☐ Examples include class hierarchies where a feature is renamed and appears as a possible variant of a dynamically bound feature.

☐ Examples include a scenario where the feature name is misleading with respect to a possible variant of a dynamically bound feature.

**Sample questions**

A routine takes an argument $d$ of type $DEVICE$, which has an operation $d.shutDown$, and performs $d.shutDown$. $DEVICE$ has the two descendants $COMPUTER$, which redefines $shutDown$, and $PRINTER$, which doesn’t. If the routine is called with an actual argument denoting an object of type $COMPUTER$, which version of $shutDown$ will it call?
Truc-notion graph

Figure 5.11: Dependencies of Truc Dynamic binding
5.2.10 Expression

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Feature, Feature call, Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Feature, Entity, Call expression, Operator expression, Infix operator, Postfix operator</td>
</tr>
</tbody>
</table>

**Summary**
An expression serves to denote a computation that yields a value – an object or a reference to an object. There are several variants for expressions including manifest constants, entities, operator expressions, and function calls [83]. Expressions are used in several contexts of programs, for example as a target to a feature call, as a condition in a conditional, or as the value-giving definition in an assignment.

**Role**
There are two fundamental mechanisms making up programs: describing actions and describing values (on which the actions will operate or which they produce). Expressions cover the description of values, including values obtained through computation from other values.

**Applicability**
Need to compute a value.

**Benefits**
- Provides a flexible way of describing values.

**Pitfalls**
None.

**Examples**
- **Manifest constants** are values that denote themselves, such as the integers 0, -367, the Boolean constant True, or the characters ‘A’ or ‘x’.
- **Function calls** may be unqualified or qualified and may look like this (assuming the proper class and function declarations):

```plaintext
person1.siblings.item (1)
phonebook.item (person.last_name, person.first_name).number
```

- **Expressions with operators** are mostly used for Boolean and arithmetic expressions such as in not list.is_empty or (x + 7)*12 - y.

**Common confusions**
- **Expression evaluation.** Many misconceptions of assignment happen in connection with the evaluation of expressions. It generally seems difficult for novices to understand when and how expressions are
evaluated. For a full list, see Truc Assignment in Subsection 5.2.4.

- **Query as command.** Novices have difficulties understanding that an expression needs to act as input to another construct (for example as a target to a feature call). In certain programming languages, it is possible to use an expression as an instruction, but the result is then lost. See also Truc Feature call 5.2.12). [63]

- **Operator expressions.** Novices are reluctant to using operator expressions as targets for feature calls (such as in \((a+8).\text{abs}\)). They usually store the result of such an expression in a variable and then call the feature on it.

Examples show invalid (or unwanted) use of expression as instruction.

Examples include operator expressions as targets of feature calls.

**Sample questions**
For each of the expression variants (manifest constants, function calls, and expressions with operators) give an example with a specified context.

**Truc-notion graph**

![Figure 5.12: Dependencies of Truc Expression](image-url)
5.2.11 Feature

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Member, Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Argument passing, Class, Object</td>
</tr>
<tr>
<td>Notions</td>
<td>Feature, Attribute, Constant, Export status, Feature declaration, Feature signature, Function, Local variable, Procedure, Result, Return type</td>
</tr>
</tbody>
</table>

Summary

“An operation that programs may apply to an object is called a feature” [86]. There are two main types of features: queries, which access an object and return a value or reference, and commands, which may modify objects. Creation procedures define a third category for features. They initialize newly created objects. Some features are commands and creation procedures.

Role

The features of a class define the operations that allow clients to modify or access data. The notion of feature is one of the fundamental concepts of object-oriented programming.

Applicability

Need to offer clients of a class access to the data of an object or a way to modify the object.

Benefits

- Defining a command encapsulates a set of instructions and makes it available to client classes; it thus favors reuse.
- Defining a query allows clients to access data (the data may be stored within the object or calculated by the query).
- The state of an object can only be viewed and modified through features. The features thus define an interface to a class and help abstract a class from its representation.

Pitfalls

Defining commands offers clients the possibility to change the properties of an object. The programmer must make sure that the available features ensure the consistency of the object at run time.

Examples

Consider a BANK_ACCOUNT class. Typical queries are owner and balance, typical commands are set_owner, withdraw, and deposit.
Common confusions

- **Prevailing assignment.** Very often, the routine bodies of the first examples contain only assignment statements using entities of expanded types. This fosters the impression that the work done in routines solely relies on assignment instructions. [62]

- **Name identity.** Classes with an attribute `name` or `id` may foster the misconception that this attribute is the identity of the object (such that only one object can exist with a certain `name` or `id`) or that the `name` attribute is similar to an entity that refers to the object. This may lead to a number of misconceptions; see Truc “Object” 5.2.21. [62]

- **Attributes.** Feature bodies that modify attributes often lead to the misconception that the attribute is duplicated as a local variable with the same identifier and modifications happen on the local duplicate. The misconception appears in various forms differing with respect to how the attribute is affected: the modification affects both the local variable and the attribute simultaneously, the attribute is updated at the end of the feature execution, or the attribute is unaffected. [119]

- **Side effects.** A common misconception is that writing queries with side effects is good practice. The use of procedural languages prior to instruction in object-oriented programming additionally fosters this opinion.

- **Dynamic scoping reinvented.** Novice programmers sometimes believe that instructions inside a feature body may use entities that are declared outside of their scope. For example, they try to write code that uses a local variable of a calling routine and expect the program to search for it in the calling routine if the current routine has no entity with this name. [41]

- **Function result.** Certain programming languages such as Eiffel provide a reserved entity `Result` that makes it possible to return a value from a function. The programming language treats the `Result` entity as a regular entity available for use in instructions and expressions. Several errors and peculiarities appear in connection with the `Result` entity. In particular, novices tend to forget to create or attach the resulting object. Another common issue is that they only use `Result` in the last instruction of a feature to assign it the value of another local variable that was used to calculate the outcome.

- **Undeclared variables.** Novices often forget to declare local variables that they use in their code. They generally have the tendency to use more local entities than advanced programmers.
Examples include routine bodies with feature calls (not only assignments). [62]

Examples include classes with *id* or *name* attribute that have multiple instances with similar name or id.

Examples show routine bodies that modify an attribute and explain how and when changes affect the attribute.

Examples show valid and invalid use of entities in the scope of a feature.

Examples show functions that require creation of result.

Examples show use of *Result* as regular entity.

Examples always include declaration of used entities.

**Sample questions**

Design a class *COLOR* that represents RGB colors. What are the features of the class?

**Truc-notion graph**

![Figure 5.13: Dependencies of Truc Feature](image-url)
5.2.12 Feature call

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Method invocation, Message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Feature, Object, Argument</td>
</tr>
<tr>
<td>Notions</td>
<td>Feature call, Multi dot feature call, Simple qualified feature call, Target, Unqualified feature call, Void call</td>
</tr>
</tbody>
</table>

**Summary**
Feature call is the mechanism of applying a feature to a target object [86]. The target may be explicitly defined through an expression, which at run time will be attached to a certain object. If no explicit target is given, the target is the Current object. Feature calls may contain arguments.

**Role**
Feature call is the “basic mechanism of object-oriented computation”. In an object-oriented software system, all computation is achieved by calling features on objects and no software element will ever be executed except as part of a feature call [83].

**Applicability**
Need to modify an object or access object data or state.

**Benefits**
- Fundamental to create running programs.
- Favors reuse of code by outsourcing a set of instructions into a feature and replacing them by a single feature call.

**Pitfalls**
Using non-pure queries (queries that change the state of an object) in feature calls may produce side effects. Side effects may lead to mysterious bugs that are extremely difficult to analyze and fix.

**Examples**
Consider a class `COORDINATE` with attributes `x: REAL` and `y: REAL`, and procedures `set_x_y (u, v: REAL)` and `translate (a, b: REAL)`, and a query `distance (other: COORDINATE): REAL`. Given that `p1` and `p2` are of type `COORDINATE` and have been instantiated, sample feature calls could be:

```plaintext
p1.set_x_y(17.2, 3.7)
p2.set_x_y (p1.x, 0)
p1.translate (-0.5, p1.distance (p2))
```
5.2. MODELING OBJECT-ORIENTED PROGRAMMING

Common confusions

- **Static-text execution.** Many novices have an incorrect model of control flow for feature calls. It is difficult for them to understand that a feature call results in the suspension of the calling feature (caller), a transfer of the execution control and sometimes data to a **new and unique specimen** of the called feature (callee), then a transfer of control and data back to the caller after the callee has finished. This knowledge is especially important when recursion is introduced. [48]

- **Availability of features.** “Calling a non-existent feature” seems a recurring mistake [89]. Writing feature calls demands thorough knowledge of what features are available for an entity. This requires looking up the type of the entity and the sufficiently exported features.

- **Wrong target.** Various errors in connection with feature calls originate from specifying wrong targets. For example, students use an unqualified feature call where a qualified feature call is necessary, or they explicitly specify the target to be **current** for features that are only available for unqualified feature calls. [89]

- **Object state.** Novices have difficulties in understanding that feature call instructions modify object state. [111]

- **Query as command.** It seems difficult for novice programmers to distinguish between queries and commands for feature call instructions. In certain programming languages it is possible to use a query (with a resulting value/object) in a statement, but the result is then lost. This is a common mistake. [63]

Example questions

Consider a class **WORD** that has an attribute **word: STRING**, a procedure **set_word(s: STRING)**, and a procedure **print** that displays the word on a console. It also has a function **substring(i, j: INTEGER): WORD**, which returns a new **WORD** object containing the part of the original word defined through the indices **i** and **j**. Given is an entity **w: WORD**. Write an instruction that sets the **word** entity of **w** to “summertime”. Then write an instruction that uses the **substring** function to extract “time” from **w**.
Truc-notion graph

Figure 5.14: Dependencies of Truc Feature call
5.2.13 Genericity

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Generic programming, Type parameterization, Template, Parametric polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Class, Inheritance</td>
</tr>
<tr>
<td>Notions</td>
<td>Genericity, Actual generic parameter, Constrained genericity, Formal generic parameter, Generic data structures, Unconstrained genericity</td>
</tr>
</tbody>
</table>

Summary
Genericity is a mechanism allowing classes and, in some languages, features to have generic parameters. A generic class is a template for many possible derived types and includes a formal generic parameter. To obtain a generically derived type, actual generic parameters replace the formal generic parameters. Genericity can be unconstrained (where any type can replace the formal generic parameters) or constrained, in which case the set of actual generic parameters is restricted to subtypes of the type mentioned in the constraint.

Role
Genericity makes it possible to define flexible, yet type-safe container data structures.

Applicability
Need to build data structures that may contain elements of arbitrary types.

Benefits
- Favors reuse.
- Retains type safety.

Pitfalls
Certain programming languages such as C++, implement genericity using templates. In these cases, the compiler generates a new class or function for every variant of type parameters, which may lead to bloated code.

Examples
Consider a generic STACK [G] class. The formal generic parameter G is used in the class text of stack, for example in feature signatures such as top: G. To use a generically derived type of STACK [G] in another class, G needs to be replaced by an actual generic parameter, such as in:

```
  i_stack: STACK [INTEGER]; s_stack: STACK [STRING]
p_stack: STACK [PERSON]; si_stack: STACK [STACK [INTEGER]]
```
Common confusions

- **Genericity vs. inheritance.** In programming languages that have a common ancestor class for all classes (such as `Object` in Java or `ANY` in Eiffel), creating container classes for this ancestor class is common because they may contain objects of any descendant type. For programmers who are used to this practice, it may seem that genericity replaces polymorphism.

- **Available features.** For novices it is sometimes difficult to understand what features can be called on entities declared of the type given as formal generic parameter. In the case of unconstrained genericity, only features inherited from a common ancestor class (`ANY` in Eiffel) can be used; constrained genericity broadens the range of available features to those available in the type constraint.

- **Missing actual generic parameter.** A common mistake of students is to forget to declare the actual parameter of a generic container.

- **Propagated formal generic parameter.** Novices often have troubles with propagated generic parameters. As an example, a generic class `LIST_CELL[G]` may be used as an attribute in a class `LIST`, but then `LIST` also needs to be a generic class with a generic parameter.

Examples show difference between polymorphic containers and generic containers.
- Examples discuss missing actual generic parameter.
- Examples include propagated generic parameters.
- Examples show valid and invalid use of available features in generic classes.

Sample questions

Write a generic class `FIFO_QUEUE` with a command `put` that adds an element to the end of the queue, a command `remove` that removes the first element, and a query `item` that returns the first element of the queue.
Truc-notion graph

Figure 5.15: Dependencies of Truc Genericity
5.2.14 **Hash table**

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Hash map, Associative array, Mapping, Lookup table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Algorithm, Array, Genericity</td>
</tr>
<tr>
<td>Notions</td>
<td>Hash table, Closed hashing, Collision resolution, Hash function, Hash table performance, Open hashing</td>
</tr>
</tbody>
</table>

**Summary**

A hash table is a linear container data structure that stores values at a location calculated from an associated key. It supports efficient lookup of values given their keys. For lookup, the key is transformed into a hash using a hash function; the hash is then used as an index in an array to determine the location of the value. If two keys produce the same hash, then there is the need for collision resolution. There are various approaches to collision resolution; with separate chaining, each slot in the table is a linked list storing multiple key and value pairs, with open addressing, alternative locations are searched using a fixed scheme.

**Role**

Hash tables are most useful when a large number of objects need to be stored and when the size of the data set is predictable.

**Applicability**

Need to access values by a certain key, from which a hash can be computed.

**Benefits**

- Efficient lookup based on a key (approaching constant time).
- Efficient insertion (approaching constant time).

**Pitfalls**

Depending on the quality of the hash function, a hash table can be filled up to 70% to 80%, after which too many collisions occur and its efficiency decreases. Once this happens, the hash table needs to be resized. This can be a very expensive operation.

**Examples**

A typical example for the use of a hash table is a phonebook. A hash function applied to the person’s last name calculates the index of the phone entry.
Common confusions
• Hash function. Many novices have troubles understanding that the choice of hash function and the characteristics of the stored data determine the efficiency of hash tables.

Examples include showcases of inefficient hash tables (due to data properties, inadequate hash function, or occupancy of the hash table).

Sample questions
Write a small application that stores usernames and passwords in a hash table. The username should be the key, by which fast access to the corresponding password is available.

Truc-notion graph

Figure 5.16: Dependencies of Truc Hash table
5.2.15 Information Hiding

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Encapsulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Dynamic binding, Feature</td>
</tr>
<tr>
<td>Notions</td>
<td>Information hiding, Encapsulation</td>
</tr>
</tbody>
</table>

Summary
Information hiding is the ability to prevent that certain aspects (data or behavior) of a class are accessible to its clients [83]. It is a technique for designing classes by selecting a subset of the features as the official information made available to clients. Access control mechanisms and polymorphism help apply information hiding techniques. The term encapsulation is often used interchangeably with information hiding. Not all agree on the distinctions between the two [4].

Role
Minimizes the coupling between clients and suppliers and allows changes to the private features of a class without affecting its clients.

Applicability
Need to restrict access to data and behavior and hide implementation details.

Benefits
- Eases reuse of code.
- Protects clients from unforeseeable changes in the supplier.
- Avoids that clients need to learn about implementation details.

Pitfalls
Inheritance and polymorphism can introduce extra levels of indirection, which makes testing and maintenance more difficult [51].

Examples
Consider a class `COORDINATE` with the exported features `set_x_y` for setting the coordinates in Cartesian format, and `x` and `y` for access to the Cartesian coordinate values. Additional private features `internal_x` and `internal_y` are used to store the coordinate. Thanks to information hiding, clients may ignore whether the supplier class uses the Cartesian or polar format to store the coordinate values and the supplier is able to change its implementation without affecting its clients.

Common confusions
- **Code length.** The concept of information hiding very often makes program text longer or results in more classes than if it is ignored.
“Many students consider reducing the number of lines of code and reducing the number of classes to be more important than encapsulation.” [41]

Examples discuss advantages of information hiding compared to length of class text.

**Sample questions**
Consider a class `RECTANGLE` that represents rectangular shapes. The dimension and location of a rectangle in 2-dimensional space can either be stored through two of its corners or through one corner and the horizontal and vertical side lengths. Design and implement a class interface that allows a supplier to change the rectangle’s internal representation without affecting its clients.

**Truc-notion graph**

Figure 5.17: Dependencies of Truc Information hiding
CHAPTER 5. TRUCS FOR OBJECT-ORIENTATION APPLIED TO AN INTRODUCTORY PROGRAMMING COURSE

5.2.16 Inheritance

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Generalization, Specialization, Subclassing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Class, Feature</td>
</tr>
<tr>
<td>Notions</td>
<td>Inheritance, Ancestor, Descendant, Precur-sor, Redefined feature, Renaming, Type conformance</td>
</tr>
</tbody>
</table>

Summary
Inheritance is a classification mechanism whereby a class is defined in reference to others, adding all their features to its own [83]. The descendants of a class take over (or inherit) features of its ancestors.

Role
Inheritance is one of the central concepts of the object-oriented method and has profound consequences on the software development process [83]. Inheritance enables programmers to build software satisfying the open-closed principle, such that the classes are open for extension by inheritance, while being closed and ready for client use.

Applicability
Need to create a specialization of an existing class by adding data or behavioral aspects that are not part of the inherited class; need to alter the existing implementation of routines; need to reuse code existing in a class (usually called implementation inheritance).

Benefits
- Fosters reuse of code and helps reduce complexity of systems.
- Fosters extendibility of existing class hierarchies.

Pitfalls
If an inheritance graph is deep, pieces of conceptually related code are located in non-contiguous parts of the program [24]. This makes debugging difficult since a programmer needs to flip between the class definitions in order to follow the control flow of the program.

Examples
Consider classes representing various polygonal shapes with common features such as a `draw` routine for display on the screen and a `bounding_box` function that returns the smallest rectangle holding the shape. In the inheritance structure, `POLYGON` as the topmost element implements the two features by iterating through its list of points. `RECTANGLE` inherits from `POLYGON` and provides a specialized version of `bounding_box` that directly returns its dimensions without iterating through the points.
Common confusions

- **Inappropriate use of inheritance.** A common confusion is that using inheritance is per se good and that its application should occur as often as possible. In particular, the use of inheritance hierarchies to express compositions is a common error. [26]

- **Inheritance vs. Instances.** Some novices confuse subclasses with instances. As an example, when asked to program an application with 5 buttons of differing color, they use a common ancestor class `COLOR_BUTTON` with subclasses `RED_BUTTON`, `CYAN_BUTTON`, etc. [122]

- **Taxomania.** Novices sometimes use inheritance and deferred classes over-enthusiastically resulting in complicated hierarchies. For example, they produce classes `MAN` and `WOMAN` as subclasses of `PERSON` instead of adding an attribute `gender`. [83]

□ Examples show disadvantages of inadequate use of inheritance.
□ Examples exclude taxonomic hierarchies.

**Sample questions**

Extend the class hierarchy of polygons with a class `SQUARE`. Where would you put it and what features would you redefine?

**Truc-notion graph**

![Figure 5.18: Dependencies of Truc Inheritance](image-url)
5.2.17 Instruction

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Statement, Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Assignment, Conditional, Expression, Feature call, Loop, Object creation</td>
</tr>
<tr>
<td>Notions</td>
<td>Instruction, Assignment instruction, Call instruction, Conditional instruction, Creation instruction, Loop instruction</td>
</tr>
</tbody>
</table>

**Summary**

An instruction denotes a basic operation to be performed during the program’s execution [86]. There are several variants for instructions, for example procedure call instructions, assignment, creation, conditional, and loop instructions. Conditional and loop instructions are compound instructions because they may contain instructions as part of themselves (for example as part of the loop body).

**Role**

There are two fundamental mechanisms making up programs: describing actions and describing values (on which the actions will operate or which they produce). Instructions cover the description of actions.

**Applicability**

Need to describe an action to be performed at run time.

**Benefits**

- Provides a flexible way of describing actions that define programs.

**Pitfalls**

None.

**Examples**

See the according Truc Feature call 5.2.12, Assignment 5.2.4, Object creation 5.2.22, Conditional 5.2.6 and Loop 5.2.19.

**Common confusions**

See the according Truc Feature call 5.2.12, Assignment 5.2.4, Object creation 5.2.22, Conditional 5.2.6 and Loop 5.2.19.

**Sample questions**

Given a class text (such as LIST [G] from EiffelBase), identify the instructions and their variants (loop, conditional, procedure call, assignment, or object creation).
Truc-notion graph

Figure 5.19: Dependencies of Truc Instruction
5.2.18 Linked list

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Algorithm, Genericity, Reference</td>
</tr>
<tr>
<td>Notions</td>
<td>Linked list, Cursor, List operation, List performance</td>
</tr>
</tbody>
</table>

Summary
A linked list is a linear container data structure mostly used for dynamic collections with frequent element insertions and removals. A linked list consists of a set of nodes, which store the data objects. Every node is linked to the next (and possibly the previous) node of the list via a reference. There are several types of linked list such as singly linked lists, doubly linked lists, and circularly linked lists.

Role
Linked list is one of the fundamental container data structures and useful for the implementation of other container data structures such as stacks and queues.

Applicability
Need to store a dynamic collection of elements with frequent insertions and deletions.

Benefits
- Constant time insertion and deletion of nodes at any list location.
- Unbound structure (no need for resizing).

Pitfalls
Random access (for example accessing the \( i \)-th element) takes linear time. If this occurs frequently, other data structures are better suited. Linear linked list traversal often is slower than for arrays, whose elements are stored at spatially close locations.

Examples
The figure below shows the structure of a doubly linked list. Every node of the list contains an attribute \( \text{previous} \) and an attribute \( \text{next} \). Assume that

![Figure 5.20: Example of an integer doubly linked list structure](image)

we want to insert a new node containing the number 45 after the node with value 23. This results in an update of four attachments: the new node’s
previous entity is attached to the node containing 23, its next entity refers to the node containing 7, next of the node containing 23 references the new node, and previous of the node containing 7 points to the new node.

Common confusions

- **Corner cases.** A common mistake with linked lists is to write the code for the “usual case”, but forget to handle the boundary situations such as the first and last element of a list, or an empty list. [34]

- **Container creation and initialization.** Programmers tend to forget to create and initialize container data structures (see also Truc 5.2.22).

  □ Examples include handling of empty lists, lists with only one element.
  □ Examples include special treatment for first or last element.
  □ Examples stress the need to create and initialize containers.

Sample questions

1. Using the doubly linked list of Figure 5.20, explain what references need to be reattached (1) if a new node is inserted at the end of the list, (2) if the element containing 17 is removed from the list.

2. Implement a singly linked list with an interface that provides an internal cursor and offers insertion and deletion at the cursor position.

Truc-notion graph

![Figure 5.21: Dependencies of Truc Linked list](image-url)
5.2.19 Loop

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Instruction, Primitive type</td>
</tr>
<tr>
<td>Notions</td>
<td>Loop, Exit condition, Loop invariant, Loop variant, Nested loop</td>
</tr>
</tbody>
</table>

**Summary**
A loop is a control structure that repeats a set of instructions until a specific exit condition is met (including repetition of 0 times if the exit condition immediately holds). It supports the strategy of solving a problem on a (possibly trivial) subset of its domain and extending the solution repeatedly until it covers the whole domain [86]. A loop consists of a loop body containing the instructions, an exit condition, and optional loop initialization instructions.

**Role**
The loop control structure is one of the fundamental control flow statements and needed for the implementation of many algorithms.

**Applicability**
Need to repeatedly execute a set of instructions while the number of repetitions might only be known at run time.

**Benefits**
- Allows repetition of a set of instructions without rewriting them and thus improves the readability of programs.
- In most cases more efficient than recursion.

**Pitfalls**
In comparison to recursion, iteration sometimes results in less elegant solutions.

**Examples**
Consider a procedure `print_triangle` that takes as argument `n` an integer and produces for `n = 5` the following output:

```
*  
** 
*** 
**** 
*****
```

One of the algorithms to produce the tree output uses the concatenation operator (`+`-operator) for `STRING` and the feature `count` returning the num-
ber of characters in a STRING. The algorithm outputs one line at a time using the local variable s to store the current character sequence (e.g. three stars *** for line 3) and adapts it for the next line by adding another * symbol. The loop is executed until the number of outputted lines is larger than n:

```plaintext
print_triangle (n: INTEGER)
  -- Print a triangle of size 'n'.
local
  s: STRING
  do
    from
    s := "*
    until
    s.count > n
  loop
    io.put_string (s)
    io.put_new_line
    s := s + "*"
  end
end
```

Common confusions

- **Border cases.** Many problems with loops concern the incorrect order of statements inside a loop, which may result in an execution that starts or terminates incorrectly. [29]

- **Interrupted loop.** It is a common misunderstanding that the system during the execution of a loop constantly checks the termination condition and will, as soon as it changes, interrupt the execution of the loop. In reality, the termination condition is only evaluated once in every cycle of the loop. [29, 14, 100]

- **Endless loop.** Producing endless loops due to incomplete or incorrect termination conditions is a common error.

- **Termination condition vs. continuing criterion.** Some languages use a continuing criterion in the loop instruction; others use a termination condition. These variations may lead to confusions.

□ Examples include loops with same instructions but differing order resulting in different results and different termination.
□ Examples show when termination condition is evaluated.
□ Examples discuss the difference between continuation criterions in contrast to termination conditions.
Sample questions
Write a feature that takes an INTEGER as an argument and prints a diamond with the width specified by the argument. The example below shows the diamond of width 7.

```
*  
***  
*****  
*******  
******  
***  
*  
```

Truc-notion graph

Figure 5.22: Dependencies of Truc Loop
5.2.20 Multiple inheritance

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Multiple subclassing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Notions</td>
<td>Multiple inheritance, Repeated inheritance, Name clash</td>
</tr>
</tbody>
</table>

Summary
Multiple inheritance is the unrestricted form of inheritance, whereby a class may have any number of ancestors [83]. Multiple inheritance may produce certain ambiguities that need to be resolved. As an example, name clashes occur when two features inherited from different classes have the same name; renaming one of them solves this problem. In the case of repeated inheritance, a class inherits from another through two or more paths. This may lead to additional ambiguities.

Role
Multiple inheritance makes it possible to combine multiple abstractions.

Applicability
Need to combine multiple abstractions.

Benefits
- Favors extendibility.
- Favors reuse.

Pitfalls
Multiple inheritance is a powerful mechanism that, when used imprudently, may lead to increased complexity of a system. It may also lead to a slight performance overhead compared to single inheritance [129]. Another pitfall is known as the “diamond problem”, which appears in the case of repeated inheritance. To resolve conflicts and carefully design duplicated and joined features requires experience on the side of the programmer.

Examples
Consider a class `PHONE` with features `save_contact`, `hang_up`, `call`, and `take_call` and a second class that represents digital cameras `CAMERA` with features such as `take_photo` and `save_photo`. Most of nowadays cell phones are in fact both phones and cameras and for example are able to store a taken photo together with a contact. This is one of the examples where multiple inheritance appears when items may belong to two categories.

Class `STRING` is an example of a class that needs to possess two (or
more) structural properties. It inherits from `Comparable`, which supports comparisons of two instances (e.g. to sort a list of strings in alphabetical order). `String` also inherits from `Hashable` so that it can act as a key in a hash table.

**Common confusions**

- **Client substitute.** It is a common mistake to use multiple inheritance for the client relationship. This is already a danger if single inheritance is used, but increases with multiple inheritance.

☐ Examples discuss misuse of multiple inheritance.

**Sample questions**

Assume that one of your colleagues states “Multiple inheritance is not needed. All instances of multiple inheritance could be replaced by client relationships.” Analyze this statement. In what ways is it true and in what ways is it false?

You are designing a pong game and want to reuse an existing figure hierarchy with classes `Figure` and descendants `Rectangle` and `Circle` and an existing facility that describes moving objects `Movable` with features `speed` and `direction`. Describe the class hierarchy for `Ball` and `Racket`.

**Truc-notion graph**

Figure 5.23: Dependencies of Truc Multiple inheritance
5.2.21 Object

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Class</td>
</tr>
<tr>
<td>Notions</td>
<td>Object, Instance, Value</td>
</tr>
</tbody>
</table>

Summary

“An object is a software machine allowing programs to access and modify a collection of data” [86]. “An object encapsulates the existence, state and behavior of an entity. […] The existence is limited by the instantiation of an object and its destruction in garbage collection. The state is manifested in the attributes, and the behavior is a result of feature calls that include the creation and destruction of local variables.” [118]. Objects can be manipulated through a set of well-defined operations given in its generating class and they only exist at run time.

Role

Objects are individuals of classes existing at run time. A running program thus is a set of interacting objects. The notion of object is so fundamental that it gives its name to the style of object-oriented programming.

Applicability

The concept of object is not applied in a specific situation. However, the knowledge is fundamental for the design of an object-oriented system and to understand and follow program execution.

Benefits

• Objects make it possible to view a program execution as a system of interacting objects where each object has its properties and responsibilities.

Pitfalls

Following program flow using objects is not an easy task. Information hiding techniques (such as inheritance and polymorphism) add to the difficulty.

Examples

Consider a class PERSON with attributes landlord, loved_one, and name, a creation procedure set_name, and commands set_loved_one and set_landlord. a, f and s are local variables of type PERSON. Executing the instructions below results in an object structure as indicated in Figure 5.24.

```plaintext
create a.set_name("Almaviva")
create f.set_name("Figaro")
create s.set_name("Susanna")
```
CHAPTER 5. TRUCS FOR OBJECT-ORIENTATION APPLIED TO AN INTRODUCTORY PROGRAMMING COURSE

Common confusions

- **Class versus object.** Many students have troubles distinguishing between the concept of class and the concept of object (or instance). This is particularly dangerous if the series of presented examples only show a single instance of each class. To overcome the misconception that a class can have only one instance, it is useful to work with examples that always include multiple instances of each class. [62]

- **Equality.** A common misconception is that two objects of the same class with the same state are the same object. [62]

- **Identity.** A common misconception is that two objects with the same value for a `name` attribute are the same object. [62]
Teaching material explains difference between class and object. Examples show two instances of the same class with the same state and explain that they are different objects. Examples show two instances of the same class with same id or name and explain that they are different objects.

Sample questions
Consider a class `CHARACTER` with attributes `loved_one`, `killed_by`, and `name`, a creation procedure `set_name`, and commands `set_killed_by` and `set_loved_one` that each take an argument of type `CHARACTER` and attach `loved_one` respectively `killed_by` to the passed object. `r`, `j` and `p` are local variables of type `CHARACTER`. What is the object structure after executing the instructions below?

```plaintext
create r.set_name ("Romeo")
create j.set_name ("Julia")
create p.set_name ("Paris")
r.set_loved_one (j)
r.set_killed_by (r)
j.set_loved_one (r)
j.set_killed_by (j)
p.set_loved_one (j)
p.set_killed_by (r)
```

Truc-notion graph

Figure 5.25: Dependencies of Truc Object
5.2.22 Object creation

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Instantiation, Clone, Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Feature, Feature call, Reference</td>
</tr>
<tr>
<td>Notions</td>
<td>Object creation, Creation procedure, Creation procedure declaration</td>
</tr>
</tbody>
</table>

Summary

Object creation is the mechanism of creating a concrete instance of a class at run time. Object creation is triggered by a creation call, which consists of a target and a creation procedure. It results in the creation of the object (made of a collection of fields, one for each attribute), the initialization of each field with default values, attachment of the target reference to the newly created object, and a call to the creation procedure [83].

Role

Objects are fundamental to object-oriented programming and used as targets or arguments to feature calls. The creation of objects is a prerequisite for executing a system.

Applicability

Need to add an object to the object structure.

Benefits

- Makes it possible to model changing properties of systems.
- Supports dynamic data structures.

Pitfalls

The creation of objects always entails memory consumption. Although in many modern programming languages the garbage collector takes care of freeing unused memory, it does not prevent programs from using excessive amounts of memory, nor from cycling through too much memory (creating many objects that are not referenced any more afterwards), nor from holding on to objects that are not needed any more.

Examples

Consider class COLOR with creation procedures make_black and set_rgb \((r, g, b: \text{INTEGER})\) and two variables \(c1\) and \(c2\) of type COLOR. Typical creation calls in Eiffel could look like this: `create c1.make_black` or `create c2.set_rgb(100, 0, 255)`. The result of these calls is the following part of an object structure (see Figure 5.26).
Common confusions

- **Automatic instantiation.** A general misconception is that objects are created by default, when an entity is declared. The problem results from real world analogies in which object creation is not necessarily explicitly handled. Another common misconception is that the declaration of a creation procedure is sufficient to instantiate an object. [26, 110]

- **Arguments.** Students generally seem to find argument passing a difficult concept. Therefore, a creation instruction that uses a creation procedure with arguments is also considered difficult. [110]

- **Composed creation.** Attributes of user-defined types (as opposed to basic types) may lead students to believe that there is no need to create instances of a class with user-defined attributes, if its user-defined components were already created. [110]

Examples show difference between declaration and creation instruction.
Examples use default and user-defined creation procedures.
Examples emphasize the need for creation of a user-defined object that has attributes of user-defined types after the objects for the attributes have been created.

Sample questions

Design a class `TIME` that represents a time of day. It should store the time of day in 24-hour format in features `hour` and `minute`, but offer queries to access the time in AM/PM format as well. There should be two creation procedures: one for entering it in 24-hour format and one in AM/PM format. Make sure to specify correct contracts for the class. Show the use of its creation procedures with an example.
CHAPTER 5. TRUCS FOR OBJECT-ORIENTATION APPLIED TO AN INTRODUCTORY PROGRAMMING COURSE

Truc-notion graph

Figure 5.27: Dependencies of Truc Object creation
5.2.23 Polymorphism

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Subtyping polymorphism, ad-hoc polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Argument passing, Assignment, Class, Gener-</td>
</tr>
<tr>
<td></td>
<td>icity, Inheritance, Object creation</td>
</tr>
<tr>
<td>Notions</td>
<td>Polymorphism, Dynamic cast, Dynamic type,</td>
</tr>
<tr>
<td></td>
<td>Polymorphic argument passing, Polymorphic</td>
</tr>
<tr>
<td></td>
<td>assignment, Polymorphic attachment, Poly-</td>
</tr>
<tr>
<td></td>
<td>morphic creation, Polymorphic data structure,</td>
</tr>
<tr>
<td></td>
<td>Static type</td>
</tr>
</tbody>
</table>

Summary
Polymorphism is the ability for an element of the software text to denote, at run time, objects of two or more types [83]. Inheritance supports polymorphism. It appears in two main forms: Polymorphic attachment and polymorphic data structures. Polymorphic attachment denotes the ability of an entity (declared to be of a certain static type) to become attached to objects of various types (descendants of the static type). It is based on polymorphic assignment, polymorphic argument passing, and polymorphic creation. Polymorphic data structures combine the power and flexibility of genericity and polymorphism to offer data structures that may contain objects of various types conforming to the declared actual generic parameter.

Role
Polymorphism is one of the central mechanisms supporting information hiding; it allows the supplier to hide implementation details from clients by declaring an entity to be of a certain type while at run time it may be attached to an object of a subtype. This makes it also possible to replace the implementation at run time. Polymorphism is applied in connection with many design patterns (such as the composite pattern or the strategy pattern).

Applicability
Need to attach an object of a subtype to an entity without disclosing it to clients.

Benefits
- Favors reuse.
- Protects clients from unforeseeable changes in the supplier.
- Favors extendibility.
Pitfalls
There are two situations in connection with polymorphism and dynamic binding that may lead to so called catcalls (calls to features that have a change in availability or type). In the first scenario, the descendant retypes a formal argument of a routine to be of a more specialized type than in the ancestor. In the second scenario, the descendant changes the export status of a feature to be more restricted than in its ancestor. Both of these scenarios may lead to invalid feature calls at run time.

Examples
Consider a graphics system that has a common ancestor class `FIGURE` with a deferred feature `draw` and many variants of figures such as `RECTANGLE`, `CIRCLE`, and `LINE`, which all provide their implementation of `draw`. Assume that one of the figures may be selected for manipulation in a drawing program and made available through the entity `selected_figure` declared of type `FIGURE`. This entity may at run time actually denote through polymorphic attachment an object of a subtype of `FIGURE` (for example a `RECTANGLE` object).

Common confusions
- **Static vs. dynamic type.** The concepts of an entity’s static and its dynamic type and the difference between the two are potentially confusing.
- **Polymorphism vs. dynamic binding.** It seems particularly difficult to understand that the dynamic type influences through dynamic binding the version of feature that is executed while the static type determines the features that are available for a polymorphic feature call.

☐ Examples explain difference between dynamic and static type.
☐ Examples show valid and invalid feature calls to illustrate the role of static type.
☐ Examples include feature calls explaining dynamic binding.

Sample questions
Consider a class hierarchy that represents vehicles with a common deferred ancestor class `VEHICLE` and classes `CAR`, `BIKE` and `LINE_VEHICLE`, which inherit from it. The class `LINE_VEHICLE` is also deferred and has an effective subclass `TRAIN`. Given are the following entity declarations:

```
v: VEHICLE; c: CAR; b: BIKE; l: LINE_VEHICLE; t: TRAIN
```

Which of the following assignments are valid?

```
v := l
```
5.2. MODELING OBJECT-ORIENTED PROGRAMMING

\[ t := b \]
\[ l := t \]
\[ v := c \]
\[ b := v \]

Truc-notion graph

Figure 5.28: Dependencies of Truc Polymorphism
5.2.24 Primitive type

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Basic type, Built-in type, Value type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Class, Expression</td>
</tr>
<tr>
<td>Notions</td>
<td>Primitive type, Boolean, Boolean expression, Character, Integer, Integer expression, Manifest constant, Real, String</td>
</tr>
</tbody>
</table>

**Summary**

Primitive types are data types provided by the programming language. The available types depend on the programming language implementation. The most common types are integer for whole numbers, character for single letters, digits, symbols or control characters, Boolean for the values True and False, and floating point numbers for real numbers with a limited fractional part. Primitive types typically offer operations that are efficiently implemented and have a fixed size in memory leading to a limited range of possible values. Most programming languages suggest copy semantics for primitive types.

**Role**

User-defined classes often have attributes of primitive types. Primitive types heavily influence the data aspect of an object.

**Applicability**

Need to store data as one of the available primitive types.

**Benefits**

- Operations on primitive types are implemented very efficiently.

**Pitfalls**

Trying to store a number that is larger or smaller than can be represented in the storage space leads to program crashes. As an example, the Ariane 5 crash was due to an integer overflow.

**Examples**

Typical storage sizes for integer numbers are 8, 16, 32, and 64 bits. The range of the values that an object of this type may hold depends on this size and on whether it may be negative (signed) or not (unsigned). For an 8-bit signed integer the range is $-128$ to $127$, while for a 16-bit signed integer it is $-32768$ to $32767$.

**Common confusions**

- **Division by zero.** Very often programmers tend to forget that division by zero results in a run-time exception and that the denominator
needs to be checked for this case before executing the division.

- **Overflow and underflow.** Another typical error is that the result of an arithmetic operation produces an overflow or an underflow exception.

- **Strings as expanded types.** Depending on the implementation of strings in a programming language, `STRING` may mistakenly be perceived as an expanded type although it is a reference type. This results in mistakes such as using reference equality instead of value equality and forgetting to create strings.

- **Boolean expressions in assignments.** Novices tend to circumvent complex Boolean expressions as a source in an assignment by using a conditional. For example, they will write
  
  ```
  if <cond> then b := True else b := False end
  ```
  
  instead of
  
  ```
  b := <cond>.
  ```

- Examples with numbers cover division by zero, overflow and underflow exceptions.
- Examples show most common errors with strings.
- Examples contain complex Boolean expressions in assignment source.

**Sample questions**

Calculate the number of possible values that a 32-bit integer may contain. Can you state a general formula for calculating this?

**Truc-notion graph**

![Truc-notion graph](image)

Figure 5.29: Dependencies of Truc Primitive type
5.2.25 Recursion

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Algorithm, Feature call, Genericity, Loop, Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Recursion, Backtracking, Embedded recursion,</td>
</tr>
<tr>
<td></td>
<td>Direct recursion, Indirect recursion, Recursion</td>
</tr>
<tr>
<td>Notions</td>
<td>elimination, Recursive algorithm, Recursive</td>
</tr>
<tr>
<td></td>
<td>call, Recursive data structure, Tail recursion</td>
</tr>
</tbody>
</table>

Summary
Recursion is a method of defining concepts (such as an algorithm or a data structure) in such a way that the definition involves one or more instances of the concept itself [86]. For recursive algorithms, this requires two steps: (1) define a “base case”, for which the solution is easy to compute, and (2) define a “recursive case”, which breaks down a more complex case such that it eventually arrives at the base case and assembles the overall solution from the partial solutions. A recursive algorithm relies on the ability of routines to call new instantiations of themselves with control passing forward to successive instantiations (active flow of control) and back from terminated ones (passive flow of control). Recursive data structures are data structures with references to instances of the same type and often offer features that use recursive algorithms. [48, 86]

Role
Recursion is one of the central ideas in computer science and appears in many subject areas. Recursion provides a means to create elegant solutions to many complex problems. Iteration (loops) and recursion are two problem-solving strategies that can be used interchangeably – every iterative algorithm can be transformed into a recursive algorithm and conversely with the addition of stack data structures.

Applicability
Need to solve a complex problem that can be decomposed into subproblems of the same type. Need to develop a data structure that is defined in terms of instances of the same type.

Benefits
- Provides elegant and easy to read program code.
- Helps model recursive data structures.

Pitfalls
Depending on the implementation of recursion in the programming language, it may lead to a performance overhead if compared to an iterative
solution (need to store information on the call stack).

**Examples**
A typical example of a recursive function is factorial. In its definition given below, the top part of the definition represents the base case and the lower part shows the recursive case.

\[
 n! = \begin{cases} 
 1 & \text{if } n \leq 1 \\
 n(n - 1)! & \text{if } n > 1 
\end{cases} \forall n \in \mathbb{N}
\]

The translation of this mathematical definition into a recursive feature is straightforward.

As an example of a recursive data structure, consider a class `PERSON` offering attributes `mother` and `father`. These are again of type `PERSON` and make it possible to generate a genealogical tree. A typical recursive feature of such a data structure could be `is_descendant`, which determines whether the current person is a descendant of another person.

**Common confusions**
- **Looping model.** Many novices mistakenly see recursion as looping. They expect a recursive call to have the effect of restarting the execution of the feature (possibly with new arguments) without control being passed back to the caller after the recursive call has been completed. In reality, a recursive call temporarily suspends the execution of the calling feature, passes control over to a new version of the feature until it is completed, to then continue the execution of the calling feature. [66, 70, 52]
- **Active model.** In this model the recursive calls are correctly evaluated until the base case is reached, but the passive flow (returning from callee to caller and aggregating the result) is forgotten. This misconception stays undetected if the algorithm uses tail recursion without return values, but shows in algorithms using tail recursion with return values or embedded recursion. [52, 48]
- **Step model.** Students with the step model have no concept of recursion and simply evaluate the conditional to calculate the result through the base case. [52]
- **Endless recursion.** A common mistake is to produce an endless recursion, because of missing, incomplete or incorrect non-recursive branches.

□ Examples show step-by-step execution of a recursive routine with emphasis on control flow.
□ Examples include embedded recursion and tail recursion with a return
value to unveil the looping, active, and step mental models.
□ Examples include endless recursion to emphasize importance of correct
non-recursive branches.

Sample questions
Write a recursive function that checks for a given `STRING` whether it is a
palindrome (a word that reads the same from the back as from the front).

Truc-notion graph

![Figure 5.30: Dependencies of Truc Recursion](image-url)
5.2.26 Reference

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Pointer, Link, Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Object, System execution</td>
</tr>
<tr>
<td>Notions</td>
<td>Reference, Attachment, Aliasing, Void</td>
</tr>
</tbody>
</table>

Summary
“A reference is a run-time value that is either void or attached” [83]. If the reference is attached, then it refers to a single object and thus provides a way to identify this object. In most languages, references ensure type safety: a reference of a certain type may only be attached to objects whose type conforms to it. References are orthogonal to containment, where an object instead of storing a reference to another object as attribute, stores the entire object in the attribute using an expanded type.

Role
References are fundamental for the construction of many data structures (such as linked lists or recursive data structures). They also appear in expressions and thus provide a way to exchange information between different parts of a computer program.

Applicability
Need to share objects between different parts of a program, need to construct linked structures.

Benefits
- Retains type safety.
- Provides a mechanism to share objects between different code areas.
- Makes it possible to model recursive data structures (data structures that contain themselves).
- More flexible than containment for expanded types because it does not require resizing operations if the referenced object changes.

Pitfalls
When various parts of a program share objects through aliasing, a change in the object invoked from a certain code area may have a much more global effect than expected by the programmer. This may make debugging difficult. Modeling association with references uses more memory than with expanded types since it requires storing the references additionally to the referenced object.

Examples
Consider a class `COURSE` modeling a course offered at a school. The course should contain an attribute `teacher` of type `TEACHER` storing the teacher
information and an attribute \textit{location} of type \textit{LOCATION} representing the
room where the course is given. If \textit{TEACHER} and \textit{LOCATION} were expanded
types, the object of type \textit{COURSE} would store an object of type \textit{TEACHER}
and an object of type \textit{LOCATION} inside itself. With references, the teacher
object and the location object are stored elsewhere and the course object
contains two references that help reach these external objects. Using ref-
erences makes it possible to reuse the teacher and the location object for a
second course that has the same teacher or location.

\textbf{Common confusions}

- \textbf{Single reference}. A common misconception is that only one entity
can reference an object. [62]
- \textbf{Immutable reference}. A common misconception is that once an en-
tity references an object, it will always reference it. [62]
- \textbf{Two entities are distinct}. A common misconception is that if there
are two different entities, they must refer to two different objects. [62]
  □ Examples include multiple entities that reference the same object.
  □ Examples include entities whose referenced object changes.

\textbf{Sample questions}

Give an example of

1. how references help share objects.
2. a situation where references are absolutely needed to model reality.

\textbf{Truc-notion graph}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure531.png}
\caption{Dependencies of Truc Reference}
\end{figure}
5.2.27 Stack

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Cellar, LIFO-queue, LIFO-list, Yoyo-list, Push-down list, Nesting store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Array, Genericity, Linked list</td>
</tr>
<tr>
<td>Notions</td>
<td>Stack</td>
</tr>
</tbody>
</table>

**Summary**

A stack is a container data structure that applies a last-in first-out policy: at any given time, the most recently added element (the top of the stack) is accessible. Typical stack operations are *push*, which puts an element on top of the stack, *pop*, which removes the top element, and *top*, which accesses the top element. Stacks are usually implemented using either arrays or linked lists.

**Role**

Stacks have many applications at every level in computer systems. For example, feature calls make use of a call stack to store arguments and local variables, but stacks also help efficiently parse source code and evaluate bracket terms, postfix and infix notations.

**Applicability**

Need to access stored data in reverse order.

**Benefits**

- Efficient implementation (storage requirements $O(n)$ for $n$ elements, and time requirement $O(1)$ for its operations).

**Pitfalls**

Certain implementations of stack combine the top element access with the pop operation, because the two very often appear in conjunction. This leads to a non-pure query since it will return a different value every time it is called.

**Examples**

Consider a calculator using the reverse Polish notation (postfix notation). As an example, the expression $(2 + 4) \times 8 - 9$ in infix notation is written as $2 4 + 8 \times 9 -$ in postfix notation. The algorithm may then use a stack to evaluate postfix expressions from left to right: whenever it encounters an operand, it pushes it onto the stack; whenever it encounters an operator, it takes the two top elements of the stack, applies the operator to them, and pushes the result onto the stack.
Common confusions
• **Container creation and initialization.** Students often tend to forget to create and initialize container data structures (see also Truc 5.2.22).

□ Examples stress the need to create and initialize containers.

Sample questions
Implement a calculator that uses the reverse Polish notation.

Truc-notion graph

![Diagram of dependencies of Truc Stack](image)

Figure 5.32: Dependencies of Truc Stack
5.2. MODELING OBJECT-ORIENTED PROGRAMMING

5.2.28 System execution

<table>
<thead>
<tr>
<th>Alternative names</th>
<th>Program execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies</td>
<td>Class, Feature call, Object, Object creation</td>
</tr>
<tr>
<td>Notions</td>
<td>System execution, Call stack, Current object, Object structure, Root class, Root creation procedure, Root object, System entry point, Termination</td>
</tr>
</tbody>
</table>

Summary
System execution is the process by which a system or program is executed. In particular, it describes the entry point of a program (where and how execution starts). There are various strategies for this; many programming languages use a function called `main` to define the high level functionality, create first objects, and call features on them; Eiffel, for example, requires designating a class and one of its creation procedures such that a first instance of this class (the root object) can be created. The execution of the main feature or the root procedure is there to start a process that relies on the mechanisms found in other classes. [86]

Role
Defining a main feature or root procedure supports composing systems of reusable classes.

Applicability
Need to execute, debug, or test a system.

Benefits
- Makes it possible to execute a system.

Pitfalls
Debugging a complex system is difficult because it requires understanding both the object structure and the control flow of a program.

Examples
Assume that you have a class `TEXT_ANALYZER`, which counts occurrences of letters in a given text and makes the frequencies accessible to its clients. You also have a class `FILE`, which offers features to read and write a plain text file. Using these two classes, there could be a range of systems that make use of them. For example, one system reads a file and then runs it through the text analyzer and outputs a set of statistics concerning the letter frequencies. Another system could make use of the text analyzer to decrypt an encoded message based on frequency codes.
CHAPTER 5. TRUCS FOR OBJECT-ORIENTATION APPLIED TO AN INTRODUCTORY PROGRAMMING COURSE

Common confusions

- **Static-text execution.** Many students do not have a correct model of control flow for feature calls. In particular, it seems difficult for them to understand that a feature call results in the suspension of the calling feature (caller), a transfer of the execution control and sometimes data to a new and unique specimen of the called feature (callee), then a transfer of control and data back to the caller after the callee has finished. This knowledge is especially important when recursion is introduced. [48]

Examples show control flow of feature calls.

Sample questions

1. System execution starts with the execution of a main function or the creation of a root object. When does system execution finish?
2. Provide a small example system of which you can explain every execution step.

Truc-notion graph

Figure 5.33: Dependencies of Truc System execution
5.3 Analysis of the dependency graph

5.3.1 Transitive dependencies

This section describes the transitive (direct and indirect) dependencies resulting from the Truc-notion graph of the domain model for object-oriented programming. The discussion distinguishes between outgoing and incoming links of Trucs and notions. The analysis of outgoing links organizes the Trucs according to the number of dependencies (or prerequisites) that they have. This gives an intuition of a Truc’s place in a course; Trucs with many dependencies are more likely to appear towards the end, while Trucs with few dependencies will probably appear at the beginning. When analyzing the incoming links of Trucs, focus shifts to the number of Trucs that rely on a given Truc. This gives an indication of a Truc’s importance; if many other Trucs rely on it, then it is probably central to teaching programming and will reappear throughout a course.

Table 5.1 gives an overview of the Trucs grouped by their transitive outgoing dependencies: if a set of Trucs shares their dependencies, then they are listed in one row.

Outgoing links. The first row of the table shows a core group of Trucs, which constitute a minimal set of requirements for all 28 Trucs appearing in the model. It is composed of nine Trucs: Argument passing, Class, Expression, Feature, Feature call, Object, Object creation, Reference, and System execution. Every member of the core group depends on itself and on all other members. This is an indication for cycles in the domain model (see Subsection 5.3.2). The Trucs Assignment, Inheritance, and Primitive type share the nine dependencies of the core group. They are not part of the core group, because they are not recursively dependent and because they are not mutually depending on each other.

The second set of Trucs with cyclic dependencies consists of Conditional, Loop, and Instruction. They require themselves and each other. Additionally to the nine core Trucs, they also depend on Assignment, and Primitive type. Algorithm has the same dependencies, but does not cyclically depend on itself. This group contains Trucs that can be associated to imperative programming.

All remaining Trucs require Inheritance additionally to the nine core Trucs. It is the only additional requirement for Deferred class, Genericity, and Multiple inheritance. Design by Contract additionally relies on Primitive type, while Polymorphism requires Assignment and Genericity in addition to Inheritance and the core Trucs. Dynamic binding depends on Polymorphism
### Table 5.1: Overview of transitive Truc dependencies

| # Incoming links | Recursion | Stack | Hash table | Array | Linked list | Design by Contract | Polymorphism | Information hiding | Algorithm | Argument passing | Arg. passing | Array | Assignment | Class | Conditional | Deferred class | DbC | Dynamic binding | Expression | Feature | Feature call | Genericity | Hash table | Info. hiding | Inheritance | Instruction | Linked list | Loop | Multiple inh. | Object | Obj. creation | Polymorph. | Prim. type | Recursion | Reference | Stack | System exec. | # Outgoing l. |
|------------------|-----------|-------|------------|-------|-------------|---------------------|--------------|---------------------|-----------|------------------|-----------|------|-------------|-------|------------|---------------|-----|----------------|-----------|--------|-------------|-----------|-----------|------------|-----------|-----------|------------|---------|--------------|--------|----------|----------|-----------|-------|-----------|----------|
| 5                | 28        | 3     | 12         | 28    | 9           | 0                   | 0 1          | 28                  | 26        | 28              | 8         | 0    | 1           | 20      | 11         | 12          | 11  | 12           | 14        | 14     | 14          | 10       | 9         | 9          | 20      | 18         | 17       | 14        | 12        | 14       | 10       | 9        | 20       |
and thus includes all its dependencies; similarly, Information hiding relies on Dynamic binding and shares all its requirements. This group mostly contains advanced object-oriented concepts related to inheritance.

The Trucs representing knowledge about data structures combine the dependencies of the imperative programming group with some of the object-oriented group. Linked list and Array, for example, depend on Algorithm, Conditional, Instruction, Loop, and Primitive type, as well as on Inheritance and Genericity. Hash table additionally requires Array; Stack requires both Array and Linked list; and Recursion depends on Stack.

Incoming links. The nine core Trucs are a prerequisite for all the Trucs of the domain model. This makes them fundamental for teaching object-oriented programming. The second group containing Assignment, Inheritance and Primitive type are required by 12 respectively 10 other Trucs (almost half of all Trucs); Genericity is a requirement for eight Trucs. The second cyclic group containing Conditional, Loop, Instruction, and Algorithm provides a basis for nine respectively five Trucs. Polymorphism, Dynamic binding, Array, Linked list, and Stack are prerequisite to one to three Trucs. The Trucs Deferred class, Multiple inheritance, Design by Contract, Information hiding, Hash table and Recursion do not appear as a requirement for any Truc in the model.

Transitive dependencies of notions. The transitive dependencies between notions exhibit similar characteristics as those of the Trucs. There are ten notions out of the 147 notions that form a core group such that all notions in the model transitively require them. This group contains the notions Argument declaration, Class, Feature, Feature declaration, Feature signature, Formal argument, Generating class, Instance, Object, and Type. Additionally, over half of all notions in the model transitively require the notion Expression. There are 55 notions that are unneeded by other notions.

5.3.2 Cycles

On the notion level, the domain model exhibits five circular dependencies, of which three involve the Truc Argument passing. One of these cycles is a mutual dependency between the notions Argument declaration and Feature signature; another cycle consists of the notions Argument passing, Actual argument, and Feature call; and the third cycle contains the notions Formal argument, Type, Class, Feature, Feature declaration, Feature signature, and Argument declaration. The fourth cycle on notion level involves the Trucs Class and Object and illustrates their close interrelatedness via a path through the notions Class, Object, Instance, and Generating class, back to no-
tion \textit{Class}. The fifth cycle shows the close connection between \textit{Function} and \textit{Result}.

As indicated in Subsection 5.3.1, two groups of Trucs contain cycles in their lists of dependencies. Figure 5.34(a) shows an extract of the dependency graph for the nine core Trucs \textit{Argument passing, Feature call, Feature, Class, Expression, Object creation, Object, Reference, and System execution}. This subgraph exhibits a high interrelatedness of its concepts; in particular, there are multiple pairs of mutual dependencies (such as between \textit{Class} and \textit{Object} or \textit{Feature call} and \textit{Expression}). Figure 5.34(b) shows the second group of Trucs with mutual dependencies between \textit{Instruction} and \textit{Conditional} respectively and \textit{Loop}.

5.3.3 Comparison to other models

The model of object-oriented programming described in this chapter is influenced by the fact that we are teaching introductory programming with Eiffel. The choice of Trucs and notions, the relationships between notions and the descriptions of the Trucs reflect this viewpoint. It would be wrong to claim that the model represents a generally accepted image of object-orientation and it would be wrong to claim that it is the only valid model of object-orientation.

To ensure that certain properties, in particular those of cyclic and transitive dependencies, are not entirely due to these local specifics, we have asked another instructor teaching introductory object-oriented programming with Java to model parts of his domain of teaching. His domain model includes the entities required to represent the first three lectures of the introductory Java course. Figure 5.35 shows the outcome of this experiment as a Truc-notion graph. Note that the notion dependencies are incomplete.

Compared to our model, \textit{Data type} conforms to our Truc \textit{Primitive type}, \textit{Method} to a combination of \textit{Feature} and \textit{Argument passing} with some elements of \textit{System execution} (the main method), \textit{Object} to the Truc \textit{Object} and some parts of \textit{Feature call, Reference} and \textit{Object creation}, and \textit{Variable} to \textit{Reference} with parts of \textit{Feature}. Our model has no Truc conforming to \textit{Access modifier} (Zugriffsmodiﬁer), but \textit{Feature} integrates its concepts. The Truc \textit{Polymorphism} covers the notions found in our Truc \textit{Inheritance}, and Truc \textit{Compilation unit} contains the concept of \textit{Class}, for which we have a separate Truc. Generally, this model covers the fundamental parts of our domain model, but the distribution amongst Trucs varies.

The analysis of the transitive dependencies of this model exhibits a group containing the Trucs \textit{Data type, Method, Object, and Variable}. These
5.3. ANALYSIS OF THE DEPENDENCY GRAPH

Figure 5.34: Cyclic dependencies
Figure 5.35: Truc-notion graph of Introduction to Programming at RWTH Aachen (done by D. Herding, German terms translated into English)
Trucs are suppliers for about half of the Trucs in the model. In particular, they are a prerequisite to themselves and to the Trucs Instruction, Polymorphism, Memory management, and Access modifier.

There are no cycles on notion level in this model, but there are four cycles on the Truc level: Method and Data type as well as Object and Variable are mutually dependent; additionally, there is one cycle containing the Trucs Method, Variable, and Object, and one containing Method, Data type, and Object.

There are a few differences between the two models. In the Java model the Truc Compilation unit containing the concept of class is not part of the cyclic group, while in our model the Truc Class is part of the core group. On the other hand, the Truc Data type conforming to our Truc Primitive type is part of it. This is due to the notion Command line argument in the Truc Method requesting the notion Array of Truc Data type. Another difference is that most of the other parts of the model are unconnected. This is probably due to the incomplete nature of the model. The most striking similarity between the two models is the existence of a group of Trucs that are mutually dependent and that are fundamental for a large portion of the remaining Trucs.

5.4 Application to Introduction to Programming

Applying the TrucStudio approach to a course involves two steps: (1) modeling the course on the basis of the available notions and analyzing the resulting sequence with respect to missing topics and missing prerequisites, and (2) using the checklists of recommendations for handling misconceptions and for identifying possible improvements for the teaching material.

The topic sequence of the ETH Introduction to Programming course follows the idea of the Inverted Curriculum (see Section 2.2). The Inverted Curriculum advocates an Outside-In approach: starting from the viewpoint of a consumer of library components, it gradually exposes the internals.

The first lecture on object-oriented programming of the course focuses on presenting simple qualified feature calls on predefined objects complemented with the notion of actual argument. In a second part of the lecture, style guidelines and the notion of instruction and expression follow. The next lecture covers reading the interface of a class. It shows the connection between objects, classes, and types, provides the means to understand feature signatures and shows argument declarations. The course contin-
ues with object creation, assignment, the notion of algorithm, and control structures, before going into more advanced object-oriented mechanisms like inheritance.

To prepare a model for the course, we have relied on the slides of the course and we do not take into account the weekly assignments. The model presents parts of the course in combined form: for example, the first two set of slides covering feature calls and the concept of instruction and expression are modeled as one lecture. Since the domain model focuses on a basic set of concepts needed for modeling object-oriented programming, we omit lectures that do not share this focus; for example, the model does not include the lecture on describing syntax with BNF and a lecture on topological sort. Figure 5.36 gives an overview of the resulting course model with its lectures and notions and list the set of slides they represent.

<table>
<thead>
<tr>
<th>Dealing with Objects, 180 minutes (02_objects.pptx and 03_conventions.pptx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class → Feature → Feature declaration → Feature call → Target → Object → Simple qualified call → Feature → Actual argument → Information hiding → Expression → Instruction → Value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interface of a Class, 270 minutes (04_interface.pptx and 05_logic.pptx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class → Feature → Object → Instance → Generating class → Feature declaration → Feature signature → Return type → Expression → Type → Argument declaration → Formal argument → Design by Contract → Precondition → Assertion → Postcondition → Class invariant → Boolean expression → Boolean → Infix operator → Prefix operator</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object Creation, 90 minutes (06_creation.pptx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity → Attribute → Object → Reference → Void → Attachment → Void call → Object creation → Creation instruction → Type → Creation procedure → Void → Linked list → Creation procedure declaration → Creation principle → Precondition → Class invariant → Postcondition → System entry point → Root object → Root class → Root creation procedure → System execution → Current object</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>References &amp; Assignment, 90 minutes (07_references_assignments.pptx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object structure → Reference → Value → Expanded type → Reference type → Primitive type → Boolean → Integer → Character → Real → String → Attribute → Assignment → Unqualified feature call → Simple qualified feature call → Current object → System entry point → Entity → Assignment → Integer expression → Boolean expression → Result → Reference assignment → Value assignment → Linked list → Local variable</td>
</tr>
</tbody>
</table>

Figure 5.36: Course model of “Introduction to Programming”
<table>
<thead>
<tr>
<th>Control Structures, 180 minutes (08.control.pptx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm → Conditional → If-then-else statement → If-statement → Nested conditional → If-then-elsif-else statement → Loop → Exit condition → Loop invariant → Loop variant → Linked list → Function → Nested loop → Termination</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstraction, 90 minutes (09.abstraction.pptx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature → Attribute → Procedure → Instruction → Function → Return type → Expression → Information hiding → Export status</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic Model, 90 minutes (10.dynamic_model.pptx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment → Reference assignment → Value assignment → Linked list → Aliasing → Infix operator → Prefix operator</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inheritance &amp; Genericity, 270 minutes (12.inheritance.pptx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genericity → Unconstrained genericity → Constrained genericity → Generic data structure → Formal generic parameter → Actual generic parameter → Inheritance → Descendant → Ancestor → Deferred feature → Redefined feature → Deferred class → Polymorphic assignment → Polymorphic argument passing → Polymorphic attachment → Polymorphism → Static type → Dynamic type → Type conformance → Redefined feature → Dynamic binding → Polymorphic data structure → Constrained genericity → Dynamic cast → Deferred class → Deferred feature → Interface class → Invariant inheritance → Assertion redeclaration</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Recursion, 90 minutes (13.recursion.pptx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursion → Recursive algorithm → Recursive call → Indirect recursion → Direct recursion → Recursive data structure → Tail recursion → Recursion elimination → Stack → Call stack</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Structures, 90 minutes (14.data_structures.pptx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked list → List operation → Cursor → Array → Index → Array resizing → Complexity → Worst case → Big-O notation → List performance → Array performance → Hash table → Hash function → Collision resolution → Open hashing → Closed hashing → Hash table performance → Stack</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multiple inheritance, 90 minutes (17.multiple_inheritance.pptx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple inheritance → Deferred class → Name clash → Renaming → Repeated inheritance</td>
</tr>
</tbody>
</table>

Figure 5.36: Course model of “Introduction to Programming” (continued)
5.4.1 Possibly missing notions

One of the benefits of the TrucStudio course model is its use for detecting possibly missing topics. This subsection discusses the notions of our domain model that the course Introduction to Programming does not cover. Note that the practical assignments of the course compensate for many of the topics that are missing in the lecture material.

The first lecture focuses on the notions of the Truc Feature call, in particular Feature call, Target, and Simple qualified feature call. The notion Multi dot feature call is missing in this lecture and throughout the course. The arguments against introducing this notion in the course are that long chains of feature calls occur rarely and are generally considered bad practice, violating the principle of the “Law of Demeter” [72]. The argument for including it is that students may encounter the need for composite calls. The lectures of Introduction to Programming do not cover multi-dot feature calls, but the course provides an explanation and an exercise on it in an assignment.

Another missing topic is the notion Argument passing. This indicates that the course refrains from showing the connection between actual and formal arguments. A closer look at the slides used in class shows that examples illustrate the notion of Formal argument, but the concept is not formally explained. We recommend putting more emphasis on the connection between actual and formal argument.

Manifest constants are values that denote themselves (such as -7, 0, 233, True, False, 'A', 'z'). This topic would fit into the lecture on References and Assignment, which also covers the primitive types of Eiffel. A related topic, the notion Constant, is also missing in the slides and could be included in the lecture on Abstraction. The lecture on Abstraction does not cover Encapsulation, belonging to the Truc Information hiding, but it includes it by covering export status and information hiding.

In the lecture on inheritance, the concepts of Precursor and Polymorphic creation are both missing and should be added.

Another missing notion that we included in the domain model is Backtracking. It is unclear whether its coverage should happen in Introduction to Programming or be deferred to a later course on algorithms.

The course model shows that the notions refining Instruction are not explicitly covered. This indicates that the course does not give an overview of all possible instruction variants.
5.4. Topic sequence

TrucStudio helps identify violated prerequisites in topic sequences. The process uses the rule that if a notion’s requirements are covered in the same lecture or in earlier lectures then they are considered satisfied.

In the first lecture on feature calls there are five notions that have violated prerequisites: `Object` requires `Instance`, `Feature call` requires `Argument passing`, `Feature declaration` requires `Feature signature`, and `Information hiding` requires `Encapsulation` and `Dynamic binding`. Because of the high interrelatedness of concepts, it is not surprising to find many violated prerequisites in this lecture. The next lecture on the interface of a class covers most of the required notions, except for the requirements of information hiding. This might indicate that it is too early to cover `Information hiding` at this stage of the course.

The second lecture has only one violated prerequisite: `Boolean expression` requires `Manifest constant` (through refinement of the general notion `Expression`). A closer look at the material shows that indeed the possible manifest constants for Boolean (`True` and `False`) are covered, but not the general notion of `Manifest constant`.

In the lecture on object creation there are again several reported missing prerequisites: `Entity` requires both `Local variable` and `Result`, and `Attachment` requires `Object structure`. Reappearing violated requirements are between `Linked list` and `Unconstrained genericity`, `Cursor`, and `List performance`. They appear in many of the lectures, because the teaching material uses linked list to illustrate concepts in connection with references, the object structure, and loops. All these reported prerequisite violations are non-critical since the teaching material covers linked list to the extent needed in the according context.

A more critical dependency violation appears in the lecture on references and assignment. The description of default initialization refers to the notion of `Result` as one of the entities to which the specified rules apply. Additionally, the list of possible targets for assignment instructions mentions `Result`. Since `Function` only appears almost two weeks later in the course, students cannot have an understanding of `Result` at this point.

The slides on inheritance and genericity only have two reported problems with prerequisites: `Inheritance` in our model requires `Renaming`, which appears in the lecture on multiple inheritance at the end of the course. Additionally, `Polymorphic argument passing` reports `Argument passing` as violated prerequisite. Section 5.4.1 lists `Argument passing` as one of the missing notions.

In the last three lectures, there is only one violation reported: to un-
derstand the data structure \textit{Stack}, students need to know \textit{Array}. However, the general notion of \textit{Stack} already appears in the lecture on recursion, before \textit{Array} has been covered. Since the lecturer can give a first explanation of stacks without going into implementation details, this violation is non-critical.

The slides on Inheritance and Genericity contain notions of the Trucs \textit{Inheritance, Genericity, Polymorphism, Dynamic binding, Deferred class, and Design by Contract}. It starts with \textit{Unconstrained genericity} then moves on to \textit{Inheritance} and \textit{Deferred classes}, and continues with \textit{Polymorphism}, which requires \textit{Unconstrained genericity} for \textit{Polymorphic data structure}. After this, it covers the constrained version of genericity, which requires an understanding of \textit{Inheritance}, continues with \textit{Dynamic cast}, and revisits \textit{Deferred feature}. Although the circular dependencies make it necessary to cover certain notions repeatedly, joining the first and the second part on \textit{Deferred class} could improve the structure of the slides. In addition, \textit{Dynamic cast} and contract inheritance could be moved to more appropriate positions in the lecture such that only genericity needs to be revisited.

\section*{5.4.3 Common confusions}

This part describes to what extent the examples used in class incorporate the recommendations deduced from the Truc’s collected common misconceptions. For this analysis, we went through the recommendations, Truc by Truc, and inspected the associated slides of Introduction to Programming.

The outcome is surprising: only for two (\textit{Object creation} and \textit{Reference}) out of the 28 Trucs are all the recommendations met; for another nine Trucs the recommendations are partially met; for the remaining 17 the according teaching material does not conform to the recommendations. This outcome has to be taken with caution because a small part of the recommendations applies to aspects of a Truc that the instructors intentionally leaves out. Irrespective of this, the results support the conjecture that teaching material rarely incorporates knowledge of novice misconceptions and that the Trucs help locate these deficiencies.

Based on the recommendations of the collected misconceptions, we suggest the following improvements to the slides of Introduction to Programming:

- \textit{Argument passing}: Show feature calls with invalid and valid actual arguments (with respect to the type and number); include a situation in which formal arguments are passed on as actual arguments
5.5. IMPLICATIONS ON TEACHING

To another feature; and clarify the connection between formal argument and actual argument (thus covering argument passing).

- **Conditional**: Show more examples of applied conditionals and address the concerns mentioned in the according common confusion section (the slides do not contain examples of conditional except for one in combination with loops).

- **Design by Contract**: Discuss common misunderstandings of contract inheritance.

- **Dynamic binding** and **Polymorphism**: Clarify the practical implications of entities’ dynamic and static types.

- **Feature**: Expand on function declarations and the use of Result as an entity.

- **Feature call** and **System execution**: Illustrate the control flow of feature calls at program run time.

- **Inheritance** and **Multiple inheritance**: Discuss misuse of inheritance and taxomania.


This is an incomplete list; for more details refer to the sections on common confusions.

### 5.5 Implications on teaching

Instructors face many challenges when designing courses or textbooks. Besides exhibiting great pedagogical finesse for presenting material adapted to students’ skills and interests, they must also demonstrate the ability to structure the material in a sound sequence. In particular, the sequence should at all times build on prior knowledge, it should avoid unwanted repetitions of topics throughout a course, and it should cover all basic notions.

These tasks are particularly difficult if the domain of teaching is an object-oriented programming course due to the high interrelatedness of its concepts. It complicates finding a starting point where no prerequisites are necessary, and presents challenges in ensuring that the entire subject area is covered and in sequencing the concepts without prerequisite violations.
This tight interrelatedness of the basic concepts is what Caspersen calls “one of the most challenging inherent complexities of object-orientation” [10, p. 78] and others agree with this claim [47]. The analysis of the Truc and notion graphs shown in this chapter also confirms this statement.

The circular dependencies in the Truc and notion model indicate that teaching object-oriented programming requires a spiral model; “A curriculum as it develops should revisit these basic ideas repeatedly, building upon them until the student has grasped the full formal apparatus that goes with them” [125]. They also indicate that there is no standard solution to the “Big bang problem” [10].

On the Truc level, the transitive and recursive dependencies of the nine core Trucs Argument passing, Class, Expression, Feature, Feature call, Object, Object creation, Reference, and System execution show that these Trucs – in an objects-first approach – belong in the initial phase of a course.

On the notion level, the ten notions Argument declaration, Class, Feature, Feature declaration, Feature signature, Formal argument, Generating class, Instance, Object, and Type have similar properties with respect to their dependencies as the nine core Trucs. With the exception of Argument passing and Formal argument, which can be omitted if only features without arguments are covered, it seems necessary to introduce them together.

Programming environments such as BlueJ [69] target novice programmers and allow students to explore objects and classes after covering the above notions. But if the intention is to let students write code and produce running programs, there are additional notions missing: students need preexisting objects made available through inherited attributes, declared attributes, declared local variables, or formal arguments; they also need one of the instruction variants Feature call or Assignment for the program to actually operate. Therefore, in the end for a first program they will need a number of notions coming from Trucs Class, Object, Feature, and Feature call or Assignment.

The challenge for instructors when using an objects-first approach is mainly the search for adequate examples that only rely on already covered notions. Particularly in the starting phases, when a limited set of concepts has been introduced, interesting examples are rare and difficult to create.

Many approaches circumvent the issue by focusing on imperative concepts in the first lecture: after a discussion of Class, they introduce variable declarations of primitive types such as integer and strings, operations to build expressions, and the output mechanism. In some cases, they also cover loops and conditionals. Only after the most important imperative concepts have been covered, they continue with object-oriented concepts.

The Inverted Curriculum [84, 82, 103] (see Section 2.2) takes another
approach and uses a large body of supporting software that hides complex implementations through abstraction. The examples and exercises rely on the internals of the software framework. The weakness of the Inverted Curriculum approach is that the preparation of the software framework and all examples and exercises needs to happen before the students receive the software. This produces the need for more planning and may lead to a restricted set of possible exercises. The advantage of the Inverted Curriculum approach is a clean, truly object-centered sequence of topics since there is no focus on imperative concepts at the beginning.

The ETH Introduction to Programming course implements the Inverted Curriculum. The application of the extended Truc approach to its slides identifies several possibly missing notions; amongst them are Multi-dot feature call, Argument passing, Manifest constant, Constant, Precursor and Polymorphic creation. With respect to prerequisite violations there is only the notion Result whose appearance before Function may lead to a problem.

To help improve the quality of teaching material and, in particular, the examples shown in class, we have added checklists to the Truc section on Common confusions. The checklists show a set of criteria that slides or examples illustrating a certain topic should exhibit. The use of the Trucs’ common confusions section enriched with checklists helps analyze teaching material with respect to missing treatments of novice misconceptions.

The analysis of the lecture material of Introduction to Programming indicates that many of the research findings on novice misconceptions are not put to good use. In fact, the teaching material fully complied with the recommendations of only two Trucs. This indicates that – in spite of the effort dedicated to increase the quality of the course every year – we can improve the course material using the checklists as a tool.
CHAPTER 6

TRUCSTUDIO AND ITS ARCHITECTURE

TrucStudio\(^1\) supports the extended Truc approach and offers several features that help develop, maintain, distribute and analyze the domain and course models. These tools include a customizable and extensible output generation mechanism for producing course pages, calendar files with lecture appointments or browsable lexica, visual and textual outputs for the dependency structure of the domain model, and graphical representation of the notions covered in a course.

This chapter gives an overview of TrucStudio. It shows the manipulation interfaces and the graphical representation for the domain and course models. It details on the output generation mechanism and describes a tool that produces a report of the prerequisite structure of a domain. It reports on an experiment carried out with an external tool to automatically extract notions from teaching material and it describes the architecture of TrucStudio. Some of the sections have already been published in international conferences [106, 108, 107].

6.1 Related work

The related work section of Chapter 4 (see Section 4.1) provides information on several related approaches for knowledge and concept modeling. For many of them, tool support is available.

For example, the software CmapTools [17, 90] offers tool support for concept mapping. It allows its users to collaboratively construct concept

\(^1\)Available at http://trucstudio.origo.ethz.ch
maps, publish them on the Web, search information related to a concept map on the Internet, and attach resources located anywhere to concepts or relationships.

For ontology modeling and knowledge acquisition, the open-source platform Protégé [46] offers a plug-in architecture and a wide range of plugins like OntoViz for the visualization of the ontology structures, Collaborative Protégé [134] that adds collaboration features, and the Protégé OWL Plugin [67] to edit ontologies in the Web Ontology Language (OWL [140]).

The ADISA tool [98] is one of the software systems supporting instructional design and the development of tele-education systems. It incorporates an editor for the MOT [97] model for knowledge modeling and for the instructional engineering method MISA [97].

All of these tools provide methods to develop domain models, but they target different applications than the ones TrucStudio supports. TrucStudio is specialized for the development of course and domain models that are simple with respect to the available entities and relationships, but expressive enough to help improve traditional courses; its strengths are a set of tools that help analyze structural properties and that provide extensible output generation mechanisms to support distribution of the models.

6.2 Overview

TrucStudio’s domain and course entities generally have the same properties as described in Chapter 4. For the domain model, there are three entity types: notions, Trucs, and clusters (in increasing order of granularity). A notion belongs to exactly one Truc, and it contains requires and refines links to other notions. The iconic representation of notions in TrucStudio is a blue box (see Figure 6.1). A Truc contains a set of notions, it belongs to exactly one cluster, and its notions’ requirements across Truc boundaries define its dependencies. The Truc icon visualizes its definition as a composition of notions. A cluster contains a set of Trucs, it may belong to any number of other clusters with the restriction that the cluster hierarchy forms an acyclic structure, and Truc dependencies across cluster boundaries define its dependencies. The cluster icon visualizes its intersection property. For more flexibility, TrucStudio adds explicit dependency links to Trucs and clusters. Explicit dependency links are user-defined links, as opposed to the dependency links generated through requires links between notions.

The course model provides two entity types: lectures and courses. A lecture consists of a sequence of covered notions and it has an estimated
teaching time. Each course contains a list of time slots (e.g. every Monday 2 p.m. to 4 p.m. in the semester) available for teaching it, and a list of lectures.

![Figure 6.1: Icons for TrucStudio’s entities](image)

TrucStudio’s interface (see Figure 6.2) consists of three main panels. On the top left side, a tree widget (the Truc-notion tree) displays the clusters, Trucs, and notions of the domain model. On the bottom left, another tree widget (the course-lecture tree) provides a list of all courses managed through TrucStudio and their associated lectures. A manipulation interface on the right supports editing the entities and structures.

Selecting one of the entities in the Truc-notion or course-lecture trees updates the manipulation interface. The manipulation interface can be either a form for editing the selected entity in a textual fashion, or a graphical view showing a Truc-notion graph (as in Figure 6.2) or a course diagram.

TrucStudio offers the functionality for the direct manipulation of the domain and course model in the main interface. Its supplementary tools such as output generation or the report of the dependency structure of the domain model are available through menu items in the Tools menu.

### 6.3 Domain modeling

TrucStudio supplies multiple ways to create the domain model; this section describes the direct entry and manipulation of domain entities in TrucStudio. Section 6.7 reports on an experiment with an ontology-learning tool for semi-automatic extraction of concepts from existing teaching material. This forms an alternative for defining the domain model.

An instructor models the domain of teaching in TrucStudio by creating clusters, Trucs, and notions in the Truc-notion-tree. To create a new Truc it must be associated to an existing cluster, and to create a notion there must be a valid parent Truc available. The creation of a cluster requires the selection of its parent cluster. This can be either another existing cluster or the system’s root, in which case the cluster is a top-level cluster.
Figure 6.2: Overview of the TrucStudio interface

Figure 6.3: Property view for Truc “feature call”
The Truc-notion tree displays the hierarchical structure of the entities and allows modification of it through context menus. The context menus support moving notions from one Truc to another, and moving Trucs from one cluster to another. For clusters, the context menu offers a copy operation that adds it to a new parent cluster and a deletion operation that removes it from the system.

Selecting one of the entities in the list updates the manipulation interface as in Figure 6.3 to display the properties of the selected entity. The resulting property view contains fields for the name, a URL, and a summary. They list all relations of an entity: its user-defined incoming and outgoing links and the ones defined through contained notions. The interface provides removal and insertion operations for the explicit links. The property view for Trucs additionally offers fields to edit their descriptive sections such as role, benefits, or common confusions.

While the property views enable the user to modify the single sections of an entity’s description, the graphical view (see Figure 6.2) displays all the structural information of the entities (containment, dependency, requires, and refines links). It automatically generates a layout for the graph and lets the user blend in or out any item of the graph (such as all the notions of a particular Truc, or all the requires links between notions). The user may manually rearrange the nodes, export the image to a PNG or PostScript file, and load/save the layout into an XML file format. The colors and font properties of the graph can be adapted in the graph preferences tool available in the Tools menu. The graphical view supports a restricted set of manipulation operations such as creating or removing clusters, Trucs, notions, and links. It does not offer functionality to edit the descriptive sections of the entities (such as name or summary).

### 6.4 Course modeling

The property view for lectures and courses both contain the fields for name, URL, and summary. The property view for lectures provides operations to add domain entities to the sequence of covered notions of a lecture. If a Truc or cluster is added to the lecture, then all its contained notions will be inserted at the end of the sequence of covered notions. The interface also supports reordering of the notion sequence and deletion of notions from the sequence. Additionally, it includes a rudimentary soundness check for the covered notions and reports notions that are repeated in other lectures and violations of a notion’s dependencies.

The property view for a course contains the sequence of its lectures
with associated notions, its timeslots, and a diagram visualizing the mapping of lectures to timeslots (see Figure 6.4). The interface offers a moving operation to reorder the lecture sequence and operations to add and remove timeslots.

TrucStudio also offers a visualization of a course and the notions that each of its lectures covers (see Figure 6.5). The diagram makes it possible to analyze a course or a textbook with respect to its organization; in the example, the diagram visualizes the focus of the first lectures of the Introduction to Programming course in terms of the notions of the Trucs Feature, Feature call, Class and Design by Contract. This provides a basis for objective comparisons of two courses or a course and a curriculum specification. By checking if every notion has a cross in its row, the diagram also supports detection of possibly missing topics.
6.5. OUTPUT GENERATION

A core aspect of course management is communication between the teaching staff and the students. Teachers typically provide much of the information managed with TrucStudio in other formats to students.

The task of publishing course information and keeping it up to date is often tedious and repetitive. TrucStudio provides a rich output mechanism for easy generation of outputs representing several kinds of information managed by the framework. The output mechanism is extensible through user-defined output templates supporting customization and allowing an instructor, group, or institution to enforce a common look and feel across a whole set of courses.

TrucStudio provides a basic set of templates generating the most commonly needed outputs such as course websites providing course information and an alphabetical glossary of notions. Templates include:

- Several types of web sites containing notions, Trucs, clusters, lectures, and courses.
• Generation of calendar events in ICAL format for courses.

• A primitive OWL (Web Ontology Language) output to exchange notions and Trucs with ontology tools.

Among supported formats, a particularly interesting one is the browsable lexicon. Similar to Concept Maps [17], it allows the students to review the concepts taught by highlighting relations between them. In the hyperbolic tree pane (top left part in Figure 6.6), clicking on any entity updates the right pane containing all the information about that entity and centers the tree to highlight the position of the entity and related ones in the entity hierarchy. Additionally, the glossary (lower left pane) helps find information about a particular topic.

![Figure 6.6: Browsable lexicon screenshot](image)

TrucStudio enables instructors to provide students with up-to-date information about the lectures as well as a conceptual overview of the relations between the different topics treated in the course. They can adapt the output formats to existing structures (for example to integrate them into existing course web pages or adapt them to a corporate design) and reuse them for all TrucStudio-managed courses.
6.6 Structure report

One of the main goals of TrucStudio is to provide support for the analysis of relations between domain concepts. To reach this goal, TrucStudio provides a tool that generates a report on the dependency structure of the domain model as a webpage. The report contains three main sections: (a) a report on direct and indirect dependencies of individual Trucs and their contained notions sorted in alphabetical order of Truc names, (b) a tabular representation of recursive links, and (c) a report of cycles in the dependency structure.

The structure-reporting tool offers customization of the webpage with respect to its contents. For example, it supports including or excluding the report on notion (or Truc) cycles, the descriptions of direct and indirect Truc dependencies, and enabling or disabling the production of tables of transitive dependencies.

The first part of the structure report contains a section for each individual Truc sorted in alphabetical order (see Figure 6.7). Such a section first contains the description of the Truc including a list of contained notions and its links. For each of its contained notions, a subsection describes its structural properties. The main intent of the individual descriptions of Trucs and notions dependencies is to help check that the chosen requires and refines links on notion level produce meaningful dependencies both transitively on notion level and deduced on Truc level. There are three options to customize the output of this part.

- **List all direct dependencies of a notion/Truc.** This option is available twice: once for the reporting of Truc dependencies and once for the reporting of notion dependencies. In both cases, it reports both outgoing and incoming dependencies, i.e. the number and name of the entities that the described entity directly depends on and the number and name of entities that directly depend on the described entity.

  If enabled for notions, it will list all direct dependencies including its explicit dependencies (those defined by the requires links of a notion) and its implicit dependencies (the direct dependencies inherited through refinements). For Trucs, the report also describes what notions are responsible for the implicit dependencies on Truc level.

- **List all transitive dependencies of a notion/Truc.** This option is also available separately for notions and Trucs. It reports the outgoing transitive dependencies of the described entity and the incoming transitive links (entities that transitively depend on it).
• **List all refinements of a notion.** For notions only, there is the option to include information on its refines links. The output considers both incoming and outgoing refines relations.

The second part of the structure report contains tables of outgoing and incoming transitive dependencies for all notions and one for all Trucs. The tables list all notions or all Trucs in its top row and in its first column. A cross in a cell indicates that the entity with the name found in the first cell of the row transitively requires the entity with the name found in the top cell of the column. These tables can be copied into an EXCEL sheet and used to qualitatively analyze the structural properties of the two dependency graphs. TrucStudio supports enabling or disabling the generation of the tables individually for notions and for Trucs.

The third part of the report describes the results of cycle detection. It contains two parts: a report of all cycles in the dependency graph of Trucs and a report of all cycles in the dependency graph of notions. Each of the parts can be enabled or disabled individually.

### 6.7 Semi-automatic extraction of notions

The construction of notions for a domain model is a time consuming task. To alleviate the amount of work requested by educators, tool support, even just semi-automatic, is essential. Tools for automatic extraction of concepts from text exist in the context of ontologies.

An ontology is a domain description that uses concepts and relations between these concepts, or according to Gruber “an explicit specification of a conceptualization” [55]. This makes them more general than the models generated with the extended Truc approach.

The teaching community has come to rely on ontologies particularly in online teaching, where they help classify “learning objects” and describe learning and teaching strategies [45]. Many researchers advocate the use of ontologies to produce metadata information for learning objects [30], but only a few have considered applying to education ontology learning techniques (see e.g. [18]). They use machine learning and natural language processing to extract concepts and build refines and other relationships from existing data, natural language descriptions [36].

Text2Onto [19, 18], which can produce OWL files, is one of the available ontology-learning tools. It is open-source and targets data-driven change discovery using an incremental ontology learning strategy from text. It
Figure 6.7: Structure report
includes components for concept extraction, subclass-of, part-of, instance-of, equivalence, and general relations.

TrucStudio relies on Text2Onto to identify the main concepts covered in existing textbooks or slides (in the usual teaching formats: PDF, HTML and plain text) and supports importing them into the TrucStudio domain model. Preliminary tests have shown that the extraction mechanisms for relations that are available in Text2Onto would not produce meaningful results for the input data available from existing courses (slides and textbook extracts); the import mechanism thus focuses on the concepts, for which Text2Onto provides several extraction algorithms calculating relevance measures for each concept. This relevance measure can be used to reduce the number of concepts that a user has to inspect for inclusion in the TrucStudio domain model (see the import dialog in Figure 6.8).

Using Text2Onto together with TrucStudio involves the following steps: (1) call Text2Onto providing a corpus (a set of input documents) in PDF, HTML or plain text; (2) Text2Onto applies its concept extraction algorithm and calculates the relevance values for each of the found concepts; (3) it exports the concepts into an OWL file (an ontology file format); (4) TrucStudio reads the OWL file and presents the extracted concepts to the user who then can use them to build Trucs, notions and clusters.

![Import 'Text2Onto' File]

Figure 6.8: Import dialog in TrucStudio
6.7. SEMI-AUTOMATIC EXTRACTION OF NOTIONS

6.7.1 Application to teaching material

A case study to assess the approach processes teaching materials from two courses, an introductory one using Eiffel and an advanced one in Java, both held at ETH Zurich. The study gathers evidence on the quality of the extracted concepts with respect to the type of input (slides and textbooks).

The main goal of the case study was to answer the following questions: What inputs work best for the text processing (course slides or textbooks)? Can we generalize over different courses? How many concepts are extracted and thus need manual processing? What percentage of the extracted concepts is actually usable as Trucs or notions?

To answer these questions, the study used teaching materials from two courses taught in the academic year of 2006/2007: Introduction to Programming (slides\(^2\) and accompanying textbook \([86]\)) described in Chapter 2 and Java Programming (slides and handouts\(^3\)). Since the inputs to the ontology-learning tool are the slides and textbooks available from the course web pages in PDF, the results are dependent on the quality of these documents.

Introduction to Programming implements the Inverted Curriculum approach \([103]\) using Eiffel and covers fundamental object-oriented and procedural concepts such as objects, classes, inheritance, and control structures. The teaching material has been used and refined over multiple years. Its teaching material contains many examples from a real-life domain: transportation in a city. An initial assumption is that ontology learning works better on the slides of this course than on the textbook because the textbook examples are more general and may include less immediately relevant concepts.

Java Programming, an elective course, targets master-level CS students. It covers advanced Java concepts such as threads, compilation, and sockets. The course page provides slides and reading material; in the slides, most code examples appear as screenshots. The reading material is a digest of the concepts covered in the course and is meant as complementary material. (In contrast, the Introduction to Programming textbook can be used independently of the course.) These characteristics suggest that the material from this course would produce better results in ontology learning.

\(^2\)Available online at http://se.ethz.ch/teaching/ws2006/0001
\(^3\)Available online at http://se.ethz.ch/teaching/ss2007/0284
6.7.2 Results

Figure 6.9 lists on the x-axis the texts used as input to Text2Onto. For both courses, the study used a selected set of slides and the corresponding extracts from the reading material, with names starting with “slides” and “doc” respectively. The items “alldoc” and “allslides” show the results when Text2Onto processes all the reading extracts or all the slides of a course in a batch. Numbers in parentheses are the number of slides or pages.

Text2Onto processed all the listed reading extracts and slides and generated OWL files that contain the extracted concepts and their estimated relevance. Instructors then categorized the extracted concepts as relevant or not. Figure 6.9 shows the number of relevant and irrelevant concepts for each of the documents. The number at the top of each bar indicates the percentage of relevant concepts.

Extracted concepts. For both courses, the number of concepts extracted from slides is significantly lower than the number of concepts extracted from reading material. The conjecture is that slides generally present material in a condensed form, avoiding verbose explanations and full sentences. They reduce the text to include only the most relevant concepts and omit most of the noise found in reading extracts, thus the lower number of extracted concepts from slides is natural.

The comparison of the numbers of extracted concepts between the two courses indicates that for slides the density of extracted concepts per slide is similar (for Introduction to Programming 2.7 concepts per slide; for Java Programming 1.9 concepts per slide). For the reading extracts of the Java course the density is higher (22.9 concepts per page) than for the textbook extracts of Introduction to Programming (15.1 per page). This can be traced back not only to the authors’ different writing styles but also to differing purposes: while the reading extracts for Java Programming are complementary material summarizing the lectures’ topics, the Introduction to Programming document is a self-contained tutorial involving thorough explanations and examples.

Relevant concepts. The higher number of extracted concepts for reading extracts (in comparison to slides) generally also results in a higher number of relevant concepts. A possible explanation is that some concepts important to an understanding of the “big picture” show up in the instructors’ verbal explanations but not in the slides. Assuming that the slides
6.7. SEMI-AUTOMATIC EXTRACTION OF NOTIONS

Figure 6.9: Relevant vs. irrelevant concepts
as well as the reading material contain the fundamental concepts (Trucs), such missing elements can only be at the notion level. A recommendation could thus be to use slides as input if a coarse domain description suffices (providing the Trucs and most important notions), and reading material extracts if a more detailed domain description is needed.

More significant than the raw numbers of extracted and relevant concepts is the percentage of the extracted concepts that are meaningful for domain modeling. This measure reveals that for both courses the concepts extracted from slides are much more likely to be relevant than those extracted from reading material. Extracting the concepts on smaller documents provides more accurate results than using the entire set of slides or reading extracts as input to the ontology-learning tool, but comes at the cost of a much higher number of concepts to process.

The comparison of relevant concepts between the courses shows that for both the slides and the reading extracts the material of the Java course produces higher percentages. This result is also reflected in the density of relevant concepts per page or slide. For the slides of Introduction to Programming the number of relevant concepts per slide is 1.2 as opposed to 1.5 concepts per slide for Java Programming. The reading material exhibits the same tendency in an extreme form: the reading extracts from Java Programming contain 16.0 relevant concepts per page, while the Introduction to Programming textbook contains 4.4 relevant concepts per page.

**Relevance measure.** Generally, the number of raw extracted concepts is very high and results in significant manual work. Text2Onto provides a relevance rating, which can be used for sorting concepts by relevance. Assuming the rating is meaningful in the described setting, can it help removing concepts with low relevance?

To answer this question the first step is to analyze the distribution of cumulated extracted and relevant concepts for each present relevance value. Figure 6.10 shows that the cumulated number of extracted concepts per relevance value grows dramatically as the relevance gets close to zero: most concepts have a very low relevance rating. The power function \( y = \alpha + r^\beta \) (with \( r \) the relevance rating, and \( \alpha, \beta \) constants) provides a good experimental fitting of all the curves produced from the material of this study. A power function is also adequate to approximate the cumulated relevant concepts (see the grey curves in Figure 6.10), but grows slower than the cumulated extracted concepts.

It is possible to compute the parameters \( \alpha \) and \( \beta \) without user interaction for cumulated extracted concepts, but not for cumulated relevant concepts since this requires sampling.
Figure 6.10: Cumulated concepts (extracted, relevant) per relevance value

Figure 6.11 shows the percentage of cumulated relevant concepts for all the relevance values. This measure is interesting for the removal of concepts with a relevance rating below a certain limit. While the curves are generally very unsteady at high relevance values (due to the low number of extracted concepts at those values, see Figure 6.10), they exhibit a clear decrease pattern at lower values. This also applies to the curves for documents not included in Figure 6.11. This systematic pattern confirms that the relevance rating provided by Text2Onto is a valuable prediction measure, especially at low rating levels.

6.7.3 Summary

The relevance ratings provided by Text2Onto come out, in our setting, as a good predictor of relevance. It seems impossible to find a fixed value that is suitable in all cases for optimizing the percentage of relevant concepts by removing concepts below this value. Estimating the curves for cumulated extracted and cumulated relevant concepts predicts the percentage of relevant concepts for a certain relevance value, but requires user interaction to determine parameters for the curve fit of cumulated relevant concepts. The user interface of TrucStudio does not implement this strategy, but pro-
vides a slider allowing users to select a value for the relevance cut point showing the number of concepts to investigate.

The automatic extraction of concepts from course material is useful to support the creation of Trucs from scratch. Current ontology learning techniques cannot extract the Trucs automatically, but can help with extracting notion and Truc names tagged with an estimated relevance value. Depending on how the material has been designed, the extraction can lead to quite accurate results (80% for slides with no code examples), but the high number of extracted concepts remains a drawback of the approach.

6.8 Architecture of TrucStudio

TrucStudio provides a comprehensive environment backed by a systematic modeling approach, a combination strongly influenced by modern software development environments. TrucStudio is organized like a programming environment, but its elements of discourse, rather than software modules, are units of knowledge such as notions, Trucs, and clusters.

Its development started in the second half of 2007 with a Master thesis project that implemented a first prototype. Since then, five other master theses have contributed various features (for example, the output generation mechanism and the domain graph display) and several other students
have worked on it as part of a semester thesis or other course work.

6.8.1 Overall system architecture

TrucStudio is implemented in the programming language Eiffel and relies on a set of libraries: Vision2 for its GUI, Gobo for XML processing and XSLT transformations, and EiffelBase. It consists, at the time of writing, of around 280 classes containing almost 60,000 lines of code (excluding external libraries). These classes are grouped into four main clusters (see Figure 6.12).

![Figure 6.12: Overview of the TrucStudio main clusters](image)

The core of TrucStudio is the class hierarchy modeling the TrucStudio entities (clusters, Trucs, notions, lectures and courses). These classes are
located in the Eiffel cluster “model” together with a factory class that handles creation of their instances and several other utility classes.

The Eiffel cluster “main” contains controllers and view classes for various tools and manipulation interfaces included in TrucStudio. It also includes the facilities for the undo mechanism of entity manipulations, the tree views for all entities displayed on the left side of the TrucStudio interface, and the different panel views (properties, domain graph and course diagram).

The import and export tools such as the project saving and loading and the output generation framework reside in the Eiffel cluster “input/output”. This cluster also contains the classes that are responsible for saving and loading the TrucStudio projects in the TXM format.

The cluster “util” hosts a set of generic container classes and GUI elements that various tools incorporated in TrucStudio reuse.

The following sections describe the entity model, the general architecture of the graphical interfaces, and some of the output and input mechanisms in more detail.

### 6.8.2 Entity model

The entity classes `TS_CLUSTER`, `TS_TRUC`, `TS_NOTION`, `TS_COURSE`, and `TS_LECTURE` all inherit from `TS_ENTITY` (see Figure 6.13), a deferred class that contains common elements such as attributes `name`, `summary`, `uri` and setters for the attributes.
The domain model entities (clusters, Trucs, and notions) inherit from `TS_DOMAIN_ENTITY`, which is a descendant of `TS_ENTITY`. It includes the facilities for handling requirements between instances of the same type such as the requires links between two notions or the dependencies between Trucs. For each of the types there are explicit (user set) requires links and implicit requires links (calculated on the basis of other links). The implicit links of a notion are the requires links that they inherit through refining other notions (recursively); requires relations between notions of two different Trucs contribute to the implicit Truc links; and for clusters, the dependencies between Trucs of different clusters generate the clusters’ implicit links.

The main structure of the domain entities is defined through three levels of containment relations: notions belong to exactly one Truc, Trucs belong to exactly one cluster, and clusters may be contained in other clusters. The cluster hierarchy forms a directed acyclic graph as seen in the example of Figure 6.14. The dashed rectangles in the figure denote clusters, the rounded rectangles represent Trucs and the black rectangles show notions. The clusters marked with (T) are top-level clusters. Top-level clusters grant access to all the domain entities belonging to a TrucStudio project.

![Diagram of architecture of TrucStudio](image)

**Figure 6.14: Example of directed acyclic structure for domain entities**

Objects of type `TS_TRUC_MANAGER` in TrucStudio keep a list of the top-level clusters and provide access to all active domain elements in a loaded
system. It also provides access to a list of courses that then contain all the lectures of a project.

TrucStudio implements insertion and deletion of entities in a system through rooting and unrooting the elements (adding or removing their containment relation). To find out whether a given element is in the active system, the class `TS_ENTITY` requires the effective descendants to implement a query `is_rooted`. The class `TS_SHARED_MODEL_EVENTS` provides event queues for observers of the entity model. If an entity is added to the system, the according event queue (such as `notion_add_event` for notion insertion) notifies its subscribers.

**6.8.3 Graphical interfaces**

TrucStudio combines a wealth of graphical interfaces and uses Vision2 for the GUI elements. It generally separates model, view, and controller and uses an event-based observer mechanism. The views subscribe to the event queues provided in `TS_SHARED_MODEL_EVENTS` and update themselves upon notification, while the controllers trigger modifications of the model by reacting to user requests.

The class `TS_C_MAIN_WINDOW` is the controller for the main window and is responsible for putting together the main interface with its menu items, toolbar buttons, the two tree views, and the view panels. The view panels support switching between the manipulation view for single entities and the graph view of the domain entities.

TrucStudio provides an undo/redo mechanism for entity manipulations. The implementation of the history functionality relies on the pattern library versions [5] of the command and composite patterns [44] (see Figure 6.15).

The class `TS_MAIN_CONTROLLER` provides access to the history and offers a set of features that generate the most commonly needed history commands. For example, the feature `add_notion` creates a composite history command and executes it. The history command contains two redo/execute steps: it first adds the notion to an existing Truc and selects the new notion. The undo steps contain the notion removal from its parent Truc and selection of the former parent Truc (see Listing 6.1).

**Listing 6.1: Command generation for notion insertion**

```plaintext
add_notion (a_truc: TS_TRUC; a_notion: TS_NOTION) is
  -- Add 'a_notion' to 'a_truc'.
  -- Select the new notion afterward.
  -- Use history commands for this.
```

---

---
require
    a_truc_exists: a_truc /= Void
    a_notion_exists: a_notion /= Void
local
    s_c: HISTORY_COMMAND
    c_c: COMPOSITE_COMMAND

do
    create c_c.make
    create s_c.make_with_undo (agent do_nothing, agent
        select_truc (a_truc), false)
    c_c.add (s_c)
    create s_c.make_with_undo (agent a_notion.add_truc (a_truc), agent a_notion.del_truc, false)
    c_c.add (s_c)
    create s_c.make_with_undo (agent select_notion (a_notion), agent do_nothing, false)
    c_c.add (s_c)
    c_c.execute
end

Most of the controllers use the commands of TS_MAIN_CONTROLLER to trigger history-controlled modifications in the entity model, because the setup of the commands is non-trivial. For certain operations, however, the controllers circumvent the history and directly manipulate the model without recording the changes in the history. This happens for example when a new project is loaded from XML.

6.8.4 TrucStudio XML (TXM)

TrucStudio stores its projects in an XML derivative format, called TXM (TrucStudio XML). Listing 6.2 shows an extract of a TXM file.

The entities have a unique URI of the form ts://host/type/uuid/name, where host defines the location (in general “local”), type the entity type (such as “notion”, “truc”, “course”), uuid a unique identification string, name the current entity name. The use of a unique identifier (uuid) supports the handling of entities that may change name over time.

The TXM format stores the descriptive sections of Trucs in CDATA (un-parsed Character DATA [141]) sections. XML parsers ignore these sections when parsing XML files, so that they may contain text that otherwise would be recognized as markup possibly resulting in parsing errors. Using the CDATA sections for the descriptive sections of a Truc makes it possible to include descriptions that are not well-formed XML. This does not restrict the user to enter valid XML or HTML code. In fact, we have in-
tegrated the Truc descriptions shown in Section 5.2 in the TXM file using HTML tags for formatting. TrucStudio’s output generation mechanism can handle the unparsed HTML code when generating webpages.

6.8.5 Generic output generation

The TXM file format also plays a central role for the implementation of the generic output generation mechanism (see Section 6.5). The output generation mechanism heavily relies on XSLT (Extensible Stylesheet Language Transformations [139]). XSLT is an XML-based language for transforming an input XML document into another format (not necessarily an XML derivative).

TrucStudio implements the output generation in a wizard-like form. The process consists of three main steps. In the first step, TrucStudio presents available output choices to the user, from which he selects one. In the background, this selection results in the selection of an XSL file, which will be applied to an input file.

After having chosen the output variant, a number of entities need to be selected for inclusion in the output. This selection makes it possible to produce outputs for parts of the model. TrucStudio then creates a TXM
Listing 6.2: Example TXM file

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<project version="0.7">
  <cluster uri="ts://local/cluster/43E66637-45FF-40C3-8ECF-CD1EFE3DCEEF/Object-oriented programming fundamentals" name="Object-oriented programming fundamentals" toplevel="True">
    <truc_ref uri="ts://local/truc/69D62C2D-254B-4198-80DC-3D282A941675/Assignment" />
    ...</cluster>
  <truc uri="ts://local/truc/69D62C2D-254B-4198-80DC-3D282A941675/Assignment" name="Assignment">
    <alt_name><![CDATA[Association]]></alt_name>
    <summary><![CDATA[Assignment is the mechanism by which ...]]></summary>
    <role><![CDATA[Assignment and object creation are the two mechanisms for attachment of references...]]></role>
    <applicability><![CDATA[Need to reattach the target reference to the object to which the source reference is attached.]]></applicability>
    ...</truc>
  <notion uri="ts://local/notion/A178710E-5A13-4906-9CAE-742F4D16D6F9/Assignment" name="Assignment" central="True">
    <requires uri="ts://local/notion/89B7286C-740A-41A4-ABF3-32D573015B8A/Expression" />
    <requires uri="ts://local/notion/1DA62BF-D656-4C60-B48A-C3921DAF6F2/Entity" />
    <refines uri="ts://local/notion/0F93A030-90D7-44EE-8D6E-33394E2D77DB/Reference assignment" />
    <refines uri="ts://local/notion/97E6F0CE-61A4-48BD-A173-0F03556FB296/Value assignment" />
  </notion>
</project>
```
file (overwriting the file located at xsl/in.txm) containing the selected entities only.

In a third step, the user selects a destination for the output. Depending on the output choice, this will be a directory or a file name and location.

Finally, TrucStudio uses the GOBO XSLT processor to run the XSL transformation chosen in step one on the generated input TXM and creates the final output documents (see Figure 6.16).

The entries of the list of output choices, presented in step one, are loaded dynamically from a configuration file in XML format. This file is located at xsl/templates/templates.xml. Listing 6.3 shows an extract of it.

For every available output choice, it contains a <template> XML node. The subnodes <name> and <description> specify the name that appears in the list of choices respectively a description of the final output, both integrated in the wizard interface. The <resources> section defines preparatory actions as for example, copying images or applets used in generated webpages, or creating subfolders in the destination directory. The two available actions are copy and create, both applicable to directories (indicated by the tag <dir>) or single files (tag <file>).

The subnode <xsl> gives the location of the main XSL file. The attribute type indicates whether the output will result in a single document or a set of documents. If it is set to "single", then the wizard opens an instance of EV_FILE_SAVE_DIALOG for the user to choose the location of the destination file; if it is set to "multiple" it will be a EV_DIRECTORY_DIALOG to select the destination directory.

The file specified in the <xsl> tag of templates.xml only specifies the entry point for a transformation. Through <xsl:include> tags, the XSL file can reuse transformations defined in other files.

Listing 6.4 shows an example XSL file that includes two external XSL files: style_imp.xsl and webpages_imp.xsl. The included XSL files
6.8. ARCHITECTURE OF TRUCSTUDIO

Listing 6.3: Templates configuration file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <output_templates>
    <template type="model">
      <name>Webpages</name>
      <description>Webpages for each domain entity</description>
      <resources>
        <dir action="copy" from="hypertree/image" to="image"/>
        <dir action="copy" from="hypertree/image" to="image"/>
        <dir action="create" to="entities"/>
        <dir action="create" to="scripts"/>
      </resources>
      <xsl type="multiple">webpages.xsl</xsl>
    </template>
    ...
  </output_templates>
</templates>
```

define the behavior of the templates multipleoutput_specialized and make.css. Additionally, the XSL file produces a file containing JavaScript and an index.html.

The architecture of the output generation makes it possible to extend the templates in two steps: First, the configuration file containing the available templates must contain the new template (see Listing 6.3 for an extract of it); and second, a master XSL must take care of the transformation (see Listing 6.4 for an example master template). Extending the output mechanism requires knowledge of XML, knowledge of XSLT, and an understanding of the TXM file structure.

6.8.6 Structure report

The structure report tool of TrucStudio supports the analysis of the dependency structure of domain entities by generating an HTML report that optionally lists all direct links of entities, their transitive (indirect) dependencies, tabular representations of the transitive dependencies and cycles in the dependency structure.
Listing 6.4: Example of an XSL file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
  <xsl:include href="imp/style_imp.xsl"/>
  <xsl:include href="imp/webpages_imp.xsl"/>
  <xsl:param name="destination"></xsl:param>
  <xsl:template match="project">
    <xsl:result-document href="{$destination}scripts/scripts.js" format="txt">
      <xsl:text>
        function change_to(id) {
          //var id= extract_id_from_url(url);
          //var type= extract_type_from_url(url);
          parent.location.href = "../entities/"+id+".html ";
        }
      </xsl:text>
    </xsl:result-document>
    <xsl:call-template name="make_css">
      <xsl:with-param name="destination" select="concat(  $destination, 'scripts/style.css')"></xsl:with-param>
    </xsl:call-template>
    <xsl:call-template name="multipleoutput_specialized">
      <xsl:with-param name="destination" select="$destination"/>
      <xsl:copy-of select="entities/"/>
    </xsl:with-param>
  </xsl:template>
  <xsl:template match="/">
    <html>
      <head>
        <meta http-equiv="refresh" content="0; URL=entities/listindex.html"/>
      </head>
    </html>
  </xsl:template>
</xsl:stylesheet>
```
The detection algorithm for transitive dependencies uses a depth first search and marks visited elements in a Boolean matrix. TrucStudio uses two of these Boolean matrices – one for the transitive dependencies between notions and one for transitive dependencies between Trucs. Every row in the matrix corresponds to the outgoing transitive dependencies of an entity. The incoming transitive dependencies of the same entity can be found in the respective column of the matrix. The different output variants then look up the contents of the matrices to produce the textual and tabular representations of the transitive dependencies.

6.9 Summary

Curriculum and course planning is a key step in developing quality educational programs, but current practices very often lack a systematic approach. TrucStudio addresses this issue by providing tool support for modeling domains and courses based on the extended Truc approach.

The main interface of TrucStudio consists of a tree view for domain entities (clusters, Trucs and notions), a second tree view showing all courses and lectures, and a main manipulation interface. If the property view mode is enabled, the manipulation interface displays the properties of an entity in a form; if it is in graph view mode, it generates a graphical representation of the model.

The property view provides support for editing the descriptive elements of a selected entity and manipulating its relations to other entities. For lectures, the property view provides a rudimentary soundness check that reports possibly violated dependencies of the notions covered in a lecture; for courses, it contains a visualization of the mapping of lectures to timeslots.

The graph view provides graphical representation of the domain model as a Truc-notion graph. The look and feel of the graph can be adapted through manual layout of its elements, hiding or showing certain link or entity types, or through setting the colors and fonts. TrucStudio supports saving and loading the layout of the graph from XML and it offers export to PNG and PostScript files. For the course model, the graph view displays a diagram illustrating the lectures and their contained notions. The course diagram helps detect the focus of the single lectures and makes it possible to identify possibly missing notions in a course.

TrucStudio additionally supplies an extensible and flexible output generation mechanism, which makes it possible to produce a range of documents from the domain and course models (for example course webpages
or the browsable lexicon). The output generation helps distribute the Trucs and other selected information captured in the TrucStudio projects to other instructors or students.

We have also experimented with mechanisms that aim at easing the generation of domain models. The results of these tests show that the mechanisms are far from perfect, but they provide a direction for future work that could lead to promising outcomes.
CHAPTER 7

FUTURE WORK

This thesis contributes to three research directions connected to the teaching of programming and provides several possibilities for extensions and improvements of the research results. This chapter discusses directions for future research grouped by topic.

7.1 Student backgrounds

The study on prior programming knowledge of entering CS students offers many possible directions for future work. In particular, several points listed in the threats to validity (Section 3.3) should be addressed in the future:

- The results are based on self-assessment. A possible extension could make use of more objective measures through entrance tests. These tests could include a variety of questions. For example, they could ask students who claim to have prior programming knowledge to write pseudo-code solutions to algorithmic problems or to draw concept maps of object-oriented programming concepts. These tests could then help verify the accuracy of the data collected in our questionnaires.

- The similarities in the data collected at two European universities located in distinct countries indicate that the composition of the student body with respect to their prior programming experience is a global phenomenon. To substantiate this conjecture, it is necessary to broaden the scope of the study to more universities and countries.
The investigated student bodies consisted mostly of entering students with CS as their major. Since programming has nowadays reached the world at large, the study could be extended to students of other fields who are taking programming classes as part of their education. It would be interesting to see how these groups differ from the CS majors with respect to their prior computing experience.

The current study does not investigate how prior programming knowledge influences performance in the programming course. In particular, it would be interesting to explore whether students’ backgrounds influence the particular difficulties that they experience in the programming courses. Are there typical problems that only novices have and are there challenges that only the experienced programmers face? Such research could address the general question of whether students can, based on their prior programming exposure, be grouped into experience classes that then influence how instructors teach programming.

7.2 Trucs for object-oriented programming

This thesis describes an approach for modeling courses and their contents. It also provides a set of Trucs for object-oriented programming ready for use by instructors. Future work on the Trucs could follow two directions: (1) extending and improving the Trucs, and (2) improving the modeling techniques.

For many of the Trucs presented in Section 5.2, research studies have identified the most common confusions, but for certain topics, such studies are missing. The Trucs give an idea of possible topics on which future studies could focus; for example, many of the advanced topics such as deferred classes, genericity and multiple inheritance as well as Design by Contract and most of the container data structures could benefit from supplementary attention.

In the sections on common confusion of the Trucs (see Section 5.2), this thesis lists a set of criteria for examples and course contents. The section on examples does not reflect this knowledge and should in the future integrate it.

The Truc section on sample questions also does not draw on knowledge of common confusions. A future improvement of the Trucs could therefore tackle the production of sample questions that help
identify students’ misconceptions. This would then make the Trucs a diagnosis tool for instructors.

- The Truc descriptions currently mostly contain descriptions of technical aspects. A possible extension of the Trucs with pedagogical resources such as real-world analogies, or iconic and enactive representations of certain operations and concepts could help improve instruction substantially.

- The set of Trucs and the resulting domain model are based on the use of Eiffel as instructional programming language. Future work could investigate in what respect other object-oriented programming languages inflict changes on the model.

- This thesis provides the Trucs for object-oriented programming. Similar pedagogical resources would be useful for other areas of teaching such as algorithms and data structures, database systems, or concurrent programming.

### 7.3 TrucStudio

TrucStudio supports the Truc modeling approach. The TrucStudio tool as it stands today incorporates many advanced features, but some of these could be extended and improved.

Currently, TrucStudio does not provide any support for text formatting when entering the Truc descriptions; it treats all textual descriptions as plain text. An improved interface could offer formatting options similar to those found in simple text editors and in the background produce, for example, HTML code.

The structure report is currently implemented as a separate tool. In the future, we plan to integrate it in the output generation mechanism. This requires producing XSL templates that deduce implicit and indirect links and detect cycles or adapting the input XML (currently a TXM file) to include this information.

More research could also go into the automatic extraction of notions by using other techniques for extraction or other resources such as book indexes or glossaries. We have also started the development of a tool that automatically extracts lectures and their covered notions from teaching material in plain text format.
Another idea is to provide a subset of the functionality of TrucStudio in a web application similar to a Wiki allowing online collaboration to develop and maintain the Trucs and their notions.

This thesis refrains from quantitatively assessing the TrucStudio tool. The application of the approach to object-oriented programming and Introduction to Programming in a case study gives an indication of how it can help improve instruction, but a broader assessment needs to confirm this. In particular, we have not formally evaluated whether providing the resources to students helps them in learning programming.
CHAPTER 8

CONCLUSIONS

Teaching programming is one of the seven grand challenges of computing education [80] and it is the focus of many research efforts in computer science education as evidenced by numerous developments and studies in the field.

Advancing research with the goal of improving programming instruction can take many directions: developing new tools such as development environments targeted to novices, intelligent tutoring systems, or algorithm visualization tools; improving or extending existing ones; experimenting with new approaches with respect to the focus of instruction; providing additional resources to students and instructors; detecting common misconceptions when learning to program; etc.

This thesis contributes to several directions aiming at the improvement of programming instruction: it investigates the student diversity of introductory programming courses and the complexity of object-oriented concepts, which make teaching (and learning) programming difficult. The thesis also provides practical tools for the improvement of existing educational material.

The main focus and domain of application is the Introduction to Programming course for entering CS majors at ETH, which acts as a case study for the application of the developed tools and concepts and whose participants provided most of the collected data. The course follows an Inverted Curriculum approach and exposes students early to a large software framework.

As many introductory programming courses at the university level, Introduction to Programming faces a particularly diverse student body with respect to prior programming knowledge. To answer questions with respect to this prior knowledge, we have developed a questionnaire that
These data help depict the typical student taking Introduction to Programming. He already knows one programming language well and two or three programming languages a little. He most probably has learned these languages on his own and thus might not have good design principles or a clear idea of how things should be coded. But he has had exposure to programming before entering university. The outcome shows how broadly parts of CS concepts have reached some of the world at large, including the younger segments of the population.

The challenge, for those who teach introductory programming, is that we can neither ignore this background nor rely on it. The collected data show that about one third of the students have no or only very little programming expertise; roughly another third of the students know one or two programming languages well or very well; and another third have profound knowledge of three or more programming languages.

This distribution illustrates the wide range of student backgrounds that instructors of the Introduction to Programming course face. Moreover, there is currently no indication of a change in these numbers. One of the consequences could be to restructure the computer science curriculum accordingly. For example, it could provide additional courses to the novices and those that need more training.

The collected data indicate that the distributions of programming expertise amongst students are not restricted to ETH, but part of an international phenomenon. This is backed by a comparison of the data of ETH participants to those of students attending the University of York. The results show no significant differences between the two universities, except in the particular programming languages that the students know and in single items of their computer literacy.

The challenge of handling the diverse student body is one of many complexities that CS1 instructors face. When teaching object-oriented introductory programming another complexity becomes an issue: object-oriented concepts are manifold and tightly interrelated leading to difficulties in ensuring a complete and sound coverage with respect to prerequisites.

The contributions of this thesis follow two main directions to tackle the described difficulties: on one side, it proposes a modeling approach for teaching domains and associated courses yielding the possibility to check prerequisites and completeness of covered concepts complemented with tool support; and on the other side, it offers a comprehensive application of this approach to object-oriented programming and the course Introduction to Programming resulting in resources to improve its teaching.
The modeling approach is based on the notion of Truc [85] complemented with additional entities (notions and clusters) for the domain model and the entities course and lecture for the course model. It defines a set of relations between these entities, which provide structural information with respect to dependencies between concepts. These relations are the basis for calculating prerequisite violations and verifying the claim of high interrelatedness between concepts.

To support the generation of the domain and course models, we have developed a supporting tool, TrucStudio. TrucStudio helps educators devise, analyze, and maintain the models. The methodology is graphical and conjectured to be suited for an intuitive understanding. It provides visualizations of the domain and the course model to help ease analysis. In the same line, TrucStudio incorporates further analysis tools for the dependency structure of the modeled concepts. This brings both normative and descriptive support to the analysis of interrelatedness of concepts, and yields clear views of previous knowledge and prerequisites existing in teaching material.

An essential overhead is the generation of domain models. We have experimented with semi-automatic extraction of notions from teaching material. The results are promising, but not satisfactory and need further work. Another critical need is the generation of output from the given domain and course descriptions to ease distribution and reuse of these resources. TrucStudio’s output mechanism provides textual and graphical outputs as part of the TrucStudio distribution, but also allows instructors to create their own output formats and forms.

To assess TrucStudio and the extended Truc approach, we have applied them to object-oriented programming (with a focus on the programming language Eiffel) and the Introduction to Programming course. The result is a domain model consisting of 28 Trucs ranging from imperative programming constructs (such as Loop and Conditional) and object-oriented concepts (such as Object, Class and Inheritance), to data structures (for example Array and Hash table).

The Trucs contain descriptions of alternative names, summary, application, role in the field, benefits, pitfalls, examples, sample tests, and common confusions. Graphs of the contained notions and dependent Trucs complement these sections.

The domain graph for object-oriented programming builds the basis for the analysis of concept interrelatedness. This analysis suggests that there exists a core group of nine Trucs containing Argument passing, Class, Expression, Feature, Feature call, Object, Object creation, Reference, and System execution. Taking into account transitive dependencies, these Trucs are mu-
tually dependent on each other, and all other Trucs in the model require their coverage as a prerequisite. This result confirms the claim of high interrelatedness of concepts, but it also narrows down the scope of involved concept groups, which in return may help educators develop their teaching sequence.

Another main contribution of the developed Trucs is the collection of common confusions. They contain results from a large body of research over the past twenty years. To provide instructors with a tool to improve their teaching we have developed, based on the collected misconceptions, checklists with recommendations for examples and other teaching material such as slides. These checklists are reusable by a large range of instructors and allow them to check their teaching material for conformance.

The application of the TrucStudio approach also includes the course model of Introduction to Programming. The analysis of the prerequisite structure shows only few prerequisite violations, but there are several concepts missing in the course. We also added an informal application of the common confusions checklists to the teaching material. The overall outcome suggests that the current teaching material complies with only few of the recommendations, and the conjecture is that applying the Trucs will improve the course.
BIBLIOGRAPHY


[44] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


