Software Systems — Draft version

(Course code: 201500111)
Module 2 of the TCS and BIT curricula
University of Twente

Christoph Bockisch
Twan Coenraad
Frans van Dijk
Maarten Everts
Martijn Hoogesteger
Marieke Huisman
Sophie Lathouwers
Luís Ferreira Pires
Renate van Luijk
Arend Rensink
Laurens Rouw
Klaas Sikkel
Jip Spel
Leon de Vries

2016 – 2017

Contents

Introduction 1
Overview of the Module . . . 1
Learning Objectives . . . 2
Modes of Instruction . . . 3
Project Groups and Lab Groups . . . 3
Study Material and Systems . . . 3
Tests and Grades . . . 4
Mandatory Activities . . . 6
Signing off Mandatory Exercises . . . 7
Teachers and Assistants . . . 7
Monitoring and Evaluation of the Module . . . 8

Project Descriptions 9
Design project . . . 9
Case Description: Barchester City Council Parking System . . . 9
What to hand in . . . 11
Submission and grading . . . 12
Programming project . . . 13
Rules of the Board Game . . . 13
Global Description of the Game Application . . . 13
Communication Protocol . . . 14
Functional Requirements of the Application . . . 14
Guidelines for the Report . . . 14
Packaging for Submission . . . 16
Minimal Requirements for the Project . . . 17
Project Activities . . . 18
Important Dates . . . 18
Possible Extensions of the Application . . . 19
Grading . . . 20
Grading Criteria . . . 20

1 Week 1 23
1.1 Overview . . . 23
1.1.1 Mandatory Presence . . . 23
1.1.2 Expected Self-Study and Project Work . . . 24
1.1.3 Materials for This Week . . . 24
1.1.4 Tool Installation Session . . . 24
1.2 Academic Skills . . . 24
1.2.1 Assignments . . . 24
1.3 Design . . . 25
1.3.1 Project . . . 25
1.3.2 Laboratory session 1a (Activity Diagrams) . . . 25

1.3.3 Recommended exercise 1a (Activity Diagrams) . . . 27
1.3.4 Laboratory session 1b (Use cases) . . . 28
1.3.5 Recommended exercise 1b (Use Cases) . . . 32
1.4 Programming . . . 35
1.4.1 Laboratory exercises . . . 35
1.4.2 Recommended exercises . . . 40

2 Week 2 **41**
2.1 Overview . . . 41
2.1.1 Contents of This Week . . . 41
2.1.2 Mandatory Presence . . . 42
2.1.3 Expected Self-Study and Project Work . . . 42
2.1.4 Materials for this Week . . . 42
2.2 Academic Skills . . . 42
2.2.1 Assignments . . . 42
2.3 Design . . . 43
2.3.1 Laboratory session 2a (Class Diagrams) . . . 43
2.3.2 Recommended exercise 2a (Class Diagrams) . . . 46
2.3.3 Project . . . 47
2.3.4 Laboratory session 2b (Sequence Diagrams) . . . 47
2.3.5 Recommended exercise 2b (Sequence Diagrams) . . . 52
2.4 Programming . . . 53
2.4.1 Laboratory exercises . . . 53
2.4.2 Recommended exercises . . . 57
2.4.3 Bonus exercises . . . 58

3 Week 3 **59**
3.1 Overview . . . 59
3.1.1 Contents of This Week . . . 59
3.1.2 Mandatory Presence . . . 59
3.1.3 Expected Self-Study and Project Work . . . 60
3.1.4 Materials for this Week . . . 60
3.2 Academic Skills . . . 60
3.2.1 Peer feedback session . . . 60
3.3 Design . . . 60
3.3.1 Laboratory session 3a (State Machine Diagrams) . . . 60
3.3.2 Recommended exercise 3a (State Machine Diagrams) . . . 62
3.3.3 Project . . . 63
3.3.4 Laboratory Session 3b (Versioning) . . . 63
3.4 Programming . . . 66
3.4.1 Java 8: Default interface methods . . . 66
3.4.2 Laboratory exercises . . . 66
3.4.3 Recommended exercises . . . 69
3.4.4 Bonus exercises . . . 69

4 Week 4 **71**
4.1 Overview . . . 71
4.1.1 Contents of This Week . . . 71
4.1.2 Mandatory Presence . . . 71
4.1.3 Expected Self-Study and Project Work . . . 72
4.1.4 Materials for this Week . . . 72
4.2 Academic Skills . . . 72
4.2.1 Assignments . . . 72
4.3 Design . . . 73
4.3.1 Laboratory session 4 (Software Metrics) . . . 73
4.3.2 Project . . . 74

4.4 Programming . . . 74
4.4.1 Laboratory exercises . . . 74
4.4.2 Recommended exercises . . . 77
4.5 Mathematics: Euclides, Leibniz, and Newton in Computer Science . . . 77
4.5.1 Mathematics for BIT Students . . . 77
4.5.2 Mathematics for TCS Students . . . 78

5 Week 5 **79**
5.1 Overview . . . 79
5.1.1 Contents of This Week . . . 79
5.1.2 Deadline . . . 79
5.1.3 Mandatory Presence . . . 79
5.1.4 Expected Self-Study and Project Work . . . 80
5.1.5 Materials for this Week . . . 80
5.2 Design . . . 80
5.2.1 Test . . . 80
5.2.2 Project . . . 81
5.3 Programming . . . 81
5.3.1 Laboratory exercises . . . 81
5.3.2 Recommended exercises . . . 84
5.3.3 Bonus exercises . . . 85

6 Week 6 **87**
6.1 Overview . . . 87
6.1.1 Contents of This Week . . . 87
6.1.2 Mandatory Presence . . . 87
6.1.3 Expected Self-Study and Project Work . . . 88
6.1.4 Materials for this Week . . . 88
6.2 Academic Skills . . . 88
6.2.1 Peer feedback session . . . 88
6.3 Programming . . . 88
6.3.1 Laboratory exercises . . . 88
6.3.2 Project . . . 94
6.3.3 Recommended exercises . . . 95
6.3.4 Bonus exercises . . . 95

7 Week 7 **99**
7.1 Overview . . . 99
7.1.1 Contents of This Week . . . 99
7.1.2 Mandatory Presence . . . 99
7.1.3 Expected Self-Study and Project Work . . . 100
7.1.4 Materials for this Week . . . 100
7.2 Programming . . . 100
7.2.1 Laboratory exercises . . . 100
7.2.2 Recommended exercises . . . 108
7.2.3 Project . . . 110
7.2.4 Bonus exercises . . . 111

8 Week 8 **115**
8.1 Overview . . . 115
8.1.1 Contents of This Week . . . 115
8.1.2 Mandatory Presence . . . 115
8.1.3 Expected Self-Study and Project Work . . . 115
8.2 Programming . . . 116
8.2.1 Project . . . 116

9 Week 9 **117**
9.1 Overview . . . 117
9.1.1 Contents of This Week . . . 117
9.1.2 Mandatory Presence . . . 117
9.1.3 Expected Self-Study and Project Work . . . 117
9.2 Programming . . . 118
9.2.1 Project . . . 118

10 Week 10 **119**
10.1 Overview . . . 119
10.1.1 Contents of This Week . . . 119
10.1.2 Mandatory presence . . . 119
10.1.3 Expected Self-Study and Project Work . . . 119
10.2 Programming . . . 120
10.2.1 Project . . . 120

A Java Modeling Language **121**
A.1 JML Method Contracts . . . 121
A.2 Class Invariants . . . 125

Introduction

This is the manual used during the module "Software Systems" (M1.2) of the BSc curricula for Computer Science (TCS) and Business & IT (BIT). The manual contains information about the organisation of the module and (especially) the exercises and assignments to be done.

Overview of the Module

Overall Purpose In this module you are introduced to the design, implementation and testing of software systems, and to carrying out larger projects independently. For the design of software systems, you learn to use Software Engineering models, particularly the UML diagrams (use case diagrams, activity diagrams, class diagrams, sequence diagrams and state machine diagrams), and you get acquainted with the waterfall software development processes. For the programming of software systems, you learn the core concepts of structured programming, object-orientation and multi-threading with the help of the Java programming language, with attention to correctness by means of preconditions and postconditions. For testing software systems, you learn to distinguish among the different levels at which testing can be performed (specially unit testing and system testing), the principles underlying a test plan, and some relatively simple testing techniques. Attention is also given to elementary project management skills and academic skills (planning and self-management).

The contents of this module builds upon the knowledge of algorithms and recursion acquired in Module 1.1.

Position in the Curriculum This module is offered in the second period (Quarter 1B) of the Computer Science (TCS) and Business & IT (BIT) Education Programmes. Module 1.1 ("Pearls of Computer Science" resp. "Introduction to BIT") is a prerequisite for this module, since this module builds upon the introduction on programming (imperative and functional, for algorithms and recursion, respectively) learned there. This module is itself a prerequisite for Module 1.4: Data & Information, which builds upon the design and programming skills learned here.

Threads This module consists of four threads, namely:

- *Design (D)*: methods and techniques for the high-level design of software systems.
- *Programming (P)*: methods and techniques for the programming and testing of software systems.
- *Academic skills (A)*: techniques for project management, planning, time- and self-management, and reflection on one's own behaviour with regard to planning.
- *Mathematics (M)*: the so-called 'Newton' package, which covers the theory of mathematical functions and integrals.

In this module, you perform two projects, one in the Design thread (D-project) and one in the Programming thread (P-project).

Even though the content is divided over four different threads, there is a close connection between them, since skills from one thread are necessary later in other threads.

Software Systems as Minor, Pre-Master etc. If you are not a student of the regular TCS or BIT Education Programme, there are various possibilities to follow parts of the module:

- *Software Systems as Minor Module (12 EC)*: with in addition an extra mathematics course: *Introduction to Mathematical Analysis (3EC, 201400385).*
- *Design and Programming (12 EC)*: only the Design, Programming and Academic Skills threads.
- *Design theory and project (4 EC)*: only the Design and Academic Skills threads.
- *Programming theory and project (8 EC)*: only the Programming threads.

Learning Objectives

Concerning Software Design, after successfully finishing this module you are capable of:

- Specifying an existing software system or a software system under design in terms of UML models (including class diagrams, activity diagrams and state machines);
- Interpreting these models, and explaining the relation between different models and between each model and the software code, as well as the usefulness of defining models in addition to writing software code;
- Explaining the commonly recognised phases of software development;
- Applying version management in software development projects;
- Explaining basic software metrics and using them to assess quality characteristics of a code base

Concerning Programming, after successfully finishing this module you are capable of:

- Explaining and applying the core concepts of imperative programming, such as variables, data types, structured programming statements, recursion, lists, arrays, methods, parameters, and exceptions;
- Explaining and applying the core concepts of object-orientation, such as objects, classes, values, types, object references, interfaces, specialisation/inheritance and composition;
- Using the Model/View/Controller pattern when developing applications;
- Writing simple multi-threaded programs, explaining the operation and problems (race conditions) of concurrent threads, and using synchronisation mechanisms such as monitors, locks and wait sets;
- Writing programs using basic network mechanisms, based on sockets;
- Explaining and applying the basic concepts of security engineering and applying them to Java programs;
- Writing software of average size (around ten classes) in Java, by using the concepts mentioned above, including the use of algorithms for searching and sorting data;
- Documenting software of this size, by using JML-based preconditions, postconditions and (class) invariants, and (informally) justifying the correctness of the implemented software;
- Explaining how this software can be tested, defining and executing a test plan, and measuring and improving test coverage.

For academic and project skills, attention is given to planning, time- and self-management, and reflection on one's own behavior with regard to planning. After successfully finishing this module you are capable of:

- Describing the major principles of effective time management;
- Describing personal strengths and weaknesses regarding the execution of these major principles with regard to study behavior;
- Applying these principles to make a personal planning for a shorter period (a week) and a medium long-term period (for instance a module or a medium-sized project);
- Giving and receiving peer feedback;
- Identifying major personal pitfalls concerning procrastination behavior;
- Applying several techniques and tricks in overcoming procrastination behavior.

Concerning Mathematics, after successfully finishing this module you are capable of:

- Working with limits and the definitions of continuity and differentiability and applications;
- Investigating functions in two variables;
- Working with elementary properties of integrals and calculate integrals using different techniques;
- Working with power series and Taylor series.

Modes of Instruction

A number of different modes of instruction are used in this module. Some of them have already been applied in Module 1.1.

Lecture Traditional lecture, where a teacher explains theory and examples. The lecture is intended to provide sufficient context and to motivate why you should study the material. It does not replace the written material. Some lectures also include exercise sessions.

Tutorial You work on exercises and get feedback from a teacher, in groups of 20-30 students.

Laboratory You work on lab exercises supervised by a student assistant, in pairs. The exercises have to be signed off. Participation is mandatory. The rooster.utwente.nl web site calls this *Practical*.

Self-study supervised Whenever this mode of instruction appears in rooster.utwente.nl there is an instructor (teacher or student assistant) present to answer your questions.

Project supervised You work on one of the projects. The D-project (due in Week 5) is performed in groups of four, while the P-project (due in Week 10) is performed in pairs. Whenever this mode of instruction appears in rooster.utwente.nl there is an instructor (teacher or student assistant) present to answer your questions.

Peer feedback You give feedback on each other's work, typically in project groups.

Seminar You have the opportunity to ask questions for exam preparation.

Diagnostic test An exercise (typically done individually) to test how well you are acquainted with the material. Participation is mandatory.

Project Groups and Lab Groups

All lab exercises and the programming project should be done in pairs. The design project will be done in groups of four. The group distributions will be made available on Blackboard during the first day of the module.

Study Material and Systems

For the Design and Programming threads we have selected a book, and a set of software tools. Furthermore some additional material is used to cover some topics not addressed by the book. For the Mathematics thread, the material is listed on the Blackboard organisation for *Mathematics A B C1 D1*. The full list of resources for is as follows:

Academic skills R. van Tulder *Skill Sheets*, Pearson, 2012.
This book has also been used in Module 1.1.

Design For the design thread there is no mandatory book, instead the lecture slides serve as the main course material. Additional handouts are available for the topics Use Cases (Week 1) and Software Metrics (Week 4).

Programming J. Niño and F.A. Hosch. *An Introduction to Programming and Object-Oriented Design Using Java*. Wiley, 3rd edition, 2008.

For Week 2 (Specification and Testing), a short introduction to JML is provided in Appendix A.

For Week 7 (Concurrency and Networking), the material is discussed in:

Chapter 14, until *BlockingQueues* (p. 819 - 877) from C.S. Horstmann and G. Cornell, *Core Java, volume I: Fundamentals*. Prentice Hall, 9th edition, 2012.

Chapter 3, until *Making URL Connections* (p. 185 - 210) from C.S. Horstmann and G. Cornell, *Core Java, volume II: Advanced Features*. Prentice Hall, 9th edition, 2012.

Both *Core Java* books are available in the university library and in the InterActief room.

For the lectures on security, relevant material is discussed in:

Chapter 5 from Ross Anderson, *Security Engineering*. Wiley, 2nd edition, 2008. Available for free at `http://www.cl.cam.ac.uk/~rja14/book.html`.

Manual This manual contains all relevant information about the projects and exercises for the A, D and P threads. For information about the Mathematics thread we refer to the Blackboard organisation for *Mathematics A B C1 D1*, with the exception of the special session in Week 4 that is about the use of Mathematics in Computer Science and Business Information Technology.

Per week this manual contains the following information:

- The contents and relevant material;
- An overview of mandatory activities;
- An estimate of the required self-study effort;
- Mandatory laboratory exercises;
- Self-study exercises to prepare for the lectures;
- Recommended additional exercises that can be solved in self-study sessions; and
- Challenging bonus assignments.

The project descriptions are given in a separate chapter, preceding the material per week.

Additional material on JML is provided as an appendix of the manual.

In the margins of the programming exercises, we sometimes refer to relevant parts of the study material, which could help you to make the exercises. In this marginnote, we use N & H to refer to J. Niño and F.A. Hosch, *An Introduction to Programming and Object-Oriented Design Using Java*, and CJ to refer to the chapters in the Core Java books by Horstmann and Cornell.

Solutions to selected exercises Answers to the *recommended exercises* of threads D and P will be made available on Blackboard.

Software Systems Used. The following tools are used in this module:

- Eclipse (`http://eclipse.org`) as a general tool for implementing and testing Java programs.
- Modelio (`http://www.modelio.org`) as a general tool for drawing UML diagrams.
- OpenJML (`http://jmlspecs.sourceforge.net/`) for the type checking of software specifications.
- EMMA (`http://emma.sourceforge.net/`, `http://www.eclemma.org/`) as a standalone tool and an Eclipse plugin for test coverage.
- Tools for automatically checking programming code style, such as, for example, CheckStyle (`http://eclipse-cs.sourceforge.net/`).
- Metrics Eclipse plugin (`http://metrics.sourceforge.net/`) to extract some metrics from programming code.

Information about the material necessary for the Mathematics thread can be found on the Blackboard organisation for *Mathematics A B C1 D1*.

Tests and Grades

There are some conditions that have to be fulfilled to be allowed to participate in the tests:

- to be allowed to participate in the Mathematics test, you have to attend at least 10 out of the 12 tutorial and supervised self-study sessions;
- to be allowed to participate in the Design test, all Design lab exercises should be signed off; and
- to be allowed to participate in the Programming test, all Programming lab exercises should be signed off.

To successfully complete this module you have to:

- participate in all mandatory activities (see below);
- sign off all mandatory exercises, i.e.,

  - submit answers to all academic skills assignments via Blackboard;
  - sign off all exercises during the lab sessions;
- have a minimum of 5.5 for two of the tree tests (Mathematics, Design, and Programming) and a minumum of 5.0 for the third test;
- have a minimum of 5.5 for the average of the three tests;
- have a minimum of 5.5 for each of the projects (Design and Programming).

If the above criteria are satisfied, the final grade of this module is determined by the average of the following results (all having the same weight):

- Mathematics test grade (individual);
- D-project grade (in groups of 4 students);
- D-test grade (individual);
- P-project grade (in groups of 2 students); and
- P-test grade (individual).

An overview of the grading schema for this module is given in Table 1 (copied from the Teaching and Examination Regulations).

Table 1: Grading schema

201500111 Software Systems						
Module part		Type of assessment	Individual / Group	Weight within part (%)	Minimum grade	Weight part (%)
I	Math B2	Written test	I	100	5.5*	20
II	Design	Written test	I	100	5.5*	20
		Assignment	I	Pass		
	Programming	Written test	I	100	5.5*	20
		Assignment	I	Pass		
				Sub-weighted average	5.5*	
	Design Project	Report	G	100	5.5	20
	Programming Project	Product	G	100	5.5	20
		Report	G			
	Academic Skills	Assignment	I	Pass	Pass	0
				Weighted average	5.5	

*Out of the marked (*) module component grades ONE mark lower than 5.5, but at least 5.0 (5.0 = <rate <5.5) is allowed IF it's sub-weighted average is 5.5 or higher.

Amendments to the projects (Design and Programming) necessary to improve the project grade can be handed in before or at the end of Week 10. If needed, you can make amendments to both projects. Note, however, that when an amendment to a project is necessary after the initial deadline, the grade for that project cannot be more than 5.5.

Extra Information about the D- and P-tests An explanation and example tests are available on Blackboard. Both the D-test and P-test are open book. You are allowed to bring:

- a paper version of the module manual;
- a paper copy of the lecture slides; and
- a paper version of the book.

You are *not* allowed to bring the following:

- solutions to recommended exercises as published on Blackboard; and
- your own material (print outs of laboratory exercises, or notes in whatever form).

It is allowed to have highlighted text with a text marker, but it is not allowed to have any hand-written notes in the manual, slides or book.

It is being investigated whether it will be possible to do the P-test on a laptop. More information about this will be provided during the module.

Grading schema for Students that take Software Systems as Minor, Pre-Master etc. If you are not following the regular TCS or BIT Education Programme, we have the following grading schemas:

- *Software Systems as Minor Module (12 EC)*:
  - participate in all mandatory activities other than mathematics;
  - sign off all mandatory exercises (as above);
  - have a minimum of 5.0 for each of the tests (Design and Programming);
  - have a minimum average of 5.5 for the tests (Design and Programming); and
  - have a minimum of 5.5 for each of the projects (Design and Programming).

  The final grade is determined by the average of the following results:
  - D-project grade (in groups of 4 students);
  - D-test grade (individual);
  - P-project grade (in groups of 2 students); and
  - P-test grade (individual).

  In addition, you follow separately the mathematics course *Introduction to Mathematical Analysis (3EC, 201400385)*.
- *Design and Programming (12 EC)*: following only the Design, Programming and Academic Skills thread. The same grading schema as for the minor module applies, but without the additional mathematics course.
- *Design theory and project (4 EC)*: following only the Design and Academic Skills thread.
  - participate in all mandatory activities for the design and academic skills thread;
  - sign off all mandatory exercises for the design and academic skills thread;
  - have a minimum of 5.5 for the Design test; and
  - have a minimum of 5.5 for the Design project.

  The final grade is determined by the average of the following results:
  - D-project grade (in groups of 4 students) and
  - D-test grade (individual).
- *Programming theory and project (8 EC)*: following only the Programming thread.
  - participate in all mandatory activities for the programming thread;
  - sign off all mandatory exercises for the programming thread;
  - have a minimum of 5.5 for the Programming test; and
  - have a minimum of 5.5 for the Programming project.

  The final grade is determined by the average of the following results:
  - P-project grade (in groups of 2 students) and
  - P-test grade (individual).

Mandatory Activities

For the following activities in the module, presence and active participation is mandatory:

- All academic skills peer feedback sessions (Weeks 3 and 6);
- The mathematics case session (Week 4);
- At least 10 out of the 12 mathematics tutorial and supervised self-study sessions;
- All diagnostic tests (Weeks 3, 6 and 7);
- The following programming sessions, which are related to the programming project:
  - The project session on global design in Week 6,

- The project session on the group protocol in Week 7,
- the peer feedback sessions in Weeks 8, 9 and 10.

In the description for each week, you can find more information about the purpose of these sessions.

If you have a good reason why you cannot attend a mandatory session, you should always contact the module coordinator about an alternative.

For the academic skills lectures in Week 1, 2 and 4, your presence is expected. All mandatory academic skills exercises are explained and can be (at least partly) completed during these lectures.

Signing off Mandatory Exercises

To sign off the mandatory exercises, the following procedure is used:

- For academic skills the exercises that have to be handed in on Blackboard are:
  - A-1.1 Personal planning for the second week of the module (deadline: Sunday Week 1, 23.59 CET);
  - A-2.1 Questionnaire on procrastination behavior (deadline: Sunday Week 2, 23.59 CET).

  Other exercises for academic skills will be discussed and signed off during the peer feedback sessions, so you are expected to bring them with you.
- For design, you do the exercises in *in pairs*. Some exercises are marked with a ☞ symbol. This indicates that you should ask a student assistant for feedback when you have finished this exercise. If it is (nearly) OK, the assistant will sign off the exercise. If important aspects of the solution are missing or suboptimal, you will be asked to improve your design.
  When you come to the lab sessions well prepared (i.e., you have digested the contents of the lecture) it should be doable to finish all exercises during the session. However, if some exercises have not been signed off, please finish them at home. Ultimately, all design exercises should be signed off a week after the lab session. Pending exercises can be signed off at the beginning of a next lab session, but please do not spend the lab sessions working on exercises from past sessions. Each lab session is dedicated to one particular topic. It is important that you work on *that* topic during the lab session, so that you prepare yourself for the tasks in the design project.
- For programming,you do all exercises *in pairs*. Some exercises are marked with a ☞ symbol. These indicate *sign off points*. If you are finished with the marked exercise, you can ask a student assistant to check your solutions of the preceding exercises (including the marked exercise). All exercises for Week n should be signed off at the Monday of Week $n+1$, except for week 4, which should be signed off at the end of week 4. The lab sessions on Monday are especially dedicated to signing off; during the other practical programming sessions, questions will be given priority over signing off.

In principle, the pairs for the design and the programming lab sessions should be the same.

It is very important to sign off all exercises within the allocated time. If you fail to do so, you might not be allowed to take part in the tests.

Teachers and Assistants

The module coordinator is Marieke Huisman. She can be reached by mail (M.Huisman@utwente.nl), and her office is Zilverling 3039. She is not available on Wednesdays. The contact details of the other teachers and student assistants and their individual responsibilities can be found on the Blackboard site of this module.

For questions about the contents, the exercises, or the projects, you can always ask the assistant at the next lab session.

Monitoring and Evaluation of the Module

If there are problems with the organisation, the programming environment or the manual, please contact the responsible teachers and/or the module coordinator.

During the module, there will be several intermediate evaluation sessions, during the break immediately following a lecture. The dates and times for these evaluation sessions will be announced during the first lecture of the module. Additionally, you will be asked to fill in a questionnaire in Week 5 and Week 9.

Project Descriptions

This chapter describes the projects that you have to carry out in the course of the module Software Systems. The Design Project is due in Week 5, the Programming Project is due in Week 10.

Design project

Make a design for a medium-sized system. It is expected that you do this project assignment in teams of four students.

In this project you make a design by means of UML diagrams for a software system for parking houses. There are multiple business processes to be analysed, a number of use cases to be identified and described, and a nontrivial data model. Most of the information needed is described below. However, some additional information is to be acquired by means of an interview (in Week 2).

On pages 11–12, after the case description, it is explained in detail

- which documents and diagrams to hand in,
- how to package and submit them, and
- how this will be graded

If you have successfully completed a design project in previous years (i.e. you do Software Systems for the second time) you will get a different assignment. The task is to study and compare designs made by different design teams, and use this to create an optimized design for the parking system. A more precise project specification is available as separate document (not included in this manual).

Case Description: Barchester City Council Parking System

Barchester City Council operates seven car parks in the centre of Barchester[1]. The Council has a requirement for a new system to control its car parks. This system must provide for the day-to-day operation of each car park—issuing tickets, handling payment and controlling barriers—and the management of car parks—recording problems, issuing season tickets and monitoring service level agreements with the security company that guards the car parks.

The car park operational system controls entry to and exit from a car park and payment for car parking.

There are two types of users: ordinary customers, who pay for their use of each car park at the time they use it, and season ticket holders, who pay a fixed amount in advance for parking for three, six or twelve months in a specific car park. Season ticket holders are allocated parking spaces in designated areas that are not available to ordinary customers from Monday to Friday. Season tickets are for weekdays only; the designated spaces are available to all customers at week-ends. No more than 10% of the spaces in a car park are allocated to season ticket holders.

[1]This case description was written by Bennett, McRobb, and Farmer, as additional case study material to their book. The case description in this manual has some minor modifications compared to the original version.

Entry to the car park

When a car approaches an entry barrier, its presence is detected by a sensor under the road surface, and a 'Press Button' display is flashed on the control pillar

The ordinary customer must press a button on the control pillar, and a ticket is printed and issued. The ticket must be printed within five seconds. A 'Take Ticket' display is flashed on the control pillar. If the car park is full, no ticket is issued, and a 'Full' display is flashed on the control pillar. If a vehicle leaves the car park, then the 'Press Button' display is activated again when there is a vehicle waiting.

When the customer pulls the ticket from the control pillar, the barrier is raised.

The season ticket holder does not press the button, but inserts his or her season ticket into a slot on the control pillar. A check is made that the season ticket is valid for this car park and has not expired, that it is a weekday and that the season ticket holder is not recorded as having already entered this car park and not left. If all these checks are passed, then the barrier is raised. The checks must take no longer than five seconds. A record is made of the time of entry for that season ticket holder.

A sensor on the other side of the barrier detects when the car has passed and the barrier is lowered.

The ticket issued to each ordinary customer has a bar code on it. The bar code has a number on it and the date (ddmmyyyy) and time (hhmmss) of entry to the car park. The number, date and time of entry are also printed on the ticket in human readable form.

The details of the ticket are stored: ticket no., issue date, issue time, issuing machine.

The number of vehicles in the car park is incremented by 1 and a check is made against the capacity of the car park. If the car park is full, then a display near the entrance is switched on to say 'Car Park Full', and no further tickets are issued until a vehicle leaves the car park.

Payment

When the ordinary customer is ready to leave, he or she must go to a pay station to pay. The ticket is inserted into a slot, and the bar code is read. The ticket bar code information is compared with the stored information. If the dates or times are not the same, the ticket is ejected, and the customer is told (via an LCD display) to go to the office. In the office, the attendant has a bar code reader and can check a ticket. Typically the problem is damage to the bar code on the ticket, and the attendant can use the office system to calculate the charge, take payment and validate the ticket (see below).

At the pay station, if the ticket dates and times are the same as the bar code dates and times, then the current date and time are obtained, and the duration of the stay in the car park is calculated. From this the car park charge is calculated and displayed on the LCD display. Calculation and display of the charge must take no more than two seconds.

There are two tariffs: a short-stay tariff and a long-stay tariff. Both short-stay and long-stay have higher rates for weekdays from 8.00 am to 6.00 pm, and lower rates for entry after 6.00 pm and at week-ends.

If no change is available, this information is displayed on the LCD display. The customer must then insert notes or coins to at least the amount of the charge. Each note or coin is identified as it is inserted and the value added to an accumulated amount and displayed on the LCD display. Invalid notes are ejected from the note slot. Invalid coins are dropped through into the return tray. A message is displayed on the LCD display.

As soon as the amount accumulated exceeds the charge, the ticket is validated. The current date and time are added to the stored data for that ticket (payment date, payment time).

If the amount entered exceeds the charge and change is available, then the amount of change is calculated and that amount of change is released into the return tray. Otherwise, no change is given. In either case, a message is displayed on the LCD display.

The ticket has the payment date and time printed on it and is ejected from the ticket slot.

A message is displayed telling the customer to press the 'Receipt' button if they need a receipt. If they press this button, a receipt is printed and ejected into the receipt tray. The receipt shows the Council address, address of the car park, VAT number, date and amount paid.

A message is displayed for the customer telling them to take the ticket back to their car and leave the car park within 15 minutes.

Leaving the car park

When the customer drives up to the exit barrier, the car is detected by a sensor, and an 'Insert Ticket' display is flashed on the control pillar. The customer must insert the ticket. The bar code is read and a check is made that no more than 15 minutes have elapsed since the payment time for that ticket. If more than 15 minutes have elapsed, an intercom in the control pillar is activated and connected to the attendant in the car park office. The customer can talk to the attendant, and the attendant can view the details of the ticket on his or her computer. The attendant can activate the barrier remotely, for example if there is a queue to get out and the customer is likely to have been reasonably delayed.

If no more than 15 minutes have elapsed, the barrier is raised. A sensor on the other side of the barrier detects when the car has passed and the barrier is lowered.

The number of vehicles in the car park is decremented by 1 and a check is made against the capacity of the car park. If the car park was full, then the display near the entrance is switched to say 'Spaces', and a check is made to see if any vehicles are waiting. If they are, then the control pillar for the first waiting vehicle is notified. If the driver of the vehicle waiting there does not press the button (for example, because they have backed out and left), then the control pillar for the next waiting vehicle is notified.

At any time, the attendant can view the status of a pay station or a barrier control pillar. Once a connection is made, the status is updated every 10 seconds.

Season ticket holders do not have to go to the pay station, when they are ready to leave the car park, they go to the exit and insert their season ticket into a slot on the exit barrier control pillar. The barrier is raised and a record is made of the time at which the season ticket holder left.

Security visit recording

The City Council has a contract with security companies to visit the car parks at regular intervals. The contract specifies the number of visits per day to each car park and the minimum duration of each visit. Each car park has an office to which the security guards have access. In the office is a card reader similar to the one used for reading season tickets in the control pillars. When a security guard arrives in a car park, he or she puts a card into the card reader and the date and time of arrival is recorded. When the security guard leaves, he or she puts the card in again, and the departure time is recorded. (This card also allows security guards to enter and leave the car park in the same way as season ticket holders. However, this is not used to record the arrival and departure of security guards, as they may not be able to enter with a vehicle if there is a queue of cars at the barrier.)

Currently, the City Council uses two security companies, but could use more or only one in the future. Each security company is issued with a specific number of cards, depending on the number of car parks they are responsible for. Each security company is responsible for specific car parks.

Management functionality

All requirements for operational use have been described above. The City Council would like to obtain further management information from the system. It is not included in this document because the requirements were not finalized at the date of writing. The responsible person from the City Council has invited the IT company to visit him to discuss this.

What to hand in

The final project deliverable should contain the following items.

- A brief report, describing the contents and, if applicable, anything you want to communicate (e.g. why you have made particular design choices). In an appendix you should mention who the team members are and what everybody's contribution to the final deliverable is.
- A report about the interview conducted in Week 2. It should briefly describe the facts (whom you interviewed, when, who conducted the interview, etc.) and give a complete account of the relevant information that you extracted from the interview.
- Appropriate[2] activity diagrams describing relevant business processes (at least 2). [Lecture 1a]

[2] Several items in this list of deliverables mention that you should make *appropriate* choices. What is appropriate? In this context it means that you should choose processes/objects worth modelling, because they are not entirely trivial, and having the model on paper

- A glossary. [Lecture 1b]
- A complete requirements list (with requirements and use cases) [Lecture 1b]
- Use case diagrams for the complete system. [Lecture 1b]
- An actor list. [Lecture 1b]
- Complete (brief) use case descriptions. [Lecture 1b]
- Extended use case descriptions of representative use cases (at least 4). [Lecture 1b]
- A complete class diagram, following the style of the design lectures and exercises. [Lecture 2a]
- Appropriate sequence diagrams of relevant system processes (at least 2). [Lecture 2b]
- Appropriate state machine diagrams (at least 2). [Lecture 3a]

Submission and grading

Packaging and submission

- Create a folder `diagrams` in which you put screenshots (`.png` files) of all diagrams of your project. Please make sure that the file names indicate what the diagrams are about.
- Create a folder `modelio` in which you store the modelio project(s) (exported as `.zip` files) from which the screenshots originate.
- Merge all the text documents into a single PDF document `report-group⟨nr⟩.pdf`
- Package all of the above into a single zip file `d-project-group⟨nr⟩.zip`
- Submit it by means of the Blackboard assignment.

Important Dates

Week 5	Sun 23:59 CET	Submission deadline for maximal project grade
Week 10	Fri 23:59 CET	Submission deadline for maximum 5.5 grade

Grading

If your submission satisfies what is requested (i.e. it contains all the items mentioned under "what to hand in") and your models are roughly OK (not necessarily perfect) you can expect at least a 5.5. In determining the grade, the following quality criteria will be taken into account.

Good quality, increasing your grade (from more important to less important):

- Completeness – you didn't overlook any requirements.
- Appropriate explanations – if you have made particular design choices, can you tell us what and why.
- Appropriate use of specification techniques (like inclusions and extensions in use case diagrams, generalization in class diagrams). Use them where they help, but be aware that too much structure does not help to make a diagrams understandable.
- Appropriate choice of elements to specify (activity diagrams, extended case descriptions, sequence diagrams, state diagrams). There is no point in including them if they are trivial.
- Appropriate choice of names for classes, use cases, etc.

Bad quality, decreasing your grade (from more important to less important):

- Missing stuff—diagrams or tables that were asked for are not included.
- Requirements that are ignored or misrepresented in the design
- Errors in the use of UML—a missing arrow head will not cost you points; systematically forgetting cardinalities in a class diagram will.
- Unclear explanations and badly structured text.
- A badly packaged submission, not following the description above.

Not part of the grading criteria:

- Grammatical correctness (as long as it does not degrade the readability of the report).

helps to understand exactly what the system should do. In other words, as a design effort it makes sense to model them.
An example of what *not* to do: For two state machine diagrams one could model the barrier at the entrance of the car park (having two states: *up* and *down*), and the barrier at the exist of the car park (*idem*). This would not be regarded as a meaningful contribution to the design.

- Graphic quality of the diagrams (as long as they are clearly readable).
- Lay-out of the documents.

In the unfortunate case that your group has to resubmit an improved version, you will receive a list with specific requests that have to be satisfied to acquire a 6.

Programming project

Develop a distributed client/server program to play a board game, and describe the design in a report. It is expected that you make this project assignment in pairs.

This section describes the requirements for the programming project. Purpose of the project is to demonstrate the design and programming skills you have acquired during this module. You will demonstrate this by developing a client/server application to play a board game.

The application should at least provide the following functionality:

- a game server which offers the possibility to play the game;
- a client that can connect with the server and allow a player to participate;
- textual user interfaces (TUIs) for the server and client;
- enforcement of the game rules;
- support for playing the game over the network;
- support for multiple players per game;
- support for computer players.

During Week 6 and 7 there are special sessions dedicated to the project. Participation in these sessions is *mandatory*. During the session of Week 6, you will discuss the global design of the application; and during the session of Week 7, you will design the communication protocol. More information about the purpose of these sessions is given in the corresponding chapters of thie manual.

The special sessions in Weeks 6 and 7 require you to start developing individual components of the project. Make sure that you indeed do this; if you wait until the last weeks of the module before beginning the development of the game, you will probably lack time. Additionally, several peer feedback sessions are planned where you can discuss your progress with other members of your tutorial group.

Rules of the Board Game

The rules of the board game to be implemented will be made available on Blackboard.

Global Description of the Game Application

After starting a client, the user can enter the IP address and port number of the server, and his own name. After entering this information, the client will log on to the server. The client will wait for the server to signal that there is another client logged on. When a second client has logged on, the game can start, or (in case of a multi-player game) the clients can decide to wait for more players. When the game is ready to be started, the server can assign arbitrary colours to the players, or the players can choose their colour in order of arrival.

After a game has started, the server should be ready for incoming requests from new clients that would like to play the game. Thus, it should be possible to play several games simultaneously on the same server.

The game itself could proceed as follows. The player whose turn it is enters a move, taking into account the rules of the game. The client checks the move, and when it is legal, it sends it to the server.

The server also checks legality of the move, and if it is legal, it will send this information to all participating clients, who can then update their internal game state. The turn then moves to the next player, who should again enter a legal move. This procedure proceeds until the rules of the game indicate the game has finished.

Communication Protocol

Your client application should be able to communicate with server applications from other students within your tutorial group, and vice versa. As a consequence, all students within the same tutorial group should use the same *protocol* for client/server communication. The protocol describes which data will be exchanged between the client and the server, and in which order. This data will consist of the moves in the game, and inspection requests for the leader board.

The protocol will be determined during a tutorial group meeting in Week 7. More information about how the protocol can be defined is given in Section 7.2.3.

Remark It is possible that your tutorial group decides on a client/server-interaction than the one discussed above, with a correspondingly different distribution of responsibilities between client and server.

Functional Requirements of the Application

Your application should implement a number of requirements.

For the *server*, the following requirements should be implemented

1. When the server is started, a port number should be entered that the server will listen to.
2. If the port number already is in use, an appropriate error message is returned, and a new port number can be entered.
3. A server should be able to support multiple instances of the game that are played simultaneously by different clients.
4. The server has a TUI that ensures that all communication messages are written to `System.out`.
5. The server should respect the protocol as defined for the tutorial group during the project session in Week 7, *i.e.*, the server should be able to communicate with all other clients from the tutorial group.

For the *client*, the following requirements should be implemented

1. The client should have a user-friendly TUI, which provides options to the user (*e.g.*, possibility to a enter port number and IP address) to request a game at the server.
2. The client should support human players, and computer players with (some) artificially intelligent behaviour.
3. The thinking time of the computer player (and thus the power of the artificial intelligence) should be a parameter that can be changed via the client TUI.
4. The client provides a *hint* functionality. This shows a human player a possible move, as indicated by the computer player. The move may only be proposed, the human player should have the possibility to decide whether to play this move, or make a different one.
5. After a game is finished, the player should be able to start a new game.
6. If a player quits the game before it has finished, or the client crashes, the other player(s) should be informed, and the game should end cleanly. In this case, the other player(s) should be allowed to register again with the server for a new game.
7. A server might at all times disconnect. The clients should react to this in a clean way, closing all open connections *etc.*
8. The client should respect the protocol as defined for the tutorial group during the project session in Week 7, *i.e.*, the client should be able to communicate with all other servers from the tutorial group.

Finally, as a *global requirement*, the client and the server should always be in the same game state.

Warning It might be possible to find an implementation of the game on the Internet. Copying such an implementation will be considered as fraud, and will be communicated to the Examination Board. At any moment in time, you should be able to explain your own solution to the lecturers.

Guidelines for the Report

The report may be written in English or Dutch. The *target group* of this report is a *software maintainer*, *i.e.*, somebody that did not necessarily write the code himself, but at a later occasion should extend or improve it, and therefore should be able to understand the overall design of your application, understand the purpose and functionality of a class, be able to check whether at least the existing tests still pass, etc..

Discussion of the Overall Design
The design of the application describes the overall structure of the system, in terms of the classes and their relationships. It should describe the following:

1. Class diagrams, with an explanation, for example per package and globally. Make sure your layout reflects the structure of your application, *e.g.*, by repositioning classes, and removing unnecessary details. Feel free to add extra labels and notes.
2. A systematic overview of which part of the requirements is implemented by which classes. If you did not manage to implement all requirements, this should be indicated here.
3. The use of the *Observer* and *Model-View-Controller* patterns.
4. Formats for data storage and communication protocols. It is not necessary to repeat the protocol as defined for the tutorial group, you can simply refer to it.

Discussion per Class
Every class in the application should be discussed by summarising:

1. the role of the class in the system;
2. the responsibilities of the class;
3. the other classes that are used by this class to achieve its responsibilities;
4. if there are any special cases in the class's contract;
5. any precautions taken to fulfill the preconditions in the contract of the server classes.

Test Report
As always, thorough testing of the application is an integral part of its development. You are expected to discuss and hand in both unit tests and system tests.

If you discovered errors in your implementation during testing, and had no time to fix these errors, you should indicate these results in this part of the report.

Unit Testing. Unit tests test the functionality of a class in isolation. Various techniques exist for this purpose:

- Using dedicated test classes, as you have learned during this module. This is the most thorough testing method for unit tests.
- Simulating a part of the system manually via `telnet`, to test communication over a network.
- Adding a `main` method that creates several instances of the class under test, and then invokes several methods on these objects.
- Visual inspection of the UI.

The report on the unit tests should describe for each class separately the following information:

1. how the class has been tested (in isolation, together with other classes, not at all);
2. which test technique has been applied;
3. if appropriate, test programs that have been developed;
4. test results and expected results; and
5. how much percent of each method is covered by tests (the coverage should be determined using Emma). Discuss reasons and consequences of low coverage in a class.

You are allowed to use JUnit for unit testing, but you should clearly document how it has been used.

The test report should provide sufficient information for the reader to repeat the tests. Test programs should be included in the implementation.

System testing. System tests will try out the complete functioning application as a whole; i.e., a system test will typically consist of a complete run-through of a game. The functionality requirements should serve as the basis for these tests, as the implementation should fulfill those. Each requirement may be tested separately by creating one or several test executions with different usage scenarios (both good and bad usage).

The system tests should also be described in the report. Again, describe which aspects of the system have been tested and how. In particular, you should describe:

1. which functionality has been tested;
2. the test execution, in such a way that the reader is able to repeat the test;
3. the expected behaviour of the system; and
4. the actual behaviour of the system (if different from the expected behaviour).

Metrics report
Include a brief section describing the values of the various software metrics you have learned about during this module, with an analysis of the consequences for the quality of your code.

Reflection on Planning
In Week 7, you are asked to make a planning for the project, and to discuss this with a student assistant. During the last weeks of the module, you should keep track of how your planning corresponds with your actual progress, and adapt your planning if necessary.

In your report you should reflect on this, and in particular you should describe the following:

1. How was your planning influenced by your experiences with the planning and time writing during the Design project?
2. To what extent did your planning correspond to the actual progress during the project weeks? What made you deviate from your planning? For example:
   - Were there tasks within the project that took significantly more or less work than you had expected? If so, how come?
   - Were there significant losses of time or momentum in your projects (one or more periods in which you just did not seem to make the progress you had intended)? Looking back, how can you explain this?
3. Which countermeasures did you take to compensate deviation from your original planning? What was the impact of this on the intended scope or quality of the project?
4. What did you learn from this experience for your next (project) planning? Take your answers for answers for question 2 and 3 into account and ask yourself how you would want to prevent this or deal with this a next time.
5. Imagine you are next year's student assistant for this project. Please describe at least two dos and don'ts that you intend to tell your students to help them with their planning.

This reflection paragraph also should be uploaded in your academic skills portfolio.

Packaging for Submission

The implementation must be handed-in as a single ZIP archive. To create this archive, use EXPORT... from the FILE menu in Eclipse. Next, select ARCHIVE FILE from the GENERAL category. Select your project and press finish. The ZIP archive should have the following content and structure:

- All self-defined classes should be stored in a single directory hierarchy.
- There should be a README file with information about installation and starting the game, indicating for example which directories and files are necessary, and conditions for the installation. After reading this file, a user should be able to install and execute the game without any problem. The README file should be located in the root folder of the project.
- Documentation of all self-defined classes (as Javadoc-generated HTML) in a directory hierarchy separate from the source files.
- Any non-standard predefined classes and libraries should be included in `jar`-files.

Typical causes that make the installation and compilation procedure fail are names and paths or hard-coded URLs. **Test this before submitting your project.**

If your program contains references to the file system, e.g., to load image files, make sure to be platform independent. If may be that your application will be executed under a different operating system than you used during development. For further information, see the documentation for `java.lang.File` and in particular the constants `pathSeparator` and `separator` defined in this class.

Minimal Requirements for the Project

This paragraph provides a check list with minimal requirements for your implementation and your report. If you fulfill these requirements, your project will be graded with at least a 5.5.

Minimal Requirements for the Implementation

The implementation should satisfy the following requirements:

- The application should implement all the functional requirements discussed in the previous paragraph.
- The program should compile without any problems.
- The implementation should make use of the *Model-View-Controller* pattern and the *Observer* pattern.
- All self-defined classes must be in self-defined packages, the classes should be organized in at least three different packages.
- The code must follow a good coding style, including layout and variable naming. Specifically, it must not produce any checkstyle warnings (see below for details).
- Result values of methods should not be used to encode error states; instead, exceptions and exception handling should be used, and all exceptions explicitly thrown in own code should be self-defined.
- For the three most complex classes in the server (which can also be classes that are shared with the client; see below for a definition of "most complex"): the classes and all their methods should be documented in Javadoc, with JML pre- and postconditions and class invariants (JML specifications must type-check with OpenJML).
- The server should be multi-threaded to support multiple concurrent games being played.
- There should be test classes and test runs. The tests should try different situations described in the functional requirements and game rules (e.g., the specified port for the server is free or in use; different number of players; or playing an illegal move)
- For the three most complex server classes (see below), the test coverage should at least be 50%, determined using Emma.

Checkstyle Warnings You should download the checkstyle configuration from Blackboard (go to COURSE MATERIAL and find the checkstyle configuration file as well as an instruction for importing it in the `Tools` item). Each violation of a check in the configuration will be reported in the Eclipse PROBLEMS view as a warning. As type of the warning "Checkstyle Problem" will be specified. Your project should not contain any such warnings produced by checkstyle.

Three Most Complex Classes Extensive documentation and test coverage (as detailed above) should be provided for the three most complex self-defined classes in the server application. The server application is formed by all classes that are necessary to run the server, some of these classes may be shared with the client application. To determine the complexity of classes, you should use the Eclipse Metrics tool and in particular the provided metric "Weighted Methods per Class" (WMC).

Minimal Requirements for the Report

The report should describe the following:

- A discussion of the overall design of the application:
  - Document the use of the Model-View-Controller pattern: which classes and packages play the roles of model, view and controller?
  - A class diagram of the contents of the package holding your model classes (according to the Model-View-Controller pattern). The class diagram should show all public methods and fields, fields with a type that is contained in the class diagram should be represented as association.
  - For each functional requirements of the server application, the report should contain a discussion how it is implemented and in which class(es).
- A clear description of formats for data storage and communication.
- For each self-defined class:
  - The responsibilities of the class in the system. I.e., which functional requirements does it implement? If applicable, which rule(s) does it implement? Which role does it play in the implementation of the Model-View-Controller or Observer pattern (if any)?

  - For all classes that it refers to (i.e., as a field type, super type, argument type, local variable type or by calling its constructor) you should describe the purpose of using this class.
- For each of the three most complex server classes:
  - Which precautions must be taken to fulfill the preconditions in the contract of the class.
- A report of the unit tests for each of the three most complex server classes:
  - Besides the class under test, which other self-defined classes are executed during each test.
  - The expected results for each test together with an explanation of the expected result (e.g., refer to the game rules or functional requirements), as well as the actual result when running the test.
  - A discussion of the test coverage of the class under test. Explain whether you consider the reached level of coverage high or low. For those parts of the code that are not covered, you should also discuss why they are not covered (e.g., why was it difficult to write tests that cover them).
- A reflection on your planning and self-management during the project:
  - How was your planning influenced by your experiences with the planning and time writing during the Design project?
  - How much did your planning correspond to the actual progress during the project weeks?
  - What changes did you make to your planning?
  - What did you learn from this experience for your next (project) planning?

It is *not* necessary to copy your source code or your Javadoc-generated documentation in the report.

Project Activities

Planning Exercise P-7.1 asks you to write a planning for this project. To successfully complete the project, you should at least plan during which periods you are going to work on *designing*, *implementing*, *documenting code*, *testing*, and *writing the report*. Moreover, for these tasks, and in particular for the implementation and report writing tasks you should think of sub-tasks and plan these as well.

The planning should be discussed with a student assistant. He or she might give some suggestions how to adjust your planning, if your planning is unrealistic. Remember that the final goal of the project is to create a working program and a decent report. The student assistant has also done this during his/her first year, therefore it is important to take his feedback into account.

During the last weeks of the module, the student assistant can help you to adjust your planning, if necessary.

Peer Feedback In Weeks 8, 9, and 10, peer feedback sessions are planned, where you can discuss your progress so far. In the sections on the project for this week, you can find more detailed guidelines for the peer feedback. The pairing for the peer feedback will be announced via Blackboard.

Remember what you learned about giving feedback during Module 1. Try to judge all components, and where appropriate provide additional details to support your judgement. It is strongly recommended that you discuss the feedback face-to-face.

Remember that fellow students are also doing this for you, so take your review job seriously. Moreover, understanding the approach from other people might also help your own implementation.

Tournament Wednesday afternoon in Week 10, a tournament will be organised where the different computer players will compete against each other. Bonus points can be gained by winning the tournament.

Important Dates

Week	Day	Hour	Activity
Week 6	Thu	2	Kick-off lecture on the programming project
Week 6	Thu	6–7	Tutorial group meeting on overall design
Week 7	Wed	1–4	Discuss project planning with student assistant
Week 7	Wed	7–8	Tutorial group meeting on communication protocol
Week 8	Tue	6	Peer feedback on game logic implementation
Week 9	Tue	6	Peer feedback on complete implementation

Week 10	Tue	6	Peer feedback on documentation
Week 10	Wed	6–9	Tournament
Week 10	Wed	23:59 CET	Submission deadline for maximal project grade
Week 10	Fri	23:59 CET	Submission deadline for maximum 5.5 grade

Possible Extensions of the Application

If you have sufficient time, you can extend your game, following one of the suggestions below. Implementing more extensions makes your project more advanced and gives you a better grade. Each extension can give you 0.4 points extra at most. *However, extensions will only be graded if the application respects the basic functional requirements, i.e., if it is graded with at least a 5.5.*

Note that some of the extensions below also require extending the server and the communication protocol. You should make sure that trying to use the extended functionality with a server from a different team does not lead to a crash.

GUI

You can build a GUI for your game in addition to the TUI. Make sure that your GUI is scalable and properly tested.

Chat Box

When the game application is developed as described above, it can only be used to play the game. It can be fun to have the possibility to communicate with your opponents during the game. Therefore, a possible extension is to add an option to the client UI allowing the user to enter texts. After pressing the return button, the message is sent to the server, who then sends it to the other clients participating in the game, after which the message is shown on all client UIs.

Challenge

Up to now, clients are connected in order of logging on to the server. It would be nicer if a client could choose from the registered clients against who to play. To achieve this, the following changes will be necessary:

- a client should know which other clients are registered;
- a client should be able to choose his opponents;
- a client should be able to refuse a game.

Leaderboard

The server could maintain a leaderboard, providing scoring information about all the games played on the server. It should be possible to retrieve this information in various ways (*e.g.*, overall high score, best score of the day, best score of a particular player).

The leaderboard could provide the following functionality:

- methods to add new scores to the leaderboard database;
- for each score, it should be possible to see at which date and time it is achieved, and by which team (or player);
- methods to inspect the scores, *i.e.*, the top n scores, all scores above a certain value, all scores below a certain value, average score, average score of the day *etc.*; and
- methods to inspect team results, *i.e.*, the best result of a team, the day where the team had the best average results, *etc.*.

The overall performance should be as good as possible, meaning that you have to choose a representation that suits your implementations.

Security

As soon as a leaderboard is in place, players might want to cheat to get to the highest place. In addition, simply accepting any connection leaves one vulnerable to Denial-of-Service attacks. One way to help prevent all this is to properly authenticate clients to the server. Simple password-based approaches are a possibility (but how would you prevent the password from being sniffed on the network?), as well as more advanced signature-based (PKI) approaches (for those who did the cryptography pearl of Module 1).

Grading

Both the implementation and the report will be graded separately with a grade between 1.0 and 10.0. The final grade for the project is the average of the two grades.

The minimal requirements as sketched above are necessary to score 5.5 points overall for the project assignment. If your implementation and/or report does better than the minimum requirements (e.g., you documented more parts, provide proper loop-invariants, provide more extensive tests, or discuss the design of larger parts of your system in the report), you can get up to a maximum 10.0 points.

If you submit your project on Friday, and if it fulfills the minimal requirements, you will score 5.5 points for the project. In this case, no bonus points will be awarded.

Rules for Bonus Points

There are several ways to earn bonus points. Bonus points will be added to the average grade of the implementation and report (see above). It is not possible to earn more than 10.0 points in total.

- The protocol as agreed upon in the tutorial group will be maintained and kept up-to-date by a single student. If this task is fulfilled properly, the student will get a bonus of 0.5 points.
- The winner of the tournament within an tutorial group will get 1.0 bonus points, the runner-up will get 0.5 bonus points.
- For each extension discussed above, you can obtain at most 0.4 bonus points.

All bonus points are added to the overall project result.

Grading Criteria

Finally, we give the list of grading criteria that will be used to grade the project assignment. You can use this list to judge your own progress with the project.

Code

1. The program has all the required components.
2. There is a README file with installation and execution instructions.
3. The program compiles and executes without errors.
4. The program has been sufficiently documented, with Javadoc, and JML pre- and postconditions, and class invariants.
5. The implementation of large and/or complex methods has been documented internally, including the use of loop invariants.
6. The program layout is understandable and accessible.
7. Packages and accessibility is used in a sensible way.
8. The program has a user friendly UI.
9. Bytecode of predefined, external, non-standard Java classes is submitted with the code.
10. All used test classes and test executions are submitted with the code.

Design

1. The overall design is logical.
2. The implementation corresponds to the design in the report.
3. The Model-View-Controller and Observer patterns have been used well.
4. The program is divided in a logical way into classes.
5. All classes have detailed documentation.

Programming Style

1. Names of classes, variables and methods are well-chosen and clear.
2. Code is efficient and neatly implemented.
3. The program is easily maintainable (use of constants, variable names etc.)
4. The tutorial group's communication protocol has been properly implemented.
5. The exception mechanism is used appropriately.
6. Concurrency constructs are used properly.
7. Artificial intelligence for a computer player has been implemented.

Testing

1. Appropriate unit tests are provided.
2. Appropriate system tests are provided.
3. Sufficient test coverage is reached.
4. Tests are well documented.
5. All classes in the system have been tested by unit testing.

Extensions

1. The chat functionality is correctly implemented and documented.
2. The challenge functionality is correctly implemented and documented.
3. A GUI is correctly implemented and documented.
4. The leaderboard functionality is correctly implemented and documented.

Report contents

1. The report has all the required components.
2. The general design is described clearly.
3. The class diagrams have been explained clearly.
4. The choices made in the overall design are well-documented and motivated.
5. There is an overview of which functionality is implemented by which classes.
6. It is described how the Observer and Model-View-Controller patterns are used.
7. Formats for data storage and communication protocols are described clearly.
8. The role and responsibilities of classes is described clearly.
9. The choices in class implementations are well-documented and motivated.
10. The class implementations have been written for the appropriate target group.
11. The connections between classes are described clearly.
12. Special cases in the class's contract are clearly described.
13. Test activities are described clearly.
14. The test activities cover the whole application.
15. The test results have been analysed.
16. The results are written for the appropriate target group.
17. There is a reflection on the planning and its necessary re-adjustments.

Report structure and language

1. The user can quickly see what the text is about.
2. The text is logically structured.
3. The necessary information can be found easily.
4. The language in the report is understandable for the target group.
5. Spelling is correct and consistent.
6. Sentences are well-constructed.

Week 1

1.1 Overview

Getting Started with the Tools For this module, everybody should have a working tool environment. Therefore, on the very first day of the module there is a special tool installation session. After this session, everybody is assumed to have a working tool environment. The tool installation session will be followed by a first programming lab session.

Academic Skills This week contains one lecture dedicated to academic skills: a 1-hour session on Thursday. During this meeting the general content and structure of the sessions devoted to academic skills shall be presented.

You will receive guidelines on making the planning for the self-monitoring schedule. This assignment needs to be completed by the end of the week and handed in on Blackboard.

Design The activities in this week cover the following topics

- Project: getting started, see 1.3.1.
- *Activity Diagrams*. Lecture 1a gives a general introduction to Design and UML and introduces *Activity Diagrams*. We use these, for now, to describe (business) processes in the real world. See 1.3.2 for the corresponding exercises in Lab Session 1a.
- *Use cases*. Use cases describe the functionality of a system at a very high level from the perspective of the user. Discovering use cases includes interviewing prospective stakeholders, in order to find out which functionality is desired. Lecture 1b introduces use cases as well as interview techniques. See 1.3.4 for exercises in Lab session 2a.

Programming This week the following topics will be covered:

- From Python to Java
- Classes, objects, and variables in their diverse appearances

1.1.1 Mandatory Presence

During the following activities, your presence is mandatory.

- Tool installation session (Mon 6–7)
- (M) Self-study supervised (Tue 6 – 7)
- (M) Tutorial (Thu 3 – 4)

1.1.2 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 1 hour to make the academic skills exercises;
- 4 hours self-study for mathematics;
- 2 hours self-study for the design thread; and
- 2 hours self-study for the programming thread.

1.1.3 Materials for This Week

Academic Skills No reading material for this week.

Tool installation

A document describing the tools and how to install them can be found on Blackboard under "Course Materials".

Design

- Slides from lectures 1a and 1b
- Excerpts of Chapters A1 and A2 of S. Bennett, S. McRobb, R. Farmer: *Object-Oriented Systems Analysis and Design using UML*, McGraw-Hill, Maidenhead, UK, Fourth Edition, 2010.
- Van Tulder, Skill Sheets, D4 (recommended further reading: D2, D3, D6)

Programming

Lecture Niño & Hosch, Chapters 0–4

Laboratory The following predefined files are provided on Blackboard:

- `ss/week1/Password.html`
- `ss/week1/hotel/Guest.html`
- `ss/week1/hotel/Room.java`
- `ss/week1/test/DollarsAndCentsCounterTest.java`
- `ss/week1/test/PasswordTest.java`
- `ss/week1/test/GuestTest.java`
- `ss/week1/Dlab-1a-modelio.zip`
- `ss/week1/Dlab-1b-modelio.zip`
- `ss/week1/Dlab-1b-TheatreTickets/ActorsList.rtf`
- `ss/week1/Dlab-1b-TheatreTickets/Glossary.rtf`
- `ss/week1/Dlab-1b-TheatreTickets/RequirementsAndUseCases.rtf`
- `ss/week1/Dlab-1b-TheatreTickets/UseCaseDescriptions-brief.rtf`
- `ss/week1/Dlab-1b-TheatreTickets/UseCaseDescriptions-extended.rtf`

1.1.4 Tool Installation Session

A document with installation instructions can be found on Blackboard under "Course Materials".

You are kindly requested to download the tools as described in the document *before* the session. If everybody would try to download all at the same time, chances are that the wireless network cannot cope with this.

To see if your installation works properly, make introductionary exercises D-1.1, and P-1.1 – P-1.2.

1.2 Academic Skills

1.2.1 Assignments

A-1.1 Planning for Week 2

Fill in your planning for the next week using the appropriate columns (P) of the self-monitoring sheet before the start of Week 2. Take care that you schedule your obligations first. This is the so called "hard

landscape" of your planning. Secondly, think about the amount of time you need for physical care (for instance adequate sleep, meals, showering). After that you can fill the gaps with the types of activity that are applicable to your situation.

To make it easier to compare and discuss your planning with others during the peer feedback session, please use these colors:

- Sleep – grey
- Scheduled study obligations – green
- Independent (self) study – yellow
- Domestic chores (cooking, cleaning, administration) – light blue
- Work (e.g., weekend job) – dark blue
- Meals and other self-care – red
- Travel – orange
- Active recreation (sports, going out, student society, cultural activities) – purple
- Independent relaxation (reading, television, surfing, gaming) – pink

Feel free to add other colors if you have clusters of activities that take up a fair amount of your time, that are not mentioned above.

You have to submit your answer **at the latest Sunday of Week 1 at 23:59 CET** via Blackboard..

1.3 Design

1.3.1 Project

D-P.1 The purpose of this first project hour is to meet your team, to clarify what is expected of you, and to do some planning.

- Study the description of the design project and the case description of the Barchester parking system (page 9). For which business processes would it be useful to make an activity diagram, in order to acquire a good understanding of the details of the process? (but wait with drawing the diagrams until you have done lab session 1a).
- Next week you are expected to interview a representative of the Barchester City Council, see Section 2.3.3. How to conduct interviews is one of the subjects of lecture 1b, but the sooner you get into contact with the Council representative, the better, if you want to have the interview at a time that suits your group.

1.3.2 Laboratory session 1a (Activity Diagrams)

The laboratory session is done in groups of two students.
Whenever you have completed an exercise marked ☞, please ask a student assistant for feedback.

D-1.1 Introductory exercise to Modelio.

In the lab files for this week on Blackboard, you will find the file `Dlab-1a-modelio.zip`. Save this file, open Modelio, import the project, and open it. You will find a incomplete version of the activity diagram shown in Figure 1.1. Add the missing branch (called *Decision* in Modelio), activity (called *Action* in Modelio), and merge, and add/adapt the control flows so that it is equivalent to Figure 1.1.

In the following series of exercises we will make an activity diagram of the information gathering process of the police, described below[1].

[1] *Disclaimer*: The texts below are based on a case study in one of the regional police forces in the Netherlands before a major national reorganization took place, which started in 2013. Procedures could be different now.

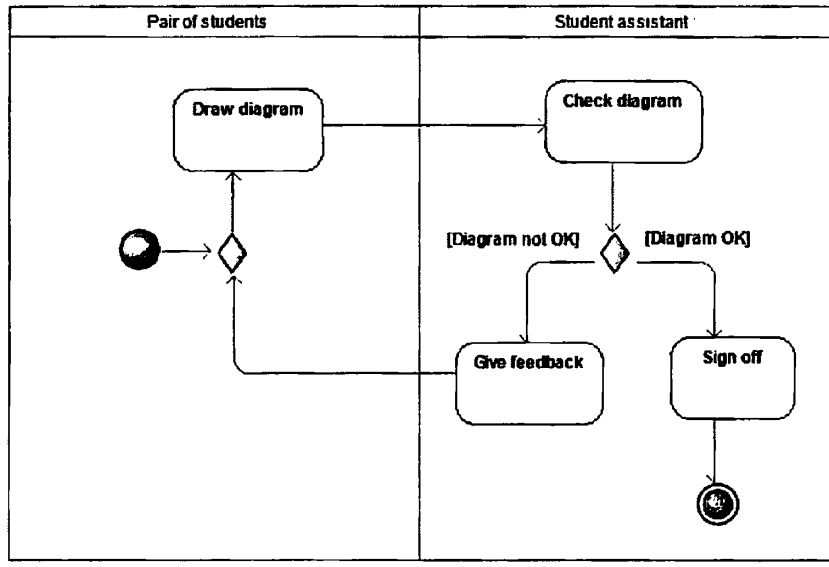


Figure 1.1: Activity Diagram for introductory exercise

Information gathering by the police

The Polderland Regional Police often have difficulties satisfying information requests from the Ministry of the Interior and Kingdom Relations in The Hague. All information is stored in the Police Business System (PBS), but the difficulty lies in retrieving the information from the system. There is no problem when it concerns standard information, which has to be provided at regular intervals. But sometimes there are requests concerning new topics of interest. As an (imaginary) example, the Ministry might suddenly have an increased interest in crimes related to racial tensions and ask for statistics on the last five years. For incidents that happened in the past, it may not always be clear whether they were linked to racial tensions, because it wasn't considered from that perspective at the time. Past incidents are not always tagged appropriately according to current political interests.

The search functions in PBS could possibly be improved to retrieve more information, since the system is more than twenty years old. But before a project is initiated to improve things, it make sense to investigate exactly which procedures are used by the Police force and PBS.

The purpose of PBS is to register all facts about incidents. An incident is any situation or event that calls for police involvement in some way. Incidents can be (telephonic notifications of) crimes and accidents; offences observed by police officers; civilians who come to the police office to report thefts, lost properties found in the street which someone brings to the police, etc., etc.

A record of an incident is called a "mutation" in PBS. It is a confusing term (a mutation can be updated and still be the same mutation), and no one knows why, in a distant past, it was called that way. But everybody uses the term so we'll stick with it.

D-1.2 Create an (initial) activity diagram of the process of information gathering by the Police that includes the following facts:

- If someone phones the police, they are connected to the incident room. If it really is an incident, then the incident room officer has to do two things: create a mutation in PBS and send an officer to the scene of the incident. What is done first is up to the incident room officer (who may decide one way or another, depending on the circumstances).
- In fact a mutation consist of two parts: a basic part that is filled in by the incident room officer, and an additional part with a report of the police officer who visited the scene. Obviously, the latter part is filled in sometime later, when the police officer has the time and occasion to report.

☞ **D-1.3** Extend the activity diagram by incorporating the following:

- Sometimes a police officer on duty observes an offence (which has not been reported to the incident room), and takes appropriate action. In that case the police officer fills in both parts of the mutation.

☞ **D-1.4** Extend the activity diagram by taking the following into account:

- Many police officers care a lot more for their primary tasks than for administration. To ensure that incident registration is up to standards, the Polderland Regional Police has created a Data Management department. Employees in this department—we call them data managers—inspect mutations for completeness. They can ask police officers to improve their reporting. (For this task they have access to other sources of information which are not stored in PBS, including the police officers' daily reports.) Mutations are often updated in the days after the incident, so Data Management inspects mutations two weeks after their creation. If a mutation is not up to standards, a data manager has two options. Sometimes the data manager improves the mutation him/herself and notifies the police officer. In this way, new officers, who don't have much experience with this type of reporting, learn what is expected of them. It is hoped they will do better next time. Alternatively, the data manager can send a request to the police officer to improve the mutation. The police officer should then do it themselves.

D-1.5 If you have finished Exercise D-1.4 before the end of the session, please continue with the following extension. You don't have to sign this off, but you should ask a student assistant for feedback if you have found the time to do this exercise.

- In Exercise D-1.4 it is assumed that police officers always comply when they are asked to improve a mutation. Some officers are quite stubborn, however. The improvements are sometimes okay, sometimes still poor quality, and sometimes not made at all. Therefore, if a police officer has been asked to improve a mutation, Data Management carries out another inspection two weeks later. However, if it is still not good enough, Data Management does not have the authority to sanction the police officer. What is can do, though, is notify the manager of the police officer.

1.3.3 Recommended exercise 1a (Activity Diagrams)

D-1.6 Make an activity diagram for X-rays in the hospital, according to the case history described below.

Radiology administration

In the Polderland regional hospital, despite the overwhelming number of computer systems, not all processes have been automated yet. A process that is still largely paper-based is making appointments for medical examinations. For example: for a request for radiography (X-ray photographs), a paper form has to be filled in. A medical specialist in another department, or a doctor from outside the hospital (typically a general practitioner) fills in the radiography request form and gives it to the patient. The patient contacts the secretariat of the Radiology department and makes an appointment at a convenient time.

Making the requests electronic, rather than paper-based, will increase both the efficiency and the reliability of the process, according to the hospital management. In the current way of working, information has to be copied manually from the paper form into the hospital's information system. This is inefficient and, more importantly, it can lead to errors. Therefore, a new procedure has been defined. Your task is to make a specification in the form of an Activity Diagram of the process that is described below.

A few introductory remarks:

- Different medical examinations may have different procedures. To keep things concrete and simple, the case description is limited to making X-ray photographs.

- Also, we ignore the fact that different parts of the procedures described below will be embedded in different systems which are already present in the hospital. The purpose here is to clarify the process steps themselves, not the systems in which these steps will be embedded.

Radiographic examination
A radiographic examination consists of a set of X-ray photographs and a report by a radiologist about the findings in the photographs. In the new process, requesting and carrying out a radiographic examination is done as follows.

- A medical specialist in the Polderland hospital usually requests a radiographic examination during a consultation with the patient. From within the patient's electronic record, the specialist needs can open a request form with a single mouse click. The patient's essential data will be included automatically. The specialist adds a reason for the request.
- After the request has been filed by the medical specialist, the patient should make an appointment for the examination. A patient can visit or phone the Radiology secretariat for an appointment. A secretary will record an appointment in the system for a time that suits the patient. (*We ignore special cases where the request is urgent and perhaps the patient is incapacitated, then the medical staff will make an immediate appointment on their behalf. You don't have to model that.*)
- General practitioners in the region can also make a request for a radiographic examination of a patient. In this case the request will be made electronically as well, but chances are that the patient data are not as complete as the hospital would like to have them.
  Consequently, if a patient contacts the Radiology secretariat, the secretary should do two things: make an appointment for the patient and check whether the patient data are complete. If not, ask the patient for the missing data and enter them into the system.
- At the appointed time the patient checks in at the Radiology secretariat. A secretary places a patient on the work list for that day. Usually it takes 10-15 minutes until it is the patient's turn, in exceptional cases it could take a bit longer.
  Unfortunately, it happens that patients do not turn up. If a patient hasn't checked in one hour after the appointed time, a letter is generated automatically, asking the patient to make a new appointment. Later that day a secretary prints the letter and sends it by (physical) mail to the patient.
- The X-rays are made by a radiology assistant. The assistant positions the patient so that the right body part will be photographed from the right angle and then hides behing a protective cover for making the X-ray. When all reqested X-rays have been made in this fashion, the assistant inspects the X-rays. If one or more are not good enough, e.g. because the patient moved, these X-rays these will be replaced by new X-rays.
- The most important step in the examination is that one of the radiologists will study the X-rays and write a report. Because it their area of specialization, radiologists may see things that would be overlooked by other doctors. As the radiologist writes the report directly into the system, the report (with the X-rays attached) can be sent automatically to the doctor who requested them. The requesting doctor will (look at the X-rays and) read the report from the radiologist. The process ends here, it is up to the doctor how en when to inform the patient of the findings.

1.3.4 Laboratory session 1b (Use cases)

The laboratory session is done in groups of two students.

☞ **D-1.7** Figure 1.2 gives a very simple use case diagram for a part of the Police case study from Section 1.3.2. You find the diagram in the Modelio project `Dlab-1b-modelio.zip` in this week's lab files on Blackboard. Please extend it with the following information, making use of ≪include≫ and ≪extend≫ relations between use cases.

- If someone want to improve or inspect a mutation, they first have to search for the right mutation.
- A police officer who completes a mutation may have to create the mutation first.
  But also (if the mutation already existed), the police officer may have to search for it.
  (From the case description you can deduct that *either* the former *or* the latter applies. There is no way to express this in the syntax for use case diagrams, so you may ignore that there is a binary choice—this is the kind of logic you want to model in process diagrams, rather than use case diagrams.)

The next series of exercises D-1.8 – D-1.15 all relate to a case study about handling theatre tickets.

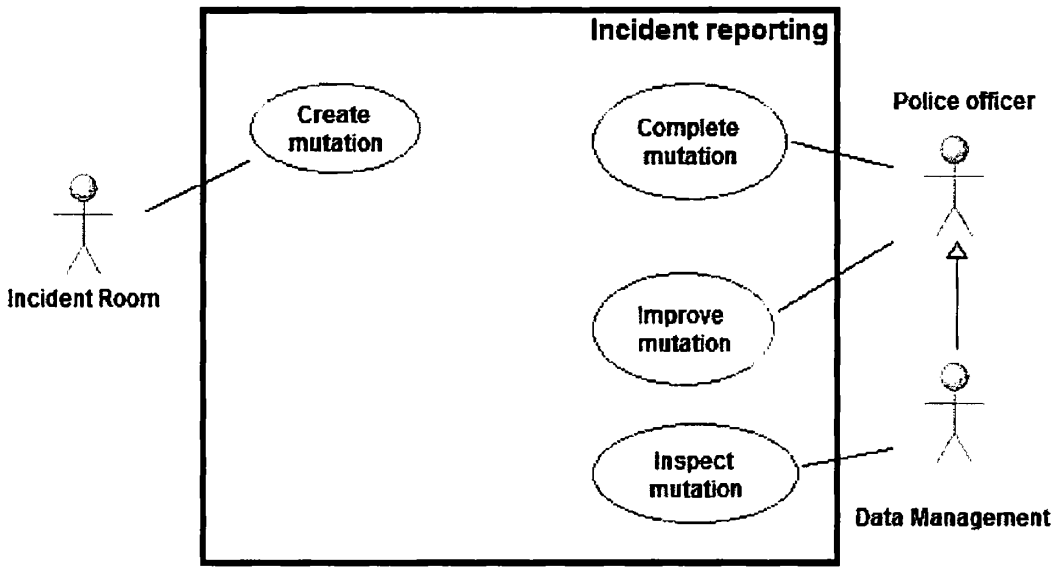


Figure 1.2: Activity Diagram voor incident reporting (Exercise D-1.7)

Theatre tickets: Introduction

State subsidies for cultural activities have been repeatedly decreased over the last few years. As a consequence, the cultural sector in the region Polderland requires major restructuring. This does not only involve theatre companies and orchestras, the theaters themselves are also having a harder time making ends meet. In order to cut costs, the different theatres in the region have decided to merge their organizations. As a result, there is now a single "Polderland Theatre" of which the previously independent theatres have become branches.

One of the steps that should reduce costs is to issue a joint Polderland Culture Programme (as a paper brochure and online) for the theatre season (running from September to June). In addition, the administrations will be merged, with headquarters in Water City, the largest town in the region. A computer system is needed that can be used by the central administration in Water City, as well as the staff of the various theatres throughout the region.

The Polderland Culture Programme is also regarded as an important marketing instrument. It lists all performances of all cultural productions in the region for the whole season. This creates a much larger variety than any of the individual theatres could offer by itself: theatre and dance performances, music in different styles, opera, musical, and cabaret.

Furthermore, the joint programme makes is easier to attract sponsors. They are mentioned as sponsor and get free advertisements.

For this new set-up an appropriate computer system is necessary. *The new system will need all kinds of functionalities, but here we only consider selling tickets and related matters. We disregard customer administration, mailing to sponsors, entering the details of performances into the system, etc., etc.*

For most performances there is standard price of € 23.50 for a ticket, but some productions are more expensive. In some cases only the première performance of a production is priced differently, tickets for the following performances have a regular price. The system with different prices for different seating areas has been abolished some time ago. All tickets for a performance cost the same, whether you sit in front, in the back or on the balcony. Holders of a "Cultural Youth Passport" or a "Culture Card Polderland" receive 25 % reduction on most performances. In some cases there is no reduction.

Tickets are sold for a particular *performance*. A *production* (e.g. the theatre play "Hamlet") will have different performances. In some cases, all performances are at one particular location, in other cases performances are at different locations across Polderland. A performance takes place in a *hall*. One theatre can have different halls (e.g. the Concert House in Water City has a large hall for symphonic music and pop concerts, and a smaller hall for chamber music performances).

Term	Description
Production	A play / ballet / concert / ..., which will be performed one or more times
Hall	Venue where a performance takes place. Polderland Theatre comprises different theatres, each of which could have multiple halls.
Polderland Cultural Programme	(Published as a paper brochure but also available online:) programme of all performances in all theatres for the whole season
...	...

Figure 1.3: Partial glossary for the Polderland Theatre case (Exercise D-1.8)

Ordering tickets in advance

Of course it is possible to buy tickets at the box office of a theatre. However, we'll disregard that for the moment. We start with modelling two ways of buying tickets: by means of a paper order form and through internet.

The (paper version of the) Polderland Culture Programme includes an order form with a complete listing of performances. For each performance the customer can indicate how many tickets they want, and which reduction applies to which number of tickets. Also, a bank account number must be provided. By signing the order form, the customer authorizes Polderland Theatre to conduct a debit payment (i.e., to order the bank to transfer the money from the customer's account to the theatre's account). For regular visitors (who make sure that they order early), the costs can run into hundreds of euros. So Polderland Theatre has arranged that, for orders that arrive the before the 1st of September, the payment will be not be debited immediately, but at the start of the season.

Of course it is not required to send the order form back before the start of the season. Some visitors pick up the Programme at their first theatre visit in the new season, and they could order further tickets by means of the order form.

When an order form arrives at the central administration in Water City, an administrative staff member enters the data into the computer system. If there are still tickets available for the requested performances, these are booked and printed. The paper tickets are mailed (by surface mail) to the customer. On the order form, customers can indicate preferences for seats they like to get, but there is no guarantee that these seats will still be available. If there are still tickets available, but not for the requested seats, different (possibly similar) seats will be booked for this customer. The administrative staff use a graphical interface, showing all seats that are still available for a performance, to make the booking.

Customers also can buy tickets through the Internet. Customers who book through the Internet have the advantage that they can select their own seats from those still available. The internet application uses the same graphical application to show which seats are still available. You can select the desired seats and book them. Upon payment (always immediately for internet bookings), you get the tickets as a PDF file for printing. These tickets contain a bar code that is scanned by the theatre staff when you enter.

D-1.8 Figure 1.3 gives a partial glossary. Extend this with two more terms that you consider useful to include and give appropriate descriptions. You find the document in this week's lab files on Blackboard.

D-1.9 Figure 1.4 gives a requirements list for the Theatre case study (so far). The requirements have already been filled in, your task is to enter appropriate use cases in the second column.
Please remember that requirements and use cases usually do not map one-to-one. It is possible that a requirement is implemented by a small set of use cases, rather than a single use case. It is also possible that a use case serves multiple requirements.
You find the document in this week's lab files.

☞ **D-1.10** Make a use case diagram that includes all the use cases identified in D-1.9.
For your convenience, an actor list has been provided in Figure 1.5

For Exercises D-1.11 to D-1.15, please also consider the following information.

	Requirement	Use case(s)
1	To process an order form	...
2	To debit payments in September	...
3	To find seats by means of graphical interface	...
4	To book tickets through Internet	...

Figure 1.4: Requirements for the Polderland Theatre system (so far) (Exercise D-1.9)

Actor	Description
Customer	A person buying tickets for him/herself and possibly other visitors of a performance.
Administrative staff	Staff of Polderland Theatre who process order forms.

Figure 1.5: Actors for the Polderland Theatre system (so far) (Exercise D-1.10)

Buying tickets at a box office

One of the innovations of the new computer program is that one can buy tickets for a performance *anywhere in the region* at the box office of any branch of Polderland Theatre. This should increase convenience for persons who want to book in advance. Rather than sending a form to the central administration, you can go to the nearest theatre for all tickets you'd like. (But you do not have to buy tickets in advance, you can also go to a performance and by tickets on the spot, if places are still available.)

If you buy tickets at the box office, the cashier can show you on the graphical interface which seats are still available (the same graphical interface that is used for processing order forms as well as internet bookings). If there are seats that you like, the cashier will sell you the tickets. Tickets bought at the box office always have to be paid immediately, even if the booking is made before the start of the season.

Sometimes it happens that customers cannot visit a performance for which they have tickets. It is possible to return the tickets up to 48 hours before the start of the performance. This can be done at the box offices of a theatre (any theatre, not necessarily the one where the performance takes place). The cashier who takes back the tickets releases the seats in the computer system and gives the customer a voucher, with € 3.– per ticket subtracted as administration fee. These vouchers can be used to buy tickets for all theatres in Polderland. A refund is not possible, but the vouchers are valid for five years. For tickets that have not yet been paid (i.e. tickets purchased by means of the order form and returned before the start of the season) the return service is free of charge.

It also happens that customers mislay their tickets or forget to bring them with them to the theatre. Up to 30 minutes before the start of a performance it is possible to ask for duplicate tickets, provided that the customer can prove that they are the customer who ordered the tickets. The cashier looks up which seats were reserved for this customer and prints duplicate tickets. This service costs € 3.50 per ticket.

Excerpts from an interview with Anneke de Wit, employee at Rivertown Theatre

...

What is your function at Rivertown Theatre?

My job title is 'Business Manager'. But that sounds a lot more impressive than it is. We're only a small theatre here, with very few staff, and I work shifts as a cashier like all of us.

Will the merge have a lot of impact on you?

We'll have to see. Some colleagues are worried that people in Rivertown will go to Water City more often, rather than to our local theatre. But I don't think it will make much difference.

Why do you think so?

In the past you had to order tickets from the theatre you wanted to go to. After the merge, you can buy tickets for Water City from our local theatre. Additionally, you get a programme that lists everything in the region, and I cannot deny that they have a lot more to offer. But most of our customers like to come here because they feel it's 'their' theatre—and it's convenient that it is near by.

	Use case	Description
1	Process order form	Administrative staff member makes bookings for all performances indicated on the order form, as far as seats are still available. Indicated preferences will be taken into account, but there is no guarantee that the customer will get the desired seats. May include Debit payment.
2	...	...

Figure 1.6: Template for brief use case descriptions (Exercise D-1.14)

It is expected that most people will order tickets by means of the order form from the season's programme. Does this mean that you will have much less work at your Rivertown box office?

No, that is a misunderstanding. You can send the order form to the central administration in Water City, but you can also drop it off at our local theatre. This is convenient, as we're located right in the town centre. When there are no customers, the cashier processes these order forms. In that sense we are also administrative staff of Polderland Theatre.

Are there any things that we should keep in mind when designing the new booking software?

What is inconvenient in the current system is processing returned tickets. That should be simple. It does not happen a lot, but every time someone returns tickets I have to figure out again how it works. The same when people ask for duplicate tickets. This is even worse, because they're in the queue for the show and then I have to waste time searching for what to do.

It will be much appreciated if you can make this more intuitive and user-friendly.

OK, thanks a lot! We'll look into this.

...

D-1.11 Extend the list of actors in Figure 1.5 so that it covers the whole case desciption. You find the document in this week's lab files.

D-1.12 Extend the list with requirements and use cases (see D-1.9) so that it covers the whole case description.

☞ **D-1.13** Extend the use case diagram with all use cases identified in D-1.12.

D-1.14 Make a complete listing of brief use case descriptions for the use cases in D-1.12 / D-1.13. Figure 1.6 gives a template. You find the document in this week's lab files.

☞ **D-1.15** Make an extended use case description for *Process order form* by completing the description given in Figure 1.7. You find the document in this week's lab files.
(The way the use case is modelled here is that the administrative staff member enters some *user data: name or postal code or whatever. If it uniquely identifies the customer the system shows the customer and the use case proceeds to step 3. Otherwise the staff member can add more data or perhaps select a customer from the remaining matches.*
Similarly for selecting the right performance insteps 3–4.
This is one of many possible ways to implement finding the right customer, the case description does not provide further information. Typically, exended use case description add further design details which are not present in the use case diagram.)

1.3.5 Recommended exercise 1b (Use Cases)

Exercises D-1.16 and D-1.17 are related to the following case study about medication support.

Process order form			
Actor action		**System Response**	
1	Enters some customer data	2	Shows all data for this customer
3	Enters some data of a performance	4	Shows all data for this performance
5	...	6	...
Alternatives:			
1–2	Repeat if customer data entered so far identify more than one customer		
1–2	Create new customer if the customer is not yet known to the system		
3–4	Repeat if preformance data entered so far identify more than one performance		
...	...		

Figure 1.7: Incomplete extended use case description (Exercise D-1.15)

Medication support

Many elderly people suffer from various illnesses and complaints, for which they are given a variety of different medication. When, in addition, their memory isn't what it used to be, it becomes very difficult to remember when to take which pills. Another complication is that elderly people sometimes cannot remember that they already took their pills 10 minutes ago, and could be tempted to take them again. This is problematic, as some pills are harmful when you take too many.

In nursing homes this leads to the following practice. Medication is stored in a locked cupboard in the apartment of the elderly person, but they do not have a key to this cupboard. At regular times a nurse passes by, retrieves the appropriate medication from the cupboard, and sees to it that the medication is taken.

This can be improved with modern technology. Care centre "The Westwolds" in the city of Barchester will run a pilot with so-called medication dispensers, which make the right medication available at the right time. Clients of The Westwolds (the care center prefers to speak of "clients", rather than "patients") include inhabitants of the nursing home with the same name, as well as clients outside the nursing home, in Barchester and a dozen villages in East Barsetshire. These are elderly people living at home, but in need of home care.

Using dispensers will not be a solution for all clients. For some clients it is important that nursing staff makes sure that the pills are actually taken. However, there is a large enough group of clients who can look after themselves, as long as they get the right pills at the right moment.

You will be asked to make some design models for the central information system that will be used in this pilot, based on the following information.

Note: The medication dispensers themselves are *not* part of the information system. They can be regarded as external actors that exchange information with the system.

- A medication dispenser contains a series of packets with pills. It is connected to the central information system of The Westwolds. When it is time for a client to take their pills, the information system sends a message to the dispenser. The dispenser unlocks itself, so that the client can take out the packet with the right pills. When the client takes out the packet, the dispenser sends a message back to the information system. The information system registers the date and time that the medication was taken from the dispenser. If necessary, the client is reminded several times that they should take the medication from the dispenser. If it takes too long, a nurse visits the client to find out what is wrong.
  In more detail:
  - The dispenser is unlocked m minutes in advance of the set medication time. When the medication should be taken, the client receives a message. If they do not take the medication, the message is repeated every k minutes, up to a maximum of $n-1$ times.
  - At the n-th time the message is not sent to the client, but an alarm is sent to the nursing staff. A nurse then visits the client. When the nurse has returned from the client, they report their findings in the system.
- There are various ways in which messages can be sent to a client: a text message to a mobile phone,

Actor	Description
Nurse	A nurse provides care to Westwolds clients; a nurse is authorized for all system functions.
Staff	Staff is authorized for some system functions (for activating and deactivating service plans but not changing service plans).
Dispenser	Device that releases mediation to clients. The device is controlled by the central information system, it sends return messages.
Clock	Some things happen automatically at the appropriate moment. This can be modelled by using the clock as a (pseudo-)actor.

Figure 1.8: Actor list for the medication dispenser service (Exercises D-1.16, D-1.17)

iPad, or television, with or without an additional sound signal. The options could be extended in future.

- Obviously the client's apartment needs to be equipped with a dispenser that is connected to the central computer system.
- For each client who makes use of this service, a so-called service plan has been set by a nurse. A service plan has a maximum of four times a day when medication should be taken. For each client these times can be different (some people rise and go to sleep earlier than others). The service plan also records in which way the client is to be notified. Furthermore, the values of m, k, and n as described above have to be defined. There is choice of several *service patterns* with fixed values for m, k, and n. (A service pattern for heart problems is different from a service pattern for rheumatic conditions. For the latter, timely medication is much less important, so there are more repeats with longer intervals.)

  By the way: not all of the client's medication needs to be handed out through a dispenser. For example, if someone is prescribed pills for high blood pressure and an ointment for a dermatological disease, typically the pills will be distributed by the dispenser, but for the ointment it is not needed (and it would be impractical to distribute it in daily rations).
- When a nurse enters a (new) service plan, the service plan is activated automatically.

  It is possible to deactivate a service plan, e.g. for periods during which the client is not at home. The service plan can be reactivated at any time. Activating and deactivating a service plan can be done by all Westwolds staff; for changing other service plan details only the (properly certified) nurses are authorized.
- Occasionally the medication of a client is changed. If this happens, a nurse adapts the service plan.

  When a service plan is changed, the current service is automatically deactivated (if it was active at that moment). When the changed service plan is stored, the (changed) service will be automatically reactivated.
- An obvious condition is that the medication is physically present in the dispenser. From time to time a nurse visits the client's apartment and refills the dispenser. The dispenser contains a series of packets. Every time the client needs to take medication the next packet is released. (So the nurses should take proper care filling the dispenser when different medication is taken at different times. The dispenser has no knowledge of the contents of the packets.)

  What the dispenser does detect is when the last packet is taken out by the client. If that happens, the dispenser not only sends a message that the medication has been taken, it also sends a signal that the dispenser is empty.

An actor list for the central information system has already been compiled, it is shown in Figure 1.8

D-1.16 Make a glossary for the case description with the five terms that you consider the most important to be included.

D-1.17 Make a use case diagram for the central information system for medication support.

1.4 Programming

1.4.1 Laboratory exercises

You should always read the text between the exercises! This text will give you hints and instructions that will help you to make the exercise. Student assistants will expect you to have read these text, otherwise they may not help you.

Hello World

For more info, read N & H Appendix I

The following exercises are intended to familiarise you with the mechanics of compiling and running a JAVA program. You will first do this using minimal tool support: a plain text editor for typing a program, and command-line calls to the compiler and virtual machine to get it running. Subsequently, you will learn how to do the same using ECLIPSE as an Integrated Development Environment (IDE).

We start with the simplest program imaginable, which does nothing but print a message on your console.

Before you can use the command-line compiler you have to set-up your Path variable. At the following url you can find how to do this: `https://www.java.com/en/download/help/path.xml`.

```
package ss.week1;

/**
 * Hello World class.
 * @author Arend Rensink
 */
public class Hello {
    /**
     * @param args command-line arguments; currently unused
     */
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }

}
```

For this assignment, create a new *directory* on your system, eg `c:\softwaresystems\src` This is the place in which you will create your JAVA source files. Preferably it should be a "clean" directory not used for any other stuff; it is typically called `src` or `java`.

P-1.1 Type in the program above in a plain text editor. Make sure of the following:

- The file should be called `Hello.java`
- The file should be located in a subdirectory `ss\week1` of your source directory.

Now you have a file at the following location: `[source directory]\ss\week1\Hello.java`. Open a command prompt [2] at your *source directory* and *compile* your program using[3]

```
javac ss\week1\Hello.java
```

You compiled a java file, and in the directory `ss\week1` a new file appeared. What do you think you did?

P-1.2 In the previous exercise you compiled `Hello.java`, in this exercise you will run the file. Run in a command prompt at your `source directory` the following command:

[2] Goto `http://windows.microsoft.com/nl-nl/windows/command-prompt-faq` if you do not know what a command prompt is.

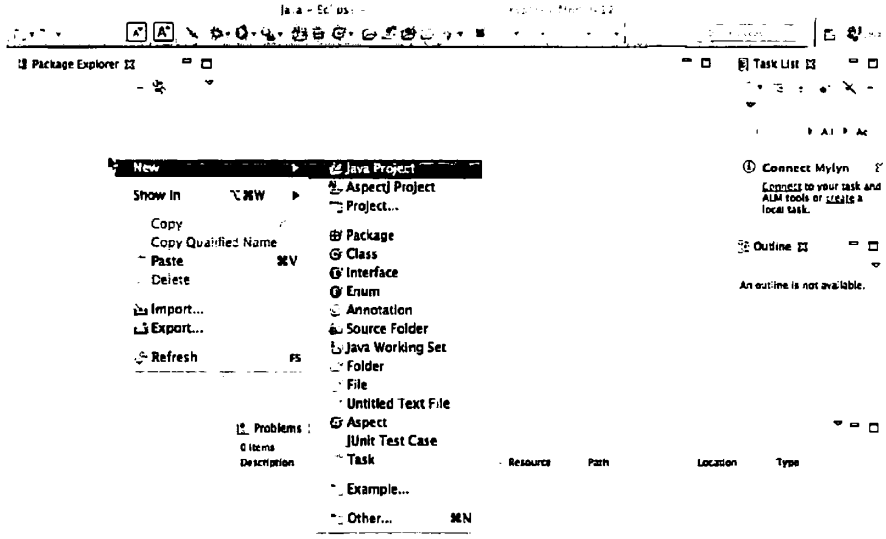
[3] Note that on a Linux or Apple system, all backslashes separating directory names and file (or other directory) names, as in `ss\week1\Hello.java`, should become slashes, as in `ss/week1/Hello.java`.

```
java ss.week1.Hello
```

What happend? And what do you have to do to change the text on the screen to *Hello softwaresystems student*? Try this out.

Instead of calling javac and java in a command prompt we will use the Eclipse IDE. Eclipse makes compiling and running code a lot easier. More information about Eclipse is available online: `http://help.eclipse.org`. In the online Eclipse help, the most relevant part is on the "Java development user guide", which you can choose from the list of contents at the left-hand side.

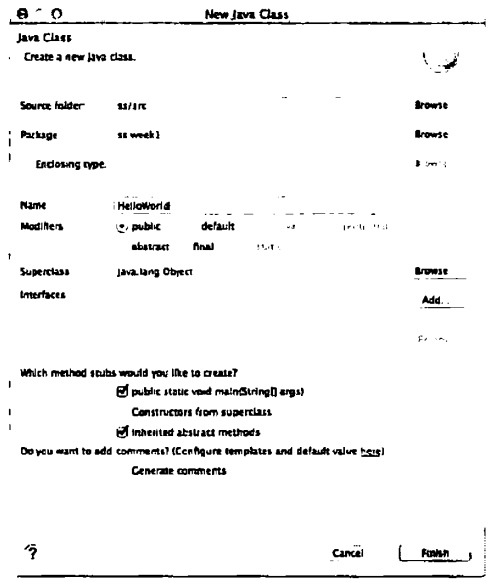
☞ **P-1.3** Now, start Eclipse. At the left of the window, you should see the PACKAGE EXPLORER. Move your mouse in there, and right-click. Choose NEW in the context menu, and next choose JAVA PROJECT.



Enter a project name: `softwaresystems`, and click FINISH. Eclipse will automatically create a folder called `src` in the new project.

Select your new project in the PACKAGE EXPLORER, right click to get the menu where you can select NEW again. Now you have more options. Select to create a PACKAGE, and call it `ss.week1`.

Next inspect the contents of your new project, and select the package `ss.week1`, and add a new CLASS to it. Call this class `HelloWorld`. Select that you would like to create a method stub for `public static void main(String[] args)`.



Copy the contents of the `main` method into your file. Save your file (via the FILE menu, or using CTRL-S). Now we will run the class: select RUN in the RUN menu. The output of the program execution will be printed in the Eclipse CONSOLE view.

Now try to introduce some small errors in the file: for example: rename your package declaration to `week2`, or remove the `;` after a statement. What happens? What happens when you hover on the little red cross?

Three-Way Lamp

First, we will practice how the real world can be modelled in a program.

For more info, read N & H Section 1.1

P-1.4 Make exercises 1.5 – 1.9 from Niño & Hosch *(note: not the self-study exercises)*.

Next we are going to develop a three-way lamp application. A three-way lamp is a light switch with four different options: off, on with low light, on with medium light, and on with full light. First you should specify the intended behaviour of the application.

☞ **P-1.5** Make exercises 1.2 and 2.4 of Niño & Hosch. Before continuing with the following exercises, save a copy of your solution to show it to the student assistant.

For more info, read N & H Section 1.2 & 2.2.1

The next step is to provide a working implementation for the lamp.

P-1.6 Make exercise 2.14 of Niño & Hosch. Use for the `switchSetting` method the modulo (%) operator and test it by checking if the code compiles.

For more info, read N & H Section 2.3 & 2.4

Finally, we also want to test our implementation.

☞ **P-1.7** Make exercise 3.8 of Niño & Hosch.

For more info, read N & H Section 3.6

Hotel

On Blackboard, several auxiliary classes and tests are available, which should be used during the exercises. Go to Blackboard and download the ZIP file with the provided material for Week 1 to any folder on your local disk and remember this folder. In Eclipse, select from the FILE menu IMPORT. In the dialog which now opens, choose from the category GENERAL the option ARCHIVE FILE and then press NEXT. In the following window, click the BROWSE button next to the input field FROM ARCHIVE FILE and select the ZIP file you just downloaded. Then click BROWSE next to the input field INTO FOLDER. Here you select the folder `src` in your project. Finally, click the FINISH button. This will add all the provided material to your

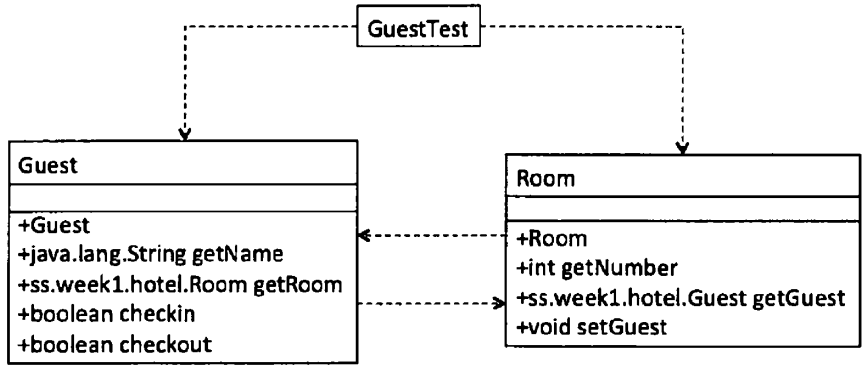


Figure 1.9: Class Diagram for Hotel Application

project. You can now find the source code for the class `Room` as well as the documentation for class `Guest` in the package `ss.week1.hotel`. Additionally, in the package `ss.week1.test` you can find several test classes for this week's exercises.

Note: Eclipse will show compilation errors in the classes which you just imported. The reason is that the provided classes already make reference to the classes that you will develop during this assignment. After completing assignment for class `DollarsAndCentsCounter`, the compilation errors in `DollarsAndCentsCounterTest` must be gone. The remaining errors can be ignored. When you want to run `DollarsAndCentsCounterTest` after this assignment, Eclipse will tell you that the project contains compilation errors, you can ignore this and press PROCEED in the dialog. Similarly, when you have developed `Guest`, the compilation errors in `Room` and `GuestTest` should disappear, but there will still be compilation errors in `PasswordTest`, which can be ignored.

In Exercise P-1.7 you made a very simple test and a more complex test based on the `assertEquals` method. If you makes tests for a larger system you want to automate this as much as possible. There are several tests suits to make testing easier, one of them is JUNIT. More information about testing and JUNIT can be found in Niño & Hosch [Chapter 6]. The test classes that you just downloaded are developed using JUNIT. Next week, you will learn to develop JUNIT test classes yourself.

As a warming-up, we first do some basic exercises from Niño & Hosch.

For more info, read N & H Section 4.1-4.2, 2.4

☞

P-1.8 Make exercise 4.2 of Niño & Hosch about Booleans.

P-1.9 Make exercises 4.7 and 4.13 of Niño & Hosch about conditionals. For exercise 4.13, a test class `ss.week1.test.DollarsAndCentsCounterTest` is provided in the ZIP file that you downloaded from Blackboard. Use this test class to make sure your solution functions correctly for all cases.

During this course, there will be several exercises where you develop parts of a hotel application. This exercise is the first: you will be asked to develop a class `Guest`. We will first develop a "stub" implementation for class `Guest`. This stub implementation contains all methods of the class, but with minimal bodies. In the following exercises, we will develop the class further.

P-1.10 Create a class `Guest` in the package `ss.week1.hotel` containing all methods required according to Figure 1.9. Use the documentation `Guest.html` and the class `Room.java` to find out which parameters are necessary. Make sure that the compiler finds no problems in `Guest`. This means that any method that has a non-`void` return type should contain a `return` instruction with an appropriate value (see also the blue black on page 69 - 71 of Niño & Hosch). Typically,

- if the return type is `int`, use `return 0;`
- if the return type is `boolean`, use `return false;` and
- if the return type is the name of a class, use `return null.`

Apart from the `return` statements, the method bodies should be empty.

Also the classes `GuestTest` and `Room` should now contain no compilation errors anymore.

☞ **P-1.11** Run the program `ss.week1.test.GuestTest` to test your implementation. Copy the output to a text file, and add a manual explanation of the differences in the output.

Before really starting on the implementation, an intermediate step is to add documentation to the class.

For more info, read N & H Appendix I

P-1.12 Add `javadoc` comments to your class `Guest` that describes the intended behaviour of the class. You can use `Room.java` as a source of inspiration for the formatting of your documentation. Use tags such as `@author`, `@version`, `@return`, and `@param`, and try to make useful and original documentation.

To generate the HTML files from JavaDoc, first select your project in the PACKAGE EXPLORER. Next, select PROJECT from the menu and then GENERATE JAVADOC. In the dialog which opens now, you can keep all default entries and simply click FINISH. Now you will be asked if the folder into which the files are generated should be used as the JavaDoc location for your project. You can choose YES here, so that Eclipse will use this folder to search for documentation of your classes. Eclipse will now generate several HTML files into the `doc` folder in your project.

☞ **P-1.13** Generate JavaDoc for your project. You can open the file `doc/index.html` to start browsing the generated documentation. What information is included in the JavaDoc? Explain why JavaDoc can be useful.

P-1.14 Make exercise 2.16 from Niño & Hosch.

P-1.15 Complete the implementation of class `Guest` by developing non-trivial method bodies. Make sure your method implementations respect your specifications.

Method `checkin` should do the following: if the room that it receives as argument is not occupied yet (checked by calling `getGuest` on this room object), then the current guest is assigned to this room (by using `setGuest`), otherwise the method returns `false`. Notice that the current guest is the receiver of the current method, *i.e.*, the `this` object.

Test your implementation with the `ss.week1.test.GuestTest`, make sure your implementation passes the test.

Finally, the information provided by the test about the tested guest and room can be a bit difficult to understand. Therefore, we will finish by providing a way to give a more informative textual description of the objects. This could be for example the name of the guest, or a room number. As described in Niño & Hosch [§3.5.2], we can use the Java method `toString` for this purpose.

☞ **P-1.16** Add a method `toString` to the classes `Guest` and `Room`. For each guest, it should provide a description `Guest ...`, and for each room, a description `Room ...` (where `...` denotes the name and the room number, respectively). Check how the result of test is improved.

For more info, read N & H Section 3.5.2

Passwords

This exercise asks you to implement a class `Password`. It should provide operations to compare the password with an arbitrary string, to check whether this is the correct password; and to change the password. Next week, the password class will be used to develop a safe application, to be used in a hotel room.

In the ZIP file you downloaded from Blackboard, you can find the (`.html`) documentation of the `Password` class (in package `ss.week1`).

For more info, read N & H Section 2.5.1

P-1.17 Why is there a constant `INITIAL` to initialise the password upon construction, instead of initialising it simply to the empty string `""`?

P-1.18 Why is the constant `INITIAL` declared as `public` instead of `private`?

As the specification for `Password` is given, it should not be too difficult to make a stub for `Password.java` with empty constructor and method bodies, where necessary with a trivial `return` statement, as in Exercise P-1.10 for `Guest.java`.

P-1.19 Develop a stub implementation for `Password` respecting its specification; make sure the stub implementation compiles.

Now we are ready to fully implement `Password`.

☞ **P-1.20** Complete your method bodies for class `Password`. Method `setWord` should do the following:

- Check if the old password is correct;
- Check if the new password is acceptable;
- If so, update the password.

Use the methods of `Password` that provide this functionality. Use class `ss.week1.test.PasswordTest` that is contained in the ZIP file that you downloaded from Blackboard to test your implementation.

1.4.2 Recommended exercises

P-1.21 Make the following exercises from Niño & Hosch.

- 1.3
- 2.2, 2.13
- 2.6
- 3.2, 3.4, 3.6

Week 2

2.1 Overview

2.1.1 Contents of This Week

Academic Skills This week there is one lecture dedicated to academic skills: a 2-hour meeting on Monday. During the lecture an overview of the basic principles of time management will be discussed in more detail: prioritizing, planning, and monitoring execution. We will focus on identifying the most important tasks for your individual study plan, both on short term as with regard to a semester perspective. The goal is to identify those factors that will enhance your ability to study effectively and efficiently.

The lecture will also focus on self-awareness. One of the major pitfalls in executing any plan is procrastination behaviour. During the lecture you will do a short assessment that will help you to look into some major factors causing procrastination and what to do about them.

The self-assessment on procrastination need to be completed by the end of the week and handed in on Blackboard. The timekeeping will be signed off in next week's peer feedback session.

Design The design activities in this week cover the following topics

- *Class Diagrams* describe the information structure of a design. See 2.3.1 for the exercises for Lab Session 2a.
- *Sequence diagrams* describe the interaction between system components. See 2.3.4 for the exercises for Lab Session 2b.
- Project work, continued from last week, see 2.3.3.

Programming This week the following topics will be discussed:

- Specifications and programming by contract (using the Java Modeling Language). This will be done in a double lecture: the first 45 minutes will discuss the theory and concepts, the second 45 minutes a few lab exercises on specifications will be discussed plenary.
- Testing: unit testing, test plan, and test framework

Note: during Week 4, an optional lecture will be given on how the Java Modeling Language can be used not only for specification, but also to validate correctness of an application.

2.1.2 Mandatory Presence

During the following activities, your presence is mandatory.

- (M) Self-study supervised (Tue 6 – 7)
- (M) Tutorial (Thu 3 – 4)

2.1.3 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 1 hour to make the academic skills exercises;
- 4 hours self-study for mathematics;
- 1 hour self-study for the design thread;
- 2 hours design project work beyond the session with assistance; and
- 2 hours self-study for the programming thread.

2.1.4 Materials for this Week

Academic Skills
Van Tulder, Sections B8, B9 and B10.

Design
Slides from lectures 2a and 2b.

Programming

Lecture Niño & Hosch, Chapters 5–8
Laboratory The following predefined files are provided on Blackboard:
- `ss/week2/test/HotelTest.java`
- `ss/week2/test/RoomTest.java`
- `ss/week2/Dlab-2b-modelio.zip`
- `ss/week2/Rectangle.java`

JML Appendix A on the Java Modeling Language
Tools Use of the OPENJML and EMMA plugins

2.2 Academic Skills

2.2.1 Assignments

A-2.1 Evaluation of procrastination behaviour Pick one (type of) activity where you are currently procrastinating more than you would wish. For this (type of) activity, fill out the questionnaire on procrastination behaviour you can find on Blackboard. As you add up the total scores, you can make your diagnosis: is your main problem in the area of value, expectancy, delay or impulsiveness. Try to find a quick win for improvement, by naming some concrete actions you can take.

You have to submit your answer **at the latest Sunday of Week 2 at 23:59 CET** via Blackboard.

A-2.2 Monitoring execution of personal planning (timekeeping) During this week you are expected to write down how you are actually spending your time. Use the self-monitoring schedule you have used to make your planning for this week and complete the columns U and R for actual execution (U) and results (R) during this week.

Please note: you will only get full benefit of this exercise if you are completely honest on the match between planning and execution. Rather note that you have not put in the hours you have planned or you have not been as productive you would have wished, instead of giving a favourable impression. Reflection on this will be discussed during the peer feedback session in Week 3. The better you know where you got off track, the more useful this will be for you. You don't have to submit this exercise,

however you should bring your time monitoring sheet with you to the peer feedback session to be signed off there.

2.3 Design

2.3.1 Laboratory session 2a (Class Diagrams)

The laboratory session is done in groups of two students.

All lab exercises are concerned with the following case description.

CD Rental
The CD Rental was founded in the 1980ies by students of the UT (then: THT) and at the time located on campus. By the end of the century the CD Rental moved to the second floor of the Enschede public library in the town centre and merged the library's CD collection with its own. It is still located there. Meanwhile, the collection has grown to over 100,000 CDs and 15,000 records on other media (vinyl, DVD, and, more-recently, Blu-ray).

For many young persons CDs are a medium from the past. Music can easily be obtained online, legally or illegally. Nevertheless the CD Rental satisfies a need. People may want to read the liner notes, enjoy browsing the physical collection, of have other reasons for getting their music offline. Tens of thousands of CDs are rented per year.

For every rented CD[1] a small amount of money is paid to the body that distributes revenues to the parties with legal rights. What a lot of people don't know is that Dutch legislation on authorship explicitly allows you to make a digital copy of a rented CD for personal use. Ripping rented CDs is entirely legal as long as you don't distribute it or sell it.

Anyone can become a member of the CD Rental for a fixed yearly rate, and then rent an unlimited amount of CDs for a small fee per item.

Some time ago, the CD Rental started a collaboration with the *Overijsselse Biblitoheekdienst* (OBD) (collective of libraries in the province of Overijssel). From then on, any person who is a member of any library of the OBD can rent CDs at the CD Rental.

As a result of this collaboration, the term "member" needed to be redefined. The computer system differentiates between external members (i.e. members of OBD libraries) and normal members. Normal members have some privileges:

- *New CDs.* The first three months that a CD is in possession of the CD Rental, it is rented only to normal members, not to external members.
- *Reservations.* Normal members can reserve CDs. These are set apart (for CDs that are currently rented out: after their return) on a shelf with reserved items so that other customers cannot rent them. The member receives an e-mail message when the reserved CD can be picked up. This service is free of charge.

For the information system the following points could be relevant.

- For each CD (or other medium) the following information is stored in the system:
  - Title.
  - Artist's name (for most kinds of music this is the artist/group, for classical music it is the composer).
  - Year in which the CD was released.
  - Registration date (when it was obtained by the CD Rental—not necessarily in the year in which the CD was released).
  - Medium (vinyl, CD, SACD, DVD, Blu-ray)
  - Item code: unique identification, e.g. "CP44661". Every item carries a bar code label with the item code.

[1] Throughout this text we speak of "CDs", instead of the more precise but much more cumbersome "CDs or other media".

  - "Three-letter code": usually the first three letters of the name of the artist. (For example "DEL" for Ilse DeLange). CDs in the collection are sorted by three-letter code. Three-letter codes—despite their name—are strings of 2–8 characters. For example the CD "Hitzone 56" is a compilation of tracks from different artists (so the artist's name is "various") and the three letter-code is "DIV/HIT". CDs of the group U2 have the three-letter code "U2".
  - Music category. The CD Rental categorization comprises five major styles: Pop; Classical music; Jazz; Blues; World music. These are subdivided into some hundred different categories. Every category is part of a single music style, e.g.: "Baroque" is classical music, "Techno" is pop music.
  - Information: All other information about this CD in the database. For classical music this may include orchestra, conductor, soloists, etc.

- Newly obtained CDs are registered in the system by a back office employee. The first three months these items are rented only to normal members. After three months they cease to be new and all members can rent them.

- For some CDs that are in high demand, there are multiple copies available. For example, there are two copies of "Incredible" by Ilse DeLange, with item codes CP44661 and CP44662.

- The CD Rental catalogue is available on the website. If a CD has been rented out, it is possible to see the date when it should be back, and whether there are any reservations for this CD. The joint *Library Catalogue Overijssel* of the OBD libraries is linked to the CD Rental system, so that library members can browse the CD Rental catalogue there.

- Renting a CD works as follows. The customer can visit the CD Rental, browse through the collection, possibly listen to CDs on a CD player, and rent the CDs at the counter. An employee enters the information into the system by scanning the customer's membership pass (for external members: the library pass) and the bar code on each CD. For each CD that is rented out, the employee checks it for damage (every CD has a 'scratch card' on which scratches and other irregularities are indicated) and demagnetizes it (turns off the magnetic security label so that the alarm doesn't ring when the customer leaves the premises).
  CDs are rented for three weeks. For late returns, an additional daily fee is charged.
  (To keep things simple we disregard anything related to payment.)

- A member of an OBD library can also rent CDs through *BookFinder*, the OBD system for interlibrary loans. Library members who browse the Library Catalogue Overijssel are automatically transferred to *BookFinder*, but *BookFinder* is not linked to the information system of the CD Rental. Once a day an employee prints the requests that have arrived through *BookFinder* and sends each CD with the print to the requesting library. The library member then can collect the CD at their local library.
  For the employee at the CD rental there is little difference between renting a CD to a customer at the counter and renting a CD through *BookFinder*. In both cases the rental data about the customer and the CD have to be entered into the system. The main difference is that a CD rented through *BookFinder* is rented to *the library* (not the library member). To that end, a special library number (uniquely identifying the library) is manually entered in the field where otherwise the customer's pass number would be scanned. The CD is demagnetized and scanned for scratches as usual, and then prepared for transport.

- Reservations can only be made through the Internet. There can be multiple reservations for one CD (by different members; it is not possible to make a second reservation for a CD which they currently have reserved already). Reservations are processed by the system in the order in which they arrive.

  - If a member makes a reservation for a CD which is currently present in the CD Rental, a message appears in the top of the screens at counter (*"1 reservation should be set apart (CP44661)"*). At a quiet moment, when there is no customer at the counter, an employee takes the CD from the collection and scans it in the system. The CD gets is given special status "set apart" and the member who reserved it receives an automatically generated e-mail. The employee puts the CD on a special shelf behind the counter.

  - If someone makes a reservation for a CD that is not currently available at the CD Rental, then the reservation remains pending until the CD (in case of multiple copies: one of the CDs) is returned. The returned CD is then automatically allocated to the next member who reserved it. On the employee's screen a message appears that the returned CD—after inspection—should be scanned again to be set apart.

- From the moment that a CD has been reserved, it can only be rented to the member to whom it has been assigned. If an employee by mistake tries to rent it to another customer, the system refuses and gives a message that this CD cannot be rented to this customer.
  A CD that has been set apart should be collected within 72 hours. After that period the reservation expires. The member is notified by another automatically generated e-mail message. If there were more reservations for this CD, it passes to the next member who reserved it. If there are no more reservations, the employee at the counter is informed by a message in the top of the screen that certain CDs that were set apart should be returned to the collection.

- When a rented CD is returned, the employee at the counter scans it and magnetizes it. Again, the CD is checked for damage. (When it is busy, this can be delayed until a quieter moment.) If the returned CD turns out to be reserved, then, as describe above, a message appears in the screen asking the employee to inspect it, scan it again, and set it apart.
  When the OBD transportation service returns CDs that were rented through *BookFinder*, the employee registers the return separately in the two different systems: the *BookFinder* system and the CD Rental information system are not linked. For the CD Rental information system there is no difference with CDs that are returned by customers in person. Also it is magnetized and inspected as usual.

- Data about rentals remain stored in the system after the CDs have been returned. These include the date and time[2] it was rented, the date and time it was returned, and the customer to whom the CD was rented. The back office can use these data to generate reports and statistics. Data about reservations are deleted from the database when the reserved CD is rented out or when the reservation expires.

- Because of the new situation, employees at the counter are instructed to pay attention the different types of membership

  - the CD Rental has its own members, as before, who are now called *normal members*. The following information about a normal member is stored in the system: name; address; e-mail address; membership passes (multiple passes are possible, see below); start date of membership; date when membership will expire.
    When the (normal) membership has expired, the member can renew this the next time they rent CDs and pay membership for another twelve months.
  - A library member of an OBD library who wants to rent CDs at the counter should first have his library pass registered by an employee. In the CD Rental system the person will be registered as an *external member*. This registration remains valid as long as the person remains library member.
    From external members, only pass number and e-mail address are registered. Other data are not needed. The pass number uniquely identifies the library of whom the pass owner is a member (for example: all pass numbers starting "0330" are from members of the public library in Enschede), so the CD Rental can get further customer data from the library in the rare cases where that is needed. The e-mail address is occasionally used by back the office administrator to notify a customer that a CD is long overdue.
  - An external member can become a normal member by paying a (reduced) additional membership fee to the CD Rental. They are stored as a normal member in the system receive a CD Rental membership pass. The library pass number and CD Rental pass number are linked to the same person.

- A normal member can have multiple passes. They have at least one CD Rental pass, possibly more: sometimes passes get lost, in which case the member gets a new pass and the old one is blocked.

[2] Typically the *Time* attribute includes the date, so there is no need for a separate *Date* attribute.

However, the old pass number remains stored in the administration. In addition, the member can have a library pass number (or several library pass numbers for similar reasons).
For each pass the following data are recorded in the system: pass number; blocked or not blocked; kind of pass (library pass or CD Rental pass).
Rentals are associated with a customer, *not* with a specific pass. If CDs rented on one pass are overdue and the customer tries to rent new CDs on another pass, it is immediately detected that this is the same person.

- For rentals through *BookFinder*, it is the library to which the CD is sent that is registered as customer in the CD Rental information system. To that end, for every OBD library the following information is stored: name; address; telephone number; library number.

D-2.1 Consider, for the moment, only three classes: *Reservation*, *CD*, and *Member*. How can these classes be associated with each other in a 'snapshot' view? (i.e. the system stores current reservations, but does not retain reservation information from the past). Which options exist? which one would you consider preferable and why? Make a sketch (on paper).

D-2.2 Consider, for the moment, only three classes: *Rental*, *CD*, and *Customer*. How can these classes be associated with each other in a historical view? (i.e. data about past rentals are *not* deleted when a CD is returned). Make a sketch (on paper).

☞ **D-2.3** Study the various forms of membership used for renting CDs (there are three). There are various ways in which this can be represented in a generalization hierarchy. A flat hierarchy has one superclass with any number of subclasses. In a deeper hierarchy a subclass can be the superclass of a sub-subclass, etc.
Draw two different stuctures (on paper).

If you take into account that anyone who rents a CD at the counter needs a pass, but libraries (to whom CDs are sent) don't have pass, which generalization hierarchy is preferable? Add the membership pass to the diagram.

D-2.4 Draw a class diagram that integrates the partial solutions of D-2.1, D-2.2, and D-2.3. (Combining these, however, yields a class diagram that is not yet complete and possibly not even correct. This will be mended in the next exercise.)

☞ **D-2.5** Study the case description again and check it for details that have not been modelled yet and situations that cannot be adequately represented in the class diagram so far.
One of the things you have to consider is the scenario that the CD Rental has three copies of a brand new CD, all of which are rented out, but there are 5 reservations for it. How is this going to be handled by the system? How should this be represented in the class diagram?

2.3.2 Recommended exercise 2a (Class Diagrams)

D-2.6 Make a class diagram for the following case description.

Note: the generalizations in this case are quite tricky (more difficult than what will be asked in the test)

Ship Rental

The newly founded company Ship Rentals Ltd. needs a system for its administration. The company offers sailing ships and motor ships for rent. Some of the ships are owed by Ship Rentals itself, others are chartered by Ship Rentals from the ship owners.

For each ship the following information should be stored: its name, ship type, year of construction, length, draught (maximum vertical space below the water surface), number of sleeping places on board, whether it is chartered or owned, and the current location of the ship. Ships of the same type have the same length, draught, and number of sleeping places. Furthermore, every ship is identified by a unique number.

For each type of sailing ship, the size (surface area) of the sails and the height of the mast (if it has more than one: the tallest mast) is stored. For each type of motor ship the fuel type and motor capacity are

stored. For each chartered ship, the owner's name, address, postal code, and municipality are known. Also, for each chartered ship there is a charter contract stating the contract number, start date, end, date, and the charter price (for contracts spanning multiple years this is the charter price per year). For each owned ship, the date is known on which it was acquired by Ship Rentals.

Renting a ship starts with filling out a form with information about the customer and requirements for the ship to be rented. Based on this information, Ship Rentals will make a offer and send this with a rental contract to the customer.

A rental contract is identified by a unique number. The contract contains the following information about the customer: name, address, postal code, and municipality of the customer, identification (passport or driving licence number). The contact also states the date that the is contract sent, the name and type of ship, the begin date and end date of the contract, the port of departure, the port of arrival, and the price for the rental. It is possible that the ship needs to be transported from/to another port before/after the rental. If this is the case, the contract will mention transport costs.

The offer with the rental contract is sent to the customer. It they accept, they should send a signed copy back to Ship Rentals within two weeks, and make a down payment (a percentage of the rent payed in advance). When the down payment has been received by Ship Rentals, the contract is confirmed. If the down payment is not received within two weeks, the contract is cancelled. It is possible that the same customer has multiple contracts with Ship Rentals.

When the customer returns the ship in the port of arrival, the date of return needs to stored. In most cases this is the end date of the contract, but due to bad wheather or other reasons it could happen that the ship is returned too late. After the ship has been returned, the balance (rental price, possibly including surcharge in case of late return, plus transport costs if applicable, minus down payment) can be calculated and a bill is sent to the customer. The customer should pay this in a single installment.

If no payment is received within four weeks, Ship Rentals will send a reminder. A second reminder is sent after another four weeks. If no payment follows during the next four weeks, the file is handed over to a collection agency, which will try to recover the money with various legal means.

2.3.3 Project

D-P.2 For the project, you can work on the following tasks:

- Draw the activity diagrams for the business processes which you identified last week.
- Conduct an interview with a representative of the Barchester City Council, in order to elicit requirements for management functions of the system. On Blackboard you can find a list indicating which group should contact which person. The interview can be conducted any time this week, not necessarily Wednesday afternoon.
  Please note: Further useful tips for interviewing, beyond Lecture 1b, can be found in Skill sheets D2, D3, D4, D6.
- You can start working on the requirements list, the glossary, the actor list, the use case diagram and the use case descriptions even if you have not yet conducted the interview.

2.3.4 Laboratory session 2b (Sequence Diagrams)

The laboratory session is done in groups of two students.

In this laboratory session we make a further analysis of some processes in the CD rental that was the subject of laboratory exercise 2a.

Renting out CDs

The customer, having selected some CDs, hands these to the employee at the counter. The employee scans the customer's pass and, after that, the bar codes of the CDs. When all CDs have been scanned, the employee inspects the CDs for scratches, demagnetizes the CDs, and hands them to the customer. Subsequently the employee handles payment—only payment in cash is accepted—and enters into the system that payment has been processed. The system asks the employee to confirm the end of the transaction.

Figure 2.1 shows the sequence diagram of renting CDs to a customer:

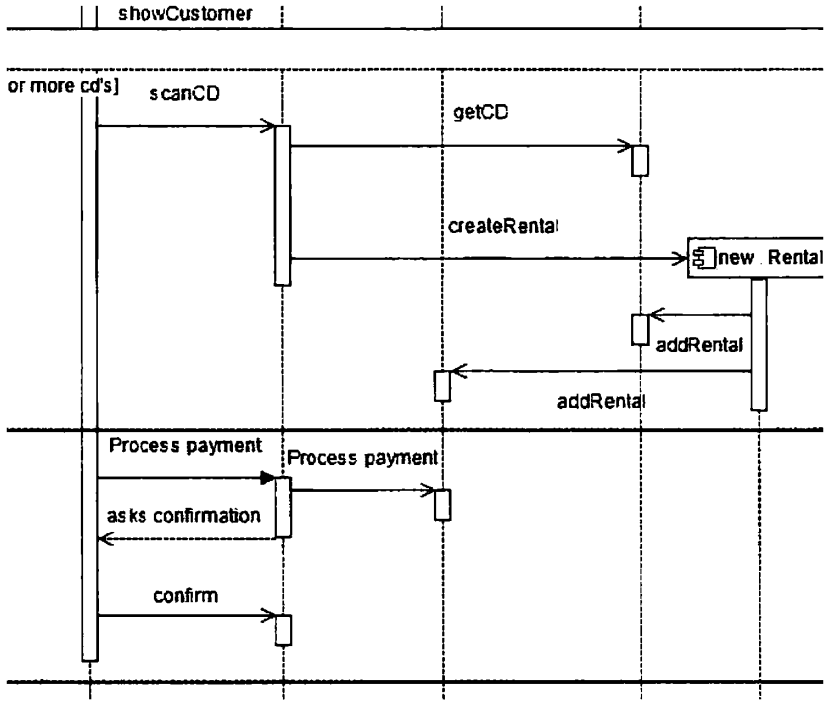


Figure 2.1: Sequence diagram for renting out CDs (Exercise D-2.7)

- The employee is the only person interacting with the system (similar to the cashier handling theatre tickets in Lab session 1b).
- The diagram contains life lines for the actor, for a control object *:RentOut* and for objects from the class diagram (Lab session 2a) that are relevant to renting out a CD.
- Return messages are not shown in many cases. However, return messages from *:RentOut* to *:Employee* are shown where it is useful to know what the employee actually sees on the screen.
- Various activities *outside* the system are not respresented in the sequence diagram: The customer hands the CDs to the employee, the employee inspects the CDs for scratches and demagnitizes them, the customer gives money and may get change, the employee hands the CDs back to the customer. In a sequence diagram we model what happens at the user interface and inside the system, we're not concerned with activites outside the system.

Please note the differences with activity diagrams in the first lecture: we used activity diagrams to model processes in the real world[3]. Moreover, in the activity diagram for a library, we only modelled borrowing a single book. This was to keep the model simple, taking for granted that borrowing multiple books would be similar. In sequence diagrams we model some more procedural detail, showing *how* a use case is executed, at same level of abstraction as the exended use case descriptions.
In this case: For renting out a set of CDs we want to model explicitly that the membership pass is scanned once, before all of the CDs are scanned.

D-2.7 We can simplify Figure 2.1 by moving an interaction fragment from this diagram to a subdiagram. *(Having a smaller diagram will be useful in D-2.8, where we will extend the sequence diagram.)*

Make a subdiagram *createRental* that covers the interactions for creating a new rental (i.e., the create message for a new rental object and the messages to add this rental to the given CD and customer). In Figure 2.1, replace the elements that are no longer needed by a reference to the subdiagram.
You find Figure 2.1 in lab files for this week.

Renting out CDs (continued)
The renting-out process, as modelled in Figure 2.1, does not take reservations into account, this needs to be adapted.

When the renting out starts, the employee has a small pile of CDs to be rented to this customer. Some of these (possibly 0) were taken from the collection by the customer and handed to the employee, some of these (possibly 0) were taken from the shelf with reserved CDs by the employee. Scanning these CDs for rental makes no difference for the employee, but the system will treat them slightly differently. If there was a reservation, then, in addition to creating a rental object, the reservation is to be deleted.

For a robust process, we make no assumption about the order in which the CDs are scanned. First the non-reserved and then the reserved CDs; first the reserved and then the non-reserved CDs; a mixture of reserved and non-reserved—any order should be acceptable.

To make things more complicated, it can happen that a rental is refused by the system when the employee scans the CD. There are two reasons why a rental can be refused:

- The customer is an external member, who has picked up a new CD and handed it to the employee. External members are not allowed to rent new CDs, so the system refuses to create a rental.
- The employee tries to rent a reserved CD to the wrong customer. This happens occasionally when there are different copies of one CD, and two copies are on the shelf with reserved items. Each of these copies is allocated to a specific customer. If, by mistake, you try to rent one copy to the customer for whom *the other* copy has been reserved, the system doesn't accept it.

☞ **D-2.8** Extend the modified diagram from Figure 2.1 (Exercise D-2.7) to include the following:

- the handling of reserved CDs,
- attempting to rent out CDs which will be refused by the system.

[3] This is not a restriction by UML. Activity diagrams can also be used to model how different components within a computer system interact. Similar for sequence diagrams. It is in the context of *this course* that we use activity diagrams for modelling activities in the real world and sequence diagrams for specifying use cases in more detail. All UML models can be used at any level of abstraction.

Registering returned CDs

At the CD Rental, employees have complained that the procedure for registering returned CDs is not logical and should be redesigned.

Most customers return several CDs, not a single CD. The employee at the counter scans all the CDs returned by one customer before starting another task (helping another customer; checking returned CDs for scratches; whatever needs to be done)—at least, this is what employees would like to do.

If a customer who returns CDs has reservations that were set apart on the shelf, the system will interrupt the employee when the first returned CD has been scanned. The system detects that reservations are waiting for this customer, and will ask the employee to take the reserved CDs from the shelf. The employee is forced to do this before they can continue scanning the remaining returned CDs.

In more detail, the following happens:

- Normally, when an employee scans a returned CD, a screen will be shown as in Figure 2.2. The screen shows a list of returned CDs, the last one that was scanned at the top. The information about previous returns scrolls down and eventually disappears from the screen. *(At the top there is a message about CDs that still have to be set apart. This is not related to the current customer; when reservations have to be set apart it is shown in the top of* every *screen)*
- However, if a CD has been returned that was borrowed by a customer for whom reservations are waiting, the employee wil get a different screen, as shown in Figure 2.3. The screen shows a message about the (first) reservation that has been set apart for this customer.
- The employee searches the shelf with reserved CDs that have been set apart for this item and puts in on the counter.
- The employee deletes the message.
- This is repeated until all messages about reservations are dealt with.
- Then, at last, the screen in Figure 2.2 returns, showing that the (first) CD of this customer has been successfully returned.
- The employee scans the remaining CDs that were returned by this customer.

A model of this process is shown in Figure 2.4. Note that it does *not* contain a loop for the case that a customer returns more than one CD. The reason for this is that the system is not concerned whether CDs are returned in groups or individually. Registering a returned CD is complete transaction. Registering two returned CDs are two separate transactions. What happens in such a transaction is that the rental object is looked up and a return time for the rental is stored (and information about reservations for this customer is retrieved). Whether the next returned CD is from the same customer or from another customer doesn't matter, the customer's identity is not needed to carry out the transaction.

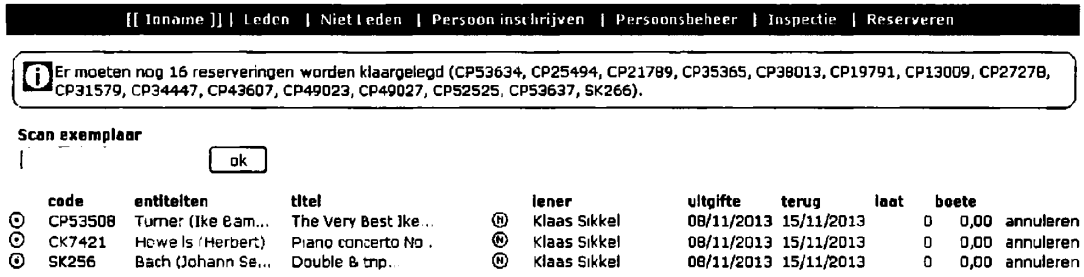
For renting out CDs the situation was different. Renting out a CD involves both scanning the membership pass and scanning the CD. If there are several CDs, the membership pass needs to be scanned only once. Therefore, renting multiple CDs has been modelled as a single, composite transaction.

For returning CDs, there is no need for a composite transaction. Each return can be handled separately.

Employees experience this very differently. A customer returns a pile of CDs, and handling the whole pile is felt to be one coherent sequence of actions. This is interrupted by the pop-up screen which forces the employee to engage in another activity. While the employee handles the reservations, a pile of half-finished business remains on the counter. Employees don't like half-finished business, particularly when it's busy: it clutters the counter with piles for which you have to remember what their status is.

This is an interesting example of a piece of design that is perfectly logical from the system's perspective (registering a returned CD is a transaction by itself, not related to other returned CDs) but illogical from the end user's perspective (registering all CDs returned by one customer is a coherent sequence of actions). In terms of Requirements Analysis (*Pearls of Computer Science*): all system-level requirements have been duly implemented, but a business-level requirement was overlooked. From the perspective of the employees, this is a design error.

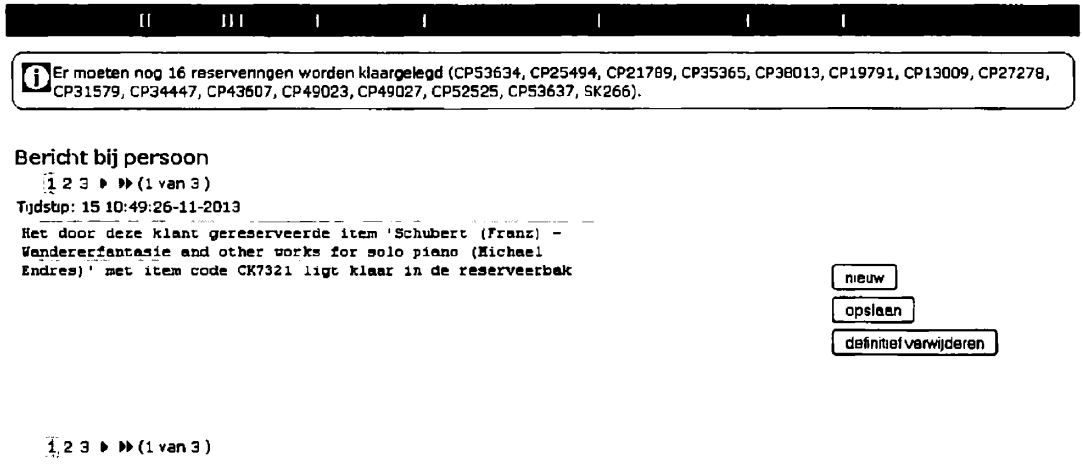
With this analysis of why the employees are unhappy about the system design, it is not difficult to come up with a solution:



A screenshot from the user interface—in Dutch—for the employee. It displays

- *a top menu bar "Return | Members | [...]";*
- *a message: "16 reservations have to be set apart (CP 53634 [...])";*
- *a text box "Scan item" (can be entered by hand if scanning the bar code doesn't work);*
- *a listing of the most recently returned CDs.*

Figure 2.2: User interface for the registration of returned CDs (Exercise D-2.9)



A screenshot from the user interface—in Dutch—for the employee. It displays

- *a top menu bar (greyed out);*
- *a message: "16 reservations have to be set apart (CP 53634 [...])";*
- *a screen title "Message about a person";*
- *an indication that the first of three messages is shown;*
- *a text box displaying the message: "The item 'Schubert (Franz) – Wanderfantasie and other works for piano solo (Michael Endres)' with item code CK7321 has been set apart for this customer on the shelf for reserved items";*
- *buttons "new", "store", "permantly delete", (grayed out:) "continue".*

Figure 2.3: Message that a reserved CD has to be taken from the shelf (Exercise D-2.9)

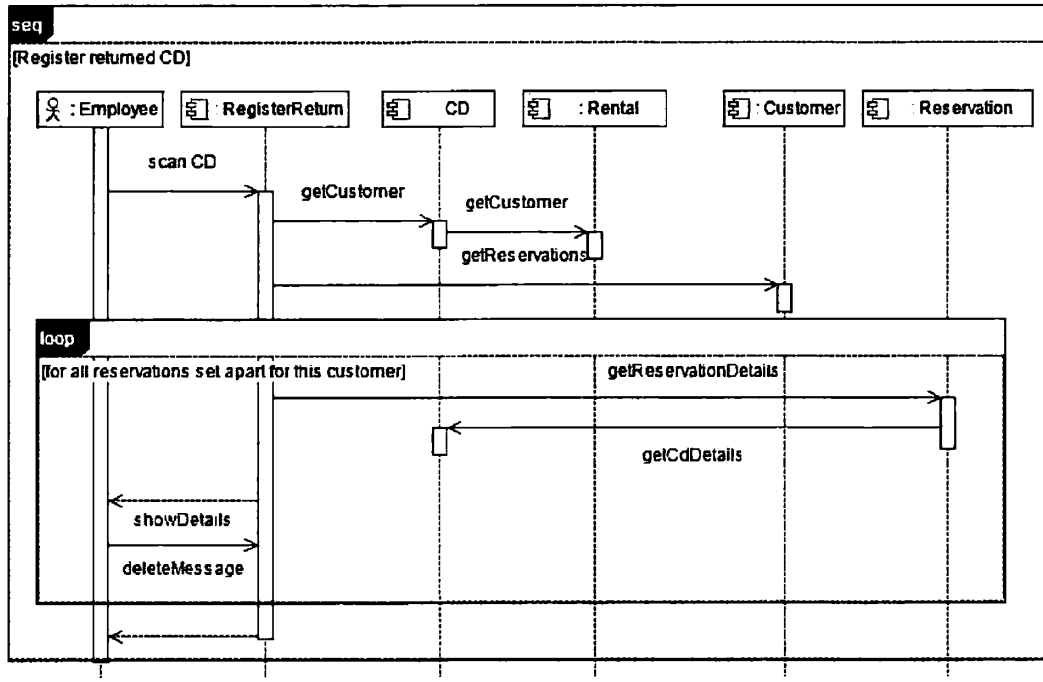


Figure 2.4: Sequence diagram for returned CDs and getting CDs that were set apart (Exercise D-2.9)

- Do not force the employees to interrupt what they are doing, but let them organize the work as they think best.
- Instead, display a message in the top of the screen that reserved CDs have to be taken from the shelf for customer *X*, similar to the message about items that have to be set apart.

The employee will then look for reserved items after they have finished scanning the returned CDs.

☞ **D-2.9** Redesign the sequence diagram in Figure 2.4. Make two separate sequence diagrams, in line with the above explanation.
You find Figure 2.4 in lab files for this week.

2.3.5 Recommended exercise 2b (Sequence Diagrams)

Medication support
Exercise D-2.10 is related to the case study about medication support from the recommended exercise 1b (Section 1.3.5, page 32). Please (re-)read the full case description there, as well as the following additional information.

Changing a service plan
Changing a service plan is organized as follows (described in more detail here than in Section 1.3.5).

When a nurse starts this function, they first choose the client for whom the service plan is to be changed. When this client has been chosen, their service plan is deactivated (only if it was active at the time). Subsequently, the following changes can be made;

- change the dose of a particular medicine,
- indicate that a medicine is no longer taken by the client,
- indicate that a new medicine has been prescribed to the client
- change the times at which the dispenser releases the medication.

Elderly people often use multiple medicines for multiple illnesses and conditions, so each of these changes can be applied more than once when the service plan is adapted. There is no prescribed order, a nurse can do the steps in any order they like.
(It is not possible to change the service pattern. Would that be required, then a new service plan has to be installed—which is something different from changing a service plan)
Finally, the service is activated.

The main structure of the sequence diagram has been elaborated in Figures 2.5 and 2.6. Figure 2.5 models the service plan in the manner that was used in Lab session 2b: with a lifeline for a control object for the use case. Figure 2.6 is a simplification (see the lecture slides). The control object is not represented in the diagram: the actor interacts directly with the relevant objects.

The structure of Figure 2.6 avoids a lot of double arrows. Therefore it is easier to continue on the basis of Figure 2.6. For the essential part of the sequence diagram there is a reference to another sequence diagram *Change services*.

D-2.10 Make a sequence diagram for *Change services* that fits to the top-level design in Figure 2.6.

2.4 Programming

2.4.1 Laboratory exercises

Rectangle

For more info, read N & H Section 5.3

Listing 3.4 in Niño & Hosch (page 149) contains an informal specification for class `Rectangle`. In the material provided for this week, there is a stub-implementation for this class. In these exercises, you will make write a formal specification for this class, provide an implementation and validate the implementation. Create the package `ss.week2` and put this implementation into this package.

For more info, read Appendix A

P-2.1 Write a JML specification for class `Rectangle` that captures its informal specification. Your specification should have pre- and postconditions whenever this is meaningful, and specify appropriate class invariants. Use the OpenJML Eclipse plug-in to make sure your specification typechecks.

For more info, read N & H Section 5.1.1

P-2.2 Implement the method `area` in class `Rectangle`, and add assertions to validate your specifications, as explained in Niño & Hosch. The assertions should be sufficient to validate pre- and postconditions and the class invariants. For the query methods, it is sufficient to assert that the class invariant holds in the beginning of the method. Explain why.

P-2.3 Your specifications of `area` and `perimeter` are probably specific to rectangles. How could they be generalised to apply to arbitrary geometric figures? What would be the assertions corresponding to the postcondition in that case?

P-2.4 Now implement the remainder of class `Rectangle`, and add assertions to validate your specifications.

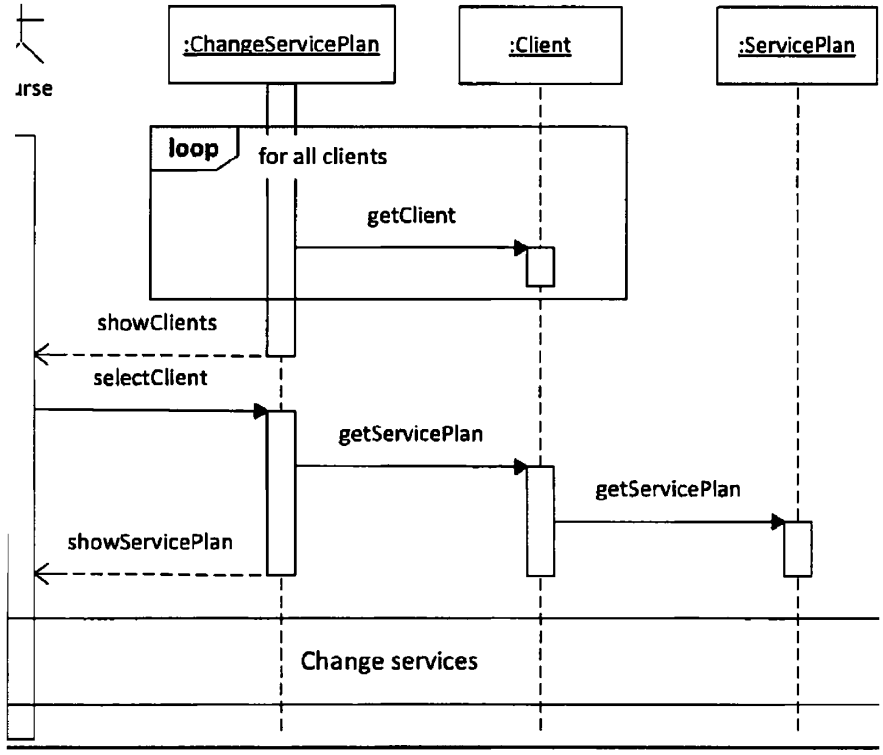
Last week, you built a simple test for the three-way lamp. A more complex test can be found in `ss.week1.test.GuestTest`, which automatically compares expected outcomes with the real outcomes.

☞ **P-2.5** Create a test class to validate `Rectangle.java`. Use `ss.week1.test.GuestTest` as an example. Use the `assertEquals` method to compare expected and real results.

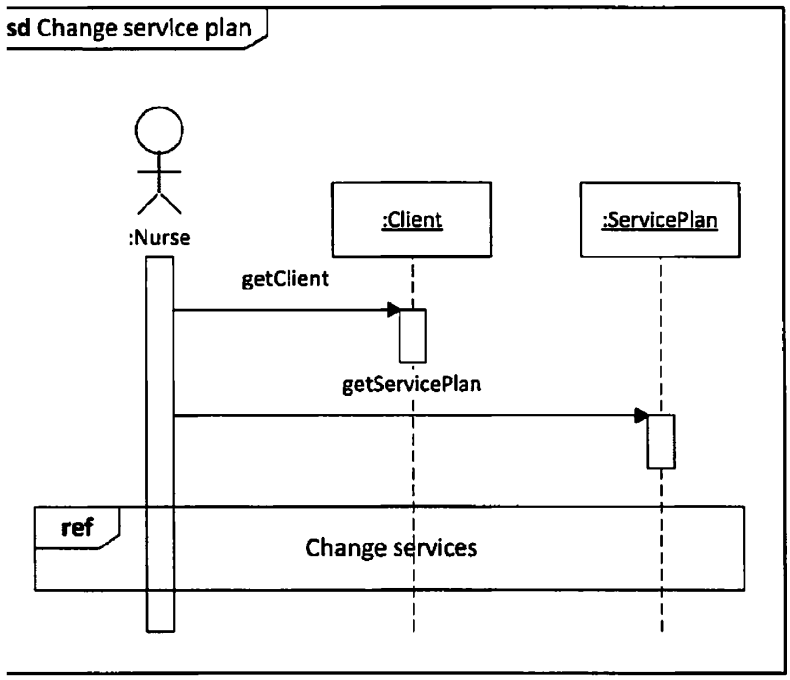
For more info, read N & H Section 6.3

Three-Way Lamp

Last week, as part of Exercises P-1.5 and P-1.6, you developed an informal specification and an implementation of a three-way lamp. **Copy** the implementation from Week 1 to Week 2 in the following way. In the Eclipse PACKAGE EXPLORER view navigate to the package `ss.week1` in the project `ss` and expand the package. Now select the class you want to copy and right-click it to open the context menu. In the context menu select COPY. Now select the package `ss.week2`, again right-click it and select `Paste` from the context menu. Eclipse will automatically adapt the package declaration in the first line of the class definition.



.5: Sequence Diagram for *Change serviceplan* (elaborate version) (Exercise D-2.10)



6: Sequence Diagram for *Change serviceplan* (simplified version) (Exercise D-2.10)

P-2.6 Formalise your specifications in JML, and type check them with OpenJML. Next, add assertions in your implementation that capture the specification. *Note:* for the JML specification you have to explicitly ensure that OFF goes to LOW, LOW goes to MEDIUM etc.

The three-way lamp implementation is a typical example where the state of the object only can take one of a limited set of values: the lamp is either *off*, *low*, *medium*, or *high*. In your implementation, you will have used constants (*i.e.*, `public static final` variables, probably of type integer) to define these possible conditions. In your specification, you will have specified a class invariant that the state of the lamp always holds exactly one of these options. If you would want the lamp also to be initialised in one of these states, and pass this as a parameter to the constructor, then you would have to specify a precondition that the integer argument to the constructor should be equal to one of these constants.

For more info, read N & H Section 5.4

This pattern occurs very frequently in Java programs, and therefore a special *enumeration* type construct has been provided that allows to capture precisely this situation.

The syntax to define an enumeration type is as follows:

```
public enum <TypeName> { <val1>, <val2>, ..., <valn> }
```

This declaration may be at top-level, *i.e*, it does not have to be inside a class. Values of the enumeration are now referred to as `<TypeName>.<val1>`, `<TypeName>.<val2>` etc., and variables may be declared to be of type `<TypeName>`. If a variable is declared to be of an enumeration type, the class invariants and preconditions as mentioned above are no longer necessary, as the Java compiler guards agains type violations. As values (`<val1>`, `<val2>`, etc.) you can specify arbitrary Java identifiers, e.g., `off`, `low`, etc. Java identifiers can consist of characters, digits and the underscore character "_" and they must start either with a character or an underscore. *However, by convention, Java enumeration values, like constants, are written in all-caps style. We strongly advise you to adhere to this convention.*

Enumeration types have not always been part of Java; they have been added to the source code language in Java 1.5. The compiler transforms them into special classes. Thus, enumerations are only syntactic sugar, but very useful. You can find more information and examples in Niño & Hosch (§5.4: Enumeration classes).

☞ **P-2.7** Declare an enumeration type for the state of the three-way lamp. Make a copy of your original file, and then rewrite your specification and implementation to cater for the use of enumeration types. Make sure your `ThreeWayLampTest` still runs as expected. *Note:* you cannot use the modulo (`%`) operator on values of your **`enum`** type. However, you can use the **`switch`**-statement.

For more info, read N & H Section 4.8

Hotel

In this exercise, we will make a further extension to our hotel application. Each room of our hotel has a safe, which guests can rent upon request. Each safe is protected by a code: it only opens when a valid code is entered. Additionally, each safe can be (de)activated: only an active safe can be opened, and activating can only be done with the correct code.

Create the new package `ss.week2.hotel` and copy your class `Password` from the package `ss.week1` to this new one.

P-2.8 Design and specify a class `ss.week2.hotel.Safe` with the functionality as described below. Use last week's class `Password` to store the safe's password. The specification should be an empty stub, with comments and JML pre- and postconditions.

The class should contain the following commands:

- `activate`: receives a `String` with a password text as a parameter, and activates the safe if the password is correct;
- `deactivate`: without parameters, closes the safe and deactivates it;
- `open`: receives a `String` with a password text as a parameter, opens the safe if it is active, and the password is correct;
- `close`: without parameters, closes the safe (but does not change its activity status).

and the following queries:

- `isActive`: indicates whether the safe is active;
- `isOpen`: indicates whether the safe is open;

- `getPassword`: returns the password object on which the method `testWord` can be called to check the password.

Additionally, you should decide yourself about the constructor's functionality.

Make a class diagram of your design for class `Safe`.

In principle, this class can be tested in the same way as last week. This time, you are asked to develop and program your own test, rather than copying and modifying an existing one.

Put your tests in a separate package `ss.week2.test`. This means that your tests use classes from a different package than the one they themselves are located in; these classes must therefore be declared in an import statement. See Niño & Hosch (§2.9) for the Java syntax.

To create a JUNIT class in ECLIPSE:

- Select and right-click the package `week2.test`
- Choose NEW → JUNIT TEST CASE;
- Enter the name of the test class and select the option to create a `setUp()` method;
- Press FINISH.

You can also create a *test suite* to bundle tests and run them at once. How to do this can be found at the JUNIT website: `https://github.com/junit-team/junit/wiki/Aggregating-tests-in-suites`.

For more info, read N & H Section 6.2 & 6.3

P-2.9 Make a test plan for `Safe`, using your own specification as starting point, and implement a JUNIT test `ss.week2.test.SafeTest`, following the instructions above to create a fresh test class.

In the generated test class, you will see two as yet empty methods: one called `setUp`, which is annotated with `@Before`, and one called `test`, which is annotated with `@Test`.

- When JUNIT runs a test class, the method annotated `@Before` is automatically run *before any test case*. This is therefore a good place to put any initialisation that creates a useful initial state for your test. For instance, in `SafeTest` you can initialise an instance variable of type `Safe` with a fresh instance of the class.
- A method annotated with `@Test` is called a *test case*. When JUNIT runs a test class, the test cases are executed and reported individually. It is good practice to create many small test cases and give them meaningful names; for instance, in `SafeTest` you can create test cases `testActivateCorrectPassword`, `testActivateWrongPassword`, `testOpenCorrectPassword` etc., each of which tests for that single property.

Now you are ready to implement and test the class `Safe`.

P-2.10 Implement the class `Safe` as you specified in Exercise P-2.8 above. Specify class invariants where appropriate. Additionally, add **`assert`**-statements capturing your method specifications and class invariants. Test your class using the class `SafeTest`.

P-2.11 Execute the class `SafeTest` again, but this time while measuring the test coverage. For this purpose, execute the test via the COVERAGE menu (rather than the RUN menu). (You need to have installed the EMMA plugin).

The COVERAGE view of ECLIPSE shows which percentage of the code has been executed during the test run. When you double click on a class in this view, it will be opened in the editor. Lines in the editor are highlighted in different colors: green means the line was fully covered, yellow means only some statements on the line have been executed, red means the line has not been executed at all. Answer the following questions:

- Which packages and classes have been covered the least/ the most?
- Why is this the case and is it a problem that some classes are covered to 0%?
- Can you improve your test class to increase the coverage?

Your implementation of class `Safe` contains several **`assert`** statements. As described in Niño & Hosch, these statements are only executed if the virtual machine (i.e., the JAVA application) is called with a special option; otherwise, they are simply skipped during execution.

☞ **P-2.12** Give your class `Safe` a `main` method that calls the constructor or a method of `Safe` so that the precondition is violated. Execute the program in Eclipse in the following two ways:

- Executing as usual: select the class `ss.week2.hotel.Safe` in the editor, then choose RUN → RUN AS → JAVA APPLICATION.
- Executing with assertion checking enabled: choose RUN → RUN CONFIGURATIONS. In the dialog that opens, select the run configuration `Safe` from the category JAVA APPLICATION in the list to the left. Next, choose the tab ARGUMENTS in the right part of the dialog window. In the text box VM ARGUMENTS enter `-ea`, and select RUN. (If you want to perform the usual execution again, repeat these steps and remove `-ea` from the VM ARGUMENTS.)

What does the option `-ea` mean? Describe the effect of both executions, and explain the difference.

Now copy your classes `Guest` and `Room` from `ss.week1.hotel` to `ss.week2.hotel`.

P-2.13 Add a field `safe` to the class `Room`. This attribute should be initialised in the constructor of `Room`, and an appropriate query should be defined for it.

Add appropriate tests to `RoomTest` (provided on Blackboard).

Now we are going to design and implement a class `Hotel` combining all these classes. For simplicity, we assume that the hotel only has 2 rooms. Guests can check in by using their name, and there cannot be two different guests with the same name. Checking in is protected by a password, that is, the check in operation only succeeds if the password argument is correct.

P-2.14 Design and specify a class `ss.week2.hotel.Hotel` with the following functionality:

- A command `checkIn` that receives two `String` objects as parameters. The first parameter is the password for checking in; the second parameter is the name of the guest. The method returns a `Room` object with a (new) `Guest` of the given name checked in; or `null` if either the password is wrong, there already is a guest with this name, or the hotel is full.
- A command `checkOut`, receiving the name of a guest as a parameter. The guest is checked out, and the safe in his room is deactivated. If there is no guest with this name, nothing happens.
- A query `getFreeRoom`, returning a `Room` where there is no guest checked in, or `null` if there is no free room available.
- A query `getRoom`, receiving the name of a guest as parameter, returning the `Room` object where the guest is checked in, or `null` if there is no such room.
- A query `getPassword`, returning the `Password` object of the hotel (which is used by `checkIn`).
- A query `toString` that gives a textual description of all rooms in the hotel, including the name of the guest and the status of the safe in that room.

Additionally, the hotel should have a property `name`, which is set at initialisation, with an appropriate query. Extend the class diagram of P-2.8 with the new classes; the class diagram should show all classes in the package `ss.week2.hotel`.

The test plan and test class have been developed for you; you can find them as `ss.week2.test.HotelTest` on Blackboard.

☞ **P-2.15** Implement the class `Hotel` as designed and specified above. Where appropriate insert class invariants and `assert` statements. Test your implementation using the provided test class.

2.4.2 Recommended exercises

P-2.16 Make the following exercises from Niño & Hosch.

- First line of 5.2, 5.7 (but write your specifications in JML, and typecheck using OpenJML).
- 6.2 and 6.5. In 6.5 you can assume that the `Employee` has methods `setRate()` en `setHours()`. Also write the appropriate `setUp` and test methods.
- 8.2a, 8.4 and 8.8.

2.4.3 Bonus exercises

P-2.17 Consider class `Rectangle` and its specification, as developed in Exercises P-2.1–P-2.3. Remove the **`assert`**-statements from `Rectangle`, and adapt the method `main` in `RectangleTest` so that it just executes the code but without invoking the tests

- Use the run-time checker to validate your implementation w.r.t. the specification.
- Now introduce an error in your implementation, and use the run-time checker to detect the error.
- Finally try to use the static checker and try to complete your specifications so that it is accepted by the static checker.

P-2.18 Do the same as above, but for your specification and implementation of the `ThreeWayLamp`.

Week 3

3.1 Overview

3.1.1 Contents of This Week

Academic Skills This week contains one peer feedback session dedicated to academic skills on Monday. This meeting will consist of a 30-minutes peer feedback meeting in which you are expected to be present and actively participate.

Design The activities in this week cover the following topics:

- *State machine diagrams.* These describe states that system components can be in, and transitions between states. See 3.3.1 for exercises for Lab Session 3a.
- *Versioning.* How to handle different versions when multiple persons are working on the same project. This is in fact a project management issue (more than design or programming). In this module is it included in the Design thread. See §3.3.4 for the exercises for Lab Session 3b.
- Project work, continued from last week, see 3.3.3;

Programming This week the following topics will be discussed:

- Abstraction and inheritance (notice that this coincides with the notion of *generalisation* discussed in the design thread)
- Overriding methods
- Specification and method contracts in the context of inheritance and method overriding
- Introduction to Security Engineering: threat modeling and mitigation approaches.
- A diagnostic test that will allow you to judge for yourself how well you have understood the basic programming concepts

3.1.2 Mandatory Presence

During the following activities, your presence is mandatory.

- (A) Peer feedback session (Mon 6 - 7)
- (M) Self-study supervised (Tue 6 – 7)
- (M) Tutorial (Thu 3 – 4)
- (P + D) Diagnostic test *Basic Programming and Design Concepts* (Thu 9)

3.1.3 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 4 hours self-study for mathematics;
- 2 hours self-study for the design thread;
- 2 hours design project work beyond the session with assistance; and
- 2 hours self-study for the programming thread.

3.1.4 Materials for this Week

Academic Skills No reading material this week.

Design

Slides from lectures 3a and 3b

Programming

Lecture Niño & Hosch, Chapters 9–11

Laboratory The following predefined files are provided on Blackboard:

- `ss/week3/OperatorWithIdentity.java`
- `ss/week3/test/AdditionTest.java`
- `ss/week3/test/MultiplicationTest.java`
- `ss/week3/test/TimedPasswordTest.java`
- `ss/week3/test/PricedRoomTest.java`
- `ss/week3/test/PricedSafeTest.java`
- `ss/week3/hotel/Bill.html`

See §3.4.1 for a brief discussion of some recent extensions to Java (since version 8), enabling useful new features that did not exist when Niño & Hosch was written (and partially contradict what is stated there).

3.2 Academic Skills

3.2.1 Peer feedback session

In groups of 4 students, the project group for the Design project, you are asked to reflect on the execution of your personal week-planning. *You should bring your planning & monitoring sheet which you have kept in Week 2 with you to your peer feedback session.*

The remaining time of the scheduled 2 hours can be used for project work

3.3 Design

3.3.1 Laboratory session 3a (State Machine Diagrams)

The laboratory session is done in groups of two students.

In this session we will look at the CD Rental again. See Section 2.3.1, page 43, for the full case description. In a few steps, we will make a state machine diagram for the state of a CD.

CD Rental (simplified version)

For the first version of the state machine, we'll simplify things a bit.

CDs can be rented to customers. Renting out and registering returns is done by an employee of the CD Rental. Members can make a reservation through the internet. A reserved CD can only be rented to the person who reserved it. If there are more copies of a CD, and a reservation is made, one of these copies will be reserved. If there are more reservations than copies available, the reservations will be dealt with in the

order in which they were made. That is: if a copy of a reserved CD is returned, it is assigned to the next reservation in the queue.

For the time being we disregard setting apart reserved CDs. In other words, setting apart a CD (for now) does not change its status. CDs can be in the CD Rental (whether in the collection or set apart does not matter) or rented out.

We also disregard that CDs can be new (and we don't consider entering new CDs into the collection).

Likewise, we're not interested in the fact that not everyone can rent every CD. If renting out a CD is refused for whatever reason, then—from the perspective of the state machine—nothing happens; the state of the CD doesn't change.

Remember that state machines are concerned with the state of an object as represented within the system. If an inspection of a CD at the counter reveals some new scratches, these are added by an employee to the physical scratch card. So, in a real-world sense, the state of the CD has changed. But this is not represented in the information system and therefore not represented in the state machine.

☞ **D-3.1** Make a state machine for a CD for the simplified version of the CD rental.

Before you start drawing a diagram, you'd have to ask yourself:

- What are the relevant events (use cases that will correspond to transitions in the state machine)?
- Which states are needed to represent all possibilities?

Hint: Take into account that a member can reserve a CD at any time. What happens when a member reserves a CD while it is rented to someone?

Setting apart reserved CDs

The first extension that should be made to the simplified description above is the handling of reserved CDs.

If a (not-reserved) CD is reserved by a member, the employees are notified by a message in the screen (see the user interface in Figures 2.2 and 2.3 on page 51). When an employee picks up this CD from the collection they scan it (in a 'Reservation handling' screen) and put it on the shelf. By scanning it, the system knows that is has been set apart. When the reservation has been made, but the CD has not yet been set apart, it is not possible to rent it out. Even if the customer who reserved it visits the CD rental, takes it from the collection and hands it to the employee, it still needs to be registered as 'set apart' before it can be rented.

If a CD with a pending reservation has been returned, the screen will show a message that the returned CD has to be set apart for the next reservation. In this case, also, the employee has to scan it as 'set apart' before they put it on the shelf. (The reason for this design is that is happens quite often, mostly when it's busy, that employees overlook that a returned CD is reserved for someone else, so it gets returned to the collection. But it will show up in the list of other CDs to be set apart, and eventually be picked up and set apart on the shelf.)

When a CD is scanned by an employee as 'set apart', the (next) person who reserved it automatically gets an e-mail with a request to pick it up within 72 hours. If it hasn't been rented out within 72 hours, the reservation expires. In case there are no further reservations, the CD is still set apart, but no longer reserved. A message to that effect will appear in the screen. An employee should take it from the shelf, scan it as 'no longer set apart,' and return it to the collection. A CD that has been set apart but is no longer reserved, can not be rented out until the 'set apart' status is withdrawn. However, it could be reserved by a member (*at any time*, thus also in this situation). If that happens, it should remain to be set apart, and the message on the screen simply disappears.

D-3.2 Adapt the state machine accordingly.

The life cycle of a CD

Regularly, new CDs are added to the collection.

Newly acquired CDs are registered in the system by the back office. The back officer enters the required details into the system, and a new item code will be assigned.

Exactly three months after the time and date of registration, the status will automatically change from *new* to *not new*.

In principle, CDs are never removed from the collection. Public libraries (including all OBD libraries) has what they call "rational collection management": if a book has not been borrowed for a long enough period, it is no longer needed in the collection and can be discarded. The CD Rental boasts to have "irrational

collection management," nothing is ever discarded. (*Reality is not quite so simple, but here we may disregard handling of CDs that are stolen, lost by the customer, damaged beyond repair, etc. When possible, these are replaced by new copies at the expense of the customer*).

☞ **D-3.3** Include the CD life cycle in the state machine diagram.
Hint: take into account that you don't know what the state of a CD will be when its status is changed from *new* to *not new*.

3.3.2 Recommended exercise 3a (State Machine Diagrams)

D-3.4 Draw a state machine diagram for the state of a pet, according to the case description below.

Pet Registration Database
Following up on pressure from the Parliament, The Ministry of Agriculture, Nature, and Food Quality is drafting a regulation for nation-wide registration of pets. A successful regulation will consist of the following three elements:

- The technical basis. Registration will make use of so-called RFIDs, which can be implanted subcutaneously in animals. For this purpose, an RFID is stored in a non-decomposable synthetic tube with a lenghth of 12 mm and a diameter of 2 mm, with a total weight of 0.11 g. An RFID has no power source of its own, but its contents can be read by applying an electromagnetic field.
- Operation of a nation-wide database. The purpose of the database is to register pets that have gone missing or have been stolen, so that the rightful owner can be traced when the pet has been found.
- Rules and procedures for the registration of pets. In future, it is expected, the registration of certain types of animals (cats, dogs, horses) will become mandatory. In the foreseeable future, registration is voluntary.

The database will be run by the Animal Database for the Netherlands (ADN). Employees of ADN wil operate and maintain the system, and will provide statistics to relevant government bodies. ADN also maintains the register of authorized parties (vet practices and animal shelters) that can add or change registrations of animals.

Some registrations can be made by the pet's owner, some only by authorized parties, some by all parties involved.

- Everyone who owns a pet can have it registered by a veterinarian. The vet (or an assistant) implants the RFID in the pet, gives the owner a registration certificate (on paper) with all the information and registers the pet with ADN. The RFID code consists of a unique 15-digit number. The composition of this number has been standardized across the EU. The first three digits identify the country ('528' for the Netherlands), the next three digits identify the RFID producer, the remaining nine digits identify the animal.
- A change in address can be entered into the system by the owners themselves. Changing the ownership of an animal can be reported by the old owner on the ADN website. To make sure that this is done properly, ADN will contact the new owner for a confirmation.
- Occasionally it happens that pets run away. Also, certain types of animals can get stolen (e.g. koi carps, who do get RFIDs for this very reason). If an animal is missing, the owner can report this by means of a web form on the ADN website.
- When people find a run-away pet, they can bring it to the nearest animal shelter. Depending on the circumstances one can also call the animal ambulance to pick it up. An employee of animal shelter scans the pet's RFID code, registers it as "found" with ADN, and gets the information about the owner from ADN. ADN registers the date on which the animal was found and the animal shelter that made the registration. If an e-mail address is known, the owner automatically gets a notification where the pet can be picked up. In addition, the animal shelter always phones the owner. When the pet is picked up, an employee of the animal shelter records in the ADN database that this has happened.
  With found pets, the following special cases can be distinguished:

  - It could be dead, or so badly wounded that is has to be killed.
  - Sometimes an animal is found that carries no RFID. The animal shelter always implants an RFID (unless the animal has to be killed) en registers it with ADN. An owner may turn up later, in which case the animal shelter updates the registration.
  - It also happens that a pet with RFID is found, which has not been reported missing. In that case the animal shelter contacts the owner. The fact that the animal was not missing could be an indication of suspicious circumstances. The animal shelter may contact the Animal Protection Society, which could further investigate the case.
    If the owner cannot be traced or refuses to pick up the pet, the animal shelter contacts the Animal Protection Society, who can dispossess the owner.
  - Animals without owner stay in the animal shelter until a new owner can be found. Fortunately, there are a lot of people who get a new pet from the shelter. If this happens, the animal shelter updates the registration.
    If the shelter's capacity if fully used, animals for which no new owner can be found have to be killed.

- If a pet is still missing after three months, ADN will change it's status from "missing" to "lost forever". The owner gets a letter in which it is explained that, unfortunately, their pet has not been found and, after so much time, it is unlikely that this will ever happen. Nevertheless, once in a while, a permanently lost pet is found. In that case the procedure is the same as for a missing pet.
- If an animal dies, any of the concerned parties (owner, vet (assistant), animal shelter employee) can register this with ADN. Many owners forget to do this, creating some data pollution in the ADN database.

3.3.3 Project

D-P.3 This week you could finalize the requirements list, glossary, and the actor list and get a draft version of use case diagrams, and use case descriptions. You may also want to make a start with the class diagram.

3.3.4 Laboratory Session 3b (Versioning)

This laboratory session is done individually.

One of the practices we strongly suggest you to adopt from the start in *any* project you work on — be it individually or collaboratively — is to version your files. This wil put you in control of your own work: you can undo, restore, branch, merge, and share everything with much less effort and problems. (After working through the exercises in this lab sessions you will know what these terms mean, if you don't already.)

For this module, the versioning system of choice is GIT. Among other things, this has the advantage of being widespread, efficient, and supported out-of-the-box by ECLIPSE. Compared to some other systems such as SVN (SubVersioN), GIT is conceptually more complicated; reason enough to devote this lab session to getting acquainted.

Even apart from the general usefulness of versioning in general and GIT in particular, you will be expected to use GIT for the final project of the module. All in all, we assume that getting to know it is something you are motivated for without artificial pressure. Hence, the only thing you have to show to get this lab session signed off is that you have created a GIT repository on some external host, with some branches and snapshots. If you complete the exercises, you are sure to have achieved that state.

D-3.5 Go through the online GIT tutorial at `https://try.github.io`. This will acquaint you with some of the basic concepts and commands of GIT used on the command line.

D-3.6 What does GIT stand for?

Though the above is a nice way to get a first idea about GIT and to see things in motion, it is not very systematic. We strongly recommend you to read up on the underlying concepts in one of the many tutorials around. A good one is `https://www.atlassian.com/git/`.

Central and local repositories At this point, it's time to really get going and create a repository of your own. This and the next exercises are spelled out on a quite fine-grained level. Do not just dumbly follow the steps: think actively on what is happening, and ask assistance if you feel that you do not understand what's going on! Also, feel free to try out things on your own. (In the meanwhile, do realise that GIT is very powerful and flexible, making it seem like a monster when you meet it the first time. Quite likely you will never have to use any of the more advanced features, and it's quite OK to ignore them.)

D-3.7 (Your first repository)

1. Get an account at GitHub `https://github.com` or Bitbucket `https://bitbucket.org`.
2. Create a (bare, initially empty) remote repository at that account.
3. In ECLIPSE, go to the GIT perspective and create a local repository on your machine, with the remote one as its origin.[1]
4. Go back to the JAVA perspective, create a fresh JAVA project and put a trivial, single class `Hello` in it.
5. Add your project to the (local) repository using TEAM → SHARE from the project's context (i.e., right-click) menu.
6. Modify `Hello` and commit it to your local repository (via TEAM → COMMIT)
7. Push your local repository to the origin (via TEAM → PUSH TO UPSTREAM). (You could actually have used COMMIT AND PUSH in one go in the previous step.)
8. Check your original, online repository: this should now contain the latest version of `Hello`.

Rolling back One of the greatest advantages of versioning is, as the word implies, the ability to retrieve an older version in case you've made a change that you regret. In fact you can go one better: you can not just undo the latest change, but *any change* in the past as long as later changes are not dependent on the one you want to undo. The idea is that you actually apply the change in reverse.

D-3.8 (Rolling back)

1. Add a method to your `Hello` class (from Exercise D-3.7) and commit it. Note that you have to enter a *commit message* every time you do this.
2. Add another class to your project, and commit it as well. This change is clearly independent from the previous one.
3. Select your project, and select TEAM → SHOW IN HISTORY from the context menu. This opens a view showing the sequence of commits for the project,as stored in the repository (with your commit messages to guide you as to what happened: a good reason to use meaningful messages!)
4. Select the one but last commit in the history view, and select REVERT COMMIT from the context menu. This should result in the removal of the method added in step 1 above, while the class added in step 2 remains. You just undid a change made two steps ago, while the change that followed it was unaffected!
5. Note that another item just appeared in the history, reflecting the reversion of the earlier commit. You can also undo the undo, and so on.

Pushing and pulling Remember, you have a local repository (on your machine) and a central one (on github or bitbucket). Committing means you put the changes in your project into the local repository, *pushing* means you send the changes in your local repository to the central one. You can do this in one go, but it can be advantageous not to do so straight away — for instance, if you're currently not connected to the internet, or if you want to complete a bunch of edits before exposing them to anyone else.

This setup implies that there can be many local repositories each "connected up" with the same central one. (This central repository is called the *origin*.) These different local repositories are typically on different machines, and may be under the control of different users who make their own choices. GIT goes a long way towards making this all work together smoothly; in particular, taking care that edits by different users can be integrated afterwards. Of course that can't always work, for instance if those edits *conflict*, meaning that they did incompatible things with the same file.

D-3.9 (Pushing and pulling)

[1] The origin is sometimes also called the *upstream* repository.

1. Create another local repository, on you lab partner's machine, by cloning the central repository that you created in Exercise D-3.7. (If you do not have a second machine available, you have to start up a second copy of ECLIPSE with another workspace, as it is not possible to have two distinct projects with the same name.)
2. In ECLIPSE on that other machine, re-create your JAVA project based on the new repository. You can do this by selecting IMPORT PROJECTS from the context menu of the new repository (in the GIT perspective), and then selecting IMPORT EXISTING ECLIPSE PROJECTS.
3. Do some edits in one of the local repositories, commit and push. (Maybe you still have some unpushed edits left over from Exercise D-3.8.)
4. In the *other* local repository, execute a *pull* by selecting TEAM → PULL from the project's context menu. This should result in an update such that the state of these two projects is now the same.
5. In the two local repositories (sharing the same central one), independently edit the same file in two separate places, and try to push the changes from both repositories to the origin. What happens?
6. Do this again, but now editing the same lines of the same file. What happens?

The last two steps involve *conflict resolution*. If GIT can discover that two changes affect different parts of a file, it will resolve them automatically; if not, this is left to the user.

Branching and merging The last concept we'll cover here is that of *branches* in a repository. A branch is a copy *within* a repository, with its own name, of all your files. You can work on (modify, improve, adapt) a branch without affecting other users such as your project partner (as long as they work on other branches), *even while committing and pushing your changes*. Essentially, your changes stay within your branch. However, at a later stage you can *merge* your branch back into the main development stream — which effectively is nothing else than a branch itself, usually called the *master*; or, alternatively, merge changes from the master branch into yours (or indeed, from/to any other branch).

☞ **D-3.10** (Branching and merging)

1. In your GIT project, select TEAM → SWITCH TO → NEW BRANCH. Call the new branch `try` or some other clever name. Do *not* select CONFIGURE UPSTREAM FOR PUSH AND PULL. Note that you can see which branch your project is on in the PACKAGE EXPLORER view of ECLIPSE— as well as the number of unpushed commits.
2. Create a new class in your project, and add a method to another class. Commit and push. These edits are now part of the new branch, but are *not* visible in the master branch.
3. Switch to the `master` branch. Note that your `try` edits are now gone. Do some *different* edits here.
4. Switch back to `try`. Select TEAM → MERGE, select the `master` and do MERGE. The result should combine the edits you did in the two branches.

At the end of this exercise, show a student assistant your repository with the various branches and commits, to get the lab session signed off.

Branches are a great tool if you want to develop a new feature in isolation, without affecting anyone else working on the project. It really is the thing that makes versioning indispensible in larger projects. You are very much recommended to create branches liberally, even for small extensions or bug fixes. Just do remember to merge the master branch into yours on a regular basis — and only merge back when you're done. When you're *really* done, you can even delete the branch altogether. (This is an old-fashioned concept called *cleaning up*.)

Take it from here This is just the beginning, but it should allow you to work fruitfully with GIT. Staging areas, submodules, rebasing, subtrees... maybe it's all in store for you; but even if this is not your cup of tea, just remember: version everything, make it a habit, and you will very soon be glad you did.

3.4 Programming

3.4.1 Java 8: Default interface methods

The Java programming language has evolved quite a bit with respect to the book we use for this module. There is simply no time to discuss all language features, but one recent *very* important extension, which greatly enhances the power of the language to keep programs more concise and avoid code duplication, is the concept of a *default method* in an interface. Essentially, a default method is a non-abstract interface method that is inherited by all implementing classes (in direct contradiction to Section 9.2.1 of Niño & Hosch, which states that interfaces may only have abstract methods). Since interfaces may *not* have instance variables, such a default method can only provide functionality by calling other methods. Still, this is enormously useful in order to provide rich interface functionality without forcing all classes that implement such an interface to actually implement all its methods. In fact, with this extension, Java has come much closer to supporting multiple inheritance.

This new functionality is only supported in newer versions of Java; in particular, for the interface methods you have to have Java 8.

For instance, in Java 8, the interface `Player` in Niño & Hosch, Section 9.2.1 can be extended with a method

```
/** Indicates if this Player has not yet taken a turn. */
public default boolean firstTurn() {
    return numberTaken() == 0;
}
```

This would immediately provide *all* implementing classes with a method of that name. It could still be overridden at need. For further information see `http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html`.

3.4.2 Laboratory exercises

Binary Operators

For more info, read N & H Section 9.2.2

P-3.1 Make exercise 9.4 from Niño & Hosch. Note that the "identity" of an operation is that special number, also called neutral element, where the operation has no effect. For example, the neutral element of addition is 0, because $x+0=0+x=x$ for every x. Similarly, the neutral element for multiplication is 1. The implementation of `OperatorWithIdentity` is provided on Blackboard. Use `ss.week3.test.AdditionTest` and `ss.week3.test.MultiplicationTest` to validate your implementation.

Passwords and Checkers

In this exercise, we will build on the class `Password` from Exercise P-1.20.

The `Password` implementation you made in Week 1 accepts only passwords longer than 6 characters and it may not contain a space. But we want a flexible check to test if a password meets the requirements (e.g. length and/or containing letters and digits) depending on how we use the class. We will use an interface to make the test for new passwords more flexible. Your solution should be an instance of the *Strategy* pattern discussed in Section 9.6 of Niño & Hosch.

For more info, read N & H Section 9.2.1

P-3.2 Write an interface `ss.week3.pw.Checker` with a single method `boolean acceptable(String)`. An implementation of this method should return `true` if the parameter (according to the implementation's criteria) is an acceptable password, *i.e.*, it cannot be guessed too easily.

P-3.3 Write two implementations of the interface `Checker`:

- A class `BasicChecker` that checks if the string is at least 6 characters long and does not contain any spaces, *i.e.*, the criterion that was also implemented in Exercise P-1.20.
- A class `StrongChecker` that inherits from `BasicChecker` and that checks *in addition* whether the string starts with a letter, and ends with digit.

Hint: use methods `charAt` and `length` from class `String` to find the first and last character in a string, and appropriate methods from the class `java.lang.Character` to test whether a character is a letter or a digit.

As discussed before, also the initial state of an object should respect the object's class invariant. Concretely, in this case it means that the initial password should also be "acceptable". Because we can implement different criteria, we need to find a way to define an initial, acceptable password.

☞ **P-3.4** Extend the interface `Checker` with a method `generatePassword` that returns an acceptable string. Also extend your implementations of `BasicChecker` and `StrongChecker` with appropriate implementations of `generatePassword`. You are allowed to use a constant in your implementations.

For more info, read N & H Section 10.3

The intended client of a `Checker` is a `Password`. We will now adjust the password implementation. First read Section 10.3 of Niño & Hosch.

P-3.5 Extend the implementation of `ss.week2.hotel.Password` with the fields `checker` (of type `Checker`) and `factoryPassword` (of type `String`), with appropriate queries (the intention of the `factoryPassword` field is to initialise the `Password` object with a known value, which can be passed on to a password client, who can then change it).

`Password` should have two constructors:

- The first constructor receives a `Checker` as parameter and sets `checker` and `factoryPassword` to an appropriate value.
- The second constructor has no parameters. It sets the `checker` to be a `BasicChecker`, and it does so by calling the first constructor.

A further way to protect passwords is by making them *expire*, *i.e.*, after a certain amount of time, the password can no longer be used.

To register times, you can use the method `currentTimeMillis` from the class `java.lang.System`. This method indicates the time passed since 1st of January 1970 in milliseconds. By comparing the results of two calls of `currentTimeMillis`, you can see how much time has passed.

For more info, read N & H Section 10.2

P-3.6 Specify and implement a class `TimedPassword` that inherits from `Password`. It should have a field `validTime` that indicates how long a password is valid, and a method `isExpired` that indicates whether the password has expired. The class should have two constructors: one that has the expiration time as an argument, and one that sets the experation time to a default value. Whenever the password is reset, the validity period restarts.

Make sure that when the `TimedPassword` object is constructed, `validTime` immediately should have a sensible value. Use `ss.week3.test.TimedPasswordTest` to validate your implementation.

☞ **P-3.7** What will go wrong if the method `testPassword` in `TimedPassword` is overwritten in such a way that it always returns `false` whenever the password is expired?

Hotel Bill

To start, copy all your hotel classes from `week2.hotel` to `week3.hotel`, make sure you update the packages as well.

In the next exercise, we will extend the hotel system to print a bill for a particular room. For this, it is necessary to assign a price to each hotel room. In addition, a guest should pay for the activation of the safe. However, this is a fixed amount, independent of the number of nights of his stay.

For more info, read N & H Section 10.5.3

First we develop a class `Bill`. To make this as generic as possible, a bill will consist of *items*. For each item there will be a description, and an amount associated to it. It seems natural to use an interface for this. As this interface is specific to the class `Bill`, it will be declared as a nested interface (see the files on Blackboard).

For the formatting of the bill, we will use the static method `format` from class `String`. The first parameter of this method is a string that indicates the expected format, the remaining parameters contain the text that is to be formatted.

For more info, read the webpage

The Java API contains detailed explanation of this method. The method `String.format` internally uses the class `java.util.formatter` to format the output, therefore examples for defining the expected format as well as a detailed description can be found in the API documentation of the class `java.util.Formatter` (see `https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html`).

P-3.8 Develop a class `ss.week3.Format`, containing a static method `printLine(String text, double amount)` that ensures that `text` and `amount` are printed on standard output, and a `main` method that tests your implementation.

Each line should be formatted such that first the text is written and then the amount, whereby the amount is aligned at the decimal dot across all lines, as shown below.

```
text1              1.00
other text       -12.12
something          0.20
```

For the class `Bill`, the intention is that we format the text without printing it directly to standard output (`System.out`). It is useful to separate the formatting from the printing, because for example when we build a test we do not always want all the text to be printed on screen (we might want to write it into a special logfile instead). Therefore, each instance of `Bill` receives a `PrintStream` object when it is constructed, and this `PrintStream` object is used to print to. If we want to print to the standard output (console) we can pass the `PrintStream System.out` to the constructor. In later weeks, we will see examples of other output streams.

P-3.9 Program a class `ss.week3.hotel.Bill` using a nested interface `Item`. In the ZIP provided on blackboard you can find a specification (`Bill.html`). Make sure that your implementation respects the specifications given.

The nested interface `Item` will be implemented by classes that are not nested classes of `Bill` themselves. Therefore, note that instead of declaring them with `implements Item`, you should use `implements Bill.Item`.

Before implementing different items, we will first test the general behaviour of `Bill`.

P-3.10 As we do not have any implementations of `Bill.Item` yet, it is difficult to test the class `Bill`. Therefore you have to make a basic implementation of `Bill.Item`. This constructor should take a `String` text and a `Double` amount. Make sure the method `getAmount` returns the amount and the `toString` method returns the text.

Methods `newItem` and `finish` attempt to write to the `PrintStream`. It is not clear how to test that using our setup. Therefore, in this case, simply set `PrintStream` to null, so there will be no output. (If you wish to test your formatting of the bill, you can always add a method `main` to the class and write to standard output, i.e., pass `System.out` as argument to the constructor.)

☞ **P-3.11** Write a JUNIT test class `ss.week3.test.BillTest` and use this to test your implementation of `Bill`. Use EMMA to check the coverage of your tests.

Now we will ensure that the hotel rooms and the safe indeed cost some money. We do this by inheriting from the existing classes `week3.hotel.Safe` and `week3.hotel.Room` – instead of changing the classes themselves.

P-3.12 Specify and implement a class `ss.week3.hotel.PricedSafe`. It should extend `ss.week3.Safe` and implement `Bill.Item`. The price of the safe is a parameter of the constructor. Remember to implement the method `toString`; this should also show the price of the safe. Use test class `ss.week3.test.PricedSafeTest` to validate your implementation.

P-3.13 Specify and implement a class `PricedRoom` that extends `ss.week3.hotel.Room` and implements `Bill.Item`. The class `PricedRoom` should have a constructor that receives a room number, a room price and a safe price.

In the constructor a new `PricedSafe` should be created and given to the parent(`Room`). Therefore you should add an extra constructor to `Room` which receives and uses that given Safe.

Also override the `toString` method, so that it returns the price per night.

Use test class `ss.week3.test.PricedRoomTest` to validate your implementation.

Finally, it is the responsibility of the hotel to produce the bill. A bill consists of 1 item per night, and if appropriate an item for an activated safe, if the safe is a priced safe.

P-3.14 Write a method `Hotel.getBill` that receives as parameter the name of a guest, the number of nights the guest spent in the hotel, and a `PrintStream` where the bill should be printed. If there is no guest with the given name, or if the guest stays in a free room (*i.e.*, not a `PricedRoom`), the method `getBill` should return the value `null`.

Remark: Of course, the bill should also include the use of the safe.

3.4.3 Recommended exercises

P-3.15 From Niño & Hosch:

- 9.2, 9.3, 9.7, and 9.8.
- 10.4, and 10.5.
- 11.3, 11.4, and 11.5.

P-3.16 Write a testplan and write a test class for your implementations of the `Checker` interface. Use Emma to check the coverage of your tests.

3.4.4 Bonus exercises

P-3.17 In Exercise P-3.4, it was sufficient to define the initial password as a constant. Of course, in a more realistic implementation, this should be generated randomly. You can use the method `Math.random()` for this. For example, the expression `(char)('a' + 26*Math.random())` returns an arbitrary lower case letter, while `(char)('0'+10*Math.random())` returns an arbitrary digit. Using expressions of this kind, you can write a class `week4.pw.Random`, implementing a method `randomString` that returns a random string, consisting of random lower case letters and digits in arbitrary order. Then use this class to implement a class `RandomChecker`. This class receives another checker implementation as parameter, and then initialises the password by generating random strings until an acceptable string has been generated.

P-3.18 Develop a class hierarchy to encode and combine different password criteria. The top of the hierarchy should be an interface `Criterion`, containing a method `acceptable` defining the acceptability criterion. Define your class hierarchy in such a way that you avoid code duplicate for your `acceptable` method as much as possible.

Hint: typically, at a high level in your hierarchy you will have classes such as `AndCriterion`, combining two different criteria, and requiring that both criteria should be respected for the password to be acceptable.

P-3.19 Extend your hotel bill application so that it can contain an item `Nights` that produce a single item on the bill for the total number of nights the guest stayed in a priced room.

Week 4

4.1 Overview

4.1.1 Contents of This Week

Academic Skills This week contains one lecture dedicated to academic skills: a 2-hour meeting on Wednesday. We will make the step from personal planning & execution with a week-perspective, to medium long-term planning (module-planning) and project planning. We will focus on several golden rules from project management and will incorporate some principles from the LEAN methodology to make project work more effective. One of those principles is visual management: a way to keep track of key indicators of your module-planning or project by using a performance board. Keep in mind that even if you find it easy to keep an overview for this particular project, it is a good practice to develop a skillset for later on when project work will become increasingly more complex.

Design The activities in this week cover the following topics:

- *Software Metrics*, see §4.3.1 for the exercises for Lab Session 4.
- Project work, see 4.3.2.
- *Self-study: Example test.* In order to prepare for next week's test it is strongly advised to do the example test on Blackboard. On Monday in Week 5 the answers will be discussed in a plenary session. It will only make sense to be there, however, if you did try the example test.

Programming This week the following topics will be discussed:

- Continuation of inheritance, abstraction, and interfaces
- Lists
- Arrays in Java (building on the knowledge of Week 2 of Module 1)
- Different list implementations

Additionally, there is an optional 1-hour lecture on Thursday about how JML specifications can be validated (during program execution or statically). The content of this lecture is not part of the study material of this module.

4.1.2 Mandatory Presence

During the following activities, your presence is mandatory.

- (M) Session on mathematics in TCS and BIT (Tue 6–9)

4.1.3 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 4 hours self-study to do the example test; and
- 8 hours self-study for the programming thread.

4.1.4 Materials for this Week

Academic Skills Van Tulder B2 and B9.

Design Slides from lecture 4.

Programming

Lecture Niño & Hosch [Ch. 9–11, Ch. 12–14, 21] (Chapter 9–11 is the same as last week; this week different aspects of the same topic will be covered)

Laboratory The following predefined files are provided on Blackboard:

- `ss/week4/Util.java`
- `ss/week4/Exercises.java`
- `ss/week4/MergeSort.java`
- `ss/week4/DoublyLinkedList.java`
- `ss/week4/LinkedList.java`
- `ss/week4/tictactoe/Board.java`
- `ss/week4/tictactoe/Game.java`
- `ss/week4/tictactoe/Mark.java`
- `ss/week4/tictactoe/TicTacToe.java`
- `ss/week4/tictactoe/Player.java`
- `ss/week4/tictactoe/HumanPlayer.java`
- `ss/week4/test/BoardTest.java`
- `ss/week4/test/ConstantTest.java`
- `ss/week4/test/DoublyLinkedListTest.java`
- `ss/week4/test/ExercisesTest.java`
- `ss/week4/test/ExponentTest.java`
- `ss/week4/test/LinearProductTest.java`
- `ss/week4/test/LinkedListTest.java`
- `ss/week4/test/MergeSortTest.java`
- `ss/week4/test/ProductTest.java`
- `ss/week4/test/SumTest.java`

Tools Use of the CHECKSTYLE plugin

4.2 Academic Skills

4.2.1 Assignments

A-4.1 Performance Board As a group, make a performance board for your design project. Use this to keep track of your progress and results as a team on this project for the work you still have to do. Also take into account the progress you have already made and the lessons-learned: what indicators would have helped you? Choose your key indicators wisely, be creative and have fun with designing and using it!

Bring your performance board with you to illustrate your progress and results during the peer feedback session in week 6. Several examples and guidelines will be shared during class. This exercise will be signed off along with presence at the peer feedback session by the peer feedback leader.

4.3 Design

4.3.1 Laboratory session 4 (Software Metrics)

This laboratory session is done in groups of two students.

To prepare for this exercise, create a new ECLIPSE Java project and copy the entire source directory `ss.week3.hotel` (the result of last week's programming lab session) to the `src` directory of that new project (keeping the directory structure intact). Refresh the project in ECLIPSE. It should compile without errors. Make sure the Metrics plugin has been installed and has been enabled for your project (WINDOW → PREFERENCES → METRICS PREFERENCES → ENABLE METRICS), and display the Metrics view (WINDOW → SHOW VIEW → OTHER, then METRICS → METRICS VIEW).

The actual calculations done by the ECLIPSE Metrics plugin are discussed on `http://metrics2.sourceforge.net/`. For some of the metrics, the definition has been taken from *OO Design Quality Metrics, An Analysis of Dependencies.*

D-4.1

1. In WINDOW → PREFERENCES → METRICS PREFERENCES, change the ordering so that VG, WMC, CA, CE and LCOM are shown on top.
2. Check the value of "Method Lines of Code" for a couple of methods in the different classes of this project. Does this value depend on how the code is formatted? Does it depend on the number of lines of comment? (Devise and perform a simple experiment to answer this question!)
3. What is the value calculated by the Metrics plugin for the Afferent and Efferent coupling? Explain this uotcome, using the definition of the metrics on `http://metrics2.sourceforge.net/`.
4. Calculate by hand the value of "Weighted Methods per Class" for the `Hotel` class according to the calculations discussed during the lecture. What is the value calculated by the Metrics plugin?

As already stated in the slides, the "Lack of Cohesion in Methods" (LCOM) has different definitions in the literature. In all cases, high values are to be avoided.

☞ **D-4.2** For the `Guest` class, make a table showing which attributes are accessed by which methods.

1. Calculate by hand the value of "Lack of Cohesion in Methods" according to the calculations discussed during the lecture.
2. Also calculate the value according to the definition of `http://metrics2.sourceforge.net/`.
3. If you change all read references to the attributes into calls of the corresponding `get`-methods, what happens to the value of LCOM? Do you think this is reasonable, given what LCOM is trying to measure?

For the following question, consider the `toString` method of the class `ss.week4.tictactoe.Board` provided on Blackboard for this week. Consider the following method from this class:

```
1     public String toString() {
2         String s = "";
3         for (int i = 0; i < DIM; i++) {
4             String row = "";
5             for (int j = 0; j < DIM; j++) {
6                 row = row + "␣" + getField(i, j).toString() + "␣";
7                 if (j < DIM - 1) {
8                     row = row + "|";
9                 }
10            }
11            s = s + row + DELIM + NUMBERING[i * 2];
12            if (i < DIM - 1) {
13                s = s + "\n" + LINE + DELIM + NUMBERING[i * 2 + 1] + "\n";
14            }
15        }
16        return s;
17    }
```

Again create a fresh ECLIPSE project, and copy only this class to it.

☞ **D-4.3**

1. Draw the flow graph that represents the cyclomatic complexity of this `toString` method of the `Board` class.
2. What should be the result of the McCabe cyclomatic complexity (VG) of this method according to the calculations discussed during the lecture? What is the value calculated by the Metrics plugin?
3. Change the method by removing lines 5–10 from `toString` and putting them into a separate **`private`** method, and instead calling that method from `toString`. The new method needs to get `i` and `row` as parameters. (You can let ECLIPSE do this refactoring for you by selecting the lines in question and then selecting REFACTOR → EXTRACT METHOD.) What is the cyclomatic complexity of the refactored `toString` and of your extracted method?

4.3.2 Project

D-P.4 Follow your planning!

4.4 Programming

4.4.1 Laboratory exercises

Arrays

We will first do some exercises to become acquainted with arrays in Java. Notice that arrays in Java are more low-level than arrays in Python. The most important difference is that they have a fixed size, and they cannot be dynamically extended. If you need more space to store your data, you need to allocate a new array with a larger size, and copy all the information from the original array to the new array. Java provides an efficient copy method for this purpose: `java.util.ArrayCopy(src, dest, srcIndex, destIndex, length)`, which is implemented natively. To see how this is used, look for example at the implementation of the method `add` in the `ArrayList` implementation of `java.util.Collection`.

For more info, read N & H Section 13.1

P-4.1 Make exercises 13.1 and 13.2 from Niño & Hosch. You can use the boilerplate code in `Exercises` for your implementations, and `ExercisesTest` to validate them.

Checkstyle

On Blackboard, the class `Util` is provided.

☞ **P-4.2** The class `Util` violates the configured checkstyle coding conventions in several ways. Modify the class implementation such that the meaning of the class implementation does not change, but that the coding conventions are satisfied.

Mathematical Functions

In this exercise, we will use abstraction to provide support for symbolic mathematical functions. The operations that will be supported are discussed during the mathematics line of this module.

P-4.3 Develop an interface `Function` with methods providing support for the following functionality:

- `apply` executes the function to an `argument` of type double.
- `derivative` returns the `Function` object that is the derivative of the current object.
- `toString` returns a nice string representation of the function.

In the following series of exercises, you are asked to implement for different mathematical functions the `Function` interface. For your convenience, the definitions below summarise the rules to compute the derivatives and integrands of the functions used in this exercise.

Constant $f(x) = c$
$f'(x) = 0$
$F(x) = c \cdot x$

Exponent $f(x) = f(x) = x^n$
$f'(x) = n \cdot x^{n-1}$
$F(x) = \frac{1}{n+1} \cdot x^{n+1}$

Sum $f(x) = g(x) + h(x)$
$f'(x) = g'(x) + h'(x)$
$F(x) = G(x) + H(x)$

Product $f(x) = g(x) \cdot h(x)$
$f'(x) = g'(x) \cdot h(x) + h'(x) \cdot g(x)$
no standard rule available for integrand

Linear Product $f(x) = n \cdot g(x)$
$f'(x) = n \cdot g'(x)$
$F(x) = n \cdot G(x)$

Polynomial $f(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \cdots + a_1 \cdot x + a_0$
$f'(x) = a_n \cdot n \cdot x^{n-1} + a_{n-1} \cdot (n-1) \cdot x^{n-2} + \cdots + a_1$
$F(x) = a_n \cdot \frac{x^{n+1}}{n+1} + a_{n-1} \cdot \frac{x^n}{n} + \cdots + a_1 \cdot \frac{x^2}{2} + a_0 \cdot x$

For all classes below, test classes are provided to validate your implementations of `apply` and `derivative`.

P-4.4 Develop a class `Constant`, implementing the interface `Function` with a constant function. The value of the constant should be passed as an argument to the constructor. The `apply` method should return the value. The `derivative` should return a new `Function` with the derivative of a constant (a new Constant with value zero). And the `toString` method should return the constant value.

For more info, read N & H Section 10.4.2

P-4.5 Develop a class `Sum`, which defines the sum of two `Functions`. The `Functions` should be passed as arguments to the constructor.

Note: the derivative should be a `new Sum()` of the derivatives of `Functions` passed to the constructor.

P-4.6 Develop a class `Product`, which defines the product of two `Functions`. The `Functions` should be passed as arguments to the constructor.

P-4.7 Develop a class `LinearProduct` extending `Product`, which defines the product of a `Function` with a `Constant` function. Reuse as much as possible from the implementation of `Product`, but try to make your implementation as optimal as possible.

P-4.8 Develop a class `Exponent`, implementing the interface `Function`, which resembles x^n . Where the exponent n should be passed as an `Integer` to the constructor.

☞ **P-4.9** Develop a class `Polynomial`, implementing the interface `Function`, to represent arbitrary polynomial functions. The parts should be given to the constructor as an array from `doubles`.

Hint: A possible way to implement this is to initialise an array of `LinearProducts` (`Constant`, `Exponent`) in the constructor. Note that to calculate the derivative, you only have to return the sum of the derivative of all elements in the array.

P-4.10 Develop an interface `Integrandable` with an method `integrand`. This method should return the integrand of a function.

P-4.11 Why should it be necessary to create a separate interface for `integrand`, instead of adding it to the `Function` interface?

P-4.12 Modify the `Function` classes, so they implement `Integrandable` if possible. Implement the `integrand` methods.

P-4.13 Make a JUNIT `Test` class for `Polynomial`, similar to tests for the other `Functions`. Test also for the `integrand` method.

P-4.14 Make a class `Homework` with the following functionality: Make a main method where you print and test several combinations of the created functions. You should print the functions and their outcome. e.g.:

```
LinearProduct f1 = new LinearProduct(new Constant(4),new Exponent(4));
Function f2 = f1.integrand();
Function f3 = f1.derivative();
System.out.println("f(x) = " + f1 + ", f(8) = " + f1.apply(8));
System.out.println("f(x) = " + f2 + ", f(8) = " + f2.apply(8));
System.out.println("f(x) = " + f3 + ", f(8) = " + f3.apply(8));
```

Which has the following outcome:

```
f(x) = (4.0) * (x^4), f(5) =  16384.0
f(x) = (4.0) * ((0.2) * (x^5)), f(5) =  26214.4
f(x) = (4.0) * ((4.0) * (x^3)), f(5) =  8192.0
```

☞ **P-4.15** Draw the class diagram of the `Function` hierarchy.

Sorting

During Pearl 001 (week 2) of Module 1, you developed Python implementations of various sorting algorithms, such as bubble sort, and merge sort. See for example `http://y2u.be/EeQ8pwjQxTM` for an explanation of merge sort.

For more info, read N & H Section 12.3, 14.1 and the movie

P-4.16 Reimplement the merge sort algorithm from Pearl 001 in Java for `Lists` of `Elem`, assuming that `Elem extends Comparable<Elem>`. You can use the boilerplate code in `MergeSort` for your implementation, and `MergeSortTest` to validate it.

Linked Lists

Next, we will study typical linked list implementations.

For more info, read N & H Section 21.3

P-4.17 Make exercise 21.4 of Niño & Hosch. You can use the boilerplate code in `DoublyLinkedList` for your implementation, and `DoublyLinkedListTest` to validate it.

☞ **P-4.18** Make exercise 21.6 of Niño & Hosch. Create a function `findBefore` in `LinkedList` and give it the behaviour as described for the first version of the `find` method. After that make exercise 21.7 of Niño & Hosch. Create a method `remove` in `LinkedList` that uses the method `findBefore`. You can use the boilerplate code in `LinkedList` for your implementation, and `LinkedListTest` to validate it.

For more info, read N & H Section 21.1

Tic-Tac-Toe

In this exercise we will develop an implementation of the game Tic-Tac-Toe (in Dutch: boter kaas en eieren). We follow the structure of the Nim game as described in Niño & Hosch (Chapter 8); it is useful to read through this chapter first.

Tic-tac-toe is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a 3x3 grid. The player who succeeds in placing three respective marks in a horizontal, vertical, or diagonal line wins the game, see `http://en.wikipedia.org/wiki/Tic-tac-toe`.

In our version of tic-tac-toe game, the board is represented by a one-dimensional array of `3*3=9` `Mark`-objects[1]. The relation between the array and the board is as follows:

```
 0 | 1 | 2
---+---+---
 3 | 4 | 5
---+---+---
 6 | 7 | 8
```

[1] It is also possible to represent the board with an `3*3` matrix van `Marks`.

An empty field is represented by a `Mark.Empty`, a cross can be written as `Mark.XX` and a circle by `Mark.OO`. The class `Mark` has a method `toString` to create a one-character `String` of a `Mark`-object.

The class `Board` has several game related methods, like `hasRow`, `hasWinner`, etc.

P-4.19 Implement the missing methods of the class `Board`. Make sure your implementation is valid according to their pre- and postconditions. Remember to use constants instead of "hard-coded" values. You can use `LinkedListTest` to validate your implementations.

The class `week4.Game` controls the interaction between the `Players` and the `Board`. It also shows a textual representation of the board in the console. The class `tictactoe/Game` available on Blackboard is almost finished; only the method `play` needs to be implemented.

P-4.20 Implement the method `play` of `Game` according to its specification.

To play the game we need an executable class (i.e., a class with a `main` method). We will call this class `TicTacToe`. The main method only has to create a new `Game`-object and call the method `start`.

To determine the names of the players we will use the possibility to give command-line parameters when executing a program. The `main` method always looks as follows:

```
public static void main(String[] args)
```

The argument of type `String[]` is filled with the arguments provided when starting the program.

If we start `TicTacToe` as follow:

```
java week4.TicTacToe mies wim
```

then the array of strings `args` will have the value [`mies`,`wim`], i.e., the array has a length of two (`args.length==2`), `args[0]` has the value `"mies"`, and `args[1]` has the value `"wim"`.

You can refer back to Exercise P-2.12 from Week 2 for an explanation how to specify command-line arguments when running a class using Eclipse. Instead of specifying VM Arguments, enter your options in the text box Program arguments.

☞ **P-4.21** Write the class `TicTacToe` and implement the main method. When starting the game the names of the players should be given as an argument, and these arguments should be used to create the `Player` objects. Test the system by playing some games. This class will be further elaborated next week.

4.4.2 Recommended exercises

P-4.22 From Niño & Hosch:

- 13.3, 13.4, 13.8, 13.9

4.5 Mathematics: Euclides, Leibniz, and Newton in Computer Science

The goal of the mathematics session this week is to show you the link between the mathematics of this and the previous module and TCS and BIT. Participation in this session is mandatory.

4.5.1 Mathematics for BIT Students

BIT students will work on a case concerning a resourcing problem of a company. The problem is how much resources of various kinds (materials, personnel, etc.) must be purchased for the next period, in order to maximize the company's profit. In this context, also dynamic prizes and market share will be considered. Optimization techniques of Math B2 will be indispensable for analyzing this problem.

4.5.2 Mathematics for TCS Students

During the 6th and 7th hour, there will be 'speed dates' with several researchers. These researchers are from different departments: Computer Science, Electrical Engineering, and Mathematics, but their research is always related to Computer Science.

You will be divided in groups consisting of 3 project groups, i.e., of approximately 12 students. These groups will be discussing with the participating researchers to find out how he or she uses the mathematics of these two modules for research. There will be 3 rounds with speed dates:

- Round 1: 13:45 - 14:15
- Round 2: 14:20 - 14:50
- Round 3: 14:55 - 15:25

Purpose of the discussion is that you find out the following:

- what mathematics does the researcher use?
- how is the mathematics used?
- can you find applications of this mathematics in every day life?
- what are the computer science applications of this mathematics?
- is the mathematics mainly applied, or is new theory being developed?
- what mathematical knowledge are you still missing related to this research?

We will provide you with a list of mathematical topics that are the subject of the first and second module.

☞ **M-1.1** In the room, there will be sheets of papers with the mathematical topics. There will also be small memo notes available. After each discussion, stick a note with the name of a researcher and a short description how the research relates to the mathematics topic on the appropriate piece(s) of paper.

☞ **M-1.2** After the speed dates, you have 45 minutes (the 8th hour) to reflect on what you learned. Concretely, you are asked to do the following: with your own project group, prepare a 4-minute presentation to inform the other students about your speed dates. During your presentation, you should answer the following questions:

- Of which mathematical applications were you aware already before the speed dates?
- Of which mathematical applications did you learn today?
- Which applications did you appreciate most, and why?
- How does this affect your view on the mathematics of this module?

Feel free to add other points that you feel should be addressed. There is **no** need to prepare any slides. We expect all group members to contribute to the presentation.

During the last hour of the afternoon (the 9th hour), the groups give their presentations. For reasons of time, we will divide into three separate groups.

Week 5

5.1 Overview

5.1.1 Contents of This Week

Academic Skills This week contains no academic skills activities.

Design The design activities in this week cover the following topics:

- *Question and Answer session.* The answers to the example exam will be presented. Questions about the example exam and any other design matters will be answered.
- *Test* (Wed 1–3), see 5.2.1.
- Project work, see 5.2.2.

Programming This week the following topics will be discussed:

- Recursive data structures.
- Maps.
- Generics.
- Complexity of implementations.
- Security engineering: hash functions, Java security properties, side-channel attacks, and using security libraries.

Recursion has been introduced in Module 1 (Pearl 011, week 4); this week we will see how recursion is used in Java.

5.1.2 Deadline

The deadline for design project submission is **Sun 23:59**, see 5.2.2.

5.1.3 Mandatory Presence

During the following activities, your presence is mandatory.

- (M) Self-study supervised (Tue 6 – 7)
- (D) Design test (Thu 1–3)
- (M) Tutorial (Thu 6 – 7)

5.1.4 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 4 hours self-study for the mathematics thread;
- 3 hours self-study for final preparations for the design test;
- 4 hours design project work beyond the sessions with assistance;
- 2 hours self-study for the programming thread; and

5.1.5 Materials for this Week

Academic Skills No reading material this week.

Programming

Lecture Niño & Hosch [Chapters 19, 20, and 22.] Chapter 5 of "Security Engineering", sections 5.3–5.3.1.2 and 5.6–5.6.2.

Laboratory The following predefined files are provided on Blackboard:

- `ss/week5/HumanPlayer.java`
- `ss/week5/MapUtil.java`
- `ss/week5/EncodingTest.java`
- `ss/week5/test/CompatibleTest.java`
- `ss/week5/test/ComposedTest.java`
- `ss/week5/test/InverseBijectionTest.java`
- `ss/week5/test/InverseTest.java`
- `ss/week5/test/IsOneOnOneTest.java`
- `ss/week5/test/IsSurjectiveOnRangeTest.java`

5.2 Design

5.2.1 Test

The test has the same type of questions as the lab exercises (and the recommended exercises). Unlike the lab session, you have to give your solutions on paper. 85% of the test will be about UML diagrams, 15% of the test will be about software metrics.

For the UML part, you will be asked to draw a selection of the following diagrams for a given case description:

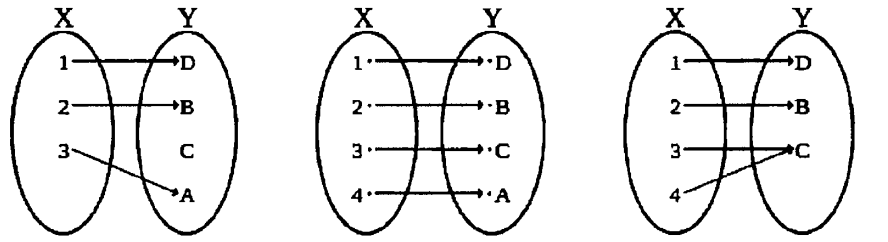
- activity diagram,
- use case diagram (possibly including small sections of a list of requirements and use cases, a list of actors, a glossary, a list of use case descriptions, and/or extended use case descriptions—you will not be asked to make long lists that involve a lot of writing)
- class diagram,
- sequence diagram,
- state machine diagram.

A use case diagram and a class diagram are *always* included in the test.
From the activity diagram, sequence diagram and state machine diagram *at least two* will be included. Usually one these three is dropped. Which one depends on the case study. It is possible that all three diagram types are included, in which case the diagrams will be smaller than usual.

The test is an open book test. See the section *Tests and Grades* (p. 5) in the Introduction for what materials you can bring to the test.

How to prepare for the test

For the UML part of the test, all you need to know is covered in the lecture slides. Note, however, that no factual knowledge is asked. The best way to prepare for the test is by doing the recommended exercises and



Injective but not surjective | Injective and surjective | Surjective but not injective

Figure 5.1: Examples of injectivity and surjectivity

the example tests. You should look at the solutions *after* you gave it a serious try—and consult the slides if you have difficulties understanding the solutions.

For the Software Metrics part of the test, you will be expected to understand what the metrics intend to achieve. The test will probably ask you to compute one metric or the other—cyclometric complexity being a prime example. This is in fact the only one you are expected to know by heart; and in the process, you are expected to understand how to construct the flow graph of a method, at least to the degree where you can count the decision points. For other metrics, if you are asked to compute them, the definition will be included.

5.2.2 Project

D-P.5 Finalize the design project. Consult the sections *What to hand in* and *Submission and grading* (p. 11) in the project descriptions. The project is due **Sun 23:59**.

5.3 Programming

5.3.1 Laboratory exercises

Function Properties

For more info, read the documentation

The interface `java.util.Map<K, V>` can be used to model a mathematical function. For each *key*, a `Map` returns the corresponding *value*. The map's `keySet()` corresponds to the domain of the function, i.e., for which the function is defined. For example, given a function $f : \mathbb{N} \to \mathbb{N} : x \mapsto x+1$, we can model this as a map `mapF`, and if 1 is in the `keySet()` of `f`, then `mapF.get(1)` should return 2. For more information, see `https://docs.oracle.com/javase/8/docs/api/java/util/Map.html`.

In this exercise we develop a class `MapUtils` that defines several auxiliary functions to define commonly used properties and definitions of mathematical functions:

- a function $f : X \to Y$ is *injective* or *one-on-one* if (see Figure 5.1):

$$\forall x_1 x_2 \in X . x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$$

- a function f is *surjective* if it maps to all values in its range (see Figure 5.1):

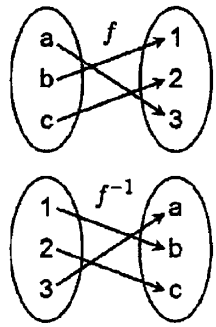
$$\forall y \in Y . \exists x . f(x) = y$$

- the *inverse* of a function $f : X \to Y$ is the function $f^{-1} : Y \to X$, such that (see Figure 5.2):

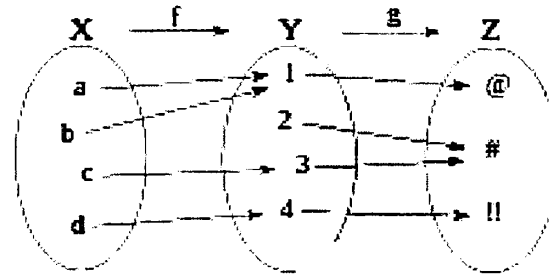
$$\forall x \in X . f^{-1}(f(x)) = x$$

- the *composition* of two functions $f : X \to Y$ and $g : Y \to Z$ is a function $h : X \to Z$ such that (see Figure 5.2):

$$\forall x \in X . h(x) = g(f(x))$$



A function and its inverse



Function Composition

Figure 5.2: Examples of inverse and function composition

P-5.1 Implement a static method `isOneOnOne` that checks whether a `java.util.Map<K,V> f` passed as a parameter is an injection. This is, `isOneOnOne` returns true if for all `v` in the value set of the map `f`, there exists exactly one key `k` in the map's key set, such that `v == f.get(k)`.

Write a JML method contract for this method, which specifies what is returned by this method.

Not all mathematical properties can be implemented directly. To check whether a method is surjective, *i.e.*, for all values in the range of the function, there is a value in the domain that maps to it, we have to pass the intended range as an explicit argument.

P-5.2 Implement a static method `isSurjectiveOnRange` that checks whether parameter map `f` is surjective. Add an parameter `java.util.Set<V>` range, and check for all elements in `range` there is a key such that `f` maps to this element.

Give the method an appropriate JML method contract, which specifies what is returned by this method.

Next we will define the inverse of a function. We make two variations, one that defines the inverse of an arbitrary function, and one that defines the inverse of a bijection, *i.e.*, of a function that is one-on-one and surjective.

P-5.3 Implement a static method `inverse` that given a map of type `Map<K,V>` returns a map of type `Map<V, Set<K>>`. Explain why you need the result type to be a `Map<V, Set<K>>` instead of `Map<V,K>` in this case.

Additionally, implement a static method `inverseBijection` that returns a map of type `Map<V, K>` if the function is injective and surjective.

Give appropriate JML method contracts for your methods, which specify what is returned by the methods.

As a last step, we will define two methods to compose two functions.

☞ **P-5.4** Implement a static method `compatible` that checks whether the two maps passed as a parameter can be composed, *i.e.*, whether all values in the value set of the first map are in the key set of the second map.

Next, implement a static method `compose` that defines the composition of two maps, provided they are compatible.

Both methods should have appropriate JML method contracts, which specify what is returned by the methods.

Strategy

For more info, read N & H Section 9.6

We will extend last week's Tic-Tac-Toe implementation with support for a computer player. To do this, we need to change the functionality of the `Player`, using the `Strategy` pattern (see §9.6 of Niño & Hosch).

P-5.5 Specify an interface `ss.week5.Strategy` to determine the next move for the tic-tac-toe game. The interface should have the following two methods:

- `public String getName()` that returns the name of the strategy;
- `public int determineMove(Board b, Mark m)` that returns a next legal move, given the `Board b`, for the player with `Mark m`.

P-5.6 First we develop a naive strategy where `determineMove` returns an arbitrary empty field on the board.

Specify and implement the class `ss.week5.NaiveStrategy`, implementing the interface `Strategy`. The strategy name is `''Naive''`.

As said, the method `determineMove` chooses an arbitrary empty field. Therefore, you should first construct a collection of empty fields (using interface `java.util.Set`). Afterwards, you can use the method `Math.random` to select one of these empty fields.

P-5.7 Specify and implement a class `ss.week5.ComputerPlayer` that extends the class `ss.week5.Player` from the tic-tac-toe game. A `ComputerPlayer` always has a strategy.

The name of the `ComputerPlayer` is the name of the strategy, followed by a hyphen (`''-''`), followed by a representation of the player's mark.

The class `ComputerPlayer` should have two constructors:

- `public ComputerPlayer (Mark mark, Strategy strategy)` that constructs a computer player using the given mark and strategy; and
- `public ComputerPlayer (Mark mark)` that constructs a computer player using the given mark and a naive strategy.

The method `determineMove` from `ComputerPlayer` of course should use the computer player's strategy. Additionally, it should provide functionality to inspect and update the strategy.

P-5.8 Update the class `week5.TicTacToe` such that naive computer player can play the game. If the name "`-N`" is given as an argument, this means this player is a ComputerPlayer with a naive strategy. For example, if you start the program with the arguments wim -N, as in

```
java week8.TicTacToe wim -N
```

a game will be created with `wim` (as `X`-humanplay) against `naive-computer-O`.

Test the program by playing some games against the naive computer player.

Of course, with the naive computer player's strategy, the computer is not very likely to win. Therefore, we will develop a smarter strategy, thinking one move ahead.

P-5.9 Specify and implement a class `ss.week5.SmartStrategy`, implementing the `ss.week5.Strategy` interface. The method `determineMove` returns an empty field using the following recipe:

- If the middle field is empty, this field is selected;
- If there is a field that guarantees a direct win, this field is selected.
- If there is a field with which the opponent could win, this field is selected.
- If none of the cases above applies, a random field is selected.

Hint It might be useful to make a copy of the board for the implementation of `determineMove`.

The name of this strategy is `''Smart''`.

☞ **P-5.10** Update the class `week5.TicTacToe` in such a way that also the smart computer player can play the game. If the name "`-S`" is given as an argument, this means this player is an ComputerPlayer using a smart strategy.

Test the program by playing some games against the smart computer player. Also play some games where the naive computer player plays against the smart computer player.

Encoding

When dealing with security-related subjects such as cryptographic hashes, MAC's, etc., one often needs to handle binary data (i.e., raw bytes). To be able to represent such binary data correctly in plain, normal (UTF-8) text you can use encoding methods[1] such as Hex[2] and Base64[3]. There is no need to implement these encoding schemes yourself; simply use existing libraries for that. In preparation for the assignments in week 6, the next few assignments will show you how to use one of such libraries: the Apache Commons Codec library (see `https://commons.apache.org/proper/commons-codec/`). This boils down to simply using the static methods of the classes `org.apache.commons.codec.binary.Hex` and `org.apache.commons.codec.binary.Base64`.

P-5.11 To get you started, you are given `EncodingTest.java`. Using the Apache Commons Codec library, it prints out the hexadecimal encoding of the ASCII string "Hello World". It does this by first getting the raw bytes representation of the string using the `getBytes` method of the `String` class. This is then provided to the `encodeHexString` method of the `org.apache.commons.codec.binary.Hex` class. Download the jar-file of the Apache Commons Codec library and use it in your Eclipse project for `EncodingTest.java`. The provided code should print out `48656c6c6f20576f726c64` as the hexadecimal representation of the ASCII string "Hello World". Now change the input string to "Hello Big World", how does the hexadecimal output change?

P-5.12 Of course the same library also has support for converting a hexadecimal representation back to bytes. Add a few lines of code to `EncodingTest.java` such that it converts the hex string `4d6f64756c652032` to bytes (a byte array) and then prints the result when these bytes are turned into a String. See the documentation for the Apache Commons Codec library which method to use. Use `new String(bytearray)` to create a string from a byte array. Additional hint: the String class has a method `toCharArray` to convert a string to an array of characters.

☞ **P-5.13** As stated above, Base64 is another way to represent binary data in plain (ASCII) text (see Wikipedia's page on Base64 for more information). It is for example often for email attachments. In this exercise you will play a bit with Base64 by adding a few more lines of code to `EncodingTest.java`.

- First, based on your experience with the `Hex` class, encode the string "Hello World" in Base64.
- Next, take the hex string `010203040506`, decode it to a byte array, encode this byte array with Base64, and print it. What is the output? What is the length of the Base64 representation? Can you see an advantage of Base64 over Hex encoding?
- Then, decode the Base64 string `U29mdHdhcmUgU3lzdGVtcw==` and print it.
- Finally, produce the Base64 encoding for each of the following strings: "a", "aa", "aaa", "aaaa", "aaaaa", ..., "aaaaaaaaaa". What do you notice?

5.3.2 Recommended exercises

P-5.14 From Niño & Hosch make the following exercises:

- 12.2, 12.3, 12.4, 12.5, 12.11, 12.12, and 12.13
- 14.8, 14.9.
- 19.3 and 19.6
- 22.4 (the method `concatenate` for two `List<Element>` objects), 22.6 (the method `isPalindrome` for a `List<Element>` object).

[1] Note: this is not the same as encryption! Encodings such as Hex and Base64 are easily reversible without any key material.
[2] See `http://en.wikipedia.org/wiki/Hexadecimal`
[3] See `http://en.wikipedia.org/wiki/Base64`

5.3.3 Bonus exercises

P-5.15 In Exercise P-5.9 you have developed a smart strategy for tic-tac-toe. However, this strategy is not always satisfactory, as it thinks only one move ahead. However, it is possible to implement a *perfect strategy*, using a recursive algorithm. If there is a possibility to win the game, this strategy will lead to victory. If there is a possibility for a draw, the strategy will never lead to a loss. First we define some terminology, before describing the algorithm:

- A move is *winning* (for the player whose turn it is) if after the move one of the following situations applies:
  - The player has won;
  - If the opponent can still make a move, then all possible moves of the opponent will make the opponent *loose*.
- A move is *losing* (for the player whose turn it is) if after the move the opponent can make a move such that the opponent *wins*.
- In all other cases, the move is *neutral*.

Using this terminology, we describe an algorithm that given a game situation and the knowledge whose turn it it, returns the best move for this player, together with an estimate of the *quality* of this move (winning, losing, or neutral):

- The best move so far, and its quality is stored using auxiliary variables, for example `bestMove` and `bestQual`;
- For every possible move, the following happens:
  1. The player whose turn it is, does the move (as a tryout)
  2. Next the quality `qual` of the move will be determined
     (a) If the player whose turn it is, now has won, the `qual` will be *winning*.
     (b) If the opponent now has not won, the best move of the *opponent* will be determined by a recursive call to the algorithm. Call the result of this call `oppQual`.
     - If `oppQual` is "winning", then `qual` will be "losing".
     - `oppQual` is "losing", then `qual` will be "winning".
     - Otherwise, `qual` will be "neutral".
  3. If `qual` is better than the current value of `bestQual`, then `bestQual` will be replaced by `qual`, and the value of `bestMove` will be replaced by the current move.
  4. Finally, the tried move will be removed from the list of possible moves, so the remaining ones can be tried.
- The result of the algorithm is the values of `bestMove` and `bestQual`.

The algorithm can be made more *non-deterministic* by making a random choice in step 3 whether to replace the `bestMove` by another move of the same quality, or not.

Figure 5.3 gives an overview of all possible moves of the X-player that can lead to a 'perfect' strategy. In the figure, the X-player first chooses the left upper corner, and the steps can be followed by (recursively) zooming in on the position chosen by the O-player.

Specify and implement a class `ss.week5.PerfectStrategy`, implementing the `ss.week5.Strategy` interface. The method `determineMove` should use the algorithm described above.

The name of this strategy is "`Perfect`".

Would this algorithm also work for a game of chess? Why, or why not?

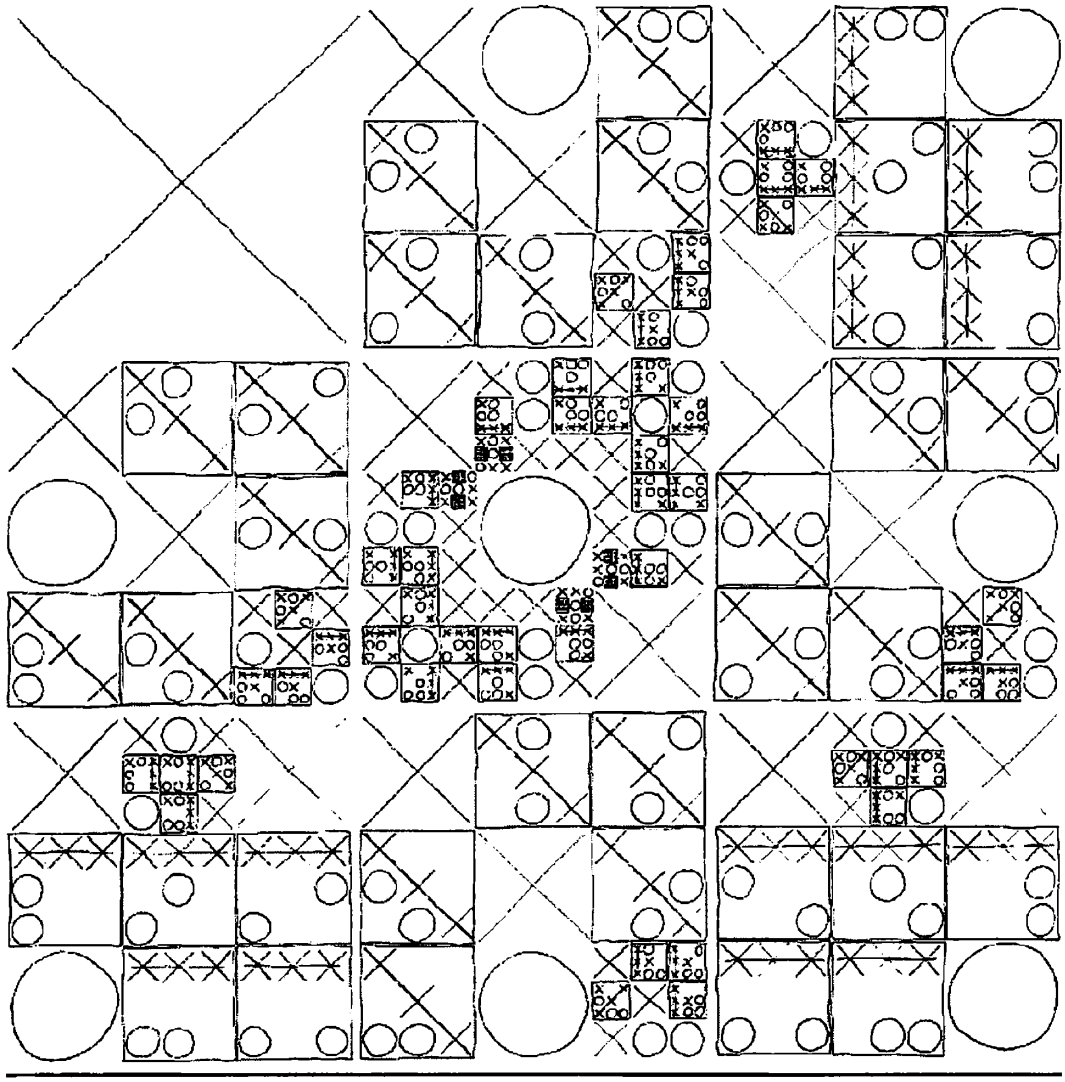


Figure 5.3: Possible perfect moves for player X.

Week 6

6.1 Overview

6.1.1 Contents of This Week

Academic Skills This week contains one peer feedback session dedicated to academic skills on Wednesday. You will have a 30-minutes peer feedback session with your project group.

Design This week contains no session on design.

Programming This week the following topics will be discussed:

- Input/output: reading from standard input, and from a file, and writing to standard output, and to a file.
- The model-view-controller (MVC) pattern.
- Exceptions.
- A diagnostic test where you can judge for yourself how well you have understood the concepts related to inheritance and overriding.
- Kick-off for the programming project.

Additionally, there is an optional 2-hour lecture on Wednesday about Graphical User Interfaces (technical and user aspects). The contents of this lecture is not part of the study material of this module; your participation is optional.

6.1.2 Mandatory Presence

During the following activities, your presence is mandatory.

- (M) Self-study supervised (Tue 6 – 7)
- (A) Peer feedback session on project planning (Wed 6–7)
- (P) Diagnostic test *Inheritance and Abstraction* (Wed 8)
- (M) Tutorial (Thu 6 – 7)
- (P) Project meeting on overall design of the programming project (Thu 6 –7)

6.1.3 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 4 hours self-study for the mathematics thread;
- 2 hours self-study for the programming project; and
- 7 hours self-study for the programming thread.

6.1.4 Materials for this Week

Academic Skills Please take note of the important do's and don'ts on giving and receiving feedback that are described in Van Tulder G9.

Programming

Lecture Niño & Hosch, Chapter 15, 16, 18. For Chapter 18, you can ignore all GUI-related remarks, and focus on the MVC pattern.

Laboratory The following predefined files are provided on Blackboard:

- `ss/week6/Zipper.java`
- `ss/week6/cards/Card.java`
- `ss/week6/cards/CardReader.java`
- `ss/week6/challenge/ttt/Board.java`
- `ss/week6/challenge/ttt/Game.java`
- `ss/week6/challenge/ttt/Mark.java`
- `ss/week6/voteMachine/gui/ResultJFrame.java`
- `ss/week6/voteMachine/gui/VoteFrame.java voteMachine/gui/VoteGUIView.java`
- `ss/week6/voteMachine/VoteView.java`
- `ss/week6/dictionaryattack/DictionaryAttack.java`
- `ss/week6/test/ArgumentExceptionTest.java`
- `ss/week6/test/CardTest.java`
- `ss/week6/test/DictionaryAttackTest.java`
- `ss/week6/test/LeakedPasswords.txt`
- `ss/week6/test/VoteMachineModelTest.java`
- `ss/week6/test/ZipperTest.java`

6.2 Academic Skills

6.2.1 Peer feedback session

In the groups formed for the Design project, you are asked to reflect on your planning and group work of this last project. In this session your peer feedback leader will facilitate a discussion about the ways planning and group work went well and about which things –with hindsight– could be improved. The performance board which you have kept since week 4 serves as an aid in this discussion. You should be able to show this performance board to explain and reflect on your progress and the results for this project.

6.3 Programming

6.3.1 Laboratory exercises

Textual User Interface

We will first develop a basic textual user interface. The Java API `java.util.Scanner` will be used to interpret the data input. Look at the specification of this class, and in particular of the following methods:

- The constructor `Scanner(InputStream)` that creates a scanner from an "input source". A commonly used input stream in Java is `System.in`, which denotes the standard input.

- The constructor `Scanner(String)` that creates a scanner from an arbitrary `String` object.
- The methods `hasNext` and `hasNextLine` that test if there is a next word or a next line, respectively, that can be read by the scanner. If the scanner is constructed for standard input, reading might have to wait until new input has been provided (ended by a carriage return, *i.e.*, pressing `Enter`).
- The methods `next` and `nextLine` that read the next word (*i.e.*, a series of connected symbols without a space), or the next line, respectively. These methods will return an error message if they are called when there is no next element to be read, therefore they should always be preceded by a call to `hasNext` or `hasNextLine`, respectively.

For more info, read N & H Section 7.2

P-6.1 Write, using the class `java.util.Scanner`, a class `ss.week6.Hello` that does the following:

- Ask for a name on standard input;
- Print the text `''Hello ''` followed by the name that has just been provided on standard input.

This should be repeated until there is no more input, or until the provided name is the empty string.

`Hint` You should import `java.util.Scanner` and start your program with the following declaration.

```
Scanner in = new Scanner(System.in);
```

The variable `in` now is the only one that should be used for input.

P-6.2 Write, using the class `java.util.Scanner`, a class `ss.week6.Words` that does the following:

- On standard input it asks for a sentence, consisting of words separated by spaces.
- It prints the words one by one on standard output.
- tt should stop if the entered sentence starts with the word `''end''`.

The program should also work if the sentence is empty, *i.e.*, has no words. Here is an example execution:

```
Line (or "end"): The quick brown fox jumps over the lazy dog.
Word 1: The
Word 2: quick
Word 3: brown
Word 4: fox
Word 5: jumps
Word 6: over
Word 7: the
Word 8: lazy
Word 9: dog.
Line (or "end"): end
End of programme.
```

Hint To break the sentence into separate words, you should also use a `Scanner`.

Exceptions

On the Blackboard site, you can find the class `Zipper` which implements functionality to "zip" two Strings into one: i.e., to create a new String by concatenating the characters alternating between both Strings.

The method `zip` has two preconditions and the `main` method tests the provided command line arguments to see whether they fulfill these preconditions. If this is not the case, an error message is printed; the `zip` method is only called when the preconditions are met.

☞ **P-6.3** Define three classes:

For more info, read N & H Section 15.2 - 15.3

- `ss.week6.WrongArgumentException` which extends `Exception`,
- `ss.week6.TooFewArgumentsException` which extends `ss.week6.WrongArgumentException`, and
- `ss.week6.ArgumentLengthsDifferException` which extends `ss.week6.WrongArgumentException`.

Define a new method `zip2` which performs the same functionality as `zip`, but in addition checks whether the provided arguments satisfy the preconditions. If the first precondition is violated, a `TooFewArgumentsException` must be thrown, if the second one is violated, an `ArgumentLengthsDifferException` must be thrown. The message of a thrown exception (which can be queried from an exception object using `getMessage()`) must correspond to the error message that is printed by the provided `main` method.

Now also change the `main` method such that:

- The new method `zip2` is called.
- The `main` method does not check the preconditions itself anymore.
- The functionality performed by the `main` method does not change.

Use `ss.week6.test.ArgumentExceptionTest` and `ss.week6.test.ZipperTest` to test your implementations.

Note: You may get a JML warning about not being allowed to use `String.length()` in a contract, because it is not `pure`. Ignore the warning for now.

A Card Game

On the Blackboard site, you can find an implementation of a `Card` game. We will add input and output methods to the class `Card`. As input and output often is a source of failures, your method implementations should catch these whenever necessary, and throw appropriate exceptions.

For more info, read N & H Section 16.3 - 16.4

Java supports various means of communication: via (readable) text, primitive data, and objects, respectively. In this exercise, we will practice with all three ways of communication.

Text channels Text channels are implemented by the predefined classes `java.io.PrintWriter` and `java.io.BufferedReader`. Study the documentation of these classes and pay special attention to the different constructors and the *flush* method.

P-6.4 Add a method `write` to the class `Card`. It should have a `PrintWriter` as parameter, and the result of the method should be that the object on which the method is called sends a description of itself (obtained by the use of `toString`) to the `PrintWriter`.

Test your method by adding a method `main` to `Card` where you create a `PrintWriter` instance on the basis of a file, which name is given as argument in the call of the program, for example:

```
java ss.week6.card.Card cardfile.txt
```

Let `main` create several `Card` instances and write these to file. Check that the file is indeed created, and contains readable text.

The *standard output* (`System.out`) is a `PrintStream` and thus it may also be used to create a `PrintWriter` object. Adapt your `main` method so that if no file name is given as argument, standard output will be used.

Now that we can write cards to a file, the next logical step is to construct cards based on data stored in a file. It seems a good idea to make the input format the same as the output format. The output format, described by `Card.toString` consists of two words on line: suit and rank, separated by a space.

In contrast to writing to `PrintWriter`, when reading from a `BufferedReader` you have to take exceptions into account.

P-6.5 Extend `Card` with the following method:

```
public static Card read(BufferedReader in) throws EOFException
```

The method should read from `BufferedReader in` and returns a `Card` instance on the basis of this input. Make sure that:

- `read` returns `null` if the `BufferedReader` does not allow you to construct a valid card, for example because the line that was read does not correspond to the format *"suit rank"*.
- `read` returns a `EOFException` when the `BufferedReader` is finished.

Remember that you can use the `Scanner` to split the card information in information about the suit and the rank.

☞ **P-6.6** Use `ss.week6.test.CardTest` and `ss.week6.card.CardReader` to test your method `read` from Exercise P-6.5. Check if this method `read` and `write` from Exercise P-6.4 match each other.

The provided class contains code to invoke read/write method that will be developed in the coming assignments. You can comment out the erroneous lines until the respective read/write methods are implemented.

Data Channels A second form of communication is by using a *data stream*. This supports a direct storage of primitive data values and strings (where the values are stored as a series of bytes). This kind of communication is supported by the interfaces `java.io.DataInput` and `java.io.DataOutput`, and in particular the implementations `java.io.DataInputStream` and `java.io.DataOutputStream`.

P-6.7 Implement methods `read(DataInput in)` and `write(DataOutput out)` in class `Card`. Make sure that they match: anything that is written by `write` should be readable by `read`. If anything goes wrong while writing, `write` should throw a `IOException`, while `read` should treat exceptions in the same way as `read(BufferedReader in)` in Exercise P-6.5.

Notice that a `Card` is uniquely represented by its rank and suit. Both can be represented by a `char`. Therefore, `Card` should be stored as two `char` values and not as `String`.

Use `ss.week6.test.CardTest` and `CardReader` to test your implementations.

Object Channels A third form of communication is on the basis of *object streams*, where complete objects can be written and read. This functionality is supported by the interfaces `java.io.ObjectInput` and `java.io.ObjectOutput`, implemented by `java.io.ObjectInputStream` and `java.io.ObjectOutputStream`. However, not all objects can be handled by object streams, since the objects class should be *serializable*. For more information, see the documentation of `java.io.Serializable`.

P-6.8 Extend the class `Card` with methods `write(ObjectOutput)` and `read(ObjectInput)`, similarly to the data-based methods in Exercises P-6.5 and P-6.4.

As always, it should be possible to write a `Card` object and read it back in again.

☞ **P-6.9** Which form of communication is the least costly: text-based, data-based, or object-based? Explain your answer.

Building a Voting Machine

For more info, read N & H Section 18.1 - 18.2

The Dutch government has asked you to develop a voting machine for the next parliament elections. You will use the "Model-View-Controller" principle to program this.

You can assume the usual way of voting in the Netherlands: the voter selects a candidate (called "party" from now on) and confirms his choice[1]. The machine will work through a Textual User Interface (TUI).

The voting machine should have a model, view and controller. The model consists of al the data that should be saved. The view is the layer the user sees, and the controller combines the view and the model.

For more info, read N & H Section 18.2.1

P-6.10 Design and implement the classes `PartyList` and `VoteList` in package `ss.week6.voteMachine`. Together, they form the *model* for the voting machine. The `PartyList` class should store all the parties in the system and have a method for adding a party and retrieving all parties in a `List`. The `VoteList` class should have methods for making a vote and retrieving all votes in a `Map<String,Integer>`. Use `ss.week6.test.VoteMachineModelTest` to test your implementation.

For more info, read N & H Section 18.2.5

P-6.11 The next step is to build the *controller*, which combines the model (from the previous exercise) with the view (next exercise). Add the following functionality:

- Add a static main method which creates a new VoteMachine and calls the method `start`. The method start can be empty at this moment.

[1] During this lab exercise, we will only consider votes for a party, not for individual candidates.

- Add attributes for `PartyList` and `VoteList` and initialize them in the constructor.
- Create the following methods and implement them so they update the models: `addParty(String party)` and `vote(String party)`

P-6.12 Now you have to build the *view*. Create the class `ss.week6.voteMachine.VoteTUIView` with the following functionality:

- A method `start`, which starts a loop and asks the user for input in the console;
- The `start` function should be able to identify the following user commands (you implement them later): VOTE [party], ADD PARTY [party], VOTES, PARTIES, EXIT, and HELP;
- A method `showVotes(Map<String,Integer>)` that can be called by the controller to show information about the votes;
- A method `showParties(List<String>)` that can be called by the controller to show information about the parties; and
- A `showError(String)` that can be called by the controller to show information about the errors that occurred.

P-6.13 Next we will connect the view and the controller.

- Add an attribute for the controller in `VoteTUIView`, and initialize it in the constructor, by passing a reference to the `VoteMachine` to the constructor of `VoteTUIView`.
- Add the following commands: `getParties()` and `getVotes()`, they should retrieve the information from the models and pass it to the view.
- Call the `start` method of the view in the `start` method in the constructor, to start the view when we start the vote machine.

☞ **P-6.14** Update the `VoteTUIView` such that it calls the corresponding methods in the controller when a commando is entered. Now your `VoteMachine` should work.

At this moment we do not obtain any feedback when a party is added to the list. One way to solve this is to print a message in the view when the command is executed. But what if we also support other ways to add parties to the list? Of what if we can vote also through a network? We will solve this by notifying the view when the model is updated. This can be done by using the Observer pattern, see Section 18.2 of Niño & Hosch.

For more info, read N & H Section 18.2.2 - 18.2.3

P-6.15 Implement the Observer pattern:

- Let `PartyList` and `VoteList` extend `Observable` and call `setChanged` and `notifyObservers` when you add a party or a vote. Add as a parameter from `notifyObservers` a string named "party" or "vote", to specify what changed.
- Let `VoteTUIView` implement `Observer` and add the `update` method to `VoteTUIView`. This method should show a message when a vote or party is added. Use the second parameter to check if a party or vote was changed.
- Update the controller `VoteMachine` such that the view is added as an observer to the party- and vote-list, use the `addObserver` method.

P-6.16 What is the advantage of using the MVC (Model, View, Controller) pattern? Explain this in terms of reusability.

☞ **P-6.17** On Blackboard, a package `week6.voteMachine.gui` is provided, implementing the interface `week6.voteMachine.VoteView`. Change your `VoteTUIView` such that it also implements the `VoteView` interface. Update `VoteMachine` such that its view attribute is of type `VoteView` instead of `VoteTUIView`. Your application should still work.

If you did this correctly, you can change the code that constructs the view to:

```
this.voteView = new VoteGUIView(this);
```

This gives you a GUI for your vote machine.

Password dictionary attack

Suppose now that the fictious company BigSimpleCorp has a simple online service that authenticates its users through passwords. However, somehow, the password file for their online service was leaked to pastebin (`http://pastebin.com/h0etcvvS`). Although the passwords are not stored as plain text, the only protection is the fact that the passwords appear to be hashed. A snippet of the password file:

```
alice: c0af77cf8294ff93a5cdb2963ca9f038
allen: c6009f08fc5fc6385f1ea1f5840e179f
```

As stated above, the passwords are stored using a cryptographic hash function. Cryptographic hash functions are one-way: it is easy to compute the hash for a certain input, but obtaining the original input given only a hash is (by design) very (very) hard. The reason for using it for storing passwords is straightforward: it is easy to check whether a password is correct (by hashing a password and comparing it with the hash that is stored), but in case an unauthorized person gets access to the list then the passwords are not as easy to obtain. However, dictionary attacks are still possible: given the hashes of a large number of common words (e.g., from a dictionary), one can check whether the hashes in a password file match any of the hashes of words in the dictionary. In the following few exercises you will build such a dictionary attack step-by-step. In the process you will use some of the container structure from the `java.util.Collection` hierarchy (in particular the Map). You are provided with a skeleton file for this assignment called `DictionaryAttack.java` that will act as a starting point.

P-6.18 Fill the body of the `readPasswords` method. The JavaDoc for this method describes what it should do. Hint: keep parsing of the password file simple. You could for example simply use the `String.split()` for each line (with which argument?) to obtain the username and password hash.

From the length of the string that represents the hashed password we can make a guess as for which hash algorithm is used. The hex-encoded string is 32 characters long, which means it represents 16 bytes ($16 \times 8 = 128$ bits). This matches the output of the MD5 algorithm, which is 128 bits[2].

P-6.19 Implement the `getPasswordHash` method. This method should take a password (a String), compute the MD5 hash of it, convert the resulting byte-array to a Hex-encoded string and return this. The JavaDoc for the method has more information. Java provides support for cryptographic hashes in the class `java.security.MessageDigest`. See Oracle's documentation on `MessageDigest` at `http://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html`. Check your implementation with `ss.week6.test.DictionaryAttackTest`, the hexidecimal encoding of the MD5 hash of the password "password" is `5f4dcc3b5aa765d61d8327deb882cf99`. Based on visual inspection of the password file, which users appear to have chosen "password" as their password? Note that the `MessageDigest` methods can throw exceptions. Decide for yourself how to handle those.

P-6.20 Implement the `checkPassword` method. See the JavaDoc for details and use the `getPasswordHash` you have implemented. Check your implementation with `ss.week6.test.DictionaryAttackTest`.

P-6.21 Next, compute a dictionary of password hashes by implementing the `addToHashDictionary` method. As described in the JavaDoc, it should read a file and compute the (MD5) hash of each line (use your `getPasswordHash`) and fill a dictionary (a `Map`) that can map a password hash back to the original password. Search on the internet for "most common passwords" and you will find for example a list of the 25 most common passwords. Use this list to populate a small initial dictionary to test with.

P-6.22 With the building blocks in place, now implement the dictionary attack in the `doDictionaryAttack` method. It should use the two `Map` class variables for this. Whenever a password is found, print out both the username and the password. With the dictionary file you created based on the most common passwords, you should be able to find several passwords already. To find even more you can use a larger word list. For example you could use the worldlist at `http://www.cs.duke.edu/~ola/ap/linuxwords` to make your dictionary larger. With it, you will be able to find even more passwords.

[2] See also `http://en.wikipedia.org/wiki/MD5`. Note that nowadays using MD5 for anything serious is a typically Bad Idea™. Using it to hash passwords is an even worse idea, see the rest of the text surrounding these exercises.

For passwords that are not in a dictionary, the next step is broaden the search. One approach is try all kinds of combinations of words, possibly with some extra letters or digits mixed in. Alternatively one simply try all possible passwords: the brute-force attack.

☞ **P-6.23** Someone tells you that the user Alice has used a simple four letter lowercase word as her password, how many tries would it take on average to find her password when using a brute-force approach?

Although the following assignment is optional, do read through it and the text that follows it.

P-6.24 (Optional) Write a program that finds Alice's four letter password by trying all possible combinations (even though you may have found the password already using your dictionary attack). Measure how much time it takes on average per attempt (use `System.currentTimeMillis()` for this). What other passwords can you find? Try increasing the password length, how does that change the running-time? How many passwords are you able to recover?

The above (optional) exercise illustrates how relatively easy it is to recover passwords from such a password file. Suppose it takes on average 0.000315 milliseconds to try one password. Extrapolating this to 8-letter lowercase passwords (that is, 104413532288 possible passwords) means that even with a naive implementation such a password could be recovered within 9 hours.

Based on this it should be clear that the approach used at BigSimpleCorp is not the way to go. Even more so considering GPUs and dedicated hardware are capable of brute-forcing such MD5 hashes orders of magnitudes faster.[3] Instead it is better to use a salt[4] and to use hashing functions specifically designed for the purpose of storing passwords. Such functions are designed such that they are fast enough for normal usage but too slow to effectively brute force. Examples of such hashing functions are: PBKDF2, bcrypt, and scrypt.

6.3.2 Project

Goal of the programming project is to build a board game application. See the project description in the beginning of this manual for the detailed project requirements. On the Blackboard site, you can find which game you will have to implement this year. On Thursday, there is a kick-off lecture for the programming project, your presence is mandatory.

You will develop your application in pairs.

Project Session

The purpose of this project meeting is to design together the requirements and architecture for the system. Together with an student assistant you will be discussing the game and its overall design. Before the project session, make sure that you have read the project description (in this manual) and rules of the game (on Blackboard).

Every pair of students working on the project is expected to take notes themselves, to keep track of the results of this discussion.

Concerning the game logic, in Exercise P-4.19 and further, you have developed a game logic for tic-tac-toe. The game logic controls the implementation of the board, providing for example:

- The board representation
- A check which moves are allowed
- A function to determine the winner of the game
- Functionality to translate and make moves on the board, i.e., to execute the turns in the game.

Your experiences with this exercise can serve as a starting point for your design. Typical aspects that can be discussed are the following:

- Which game rules are important for the application?
- How to implement the game logic?

[3] For example, 180 billion MD5 hashes per second, see http://www.zdnet.com/25-gpus-devour-password-hashes-at-up-to-348-billion-per-second-7000008368/.

[4] See wikipedia: https://en.wikipedia.org/wiki/Salt_(cryptography)

- What would be an appropriate representation for your board, e.g., a two-dimensional matrix, or an array.
- Which methods would be needed for playing the game.
- What would be a typical run-trough of the game, i.e., what happens between starting the application and a game-over?
- What elements are needed in the UI and how can you express them in a TUI?
- What would be a good way to represent the board. For example, would it be better to use a matrix or just a simple array?
- How should a user enter his/her moves?
- Considering the MVC pattern, which functionality belongs to the model, the view and the controller?
- What are the most important classes that you need, and what should be their functionality?

In week 7 there will be a second session where the group will decide on the communication protocol to be used between the client and the server.

6.3.3 Recommended exercises

P-6.25 Use class `Scanner` to write a textual user interface `ss.week6.ui.HotelTUI` for the class `Hotel` (from week 3). The idea is that a user can use your program to do the hotel administration using the keyboard, *i.e.*, it should be possible to check in or out a guest, and (de)activate a safe. Opening and closing of a safe is something that should be done by the guest himself, and this does not have to be supported by the interface.

An example execution would be the following.

```
Hotelbooking system Hotel Drienerburght
Command's:
 i password name ..... checkin guest with name
 o name ............... checkout guest with name
 r name ............... request room of guest
 a password name ..... activate safe
 h .................... help (this menu)
 p .................... print state
 x .................... exit
Command: i default Major
Guest Major gets room 101
Command: a default Major
Safe in room 101 is now active
Command: o Major
Command: r Major
Guest Major doesn't have a room
Command: x
```

Hint Write the following methods and use them to provide the desired functionality:

- `String readLine(String prompt)` that prints the text `prompt` on standard output and returns the line from standard input (or `null` if there is no more input);
- `Scanner readCommand()` that calls `readLine` until a nonempty line is returned and then returns this line, or returns `null` if there is no more input.

6.3.4 Bonus exercises

A GUI for Tic Tac Toe

P-6.26 Last week we developed strategies for a computer player of the Tic Tac Toe game. These exercises will help you to develop a simple *Swing* application to play the game. We will use the *Model-View-Controller* pattern again.

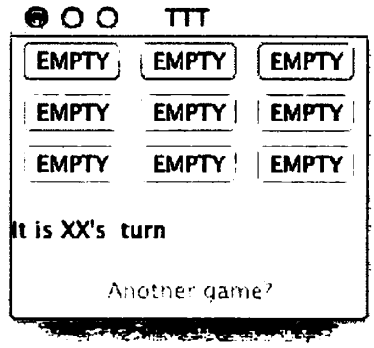


Figure 6.1: A simple GUI for Tic Tac Toe.

On Blackboard you can find the classes `Mark`, `Board` and `Game` in package `ss.week6.challenge.ttt`. They are based on the implementation form Week 5, but slightly simplified, as we do not need the `Player` class for this assignment. Instead, this application will only allow two human players to play the game on a computer.

- First we will build the GUI of the application. Figure 6.1 shows a screen shot of a basic Tic Tac Toe application, based on `JFrame`. The `JFrame` has nine `JButtons` (on a `JPanel`) that represent the board. The `JLabel` provides information about the game's state (whose turn is it, who has won, or if it is a draw). The 10th `JButton` only can be selected if the game has finished.
  Write a class `ss.week6.challenge.ttt.TTTView` that only shows the board of a Tic Tac Toe game. All Swing components should be defined as instance variables of `TTTView`.
  You do not have to make a connection with class `Game` yet. It also is not necessary yet to add an `ActionListener` to the components of `TTTView`.
- Make `Game` an `Observable`.
- Turn `TTTView` into an `Observer`.
  Add the `Observer` method to `TTTView`. This method should always reflect the state of the game within the `TTTView`. This means the following:
  - The nine `JButtons` represent the board. Therefore, they should be labelled with the `Marks` of the board. When the game is finished (there is a winner, or the board is full), no `JButton` should be enabled. If the game is not finished, only the empty buttons should be enabled.
  - Depending on the state of the game, the `JLabel` can have five difference `String` values: "X's turn", "O's turn", "X has won", "O has won", or "Draw".
  - Only when the game is finished, the `JButton` with the text "Play again?" may be selectable.

  Make sure that in the method `main` of `TTTView` a `Game` object and a `TTTView` are created. The `TTTView` object should be added as an `Observer` to the `Game` object.
  Notice: the `TTTView` object should not have a field of the `Game` object, only inside the method `update` it should be necessary to inspect information about the model's state.
- Finally add a *Controller* so that one can actually change the state of the game. Following Niño & Hosch, we use an *inner class* for this.
  Add an inner class `TTTController` inside `TTTView`. The class `TTTController` is an `ActionListener`. It only has a constructor and a method `actionPerformed`.
  In the constructor, a reference to the *Model*, *i.e.*, the `Game` object is defined. Additionally, the constructor adds a `ActionListener` to each `JButton` of `TTTView`.
  The method `actionPerformed` checks which `JButton` generated the event and then invokes the appropriate method on `Game`.

The `TTTController` object should be constructed in the constructor of `TTTView`. This means that now the constructor of `TTTView` should receive a `Game` object to pass it on to the constructor of `TTTController`.

Week 7

7.1 Overview

7.1.1 Contents of This Week

Academic Skills This week contains no session on academic skills.

Design This week contains no session on design.

Programming This week the following topics will be discussed:

- Principles of thread programming:
  - Thread creation and thread life cycle
  - Synchronisation
  - Wait-notify mechanism
- Network programming:
  - Use of sockets for communication over a network.
  - The common framework of a client/server network application.

Moreover, you have to make and discuss a planning for the programming project. On Wednesday an optional lecture will be given about interesting features of Java 8.

7.1.2 Mandatory Presence

During the following activities, your presence is mandatory.

- (M) Self-study supervised (Tue 6 – 7)
- (P) Discussing project planning with student assistant during lab session (Wed 1–4)
- (M) Question and Answer session (Thu 6 – 7)
- (P) A project session on the group communication protocol (Wed 8–9)
- (P) Diagnostic test *Exam Preparation* (Thu 6 – 7)
- (M) Mathematics test (Fri 6–7)

7.1.3 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 6 hours self-study for the mathematics thread;
- 2 hour self-study for the programming project; and
- 5 hours self-study for the programming thread.

7.1.4 Materials for this Week

Programming

Lecture Core Java I, Chapter 14, until BlockingQueues (p. 819–877), and Core Java II, Chapter 3 until Making Url connections (p. 185 - 210).

Laboratory The following predefined files are provided on Blackboard:

- `ss/week7/IntCell.java`
- `ss/week7/ConcatThread.java`
- `ss/week7/QuickSort.java`
- `ss/week7/bounce/Ball.java`
- `ss/week7/bounce/BallPanel.java`
- `ss/week7/bounce/Bounce.java`
- `ss/week7/mandel/MandelPanel.java`
- `ss/week7/mandel/MandelSet.java`
- `ss/week7/mandel/MandelThread.java`
- `ss/week7/threads/Console.java`
- `ss/week7/threads/IntProducer.java`
- `ss/week7/threads/IntConsumer.java`
- `ss/week7/threads/IntCell.java`
- `ss/week7/threads/UnsynchronizedIntCell.java`
- `ss/week7/threads/ProdCons.java`
- `ss/week7/account/Account.java`
- `ss/week7/cmdline/Client.java`
- `ss/week7/cmdline/Peer.java`
- `ss/week7/cmdline/Server.java`
- `ss/week7/challenge/BadCookieCrypto.java`
- `ss/week7/challenge/chatbox/ServerGUI.java`
- `ss/week7/challenge/chatbox/MessageUI.java`
- `ss/week7/challenge/chatbox/Client.java`
- `ss/week7/challenge/chatbox/ClientHandler.java`
- `ss/week7/challenge/chatbox/Server.java`
- `ss/week7/recipeserver/RecipeServer.java`
- `ss/week7/recipeserver/RecipeClient.java`
- `ss/week7/recipeserver/ClientHandler.java`
- `ss/week7/cmdchat/Client.java`
- `ss/week7/cmdchat/ClientHandler.java`
- `ss/week7/cmdchat/Server.java`
- `ss/week7/test/QuickSortTest.java`

7.2 Programming

7.2.1 Laboratory exercises

Project planning

☞ **P-7.1 Group planning for programming project** Make a planning for the programming project. You can, but are not required to, use the format explained during the Academic Skills lecture on planning.

To successfully complete the project, you should at least plan during which periods you are going to work on *designing*, *implementing*, *documenting code*, *testing*, and *writing the report*. Moreover, for these tasks, and in particular for the implementation and report writing tasks you should think of subtasks and plan these as well.

You should show your planning to a student-assistant and incorporate any feedback you get, as he or she might give some suggestions how to adjust your planning, if your planning is unrealistic. Remember that the final goal of the project is to create a working program and a decent report. The student assistant has also done this during his/her first year, therefore it is important to take his feedback into account.

You should have discussed your planning with the student-assistant no later than Wednesday this week.

Threads and GUIs

Sometimes we would like a Java application to do multiple things "simultaneously". For example, we would like to have a clock on the screen, play music, load a file in an editor, and react on user input. This can be achieved by using multiple threads. Even without explicitly creating and starting threads, multiple threads can be active within a Java application. For example, the garbage collector always runs in a separate thread, parallel to the main thread of the application.

Likewise, when a JAVA program uses a graphical user interface (GUI), there always is a separate thread for event handling. This is called the *event dispatch thread*). It is responsible for

- Noticing user actions, e.g., mouse clicks,
- Rendering the GUI windows and everything thereon.

If the event thread is occupied executing other pieces of code, it cannot take care of these core responsibilities and the program can become unresponsive.

The Mandelbrot Set In this exercise, we will work on a multithreaded application for drawing fractals. The application `ss.week7.mandel.MandelSet` is available from the Blackboard site. This exercise is based on Example 8.4.4 of *Martin Kalin, Object-oriented Programming in Java, Prentice Hall, 2001.* Notice that while drawing the fractal, the program is still able to react on selection in the menu.

P-7.2 If you select MENU → DRAW several times in quick succession, you can actually notice that the drawing takes place right through the menu. After analyzing the code, can you explain this effect?

P-7.3 Change in the class `MandelSet` the call to `MandelPanel.draw` to a call of `MandelPanel.drawMandel`. What is the difference when you execute the program now? Explain the difference. After doing the exercise, please undo this change.

In the given implementation, the `MandelPanel` uses a special class `MandelThread`, which extends `Thread` to create and start a parallel process. This is not always the most convenient approach.

☞ **P-7.4** Change `MandelPanel` to implement the `Runnable` interface, and adapt the creation of the `Thread` so that class `MandelThread` becomes superfluous, while the functionality of `MandelSet` is unchanged.

For more info, read CJ Section 14.1

Bouncing Balls The package `ss.week7.bounce` on Blackboard contains three classes:

- `Ball`, modelling a bouncing ball. Notice that `Ball` has a field of type `javax.swing.JPanel`, initialised in the constructor. Furthermore, `Ball` has a method `draw` that draws the ball, a method `move` that changes the position of a ball, and a method `collide` that changes the movement of two bouncing balls.
- `BallPanel`, an extension of `JPanel`, on which a number of `Ball` instances can be bouncing. `BallPanel` maintains a list of balls. Among others, it has a method `addBall` to add a bouncing ball to this list, and a method `animate` to start the movement.
- `Bounce`, the frame class of the application, with a button for adding a ball.

P-7.5 In the given implementation, the program does not draw anything at all, not even one ball. Why not?

To solve this problem, a separate `Thread` should be used, comparable to the `MandelThread` above.

P-7.6 Define an inner class `AnimateThread` within `BallPanel` that extends `Thread`. The method `run` should only call `animate`. Make sure that the constructor of `BallPanel` creates and starts an `AnimateThread`. Remove the call of `animate` from `Bounce`.

Another possibility to solve the problem of Exercise P-7.5 is to use a `javax.swing.Timer` instead of a `Thread`. The functionality of a `Timer` is to trigger the event dispatch thread to perform a certain task at regular intervals. In this particular case, that task is to animate the next frame of the bouncing balls. This is a conceptually much better solution than the `AnimateThread` because it is now the event dispatch thread taking care of rendering, and not some parallel thread that might interfere with it (like by overwriting the menu as we saw in Exercise P-7.2).

The way to program a `Timer` is to wrap the task to be triggered into an `ActionListener` that is passed into the constructor of the `Timer`, together with the period at which the timer should trigger it (in milliseconds, for instance 5).

☞ **P-7.7** Change `BallPanel` into an `ActionListener` and implement the method `actionPerformed` to draw a single animation frame — corresponding to the `moveBalls` and `repaint` calls in `animate`. Create a `Timer` instance in `BallPanel` and start this.

Synchronisation

For more info, read CJ Section 14.5

In the classes and methods seen so far, we did not bother about the functioning of a method when methods on a single object are executed via several threads simultaneously. However, it is important that classes and methods also should function correctly under these circumstances, *i.e.*, that they are *thread safe*. We first study the behaviour of the static read and print methods of the class `ss.week7.threads.Console`.

P-7.8 Write a class `ss.week7.threads.TestConsole`, extending class `Thread`. The method `run` should call a (private) method `sum` that uses `Console.readInt` to read two numbers from input, and writes their sum on output using `Console.println`. For example:

```
Thread A: get number 1? 13
Thread A: get number 2? -15
Thread A: 13 + -15 = -2
```

In the example, `Thread A` is the name of a thread. This can be assigned by passing it as a parameter to the constructor, and it can retrieved using `getName()`. Let `main` create and start **two** instances of the class.

☞ **P-7.9** Study the behaviour of `TestConsole` and try to understand the behaviour of the program, based on the program code. Why is this behaviour problematic?

For more info, read CJ Section 14.5.5

P-7.10 Develop a class `ss.week7.thread.SyncConsole` by copying `threads.Console` and making the methods `synchronized`. Copy also `TestConsole` to `TestSyncConsole` and change the references to `threads.Console` to `SyncConsole`.

P-7.11 The problem identified in Exercise P-7.9 is still not completely solved. Why not?

P-7.12 Change the method `sum` of `TestSyncConsole` to also be `synchronized`. Is there a difference with the previous exercise. If yes, explain why. If not, why not?

☞ **P-7.13** Adapt `TestSyncConsole` so that the computations of the two parallel processes no longer overlap.

For more info, read CJ Section 14.5.3

The `java.concurrency.util.locks` library declares an interface `Lock` with methods `lock()` and `unlock()` for synchronisation. A synchronized block can be replaced by declaring a `Lock`, preceding the block by a call to `lock`, and finishing the block by a call to `unlock`.

P-7.14 Study the `Lock` interface from `java.concurrency.util`. The most commonly used implementation of this class is `ReentrantLock`. Answer the following questions:

1. What does it mean for a lock to be reentrant?
2. Is this behaviour different from the `synchronized` statement?
3. What would be advantages of using a `ReentrantLock`?
4. And what would be disadvantages?

☞ **P-7.15** Reimplement `TestSyncConsole` using a `Lock`.

Producer-Consumer

For a good collaboration of parallel processes, synchronisation alone is sometimes not sufficient. In many cases, processes should wait until it is their *turn* or there are other restrictions on the order in which certain methods may be called.

As an example, we look at a typical *Producer-Consumer pattern*. Producers and consumers are executing in parallel, and they are communicating via a buffer in which there can be exactly 1 number. The producers ask the user for input values, the consumers retrieve the value from the buffer and print this on the screen. The purpose is that each value that the producer put in the buffer, is read and printed once by one consumer, in the same order as they are put in the buffer.

> *Note, that in this assignment it is not necessary to synchronise the standard input and output, as reading and writing of the values is not shown on the screen directly.*

On Blackboard you can find classes `ss.week7.threads.IntProducer` and `IntConsumer` (the parallel processes), `IntCell` (the buffer interface), `UnsynchronizedIntCell` (a preliminary implementation of `IntCell`, and `ProdCons` (the main application class).

P-7.16 Compile and execute the Java files. Execute several times. What ways can you find that the given implementation does *not* satisfy the specification? How can this be solved?

For more info, read CJ Section 14.5.5

P-7.17 Replace implementation `UnsynchronizedIntCell` of `IntCell` by a new implementation to solve the problems signalled in the previous exercise. Use this implementation in `ProdCons` instead of `UnsynchronizedIntCell`. Use the methods `wait`, `notify` or `notifyAll` (inherited from `Object`). What is better, to use `notify` or to use `notifyAll`? Why?

Hint: maintain a Boolean instance variable that indicates whether there is an unconsumed value in the buffer.

In your implementation, there are many *spurious* notifies. For example, if there is a notification to signal that the buffer contains a value to be read, it also wakes up threads that are waiting to write a value to the buffer. Therefore, it can be better to have a more fine-grained notification mechanism, and to wake up only the threads that are waiting for the condition that has just been achieved.

The `java.concurrent.util.locks` library declares a `Condition` interface for this purpose. Each implementation of the `Lock` interface can be associated with multiple conditions. Instead of calling `wait` on an object, a thread can now `await` on a condition. When this particular condition is reached, the call to `signal` on this `Condition` will awake only the threads waiting on this condition.

☞ **P-7.18** Study the documentation of interface `Condition` and implement a class `FinegrainedIntCell` implementing `IntCell` that uses a `Lock` and multiple `Conditions` instead of the `synchronized` keyword and methods `wait` and `notify(All)`. The documentation of interface `Condition` gives a producer consumer example to illustrate the usage of `Condition`.

For more info, read CJ Section 14.5.4

Concurrency Theory

In the following exercises we will investigate several smaller concurrent applications in detail to understand the different ways threads can interact with each other.

P-7.19 Consider the class `Account`.

```
package ss.week7.account;

public class Account {
    protected double balance = 0.0;

    public void transaction(double amount) {
        balance = balance + amount;
    }
    public double getBalance() {
        return balance;
```

```
    }
}
```

This class is not *thread safe*.

1. Write a class `MyThread` extending `Thread` with the following constructor:

```
public MyThread(double amount, int frequency, Account account) {
    this.theAmount = amount;
    this.theFrequency = frequency;
    this.theAccount = account;
}
```

   The constructor's paramaters indicate how many `times` a given `amount` should be added to the `account`. The `amount` may be negative (in the case, it is removed from the `account`) The only thing the thread should do is in a loop execute the transactions `times` times.
2. Write a program `AccountSync` that creates two threads that execute on the same `account`. One thread should increases the value stored in the `account` with a fixed `amount`, while the other threads removes `amount`. Both threads should do this the same number of `times`. After both threads are finished, the resulting value on the `account` should be printed on the screen.
3. Execute your program. You would want the result to be always `0.0`. However, when you run your program several times, you will see that this is not always the case. Explain why not.
4. Adapt the class `Account` to be thread safe.
5. Adapt the class `Account` in such a way that the `value` on the account will never be less than `-1000`. If a thread would like to decrease `value` further; it has to wait until the operation is allowed.
6. Indicate how the program `AccountSync` should be changed if `MyThread` would not be a subclass of `Thread`, but an implementation of `Runnable`. In that case, should the body of `MyThread` also be changed?

☞ **P-7.20** Consider the following Java program.

```
package ss.week7;

public class IntCell {
    private int contents = 0;

    public void add(int amount) {
        contents = contents + amount;
    }
    public int get() {
        return contents;
    }

    public static void main(String[] args) {
        IntCell cell = new IntCell();
        Adder a1 = new Adder(cell, 1);
        Adder a2 = new Adder(cell, 2);
        a1.start();
        a2.start();
        try {
            a1.join();
            a2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(cell.get());
    }
}

class Adder extends Thread {
```

```
    private IntCell cell;
    private int amount;

    public Adder(IntCell cellArg, int amountArg) {
        this.cell = cellArg;
        this.amount = amountArg;
    }
    public void run() {
        cell.add(amount);
    }
}
```

1. What are the possible results of this program? Motivate your answer.
2. If the calls to `a1.start()` and `a2.start()` in the method `main` are replaced by `a1.run()` and `a2.run()`, respectively, what would be the possible results of the program? What is the difference between `start()` and `run()`? Motivate your answer.
3. If in the original program, the two lines from the `try-catch` block in the method `main` (i.e., the statements `a1.join();` and `a2.join();`) are removed, what would be the possible results of the program? Motivate your answer.
4. If in the original program, the methods `add` and `get` of `IntCell` are declared `synchronized`, what would be the possible results of the program? Motivate your answer.
5. Change the body of the method `run` of class `Adder` in the original program by using `synchronized` and/or calls of the methods `wait`, `notify`, and `notifyAll`, in such a way that the program always terminates, and always prints the value 3[1].

Communication via the Command-line

Socket connection Before starting with the lab exercises, we would like to first remind you of the standard steps to create a socket connection between a server and a client.

For more info, read CJ Section 3.2

- *For the server:*
  1. Open a `ServerSocket`
  2. Accept a *client* of the `ServerSocket`
  3. Open an *inputstream* and an *outputstream* for the client `Socket`
  4. Read and write data according to a protocol (that has been agreed upon beforehand)
  5. Close both streams, the client `Socket` and the `ServerSocket`

For more info, read CJ Section 3.1

- *For the client:*
  1. Open a `Socket`
  2. Open an *inputstream* and an *outputstream* of the `Socket`
  3. Read and write data according to a protocol (that has been agreed upon beforehand)
  4. Close the streams and the `Socket`

In this exercise, we will develop an application that allows two users (a client and a server) to communicate with each other via two "Command Prompt" (or `Terminal`) windows. The connection between the client and the server will be implemented by a socket.

The user of the client can type messages on his `Terminal`. Via the `Socket` connection those messages will then be sent to the server. The user of the server receives the messages on his own `Terminal`. Similarly, the server can type messages on *his* `Terminal`, that will appear on the `Terminal` of the client. Notice that the protocol between the client and the server in this case is symmetric; apart from setting up the connection, both the client and the server provide the same functionality.

Notice further that the client/server can receive messages from its user (via the `Terminal` and from its *peer* simultaneously. Therefore, the application should also be multithreaded.

[1] Remark that the solution to this exercise enforces a particular execution order. In a way, this makes parallel execution sequential again. In these simple examples this is unrealistic, however in larger applications such techniques are sometimes necessary, to ensure that two execution steps are done in a particular order.

As mentioned, the client and the server only differ in the start up phase. A server application should be started first. When starting the server, the user should provide his name, and the *port* where the server should 'wait' for the client. To start the server, open the Eclipse launch configuration dialog and create a launch configuration to run the class `ss.week7.cmdline.Server` and specify as Program Arguments, e.g., `Alice 2727`.

For more info, read CJ Section 3.1.2

After the server application has been started, also the client application can be started. When the client is started, the user should provide his name, the hostname of the server (an IP address, or `''localhost''` if both applications run on the same machine) and the port number where the server listens to the client. Again, Program Arguments must be provided via the Eclipse launch configuration, e.g., `Bob localhost 2727`

The method `main` of the class `Client` could look as follows.

```
public static void main(String[] args) {
    if (args.length != 3) {
        System.out.println(USAGE);
        System.exit(0);
    }

    String name = args[0];
    InetAddress addr = null;
    int port = 0;
    Socket sock = null;

    // check args[1] - the IP-adress
    try {
        addr = InetAddress.getByName(args[1]);
    } catch (UnknownHostException e) {
        System.out.println(USAGE);
        System.out.println("ERROR: host " + args[1] + " unknown");
        System.exit(0);
    }

    // parse args[2] - the port
    try {
        port = Integer.parseInt(args[2]);
    } catch (NumberFormatException e) {
        System.out.println(USAGE);
        System.out.println("ERROR: port " + args[2]
                           + " is not an integer");
        System.exit(0);
    }

    // try to open a Socket to the server
    try {
        sock = new Socket(addr, port);
    } catch (IOException e) {
        System.out.println("ERROR: could not create a socket on " + addr
                + " and port " + port);
    }

    // create Peer object and start the two-way communication
    try {
        Peer client = new Peer(name, sock);
        Thread streamInputHandler = new Thread(client);
        streamInputHandler.start();
        client.handleTerminalInput();
        client.shutDown();
    } catch (IOException e) {
        e.printStackTrace();
    }
```

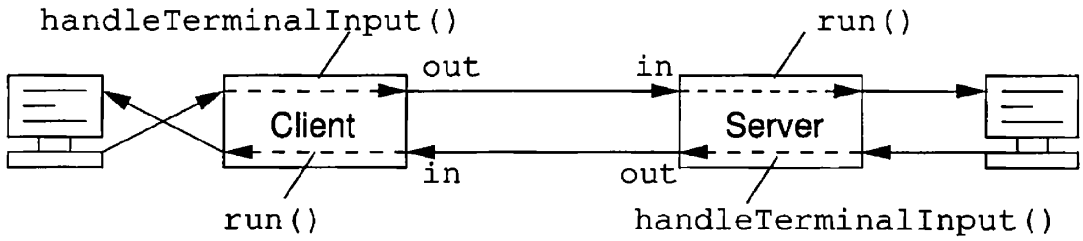


Figure 7.1: Schematic overview of client-server communication

}

First a `Socket` `sock` is constructed, based on the `addr` and `port` passed as argument. The socket is used to create a `Peer` object. Then a separate `Thread` is created that will handle the input from the socket's inputstream. Notice that this means that `Peer` is a `Runnable` object (and thus it should implement the method `run`). The method `handleTerminalInput` catches the input that is typed on the `Terminal` and sends it over the socket to the server.

Figure 7.1 gives a schematic overview of the communication between server and client application. The streams of the socket are called `in` and `out`, respectively.

The method `main` of the server is almost the same; the only different is in the checking of the arguments and the construction of the `Socket`.

The common behaviour of the client and the server is implemented in the class `Peer`. This class has the following four `protected` fields:

```
protected String name;
protected Socket sock;
protected BufferedReader in;
protected BufferedWriter out;
```

The class `Peer` has the following three methods:

- `public void run()`, which reads messages from `in` and prints them on `Terminal`
- `public void handleTerminalInput()`, which reads messages from the `Terminal` and sends these to `out`. If the string `''exit''` is entered, then the method should be finished. After sending the message to `out`, it is a good idea to make sure that the stream is emptied by using `flush()`.
- `public void shutDown()`, which closes both streams and the socket.

Additionally, the class `Peer` has the following constructor.

```
/*@
   requires (nameArg != null) && (sockArg != null);
 */
/**
 * Constructor. creates a peer object based in the given parameters.
 * @param   nameArg name of the Peer-proces
 * @param   sockArg Socket of the Peer-proces
 */
public Peer(String nameArg, Socket sockArg) throws IOException
```

On Blackboard, you can find Java frameworks for the classes `Peer`, `Server`, and `Client`.

P-7.21 Write the bodies of the methods `run`, `handleTerminalInput` and `shutDown` and of the constructor of class `Peer`. Make sure that all exceptions are treated properly. Input should be read per line, *i.e.*, each message should be finished by a newline.

The class `Server` is a class with only a `main` method. Similarly to the class `Client` it creates a `Socket` `sock` and a `Peer` object. When the `Server` application is invoked two Program Arguments must be provided (using the Eclipse launch configuration): `<name>` `<port>`

However, different from the class `Client`, first a `ServerSocket` should be created. After creating the `ServerSocket`, the server waits until a `Client` wants to connects with the `Server` over the `port`.

P-7.22 Write a method `main` of the class `Server`. Make sure that all arguments that are passed to the server are checked. After that, a `ServerSocket` should be created, and then the method should wait until a `client` wants to connect. When that happens, a `Peer` object is created, and also the input to the `Terminal` should be treated properly.

Notice that the method `main` of the `Server` has four `try/catch` blocks (*e.g.*, to check the arguments passed to the `Server`). Some of these also occur in the `main` method of the `Client`. It would be cleaner to wrap these `try/catch` blocks in methods (preferably in class `Peer`) so that they can also be used by the `Client`.

P-7.23 Test your application "locally" on your own computer.

Make sure that when you terminate the client or the server, also the peer (*i.e.*, the client or server) is closed properly. In any case, there should never be any uncaught exceptions visible on the `Terminal`.

P-7.24 When one of the two `Peer` processes closes the communication by typing `EXIT`, the 'peer' is not closing completely immediately. What is the reason? Can this be solved easily? Why not?

☞ **P-7.25** Test if your application works to communicate on different computers. Notice that you need to find the internet address of the computer that runs the server.

Networked attack

For this exercise you are given access to a simple networked recipe server and a matching client. The server however, is clumsily written and vulnerable to a security attack. First make sure you can run the recipe server and the client works as expected. For this you may have to set or change the working-directory in the "Run configuration". Another option is to move the directory containing the recipes.

☞ **P-7.26** Have a good look at the source code of the recipe server and client to have some understanding how it works. Use the given `RecipeClient.java` to create a `RogueRecipeClient.java` that retrieves data from the recipe server it actually should not. For example, can you get it to show the source code of the server? Look for the possibility of a so-called *injection attack*.

7.2.2 Recommended exercises

Concurrency Theory (extra)

P-7.27 Consider the following program fragment.

```
package ss.week7;

public class ConcatThread extends Thread {
    private static String text = ""; // global variable
    private String toe;

    public ConcatThread(String toeArg) {
        this.toe = toeArg;
    }

    public void run() {
        text = text.concat(toe);
    }

    public static void main(String[] args) {
        (new ConcatThread("one;")).start();
        (new ConcatThread("two;")).start();
    }
}
```

1. Which lines of `ConcatThread` are a critical section, and why?

2. What are the possible values of `text` after executing `ConcatThread`? Explain how these results can be achieved.
3. Adapt the method `run` of `ConcatThread` in order for the program to always terminate, ensuring that `text` has either the value `''one;two;''` or `''two;one;''`.
4. Adapt the method `run` of `ConcatThread` in such a way that `text` always has the value `''one;two;''`.

P-7.28 In this exercise, we develop a multithreaded version of the well-known quick sort algorithm. Consider the class `QuickSort`.

```
package ss.week7;


public class QuickSort {
    public static void qsort(int[] a) {
        qsort(a, 0, a.length - 1);
    }
    public static void qsort(int[] a, int first, int last) {
        if (first < last) {
            int position = partition(a, first, last);
            qsort(a, first, position - 1);
            qsort(a, position + 1, last);
        }
    }
    public static int partition(int[] a, int first, int last) {

        int mid = (first + last) / 2;
        int pivot = a[mid];
        swap(a, mid, last); // put pivot at the end of the array
        int pi = first;
        int i = first;
        while (i != last) {
            if (a[i] < pivot) {
                swap(a, pi, i);
                pi++;
            }
            i++;
        }
        swap(a, pi, last); // put pivot in its place "in the middle"
        return pi;
    }
    public static void swap(int[] a, int i, int j) {
        int tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
```

This class with the static method `qsort` is a simplified version of the quick sort algorithm discussed in Niño & Hosch [§19.3]. This version can only be used to sort an array of integer values.

Adapt the algorithm to a multi threaded version: each call to `qsort` should be done in a separate thread. (QuickSortTest) is provided which tests (QuickSort). You can use this test to quickly test your multi threaded version by changing it to use your version instead of the provided single threaded version.

This programming pattern is often called *fork-join-style programming*.

Multi-client chat

In the previous exercise you made a simple socket connection between a server and a client. But often you have multiple clients for one server. In this section you will build a multi-client chat.

When the `Server` is started with its main methods, `Clients` can log on to the `Server`.

The Server keeps on waiting for Clients that would like to connect with the Server. When a Client requests a Socket connection, the Server starts a separate ClientHandler thread that takes care of communication with this Client. The Server maintains a collection of all ClientHandler threads.

When a Client sends a message to its ClientHandler, the ClientHandler adds the name of the user in front of the message, and then passes this "personalised" messages to the broadcast message of the Server. The method broadcast offers the message to all ClientHandlers, which send it to "their" Client over the socket connection.

For this simple chat box, no (complicated) protocol is necessary. Once a Client has made a connection with the Server, he is logged on. The only thing that the ClientHandler should know is the name of the Client. Thus, the protocol specifies that the first message from the Client to the ClientHandler should always be its name.

When a Client wishes to close the connection, the user can close the Client application. This will disconnect the socket connection with the ClientHandler. When the ClientHandler notices this, by means of an exception, it will finish. However, before doing this, it first should signal this to the Server.

On Blackboard you can find incomplete Java files that you can use as a basis for this exercise.

P-7.29 Implement the missing bodies of the chatbox application. It is not mandatory that you follow the structure of the classes as provided on Blackboard. If you have a better solution, then you are free to change the classes, provided the global behaviour of the chat box remains the same.

The easiest class to implement is probably Client, as it has more or less the same functionality as the Client from the previous exercise.

P-7.30 Should the method broadcast of class Server not be declared synchronized?

P-7.31 Test your application on different computers. Your chat box application should also be able to collaborate with the client and server applications of your fellow students. Try this!

7.2.3 Project

Protocol Session

During the project session this week, you will agree with your tutorial group to on the protocol for the communication between the client and the server. One of the requirements is that your client application can communicate with the server of your fellow students (within the same tutorial group), and vice versa. Therefore, all pairs within the tutorial group should implement the same protocol for the client/server communication.

Below is a description how the communication in the game application could proceed. *However, it might be that in your tutorial group, you make different decisions about the responsibilities of the client and the server.*

After the TUI of the client has been started, a user can enter an internet address and port number of a server, and his own name. Afterwards, the client is ready to connect with the server. The client remains waiting until the server indicates that another client has also connected.

When a second client is connected, the game can begin, or the client can decide to wait for more clients (up-to a maximum number of players for the game). When the game begins, the server can assign colours to the players at random, or let the clients choose (for example in the order in which they connected). When the game has begon, the server keeps on waiting for new clients that would like to play the game. Thus, there may be multiple games played simultaneously on the server.

The game itself could proceed as follows. The player whose turn it is, decides about its next move (taking the rules of the game into account). The client checks if the move is legal, and if so, sends it to the server. The server also checks validity of the move, and if so, sends it to the other clients. The other clients update their game state accordingly. The turn now moves to the next player. The game is continued until it is finished according to the rules of the game.

The protocol should describe which data is sent between the client and the server, and in which order. For example, you might agree that the client and the server communicate via messages of class String, using Reader and Writer objects. In that case, a message could have the following format:

```
<command> <arg1> <arg2> <arg3>
```

where commands and arguments are separated by spaces (in that case, commands and arguments of course cannot contain spaces). An example of a command with a single argument is

```
join SleepingBeauty
```

With this command, a client indicates to a server that a user with the name *SleepingBeauty* would like to connect to play the game.

During this project session, you should decide which commands and arguments are necessary for the communication between client and server. In particular, you should reach agreement on the format of commands and arguments, and about the minimal set of commands that a client and server should understand. (It is of course allowed to extend the protocol for your own client/server application, to provide extra functionality to it.)

The protocol should be written down clearly, in such a way that all pairs are able to make a client and server implementation that adhere to the protocol. All commands (*e.g.* `join` in the example above) can be added to a separate Java class and this class can be made available to the other group members via Blackboard.

Game Playing Protocol

You know should be ready to start implementing the protocol as agreed upon with your tutorial group. You can make the following steps to achieve this:

- Make first a stub implementation for the game functionality, so that you have a working implementation.
- Replace the stub implementation by the intended implementation.
- Develop a test plan to test your protocol.
- Use the test plan to test your implementation on different computers.
- In particular, test if your implementation can communicate with client and server implementations of your fellow students (this functionality is needed for the tournament in Week 10).

7.2.4 Bonus exercises

The Mandelbrot Set

P-7.32 As you might have noticed, the view of a Mandelbrot set is not restored when a part of the image disappears to the background, and afterwards is moved back to the front. For this kind of recovery work, the class `JComponent` (which is a superclass of `JPanel`) the (*pure callback*) method `paintComponent(Graphics)`. The parameter `Graphics` indicates which part of the component should be redrawn (namely, the part within the rectangle that can be retrieved using `getClipBounds`). Implement this restaurant work by overwriting `paintComponent` in `MandelPanel`.

For efficiency reasons it is a good idea to store the results per pixel (*i.e.*, the colours computed per pixel) in a `width×height` matrix of `Colors` in the method `drawMandel`. This allows `paintComponent` to reread those values, instead of recomputing them.

Chatbox

P-7.33 In this exercise, we extend the previous build commandline chat. We will replace the "commandline" interface by graphical components, such as `JFrames`. Thus resulting a basic *chatbox* application. To start copy the files from cmdchat to a new package.

Figure 7.2 shows an example of a graphical user interface for a client. The fields that first were passed on the commandline are now `JTextFields` on the `JFrame`. Figure 7.3 shows an example GUI for the server.

On Blackboard you can find the class `ss.week7.challenge.chatbox.ServerGUI`. This creates a GUI for the server. Create a class `ss.week7.challenge.chatbox.ClientGUI` similar to `ServerGUI`. In this exercise we are only concerned with the user interface aspects. The GUI should have the same components as Figure 7.2.

The initial state of the `ClientGUI` should be configured as follows:

- The "Connect" button should not be selectable yet (it only becomes selectable once the fields "Hostname", "Port" and "Name" have been completed).
- While the client is not connected yet with the server, the "My Message" field should not be selectable.
- The "Messages" area should not be editable. This is where the messages coming from the server will be shown.

On Blackboard, you can also find the interface `ss.week7.challenge.chatbox.MessageUI`. Both `ServerGUI` and `ClientGUI` implement this interface. What is the advantage of this arrangement?

Now we have to adapt the cmdchat, so that it works with the GUI. For the `Server` en `Client` remove the main method and move the functionality of this main method to the GUI class. (Creation and starting of the `Client` and `Server` classes. Also replace the System.println to an output in the textboxes.

Note: The `Client` and `Server` classes should be Threads otherwise they block the GUI.

Confusing confidentiality and integrity

This exercise is mainly aimed at students who participated in the security pearl of module 1.

P-7.34 A common security mistake is to confuse confidentiality and integrity. In other words, some (real-world) implementations assume that encryption is enough to also provide integrity. There have been systems for example that used simple unauthenticated encryption to protect (browser) cookies containing session information. In the following exercise you will experience some of the problems with such an approach.

You are provided a `BadCookieCrypto` class. This class has two methods:

- `public String createCookie()`
  This method creates an encrypted cookie (encoded in Base64) containing (among others) the authorizations of the user in the browser session.
- `public boolean isAdmin(String cookie)`
  This method accepts a string containing a base64 encoded ciphertext representing the cookie. The method decrypts the cookie and determines whether the user presenting the cookie has admin rights. If so, it returns true, otherwise false.

This is what you should know about the encryption scheme: the `BadCookieCrypto` class uses CTR mode (see wikipedia) with a random IV together with an AES blockcipher and a random (64 bit) nonce and 64 bit counter. As wikipedia will tell you, in CTR mode, encryption boils down to XORing a plaintext block with the encryption of the nonce concatenated with the block counter. *So in essence, the ciphertext byte array is the result of a byte-wise XOR operation of a cipherstream and the plaintext.*

In addition, you know that the plaintext is of variable length, but it always ends with the string ";admin=N".

You now have enough information to be able to manipulate the ciphertext in such a way that the cookie token will allow a user presenting the cookie will have admin access. All without editing the `BadCookieCrypto` class!

Consider the following code and remember that the XOR operator in Java is "^".

```
byte c = 0x60 ^ 0x65;
byte b = c ^ 0x60;
```

What is the value of `b`?

Write a program that instantiates the `BadCookieCrypto` class, calls the `createCookie()` method to produce an encrypted cookie, and manipulates the the ciphertext in such a way that the `BadCookieCrypto` class is tricked into accepting the manipulated ciphertext as a valid admin cookie.
Hint: to turn a character into a byte you can use typecasting (e.g., the result of `(byte)'a'` will be a byte of value 0x60).

Week 8

8.1 Overview

8.1.1 Contents of This Week

Academic Skills This week contains no session on academic skills.

Design This week contains no session on design.

Programming This week most activities focus on the programming project. It is organised as follows.

- Monday is the last programming lab session where you can still sign off exercises.
- The rest of this week every day at least the first two or four hours after lunch, student assistants are available to help out with problems for the project. The student assistants may ask you to show your planning, and help you to evaluate your progress compared to your original planning. If necessary, they can help you to adjust your planning.
- Tuesday afternoon starts with a peer feedback session on your progress with the programming project.
- Friday morning there is a question and answer session, where you can ask all theoretical programming questions, as exam preparation.
- The remaining time you can work individually on the project.

8.1.2 Mandatory Presence

During the following activities, your presence is mandatory.

- (P) Peer feedback session on implementation of game logic (Tue 6)

8.1.3 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 20 hours self-study for the programming project.
- 4 hours self-study to prepare for the programming exam.

8.2 Programming

8.2.1 Project

Peer Feedback

In the peer feedback session (Tuesday) you have to present your implementation of the game logic to another project group. You can decide yourself with which other project group you do the peer feedback.

The purpose of this peer feedback is to:

- present your solution in a structured way to your peer;
- receive comments on your solution, possibly with proposals for improvement;
- comprehend the solution of your peer and recognize alternative solution paths;
- provide comments and proposals to your peer.

You can follow the guidelines described below for performing the peer feedback session. First one group presents their solution (this group will be called the *reader*) and the other group follows the presentation and provides comments (this group will be called the *inspector*). After discussing the solution of the first group, you switch the roles and discuss the solution of the other group. As a *reader* you should do the following:

- Present the design of the solution and explain which part of the design corresponds to which part of the assignment.
- Map the design to the implementation: which parts of the design are implemented by which classes and in which way. Explain the class invariants in your implementation.
- Present your test plan: explain each test case in terms of which part of the implementation is used, which assumptions the test case makes, and to which part of the assignment the test case corresponds.
- Perform a code walk-through (a walk-through of the complete code will take too long; therefore you should focus on the code parts you consider most difficult to write):
  - Start at the main method or at a test method and follow the control flow of the program.
  - If you reach a decision point, make an assumption on the user input and continue along the corresponding path.
  - Explain the code you reach in this walk-through statement-by-statement.
  - When you reach a method definition, also explain the signature of the method, the pre- and postconditions.
  - When you reach loops, explain the loop invariants.
  - You can repeat the walk-through and make different assumptions in order to walk along a different path.
- Point out anything else you consider interesting or challenging, or where you are uncertain about your solution.
- Note down questions or suggestions for improvement that you receive.

As an *inspector* you can pay attention to the following questions:

- Does the design correctly (and completely) reflect the assignment?
- Does the code correctly (and completely) implement the design?
- Are the invariants, pre- and postconditions correct (not too strong, not too weak)?
- Are the parameter types and result type of each method appropriate?
- Are classes, variables, methods, etc. named well (i.e., are names descriptive)?
- Are there (variable etc.) names confusingly similar?
- Are there variables/literals that should be constants?
- Are the visibility modifiers appropriate (not too strong or too weak)?
- In conditions: are the comparison operators correct (e.g., not <= instead of <, etc.), are logic operators used correctly (e.g., not & instead of &&, etc.)?
- Are meaningful JavaDoc comments provided?
- Are the used data structures appropriate?

Week 9

9.1 Overview

9.1.1 Contents of This Week

Academic Skills This week contains no session on academic skills.

Design

- Friday morning there will be a resit for the design test.

Programming This week most activities focus on the programming project. It is organised as follows.

- Monday morning there is the written exam on programming.
- This week every day the first two hours after lunch, student assistants are available to help out with problems for the project. The student assistants may ask you to show your planning, and help you to evaluate your progress compared to your original planning. If necessary, they can help you to adjust your planning.
- Tuesday afternoon starts with a peer feedback session on your progress with the programming project.
- The remaining time you can work individually on the project.

9.1.2 Mandatory Presence

During the following activities, your presence is mandatory.

- (P) Programming test (Mon 1 - 4)
- (P) Peer feedback session on protocol implementation and general progress (Tue 6)

9.1.3 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 24 hours self-study for the programming project.

9.2 Programming

9.2.1 Project

Peer Feedback

This week, the focus of the peer-feedback session is on the protocol implementation and on the general progress with implementing the game. Make sure to do the peer feedback with a different project group.

The purpose of this peer feedback is to:

- present your solution in a structured way to your peer;
- receive comments on your solution, possibly with proposals for improvement;
- comprehend the solution of your peer and recognize alternative solution paths;
- provide comments and proposals to your peer.

You can roughly follow the guidelines from last week (Section 8.2.1). Take an extra look at the following points:

- Is the protocol implemented correctly? Check this by playing a game with the server from one group and the client from another group.
- Are parts of the protocol forgotten?
- Is it possible to send corrupt data to the server (to cheat or to crash the server)?
- Does the implementation support multiple games simultaneously?

Week 10

10.1 Overview

10.1.1 Contents of This Week

Academic Skills This week contains no session on academic skills.

Design

- Friday is the deadline to hand in a repaired version of the Design project for a maximum 6.0 grade.

Programming This week most activities focus on the programming project. It is organised as follows.

- Monday and Tuesday the first two hours after lunch, student assistants are available to help out with problems with the project. Wednesday student assistants are available the whole day.
- Tuesday afternoon starts with a peer feedback session on your progress with the programming project.
- Wednesday afternoon the tournament will take place (see the project description). It is expected that at the end of the afternoon, all pairs have a working version of the project that can play against another implementation without any problems.
- Deadline for the project is **Wednesday 23:59 CET.**
- Thursday afternoon there is a resit for the written programming test.
- Friday morning there are student assistants available for project groups that have to make corrections to their implementation (both for design and programming).
- Friday afternoon there is a resit for the mathematics test.
- Friday is the deadline to hand in a version of the Programming project for a maximum 6.0 grade.

10.1.2 Mandatory presence

During the following activities, your presence is mandatory.

- (P) Peer feedback session on progress with the project implementation and report (Tue 6)
- (P) Tournament (Wed 6 - 9)

10.1.3 Expected Self-Study and Project Work

In addition to all the scheduled activities, we estimate that you need approximately:

- 10 hours self-study for the programming project.

10.2 Programming

10.2.1 Project

Peer Feedback

This week, the focus of the peer-feedback session is on the general progress with implementing the game and the report. Make sure to do the peer feedback again with a different project group.

The purpose of this peer feedback is to:

- present your solution in a structured way to your peer;
- receive comments on your solution, possibly with proposals for improvement;
- comprehend the solution of your peer and recognize alternative solution paths;
- provide comments and proposals to your peer.

You can roughly follow the guidelines from the last two weeks (Section 8.2.1). Take an extra look at the following points:

- Does the report contain all the mandatory elements?
- Is the report at the appropriate level of detail?
- Is the report well-structured?

Appendix A

Java Modeling Language

In Chapter 5 of Niño & Hosch, an annotation language is defined for *programming by contract*. During this module, we use a variation of the annotation language defined in Niño & Hosch. This language, called Java Modeling Language (or JML, for short) is a widely-used annotation language for Java. The advantage of using JML instead of the annotation language in the book is that there is tool support available. During this module, we will only use type checking facilities of the tool support, but in some of the challenge exercises, you will be challenged to experiment with more advanced tool support.

JML is a large language, with many different specification constructs. There is an online reference manual available via `http://www.jmlspecs.org`. On this webpage, you can also find references to tools and papers about JML. The remainder of this appendix describes the basic ingredients of JML (method specifications, class invariants, and loop invariants) that we will use in the Software Systems module. Method specifications and class invariants are introduced in week 2, loop invariants in week 5.

A.1 JML Method Contracts

Ingredients of a Method Contract So, what exactly is a method contract? A method contract consists of two things: it describes what is expected from the code that calls the method, and it provides guarantees about what the method will actually do.

The expectations on the caller are called the *precondition* of the method. Typically, these will be conditions on the method's parameters, *e.g.*, the argument should be a positive integer, or a non-null pointer, but the precondition can also describe that the method can only be called when the object is in a particular state. In JML, every precondition expression is preceded by the keyword `requires`.

The guarantees provided by the method are called the *postcondition* of the method. They describe how the object's state is changed by the method, or what the expected return value of the method is. A method only guarantees its postcondition to hold whenever it is called in a state that respects the precondition. If it is called in a state that does not satisfy the precondition, then no guarantee is made at all. In JML, every postcondition expression is preceded by the keyword `ensures`.

JML specifications are written as special comments in the Java code, starting with `/*@` or `//@`. The `@` sign allows the JML parser to recognise that the comment contains a JML specification. Sometimes, JML specifications are also called *annotations*. The preconditions and postconditions are basically just Java expressions (of Boolean type).

Example 1.1 *Figure A.1 contains an example of a basic JML specification. The specification should be understood independent of the implementation, therefore we just provide a stub implementation here. The*

```
package ss.jml;

public class Student {

    public static final int BACHELOR = 0;
    public static final int MASTER = 1;

    /*@ pure */ public String getName() {
        return null;
    }

    //@ ensures \result == BACHELOR || \result == MASTER;
    /*@ pure */ public int getStatus() {
        return 0;
    }

    //@ ensures \result >= 0;
    /*@ pure */ public int getCredits() {
        return 0;
    }

    //@ ensures getName().equals(n);
    public void setName(String n) {
    }

    /*@ requires c >= 0;
        ensures getCredits() == \old(getCredits()) + c;
    */
    public void addCredits(int c) {
    }

    /*@ requires getCredits() >= 180;
        requires getStatus() == BACHELOR;
        ensures getCredits() == \old(getCredits());
        ensures getStatus() == MASTER;
    */
    public void changeStatus() {
    }

}
```

Figure A.1: First JML example specification `Student`

specification contains contracts for the methods in a class `Student`*, representing a typical UT student.*

We discuss the different aspects of this example in full detail.

- For method `getName`, we specify that it is a `pure` method, *i.e.*, it may not change the state in any way (in other words: it may not have any (visible) side effects). Only pure methods may be used in specifications, because these do not change the state.

- Method `getStatus` is also pure. In addition, we specify that its result may only be one of two values: `BACHELOR` or `BACHELORMASTER`. To denote the return value of the method, the reserved JML-keyword `\result` is used.

- For method `getCredits` we also specify that it is pure, and in addition we specify that its return value must be non-negative; a student thus never can have a negative amount of credits.

- Method `setName` is non-pure, *i.e.*, it may change the state. Its postcondition is expressed in terms of the pure methods `getName` and `equals`: it ensures that after termination the result of `getName` is equal to the parameter n.

- Method `addCredits`'s precondition states a condition on the method parameters, namely that only a positive number of credits can be added. The postcondition specifies how the credits change. Again, this postcondition is expressed in terms of a pure method, namely `getCredits`. Notice the use of the keyword `\old`. An expression `\old(`E`)` in the postcondition actually denotes the value of expression E in the state where the method call started, the *pre-state* of the method. Thus the postcondition of `addCredits` expresses that the number of credits only increases: after evaluation of the method, the value of `getCredits` is equal to the old value of `getCredits`, *i.e.*, before the method was called, plus the parameter `c`.

- Method `changeStatus`'s precondition specifies that this method only may be called when the student is in a particular state, namely he has obtained a sufficient amount of credits to pass from the Bachelor status to the Master status. Moreover, the method may only be called when the student is still having a Bachelor status. The postcondition expresses that the number of credits is not changed by this operation, but the status is. Notice that the two preconditions and the two postconditions of `changeStatus` are written as separate `requires` and `ensures` clauses, respectively. Implicitly, these are assumed to be joined by conjunction, thus the specification is equivalent to the following specification:

```
/*@ requires getCredits() >= 180;
    requires getStatus() == BACHELOR;
    ensures getCredits() == \old(getCredits());
    ensures getStatus() == BACHELORMASTER;
*/
public void changeStatus() {}
```

Specifications and Implementations Notice that the method specifications are independent of possible implementations. One could imagine different implementations of this class, as long as they respect the specification. One obvious implementation is using a field `credits` that keeps track of the number of credits earned by the student. However, an alternative implementation is to keep track of a list of courses as well as the credits earned for each course and to compute the total number of credits as the sum of the credits of the individual courses.

Method specifications do not always have to specify the exact behaviour of a method; they give minimal requirements that the implementation should respect.

Example 1.2 *Consider the specification in Figure A.1 again. The method specification for* `changeStatus` *prescribes that the credits may not be changed by this method. However, method* `addCredits` *is free to update the status of the student. So for example, an implementation that silently updates the status from Bachelor to Master whenever appropriate is according to the specification.*

```
/*@ requires c >= 0;
    ensures getCredits() == \old(getCredits()) + c;
*/
public void addCredits(int c) {
   credits = credits + c;
   if (credits >= 180) {status = BACHELORMASTER};
}
```

Notice also that both `addCredits` and `changeStatus` would be free to change the name of the student, according to the specification, even though we would typically not expect this to happen. A way to avoid this, is to add explicitly conditions `getName().equals(\old(getName()))` to all postconditions. JML provides way to capture this more concisely, but we will not go into more details here (if you would like to know more, you could look at the reference manual).

Default Specifications You might have wondered why not all specifications in `Student` have a pre- and a postcondition. Implicitly though, they have. For every specification clause, there is a default. For pre- and postconditions this is the predicate `true`, *i.e.*, no constraints are placed on the caller of the method, or on the method's implementation.

Example 1.3 *Thus for example the specification of method* `getStatus` *actually is the following:*

```
/*@ requires true;
    ensures status == BACHELOR || status == BACHELORMASTER;
*/
public int getStatus() {
   return status;
}
```

However, there is one exception to this. In JML all reference values are implicitly assumed to be non-null, except when explicitly annotated otherwise (using the keyword `nullable`).

Example 1.4 *This means that the methods* `getName` *and* `setName` *have implicit pre- and postconditions about the non-nullity of the parameter and the result. Explicitly, their specifications are as follows:*

```
/*@ requires true;
    ensures \result != null;
*/
/*@ pure */ public String getName() {return "";}

/*@ requires n != null;
    ensures getName().equals(n);
*/
public void setName(String n) {}
```

Notice that the non-null by default also can have some unwanted effects, as illustrated by the following example.

Example 1.5 *Consider the following declaration of a* `LinkedList`.

```
public class LinkedList {
   private Object elem;
   private LinkedList next;
   ....
```

Because of the non-null by default behaviour of JML, this means that all elements in the list are non-null. Thus the list must be cyclic, or infinite. This is usually not the intended behaviour, and thus the `next` *reference should be explicitly annotated as* `/*@ nullable @*/`.

Specification Expressions Above, we have already seen that standard Java expressions can be used as predicates in the specifications. These expressions have to be side-effect-free, thus for example assignments are not allowed. As also mentioned above, these predicates may contain method calls to pure methods.

In addition, JML defines several specification-specific constructs. The use of the `\result` and `\old` keywords has already been demonstrated in Figure A.1, and the official language specification contains a few more of these. Besides the standard logical operators, such as conjunction[1] `&`, disjunction `|` and negation `!`, also extra logical operators are allowed in JML specifications, *e.g.*, implication `==>`, and logical equivalence `<==>`. Also the standard quantifiers ∀ and ∃ are allowed in JML specifications, using keywords `\forall` and `\exists`.

Example 1.6 *Using these, we can specify for example that an array argument should be sorted.*

```
//@ requires (\forall int i, j;  <= i & i < j & j < a.length; a[i] <= a[j]);
public ... manipulateArray(int [] a) {...
```

The first argument (`int i, j`) is the declaration of the variables over which the quantification ranges. The (optional) second argument (`0 <= i & i < j & j < a.length`) defines the range of the values for this variable, and the third argument is the actually universally quantified predicate (`a[i] <= a[j]` in this case).

Example 1.7 *An alternative way to phrase the specification in Example 1.6 is the following:*

```
//@ requires (\forall int i, j; 0 <= i & i < j & i < a.length; a[i] <= a[j]);
public ... manipulateArray(int [] a) {...
```

Validating Method Contracts A way to validate your specifications is by inserting assertions at appropriate positions in your program, as described on page 239 of Niño & Hosch. As said in Niño & Hosch, validating a postcondition means that one validate one's own implementation. Therefore, it should not be necessary to always have the postcondition check enabled.

As said, there is a wide range of tools available for JML. In particular, many of these tools provide support for run-time checking. This means that they automatically transform the code to validate the pre- and postconditions during the execution. This can be very useful in the testing phase. OpenJML that we use for type checking of JML specifications, also provides support for run-time checking.

A.2 Class Invariants

Consider again the specification of `Student` in Figure A.1. If we look carefully at the specifications and the description that we give about the student's credits, we notice that implicitly we assume some properties about the value of `getCredits` that hold throughout. For example, we wrote above:

> "a student thus never can have a negative amount of credits"

and also

> "the number of credits only increases."

But if we would like to make explicit that we assume that these properties always hold, we would have to add this to *all* the specifications in `Student`, and thus in particular also to all methods that do not relate at all to the number of credits. Thus for example, we would get the following specification:

```
/*@ requires getCredits() >= 0;
    ensures \result == BACHELOR || \result == BACHELORMASTER;
    ensures getCredits() >= \old(getCredits());
*/
/*@ pure */ public int getStatus() {return 0;}
```

[1] Since expressions are not supposed to have side effects or terminate exceptionally, in JML the difference between logical operators `&` and `&&`, and `|` and `||` is not important.

Clearly, this is not desired, because specifications would get very large, and besides describing the intended behaviour of that particular method, they also describe properties that are related to how an object can evolve over time.

Therefore, JML provides class invariants to restrict how the internal state of an object can change during the object's lifetime.

An object invariant[2] is a predicate over the object state that holds in all *visible* states of an object. A visible state of an object is defined to be any state in which a method call to the object either starts or terminates. Thus, an invariant I is implicitly added as a precondition and a postcondition to every method in the class. In addition, also the post-states of the constructor are visible states, thus any constructor has to ensure that the invariant is established.

Note that we only show how the first property is specified as an object invariant. The second property, namely that the credits can only increase, requires using the `\old` keyword, which is not supported by the `invariant` keyword in JML. There are other ways to specify invariants in JML, which also allow using `\old` expression, however, this is not discussed in this document.

Example 1.8 *Figure A.2 shows three possible invariants that can be added to interface* `Student` *(and removes method specifications that now have become superfluous). These specify that credits are never non-negative; a student's status is always either Bachelor or Master, and nothing else; and if a student's status is Master, he or she has earned more than 180 credits.*

Of course, instead of specifying invariants, one could also add these specifications to all pre- and post-conditions explicitly. However, this means that if you add a method to a class, you have to remember to add these pre- and postconditions yourself. Moreover, invariants are also inherited by subclasses (and by implementations of interfaces). Thus any method that overrides a method from a superclass still has to respect the invariants. And any method that one adds in the subclass also has to respect the invariants from the superclass. Specifying the invariants centrally as class invariants, thus leads to a very nice separation of concerns.

An important point to realise is that invariants have to hold only in all *visible object states*, *i.e.*, in all states in which a method is called or terminates. Thus, inside the method, the invariants may be temporarily broken.

Example 1.9 *The following possible implementation of* `addCredits` *is correct, even though it breaks the invariant that a student can only be studying for a Master if he or she has earned more than 180 points inside the method: if* `credits + c` *is sufficiently high, the status is changed to Master. After this assignment the invariant does not hold, but because of the next assignment, the invariant is re-established before the method terminates.*

```
/*@ requires c >= 0;
    ensures getCredits() == \old(getCredits()) + c;
*/
public void addCredits(int c) {
   if (credits + c>= 180) {status = BACHELORMASTER;} // invariant broken!
   credits = credits + c;
}
```

Validating Class Invariants If one wishes to validate the specified class invariants, assertions should be added at the beginning and end of every method call. Clearly, this is not a task that you want to do manually, but where tool support is necessary. Again, OpenJML provides support for doing this.

[2]Not to be confused with loop invariants, as discussed below.

```
package ss.jml;

public class StudentWithInv {

    public static final int BACHELOR = 0;
    public static final int MASTER = 1;

    //@ invariant getCredits() >= 0;
    //@ invariant getStatus() == BACHELOR || getStatus() == MASTER;
    //@ invariant getStatus() == MASTER ==>  getCredits() >= 180;

    /*@ pure */ public String getName() {
        return null;
    }

    /*@ pure */ public int getStatus() {
        return 0;
    }

    /*@ pure */ public int getCredits() {
        return 0;
    }

    //@ ensures getName().equals(10);
    public void setName(String n) {
    }

   /*@ requires c >= 0;
        ensures getCredits() == \old(getCredits()) + c;
    */
    public void addCredits(int c) {
    }

    /*@ requires getCredits() >= 180;
        requires getStatus() == BACHELOR;
        ensures getCredits() == \old(getCredits());
        ensures getStatus() == MASTER;
    */
    public void changeStatus() {
    }


}
```

Figure A.2: Interface `Student` with class-level specifications