Master Thesis

Teaching introductory programming with the inverted curriculum approach

Author(s): Pedroni, Michela

Publication Date: 2003

Permanent Link: https://doi.org/10.3929/ethz-a-005115128

Rights / License: In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Teaching Introductory Programming with the
Inverted Curriculum Approach

Diploma thesis

Michela Pedroni
Supervising Assistant: Bernd Schoeller
Supervising Professor: Bertrand Meyer

Department of Computer Science, ETH Zurich

September 2, 2003
Contents

1 Personal Motivation 4

I Teaching Introductory Programming - a Survey 6

2 Introduction 7
  2.1 The ‘what’ - changing curricular topics 7
  2.2 The ‘how’ - numerous teaching strategies 8
  2.3 The ‘who’ - a diverse student body 9
  2.4 Summary 11

3 Introduction to Learning Theory and Bloom’s Taxonomy 12
  3.1 Learning theory 12
    3.1.1 Behaviorism 13
    3.1.2 Cognitivism 14
    3.1.3 Constructivism 14
  3.2 Bloom’s taxonomy and programming 14
  3.3 Summary 16

4 Teaching Strategies 17
  4.1 Three didactic models 18
    4.1.1 The semiotic ladder 18
    4.1.2 Cognitive objectives taxonomy 18
    4.1.3 Problem-solving 19
  4.2 Order of topic introduction 19
    4.2.1 Bottom-up 19
      4.2.1.1 Example: Structure and Interpretation of computer programs 19
    4.2.2 Top-down 22
      4.2.2.1 Example: Big Java 22
    4.2.3 Top-down vs. bottom-up with object-orientation 22
  4.3 Large code bases vs. no code bases 24
    4.3.1 No code bases 24
    4.3.2 Large code bases 25
  4.4 Summary 25
## Contents

### 5 The Choice of a Programming Paradigm  
5.1 Procedural (imperative) programming  
5.1.1 Advantages  
5.1.2 Disadvantages  
5.2 Functional programming  
5.2.1 Advantages  
5.2.2 Disadvantages  
5.3 Object-oriented programming  
5.3.1 Advantages  
5.3.2 Disadvantages  
5.4 Summary  

### 6 The Choice of a Programming Language and Environment  
6.1 Teaching languages  
6.2 Teaching environments  
6.3 Summary  

### 7 The Inverted Curriculum, Object-Orientaion, and Eiffel  
7.1 The Inverted Curriculum  
7.2 The object-oriented paradigm  
7.3 Eiffel as a teaching language  
7.3.1 The language Eiffel  
7.3.2 The programming environment EiffelStudio 5.3  
7.4 Summary  

### 8 Conclusion  

### II Redesign of “Introduction to Programming” at ETH Zurich

### 9 A Wind of Change  
9.1 Change of teaching approach  
9.2 Change of programming language  
9.3 Change of computer science study  

### 10 Formal Issues  
10.1 General information  
10.2 Examination mode  
10.3 Organization of the home works  

### 11 Schedule of lectures  

### 12 Exercises  
12.1 Guidelines for the exercises  
12.2 Fitting exercises into the TRAFFIC application
CONTENTS

13 Course material 73
   13.1 Touch of Class text book 73
   13.2 The TRAFFIC software 75
      13.2.1 The TRAFFIC library 76
      13.2.2 The TRAFFIC application 76

14 Summary and outlook 80
Chapter 1

Personal Motivation

When I started my computer science study at ETH Zurich in fall 1997, I had no prior programming experience. I expected that the introductory programming course would teach me how to program, and I assumed that it would be fun to learn. Today I can say, that learning to program was the most frustrating experience of my computer science study.

At that time, the programming language that the students were exposed to was Oberon. Oberon, a descendant of the Pascal, Algol and Modula family, is quite a suitable language to teach introductory programming. And conforming to this, the language was not the origin of my problems. I struggled with the environment. For me, a novice programmer that was used to standard Windows applications, the radically different way of operating the Oberon environment was posing the most severe difficulties. For example, instead of using a standard menu offering copy, paste, and cut operations which then can be clicked, these operations had to be done by using a combination of mouse clicks (so called inter clicks). It happened to me more than once that instead of copying source code, I deleted it, because I had mistaken the two inter click combinations.

Besides having a difficult user interface, the Oberon environment that I used at home (under the Windows operating system) had the tendency to crash at any time. Many times I lost the code that I just had written, and I reinstalled Oberon more than once since such crashes often led to a corruption of the Oberon system itself. Furthermore, I struggled with error messages that were not understandable to me. There was one error message that haunted me through the whole course: “error 155 at position 0 - generation of new symbol file not allowed”. Back then, I didn’t understand what a symbol file is and had no idea how I should be able to fix this error. By coincidence, I found out that this error message would vanish, if I renamed my modules. This is what I did then. I started to initially call my modules A1, A2, A3, ... After the coding work was done, I gave the modules more meaningful names. This is an example of how an environment can influence the programming style of novice programmers (I changed this behavior when I started to program in a standard environment again). All in all, because of my struggles with the environment, during the first year I never felt totally comfortable with doing the programming tasks, and I probably also missed many key points that the home works meant to convey.

Today, not being a novice programmer any more, I think that the Oberon
system can be exciting, comfortable, and efficient for programming. But in my view, this environment was not at all suited for novice programmers. I believe that using another environment would have made my first experience with programming a much more enjoyable task.

When I found out that there is the possibility to write a diploma thesis that concerns the introductory programming course I enthusiastically confirmed. I was curious to see how such a course is being prepared and implemented. I also found the prospect of being able to influence the setup and to help in the development very interesting.

I am aware that this is an uncommon topic for a diploma thesis in computer science, and I believe that handling an ordinary software project as a diploma thesis would have been easier for me. But since I have been working as a programmer for more than four years gaining experience with industrial software development, I decided to tackle computer science from a totally different side: the teaching of introductory programming.
Part I

Teaching Introductory Programming - a Survey
Chapter 2

Introduction

Learning to program is a key objective in most introductory computing courses, yet many computing educators have voiced concern over whether their students are learning the necessary programming skills in those courses.


Learning to program as well as teaching programming is a challenge. The fact that many educators doubt that students after completing an introductory programming course have acquired the fundamental skills of programming reflects this. The primary goals of these courses - reasonable programming and problem-solving skills - seem to be unfulfilled by many of the students attending it. Still, lots of them pass the exam. There are two approaches to overcome this problem: don’t let students that can’t program pass (change the assessment style), or try to find a method to assure that the students learn what they are supposed to (change the course design). This survey will concentrate on the second approach.

But what are the problems of designing a successful introductory programming course? Why is this such a difficult task? The reasons for this can only be partially sketched. But to do so, we will risk a look at what we are teaching in an introductory programming course, who we are teaching it to, and how we are teaching it. This might elucidate why educators are complaining about difficulties in teaching introductory programming, why students are having trouble with it, and why the topic offers material for so many papers and discussions.

2.1 The ’what’ - changing curricular topics

Computer science is a very young, very rapidly changing and continually growing discipline. It has matured rather quickly compared to other academic areas
and still is evolving at a rapid speed. As the Joint Task Force on Computing Curricula in [42] puts it:

As we enter the new millennium, computer science is an enormously vibrant field. From its inception just half a century ago, computing has become the defining technology of our age. Computers are integral to modern culture and are the primary engine behind much of the world’s economic growth. The field, moreover, continues to evolve at an astonishing pace. New technologies are introduced continually, and existing ones become obsolete almost as soon as they appear.

The rapid evolution of the discipline has a profound effect on computer science education, affecting both content and pedagogy.

The youth of computer science is responsible for much of the difficulty of teaching it. In other sciences like mathematics, physics, or chemistry education profits from century long experience, and new advances in these fields usually do not (at least at the introductory level) affect the content of the curriculum. Technical advances in computer science, on the other hand, have an impact on the content of a curriculum by leading to a shift in curricular topics. This is also true for introductory programming courses. An example of the influence of new technologies and innovations on teaching are the ongoing discussions about starting with object-oriented programming, and the debates about whether graphics and multimedia should be included in introductory programming courses. On all of these issues, opinions diverge.

Deciding what to teach novice programmers about programming and, in particular, which programming language to teach to novice programmers, and how to teach it, is a common topic for debate within universities [9].

Different than in other sciences, there is no standard answer to the question of what exactly should be taught in an introductory programming course.

### 2.2 The 'how' - numerous teaching strategies

The question of how to teach introductory programming is closely intertwined with the question of what to teach. For example, the decision if object-oriented principles should be introduced or not influences the choice of a strategy to teach programming. On the other hand, the choice of a teaching model influences the topics that can be taught. So the dependencies between the 'how' and the 'what' are evident in both directions. As well as the 'what’, the 'how’ has been and still is the subject of many disputes and many different approaches have been proposed.

Throughout the history of computer science education, the structure of the introductory computer science course has been the subject of intense debate. Many strategies have been proposed over the years, most of which have strong proponents and equally strong detractors.
2.3. THE ‘WHO’ - A DIVERSE STUDENT BODY

Like the problem of selecting an implementation language, recommending a strategy for the introductory year of a computer science curriculum all too often takes on the character of a religious war that generates far more heat than light [42].

There have been many approaches proposed and practiced for introductory programming (see [42, 19, 38]). Each of the approaches has its advantages and its disadvantages and these again depend on the specific course setup (for example the number of students, the course topics, the course objectives, the amount of available time, prerequisites of the educational staff, and so on). So here again, there is not one optimal choice, and decisions must be made with care and cautiousness.

2.3 The ‘who’ - a diverse student body

Computers have become a mainstream equipment for academy, for industry, as well as for private users. The increasing percentage of internet users during the last couple of years is one evidence. Figure 2.1 shows the increasing percentage of Swiss people that use the Internet. The dark blue bars represent the percentage of heavy users (using the internet a couple of times a week), while the light blue bars show the percentage of Swiss residents that used the Internet at least once during the last six months. The numbers have been collected for each half-year from 1997 to 2001. Similar trends can be found for many other European and North American countries (see http://www.nua.ie/surveys/how_many_online/). This is just a sign for the importance that computation achieves in our lives. Its growing influence also affects education, in particular, the nature of the student body of introductory programming courses. “The explosion in the access to computing brings with it many changes that affect education, including a general increase in the famil-
2.3. THE 'WHO' - A DIVERSE STUDENT BODY

...
perceived to be miles ahead of them. On the other hand, students with programming skills often do not follow the course since they are initially bored by the presentation of material that is already familiar to them [6].

The goal to design a course that is interesting and challenging to both, the students that already come with good programming skills and the students that really are new to programming, is a very difficult one, maybe even a “mission impossible”. But still the effort to come as close to this goal as possible is worthwhile.

2.4 Summary

Designing and implementing an introductory programming course is a challenge. The three questions that were explored in the last three sections are the 'how', the 'what', and the 'who'. The 'who' is the variable that the educators designing a course don’t have any influence on. They have however to choose what is going to be taught and how this will be done. These choices are difficult to make because they are closely intertwined and because there is no optimal decision. The only way to go is trading off assets and drawbacks of the different choices and trying to detect and correct possible pitfalls before they occur.
Chapter 3

Introduction to Learning Theory and Bloom’s Taxonomy

Because of changes in the field of computer science, the context and content of computer science education at school is continuously evolving. But new topics alone do not lead to concrete and suitable courses. Too often, demands of the practice obscure the underlying theory. Learning theories, didactic knowledge, and computer science itself should be heeded in educational practice and the design of learning environments.

Carsten Schulte, Towards a pedagogical framework for teaching programming and object-oriented modelling in secondary education.

3.1 Learning theory

This chapter will give a short introduction into learning theory to broaden the understanding of how students learn. There are three fundamentally different ideas about the nature of learning and what the properties/nature of knowledge are. In other words the approaches not only include a view of how learning occurs, but also a view of what knowledge actually is. These three basic approaches or psychological theories are referred to as behaviorism, cognitivism and constructivism.
3.1. LEARNING THEORY

3.1.1 Behaviorism

Behaviorism is the oldest of the three theories and was very dominant in the 1950s & 60s. At its core is the classical stimulus-response pattern. In behaviorism, stimuli are impulses that come from the environment. These stimuli evoke a behavior, which is called the response. The mind is viewed as a “black box”, totally ignoring the possibility of thought processes. “Behaviorism is a theory of animal and human learning that only focuses on objectively observable behaviors and discounts mental activities. Behavior theorists define learning as nothing more than the acquisition of new behavior (see [20]).” One of the very important aspects of behaviorism is that the learner is viewed as adapting to the environment and learning is seen largely as a passive process (from the learner’s point of view). The learner merely responds to the stimuli from the environment. Actual learning occurs through conditioning until a response is under the control of the stimulus, or, in other words, until the association between a stimulus and the response is constructed. There are two different types of conditioning:

Classical conditioning: A stimulus leads to a natural reflexive reaction, which is then connected through timely or spatial closeness to a new stimulus. For example, Pavlov (1849 - 1936) conducted an experiment with a dog where whenever he placed food in front of the dog he first rang a bell. At the beginning the dog salivated because of the food, but after conditioning the ringing of the bell alone produced salivation.

Behavioral or operational conditioning: A response to a stimulus is reinforced. There are four operant conditioning mechanisms: positive reinforcement, negative reinforcement, punishment, and extinction (or non-reinforcement). Positive reinforcement builds upon the fact that rewarded responses are likely to be repeated. Negative reinforcement occurs when the disappearance of a painful or undesirable situation is connected to a distinct behavior (response). Punishment will likely lead to a suppression of responses by exposure to undesirable or painful consequences. Non-reinforcement acts on the assumption that ignoring certain behavior will lead to the extinction of the behavior (ignoring a misbehaving child usually extinguishes that behavior).

There have been many criticisms on behaviorism including the following three:

• Behaviorism does not account for all kinds of learning since it disregards the activities of the mind. They are not simple stimulus-response organisms.

• Behaviorism is not able to explain learning when there is no reinforcement. An example would be new language patterns of young children.

• Research has shown that animals adapt their reinforced patterns to new information. For instance, a rat can shift its behavior to respond to changes in the layout of a maze it had previously mastered through reinforcement.

1Information taken from [20, 34, 15]
3.1.2 Cognitivism

"Cognitive theorists recognize that much learning involves associations established through contiguity and repetition. They also acknowledge the importance of reinforcement, although they stress its role in providing feedback about the correctness of responses over its role as a motivator. However, even while accepting such behavioristic concepts, cognitive theorists view learning as involving the acquisition or reorganization of the cognitive structures through which humans process and store information [21]."

Cognitions are the processes by which an organism gains knowledge about its environment. In the human case, such processes are perception, imagination, thinking, judgment, and speech. The gained knowledge is saved in an internal structure, the so-called schema. Schemas are cognitive structures that get extended, combined, and altered to accommodate new information. The structuring process is subjective and active (a major difference to the behaviorist’s view!). However, knowledge is viewed as given and absolute just like in the behavioristic school. The use of feedback to guide and support the learner to create accurate mental connections is a key component in the cognitive theory.

3.1.3 Constructivism

Constructivism is based on the premise that each learner constructs his own reality based on his perception of experiences. Each learner has his own “rules” and “mental models” that he built up through previous experiences. Learning occurs through the search for meaning of new objects and events by adapting the prior “mental model”. Whereas in behaviorism and cognitivism, knowledge is mainly seen as given and absolute (meaning that knowledge is passive and largely automatic responses to external factors), constructivists view knowledge as an individually constructed entity and therefore learning is an active process. Thus knowledge cannot be transmitted from one individual to another, it must be reconstructed by each person.

Constructivists emphasize the importance of the learner rather than the dominance of the teacher. As opposed to behaviorism and cognitivism where the didactic and methodic presentation of the subject matter is much more relevant, in constructivism triggering and optimizing learning processes by setting up the learning environment come to the fore. The learning situation should be as authentic - regarding the future tasks in real-world - as possible, and assessment should be integrated with the tasks rather than being a separate activity. Another point that gains importance in constructivism is cooperative learning.

3.2 Bloom’s taxonomy and programming

Bloom’s taxonomy has been accepted since the 1950’s as a valuable tool for classifying cognitive skills in education. “The taxonomy contains six levels. Each successive level depends upon behaviors acquired at earlier levels [31].”
According to Lister and Leaney, Bloom’s taxonomy of Educational Objectives can be applied to the skills that are demanded from a novice programmer. Following up the six levels (beginning with the lowest level) with example tasks found in introductory programming will be described.

**Knowledge:** “the recall of specifics and universals, the recall of methods, and processes, or the recall of a pattern, structure, or setting. Bloom’s word list of activities includes ”memorize”, ”name”, ”recognize” and ”relate” [31].” In programming this would mainly be syntax-details or questions like: “State the three types of loops in Java” or “Which type of loop will be executed at least once in Java?”.

**Comprehension:** “a type of understanding when the individual knows what is being communicated and can make use of the materials or idea being communicated without necessarily relating it to other material or seeing its fullest implications. Bloom’s word list of activities includes ”restate” and ”translate” [31].” An example of a task at comprehension level would be translating pseudo code or spoken language into simple programming statements (like “Write an Eiffel statement that declares a variable that holds an integer value”).

**Application:** “is the use of abstractions in particular and concrete situations and may include general ideas, rules of procedures, generalized methods, technical principles, ideas, and theories that must be remembered and applied. Bloom’s word list of activities includes ”calculate”, ”solve”, and ”write”. We believe that many traditional programming assignments are tasks at the application level, provided the assignment task is quite specific [31].” Writing a simple class where the class interface including feature declarations is specified could be a task that conforms to the application level of Bloom’s taxonomy (providing that the implementation of the methods is not algorithmically complex or too long).

**Analysis:** “emphasizes the breakdown of the material into its constituent parts and the detection of the relationships of the parts and of the way they are organized. Bloom’s word list of activities includes ”categorize”, ”differentiate”, ”discriminate” and ”distinguish” [31].” An example exercise in Java on the analysis level could be to present students with an implementation of the Model-View-Controller and asking them to identify the parts of the code that implement the Model, the View, and the Controller.

**Synthesis:** “the putting together of elements and parts so as to form a whole. Bloom’s word list of activities includes ”create”, ”design”, ”organize”, and ”plan”. As with the application level of the taxonomy, we believe that many traditional programming assignments are tasks at the synthesis level. The difference from an application task is that a synthesis task contains more sophisticated design choices [31].” Problem-solving in the sense of taking a problem description, refining it, decomposing it into classes or subproblems, implementing the methods of classes or the solutions to the subproblems, and putting them together to solve the initial problem is seen as belonging to the synthesis level of Bloom’s taxonomy.
Evaluation: “the making of judgments about the value, for some purpose of a solution, appraising the extent to which the solution is accurate, effective, economical, or satisfying. Bloom’s word list of activities includes "assess", "evaluate", and “judge”. This level of the taxonomy maps easily to the teaching of programming, as our discipline values clear, concise and efficient code. However, traditionally it is the teachers who evaluate the students’ code. In the context of Bloom, "evaluation" is an activity the students perform, and teachers grade the quality of such evaluations. An explicit approach to "evaluation" would require the students to review some given code, and we would grade that review. Traditionally, our discipline assesses “evaluation” implicitly; under the assumption that students who design good programs are good at self-evaluation [31].”

3.3 Summary

In this excursion into psychology and philosophy, the three major learning theories – behaviorism, cognitivism, and constructivism – have been described.

Bloom’s cognitive skill levels find use in the design of exercises and the assessment of programming skills. In this respect, the six skill levels were described including short examples of tasks on the according level. Bloom’s skill level might even find their way into the design of the home works for “Introduction to Programming” at ETH Zurich.
Chapter 4

Teaching Strategies

The truth is that no ideal strategy to teach introductory programming has yet been found, and that every approach has strengths and weaknesses. Given the current state of the art in this area, we are convinced that no one-size-fits-all approach will succeed at all institutions. Because introductory programs differ so dramatically in their goals, structure, and intended audience, we need a range of strategies that have been validated by practice. Moreover, we must encourage institutions and individual faculty members to continue experimentation in this area. Given a field that changes as rapidly as computer science, pedagogical innovation is necessary for continued success.


As already mentioned, there have been described a multitude of different teaching strategies in literature. In most cases, the teaching strategies are based on a distinct idea that then influences the structure, setup, and design of the introductory programming course. Since there are just too many teaching approaches to describe them all, three fundamental issues are described in the following sections. First, the different didactic models as found in [26] will be described. These are mostly aimed at the question of how to introduce new concepts in an introductory course (for example the notion of class in a course that teaches object-oriented programming). The second section addresses the question of the order that topics are introduced. With the desire to teach object-oriented techniques in introductory programming courses, the call to rearrange the sequence of curricular topics in a top-down fashion – instead of the traditional bottom-up way – has been voiced. For each, the bottom-up and the top-down approach, an example order of topics is described. The third section describes the advantages and problems of providing students with extensive code frameworks for their exercises.
4.1 Three didactic models

Kaasboll describes in [26] three main didactic models. Such a model can be used to structure a whole course or may be repeatedly applied within a larger framework.

4.1.1 The semiotic ladder

The first didactic model is called “the semiotic ladder”. The teaching and learning sequence starts out with teaching basic syntactic constructs, then goes on to teach their semantics, while at the end teaching the pragmatics. “Its rationale is that syntactic knowledge is needed to express anything, and therefore it should precede the learning of meaning of language constructs. When knowing the meaning, the student can start to learn how to use the language for specific purposes, which is the pragmatics [26].” Usually, the “semiotic ladder” is applied repeatedly to different topics of a course, it can, however, also be applied to a course as a whole provided the syntax is small enough.

![Semiotic ladder diagram]

4.1.2 Cognitive objectives taxonomy

The second didactic model resembles Bloom’s taxonomy of cognitive objectives (see 3.2) and is entitled in [26] by “cognitive objectives taxonomy”. Here, the following four steps are used to teach programming: First, run a program. Then, read the program. Change the program. And finally, create a program (this step is optional).

![Cognitive objectives taxonomy diagram]

In the case of object-oriented teaching with special emphasis on reuse the four steps could be redefined to: First, use a class through its interface. Then, read the implementation. Change the class (adding features, changing features). And finally, create a new class.
4.2. ORDER OF TOPIC INTRODUCTION

4.1.3 Problem-solving

The third didactic model is called “problem solving”. It is strongly motivated by the constructivist’s view of how learning occurs. “Through solving problems, the students should extend their experience and repertoire of practice, and the basis for the process is the knowledge structure of the field of programming. The problem solving process is guided by methods and environments. Compared to the previously mentioned approaches, this one stresses the input and outcome of the learning process in terms of knowledge and personal behavior [26].” There has been a four-stage problem-solving process developed that has been used to teach computer science: First, try to understand the problem by structuring, dividing, clarifying, and finding sample Input/Output. Second, design a solution by finding related problems and solutions and checking the related solution against the sample Input/Output. Then, write the final solution by completing and adapting the found solutions to the problem. Finally, review the solution by testing it and summarizing what has been learned.

<table>
<thead>
<tr>
<th></th>
<th>Understanding</th>
<th>Design</th>
<th>Writing</th>
<th>Review</th>
</tr>
</thead>
</table>

4.2 Order of topic introduction

In this section we will take a look at teaching material that represents different kinds of sequences of topic introduction. Traditionally, introductory programming courses covered topics in a bottom-up fashion: They started with the standard low-level constructs of the programming language like basic data types, variables and assignment, then went on with control structures, introduced the notion of procedures, and finally, if time permitted, moved on to principles and techniques of structuring and designing large systems. With the increasing popularity of object-oriented languages and the growing need for the design of large software systems, the idea of rearranging the sequence of topics in a new way has been voiced. In such so-called top-down approaches, courses start with higher level constructs (like objects and classes) and defer low-level constructs (like assignment and control structures) to lectures later in the course. The order of topics of the first couple of lectures usually decides whether a course is taught top-down or bottom-up. The following subsections hold an example for either way of introducing curricular topics.

4.2.1 Bottom-up

4.2.1.1 Example: Structure and Interpretation of computer programs

The material in the book “Structure and Interpretation of computer programs” by Abelson and Sussman [2] has been used since 1980 at MIT for the introductory programming course. It emphasizes the role of computer languages as vehicles for expressing knowledge and it presents basic principles of abstraction and
modularity, together with essential techniques for designing and implementing computer languages. The programming language that the book uses is Scheme, a dialect of Lisp that attempts to unite the powers of Lisp and Algol. A wide variety of programming paradigms find expression in Scheme.

The book is divided into five chapters (see listing 1):

Chapter I is entitled “Building Abstractions with Procedures” and introduces the basic building blocks for simple numerical programs. It starts with programming elements like expressions, variables, compound procedures, conditionals, then introduces recursion and iteration shown on various traditional examples (tree recursion, exponentiation), and finally shows how higher-order procedures can be used to formulate abstractions.

Chapter II turns the attention from procedures to data and is called “Building Abstractions with Data”. First, the concept of data abstraction is introduced on the example of rational numbers, then the use of list to construct sequences and hierarchical structures like trees and mappings is shown. Next, symbolic data is covered, followed by a chapter on multiple representations of data using the example of complex numbers. Part II is closed with a chapter on generic operations.

Chapter III is dedicated to organizational principles that help structuring large systems. The first chapter of this part, presents the student with the view of a system as a collection of objects with local state and introduces assignment. Next, the effects of allowing assignment are discussed and the rules for evaluation are shown. Then mutable data objects are introduced, followed by concurrency and an alternative way of modeling state: data streams.

Chapter IV is called “Metalinguistic Abstraction”. It starts with a section on the metacircular evaluator that implements a subset of the Lisp dialect Scheme using the programming language Lisp. The second section enhances the previously used Scheme evaluator to handle stream programs in a more convenient way. Next, the Scheme evaluator is extended to support the programming paradigm called non-deterministic computations. Finally, an evaluator for logic programming is developed.

Chapter V is dedicated to explain how Lisp programs are executed on computers. In order to do so, processes are described in terms of a step-by-step operation of a traditional computer, a register machine. In the first section a definition of register machines is given. This definition is used in the second section to implement a simple simulator for register machines. Next, the simulator is enhanced to handle storage operations, and subsequently an evaluator that was described in Part IV is implemented as a register machine. The book terminates with a study of a simple compiler that translates Scheme programs into sequences of instructions that can be executed directly with the registers and operations of the register machine.

This book is exemplary for bottom-up topic introduction: It starts with the basic Scheme programming elements like expressions and variables, then covers procedures, and introduces in chapter 3 modularity and objects.
4.2. ORDER OF TOPIC INTRODUCTION

<table>
<thead>
<tr>
<th>1</th>
<th>Building Abstractions with Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The Elements of Programming</td>
</tr>
<tr>
<td>1.2</td>
<td>Procedures and the Processes They Generate</td>
</tr>
<tr>
<td>1.3</td>
<td>Formulating Abstractions with Higher-Order Procedures</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>Building Abstractions with Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Introduction to Data Abstraction</td>
</tr>
<tr>
<td>2.2</td>
<td>Hierarchical Data and the Closure Property</td>
</tr>
<tr>
<td>2.3</td>
<td>Symbolic Data</td>
</tr>
<tr>
<td>2.4</td>
<td>Multiple Representations for Abstract Data</td>
</tr>
<tr>
<td>2.5</td>
<td>Systems with Generic Operations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>Modularity, Objects, and State</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Assignment and Local State</td>
</tr>
<tr>
<td>3.2</td>
<td>The Environment Model of Evaluation</td>
</tr>
<tr>
<td>3.3</td>
<td>Modeling with Mutable Data</td>
</tr>
<tr>
<td>3.4</td>
<td>Concurrency: Time Is of the Essence</td>
</tr>
<tr>
<td>3.5</td>
<td>Streams</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>Metalinguistic Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The Metacircular Evaluator</td>
</tr>
<tr>
<td>4.2</td>
<td>Variations on a Scheme – Lazy Evaluation</td>
</tr>
<tr>
<td>4.3</td>
<td>Variations on a Scheme – Nondeterministic Computing</td>
</tr>
<tr>
<td>4.4</td>
<td>Logic Programming</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>Computing with Register Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Designing Register Machines</td>
</tr>
<tr>
<td>5.2</td>
<td>A Register-Machine Simulator</td>
</tr>
<tr>
<td>5.3</td>
<td>Storage Allocation and Garbage Collection</td>
</tr>
<tr>
<td>5.4</td>
<td>The Explicit-Control Evaluator</td>
</tr>
<tr>
<td>5.5</td>
<td>Compilation</td>
</tr>
</tbody>
</table>

Algorithm 1: Structure and Interpretation of Computer Programs, table of contents.
4.2. ORDER OF TOPIC INTRODUCTION

4.2.2 Top-down

4.2.2.1 Example: Big Java

The book “Big Java” by Horstmann (see [23]) is one of the examples that tackle topic introduction top-down. The book can be naturally grouped into four parts (see below).

Part I, holding chapters 1 through 8, covers the basics of object-oriented programming. It starts out with a chapter where the hard- and software setup is summarized and a first “hello-world” program is written. Within this, method calls are covered. The second chapter superficially introduces the notion of object and class in conjunction with object creation, variables, class definitions, class implementation and class interface. Students will be able to write simple classes with the knowledge they gained from chapter 2. The next chapter covers fundamental data types like int, double, and String, and introduces assignments, constants, and static methods. Chapter 4 covers applets and graphics. Next, in chapter 5 and 6 control structures are introduced. Chapter 7 takes up class design in a more profound and systematic way. The last chapter that belongs to part I covers testing and debugging issues.

Chapters 9 through 16 belong to part II. Chapter 9 discusses interfaces and polymorphism. Next, event-driven programming is covered, followed by inheritance. Chapter 12 is dedicated to GUI development. Subsequently, arrays are introduced, followed by exception handling and streams. Part II ends with a discussion of system design issues.

Part III contains chapters 17 through 20 and covers standard algorithms and data structures. First, recursion as an algorithmic method is introduced, followed by various traditional sorting and searching algorithms. Next, in chapter 19 linked lists, stacks and queues are treated. Chapter 20 introduces the set and map data types, hash tables, binary trees, tree sets and tree maps.

Part IV covers more advanced Java programming techniques like multithreading, database and network programming.

As described in the book, part III and IV go beyond a one-semester introductory programming course. The order of topic introduction – especially of part I – shows that this book is designed in a top-down way: before the standard low-level concepts like assignment, loops, and conditionals are introduced, objects and rudimentary classes are covered.

4.2.3 Top-down vs. bottom-up with object-orientation

While introducing topics bottom-up undoubtedly makes sense for courses that don’t introduce object-oriented techniques, the call to introduce object-oriented programming in an object-oriented way – meaning top-down – has been voiced. According to Hirshfield, Decker and others teaching introductory programming top-down entails many advantages over introducing object-orientation bottom-up:

- Teaching introductory programming in a bottom-up fashion focuses on ‘programming-in-the-small’. This view stands opposed to the reality in industrial practice. Therefore the students miss the important concepts of ‘programming-in-the-large’. “The emphasis in these courses is on “programming-in-the-small” and those students who succeed often leave the introductory...
Algorithm 2: Big Java, table of contents.
courses with the notion that attention to detail is the first step and the most important part to problem-solving [22].” The top-down approach focuses on ‘programming-in-the-large’ and tries to convey important object-oriented techniques and software engineering principles as opposed to syntactic, programming language dependent details. “When these students enter our CS2 course they have a better understanding and appreciation for such things as top-down design, procedural and data abstraction, information hiding and reusable software components [22].”

• Because the bottom-up courses at the beginning of the computer science study emphasize ‘programming-in-the-small’, students will have to do a major turn-around during later studies. “In an upper division software course the paradigm is reversed and conceptualization and organization of the problem solution is emphasized as paramount to the problem-solving activity and students are taught to postpone details. This mid-course reversal of instructional strategy confuses students and many graduate without a full understanding of the problem-solving approach that higher level software development courses stress [22].” By using a top-down approach the turn-around can be avoided.

• Decker and Hirshfield think that bottom-up approaches do not emphasize important concepts like reusability, information hiding, and clean structuring of solutions. “Students cannot be motivated to appreciate how things that don’t seem vital to CS1 (like the need for subprograms, parameters, information hiding, effective user interface) are truly important in real programs. [...] it should not surprise us that our students can’t appreciate the “big picture” when what we spend most of our time teaching them is the “little picture” [10].”

4.3 Large code bases vs. no code bases

With the increasing size of programs from real-world the call for stressing ‘programming-in-the-large’ rather than ‘programming-in-the-small’ – even in introductory programming courses – has been voiced. It seems intuitive that this leads to the idea of providing students with large code frameworks, in order to set up a teaching environment that resembles what they will encounter later in industry. The use of extensive code bases for introductory programming does not come without problems. Following up, a short discussion listing advantages and disadvantages of both extremes, using no code framework and using large frameworks, is given.

4.3.1 No code bases

Many introductory programming courses don’t provide large code bases to their students, but teach programming with the use of small examples.

One major advantage comes with not providing code bases: The exercises require no or only a small amount of reading of code that is provided and written by somebody else than the student. The computer science community agrees that reading code is more difficult than writing it. Therefore solving exercises
that don’t rely on code that is written by somebody else seems to be easier for students.

On the other hand, not giving code bases to students entails also two major disadvantages: First, because exercises need to be small, only introductory examples can be used. These examples are usually quite uninteresting to students that are used to flashy, very elaborate programs that they encounter outside of the introductory programming course.

4.3.2 Large code bases

In a course that offers large code bases, exercises are to be solved by modifying and extending the framework.

Using large code bases entails three major advantages: First, since exercises rely on an extensive code framework, interesting exercises can be designed. This includes the use of multimedia, graphical user interfaces and many other modern software features. Second, the provided code-base offers examples of well designed software components. Therefore students are encouraged to imitate good design and always have enough information that they can look at while programming. There have been quite a lot of teaching approaches proposed that attach great importance to students reading working, well written programs before writing their own. And third, large code frameworks help convey the “big picture” of software development. If teaching strategies want to stress concepts that come with ‘programming-in-the-large’, it should be done working on large software systems.

There are, however, also problems that come with the use of large frameworks: First, while providing a code-base allows us to come up with interesting examples, the fact that students are faced with a large amount of code that they don’t know might confuse them. This is especially the case if the given code cannot be hidden from the students. They might have the impression that they need to understand the whole framework before being able to use it. Second, using an extensive code framework involves more work for the educators. The least of additional expenses is the development of the code-base that is given to the students. The implementation of it requires carefulness and thoughtfulness on the part of the designer since it will be used as an example of good design and students will be encouraged to imitate it.

4.4 Summary

Because of the fact that there exist too many teaching approaches to describe all of them, the focus was put on three issues concerning teaching strategies: didactic models, order of topic introduction, and the use of large code bases.

There are three general didactic models that can be used to teach introductory programming: the semiotic ladder, cognitive objectives taxonomy, and the problem-solving model. As Kaasboll states “Models for teaching programming have been suggested in the literature. Interviews of teachers indicate that they normally do not relate to such models. Since referring explicitly to didactic models in teaching may improve learning, using models seems preferable [26].”

Generally, there are two major ways of topic introduction: bottom-up and top-down. While bottom-up topic introduction has been used in introductory
programming courses from the very beginning, during the last ten years instructors have started to experiment with top-down strategies. In most cases, top-down topic introduction only makes sense if it is coupled with object-orientation. To illustrate the differences between a top-down and a bottom-up topic introduction two examples have been given.

The third issue that was discussed concerns the use of code bases in introductory programming courses. While using code frameworks comes with some benefits, it also entails some problems that need to be addressed.
Chapter 5

The Choice of a Programming Paradigm

Just a decade ago, Pascal was firmly established as the nearly universal standard language for introducing computer science majors to programming. Even then, though, rumblings of discontent with standard Pascal’s lack of support for data abstraction and information hiding were beginning. [...] The increasing acceptance of the importance of non-procedural programming paradigms has further complicated the problem of choosing a vehicle for teaching programming to beginners. There is an increasing consensus that our students should be introduced to both object-oriented technology and functional programming at some point in the undergraduate curriculum. Some educators are advocating presenting one of these rather than the procedural (imperative) paradigm in the first programming course; others advocate the presentation of multiple paradigms.

Susan S. Brilliant and Timothy R. Wiseman, The first programming paradigm and language dilemma.
5.1 Procedural (imperative) programming

The imperative programming paradigm is the traditional language paradigm for introductory programming courses. It is still widely used, although its status as the only reasonable paradigm for first steps in programming has changed. In recent years it became more and more replaced by the functional and object-oriented paradigms.

5.1.1 Advantages

The following advantages can be mentioned for procedural programming:

- The procedural paradigm is the traditional paradigm for introductory programming courses and has been used widely and for many years. This results in teaching material that has been thoroughly tested and refined to fit the needs of an introductory programming course. This is also the paradigm in which most of the people that are in charge for the introductory programming courses have learned to program. Therefore they might feel more at ease to teach this paradigm as opposed to the other ones.

- Most of the other paradigms have built-in imperative programming constructs. “Students do need exposure to the traditional imperative style of programming, which remains in widespread use and which is an integral part of any object-oriented language [42].”

5.1.2 Disadvantages

- The teaching community is widely agreeing that object-orientation is not only an extension of the imperative paradigm, but requires a different way of thinking. “Object-orientation is an underlying paradigm that shapes our whole way of thinking about how to map a problem onto an algorithmic model. It determines in fundamental ways the structure of even simple programs. It cannot be “added on” to other language constructs; rather it replaces the fundamental structure of procedural programming [27].” The paradigm shift from procedural to object-oriented programming has proven to be difficult, it seems, however, that the shift backwards (from object-oriented to procedural) is much easier. “Many who have made the leap [from the procedural to the object-oriented paradigm], including us, now see that the paradigm shift is tough only in one direction - from procedural to OOP [11].”

- Imperative programming languages usually imply the need to “focus on syntax and the particular characteristics of a programming language, leading students to concentrate on these relatively unimportant details rather than the underlying algorithmic skills [42].”

5.2 Functional programming

The functional programming paradigm has received much less attention in introductory programming courses than the imperative and the object-oriented.
There have, however, been courses taught using functional languages like Scheme, Haskell, or SML.

5.2.1 Advantages

Using the functional programming paradigm has shown the following advantages:

- The minimalist syntax of functional languages and their clean semantics help the student focus on fundamental principles, essential concepts and techniques of programming. “The lightweight and orthogonal syntax of Haskell helps to relegate syntactic issues to the background and to concentrate on general programming concepts. [...] The clean semantic of functional languages leads to a good integration of the aims of programming techniques with computing concepts and theory [6].”

- Some of the important concepts of computing, for example recursion and linked data structures, come naturally with the functional programming paradigm. As a consequence of this, they can and must (see also 5.2.2) be covered earlier than with the use of another programming paradigm.

- Functional languages are not mainstream. Because a programming paradigm and language are used that relatively few students have experience with, the effect of the diversity of backgrounds of the students is reduced. “As we all know, functional languages can, by no stretch of imagination, be called mainstream. Funnily enough, this apparent disadvantage turns out to be one of the big selling points for using Haskell: as almost no first year student has any prior knowledge of Haskell, it acts as an equalizer between students who already bring some programming skills into the course and those who do not. [...] In our experience, functional languages successfully serve as an equalizer [6].”

5.2.2 Disadvantages

There are however dangers in taking a functional programming language.

- The high level of abstraction that students have to think in is not quite easy to get used to. “This approach typically requires students to think much more abstractly at an early stage than is true with more traditional programming languages. While forcing students to think in this way is certainly valuable and needs to be part of the curriculum at some point, placing it so early can discourage some students with less experience in that style of thought [42].”

- Recursion must be introduced at a very early stage of the course. Chakravarty and Keller say that “there is one topic that we have to tackle early on due to using a functional language and that students have a tendency to perceive as a significant obstacle: recursion. In imperative languages, iteration can be introduced early on as a simpler form of repetition and recursion be moved further back in the course, until trees are covered [6].”

- The transition to other paradigms can be difficult (see [3]).
5.3. OBJECT-ORIENTED PROGRAMMING

• Functional languages are not mainstream (see also 5.2.1). “Students may react skeptically to learning a language that they see as outside of the mainstream [42].” and “the perception that Scheme is not a “real world” language also presents difficulties - one respondent laments that “mom and dad have never heard of Scheme.” [3]

5.3 Object-oriented programming

Object-oriented programming has, in recent years, become the most influential programming paradigm. It is widely used in education and industry, and almost every university teaches object-orientation somewhere in its curriculum. The software community more or less agrees that teaching object-oriented programming is a good thing [27].

The main question though is whether object-oriented programming should be the first programming paradigm that students encounter in their undergraduate studies. “For a long time, object-oriented programming was considered an advanced subject that was taught late in the curriculum. This is slowly changing: more and more universities have started to teach object-orientation in their first programming course [27].” The debate on whether the object-oriented paradigm is adequate as a first paradigm, is still discussed fiercely. The trend, however, leads away from procedural or functional programming to object-orientation. The object-oriented paradigm is very powerful and has many advantages for novice programmers, there are however also dangers in teaching it as the first programming paradigm.

5.3.1 Advantages

• “Introducing the object-oriented paradigm from the beginning allows us to exploit it as a design medium [11].” The ability to use OOP as a design medium simultaneously incorporates a very strong real-world modeling ability. According to DeClue in [12]: “A very important aspect of object-oriented methods relates to the paradigm’s ability to relate problem-solving techniques directly to real-world problems.” At the same time it results in the “ability of objects to impose a natural organization on problem solutions which does not so easily occur in the structured programming paradigm [12].” Decker and Hirshfield agree with this view: “One of the beauties (to us) of the OOP paradigm is by merely describing classes one imposes an organization on a problem that not only clearly reflects those aspects of the real world being modeled, but also serves as a first crack at an algorithmic description of those same aspects. By associating functions (in C++) with classes, you have already taken a giant step, and for many students this is the toughest step by far, towards breaking a problem down into more manageable subproblems [11].”

• “From an educational standpoint, one of the most useful benefits is that the model of the cooperating objects is intuitively close to human experience and expectation [40].” Schwill agrees in [41] that “this paradigm is
consistent with the natural way of human thinking, and therefore particularly suitable for teaching computer science at an introductory level.”

- The object-oriented paradigm opens the doors for many new methods that could be used both for teaching and for learning. “By viewing object-orientation as a concept, successful research-based methods for teaching and learning concepts become available for use. These methods include presenting high-quality examples, identifying criterial attributes, comparing and contrasting using attributes, and using examples with increasing detail [12].” Another benefit is that “the ability for students to make use of ready-made objects in their applications opens a wide range of possibilities for real-world and interesting examples and exercises [29].”

- The dreaded paradigm shift can be avoided if object-orientation is the first paradigm to be instilled (see also [11]). “Problems with the paradigm shift in moving between object-oriented and non-object-oriented environments seem to be reduced. It has been found that many students whose first programming language is a procedural language, such as Pascal, experience problems in adjusting to the object-oriented paradigm. On the other hand, switching from an object-oriented language to a non-object-oriented one is not anticipated to cause as much difficulty (provided the syntax is not too different) [29].”

- The object-oriented paradigm directly supports most of the software engineering principles (like code reuse, encapsulation, incremental development, testing, and program design) that are difficult to convey using the procedural paradigm. “It [the object-oriented paradigm] elegantly supports the concepts that we have been trying to teach for many years, such as well structured programming, modularization and program design. It also supports techniques for approaching problems that have only more recently made their way into the curriculum: programming in teams, maintenance of large systems and software reuse. In short, object-oriented programming seems to be a good tool for teaching those programming methodologies that we consider important [27].”

- “Object-orientation currently is the most popular paradigm in industry. This provides a strong motivation for students, since they learn state-of-the-art technology that they can use in their later career [1].”

5.3.2 Disadvantages

While the object-oriented paradigm seems to offer many advantages for introductory programming courses there are however also some drawbacks. The most commonly cited problems of using the object-oriented paradigm seem to be the following:

- A commonly cited reason for not choosing the object-oriented paradigm is the need to fit more material into an already tight course schedule. Brilliant and Wiseman who did a survey on different programming paradigm in introductory programming courses got mixed feedback on this issue: “two responses indicating severe problems with covering all the material,
one indicating a less severe problem, and four indicating no problems [3]. “
Decker and Hirshfield [11] reported no problems with this.

• A common problem seems to find a good teaching language and environ-
ment. Kölling [27] states that “it is not object-orientation in principle
that causes the problems, but the tools available to teach it. Program-
mimg languages used are too complex and programming environments - if
they exist at all - are too confusing. Some systems used for teaching were
really developed for professional software engineers, making it difficult for
first-year students to cope; others were not “developed” at all but grew
out of historic coincidences.”

5.4 Summary

Evidently, all three programming paradigms have their assets and drawbacks.
However, it becomes increasingly clear that only two of them are adequate
for introductory teaching, namely the functional and the object-oriented. The
object-oriented paradigm has already gotten in the lead over the procedural
paradigm in many parts of computer science. It looks as if it will take over
the leadership as the teaching paradigm number one as well. This does not
mean that imperative programming should not be covered in computer science
education at all, but it should be done after one of the two other paradigms
was taught. DeClue [12] states that “In light of this possible incompatibility
[between the procedural and the object-oriented paradigm], some computer sci-
ence educators believe the continuation of current teaching practices to actually
be harmful to students. Reid (1993) asserts that delaying an introduction to
object-orientation may produce a mind-set toward programming that could per-
petuate students creating Pascal or C program organizations and merely using
the object-oriented language to write in old forms.”
Chapter 6

The Choice of a Programming Language and Environment

Indeed, many people who are not trained to be programmers would like to have this capability. However programming is well known to be a very difficult activity, especially for beginners. Some of this difficulty is intrinsic to programming, but some part of it can be relieved by careful attention to usability during the design of programming languages and tools.


One of the most important choices that must be made when designing an introductory programming course is the choice of an appropriate programming language and a programming environment. At first this decision seems only secondary, but when reading about the major problems that student face with these courses, lamentations about the language and environments that are used are the most common (for example see [27, 28]). McIver and Conway [33] state that “Learning to program is difficult. We believe that a substantial part of this difficulty arises from the structure, syntax and semantics of the programming languages which are commonly used to teach programming.” Kölling argues that “a well designed language is only half of what we need and is rendered useless by the absence of a suitable environment. With some languages, the environment was the source of the most serious problems[27]. “ If students are struggling with the environment and the language itself, how should they be able to acquire the needed programming and problem solving skills?

This part will describe requirements for teaching languages as well as for environments that are used for teaching.
6.1 Teaching languages

Programming languages that are used for teaching have to meet different requirements than programming languages that are used for the development of industrial software. “Languages are not good or bad per se; they are good or bad for a specific purpose. A language that is bad in one context can be excellent in another (or vice versa) [1].” Some characteristics of a programming language are very important for teaching purposes - others are dispensable. For example, a teaching language must emphasize a readable syntax and cleanly represent the concepts that are to be taught, but it must not be overly efficient or flexible. “Note that some issues, such as efficiency, which are often considered extremely important for production programming languages, are of little significance for a teaching language; it is only required that the language be able to be supported in a teaching environment with reasonable response time. Similarly, it is not important that the language be flexible enough to develop real-world applications (e.g. by the inclusion of operations such as arbitrary bit manipulation) - it is not intended to be used for this purpose [1].”

Following up there will be given a list of desired features that a teaching language should possess. Kölling describes in [27, 1] the characteristics that an object-oriented teaching language must conform to; the features that are listed below are motivated by his papers.

**Easy transition**

“An essential aspect of any teaching language is that what is learnt by using it must be relevant to what is needed later. [...] We, as teachers, have to ensure that students learn programming as opposed to a programming language. This means that we have to teach them the principles of programming. We might do this in any language we think appropriate. [...] Concepts learned with the first teaching language must be easily transferable to the next language following it [27].” This directly leads to the next property that a teaching language must exhibit, namely it must be based on clean concepts.

**Clean concepts**

“The concepts that we want to teach should be represented in the language in a clean, consistent and easy-to-understand way [1].” Clearly, what educators want to teach in an introductory programming course are techniques, principles and concepts that are not only valid in the particular language that has been chosen, but that also can be transposed to other languages. In order to achieve this goal, the teaching language must directly support these issues.

**Safety**

Errors that can easily be detected during compile-time or during runtime should automatically be detected, and error messages should give clearness about the cause of the errors. “There is a widely cherished belief amongst educators that one of the ways students learn best is by making their own mistakes. What is often neglected is that this mode of learning is only effective if a student’s otherwise random walk through the problem space can be guided by prompt, accurate, and comprehensible feedback on their errors [53].” Furthermore, errors
that can be omitted should be. In the case of an object-oriented language this means for example that “the language should be strongly and, as far as possible, statically typed. In dynamically typed languages the point of detection of the error might be a long way away (in time and location) from the actual source of the problem. This makes understanding and eliminating program errors an unnecessarily difficult task [27].”

**High level**

“The programmer should not need to be concerned about machine internals [27].” Since novice programmers already have to struggle with syntactical and semantical peculiarities of a programming language, they should not have to deal with hardware dependent or very low level issues. Examples of this are pointer arithmetic (which definitely has no part in an introductory programming course), bit manipulations, details of representational precision (e.g. `int` in C which has a 16 or 32-bit representation depending on the underlying hardware), and, probably most important, manual memory management.

**Simple execution model**

“The model of execution should be simple and easy to understand [1].” For example it should not be needed for the novice programmer to make decisions whether data is put on the stack or the heap. Furthermore, storage space for data (meaning objects in the case of object-oriented programming) should be allocated automatically by the system. Again, students should not have to deal with features of a language that exist purely because of efficiency reasons. If they can be automated, they should be.

**Readable and consistent syntax**

“The syntax used should be easily readable and consistent [1].” Some simple principles for doing this are:

- favor keywords over symbols
- favor familiar keywords over computer science buzzwords
- use the same syntax for the same semantics, different syntax for different semantics. This means eliminating syntactic synonyms\(^1\) as well as syntactic homonyms\(^2\).

“For both beginners and experienced programmers readability has clear advantages. It makes the learning of the language easier and helps to reduce the number of errors. [...] In an age in which we have recognized that programmers actually spend much more time reading programs text than writing them, and in which we agree that reading and understanding code may be more difficult...

\(^1\)An example for a syntactic synonym would be (in C++) `array[1]`, `*(array+1)`, `1[array]`, and `**array` which have the same semantic meaning (getting the second element of an array), but different ways of expressing it (see [33]). Note that some of these expressions are only legal in certain contexts.

\(^2\)An example for this can be found in Turing where `A(B)` has five distinct meanings (see [33]).
Small and orthogonal set of features

“The language should be as small as possible while including all important features that we want to discuss in the first year programming course [1].” In “Seven Deadly Sins of Introductory Programming Language Design” (see [33]) the first deadly sin is the “less is more” principle. They state that if the language is made up by too few concepts, it can lead to extremely obscure and unreadable code. The second deadly sin is to go into the other extreme by applying the “more is more” principle. In this case, the students are forced to learn many different language constructs which may as well obfuscate the students understanding of the underlying concepts, and often the educators only cover a subset of the language. Both, [33] and [27] regard teaching of a subset as harmful, mostly because reference materials usually are not confined to the taught subset. “Students use parts of the language that are not in the officially sanctioned subset (because they read examples from a textbook). The teacher then needs to explain why it should not be used, or accepts its use and is forced to deal with it [27].” In short: the language should neither be too small nor should it be too large.

6.2 Teaching environments

“In short: a suitable programming environment is crucial for the success of an introductory course. Of all the problems reported by educators connected to teaching object-orientation, problems with the environment used were the most frequent and the most severe [28].” The importance of a good programming environment depends on the programming paradigm that is used. “Earlier introductory courses focused on the development of algorithms in procedural or functional languages. To do this, an editor and a compiler was all that was needed for the practical part of the work [28].” But in the case of an object-oriented teaching language, where the transfer of the underlying object-oriented concepts are going to be the substantial goal, the environment has become a very important choice that has to be made. “Modern courses now use object-oriented languages and subject material taught includes testing, debugging and code reuse. This creates the need to deal with multiple source files and multiple program development tools from the very start [28].”

Because of the importance of the environment for teaching the object-oriented paradigm, it is justified to include the examination of its requirements into this survey. Kölling states that an object-oriented teaching environment must have the following seven properties:

Ease of use

“The environment must be easy enough to manage for inexperienced students to be usable for programming tasks after a very short time. This virtually implies that the environment must have a graphical user interface. [...] Ease of use also means the hiding of unnecessary detail [28].” Kölling states that most of the environments offer either too little or too much support for the inexperienced
programmer. The first category usually are environments that are a loose collection of text based tools, where the user is responsible for the interaction of these (for example the location of errors are indicated by line numbers and must be found by the user, or the files have to be handled by the user, and so on). Environments that belong to the second category usually are very sophisticated and have been developed for professional software engineers. There are many different operations that are available, but most of these are typically unneced for novice programmers. “What is needed is a system that is simple enough to be usable for beginners after a very short time, yet powerful enough to provide many of the tasks of software development easily or automatically [28].”

Integrated tools

Kölling states that the integration of tools into an environment has some benefits for novice programmers, e.g. a unified interface, smaller interfaces, increased productivity, and better functionality. Environments that consist of non-integrated tools are not suitable for beginners, but are very reasonable for experienced programmers.

Object support

The concept of objects must be directly reflected in the object-oriented programming environment. This means that for an object-oriented language an object-oriented environment must be used. Many so called object-oriented environments still are program-oriented. This means that to build an application there must be exactly one entry point defined. “The object-oriented paradigm is based on the idea that objects exist independently of each other, and that operations can be executed on them. Consequently, a user in a true object-oriented development environment should be able to interactively create objects of any available class, manipulate these objects and call their interface routines. The composition of objects at the user level should be possible. [...] In most current object-oriented environments, objects have to be wrapped in a non-object-oriented main function or script to create and invoke the first object or objects. [...] In short: the programmer must fall back to the procedural paradigm to start and test a program. Thinking in two mind-sets is required: one for thinking about the model of the application itself and one for thinking about environment interactions [28].”

Support for code reuse

“The facilitation of reuse of existing code is one of the main motivations for object-oriented programming [28].” Because the students should be encouraged to reuse code offered by libraries the environment must provide features that ease the finding of reusable code, for example a class browser.

Learning support

Kölling lists various techniques that an environment used by novice programmers should offer:
• Interaction and experimentation with objects, by for example stepping through the code and seeing variables change, etc.

• Visualization of classes and their relationships.

Group support

“Another characteristic of programming in the real world is the need to work in teams. [...] Ideally, we also want to teach our students about the techniques needed for teamwork. To do this, it is essential that the environment has some form of support for group work [28].”

Availability

As the last point, Kölling mentions that the environment should be affordable for university departments.

Concluding Kölling [28] states that

None of the requirements named above is really new. Each of them has been implemented in some existing system. The combination of these requirements, however, is what is really important in our context. In particular the combination of the requirement of ease-of-use with that for fairly sophisticated technical support may at first glance seem contradictory. We are asking for a powerful system that appears simple to the user. The degree to which this combination is achieved will be the determining factor in assessing the suitability of systems for first year teaching.

6.3 Summary

Programming languages that are used for teaching must meet different requirements than industrial programming languages. They must ease the transition to other languages by directly representing the concepts that are to be taught, they need to consist of a small, orthogonal set of language constructs, and syntactics must be at the same time readable, easy-to-understand, and consistent. Unnecessary complications of the language must be omitted (for example manual memory management).

In the case of a course where the object-oriented paradigm should be taught, special focus has to be put on the choice of a programming environment. The environment must be easy-to-use, and must directly represent the underlying object technology.

Deciding on a programming language usually is not only motivated by its suitability for teaching. Often, other issues like the familiarity of the educators with it and its popularity play a substantial role. It is, however, always valuable to analyze an already chosen language according to the listed characteristics because the gained knowledge about missing or undesired properties helps preparing the educators for problems that students might complain about.
Chapter 7

The Inverted Curriculum, Object-Orientation, and Eiffel

We should all be encouraged by the fact that the problems of teaching introductory programming from the object perspective are recognized, that a variety of strategies are being explored, and by the promise that the revolution will quickly produce a flood of new and better pedagogical aids.

William Mitchell, A paradigm shift to OOP has occurred... implementation to follow.

At ETH Zurich the introductory programming course is currently being re-designed. The new approach will follow the idea of the Inverted Curriculum (see [38, 37, 36, 35]) using Eiffel as the first programming language. With Eiffel and the Inverted Curriculum object-orientation will be taught right from the start. This chapter is dedicated to look at the chosen approach by critically checking the different features.

7.1 The Inverted Curriculum

Bertrand Meyer [38] coined the term “the Inverted Curriculum” which is based on the idea of an outside-in introduction of topics and the progressive opening of black boxes. Central to the Inverted Curriculum is the use of an extensive code framework, consisting of a specially designed library and standard Eiffel libraries. Furthermore, it emphasizes the inclusion of advanced object-oriented programming mechanisms and software engineering techniques into the curricular topics.

Rather than introducing curricular topics in a bottom-up or a top-down
7.1. THE INVERTED CURRICULUM

direction (see 4.2), the Inverted Curriculum teaches programming outside-in (see [37]). Students get equipped with an elaborate working application that reuses components from libraries. They start the course by running and exploring the application, and then extend it by adding simple code that calls features on predefined objects. This allows them to produce interesting results from the very beginning of the course. Later, creation of objects is introduced and classes are used through their abstract specifications, the contract form. Only after this, the internal implementation of the classes is gradually revealed. In this way, students first find themselves in the role of reuse customers and gradually become themselves producers of reusable components. Meyer describes in [38] his approach as follows:

Almost on day one of the course, the students will be able to produce impressive applications by reusing existing software. Their first assignment may involve writing just a few lines – enough to call a pre-built application, and produce striking results (devised by someone else!). Then they are invited to take the components making up the application and recombine them in different ways so as to produce variants of the application, or apply them to new uses.

This black-box use of pre-existing components is only the first step. As students progress, a process of progressive opening of black boxes will take place. The students are encouraged to start looking into the components themselves. The teacher may wish to specify the order in which the components are to be thus examined.

Although the Inverted Curriculum approach is not a top-down approach, it shows all the benefits that have been described for top-down teaching strategies (see 4.2.2). In conjunction with the use of a large code framework, the following elegances make the Inverted Curriculum an interesting and promising teaching approach:

- It puts emphasis on ‘programming-in-the-large’.
- The Inverted Curriculum facilitates to stress important software engineering principles and object-oriented techniques as opposed to syntactic, programming language dependent details. However, it does not neglect the teaching of low-level concepts and skills.
- Since topics are introduced outside-in, students encounter the “big” object-oriented constructs like classes and objects first. In this way, they get to know the low-level language constructs for what they are: tools that help you making your objects behave the way you want them to.
- The turn-around from ‘programming-in-the-small’ to ‘programming-in-the-large’ can be avoided.
- While being a consumer to libraries, good coding style and neat design will be appreciated by the students. This experience will show them how important these two skills are for writing reusable software.
- The use of object-oriented libraries hides at the time unneeded details from the students. This relates directly to the first disadvantage of using
code frameworks as described in 4.3.2. It will get the students used to the idea that they don’t have to know how everything has been implemented, and that they don’t have to reinvent the wheel for every new application they want to create.

- The use of libraries represents real-world behavior where reuse mostly occurs through the use of pre-existing libraries. It also addresses the common need to extend and refine existing libraries by inheriting from their classes.

### 7.2 The object-oriented paradigm

The use of the object-oriented paradigm to implement the Inverted Curriculum is of such an importance because the teaching approach makes heavy use of underlying object-oriented principles like reusability of software, information hiding, and clean structuring of software. The intuitive structuring process of object-oriented programming, meaning that software is divided into components (classes) that offer services to be used by other components, helps students that are not familiar with programming details to use the components of the provided library. Without this kind of natural structuring, students would not be able to cope with the black box flavor of the Inverted Curriculum.

Because the Inverted Curriculum is itself based on object-oriented principles that it wants to instill in students, it appears as a modern, prosperous state-of-the-art way of teaching object-orientation.

### 7.3 Eiffel as a teaching language

As explained in chapter 6, teaching languages face other requirements than programming languages that are used for industrial purposes. Since Eiffel is an object-oriented language the programming environment also needs careful consideration. In this section we will first look at Eiffel as a teaching language and then at the programming environment EiffelStudio.

#### 7.3.1 The language Eiffel

Of all the languages surveyed, Eiffel probably comes closest to fulfilling our language requirements. It is statically typed, supports object-oriented concepts in a clean way, avoids redundancy and has a clear, easily readable syntax. This combination makes it a better candidate than any other language.

Comments by teachers who have used Eiffel in university courses reflect this view. The reaction is generally positive as far as the language itself is concerned, with a list of minor criticisms [1].

Following up, the criticisms and lauds that Kölling mentions for Eiffel are going to be described and discussed:
7.3. EIFFEL AS A TEACHING LANGUAGE

Easy Transition

“Eiffel defines its constructs (small scale constructs such as loops as well as object-oriented concepts) in a clear way close to accepted mainstream views. The skills learnt with Eiffel should be easily transferable to many other languages [1].”

Clean concepts

In Kölling’s opinion, Eiffel represents most concepts in a clean way. The only source of criticism is delivered by the fact that in Eiffel two storage models are known (normal mode and expanded mode). He states that “this is unfortunate from a pedagogical point of view. Its only purpose is to increase runtime efficiency, with no justification at a conceptual level [1].” Since any extended type may as well be implemented as a reference type, this criticism is justified. It is not necessary to learn about value and reference semantics in the first programming course. On the other hand, many modern programming languages know both, value and reference semantics, and the difference must be taught somewhere in the curriculum. In this respect, covering it during the introductory programming course is not obsolete, but unfortunate.

What also needs to be mentioned at this point is the support for design by contract. The sophisticated assertion support in Eiffel is one of the features that students that have learnt Eiffel in their introductory programming course seem to value the most. “From a student point of view, the assertion mechanism is clearly the language feature which is most appreciated [39].”

Safety

“The system is mostly type safe. Eiffel is statically typed and most errors are detected at compile time. Some typing holes exist which introduce the possibility of fatal errors at runtime, but these cases are rare enough not to pose a serious problem for a first year teaching context [1].”

High Level

Since automatic garbage collection is available, it avoids one of the worst areas of low-level operations, namely non-automatic memory management. As Kölling states there exist some low-level operations but the programmers are not forced to deal with them and therefore the high level requirement is fulfilled by Eiffel.

Simple execution model

The first point of criticism that Kölling describes are again the two different storage models. As a second source of criticism he points out that Eiffel is a language that attempts to support a very large number of constructs which sometimes leads to overly complex structures. As an example he relates to the case that Normark describes in [39] where a redefined routine in a subclass needs to call the original routine in the superclass. In the version of Eiffel that was actual at the time of the paper written by Normark, to do so the programmer needed to use repeated inheritance, redefinition of the routine and renaming. In later versions of Eiffel, this complex, indirect approach to call a routine of the
superclass within the inherited routine can be achieved by using the Precursor keyword.

**Readable and consistent syntax**

“Eiffel’s syntax is based on the tradition of ALGOL and Pascal and makes extensive use of readable and meaningful keywords. Like other languages in this tradition, it is easily readable and enforces a very clear code structure. For a teaching language, this is an ideal style [1].”

**Small and orthogonal set of features**

Eiffel mostly avoids multiple constructs for the same concepts. But as Kölling states “it supports a large number of concepts. This makes Eiffel large, even though it has little redundancy [1].” Blue (see [30]), a language that was specially designed for teaching purposes, has a very restricted set of supported constructs. Comparing Blue to Eiffel it appears that Blue does not support the following constructs that are apparent in Eiffel: multiple inheritance, immediate (expanded) objects, multiple constructors, function overloading, user defined infix operators, and routine parameters. It is true, that Eiffel is a conceptually rich language, but in my opinion the size of it seems to be reasonable. Furthermore, in [30] a wish list of concepts that are not supported in Eiffel (and neither in Blue) are described. So totally opposed to what is stated in [1], there are teachers (and students) that would like even more concepts to be included in Eiffel than the ones that are already provided.

### 7.3.2 The programming environment EiffelStudio 5.3

“The most fundamental problems with Eiffel are not with the language itself, but with the libraries and the programming environment [1].” As already mentioned in the section about programming environments, in a course that teaches the object-oriented paradigm, a respectable importance lies on the suitability of the programming environment. If the programming environment is difficult to manage, use and understand, programming becomes a far more difficult task than it actually should be. Because of this, we will now look at the programming environment that will be used by the students attending the course. There will be two versions used by the students: The professional edition will be installed in the computer rooms at ETH. If the students want to work at their personal computers they will have to install the standard version of EiffelStudio that is available for free download from the internet. “The environment is needed to hide some of the complexity inherent in the implementation of object-oriented technology (such as the need to deal with multiple files) from the student. The Eiffel environment, on the other hand, adds complexity. Eiffel (at least as it is currently available) must therefore be considered unsuitable [1].” Following up, we will check if the Eiffel environment as it is now available (meaning summer 2003, four years after Kölling published his dissertation) still has the major flaws that Kölling and Normark criticize. This will be done according to the requirements stated by Kölling (see also 6.2).
Ease of use

EiffelStudio clearly is an environment for professional programmers (see figure 7.1). It offers a lot of functionality which for experienced users makes programming a comfortable and efficient task. For novice programmers, EiffelStudio could cause problems. Its interface is quite extensive and too large for beginners. One of the major problems probably is the large amount of different compilation modes (see figure 7.2). Especially for novice programmers, it is not immediately clear, which compilation mode to select for what task. Some students might not even understand what compiling really means, letting alone the meaning of incremental compiling and optimization. The second point that essentially adds to the complexity of EiffelStudio are its numerous viewing modes (see figure 7.3). Needless to say, that they are very useful, if their intention is understood. But since we are dealing with beginners, most of the views will be at least at the very beginning meaningless to these students. For example, the contract view will be confusing if design by contract has not yet been treated during the course. The flat form will make no sense to programmers that yet don’t understand how inheritance works, and viewing once features doesn’t appear to be needed by somebody who has no idea of this concept. These are just few examples of how features of an environment that are mostly valuable to expert programmers, may be a millstone around the neck of an unexperienced programmer.

Integrated tools

EiffelStudio has all needed tools integrated into it. The only point of criticism is that the diagram tool is not available in the Free edition.
Object support

EiffelStudio does not provide an interactive object tool where objects can be created and modified in an interactive way.

Support for code reuse

There is a class browser that can be used to find reusable classes. It therefore does support code reuse.

Learning support

EiffelStudio (only professional edition) comes with a diagram tool where class relationships are visualized. This tool is of great support for getting an overview of how classes fit into a hierarchy as well as showing client relationships.

Group support

Group work is not directly supported. The need for additional tools like a cvs system might arise.

Availability

Since EiffelStudio is available in a free edition this point is of no concern.

The major problems of using EiffelStudio in teaching introductory programming seems to be the availability of the large amount of functionalities. While
these undoubtedly offer a comfortable and effective way to program for experienced programmers, especially at the beginning, novice programmers will be slain by the plethora of buttons.

7.4 Summary

With the Inverted Curriculum topics are introduced outside-in leading to a progressive opening of black boxes. In this way, students start as consumers of reusable components and gradually learn to be producers. The Inverted Curriculum does not ignore the teaching of low-level programming constructs, but it puts special emphasis on software engineering concepts, in particular abstraction. This key concept is vital to the Inverted Curriculum: Abstraction is not only taught to the student, but the Inverted Curriculum itself relies on it.

The object-oriented programming paradigm supports the high requirements that the Inverted Curriculum entails regarding abstraction. In this respect, the use of sophisticated, well-coded libraries within the course are central to its success.

To teach introductory programming with Eiffel seems to be a good choice regarding the language itself. More problematic seems to be the use of EiffelStudio for novice programmers. It offers too much functionality for beginners, and will therefore probably cause problems for the students.

All in all, the Inverted Curriculum in conjunction with Eiffel seems to be a promising teaching approach. Experience will show whether the theoretical approach also works in practice.
Chapter 8

Conclusion

With the increasing spread of personal computers and the growing industrial influence of computing as well as the booming use of object-orientation, the design of introductory programming courses has become a challenge. The traditional procedural computing courses don’t comply any more to the expectations of both students and instructors, and new strategies to teach introductory programming are being developed everywhere.

Teaching introductory programming is a challenge. This is mainly due to three reasons:

- The computing community does not agree on what topics should be covered in the curriculum of an introductory programming course.
- There is no consensus on what teaching strategy should be used to teach programming to novices.
- The student body of an introductory programming course has become extremely diverse regarding their background knowledge and prior programming and computer experience.

Object-orientation, today one of the most popular programming paradigms in industry, has slowly found its way into the introductory programming courses of universities. Many programming instructors would like to teach object-oriented techniques in their first semester courses and wish to exploit the beauties and advantages of this paradigm. Yet, the teaching of object-oriented programming in a truly object-oriented way (not only as an extension to imperative programming) has shown to be far more difficult than could be assumed.

In respond to the need for a new way of teaching object-orientation to novice programmers, a teaching approach called Inverted Curriculum has come up. This teaching strategy tries to fully take advantage of object-oriented techniques and constructs, and additionally stresses key concepts of software engineering that are important to large scale projects. Abstraction – one of the key concepts of object-orientation – is central to the Inverted Curriculum. Abstraction is needed to form the black boxes that, with the outside-in topic introduction, progressively are opened up during the course. Coherently, the concept of abstraction is found in the supporting software as well. In combination with Eiffel as the first programming language, the Inverted Curriculum seems to be an auspicious and promising teaching strategy.
Part II

Redesign of “Introduction to Programming” at ETH Zurich
Chapter 9

A Wind of Change

A wind of change is blowing at the computer science department of ETH Zurich. Besides the redesign of the “Introduction to Programming” course that involves a change in the teaching approach and a change in the first programming language, the Department of Computer Science introduces the bachelor program, which changes much of the computer science study itself.

9.1 Change of teaching approach

With the Inverted Curriculum a new teaching approach is implemented at ETH Zurich. Up to last year, introductory programming had been taught using a traditional bottom-up approach (see 4.2.1). The Inverted Curriculum entails many changes to the “Introduction to Programming” course:

- It introduces object-oriented programming in a truly object-oriented way right from the start.
- It is based on and built around a specially designed, quite elaborate, large piece of software: the TRAFFIC software (see chapter 13).
- Additionally to teaching standard low-level concepts, it introduces advanced object-oriented mechanisms and software engineering techniques.
- Instead of introducing curricular topics in a traditional bottom-up way, the sequence of topic introduction is outside-in.
- Besides training standard programming and problem-solving skills, the course intends to emphasize architectural skills that are needed for designing and working on large-scale projects.

The realization of a new course always entails a great deal of work. For the Inverted Curriculum the following basic parts have been and are currently developed:

- Schedule of lectures (see 11).
- Lecture slides.
9.2. CHANGE OF PROGRAMMING LANGUAGE

- The TRAFFIC library around which the whole course is built (see 13.2).
- The text book *Touch of Class* (see 13.1).
- Home works that fit into the schedule of lectures, belong to the domain of the TRAFFIC software and can be inserted into the TRAFFIC software (see 12).

Besides developing the course material, deciding on the general setup of “Introduction to Programming” needed quite some effort. Particularly, since the discussion on the examination mode (see 10.2) was heavily influenced by the constraints that the new bachelor/master program (see 9.3) imposes, an overview of the ETH regulations was made. Furthermore, the schedule of lecture needed special care. With the introduction of new curricular topics some of the topics that were covered in former courses will be missing or only partly included. In order to be sure that the students’ abilities conform to the prerequisites that instructors of advanced courses expect from them, a comparison with former “Informatik1” courses (the equivalent to the “Introduction to Programming” course) was made.

### 9.2 Change of programming language

Traditionally, the first programming course for computer science majors at ETH Zurich has been taught using the language Oberon. Oberon is an object-oriented programming language and the latest descendant of the Pascal, Modula, and Algo family. It was developed at the ETH Zurich in the late eighties by the Professors Wirth and Gutknecht. Now, almost two decades later, the programming language for the introductory programming course will change from Oberon to Eiffel.

Eiffel, following the tradition of Oberon, cannot be called a mainstream language regarding its industrial influence. Both, Oberon and Eiffel are used in industry, but they can by no means be called “popular”, in the sense of programming languages that are in wide spread use. Choosing an “unpopular” language for the introductory programming course has the benefit that it is unlikely to be already known to the students. In this way, all students are confronted with a new programming language no matter whether they have programming experience or not. So Eiffel – as Oberon did – can act as an equalizer between already quite advanced programmers and newbies.

### 9.3 Change of computer science study

Synchronously with the introduction of Eiffel as the first programming language and the innovative new teaching approach, the Inverted Curriculum, the Computer Science Department at ETH Zurich has revised the overall structure of the computer science study. Starting in winter 2003/2004 all computer science students will participate in the bachelor/master program. The information for this description has been taken from [14, 13].

The bachelor program takes at least six semesters. After completion of the bachelor program, the degree “Bachelor of Science ETH in Computer Science” is
9.3. CHANGE OF COMPUTER SCIENCE STUDY

<table>
<thead>
<tr>
<th>First semester</th>
<th>Second semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to Programming (8)</td>
<td>Data Structures and Algorithms (7)</td>
</tr>
<tr>
<td>Logic (4)</td>
<td>Concepts in Physics (6)</td>
</tr>
<tr>
<td>Analysis I</td>
<td>Analysis II</td>
</tr>
<tr>
<td>Linear Algebra (7)</td>
<td>Discrete Mathematics (7)</td>
</tr>
<tr>
<td>Probability &amp; Statistics (5)</td>
<td>Digital Design (6)</td>
</tr>
</tbody>
</table>

Table 9.1: Courses of the basic year

given to the student. In order to complete the bachelor program a student must collect 180 credit points. The structure of the bachelor program is organized as follows:

**Basic Year**

During the basic year, ten mandatory courses (five per semester) are to be taken by the student (see table 9.1).

At the end of the semester students must collect testates for each of the courses that were taken. Testates for all ten courses are needed to be admitted to the qualifying exam in 9 subjects (Analysis I and II are tested in one exam). All 9 subjects must be tested during one examination session. There are two examination sessions per year, one during the fall semester break and one during the spring break.

A student qualifies for the second year by reaching an overall grade of at least 4.0 (grades in Switzerland range from 1.0 to 6.0 with 6.0 being the best grade). The overall grade is the weighted mean of all the grades from the exams where the weights can be taken from the table 9.1. The grade of the combined exam of Analysis I and II is weighted with 10. After passing the qualifying exam a student gets 60 credit points.

The testates are valid for two years implying that a student that starts in winter 2003/2004 needs to pass the qualifying exam until fall 2005. Each student has two chances to pass the qualifying exam. If he fails the first time, he has to retake all the exams during the second session (no matter how good the grades for the single subjects in the first run were).

**Second Year**

During the second year students will be offered a set of mandatory courses on basic subjects of computer science like Electronics, Information Theory, Introduction to Computational Science, Systems-level Programming, Theory of Computation and Algorithms, Operating Systems, Computer Architecture, Networks and Distributed Computing, Software Architecture, Formal Methods and Functional Programming, Introduction to Databases, Scientific Computation, Laboratory/Project in Software-Architecture. For each course that a student passes the according credit points are conceded. The student must collect between 53 and 61 points.
Third Year

During the third year core courses like Information Systems, Theory of Computation and Algorithms, Computational Science, Distributed Systems, Visual Computing are offered. The student must collect 24 credit points from core courses. Additionally, 6 credit points from lectures in the area of Humanities, Social and Political Sciences, and a not yet defined number of credit points from a focus-program (elected by the student) are required.
Chapter 10

Formal Issues

Implementing a new introductory programming course comes with a lot of decision-making. In the case of “Introduction to Programming”, there were two major parts that were discussed quite fiercely: the examination mode and the organization of the exercises. First, general information like number of students, failure rates, etc. are given since these statistical values influenced the discussions.

10.1 General information

Number of students

The number of students that each year start studying computer science at ETH Zurich has continually gone up until it reached its peak in 2001 (see figure 10.1). After that the number of students went down again. The reason for this break-in presumably is that in 2001 two graduating classes finished the “Kantonsschule” (pre-university college) because of a redesign of these schools. Figure 10.2 shows the percentage of female students between 1981 and 2002.

This year we are expecting between 250 and 275 students to start a computer science study and therefore to attend the “Introduction to Programming” course.

Repeating students

Since for Swiss students that graduated from the “Kantonsschule” (pre-university college) admission to the ETH Zurich is granted without any further entrance examination, the qualifying exam after the basic year takes charge of filtering out students that are not able to satisfy the high expectations and requirements of the ETH. This fact also is reflected by the high failure rates (between 40% and 60%) from the qualifying exams of the past years (see figure 10.3). Many of the students that failed the qualifying exam the first time, take again an attempt after one or two semesters. The fraction of repeating students that attend a qualifying exam the second time, can be found in figure 10.4. Notice that during the spring examination session there is a much higher percentage of repeating students than during the fall sessions.
10.1. GENERAL INFORMATION

Figure 10.1: Total number of first year computer science students.

Figure 10.2: Percentage of female first-semester students.

Figure 10.3: Failure rates at the qualifying exam.


10.2. Examination mode

As described in 9.3 the course “Introduction to Programming” will be held during the first semester of the computer science study and officially the final exam is to be held during the examination sessions. However, opinions diverged whether it makes sense to have exams during the semester. There were two models that have been discussed:

1. Stick to the ETH tradition: There is one exam during the examination session that covers the whole course. No grading is done during the semester.

2. Grading is handled similar to the way it is done at US universities: There is a mid-term exam, a final exam, and a graded project. Possibly, some home works are graded as well. Since according to ETH regulations an exam must be held during the examination session, the idea came up to give the students the choice between doing the traditional examination session exam and the US way of grading (exams during the semester). In order to “encourage” students to take the exams during the semester, the exam during the examination session would be assessed to be much more difficult to pass than the semester exams.
Both models have clear benefits and drawbacks. Especially if viewed in the context of ETH regulations that have to be followed, it is not easy to break with the traditional way of grading. Following up, most of the arguments for or against one or the other variant are listed.

**Self-assessment**

Traditionally at ETH, grading only happens based on the exams taken during the examination session. Since using the traditional model exercises are not graded, during the progression of the course students might miss and lack feedback on where they are standing. They might get a faint idea whether their work is good or bad based on the corrections of their exercises, however, they never get to know how they are doing compared to the other students. Furthermore, since the exercises are corrected by each assistant for his/her own group, the correction will vary from group to group respecting to preciseness, depth and carefulness.

Opposed to this, the US grading model attempts to give early feedback on how a student is doing. By doing so, the student comes to know whether he has to exert himself for keeping up or whether he is already doing enough. This clearly has the advantage that the insecurity about whether the course is doable for a student, vanishes earlier than with the traditional model.

**Motivation/Pressure**

With the mid-term and final exams plus the project of the US grading scheme, students are “forced” to be more motivated to study the subjects along with the course rather than only studying it during the summer holidays before the examination session. At the same time this method puts a lot more pressure on the students that are already fully occupied with many difficult topics and confronted with a new school and new way of learning. ETH Zurich is a university demanding a lot from their first semester students. Furthermore, we cannot forget about the diverse student body regarding prior programming experience. The US system might just make it impossible for programming newbies to handle the course.

Besides that, the choice of adequate testate regulations can also be used to bring up this kind of “forced” motivation. There is however, one limitation: testates cannot be made dependent on achievements but only on effort. This means that a testate regulation like “the student needs to hand in correct solutions for at least 60% of the exercises” is not allowed, because we are not to demand correctness from the students. However, a testate regulation like “the student has to hand in 60% of the exercises where a clear effort to solve the tasks can be seen” is perfectly alright. This limitation guarantees that students are able to make up for starting problems while still having a means of pushing the students to following the course right from the start.

**Holistic testing**

Another advantage of the US model is that more than one exam is used to build the grade. In the traditional ETH grading scheme, the grade that a student gets is solely based upon the outcome of the exam during the examination session.
The work that he has previously been doing within the home works during the semester is not influencing his grade. So, if a student gets caught on the wrong foot on the day of the exam during the examination session, the game is over. Whereas with the US model there is a chance to make up for a misconduct.

Besides that, the US model offers two different kinds of tests: First, the mid-term and final exams which are tests where the students solve problems and answer questions on paper without any computer support. Second, the project which is done as a homework on a computer using the programming environment and various other tools. In this way, besides pure problem-solving skills and time wise very limited questions, also interaction and handling of the computer and the programming tools, as well as the ability to create a running application and to solve quite extensive problems can be tested.

**Fair grading**

Another issue that needs to be considered, is the need to offer a fair grading process. Since “Hilfsassistenten” are not allowed to give grades all the grading work (in the US system the correction of the exams and of the project) is on the assistants. In order to keep grading as fair as possible, each assistant should be responsible for correcting a distinct part of the exam for the whole class (not just his exercise group), e.g. each assistant gets a specific task out of the exam which he has to correct for all the students. If done in another way, fairness cannot be granted since the style of correcting, time constraints, and expectations of what a student should know to get a specific grade vary from person to person. While this is a standard way of grading written exams, for the project it becomes difficult to realize. The question of how the correction work of the project exercise could be split up between several assistants is not that easy to answer. Getting one assistant to do it by himself is not reasonable regarding the large amount of students. Furthermore, there is no good way to verify that all the projects that students hand in are in fact programmed by themselves. To ensure that each student gets the grade that he deserves some kind of checking for copied code needs to be used. All these issues lead to an organizational overhead which has to be taken into account.

**Conformance**

Since at ETH the first year courses are all tested during the same examination session and must be collectively passed (see 9.3), it might be confusing and incomprehensible for students if “Introduction to Programming” is the only course that does not follow the traditional ETH scheme of grading. Furthermore, it might be conceived as quite stunning if “Introduction to Programming” falls out of the norm, whereas “Algorithms and Data Structures”, the second computer science course during the basic year, sticks to the ETH regulations. Even more, since we so to speak have to provide for both, the exams during the semester and the exam during the examination session, it starts getting complicated. There are many questions that need to be determined precisely, e.g. if a student takes both the semester exams and the examination session exam, what will his grade be? What happens to student that fail the semester exam; will they start the examination session exam with some sort of a debt? Or will the semester exams be some kind of a credit if they were done well?
10.2. EXAMINATION MODE

Besides all that, educators of the other courses (see the list in 9.1) might get the impression that “Introduction to Programming” attempts to catch the total attention of the students, implying that it is more important than any of the other courses. This clearly is not intended by the instructors of the “Introduction to Programming” course.

Both of these arguments for the traditional ETH grading scheme are important to note, yet they cannot be taken as imperative reasons for dismissing the newly proposed grading model based on US standards.

Repetition or Deferment

With the adapted US grading scheme, there are two issues that have to be taken into account: repetition of exams and deferment. As can be seen from the figures in 10.1 repetition and deferment of the exams are both quite common for first-semester students.

- Repetition: Testates are valid for two years. This means that repeating students are not forced to attend the courses a second time, but just to retake the exams. Since the overall grade for the basic year is constructed by the mean of the grades for all ten courses, a student that has to retake the exams, might even have done good at the exam for “Introduction to Programming”.

- Deferment: Students are allowed to defer their exams as many times as they want, as long as their testates are valid.

Regarding these two facts, it boils down to two questions:

1. Are the semester grades valid for both trials that the student gets to pass the qualifying exam?

2. Are the semester grades still valid in successive examination sessions after the first fall examination session if the exam has not been taken yet?

The first question probably has to be answered with a ‘no’, since according to ETH rules repeating students must retake the exams for ALL the first year courses. To the second question the answer probably should be ‘yes’ since students might have to defer their exams because of some not influenceable circumstance like illness, accidents, etc. Assuming these two answers, a scenario can be found that makes it impossible for a student to choose the semester exams: A student postpones his first try at the qualifying exam for one semester. During the spring examination session he fails it. The second try will be during the fall examination session one semester later. Because during the summer semester “Introduction to Programming” is not held, it is not possible for him to take the semester exams, but only the much harder exam during the examination session.

The major problem with the US grading system is that repeating students and students that have to defer their exams are disadvantaged. In combination with ETH regulations the US grading system leads to unfair handling of these students. Since at ETH both repetition and deferment of the exams are actually happening quite commonly, it is not possible to decide on a case-to-case basis. In this respect, the proposed US version of grading does not seem to be a fair
For the course “Introduction to Programming” at ETH Zurich in winter 2003/2004 the decision that the traditional ETH model is used has been made. Since we are already experimenting with a new teaching approach, a new programming language and some new curricular topics, we believe that we should keep the traditional ETH grading scheme for the first run of the course. This is, however, not the final decision. It has been postponed to the second run of the course, when first experiences with the Inverted Curriculum have been made.

10.3 Organization of the home works

The home works are an integral part of the course. They are responsible for enforcing taught concepts, as well as showing practical appliances of these. The exercise sessions, should offer the students the opportunity to get help with their home works and ask questions about the topics introduced during the lectures. Furthermore, new topics should be reinforced by showing adequate examples. The frequency of home works was subject to a lively debate. Two variants were taken into account:

1. Weekly home works.
2. Home works that stretch over several weeks.

Both variants have their benefits and drawbacks.

Early feedback

Weekly home works have the advantage that students get early feedback on the correctness of their home works. With larger home works stretching over several weeks it can happen that students spend days following a wrong track. At the same time, weekly home works offer the possibility to directly reinforce what has been taught during the lectures, and forces students to find out whether they understand the current subject. In this way, open questions and issues that were not yet understood, can be clarified much earlier than with home works that stretch over several weeks.

Success/Failure

If there are many small home works as opposed to a few larger home works, not succeeding to solve one of the one-week home works is not as depressing as failing to do one of the few large home works. Especially for unexperienced programmers, it is important to first present them with home works that they can achieve to solve. In this way, they also experience more often a feeling of success, than with large home works.

Organizational overhead

Besides that, small and regular home works are easier to handle for both students and assistants. It is much easier to estimate the amount of time that needs to
be spent solving a small exercise than doing an exercise stretching over several weeks. For the educators, it makes it easier to design home works that are solvable for students. For the students, some of which are already struggling with all the novelties that university holds, it makes it easier to organize their weekly schedule if they know that every week a home work for “Introduction to Programming” is due.

**Problem-solving and software engineering skills**

The main advantage of home works that stretch over several weeks is that they reinforce software engineering concepts (like specification of a problem) and train problem-solving skills and organizational skills that are needed and demanded later. When approaching a larger problem rather than a small one, a large amount of time and thinking is required to come up with a “plan of attack”, dividing the large problem into smaller subproblems. This is exactly what we want our students to learn, since it is what they will benefit from to the greatest extent. Weekly home works do not offer the possibility to train this kind of problem-solving.

For the course “Introduction to Programming” we decided to do weekly home works for the most part of the course. But since the Inverted Curriculum specifically wants to enforce software engineering concepts and attempts to emphasize the skills needed to master the solution of large-scale projects and problems we will add a so-called project exercise which stretches over several weeks at the end of the semester. Besides that, there would the possibility to do weekly home works where each of the small home works contributes to the solution of a larger one. In this way, the advantages of weekly home works can be combined with the advantage of the several-weeks home works.
Chapter 11

Schedule of lectures

The development of a schedule of lectures belongs to one of the critical points in the development of an introductory programming course. Following up, the current version of the schedule of lectures including short descriptions of the home works is given. Minor changes are still possible.

The course can be divided into three parts

- Weeks 1 to 6: Basics.
- Weeks 7 to 11: Advanced object-oriented topics, data structures and algorithmic practices.
- Weeks 12 to 14: Introduction to software engineering techniques.

The basics part of the course does not differ much from the first couple of weeks of other object-oriented introductory programming courses regarding the topics are covered. However, it does differ respecting to the order of how the topics are introduced. Conforming to the Inverted Curriculum (see section 7.1) the issues are tackled from outside-in, starting with the lectures on dealing with objects, going on to the interface of classes, creation of objects, references and assignment, and ending with control structures.

The second part of the course covers more advanced topics like fundamental data structures, inheritance, genericity, and recursion. Especially interesting is the hardware and software intermezzo in week 8 where the students get presented with information on how computers are working and which software tools are needed to produce applications. This intermezzo reflects the strategy of the Inverted Curriculum: gradually opening up black boxes. During the weeks before, the students used the computer, they used a compiler and programming environment, they worked on an operating system, but they did all this without knowing what all these items exactly are. During week 8 all these black boxes are opened up and the students are presented with a thorough understanding of what they did all along the way.

The third part of the course touches important software engineering concepts. While during the first two parts of the course, the students are asked to do weekly home works, the homework for this third part is a project which stretches over several weeks. This is especially suitable for enforcing software engineering techniques like algorithmic design, requirements engineering, and
testing and quality assurance. In this way, students will encounter some of the problems that are typical for the development of large scale software project. Besides that, the project exercise will give the students the opportunity and freedom to apply everything they learned during the whole course.

<table>
<thead>
<tr>
<th>Week</th>
<th>Lectures</th>
<th>Exercise session</th>
<th>Homework</th>
</tr>
</thead>
</table>
| 1    | Lecture 1.1: No lecture  
      Lecture 1.2: General intro, challenges of computing, organizational information | - Handout and prepare homework 1 | Homework 1: Getting started with the computers at ETH (login, mail, internet, printing) |
| 2    | Lecture 2.1: Dealing with objects  
      Lecture 2.2: Dealing with objects - Submission of homework 1 | - Demonstration: Getting started with EiffelStudio, installing the TRAFFIC library  
      - Handout and prepare homework 2  
      - Return and review homework 1 | Homework 2: Using the TRAFFIC library to build animations (e.g. city tour with certain constraints, objects are given, only features need to be called), stepwise execution of a program. |
| 3    | Lecture 3.1: The interface of a class  
      Lecture 3.2: The interface of a class - Submission of homework 2 | - Handout and prepare homework 3 | Homework 3: Use a class interface to find out which features need to be called to perform a certain task. |
| 4    | Lecture 4.1: Logic  
      Lecture 4.2: Creating objects - Submission of homework 3 | - Handout and prepare homework 4  
      - Return and review homework 2 | Homework 4: Contract an uncontracted class using the learnt logical constructs. |
| 5    | Lecture 5.1: Creating objects  
      Lecture 5.2: References and assignments - Submission of homework 4 | - Handout and prepare homework 5  
      - Return and review homework 3 | Homework 5: Copy, move and duplicate icons, (show difference between value and reference semantics). |
| 6    | Lecture 6.1: Control structures  
      Lecture 6.2: Control structures and algorithmic reasoning - Submission of homework 5 | - Handout and prepare homework 6  
      - Return and review homework 4  
      - Presence exercise on sorting (divide and conquer strategies, bubblesort, heapsort, quicksort) | Homework 6: Using nested loops e.g. by iterating over stations and calculating distances. Using conditionals for filtering e.g. stations. |
| 7    | Lecture 7.1: Fundamental data structures: arrays, lists  
      Lecture 7.2: Fundamental data structures: arrays, lists - Submission of homework 6 | - Handout and prepare homework 7  
      - Return and review homework 5  
      - Presence exercise on EBNF and syntax diagrams (e.g. show the EBNF of the Eiffel language constructs seen so far). | Homework 7: Implementing different kinds of arrays and lists. |
| 8    | Lecture 8.1: The hardware setup: computers, memories, processors  
      Lecture 8.2: The software setup: operating systems, compilers, tools - Submission of homework 7 | - Handout and prepare homework 8  
      - Return and review homework 6  
      - Presence exercise on scanners and parsers (for any simple grammar given in EBNF). | Homework 8: Test and debug an existing system (try to find bugs like pre- and post-condition violations, assignment errors, etc.). |
One of the questions that had to be tackled during the development of the schedule was whether the students who attend the course will have encountered and understood all the topics that are expected from them later on. To do so, the above schedule holding the covered topics for “Introduction to Programming” and former Informatik1 courses (the equivalent to “Introduction to Programming”) were compared. The course Informatik2 (now called “Algorithms and Data structures”) was also taken into account to see which topics have to be covered in depth, which ones can be left out and which ones can be touched only superficially. The following table shows the four courses in comparison.

The topics are ordered (more or less) in the way that they will be presented during the “Introduction to Programming” course of next winter.

L: Topic is covered during the lecture
E: Topic is covered in an exercise
-: Topic cannot or does not need to be covered
?: Not clear yet, exactly which of these subtopics will be covered

---

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Submission of homework 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Christmas break**

<table>
<thead>
<tr>
<th>Lecture 10.1: Summary (Recapitulation of things seen so far)</th>
<th>Lecture 10.2: Genericity</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Handout and prepare homework 10</td>
<td>- Return and review homework 8</td>
</tr>
</tbody>
</table>

**Homework 10**: Build a small library using inheritance.

<table>
<thead>
<tr>
<th>Lecture 11.1: Recursion and trees</th>
<th>Lecture 11.2: Recursion and trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Submission of homework 10</td>
<td>- Handout and prepare project homework</td>
</tr>
<tr>
<td></td>
<td>- Presence exercise on backtracking and dynamic programming (four/eight queens, subset sum, knapsack problem)</td>
</tr>
</tbody>
</table>

**Project homework**: Building a larger application that needs all the covered topics, including recursive algorithms (e.g., calculating shortest paths).

<table>
<thead>
<tr>
<th>Lecture 12.1: Algorithm design with an example</th>
<th>Lecture 12.2: The software process &amp; Requirements engineering and documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Return and review homework 10</td>
<td>- Work on project homework</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lecture 13.1: Testing and quality assurance</th>
<th>Lecture 13.2: Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Submission of project homework</td>
<td>- Preparation for exam: presence exercise that tries to show the kind of questions that will be asked during the exam.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lecture 14.1: Event-driven design</th>
<th>Lecture 14.2: Summary with outlook (e.g., example algorithm implemented with different kinds of programming paradigms and languages)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Return and review project homework</td>
<td></td>
</tr>
</tbody>
</table>

---

This table has been made on the basis of the course comparison that Till Bay and I did to discuss and adapt the schedule with the assistants of Prof. Meyer. Thank you, Till!
Not clear yet whether it should be covered, since it is covered by most former courses, but also by Informatik2.

<table>
<thead>
<tr>
<th>Topics</th>
<th>Intro2Prog (B. Meyer), 2003</th>
<th>Info1 (P. Widmayer), 1998</th>
<th>Info1 (J. Gutknecht), 2002</th>
<th>Info2 (P. Widmayer), 2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall setup (computers, memories, information, data)</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>-</td>
</tr>
<tr>
<td>Notion of Object</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>-</td>
</tr>
<tr>
<td>Variable</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>-</td>
</tr>
<tr>
<td>VAR parameter</td>
<td>-</td>
<td>L</td>
<td>L, E</td>
<td>-</td>
</tr>
<tr>
<td>Scope</td>
<td>-</td>
<td>L</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Feature calls</td>
<td>L, E</td>
<td>L</td>
<td>L, E</td>
<td>-</td>
</tr>
<tr>
<td>Queries, commands (procedures, routines)</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>-</td>
</tr>
<tr>
<td>Notion of Class</td>
<td>L</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Instructions and expressions</td>
<td>L</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Programming language: syntax and semantics</td>
<td>L</td>
<td>L</td>
<td>L, E</td>
<td>-</td>
</tr>
<tr>
<td>Overview of programming paradigms</td>
<td>-</td>
<td>L</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Abstract syntax tree</td>
<td>L</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>EBNF</td>
<td>L, E</td>
<td>L</td>
<td>L, E</td>
<td>L, E</td>
</tr>
<tr>
<td>Syntax diagrams</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>-</td>
</tr>
<tr>
<td>Class interface (queries, commands)</td>
<td>L, E</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Design by Contract</td>
<td>L, E</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Preconditions, postconditions, invariants</td>
<td>L, E</td>
<td>L</td>
<td>E, L</td>
<td>-</td>
</tr>
<tr>
<td>Logic</td>
<td>L, E</td>
<td>E</td>
<td>L</td>
<td>-</td>
</tr>
<tr>
<td>Void references</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>-</td>
</tr>
<tr>
<td>Creation instruction (default creation)</td>
<td>L, E</td>
<td>L</td>
<td>L</td>
<td>-</td>
</tr>
<tr>
<td>Creation procedures</td>
<td>L, E</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Root class</td>
<td>L</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Control structures (sequence, conditional, loop)</td>
<td>L, E</td>
<td>L</td>
<td>L, E</td>
<td>-</td>
</tr>
<tr>
<td>Notion of algorithm</td>
<td>-</td>
<td>L</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Run-time efficiency</td>
<td>!</td>
<td>L</td>
<td>L, E</td>
<td>L, E</td>
</tr>
<tr>
<td>Modular Design</td>
<td>-</td>
<td>L</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Assignment</td>
<td>L, E</td>
<td>L</td>
<td>L</td>
<td>-</td>
</tr>
<tr>
<td>References</td>
<td>L, E</td>
<td>L</td>
<td>L</td>
<td>-</td>
</tr>
<tr>
<td>Value and reference semantics</td>
<td>E</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Records</td>
<td>-</td>
<td>L</td>
<td>L, E</td>
<td>-</td>
</tr>
<tr>
<td>Basic data types</td>
<td>-</td>
<td>L</td>
<td>L</td>
<td>-</td>
</tr>
<tr>
<td>Type declaration</td>
<td>-</td>
<td>L</td>
<td>L, E</td>
<td>-</td>
</tr>
<tr>
<td>Type extension</td>
<td>-</td>
<td>L</td>
<td>L, E</td>
<td>-</td>
</tr>
<tr>
<td>Type compatibility</td>
<td>-</td>
<td>L</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Arrays</td>
<td>L, E</td>
<td>L</td>
<td>E, L</td>
<td>L, E</td>
</tr>
<tr>
<td>Topics</td>
<td>Intro2Prog (B. Meyer), 2003</td>
<td>Info1 (P. Widmayer), 1998</td>
<td>Info1 (J. Gulteuchter), 2002</td>
<td>Info2 (P. Widmayer), 2002</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>----------------------------</td>
<td>---------------------------</td>
<td>----------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>Lists</td>
<td>L, E</td>
<td>L</td>
<td>L, E</td>
<td>L</td>
</tr>
<tr>
<td>Lists: linear linked lists</td>
<td>?</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>Lists: double linked lists</td>
<td>?</td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lists: Ordered lists with sentinel</td>
<td>?</td>
<td>L</td>
<td>L, E</td>
<td></td>
</tr>
<tr>
<td>Lists: FIFO queues</td>
<td>?</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lists: LIFO queues</td>
<td>?</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware setup (computers, memories, processors)</td>
<td>L</td>
<td>L</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Software setup (operating system, compilers, tools)</td>
<td>L, E</td>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Object-orientation</td>
<td>L, E</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inheritance</td>
<td>L, E</td>
<td>L</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Dynamic binding</td>
<td>L</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deferred classes</td>
<td>L</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Abstract Data Type</td>
<td>L</td>
<td>L, E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Genericity</td>
<td>L</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Recursion</td>
<td>L, E</td>
<td>L, E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>Backtracking</td>
<td>E</td>
<td>L, E</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>Dynamic programming</td>
<td>E</td>
<td>L, E</td>
<td>L, E</td>
<td></td>
</tr>
<tr>
<td>Trees</td>
<td>L</td>
<td>L, E</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>Trees: Binary trees</td>
<td>?</td>
<td>L, E</td>
<td>L, E</td>
<td></td>
</tr>
<tr>
<td>Trees: Search trees</td>
<td>?</td>
<td>L, E</td>
<td>L, L</td>
<td></td>
</tr>
<tr>
<td>Trees: Self balancing trees</td>
<td>?</td>
<td>L, L</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Modeling of discrete systems</td>
<td></td>
<td></td>
<td>L, E</td>
<td>-</td>
</tr>
<tr>
<td>Search: Binary search</td>
<td>!</td>
<td>L, E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>Search: Linear search</td>
<td></td>
<td></td>
<td>L, E</td>
<td></td>
</tr>
<tr>
<td>Hashing</td>
<td>L</td>
<td>L, E</td>
<td>E</td>
<td>L, E</td>
</tr>
<tr>
<td>Sorting</td>
<td>E</td>
<td>L, E</td>
<td>L, E</td>
<td>L, E</td>
</tr>
<tr>
<td>Sorting: Insertionsort</td>
<td>?</td>
<td>L, L</td>
<td>L, E</td>
<td></td>
</tr>
<tr>
<td>Sorting: Quicksort</td>
<td>?</td>
<td>L, E</td>
<td>L, E</td>
<td></td>
</tr>
<tr>
<td>Sorting: Heapsort</td>
<td>?</td>
<td>L, L</td>
<td>L, E</td>
<td></td>
</tr>
<tr>
<td>Sorting: Bubblesort</td>
<td>?</td>
<td>L, L</td>
<td>L, E</td>
<td></td>
</tr>
<tr>
<td>Divide and conquer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graphs:</td>
<td>!</td>
<td>L, E</td>
<td>L, E</td>
<td>L, E</td>
</tr>
<tr>
<td>Graphs: reflexive, transitive hull</td>
<td>?</td>
<td>L, L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graphs: Directed graphs</td>
<td>?</td>
<td>L, E</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Graphs: Undirected graphs</td>
<td>?</td>
<td>L, L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graphs: Minimal spanning tree</td>
<td>?</td>
<td>L, E</td>
<td>L, E</td>
<td></td>
</tr>
<tr>
<td>Graphs: Graph traversal</td>
<td>?</td>
<td>L, E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Topics</td>
<td>Intro2Prog (B. Meyer), 2003</td>
<td>Info1 (P. Widmayer), 1998</td>
<td>Info1 (J. Gutknecht), 2002</td>
<td>Info2 (P. Widmayer), 2002</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-----------------------------</td>
<td>---------------------------</td>
<td>-----------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Graphs: shortest path</td>
<td>?</td>
<td>L</td>
<td>L, E</td>
<td></td>
</tr>
<tr>
<td>Topological sort</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>Random numbers</td>
<td></td>
<td>L</td>
<td>L, E</td>
<td></td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td></td>
<td>L</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>Scanner, parser</td>
<td>E</td>
<td>L, E</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>File handling</td>
<td></td>
<td>L</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Input/Output</td>
<td></td>
<td>E</td>
<td>L, E</td>
<td>-</td>
</tr>
<tr>
<td>Messages and message handles</td>
<td>-</td>
<td>L</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Software process</td>
<td>L</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Software development model</td>
<td></td>
<td>L</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Requirements engineering</td>
<td></td>
<td>L</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
<td>L</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
<td>L</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Quality assurance</td>
<td></td>
<td>L</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Reuse</td>
<td></td>
<td>L</td>
<td>L</td>
<td>-</td>
</tr>
<tr>
<td>Event-driven design</td>
<td></td>
<td>L</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Chapter 12

Exercises

12.1 Guidelines for the exercises

Designing meaningful and adequate exercises is a challenging task. Exercises are crucial to the learning process of the students because they require students to apply their knowledge in practice. Besides that, exercises help students to find out whether they really understand what they believe to have learnt. Because of the importance of exercises, special attention needs to be paid to the design of them.

In [4, 5] different processing habits of students that lead to poor learning tendencies are described. For each of them a list of suggestions to improve tasks in respect to the according processing habit are given. Following up, the outcome of this research is being summarized.

Superficial attention: Superficial attention describes the processing habit where students make no attempt to actively process the given information. In this way, personal meaning is not generated. This happens if tasks are too complex or require copying or modifying given code. The list of suggested improvements for tasks involves

- **Not always coding.** Instead of always letting student write lines of code, alternatives like tracing code, answering a series of questions, or diagrammatically presenting material may be well-suited to highlight misconceptions and convey concepts.

- **Rewards for understanding not completing.** Emphasize that understanding is rewarded not copious amount of code.

- **Outline a method of attack.** Instead of letting students tackle exercises without any guidance, offer them a plan of attack which they can follow.

- **Smaller coding questions.** The single tasks should not consume too much of the students time. Rather introduce many small tasks, instead of large ones.

Impulsive attention: Tasks that do not sufficiently emphasize the key concepts or that contain to many unfamiliar concepts may lead to the
behavior that the learner focuses on single, to him interesting parts and ignores other sections or major points. The following principles are meant to prevent impulsive attention:

- **Emphasize the key point.** This can be done by either giving students preparatory questions that shows the students the key components of the task or by specifically emphasizing the key concepts in the task itself.

- **Provide adequate resources for the introduction of unfamiliar material.** If unfamiliar concepts are part of the exercises, students should be provided with good material in order to help them understand them.

**Staying stuck:** Staying stuck happens because of the lack of any strategy to cope with getting stuck except for calling for help. There are three points where students can get stuck. Either the students don’t know how to start tackling a task, or they don’t know how to design a solution that comes in manageable components, or they get stuck at the debugging level after a full solution has been coded. To minimize staying stuck the following principles for designing tasks have been mentioned:

- **Tactics on how to start with graded help.** Don’t give away the whole solution when being asked, but only give hints. Challenge the student first.

- **Provide guidelines to writing and testing code in manageable chunks.** Include material on debugging techniques, and provide tasks that lets the student build programs in stages.

**Non-retrieval:** Tasks that introduce new concepts, or reuse concepts that were never really understood can lead to non-retrieval. In many cases, students are not able to connect earlier material to newly introduced concepts. The principles to minimize non-retrieval include:

- **Familiarity.** If concepts are reused, a referral should be made to earlier work.

- **Reinforcement by repetition.** Tasks should always include material that reinforces concepts that were covered earlier.

- **Retrieve existing understanding.** Students should be given the opportunity to use existing understanding in their tasks as well as in discussions.

**Lack of internal reflective thinking:** The learner is not aware of how single lessons, activities or tasks within the subject are linked together. He does not see the big picture. In many cases, exercises are designed at the application level rather than the analysis level of Bloom’s taxonomy. This leads to the lack of internal reflective thinking, since patterns, and the organization of parts into the overall context are not emphasized. Suggestions to encourage internal reflective thinking are:
12.2. FITTING EXERCISES INTO THE TRAFFIC APPLICATION

- **Tie the work into the “Big Ideas” of the course.** “At the start of each task specify the aims of the task and provide an explanation why a particular topic is being taught [5].”

- **Build on previous work.** Previous work should be included into tasks in order to help the students to see the context.

- **Extract the links.** “Write tasks that help the learner identify the links between each lesson and activity rather than approach topics in isolation from each other [5].”

**Lack of external reflective thinking:** The learner is not aware of how the contents of one subject are linked to other subjects or the outside world. In many cases, instructors give analogies or real life situations to create the missing links. The principles to encourage external reflective thinking include:

- **Integrate external knowledge into tasks.** Include knowledge or skills from different subjects into the tasks in order to help students utilizing all the knowledge they have.

- **Introduce new concepts with respect to external knowledge.** By justifying the introduction of new concepts in respect to external needs or problems that can only be solved with the new concept, external reflective thinking is emphasized.

There are two more guidelines that I would like to add to this list:

- **An exercise should cover as many levels of Bloom’s taxonomy as possible.** Undoubtedly, not all levels can be covered, but since each successive level of Bloom’s taxonomy builds on the mastery of the lower levels, students can be gradually led from the one of the lower levels (knowledge or comprehension) to one of the higher levels. It is obvious, that not all concepts that are reinforced through the home works offer material for all levels, but at least we should try to offer a wide variety of tasks.

- **Integrate tasks that show where common problems appear.** For example, in a homework about loops at least one task should be included that makes the student actively experience that with loops the most common problems are with the border cases (first or last loop iteration) and that the terminating condition must be chosen carefully.

### 12.2 Fitting exercises into the TRAFFIC application

Since students will be working with the TRAFFIC application from the first lesson on, we needed to assure that there exists an easy way to integrate their solved tasks into the application. The TRAFFIC application displays a list of buttons on the right which at the beginning of the course don’t offer any functionality (see figure 13.2). The students will fill in the functionality of these buttons as home works during the course. Since we want to offer them an comfortable way to do so right from the start of the course, we need to provide
12.2. FITTING EXERCISES INTO THE TRAFFIC APPLICATION

indexing

description: "Container for collecting exercise objects"

class

EXERCISE_REGISTRATION

inherit

EXERCISE_CONTAINER

create

make

feature {NONE} -- Implementation

set_up is

-- Add the exercise objects here.

do

put (create {LOOP_EXERCISE }, "Loop")

-- Add more exercises in an analogous manner.

end

end -- class EXERCISE_REGISTRATION

Algorithm 3: Class EXERCISE_REGISTRATION

them with an easy-to-use interface. There are two main classes that they will encounter from the very beginning:

- EXERCISE_REGISTRATION: the class that the exercises need to register at, telling the application the name that is to be displayed on the button and which descendant of class EXERCISE is to constitute the actions that will be taken when pressing the according button.

- EXERCISE: the ancestor of the classes that are used to implement the actions. It has a feature called “run”, that needs to be effected in descendant classes and which will define the actions that are taken when the button is pressed.

Since these will be the classes that the students encounter first, they need to be small offering only the most needed functionality. For the class texts of class EXERCISE_REGISTRATION and EXERCISE see listing 3 respectively listing 4. Additionally class LOOP_EXERCISE (see listing 5) shows how a class implementing the solution to a task on loops could look like.
12.2. FITTING EXERCISES INTO THE TRAFFIC APPLICATION

indexing

description: "Exercises for students"

defered class

EXERCISE

inherit

SHAREDMETHOD_WINDOW
SHAREDMETHOD

feature -- Basic operations

run is

-- Run the exercise code.

defered
end

console_output (x: ANY) is

-- Print to the console.

do

if x /= void then

message_window.put (x.out)

else

message_window.put ("void")

end

end -- class EXERCISE

Algorithm 4: Class EXERCISE

71
12.2. FITTING EXERCISES INTO THE TRAFFIC APPLICATION

indexing

description: "Loop exercise: output locations plus according links"

class LOOP_EXERCISE

inherit EXERCISE

feature

run is

-- outputs for each location the according roads and metro_lines

local

index: INTEGER

do from

map.city.locations.start
until

map.city.locations.off
loop

console_output (map.city.locations.item_for_iteration.description)
from

map.city.routes.start
until

map.city.routes.off
loop

from

index := 0
until

index = map.city.routes.item_for_iteration.elements
loop

index := index + 1
if map.city.routes.item_for_iteration.element (index).name

= map.city.locations.item_for_iteration.name then

console_output (map.city.routes.item_for_iteration.name)
end
end

map.city.routes.forth
end
map.city.locations.forth
end
end

end -- class LOOP_EXERCISE

Algorithm 5: Class LOOP_EXERCISE.
Chapter 13

Course material

13.1  *Touch of Class* text book

Contemporaneously with the development of the course, a text book *Touch of Class - Learning to program well with Object Technology, Design by Contract and steps to Software Engineering* is being written by Professor Meyer. The text book is very extensive and is meant as an introduction to most of the basic programming concepts including algorithms and data structures, event-driven programming, and concurrent programming. *Touch of Class* is divided into five parts (see listing 6).

Part I covers the basics. It starts out with an introductory chapter with a description of the overall setup explaining the relationship and interplay between information, data, computers, software and hardware. In the second chapter predefined objects are used by calling their features without revealing implementation details. Following up, the notion of class and class interface are described. This chapter also covers contracts and therefore shows the student an abstract, precise way of describing a class’ functionality. The subsequent chapter on logic gives the student the knowledge to fully understand, use and produce contracts. Back to programming, creation of objects is covered, followed by control structures, assignment and references. Next, fundamental data structures like arrays and lists are introduced. And as a last chapter of this part input, output and exceptions are covered. At this point, the student has all the knowledge to read and write basic program code.

Part II is entitled “How things work” and is dedicated to the internal perspective. The first chapter of this part is called “a glimpse into the hardware” and covers how data is represented, stored and processed and how computers are evolving regarding their computational power. The subsequent chapters cover the software setup including programming languages, compilers, programming tools and operating systems. The intention of part II is to explain how the software and hardware setup constrain the software system construction.

Part III introduces algorithms and data structures like recursive algorithms, searching, sorting, trees, and graphs. All these topics are covered in an object-oriented, library-driven way.

Part IV considers more advanced object-oriented techniques. It starts out with inheritance, continues with deferred classes and features and proceeds with
Part I: BASICS

1. The industry of pure ideas
2. Dealing with objects
3. The interface of a class
4. Just enough logic
5. Creating objects and executing systems
6. Control structures and routines
7. Describing syntax
8. Assignment and references
9. Fundamental data structures
10. Input, output and exceptions

Part II: HOW THINGS WORK

11. A glimpse into the hardware
12. Programming languages
13. Compilers and friends: the basic software tools

Part III: ALGORITHMS AND DATA STRUCTURES

14. Some key algorithms
15. Generic data structures
16. Recursion and trees
17. Algorithms for graphs
18. A beautiful algorithm family: Topological Sort

Part IV: OBJECT-ORIENTED TECHNIQUES

19. Inheritance
20. Deferred classes
21. Constrained genericity
22. Event-driven design
23. Going parallel: a look at concurrent programming

Part V: TOWARDS SOFTWARE ENGINEERING

24. The challenge of software quality
25. The software process
26. Writing requirements and documentation
27. Designing Graphical User Interfaces
28. Testing and debugging
29. Towards software reuse

Algorithm 6: Touch of Class table of contents
13.2. THE TRAFFIC SOFTWARE

The whole course builds around the TRAFFIC software, which is used both as a rich source for examples as well as the source for exercises. The choice of application domain required care since it needs to

- be immediately familiar to student in order to omit unneeded and distracting explanations on the context
- offer a large stock of exercises and examples that include recursive algorithms, the use of fundamental data structures, and traditional algorithms (e.g. graph algorithms, sorting, searching, etc.) for introductory programming
- call for graphics and multimedia development to capture the interest of the students
- be usable for advanced object-oriented and software engineering techniques, especially concurrent programming and event-driven design and the design of Graphical User Interfaces
- be extendible to be used for advanced courses as well

The application domain that was chosen can be described as the general notion of transportation in a city. It consists of various parts: modeling, visualization, planning, simulation, statistics. The TRAFFIC software includes a graphical frontend application and a backend library.
13.2. THE TRAFFIC SOFTWARE

13.2.1 The TRAFFIC library

The TRAFFIC library offers reusable components from which students and instructors can build and extend applications. Figure 13.1 shows the clusters of the TRAFFIC library. Currently, the TRAFFIC library provides

- Classes for generating a parse tree out of an xml file holding the description of the map, in particular class MAP_PARSER.

- Classes for the representation and processing of city elements collected by the parsing process like classes MAP, CITY, and METRO_STOP.

- Classes for the graphical display on the screen, e.g. MAP_DISPLAYER, METRO_STOP_DISPLAYER, and others.

The TRAFFIC library will continually be extended at ETH Zurich. We are also hoping for contributions by other universities that use the TRAFFIC library for teaching and by interested programmers.

13.2.2 The TRAFFIC application

For “Introduction to Programming” at ETH an application has been built making extensive use of the TRAFFIC library. The home works of the students will aim to add functionality to this application by further using the TRAFFIC library and other Eiffel standard libraries. At the beginning of the course it basically offers the following functionality:

- open a city map which will then be displayed
- zoom into the map
- zoom out of the map
- pane the map
- select stations, stops, landmarks and lines
- animated simple shortest path calculation that can be called if at least two stations or stops are selected
Figure 13.2: Traffic application
As can be seen in figure 13.2 the application has a set of empty buttons on the right of the window. These buttons will be used to integrate the functionality that was added by the students as homework into the application. Visualizations will be made directly on the map. The area on the bottom of the window (below the map) is used to display textual output. The reason for having such a console output is mainly to give students an easy way to test and debug their code by displaying strings. Debugging with the debugger of the programming environment needs experience and is, especially for beginners that do not yet know much about program execution and method invocation, much more difficult to grasp. Outputting text seems to be easier to understand: writing `console.print('anything')` outputs text on the console and students don’t have to care about stepwise execution and jumping in, out and over method calls. Later during the course, students will need to learn about using the debugger, but for the first couple of lessons they should not have to deal with yet another tool.

In Eiffel it is possible to structure the classes that are used for the application into different clusters. The clusters do not represent the hierarchical structure of the files and directories in the file system, but present the user with a well structured view of the classes that are available from the integrated libraries. In the case of the application, the following clusters are provided (see figure 13.3):

- **application**: This cluster holds application relevant classes like MAIN_WINDOW and ABOUT_DIALOG.
- **traffic**: This cluster contains the TRAFFIC library that is being developed particularly for the introductory programming course. It offers the necessary functionality to write applications modeling a city transportation system.
- **exercises**: Students will put classes that are used to solve homeworks into this cluster.
- **gobo**: This cluster holds Gobo Eiffel library classes. Gobo Eiffel offers a multitude of libraries like the Gobo Eiffel Structure Library (covering algorithms and data structures), the Gobo Eiffel Time Library (handling time and date conversions), the Gobo Eiffel XML Library, the Gobo Eiffel Parse Library, and many more (for more information see [16]).
- **vision2**: The classes of the library EiffelVision2 is located in this cluster. EiffelVision2 offers classes for graphical user interface development.
13.2. THE TRAFFIC SOFTWARE

as well as for drawing primitives. It heavily relies on WEL and GEL which are the platform dependent graphical libraries (see also [18]).

wel/gel: Under windows this cluster will be called wel, under Linux it will be named gel. The cluster holds classes stemming from WEL (Windows Eiffel Library), respectively GEL (Gtk Eiffel Library). These two libraries act as the lower tier of the two-tiered EiffelVision2 - WEL/GEL architecture. They are platform dependent and cover the platform specific graphical mechanism of Windows (WEL) and Gtk (GEL) (see also [18]).

base: Classes that are shown in this cluster, are coming from the EiffelBase Library. EiffelBase classes cover basic algorithms and fundamental data structures like lists, sets, tables, etc. (see [17]).
Chapter 14

Summary and outlook

This second part of the diploma thesis described the setup and organization of the redesigned introductory programming course at ETH Zurich. At ETH Zurich, for the introductory programming course a time of change has begun: the bachelor/master program for computer science majors will replace the traditional diploma program, the first programming language that students will be faced with will change from Oberon to Eiffel and the “Introduction to Programming” course will be based on a new teaching strategy: the Inverted Curriculum.

First, an overview of all the changes was given (chapter 9). This included a description of the organization of the bachelor program for computer science majors at ETH Zurich. Subsequently, general information like student numbers, amount of repeating students and failure rates was given. These informations were the basis to a discussion of the benefits and drawbacks of some fundamental organizational issues like the examination mode and the frequency of home works (see section 10.2 and 10.3). For the examination mode we will – at least for the first run of the course – stick to the ETH tradition. After that, the discussion will be reopened. Concerning the frequency of home works the decision was made to do them on a weekly basis during the first two parts of the course, and to require a project homework during the third part where software engineering topics are covered.

Since new material covering basic software engineering topics is added to the curriculum of the introductory programming course, it was important to come up with a reasonable schedule (see chapter 11). Furthermore, the necessity arose to check the covered topics against the topics that former Informatik1 courses (the equivalent to “Introduction of Programming”) introduced. In this way, we try to guarantee that the students that attend our redesigned course will meet the expectations and requirements of courses they will take later during their studies.

Since the students will be faced with the need of including their task solutions into the TRAFFIC software right from the start of the course, we must reassure that an easy-to-use and understandable interface exists that can be used with only little programming experience. The facilities to do this are explained in section 12.2. Besides that, a list of principles to guide the generation of good tasks was given in section 12.1. The last chapter described the course material, namely the text book Touch of Class and the TRAFFIC software.

The preparation of the course “Introduction to Programming” is still in full
swing and there are many more things to do. A major part will be the design of
good home works. This was not yet done because the TRAFFIC software was
still in a very unstable state. Now, with the software version of Milestone 2 the
creation of home works can be started. Besides this, we will need to create the
lecture slides and additional material e.g. a small how-to for EiffelStudio.

As described in part I of this thesis, I believe that most of the problems will
come with the programming environment EiffelStudio. In my opinion, while
it may be very comfortable for experienced programmers, it is too complex for
novice programmers. The preparation of a proposal for a redesign of EiffelStudio
needs a lot of research and was not possible in the given (short) time of this
thesis. It is however, still a point on the to-do list for the second run of the
course in winter 2004/2005. As for this first run, we will try to counter the
problems with a live demonstration of EiffelStudio during the exercise sessions
and a how-to that explains the use of it.

Special attention will be paid to the assessment of the course during the
semester. We will try to locate problems by making extensive use of polls
asking the students about their experience with the setup and material of the
course. This is also the point where we will learn whether EiffelStudio is as
problematic for unexperienced programmers as I fear.

All in all, the coming time promises to be exciting and suspenseful, and I’m
happy to be able to further assist in the development of the course.
Acknowledgements

I would like to thank Bernd Schoeller for his support, feedback and ideas that helped me to write this thesis. Thanks for always taking the time. Thanks to Till Bay, who accounted for a large part of the material that was the basis for the course comparison and who helped me to generate a PDF with hyperrefs. I should also single out Susanne Cech who supported me in the preparation of the mid-term presentation. Prof. Bertrand Meyer I would like to thank for giving me the opportunity of writing a diploma thesis on this subject and for valuable feedback at the mid-term presentation. Besides that, I would like to thank my parents for supporting me morally and financially through the whole course of my computer science study. Without them I couldn’t have done it.
Bibliography


