Exercise design for introductory programming
"Learn-by-Doing" basic O-O-concepts using Inverted Curriculum

Author(s):
Kessler, Marcel

Publication Date:
2004

Permanent Link:
https://doi.org/10.3929/ethz-a-005115601

Rights / License:
In Copyright - Non-Commercial Use Permitted
Exercise Design for Introductory Programming

“Learn-by-doing” basic O-O-concepts using Inverted Curriculum

MASTER THESIS

Marcel Kessler

3rd September 2004

http://se.inf.ethz.ch/projects/marcel_kessler

Student: Marcel Kessler (kesslema@student.ethz.ch)
Student-No: 99-918-849
Supervising Assistant: Michela Pedroni
Supervising Professor: Bertrand Meyer
To my parents
Abstract

In a computer science curriculum, the introductory course to programming is probably the most important course, because it can shape the student’s mind forever. In Winter 2003, ETH Zurich tried a new teaching approach called Inverted Curriculum, which is starting directly with the object-oriented paradigm. The approach is based around a library that the students first use and later modify and extend.

As the evaluation in this thesis shows, the first run of this completely new course was mostly successful. However, because preparation time was short, some material could still be improved.

This thesis lists possible improvements. One main aspect of the thesis is to present more suitable exercises. The other goal is to redesign and extend the library and increase its usability.
Contents

1 Introduction 8

2 Inverted Curriculum: The method 9
   2.1 A new approach 9
      2.1.1 Consumer-to-producer strategy 9
      2.1.2 Progressive opening of black boxes 10
      2.1.3 Outside-In: The Inverted Curriculum 10
   2.2 Summary 11

3 Winter 2003 - First try 12
   3.1 Description 12
      3.1.1 Eiffel 12
      3.1.2 EiffelStudio 13
      3.1.3 Lecture topics 14
      3.1.4 Textbook 16
      3.1.5 TRAFFIC 16
      3.1.6 Exercises 18
   3.2 Composition of the student body 21
      3.2.1 General Data 21
      3.2.2 Computer Usage 22
      3.2.3 Programming Experience 23
      3.2.4 Conclusion 26
   3.3 Course and exercises evaluation 27
      3.3.1 Assistants feedback 27
      3.3.2 Students feedback 29
      3.3.3 Other comments 35
   3.4 Conclusions 36
      3.4.1 Order of topics 36
      3.4.2 TRAFFIC 36
      3.4.3 Exercises 36
      3.4.4 Project 37
      3.4.5 Eiffel and EiffelStudio 37
      3.4.6 General remarks 37
   3.5 Try again? - Sure! 38

4 Paper reading - What others found out... 39
   4.1 Overview 39
   4.2 Exercise Design 39
      4.2.1 The Quest for Excellence in Designing CS1/CS2 Assign-
      ments [7] 39
      4.2.2 Principles for Designing Programming Exercises to Mini-
      mize Poor Learning Behaviors in Students [2] 41
      4.2.3 Characteristics of Programming Exercises that lead to
      Poor Learning Tendencies: Part II [3] 42
4.3 Games for teaching? ........................................ 43
  4.3.1 Teaching the Nintendo Generation to Program [10] .... 44
  4.3.2 Teaching with games: The Minesweeper and Asteroids
      experience [1] ........................................ 44
  4.3.3 Adding Breadth to CSI and CS2 Courses through Visual
      and Interactive Programming Projects [12] ............... 45
  4.3.4 Engaging Girls with Computers through Software Games
      [9] .................................................. 45
4.4 Other interesting findings .................................... 46
  4.4.1 Gender Differences in Programming? [4, 3] ............. 46
  4.4.3 Positive Experiences with an Open Project Assignment in
      an Introductory Programming Course [22] ................. 47
  4.4.4 Improving the CSI Experience with Pair Programming [20] 48
  4.4.5 Blooms taxonomy [6, 15, 8] ........................... 49
4.5 Conclusion: Our guidelines .................................. 52
  4.5.1 Concerning the exercises .............................. 52
  4.5.2 Project ideas ...................................... 53
  4.5.3 Games are fun! .................................... 54
  4.5.4 General ideas ...................................... 54

5 TRAFFIC 2: The Redesign 56
  5.1 Overview ............................................... 56
  5.2 The new library ........................................ 56
    5.2.1 What was wrong? .................................. 56
    5.2.2 Overview of TRAFFIC 2 ........................... 56
  5.3 FLAT_HUNT - The game ................................ 57
    5.3.1 Background ....................................... 57
    5.3.2 The idea ......................................... 58
    5.3.3 The looks ........................................ 59
    5.3.4 Behind the scenes ................................ 59
  5.4 FLAT_HUNT User & Developer Documentation ............. 60
  5.5 Summary ............................................... 82

6 The new exercises 83
  6.1 Overview ............................................... 83
    6.1.1 Description of the assignments ....................... 85
    6.1.2 The right mix ..................................... 91
    6.1.3 Exercise structure ................................ 93
  6.2 Two examples ........................................... 93
    6.2.1 My first program .................................. 93
    6.2.2 Loop exercise (Fancy animate_defeat animation) .... 95
  6.3 The final project ....................................... 96
    6.3.1 The options ...................................... 97
    6.3.2 How to form the groups? ........................... 98
    6.3.3 Just coding? ..................................... 98
6.4 Summary ................................. 98

7 Summary and outlook .......................... 99

A Project plan .................................. 101

B Style guide .................................. 105

C Exercises .................................. 109
   C.1 Assignment 1 ............................. 109
   C.2 Assignment 2 ............................. 113
   C.3 Assignment 3 ............................. 120
   C.4 Assignment 4 ............................. 126
   C.5 Assignment 5 ............................. 131
   C.6 Assignment 6 ............................. 135
   C.7 Assignment 7 ............................. 142
   C.8 Assignment 8 ............................. 147
   C.9 Final Project ............................. 153

References .................................. 159
“Designing assignments is one of the most personalized aspects of teaching, and therefore also one of the most gratifying.”

Todd J. Feldman and Julie D. Zelenski, The Quest for Excellence in Designing CS1/CS2 Assignments [7]
1 Introduction

In an university setting, learning happens through two means: Attending lectures and solving exercises. In computer science, working on exercises is for many students the more important task. Thus, 'good' exercises are very essential to the success of a course. "What are 'good' exercises?"

This is one of two core questions that this thesis tries to answer. The other question is "How can TRAFFIC be improved?" TRAFFIC is a library of classes that can be used in a Geographical Information System (GIS). TRAFFIC was especially designed for ETH Zurich's 'Introduction to Programming' course.

Since Winter 2003, this course is taught using a new approach called Inverted Curriculum. One of the main ideas of the Inverted Curriculum is that the students do not start from scratch. Instead, they work with a library: the above mentioned TRAFFIC.

Chapters 2 and 3 of this thesis describe the idea of the Inverted Curriculum and its first implementation in Winter 2003 at ETH Zurich. Chapter 3 includes a thorough evaluation of this first run.

Chapter 4 gives an overview of some interesting papers in the field of exercise design and introductory programming and draws conclusions for our own course. In Chapter 5 the whole redesign of the TRAFFIC library is discussed, and Chapter 6 takes a closer look at the new exercises. Summary and outlook are provided in Chapter 7.

Appendix A hosts the project plan, and Appendix B presents a guide to the Eiffel style. All exercise sheets that were produced during this thesis are listed in Appendix C.

\footnote{In the evaluation of 'Introduction to programming 2003', 51\% of the students agreed that transfer of knowledge happened in the lectures. Even more, 68\% of the students, claimed that the transfer happened during the exercises. For more information, see Section 3.3.}
2 Inverted Curriculum: The method

2.1 A new approach

In 1993, Bertrand Meyer published the second edition of his famous book 'Object Oriented Software Construction' [16], often just called OOSC. In chapter 29, 'Teaching the method', he writes - "from the comfort of a position in industry" [18] - about some ideas related to introductory programming courses. He advocates the use of the object-oriented method right from the beginning: "If you think object-oriented development is the right way to go, there is no reason to make a detour first" [16]. In addition, Meyer advises that teachers should use a language with syntactic clarity, where no time has to be spent on explaining the language per se. Introductory courses, which shape the student’s mind forever, should use the best technical approach, and therefore use a pure O-O language, and not a hybrid like C++.

In the last ten years, Bertrand Meyer presented his ideas about how to teach introductory programming courses in various publications. In this process he came up with different names like 'Inverted Curriculum', 'consumer-to-producer strategy', 'progressive opening of black boxes' and 'Outside-In'. At the core, all this terms express the same idea, however, they stress different aspects. The following sections describe these aspects.

2.1.1 Consumer-to-producer strategy

In a section of OOSC called 'Towards a new software pedagogy', Bertrand Meyer describes a new pedagogical technique, at that time called 'consumer-to-producer strategy'. Instead of computing the first 25 Fibonacci numbers (a typical introductory exercise) students are able to produce impressive applications almost on day one - by reusing existing software.

The students are given a good O-O library, and their first task is merely to combine existing components and assemble them into systems. Later, they will be shown the internals of the components, then they will be asked to extend and modify classes, and finally they will write their own classes. In five stages, the students start as consumers of reusable components and learn to become producers:

1. Learn to use library classes, solely through their abstract specifications.
2. Learn to understand the internals of selected classes.
3. Learn to extend selected classes.
4. Learn to modify selected classes.
5. Learn to add your own classes.
2.1.2 Progressive opening of black boxes

Another name for this method is ‘progressive opening of black boxes’: The students first use powerful tools - as clients - for their own applications. Then they “progressively lift the hood to see how things are made, make a few modifications and finally add their own extensions” [17]. From the start the focus is on powerful and possibly large examples, ideally handling graphics and thus generating a 'wow effect' even for Nintendo Generation students [10].

Two other aspects of this approach are worth mentioning: Abstraction and apprenticeship.

Abstraction

Experienced software professionals and teachers know about the importance of abstraction, and they often preach abstraction. However, “to preach is not the best way to teach. With the consumer-to-producer strategy, based on libraries, abstraction is not something to pontificate on: it is a practical and indispensable tool. Without abstraction, one cannot use libraries” [16].

Apprenticeship

Apprenticeship is a time-honored technique: You learn from the previous generation of master practitioners, and once you have understood their techniques you try to do better. In the consumer-to-producer strategy, the reusable components of the library are the 'masters', and you end up producing new components by imitating them.

Through this approach, the students should also learn the importance of well-documented code, easy design and good style.

2.1.3 Outside-In: The Inverted Curriculum

In the field of electrical engineering, there is an interesting counterpart to the consumer-to-producer strategy called 'Inverted Curriculum'. While the classical progression is from field theory to VLSI design, Inverted Curriculum is a more systems-oriented approach which goes from digital systems to device physics. Start by giving the students a user's view of the highest-level concepts that are actually applied in industry, then, little by little, unveil the underlying principles.

Outside-In [18]

This is the last and probably best known term for the new method. It incorporates the ideas of consumer-to-producer strategy, progressive opening of black boxes and Inverted Curriculum, and contrasts it with traditional bottom-up approaches. The focus is on first using good existing software 'from the outside' by calling features on given classes, and then having a closer look 'at the inside' and starting to create your own classes. But not only the order of concepts differ, there is also more emphasis on architectural skills, because successful software
development relies on the construction of models of physical and conceptual systems (model-driven architecture, [18]). Thus topics like project management and testing should also be covered.

Focusing on object technology does not mean that there are no variables and no control structures. Objects contain fields and classes contain routines. Although the stress of the Outside-In approach is clearly on 'programming in the large', low-level concepts and skills are not neglected. The traditional concepts are included, but reaching classes and objects is guaranteed, and not a nice-to-have-if-time-permits like in a bottom-up order of concept introduction.

2.2 Summary

In her diploma thesis [21], Michela Pedroni lists the following elegances of the Inverted Curriculum 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.
- 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. 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.
3 Winter 2003 - First try

In Spring 2003, ten years after writing down his ideas on teaching introductory programming in OOSC second edition, Bertrand Meyer left the “comfort of a position in industry” and became Head of the Chair of Software Engineering at the Swiss Federal Institute of Technology (ETH). He accepted the invitation to take responsibility for the CS1 course ‘Introduction to Programming’, and not denying what he wrote earlier, he tried to apply his ideas (full use of object technology, ‘Progressive opening of black boxes’, Inverted Curriculum) to this course. Following up is a description of the first run of ‘Introduction to Programming’ at ETH in Winter 2003. Some of this material is taken from 'The Outside-In method of teaching introductory programming' [18].

3.1 Description

In the years before, ‘Introduction to Programming’ was based on a traditional bottom-up approach using the language Oberon. Therefore, the whole course had to be completely redesigned. Bertrand Meyer and his group had to put a big effort in devising a complete new set of material, consisting of the following elements:

- The course itself including lecture slides and exercises.
- A new textbook called ‘Touch of Class’ [19], available to the students in electronic form.
- The supporting software-library (TRAFFIC) used for the Outside-In approach.

We will soon take a closer look at some of these elements. But before that, we have a glance at Eiffel and EiffelStudio, the language and the environment used for 'Introduction to Programming'.

3.1.1 Eiffel

Eiffel is a pure object-oriented language with the following properties (as described in 'Teaching introductory programming with the Inverted Curriculum approach' [21]):

- Object-oriented concepts are supported in a clean way.
- Syntax is clear and easily readable (no cryptic symbols, ‘=’ means equal, semicolons are optional).
- Clear constructs (e.g. loop) allow easy transition to other languages.
- Automatic garbage collection (no low level operations needed).
- Design by Contract: Sophisticated assertion mechanism allows for a more systematic approach.
Overall, Eiffel seems to be “a better candidate than any other language” [13] for ‘Introduction to Programming’. Although Eiffel is used commercially, it is not a language that students are likely to know. This helps in averaging the gap between more and less experienced students, and avoids so-called ‘Google-and-Paste’-programming (see 'The Outside-In method' [18] for more on this).

3.1.2 EiffelStudio

“A suitable programming environment is crucial for the success of an introductory course” [14].

EiffelStudio is the most widely used Eiffel development environment. It comes in a free and a professional version.

The EiffelStudio IDE features the following:

- A modern GUI with browsing, editing and debugging facilities (see Figure 1).
- Automatic documentation generation.
- Automatic architectural diagram creation (only professional version).

EiffelStudio offers a large amount of functionality. This is certainly comfortable for experienced programmers, but Michela Pedroni feared that “It offers too
much functionality for beginners, and will therefore probably cause problems for the students" [2].

3.1.3 Lecture topics

As 'Introduction to Programming 2003' was the first course conforming to the Inverted Curriculum, a whole new schedule had do be developed. The course was divided in three parts:

- Part I (Weeks 1 - 6): Basics
- Part II (Weeks 7 - 11): Advanced O-O topics (e.g. inheritance), data structures
- Part III (Weeks 12 - 14): Software engineering techniques

Table 2 gives a more detailed overview of the effective schedule in Winter 2003.
<table>
<thead>
<tr>
<th>Week</th>
<th>Lecture topics</th>
<th>Exercise topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Lecture 1.1:</strong> No lecture</td>
<td><strong>Exercise 1:</strong> Get used to ETH infrastructure (e-mail, newsgroups)</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 1.2:</strong> Introduction</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><strong>Lecture 2.1:</strong> Dealing with objects I</td>
<td><strong>Exercise 2:</strong> It's logic!</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 2.2:</strong> Logic</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><strong>Lecture 3.1:</strong> Dealing with objects II</td>
<td><strong>Exercise 3:</strong> First program / Contracts</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 3.2:</strong> The interface of a class</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><strong>Lecture 4.1:</strong> Creating objects</td>
<td><strong>Classroom exercise 1:</strong> Concepts / Logic / Contracts</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 4.2:</strong> Control structures I</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><strong>Lecture 5.1:</strong> Control structures II</td>
<td><strong>Exercise 4:</strong> Control structures</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 5.2:</strong> Syntax (EBNF)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><strong>Lecture 6.1:</strong> References and assignments</td>
<td><strong>Exercise 5:</strong> Syntax (EBNF)</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 6.2:</strong> Dynamic model I</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><strong>Lecture 7.1:</strong> Dynamic model II</td>
<td><strong>Exercise 6:</strong> References and assignments</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 7.2:</strong> Container data structures</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td><strong>Lecture 8.1:</strong> A simple program</td>
<td><strong>Exercise 7:</strong> Data structures and programming</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 8.2:</strong> The hardware setup</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td><strong>Lecture 9.1:</strong> Recursion</td>
<td>No exercise</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 9.2:</strong> Inheritance</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Christmas break</strong></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><strong>Lecture 10.1:</strong> Inheritance and genericity</td>
<td><strong>Classroom exercise 11:</strong> Recursion / Data structures</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 10.2:</strong> Testing</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td><strong>Lecture 11.1:</strong> More inheritance</td>
<td><strong>Project:</strong> Extend TRAFFIC</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 11.2:</strong> Event-driven programming I</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td><strong>Lecture 12.1:</strong> Event-driven programming II</td>
<td><strong>Project:</strong> Extend TRAFFIC</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 12.2:</strong> Undo/Redo example</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td><strong>Lecture 13.1:</strong> 'Topological sort I: Background'</td>
<td><strong>Project:</strong> Extend TRAFFIC</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 13.2:</strong> 'Topological sort II: Algorithm'</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td><strong>Lecture 14.1:</strong> From programming to software engineering</td>
<td>No exercise</td>
</tr>
<tr>
<td></td>
<td><strong>Lecture 14.2:</strong> Glance at 'Algorithm and data structures' / Project presentation</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Effective schedule 2003 [21]

A closer look at Part I clearly reveals the Outside-In approach: Starting out with 'Dealing with objects', moving on to abstraction ('The interface of a class') and ending with 'References and assignments'. Part II covers more advanced topics like data structures, recursion, inheritance and genericity. In the last few weeks software engineering techniques, e.g. testing, are introduced.

The last column in Table 2 shows an overview of the exercises. More on this in Section 3.1.6.
3.1.4 Textbook

While preparing the course, and throughout the course, Bertrand Meyer was working on a textbook called "Touch of Class" [19] (see Figure 2).

![TOUCH OF CLASS](image)

Figure 2: 'Touch of Class' textbook

Although it is still no completely finished, most of the chapters are written. The textbook is very extensive (already counting 540 pages), and covers more than will be needed in the first semester. The book is divided into 5 parts. Table 3 gives you an overview of the content of 'Touch of Class'.

3.1.5 TRAFFIC

To really support the idea of Inverted Curriculum and the Outside-In approach, a library to use, learn from and finally extend was needed. This library had to meet the following requirements [18]:

- Be familiar to the students, so that they don't have to spend time on unnecessary details.
- Provide interesting algorithm and data structure examples.
- Apply fundamental computer science concepts.
- Have an advanced GUI.
- Include graphics (to capture the attention of a new generation of students).
Table 3: 'Touch of Class' contents
• Not involve violence and aggression.

• Be extensible for advanced courses.

Bertrand Meyer and his group chose a library in the application area of transportation in a city. The library was named TRAFFIC and features the basic elements of a Geographical Information System (GIS). It can display a city with its sights and its transportation networks (see Figure 3). The city’s description is an XML-file, thus it can be retargeted to other cities.

The map of the city can be zoomed and moved, stations and sights can be selected and possible routes can be animated.

![Figure 3: TRAFFIC screenshot](image)

### 3.1.6 Exercises

When planning the new course, the frequency of the homework was also a point of discussion. Homeworks are an important part of any course, as they are responsible for enforcing the taught concepts. Between the two extremes ‘weekly homeworks’ and ‘homeworks that stretch over several weeks’ the following compromise was found:

There would be mostly weekly exercises, because they give the students a more immediate feedback, and it is less depressing if they don’t manage to solve one of these exercises. However, to reinforce problem solving and software engineering skills (another focus in the Inverted Curriculum), the last four weeks are dedicated to a bigger project exercise. The last column in Table 2 gives an overview of the exercise schedule.
To prepare the students for the exam, and to give them a chance for a self-assessment, two of the weekly exercises were so-called classroom exercises. They were paper-and-pencil exercises - more on the theoretical side - to check whether the students understood the concepts. The classroom exercises were graded, but they had no influence on the mark in the exam.

In the 'normal' exercises, the questions were diverse. The majority of the exercises was based on the TRAFFIC library. Listing 1 shows the first exercise using TRAFFIC. Listing 2 gives the respective solution. Listing 3 shows the introduction to the project exercise that took place towards the end of the semester.

### A first program

You will do your first steps in programming. The class that you will change is called PREVIEW (the class holding the example from the 'Touch of Class' book). Open this class in EiffelStudio and perform the following tasks:

1. Change the ordering of the given feature calls.

2. Output all the names of the used objects in the console window (including the console itself which is also such an object).

**Hints**

- Read the text book and solve this exercise analogously to the example described in Chapter 2.
- Don’t forget to compile your project every time you change something in the class text, before running it. (If you don’t do so, you will run a version without your latest changes.)

**To hand in**

Hand in your class text and the output of the console window that your program produces.

---

### Listing 1: First exercise task

1. Ordering

```eiffel
explore is
  -- Explore the map with changed order.
  do
    Paris.display
    Line8.highlight
    Louvre.spotlight
    Route1.animate
    Console.show (Route1.origin)
  end
```
2. Output

To output all the used objects the following lines of code must be added to feature *explor*:

```java
Console.show (Console)
Console.show (Paris)
Console.show (Louvre)
Console.show (Line8)
Console.show (Route1)
```

This call sequence yields the following output on the Console window:

```
Console: text window shown on the bottom of the TRAFFIC application
Paris: It is the capital of France. In 1995, the population of the Paris metropolitan area (Ile-de-France) was estimated at 10,982,000 inhabitants.
Louvre: one of the most famous monuments of Paris
Line 8: from Balard to Creteil-Prefecture
Route1: Route going from Porte Dauphine to Gare de Lyon
```

Listing 2: First exercise solution

---

Project exercise

The project exercise is to be solved in groups of three. Try to form mixed groups concerning your programming experience.

The project

Your task is to extend TRAFFIC with an entertainment information system. Your system should be able to store and process information about events, e.g. a movie playing at a distinct cinema, a concert that takes place in a bar, or a play that is shown in a theater. Some events that you will have to model are recurring, like a movie that is played in the same cinema everyday at the same time, or a late night movie show that is up on a weekly basis. The locations at which the events take place are LANDMARKs. Note that the same show can be located at different places, e.g. a movie may be shown in many different movie theaters. Your system should hold information both about the events (date, time, location, show, etc.) and the shows (type (movie, concert, ...), title, description, actors, directors, and genre for movies, musicians for concerts).

The data of the events should be read in from a file. How the file looks like is up to you. You will find an example text file (not XML) under [http://se.inf.ethz.ch/teaching/ws2003/37-001/downloads/project](http://se.inf.ethz.ch/teaching/ws2003/37-001/downloads/project). There has also been a map of Zurich prepared that shows all cinemas and some concert halls. It can be found under the same URL as the data file.

Listing 3: Project exercise
3.2 Composition of the student body

In the first lecture of the new ’Introduction to Programming’ course, a survey was conducted to estimate the constitution of the student body. Following up are some of the more interesting findings.

3.2.1 General Data

A total of 222 students took part in this survey. More than \( \frac{3}{4} \) of the students are aged between 19 and 22 years. However, the youngest student is only 17, while the oldest student is aged 39. Obviously, they will have a somewhat different background.

The proportion of female students is (unfortunately) still less than 10% (see Figure 4). Nevertheless, one must be conscious that man and women may approach learning in different ways. For example, Janet Carter and Tony Jenkins found out that women are more likely to approach staff directly when they need help [4, 5] (for more on this subject, see Section 4.4.1 on page 46).

![Figure 4: Gender distribution](image)

More than 80% of the students have German as their mother tongue, yet there are also 18 students speaking Italian at home. Only 5 students usually speak French. The students high school background is diverse, but 60% have a clear focus on mathematical subjects (Typus C or Physik/Anwendungen der Mathematik).

One third of the students are not new to ETH, they have already spent one year studying computer science, but unfortunately did not pass the exam. These students are taking ’Introduction to Programming’ for the second time. However, the first time they followed the old course taught by Jürg Guttenecht using a traditional teaching approach and Oberon as the programming language.
3.2.2 Computer Usage

Most of the students (93%) have been using the computer for more than 5 years. But there are also two students that started using computers less than a year ago. 4 out of 5 students use the computer every day, and 98% use it at least once a week.

Figure 5: Computer usage

Almost all the students use the computer to connect to the world by surfing the web and handling e-mails (Figure 5). Word processing and computer games are another big area of computer usage. Only one of four students is not playing computer games from time to time. This goes in line with Katrin Becker’s finding that “when students are asked, many will say that game playing is what got them interested in computers in the first place” [1]. See also Section 4.3.2 on page 44 for more on this. A closer look at Figure 6 shows that not only more than \(\frac{3}{4}\) of the boys like to play, but also almost every second girl.

Figure 6: Game playing students
A bit less than 50% use the computer for tasks like computer graphics and web design, and about the same number call themselves system administrator.

Two out of three students use the computer for programming. When it comes to operating systems, Microsoft is very strong around first year students. Only one of them reported that he knows nothing at all about Windows, while the majority thinks that they know it very well.

Linux/Unix has pretty similar figures as Mac. There is only a few who consider them as experts, and one third does not know these two operating systems at all (see Figure 7).

![Knowledge of Operation Systems](image)

Figure 7: Operating system knowledge

More than 50% of the students are mobile, meaning that they have a notebook, and 87% have (sometimes in addition) a PC at home. Unfortunately, at the time of the survey, one student seemed to have no access to a computer.

### 3.2.3 Programming Experience

The first question in this section was “Do you have programming experience (HTML does not count)?” (see Figure 8). 48 out of 222 students (22%) answered ‘no’ and could skip the rest of the survey. This number goes well with Tony Jenkins findings in 'Dealing with diversity in introductory programming' [11] (see Section 4.4.2 on page 46 for a summary of this paper), where he notices that in a computing class of 1985, every student could program to some extend, whereas nowadays, they may have done some programming, but this cannot be assumed.
So 78% of the students do already have some programming experience, however, half of them is only used to procedural programming, and only 38% of all the students have some experience with object-oriented programming. This figure is also supported by the question “How well do you know the following programming languages?” (see Figure 9).

Procedural languages like C, Pascal and Basic are a little bit better known than Java, C++ and C#. What is very interesting in the context of ‘Introduction to Programming’ is that almost nobody knows the language which is used
in this course: Eiffel. As Michela Pedroni points out in her thesis [21], having an unpopular language has the benefit that all students are confronted with a new programming language no matter whether they have programming experience or not. Thus, Eiffel can act as an equalizer between newbies and already quite advanced programmers. Apart from the languages shown in Figure 9, it became also clear that several students know web-languages like Perl and PHP.

![Figure 10: Programming concepts](image)

Figure 10 makes the difference between programming and O-O programming clear again: About 130 students know about concepts like variables, procedures, control and data structures, but only about 120 students have experience with object-orientation, and less than 100 students have ever used inheritance. Yet, when talking about these 'details', one has to keep in mind that almost 50 students have no programming knowledge at all.

Where did those students experienced in programming learn it from? Figure 11 shows that the majority studied it at home. Some were educated at high school (26%), and an ever smaller number took some programming courses. Of the students that learned to program, almost one fourth could use this skill to earn money.
In Figure 12 (Programmed Lines of Code), one can see again the 22% that never programmed at all. About 50% coded something between 10 and several 100 lines of code. Only 17% wrote programs with more than 1000 lines of code.

3.2.4 Conclusion

The student body at ETH is clearly very diverse. There are those that have barely touched a computer and those that have programmed much more than 1000 lines of code, using modern O-O techniques on new Linux notebooks bought from money they earned with making PHP-websites for large companies. Can
an introductory programming course fulfill the needs of both? The evaluation of our course in the next section will give an answer...

### 3.3 Course and exercises evaluation

Evaluation is always important, but even more so when something runs for the first time. Lots of questionnaires were filled out during the course 'Introduction to Programming'. Not only did we want to know precisely the student body (see above), we were also interested in the students feedback on the exercises and the course in general. Also the teaching assistants were asked about their impression. This section first just lists comments of assistants and students, including some graphics of the end-of-course evaluation.

After these comments, conclusions are drawn.

#### 3.3.1 Assistants feedback

Following is a list of remarks from the eleven assistants that were teaching the exercise session and had close contact with the students. The (uncommented) remarks are taken from a questionnaire developed by Michela Pedroni. They are grouped into themes for better understanding.

**Order of topics:**

- "Assignment was introduced to late."
- "Some students complained about order of topics."
- "Inheritance should be treated earlier."
- "Rethink the order of topics in the course."
- "Inverted Curriculum was no success, start from scratch like in the good old days."

**TRAFFIC:**

- "TRAFFIC needs more documentation (about architecture and classes)."
- "Reimplement TRAFFIC from scratch."
- "TRAFFIC is too slow and big."
- "Provide documentation in every class, have cooler effects."
- "Show TRAFFIC's design (in BON-notation) to the students."
- "TRAFFIC needs a better class documentation."
- "TRAFFIC needs to be completely redesigned."
Classroom exercise:

- “Classroom exercises are a good way for the students to see where they stand and what the exam might be like.”
- “First exercise was too difficult, second one was too easy.”
- Would you still do the classroom exercise next year?
  Yes: 11
  No: 0

Project:

- “Provide an introduction to GUI-design.”
- “Project based on TRAFFIC was not welcome.”
- “Students really learned programming with the project.”
- “Break project into parts, add milestones.”
- “Give those students that want to do something completely different this possibility (depending on their proposal).”
- “It is hard to motivate yourself when there is so few objectives.”
- “Avoid TRAFFIC in the final project.”
- Would you still do the project exercise next year?
  Yes: 10
  No: 1

Problems:

- “Inheritance vs. client-supplier relationship (at beginning).”
- “Inheritance vs. client-supplier relationship, object creation, ’.’-syntax.”
- “Dynamic binding.”
- “Inheritance vs. client-supplier relationship, reference vs. value, syntax.”
- “Making orderly code, compilation, understanding syntax errors.”
- “Recursion, locality of variables, inheritance vs. client-supplier relationship.”
- “Getting access to a class’ attributes.”
- “Understanding TRAFFIC, feature calls, object creation.”
- “TRAFFIC and EiffelStudio.”
- “Recursion and TRAFFIC-API.”
General remarks:

- “Make two kind of exercise groups: more experienced with a 2 hours exercise session and less/not experienced with a 4 hours exercise session.”

3.3.2 Students feedback

After every exercise, the students had to fill in a questionnaire. The questionnaire asked how much time they needed for each task, and how difficult they found it (see Listing 4).

Questionnaire

This page has to be filled out and handed in together with your exercise.

Exercise ....  Name .................  Group .................

General impression

Grade of difficulty:  (easy) O O O O O (difficult)
Time needed: 
Comments: 

Part 1

Grade of difficulty:  (easy) O O O O O (difficult)
I had problems with 

Part 2

Grade of difficulty:  (easy) O O O O O (difficult)
I had problems with 

During the following exercise session I would like to hear more about the
The students could also give feedback about the exercise in general, and suggest topics for the next exercise session. Following up is some of the praise and criticism the student mentioned. Although these comments were given for each individual exercise, here they are grouped into topics for better analysis.

Towards the end of the course, the students also took part in the official ETH evaluation. There we could ask some additional questions, which were in the form of statements where the students could say whether they agree or not. These questions were statistically analyzed and put into nice diagrams\(^2\). These diagrams are also presented under the appropriate header.

**Exercise questions:**

- “Exercises should be on topics discussed in the lecture.”
- “I did not understand the questions.”
- “Giving examples would help us understanding what is asked.”
- “Difficulty understanding the questions.”
- “Unclear questions.”
- “Hints were partly wrong.”
- “When to inherit and when to use it as a client?”
- “How important are clone, copy etc. - they only got mentioned shortly during the lecture.”

\(^2\)As this evaluation was done in German, the questions below are translated but should convey the same meaning. In the pictures, ‘MW’ stands for mean value, ‘SA’ is the variance, ‘k. Ä.’ means that no answer was given, ‘ung.’ and ‘gültig’ stand for invalid and valid respectively. ‘Anz.’ is the number of answers. ‘1’ means that the quote does not apply at all, ‘2’ is little agreement, ‘3’ is partly/partly, ‘4’ means general agreement and ‘5’ is complete agreement.
• “The exercises were formulated clearly and understandable.” (Figure 13)

![Figure 13: Clear and understandable exercises?](image13)

Classroom exercises:

• “Please print the exercises on rag-paper.”

• “Classroom exercises were useful.” (Figure 14)

![Figure 14: Classroom exercise a good idea?](image14)

Project:

• “It was a pity that we had to make a TRAFFIC extension. Everybody should have a chance to do what he wants to do.”

• “A project over several weeks is a good idea.” (Figure 15)

![Figure 15: Project exercise a good idea?](image15)
TRAFFIC:

- “Few like TRAFFIC.”
- “The software TRAFFIC was unstable and slow, even on very fast machines. I’d prefer another sample environment.”
- “TRAFFIC: too big and intransparent.”
- “TRAFFIC is too big to introduce simple concepts.”
- “Slow speed of TRAFFIC.”
- “Compilation of TRAFFIC at ETH takes half an hour.”
- “Would like to write little programs from scratch.”
- “Like this exercise because there is no TRAFFIC.”
- “Would rather work from scratch than on TRAFFIC (TRAFFIC is buggy).”
- “Finding classes, understanding structure of TRAFFIC.”
- “Writing programs from scratch would be better.”
- “TRAFFIC is missing a good documentation.”
- “Learning the basics of programming within someone else’s program is not good. Most of the time the problem is understanding TRAFFIC not O-O programming.”
- “Usually, if something doesn’t work I think that it’s my fault, but TRAFFIC also has bugs. If you can’t trust the base you’re working on, learning becomes unpleasant.”
- “I liked the idea to work with a predefined software system (i.e. TRAFFIC).” (Figure 16)

![Figure 16: TRAFFIC a good idea?](image)

- “TRAFFIC included useful examples to introduce new concepts during the lecture.” (Figure 17)
• “TRAFFIC was a useful basis for the exercises.” (Figure 18)

Figure 17: TRAFFIC: Useful examples?

Figure 18: TRAFFIC: Good exercise basis?

TRAFFIC and EiffelStudio:

• “Would be helpful to have list of useful commands.”
• “Where to find features?”
• “Where to find needed objects?”
• “How to start?”
• “Tedious to find out which features do what.”
• “What features from library to best use?”
• “How to call the right feature on the right object?”
• “As a beginner without previous knowledge, getting used to Eiffel and in particular TRAFFIC was hard.”

Eiffel and EiffelStudio:

• “Understanding what compiling, ace-file etc. means.”
• “How to use EiffelStudio, solve examples?”
• “EiffelStudio’s code editor is not user friendly.”
• “Eiffel as a programming language was easy to learn.” (Figure 19)

![Figure 19: Eiffel easy to learn?](image)

• “Eiffel helped in introducing the concepts.” (Figure 20)

![Figure 20: Eiffel useful to introduce concepts?](image)

• “Using Eiffel Studio was a pleasant experience.” (Figure 21)

![Figure 21: EiffelStudio pleasant?](image)

3.3.3 Other comments

• “Substantial improvement to last year’s course.”

• “The professor managed to present the matter in an understandable and plastic way.” (Figure 22)
Figure 22: Understandable presentation?

- “General impression of the course.” (Figure 23)

3.4 Conclusions

To improve the overall quality of ‘Introduction to Programming’ in 2004, several changes were suggested. These propositions are again grouped in topics.

The final outcome concerning TRAFFIC will be discussed in Chapter 5, the new exercises are presented in Chapter 6.

3.4.1 Order of topics

The Outside-In approach seems to have worked. However, minor adjustments are needed. For example, assignment should be introduced earlier. One reason for this is that it is very hard to have interesting ‘non-TRAFFIC’ exercises without assignment.

Another finding was that lots of students had problems with the difference between inheritance and the client-supplier relationship. This topic should also be discussed earlier.

3.4.2 TRAFFIC

The evaluation clearly showed that most improvements can be made by redesigning TRAFFIC. TRAFFIC 1.0 was very slow, unstable, big, complex and missing a good documentation. The two diagrams (Figures 16, 17) show that students opinion about TRAFFIC is pretty averaged, they did not find it very helpful.

TRAFFIC has already been rebuilt almost from scratch to make it slimmer and improve clarity (for example through better clustering, see Chapter 5). Still
very important is a good documentation that gives an overview of the whole system. TRAFFIC could also be made more attractive if it was more game-like, had nicer graphics and would play sound-effects. This is in accordance with Guzdial and Soloway's [10] discussed in Section 4.3.1 on page 44.

3.4.3 Exercises

In general, the questions should be more clear, and it would be useful to have some sample answers and maybe some hints on how to start. However, 86% were more or less happy with the exercises (see Figure 13).

The classroom exercises can be kept (only 10% did not like them, see Figure 14), but I agree with that one student suggesting that we print these big exercises on recycling paper.

With the normal exercises, there seems to be too much focus on TRAFFIC exercises. More than 50% of the students are of the opinion that TRAFFIC is not a good base for exercises (see Figure 18). Is this due to TRAFFIC’s instability and poor documentation, or because students do not like the idea to work with a big library in general? The students comments support both points.

Even though the Inverted Curriculum seems to be a good idea and should be kept, the students will want to know how to write a program from scratch. Thus, such a task should come up earlier in the exercises. The following rule-of-thumb might be successful: In each assignment, have one third TRAFFIC exercises, one third stand-alone exercise involving coding, and one third paper-and-pencil exercises.

3.4.4 Project

Giving the students four weeks to work on a longer assignment has generally worked out. Figure 15 shows that one third liked the project a lot. However, maybe the students should have the choice between a project that is based on TRAFFIC and something completely different. Our own experience in another (more advanced) course[^2] showed that an open final project can lead to very impressive results (see http://se.inf.ethz.ch/people/bay/esdgames/ for an idea).

3.4.5 Eiffel and EiffelStudio

There were no comments about problems with the language Eiffel, which can certainly be viewed as a positive feedback. 4 out of 5 students said that Eiffel was rather easy to learn, and that it helped to introduce the concepts (see Figures 19, 20).

EiffelStudio is a bit of a different story - Michela Pedroni’s fear (Section 3.1.2 on page 13) seems to have been justified. For more than 90% of the students, us-

ing EiffelStudio was not a pleasant experience (see Figure 21). The students had lots of problems finding out what classes can be used, and what features could be called on them. EiffelStudio cannot be facilitated, but we can give the students more help on how they can properly use this complex environment.

3.4.6 General remarks

One of the assistants suggested to make two kinds of exercise groups, one for the more and one for the less experienced students. This idea is definitely worth a thought, as the student body evaluation in Section 3.2.3 on page 23 showed that there indeed is an enormous gap between the various students attending 'Introduction to Programming'. Section 4.4.2 on page 46 discusses a paper talking exactly about this subject.

Overall, the students did not only like Bertrand Meyer’s presentation style (73%, Figure 22), they were also happy (50%) or even very happy (27%) with the course in general (see Figure 23).

3.5 Try again? - Sure!

In Winter 2004, Bertrand Meyer’s 10 year old idea became reality. ETH Zurich saw the first run of the new 'Introduction to Programming' course, based on approaches like Inverted Curriculum and Outside-In. A big amount of new material had to be designed, from an adopted lecture schedule to exercises, slides, a textbook and last but not least the TRAFFIC library.

This certainly was a big experiment with many a dark horse. Therefore, it is no surprise that not everything did run smooth. The last section pointed out some areas of improvement (most notable the TRAFFIC library), but in general the first try was already a success. In all probability, the second run will be even more so.
4 Paper reading - What others found out...

4.1 Overview

Teaching introductory programming is a controversial issue, and many clever brains did already think about how programming could be taught, how exercises should be designed, how students can be motivated and how a final project should look like. This chapter summarizes some of these interesting papers. First we have a look at exercise design in Section 4.2, before we move on to other topics like 'Teaching with Games' and 'Gender Differences in Programming' in Sections 4.3 and 4.4.

The papers presented here helped me in designing the exercises and thinking about introductory programming in general. In the last section of this chapter (Section 4.5) I draw my personal conclusions from these papers. The summaries following up are meant to help in understanding these conclusion. But they can also be skipped at first.

4.2 Exercise Design

In introductory programming, it is common to hand out smaller or larger assignments. These have to be solved by the student - alone or in groups - and are supposed to give them some practical experience that should complement the (more theoretical) lecture. The exercises should help the students applying their knowledge to new situations [2] and give them a deeper understanding of the subject.

First-rate homework assignments are important for the success of any course. In CS1, this is even more the case, as the lions share of what students really learn comes from the completion of several programming projects [7].

However, as Feldman/Zelenski and Carbone et Al. make clear, this is not always the case with typical CS1 exercises. While 'The Quest for Excellence in Designing CS1/CS2 Assignments' [7] focuses on techniques for engineering exceptional exercises and establishes several prerequisites that must be present in an exercise, 'Principles for Designing Programming Exercises to Minimize Poor Learning Behaviors in Students' [2] and 'Characteristics of Programming Exercises that lead to Poor Learning Tendencies: Part II' [3] identified features in programming exercises that lead to poor learning tendencies.

Following up is a summary of the most important findings of these three papers.

4.2.1 The Quest for Excellence in Designing CS1/CS2 Assignments [7]

In their paper, Feldman and Zelenski focus on one particular assignment where they use a word game called Boogle as a basis for recursion. As their students tackled this exercise with enthusiasm, they tried to figure out why it was such a good assignment. Doing so, they established five prerequisites for effective assignments:
1. The material that an assignment is intended to teach must lie at the heart of the problem it poses. The selection of appropriate examples plays an important role in successful teaching.

2. An assignment must only be challenging with respect to the material it is intended to teach. Students must be able to focus their attention on the challenges of the new material.

3. An assignment must be engaging. In this age of Sega and Nintendo, it takes a lot more for a program to be cool than it used to. Assignments should be designed with a strong audio-visual component and a high degree of interactivity. The best thing is a program that students want to write because they enjoy running it themselves.

4. An assignment must be accessible to all students. It is imperative for all students to have a fair shot at attempting all assignments.

5. The end result of an assignment must be worth the time and effort required to achieve it. When students see the end result of a programming assignment as something especially impressive, useful, or fun - a program they would like to have for themselves - they will approach the project with a heightened sense of interest and motivation. Writing a program that one can be proud of does wonders for a beginning programmer’s sense of accomplishment.

Apart from these five requirements which build a good foundation, Feldman and Zelenski found some other points that help in engineering an exceptional assignment:

- **Focus the assignment.** Clearing away nonessential components allows students to focus their energy on the more interesting aspects of the assignment. Students can be given the code for features of the program that would otherwise be eliminated. Ideally the code only loosely couples with the students work and they use it only through its public interface.

- **Don’t cut corners on materials provided to students.** Students will not be able to fix any deficiencies in a compiled library, thus its functionality must be complete and efficient enough to properly support their work.

- **Don’t add features indiscriminately.** When the basic assignment concept seems a bit too simple, various extensions can be added to spice it up or to provide extra challenges for the more motivated students. The best enhancements are those which complement the main issues of the assignment and help to further the students mastery of the key concepts.

- **Make the assignment easier for students to test and debug.** CSI/CS2 students are still fairly new to programming, and they often are not experienced with the gamut of debugging and testing techniques. To
clarify the goals the students must attain, a compiled sample application can demonstrate the behavior of a working solution.

4.2.2 Principles for Designing Programming Exercises to Minimize Poor Learning Behaviors in Students [2]

In their paper, Carbone, Hurst, Mitchell and Gunstone explore characteristics of programming tasks that affect student learning and understanding in a CS1 course. In a project called Edproj they gathered data that showed that the majority of students’ learning takes place in practical classes and in tutorials. However, it became also clear that many students just get the tasks done, meet the requirements and then stop. Students did not reflect on what they did and did not understand, and lacked strategies for improving their understanding.

During their analysis, they identified three so-called poor learning tendencies (PLTs) that act as barriers to achieving quality learning:

1. Superficial attention. No attempt to actively process the information in order to generate a personal meaning.
2. Impulsive attention. Some parts of a communication are attended to but others are overlooked.
3. Staying stuck. Lack of any strategy to cope with getting stuck except to call for help.

In the following, I summarize the features leading to these PLTs and list ways to improve the tasks.

Superficial attention:
Tasks that require reproducing or modifying code often do not gear the students into discussions to think deeply about the material; students just copy the code without a thought.

• Not always coding. Getting students to diagrammatically present material, including tasks that require tracing code, or answering a series of questions, can be used as alternatives to purely writing code.
• Rewards for understanding not completing.
• Outline a method of attack.

Impulsive attention:
Two key characteristics were found in tasks that lead student to impulsive attention: tasks that do not emphasize the key concept (students are focusing on the wrong aspect of the task) and tasks that contained too many unfamiliar concepts.
• Emphasize the key point. Either through the tasks aim or by giving the students a set of preparation questions. This also helps the student select what material is important and needs attention.

• Provide adequate resources for the introduction of unfamiliar material.

Staying stuck:
Data suggest that there are three points where students can stay stuck in tackling a programming problem. Either the student doesn’t know how to begin tackling a task, or they don’t know how to design a solution to a task in manageable components, or they become stuck at the debugging level. Students lack strategies in designing solutions for fairly large problems. The lecture notes and textbooks usually contain complete programs without strategies to design, implement, and test code in manageable chunks.

• Tactics on how to start with graded helps. Challenge a student first, don’t explain everything.

• Provide useful references and resources.

• Provide guidelines to writing and testing code in manageable chunks and include debugging strategies.

4.2.3 Characteristics of Programming Exercises that lead to Poor Learning Tendencies: Part II [3]
This paper is by the same authors and based on the same project as the paper above. However, in this paper attention is paid to features of programming tasks that led to the following three poor learning behaviors:

1. Non-retrieval. This occurs when no attempt is made to retrieve one’s own existing views and understandings which are relevant to the knowledge being presented.

2. Lack of internal reflective thinking. The learner is not thinking reflectively about the subject content as presented. Each lesson, activity or even instruction is seen isolated from each other.

3. Lack of external reflective thinking. The learner makes no attempt to link the content of the subject with the outside world or other subjects.

Again, following up is a summary of the features that lead to these PLTs and ways to improve the tasks with respect to these PLTs.
Non-retrieval:
Carbone et Al.'s work revealed that students could not remember how to code some programming features studied earlier in the semester. Students did not reflect on what they did and did not understand what they did, as a result they were confused about material that had been covered, and found it difficult to reapply later. Many students adopted the attitude that once a topic was covered it could be shelved and forgotten.

- Familiarity. When a task introduces new material a referral should be made to any earlier work that relates to it.
- Reinforcement by repetition. When designing tasks it is important to have some revision material of earlier work.

Lack of internal reflective thinking:
Students can not see how the parts of one exercise linked together. They are focused on the high demands of coding and not thinking reflectively about the subject content. Most programming activities are designed at the application level rather than the analysis level.

- Tie the work into the "Big Ideas" of the lesson. At the start of each task specify the aims of the task and provide an explanation why a particular topic is being taught.
- Build on previous work. Include exercises that use previous work.
- Extract the links. Write tasks that help the learner identify the links between each lesson.

Lack of external reflective thinking:
Programming activities often provide little in the way to encourage students to link the content of one subject with the outside world or to other subjects.

- Tasks could be designed to include components of other units.
- Real world situations can be introduced, perhaps simulated and studied as part of the task.

4.3 Games for teaching?
There are many papers that discuss the idea of using games or multimedia in CS1 assignments. "Teaching the Nintendo Generation to Program" [10] is an often cited article by Guzdial and Soloway, in which they argue why traditional 'Hello, World!' assignments are described as tedious and dull by students of the Nintendo Generation. In her paper 'Teaching with games: The Minesweeper and Asteroids experience' [1], Katrin Becker points out the value of computer games as motivator. Jimenez-Peris et Al. present the programming projects
they have successfully used in CS1 in 'Adding Breadth to CSI and CS2 Courses through Visual and Interactive Programming Projects' [12]. In 'Engaging Girls with Computers through Software Games' [9], Gorriz and Medina discuss a new point: “Are games also for girls?”

4.3.1 Teaching the Nintendo Generation to Program [10]

“Why are we doing such a poor job at getting and keeping students in computer science?” is what Guzdial and Soloway ask in their article, and this is their suggestion: Computer science educators are using an outdated view of computing and students.

Learning scientists have found over and over again that engaging the students is critical to deep learning. Guzdial and Soloway think that this is achieved best by creating media. They believe that Nintendo generation students will prefer learning about array manipulation where the example results in producing sound as opposed to sorting payroll IDs or doing linear searches for student names.

We have used ‘Hello, World!’ for the past 25 years because text was the medium that was easiest to manipulate with the given technology. Today’s technology can manipulate sound, graphics, and video with the same responsiveness and ease.

In using the media of the students, educators are tacitly saying: “We value you and your ideas.” Guzdial and Soloway are convinced that students will not miss that gesture.

4.3.2 Teaching with games: The Minesweeper and Asteroids experience [11]

Many academics consider the entire games industry to contain little scholarly merit, and they do not believe that computer games are generally the result of serious work or worthy of serious attention.

However, when students are asked, many will say that game playing is what got them interested in computers in the first place. Regardless of where they end up in their careers, many start off with a desire to become games designers.

While having fun is not typically high on the list of teaching goals, its value should not be underestimated. Students who are having fun work harder, longer, and are more apt to expand on what is taught.

Games are fun. This one fact alone sets them apart from the majority of the projects assigned. But games also hold the potential for the integration of almost all of the concepts and techniques taught in a typical CS degree program.

For example, the well-known John Conway creation: The Game of Life (see also 6.3) has withstood the test of time for good reasons. It exercises several useful concepts (like the use of random numbers, 2-D arrays, and functions). It is well within the realm of "do-able" by a first year class with little programming experience, and it is fun.

Unfortunately though, most programming assignments are what most students would classify as boring: mathematical problems, list processing and oth-
ers all succeed in helping students to learn the concepts but few are met with any real enthusiasm and fewer still inspire true creativity.

In her CS1 class, Katrin Becker created an ASCII version of "Minesweeper", and her students were enthusiastic. "Minesweeper" is a game that almost all students have tried. This puts the problem they are to solve in a context with which they are already familiar.

Because first year students often begin with dramatically different backgrounds and abilities, several levels of completion (stages) were described in the assignment specifications. For example, students were provided with a pre-written base class that they could use if they wished. Katrin Becker also tries to address students of superior ability by including a bonus category with each assignment.

4.3.3 Adding Breadth to CSI and CS2 Courses through Visual and Interactive Programming Projects [12]

Jimenez-Peiris, Khuri and Patino-Martinez have the opinion that video game programming is one of the most motivating topics for computer science students. For example, the game Asteroids consists of a collection of interacting objects. Objects like asteroids and spaceships have many similarities that can be exploited with the use of inheritance.

What Jimenez-Peiris et Al. find very important is that instructors provide some background material that alleviates some of the difficulties associated with challenging projects like this.

4.3.4 Engaging Girls with Computers through Software Games [9]

In their paper, Gorriz and Medina have a closer look at what kind of games are needed to make girls interested in computer science. This is what they found:

- Women and girls tend to prefer adventure and games with a narrative.
- Girls prefer collaboration to competition (they do not play games only to win).
- They are bored by repetitive games that require a player to start over again each time he or she loses or dies.
- Girls are more interested in creating than destroying.
- Girls prefer games that require thought and puzzle.
- Girls like to apply problem solving skills rather than test their eye-hand reflexes.
4.4 Other interesting findings

This section summarizes some papers that do not fit in the two categories above, but that I find interesting anyhow.

Janet Carter and Tony Jenkins take a more general look at 'Gender Differences in Programming' [4, 5]. Jenkins together with John Davy thought about how to deal with the wide variety of backgrounds of CS1 students ('Dealing with Diversity in Introductory Programming' [11]). Sindre, Line and Valvag propose one way to solve this problem, by assigning an open project, where students of the same skill form groups and choose a task that is at their level of ambition ('Positive Experiences with an Open Project Assignment in an Introductory Programming Course' [22]). Nagappan et Al. did some research on programming in groups of two in their paper 'Improving the CS1 Experience with Pair Programming' [20]. The last paper discussed is B. Bloom’s paper about taxonomy [6, 15].

4.4.1 Gender Differences in Programming? [4, 5]
Janet Carter and Tony Jenkins found evidence to suggest that men and women may approach learning in different ways. Women are more likely to be interested in learning for its own sake, have a greater fear of failure, and are less likely to cheat. Women also seek help in different ways when learning to program; they are much more likely to approach staff directly, while male students generally prefer impersonal mechanisms.

4.4.2 Dealing with Diversity in Introductory Programming [11]
Students approach Higher Education from an increasingly wide variety of backgrounds, and with an increasingly wide range of skills and expectations. However, often teaching and support methods in place do not meet the needs of this increasingly diverse student body.

Jenkins and Davys experiences at Leeds allowed them to identify four very broad categories:

- Rocket Scientists: already highly proficient programmers.
- Averages: they will pass the module well.
- Strugglers: students who will find the course challenging, but who would be expected to pass.
- Serious Strugglers: students who will find the course extremely difficult, and who will not pass without significant additional support and encouragement.

After attempting to classify the students based on their own assessment, which turned out to be not very reliable, they devised an aptitude test. This classification now allowed implementing separate routes through the module for each group:
The Rocket Scientists were removed from the mainstream teaching. The assumption was that they would learn very little there, but could learn much more by undertaking more advanced work. They were asked to choose a suitable programming project which they would complete over the semester.

For the Averages, there was little need to change from the established approach.

The Strugglers was the group to which most attention was to be devoted. This group followed the same lecture course as the others, but instead of a tutorial they had a supervised session in the laboratory. This session consisted of a programming task covering the same material as the lectures. A number of staff and demonstrators were on hand to provide immediate assistance.

The Serious Strugglers were offered (no student was forced to do anything) an extra tutorial class.

According to the authors, the attempt to classify students has been a success. The module results showed fewer students failing than in previous sessions.

4.4.3 Positive Experiences with an Open Project Assignment in an Introductory Programming Course [22]

In the 'Object-oriented Programming’ course at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway, the assignments are a mixture of individual exercises on a weekly basis, and a somewhat larger project. This project is lasting for five weeks and has to be done by groups of four students.

This larger project was changed from one with fixed requirements set by the staff to an open assignment, where each group made a computer game according to their own preferences.

The project trains the students’ abilities to compose larger programs needing several classes and using all or most of the language concepts and features that the course covers. Moreover, it is supposed to stimulate their creative and collaborative abilities. The project must be big enough to make sense as a group exercise. If it is too small, it will be hard to find a useful breakdown of the solution into modules that separate group members can implement.

Sindre, Line and Valvag realized that a one-size-fits-all project with fixed requirements does embody a lot of problems:

- Too little challenge for more experienced students, who would be able to finish more quickly working alone.
- Too hard for less experienced students.
- Widespread copying of bigger or smaller parts of the solution between groups.
- Low inspiration. All groups are doing the teacher’s thing. So the students feel little ownership for their work.
• It is hard to define an assignment that makes a really useful program and is yet accomplishable for four freshmen in five weeks.

Thus, they proposed three changes:

**New principles for the grouping of the students:**
Preferably, all 4 members of a team should be at approximately the same skill level. Thus, the less clever, teamed with other students on the same level, had to struggle together instead of simply leaving the work to other group members.

**Computer games were the chosen product, instead of more boring tasks of previous years.**
Arguments are:

• Game programming exercises are more fun, thus motivating students more easily.
• Games are programs that the students may use and improve after the deadline.
• A game can proudly be shown to friends and relatives.
• The students themselves are plausible end-users and thus able to determine the quality of their programs.

**The assignment was made open instead of the fixed requirements of previous years:** Make any kind of computer game.
This has the following gains:

• A huge reduction of the copying problem.
• More inspired students.
• It is easier for each group to determine its own level of ambitions.

The student response has been overwhelmingly positive. Apparently, both more and less experienced students worked harder with this project than with previous ones. The exam results also indicated that the students had a better learning experience. The open project stimulates inventiveness and curiosity.

4.4.4 Improving the CS1 Experience with Pair Programming [20]
Various research indicates that pair programmers produce higher quality code in essentially half the time taken by solo programmers. An experiment by Nagappan et Al. shows that student pair programmers were more self-sufficient and generally perform better on projects and exams.

It is also known that in industry, software developers generally spend 30% of their time working alone, 50% of their time working with one other person,
and 20% of their time working with two or more people. Studies also show that cooperative and collaborative pedagogies are beneficial for students.

Nagappan's study provides strong results of the following findings:

- Pair programming helps in the retention of more students in the introductory computer science stream.
- Students in paired labs have a more positive attitude towards working in collaborative environments.
- Students in paired labs display more active participation in their learning.
- Paired student questions display higher order thinking such as application, synthesis, and evaluation (see next section for more on these terms).

4.4.5 Blooms taxonomy [6, 15, 8]

B. Bloom established a hierarchy of educational objectives which divides cognitive objectives ranging from the simplest behavior to the most complex. This hierarchy is generally referred to as Bloom’s Taxonomy and classifies questions in 6 levels, from K1 (the easiest) to K6 (the most difficult). Following up is an overview of these levels. For each level, there is first a short description followed by some typical words used in these types of assignments.

If the discussed level is present in one of our new exercises (which can be found in Appendix C), a reference to the respective assignment is made.

1. Knowledge (K1)

Knowledge is defined as the remembering of previously learned material. This may involve the recall of a wide range of material, from specific facts to complete theories, but all that is required is the bringing to mind of the appropriate information. Knowledge represents the lowest level of learning outcomes in the cognitive domain.

Words that apply:
- define
- describe
- identify
- label

Example:
- No example in our exercises

2. Comprehension (K2)

Comprehension is defined as the ability to grasp the meaning of material. This may be shown by translating material from one form to another
(words to numbers), by interpreting material (explaining or summarizing), and by estimating future trends (predicting consequences or effects). These learning outcomes go one step beyond the simple remembering of material, and represent the lowest level of understanding.

Words that apply:

- classify
- discuss
- estimate
- give examples
- restate (in own words)
- summarize

Example:

- Exercise 1 in Assignment 3 (C.3)

3. **Application (K3)**

Application refers to the ability to use learned material in new and concrete situations. This may include the application of such things as rules, methods, concepts, principles, laws, and theories. Learning outcomes in this area require a higher level of understanding than those under comprehension.

Words that apply:

- assess
- compute
- develop
- implement
- relate
- solve

Example:

- Exercise 3 in Assignment 3 (C.3)

4. **Analysis (K4)**

Analysis refers to the ability to break down material into its component parts so that its organizational structure may be understood. This may include the identification of parts, analysis of the relationship between parts, and recognition of the organizational principles involved. Learning outcomes here represent a higher intellectual level than comprehension and application because they require an understanding of both the content and the structural form of the material.

Words that apply:
• break down
• illustrate
• prioritize
• subdivide

Example:

• Exercise 2 in Assignment 6 (C.6)

5. **Synthesis (K5)**

Synthesis refers to the ability to put parts together to form a new whole. This may involve the production of a unique communication, a plan of operations (research proposal), or a set of abstract relations (scheme for classifying information). Learning outcomes in this area stress creative behaviors, with major emphasis on the formulation of new patterns or structure.

Words that apply:

• adapt
• compare
• create
• model
• reorganize

Example:

• Exercise 1 in Assignment 4 (C.4)

6. **Evaluation (K6)**

Evaluation is concerned with the ability to judge the value of material for a given purpose. The judgments are to be based on definite criteria. These may be internal criteria (organization) or external criteria (relevance to the purpose) and the student may determine the criteria or be given them. Learning outcomes in this area are highest in the cognitive hierarchy because they contain elements of all the other categories, plus conscious value judgments based on clearly defined criteria.

Words that apply:

• criticize
• interpret
• judge
• justify
Example:

- No example in our exercises

Using Blooms classification is important in designing exercises: “Without taxation, about 75% of all exercises only ask for reproduction of studied material” [8].

4.5 Conclusion: Our guidelines

From all this various papers, we want to extract some properties that could prove useful in our course and our exercises. The ideas are grouped into four sections:

- Guidelines for the exercises
- Ideas for the project
- Why games might be useful
- General remarks

These guidelines are not just theory - we directly applied them to our exercises. All the new exercises can be found in Appendix C. For each guideline, we make a forward reference to an appropriate exercise in the appendix.

4.5.1 Concerning the exercises

- An assignment must only be challenging with respect to the material it is intended to teach. [7]
- Clearing away nonessential components allows students to focus their energy on the more interesting aspects of the assignment. [7]

Using the Outside-In approach and the TRAFFIC library is very useful to abstract ‘unnecessary’ details. A good example is the loop exercise in Assignment 5 (Appendix C.5). See also Section 6.2.2 for more on this exercise.

- To clarify the goals the students must attain, a compiled sample application can demonstrate the behavior of a working solution. [7]

In some exercises, like in Assignment 4 (Appendix C.4), the students get the screenshot of a possible solution as an example. If the question could raise ambiguity problems, an example solution (like in Assignment 6 in Appendix C.6) is given.

- Not always coding. Get students to diagrammatically present material, include tasks that require tracing code. [2]
About one third of the exercises do not involve direct coding. There are multiple choice questions (Appendix C.7), tasks where the students have to write something in their own words (Appendix C.3), and one exercise where the students have to draw the run-time structure of an object (Appendix C.6). Above all, two out of the ten assignments are classroom exercises, which involve no direct coding at all.

- Emphasize the key point through the tasks aim. [2]

In every exercise, we first state what the goals are.

- Provide adequate resources for the introduction of unfamiliar material. [2]

When new concepts are used for the first time, like input/output in Assignment 4 (Appendix C.4), clear information is given on how to use it.

- Provide guidelines to writing and testing code. [2]

Wherever students might encounter problems, the exercises has some hints ready giving the students ideas on how to tackle the problem (e.g. in Assignment 2 in Appendix C.2).

- Include debugging strategies to help the students when getting stuck. [2]

Already in Assignment 3 (Appendix C.3), the students are introduced to Eiffel-Studio’s debugging mode.

- When a task introduces new material a referral should be made to any earlier work that relates to it. [2]

Assignment 6 (Appendix C.6) makes use of debugging techniques, which are explained already in Assignment 3 (Appendix C.3).

- Without taxation, about 75% of all exercises only ask for reproduction of studied material. [8]

Every exercise is categorized in one of the six levels of Bloom’s Taxonomy. In each assignment, there are exercises of different levels. Over all assignments, exercise of the levels between K2 to K5 are present (see also Chapter 6).

4.5.2 Project ideas

- A one-size-fits-all project with fixed requirements does embody problems like low inspiration and widespread copying of parts of the solution between groups. [22]

- On open assignment reduces the copying problem and makes it easier for each group to determine its own level of ambitions. [22]
In the final project (Appendix C.9), there are three possible options, all very open themselves: Extend FLAT_HUNT, implement Conway’s Game of Life, or pursue an own idea.

- Members of a team should be at approximately the same skill level. [22]

The project exercise 2004 asks the students to form uniform groups concerning their programming exercise, in contrast to the project in 2003 which asked for mixed groups (Section 3.1.6). See also Section 6.3 on page 96 and Appendix C.9 for more information about the final project.

4.5.3 Games are fun!

- Game programming exercises are more fun, thus motivating students more easily. [22]
- The students themselves are plausible end-users. [22]
- Game playing is what students got interested in computers. [1]
- Video game programming is one of the most motivating topics for computer science students. [12]

The traditional TRAFFIC library and the examples in the ’Touch of Class’ book are mostly dealing with a Geographical Information System. For the exercises, a new application called FLAT_HUNT was developed. FLAT_HUNT is a multiplayer game similar to the board-game 'Scotland Yard'.

- Use a game that almost all students have tried. This puts the problem they are to solve in a context with which they are already familiar. [1]

As 'Scotland Yard' is a very famous game, most of the students will know it and should be able to play it straight away.

- Girls prefer collaborative games and games that require thought. [9]

FLAT_HUNT is a round-based multiplayer game. Several students have to work hand-in-hand to find the estate agent which rents them a flat in Zurich. Lots of strategies can be used to achieve this, and it is far away from violent first-person shooter games.

4.5.4 General ideas

- The selection of appropriate examples play an important role in successful teaching. [7]

As lots of the exercises are based on the TRAFFIC library, this library itself is already an appropriate example. For some other exercises, examples are given directly in the exercise text, like in Assignment 3 (Appendix C.3).
• The majority of students’ learning takes place in practical classes. [2]

• Engaging the students is critical to deep learning. This is achieved best by using sound and graphics. [10]

• Assignments should be designed with a strong audio-visual component and a high degree of interactivity. [7]

As most of the exercises build on the improved TRAFFIC library, the ‘Nintendo Generation’ should be happy. Again, the loop exercise in Assignment 5 (Appendix C.5) is a typical example.

• Classify students based on an aptitude test and implement separate routes through the course for each group. [11]

This idea was also suggested by one of the assistants. Unlike Jenkins and Davys, I would not advocate four groups, but I think that making two groups (less/not experienced and more experienced) would already have a big benefit. The less experienced students (at least 22%, see Section 3.2.3 on page 23) could have a longer exercise session (3 hours) and maybe an additional 1 hour lab session, supervised by an assistant. For the more experienced students, 2 hours exercise session should be enough.

• Pair programmers generally perform better on projects and exams. [20]

This also sounds like a good idea. Maybe some of the exercises (especially those towards the end of the semester) could be solved in groups of two.
5 TRAFFIC 2: The Redesign

5.1 Overview

“TRAFFIC 1.0 was very slow, unstable, big, complex and missing a good documentation.” This was the hard conclusion drawn in Section 3.4.2 on page 36. Lot of things had to change - and they did change. Michela Pedroni worked on the redesign of TRAFFIC, which is shortly described in Section 5.2. To make TRAFFIC more interesting, I implemented a new application called FLAT_HUNT. FLAT_HUNT is described in Section 5.3.

Section 5.4 contains the documentation the students get when they start using FLAT_HUNT. That documentation is not only useful for the students, but also for the reader of this thesis. Especially the second part “Developer Guide” contains an interesting overview of how TRAFFIC and FLAT_HUNT work. Section 5.5 contains a summary of this chapter.

5.2 The new library

5.2.1 What was wrong?

The library TRAFFIC is an integral part of the Outside-In approach (see Chapter 2). The idea is that the students don’t start from scratch, but learn programming by first using a library as a consumer, and then - by gradually starting to modify and extend this library - become a producer. So there is no way around this library. The students have to work with it, whether they like it or not.

And unfortunately, last year they did not like it. They complained about the “slow speed of TRAFFIC” and that it was “too big and intransparent” (see Section 3.3.2). Also the assistants were not happy. They asked for a complete redesign of TRAFFIC and a better documentation including BON-notation.

5.2.2 Overview of TRAFFIC 2

This complete redesign did happen - TRAFFIC was reimplemented from scratch. TRAFFIC itself is now a generic GIS\(^4\) library which is used by two applications:

FLAT_HUNT, which is described in the next section, and a system called TOUCH_APPLICATION.

The TOUCH_APPLICATION is the equivalent to last year’s TRAFFIC and features the examples from the ‘Touch of Class’ textbook. See Figure 24 for a screenshot.

\(^4\)Geographic Information System
The new TRAFFIC library is more clearly presented thanks to a better clustering. This clustering is described in Chapter 7 of the FLAT_HUNT User & Developer Documentation (see Section 5.4 on page 60). TRAFFIC 2 is more stable and faster, however, as we wanted to keep the XML-maps because they are very flexible, opening a big map still takes some time.

TRAFFIC 2 has fewer classes and is less complex, yet still generic and extensible. There is one main entry point, which is the class CITY. This is also described in the User & Developer Documentation. That documentation includes a BON-diagram of some TRAFFIC clusters to help the students understand the library.

5.3 FLAT_HUNT - The game

5.3.1 Background

When we first exchanged ideas about TRAFFIC’s redesign, we all agreed that it should be faster and easier understandable. However, something else was missing, too: Fun!

After reading papers like 'Teaching the Nintendo Generation to Program' [10] and 'Teaching with games: The Minesweeper and Asteroids experience' [1] (see Section 4.3 for summaries) we decided that we should make use of the TRAFFIC library in a game. The following statements from Section 4.5.3 support this idea:

- Game programming exercises are more fun, thus motivating students more easily. [22]
- Game playing is what students got interested in computers. [1]
- The students themselves are plausible end-users. [22]
But what kind of game? For sure, it had to be a game that would work well with the TRAFFIC library. So it had to be a game that used some kind of public transport system.

We tried to also take care of the following two suggestions by Katrin Becker and Gorriz/Medina (summaries again in Section 4.3):

- Use a game that almost all students have tried. This puts the problem they are to solve in a context with which they are already familiar. [1]
- Girls prefer collaborative games and games that require thought. [9]

5.3.2 The idea

We tried games like 'Railroad Tycoon' until we found what we were looking for: Scotland Yard. Scotland Yard is a twenty years old board game with the following description:

“One player takes the part of Mr. X and slinks around the city of London, leaving clues for the others who are the Scotland Yard detective team. Was the elusive Mr. X last seen at Buckingham Palace or Madame Tussaud's? Did he travel by bus or taxi? Strategy skills and teamwork will help Scotland Yard players trap Mr. X and win the game. If, on the other hand, Mr. X manages to escape capture, he wins!"

Figure 25 shows the London map on which Scotland Yard is played.

Figure 25: Scotland Yard board game

Scotland Yard was exactly what we needed. A collaborative strategy game that plays on the map of a city. However, we didn’t want to make an exact copy. For instance, why should it be the city of London? We decided to base our game on the map of Zurich, which features an extensive public transport system and is well known to all our students.
And why does it have to be a detective story? Our story does also involve an agent, but it is an estate agent that leases flats in Zurich. The equivalent to the Scotland Yard team are the flat hunters, fresh students at ETH who try to find a flat in Zurich.

The complete story can be found in Chapter 2 of the *FLAT_HUNT Documentation* (see Section 5.4). The Scotland Yard board game itself is very well known especially in German speaking areas. Most of the students will already be familiar with the rules and can play FLAT_HUNT straight away. For those not knowing Scotland Yard, Chapter 3 in the *Documentation* explains how FLAT_HUNT is meant to be played.

### 5.3.3 The looks

FLAT_HUNT (see Figure 26) looks similar to the TOUCH_APPLICATION, which is no surprise, as they are both based on the TRAFFIC library. One difference is the map, which is not Paris, but Zurich. The red circle indicates the current player, while the blue highlighted places mark the locations the player can move to. At the bottom of the screen, the info boxes are located. They contain statistics about the current player and the game state. All the features are described in Chapter 3 of the *FLAT_HUNT Documentation* in Section 5.4.

![Figure 26: FLAT_HUNT screenshot](image)

### 5.3.4 Behind the scenes

Most of the design aspects are explained in Part II (Chapters 5 and 6) of the *FLAT_HUNT Documentation*. FLAT_HUNT is a rather simple application,
and the students should not find it difficult to get used to it. Despite its simplicity, FLAT_HUNT is a good basis for many exercises. It contains practical examples of inheritance and polymorphism as well as client-supplier relationship.

FLAT_HUNT is not one of those superficial O-O introduction programs. If it was not programmed using the object-oriented paradigm, one would easily get lost.

The FLAT_HUNT application also shows the power of reuse. By using libraries, the FLAT_HUNT system makes use of 1153 classes and counts more than 200'000 lines of code.

5.4 FLAT_HUNT User & Developer Documentation

The following pages contain a copy of the FLAT_HUNT User & Developer Documentation. This documentation is handed out to the students in the second week, when they start with Assignment 2 (see Appendix C.2). It should help them understanding FLAT_HUNT, from the user’s and the developer’s perspective. The first part is the User Guide. Part II is for the developers point of view. The writing style of the whole documentation is a bit less formal, but this is on purpose.

Even though the documentation is primarily aimed at the students, especially Part II includes a lot of information about TRAFFIC and FLAT_HUNT, like a description of the clustering and of some of the classes. As these topics are not covered in the sections above, the FLAT_HUNT User & Developer Documentation forms an integral part of this thesis.
<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part I User Guide 2</td>
</tr>
<tr>
<td>Introduction 3</td>
</tr>
<tr>
<td>About this document 4</td>
</tr>
<tr>
<td>Legal stuff 4</td>
</tr>
<tr>
<td>What's the story? 5</td>
</tr>
<tr>
<td>How to play? 6</td>
</tr>
<tr>
<td>General rules 6</td>
</tr>
<tr>
<td>Game modes 7</td>
</tr>
<tr>
<td>Other things to do... 8</td>
</tr>
<tr>
<td>Part II Developer Guide 10</td>
</tr>
<tr>
<td>How it works 11</td>
</tr>
<tr>
<td>Overview 11</td>
</tr>
<tr>
<td>FLAT_HUNT 12</td>
</tr>
<tr>
<td>Cluster Controller 13</td>
</tr>
<tr>
<td>Cluster Model 13</td>
</tr>
<tr>
<td>Cluster View 14</td>
</tr>
<tr>
<td>The states of the game 15</td>
</tr>
<tr>
<td>Game loop 15</td>
</tr>
<tr>
<td>TRAFFIC - the library 17</td>
</tr>
<tr>
<td>Cluster Graph 17</td>
</tr>
<tr>
<td>Cluster City 17</td>
</tr>
<tr>
<td>Cluster Visualization 18</td>
</tr>
<tr>
<td>Guided &quot;walk-through&quot; 19</td>
</tr>
</tbody>
</table>
Part I
User Guide
Introduction

Welcome to FLAT_HUNT!

FLAT_HUNT is a simple adaptation of the well-known board game “Scotland Yard” (see Figure 1). Instead of some agents hunting Mr. X all around London, it is about a group of students starting off at ETH Zurich. To make their student life a bit more pleasant, they are desperately trying to find a flat in this little big city. But to get a flat, they must first meet the estate agent, who is running all around Zurich showing his flats to other people…

Figure 1 FLAT_HUNT screenshot
About this document

This first part is the FLAT_HUNT User Guide. The next chapter goes a bit more into the story of FLAT_HUNT. Chapter 3 tells you how this game is meant to be played.

Part II (Chapters 4 to 8) explains in detail how the game works, describes the classes used and highlights some of the features.

Words in all UPPERCASE usually refer to classes of FLAT_HUNT or TRAFFIC. Words in small letters and in italic describe features of classes. Cluster names are starting with uppercase in italic. And this is actual Eiffel code.

Legal stuff

The South Park icons used in FLAT_HUNT are courtesy of Trey Parker & Matt Stone.

The idea to FLAT_HUNT is based on Scotland Yard, a board game courtesy of Ravensburger.
Chapter 2

What’s the story?

As the title suggests (and the introduction mentions), it is all about finding a flat in Zurich…

However, this is not so easy… There is this guy, the estate agent (Figure 2 a), who is renting flats. The problem is that he is always busy showing flats to other customers (e.g., other students), and even in his office they don’t really always know where exactly he is. The only thing they know is what kind of transport he is moving around with. This is because the estate agent is taking part in a new VBZ project called “Customer tracking”.

In collaboration with ETH, they equipped some volunteers with transponders. These transponders gather information like current position and type of transport, and send it in real-time to the office. However, for privacy reasons, only the type of transport can be accessed all the time.

Once in a while, the estate agent calls his office to tell the secretary which flat he is currently visiting. So sometimes, the people there in the office can tell “you” where to look for the guy… “You” meaning yourself (Figure 2 b) and your friends (Figure 2 c), the guys you want to share the flat with…

1 Verkehrsbetriebe Zürich (Zurich Public Transport)
Chapter 3

How to play?

Playing FLAT_HUNT is not very difficult, especially for those that know the game "Scotland Yard"…

This chapter is structured in three sections. The first section describes the general rules of the game. Then the different modes are explained more precisely. The last section is about other stuff like how to open a new map or how to start a new game.

General rules

The game lasts for at most 23 rounds. In these 23 rounds, the flat hunters try to find the estate agent, while he tries to avoid them (this is because he would rather rent the flats to elderly couples, since presumably they make fewer parties in the middle of the night…).

In each round, every player can make one move on the public transport system. The estate agent is the first, then it’s the hunters turn. One move is either

- one or two stops by tram (coloured lines),
- one stop by train (thick black lines),
- or one stop by bus (thin dashed blue lines).

A move with a certain transport can only be made if one has still enough tickets (see Figure 3), if there is a connection (obviously), and if there is no other player at that destination (and in the case of tram lines, if there is no hunter in between).

![Figure 3 Tickets left for Hunter 1](image)

2 By far the easiest way to move around in Zurich (except for bicycle)
Attention: If you are at a bus-only stop, and you run out of bus tickets, you will get stuck there forever, so be careful…

The possible places one can move to have a blue highlighted outline (see Figure 4). To make a move, just click on one of those highlighted places. The red circle centers on the player whose turn it is, and in the information panel at the bottom, the current player's tab is highlighted and you can see the player's image.

Figure 4 Highlighting of possible moves

Game over?
The game is over when

a) the hunters could not find the estate agent within 23 rounds,
b) one flat hunter moves onto the place where the agent currently is,
c) or the hunters encircle the estate agent so that he cannot move anymore.

In case a), the winner is the estate agent (he does not have to rent his flat to students), whereas in b) and c) it is the hunters that win, as they get to meet the estate agent on time and thus manage to find a flat.

Game modes
There are four modes to play FLAT_HUNT: Hunt, Escape, Versus, and Demo. Depending on the mode, zero (Versus), one (Hunt/Escape) or two (Demo) parts are taken over by the computer.

Hunt
This is probably the most typical situation; the player tries to find the agent, which is played by the computer. Thus, the player only knows about every fifth move where the agent just was… The exact route of the agent can be seen at the bottom left (see Figure 5). The agent shows himself only in rounds number 1, 3, 8, 13, 18, and 23.
**Escape**
This is just the opposite: The agent is played by you, and the hunters are played by the computer. The hunters always move as close in your direction as possible, as they somehow manage to decode your transponder signal, and thus always know your precise location (so much for privacy...). You just have to try to avoid them as long as possible...

**Versus**
This is the multiplayer mode. One of the players is the agent; the other plays all the hunters. While the player of the agent is making a move, the player of the hunters is supposed to look away...

**Demo**
This mode is more or less the opposite of the buzzword “interactive”, but is about as entertaining as watching fish in an aquarium. The computer is playing against himself, trying to catch the agent as fast as possible.

**Other things to do...**

**Start new game**
At any time, a new game can be started. Just select File ➔ Start... (see Figure 6) or press [Enter] and the game dialog (see Figure 7) will appear. There you can select the game mode and specify the number of hunters. You can also use this dialog to alternate between the small and the big map of Zurich.

![Figure 6 File menu](image-url)
The FLAT_HUNT game comes with two maps: “zurich_big_city.xml” and “zurich_little_city.xml”. By using the File menu (see Figure 6) or [Ctrl-O] the map can be changed.

Buttons and keyboard short-cuts
Figure 8 shows the button panel in FLAT_HUNT. From left to right, they have the following function:

- Zoom in and out (you can also use your mouse wheel for this).
- Selection mode: Normal mode to play the game.
- Move mode: If you want to move the map (can also be done by using the right mouse button). Go back to “selection mode” afterwards.
- Center map on current player.
- Switch the place names on/off.

If you press [Space] any time during the game, the game will stop after the next players move until you press [Space] again.
Part II
Developer Guide
Chapter 4

How it works

This and the following chapter should help you to understand how exactly FLAT_HUNT works. First you will get an overview of the whole system’s organization, and then we will have a look at some important classes in more detail (Chapter 5).

The different states the FLAT_HUNT game can be in are described in Chapter 6. In Chapter 7 we will have a short glance at the TRAFFIC library, which is also used in the TOUCH_APPLICATION. This developer guide finishes with a “walk-through” of a typical game in Chapter 8.

Overview

When opening FLAT_HUNT in EiffelStudio, the cluster view in the bottom right corner shows three clusters: Dependencies, Flat_hunt and Traffic. Figure 9 shows them already expanded.

Cluster Flat_hunt is described in Chapter 5 and 6, and Chapter 7 introduces the Traffic cluster.

The cluster Dependencies contains several libraries that are used in FLAT_HUNT, like Time to find out how many milliseconds have passed, Gobo which includes data structures like DS_LINKED_LIST or Vision2 on which the graphical user interface (GUI) of FLAT_HUNT is based.
To remove complexity, FLAT_HUNT is structured in three clusters (see Figure 10): Model, View and Controller. In each cluster, there are several classes, and sometimes there are subclusters. Like in any object-oriented system, these classes are connected. This is symbolized by the red and blue arrows.

Red arrows describe an inheritance relationship. For example, class BOT (7) inherits from class BRAIN (6). This has the effect that class BOT can do the same things as class BRAIN, because it inherits all the features from class BRAIN. Usually, the class that inherits (BOT) can do some additional things which are not defined in the parent class (BRAIN).

Blue arrows stand for a client-supplier relationship. The class the arrow points to is the supplier, the other class is the client that makes use of the supplier class. In FLAT_HUNT, class GAME (3) is a client of class PLAYER (4). Using the client-supplier relationship allows the client to access features on the supplier.

Figure 10 FLAT_HUNT class diagram
Cluster Controller

Cluster Controller is the fundamental cluster in FLAT_HUNT. Here are the classes that “control” the action. They make sure that the displayer classes in cluster View display the proper information, which they get from the Model classes. For example, feature prepare in class GAME controls the display update by calling

caller_player.display.display_after_move.

- **START (1):** This is the entry point to FLAT_HUNT. Almost all features are inherited from class MAIN_CONTROLLER. There is only one feature called start which is indeed the start of the whole system. When you run FLAT_HUNT, this is where it all begins.

- **MAIN_CONTROLLER (2):** The MAIN_CONTROLLER is (as the name suggests) responsible for many things. It provides access to the MAIN_WINDOW, to class GAME (3) and to the whole TRAFFIC library, which is responsible for the visualization of the city (not visible in Figure 10, see Chapter 7).

- **GAME (3):** Class GAME features the FLAT_HUNTER logic. It knows which players turn it is. And as it is an heir to class STATE_CONSTANTS, it also knows the game state (see Chapter 6).

Cluster Model

In the cluster Model, there are two important parent classes: Class PLAYER and class BRAIN. PLAYER is the parent of FLAT_HUNTER and ESTATE_AGENT, and BRAIN is the parent of HUMAN and BOT. These Model classes describe the internal representation of “real world” objects. Here’s a description of four of these classes:

- **PLAYER (4):** Class PLAYER knows the basic things one needs to know about a player of FLAT_HUNT, like how many tickets are left. It features the commands play and move (see Chapter 6) and has either a HUMAN or a BOT brain.

- **ESTATE_AGENT (5):** This is one of the two heirs of class PLAYER. It has some additional information that is special for an estate agent player like knowing where he last showed himself.

- **BRAIN (6):** Class BRAIN includes the intelligence to choose the next move.
• **BOT (7):** BOT is one of the two intelligences to choose a move. In contrast to class HUMAN, this is an artificial intelligence (AI).

![Cluster View](image)

**Cluster View**

This cluster's job is to make sure that the user sees what is going on. It includes all windows and dialog boxes, as well as displayers for the game players and the game itself:

• **PLAYER_DISPLAYER (8):** This class displays the player on the map and prints the amount of tickets left. PLAYER_DISPLAYER knows this information because of the client-supplier relationship with class PLAYER. PLAYER_DISPLAYER also features the `animate_defeat` animation.

• **GAME_DISPLAYER (9):** … displays the game statistics, like whose turn it is and whether it is game over or not. This class does not exist at the beginning. In assignment 4 it will be your task to create this class.

• **MAIN_WINDOW (10):** MAIN_WINDOW includes not only the canvas on which the map is painted, but also all the info_boxes, the buttons and the menu.
Every game has at least two states: playing and game over. FLAT_HUNT has six states in total; three playing states and three game over states (see Figure 14). These game states are defined in class STATE_CONSTANTS:

Agent_stuck, Agent_caught, Agent_escapes, Prepare_state, Play_state, Move_state: INTEGER is unique

**Game loop**

For each player in each round in FLAT_HUNT, the game goes through the following states: Prepare, Play and Move. In addition, there are three game over states: Agent_stuck, Agent_caught and Agent_escapes.

- **Prepare**: If the game is in state Prepare, the current player gets a red circle and the possible moves are calculated and displayed. If the current player is the estate agent, and there are no possible
moves, the agent is stuck and thus the game is over (state `Agent_stuck`, 1). If that is not the case, the game goes in state Play (2).

- **Play**: In state Play, if the current player is played by a human, the game waits until the human player clicks on one of the places that are highlighted. If the player is controlled by an artificial intelligence, then the best of the possible moves is calculated. The game then goes in state Move (3).

- **Move**: Move does perform the move selected in state Play. After the move, the game checks if the player hits the place of the estate agent. If that is the case, the game goes into state `Agent_caught` (4). If the agent did not get caught, and the round number is greater than 23, then the estate agent is the winner and the game goes into state `Agent_escaped` (5). If none of the above is the case, then it’s the next player’s turn and the game loop starts again in state Prepare (6, see also example below).

In the classes `MAIN_CONTROLLER`, `GAME` and `PLAYER`, you can find the features `prepare`, `play` and `move` that deal with these game states. As an example, let’s have a look at feature `move` in class `GAME`:

```plaintext
move is -- Make the chosen move.
do current_player.move
  if current_player.location = estate_agent.location and current_player /= estate_agent then
    state := Agent_caught
  else
    state := Prepare_state
    next_turn
  end
end
```

In the classes `MAIN_CONTROLLER`, `GAME` and `PLAYER`, you can find the features `prepare`, `play` and `move` that deal with these game states. As an example, let’s have a look at feature `move` in class `GAME`:

```plaintext
move is -- Make the chosen move.
do current_player.move
  if current_player.location = estate_agent.location and current_player /= estate_agent then
    state := Agent_caught
  else
    state := Prepare_state
    next_turn
  end
end
```
TRAFFIC – the library

Like the TOUCH_APPLICATION, FLAT_HUNT also makes use of TRAFFIC, a library of classes especially designed for ‘Introduction to programming’. This chapter will give a short overview of the TRAFFIC library.

TRAFFIC consists of three clusters: Graph, City and Visualization.

Cluster Graph

This small cluster features three classes: GRAPH, NODE and EDGE. They build the basis for the city’s transport network, as the places in the city are based on class NODE and the transportation links are implemented using EDGEs. This is achieved by inheriting from class GRAPH in class NETWORK:

```plaintext
class NETWORK
inherit GRAPH [PLACE, LINK]
```

Cluster City

The FLAT_HUNT application uses this cluster to display a map of Zurich. Class CITY is the main entry point if access to places or lines is needed, because it contains lists of all the PLACES, LINES and LINKs:

```plaintext
links: DS_LINKED_LIST [LINK]
-- Container in which all links of the city are stored
```

Through the feature transport_network of type NETWORK (see cluster Graph above), feature calculate_possible_moves in class GAME can find out the next possible moves:

```plaintext
outgoing_links := city.transport_network.outgoing_links (a_player.location)
```

Figure 15 shows the important classes in cluster City including all inheritance relations (red arrows).
Cluster City includes a subcluster called Input. This cluster contains classes that help reading in the map of the city, which is in XML format. Below you can see some lines of zurich_little_city.xml:

```xml
<city name="Zurich (little)">
  <places>
    <place name="Bahnhof Stadelhofen"/>
    <place name="Bellevue"/>
    <place name="Boersenstrasse"/>
    <place name="Buerkliplatz"/>
  </places>
  <lines>
    <line name="11" type="tram">
      <line_displayer thickness="4">
        <color red="0" green="154" blue="0" type="standard"/>
        <color red="10" green="184" blue="20" type="highlight"/>
      </line_displayer>
      <link from="Bahnhof Stadelhofen" to="Bellevue" direction="undirected">
        <point x="910" y="495"/>
        <point x="843" y="495"/>
      </link>
      <link from="Bellevue" to="Buerkliplatz" direction="undirected">
        <point x="843" y="495"/>
        <point x="750" y="495"/>
      </link>
      <link from="Buerkliplatz" to="Boersenstrasse" direction="undirected">
        <point x="750" y="495"/>
        <point x="735" y="495"/>
        <point x="710" y="520"/>
      </link>
    </line>
  </lines>
</city>
```

As you can see, the elements places, lines and links directly relate to the classes PLACE, LINE and LINK in cluster CITY.

**Cluster Visualization**

This cluster contains mostly DISPLAYER classes like CITY_DISPLAYER or PLACE_DISPLAYER. They help in displaying the city's objects. The subcluster Graphics features lots of DRAWABLE classes (like DRAWABLE_ROUNDED_RECTANGLE which is used in class PLACE_DISPLAYER) as well as the CANVAS class on which everything is drawn.
Guided “walk-through”?

What happens when you start FLAT_HUNT? In this last chapter we will go step-by-step through a typical FLAT_HUNT game. However, because there are lots of details involved, we concentrate on the more important steps…

1. A call to `start_game` in `MAIN_CONTROLLER` creates a game of the proper gaming mode by calling `game.make`.
2. `make` in class `GAME` creates the players using class `PLAYER_FACTORY` and sets the game state to `Prepare`.
3. In class `PLAYER_FACTORY`, for example the estate agent is created using `estate_agent.make` in feature `build_players`.
4. This creates a HUMAN or BOT brain depending on the value of `bot_estate_agent`.
5. Back to class `MAIN_CONTROLLER`: Feature `idle_action` gets called whenever nothing is going on, i.e. now. `idle_action` checks whether the game is in one of the three game loop states, and calls the corresponding feature in class `MAIN_CONTROLLER`. In the first run, this is `prepare`…
6. …which centers the city map on `game.current_player` and then calls `game.prepare`.
7. `prepare` of class `GAME` first calculates the estate agents possible moves (see also Section “Cluster City” in Chapter 7). If there are no possible moves (`current_player.possible_moves.is_empty`) then it’s either the next player’s turn or the state is set to `Agent_stuck`. Otherwise it’s state := `Play_state`.
8. With that, the call to `prepare` (Step 5) comes to an end and control goes back to feature `idle_action` of class `MAIN_CONTROLLER`. According to the present state, `idle_action` will now call `play` which then calls `game.play`.
9. This calls `current_player.play(place)`, where `place` is the last place the user clicked on. `place` is then passed on to class `BRAIN`.
10. `choose_move` in class `PLAYER` is deferred, which means that `choose_move` of class `ESTATE_AGENT` or `FLAT_HUNTER` gets called, depending on whether the current player is a hunter or an agent.
11. `FLAT_HUNTER`s `choose_move` calls `choose_estate_agent_move` on class `BRAIN`. 
12. Again, this is a deferred feature. Let’s assume that the agent is played by a human player. The agent’s move is chosen in the brain of class HUMAN (which inherits from BRAIN) which checks whether the place clicked on (see Step 9) is one of the destinations of the possible moves (see Step 7).


14. In class GAME, the state is now changed to Move_state.

15. Feature idle_action in class MAIN_CONTROLLER: After play, if the player is a BOT, sleep_and_process takes a short break…

16. Next comes move which calls game.move which in turn calls current_player.move.

17. location := next_move.other_end (location) in class PLAYER sets the player to the new position.

18. Back to move in class GAME: The state is either set back to Prepare_state and it’s the next player’s turn (next_turn) or the state is set to Agent_caught.

19. The last if in idle_action tests whether the game is in one of the game over states via game.is_game_over. Class GAME inherits this feature from class STATE_CONSTANTS.

20. If the above test yields True, end_game of class MAIN_CONTROLLER is called and displays the game over messages.

We hope that this little walk-through could give you an idea of what’s going on when you run FLAT_HUNT, and that you enjoy both playing and working with FLAT_HUNT.
5.5 Summary

This chapter described the most important change from 'Introduction to Programming 2003' to 'Introduction to Programming 2004': TRAFFIC’s redesign. The problems of TRAFFIC 1.0 are mentioned, followed by a description of how TRAFFIC 2 tackles these problems by making TRAFFIC slimmer and easier accessible.

TRAFFIC is now really a library and not a combination of an application and a library. There are two new applications based on the TRAFFIC library: The TOUCH_APPLICATION for the book examples and FLAT_HUNT, a Scotland Yard like game on which the exercises are built. These two applications should be easier understandable and less error-prone than the 'old' TRAFFIC.

To help the students get used to FLAT_HUNT, not only the game rules, but also FLAT_HUNT's design is described in the FLAT_HUNT User & Developer Documentation.

An important goal of the Inverted Curriculum is that the students have an interesting application to work on and learn from. Hopefully, FLAT_HUNT and TRAFFIC 2 serve this goal better than TRAFFIC 1.0.
6 The new exercises

Next to redesigning TRAFFIC, the other goal of this thesis was to devise a new set of exercises. Even though the students were already generally happy with the exercises (see Section 3.3 on page 27), there was room for improvement, for example regarding an easier understanding of the questions. And as several exercises are closely related to TRAFFIC, and TRAFFIC has changed significantly, exercises had to be changed as well.

The next section (6.1) gives an overview of the new lecture schedule and the exercises layout. All the assignments are listed with a short description of each exercise. In Section 6.2, we take a closer look at two exercise examples ('My first program' and 'Fancy loop exercise'), before we consider the new final project in Section 6.3. A summary concludes this chapter.

Remark: All assignments (including the final project) can be found in Appendix C.

6.1 Overview

As mentioned in Section 3.4.1 on page 36, the evaluation suggested some minor adjustments in the lecture schedule. The biggest problem was the late introduction of assignment (':='). We suggest that assignment is introduced directly after object creation (Week 4 in Table 5). The reason for this is not only the assistants and students critical feedback, but also the fact that it is hard to devise interesting exercises without assignments. The rest of the schedule will be more or less the same as in 2003. Table 5 gives a more detailed overview of the planned lecture and exercise schedule for Winter 2004.
<table>
<thead>
<tr>
<th>Week</th>
<th>Lecture topics</th>
<th>Exercise topics</th>
</tr>
</thead>
</table>
| 1    | **Lecture 1.1:** No lecture  
**Lecture 1.2:** Introduction | **Exercise 1:** Get used to ETH infrastructure (e-mail, newsgroups) |
| 2    | **Lecture 2.1:** Dealing with objects I (Objects, feature calls, arguments)  
**Lecture 2.2:** Dealing with objects II (Syntax and semantics) | **Exercise 2:** First program / Command vs. query |
| 3    | **Lecture 3.1:** The interface of a class (Classes vs. objects, contracts)  
**Lecture 3.2:** Invariants and logic | **Exercise 3:** Classes / Features / Contracts |
| 4    | **Lecture 4.1:** Creating objects (Identifiers, void references, root class)  
**Lecture 4.2:** Assignment, references, and object structure | **Exercise 4:** Object creation |
| 5    | **Lecture 5.1:** Control structures I (Algorithms, sequences, conditionals)  
**Lecture 5.2:** Control structures II (Loops) | **Exercise 5:** Loops / Conditionals |
| 6    | **Lecture 6.1:** Dynamic Model I (Uniform access, variables, types, garbage collection)  
**Lecture 6.2:** Describing the Syntax (BNF) | **Classroom exercise I:** Concepts / Syntax / Contracts |
| 7    | **Lecture 7.1:** Dynamic model II  
**Lecture 7.2:** Inheritance I (Polymorphism, deferred classes) | **Exercise 6:** Value vs. reference / Inheritance |
| 8    | **Lecture 8.1:** Inheritance II and genericity  
**Lecture 8.2:** Inheritance III (Multiple inheritance) | **Exercise 7:** Dynamic binding / Aggregation vs. inheritance |
| 9    | **Lecture 9.1:** Data structures I (Linked list)  
**Lecture 9.2:** Data structures II (Stack) | **Exercise 8:** Generics / Lists |
| 10   | **Lecture 10.1:** Recursion  
**Lecture 10.2:** A glimpse into the hardware | **Project:** Extend Flat_hunt / Game of Life / Own ideas |
| 11   | **Lecture 11.1:** Testing  
**Lecture 11.2:** Event-driven programming I | **Classroom exercise II:** Recursion vs. loop / Data structures / Constrained genericity |
| 12   | **Lecture 12.1:** Event-driven programming II  
**Lecture 12.2:** Undo/Redo example | **Project:** Extend Flat_hunt / Game of Life / Own ideas |
| 13   | **Lecture 13.1:** Topological sort I: Background  
**Lecture 13.2:** Topological sort II: Algorithm | **Project:** Extend Flat_hunt / Game of Life / Own ideas |
| 14   | **Lecture 14.1:** From programming to software engineering  
**Lecture 14.2:** Glance at 'Algorithm and data structures' / Project presentation | **Project:** Extend Flat_hunt / Game of Life / Own ideas |

Table 5: Planned schedule 2004
6.1.1 Description of the assignments

Following up is a short overview of all the exercises to appear in Winter 2004. The exercises are grouped in their respective assignments, with a short overview of the topics covered in that assignment. For each exercise, the goals are stated, followed by a short task description. There are three properties to group the exercises into categories:

- FLAT_HUNT related
- Stand-alone program
- Paper-and-pencil

For each property, if it is preceded by an 'x' and in bold italics, the exercise conforms to this property. If there is an 'o' and no bold italics, the property does not hold for the exercise. These properties are useful to achieve the right mix (see Section 6.1.2). Last but not least, for every exercise, the classification according to Bloom’s Taxonomy (see Section 4.4.5) is given.

Assignment 1: “Hello ETH”

Topic(s):
- Get used to ETH infrastructure (e-mail, newsgroups)

1. Get started

Goal(s):
- Students get used to ETH (and its infrastructure).
- Make sure everybody has a working e-mail account.

Description:
Students have to change their password and register to the course.

Categories:
- FLAT_HUNT
- Stand-alone
- Paper-and-pencil
- Taxonomy: K3

2. Information sharing

Goal(s):
- Students get used to the forum.
- Students post a (possibly first) message in a forum.
- Students visit some interesting websites.

Description:
Students have to visit the forum and post a message.

Categories:
- FLAT_HUNT
- Stand-alone
- Paper-and-pencil
- Taxonomy: K3
Assignment 2: Taking off

Topic(s):

- First program
- Command vs. query

1. My first program

Goal(s):

- Students get EiffelStudio running.
- Students get TRAFFIC compiled.
- Students write their first feature calls.
- Students see the effects of contracts.

Description:

Students start FLAT_HUNT by writing feature calls on predefined objects.

Categories:

| FLAT_HUNT | Stand-alone | Paper-and-pencil | Taxonomy: K2 |

2. Command vs. Queries

Goal(s):

- Understand the difference between commands and queries.

Description:

Students have to classify features of a given class in one of the two categories command or query.

Categories:

| FLAT_HUNT | Stand-alone | Paper-and-pencil | Taxonomy: K4 |

Assignment 3: Of Objects and Contracts

Topic(s):

- Classes
- Features
- Contracts

1. Class vs. Objects

Goal(s):

- Students understand the difference between the notion of class and the notion of object.
Description:
Students have to describe the difference between class and object in their own words.

Categories:

| Flat_Hunt | Stand-alone | Paper-and-pencil | Taxonomy: K2 |

2. Feature reading

Goal(s):

- Students discover and understand FLAT_HUNT.
- Students get used to EiffelStudio.
- Students get used to the .-notation.
- Students experiment with the debugging mode.

Description:
Students have to figure out the return type of certain feature calls.

Categories:

| Flat_Hunt | Stand-alone | Paper-and-pencil | Taxonomy: K4 |

3. Contracts

Goal(s):

- Students use and understand contracts.
- Students appreciate the advantages of Design by Contract for debugging.

Description:
Students have to fix/add contracts in a given class.

Categories:

| Flat_Hunt | Stand-alone | Paper-and-pencil | Taxonomy: K3 |

Assignment 4: Object creation

Topic(s):

- Object creation

1. My first application

Goal(s):

- Students write their first application from scratch.
- Students create and use a new class.
- Students learn about basic input/output in Eiffel.
Description:
Students have to write a temperature converter application from scratch.

Categories:
- FLAT_HUNT
- Stand-alone
- Paper-and-pencil
- Taxonomy: K5

2. GAME_DISPLAYER

Goal(s):
- Students learn how to refactor a class.
- Students create and use a new class in FLAT_HUNT.

Description:
Students have to refactor a class.

Categories:
- FLAT_HUNT
- Stand-alone
- Paper-and-pencil
- Taxonomy: K5

Assignment 5: Loops

Topic(s):
- Conditionals
- Loops

1. Where is “Central”?

Goal(s):
- Students understand loop structure and conditionals.
- Students realize the importance of proper stop criterions.

Description:
Students have to correct two given loop extracts.

Categories:
- FLAT_HUNT
- Stand-alone
- Paper-and-pencil
- Taxonomy: K2

2. Fancy animate_defeat animation

Goal(s):
- Students play around with loops and conditionals.
- Students are creative and make FLAT_HUNT look nicer.

Description:
Students have to program a loop to display a nice animation.

Categories:
- FLAT_HUNT
- Stand-alone
- Paper-and-pencil
- Taxonomy: K5
3. **Loop painting**

**Goal(s):**

- Students try nested loops.

**Description:**

Students have to write a nested loop to print out given ASCII character pattern.

**Categories:**

| FLAT_HUNT | Stand-alone | Paper-and-pencil | Taxonomy: K3 |

**Assignment 6: “equal” = “equal”?

**Topic(s):**

- Value vs. reference
- Object structure
- Inheritance

1. **Value vs. reference, deep vs. shallow**

**Goal(s):**

- Students understand the difference between value and reference.
- Students understand the difference between deep and shallow.

**Description:**

Students have to answer multiple choice questions concerning the above topics.

**Categories:**

| FLAT_HUNT | Stand-alone | Paper-and-pencil | Taxonomy: K2 |

2. **Object structure**

**Goal(s):**

- Students understand how objects are linked together.

**Description:**

Students have to draw a picture of a run-time object structure.

**Categories:**

| FLAT_HUNT | Stand-alone | Paper-and-pencil | Taxonomy: K4 |

3. **Inherited fraction**

**Goal(s):**
• Students inherit from a class.
• Students use infix/prefix notation.

Description:
Students have to inherit from a given class and implement the missing features.

Categories:
- FLAT_HUNT
- Stand-alone
- Paper-and-pencil
- Taxonomy: K5

Assignment 7: More Inheritance

Topic(s):
• Dynamic binding
• Aggregation vs. inheritance

1. Dynamic exercise

Goal(s):
• Students understand the effects of dynamic binding.

Description:
Students have to look at code snippets and decide whether they are valid or not.

Categories:
- FLAT_HUNT
- Stand-alone
- Paper-and-pencil
- Taxonomy: K3

2. From Client/Supplier to Multiple Inheritance

Goal(s):
• Students adopt the FLAT_HUNT player-brain mechanism so that it makes use of multiple inheritance.

Description:
Students have to change given classes to support multiple inheritance.

Categories:
- FLAT_HUNT
- Stand-alone
- Paper-and-pencil
- Taxonomy: K5
Assignment 8: Genericity

Topic(s):

- Genericity
- Lists

1. Santa needs help

Goal(s):

- Students learn how to use lists of lists.
- Students have a first glimpse at recursion.

Description:

Students have to iterate through a list of lists.

Categories:

\[ \text{x FLAT\_HUNT} \quad \text{o Stand-alone} \quad \text{o Paper-and-pencil} \quad \text{Taxonomy: K3} \]

2. current = next.previous

Goal(s):

- Students implement a more complex data structure.
- Students play around with references.
- Students use genericity.

Description:

Students have to write a generic list class.

Categories:

\[ \text{o FLAT\_HUNT} \quad \text{x Stand-alone} \quad \text{o Paper-and-pencil} \quad \text{Taxonomy: K5} \]

6.1.2 The right mix

In last years course, most of the exercises were based on TRAFFIC. According to the Outside-In approach, it was not planned that the students would write applications from scratch. All the exercises should make use of the library. However, the evaluation showed that the students want to know how a program works from the start until the end. They want to have complete control over the whole program. Thus we decided that there should be some exercises where the students can write programs from scratch.

These are somewhat 'typical' CS1 exercises, like for example the Celsius-to-Fahrenheit converter in Assignment 4 (Appendix C.4). They are not as 'fancy' as some TRAFFIC exercises, but they are assumed to give the students a different perspective on software engineering.

In addition to the classroom exercises, we decided to include non-programming exercises also in the normal assignments. This goes in line with our conclusions.
in Section 4.5.1 on page 52. Examples are Exercise 1 in Assignment 3 (Appendix C.3) and Exercise 2 in Assignment 6 (Appendix C.6).

Finally, we came up with the following recipe for a prototypical assignment:

- 1/3 FLAT_HUNT modifications/extensions
- 1/3 stand-alone programs
- 1/3 paper-and-pencil exercises

Table 6 lists the categories and the taxonomy of the assignments described in Section 6.1.1. It shows that in almost all the later assignments, there are also stand-alone and paper-and-pencil exercises. The taxonomy ranges from K2 to K5.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Exercise</th>
<th>FLAT_HUNT</th>
<th>Stand-alone</th>
<th>Paper-and-pencil</th>
<th>Taxonomy (K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>x</td>
<td>o</td>
<td>o</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>o</td>
<td>o</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>o</td>
<td>x</td>
<td>o</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>x</td>
<td>o</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>o</td>
<td>o</td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>o</td>
<td>o</td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>x</td>
<td>o</td>
<td>o</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6: Assignment category and taxonomy overview
6.1.3 Exercise structure

To make things easier for the students, all exercises are structured as follows:

- Goal
- Description
- To do
- Hint/Note
- To hand in

The Goal section should help the student to get the key point of the exercise (see 4.5.1 on page 53). Description gives an overview of the task, while To do guides the student step by step through the exercise. Often there is a Hint section which provides help for the less experienced students (see 4.5.1 on page 53). The last section is called To hand in and specifies exactly what needs to be submitted to complete this exercise.

6.2 Two examples

Appendix C contains totally 19 exercises that range from command-query separation to multiple inheritance and genericity. The next two subsections present two sample exercises that rely on FLAT_HUNT and the TRAFFIC library. For each of them, first the exercise text as it is on the hand-out is given. Then some interesting points of the exercise are discussed in more detail. These discussions include references to the findings of Section 3.4 and Chapter 4.

6.2.1 My first program

Exercise text (see also Appendix C.2)

Remark: In the assignment, the text below is preceded by a manual on how to install EiffelStudio and get everything ready.

Goal

- Get EiffelStudio running.
- Get TRAFFIC compiled.
- Write your first feature calls.
- See the effects of contracts.
Description
You will do your first steps in programming. The class that you will change is called START. Open this class in EiffelStudio (see Figure 1) and perform the following tasks:

To do

1. Without changing anything in the class, click on 'Launch'. This will start the application, however, nothing will happen. Close the application.

2. In the feature start, between the 'do' and the 'end', fill in the following text:
   
   ```
   open_map ("zurich_big_city.xml")
   game.set_game_mode (Hunt)
   game.set_number_of_hunters (3)
   start_game
   ```

3. Compile the project again and launch it. It takes some time to load the map, but then you should be able to play FLAT_HUNT. It should look like the screenshots in the FLAT_HUNT documentation.

4. Try changing the modes of the game (see FLAT_HUNT documentation for more information):
   
   - Load a different map
   - Try different game modes
   - Change the number of players (try also negative values and values greater than 8)

   What happens if you change the order of the calls?

5. If you want the game to start in full screen mode, add the following statements:
   
   ```
   main_window.set_position (0, 0)
   main_window.set_size (1024, 768)
   ```
Hint

- Don’t forget to compile your project every time you change something in the class text, before running it. (If you don’t do so, you will run a version without your latest changes.)

- If something goes wrong (especially in Task 4) just click on ‘Kill’ and undo your changes.

- If you are curious about something, don’t hesitate to take a closer look. However, don’t be disappointed if you have to take another look in one of the following exercises.

To hand in

There is nothing to hand in. We assume that you did spend some time with the application.

Discussion

This exercise is the Outside-In analogon to a ‘Hello World!’ exercise in another curriculum. The only thing the students have to do is to call a number of features with the right arguments. By using the predefined class GAME, the students also have to call features on other objects.

Instead of displaying some letters, these few calls start a whole game with graphic and sound, hopefully satisfying the needs of the ‘Nintendo Generation’ [10].

By trying unallowed values in Task 4, the students will already see the advantage of Design by Contract, without somebody having to tell them the details yet.

If the students get stuck, the Hint section should help them to get back on track again. The students are invited to have a deeper look if they feel the engineers urge to know what is going on.

6.2.2 Loop exercise (Fancy animate_defeat animation)

Exercise text (see also Appendix C.5)

Goal

- Play around with loops and conditionals.

- Be creative and make FLAT_HUNT look nicer.

Description

In the class PLAYER_DISPLAYER in FLAT_HUNT, there is a feature called animate_defeat.

This feature gets called either on the estate agent when the hunters find him, or on all the hunters if the agent can escape. Up to now, animate_defeat just draws a black circle. There is a loop prepared, but for now, the loop is empty.

Your task is to fill this loop and try to make a nice animation whenever the game is over. This could for example look like Figure 1. However, instead of circles, you might also want to use lines or rectangles, and maybe play with colors.

Probably it gets even cooler if you add some conditionals as well.
Exercise Design for Introductory Programming

Note

- Make sure that loop assertions in the menu 'Project configuration' are enabled.
- Have a look at the comments in the source code.

To do

- Fill in the loop in feature *animate_defeat* in class *PLAYER_DISPLAYER*.

To hand in

Hand in your version of feature *animate_defeat* plus a screenshot of your animation.

Discussion

Usually, when loops are introduced, students have to implement a countdown or something similar, which is just printing out some numbers. Thanks to the library we are using, the students can be a bit more creative and are likely to play around with the loop structure (in contrast to just meeting the minimal requirements).

The students don't have to write a small application before they can start with the real goal of the exercise. A whole application is already there, the students can get straight to the point (see also Section 4.5.1). The picture of a possible solution should give the students a clear idea of what is expected (compare with Section 4.5.1).

6.3 The final project

Last year's project was to extend TRAFFIC with an entertainment information system for cinemas and theaters. The system had to hold information both about the events (date, time, location, show, etc.) and the shows (title, description, actors, directors, etc.). See 3.1.6 on page 20 for more information.

The students were already pretty happy with that project (see Section 3.4.4 on page 37). However, some students would have preferred to do something unrelated to TRAFFIC, or to implement their own ideas. The new project should give them more freedom, as it features three options:

- Option A: Extend FLAT_HUNT in some way
• Option B: Implement Conway’s Game of Life
• Option C: Do something they’ve always wanted to do

6.3.1 The options

The next three subsections describe these options in more detail. They are taken from the project exercise (Appendix C.9):

Option A: FLAT_HUNT extension

By now, you have done several exercises with FLAT_HUNT, and probably you were annoyed about something, or you desperately missed a certain game feature. The time has come where you can extend FLAT_HUNT and give it your personal touch. Possible ideas are:

• Improve the artificial intelligence
• Add new game modes
• Make FLAT_HUNT net-playable (and add a chat-client)
• Add special cards (e.g. to block a player)
• Add new types of transport (e.g. a jetpack)
• Make FLAT_HUNT look nicer (e.g. add a small overview map)
• Make FLAT_HUNT real-time

Option B: Game of Life

The Game of Life is not a game in the conventional sense. There are no players, and the game is not about winning or losing. It is more like a simulation that runs according to some specified rules.

To find out more about the Game of Life, visit the very interesting webpage http://www.math.com/students/wonders/life/life.html.

Your job is to design an Eiffel version of Game of Life. Whether you build a nice GUI (Graphical User Interface) or make it text-based is completely up to you.

Option C: I’ve always wanted to

As the title suggest, you can do almost whatever you want. Almost just means that your assistant has to agree with what you have in mind, and that your idea is worked out as clearly as they have to be in options A and B.

If you intend to do something that involves graphics and multimedia, you might want to check out ESDL. ESDL is a wrapper for SDL, the Simple DirectMedia Layer library that is very popular among Linux game developers. Find more information about ESDL on http://eiffelsdl.sourceforge.net/.

Another important point: Your project must not have any offensive content, and it should not use copyrighted material.
6.3.2 How to form the groups?

Taking on the idea from the Norwegian University of Science and Technology (NTNU), we will not form heterogeneous groups (with the idea that the more experienced students can help the newbies), but make homogeneous groups of the same skill (see Section 4.4.3 on page 47 for more on this idea).

The introduction to the project exercise reads like this (Appendix C.9):

Please do not solve this assignment alone. Actually, you should do the project in groups of three. Try to form uniform groups concerning your programming experience, so that you can choose a task that is equally challenging for all the three of you.

6.3.3 Just coding?

Common to all the three options is the following: To keep students from doing nothing until the last minute or going in the completely wrong direction, two weeks after the hand-out of the project exercise the students have to hand in a short report describing their planned system and their design decisions.

Only after this they should start with the implementation. This approach should also get the students used to professional software engineering, which is another part of the Inverted Curriculum method.

6.4 Summary

Designing exercises for an introductory teaching course is an interesting task, especially when the course is based on an approach like Inverted Curriculum, which extensively makes use of a library to introduce the concepts.

Only minor changes had to be made to the schedule, but several new exercises were designed, not only because the TRAFFIC library has changed, but also because the students criticized that there were no 'exercises from scratch'.

Two sample exercises were presented, the 'Hello, World!' à la Outside-In and the graphical loop exercise. All the other exercises can be found in Appendix C. In the last section, we had a look at the new project exercise with three different options.
7 Summary and outlook

On Tuesday, October 19, 2004, 'Introduction to Programming' will go in round two, after a successful first try in Winter 2003. For the second time, 'Introduction to Programming' is based on the Inverted Curriculum idea:

- Put emphasis on 'programming in the large'.
- Stress important software engineering principles and object-oriented techniques as opposed to syntactic, programming language dependent details.
- Introduce topics Outside-In: Classes and objects first.
- Use an object-oriented library that hides at the time unneeded details from the students.
- Represent real-world behavior where reuse mostly occurs through the use of pre-existing libraries.

In this thesis, we analyzed the first run of Winter 2003 and came to the following conclusions for the second round:

- The Outside-In order of topics was generally appreciated, but assignment should be introduced earlier.
- Some of the students had problems understanding the exercises. The questions should be more clear. The students also asked for more exercises that are not based on TRAFFIC.
- Eiffel is a language that is suitable for introductory programming. However, EiffelStudio is difficult to use, and the students need more guidance.
- TRAFFIC, the library used, posed many problems: It was slow, unstable and too complex. The majority of the students did not like the idea to work with TRAFFIC.

TRAFFIC had to be redesigned. TRAFFIC 2 is less complex and serves as a library for two example applications: The TOUCH_APPLICATION which holds the examples from the 'Touch of Class' textbook, and FLAT_HUNT. FLAT_HUNT is an easy-understandable game that is played on a public transport map of the city of Zurich. Having a game as an example application should motivate the students even more.

To help the students play and work with FLAT_HUNT, the FLAT_HUNT User & Developer Documentation was written. This documentation contains a description of the story and the rules of the game in Part I, and a design and class overview for FLAT_HUNT and TRAFFIC in Part II.

Apart from the redesign of TRAFFIC, the other main goal of this thesis was to devise new exercises. When designing introductory programming exercises, one should
• clarify the goals the student must attain,
• emphasize the key points,
• clear away nonessential components,
• provide guidelines to writing and testing code,
• and include debugging strategies.

Many of the exercises are based on FLAT_HUNT. By making use of a library, more interesting exercises like the 'Fancy loop exercise' are possible. Nevertheless, there are also some stand-alone exercises, where the students can write an application from scratch, and some paper-and-pencil exercises that do not involve coding but gear the students to think deeply about the presented material. The final project is not only based on FLAT_HUNT anymore - there are two other options to give the students more freedom.

Many things have changed in these seven months between the last lecture of 'Introduction to Programming 2003' and the start of 'Introduction to Programming 2004' in October.

This Winter's run will again be evaluated in detail. Hopefully the analysis in Spring 2005 will show that the improvements presented in this thesis helped in making 'Introduction to Programming 2004' an even bigger success.
A Project plan

Exercise Design for Introductory Programming
“Learn-by-doing” basic OO-concepts using Inverted Curriculum

PROJECT PLAN

Master Thesis
Thursday, March 4 – Friday, September 3 2004
Marcel Kessler
9-th semester
kesslema@student.ethz.ch
Supervising assistant: Michela Pedroni
Supervising professor: Bertrand Meyer

1. PROJECT DESCRIPTION

Overview

In Winter 2003/2004, a new introductory course on programming was introduced at the ETH Zurich [1]. A new approach, called Inverted Curriculum, was used for the first time. In this approach, students step-by-step grow from consumers to producers of a library (called “TRAFFIC”) especially designed for this purpose. That is why Inverted Curriculum is also known as „progressive opening of black boxes“ or the „outside-in method“.

Exercises were designed in a way that first the students’ application just uses the components of the library (the “black boxes”), and later the students gradually plunge deeper into the library code by requesting modification and extensions of the library code. This way, the students should appreciate the benefits of abstraction, reuse and a good object-oriented framework right from the start.

After a first mostly successful test on the students, the especially designed library TRAFFIC and the exercises belonging to it are to be redesigned and improved, to get even closer to the goal of having a perfect library that can be used and extended as intended in the Inverted Curriculum.

Scope of the work

The master thesis is composed of 5 parts:
- Background: Getting used to TRAFFIC and the “Introduction to Programming” course
- Theory/Reading: Criteria and general rules for designing educational meaningful exercises in computer science introduction
- Framework: Form (goals, difficulty (mastery concept), hints, Lernaufgaben [2] etc.) and method of access to the exercises (and the master solutions) and the final project
- Exercises: Evaluating the “old” exercises, designing new exercises and proper master solutions, designing additional exercises (for exam preparation)
- TRAFFIC: Playing a part in the redesign of TRAFFIC (additional needs for new exercises) and integrating the exercises into the TRAFFIC software

**Intended results**

- Report: A documentation of all the work that was performed during the thesis
- Practical work: The appropriate exercises and solutions (integrated in TRAFFIC)

**Personal motivation**

I had almost decided on another diploma thesis when I heard about this topic, and after giving it some thought it was clear for me that I had to switch to this thesis. One reason for this was that I am doing “education” as my minor subject and that I am very much interested in teaching and didactics. The other reason was that I will have the chance to design exercises that will be used by several hundred students and might therefore have some (hopefully positive) impact on some human beings instead of disappearing somewhere deep down in a drawer.

**2. BACKGROUND MATERIAL**

**Reading list**


3. PROJECT MANAGEMENT

Objectives and priorities
1. Design useful exercises for the course “Introduction to Programming” to be held winter 2004
2. Adaptation of TRAFFIC to fit the new exercise needs
3. Supply additional exercise for self-study and exam preparation

Obviously, the first two priorities cannot be looked at independently, yet the main focus of this thesis is the exercises.

Criteria for success
1. The students are happy with the new exercises

4. PLAN WITH MILESTONES

Project steps
Project start: Thursday, March 4 2004
Theory read and documented: Tuesday, 23 March 2004
Problems of “first-try” evaluated: Friday, 26 March 2004
Decision on framework and integration into TRAFFIC: Tuesday, 30 March 2004
Prototype developed: Monday, 17 May 2004
Exercises and solutions designed: Tuesday, 6 June 2004
Exercises pre-tested on certain students: Monday, 19 July 2004
Report completed: Thursday, 2 September 2004

Deadline
Friday, September 3, 2004
REFERENCES


B Style guide

DO IT WITH STYLE – A Guide to the Eiffel Style


If you have any corrections or comments, please send an e-mail to Marcel Kessler (kesslema@student.ethz.ch).

CHOOSING THE RIGHT NAMES

• For feature and class names, use full words, not abbreviations, e.g. call number, not num.
• Do not hesitate to use several words connected by underscores, as in ANNUAL_RATE.
• For features, there is seldom a need for more than two or possibly three underscore-connected words.
• Do not include in a feature name the name of the underlying data abstraction (which should serve as the class name).
  o The feature giving the part number in class PART should be called just number, not part_number.
• Sometimes, every instance of a certain class contains a field representing an instance of another class. Although you should try to find a more specific name, you may, if this fails, just declare the feature as rate: RATE.
• Local entities and arguments of a routine only have a local scope, so they do not need to be as evocative.
• Arguments to functions usually have a prefix a_, like in print_name (a_name: STRING).

move (i: INTEGER) is
   -- Move cursor i positions, or after if i is too large.
   local
      c: CURSOR, counter: INTEGER, p: like FIRST_ELEMENT
   ...

remove is
   -- Remove current item; move cursor to right neighbor.
   local
      succ, pred, removed: like first_element
   ...

• If succ and pred had been features they would have been called successor and predecessor.
Letter case

- Class names appear in all upper case: POINT, LINKED_LIST...
- Names of attributes, routines etc. appear in all lower case: balance, deposit, succ, i.
- Constant attributes have their first letter in upper case and the rest in lower case: Pi: INTEGER is 3.1415926524; Welcome_message: STRING is "Welcome!"
- A few reserved words are written with an initial upper case since they are similar to constants, they include Current, Result, Precursor, True and False.

Grammatical categories

- For class names, you should always use a noun, possibly qualified as in LONG_TERM_SAVINGS_ACCOUNT.
- Routine names should faithfully reflect the Command-Query separation principle:
  - Procedures (commands) should be verbs in the infinitive or imperative: make, move, deposit, set_color.
  - Attributes and functions (queries) should never be imperative or infinitive verbs; never call a query get_value, but just value.
- Non-boolean query names should be nouns, such as number.
- A frequent convention for boolean queries is the is_ form, as in is_empty.

HEADER COMMENTS AND INDEXING CLAUSES

Instead of the long comment in tangent_from (p: POINT): LINE is

```
-- Return the tangent line to the circle going through the point p,
-- if the point is outside of the current circle.
require
   outside_circle: not has (p)
```

just write

```
-- Tangent from p.
```

because of the following reasons:

- The comment for a query, as here, should not start with “Return the…” or “Compute the…” . Simply name what the query returns, typically using a qualified noun.
- We can get rid of the auxiliary words, especially the, where they are not required for understandability.
- Another mistake is to have used the words line to refer to the result and point to refer to the argument: this information is immediately obvious from the declared types, LINE and POINT.
• Header comments for commands (procedures) should end with a period. For boolean-valued queries, the comment should always be in the form of a question, terminated by a question mark:

\[\text{has (v: G): BOOLEAN is} \]
\[\text{-- Does 'v' appear in list?} \]

• Software entities — attributes, arguments — appearing in comments in the source text should always appear between an opening quote (“backquote”) and a closing quote.

Because an exported attribute should be externally indistinguishable from argumentless functions — remember the Uniform Access principle — it should also have a comment:

\[\text{count: INTEGER} \]
\[\text{-- Number of students in course} \]

**TEXT LAYOUT AND PRESENTATION**

The textual layout of the notation follows a **comb-like structure**; the idea is that a syntactically meaningful part of a class, such as an instruction or an expression, should either:

• Fit on a line together with a preceding and succeeding operators.
• Be indented just by itself on one or more lines.

\[\text{if } c \text{ then } a \text{ else } b \text{ end} \]

or

\[\text{if} \]
\[c\]
\[\text{then} \]
\[a\]
\[\text{else} \]
\[b\]
\[\text{end} \]

or

\[\text{if } c \text{ then} \]
\[a\]
\[\text{else } b \text{ end} \]

**Spaces**

You will use a space:

• Before an opening parenthesis, but not after: \(f(x)\).
• After a closing parenthesis unless the next character is a period or semicolon, but not before. Hence: \(proc1(x); x := f1(x) + f2(y)\).
• After a comma but not before: \(g(x, y, z)\).

Spaces should appear before and after arithmetic operators, as in \(a + b\).
A layout example

indexing
  description: "Example for formatting"

class
  EXAMPLE
  inherit
    MY_PARENT
      redefine f1, f2 end
  MY_OTHER_PARENT
    rename
      g1 as old_g1, g2 as old_g2
      redefine
        g1
        select
          g2
        end
    create
      make
    feature
      -- Initialization
      make is
        -- Do something.
        require
          some_condition: correct (x)
        local
          my_entity: MY_TYPE
        do
          if a then
            b, c
          else
            other_routine
            new_value := old_value / (max2 - max1)
          end
        end
    feature
      -- Access
      my_attribute: SOME_TYPE
        -- Explanation of its role (aligned with comment for make)
        ...
    invariant
      upper_bound: x <= y
  end
C Exercises

C.1 Assignment 1

Assignment 1: “Hello ETH”

Hand-out: 19 October 2004
Due: 25 October 2004

The goal of this exercise is to familiarize you with the infrastructure provided by ETH Zurich. Especially the use of email and the web-based forum will be treated. This assignment should be solved alone.

1. Get started

Goal
- Get used to ETH (and its infrastructure).
- Make sure you have a working e-mail account.

To do
1. Log in to a computer in one of the computer rooms. The locations of the computer rooms are listed on the information sheet that you received.
2. Log on to the n.ethz.ch account administration page (https://n.ethz.ch/cgi-bin/admin_tool/main.cgi) to change your assigned password. After logging in you should see the welcome screen (Figure 1: Welcome screen). Select “Passwort ändern”, change your password¹ (Figure 2: Change password screen) and logout. After this you can log on to ETH’s student web-mail interface (http://mail.student.ethz.ch/) with your user name and your newly set password.

¹ You absolutely need to remember this password; yet it has to be complex enough for security reasons. See password recommendations at CERN: http://security.web.cern.ch/security/passwords/
3. Try to find the Eiffel development environment - EiffelStudio - on the machine that you are using.
4. Make sure that you have subscribed to this course (and all the other courses you visit this term) on http://www.einschreibung.ethz.ch. Otherwise you might not get your testat.

Figure 1: Welcome screen

Figure 2: Change password screen
Hint

- You will find your n.ethz login and password also useful on many other ETH web pages, like the above mentioned http://www.einschreibung.ethz.ch.

To hand in

As soon as you have subscribed to the course (Task 4), this first exercise is accomplished. There is nothing to hand in.

2. Information sharing

Goal

- Get used to the forum.
- Post your (possibly first) message in a forum.
- Visit some interesting websites.

To do

2. Create an account in the forum (Click on “Registrieren”).
3. Find the thread of your group and post a message there. The message should include a link to a website that is of relevance to either:
   - Starting out at ETH
   - Student life in Zurich
   - Introduction to programming & Eiffel
4. Visit some of the posted links of your group members.
Hint
- Try to find a page that could be of real interest to all your fellow students.

Example:
- Homepage of "Verein der Informatik Studierenden" (VIS):
  http://www.vis.ethz.ch/; Contains information about the computer science degree, job and book offers as well as the forum…

To hand in
There is nothing to hand in. The second exercise is accomplished when you have posted at least one message in the forum.
Assignment 2: Taking off

Hand-out: 25 October 2004
Due: 1 November 2004

In this assignment you will start your first Eiffel-program.
Please solve this assignment alone.

1. My first program

Goal
- Get EiffelStudio running.
- Get TRAFFIC compiled.
- Write your first feature calls.
- See the effects of contracts.

Preparation
Doing this preparation will get you ready to write your first program and therefore to solve the main part of this exercise.
1. Download and install EiffelStudio from http://www.eiffel.com/downloads if you intend to work at home.
2. Download the TRAFFIC software (traffic.zip) from http://se.inf.ethz.ch/teaching/
3. Unzip traffic.zip to a directory of your choice.
4. Start EiffelStudio.
5. Select “Open existing Ace (control file)” (see Figure 1).
6. Browse to where you unzipped TRAFFIC.
7. Click “Next”.
8. Make sure that “Compile the generated project” is checked, and click “OK” (see Figure 2).
9. Have a look at the FLAT_HUNT documentation (Flat_hunt_docu.pdf), which you can find in subdirectory example/flat_hunt/doc.
Description
You will do your first steps in programming. The class that you will change is called START. Open this class in EiffelStudio (see Figure 4) and perform the following tasks:

To do
1. Without changing anything in the class, click on “Launch” (see Figure 3). This will start the application, however, nothing will happen. Close the application.
2. In the feature start, between the do and the end, fill in the following text:
   
   ```
   open_map ("zurich_big_city.xml")
   game.set_game_mode (Hunt)
   game.set_number_of_hunters (3)
   start_game
   ```

3. Compile (see Figure 3) the project again and launch it. It takes some time to load the map, but then you should be able to play FLAT_HUNT. It should look like the screenshots in the FLAT_HUNT documentation.
4. Try changing the modes of the game (see FLAT_HUNT documentation for more information):
   - Load a different map
• Try different game modes
• Change the number of players
  (Try also negative values and values greater than 8)
  What happens if you change the order of the calls?
5. If you want the game to start in full screen, add the following statements:
   
   ```
   main_window.set_position (0, 0)
   main_window.set_size (1024, 768)
   ```

   **Hint**

   • Don’t forget to compile your project every time you change something in the class text, before running it. (If you don’t do so, you will run a version without your latest changes.)
   • If something goes wrong (especially in task 4) just click on “Kill” (see Figure 5) and undo your changes.
   • If you are curious about something, don’t hesitate to take a closer look. However, don’t be disappointed if you have to take another look in one of the following exercises.

   ![Figure 5 Recover from a problem](image)

**To hand in**

There is nothing to hand in. We assume that you did spend some time with the application.

### 2. Commands vs. Queries

**Goal**

• Understand the difference between commands and queries.
Summary

Feature
an operation available on a certain class of objects

Command
Goal: produce a change on an object

Query
Goal: obtain properties of objects

Description
The class below is a simplified and cropped version of the class GAME appearing in FLAT_HUNT:

```plaintext
class GAME
inherit STATE_CONSTANTS
create
make
feature -- Initialization
make is
    -- Creation procedure
    do
        create game_displayer.make (Current)
        create checkpoint.make
        checkpoint.fill ([< 3, 8, 13, 18, 23 >])
    end
start_game is
    -- Prepare the board, start the game.
    local player_factory: PLAYER_FACTORY
    do
        create player_factory.make (city)
        create player.make (hunter_count + 1)
        current_round_number := 0
        turn := 0
        state := Prepare_state
    end
```
if game_mode = Hunt then
  player_factory.build_players (True, False, hunter_count)
elseif game_mode = Escape then
  player_factory.build_players (False, True, hunter_count)
elseif game_mode = Versus then
  player_factory.build_players (False, False, hunter_count)
elseif game_mode = Demo then
  end
end
players := player_factory.players
estate_agent := players.first
check
end
if game_mode = Hunt then
  estate_agent.display.set_hunt_mode (True)
end
next_turn
end

feature -- Status setting
set_game_mode (a_mode: INTEGER) is
  -- Set mode of game.
  do
    game_mode := a_mode
  end
end

feature -- Status report
is_occupied (loc: PLACE): BOOLEAN is
  -- Check if `loc' is already occupied by another player.
  do
    from
      players.start
      players.forth
    until
      players.off or Result
    loop
      if players.item.location = loc then
        Result := True
      end
    players.forth
  end
end

feature -- Basic operations
play is
  -- Give player possibility to make a move.
  do
    current_player.play (place)
    set_selected_place (Void)
    if current_player.next_move /= Void then
      state := Move_state
    end
  end
end

move is
  -- Make the chosen move.
  do
    current_player.move
    if current_player.location = estate_agent.location
      and current_player /= estate_agent then
      state := Agent_caught
    else
      state := Prepare_state
    end
    next_turn
  end
end
To do

1. Classify the features of class GAME according to the feature classification given in the summary. Are they commands or queries?

<table>
<thead>
<tr>
<th>Commands</th>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hints

- The number of rows in the table above is not a hint. There may be more or fewer rows in the final table.

To hand in

Hand in your classification of the features of class GAME.
Assignment 3: Of Objects and Contracts

Hand-out: 1 November 2004
Due: 8 November 2004

This assignment is about classes, features and contracts.
Please solve this assignment alone.

1. Class vs. Objects

Goal
- Understand the difference between the notion of class and the notion of object.

To do
1. Describe in your own words the difference between a class and an object.
2. Find an analogy (see Example).

Sample answer:
2. The class can be looked at as the blueprint of a machine, while the object is the actual machine, which is built according to the blueprint.

To hand in
Hand in your answers to question 1. and 2.
2. Feature reading

Goal
- Discover and understand FLAT_HUNT.
- Get used to EiffelStudio.
- Get used to the .–notation.
- Experiment with the debugging mode.

Summary
*Feature call:*
The fundamental mechanism of program execution: apply a “feature” to an “object”
*Basic form:* `your_object.your_feature`

*Example:* `main_window.show`
Queries don’t “do” anything, but yield a return value, e.g. `estate_agent.displayer` yields an object of type `DISPLAYER`, the estate agent’s displayer.
As this result types are objects themselves, it is possible to call features directly on them, e.g. `estate_agent.displayer.set_visible(True)`.
For the same reason, it is possible to use them as arguments, e.g. `enough_tickets(temp_link.type)`.

*Hint*
- In EiffelStudio, if you press <CTRL> + <SPACE> after typing the name of an object followed by the ‘.’, you will get a list of all the features that can be called on this object (see Figure 1).

- What happens when you additionally press <SHIFT> at the same time?
- To get an overview of FLAT_HUNT, have a look at the FLAT_HUNT documentation, which is located in your TRAFFIC-folder, in subdirectory `example/flat_hunt/doc`.
- You can run your program step-by-step if you set a breakpoint at the beginning of the program. This is done by right-clicking on a feature, then moving the mouse pointer over the icon highlighted in Figure 2 and dropping it there. After launching the application, the execution will stop at the (first) breakpoint. By using the buttons highlighted in Figure 3, you can run the application step-by-step. During this, have a look at the different parts of the window.
This technique of clicking-and-dropping is called pick-and-drop mechanism. It is used extensively in EiffelStudio. Just try for example to 'pick' a class or feature name and 'drop' it in some other (or even the same) part of the window, and see what happens.

**Description**

For each statement, try to figure out what the different return types are and specify for which of the following feature calls (not necessarily from FLAT_HUNT) they could be the argument:

a) put_string (s: STRING)

b) increase_by_one (i: INTEGER)

c) set_state (b: BOOLEAN)
Example:

Question:
0. game.is_game_over in feature idle_action in class MAIN_CONTROLLER

Answer:
0. game: GAME, is_game_over: BOOLEAN; c) set_state

To do
1. game_dialog.number_of_hunters.value in feature start_game_after_dialog in class MAIN_CONTROLLER
2. game.game_displayer.info_text in feature prepare in class MAIN_CONTROLLER
3. game.estate_agent.displayer.visible in feature prepare in class MAIN_CONTROLLER
4. player.location.out in feature print_location in class PLAYER_DISPLAYER
5. player.brain.generating_type.substring (1, 3).is_equal ("HUM") in feature display_after_move in class PLAYER_DISPLAYER

To hand in
Hand in your answers (similar to the example) to tasks 1. - 5.

3. Contracts

Goal
- Use and understand contracts.
- Appreciate the advantages of “Design by Contracts” for debugging.

Summary
A client calling a feature must make sure that the precondition holds before the call. A feature must make sure that, if its precondition held at the beginning of its execution, its postcondition will hold at the end. A class invariant must hold as soon as an object is created, then before and after the execution of any of the features of the class that are available to clients.

Description
You will first overwrite the class player.e with a new version you can download. This new PLAYER class does have a few contracts, however, most of them are wrong, and even more are missing the condition, only the tag is there... Looks like you will have to fix something...
This exercise needs contracts (assertions) to be enabled. Make sure that they are checked in your ‘Project configuration’ (Figure 4, bottom). To open ‘Project configuration’, click on the highlighted button (see Figure 4).

To do
1. Download the file ‘player.e’ from http://se.inf.ethz.ch/....
2. Copy the file in your TRAFFIC directory, subfolder ‘example\flat_hunt\model\player’. Allow overwrite.
3. Start EiffelStudio, open FLAT_HUNT.
4. Make sure that assertions are enabled (see Important above).
5. Recompile (melt) FLAT_HUNT (press ‘F7’).
6. Launch FLAT_HUNT (‘F5’).
7. Correct the wrong assertions.
8. Fill in appropriate contracts where only the tag is given and the condition is missing.
9. Do not forget the class invariants.
Example:

```haskell
8. enough_tickets (a_type: STRING): BOOLEAN is
   -- Check if player has tickets to use this link.
   require
     valid_ticket_type: is_valid_type (a_type)
   do
     ...
   ensure
     only_true_if_tickets_available: Result implies
       (rail_tickets > 0 or tram_tickets > 0 or bus_tickets > 0)
   end
```

Hint

- To get an idea of possible assertions, just have a look at other classes in FLAT_HUNT and TRAFFIC

To hand in

Hand in a print-out of the corrected class.
Assignment 4: Object creation

Hand-out: 8 November 2004
Due: 15 November 2004

Today you are going to write your first stand-alone program. Please create the solution to this assignment alone…

Summary

To create an object:

- If class has no create clause (i.e. uses default_create), use basic form, 
  create x
- If the class has a create clause listing one or more procedures, use 
  create x.make (...) 
  where make is one of the creation procedures, and “(...)” stands for arguments if any.
1. My first application

Goal
- Write your first application from scratch.
- Create and use a new class.
- Learn about basic input/output in Eiffel.

Description
In this exercise you will write your first application from scratch. You will probably find it not that fancy and it cannot do a lot, but 'Hey!', it might be your first complete program...

You have to write an application that can convert temperatures from Celsius to Fahrenheit. The application should consist of two classes, one called TEMPERATURE (the root class), and the other called CONVERTER.

CONVERTER has one feature called to_fahrenheit, which takes an INTEGER and directly prints the calculated result. There is no need to return the calculated value.

TEMPERATURE will have one feature make, which creates the converter, reads an INTEGER and calls the feature to_fahrenheit from class CONVERTER. It also needs a feature converter to have access to CONVERTER.

TEMPERATURE is a client of CONVERTER.

Things you need to know
- To print something on the screen, use io.put... (use <CTRL> + <SPACE> to see the different possibilities).
- To get user input, use io.read... to read input into a buffer, followed by io.last... to access the last read element.
- …and most important, the “magic” formula: Fahrenheit = 9/5 * Celsius + 32.

To do
1. Open EiffelStudio.
2. In the opening dialog, create a new project of type 'Basic application (no graphics library included)'. If this dialog does not appear, select File ➔ New project.
3. Enter the values from Figure 1.
4. Create a new class using the button highlighted in Figure 2.
5. Call the new class CONVERTER (see Figure 3).
6. Create new class TEMPERATURE.
7. Complete the two classes to get the functionality described above.
Figure 1: New project dialog

Figure 2: What you get after compilation
Figure 3 Create CONVERTER class

Example:
This is about how your application should look like:

Remark
- Celsius temperatures have a property that almost shouts for a precondition...
- Try to build your class according to the EIFFEL style rules. A short overview of these can be found in “DO IT WITH STYLE” (http://www.se.inf.ethz.ch/...).

To hand in
Hand in your two classes temperature.e and converter.e.
2. GAME_DISPLAYER

Goal
- Learn how to refactor a class.
- Create and use a new class in FLAT_HUNT.

Description
In the FLAT_HUNT documentation, in Chapter 5, Cluster View, a class GAME_DISPLAYER is described. This class is supposed to display the game statistics, like whose turn it is and whether it is game over or not. The problem is, this class does not yet exist.

The features that should be in this class were put into class GAME in cluster Controller. However, the clustering would be much nicer if the features dealing with visualization were in this class GAME_DISPLAYER in cluster View.

This distinction between Controller and View is used in several places in FLAT_HUNT, e.g. in PLAYER vs. PLAYER_DISPLAYER. Your task will be to apply this pattern also to the class GAME. In the end, it should look like Figure 4.

To do
1. Make a new class GAME_DISPLAYER in Cluster View, Subcluster Game_displayers, which takes the features info_text, display_end_game, display_before_prepare, display_stats and print_flat_hunter_locations away from class GAME in a separate displayer class.
2. Add reasonable contracts.
3. Test your new FLAT_HUNT.

Hint
- GAME_DISPLAYER has to be created somewhere.
- Probably you need a reference to GAME in your new class.
- Maybe you have to change other classes as well.

To hand in
Hand in your class texts of classes game.e and game_displayer.e.
C.5 Assignment 5

Assignment 5: Loops

Hand-out: 15 November 2004
Due: 22 November 2004

Please solve this assignment alone.

Hahaha! This joke in the programming language C would look in Eiffel about as follows:

Summary

```c
int main(void)
{
    int count;
    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

```
initialization

invariant

loop instructions

exit condition
```
1. Where is “Central”?

**Goal**
- Understand loop structure and conditionals.
- Realize the importance of proper stop criterions.

**Description**
This first part intends to convey the importance of choosing the right stop criterions. In the following class text extract we want to loop through a list of places and search for the place called “Central”. When we have found it, the loop stops and we do something to the place (if it was found). The two code extracts below are supposed to do everything as just described.

**To do**
1. For each version decide whether it does what it is supposed to.
2. If you think it is not OK, then correct the errors.
3. Specify exactly what the problem was.

**Note**
- You may assume that all named objects are not Void and have been declared and created outside of these extracts. `set_found` just sets `found` to either True or False.

```
Version a)
from
  places.start
set_found (False)
until
metro places.after or found
loop
  if (places.item.name = "Central") then
    set_found (True)
  else
    places.forth
  end
if (not places.after) then
  "Perform some operations on the found place"
end
```

2
To hand in
This is a pen-and-paper exercise; you do not need to code in EiffelStudio.
Hand in your answers to Task 1, the corrected versions (Task 2) and the description of the bugs (Task 3) of a) and b) (if necessary).

2. Fancy animate_defeat animation

Goal
- Play around with loops and conditionals.
- Be creative and make FLAT_HUNT look nicer.

Description
In the class PLAYER_DISPLAYER in FLAT_HUNT, there is a feature called animate_defeat. This feature gets called on the estate agent when the hunters find him, or on all the hunters if the agent can escape. Up to now, animate_defeat just draws a black circle. There is a loop prepared, but for now, the loop is empty.

Your task is to fill this loop and try to make a nice animation whenever the game is over. This could for example look like Figure 1. However, instead of circles, you might also want to use lines or rectangles, and maybe play with colors. Probably it gets even cooler if you add some conditionals as well…

Figure 1 Example animate_defeat animation
Note

- Make sure that ‘loop assertions’ in ‘Project configuration’ are enabled.
- Have a look at the comments directly in the source code.

To do

1. Fill in the loop in feature `animate_defeat` in class `PLAYER_DISPLAYER`.

To hand in

Hand in your version of feature `animate_defeat` plus a screenshot of your animation.

3. Loop painting

Goal

- Try nested loops.

Description

You can use loops within loops to display certain figures like the following:

![Figure 2 Checkered triangle with size 8](image.png)

To do

1. Write a program that asks the user to input a value, and then displays a checkered triangle of the given size as in Figure 2. Be aware that stars and white space should be alternating.

To hand in

Hand in a print out of your class text.
Assignment 6: “equal” = “equal”?

Hand-out: 29 November 2004
Due: 6 December 2004

We hope that’s not the case with you…
This exercise is not only about equality, but also about object structures and, for the first time, inheritance.
Please solve this assignment alone.
1. Value vs. reference, deep vs. shallow

Goal
- Understand the difference between value and reference.
- Understand the difference between deep and shallow.

Summary

An object is made of a number of values called fields.

A value is either an object or a reference.

It is an object if it is an expanded type (e.g. INTEGER).

A reference is either Void or attached to an object.

Example:

Eiffel offers the following commands:

- \( x = y \) compares two references
- \( \text{equal} \ (x, y) \) compares two objects (shallow comparison)
- \( \text{deep_equal} \ (x, y) \) deep comparison of objects \( x \) and \( y \)
- \( x := y \) assigns \( x \) to the object denoted by \( y \) (reference assignment)
- \( x \text{.copy} \ (y) \) shallow copy of \( y \) on object \( x \)
- \( x := \text{clone} \ (y) \) creates new object attached to \( x \) as a shallow copy of \( y \) (calls \( \text{copy} \))
- \( x := \text{deep_clone} \ (y) \) creates a new object attached to \( x \) as a deep copy of \( y \)

This is how the object structure before and after some commands looks like:
And here is an analogy:
Consider a HTML-page. You can compare the normal text with objects, and the links
with references.
Now, if you just save the web page, only the current page will be saved, the links will
still point to the same pages in the WWW.
However, there are some programs that allow you to make something like a deep copy,
where (down to a certain level) also the referenced pages are downloaded, and the links
are changed to the local copy of these references pages.

Description
This is a multiple choice exercise. First, make sure that you understood the theory above,
and then try to answer questions 1 to 5.

To do
1. Suppose that the instruction x := clone(y) has just been executed successfully,
   which of the following statements are true?
   (a) equal(x, y)
   (b) x = y
   (c) deep_equal(x, y)
2. Suppose that the instruction `x.copy(y)` has just been executed successfully, which of the following statements are true?

(a) `equal(x, y)`  
(b) `x = y`  
(c) `deep_equal(x, y)`

3. Suppose that the instruction `x := y` has just been executed, which of the following statements are true?

(a) `equal(x, y)`  
(b) `x = y`  
(c) `deep_equal(x, y)`

4. Is the statement in the title of this assignment ("equal" = "equal")...

(a) ... True?  
(b) ... False?

Explain why!

**To hand in**

Hand in your multiple choice answers to questions 1 to 4, and the reason for your answer in question 4.

### 2. Object structure

**Goal**
- Understand how objects are linked together.

**Description**

This exercise expects you to draw a picture of the run-time object structure when playing `FLAT_HUNT`. It should help you to realize how objects are connected with other objects.

**To do**

1. Open the `FLAT_HUNT` project.  
2. Open class `ESTATE_AGENT` and set a breakpoint on feature `choose_move`.  
   (Check assignment 3 if you forgot how to do this.)  
3. Start `FLAT_HUNT`.  
4. When `FLAT_HUNT` stops at the breakpoint, draw a picture of the current object structure. Only follow the references down to a level of one indirection.  
5. When you have finished the painting, answer the following questions:  
   a) What is an expanded type?  
   b) Are there any expanded types in your picture? Which?
6. a) What is the difference of ":=" for expanded types / normal types?
b) What is the difference of "equal" for expanded types / normal types?

Example
This could be the beginning of your drawing if the breakpoint was set in class GAME:

Hint
- There might be backward-references.

To hand in
Hand in your picture of the object structure, and your answers to questions 5 and 6.

3. Inherited fraction

Goal
- Inherit from a class.
- Use infix/prefix notation.

Summary
Just an example from the lecture:

```plaintext
class POLYGON
  create
  make

  feature
    vertices: ARRAY [POINT]
    vertices_count: INTEGER
    perimeter: REAL
      do
        from ... until ... loop
          Result := Result + (vertices @ i).distance (vertices @ (i + 1))
        end
      end

  invariant
    vertices_count >= 3
    vertices_count = vertices.count
end
```
Description

NUMERIC is a deferred class in the library EiffelBase that exports the following features:

- one
- zero
- divisible
- exponentiable
- infix "+"
- infix ","
- infix "/"
- infix "*"
- prefix "+"
- prefix "+"

Your task is to implement class FRACTION (\texttt{enumerator}) inheriting from NUMERIC.

The following test class should work with your implementation without any changes:

```eiffel
class FRACTION_TEST
create make
feature -- Initialization
  a, b, c: FRACTION
make is
  do
    -- Test the class FRACTION.
  end
end
```

class RECTANGLE inherit
  POLYGON
redefine perimeter
end
create make
feature
  diagonal, side1, side2: REAL
  perimeter: REAL is
    -- Perimeter length
    do
      Result := 2 * (side1 + side2)
    end
invariant
  vertices_count = 4
end

Description
NUMERIC is a deferred class in the library EiffelBase that exports the following features:

- one
- zero
- divisible
- exponentiable
- infix "+"
- infix ","
- infix "/"
- infix "*"
- prefix "+"
- prefix "+"

Your task is to implement class FRACTION (\texttt{enumerator}) inheriting from NUMERIC.

The following test class should work with your implementation without any changes:

```eiffel
class FRACTION_TEST
create make
feature -- Initialization
  a, b, c: FRACTION
make is
  do
    -- Test the class FRACTION.
  end
end
```

class RECTANGLE inherit
  POLYGON
redefine perimeter
end
create make
feature
  diagonal, side1, side2: REAL
  perimeter: REAL is
    -- Perimeter length
    do
      Result := 2 * (side1 + side2)
    end
invariant
  vertices_count = 4
end
```
create a.make (1, 2)  
create b.make (3, 4)

io.put_string ("Calculating with fractions:" + "\n")  
io.put_string ("a : " + a.out)  
io.put_string ("b : " + b.out)

c := a + b  
io.put_string ("a + b : " + c.out)

c := a - b  
io.put_string ("a - b : " + c.out)

c := a * b  
io.put_string ("a * b : " + c.out)

c := a / b  
io.put_string ("a / b : " + c.out)

end

You can download the source of this class from [http://www.se.inf.ethz.ch](http://www.se.inf.ethz.ch)/...

To do
1. Create a new project with root class FRACTION_TEST.
2. Copy and paste the class above in the root class.
3. Create a new class, inherit from NUMERIC and implement the missing features.

Hint
- To reduce a fraction, you can use one Greatest Common Divisor (GCD) algorithm, for example the Euclidian algorithm. Try to find this one on the web and adapt it to Eiffel.
- In Eiffel, integer division is done with \\
  ; integer remainder (modulo) with \\.
- Have a closer look at class FRACTION_TEST if you don’t know where to go.

Remark
- Do not forget contracts. This example does again have a very obvious invariant…

To hand in
Hand in the full source of your class fraction.e.
Assignment 7: More Inheritance

Hand-out: 6 December 2004
Due: 13 December 2004

{ASSIGNMENT_6} Precursor -- Please solve this assignment alone.

1. Dynamic exercise

Goal

- Understand the effects of dynamic binding.

Summary

For a description of classes POLYGON and RECTANGLE have a look at Assignment 6.
Assume:

\[ p: \text{POLYGON}; \ r: \text{RECTANGLE} \]
\[ x: \text{REAL} \]

Permitted:

\[ x := p.\text{perimeter} \]
\[ x := r.\text{perimeter} \]
\[ x := r.\text{diagonal} \]
\[ p := r \]

NOT permitted:

\[ x := p.\text{diagonal} \text{ (even just after } p := r \text{ !)} \]
\[ r := p \]
Description
Consider the following inheritance hierarchy:

And the corresponding class texts:

```ruby
class BOOK
  create
  make
  feature -- Initialization
  make in
    do
      -- Initialize book.
    end
  feature -- Output
  print_book is
    do
      io.put_string("This is a book.%N")
    end
end

class TEXTBOOK
  inherit BOOK
  rename
    print_book in print_textbook
  redefine
    print_textbook
  create
  make
  feature -- Output
  print_textbook is
    do
      io.put_string("This is a textbook.%N")
    end
end

class COMICS
  create
  make
  print_book
  print_book
  print_textbook
  print_comics
  print_textbook++
  print_comics++
```

```
ETHZ D-INFK  Introduction to programming – Exercises
Prof. Dr. B. Meyer  Winter 2004-2005

Description
Consider the following inheritance hierarchy:

And the corresponding class texts:

```ruby
class BOOK
  create
  make
  feature -- Initialization
  make in
    do
      -- Initialize book.
    end
  feature -- Output
  print_book is
    do
      io.put_string("This is a book.%N")
    end
end

class TEXTBOOK
  inherit BOOK
  rename
    print_book in print_textbook
  redefine
    print_textbook
  create
  make
  feature -- Output
  print_textbook is
    do
      io.put_string("This is a textbook.%N")
    end
end

class COMICS
  create
  make
  print_book
  print_book
  print_textbook
  print_comics
  print_textbook++
  print_comics++
```
Examples:

Question 1: Is the following code valid? Explain why or why not.

\[ b: \text{BOOK} \]
\[ \text{create} \ b.\text{make} \]
\[ \text{b.print_book} \]

Answer 1: Yes, because \( b \) is of type \( \text{BOOK} \) and class \( \text{BOOK} \) has a feature \( \text{print_book} \).

Question 2: The code presented in question 1 is valid. What message is printed when executing this code?

Answer 2: “This is a book.”

To do

1. Is the following code valid? Explain why or why not.

\[ b: \text{BOOK} \]
\[ \text{create} \ (\text{TEXTBOOK}) b.\text{make} \]
\[ b.\text{print_book} \]

2. Is the following code valid? Explain why or why not.

\[ b: \text{BOOK} \]
\[ \text{create} \ (\text{TEXTBOOK}) b.\text{make} \]
\[ b.\text{print_textbook} \]
3. One of the code samples presented in question 1 or 2 is valid. What message is printed when executing this code?

4. Is the following code valid? Explain why or why not.
   
b: BOOK
   t: TEXTBOOK
   create t.make
   b := t
   b print_book

5. Is the following code valid? Explain why or why not.
   
b: BOOK
   t: TEXTBOOK
   create t.make
   b := t
   b print_textbook

6. One of the code samples presented in question 7 or 8 is valid. What message is printed when executing this code?

7. Is the following code valid? Explain why or why not.
   
b: BOOK
   c: COMICS
   create {COMICS} h.make
   c := h
   c print_book

8. Is the following code valid? Explain why or why not.
   
b: BOOK
   c: COMICS
   create {COMICS} h.make
   c := h
   c print_comics

9. One of the code samples presented in question 7 or 8 is valid. What message is printed when executing this code?

To hand in
Hand in your answers to questions 1 to 9.
2. From Client/Supplier to Multiple Inheritance

Goal
- Adopt the FLAT_HUNT player-brain mechanism so that it uses multiple inheritance.

Description
Up to now, the player-brain mechanism in FLAT_HUNT used the so-called “client-supplier” relationship (see Figure 1). Your task is to change this into a relationship using multiple inheritance. So the different players should not “use” the brain anymore, but just “have” a brain, and directly access its features. For a hint how it might look like, have a look at Figure 2.

To do
1. Make use of multiple inheritance in the player-brain mechanism.
   You may not only have to add new classes, but probably also change several existing classes. Make sure that FLAT_HUNT (including contracts) still works.
2. Draw a BON-diagram (similar to Figure 1) of the new situation.
3. Answer the following questions:
   a) What is the advantage of the new relationship?
   b) What are the drawbacks?
   c) Which version do you prefer? Why?

To hand in
Hand in all your new classes (the changed ones are not needed), the BON-diagram and your answers to questions a) to c) of task 3.
Assignment 8: Genericity

Hand-out: 13 December 2004
Due: 20 December 2004

For the last time (at least in this course, and except for the exam):
Please solve this assignment alone.
1. Santa needs help

Goal
- Learn how to use lists of lists.
- Have a first glimpse at recursion.

Description
Santa needs your help: Since he is very short on time this year, but still wants to make happy as many children as possible, he decided to only visit large metro stations in Paris. He wants to make a tour passing these stations as fast as possible. Since he is curiously following the Introduction to Programming lectures over the Internet and saw the promising TOUCH_APPLICATION described in A touch of class, he thinks that we can help him with his plan.

To make things easier for us, he agreed on the following definition of large metro stations: a metro station is large if it has more than 8 incoming or outgoing connections.

Your task is to do the following: Find the large metro stations and the shortest route to visit all of them, and display and animate it. Santa does not care about the order in which the single stations are visited, he just wants to be done as fast as possible.

To do
1. Generate a new subcluster santa of cluster exercises in your TOUCH_APPLICATION. Download the file shortest_path.e found under http://se.inf.ethz.ch/teaching/ws2003/37.001/downloads/santa and put it into your newly created cluster.
2. Find the large stations that Santa wants to visit (see definition of large metro stations above).
3. Use feature Map.calculate_route to calculate the shortest path for the set of large metro stations. Map.calculate_route uses Map.station_selection to find out which stations or landmarks are to be visited by the route. Note that the order of the elements in Map.station_selection is the order in which the route visits these places. So the order does matter to the route calculation algorithm. In order to find the fastest route you will need to try all possible permutations of the large metro stations. For this, a feature permute in class SHORTEST_PATH has been prepared. The shortest path is defined as the route that has the least route segments.
4. Highlight and animate the fastest route by using the features Map.display_route to highlight and Map.play to animate the route.

Hint
- If you are curious about what the feature permute exactly does, have a closer look and/or ask your assistant.
To hand in
Hand in the class text of SHORTEST_PATH.

2. Current = next.previous

Goal
- Implement a more complex data structure.
- "Play around" with references.
- Use genericity.

Description
This exercise has two parts. In the first part, you have to implement a data structure called doubly linked list. In the second part, you have to add genericity to the list.

The doubly linked list consists of two classes: INTEGER_LIST_CELL and INTEGER_LIST. It looks as follows:

- INTEGER_LIST_CELL holds an INTEGER as the cell content and has a previous and a next reference to an INTEGER_LIST_CELL.
- INTEGER_LIST has a reference to the first and the last cell of the list.

Additionally, INTEGER_LIST should provide at least the following features:
1. extend (val: INTEGER) -- Append a integer list cell with content 'val' at the end of the list.
2. prune (val: INTEGER) -- Remove the first cell that has value 'val'.
3. search (val: INTEGER): INTEGER_LIST_CELL -- Search for the first element with content 'val', return 'Void' if no cell is found.
4. count: INTEGER -- Number of cells in the list.

Class INTEGER_LIST_CELL is given:

Indexing
description: "Cells holding an integer value, used for doubly linked lists"
class INTEGER_LIST_CELL
create (INTEGER_LIST)
set_value
feature -- Access
  value: INTEGER
    -- Content that is stored in the list cell
  next: INTEGER_LIST_CELL
    -- Reference to the next integer list cell of a list
  previous: INTEGER_LIST_CELL
    -- Reference to the previous integer list cell
feature {INTEGER_LIST} -- Element change
  set_value (x: INTEGER) is
    -- Set 'value' to 'x'.
    do value := x
       ensure value_set: value = x
    end
  set_next (el: INTEGER_LIST_CELL) is
    -- Set next to 'el'.
    do next := el
       ensure next_set: next = el
    end
  set_previous (el: INTEGER_LIST_CELL) is
    -- Set previous to 'el'.
    do previous := el
       ensure previous_set: previous = el
    end
invariant
  next_linked: next = void or else next.previous = Current
  previous_linked: previous = void or else previous.next = Current
end

Also given is a small TEST_APPLICATION:

class TEST_APPLICATION
create
make
Your first task will be to write the class INTEGER_LIST with all necessary features. The classes INTEGER_LIST_CELL and TEST_APPLICATION should not be changed.

The second task is to rewrite all the three classes and make the list generic.

**To do**

**Part I**
1. Download the sources for classes INTEGER_LIST_CELL and TEST_APPLICATION from [http://www.sc.inf.ethz.ch/...](http://www.sc.inf.ethz.ch/...).
2. Create a new project, copy and paste these two classes.
3. Write the new class INTEGER_LIST.
4. Test your application.

**Part II**
5. Extend all the classes to support not only INTEGERs. You may now change classes INTEGER_LIST_CELL and TEST_APPLICATION.
6. Test your application. You may now also change class TEST_APPLICATION.

**Hint**
- Some additional "helper" features might be useful.
- The output of Part I should look like Figure 2.
To hand in

Hand in the “generic” class text of all the three classes (INTEGER_LIST,
INTEGER_LIST_CELL, TEST_APPLICATION), as well as the output you get with
your modified class TEST_APPLICATION.
C.9 Final Project

Please do not solve this assignment alone. Actually, you should do the project in groups of three. Try to form uniform groups concerning your programming experience, so that you can choose a task that is equally challenging for all the three of you.

1. The project(s)
As you will spend a considerable amount of time on the final project, we want you to be very happy with it. Therefore, we give you the choice between three different tasks. And each of these tasks also gives you lots of freedom.

Option A: FLAT_HUNT extension
By now, you have done several exercises with FLAT_HUNT, and probably you were annoyed about something, or you desperately missed a certain game feature. The time has come where you can extend FLAT_HUNT and give it your personal touch. Possible ideas are:

- Improve the artificial intelligence
- Add new game modes
- Make FLAT_HUNT net-playable (…and add a chat-client)
- Add special cards: e.g. to block a player
- Add new types of transport: e.g. jetpack
- Make FLAT_HUNT look nicer: e.g. add a small overview map
- Make FLAT_HUNT real-time
- …
Option B: Game of Life
The Game of Life is not a game in the conventional sense. There are no players, and no winning or losing. It is more like a simulation that runs according to some specified rules. To find out more about the Game of Life, visit the very interesting webpage http://www.math.com/students/wonders/life/life.html. Your job is to design an Eiffel version of Game of Life. Whether you build a nice GUI (Graphical User Interface) or make it text-based is completely up to you.

Option C: I've always wanted to...
As the title suggest, you can do almost whatever you want. 'Almost' just means that your assistant has to agree with what you have in mind, and that your idea is worked out as clearly as they have to be in options A and B.
If you intend to do something that involves graphics and multimedia, you might want to check out ESDL. ESDL is a wrapper for SDL, the Simple DirectMedia Layer library that is very popular among Linux game developers. Find more information about ESDL on http://eiffelsdl.sourceforge.net/.

Another important point: Your project must not have any offensive content, and it should not use copyrighted material.

2. The tasks

a) Analysis and Design
Regardless which option you took, before you start to hack your application, we want you to sit together with your group and discuss how to best model your system.
- What is the goal?
- What are the requirements?
- How should the final system look like?
- What are the advantages/drawbacks?
- How do the classes work together?
- Who of your group is doing which parts?

You are supposed to write a short report describing your system and your design decision. This report should include a BON diagram of your classes. You have to e-mail this document to your assistant until Sunday, 16 January 2005.

b) Implementation and testing
Realize the system you designed. Make sure that your code is readable, well-commented and equipped with contracts.
Then test your system by writing down some use cases explaining how the system should react and make sure that it actually reacts as expected.
In addition to the source code, we want you to write four (short) documents:
- A user guide describing how to use the application.
- A developer guide describing your system.
- A test report describing your tests and results.
- An experience report describing your teamwork, what you learned and what difficulties you encountered.

3. The presentation
In the last exercise session (31 January 2005), you have to present your application. The audience will be your exercise group and your assistant. Your presentation should take about 10 minutes, and you are not only to show the final release of your system, but also to talk about your design decisions and your experiences during this project. Every group member should get the chance of saying something.
4. The grading criteria

4.1 Design (25%)
- Soundness
- Extendibility
- Ease of use

4.2 Quality of contracts (15%)
- Preconditions
- Postconditions
- Class invariants

4.3 Documentation (20%)
- User and developer guide, test and experience report

4.4 Test (10%)
- Correctness of the application (Does it often crash?)

4.5 Quality of code (10%)
- Style guidelines
- Quality of code

4.6 Effort devoted to the project (20%)
- Not much to say here…

5. To hand in

16 January 2005
Hand in the description of your system (Task 2. a) by e-mail.

31 January 2005
Hand in your implementation in electronic form (ACE-file, e-files and additional data files), together with user guide, developer guide, test report and experience report and presentation. Please make sure that you:
- Do not include the EIFGEN directory, nor the following files: .epr, .rc, .wb
- Include the ACE-file, all .e-files, and any other relevant files (pictures…)
- Make all filenames (including extensions) lowercase, without white spaces, and only using forward slashes (/)
- Create a directory called “/doc” and put there all your documentation as PDF
- Remove all absolute paths in your code
- If you had to change library classes, document all changes
6. Final remark
This exercise gives a lot of freedom concerning the amount of work you spend. You may improve and upgrade the functionality of your application to any extent you want. Feel free to use your imagination! There are no limits. The main goal is to program and gain experience in extending/developing larger systems.
Acknowledgments

I am grateful to Michela Pedroni for being an excellent supervisor, giving me lots of freedom yet helping me when I needed it. I thank Prof. Bertrand Meyer for giving me the chance to do my master thesis in the area of teaching. I am also grateful to Prof. Werner Hartmann for his helpful inputs, and to Till Bay for his ideas and help. Last but not least I thank my parents for supporting me - not only financially - through all my studies.
References


