Software Systems

(Course code: 201700117)
Module 2 of the CS and BIT curricula
University of Twente
2019-2020 Academic year

Module team:
Luís Ferreira Pires (*Module Coordinator*)
Joke van Staalduinen (*Design Thread*)
Marcus Gerhold (*Design Thread*)
Tom van Dijk (*Programming Thread*)
Maarten Everts (*Security*)
Jasper de Jong (*Mathematics Line Coordinator*)
Yeray Barrios Fleitas (*Skills Thread*)
Wim Kamerman (*Manual revision and Programming Project support*)

15th January 2020

Contents

Introduction **1**
- Module Overview . . . 1
- Learning Objectives . . . 2
- Modes of Instruction . . . 3
- Project Groups and Lab Groups . . . 3
- Study Material and Systems . . . 3
- Tests and Grades . . . 5
- Mandatory Activities . . . 7
- Signing off Mandatory Exercises . . . 7
- Teachers and Assistants . . . 8
- Monitoring and Evaluation of the Module . . . 8
- Acknowledgements . . . 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
  - Global Description of the Game Application . . . 13
  - Communication Protocol . . . 13
  - Functional Requirements of the Application . . . 13
  - Report Requirements . . . 14
  - Possible Extensions of the Application . . . 16
  - Packaging for Submission . . . 17
  - Grading . . . 17
  - Grading Criteria . . . 18
  - Project Activities . . . 20
  - Important Dates . . . 20

1 Week 1 **21**
- 1.1 Overview . . . 21
  - 1.1.1 Mandatory Presence . . . 21
  - 1.1.2 Expected Self-Study and Project Work . . . 22
  - 1.1.3 Materials for This Week . . . 22
  - 1.1.4 Tool Installation Session . . . 22
- 1.2 Academic Skills . . . 22
  - 1.2.1 Assignments . . . 22
- 1.3 Design . . . 23
  - 1.3.1 Project . . . 23
  - 1.3.2 Laboratory session 1a (Activity Diagrams) . . . 23
  - 1.3.3 Recommended exercises 1a (Activity Diagrams) . . . 25

1.3.4 Laboratory session 1b (Use cases) . . . 26
1.3.5 Recommended exercises 1b (Use Cases) . . . 30
1.4 Programming . . . 32
1.4.1 Laboratory exercises . . . 32
1.4.2 Recommended exercises . . . 39

2 Week 2 **41**
2.1 Overview . . . 41
2.1.1 Contents of This Week . . . 41
2.1.2 Mandatory Presence . . . 41
2.1.3 Expected Self-Study and Project Work . . . 41
2.1.4 Materials for this Week . . . 42
2.2 Academic Skills . . . 42
2.2.1 Assignments . . . 42
2.3 Design . . . 42
2.3.1 Laboratory session 2a (Class Diagrams) . . . 42
2.3.2 Recommended exercises 2a (Class Diagrams) . . . 43
2.3.3 Project . . . 46
2.3.4 Laboratory session 2b (Sequence Diagrams) . . . 46
2.3.5 Recommended exercises 2b (Sequence Diagrams) . . . 49
2.4 Programming . . . 50
2.4.1 Laboratory exercises . . . 50
2.4.2 Recommended exercises . . . 55

3 Week 3 **57**
3.1 Overview . . . 57
3.1.1 Contents of This Week . . . 57
3.1.2 Mandatory Presence . . . 57
3.1.3 Expected Self-Study and Project Work . . . 58
3.1.4 Materials for this Week . . . 58
3.2 Academic Skills . . . 58
3.2.1 Assignment . . . 58
3.3 Design . . . 58
3.3.1 Laboratory session 3a (State Machine Diagrams) . . . 58
3.3.2 Recommended exercises 3a (State Machine Diagrams) . . . 60
3.3.3 Project . . . 61
3.3.4 Laboratory Session 3b (Versioning) . . . 61
3.4 Programming . . . 65
3.4.1 Laboratory exercises . . . 65
3.4.2 Recommended exercises . . . 71

4 Week 4 **75**
4.1 Overview . . . 75
4.1.1 Contents of This Week . . . 75
4.1.2 Mandatory Presence . . . 75
4.1.3 Expected Self-Study and Project Work . . . 76
4.1.4 Materials for this Week . . . 76
4.2 Academic Skills . . . 76
4.2.1 Assignments . . . 76
4.3 Design . . . 77
4.3.1 Laboratory session 4 (Software Metrics) . . . 77
4.3.2 Project . . . 78
4.4 Programming . . . 78
4.4.1 Laboratory exercises . . . 78
4.4.2 Recommended exercises . . . 82
4.5 Mathematics: Relevance for Computer Science and Business Information Technology . . . 82

5 Week 5 **83**
5.1 Overview . . . 83
5.1.1 Contents of This Week . . . 83
5.1.2 Deadline . . . 83
5.1.3 Mandatory Presence . . . 84
5.1.4 Expected Self-Study and Project Work . . . 84
5.1.5 Materials for this Week . . . 84
5.2 Design . . . 84
5.2.1 Test . . . 84
5.2.2 Project . . . 85
5.3 Programming . . . 85
5.3.1 Laboratory exercises . . . 85
5.3.2 Recommended exercises . . . 88

6 Week 6 **91**
6.1 Overview . . . 91
6.1.1 Contents of This Week . . . 91
6.1.2 Mandatory Presence . . . 91
6.1.3 Expected Self-Study and Project Work . . . 91
6.1.4 Materials for this Week . . . 92
6.2 Academic Skills . . . 92
6.2.1 Peer feedback session . . . 92
6.3 Programming . . . 92
6.3.1 Laboratory exercises . . . 92
6.3.2 Project . . . 100
6.3.3 Recommended exercises . . . 100

7 Week 7 **105**
7.1 Overview . . . 105
7.1.1 Contents of This Week . . . 105
7.1.2 Mandatory Presence . . . 105
7.1.3 Expected Self-Study and Project Work . . . 105
7.1.4 Materials for this Week . . . 106
7.2 Programming . . . 106
7.2.1 Laboratory exercises . . . 106
7.2.2 Recommended exercises . . . 110
7.2.3 Project . . . 113

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 . . . 116
8.2 Programming . . . 116
8.2.1 Project . . . 116

9 Week 9 **119**
9.1 Overview . . . 119
9.1.1 Contents of This Week . . . 119
9.1.2 Mandatory Presence . . . 119
9.1.3 Expected Self-Study and Project Work . . . 119
9.2 Programming . . . 120
9.2.1 Project . . . 120

10 Week 10 **121**
10.1 Overview . . . 121
10.1.1 Contents of This Week . . . 121
10.1.2 Mandatory presence . . . 121
10.1.3 Expected Self-Study and Project Work . . . 121
10.2 Programming . . . 122
10.2.1 Project . . . 122

A Testing **123**
A.1 Types of tests . . . 123
A.2 When should tests be written? . . . 124
A.3 Testing and verification . . . 125
A.4 JUnit 5 . . . 126

Introduction

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

Module Overview

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 some 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. In this respect, this module builds upon the knowledge of algorithms and recursion acquired in Module 1.1. 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, time management and self-management). This module also contains the Calculus 1B course of the Mathematics line, which covers the theory of integrals, power series, differential equations and complex numbers.

Position in the Curricula This module is offered in the second period (Quarter 1B) of the Computer Science (CS) 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 to 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.

Education Threads This module consists of four education 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 management and self-management, and reflection on one's own behaviour with regard to planning.
- *Mathematics (M)*[1]: the Calculus 1B course, which covers the theory of mathematical functions and integrals.

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

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

[1] This thread is called *Mathematics Line* on Canvas.

Software Systems in other study programmes (as minor, premaster etc.) If you are not a student of the CS or BIT Education Programme, you may follow parts of this module in one of the following variants:

- *Software Systems as Minor Module (12 EC)*: only the Design, Programming and Academic Skills threads, with the additional 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 thread.

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 by using UML models (at least class diagrams, activity diagrams and state machines), with the help of software tools that are suitable for this purpose.
- Analysing the relations between different models on the one hand, and between each model and the software code on the other hand.
- Explaining the commonly recognized phases of a structured software development process.
- Measuring and interpreting basic software metrics to assess the quality characteristics of a code base.

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

- Applying the core concepts of imperative programming, such as variables, data types, structured programming statements, recursion, lists, arrays, methods, parameters and exceptions.
- Applying the core concepts of object-orientation, such as object, class, value, type, object reference, interface, specialisation / inheritance, and composition.
- Implementing interactive applications using the Model/View/Controller pattern.
- Applying basic synchronisation mechanisms, such as monitors, locks and wait sets, to the problems of concurrent threads (race-conditions).
- Implementing client-server programs using basic network mechanisms such as Java sockets.
- Applying the basic concepts and techniques of security engineering to address the challenges of producing secure software.
- Implementing software of average size (10-20 classes) in Java by using the core concepts of imperative programming and object-orientation.
- Documenting software of average size (10-20 classes) by defining preconditions, postconditions and (class) invariants. Defining and performing a test plan for software of average size (10-20 classes) with appropriate test coverage.

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

- Applying the most important principles and techniques for effective time management, like personal planning elaboration and reflection, identification of personal strengths and weaknesses, giving and receiving feedback, and procrastination avoidance, in a project of limited complexity.

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

- Working with vectors and elementary properties of functions, especially with the rules of differentiability: applying elementary vector operations, calculating dot product and cross product, applying elementary properties of functions, calculating derivatives using differentiation rules and the derivatives of elementary functions;
- Solving constant-coefficient linear differential equations and work with complex numbers: solving homogeneous equations using the characteristic equation, solving first and second order equations using the method of undetermined coefficients, solving initial / boundary value problems, plot (sets

of) complex numbers in the plane, calculating absolute value and argument of a complex number to express the complex number in polar form, applying the complex arithmetic operations, finding roots of a complex number and solving binomial equations.

Modes of Instruction

The following modes of instruction are used in this module, some of which have already been used 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 you to study the course material. It is therefore not meant to replace the course material but rather to complement it. Some lectures also include exercise sessions.

Tutorial You work on exercises and get feedback from a teacher or teaching assistant.

Laboratory You work in pairs on the lab exercises, supervised by a teaching assistant. The exercises are signed off and participation is mandatory. In rooster.utwente.nl this mode of instruction is called *Practical*.

Self-study supervised You perform self-study, and an instructor (teacher or teaching assistant) is present in the room to answer your questions.

Project supervised You work on one of the projects, and an instructor (teacher or teaching assistant) is present in the room to answer your questions. The D-project (due in Week 5) is performed in groups of four students, while the P-project (due in Week 10) is performed in pairs.

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

Seminar You have the opportunity to ask questions to help you prepare for the exam.

Diagnostic test You perform exercises (typically individually) to assess how well you are acquainted with the material. Participation is mandatory.

The module schedule with the actual sessions (lectures, tutorials, etc.) can be found in rooster.utwente.nl and on Canvas. In case these systems show contradictory information, the information found in rooster.utwente.nl prevails.

Project Groups and Lab Groups

All lab exercises and the programming project should be performed in pairs. The design project will be done in groups of four students. The procedure to form groups will be announced on Canvas before the first day of the module.

Study Material and Systems

For the Programming thread we have selected a book and a set of software tools. Furthermore, some additional material is used to cover some topics not addressed in the book. For the Mathematics line, the material is listed on the Canvas site of the *2019-2020 Mathematics line*. A link to this site can be found on the Canvas site of this module. The complete list of resources for this module is the following:

Academic skills No mandatory book, but instead the lecture slides and handouts are the main course material.

Design No mandatory book, but instead the lecture slides are the main course material. There is an additional handout for the topic Use Cases (Week 1); for Software Metrics (Week 4) recommended reading will be made available on Canvas.

Programming David J. Eck. *Introduction to Programming Using Java*. Version 8.1, July 2019. Available at http://math.hws.edu/javanotes/

For Week 7 (Concurrency and Networking), the material consists of:

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 Inter*Actief* room.

For the security topic, the relevant material is:

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 line we refer to the Canvas site of the *2019-2020 Mathematics line*, with the exception of the special session in Week 4, which is about the use of Mathematics in Computer Science and Business Information Technology.

This manual has one chapter for each week of the module. These chapters contain the following information:

- Contents and relevant material;
- Overview of mandatory activities;
- Estimate of the required self-study effort, except for the activities of the Mathematics line;
- 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 D and P projects are described in a separate chapter that immediately follows this introductory chapter.

Additional material on testing is provided in Appendix A.

In the margins of the programming exercises, we sometimes use notes to refer to parts of the study material that should consult if you need help with these exercises. In these notes, ECK refers to David J. Eck. *Introduction to Programming Using Java*, and CJ refers to the chapters of the Core Java books by Horstmann and Cornell.

You should always read the text between the exercises in this manual! These pieces of text give you hints and instructions that will help you make the exercise. The teaching assistants expect you to have read these pieces of text, otherwise they may not help you.

Software Development Tools The following software development tools are used in this module:

- Eclipse (http://eclipse.org), which is a comprehensive tool for implementing and testing Java programs (amongst others).
- Visual Paradigm (http://www.visual-paradigm.com), which is a general tool for drawing UML diagrams. Visual Paradigm can be integrated with Eclipse as a plugin.
- EMMA (http://emma.sourceforge.net/, http://www.eclemma.org/), which is a standalone tool and an Eclipse plugin for test coverage.
- CheckStyle (http://eclipse-cs.sourceforge.net/), which is an Eclipse plugin for automatically checking programming code style.
- Metrics Eclipse plugin (http://metrics.sourceforge.net/), which allows some metrics to be extracted from programming code.

Information about the software tools necessary for the Mathematics line can be found on the Canvas site of the *2019-2020 Mathematics line*.

Tests and Grades

The following conditions have to be fulfilled:

- To be allowed to participate in the Design test, all mandatory Design lab exercises should be signed off; and
- To be allowed to participate in the Programming test, all mandatory Programming lab exercises should be signed off.

For the conditions related to the Mathematics line, we refer to the Canvas site of the *2019-2020 Mathematics line*.

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 Canvas;
  - Sign off all mandatory exercises during the lab sessions;
- Get a minimum of 5.5 for two of the tree tests (Mathematics, Design, and Programming) and a minimum of 5.0 for the third test;
- Get a minimum of 5.5 for the average of the three tests;
- Get 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 four students);
- D-test grade (individual);
- P-project grade (in groups of two students); and
- P-test grade (individual).

Table 1 gives an overview of the grading schema of this module.
The projects (Design and Programming) submitted before the first deadline and that do not require amendments will receive a bonus of 0.5 points. Projects that require amendments have to be submitted before the last (final) deadline and will not receive bonus points.

Extra Information about the D-tests and P-tests Instructions and example tests are available on Canvas. Both the D-test and P-test are open book. You are allowed to use the following materials:

- This manual;
- Slides of the lectures;
- Books specified as course material for the module, or copies of the required pages of these books);
- Simple calculator;
- Dictionary.

You may *not* use any of the following:

- Solutions of any exercises published on Canvas, such as recommended exercises or old tests;
- Your own material (copies of (your) code, solutions of lab assignments, notes of any kind, etc.).

Text in the material brought to the tests can be highlighted with a text marker, but hand-written notes in the manual, slides or book are not allowed. Violations of these rules can make your test invalid.

We are investigating whether the P-test can be performed on a laptop. More information about this will be provided during the module.

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.

Grading schema for Students from other study programmes (minor, premasters, etc.) If you are not following the CS or BIT Education Programme, your passing conditions and grading schema are:

- *Software Systems as Minor Module (12 EC)*:
  - Participate in all mandatory activities other than Mathematics;
  - Sign off all mandatory exercises (as above);
  - Get a minimum of 5.0 for each of the tests (Design and Programming);
  - Get a minimum average of 5.5 for the tests (Design and Programming); and
  - Get 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 have to pass 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 conditions and grading schema as above for the minor module apply, 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;
  - Get a minimum of 5.5 for the Design test; and
  - Get 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 four 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;
  - Get a minimum of 5.5 for the Programming test; and
  - Get 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 two students) and
  - P-test grade (individual).

Mandatory Activities

Presence (and active participation) is mandatory in the following activities of this module:

- All academic skills workshop sessions (weeks 2 to 5);
- Mathematics case session (week 4);
- All diagnostic tests (weeks 3, 6 and 7);
- The following programming sessions, which are related to the programming project:
  - Project session on global design in week 6,
  - Project session on the group protocol in week 7,
  - Peer feedback sessions in weeks 8, 9 and 10.

These activities are mandatory, but if personal (severe) circumstances have made you incapable of attending one of these activities, you should contact the module coordinator to discuss the situation and agree on an alternative.

Although not mandatory, your presence is expected in the Academic skills sessions in weeks 1 to 5, in which the mandatory Academic skills exercises are explained.

Signing off Mandatory Exercises

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

Academic skills The exercises that have to be handed in via Canvas are:

- A-1.1 Personal project goals and expectations for week 2
  Deadline: Sunday of week 1 at 23:59 CET
- A-2.1 List of tasks in your daily life
  Deadline: Sunday of week 2 at 23:59 CET
- A-3.1 Evaluation of the effectiveness of our time management plan *Deadline*: Sunday of week 3 at 23:59 CET
- A-4.1 Module 3 Time Management Plan (MTPM)
  Deadline: Sunday of week 4 at 23:59 CET

The other academic skills exercises will be discussed and signed off during the peer feedback sessions, so you are expected to bring them with you.

Design You do the exercises *in pairs*. An exercise marked with a ☞ symbol indicates that you should ask a teaching assistant for feedback when you have finished this exercise. If the exercise has been performed (nearly) satisfactorily, the assistant will sign off the exercise. If important aspects of your solution are missing or suboptimal, you will be asked to improve your design. If you come to the lab sessions well prepared (i.e., you have studied the contents of the lecture) it should be possible to finish all exercises during the session. However, if some exercises have not been signed off, you should 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 do not spend the lab sessions working on exercises from past sessions. Each lab session is dedicated to one particular topic, so you work on *that* topic (and nothing else) during the lab session, so that you prepare yourself properly for the design project.

Programming You do all exercises *in pairs*. An exercise marked with a ☞ symbol indicates *sign off points* that if you are finished with this exercise you can ask a teaching assistant to check your solutions of the preceding exercises (including the marked one). All exercises for week n should be signed off on 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.

Students are expected to work in the same pairs in both the programming and design lab sessions.

You are expected to sign off *all* exercises within the allocated time. If you fail to do so, you may not be allowed to perform the tests and consequently fail the whole module.

Teachers and Assistants

For questions about the contents, the exercises or the projects, you can always ask a teacher or teaching assistant during a lab session. For problems with group partners and personal problems, you should contact the housekeeper of the house to which you are assigned. The assignment of students to houses and the housekeepers will be published on Canvas.

The module coordinator is Luís Ferreira Pires. He can be reached by mail (l.ferreirapires@utwente.nl), and his office is Zilverling 2011. The contact details of the other teachers and student assistants and their individual responsibilities can be found on the Canvas site of this module.

Monitoring and Evaluation of the Module

If there are problems with the organisation, the programming environment or the manual, do not hesitate contact the responsible teachers and/or the module coordinator.

During the module, there will be two intermediate module evaluation sessions in weeks 3 and 7 (Tuesday during the lunch break). You will be invited to attend these sessions via Canvas.

Acknowledgements

Many people have contributed to this manual since its first version in 2013-2014. We acknowledge the valuable contribution of Kevin Alberts, Christoph Bockisch, Twan Coenraad, Frans van Dijk, Martijn Hoogesteger, Sophie Lathouwers, Renate van Luijk, Laurens Rouw, Jip Spel and Leon de Vries. Wim Kamerman deserves a special mention since he played a crucial role in the 2019-2020 revision of this manual. We also acknowledge the efforts of Arend Rensink, who set up the automated LaTeX-based environment that facilitates the editing of this manual, avoiding text duplication and keeping the pieces of code shown in the manual syntactically correct and executable.

Project Descriptions

This chapter describes the projects that you have to carry out in this module. 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. You are expected 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[2]. 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.

[2]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[3] activity diagrams describing relevant business processes (at least 2). [Lecture 1a]

[3] 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 (data model, 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 clearly indicate what the diagrams are about, e.g. `AD1-entry.png`. You don't have to submit the VP project(s) in which the diagrams were created.
- Merge all the text documents into a single PDF document `report-group⟨nr⟩.pdf`
- Package the report and the diagrams folder into a single zip file `d-project-group⟨nr⟩.zip`
- Submit it by means of the Canvas assignment.

The submission deadline is at the end of week 5. In order to encourage timely submission, groups that hand in the project by the end of week 5 will get a bonus of 0.5 points on the project grade. Groups whose project is graded insufficient and groups who miss the deadline will get one second chance. The ultimate, hard deadline (i.e. missing it causes you to fail the module) is by the end of week 10.

Important Dates

Week 5	Sun 23:59 CET	Submission deadline (0.5 bonus for timely submission; repairs are allowed if the grade is insufficient)
Week 10	Fri 23:59 CET	Submission deadline for repaired and late projects

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

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.

- 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).
- 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 in order to get at least 5.5 as final mark when the project is resubmitted in Week 10.

Programming project

Develop a distributed client/server program to play a board game, and describe the game design in a report. You are expected to perform this project assignment with a peer student in a *pair*. The pairs are published on Canvas. The purpose of the project is to demonstrate the design and programming skills you have acquired during this module.

The board game to be implemented in the programming project will be published on Canvas, as well as the rules of this game.

Global Description of the Game Application

The game application consists of a *client* program that directly interacts with each player, and a *server* program that controls the games. After starting the client, the user's name, IP address and port number of the server are asked for. After entering this information, the client connects to the server and waits for the server to signal that another client has also connected. When a second client connects to the server, a game can start, or the clients can decide to wait for more players if the game allows for more than two players.

After a game has started, the server should be ready for requests from new clients that would like to start a game. A server should therefore be able to support several games simultaneously.

A game normally proceeds as follows. The player who has the turn enters a move in accordance with the rules of the game. The client then checks the move, and if it is legal, it sends the move to the server. The server also checks if the move is legal, and this is the case it sends this information to all participating clients, who can then update their internal game state. The turn then moves to the next player, who again is expected to enter a legal move. This procedure goes on until the game is finished according to its rules.

Communication Protocol

Your client and server applications should be able to communicate with the respective applications of the other pairs in your tutorial group, which implies that all pairs of a tutorial group should use the same *protocol* for client/server communication. The protocol describes which data is exchanged between the client and the server, and in which order and format. Among other things, the data represent the moves in the game.

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

Functional Requirements of the Application

Your *server* must fulfil the following requirements:

1. When the server is started, a port number should be entered that the server will listen to.
2. If the port number is already in use, an appropriate error message is returned, and a new port number can be entered.

3. The server should be able to support multiple instances of the game being played simultaneously by different clients. To support this, the server must be multithreaded.
4. The server has a TUI that ensures that all communication messages are written to the console (`System.out`).
5. The server should respect the protocol as defined by the tutorial group during the project session in Week 7, *i.e.*, the server should be able to communicate with all other clients from other pairs in the tutorial group.

Your *client* must fulfil the following requirements:

1. The client should have a user-friendly TUI that provides options to the user (e.g., possibility to enter a port number and IP address) to request a game to the server.
2. The client should support multiple human players per game (up to the number that the game rules require), 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 that shows a possible move to a human player, as calculated by the computer player. The move may only be proposed, so that the human player has 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, e.g., closing all open connections.
8. The client should respect the protocol as defined for the pair of tutorial groups during the project session in Week 7, i.e., the client should be able to communicate with all the servers from the other pairs in the tutorial group.

Finally, the following *global requirements* should be fulfilled

1. The client and the server should synchronise their game state.
2. All user input should be validated and handled accordingly (for example, check if the specified port is a number. If it is not, ask again).
3. Your implementation should make use of the *Model-View-Controller* pattern.
4. 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.

Checkstyle Warnings You should download the checkstyle configuration from Canvas (go to the TOOLS INSTALLATION SESSION 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.

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.

Report Requirements

The report must be written in readable English. The *target audience* of this report is a *software maintainer*, i.e., somebody who did not necessarily write the code himself, but at a later occasion will have to extend or improve it. Therefore, this person should be able to understand the overall design of your application, understand the purpose and functionality of each class, be able to check whether the application passes at least the existing tests, etc.

Discussion of the Overall Design Your report should discuss the overall design of the application (global structure of the system), in terms of the classes and their relationships. The overall design should consist of the following:

1. Class diagrams, with explanations of the global design and the roles and responsibilities of each package and class. Make sure your diagram layout reflects the structure of your application, e.g., by repositioning classes and removing unnecessary details. Feel free to add extra labels and notes. 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 an association[4].
2. A systematic overview of which functional requirements are implemented by which classes and methods. If you did not manage to implement all requirements, this should be mentioned here.
3. A description of how you implemented the *Model-View-Controller* pattern: indicate which classes and packages play the role of model, view and controller in your application design.

Testing Testing is an integral part of the development of an application. Therefore, in the report you are expected to discuss both unit tests and system tests. Also, if you discovered errors in your implementation during testing and had no time to fix these errors, you should mention them in this part of the report. Mention at least one discovered and fixed bug and include the test cases of this bug.

See also See Appendix A

Unit Testing. All complex (non-trivial) methods should be tested with (at least) one of the following methods, if possible:

- Using dedicated test classes, preferably using JUNIT, as you have learned in this module. This is the most thorough testing method for unit tests.
- Simulating a part of the system manually via `telnet`, to test the communication over the 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.

For each class, the report contains the following information concerning the unit tests:

1. Strategy used to test the class (in isolation, together with other classes, not at all);
2. Test method that has been applied (see above);
3. If applicable, test programs that have been developed;
4. Test results and expected results;
5. Test coverage percentage of each method to be determined using the Emma plugin. In case of low coverage, discuss reasons and consequences for this class.

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

System testing. System tests are used to assess the functionality of the application as a whole; i.e., a system test will typically consist of a complete run-through of a game. The functional requirements should serve as the basis for these tests, since the implementation should fulfil these requirements. Each requirement may be tested separately by creating one or several test executions with different usage scenarios (of both expected and incorrect usage). Your system tests should also be described in the report, indicating which aspects of the system have been tested and how. Appendix A.2 contains an example of how to document a system test.

Metrics report The report should contain a (brief) section that contains the values of the software metrics you have learned about during this module. Include an analysis of the consequences for the quality of the code[5].

[4] *Programming-only* project pairs are not required to precisely follow the UML syntax. However, we still expect some graphical representation of your class structure, with the classes and its relationships.

[5] *Programming-only* project pairs can ignore this requirement. However, a remark about this should be made in the report in order to avoid confusion.

Reflection on planning In Week 7, you are asked to make a planning for the project, and to discuss this with a teaching 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. 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 project tasks that took significantly more or less work than you had expected? If so, why did this happen?
   - 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 do you explain this?
3. Which countermeasures did you take to compensate for deviations 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 to the other questions into account and ask yourself how you would want to prevent this or deal with this next time.
5. Suppose that next year you are a teaching assistant for this project. Give at least two do's and don'ts that you would tell your students to help them with their planning.

Possible Extensions of the Application

If you have sufficient time, you can extend your game according to the suggestions below. Implementing more extensions makes your project more advanced and will result in a better grade. For each extension, you can get a maximum number of points. *Extensions will only be graded if the application without extensions is sufficient for a pass, i.e., if it is graded with at least a 5.5.*

Some of the extensions may require the server and the communication protocol to be extended or modified. You should make sure your client and server still correctly work with the servers and clients from different pairs, respectively after the extended functionality is implemented.

Chatbox (max +0.3 points)

A game application implemented as described above can only be used to play the game. It would be cool if we had the possibility of communicating with your opponents during the game. Therefore, a possible extension is to extend the server and client UI to allow players to send short pieces of text to the other players.

Challenge (max +0.3 points)

According to our initial description, players play against each other after their clients connect to the server. An interesting extension would be to allow a player to choose other players to play against from a set of registered players. To achieve this, the following changes are expected to be necessary:

- A player should know which other players are registered;
- A player should be able to choose its opponents;
- A player should be able to refuse a game.

Leaderboard (max +0.3 points)

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 of your system should be negatively influenced by this extension, which implies that you have to choose a representation that suits your implementation.

Security (max +0.4 points)

As soon as a leaderboard is in place, players might want to cheat to get to the highest place. Also, simply accepting any connection makes servers vulnerable to Denial-of-Service attacks. One way to prevent 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).

GUI (max +0.4 points)

In addition to the TUI you can build a GUI for your game. Make sure your GUI scales (i.e., it keeps its appearance when it is scaled) and is properly tested.

Packaging for Submission

The implementation must be handed in as a *single ZIP archive* file via Canvas. This file must include your *report as a single PDF file* and a *ZIP archive of your implementation code*.

You can generate your implementation archive in ECLIPSE using the EXPORT... item from the FILE menu, selecting ARCHIVE FILE from the GENERAL category, selecting your project and pressing finish. The ZIP archive with your implementation should contain the following:

- All source files of the self-defined classes, stored in a single directory hierarchy.
- Documentation of all self-defined classes (HTML pages generated with JavaDoc) in a directory hierarchy that is separated from the source files.
- Any non-standard predefined classes and libraries, to be included as `jar`-files.
- A `README` file, located in the *root folder*, containing:
  - Information about installation, indicating, for example, which directories and files are necessary and which conditions apply for installation
  - Steps on how to start the game, including example start commands if applicable.

Before submitting this package, make sure your program compiles without problems with the files that you will submit!

Warning: Typical issues that make the installation and compilation procedure fail are names and paths or hardcoded URLs. *Test this before submitting your project!* If your program contains references to the file system, e.g., to load image files, make sure these references are platform-independent. It is possible that some user will run your application under a different operating system than the one you used to development the application. For further information, see the documentation for `java.lang.File` and in particular the constants `pathSeparator` and `separator` defined in this class.

Grading

You will receive one grade for the project, which is derived from the grading criteria listed on page 18. The *early bird submission deadline* is on Wednesday (23:59) in Week 10. In order to encourage timely submission, pairs that hand in the project before this early bird deadline will get a bonus of 0.5 points on the total project grade. Pairs that miss the early bird deadline must hand in their submission latest on Friday (23:59) in Week 10. This is a *hard, final deadline*, i.e., missing it causes you to fail the module.

Rules for Bonus Points

There are several ways to earn bonus points, which are 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 groups is maintained and kept up-to-date by one programming project pair or by a single student. If this task is fulfilled properly, each student in the pair gets a bonus of 0.5 points. If an individual student maintains the protocol, this student gets the 0.5 bonus points individually.
- The students in the pair that wins the tournament of a tutorial group get 1.0 bonus points each, and the students in the runner-up pair get 0.5 bonus points each.
- Various project extensions are possible, for which you can earn bonus points. More information can be found on page 16.

These bonus points are added to the overall project result.

Grading Criteria

The grading criteria that will be used to grade the project assignment are described in the sequel. You can apply these criteria to judge your own progress. There are two grading scales, depending on the criteria: *fail/pass* and *fail/pass/good/excellent*. For the latter scale, you will fail if you do not meet the criteria mentioned at *pass*, and the *good* grade will be assigned if your implementation falls in between the *pass* and *excellent* criteria.

You will get a 5.5 for the project assignment if you have successfully included or implemented all *pass* criteria that are indicated below with *pass required*. You can improve your score by implementing all criteria mentioned below as *excellent*, up to a maximum score of 10.0. You will fail the project if one or more of the criteria indicated with *pass required* are not met.

Packaging

- Your submission is handed in via Canvas and according to the requirements defined in "Packaging for Submission" on page 17. Grading scale: fail/pass. *Pass required.*

Global Design

1. The overall design (UML & Code). Grading scale: fail/pass/good/excellent
   - *Pass:* (a) The implementation code is generally structured according to the *Model-View-Controller* pattern. In the report you elaborated on how you implemented the *Model-View-Controller* pattern. If certain functionality is not separated as it should, a description of the necessary changes is included in the report. (b) The program is divided into classes and packages in a logical way. (c) A class diagram for all model classes is included and described in the report. *Pass required.*
   - *Excellent:* The implementation fully adheres to the *Model-View-Controller* pattern, and concrete classes are shielded by interfaces whenever applicable. The report elaborates on all specific design choices, and includes (not automatically generated) UML class diagrams for all classes. All functional requirements are defined and elaborated in the report.
2. Class Specifications. Grading scale: fail/pass/good/excellent
   - Pass: The three most complex classes are fully specified with preconditions and postconditions, as well as (class) invariants. To determine the complexity of classes, you used the Eclipse Metrics tool and in particular the provided metric "Weighted Methods per Class" (WMC). *Pass required.*
   - Excellent: All classes and all methods are fully specified with pre- and postconditions, as well as (class) invariants.

Programming

1. Code Quality. Grading scale: fail/pass/good/excellent

  - *Pass:* Names of classes, variables and methods follow the conventions prescribed in the module (checked with the Checkstyle plugin) and have intelligible names. Constants are used where applicable to make the software easier to maintain. The code is neat: not unnecessarily complicated, no excessive duplicate code is present and no code is unused. The coverage of the project is calculated and explained in the report. You will pass if most of these conditions are largely met. *Pass required.*
  - *Excellent:* All of the above conditions are fully met. All software metrics you have learned about during this module are calculated, analysed and explained in the report.

2. Functional Requirements. Grading scale: fail/pass/good/excellent
   - *Pass:* A functioning multi-threaded application in which multiple games can be played simultaneously with your client/server implementation over the network. Your implementation adheres to the basic rules in the protocol. We specifically take the quality of the networking, multithreading and input validation into account. *Pass required.*
   - *Excellent:* Your implementation fully adheres to the set protocol, handles all networking exceptions well with custom exceptions and contains all configuration options mentioned in the functional requirements.
3. Game Rules. Grading scale: fail/pass/good/excellent
   - *Pass:* Your implementation globally complies with the game rules. *Pass required.*
   - *Excellent:* Your implementation fully complies with all game rules.
4. Documentation. Grading scale: fail/pass/good/excellent
   - *Pass:* Most classes, methods and tests are documented with minimal useful JavaDoc tags (description + parameter definition). *Pass required.*
   - *Excellent:* All classes, methods and tests are documented with extensive and detailed JavaDoc. Complex parts of code are documented with inline comments.

See also Appendix A on testing

Testing

1. Unit tests. Grading scale: fail/pass/good/excellent
   - *Pass:* The most complex methods of the three most complex classes are properly tested with JUnit. The most complex methods have 100% coverage. The chosen classes and test coverage are mentioned and justified in your report. *Pass required.*
   - *Excellent:* All complex (non-trivial) methods that are tested with JUNIT tests, `telnet`, `main` methods or other unit testing methods, whenever possible. The tests and their coverage are discussed and justified in the report.
2. System Tests. Grading scale: fail/pass/good/excellent
   - *Pass:* The report describes how five functional requirements have been tested as discussed in Appendix A, or any similar reference. Examples of functional requirements are *"after a game is finished, the user can start a new game"* and *"a user should be able to input another port if a specified port is invalid"* from the *"All user input should be validated"* functional requirement. *Pass required.*
   - *Excellent:* All implemented functional requirements are system tested and described in the report as mentioned above.

Reflection on Planning

1. Your reflection on planning includes all elements as described in the Report Requirements on page 14. Grading scale: fail/pass. *Pass required.*

Report

1. *Structure:* The reader can quickly understand what the text is about, the text is logically structured and all necessary information can be easily found. Grading scale: fail/pass. *Pass required.*
2. *Writing:* The language in the report is understandable for the target group, spelling is correct and consistent and sentences are well-constructed. Grading scale: fail/pass. *Pass required.*

Extensions

1. The chat functionality is correctly implemented and documented. (+max 0.3)

2. The challenge functionality is correctly implemented and documented. (+max 0.3)
3. The leaderboard functionality is correctly implemented and documented. (+max 0.3)
4. The security functionality is correctly implemented and documented. (+max 0.4)
5. A GUI is correctly implemented and documented. (+max 0.4)

Project Activities

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 with a teaching assistant, 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 this 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.

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 teaching assistant, who might give some suggestions on how to adjust your planning, in case your planning is unrealistic. Remember that the goal of the project is to create a working program and an appropriate report. Teaching assistants have done this project during their first year as well, so their feedback should be taken into account.

During the last weeks of the module, the teaching assistants can help you 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 sessions on the project for each week, you will find more detailed guidelines for the peer feedback. The pairing for the peer feedback will be announced via Canvas.

Apply what you learned about giving feedback so far in the Academic skills line. Try to judge all components, and where appropriate, provide additional details to support your judgement. We strongly recommend that you discuss your feedback face-to-face.

Do not forget 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 in which the different computer players will compete against each other. Bonus points can be gained by winning the tournament (see the Rules for Bonus Points section on page 18).

Important Dates

Week	Day	Hour	Activity
Week 6	Tue	8	Kick-off lecture on the programming project
Week 6	Wed	3–4	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 8	Wed	6–7	Tutorial group meeting to discuss written-down version of the protocol
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 0.5 bonus
Week 10	Fri	23:59 CET	Final submission deadline

Week 1

Week 1

1.1 Overview

Getting Started with the Tools For this module, all students should have a working tool environment. For this purpose, on the very first day of the module there is a special tool installation session (see Canvas).

Academic Skills This week, there is a 1-hour workshop dedicated to academic skills. The general content and structure of the sessions devoted to academic skills will be presented. We will also talk about the platform that we will use to visualize the theoretical content that must be studied before each work session. Finally, a series of short questionnaires will be offered to facilitate the task to be developed in the next session.

Design The activities in this week cover the following topics

- Project: getting started, see Section 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 Section 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 Section 1.3.4 for exercises in Lab session 2a.

Programming This week the following topics will be covered:

- Setting up and using your tool environment.
- Values and variables.
- Control flow.
- Simple interactive programs with textual input/output.

1.1.1 Mandatory Presence

During the following activities, your presence is mandatory.

- Tool installation session (Mon 6–7)

1.1.2 Expected Self-Study and Project Work

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

- 1.5 hours to make the Academic skills exercises;
- 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: See Canvas ("Course Materials").

Design: Slides of lectures 1a and 1b

Programming: ECK, Chapters 1–3

Laboratory: The following predefined files (for both Programming and Design) are provided on Canvas.

- `ss/week1/CountDivisors.java`
- `ss/week1/Dlab-1a-intro.vpp`
- `ss/week1/Dlab-1b-incident-reporting.vpp`
- `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`

Additionally, to be used in other weeks as a generic library:

- `ss/utils/TextIO.java`

1.1.4 Tool Installation Session

A document with installation instructions can be found on Canvas.

You are kindly requested to download the tools as described in the document *before* the session. If everybody tries to download everything that is necessary at the same time during the session, chances are that the wireless network will get clogged.

We strongly suggest that you perform Exercises D-1.1 and P-1.1 – P-1.2 in order to check if your installation works properly.

1.2 Academic Skills

1.2.1 Assignments

A-1.1 Writing personal project goals and expectations for week 2

Next week we will write a group contract, a tool that will allow us to configure a series of rules that will facilitate intercommunication between group members. To begin preparing this document it is necessary that we know ourselves, our desires, ambitions, expectations and limitations. Your objective will be to write a list of the goals and expectations related to your design project in this module. To do this, in Canvas you will find a template with a series of key questions that will help you write this list

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

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 Visual Paradigm.

In the lab files for this week on Canvas, you will find the file `Dlab-1a-intro.vpp`. Open this file with Visual Paradigm (VP). You will find a incomplete version of the activity diagram shown in Figure 1.1. Add the missing branch (called *Decision Node* in VP), activity (called *Action* in VP), and merge (*Merge node* (found under Decision node) in VP), and add/adapt the control flows so that it is equivalent to Figure 1.1.
Please note: VP offers the option to add a condition to the decision node (so that you could annote the outgoing control flows with "yes" and "no"). We use the standard UML notation where the conditions are given as annotations to outgoing arrows.

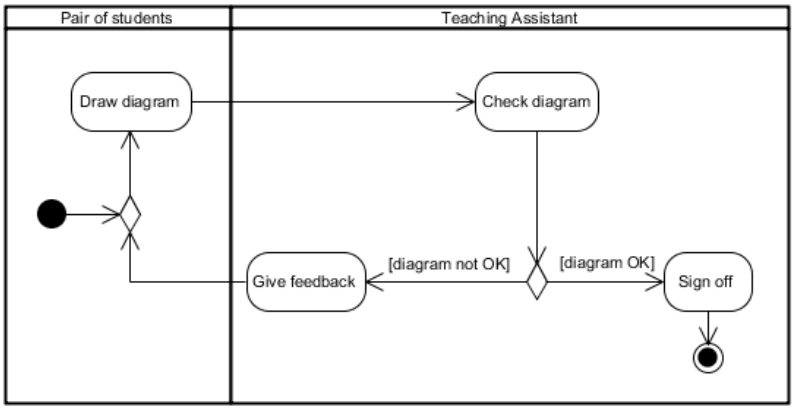


Figure 1.1: Activity Diagram for introductory exercise

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 several years ago. Procedures could be different now.

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 (*Optional*) 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 exercises 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

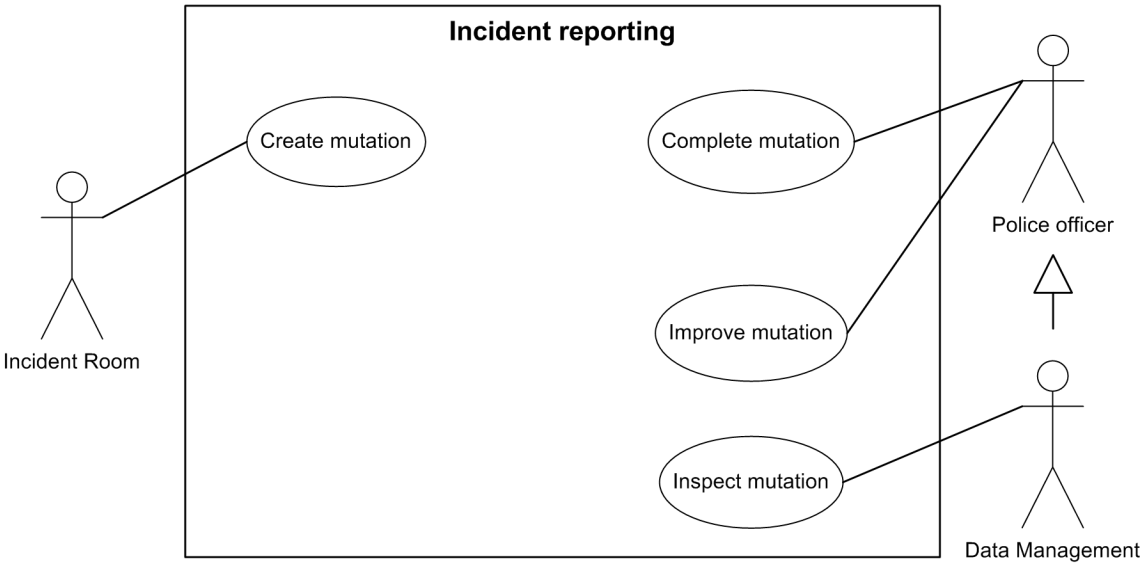


Figure 1.2: Use Case Diagram voor incident reporting (Exercise D-1.7)

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 Visual Paradigm project `Dlab-1b-incident-reporting.vpp` in this week's lab files on Canvas. Please extend it with the following information, making use of ≪include≫ and ≪extend≫ relations between use cases.

- If someone wants 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.

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).

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.

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)

	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)

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 Canvas.

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.

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

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)

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.

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.

	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)

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)

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 exercises 1b (Use Cases)

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

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, 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.

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)

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

Hello World

The following exercises are intended to familiarise you with the process 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).

See also ECK Sect. 2.6.2 for information on ECLIPSE

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

P-1.1 For this assignment, create a new *working directory* on your system called, e.g., `c:\softwaresystems\java` in a Windows system. This is the place in which you can create your JAVA source files from now on. Preferably this should be an empty directory not used for any other stuff.

- Within your working directory, create a subdirectory `ss`, and inside this directory another subdirectory `week1`.
- Inside that directory, create a file called `Hello.java`
- Edit the file with a plain text editor (e.g., Notepad or Notepad++ for Windows, and TextEdit or BBEdit for the MacOS), and type in the following:

```
package ss.week1;

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

- Open a command-line window and go to your working directory.
- *Compile* your program by typing[2]

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

- Inspect the subdirectory `ss\week1` of your working directory. What happened?

See also ECK Sect. 2.6.6 and 4.5.2 for Java packages

In Exercise P-1.1, class `Hello` is defined in package `ss.week1` (see first line of the given code). This implies that the compiler expects to find the `.java` file in directory `ss\week1`, and the compilation will fail if the file is not in this directory.

P-1.2 In the previous exercise, you compiled `Hello.java`, and now you will run the compiled program.

- In a command line window at your working directory, type the command:

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

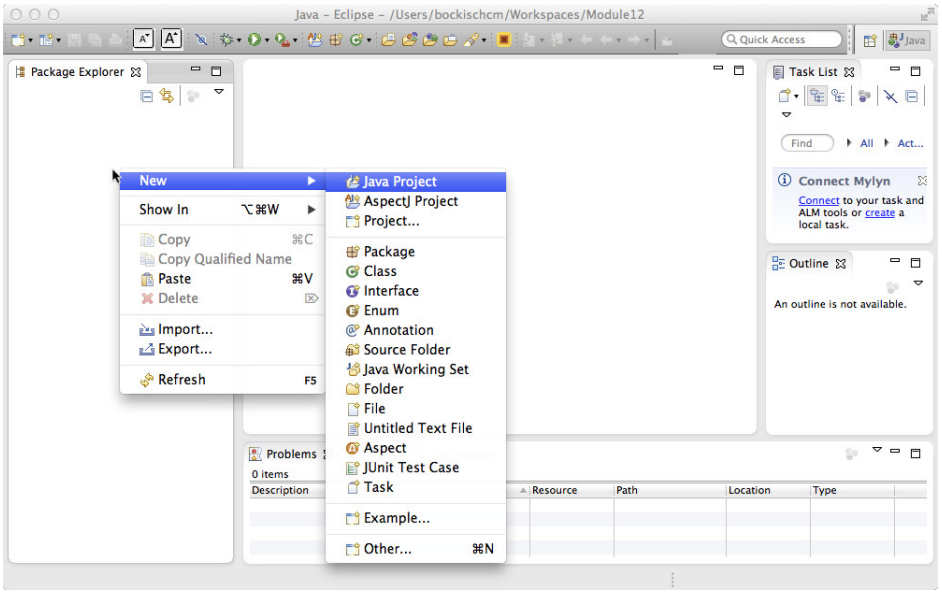
  What happened?
- Suppose that you actually want to print *Hello Software Systems student* on the screen. What do you have to do to make this happen?

Instead of calling `javac` and `java` from a command line window, in the remainder of this course we will use the ECLIPSE IDE. ECLIPSE makes compiling and running code a lot easier. More information about ECLIPSE is available online at http://help.eclipse.org. The most relevant information can be found there in the JAVA DEVELOPMENT USER GUIDE, which you can reach from the list of contents on the left-hand side of the ECLIPSE HELP.

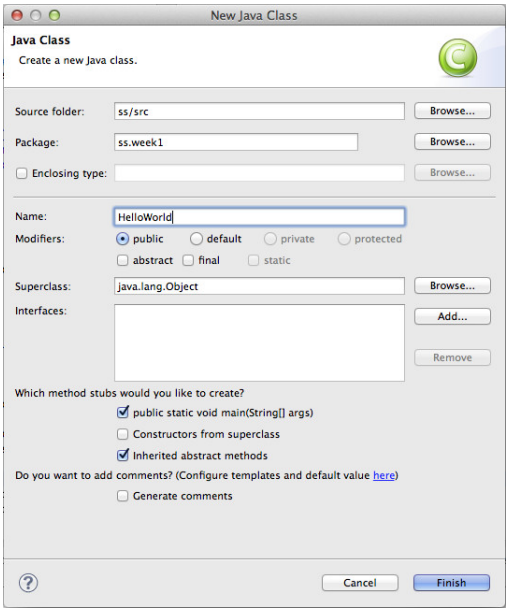
P-1.3 Start ECLIPSE and take the following steps:

- On the left of the window, you should see the PACKAGE EXPLORER. Right-click on the PACKAGE EXPLORER to select it, choose NEW in the context menu, and then choose JAVA PROJECT.

[2] On a Linux or MacOS system, backslashes to separate directory names and file (or other directory) names, as in `ss\week1\Hello.java`, become slashes, as in `ss/week1/Hello.java`.



- Enter a project name, e.g., `softwaresystems`, and click on 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 and 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, select the package `ss.week1`, and add a new CLASS to it. Call this class `HelloWorld`. Indicate that you would like to create a method stub (method declaration with an empty body) for `public static void main(String[] args)`.



- Copy the contents of the `main` method of the `Hello.java` you created in Exercise P-1.1 to the `HelloWorld` file. Save your file (via the FILE menu or using CTRL-S).
- Now you will run the class by selecting RUN in the RUN menu. The output of the program execution will be printed in the ECLIPSE CONSOLE view.

P-1.4 Now try to introduce some small errors in your `HelloWorld`. For example, change the package name to `ss.week2`, or remove the `;` after a statement. What happens? What happens when you hover over the error indicators (the little red cross in the margin, or the squiggly line below a piece of program)?

Importing files from Canvas into ECLIPSE

On Canvas, a number of auxiliary classes and tests are available every week, which should be used during the exercises. The instructions below can be used to import these files into ECLIPSE:

1. Download the ZIP file with the predefined files from Canvas, and remember its location.
2. In ECLIPSE, select FILE → IMPORT.
3. In the dialogue that now opens, choose GENERAL → ARCHIVE FILE and then press NEXT.
4. Press BROWSE next to the input field FROM ARCHIVE FILE, and select the downloaded ZIP file.
5. Press BROWSE next to the input field INTO FOLDER, and select the folder `src` in your project.
6. Finally, click the FINISH button.

You will be asked to repeat this procedure every week from now on, so you can do that now for week 1. If you succeeded in the above, you can now find the source code for some classes that you will need to do the exercises of this week in subdirectories of your `src` folder, like, for example, the `TextIO` class in subdirectory `ss/utils` and class `CountDivisors` in subdirectory `ss/week1`.

Important: Follow the instructions of the Javadoc section that can be found at the end of the 'Tools Installation session' page on Canvas. After completing these instructions, your documentation will be properly generated during the module.

Values and Variables

Below you are asked to perform a couple of exercises from ECK in order to learn how values and variables can be used in Java.

P-1.5 (ECK *Exercise 2.2)*

Write a program `RollTheDice.java` in package `ss.week1`[3] that simulates rolling a pair of dice and prints the results. You can simulate rolling one die by choosing one of the integers 1, 2, 3, 4, 5, or 6 at random. The number you pick represents the number on the dice after it is rolled. The expression

See also ECK Sect. 2.3.1 for Math functions

```
(int)(Math.random()*6) + 1
```

does the computation to select a random integer between 1 and 6. Assign this random integer to a variable to represent one of the dice that are being rolled. Do this twice and add the results to get the result. Your program should report the number showing on each die as well as the final result. For example:

```
The first die comes up 3
The second die comes up 5
Your final result is 8
```

P-1.6 (ECK *Exercise 2.5)*

If you have `n` eggs, then you have `n/12` dozen eggs, with `n % 12` eggs left over. This is essentially the definition of the `/` and `%` JAVA operators for integer values. Write a program `ss.week1.GrossAndDozens.java` that asks the user how many eggs she has and then tells her how many dozens of eggs she has and how many extra eggs are left over.

Hint: To ask the user and read the answer (an integer value) in your program you can use the methods `System.out.println()` and `TextIO.getlnInt()` (from ECK). Don't forget to import the `TextIO` class to your program by adding the following line just below the package declaration:

See also ECK Sect. 2.4.1 for text output and 2.4.2 for text input

```
import ss.utils.TextIO;
```

A gross of eggs is equal to 144 eggs. Extend your program so that it tells the user how many gross, how many dozen, and how many left over eggs she has. For example, if the user says that she has 1342 eggs, then your program would respond with

[3] From now on we will indicate the package by prefixing it to the name of the class. In this case, the class (program) would be called `ss.week1.RollTheDice.java`.

```
Your number of eggs is 9 gross, 3 dozen, and 10
```

since 1342 is equal to `9*144 + 3*12 + 10`.

Control flows

Below you are asked to perform a couple of exercises to learn how to define control flows in Java.

P-1.7 (ECK *Exercise 3.2)*
Which integer between 1 and 10000 has the largest number of divisors, and how many divisors does it have? Write a program `ss.week1.MostDivisors.java` to find the answers and print out the results. It is possible that several integers in this range have the same, maximum number of divisors. Your program only has to print out the smallest of them.

Hints: In `ss.week1.CountDivisors.java`, you will find the source code of the divisors example of ECK Section 3.4.2. You can use this code as starting point for this exercise. To find a maximum value, the basic idea is to go through all the integers, storing in local variables in your program both the largest number of divisors that you have seen *so far* and the integer with the most number of divisors, respectively.

P-1.8 (ECK *Exercise 3.6)*
Exercise P-1.7 asked you to find the number in the range 1 to 10000 that has the largest number of divisors. You only had to print the smallest of such an integer. Copy the contents of `ss.week1.MostDivisors.java` to a new class, `ss.week1.MostDivisorsWithArray.java`, and modify the program in such a way that it prints *all* numbers that have the maximum number of divisors. Use an array with a length of 10000 and try storing the number of divisors of each integer number from 1 to 10000 in this array. At the end of the program, you can go through the array and print out all the numbers that have the maximum count. The output from the program should look like this:

```
Among integers between 1 and 10000,
The maximum number of divisors was 64
Numbers with that many divisors include:
   7560
   9240
```

☞ **P-1.9** The Babylonian algorithm to compute the square root of a positive number n is defined as follows:

1. Make an initial guess of the answer (you can pick $n/2$ as your initial guess).
2. Compute `r = n / guess`.
3. Set `guess = (guess + r) / 2`.
4. Go back to step 2 for as many iterations as necessary. The more you repeat steps 2 and 3 the closer `guess` will become to the square root of n.

Write a program `ss.week1.BabylonianAlgorithm.java` that asks the user to input a `double` value using method `getDouble()` from `TextIO`, assigns it to n and iterates through the Babylonian algorithm until the last two guess values are within 1% of each other. Output all guesses as a `double` with two decimal places by using the `String.format` method.

See also ECK Sect. 2.4.1 for text output and 2.4.2 for text input

Three-Way Lamp

Now you will implement 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. The lamp is therefore either *off*, *low*, *medium* or *high*.

P-1.10 Write a program `ss.week1.ThreeWayLamp.java` that repeatedly asks the user for input to change the state of the lamp. You can do this by using the method `getln()` of `TextIO` to get a line of textual input, and assigning the contents of this line to a `String` variable. This input should then be checked to

determine whether it is valid, the state of the lamp is changed if necessary, and some feedback is given to the user.

The following input options are offered to the user (as `String` values):

- **OFF**: Set the lamp to OFF *(default value)*
- **LOW**: Set the lamp to LOW
- **MEDIUM**: Set the lamp to MEDIUM
- **HIGH**: Set the lamp to HIGH
- **STATE**: Print the current setting of the lamp
- **NEXT**: Change to the next setting, observing the order OFF → LOW → MEDIUM → HIGH → OFF
- **HELP**: Show a help menu, explaining how the user should interact with the program
- **EXIT**: Quit the program

See also ECK Sect. 2.3.4 for **enum**

In your implementation, you are expected to use an enumerated type (**enum**) to define the possible states of the lamp.

The expected structure of the `main()` method body is represented by the following pseudo code:

```
1:  Print menu;
2:  exit ← false;
3:  while ¬exit do
4:      input ← input string from stdin
5:      if input = OFF then
6:          Set light to off;
7:      else if input = LOW then
8:          Set light to low;
9:      else if input = MEDIUM then
10:         Set light to medium;
11:     else if input = HIGH then
12:         Set light to high;
13:     else if input = STATE then
14:         Print state;
15:     else if input = NEXT then
16:         Test state value and go to the next;      ▷ Could be implemented as a method (subroutine)
17:     else if input = HELP then
18:         Print menu;
19:     else if input = EXIT then
20:         exit ← true;
21:     else
22:         Print error message;
23:     end if
24: end while
```

See also ECK Sect. 3.6.2 for examples of input menus and the **switch** to handle options

Instead of nested **if**s you are asked to use the **switch** statement to check the input values and act accordingly.

Simple Hotel Application

In the first weeks of this module you will perform a number of exercises where you develop parts of a hotel application step by step.

P-1.11 The first exercise is to print a bill for the guests. We start very simple, so the bill has only one item with a fixed description that you define in your program as a `String` and a fixed **double** value, as for example:

```
Hotel night 1x     85.00
```

Write a program that prints this bill containing a single line. Create a new package `ss.week1.hotel` and a new class `ss.week1.hotel.SimpleBillPrinter.java`. In this class:

1. Declare a `String` variable that will contain the formatted bill `String` to be printed out.
2. Declare a `String` variable to contain the description and assign a `String` value to it (e.g., `"Hotel␣night␣1x"`).
3. Declare a **`double`** value to contain the amount of money associated to this item, and assign a **`double`** value to it (e.g., `85.00`).
4. Correctly format the contents of the first `String` you defined by using `String.format` or `System.out.printf`, so that everything is aligned like in the example. In other words, the price should be right-aligned with precisely two decimals.
5. If this haven't been done yet, print the `String` variable that contains the formatted line.

See also ECK Sect. 2.4.1 for formatting `String` values

☞ **P-1.12** Use class `TextIO` to write a textual user interface `ss.week1.hotel.HotelTUI`. The idea is that the hotel staff can use your program to do the hotel administration using the keyboard, i.e., it should be possible to check a guest in or out.

Since you do not have a hotel with rooms and guests yet (you will develop this in week 2) we use a simplified hotel with just one room and zero or one guest. The guest will be represented by a `String` value that can either be **`null`** (no guest checked in the room) or the name of the guest. You will also need two more `String` variables, to store the name of the hotel and the name of the room, respectively.

An example execution would be the following.

```
Welcome to the Hotel booking system of the U Parkhotel
Commands:
 i name ............... check in guest with name
 o name ............... check out guest with name
 r name ............... request room of guest
 h ..................... help (this menu)
 p ..................... print state
 x ..................... exit

Command: i Richard
Guest Richard gets room 101
Command: o Richard
Command: r Richard
Guest Richard doesn't have a room
Command: x
```

Take the following into account to implement this program:

- Use constants to define the command characters (e.g. `i`, `o`, `r`) in your program. Constants can be defined by declaring them with **`static final`**. For example, a constant to represent the *checkin* command with value `'i'` would be defined as **`static final char`** `IN = 'i';`. By convention, constants are defined in all caps.
- Remember that Strings can be null! Check this in your code to avoid execution errors.

The expected structure of the `main()` method body is represented by the following pseudo code:

```
1: Print menu;
2: exit ← false
3: while ¬exit do
4:     input ← line from stdin;
5:     split[] ← line split into words;
6:     command ← split[0];
7:     parm ← null
```

```
8:    if split.length > 1 then
9:        parm ← split[1];
10:   end if
11:   if command has one letter then
12:       if command = IN then
13:           if parm = null then
14:               Print error message;
15:           else if Room not taken then
16:               guest ← par;
17:           end if
18:       else if command = OUT then
19:           if parm = null then
20:               Print error message;
21:           else if Room not taken by given guest then
22:               Print error message;
23:           else
24:               guest ← null;
25:           end if
26:       else if command = ROOM then
27:           if par = null then
28:               Print error message;
29:           else if Guest not found then
30:               Print error message;
31:           else
32:               Print room information;
33:           end if
34:       else if command = PRINT then
35:           Print hotel information;
36:       else if command = HELP then
37:           Print menu;
38:       else if command = EXIT then
39:           exit ← true
40:       else
41:           Print error message;
42:       end if
43:   else
44:       Print error message;
45:   end if
46: end while
```

Similarly to Exercise P-1.10, instead of nested **`if`**s you are asked to use the **`switch`** statement to check the input values and act accordingly.

To perform the step in line 5, use method `String.split()` (from class `String`) to split the input line into an array using spaces as delimiters.[4]

1.4.2 Recommended exercises

P-1.13 Make the following exercises from ECK:

- Exercises 1.4, 1.6 and 1.7 (pages 68 and 69).
- Exercises 3.3 and 3.4 (page 129).

[4] Line `String [] split = input.split("\\s+");` does this trick by copying to array `split` the parts of string `input` that are separated by spaces. If you Google "separate string spaces java" you can find an explanation for this.

Week 2

Week 2

2.1 Overview

2.1.1 Contents of This Week

Academic Skills This week there is one workshop dedicated to academic skills. We will work in groups to create a draft of a group contract, so all the members of your group have to attend the work session. We will focus on synchronizing the team's work moments, the roles that each one will take, the time slots that will be allocated to the work as well as the goals and expectations of the group in their performance. We will also spend time defining what our ideal behaviours should be for teamwork and what should be the consequences of repeatedly breaching contract rules. At the end of the session, the draft contract is expected to be completed on time, and it will be reviewed and signed by the group during the week as part of an assignment.

Design The design activities in this week cover the following topics

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

Programming The following topics will be discussed this week:

- Classes and objects.
- Program by contract (invariants, preconditions and postconditions).
- Testing: unit testing, test plan and test framework.

2.1.2 Mandatory Presence

There are no mandatory activities this week.

2.1.3 Expected Self-Study and Project Work

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

- 2.5 hours to make the Academic skills exercises;
- 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
Set of videos proposed in Edpuzzle.

Design
Slides from lectures 2a and 2b.

Programming

Lecture ECK, Chapter 4–5.4
Laboratory The following predefined files are provided on Canvas:

- `ss/week2/Student.java`
- `ss/week2/hotel/Room.java`
- `ss/week2/hotel/Guest.html`
- `ss/week2/test/GuestTest.java`
- `ss/week2/test/HotelTest.java`
- `ss/week2/test/RoomTest.java`
- `ss/week2/test/DollarsAndCentsCounterTest.java`
- `ss/week2/Dlab-2b-lendBooks.vpp`

Tools EMMA Eclipse plugin

2.2 Academic Skills

2.2.1 Assignments

A-2.1 List of tasks in your daily life. Next week we will work on the construction of a time scheme that will allow us to have a better management of our tasks and the energy used in each of them. The objective is to optimize time management, where we will try to minimize stress situations and maximize the performance of your performance during this module. In order to be able to carry this out, it will be necessary again that you do introspection to know better every aspect of our lives that need to be managed. We ask you to write a list of each and every one of the tasks that you carry out in your life and that take you more than 10 minutes. For this we ask you to use the template that we will provide you on Canvas. Once you are sure that you have not forgotten to write any task, we ask that you assign an amount of time to each of these tasks (how long would it take you to carry out this particular task?). The estimation of time is of most importance, so we ask you to try to be as precise as possible, avoiding being too optimistic or pessimistic (for example: it will take at least 30 minutes to brush my teeth and about 15 minutes to finish the exercises of this week).

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

2.3 Design

2.3.1 Laboratory session 2a (Class Diagrams)

The laboratory session is done in groups of two students.

☞ **D-2.1** Make a class diagram for the following case description.

Outdoor Holiday Tours (OHT) is a travel company specialized in outdoor group travels. It started as an initiative of a small group of tour guides who organized their own tours, with a bare minimum of

administration. But due to enthusiastic posts on social media they get more and more participants, and the program is expanding. This calls for a more professional organization. One of the things OHT needs is a proper booking system. You are asked to help making a design for this system.

OHT organizes hiking tours in different parts of the world. They also do canoe tours, but these take place only in Europe. Accommodation during a tour is in simple guest houses or tents provided by OHT. The length of a tour ranges from a few days to a few weeks. Some tours are more demanding than others. The difficulty of a tour is indicated by a number of footprints ranging from one footprint (very easy) to seven footprints (very challenging). The full program with information about all tours is shown on the website. Some tours are quite popular and are offered multiple times per year. In particular the canoe tours in France enjoy high demand.

Prospective participants can make a booking on the website (OHT consistently speaks about participants, rather than customers; active participation in the group is key to making a tour successful). *However, we'll disregard bookings for now and deal with that in the next exercise.* It may happen—fortunately not very often—that OHT has to cancel a tour when the minimum number of participants has not been reached.

A tour has a name; description; number of footprints; maximum ascent per day (only for hiking tours, not for canoe tours); minimum and maximum number of participants; number of days; start date; price per person.
The price per person for a tour can vary; typically the same tour is a bit more expensive in the peak season. When a tour has been cancelled, this is also indicated in the system.
For canoe tours OHT locally hires canoes. From a canoe rental company the name, address, telephone number, and name of the owner are known.
It is possible that OHT has rental contracts with multiple companies for one tour. E.g. the Rivers of Aquitane Tour includes canoeing on the rivers Lot and Dordogne; the canoes are rented locally with a different company. It is also possible that OHT has different rental contracts with one company for different tours. E.g. there is a Basic Ardeche Tour and an Advanced Ardeche Tour, for which different contracts have been made with one company. A rental contract has a start date, end date, and a price per canoe for the whole tour.

☞ **D-2.2** Extend the class diagram of D-2.1 with the following information.

Each tour has a tour guide. For a tour guide is known: name; address; gender; SSN (social security number). It is possible, however, that a tour is in the system (and thus can be booked) while no guide has been assigned yet.

Prospective participants can make a booking on the website. A booking can be made for one person or for a small group of up to five persons (friends or family going together). A booking through the website is provisional, in the sense that your booking will be deleted if you don't pay an advance payment in time. The advance payment, a certain percentage of the tour price, should be paid within ten days. If not, the (provisional) booking will be cancelled automatically. Sometime before the start of the travel, a final payment is due.

Each booking has a unique booking number. For each booking it is known how much money has been paid so far. When a booking is made, the following information about a participant is stored: name; address; gender; birth date; diet; telephone number; e-mail address. However, if the booking is for more than one person, telephone number and e-mail address are given only for the contact person, not for the other participants in the same booking. The attribute diet states special dietary requirements of the participant (if any).

2.3.2 Recommended exercises 2a (Class Diagrams)

Class diagrams are the most challenging type of UML diagram. To allow for enough training, there are two recommended exercises.

D-2.3 Make a class diagram for the following case description
(The size and difficulty of this exercise is comparable to what you can expect in a test)

CD rental

The CD Rental was founded in the 1980ies by students of the UT (then: THT) and at the time located on campus. Currently, it is located on the second floor of the Enschede public library in the town centre. Although the use of CDs has declined a lot over the years, there is still a group of, primarily elderly, people for whom this is the preferred medium for listening to music.

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 ("Normal members"). In addition to that, The CD Rental collaborates with the *Overijsselse Biblitoheekdienst* (collective of libraries in the province of Overijssel). Any person who is a member of any library of the OBD can rent CDs at the CD Rental. ("External members").

Only normal members can reserve CDs. CD Rental staff will set them 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.
For the information system the following points could be relevant.

- For each CD 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).
  - Item code: unique identification, e.g. "CP44661". Every item carries a bar code label with the item code.
  - "Three-letter code": usually the first three letters of the name of the artist. (For example "DEL" for Ilse DeLange).
  - 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.

- 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.

- Renting a CD works as follows. The member 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.
  CDs are rented for three weeks. For late returns, an additional daily fee is charged *(but 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 inter-library 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 scanned as usual and 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 for one member to make a second reservation for a CD which they currently have reserved already). Reservations are processed in the order in which they arrive.

- CDs can be returned at the CD Rental desk or, if the CD was rented through *BookFinder*, by the OBD transportation service. Returned CDs are scanned so that the system knows they are available again. If a returned CD has been reserved, it will be set apart.

- Data about rentals remain stored in the system after the CDs have been returned. These include the customer to whom the CD was rented, the date and time it was rented, the date and time it was returned (N.B. the *time* attribute includes the date, so there is no need for a separate *date* attribute. 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.

- The following data about membership need to be stored:

  – For normal member, the system should know: name; address; e-mail address; membership passes (multiple passes are possible, see below); start date of membership; date when membership will expire.

  – For an external member (member of an OBD library who rents CDs directly from the CD rental at the counter), only pass number (possibly multiple pass numbers) and e-mail address are stored.

  – A member can have multiple passes. (Sometimes passes get lost, in which case the member gets a new pass and the old one is blocked. However, the old pass number remains stored in the administration.) 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).

  – 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-2.4 Make a class diagram for the following case description.
(The generalizations in this exercise are more complicated than what you can expect in a 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 Canvas 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.

☞ **D-2.5** Figure 2.1 shows the sequence diagram for lending books to a customer of the library, which was discussed extensively in the lecture. It is available in `Dlab-2b-lendBooks.vpp`in this week's lab files. Extend the sequence diagram to incorporate the following additional information:

- Some books are not lendable, i.e., they can be read in the library, but you cannot take them home. An attempt to lend out a book that is not lendable will fail, the system displays that the book is not lendable
- If books are returned too late, a small fee is due. Preferably this is paid immedately when the books are returned. But sometimes customers have no money with them, in which case it can be paid later. However, if the debt has reached a certain threshold (currently € 10) no books should be lent to this customer. (Note: you don't need to add options for payment, this is not part of the lending process, but consider the possibility that lending to this customer will be refused.)

The next exercise (D-2.6) is related to the following case study.

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.

Care centre "The Westwolds" in Barchester has adopted 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. *Section 1.3.5 (recommended exercise 1b) describes a pilot with automatic medication dispensers. Here we will only be concerned with the proper medication for clients of The Westwolds, not how the medication is distributed.*

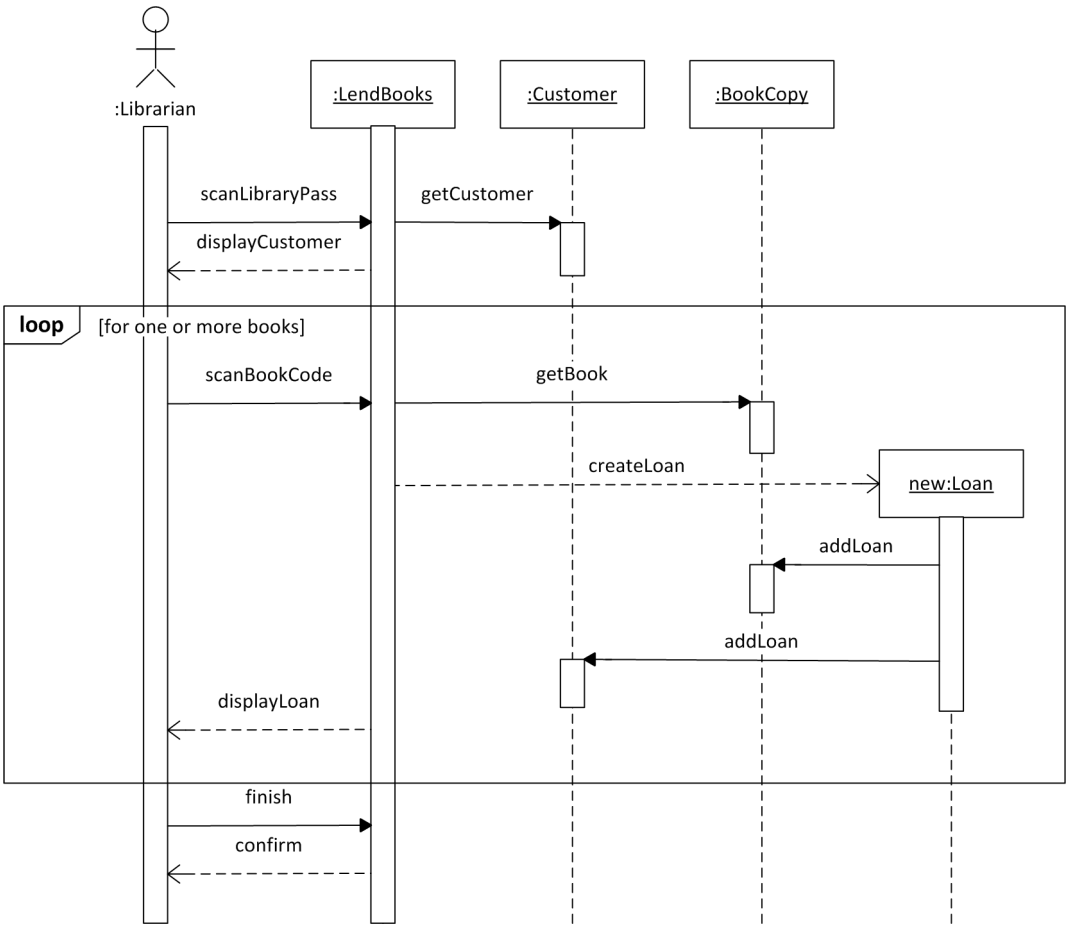


Figure 2.1: Lending library books (Exercise D-2.5)

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. For each client who makes use of this service, a so-called service plan has been set by a nurse, indicating wich medication should be taken, in which dose, and when (maximum three times per day).
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.

Figure 2.2 shows part of a use case diagram for the information system used by The Westwolds. Figure 2.3 shows part of a class diagram for this system.

Changing a service plan
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 number of times per day and/or 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 a client is visited for 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.
Finally, the service is activated.

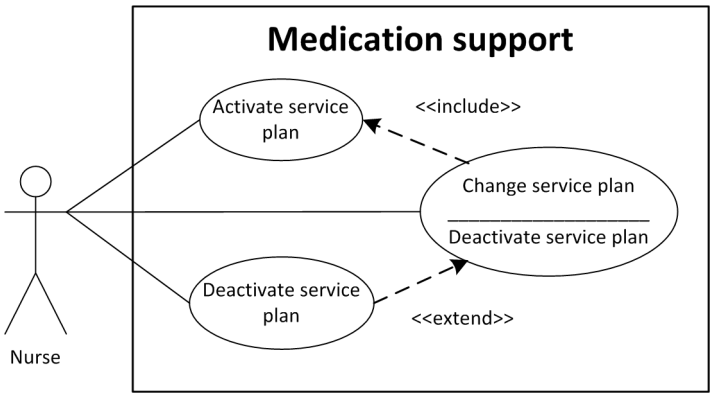


Figure 2.2: Partial Use Case Diagram for medication support (Exercise D-2.6)

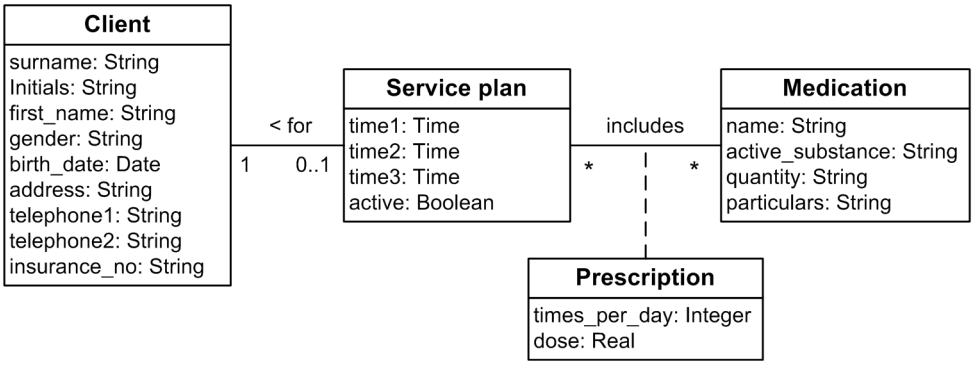


Figure 2.3: Partial Class Diagram for medication support (Exercise D-2.6)

The main structure of the sequence diagram has been elaborated in 2.4. The control object is not represented in the diagram: in the model the actor interacts directly with the relevant objects. For the essential part of the sequence diagram there is a reference to another sequence diagram *Change services*.

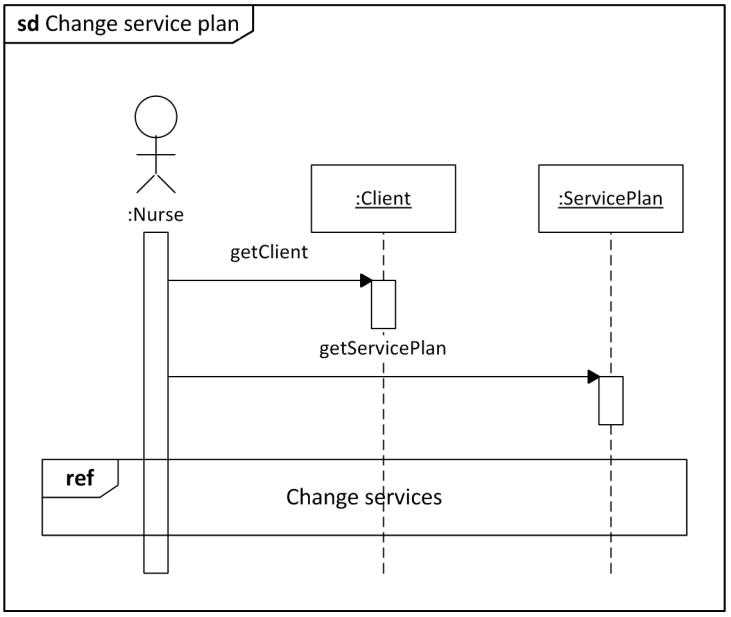


Figure 2.4: Sequence Diagram for *Change serviceplan* (without control object) (Exercise D-2.6)

☞ **D-2.6** Make a sequence diagram for *Change services* that fits to the top-level design in Figure 2.4.

2.3.5 Recommended exercises 2b (Sequence Diagrams)

D-2.7 Make a sequence diagram for booking a tour with Outdoor Holiday Tours (see Section 2.3.1 for the case description). To limit the amount of drawing, you don't have to include a control object (and, as a consequence, you don't have to draw return messages).

Making a booking involves the following steps:

- Selecting a tour;
- Creating a new booking for that tour;
- Adding the participants with the requested data for each participant;
- Confirming the booking when it has been completed.

Please note:

- If a booking is made for multiple participants, contact details are needed only for the first participant (the contact person), not for the other participants.
- Some participants have special dietary requirements due to an allergy or according to their religious background. where applicable, these can be added.

D-2.8 You are asked to make a sequence diagram for the following case decription.

Applying for admission to the University

For students from the Netherlands, there are standard procedures to enroll in a university program. If you come from another country it is more complicated – an admission board has to decide whether an applicants fulfil the prerequisites, based on the evidence that the applicants can provide of their previous education. The number of foreign students has increased over the last few years. A new system is needed to support the handling of these applications.

Any person who wants to apply for a study program at the university can file an application through the website. The first step is starting an application for admission. This will provide you with login details

that you can use to edit the application. A lot of information and documents have to be collected and uploaded; applicants usually do multiple edits. As a final editing step, when everything is complete, you can submit the application, changing its state from 'draft' to 'submitted'. Submitted applications will go to the admission office, which make sure that the application is handled by the admission board of the program and eventually informs the applicant about the decision that has been taken.

Requested is a sequence diagram for editing an application for admission to the University by the applicant. In order to edit an application you have to open it first. We can distinguish two cases: starting a new application or opening a previously created application. This is shown in Figure 2.5. (*In either case starting/opening the application will need some interaction with the web server to authenticate the user, which we disregard for convenience*).

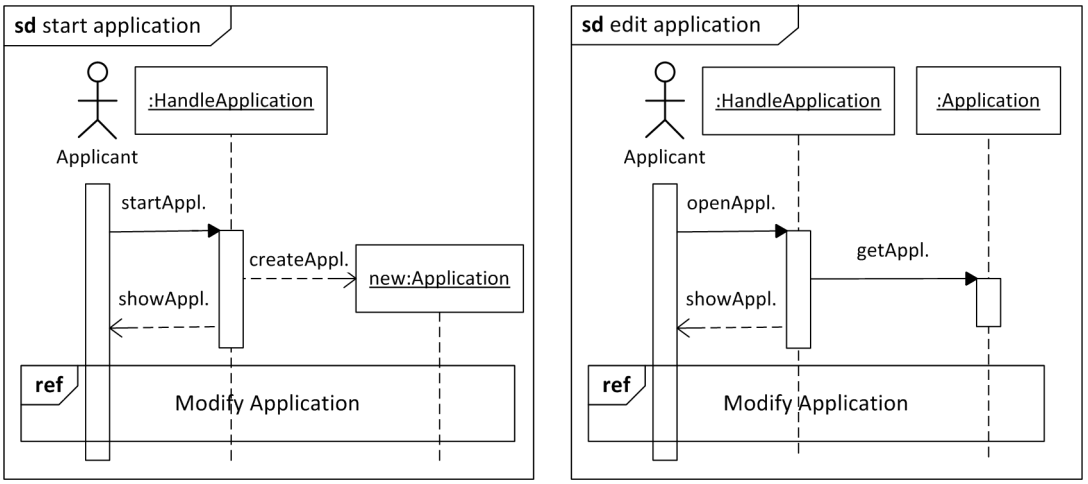


Figure 2.5: Creating/editing an application for university admission (Exercise D-2.8)

You are asked to give a sequence diagram for *Modify Application*. Modifying the application could involve the following steps:

- editing one or more information fields in the application;
- adding a PDF document as attachment to the application;
- submitting the application (i.e. modifying its status).

During an editing session, the first two steps can be done any number of times, in any order. One can also stop and come back at another time (re-invoking *edit application*) However, when the application has been submitted, no further editing is possible.

2.4 Programming

2.4.1 Laboratory exercises

Import the contents of this week's lab files from Canvas to your ECLIPSE project by following the instructions mentioned on page 35.

If you succeeded in the above, you can now find the source code for the class `Room` as well as the documentation for class `Guest` in package `ss.week2.hotel`. Additionally, in package `ss.week2.test` you can find several test classes for this week's exercises.

Important: ECLIPSE will show compilation errors in the classes that you have just imported. This is because the predefined classes make references to the classes that you will develop during this lab session. After completing the assignment for class `DollarsAndCentsCounter` in Exercise P-2.8, for instance, the compilation errors in `DollarsAndCentsCounterTest` should be gone.

P-2.1 In this exercise you will extend the code of the given class `ss.week2.Student`. The instance variable *score* of type **`int`** represents a student's test score. Write a new method `passed` that returns **`true`** if the student's score is 70 or above, and returns **`false`** otherwise. Do you need a conditional (**`if`**) statement to write this method?

Object-oriented Three-Way Lamp

Last week you implemented a `ThreeWayLamp` program by filling in the `main` method with all the intended functionality. This week you will adapt this program to make it object-oriented.

P-2.2 You will define a JAVA class to model your three-way lamp. What query (or queries) should the class support? Which commands?

P-2.3 Create a new JAVA class `ss.week2.ThreeWayLamp` and write a *stub implementation* for this class. A stub implementation contains all methods of the class, but with minimal bodies, only to make sure the code compiles without errors.

Work on the code until there are no compilation errors in the `ThreeWayLamp` class. This means that any method that has a return type that is not **`void`** should contain a **`return`** instruction with an appropriate value. 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`**.

The method bodies in these cases should only contain these **`return`** statements, while a method with a **`void`** return type should be empty.

See also Appendix A

P-2.4 Complete the specification of the `ThreeWayLamp` class by defining invariants, preconditions and postconditions to your stub implementation as comments. You can define invariants, preconditions and postconditions in the JavaDoc documentation by using the `@invariant`, `@requires` and `@ensures` tags, respectively. Define these conditions in plain English or pseudo code, and try to be as precise as possible.

Now add assertions (**`assert`** statements) to the method bodies that check whether the preconditions of these methods are satisfied.

In this specification you have to explicitly define that after OFF the lamp goes to LOW, after LOW it goes to MEDIUM, etc.

See also Appendix A

P-2.5 Write a test for the `ThreeWayLamp` class based on the conditions you defined in Exercise P-2.4. Create a class `ThreeWayLampTest` in package `ss.week2.test` that implements your tests. Your class should test the following cases:

- If after being created the lamp is `OFF`;
- If the sequence `OFF` → `LOW` → `MEDIUM` → `HIGH` → `OFF` is properly implemented.

The `ThreeWayLampTest` class should have the following elements:

- A (private) variable of type ThreeWayLamp to hold an object to be tested.
- A `setUp` method that creates the ThreeWayLamp object to be tested.
- A method for each of the two test cases above;
- A `runTest` method that calls `setUp` and the methods of the test cases.

The `main` method of the `ThreeWayLampTest` class should create a `ThreeWayLampTest` object and call its `runTest` method.

☞ **P-2.6** Implement the methods that you specified in Exercise P-2.4 by filling in the body the methods in the stub implementation with the actual intended functionality of the class. The result should compile without errors and it should pass the `ThreeWayLampTest` tests you wrote in Exercise P-2.5.

P-2.7 Finally you will now combine your efforts of last week (Exercise P-1.10) with your newly created `ThreeWayLamp` class. Create a new class `ss.week2.ThreeWayLampTUI` and copy the contents of `ss.week1.ThreeWayLamp.java` to this class. Edit the `ThreeWayLampTUI` in such a way that it uses your newly created `ss.week2.ThreeWayLamp` class, so remove the pieces of code that are covered by your `ss.week2.ThreeWayLamp` class. In its `main` method, class `ThreeWayLampTUI` should create a `ThreeWayLamp` object and call the methods of this object whenever an input option is chosen by the end user.

Hint: Create a separate method that prints the help menu, and remove references to the **`enum`** you declared last week in the `ss.week1.ThreeWayLamp` class.

Dollars and cents counter

In the next exercise you will use the predefined class `DollarsAndCentsCounterTest`. When inspecting this class, you will notice that it does not have a `main` method, which means that it cannot run as a standalone Java application. Instead, this is a so-called JUNIT *test class*: all methods preceded with the tag `@Test` are so-called *test methods*. In ECLIPSE, you can run this in a similar way as an application class (i.e., select the class in question, then select Menu RUN → item RUN, or CTRL-F11 as a shortcut), however, the result will be different: instead of output in the console view, you will get a new JUNIT view with a progress bar that colours green if all tests methods finish correctly, or red if there are errors in your code. If there are errors, then by selecting them you can find out where in the code these errors occurred.

P-2.8 Implement and test a dollars and cents counter. The counter should provide the following functionality:

`public int` `dollars()`
The dollar count.
@ensures a return value that is bigger or equal to 0

`public int` `cents()`
The cents count.
@ensures a return value in the range of 0 to 99

`public void` `add(`**`int`** `dollars, `**`int`** `cents)`
Adds the specified dollars and cents to this *Counter*.

`public void` `reset()`
Reset this *Counter* to 0.
@ensures this *Counter* is set to 0 dollars and 0 cents

Implement class `ss.week2.DollarsAndCentsCounter`. In the files you imported from Canvas you will find the test class `ss.week2.test.DollarsAndCentsCounterTest`. Run this class and improve your implementation until it passes all test cases.

Inspecting the JAVA `String` class

P-2.9 Using a web browser, access the Java API documentation for the predefined class *String* at https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html. Scan the class definition and answer the following questions:

1. In which package is the `String` class contained?
2. How many constructors does this class have? How do they differ?
3. Several methods can be used to construct new `String` instances, which are often called *factory methods* because they 'produce' objects. How many methods are available that build `String` instances?

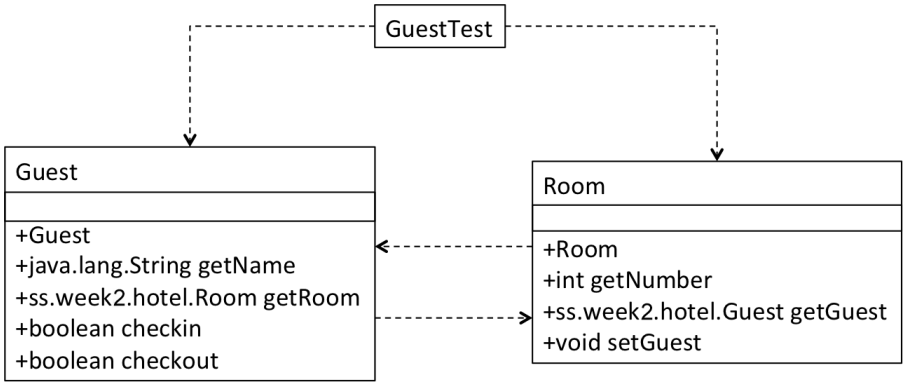


Figure 2.6: Class Diagram for Hotel Application

Object-oriented Hotel

Last week you implemented a nice user-friendly TUI for your hotel. From this week on, you will develop various parts of the hotel, starting with the `Guest` class.

P-2.10 Create a class `Guest` in package `ss.week2.hotel`, and add a stub implementation of all methods of this class listed in Figure 2.6. Use the documentation `Guest.html` and class `Room.java` to find out which method parameters are necessary. Improve your stub implementation until there are no compilation errors. Once this is done properly, the predefined classes `GuestTest` and `Room` should not give any compilation errors.

P-2.11 In ECLIPSE, run the `ss.week2.test.GuestTest` JUNIT tests that you downloaded from Canvas (see Exercise P-2.8) in order to test your implementation. Explain what happens.

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

P-2.12 Add JavaDoc comments to your class `Guest` that describe 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`, `@return`, `@param`, `@requires` and `@ensures` to make useful and original documentation.

To generate the HTML files from JavaDoc comments, first select your project in the PACKAGE EXPLORER. Next, select PROJECT from the menu and then GENERATE JAVADOC. In the dialogue that opens after that, 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.

Make sure you have followed the instructions of the Javadoc section of the 'Tools installation session' page on Canvas. In this way the documentation will be properly generated.

P-2.13 Generate JavaDoc documentation for your project. Once the documentation is generated, you can open the file `doc/index.html` to start browsing the documentation. Which information is included in the JavaDoc documentation? Explain why JavaDoc can be useful.

P-2.14 Implement your class `Guest` by defining method bodies that comply with the method descriptions.

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`. The current `Guest` is the receiver of the current method, *i.e.*, the object represented with `this`.

Improve your implementation until it passes the tests performed by the predefined `GuestTest` class.

Before your program passed all tests, probably it still contained many errors. You may have noticed that the runtime error information provided by the Java Virtual Machine (JVM) can be difficult to understand, because it contains some strange numbers that refer to the internal representation of `Guest` and `Room` objects, which is only intelligible for the JVM.

P-2.15 Intentionally insert an error in your implementation of `Guest`, for instance, by omitting the assignment to `room` in `checkin`, and study the error message generated by `GuestTest`. You should see something like

```
java.lang.AssertionError:
     expected:<ss.week2.hotel.Room@5cdd8682> but was:<null>
```

See also ECK Sect. 5.3.2 for method `toString`

Therefore, it is advisable to define more informative textual descriptions of objects. This could be, for example, the name of the guest or a room number, which can be defined in the string representation of these objects in method `Object.toString()` (string representation of an `Object`).

P-2.16 Add a method **`public`** `String 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). Now check if the error message became more informative.

You will now further extend your hotel application.

P-2.17 Each room of our hotel has a *safe*, which guests can rent upon request. Each safe can be open or closed. Additionally, each safe can be (de)activated, and only an active safe can be opened. This safe seems kind of useless now, but next week you will extend it with a protective password to make it more interesting.

Implement a class `ss.week2.hotel.Safe` that has two (private) instance variables to keep information about whether the safe is open or closed, and activated or deactivated, respectively, and proper (public) methods to query and modify these instance variables (see below). Because this class is relatively simple, you will implement it directly, i.e., without writing a specification and a testing class.

The class should contain the following commands:

- `activate`: without parameters, activates the safe;
- `deactivate`: without parameters, closes the safe and deactivates it;
- `open`: without parameters, opens the safe if it is active;
- `close`: without parameters, closes the safe (but does not change its active/inactive status).

and the following queries:

- `isActive`: returns **`true`** if the safe is active, **`false`** otherwise;
- `isOpen`: returns **`true`** if the safe is open, **`false`** otherwise.

Additionally, define and implement the *default constructor* of this class (constructor with no parameters), with proper default (start) values for the class instance variables.

Now you will assign a `Safe` to a `Room` and test the `Room`.

☞ **P-2.18** Add an instance variable of type `Safe` to the class `Room`. This instance variable should be initialised in the constructor of `Room`, and an appropriate query should be defined to get it.

Give class `Room` *two* constructors: one with two parameters, **`int`** `number` and `Safe safe` (used to initialise the instance variable `safe`), and one with a single parameter `number` that creates a new `Safe`. The latter can call the former, by using as first (and only) line:

```
this(number, new Safe());
```

Add a test case to `week2.test.RoomTest` (provided on Canvas) that checks that the room in its initial state contains the `Safe` that was passed to the constructor. Run the test and improve until it passes.

Now you are going to specify and implement a simple class `Hotel` by combining all these classes. For simplicity, assume that the hotel has 2 rooms and that guests can check in by using their name. Also assume that different guests have different names.

P-2.19 Specify a class `ss.week2.hotel.Hotel` with the following functionality:

- A command `checkIn` that receives one `String` object as parameter, indicating the name of the guest. The method returns a `Room` object with a (new) `Guest` of the given name checked in, or **`null`** in case there is already a guest with this name or the hotel is full.
- A command `checkOut` that receives the name of a guest as a parameter. The guest is checked out, and the safe in the room is deactivated. Nothing happens if there is no guest with this name.
- A query `getFreeRoom` that returns the `Room` into which the guest can checked in, or `null` if there is no free room available.
- A query `getRoom` that receives the name of a guest as parameter, returning the `Room` object into which the guest has checked in, or `null` if the guest cannot be found in any room.
- 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.

Complete your specification by adding class invariants, preconditions and postconditions as comments, and `assert` statements to check preconditions in the method bodies.

Additionally, the hotel should have an instance variable `name` with an appropriate query. This instance variable is set when the object is initialised. Extend the class diagram of Exercise P-2.10 with the new classes; the class diagram should show all classes in the package `ss.week2.hotel`.

Class `ss.week2.test.HotelTest` (available on Canvas) contains JUNIT tests that can be used to test your `Hotel` class.

☞ **P-2.20** Implement the class `Hotel` as you specified above. Improve your implementation until it passes tests of `ss.week2.test.HotelTest` without errors.

Hotel TUI Integration

Now you will finally connect your `HotelTUI` from last week with your newly created `Hotel` class.

☞ **P-2.21** **Copy** the `HotelTUI` implementation from package `ss.week1.hotel` to `ss.week2.hotel`.

- In this exercise, you will not run the TUI in static context in the `main()` method. Instead, create two new methods in class `HotelTUI`, namely **`public void`** `start()` and **`public void`** `printHelpMenu()`, and a constructor that has only the name of the hotel as argument.
- Move all code from the `main()` method to the `start()` method.
- Change the main method to create a new `HotelTUI` object and start it with the statement `(new HotelTUI("U␣Parkhotel")).start();`
- Remove all locally declared variables that represented the attributes of a hotel in week 1 (name of guest, room, hotel, etc.).
- Declare a variable of type `Hotel`, initialise it in the `HotelTUI` constructor and update all switch cases to work with this `Hotel` object.
- Implement the `printHelpMenu()` method.

Make sure the user is adequately informed about the result of each action by means of feedback given on the standard output. For example, when `hotel.checkIn()` is called, let the user know whether the check in succeeded or not.

2.4.2 Recommended exercises

P-2.22 Make the following exercises from ECK:

- Exercises 5.1, 5.2 and 5.3.

Week 3

Week 3

3.1 Overview

3.1.1 Contents of This Week

Academic Skills This week there is one workshop dedicated to academic skills. Is everything you planned to do equally (and really) important? How to prioritize things when everything seems a priority? During the first half of the session we will focus our attention on using the Eisenhower matrix to understand how to prioritize the tasks in our lives. Then, we will begin to configure our personal time management plan (PTMP) based on all the information that we have previously collected. This plan includes the task and time management of a full week of the module.

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 Section 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 Section 3.3.4 for the exercises for Lab Session 3b.
- Project work, continued from last week, see Section 3.3.3;

Programming This week the following topics will be discussed:

- Interfaces and inheritance. These concepts are related with the notion of *generalisation* discussed in the Design thread.
- Subtyping and method overriding.
- Introduction to Security Engineering: threat modelling and mitigation approaches.

3.1.2 Mandatory Presence

During the following activities, your presence is mandatory.

- (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:

- 2 hours to make the Academic skills exercises;
- 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: Set of videos proposed in Edpuzzle.

Design

Slides from lectures 3a and 3b

Programming

Lectures ECK, Sections 5.5–5.8

Laboratory The following predefined files are provided on Canvas:

- `ss/week3/bill/Bill.html`
- `ss/week3/bill/Bill.Item.html`
- `ss/week3/password/BasicPassword.html`
- `ss/week3/test/BasicPasswordTest.java`
- `ss/week3/test/HotelTest.java`
- `ss/week3/test/PricedRoomTest.java`
- `ss/week3/test/PricedSafeTest.java`

3.2 Academic Skills

3.2.1 Assignment

A-3.1 Evaluating the effectiveness of our time management plan. Do you remember the time management plan we made in this week's session? It is now time to test it and see how your planning fits your real life. Were all the tasks of this week conducted as planned or have things come up that you had not initially thought about? Those that were planned, could be carried out in the stipulated time or did you have to make changes? We ask you to extend your PTMP document with this information day by day during this week so that you learn what your true work rhythm is. Use the example we provide on Canvas to carry it out.

You have to submit your answer at the latest on Sunday of Week 3 at 23:59 CET.

3.3 Design

3.3.1 Laboratory session 3a (State Machine Diagrams)

The laboratory session is done in groups of two students.

Repair service of an internet shop (first version)

RedHot Ltd. Is an internet shop for consumer electronics, such as tablets and smartphones. It claims to distinguish itself from similar shops by its superb customer service. If you have a problem with any product purchased from RedHot you can call them for advice, and you'll be connected to a knowledgeable person within a minute or two. E-mails are answered within a few hours. You can also visit them physically in their Rotterdam shop, where service staff can help you select the best choice before you buy something.
In this case study we disregard most of their services and focus only on the repair service.

Any article that you bought at RedHot can be sent back for repair when it breaks down. If a customer wants to return an article for repair, they call the RedHot Customer Service. The Customer Service employee inquires what the problem is and creates a service record for this article in the system, containing a brief description of the problem. The customer than (automatically) gets an e-mail with instructions. As an attachment, a label to a freepost number (i.e. postal charges paid by receiver) is included, which the customer should use to send a parcel to RedHot. The label also carries a bar code linking parcel to the service record, so that RedHot knows what to do with it when it arrives.

When an article arrives at RedHot, its label is scanned by the Technical Service department; the arrival date and time is recorded in the system. Technical Service will first make an estimate of the repair costs. A Technical Service employee enters the cost estimate into the system, possibly with a brief report to explain what the estimate is based on. The customer automatically gets an e-mail with the estimate and the report, asking the customer to give permission for the repair (by clicking a link in the e-mail) or to cancel the repair (by another link).

When the customer gives permission for repair, this includes giving permission to RedHot to debit the repair costs to the customer's bank account or credit card. The Technical Service department of RedHot will then carry out the repair. If the customer does not give permission for repair, the article will be returned to the customer.

In principle, every repair should be finished within a week upon arrival. If for whatever reason the repair takes longer, after a week RedHot will send an e-mail with apologies an a status update. When the repair is completed, a Technical Service employee finalizes the repair by filing a repair report and indicating the costs of the repair. The costs that will be charged (automatically) are never higher than the given estimate (even though the real costs could be, at the expense of RedHot)—but it could be lower if the repair was easier than anticipated.

Finally the repaired article is sent back to the customer. The date and time that the article is sent (more precisely; the date and time that the address label for the return parcel is printed) is stored.

Three weeks after the repair, the RedHot system sends a questionnaire to the customer, asking wether the customer is satisfied with the service. RedHot claims to give excellent service, therefore it is important to monitor how the customers appreciate it.

In the unfortunate case that a repaired product breaks down again, the same process is followed a second time.

☞ **D-3.1** Make a state machine for the state of an article.

Hints:

- First, make a list of relevant events (i.e., use cases that will cause a change in the state of the article).
- Please note that some transitions change the state of an article (as it gets repaired during the process), while other transitions only change the location of the article (not part of the repair itself). For a systematic approach it is useful to include both aspects in the description of a state (for example: article at RedHot, repaired).
- Please be aware that the state machine describes *the state of an article in RedHot's administration system*, not what happens with the article in real life. It could happen, for example, that an article stops working and the owner throws it away. This is not reported to RedHot, not registered in the system, and therefore does not lead to a change of state.

Repair service of an internet shop (extended version)

Some details, left out in the first version, still need to be included.

- Warrantry. Most articles come with a warrantry period of a few years. When an article breaks down during the warrantry period, repairs are free of charge. Therefore, if warrantry applies the customer does not have to give permission and there is no need to make a cost estimate first.
- Repair by the supplier. In some cases, RedHot does not carry out the repair itself, but sends it back to the supplier of the article. In the process as it is presented to the customer it does not make any difference. RedHot makes an estimate of the costs. When the customer gives permission for repair, the article is sent to the supplier (which of course is registered in RedHot's administration). When the repaired article arrives back from the supplier it is registered as repaired and then sent back to the client as usual.

☞ **D-3.2** Extend the state machine with the new details.
The extended state machine should have a transition "warrantry expires". Please be aware that this can happen at any moment in time, also during a repair. (However, the repair is still free of charge if the article was reported broken before the warrantry expired).

3.3.2 Recommended exercises 3a (State Machine Diagrams)

D-3.3 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 Gitlab, with some branches and snapshots. If you complete the exercises, you are sure to have achieved that state.

D-3.4 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-3.5 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. (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-3.6 (Your first repository)

1. For this module, and in module 4, the GitLab server of the SNT is used. All students already have access to it with their student account. So, go to `https://git.snt.utwente.nl/` and login with your student number and password.
2. Create a new (bare, initially empty) remote repository on that account.
   (a) Click NEW PROJECT or go to the new repository page directly: `https://git.snt.utwente.nl/projects/new`.
   (b) Give the repository a fitting name in the PROJECT NAME field, like `softwaresystems-lab3b`.
   (c) Make sure that you create a *blank* repository, do not use a template. We will fill the repository ourselves.
   (d) Click CREATE PROJECT.
3. Create a new ECLIPSE project, create a local repository and link the local repository to the remote repository.
   (a) In ECLIPSE, create a new empty Java project and add a trivial, single class `Hello` in it.
   (b) Go to the GIT perspective (via WINDOW → PERSPECTIVE → OPEN PERSPECTIVE → OTHER... → GIT).
   (c) In the toolbar of the "Git Repositories" tab, click the icon labeled "Create a new Git Repository and add it to this view", or, if you have no repositories, click the link labeled "Create a new local Git repository". Use the project directory as the repository directory.[1]
   (d) Open the repository with the arrow next to it. Right click REMOTES and click CREATE REMOTE....
   (e) Use `origin`[2] as the remote name and click Ok. The URI is the link to your remote repository on the SNT GitLab. You can find this link on the web page of your repository. Use the HTTPS link for now, not the SSH one,[3] it will look something like `https://git.snt.utwente.nl/<username>/<repositoryname>.git`.[4]
4. Now we will actually use the repository. We will make some changes to the project, 'commit' them to the local repository and 'push' them to the remote repository.
   (a) Make some changes to the `Hello` class. (Go back to the JAVA perspective first.)
   (b) Now we have to tell Git that we made changes to the file and want to make them permanent. This is done by *staging*, and then *committing* the file. Right-click on the project, followed by TEAM → COMMIT....[5]
   (c) The "Git Staging" panel will open. Here, we can select files whose changes we want to add to the commit by dragging them from the "Unstaged Changes" section to the "Staged Changes". This will *stage* the files, i.e. tell Git that we want to add the changes to our repository. Stage your `Hello` file now.[6]
   (d) After we have staged all of the files we want (in this case only `Hello`), we need to *commit* them. This will group all of the changes to the staged files together into one batch, which can later be viewed or even reverted when needed. Each commit has a message to describe what changes were made. Add a commit message for your changes in the "Commit Message" box, and click COMMIT.[7]
   (e) Now, the changes have been saved to our local repository. However, the repository we have made on GitLab does not know about those changes yet. This is what *pushing* is for. Pushing

[1] This is analoguous to running the `git init` command from the Git tutorial from before.
[2] The origin is sometimes also called the *upstream* repository.
[3] Setting up and using an SSH key is somewhat out of the scope of this lab. If you already have an SSH key setup and know how to use it, it's fine to use the SSH link as well.
[4] This is analoguous to the `git remote add origin <url>` command.
[5] If this menu item does not exist, choose TEAM → SHARE PROJECT first, then try again.
[6] Dragging the files in Eclipse is analoguous to running the `git add <filename>` command.
[7] Adding a message and clicking "Commit" is analoguous to running the `git commit -m "<message>"` command.

synchronizes all of your local changes with the remote repository. Push your local repository to the origin now. Right click the project, followed by TEAM → PUSH BRANCH 'MASTER'.... (Note that you could have actually used COMMIT AND PUSH to commit and push in one go in the previous step.)

5. Go to your repository on GitLab; the repository should now contain your changes to `Hello`. Also check the "Commits" page while you're there; you can see your commit here, too. You can also click the commit to show a detailed view of the changes.

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-3.7 (Rolling back)

1. Add a method to your `Hello` class (from Exercise D-3.6) 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. Don't change anything in the `Hello` class. 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 commit history for the project, as stored in your local repository (with your commit messages to guide you as to what happened: a good reason to use meaningful messages!). The most recent commit is shown at the top, with your first commit at the bottom.
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. This is a separate commit, 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 GitLab). *Staging* changes means you tell Git about some changes you made in your project, *committing* means you add the changes into the local repository and *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 all make their own changes. 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-3.8 (Pushing and pulling)

1. Now it's time to see how Git handles multiple users, and how changes by multiple users are merged together.
   (a) On the web page of your repository on GitLab, add your partner as a collaborator to the project. Go to SETTINGS → MEMBERS in the left bar, search for your partner's student number or name and add him/her to the project with the MASTER role.
   (b) On your partner's laptop, clone the central repository into a folder. You can do this from the Git perspective, so switch to that first (on your partner's laptop!).
   (c) In the toolbar of the "Git Repositories" tab, click the icon labeled "Clone a Git Repository and add the clone to this view", or, if you don't have any repositories, click the link labeled "Clone a Git repository".
   (d) Paste the URI to the repository in the URI field. This is the same URI that you used in Exercise D-3.6 when you added the remote repository to your local repository. Click "Next" until you get to the page where you need to choose a destination directory.

   (e) Pick a directory to clone the repository to. This can be anywhere you want, but for ECLIPSE it's easiest if you put it somewhere in your ECLIPSE workspace folder.
2. Next, we need to import the project into ECLIPSE from the newly cloned repository. Right-click the repository, choose IMPORT PROJECTS... from the menu, then click Finish to import the project into ECLIPSE.
3. Make some edits in one of the two local repositories you have now, commit them, and push them to the central repository.
4. In the *other* local repository, execute a *pull* by selecting TEAM → PULL from the project's context menu (in the JAVA perspective). 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 commit and push the changes from both repositories to the origin. What happens?
6. If the push failed in the last step, try to pull the reppsitory first, and then push again. What happens?
7. Make sure both repositories are up-to-date with the upstream by pulling both of them.
8. Edit the file again on both of the local repositories, but now editing the same lines of the same file. What happens when you commit and push on one repository, and then commit and pull on the other repository? If something happened, fix it and commit and push it. Then, make sure both repositories are up-to-date with the remote repository again.
9. Look at the repository history again (Right-click the project folder, and choose TEAM → SHOW IN HISTORY). You can see the commits you have made on your repository and the other repository, and some *merges* that were done to combine the changes.

The last few 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* branch; or, alternatively, merge changes from the master branch into yours (or indeed, from/to any other branch).

☞ **D-3.9** (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 behind the project name 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. Note that the menu item to push is now called PUSH BRANCH 'TRY' instead of PUSH BRANCH 'MASTER'. These edits are now part of the new branch, but are *not* visible in the master branch.
3. Switch to the `master` branch (TEAM → SWITCH TO... → MASTER). Note that the edits you just made on the `try` branch are now gone. Do some *different* edits here, in *different* files, and commit and push them.
4. Switch back to the `try` branch. Select TEAM → MERGE, select the `master` and do MERGE. The result should combine the edits you did in the two branches. Push the result to the upstream.

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 Laboratory exercises

This week you will further extend your Hotel application.

Passwords and Checkers

We start by implementing a `BasicPassword` class that represents passwords. You will use it to protect your safe later on in this week. The class should provide operations to compare the password with an arbitrary `String`, to check whether a certain `String` is the correct password and to change the password.

P-3.1 Study the documentation of the `BasicPassword` class. The documentation is included in the predefined files (subdirectory `ss.week3.password`).

See also ECK Sect. 4.7.2 for constants

1. Why is there a constant `INITIAL` to initialise the password upon construction, instead of initialising it simply to an empty `String`?
2. Why is the constant `INITIAL` declared as **`public`** instead of **`private`**?

Since you already have the specification of `BasicPassword`, it should not be too difficult to make a stub implementation for `BasicPassword.java`, with an empty constructor and empty method bodies, where necessary with a default **`return`** statement to avoid compilation errors.

P-3.2 Develop a stub implementation for `BasicPassword` in package `ss.week3.password`, respecting its documentation. Make sure the stub implementation compiles.

P-3.3 Now you are expected to complete your implementation of `BasicPassword`. In particular, 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.

In the implementation of `setWord()`, call other methods of `BasicPassword` wherever you can.

Use the `ss.week3.test.BasicPasswordTest` JUNIT tests found from Canvas to test your implementation.

Your current `BasicPassword` implementation accepts only passwords longer than 6 characters that do not contain a space. However, we want a flexible check to test if a password meets the requirements (e.g., length restrictions or presence of letters, digits and special characters, or both) depending on how we use the class. We will define an *interface* to make the test for new passwords more flexible by allowing different implementations of this test.

See also ECK Sect. 5.7 for interfaces

P-3.4 Write an interface `ss.week3.password.Checker` with two methods:

- **`boolean`** `acceptable(String)`, which should return **`true`** if the parameter is an acceptable. Give a **`default`** implementation that checks if the `String` is at least 6 characters long and does not contain any spaces, *i.e.*, the same criterion implemented in Exercise P-3.3.
- `String generatePassword()`, which should return an acceptable `String` (and does not have a default implementation).

Define preconditions and postconditions, as well as useful documentation.

P-3.5 Give two classes that implement the `Checker` interface:

- A class `BasicChecker` that only implements the default check;
- A class `StrongChecker` that inherits from `BasicChecker` and in method `acceptable()` checks *in addition* whether the `String` starts with a letter and ends with digit.

Hint: Use methods `charAt` and `length` from class `String` to get 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.

The intended client of a `Checker` is a `Password`. You will now adjust the password class you implemented before in `BasicPassword.java`.

P-3.6 Copy `ss.week3.password.BasicPassword.java` to `ss.week3.password.Password.java`, and then extend `Password.java` with an instance variable `checker` (of type `Checker`) and an instance variable `factoryPassword` (of type `String`), with appropriate queries. The purpose of the `factoryPassword` instance variable is to initialise the `Password` object with a known predefined password `String`, which can be passed to a password client, who can change it later.

See also ECK Sect. 5.6.3 for constructors and superclasses

Class `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` by creating a `BasicChecker` object and calling the first constructor with this object as parameter.

Printer

In the next exercises, you will extend the hotel system with the functionality to create and print a bill for a particular room. You will first define a `Printer` interface, and after that you will implement a `Bill` class. `Bill`s can be printed using implementations of the `Printer` interface. In order to do this, you will reuse the functionality to print a bill item from week 1 (`SimpleBillPrinter.java` in Exercise P-1.11). You will again use the static method `format` from class `String`, whose purpose is to create a formatted `String` from a number of input objects.

P-3.7 Define an interface `ss.week3.bill.Printer` with the following methods:

- Method `String format(String text, double price)`, which returns a formatted line listing the item and price, ending on a newline (i.e., with the character `'\n'` at the end of the `String`);
- Method `void printLine(String text, double price)`, which uses `format` to send the combination of `text` and `price` to the printer in a way that is specific to the implementation.

Give your `Printer.format` method a **`default`** implementation, which calls `String.format` to format the `text` and `price`, such that the statements

```
p.printLine("Text1", 1.0);
p.printLine("Other text", -12.1212);
p.printLine("Something", .2);
```

(where `p` is a `Printer`-instance) results in the output

```
Text1                1,00
Other text         -12,12
Something            0,20
```

In other words, the prices should be right-aligned with precisely two decimals each, just like in class `SimpleBillPrinter.java` that you created in Exercise P-1.11.

P-3.8 Program the following `Printer`-implementations:

1. A class `ss.week3.bill.SysoutPrinter`, whose `printLine` method directly prints to the standard output. Give `SysoutPrinter` a `main` method that calls `printLine` a couple of times with examples to show how this method works.
2. A class `ss.week3.bill.StringPrinter`, whose `printLine` method collects all lines in a single `String`, without directly printing it on the standard output. `StringPrinter` should have a method `getResult` that returns the collected string.

Hotel Bill

Now that you have different implementations of the `Printer` interface, you can implement a class `Bill` that prints a bill. To make this as general as possible, a bill will consist of *items*, where each item has a *description*, and an *amount* associated to it. In order to allow multiple alternative implementations, it is advisable to use an interface for this. Since this interface is specific to the class `Bill`, it will be declared as a *nested interface*.

See also ECK Sect. 5.8 for nested classes and interfaces

For the `Bill` class, the intention is that we format the text without printing it directly to standard output (`System.out`). In general, it is beneficial to separate formatting from printing; for example, if when implementing a test program we may not want all the text to be printed on screen, but we might want to write it into a special log file instead. Therefore, each instance of `Bill` receives a `Printer` object when it is constructed. If we want to print to the standard output (console) we can pass the `SysoutPrinter` to this object, otherwise, we can use a `StringPrinter`. Using the `StringPrinter` we can query the result after adding items.

P-3.9 Implement a class `ss.week3.bill.Bill` that has a nested interface `Item`. The files imported from Canvas contain the specification of this class and of the interface (`ss.week3.bill.Bill.html` and `ss.week3.bill.Bill.Item.html`, respectively). Make sure your implementation respects the given specifications.

The nested interface `Item` will be implemented by classes that are not nested classes of `Bill` themselves, such as, for example, a `Room`. Therefore, instead of declaring them with **`implements`** `Item`, you should refer to them with **`implements`** `Bill.Item`, since the `Item` interface is defined inside the scope of the `Bill` class.

Before implementing different items, you should first test the general behaviour of `Bill`.

☞ **P-3.10** Write a JUNIT test `ss.week3.test.BillTest` for your implementation of `Bill`. To create a JUNIT class in ECLIPSE:

- Select and right-click the package `week3.test`
- Choose NEW → JUNIT TEST CASE;
- Select NEW JUNIT JUPITER TEST;
- Enter the name of the test class and select the option to create a `setUp()` method;
- Press FINISH.

In the generated test class, you will see two empty methods: `setUp()`, which is annotated with `@BeforeEach`, and `test()`, which is annotated with `@Test`.

- When JUNIT runs a test class, the method annotated `@BeforeEach` is automatically run *before any test case*. Therefore, this is a good place to put any initialisation that creates an appropriate initial state for your test. For instance, in `BillTest` you can initialise an instance variable of type `Bill` 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, assign them to a specific requirement (or condition) and give them meaningful names. For instance, in `BillTest` you can create test cases `testBeginState` and `testNewItem`, each of which tests one specific requirement of the class to be tested.

Your `BillTest` JUNIT test class should have two methods, to test the `begin state` (`Bill` has no items), and whether items are inserted accordingly and the bill can be properly closed.

When implementing `BillTest`, take the following issues into account:

- In order to be able to test the `Bill` class, you have to create a stub implementation of `Bill.Item`. This can be done using a *nested class* `Item` inside `BillTest`. The constructor of `BillTest.Item` should take a `String` text and a `double` amount; its `toString` method should return the text, and its `getAmount` method should return the amount.
- The constructor of `Bill` takes a `Printer` object. For the purpose of `BillTest`, a `StringPrinter` is the best choice, since it collects the printed items in a `String` that can be tested afterwards.

- To test **doubles** for equality in a JUNIT test, you should use the method `assertEquals(double expected, double actual, double epsilon)`; see the documentation of this method for more information.

Password-Protected Hotel Safe

Now that you have a `Bill` and you can print it, you can implement `Items` that hotel guests need to pay for. We start by developing a `PricedSafe`, which is password-protected. Hotel guests can upgrade to this password-protected safe by paying a fee. Afterwards, you will implement a `PricedRoom` that costs money and includes a `PricedSafe` by default.

You will implement this functionality by defining subclasses of the existing classes `ss.week2.hotel.Safe` and `ss.week2.hotel.Room`, instead of modifying the classes to include this new functionality. Therefore, copy these classes to the `ss.week3.hotel` package and update their **`package`** declaration accordingly.

P-3.11 Specify a class `ss.week3.hotel.PricedSafe` that extends the `ss.week3.hotel.Safe` class and implements the `ss.week3.bill.Bill.Item` interface. The price of the safe is a parameter of the constructor. Additionally, `PricedSafe` should be password-protected, so that the safe only opens if a valid password is entered. Each `PricedSafe` can be (de)activated. Only an activated safe can be opened, and the safe can only activated with the correct password. Use an instance variable of type `ss.week3.password.Password` to store the password of your safe.

Therefore, specify the following commands:

- `activate`: receives a `String` with a password text as a parameter, and activates the safe if the password is correct;
- `activate`: without parameters, overrides the parent method, gives a warning and does not activate the safe;
- `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;
- `open`: without parameters, overrides the parent method and does not change the state of the safe;
- `close`: without parameters, closes the safe (but does not change its activation status).

and the following query:

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

In this exercise, you are also asked to specify the method `toString()`, which should include the price of the safe. You can already implement this method here in order to define how the `Safe` object will be represented as a `String`. This is necessary to implement a proper test case later on.

To get an overview of the classes in your system, draw a class diagram of your design for class `Safe`.

See also Appendix A

P-3.12 File `ss.week3.test.PricedSafeTest` provided on Canvas allows you to test your `PricedSafe` implementation. However, this JUNIT test class only tests whether your `PricedSafe` correctly implements `Bill.Item`. Extend this test so that the following test cases are also covered:

- Test if method `getAmount` works properly;
- Test if method `toString` works properly;
- Assert that a deactivated safe can be activated with the correct password and is activated and closed after that;
- Assert that a deactivated safe cannot be activated with an incorrect password (remains deactivated and closed);
- Test if after trying to open a deactivated safe with the correct password the safe is indeed deactivated and closed;
- Test if after trying to open a deactivated safe with an incorrect password the safe is indeed deactivated and closed;

- Assert that after activating a safe with the correct password it cannot be opened with an incorrect password, but after being opened with the correct password it is activated and open;
- Test if after activating and opening a safe with the correct password, and closing it, the safe is closed and activated;
- Test if after closing a deactivated safe, it is closed and deactivated.

P-3.13 Now implement the `PricedSafe` class as you specified in Exercise P-3.11. Improve your implementation until it passes the `PricedSafeTest` tests.

☞ **P-3.14** Execute class `PricedSafeTest` again, but this time also measure the test coverage. For this purpose, execute the test via the COVERAGE menu (rather than the RUN menu). ECLIPSE uses the EMMA plugin to measure the coverage, which you should have installed before.

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 are some classes covered to 0%? Is this a problem?
- Can you improve your test class to increase the coverage?

See also ECK Sect. 8.4.1 for assertions

If you followed the instructions properly, your implementation of `PricedSafe` contains a couple assertions (**`assert`** statements). These statements are only executed if the virtual machine (i.e., the `java` program) is called with the special option `-ea` ('enable assertions'), otherwise, they are simply skipped during execution.

In the tool installation session you should have enabled execution with assertion checking for all classes in your project by adding `-ea` to the default VM ARGUMENTS.

P-3.15 Give your class `PricedSafe` a `main()` method that calls the constructor or a method of `PricedSafe` in such a way that the precondition is violated. Execute the program. Which error do you see? In which cases are assertions useful?

P-3.16 Implement a class `PricedRoom` that extends `ss.week3.hotel.Room` and implements `Bill.Item`, taking the following issues into account:

- The constructor should receive a room number, a room price and the cost of the safe.
- The constructor should create a new `PricedSafe` and pass it to the parent constructor (`Room`).
- The result of `toString` should also include the price per night.

Run the `ss.week3.test.PricedRoomTest` JUNIT tests from the Canvas files to test your implementation, and improve it until it gives no errors.

Hotel Class & TUI

Finally, it is the responsibility of the hotel to produce the bill. A bill consists of one item per night, and an item for the safe, in case it is a `PricedSafe`.

☞ **P-3.17** Copy class `ss.week2.hotel.Hotel` to package `ss.week3.hotel`. Update the package declaration and change the constructor so that the first room in the hotel is a `PricedRoom`. This is necessary for the JUNIT tests to work properly.

Add a method `getBill` to this class that receives as parameter the name of a guest, the number of nights the guest spent in the hotel and a `ss.week3.bill.Printer` for the bill. If there is no guest with the given name, or if the guest stays in an 'standard' room (*i.e.*, not a `PricedRoom`), the method `getBill` should return the value `null`.

Do not forget that the bill should also include the use of the safe!

Test your implementation with the `ss.week3.hotel.HotelTest` JUNIT tests from the Canvas files, and improve this implementation until it passes the tests without errors.

☞ **P-3.18** Copy the class `ss.week2.hotel.HotelTUI` to package `ss.week3.hotel`, update the package declaration and add the following functionality:

- A new command *b name nights* that prints the bill for a `Guest` with name `name` for a number of nights. Use the method `Hotel.getBill()` and print the bill to the standard output using a `SysoutPrinter`.
- The option to activate a `PricedSafe` using a second argument `password`. If the `Safe` in a `Room` is a `PricedSafe`, the password argument must be provided. If it is a regular `Safe`, the second argument can be left empty.

We suggest the execution of these commands to look as follows:

```
Welcome to the Hotel booking system of the U Parkhotel
Commands:
 i name ........... checkin guest with name
 o name ........... checkout guest with name
 r name ........... request room of guest
 a name password .. activate safe, password required for PricedSafe
 b name nights..... print bill for guest (name) and number of nights
 h ................ help (this menu)
 p ................ print state of the hotel
 x ................ exit

i Richard
Guest Richard is checked into room 101

a Richard
Wrong params at activation (password required)

a Richard default
Safe in room 101 of guest Richard has been activated.

p
Hotel U Parkhotel:
  Room 101 (100.0/night):
      rented by: Guest Richard
      safe active: true
  Room 102:
      rented by: null
      safe active: false

b Richard 2
Room 101 (100.0/night)      100.00
Room 101 (100.0/night)      100.00
Safe for  10.00              10.00
Total                       210.00

o Richard
Guest Richard successfully checked out.

p
Hotel U Parkhotel:
  Room 101 (100.0/night):
      rented by: null
      safe active: false
  Room 102:
      rented by: null
      safe active: false
```

Hint: Define a constant for the new command character `b` and add it as a new **`case`** in the **`switch`** statement of the menu.

3.4.2 Recommended exercises

Hotel bill improvement

P-3.19 Improve your hotel bill class in Exercise P-3.9 so that it contains an item `Nights` that produce a single item on the bill for the total number of nights the guest stayed in a priced room.

Timed password

An additional 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.

P-3.20 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 expiration 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. Implement your own JUNIT tests to test your implementation.

P-3.21 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?

Password checkers

P-3.22 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-3.23 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.

Interfaces

P-3.24 (*Adapted from Niño en Hosch, Exercise 9.2*)
Consider an interface `Comparable` defined as follows:

```
public interface Comparable {

    /**
     * Checks whether this object is greater than the other
     *
     * @requires this.isComparableTo(other)
     * @param other object to the compared
     * @return true if this is greater than other
     */
```

```
    public boolean greaterThan (Comparable other);

    /**
     * Checks whether this object can be compared to the other
     *
     * @requires other != null
     *
     * @param other object ot be compared
     * @return true if objects can be compared
     */
    public boolean isComparableTo (Comparable other);

}
```

Assume that a `Date` class has the following queries:

```
public int getDay();
public int getMonth();
public int getYear();
```

with meanings that are straightforward. Define the class `Date` as an implementation of the interface `Comparable` and complete its implementation so that it properly represents dates. A `Date` can only be compared to another date and a later date is greater than an earlier date (e.g., 1 January 2020 is greater than 30 November 2019).

P-3.25 (*Adapted from Niño en Hosch, Exercise 9.7*)
A chess board is made up of 64 squares, 8 rows and 8 columns. Rows are numbered 1 to 8 from bottom to top, and columns are numbered 1 to 8 from left to right.

Define and interface `Piece` to model the movement of chess pieces. This interface has a query `(int row, int column)` that tests whether this is a valid move for a piece, and a command `moveTo(int row, int column)` that actually performs this move.

A bishop moves diagonally and a rook moves vertically and horizontally. Define classes `Bishop` and `Rook` that implement interface `Piece` to represent the movement of a bishop and a rook, respectively. To simplify this implementation you should ignore that other pieces may be on the board.

P-3.26 (*Adapted from Niño en Hosch, Exercise 9.8*)
Consider a class `Employee` that represents employees in a company. In addition to getter and setters for its private instance variables, this class has the following methods:

```
    /**
     *  Returns the number of hours this Employee worked
     *  in the current period.
     */
    public int hours()
    /**
     * Returns this Employee's pay for the period.
     */
    public int pay()
```

Different kinds of employees are defined in different ways depending on their employment contracts, and suppose this can change at any time. Therefore it is advisable to define a class to which the `Employee` class can delegate the execution of the payment, and make this class an implementation of an interface with a method `pay` that is called by the `Employee` class to calculate the actual payment.

Define an interface `PayCalculator` with a method `pay` to calculate the payment, and implement class `Employee` so that it calls an instance of this interface. Class `Employee` should have an instance variable of type `PayCalculator` in order to be able to call the `pay` method properly.

Now implement two classes that implement interface `PayCalculator` to pay the employee with some fixed hourly rate and another one that pays 1.5 as much for overtime. Pay attention to the information

that the `PayCalculator` implementations need in order to perform their work and make it possible for them to get this information.

Write a program or JUNIT tests to test your implementation.

Week 4

Week 4

4.1 Overview

4.1.1 Contents of This Week

Academic Skills This week there is one workshop dedicated to academic skills. We will begin this session by taking a global balance of the average effectiveness of our PTMP documents. How accurate were our estimates? We will review some of the basic concepts explained above about the time management cycle and how to improve our planning in the next iteration. This is something that we will also do as part of this week's assignment, which will save us some of the work.

We are also going to dedicate this workshop to work on the aspects of our life in which we spread. Why is this happening to us and how could we detect our procrastination moments? How could we give you a solution? We will develop our own profile to reveal what our strengths and weaknesses are.

Design The activities in this week cover the following topics:

- *Software Metrics*, see Section 4.3.1 for the exercises for Lab Session 4.
- Project work, see Section 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 Canvas. 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:

- Lists and arrays.
- List implementations.
- Collections: maps and sets.

4.1.2 Mandatory Presence

During the following activities, your presence is mandatory.

- (M) Session on Mathematics in CS and BIT (Wed 6–9)

4.1.3 Expected Self-Study and Project Work

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

- 2 hours to make the academic skills exercises;
- 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 Set of videos proposed in Edpuzzle.

Design Slides from lecture 4.

Programming

Lecture ECK, Sections 7.1–7.3, 9.1–9.2 and 10.1–10.2
Laboratory The following predefined files are provided on Canvas:

- `ss/week4/ArrayExercises.java`
- `ss/week4/DoublyLinkedList.java`
- `ss/week4/LinkedList.java`
- `ss/week4/MapUtil.java`
- `ss/week4/MergeSort.java`
- `ss/week4/Util.java`
- `ss/week4/test/ArrayExercisesTest.java`
- `ss/week4/test/BoardTest.java`
- `ss/week4/test/DoublyLinkedListTest.java`
- `ss/week4/test/LinkedListTest.java`
- `ss/week4/test/MergeSortTest.java`
- `ss/week4/tictactoe/Board.java`
- `ss/week4/tictactoe/Game.java`
- `ss/week4/tictactoe/HumanPlayer.java`
- `ss/week4/tictactoe/Mark.java`
- `ss/week4/tictactoe/Player.java`
- `ss/week4/test/CompatibleTest.java`
- `ss/week4/test/ComposedTest.java`
- `ss/week4/test/InverseBijectionTest.java`
- `ss/week4/test/InverseTest.java`
- `ss/week4/test/IsOneOnOneTest.java`
- `ss/week4/test/IsSurjectiveOnRangeTest.java`

Tools Use of the CHECKSTYLE plugin

4.2 Academic Skills

4.2.1 Assignments

A-4.1 Module Time Management Plan (MTPM) We will make the step from personal planning and execution with a week-perspective, to medium long-term planning (module planning). With what we have learned in the last two weeks, we ask you to make a new time management plan, but this time the plan must contemplate the complete module, i.e., the next 6 weeks. Your MTMP will be evaluated, so we ask that before sending it you ensure that your work meets all the minimum quality by making use of the Checklist that we will provide you on Canvas, otherwise you will have to make further modifications to pass this part of the module.

You have to submit your answer at the latest Sunday of Week 4 at 23:59 CET.

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` to the `src` directory of that new project (keeping the directory structure intact). (If you're still working on week 3, you can also take `ss.week2.hotel`.) Please pay attention to the following:

- Make sure the Metrics plugin has been installed and has been enabled for your project, as follows. Select your metrics project in the Package Explorer menu. Under the right button, select PROPERTIES, then METRICS → ENABLE METRICS.
- Display the Metrics view (WINDOW → SHOW VIEW → OTHER, then METRICS → METRICS VIEW).
- Make sure your project is *without compilation errors*, and refresh it when necessary (PROJECT → CLEAN…)

The actual calculations done by the Eclipse Metrics plugin are discussed on http://metrics2.sourceforge.net/.

D-4.1 In this exercise we'll look into some elementary metrics and explore how they are handled by the Eclipse plugin.

1. In WINDOW → PREFERENCES → METRICS PREFERENCES, change the ordering so that MLOC, VG (cyclomatic complexity), WMC, LCOM, CA, CE are shown on top.
2. For trying out MLOC ("Method lines of Code"), select a larger method in one of the classes of this project.
   - Check the value of MLOC for this method.
   - Insert a new comment of a few lines. What happens to MLOC?
   - Reduce the number of lines of code by changing the formatting. What happens to MLOC?
   - Let Eclipse reformat the code for this method (apply SOURCE → FORMAT ELEMENT). What happens to MLOC?
3. What is the value calculated by the Metrics plugin for the Afferent and Efferent coupling? Explain this outcome, using the definition of the metrics on http://metrics2.sourceforge.net/.
4. Calculate by hand the value of "Weighted Methods per Class" for the class `Hotel`, i.e., the sum of the cyclomatic complexities of all methods in the class. (You can find the cyclomatic complexity for each individual method in Eclipse.) What value for WMC is calculated by the Metrics plugin?

☞ **D-4.2** In this exercise we'll look at "Lack of Cohesion in Methods" (LCOM). As stated in the lecture slides, there are different definitions in the literature.

For the class `Guest`, make a table (on a sheet of paper) showing which attributes are accessed by which methods.

1. Calculate by hand the value of "LCOM1" (Chidamber & Kemerer).
2. Also calculate the value of "LCOM2" (Hencerson, Constantine & Graham; also the LCOM computed by the plugin).
3. Check the LCOM for `Guest` as calculated by the plugin. Does it correspond to your own calculation? If not what did go wrong?
4. The body of the method `toString` is a line (similar to)

   ```
   return "Guest " + name;
   ```

   Replace this by

   ```
   return "Guest " + getName();
   ```

   Can you explain what happens to the computed value of LCOM? Do you think this is reasonable, given what LCOM is trying to measure?

For the following question create a fresh Eclipse project an copy the package `ss.week4.tictactoe` to this prroject (the package is included in the lab files for this week). From the class `Board` consider the method `toString`:

```
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    }
```

☞ **D-4.3**

1. Draw (on paper) the flow graph that represents the cyclomatic complexity of the method `Board.toString`.
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. *Refactor* the method by moving lines 4–10 from `toString` to a separate method `rowToString`, which is then called from within `toString`. Eclipse can do the refactoring for you: select the lines in question and then select 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

In the next exercises you will work with arrays in Java. When compared to Python arrays, Java arrays are closer to the machine level. The most important difference is that Java arrays 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; namely `java.util.ArrayCopy(src, dest, srcIndex, destIndex, length)`. To see how this method works, look, for example, at the implementation of the method `add` in the `ArrayList` implementation of `java.util.Collection`.

See also ECK Sect. 7.1 and 7.2 for arrays

P-4.1 Implement the following two methods of the `ss.week4.ArrayExercises` class that has been provided in the lab files of this week:

- A method `countNegativeNumbers` that takes an `int[]` as argument, and returns the number of negative values in the array.

- A method `reverseArray` that takes an `int[]` as argument, and reverses the values in the array. For instance, if the argument contains 5 elements [1, 3, 7, 9, 4] before the method is executed, then it should contain the values [4, 9, 7, 3, 1] after the method is executed. This method should do this without creating a new array.

Use the `ss.week4.test.ArrayExercisesTest` JUnit tests from Canvas to test your implementation.

Checkstyle

Now consider the class `Util` that can be found in the files downloaded from Canvas.

P-4.2 This class violates the configured Checkstyle coding conventions in several ways. Run Checkstyle and write down the violations. Now modify the class implementation such that the functionality of the class implementation does not change, but the coding conventions are satisfied.

Sorting

In Module 1, you developed 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.

See also ECK Sect. 9.1, 10.2.1 and 10.2.2 for lists

P-4.3 Reimplement the merge sort algorithm from the Module 1 in Java for `Lists` of `Elem`, assuming that `Elem extends Comparable<Elem>`. [1] Use the provided stub code in class `MergeSort` for your implementation, and `MergeSortTest` to test it.

Linked Lists

Next you will implement linked lists, as discussed during the lectures.

☞ **P-4.4** Implement the methods `add(int index, Element element)` and `remove(int index)` of the `DoublyLinkedList` class provided this week. Use `DoublyLinkedListTest` to test your implementation.

See also ECK Sect. 9.2 for linked lists

P-4.5 Implement the following two methods in the provided `LinkedList` class.

- `private Node findBefore (Element element)` that returns the `Node` immediately before the first occurrence of the specified `Element`. If the specified `Element` is first in the list, or is not contained in the list, the method returns `null`
- `public void remove (Element element)` which removes the first occurrence of the specified `Element` from a list. Use the method `findBefore` to implement this method.

Use the provided `LinkedListTest` JUnit tests to test your implementation.

Tic Tac Toe

In this exercise you will implement the popular game Tic Tac Toe (see https://en.wikipedia.org/wiki/Tic-tac-toe).

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.

In our version of the Tic tac toe game, the board is represented by a one-dimensional array of `3*3=9` `Mark`-objects[2]. The relation between the array and the board is shown below:

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

[1] If you have not done Module 1 you can still refer to the description of the merge sort algorithm at https://en.wikipedia.org/wiki/Merge_sort.

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

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

Classes `ss.week4.tictactoe.Player`, `ss.week4.tictactoe.HumanPlayer` and `ss.week4.tictactoe.Mark` on Canvas implement a generic player, a human player and a mark, respectively. Classes `Board` and `Game` are also given, but are incomplete, so you are expected to complete them.

P-4.6 Implement the missing methods of the class `ss.week4.tictactoe.Board`. Make sure your implementation is valid according to their preconditions and postconditions. In your implementation you should use constants instead of hard-coded values. Use the `BoardTest` JUNIT tests from Canvas to test your implementation.

P-4.7 Class `ss.week4.tictactoe.Game` controls the interaction between the `Player`s and the `Board` and shows a textual representation of the board in the console. This class is incomplete, because method `play()` still needs to be implemented. Implement method `play()` of `Game` according to its specification.

A class with a `main` method is necessary to play the game. This class will be called `TicTacToe`. The `main` method has to create a new `Game`-object and call method `start`. To start the program with the names of the players, you will enable the user to to provide command-line parameters.

See also ECK Sect. 4.3.6 for how `main()` gets arguments from the start

The `main` method has the following signature:

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

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

If you start `TicTacToe` as follows:

```
java week4.TicTacToe Alice Bob
```

then the array of Strings `args` will get the value [`Alice`, `Bob`], i.e., the array has a length of two (`args.length == 2`), `args[0]` has the value `"Alice"`, and `args[1]` has the value `"Bob"`.

You can specify command-line arguments when running a class using ECLIPSE in its Run configuration. Your options can be entered in PROGRAM ARGUMENTS text box of the ARGUMENTS tab.

P-4.8 Define the class `TicTacToe` and implement its `main()` method accordingly. When starting the game, the names of the players should be given as arguments, and these arguments should be used to create the `Player` objects. Test your system by playing some games.

☞ **P-4.9** To make your system more user-friendly, ask for the names of the players using `TextIO` when no run configuration arguments are given. This creates an improved user experience.

Function Properties

The interface `java.util.Map<K, V>` can be used to implement a mathematical function. For each *key*, a `Map` returns the corresponding *value*. The `keySet()` of a map corresponds to the domain of the function, i.e., for which types (value ranges) 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/en/java/javase/11/docs/api/java.base/java/util/Map.html`.

See also ECK Sect. 10.3 for the Map interface

In this exercise, you will implement a class `MapUtil` 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 4.1):

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

- a function f is *surjective* if it maps to all values in its range (see Figure 4.1):

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

- the *inverse* of a function $f : X \to Y$ is the function $f^{-1} : Y \to X$, such that (see Figure 4.2):

$$\forall x \in X . f^{-1}(f(x)) = x$$

- 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 4.2):

$$\forall x \in X . h(x) = g(f(x))$$

P-4.10 Implement the static method `isOneOnOne` in the provided `MapUtil` class that checks whether a `java.util.Map<K,V> f` passed as a parameter is an injection, i.e., `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 preconditions and postconditions where applicable for this method that at least specify what this method returns. Test your implementation using the provided `IsOneOnOneTest` JUnit tests.

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-4.11 Implement the static method `isSurjectiveOnRange` that checks whether parameter map `f` is surjective. Add a parameter `java.util.Set<V> range`, and check for all elements in `range` there is a key such that `f` maps to this element.

Write preconditions and postconditions where applicable for this method that at least specify what this method returns. Test your implementation using the provided `IsSurjectiveOnRangeTest` JUnit tests.

Next you will implement the inverse of a function. There are two alternatives, namely 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-4.12 Implement the static method `inverse` that given a `Map<K,V>` object 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 the static method `inverseBijection` that returns a map of type `Map<V, K>` if the function is injective and surjective.

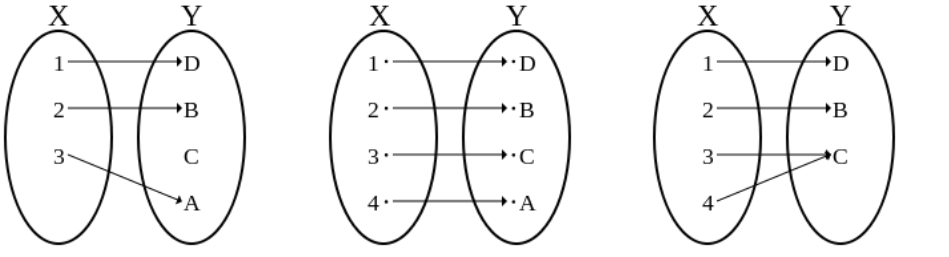
Write preconditions and postconditions where applicable for this method that at least specify what this method returns. Test your implementation using the provided `InverseTest` and `InverseBijectionTest` JUnit tests.

Finally, you will implement two methods to compose two functions.

☞ **P-4.13** Implement the static method `compatible` that checks whether the two maps passed as 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.

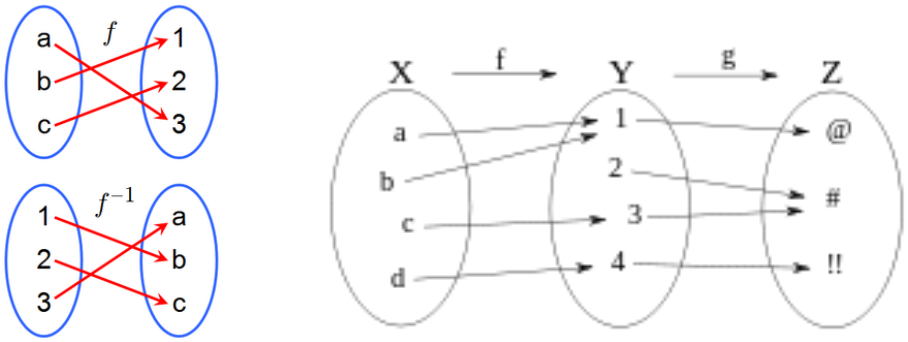
Next, implement the static method `compose` that defines the composition of two maps, provided they are compatible.

For both methods, write preconditions and postconditions where applicable. They should at least specify what each method returns. Test your implementation using the provided `CompatibleTest` and `ComposedTest` JUnit tests.



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

Figure 4.1: Examples of injectivity and surjectivity



A function and its inverse Function Composition

Figure 4.2: Examples of inverse and function composition

4.4.2 Recommended exercises

P-4.14 Make the following exercises from ECK:

- Exercises 7.1, 7.2, 7.3 and 7.5.

4.5 Mathematics: Relevance for Computer Science and Business Information Technology

The goal of the Mathematics session this week on Thursday 6–7 is to show you the relevance of Mathematics for CS and BIT. Participation in this session is mandatory for the first-year CS and BIT students. Premasters, minor, resit and double students are exempted.

Previously we recorded a collection of interviews with researchers from the Computer Science department about how they use Mathematics in their research. You are expected to listen to these interviews beforehand and discuss them in a group during the first part of the session. This discussion will be then compiled in a short presentation of 4 minutes (1 minute to switch groups) to be delivered in the second part of the session. The whole procedure, including the interviews and the groups, will be published on Canvas.

In your presentation, you should include the following:

- Of which Mathematical applications were you aware already before watching the interviews and discussing them?
- 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 topics of this module?

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

Week 5

Week 5

5.1 Overview

5.1.1 Contents of This Week

Academic Skills This week there is one workshop dedicated to academic skills. Our last session on academic skills will be devoted to reflecting on the effectiveness of group contracts during this first half of the module. Did the contracts work? Did they have to be used during these last weeks? How often? What things went well and what things went wrong? What things would you add to a next version of the contract?

The second part of the workshop will be devoted to providing feedback on the MTMP of other colleagues following the rules learned from effective feedback. We will use a rubric provided by the teacher to evaluate qualitatively and quantitatively the work of a partner, and give you clues so that you can improve your design of the time management plan, at the same time that she will do the same with us.

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 Section 5.2.1.
- Project work, see Section 5.2.2.

Programming This week the following topics will be discussed:

- Streams input and output.
- MVC pattern.
- Exceptions.
- Security engineering: hash functions, Java security properties, side-channel attacks, and using security libraries.

5.1.2 Deadline

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

5.1.3 Mandatory Presence

During the following activities, your presence is mandatory.

- (D) Design test (Thu 1–3)

5.1.4 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;
- 3 hours self-study for final preparations for the Design test;
- 4 hours Design project work beyond the sessions with assistance; and
- 2 hours self-study for the Programming thread.

5.1.5 Materials for this Week

Academic Skills No reading material this week.

Programming

Lecture ECK, Sections 3.7 and 11.1
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 Canvas:

- `ss/week5/Zipper.java`
- `ss/week5/test/ArgumentExceptionTest.java`
- `ss/week5/test/EncodingTest.java`
- `ss/week5/test/ZipperTest.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. 85–90 % of the test will be about UML diagrams, 10–15 % 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.

For the Software Metrics part of the test, you will be expected to understand what the metrics intend to achieve. The test may ask you to compute a metric for a given case and may also ask questions testing your understanding of the nature of a particular metric.

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 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. Cyclometric complexity is in fact the only metric 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**.
If you submit the project in time you get 0.5 bonus for the project grade and, if the project is not yet graded sufficient, specific feedback on how to improve the project before the end of the module.

5.3 Programming

5.3.1 Laboratory exercises

Strategy

See also ECK Sect. 5.7.2 for interfaces

This week you will extend the Tic-Tac-Toe implementation of week 4 in order to support an automated (computer) player. To do this, we need to change the functionality of the `Player`, using an interface as a type.

P-5.1 Specify an interface `ss.week5.tictactoe.Strategy` to determine the next move for the Tic-Tac-Toe game. This interface must have the following two methods:

- **`public`** `String getName()` that returns the name of the strategy;
- **`public int`** `determineMove(Board board, Mark mark)` that returns a next legal move, given the `Board board`, for the player with `Mark mark`.

P-5.2 First you have to develop a naive strategy (called `"Naive"`), in which method `determineMove()` returns an arbitrary empty field on the board.

Specify and program the class `ss.week5.tictactoe.NaiveStrategy` that implements this naive strategy by implementing the `Strategy` interface.

In order to choose an arbitrary empty, in method `determineMove()` you should first construct a collection of empty fields (e.g., with an array of integer values). Afterwards, you should select a random empty field out of this collection. You can use method `Math.random()` for this purpose.

Copy class `ss.week4.tictactoe.Player` to `ss.week5.tictactoe.Player` and update the **`package`** declaration accordingly.

P-5.3 Specify and implement a class `ss.week5.tictactoe.ComputerPlayer` that extends class `ss.week5.tictactoe.Player`. A `ComputerPlayer` always has a strategy, where name of the `ComputerPlayer` is the name of the strategy, followed by a hyphen (`"-"`), followed by a representation of the player's mark.

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.

As expected, method `determineMove()` from `ComputerPlayer` should use the computer player's strategy. Additionally, it should provide the functionality to inspect and update the strategy (getters and setters, respectively).

P-5.4 Copy your class `ss.week4.tictactoe.TicTacToe` to `ss.week5.tictactoe.TicTacToe` and update it such that the game can be played with a naive computer player. If the name `"-N"` is given either as an argument or as input, a ComputerPlayer with a naive strategy should be created. For example, if you start the program with the arguments `wim -N`, as in

```
java week5.tictactoe.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.

You probably have noticed that the computer is not very likely to win if it plays with the naive strategy. Therefore, next you will develop a smarter strategy, which thinks one move ahead.

P-5.5 Specify and implement a class `ss.week5.tictactoe.SmartStrategy` that implements the `ss.week5.tictactoe.Strategy` interface. Method `determineMove()` returns an empty field using the following strategy:

- 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 is advisable to make a copy of the board in the implementation of `determineMove()` to calculate what happens next when a field is selected.

The name of this strategy is `"Smart"`.

☞ **P-5.6** Update the class `week5.tictactoe.TicTacToe` in such a way that also the smart computer player can play the game. If the name `"-S"` is given, this means the player is a 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, MACs, etc., one often needs to handle binary data (i.e., raw bytes). To be able to represent these binary data correctly in plain, normal (UTF-8) text you can use encoding methods[1], such as Hex and Base64. There is no need to implement these encoding schemes yourself, but you can simply use existing libraries that implement these encodings. In the next few assignments you will work with one of these libraries, namely 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`.

See also See http://en.wikipedia.org/wiki/Hexadecimal

See also See http://en.wikipedia.org/wiki/Base64

P-5.7 To get you started, you are given the class `ss.week5.test.EncodingTest.java`. This class uses the Apache Commons Codec library, which can be obtained from https://commons.apache.org/proper/commons-codec/download_codec.cgi. Download file `commons-codec-1.13-bin.zip` and extract its contents. The bytecode of the library is in a Java archive (JAR) file called `commons-coded-1.13.jar`.

Now you have to add this library to your Java project in order to run the `EncodingTest` program. The most elegant and straightforward way to add a library to an ECLIPSE project is to copy the jar-file with

[1] *This is not the same as encryption!* Encodings such as Hex and Base64 are easily reversible.

the library (`commons-codec-1.13.jar` in our case) to the `lib` directory of the project (create one if it does not exist), choose JAVA BUILD PATH in the project PROPERTIES, choose the tab LIBRARIES, select CLASSPATH, click on ADD JARS and point to the jar-file (`lib/commons-codec-1.13.jar`).

The class `EncodingTest` prints out the hexadecimal encoding of the ASCII string "Hello World" by first getting the raw bytes representation of the string using the `getBytes` method of the `String` class. These raw bytes are given to the `encodeHexString` method of the `org.apache.commons.codec.binary.Hex` class. The provided code prints 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-5.8 This same library also supports the conversion of an hexadecimal representation back to bytes. Add a few lines of code to the `EncodingTest` class so 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`. Check the documentation for the Apache Commons Codec library at https://commons.apache.org/proper/commons-codec/archives/1.13/apidocs/index.html to determine which method to use. Use `new String(bytearray)` to create a string from a byte array.

Hint: the `String` class has a method `toCharArray` to convert a string to an array of characters.

☞ **P-5.9** Base64 is another way to represent binary data in plain (ASCII) text, which is often used for email attachments. In this exercise you will play with Base64 encoding by adding a few more lines of code to class `EncodingTest`.

- 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?

Exceptions

On Canvas, 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-5.10** Define three classes:

See also ECK Sect. 8.3

- `ss.week5.WrongArgumentException`, which extends `Exception`,
- `ss.week5.TooFewArgumentsException`, which extends `ss.week5.WrongArgumentException`, and
- `ss.week5.ArgumentLengthsDifferException`, which extends `ss.week5.WrongArgumentException`.

Define a new method `zip2()` that 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 JUNIT tests `ss.week5.test.ArgumentExceptionTest` and `ss.week5.test.ZipperTest` from Canvas to test your implementations.

5.3.2 Recommended exercises

P-5.11 Make the following exercises from ECK:

- 9.2, 9.3, 9.4 and 9.5.
- 10.1, 10.2, 10.3 and 10.4.

Tic-Tac-Toe Perfect Strategy

P-5.12 In P-5.5 you developed a smart strategy for the Tic-Tac-Toe game. 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 5.1 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` that implements the `ss.week5.Strategy` interface. 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?

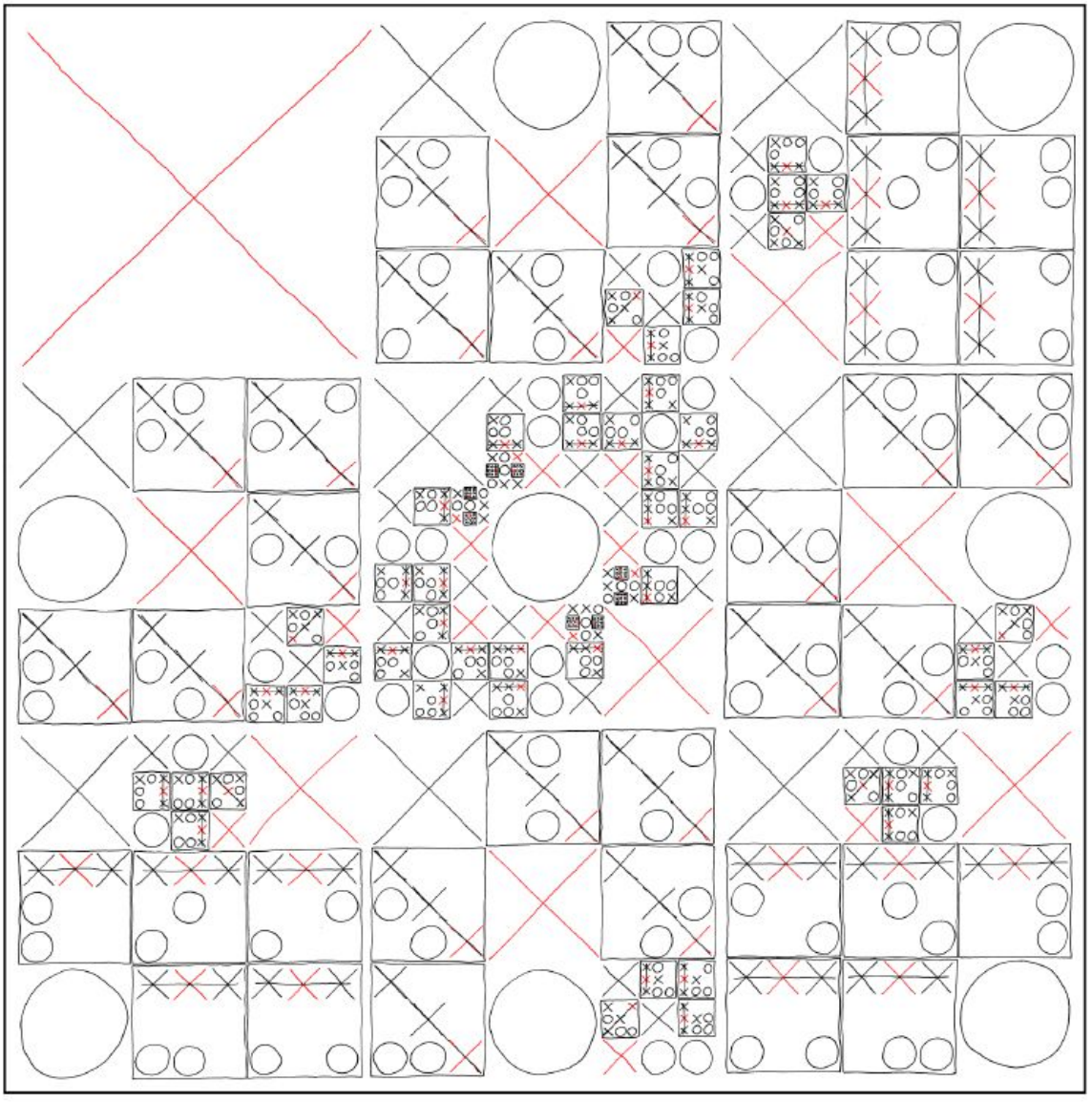


Figure 5.1: Possible perfect moves for player X.

Week 6

Week 6

6.1 Overview

6.1.1 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 programming with threads:
  - Thread creation and thread life cycle
  - Synchronisation between threads
  - The wait-notify mechanism
- Kick-off for the programming project.

Additionally, there is an optional 1-hour lecture on Wednesday presenting tips and tricks for practical software development, such as the use of debuggers.

6.1.2 Mandatory Presence

During the following activities, your presence is mandatory.

- (P) Diagnostic test (Wed 1–2)
- (P) Project meeting on overall design of the programming project (Wed 3–4)

6.1.3 Expected Self-Study and Project Work

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

- 2 hours self-study for the Programming project; and
- 7 hours self-study for the Programming thread.

6.1.4 Materials for this Week

Programming

Lecture
- Core Java I, Chapter 14, until BlockingQueues (p. 819–877)
- ECK, Sections 12.1–12.3

Laboratory The following predefined files are provided on Canvas:
- `ss/week6/ConcatThread.java`
- `ss/week6/IntCell.java`
- `ss/week6/QuickSort.java`
- `ss/week6/Words.java`
- `ss/week6/account/Account.java`
- `ss/week6/bounce/Ball.java`
- `ss/week6/bounce/BallPanel.java`
- `ss/week6/bounce/Bounce.java`
- `ss/week6/cards/Card.java`
- `ss/week6/challenge/ttt/Board.java`
- `ss/week6/challenge/ttt/Game.java`
- `ss/week6/challenge/ttt/Mark.java`
- `ss/week6/dictionaryattack/DictionaryAttack.java`
- `ss/week6/mandel/MandelPanel.java`
- `ss/week6/mandel/MandelSet.java`
- `ss/week6/mandel/MandelThread.java`
- `ss/week6/test/CardTest.java`
- `ss/week6/test/DictionaryAttackTest.java`
- `ss/week6/test/LeakedPasswords.txt`
- `ss/week6/test/VoteMachineModelTest.java`
- `ss/week6/test/QuickSortTest.java`
- `ss/week6/threads/Console.java`
- `ss/week6/threads/IntProducer.java`
- `ss/week6/threads/IntConsumer.java`
- `ss/week6/threads/IntCell.java`
- `ss/week6/threads/UnsynchronizedIntCell.java`
- `ss/week6/threads/ProdCons.java`

6.2 Academic Skills

6.2.1 Peer feedback session

In the Design project groups, 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.

As the peer feedback session lasts only 20 minutes, *please make sure that you arrive exactly on time.*

6.3 Programming

6.3.1 Laboratory exercises

Textual User Interface using Scanner

Until now, to implement Textual User Interfaces (TUIs) you have used the `TestIO` class, which is a library introduced in ECK to facilitate the implementation of TUIs for beginners. The JAVA standard class for

implementing TUIs is the `java.util.Scanner` class. You will use this class in the following exercise to interpret data input and use these data in your program. Start by checking the specification of the `Scanner` class in the JAVA API, in particular the following methods:

- Constructor `Scanner(InputStream)`, which creates a `Scanner` from an *input source* (`InputStream` object). A commonly used input stream in Java is `System.in`, which denotes the standard input.
- Constructor `Scanner(String)`, which creates a `Scanner` from an arbitrary `String` object.
- Methods `hasNext()` and `hasNextLine()`, which test if there is a next word or a next line in the input stream, respectively, which can be read by the `Scanner`. If the `Scanner` is constructed with the standard input as input stream to read user input, the `Scanner` may have to wait until new data have been provided (ended by a carriage return, i.e., by pressing `Enter`).
- Methods `next()` and `nextLine()`, which read the next word (i.e., a series of connected symbols without a space) or the next line, respectively, from the input stream. These methods return an error message if they are called and there is no next element to be read. Therefore, a call to `hasNext()` and `hasNextLine()` should precede a call to `next()` and `nextLine()`, respectively.

See also ECK Sect. 2.4.6 and 11.1.5 for Scanner

P-6.1 Using the class `java.util.Scanner`, implement a class `ss.week6.Words` with the following behaviour:

- On the standard input it asks for a sentence, consisting of words separated by spaces.
- It prints the words one by one on standard output.
- It stops if the entered sentence starts with the word `"end"`.

The program should also work if the sentence is empty, i.e., if the sentence has no words. An example execution is as follows:

```
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: You can create a `Scanner` with the contents of the sentence (line) and use it to break the sentence into separate words.

Card Game

See also ECK Sect. 11.1 for I/O streams

On Canvas, you can find class `ss.week6.cards.Card`, which represents a game card. In this exercise you will add input and output methods to this class. Java supports various techniques for input/output communication, namely via (readable) *text*, *primitive data* and *objects*. You will get acquainted with such input/output communication streams in the following exercises. Since input and output are often sources of failures, your method implementations should capture them whenever necessary, and throw appropriate exceptions.

The following exercise is about *text channels*, which 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-6.2 Add a method `write()` to the class `ss.week6.cards.Card` with a `PrintWriter` as parameter. The method should send a representation of the object on which the method is called (obtained by the use of `toString()`) to the `PrintWriter`.

Test your method by adding a method `main()` to class `Card` that either creates a `PrintWriter` object to write to a file if a file name is given as argument to the program, or writes its output to the standard output (`System.out`) if no file name is given. For example, if the program is called with

```
java ss.week6.cards.Card cardfile.txt
```

the output is written to file `card.txt`.

Method `main()` should create three or four `Card` objects and write them to a file or the standard output. To test your program, check whether the file is indeed created and whether it contains the intended text.

Now that you can write cards to a file, the next step is to construct `Card` objects based on data stored in a file. This can be facilitated if the input format is the same as the output format that you used to write the cards to the file. The output format has been defined in `Card.toString()`, and consists of two words on a line to represent the suit and the rank, separated by a space.

In contrast to writing to `PrintWriter`, when reading from a `BufferedReader` you have to take exceptions into account.

☞ **P-6.3** Add to the `Card` class a method with the following signature:

```
public static Card read(BufferedReader in) throws EOFException
```

Method `read()` should read from `BufferedReader in` and return a `Card` instance on based on this input. Make sure method `read()` works as follows in case of errors:

- Returns **`null`** if the `BufferedReader` does not allow for the construction of a valid card, for example, because the line that was read does not comply with the format `"suit␣rank"`.
- Returns a `EOFException` when the `BufferedReader` is finished.

Hint: you can use the `Scanner` class to parse the card information in order to obtain the suit and the rank.

Modify your `main()` method to check whether method `read()` and method `write()` from Exercise P-6.2 match, i.e., if `read()` can properly read the cards written by `write()`. After writing objects to a file, your `main()` method should now open this file, read each object and compare it with the one that was written (using `equals`). Show an error message if the objects differ.

The other techniques to write and read objects are *data streams* and *object streams*. For you to have more time to explore other topics, and because the data and object streams are structurally very similar to the *text streams*, we have not included exercises on them here. You can find such exercises in the recommended exercises of this week.

Password dictionary attack

Suppose that the fictitious company BigSimpleCorp has a simple online service that authenticates its users through passwords. 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 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 hash functions 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, 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 incomplete file for this assignment called `ss.week6.dictionaryattack.DictionaryAttack.java`. You will complete it during the exercises.

P-6.4 Fill the body of the `readPasswords()` method of the `DictionaryAttack` class. The JavaDoc for this method describes what it should do.

Hint: keep the parsing of the password file simple. You could, for example, simply use the `String.split()` method 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 has been 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[1].

P-6.5 Implement the `getPasswordHash()` method of the `DictionaryAttack` class. 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 contains more information. Java provides support for cryptographic hashes in the class `java.security.MessageDigest`. See Oracle's documentation on `MessageDigest` at https://docs.oracle.com/en/java/javase/11/docs/api/java.base/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-6.6 Implement the `checkPassword()` method. See the JavaDoc documentation for details, and use the `getPasswordHash()` method you have implemented. Check your implementation with the `ss.week6.test.DictionaryAttackTest` JUNIT tests that can be found on Canvas.

P-6.7 Next, compute a dictionary of password hashes by implementing the `addToHashDictionary()` method of the `DictionaryAttack` class. As described in the JavaDoc documentation, this method should read a file and compute the (MD5) hash of each line (use your `getPasswordHash()` method for this). It should then fill a dictionary (a `Map` interface implementation) 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 dictionary to start your tests.

☞ **P-6.8** With the building blocks in place, now implement the dictionary attack in the `doDictionaryAttack()` method of the `DictionaryAttack` class. This method should use the two `Map` instance variables. 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 already able to find several passwords. To find even more passwords, you can use a larger word list. For example, you can use the wordlist at http://www.cs.duke.edu/~ola/ap/linuxwords to make your dictionary larger.

For passwords that are not in a dictionary, the next step is to broaden the search. One approach is try all kinds of combinations of words, possibly with some extra letters or digits mixed in. Alternatively, one can simply try all possible passwords, the so called 'brute-force attack".

P-6.9 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 marked optional, do read through it and the text that follows it.

P-6.10 (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()`). 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?

[1] See also http://en.wikipedia.org/wiki/MD5. Using MD5 for anything that requires some level of security is a typically Bad Idea™. Using it to hash passwords is an even worse idea as you will see in the rest of these exercises.

Exercise P-6.10 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.[2] Instead, it is better to use a salt[3] 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.

Threads and GUIs

Sometimes we would like a Java application to do multiple things "at the same time". 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 is always a separate thread for event handling, called the *event dispatch thread*. This thread is responsible for

- Capturing user actions, e.g., mouse clicks,
- Rendering the GUI windows and the actions related to them.

If the event dispatch thread is busy executing other pieces of code, it cannot take care of these core responsibilities. The program then becomes unresponsive. Therefore, if after an event some calculation must be performed, this calculation is often performed on a different thread, so that the event dispatch thread stays available for user input.

The Mandelbrot Set In this exercise, you will work on a multi-threaded application for drawing fractals called `ss.week6.mandel.MandelSet` that is available on Canvas. This exercise is based on Example 8.4.4 of *Martin Kalin, Object-oriented Programming in Java, Prentice Hall, 2001*. When executing this application, notice that while drawing the fractal, the program is still able to react on selection in the menu.

P-6.11 Run the `MandelSet` application. Notice that if you select MENU → DRAW several times in quick succession, the drawing takes place right through the menu. After analysing the code, can you explain this effect?

P-6.12 In the `MandelSet` class, change the call to `MandelPanel.draw()` to a call to `MandelPanel.drawMandel()`. What is the difference when you execute the program now? Explain this difference. After doing the exercise, please revert this change.

In the given implementation, the `MandelPanel` class uses a special class `MandelThread` that extends `Thread` to create and start a parallel process. This is not always the most convenient approach.

☞ **P-6.13** Change the `MandelPanel` class to implement the `Runnable` interface, and adjust the creation of the `Thread` so that class `MandelThread` becomes superfluous, while the functionality of `MandelSet` is unchanged.

See also CJ Section 14.1

Bouncing Balls Package `ss.week6.bounce` on Canvas contains three classes:

- `Ball`, which models a bouncing ball. Class `Ball` has an instance variable of type `javax.swing.JPanel`, which is 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.

[2] For example, 180 billion MD5 hashes per second, see https://www.zdnet.com/article/25-gpus-devour-password-hashes-at-up-to-348-billion-per-second/.

[3] See wikipedia: https://en.wikipedia.org/wiki/Salt_(cryptography)

- `BallPanel`, which extends `JPanel` and maintains a list of `Ball` objects to bounce. Among others, it has a method `addBall()` to add a bouncing ball to this list and a method `animate()` to start bouncing.
- `Bounce`, which is the frame class of the application, with a button for adding a ball.

P-6.14 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-6.15 Define an inner class `AnimateThread` in `BallPanel` that implements `Runnable`. Method `run` of `AnimateThread` should only call `animate` in order to start the animation. Do not forget to remove the call to `animate` from `Bounce`.

Another solution to the problem of Exercise P-6.14 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, this task is to animate the next frame of the bouncing balls. This is conceptually a much better solution than defining the `AnimateThread`, because in this case the event dispatch thread takes care of rendering, and there is no parallel thread interfering with it, like by overwriting the menu, as we saw in Exercise P-6.11.

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-6.16** Change `BallPanel` so that it implements the `ActionListener` interface. In method `actionPerformed()`, it should draw a single animation frame that corresponds to the `moveBalls()` and `repaint()` calls in `animate()`. Create a `Timer` object in `BallPanel` and start this timer by calling its `start()` method.

Synchronisation

See also 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, classes and methods should also function correctly under these circumstances, i.e., they should be *thread safe*. We first study the behaviour of the static read and print methods of the class `ss.week6.threads.Console`.

P-6.17 Write a class `ss.week6.threads.TestConsole` that implements class `Runnable`. The `run()` method of `TestConsole` should call a (private) method `sum()` that uses `Console.readInt()` to read two numbers from the standard input, and writes their sum to the standard output using `Console.println()`. A possible execution of this program can be:

```
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 a `String` as the second parameter in the `new Thread()` call, like so:

```
new Thread(new TestConsole(), "Thread A").start();
```

The name of the thread can be retrieved by calling `Thread.currentThread().getName()`. Method `main()` should create and start *two* instances of `TestConsole`.

P-6.18 Study the behaviour of `TestConsole` and try to understand the behaviour of the program, based on the program code. Why is this behaviour problematic?

☞ **P-6.19** Develop a class `ss.week6.threads.SyncConsole` by copying `threads.Console` and making the methods `synchronized`. Also copy `TestConsole` to `TestSyncConsole` and change the references to `threads.Console` to `SyncConsole`.

See also CJ Section 14.5.5

P-6.20 The problem identified in Exercise P-6.18 is still not completely solved. Why not?

P-6.21 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-6.22 Adapt `TestSyncConsole` so that the computations of the two parallel processes no longer overlap.

See also CJ Section 14.5.3

The `java.concurrency.util.locks` library declares an interface `Lock` with methods `lock()` and `unlock()` for synchronisation. A synchronised 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-6.23 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-6.24** Reimplement `TestSyncConsole` by using a `Lock` to protect the critical section from being used "simultaneously" by two threads.

Producer-Consumer

For a proper 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 execute in parallel, and they communicate via a buffer with room for exactly *one* number. A producer asks the user for input values and a consumer retrieves a value from the buffer and prints this on the screen. The purpose is that each value that a producer puts in the buffer, is read and printed once by a consumer, in the same order as they are put in the buffer. Multiple producers and consumers may be operating simultaneously.

Hint: In this assignment, it is not necessary to synchronise the standard input and output, as reading and writing of values is not directly shown on the screen.

On Canvas, you will find the following classes:

- `ss.week6.threads.IntProducer` and `ss.week6.threads.IntConsumer`, which are the parallel processes;
- `ss.week6.threads.IntCell`, which is the buffer interface;
- `ss.week6.threads.UnsynchronizedIntCell`, which is an initial implementation of `IntCell` and
- `ss.week6.threads.ProdCons`, which is the main application class.

P-6.25 Run the `ProdCons` application several times. In which ways does this implementation fail to satisfy its specification? How can this be solved?

See also CJ Section 14.5.5

P-6.26 Replace the `IntCell` implementation in `UnsynchronizedIntCell` by a new implementation class to solve the problems signalled in the previous exercise. Use this new class in `ProdCons` instead of `UnsynchronizedIntCell`. Use the methods `wait()`, `notify()` or `notifyAll()` (inherited from `Object`). Which method is preferable? `notify()` or `notifyAll()`? Why?

Hint: Define a **`boolean`** instance variable to indicate 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-6.27** Study the documentation of interface `Condition` and implement a class `FinegrainedIntCell` that implements `IntCell` by using a `Lock` and multiple `Condition`s, instead of the `synchronized` keyword and methods `wait` and `notify()` or `notifyAll()`. The documentation of interface `Condition` gives a producer-consumer example to illustrate the usage of `Condition`.

See also CJ Section 14.5.4

Concurrency Theory

In the following exercises, you will investigate several smaller concurrent applications in detail to understand the different ways in which threads can interact with each other.

P-6.28 Consider the class `ss.week6.account.Account` that can be found on Canvas.

This class is not *thread safe* because the transactions are not protected and can be executed by multiple threads simultaneously, so that their integrity cannot be guaranteed.

1. Write a class `MyThread` that implements `Runnable` with the following constructor:

```
public MyThread(double amount, int frequency, Account account) {
    this.theAmount = amount;
    this.theFrequency = frequency;
    this.theAccount = account;
}
```

   The constructor's parameters indicate how many `times` a given `amount` should be added to the `account`. The `amount` may be negative, in which it is subtracted from the `account`. `MyThread` should then be executed on a new thread to execute the transactions `times` times in a loop.
2. Write a program `AccountSync` that creates two `MyThread` threads that execute on the same `account`. One thread should increase the value stored in the `account` with a fixed `amount`, while the other thread removes the same `amount`. Both threads should do this the same number of `times`, so that the expected result is `0.0`. After both threads are finished, the resulting value on the `account` should be printed on the screen.
3. Execute your program. Although you would expect the result to be always `0.0`, when you run your program several times you will notice that this is not always the case. Explain why not.
4. Modify class `Account` to make it thread safe.
5. Modify class `Account` now in such a way that the `value` on the account will never be smaller than `-1000`. In case a thread wants to decrease `value` further than that, it has to wait until the operation is allowed, i.e., until enough money has been deposited.

☞ **P-6.29** Consider the JAVA program `ss.week6.IntCell` that can be found on Canvas.

1. What are the possible results of running 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[4].

[4]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.

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 Canvas, 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 Canvas).

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-4.6 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?
- 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

Card game (cont.)

In Exercise P-6.2 and Exercise P-6.3 you implemented `read` and `write` methods to write to and read from a text stream, respectively. Now you can do something similar with data and object streams.

Data Streams Another 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-6.30 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-6.3.

A `Card` is uniquely represented by its rank and suit, which can both be represented by a `char`. Therefore, `Card` should be stored as two `char` values and not as `String`.

Modify the `main` method of your `Card` implementation to test these methods.

Object Channels Yet another form of communication is based on *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-6.31 Extend the class `Card` with methods `write(ObjectOutput)` and `read(ObjectInput)`, similarly to the data-based methods in Exercise P-6.3 and Exercise P-6.2.

As always, it should be possible to write a `Card` object and read it back in again.

P-6.32 Which form of communication is the least costly: text-based, data-based, or object-based? Explain your answer.

Concurrency Theory

P-6.33 Consider the following program fragment:

```
package ss.week6;

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-6.34 In this exercise, we develop a multithreaded version of the well-known quick sort algorithm. Consider the class `QuickSort`.

```
package ss.week6;


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 version of the quick sort algorithm can only be used to sort arrays of integer values. Adapt this algorithm to a multi-threaded version in which each call to `qsort` is done in a separate thread. The `QuickSortTest` JUNIT test class is provided to test `QuickSort`. You can use this test to test your multi-threaded version of the quick sort algorithm by changing it to use your version instead of the provided single-threaded version.

The programming pattern used in this exercise is often called *fork-join-style programming*.

Tic Tac Toe GUI

P-6.35 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.

On Canvas 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 `JButton`s (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`.

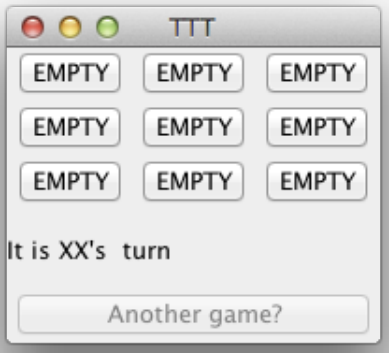


Figure 6.1: A simple GUI for Tic Tac Toe.

- 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 `Mark`s 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.
  Hint: 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. You can use an *inner class* `TTTController` inside `TTTView` for this. 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

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:

- The Model-View-Controller (MVC) pattern
- Network programming:
  - Use of sockets for communication over a network.
  - Networking and multithreading.
  - Framework of a client/server network application, including protocol.
- Graphical user interfaces.

Moreover, you have to make a planning for the programming project and discuss it with a teaching assistant.

7.1.2 Mandatory Presence

During the following activities, your presence is mandatory.

- (P) Discussing project planning with student assistant during lab session (Wed 1–4)
- (P) A project session on the group communication protocol (Wed 7 – 8)
- (P) Diagnostic test *Exam Preparation* (Thu 6 – 7)

7.1.3 Expected Self-Study and Project Work

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

- 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 3 until Making Url connections (p. 185 - 210)
- ECK, Sections 11.4, 12.4, 12.5 and Chapter 14.

Laboratory The following predefined files are provided on Canvas:
- `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/recipeserver/recipes/bread.txt`
- `ss/week7/recipeserver/recipes/spaghetti.txt`
- `ss/week7/recipeserver/recipes/pie.txt`
- `ss/week7/recipeserver/recipes/cake.txt`
- `ss/week7/hotel/client/HotelClient.java`
- `ss/week7/hotel/client/HotelClientView.java`
- `ss/week7/hotel/exceptions/ExitProgram.java`
- `ss/week7/hotel/exceptions/ProtocolException.java`
- `ss/week7/hotel/exceptions/ServerUnavailableException.java`
- `ss/week7/hotel/protocol/ClientProtocol.java`
- `ss/week7/hotel/protocol/ProtocolMessages.java`
- `ss/week7/hotel/protocol/ServerProtocol.java`
- `ss/week7/hotel/server/HotelClientHandler.java`
- `ss/week7/hotel/server/HotelServer.java`
- `ss/week7/hotel/server/HotelServerView.java`
- `ss/week7/hotel/server/HotelServerTUI.java`
- `ss/week7/hotel/test/HotelClientTest.java`
- `ss/week7/cmdchat/Client.java`
- `ss/week7/cmdchat/ClientHandler.java`
- `ss/week7/cmdchat/Server.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.

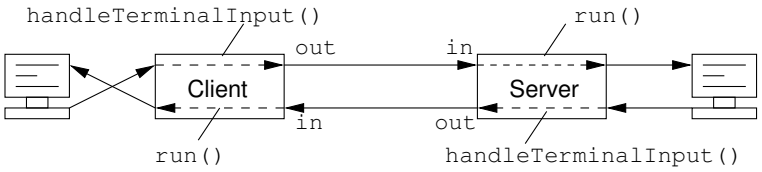


Figure 7.1: Overview of client-server communication

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 a teaching assistant no later than Wednesday this week.

P-7.2 What is the advantage of using the MVC (Model, View, Controller) pattern? Explain this in terms of reusability.

Communication via the command line

In this exercise, you will develop a simple application that allows two users (a client and a server) to communicate with each other via two *console windows* (also called *terminal* or *command prompt*). The connection between the client and the server will be implemented by using the `Socket` class.

The user of the client can type messages on her console and these messages are sent to the server through a `Socket` connection. The user of the server receives the messages on her own console. Similarly, the server user can type messages on her console, which are forwarded to the console of the client. The protocol between the client and the server in this case is symmetric, since apart from setting up the connection, both the client and the server provide the same functionality.

The client/server application programs should be able to receive and handle messages from its user (via the console) and from its *peer* simultaneously, which means that these applications should be multi-threaded.

On Canvas, you can find the files `ss.week7.cmdline.Server`, `ss.week7.cmdline.Client` and `ss.week7.cmdline.Peer`, which you can use as starting point for implementing a server, a client and their common behaviour, respectively.

As mentioned before, the client and the server only differ in the start up phase. The server application should be started first, and its user should provide her *name* and the *port* where the server should wait for a connection request from the client. To start the server for a specific user (name) at a specific port, open the Eclipse Run configuration dialogue, create a run configuration to run the class `ss.week7.cmdline.Server`, and specify Program Arguments `<name> <port>` in the Arguments tab. For example, to run the server for user 'Alice' on port 2727 you should give `Alice 2727` as Program arguments.

See also CJ Section 3.1.2

The client application should be started after the server application has been started. When starting the client, its user should provide her name, the host name of the server (an IP address, or `localhost` if both applications run on the same machine) and the port number on which the server listens to the client. Again, Program Arguments must be provided via the Eclipse launch configuration, e.g., `Bob localhost 2727`.

Now inspect the `main` method of the `ss.week7.cmdline.Client` class. In this method, first a `Socket sock` is constructed, based on the `addr` and `port` passed as argument to the program. The socket is used to create a `Peer` object. Then, a separate `Thread` is created to handle the input from the socket's `InputStream`. This means that `Peer` should be a `Runnable` object, and thus it should implement method `run`. Method `handleTerminalInput` catches the input that is typed in the console (standard input) 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.

Method `main` of `ss.week7.cmdline.Server` is similar, but the only differences are when the arguments are checked and the `Socket` instantiation.

The common behaviour of the client and the server is implemented in class `Peer`. This class has instance variables to hold the name of the peer, the `Socket` through which communication takes place, and the input

and output streams (`BufferedReader` and `BufferedWriter`, respectively).

Class `Peer` has the following three methods:

- **`public void`** `run()`, which reads messages from `in` and prints them on the standard output.
- **`public void`** `handleTerminalInput()`, which reads messages from the standard input and sends these to `out`. If the string `"exit"` is entered, then the method should return. After sending the message to `out`, the stream should be emptied by using `flush()`, in order to release operating systems resources and avoid memory leaks.
- **`public void`** `shutDown()`, which closes both streams and the socket.

Additionally, class `Peer` has a constructor with the following signature:

```
/**
 * @requires (nameArg != null) && (sockArg != null)
 * @param   nameArg name of the Peer process
 * @param   sockArg Socket of the Peer process
 */
public Peer(String nameArg, Socket sockArg) throws IOException
```

On Canvas, you can find Java stubs for the classes `Peer`, `Server`, and `Client`.

P-7.3 In the `Peer` class, implement the methods `run`, `handleTerminalInput` and `shutDown`, and the class constructor. Make sure all exceptions are treated properly. Input should be read per line, i.e., each message should be finished by a newline.

Class `Server` has only a `main` method. Similarly to 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, in contrast with class `Client`, first a `ServerSocket` needs to be created. After creating the `ServerSocket`, the server waits until a `Client` wants to connect with it over the given `port`.

☞ **P-7.4** Write the method `main` of the class `Server` inspired by the `main` method of class `Client`. 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.

Test your application "locally" by starting both programs in a single computer and exchanging messages between them. 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 console. Improve your implementation until it works as expected. Once the application works locally, test if your application also works to communicate on different computers. In this case, you will have to discover the internet address of the computer that runs the server to give it as parameter to the client!

Method `main` of the `Server` should have four `try/catch` blocks (e.g., to check the arguments passed to the `Server`). Some of these blocks are also defined in the `main` method of the `Client`, so it would be neater to wrap these `try/catch` blocks in methods (preferably in class `Peer`), so that they can be reused in the `Client` code.

P-7.5 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?

Networked attack

For this exercise, you are given access to a simple networked recipe server and a matching client. However, the server is clumsily written and vulnerable to a security attack. First make sure you can run the recipe server and the client works as expected. The code necessary to run the server and the client can be found in package `ss.week7.recipeserver` on Canvas. The server and the client can be executed by running the programs `RecipeServer` and `RecipeClient`, respectively. To run the `RecipeServer` properly, you have to change the working directory in the RUN CONFIGURATION (tab ARGUMENTS) by changing the WORKING DIRECTORY from DEFAULT to the `src/ss/week7/recipeserver` directory of your project. This is necessary otherwise the server will not find the recipes, which are in directory `recipes` that is now is a subdirectory of package `ss.week7.recipeserver`.

☞ **P-7.6** Inspect the source code of the recipe server and client to understand how it works. Use the given `RecipeClient` class to implement a `RogueRecipeClient` class 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*.

Networked Hotel Application

You will now develop your final application for the lab weeks: a networked hotel application. Step by step, you will work towards a system in which there is one server (with the hotel application) and multiple clients that are to be operated by the hotel staff. The clients enable the staff to check guests in and out, to activate safes, etc. Essentially, it is a networked *HotelTUI*.

In the files provided on Canvas you can find five subpackages of the `ss.week7.hotel` package:

- Package `client`, which contains the client classes;
- Package `exceptions`, which contains the exception classes used by both the client and the server;
- Package `protocol`, which contains the protocols for the client and the server, as well as the protocol syntax;
- Package `server`, which contains the server classes;
- Package `test`, which contains a client JUNIT test.

Parts of the server and client are given and you are asked to complete these implementations. We first explain how the server and client are expected to communicate over the network. This communication must be structured in terms of a *protocol*, which defines rules and message format to which both the server and the client must adhere to in order to properly communicate.

P-7.7 Inspect the `ClientProtocol` and `ServerProtocol` interfaces in package `ss.week7.hotel.protocol` and try to understand how the client and the server are expected to communicate with each other. The two interfaces contain the methods that the client and server must implement in order to adhere to the protocol, respectively. The server and the client are both already defined to implement the protocol interface, but the implementation of the methods is missing.

Implement all methods in `HotelClient` that are defined in the `ClientProtocol` interface. You can test your implementation by running the provided *confirmer test server* `ss.week7.hotel.serverconfirmer.jar`. This server is called a *confirmer* test server because it sends a confirmation for every message it receives from a client. The confirmation message includes the original received message, unless the client sends a `HELLO` or `EXIT` message according to the protocol. You can execute the server `jar` file in a terminal (command prompt) by typing `java -jar serverconfirmer.jar`.

To test your implementation, start the confirmer server at port 8888. Then, run the JUNIT `HotelClientTest` in the `test` package, which will execute a predefined sequence of requests to the server. Your client implementation should print the server responses to the standard output, preceded by `"> "`, in case you call the `readLineFromServer()` method. The test will compare what is written to the standard output with the expected response from the server, which will determine whether the client sent the correct request.

P-7.8 Now that your client behaves according to the protocol, complete the server by implementing all methods in `HotelServer` that are defined in the `ServerProtocol` interface. For now, the `do*` methods should return an arbitrary `String`, which you can use to test the whole system. At a later stage, you will connect the server with the hotel model in order to provide the intended functionality.

Next, implement the `handleCommand()` method in class `HotelClientHandler`. For this, you can reuse parts of the `HotelTUI` that you developed earlier in week 3.

To test your implementation, create a new test case in the JUNIT `HotelClientTest` class. Copy the contents of the `testConfirmer()` test case and edit the expected values to match the arbitrary `Strings` you defined earlier in the `do*` methods of the `HotelServer`. Run the test and check whether the server works correctly and returns the expected values.

The server and client now both adhere to the protocol and are compatible with each other. The next step is to implement their views. Two interfaces are given: `ss.week7.client.HotelClientView` and

ss.week7.server.HotelServerView. In the sequel, you will develop a TUI for the client and finish the view of the server.

P-7.9 Create a new class `hotel.client.HotelClientTUI` that implements the `HotelClientView` interface. Define a class variable of type `HotelClient` that is initialised through a constructor parameter. Take a look at the `HotelClientView` interface for the documentation of the methods to be implemented.

Now change the `HotelClient` class to use the new TUI. Make the following changes:

- Initialize the TUI in the constructor of `HotelClient`.
- Instead of printing messages to `System.out`, send them to the TUI using the `showMessage()` method.
- Implement the `start()` method according to the given JavaDoc documentation.

Test your implementation by sending various requests to your server.

Your client is ready now, but actually the hotel functionality still needs to be implemented. Therefore, on the server side you have to make the user experience more interactive and connect it with the hotel model in order to support the intended behaviour.

P-7.10 Finish the implementation of class `HotelServerTUI` by implementing the `getString()`, `getInt()` and `getBoolean()` methods according to the documentation in the `HotelServerView` interface.

These methods are already used by the `HotelServer` to ask for user input, but until now they have returned hard-coded responses. If you implement the methods correctly, the user is asked for the port on which the server should be started and whether a new connection should be established after a connection terminates.

☞ **P-7.11** Now connect the server with the hotel application you created in the first weeks. Add a class variable of type `Hotel` to the `HotelServer` class. Then:

- Implement the `getHotelName` and `setupHotel` methods according to the given JavaDoc documentation. In the `setupHotel` method, you should ask for the name of the hotel by using the `getString` method of the hotel TUI.
- Update all `do*` methods to check the method arguments and execute the requests in the hotel model according to the JavaDoc documentation. You should return adequate `String` responses.

Finally, test your implementation by starting a server, connecting multiple clients to this server, and executing all possible requests to the server. Check whether the hotel model is updated as it should be and improve your code until it works as intended.

Congratulations! You are now ready to take your networking skills to the next level in your project.
Good luck!

7.2.2 Recommended exercises

Multi-client chat

In Exercises P-7.3 to P-7.4, you made a simple socket connection between a server and a client, but often you have multiple clients for one server. In this exercise, you will build a multi-client chat.

When the `Server` is started with its main methods, `Client`s can log on to the `Server`.

The `Server` keeps on waiting for `Client`s 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 `ClientHandler`s, 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

Figure 7.2: Example GUI for a client.

`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 Canvas you can find incomplete Java files that you can use as a basis for this exercise.

P-7.12 Implement the missing bodies of the chatbox application. The structure of the classes provided on Canvas is not mandatory. If you have a better solution, then feel 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-7.13 Should the method `broadcast` of class `Server` not be declared `synchronized`?

P-7.14 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!

Chatbox

P-7.15 In this exercise, we extend the previous build command line chatbox by replacing the TUI by graphical components, such as `JFrame`s. 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 Canvas, 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:

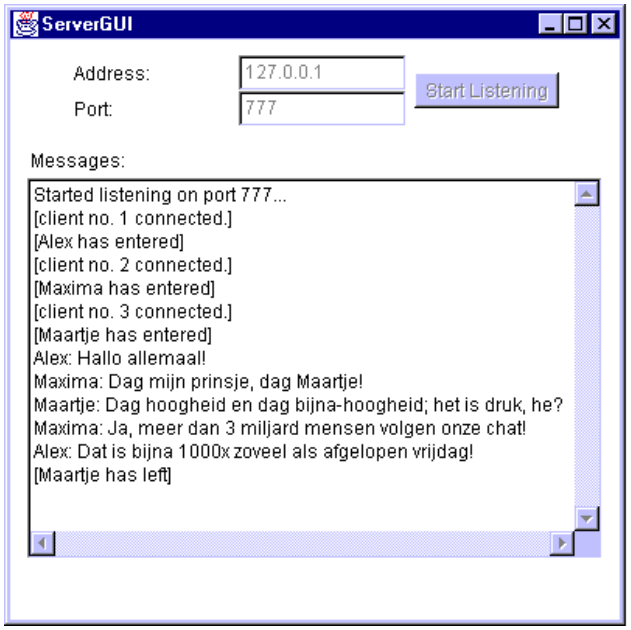


Figure 7.3: Example GUI for a server.

- 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 Canvas, 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.

Hint: The `Client` and `Server` classes should be `Threads` otherwise they block the GUI.

The Mandelbrot Set

P-7.16 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 `Color`s in the method `drawMandel`. This allows `paintComponent` to reread those values, instead of recomputing them.

Confusing confidentiality and integrity

This exercise is mainly aimed at students who participated in the Security pearl of Module 1.

P-7.17 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).

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 Canvas.

Next week, there will be another session on the protocol, where remaining uncertainties in the written-down version of the protocol can be discussed, and questions can be asked.

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).

Week 8

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.
- Wednesday morning during the lab session the written-down version of the protocol, as agreed upon during last week's protocol session, will be discussed. If necessary, remaining uncertainties will be resolved.
- 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.

On Wednesday a 1-hour optional lecture will be given on powerful and cool features of Java 8 that were otherwise ignored during the module, such as lambdas and streams. The content of this lecture is not part of the study material of this module.

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

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

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.

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.
- Early bird 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 final deadline to hand in a version of the Programming project.

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

Testing

Testing is a crucial activity in software development, independent of the development method applied in the software development project (e.g., traditional waterfall or agile development). Testing is crucial because it is extremely difficult to write bug-free code in one shot. Even if this were possible, the software developer would still have to demonstrate that the software works properly, and this can only be done by testing it somehow. Testing becomes even more important when the complexity of the system increases, so that different parts of the system need to be tested separately, and the developer also needs to test if these parts properly work together. Therefore, in this appendix we briefly explain some concepts related to the testing of software systems and give pointers to more information.

A.1 Types of tests

Three types of testing are generally identified in the literature:

1. *Unit testing*
   Consists of testing *a single software unit*, which is typically a *class* but sometimes a *method*. With unit testing the software unit is tested to assert whether it behaves on its own as expected.
2. *Integration testing*
   Consists of testing *multiple software units* to assert whether they correctly work together. In some cases integration testing can be used to automatically test the entire system.
3. *System testing*
   Consists of testing the *entire system* to assert whether it behaves as expected and/or whether it fulfils some specific requirements. For example by trying to make the system behave wrongly, or by playing out earlier defined scenarios called *user stories* (interactions that follow from the requirements) to test a feature from the user's perspective.

Unit testing and integration testing are performed automatically, which means that the tests are driven by pieces of software. When the software is changed by the developer (for example, a method is changed for some reason), the software needs to be tested and if these tests are automated they can be easily repeated to check if errors have been introduced with the change. Errors caused by changes are called *regressions*.

In order to get confidence in the software systems we develop, we need a healthy mixture of unit testing and integration testing, because many bugs arise when different units interact. Errors discovered during system testing are typically more expensive (or time-consuming) to fix than errors discovered during integration testing. Similarly, errors discovered during integration testing are often more expensive to fix than

errors discovered during unit testing, so we should try to find and fix most errors as early as possible, i.e., when performing unit testing. This is why we stress unit testing so much in this module.

We further distinguish between *black-box* and *white-box* testing:

1. *Black-box testing*
   In black-box testing, no knowledge of the internal behaviour or structure of the *system under test* is used by the tester. Internal behaviour can be, for example, the internal state of an object or the precise calculations that are performed. Instead, the tester uses the system specification, which relates inputs to outputs of the system or describes how the system should interact with its environment. Therefore, in this case only functionality and requirements are tested, not internal behaviour.
2. *White-box testing*
   In white-box testing, knowledge of the internal structure of the code (e.g., control flow, internal variables or class structure) is available and used to design tests. In this case, we can test various branches and check whether special "edge cases" are handled correctly. Examples are integer overflows, catching exceptions, handling bad input correctly, etc. We can also reason about so-called *class invariants*, which are conditions about the state of all instances of a class that must always be true before and after each method call. An example of a class invariant for a hotel application is that a hotel must not have a negative number of guests.

A.2 When should tests be written?

A common rule of thumb is to "test early and test often". An approach called *test driven development* prescribes that the tests should be written first, even before the units to be tested are implemented. In this way, the tests can provide guidance as to how the unit is intended to be used. The system is then implemented when the tests of all units and the integration tests succeed. In addition, whenever a bug is found when using the system, first try to isolate the bug by writing a test that fails due to the bug, and then fix the bug. The bug is considered to be fixed when the test passes successfully.

When writing a test, the goal of the test should be defined explicitly. Some examples are:

- Test whether the state of class `C` is properly updated by method `m1()`.
- Test whether method `m2()` handles each special case correctly.
- Test whether two objects `o1` and `o2` (instances of classes `C1` and `C2`, respectively) interact properly.
- Test whether the result of a sorting procedure works properly for all special cases.

Test coverage is a rough estimate of how much of the code is tested, by considering which lines of code are executed when a test is performed and which conditional branches are taken. Sometimes, software developers aim to achieve 100% testing coverage, in which case they only focus on how to get maximum coverage. In practice this means that tests are defined *just to cover the lines*, without considering quality of the tests. However, the ultimate goal of testing is to give confidence that the software does not contain errors, and that it functions according to its specifications (i.e., it fulfils its requirements). High-quality tests should therefore allow the developers to spot (isolate) software errors, not tests that are only defined to increase coverage. See for example Exercise P-3.12. Important things to test are functionalities that are high-risk or essential, such has handling passwords correctly. Most functions that contain more than a few lines or that contain branches, other method calls, or loops, are sufficiently complex to require testing. Testing is done not to convince the programmer that the implementation is correct, but to convince others (including their future self) that the implementation is correct.

After testing, the tester can write a *testing report*, which describes the performed testing.

Testing Report for FR 5
FR 5: *The name or description of Functional Requirement 5*
Expected behaviour: *A detailed description of how the software should behave, that is, the relationship between inputs and outputs, how it should interact with the environment or with other parts of the system, etc.*
Testing result • *Step 1* • *Step 2* • *Step 3* *Some screenshots, etc*

A.3 Testing and verification

When reasoning about the correctness of programs, we often use the following concepts:

1. *Preconditions* and *postconditions*
   A precondition is a condition that is true *before* a line of code or a function call, whereas a postcondition is true *after* execution of that line of code or function call. Often someone wants to prove that "if the precondition is true, *then* the postcondition will also be true", or similarly, "the postcondition will hold if the precondition is satisfied."
   In this module, we use `@requires` and `@ensures` in the Javadoc documentation of a method to explicitly write preconditions and postconditions. This documentation forces the programmer to be clear and unambiguous about the (intended) semantics (logical meaning) of the implementation.
   When generating JavaDoc documentation, the `@requires` and `@ensures` tags are not exported by default. Please refer to the tool installation guide on Canvas to find out how you can include them in your HTML export.
2. *Invariants*
   An invariant is a condition that is always true, or that is always true between method calls. For example, the number of guests in a room is always ≥ 0.
3. *Loop invariants*
   Loop invariants are special kind of invariants for *loops*, such as while-loops and for-loops. The invariant must be true before (and after) each iteration of the loop. Loop invariants are used to reason about the correctness of a loop: if something is true before the loop is executed, and we can prove that each iteration of the loop preserves the invariant, then the invariant is also true after the loop terminates.
   An example of a loop invariant is the following: imagine that we get an array of n numbers and that we want to compute the sum of all its numbers. We use a for-loop that increments i from 0 to $n-1$ and each iteration performs `sum = sum + a[i]`. Variable `sum` is initialized to 0. We could then have the loop invariant that sum $= a[0] + a[1] + \ldots + a[i-1]$, which is true before the loop starts, between each iteration, and at the end, when $i = n$.
4. *Assertions*
   Assertions are "runtime checks" in the code that check whether a certain condition is true when the assertion is run. Assertions are often used at the start of a method to check whether the preconditions are satisfied, or at the end of a method to check whether the postconditions are satisfied. Programmers often use assertions to check whether the program actually works in the way they think it works.
   In JAVA, assertions are only checked if they are enabled, for example, by using `java -ea` or by configuring the IDE correctly. Please refer to the tool installation guide on Canvas to see how you can enable assertion checking for your full project.
5. *Mocks* and *stubs*
   Mocks and stubs are classes or programs that simulate the behaviour of the classes or programs that a system under test depends on. For example, if a method is expected to interact in a certain way with another object, then we can provide a stub that simulates the behaviour of the other object and

records how the system under test interacted with it. A mock or stub is not a fully implemented class or program, as this is typically not needed for testing the system under test.

Preconditions, postconditions and invariants are often used not just for testing, but for *program verification*, that is, rigorously proving that software is correct. The main difference between testing and verification is that testing does not guarantee that the software is correct, while verification provides guarantees that the software implementation matches its specification. Program verification is often more difficult than testing because it requires formal reasoning and advanced tools.

A.4 JUnit 5

In this course, we use JUNIT 5 for testing. JUNIT 5 is supported by development environments such as ECLIPSE and IntelliJ. For more information on JUNIT 5 we suggest the tutorials https://www.vogella.com/tutorials/JUnit/article.html and https://www.javaguides.net/p/junit-5.html.

JUNIT uses annotations to control the execution of tests. Therefore, in order to understand and define JUNIT 5 tests you should learn how the `@Test`, `@BeforeEach`, `@BeforeAll`, `@AfterEach` and `@AfterAll` annotations work, and how to use the various `assertions`.