

Software Systems

(Course code: 202001023)

Module 2 of the TCS curriculum

University of Twente

2021-2022 Academic year

Module team:

Tom van Dijk (*Module Coordinator*)

Peter Lammich (*Design*)

Marcus Gerhold (*Design*)

Vadim Zaytsev (*Design*)

Ömer Şakar (*Programming*)

Faizan Ahmed (*Programming*)

Marieke Huisman (*Programming*)

Reza Soltani (*Programming*)

Iman Hemati Moghadam (*Programming*)

Mohammed El-Hajj (*Security*)

Georgiana Caltai (*Skills*)

Jasper de Jong (*Mathematics Line Coordinator*)

Rico van Lingen (*Mathematics Line*)

Tina Holtkamp (*Module Support*)

9th November 2022

Contents

Introduction	1
Module Overview	1
Learning Objectives	2
Modes of Instruction	3
Project Groups and Lab Groups	3
Communication	3
Help I have a Question	4
What can I expect from Teaching Assistants?	4
What can I expect from my Partner?	4
Study Material and Systems	5
Tests and Grades	6
Mandatory Activities	7
Signing off Mandatory Exercises	7
Monitoring and Evaluation of the Module	8
Acknowledgements	8
Pair programming	9
Motivation for pair programming	9
Traditional pair programming	9
Strong-style pair programming	9
Advantages of strong-style pair programming	10
General tips	10
Rules during Module 2	11
Design project	13
Case Description: Tarp learning management system	13
What to hand in	15
Submission and grading	15
Programming project	17
Detailed description of the client and server functionality	17
Description of other code requirements	18
Description of the report contents	19
Possible Extensions of the Application	20
Packaging for Submission	21
Grading	21
Activities and important dates	23
1 Week 1	25
1.1 Overview	25
1.1.1 Mandatory Presence	25
1.1.2 Expected Self-Study and Project Work	25
1.1.3 Materials for This Week	26
1.1.4 Tool Installation Session	26
1.2 Design	26

1.2.1	Project	26
1.2.2	Laboratory exercises on modelling	26
1.2.3	Laboratory exercises on testing	27
1.3	Programming	28
1.3.1	Laboratory exercises	28
1.3.2	Recommended exercises	36
2	Week 2	37
2.1	Overview	37
2.1.1	Contents of This Week	37
2.1.2	Expected Self-Study and Project Work	37
2.1.3	Materials for this Week	37
2.2	Design	38
2.2.1	Project	38
2.2.2	Laboratory exercises on class diagrams	38
2.2.3	Laboratory exercises on sequence diagrams	39
2.2.4	Recommended exercises on class diagrams	42
2.2.5	Recommended exercises on sequence diagrams	45
2.3	Programming	46
2.3.1	Laboratory exercises	46
2.3.2	Recommended exercises	54
3	Week 3	55
3.1	Overview	55
3.1.1	Contents of This Week	55
3.1.2	Mandatory Presence	55
3.1.3	Expected Self-Study and Project Work	55
3.1.4	Materials for this Week	56
3.2	Design	56
3.2.1	Project	56
3.2.2	Laboratory exercises on version control	56
3.2.3	Laboratory exercises on activity diagrams	60
3.2.4	Laboratory exercises on use case diagrams	62
3.2.5	Recommended exercises on activity diagrams	65
3.2.6	Recommended exercises on use case diagrams	67
3.3	Programming	69
3.3.1	Laboratory exercises	69
3.3.2	Recommended exercises	75
A	Testing and Design By Contract	79
A.1	Types of tests	79
A.2	When should tests be written?	80
A.3	JUnit 5	81
A.4	System testing	81
A.5	Design by Contract	82
B	Java Modeling Language	85
B.1	JML Method Contracts	85
B.2	Class Invariants	89
B.3	Loop invariants	91
C	Design Project Rubric	93
D	Programming Project Rubric	99

Introduction

This is the manual used during the Module 2 “Software Systems” of the BSc curriculum for Computer Science (TCS). The manual contains information about the organisation of the module and 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 and in a small group.

For the design of software systems, you learn to use Software Engineering models, particularly several widely used UML diagrams (use case diagrams, activity diagrams, class diagrams, sequence diagrams and state machine diagrams), and you get acquainted with the waterfall software development process, and you learn how to obtain requirements for a software system and translate them to a software design.

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, sufficient testing, and proper documentation. In this respect, this module builds upon the knowledge of algorithms and recursion acquired in Module 1.

For testing software systems, you learn to distinguish among the different levels at which testing can be performed (especially unit testing and system testing), the principles underlying a test plan, and some relatively simple testing techniques.

Attention is also given to the study of academic skills for the improvement in a number of soft skills. These focus on techniques to effectively manage a software development project within a team.

The Software Systems module also contains the Calculus 1B course of the Mathematics line, which covers the theory of integrals, power series, differential equations and complex numbers. The Mathematics line is, however, not covered in this manual.

Position in the Curriculum This module is offered in the second period (Quarter 1B) of the Computer Science (TCS) Education Programme. The module is also offered as a minor to various other programmes of the University.

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.
- *Skills (S)*: techniques to effectively manage a software development project within a team.
- *Mathematics (M)*: the Calculus 1B course, which covers the theory of mathematical functions and integrals.

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

The threads Design, Programming and Skills are contained in the study unit Software Systems Core (202001024). The Mathematics thread is contained in the study unit Calculus 1B (202001197).

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, with the help of software tools that are suitable for this purpose.
- Analysing relations among entities within a model, among different models and between each model and software code.
- Performing requirements eliciting interviews, and integrating knowledge gained from them into appropriate models of a software system.
- 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-oriented programming, such as encapsulation, abstraction, inheritance and polymorphism.
- Applying software design patterns to avoid tight coupling between different components of software, such as use of interfaces, the listener pattern and the Model-View-Controller pattern.
- Explaining problems with concurrent threads (race conditions) and applying basic synchronisation mechanisms to eliminate data races.
- Implementing client-server programs using 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.
- Collaborating with other students according to the pair programming method.

Attention is also given to the development of various skills for the improvement in project management. After successfully finishing this module you are capable of:

- Defining your role and responsibilities within a given project team.
- Eliciting and negotiating requirements, and establishing SMART goals for the successful implementation of a software development project.
- Managing time effectively, by applying tools and methodologies for project scheduling, and deciding and prioritising personal tasks within the project.

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

1. work with elementary properties of integrals and calculate integrals using different techniques, for functions of 1 variable
 - formulate Riemann sums
 - formulate and use the Fundamental Theorem of Calculus
 - calculate integrals using anti-derivates
 - calculate integrals using the substitution method
 - calculate integrals using the technique of integration by parts
 - calculate improper integrals using limits
2. work with power series and Taylor series, for functions of 1 variable
 - calculate the convergence radius by the ratio test
 - calculate Taylor series and polynomials
3. solve linear differential equations
 - solve first order equations using integrating factor
 - solve second order homogeneous equations with constant coefficients using the characteristic equation

- solve first and second order equations with constant coefficients using the method of undetermined coefficients
 - solve initial / boundary value problems
4. work with complex numbers
- plot (sets of) complex numbers in the plane
 - calculate absolute value and argument of a complex number to express the complex number in polar form
 - apply the complex arithmetic operations
 - find roots of a complex number and solve binomial equations

Modes of Instruction

The following modes of instruction are used in this module:

Self-study	You study the material, which consists of the programming and design topics (short videos) as well as the digital book found at https://math.hws.edu/javanotes/ . It is recommended to make notes for yourself, including questions you would like answered.
Q&A sessions	You ask questions to the teachers during the Q&A sessions. These sessions are recommended.
Lab sessions	You work in pairs on the lab exercises, supervised by teachers and teaching assistants. The exercises are mandatory and will be signed off during the lab exercises.
Project supervised	You work on one of the projects, and an instructor (teacher or teaching assistant) is present to answer your questions.
Workshop	These are sessions where you are expected to prepare some topics in advance and then dedicate the work time in the development of some related activity so that you put into practice what you learned previously.
Diagnostic test	You perform exercises (typically individually) to assess how well you are acquainted with the material. Participation is mandatory.

Project Groups and Lab Groups

All lab exercises and the programming project are performed in pairs. The design project is done in groups of four students. Groups are formed by the teachers and teaching assistants of the Module.

Please study the chapter on pair programming (page 9) for the explanation of **strong-style pair programming** which you are expected to use during the Programming lab sessions.

There are a few special cases:

- AM-TCS Double-degree students should be paired with other double-degree students
- Minor students should be paired with other minor students
- Resit students should be paired with other resit students

Communication

For communication of information, we use several systems.

- On **Canvas** you can find official announcements, the official student group assignment, materials, topic videos and slides, files for the lab sessions, and your grades.
- For lab sessions and questions, we use **Discord**. There are channels to ask general questions about design, programming, the design project, and the programming project. When you participate remotely during the lab sessions, you are required to use the dedicated voice channels, so the teaching assistants can quickly find you.
- We also use **Horus** for the lab sessions, where you can find a record of your sign-offs, and which is used to indicate to teaching assistants that you have a question or want to sign off exercises.

- Q&A sessions are physical this year. **Prepare for Q&A sessions by writing down your questions at a location announced on Canvas.** The teacher prepares answers to questions asked ahead of time and you can ask follow-up questions during the Q&A sessions.

Help I have a Question

Every student in the module can ask questions to the following people:

- Your **group teaching assistant**. Every student will be assigned to a group teaching assistant as a primary point of contact. The group teaching assistant is an experienced teaching assistant who can likely answer your question. Rule #1 of having a group teaching assistant is to keep your group teaching assistant informed.
- Your **housekeeper**. For various organisational issues, cases of sickness, conflicts in your groups, etc., talk to the housekeepers and keep the group teaching assistant informed as well.
- The **study advisers**. If there are personal issues or private matters that impact your study in any way, seek their advice. They are hired to help you and no topic is embarrassing.
- The **teachers**, during the Q&A sessions.
- The **teaching assistants**, during the Lab sessions.

In Module 2, each house is supported by a team consisting of the housekeeper and the assigned group TAs. The housekeeper will be the same person for every module and will therefore know you better than the group TAs. Simultaneously, the group TAs are more experienced with the content of Module 2.

Please ask non-personal questions, such as questions about the content or the organisation, on Discord. This way, other students can benefit too.

What can I expect from Teaching Assistants?

During the Lab sessions, teaching assistants will be walking around to help you with any question you might have. However there are limitations to this, thus you can expect the following things from them:

- You can expect from teaching assistants that they are prepared for the Lab session. Simultaneously, you can expect that they sometimes cannot answer every question. Many teaching assistants are a teaching assistant for the first time.
- Teaching assistants will not give direct answers to exercises. You can expect them to guide you into the right direction.
- Teaching assistants are only allowed to sign-off one exercise point at the time. If you want to sign-off more you have to rejoin the queue.
- Teaching assistant are not allowed to sign-off exercises outside Lab session hour. Asking them to do this is also not allowed.
- Your group teaching assistant is allowed to make an exception to the rule above. This exception can only be made if there is a good reason to do so.
- Asking questions in Discord is encouraged, however you should not expect a response from a teaching assistant in a timely manner outside Lab hours.
- Teaching assistants have no obligation to answer questions asked in a private chat. Your group teaching assistant is an exception to this rule.
- You can expect **queues** during Lab sessions. Queue lengths can vary from 15 minutes to sometimes 60 minutes in busy hours. If you don't want to wait forever in a queue, then come early to the Lab sessions and queue for signing off as soon as you finish a sign off point.

What can I expect from my Partner?

During the module you will be paired up with another student for the entire module. Together you will work on the exercises and the project. You can expect the following things from your partner:

- The ambition and motivation to pass module 2 Software Systems.

- Your partner should respond within 1 working day.
- Your partner should be available to work with you on the exercises during the Lab sessions.
- In the event that your partner is ill or has other personal circumstances, your partner should inform you and your group teaching assistant as soon as possible.

It is important to realise that your partner can also expect the points above from you.

Study Material and Systems

The study material and software tools used in Software Systems Core is the following:

Skills All the necessary material will be published on Canvas in the form of text and videos.

Design The topic videos are the main course material, the slides are available too. The material is divided into the following levels:

- [L1Tx]: introduction, history, terminology, software (development) lifecycle, software modelling
- [L2Tx]: the use of models (UML and otherwise) in different phases of software development
- [L3Tx]: structural modelling
- [L4Tx]: behavioural modelling
- [L5Tx]: supplemental modelling
- [L6Tx]: software analytics
- [L7Tx]: conclusion

There are 1–6 pre-recorded topics per level, and one more per level can be requested by students.

Programming The course material consists of a digitally available book, the topic videos, and the lab exercises. We use the following book:

David J. Eck. *Introduction to Programming Using Java*. Version 8.1.3, August 2021. Available at <http://math.hws.edu/javanotes/>

For the security topics, additionally 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 in Software Systems Core. It does *not* contain information about the Mathematics line.

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;
- Mandatory laboratory exercises;
- Self-study exercises to prepare for the Q&A sessions;

The Design project and Programming project are described in a separate chapter that immediately follows this introductory chapter.

Additional material on testing is provided in Appendix A. Additional material on JML specifications is provided in Appendix B.

We sometimes refer to parts of the study material. Here, 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.

It is important to **study** the book and the topic videos on Canvas, prior to the lab exercises. There is time allocated in the schedule for self-study.

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 that you 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:

- INTELLIJ (<https://www.jetbrains.com/idea/>), 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 into your IDE with a plugin.
- IntelliJJML (<https://gitlab.utwente.nl/fmt/intellijml/-/releases>), which is a plugin for INTELLIJ that supports syntax checking of software specifications written in JML.
- Checkstyle (<https://plugins.jetbrains.com/plugin/1065-checkstyle-idea>), which is a plugin for automatically checking programming code style.
- MetricsReloaded (<https://plugins.jetbrains.com/plugin/93-metricsreloaded>), which is a plugin for INTELLIJ that can compute various software metrics from programming code.
- Sonar Lint (<https://www.sonarlint.org/>), which is a plugin for your IDE. Sonar Lint can help you detect issues with your code.

Consult Canvas for a guide on installing these tools on your computer. On the Monday of week 1 there is a mandatory installation session scheduled to install everything before you start with the lab sessions.

Tests and Grades

Exceptions to the following rules can only be obtained via the Examination Board of TCS, and will typically only be granted based on personal circumstances outside of your control.

The following conditions have to be fulfilled before the Design test and the Programming test:

- You are only allowed to participate in the Design test if all mandatory Design lab exercises have been signed off before the Design test.
- You are only allowed to participate in the Programming test, if all mandatory Programming lab exercises have been signed off before the Programming test.

To successfully complete Software Systems Core, you have to:

- Participate in all mandatory activities and sign off all mandatory exercises:
 - All mandatory Design lab exercises must be signed off
 - All mandatory Programming lab exercises must be signed off
 - All mandatory Skills assignments must be passed
 - You must participate in the Mathematics case if you are a first-year TCS student
 - You must participate in the Design and Programming diagnostic exams
 - You must participate in the Programming project tournament
- Get a minimum of 5.0 for the Design test and the Programming test
- Get a minimum of 5.5 for the Design test *or* the Programming test
- Get a minimum of 5.5 for the Design project and the Programming project
- Get a minimum of 5.5 in $(\text{Design test grade} + \text{Design project grade})/2$
- Get a minimum of 5.5 in $(\text{Programming test grade} + \text{Programming project grade})/2$

This means you have to get at least a 5.5 for the tests and the projects, except you can get one insufficient grade in one of the tests which you must then compensate with a high grade in the corresponding project. For example, if you obtain a 5.0 grade for the Design test, you will need to obtain at least a 6.0 for the Design project.

If the above criteria are satisfied, then the final grade of Software Systems Core is determined by the average of the following results (all having the same weight):

- Design project grade (in groups of four students);
- Design test grade (individual);
- Programming project grade (in groups of two students);
- Programming test grade (individual).

Project repairs and deadlines The deadlines for the Design project and the Programming project can be found on pages 13 and 17. If the grade for the Design project is insufficient, it is possible to repair the Design project by resubmitting before the end of week 10. Repaired Design projects have a maximum grade of 6.0. The Programming project cannot be repaired.

Late submission policy Late submissions of the two projects have a reduced maximum grade, with a reduction of 1.0 point after the deadline has passed, and a further reduction of 1.0 point every 24 hours after the deadline has passed. For example, if you submit 30 hours after the deadline, the maximum grade will be reduced by 2.0 points. This means a project that would receive a grade 9 but handed in two days after the deadline will receive a grade 8. It is not possible at all to submit anything after the Module has ended.

Tests Instructions and example tests will be made available on Canvas. Use the example tests to identify your weak points. Please notice that the real tests may be completely different from the example tests. The Design test and the Programming test are open book and performed via Remindo. You are allowed to use the following materials:

- This manual;
- Slides of the topic videos;
- Books specified as course material for the module, or copies of the required pages of these books;
- The available programs on the Chromebooks used during the test
- A simple calculator;
- A dictionary.

You may *not* use any of the following:

- Solutions of any exercises published on Canvas, such as recommended exercises or example 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.

During the written tests in the Remindo environment, you will have access to the manual, to the slides, and to the ECK book. You will be able to use the tools IntelliJ IDEA and Visual Paradigm during the written tests. Any updates to these rules will be announced on Canvas.

Mandatory Activities

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

- The general session and the installation session on the first day of week 1
- All Skills sessions;
- Mathematics case session (week 4) for first year TCS students;
- The diagnostic tests (weeks 3 and 7);
- The programming project tournament (week 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.


Keep in mind, as described below, that lab exercises are mandatory and need to be signed off. Therefore you might as well treat all sessions as mandatory.

Signing off Mandatory Exercises


Most exercises in the lab sessions are mandatory to be signed off. This is in part to ensure that the exercises are performed in the way that we teach the material, and in part to assess your understanding of the materials.

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

Skills Deadlines of Skills submissions are communicated via Canvas.

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 topics) 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.

There are only so many Lab sessions. The deadline for Design lab exercises is in week 4.

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). In order to remain on schedule, all exercises for a week for should be signed off on the Monday of the next week.

- Exercises are signed off **in order**.
- Only one sign off point at a time; you should not wait with signing off.
- Student pairs are signed off **together**. If your partner is missing, you cannot sign off.
- Signing off happens **only during lab sessions**.

Please study the chapter on pair programming (page 9) for the explanation of **strong-style pair programming** which you are expected to use during the Programming lab sessions.

The Programming lab sessions on Monday are especially dedicated to signing off; during the other practical programming sessions, questions have priority over signing off.

There are only so many Lab sessions. The deadline for Programming lab exercises is in week 8.

Students are expected to work in the same pairs in both the programming and design lab sessions. Signing off during lab sessions will be done via **Horus** and **Discord**.

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.

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 (module2-tcs@utwente.nl)

During the module, there will be two intermediate module evaluation sessions, *probably* in weeks 3 and 7 during the Tuesday lunch break. These are organised by CEEP. 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 Hoo-gesteger, Sophie Lathouwers, Renate van Luijk, Laurens Rouw, Jip Spel, Leon de Vries, Wim Kamerman, Remco Abraham, Rick de Vries, Andrei Popa, Silas de Graaf and Daniël Floor, Joris S. Kuiper I, Alexander Stekelenburg, Brianna Dringa, Anissa Donkers, Irvine Verio, Valeria Veverita, Zhang Fay. 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.

Pair programming

During the Programming lab sessions, we ask that you work primarily using the **strong-style pair programming** approach. You can also use this approach for the project.

Motivation for pair programming

Roughly speaking, we could recognize different styles of working together:

1. One student does the exercise, the other student watches passively
2. Both students work individually, occasionally asking each other questions
3. Traditional pair programming where the driver works on the details of coding and the navigator keeps track of the bigger picture and checks for bugs
4. Strong-style pair programming where the navigator gives instructions to the driver

The first two styles have obvious downsides, as there is a real risk that one student who is a bit behind is only getting further behind. There also is less cooperation or shared learning between the partners.

Writing code is only one aspect of programming; it is more important to think about what you are going to code and the best way to do this when you work in a team is by organizing your thoughts and plans together. If done correctly, the result of pair programming is higher software quality obtained in less time. Pair programming is also used in the industry. It is in fact a skill that requires some effort to learn. Thus if you learn this technique now, it becomes a tool that you can use in the future.

Traditional pair programming

In traditional pair programming, two people share the same computer. One person is the **driver** and the other person is the **navigator**. Just as with a car, the driver is concerned with the details of operating the computer writing code, while the navigator keeps an eye on the bigger picture. The driver explains their actions as they write the code, while the navigator thinks ahead, asks questions and performs real-time code review, checking for errors.

In traditional pair programming, if the navigator has an idea, they take the keyboard and become the driver.

Strong-style pair programming

With strong-style pair programming, the driver essentially implements high-level instructions from the navigator. When the driver has an idea, they hand the keyboard to the navigator and swap roles. To produce code, both developers must be actively involved.

The driver and the navigator have distinct roles.

The **navigator** has to pay attention to the driver and give appropriate instructions: not too detailed, not too vague. They should guide the driver when necessary and enable them to work at their pace. While the driver is working, they should already think about the next thing the driver should do, and plan a few steps ahead. The navigator should consider the list of things to do and what would be the right order to do these things. The navigator never grabs the keyboard to implement their idea: they work through the driver.

The **driver** should not merely type what they are told to type, but be actively engaged: Ask questions if you feel something should be different; talk back to the navigator if they are too vague or too detailed; you can add your own intelligence to how you execute the instructions from the navigator. At the same time, the driver needs to trust the navigator and it can even be helpful to try it one way, then afterwards switch roles and try it another way. For example, the driver could agree to go with the navigator's direction for 10 minutes and then evaluate the result. Sometimes it can be better to hold a question so that the navigator's flow is not broken; however, if the driver does not understand what they have been doing, they should ensure they understand it afterwards.

When working together, one person is a navigator and the other the driver. Typically, whoever has an idea should navigate. Sometimes one person is more experienced than the other and in that case, the more experienced person often starts as the navigator. It is also important to switch regularly, either after a fixed amount of time or after completing a task.

Advantages of strong-style pair programming

Strong-style programming solves a number of problems with other ways of working together.

- With strong-style pair programming it is literally impossible if one student struggles with the exercises, that the other student grabs the keyboard and finishes the exercise.
- It completely prevents the navigator from disengaging and becoming distracted.
- It solves the expert/novice problem when the expert is the navigator and the novice is the driver. The expert is not slowed down by the novice, the novice obviously benefits by learning from the expert, and the expert benefits from being able to think at a higher level in the role of the navigator. In fact, strong-style pair programming is a solid method for onboarding new developers on a project.

There are more advantages:

- Different people have different perspectives. Code is always a result of two different people thinking about the program. The strong style forces thinking aloud, so an idea goes from thinking to talking, is then understood by the driver and translated to code, after which the navigator can review the result. This increases the quality of the code and reduces the number of errors.
- Collaboration is better when both partners are involved in the code, rather than having worked on separate parts of the program. Especially with the programming project, this avoids that one person has not seen half the submitted project.
- There is less procrastination because it is easier to stay active.
- Practicing strong-style pair programming also practices a number of other useful skills: giving and receiving instructions, giving and receiving feedback, communicating clearly about design decisions and implementation choices, working out disagreements.

General tips

- Communication is key. With strong-style pair programming it is literally impossible to work in silence. With traditional pair programming there is a risk that the navigator gets bored or distracted when the driver does not engage the navigator with what they are doing and why.
- When doing pair programming remotely, you can use plugins such as Code With Me in IntelliJ, so that you can still work on the same computer.
- Take regular breaks, because working as a pair in this way can be quite intensive. Breaks can be useful to take a step back, to give feedback to each other, to take a walk and to stay hydrated.
- Remember that pair programming is a skill to learn and simultaneously involves building a relationship with your partner. In the end, you need to adjust to each other to form a good team. This also means that it can take some time to find the right approach for the collaboration. Give each other feedback on the communication and pay attention to the feedback from your partner.

Rules during Module 2

The implementation of pair programming in Module 2 is as follows:

- Students are expected to use strong-style pair programming during the lab sessions.
- Students are required to sign off in pairs. Only students who are temporarily without a partner may sign off individually, or students who are behind after getting a new partner.
- Teaching assistants ask questions to both partners when signing off; if one person does not understand the solution, then this is an indication of poor collaboration.
- Pairs are formed by the teachers and the house team; students do not choose their partners. We try to pair students together who have a similar skill level; for first year students we pair based mostly on the grade for the Algorithms pearl in Module 1; we pair resit students with other resit students.
- If you are unhappy with your partner, you can ask your group TA for a change under specific circumstances:
 - A partner who is structurally unavailable
 - A persisting conflict
 - A problematic skill/experience difference

In all cases, your group TA is instructed to first find a solution that does not involve changing the pairs.

Design project

Make a design for a medium-sized system. You are expected to 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 15–15, 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: Tarp learning management system

The Canvas learning management system is supposed to be replaced by its fierce competitor Tarp. You are in charge to design Tarp based on the following project description. In general, the learning management system is supposed to be available for stakeholders on a day-to-day basis.

Account Management System. To prevent students from entering and editing their own grades, Tarp needs to be able to manage different roles and users. This includes course relevant staff members, such as the module coordinator(s), teaching staff and teaching assistants (TAs).

For GDPR (General Data Protection Regulations) reasons, the system should not store unnecessary personal data of its users. However, at least, the system should store the names and email addresses of users (such that identification and communication is possible). For students, the student number (s-number), and for staff the m-number shall be stored, as this is required for identification to other university bodies like the exam office.

We assume that the exam office initially creates a course and assigns a module coordinator to it. The module coordinator then invites other users to the course. In general, a user can promote another user at most to their own role according to the hierarchy: “module coordinator > teacher > TA > student” A TA may invite users as TAs or students and a teacher may invite users as students, TAs, or teachers, etc. Removing users from a course can only be done by users with a higher role in the hierarchy: the module coordinator can remove teachers, TAs, and students; TAs can remove students, etc.

Notably, these roles depend on the courses, and thus, roles may differ with different courses. To illustrate, a person might be a TA in a first year course they took last year, but they might currently be enrolled as a student in a third year course themselves. Similarly, to keep up to date, teachers sometimes also register as students of courses offered by the Centre for Educational Support.

Creating Assignments and Uploading Files. Teachers should be able to create assignments for their course. The assignment should contain a textual description of the task, as well as (optionally) some attached files, e.g., source code templates or sample data.

Moreover, assignments come with a deadline and a closed-for-submission date, so students can plan accordingly. Submissions after the deadline will be flagged as “late”, and no submissions are possible at all past the closed-for-submission date.

The teacher can allow students to re-submit before the closed-for-submission date, and the system will only record the last submission. Also, the teacher should be able to limit the file types whenever an upload is required, e.g., a project may require a program in `zip` format or a write-up in `pdf` format.

Sometimes, teachers would like to create an assignment, but don’t want to share it with students right away. Rather, it remains hidden from them, until it is published. This could be the case when they want co-teachers or TAs to have a look at the assignment before it finally gets rolled out. The creator of an assignment also decides if other teachers of the same course or even TAs can modify the assignment. At the very least, other teachers shall be able to attach comments to the assignment.

To track progress once a submission was uploaded, indications are given to a student. A submission can be under correction, graded with hidden grades, and graded with published grades. The latter is to ensure fairness, so that all students receive their grades at the same time.

Auto-grading. In particular for programming assignments, teachers may want to use (semi)-automatic grading. Note that the tools used for such grading are not part of the system, and teachers will most likely want to use their own tools. Thus, Tarp provides an easy interface to use such tools. A minimalist solution is to allow the teacher to (bulk) download all submissions, annotate them with whatever tools they want, and then (bulk) upload the annotated submissions and associated grades. A more complete solution also allows for test-case reports, where the results of an external automated test suite are forwarded to the students immediately, such that they can still change their submissions before the deadline has passed. Optionally, the system should record how many submission attempts a student has made, and use this number for either reducing the grade or limiting the number of allowed submission attempts.

Plagiarism Checker. Tarp shall support plagiarism checkers. The checker needs not be part of the system directly, but an interface to communicate with such a tool should be present in Tarp. Thus, similarly to the auto-grader, all submissions must be made available (in bulk) to the plagiarism checker.

Note that passing on student submissions (potentially linked to student names) is a serious privacy issue if the plagiarism checker is from an external company. Thus, Tarp shall provide a design that leaks as few information as possible to the external tool.

If an assignment submission is under suspicion of plagiarism, it is flagged for a teacher to inspect it closer, and the report of the plagiarism checker is attached to the submission.

Grade management. Tarp is supposed to facilitate easy grade management for all parties:

Teachers and TAs should be able to grade submissions. Thus, to ensure a fair grading, they have to see all grades of their courses. While TAs can only change unpublished grades they gave themselves, teachers can change any grades of their course. Changing a published grade, though, requires a textual justification that is stored with the grade change. (e.g., “answer model changed”, or “question 3 was removed from the assignment/exam”)

Students should be able to see their own grades only. Grades will be hidden first, and the teacher will publish all grades simultaneously, to ensure that all students receive their grades at the same time.

Submissions that come with an exam review have a preliminary and a final grade. The final grade is set by the teacher after the exam review. Usually, it will be the same as the preliminary grade. As for the preliminary grade, it is first hidden, and then published for all students at the same time.

At the end of the course, an overall grade is computed from the individual submission and exam grades. The overall grade is entered by the teacher manually, or computed automatically by the system according to rules set by the teacher. The teacher can manually override an automatically computed grade, but this will be logged with a justification text.

When all grades are final, the overall grades can be submitted to the exam office. From this point, no grade changes are possible in the system (those would be handled by the exam office now).

The students shall be notified about any grades and changes when they become visible to them. While usually the grades of all students are published/submitted at the same time, exceptions for individual students must always be possible, e.g., if an individual deadline extension has been granted, or grades are held back due to a pending fraud case.

What to hand in

The final project deliverable should contain the following items (the [LxTx] marks refer to the relevant topics you might find helpful to revisit):

- 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 [L2T1] conducted in Week 1 or 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.
- A list of stakeholders [L2T1] and actors [L5T1].
- Use case diagrams [L5T1] for the complete system, with brief descriptions.
- Extended user stories [L2T5] of representative scenarios (at least 2).
- A list of functional and non-functional requirements [L2T1].
- A test strategy (with testing objectives and defect classification) and a test plan [L2T5].
- Activity diagrams [L4T1] describing relevant business processes (at least 2).
- A complete class diagram [L3T2] of the system.
- Sequence diagrams [L4T2] of relevant system processes (at least 2).
- Appropriate state machine diagrams [L4T3] (at least 2).
- A list of metrics [L6T1] suitable for measuring the quality [L6T5] of the system, and possible appropriate conventions [L6T4].

All the deliverables from this list should be *appropriate*. What is appropriate? In this context it means that you should choose processes and objects worth modelling, because they are not entirely trivial, and having an explicit model helps to understand exactly what the system should do. In other words, as a design effort, it makes sense to model them.

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. 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. **Repaired Design projects have a maximum grade of 6.0.** Late submissions receive a lower grade, with a reduction of 1.0 point after the deadline has passed, and a further reduction of 1.0 point every 24 hours after the deadline has passed. For example, if you submit 30 hours after the deadline, the grade will be reduced by 2.0 points.

Important Dates

Week 5	Sun 23:59 CET	Submission deadline; 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.

Rubrics The project is graded according to the rubric that can be found in appendix C (page 93 onwards).

The rubrics consist of five columns, where each column corresponds to a specific grade (indicated at the top of the rubric). For each row, the cells indicate what is required to get the corresponding points on that component of the grade. If multiple cells are applicable, the grade corresponding to the *leftmost* applicable cell is selected.

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

During the module, you have learned all the tools you need to build a high quality software product. In this project, you will be putting this knowledge in practice by developing your own software product. Together with your partner, you will be developing a client-server multiplayer game. The game description will be available on Canvas when the project starts. The goal is to demonstrate the design and programming skills you acquired.

The game you will be developing will be based on a client-server architecture. This means that you essentially have to deliver two products: a client and a server. The server controls the game, and the client serves as one player of the game. Of course, the client and the server will need to be able to communicate with each other. Since it would be much more fun if your client and server could not only communicate with each other, but also with the client and server of your fellow students, we have already designed a communication protocol for you to use. You can find the protocol on Canvas when the project starts. In order to verify whether your game adheres to this protocol, a pre-compiled reference client and server is available on Canvas by then as well.

Since your client and server should be able to communicate with all other clients and servers developed by your fellow students, there is an opportunity for competition. You should not only be able to play the game as a human, your client should also be able to play by itself! This means your client needs to have a computer player. At the end of this module, there will be a tournament to determine who has the best computer player. As an added incentive, the winner and runner-up per house will receive bonus points on their final grade.

In order to achieve this goal of having a fully functional client-server game implementation, we expect you to adhere to all coding standards and good practices you have learned throughout this module. We expect you to deliver a preliminary design at the start of the project and a complete (design) report at the end. We expect your code to be structured in a sensible way, by distinguishing different responsibilities and by designing and implementing the software such that individual parts like the user interface could be changed without affecting the rest of the code. We expect good documentation and good application of the object-oriented programming concepts that you have seen throughout this module. Finally, we expect you to have extensively tested your product according to a clear plan in your report. Testing is a very important part of your grade. You are expected to write automated unit tests for important parts of your project, especially to ensure that your implementation of the game logic is correct. You are also expected to write system tests: instructions for a developer to test certain aspects of your software.

The next sections will go more into the details of the requirements and the grading criteria. Detailed grading criteria can be found in appendix D (page 99 onwards).

Detailed description of the client and server functionality

Client-server architecture As mentioned above, you will build a separate client and server. Two clients can connect to the same server, after which a game can be initiated. As the game is played, updates (e.g. executed moves) are communicated between client and server via the predefined protocol. Using this information, both clients keep track of the state of the game, and can prompt the user for possible moves via a user interface. Both client and server should verify the legality of a move before sending it over the network. Only valid moves may be sent. When a game is finished, the users should be able to play a next game immediately after: client and server should not need to re-establish a connection.

The protocol must be followed exactly. It is not allowed to add commands. It is also explicitly not allowed to create a “thin clients” where the user writes the communication messages with the server.

Server functionality It could be possible that multiple distinct servers run on the same computer. For this reason, the server should prompt the user for a port number when it is started. When the port is already taken, it should ask the user for an alternative port to try again. Once initialized, no interaction between user and server should be required to play games. Furthermore, the server should support many players connecting to the server and playing games simultaneously.

Client functionality In order to connect to the server, the client should request the server IP and port from the user when starting. Additionally, the user should enter a “username”, which is used for various purposes in the communication protocol. This name should be unique in the server: if the name is already taken, the server will send an appropriate reply and the client should ask the user for an alternative username. Once connected, the user should queue for a game, wait for an opponent and then start playing. Ideally, the TUI also helps the user where possible, for example by having a “help” command to see a list of available commands. When it is the player’s turn, the user should be able to enter a move or request a (legal) move as a *hint* from the client.

Your client TUI should be designed with novice players in mind, for example your family members or friends should be able to play a game using your client TUI. Your implementation should generate protocol messages instead of the user; the user should not even be aware that there is a specific protocol that the client uses to communicate with the server.

Computer players You are required to integrate an AI into your client, which can play the game instead of a regular human behind a keyboard. This AI should have an adjustable difficulty level, you should have at least two difficulty levels. The user should be able to indicate who should decide on moves (AI and corresponding difficulty level, or a human) via the TUI at the start of the game.

Expecting the unexpected Unfortunately, you cannot expect all parties to behave as expected. There might be clients and servers that do not adhere to the protocol. This might be because of an implementation mistake, or a malicious attempt by someone else’s server or client to cheat. When playing a game using your client on a server not adhering to the protocol (or vice versa), your product should not crash under any circumstance. This means that your product (both client and server) should be able to handle randomly dropping connections, malformed networking messages or illegal moves without crashing themselves.

Sometimes network connections break down resulting in an unintended lost connection. When a connection to a server drops, your client should inform the user what happened and then terminate cleanly (e.g. without some thread dumping a stacktrace or hanging forever). When a connection to a client drops, your server should inform the other client (when in a game) and continue operating normally. It is *never* allowed for your software to crash or hang due to getting unexpected (network or user) input.

This is also related to the concept of *defensive programming*: eliminating as many foreseeable problems as possible so that the software remains functional under unforeseen circumstances.

Description of other code requirements

Design and quality of code The stability and the maintainability of your software is heavily dependent on the design of your program, both the high-level architectural design into components and packages, as well as the low-level detailed design using design patterns and the principles of object-oriented programming. Your code should make use of design patterns that we teach in the module, in particular you should make use of Listener patterns as well as the Model-View-Controller pattern such that someone could write a new user interface without modifying the rest of your program. Your code should follow the practices that we have seen during the module: proper use of inheritance, encapsulation, abstraction and composition, as well as following Java code conventions (as enforced by Checkstyle). In addition, you should make use of (custom) exceptions where appropriate.

Tests In order to guarantee smooth running of the software, as well as prevent bugs in the future, proper testing should be done. You are expected to write automated JUnit tests, covering as much of your codebase as possible, but at least the implementation of the game logic. Start with writing tests for the game logic, and continue adding tests for other parts of your system afterwards. Furthermore, you should keep good testing practices in mind: make sure each test only tests one clearly defined piece of code or one clearly defined case. In addition, we expect you to write system tests in the report (see below).

Documentation Like any piece of software, proper documentation should be written to aid future readers in understanding the code. For Java, this comes down to having proper Javadoc for classes and methods, but also helpful in-line comments to explain *why* something is done the way it is¹. For the classes in the game logic, you should also add proper pre- and postconditions to the Javadoc documentation. In practice, you should write Javadoc before implementing classes and methods and you should write comments while writing code. Writing documentation helps to think before you code. Similarly, you are asked to specify your code, in particular the game logic, using JML preconditions, postconditions and invariants.

Description of the report contents

The source code is not the only deliverable for the project. You are also asked to submit a report, in which you explain the architecture of your software. Additionally, you are asked to elaborate and reflect on various decisions that were made before and during the implementation, as well as the testing methodology. The paragraphs below explain what sections your report should consist of, and how they relate to each other. For the sections related to the design of your code, your target audience is a future software maintainer. Focus on what would be important to someone who might need to adapt your code in the future.

Explanation of realised design In this section, you explain the design that you realised in the end. It is your objective to guide the software maintainer through the classes and packages that your project is composed of, and (where necessary) to explain what a class does and how it relates (i.e. what purpose it serves) to other classes. To give this explanation some structure, we expect you to create a list of responsibilities (e.g. “manage the game board”) and a list of classes and the relationship between the responsibilities and the classes and to explain your design choices. You should also justify how you divide your program into different components and packages. You should use class diagrams to aid your explanations where appropriate. These class diagrams only need to display a portion of all classes. As an example, you can have a class diagram with all model classes before the actual explanation, so that the explanation can go over the diagram in a clear way. It is fine if the same class appears in multiple diagrams. The class diagrams should show associations between classes. In addition, you should use sequence diagrams to show how classes cooperate during certain execution flows, such as processing a particular user or network input. Feel free to pick other execution flows, however: whatever is most appropriate for your project. After all, your goal is to explain all major design decisions to a future maintainer of the code.

Concurrency mechanism Both your client and server will make use of multiple threads to function properly. As a consequence, concurrency issues could pop up in multiple areas in both applications. In this section of the report, you should explain where new threads are created and what purpose they serve. You should also identify all fields that could be accessed by multiple threads, what could potentially go wrong, and what you did to ensure that no problematic race conditions occur. Be specific and concrete: merely stating “we use locks everywhere” is not a sufficient explanation. Since a simple concurrency design is often the best, this section does not have to be particularly long, but it should answer all questions about concurrency in your application. It could be that you discover a flaw in the design, which is discovered so late that it cannot be resolved before the deadline. In that case, you should explain what the issue is, and how you would have solved it, if you had more time.

Reflection on design As opposed to the “explanation of realised design”, this section should contain things that you did *not* end up doing. Compare the realised design with the design you submitted at the start of the project. In what way did things improve? Why were these changes necessary? What part of the initial design worked out well? You should also look ahead: if you were to make this project again, what part of the design would you do differently? Why could you not make such improvements during the project? You need to reflect on the initial design that you made in the first project week, and on the final realized product. This section should be about possible program design improvements, not about what could be improved in the process.

¹What is explicitly *not* requested, are comments explaining *what* is being done. Your function/variable naming should already make that clear. Reasons for doing something are much more helpful, especially when those reasons are not immediately obvious.

System tests Many tests cannot easily be fully automated. One category of tests for which this is particularly true is system testing. This section should systematically describe procedures to test the system as a whole (client and server either individually or cooperatively). The tests should be reproducible by a future maintainer with no pre-existing knowledge about the software. Each system test is essentially a list of instructions that any future maintainer could perform to test part of the product. You should also list the expected outputs for the different steps. Focus on visiting all features of the client and server, including the “bad flows”, such as disconnections and unexpected inputs. See also appendix A (page 79 onwards) for more information on system testing.

Overall testing strategy The system tests above are only one part of your testing strategy. There should be a strategy involved in testing the different parts of your system using your unit, integration and system tests. In this section, it is your goal to convince the maintainer that your testing strategy has led to a stable application. The recommended way to do so is by combining the results from code metrics (e.g. test coverage or class complexity) with the contents of your tests in a concise manner. Explain why the tests are structured the way they are, and why this leads to a stable product. Explain how one test covers what another test does not, why certain features are tested the way they are, and why there is a focus on a particular area. Undoubtedly, you will have tested some part of the system less than others. Explain what this component is, and why (despite that) your testing strategy is still a solid one.

Reflection on process This is, quite likely, the first time that you have made a software project of this size. As such, it is only natural that not all things went as well as predicted. Perhaps some activities should have been done earlier, before or in parallel with some other activities, or perhaps there was an issue in the collaboration process somewhere. In this section, you should describe the positives and negatives of your initial planning and the collaboration. You should mention how to improve on the things that did not go well, as to prevent similar issues in future projects. This complete section should be done *individually*. Your report should clearly indicate which group member produced which reflection. You are also asked to provide a rough estimation of how the work was performed: who did what, which parts were done together, which parts were done individually.

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 additions to your networking code, as described in the protocol documentation for these extensions. You should make sure your client and server still correctly work with the servers and clients from different pairs, after the extended functionality is implemented.

Challenge (max +0.3 points)

While playing with a random opponent is the default method, sometimes you may want to play against a specific opponent. One extension described in the communication protocol allows challenging another player. If the challenge is accepted, the server creates a new game with that player.

Chatbox (max +0.3 points)

A game application implemented as described above can only be used to play the game. It would be nice 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 a global chat or to individual players.

Ranking (max +0.3 points)

The server could maintain a player ranking, acquired by maintaining statistics about the performance of players on the server. Possible statistics are winrate, total amount of wins or more advanced metrics such

ELO-rating. It is acceptable if the calculation is done using only data about the period that the server is currently running: persisting the history to storage to “survive” reboots is not required.

Security (max +0.4 points)

Since a client can freely choose its username, a client could pretend to be someone else (for example, to influence the ranking). One way to prevent this is through authentication. The protocol describes a way to authenticate a player using public-/private-key algorithms. This key should be stored locally, to ensure that the same key-pair can be used for subsequent connections from the client. During connection initialization, the server verifies the authenticity of the client using a challenge.

While in principle nothing secretive is communicated, it might be desirable to add encryption to your client and server to ensure message confidentiality. The protocol describes extra commands to allow for shared key negotiation, which prevents eavesdroppers from listening in on your client-server communication.

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*. We do not look at other attachments such as images of class diagrams that are not in the report.

You can generate your implementation archive in INTELLIJ using the `EXPORT...` item from the `FILE` menu, and choosing `PROJECT TO ZIP FILE`. The ZIP archive with your implementation should contain the following:

- A directory `src` containing all source files, stored in a single directory hierarchy.
- (Optional) A directory `test` containing all source files of the unit test classes, if you use a separate test sources root.
- A directory `docs`, containing the documentation of all self-defined classes (HTML pages generated with `JavaDoc`) in a directory hierarchy that is separated from the source files.
- (Optional) A directory `libs` containing any non-standard predefined classes and libraries, to be included as `jar`-files.
- A `README` file, located in the *root folder* of the ZIP archive, containing:
 - Information about building and testing your software, indicating, for example, which directories and files are necessary and which conditions apply, such as versions of libraries.
 - Steps on how to start the game, including example start commands if applicable.
 - See also <https://www.makeareadme.com> for suggestions on writing a great `README`.

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

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 develop the application. For further information, see the documentation for `java.lang.File` and in particular the constants `pathSeparator` and `separator` defined in this class. You could also test your application in a freshly installed virtual machine with an operating systems that differs from your own to be absolutely sure about the compatibility.

Grading

The project is graded according to the rubric that can be found in appendix **D** (page 99 onwards).

Rubrics The rubrics consist of five columns, where each column corresponds to a specific grade (indicated at the top of the rubric). For each row, the cells indicate what is required to get the corresponding points on that component of the grade. If multiple cells are applicable, the grade corresponding to the *leftmost* applicable cell is selected.

Desk reject If a deliverable is missing (source code or report) or required parts (game logic, server/client) are clearly unfinished, then the project is rejected and not graded. The server and client are deemed “not clearly unfinished” if it is possible to use the server with the reference client and the client with the reference server to play a game as a human player and with the AI.

Initial design You need to hand in an initial design in week 8. This assignment is mandatory. The purpose of this design is to get you to think about the design of your project in advance. This initial design will also be used to reflect upon in the report. If you do not hand in an initial design, you automatically fail that part of the report. What exactly you hand in as an initial design and how much you want to design, is up to you. The amount should be sufficient to reflect back upon in your final report. It could for example consist of a class diagram and/or sequence diagrams, a list of responsibilities and a preliminary list of classes and components/packages, perhaps already some descriptions of what threads you are going to need, some description of where you expect to need synchronization/locking to prevent concurrency issues, et cetera. You do not get feedback from teachers or teaching assistants on the initial design; it is meant for reflection in the report. However, it is probably a good idea to submit a design of reasonable quality, not only to improve the quality of your end product but also to be able to do a proper reflection. If you miss the deadline for this, you will simply get 0 points for “reflection on design” in the report.

Midway submission You can submit a partial project in week 9. If you do this before the deadline and the following points are present with reasonable quality, then you get +0.5 bonus points if your project is sufficient for a pass without the bonus points.

- All game rules are implemented.
- All public classes and methods of the game logic have Javadoc.
- Complex methods of the game logic have comments.
- There are unit tests for the game logic, including at least testing whether a move is performed correctly, testing a gameover condition, and testing a random play of a full game from start to finish including checking the gameover condition.
- There is an initial version of the report including a section on the overall testing strategy with incorporation of coverage metrics of the game logic and an explanation of the test plan for the game logic.

During grading, the grader will check presence of the above items in the midway submission. If present and the entire project is sufficient, then the bonus point is granted.

Functional Requirements The following requirements are deemed “crucial”:

1. A standard game can be played on both client and server in conjunction with the reference server and client, respectively.
2. The client can play (on a server) as a human player, controlled by the user.
3. The client can play (on a server) as a computer player, controlled by AI.

If the project is clearly not sufficient to fulfill the above requirements, then it is not graded.

In addition, the following requirements are deemed “important”:

1. When the server is started, it will ask the user to input a port number where it will accept connections on. If this number is already in use, the server will ask again.
2. When the client is started, it should ask the user for the IP-address and port number of the server to connect to.
3. When the client is controlled by a human player, the user can request a possible valid move as a hint via the TUI.
4. The client can play a full game automatically as the AI without intervention by the user.
5. The user can adjust the AI difficulty via the TUI.
6. Whenever a game has finished (except when the server is disconnected), a new game can be played without needing to establish a new connection in between.
7. All communication outside of playing a game, in particular the handshake and feature negotiation, works on both client and server in conjunction with the reference server and client, respectively.
8. Whenever a client loses connection to a server, the client should gracefully terminate.
9. Whenever a client disconnects during a game, the server should inform the other client(s) and end the game, allowing the other player to start a new game.

These important requirements are part of the functionality grade in the rubric. We suggest that you also write system tests in the report to test these requirements.

Grade calculation The grade G will be between 1.0 and 10.0, and calculated using the formula $G = \min(10, \max(1, 0.2 * F + 0.4 * S + 0.4 * R + B))$, where:

- F is the functionality grade, calculated as a weighted average using the percentages from the functionality rubric,
- S is the software grade, which is calculated in an identical manner using the software rubric,
- R is the report grade, which is calculated in an identical manner using the report rubric, and
- B is the sum of the accumulated bonus points or 0 if $0.2 * F + 0.4 * S + 0.4 * R < 5.5$.

Activities and important dates

Tournament 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 scoring high in the tournament within your house. A full point is earned by the winner, 0.5 points is earned by the runner-up. The bonus is added to the grade in a similar way as extensions, i.e. as a flat bonus to the overall grade provided that the grade is passing without the bonus (part of B in the grading formula).

Important Dates The following table lists the several important dates for the project.

Week	Day	Hour	Activity
Week 6	Tue	3–4	Kick-off lecture on the programming project
Week 7			Discuss project planning with student assistant
Week 8	Wed	23:59 CET	Initial design deadline (via Canvas)
Week 9	Wed	23:59 CET	Midway submission deadline (via Canvas)
Week 10	Wed	23:59 CET	Submission deadline
Week 10	Fri		Tournament (schedule announced separately)

Suggestions We have a few tips to get you started.

- Read the entire project description and the rubric first
- Use the rubric as a checklist before submitting
- Write Javadoc before you implement classes and methods
- Write comments while coding, not afterwards
- Use Checkstyle from the start
- Programming is a team effort. Use the group contract to make an agreement with your partner on expectations and how to do the project. Meet daily to discuss progress.
- While pair programming is not mandatory for the project, we recommend that you use it.

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).

Design The activities in this week cover the following topics

- Level 1 Topics: Q&A session on [L1Tx].
- Level 2 Topics: Q&A session on [L2Tx].
- Project: getting started, see Section 1.2.1.
- Lab: S/P/E systems, their stakeholders and requirements, see Section 1.2.2.
- Lab: Test policies, strategies and stories, see Section 1.2.3.

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.
- Debugging: setting breakpoints, stepping, inspecting

1.1.1 Mandatory Presence

During the following activities, your presence is mandatory.

- General introduction session (Mon 1–2)
- Tool installation session (Mon 3–4)

1.1.2 Expected Self-Study and Project Work

In addition to the scheduled activities (which include watching topic videos), we estimate that you need approximately:

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

1.1.3 Materials for This Week

Tool installation: See Canvas (“Course Materials”).

Design:

Watch videos [L1T1], [L1T2], [L1T3] to get started.

Watch videos [L2T1], [L2T2], [L2T3], [L2T4], [L2T5], [L2T6].

Programming: ECK, Chapters 1–4; the week 1 topic videos

Laboratory: The following lab files are provided on Canvas.

- checkstyle-ss-2020.xml
- src/ss/week1/BrokenFibonacci.java
- test/ss/week1/BrokenFibonacciTest.java
- test/ss/week1/FibonacciTest.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.

1.2 Design

1.2.1 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 start planning. Watch the topic videos beforehand as they will help you in this.

- Study the description of the design project and the case description of the Tarp learning management system (page 13). Think ahead on what will be the consequences of the project details on different phases/elements of your system lifecycle [L1T2].
- Within the first two weeks you are expected to interview [L2T1] a relevant stakeholder, see Section 2.2.1. The sooner you get into contact with the relevant representative, the better, if you want to have the interview at a time that suits your group. Reach out now to make an appointment for this week or next week!
- Make a list of stakeholders [L2T1] of the Tarp learning management system.
- Identify the core functionality of the system, and write down its main functional requirements [L2T1].
- List non-functional requirements relevant for this system and its stakeholders [L2T1].
- Sketch a test plan [L2T5] of your project—how can you make sure that your design will be good?

1.2.2 Laboratory exercises on modelling

All laboratory sessions, unless specified otherwise, are done in groups of two students.


Whenever you have completed an exercise marked , please ask a teaching assistant for feedback.

 **D-1.1** Classify each of the following software systems either as an S-system, P-system or E-system as defined by Manny Lehman and explained in [L2T4]. Give a brief justification for your classification of each system.


Example: Coffee machine: S-system - users selects drink, machine makes the drink following given recipe

- An automatic sudoku solver.


- An interactive sudoku playing app.
- A student calendar app with scheduled lectures.
- The Tarp learning management system (p. 13).
- Untappd (a social network that tracks which beer you drank and shares it to your friends).
- Airbnb
- A PDF reader.
- A compiler for a textual programming language like Java.
- A compiler for a graphical modelling language like UML.
- An augmented reality set.

 **D-1.2** Go through the list of the systems in **D-1.1** again, and discuss what requirements [**L2T1**] will be in place if you were *forced* to make each of them as an S-system.

Example: Coffee machine: No changes, as it is already an S-system.

 **D-1.3** Go through the list of the systems in **D-1.1** again but this time, envision each of them as an E-system.

Example: Coffee machine: some account management system could be implemented such that the machine keeps track of customer's previous orders and can recommend new drinks based on this data.

 **D-1.4** Go through the answers you have for **D-1.1**, **D-1.2** and **D-1.3** and choose 3 systems from each question. Describe which prescriptive, descriptive and predictive models [**L2T3**] could be useful in the process of creating those systems. Try to use models from each prescriptive, descriptive and predictive at least once.

Example:

Coffee machine -> Prescriptive model : recipe tells the machine how to make the drinks

Washing machine -> Descriptive model: manual teaches users how to operate the machine

*project -> Predictive model : effort estimation predict how much time will be needed to successfully design the system

1.2.3 Laboratory exercises on testing

 **D-1.5**

Go through the answers you have for **D-1.1**, **D-1.2** and **D-1.3** and choose 3 from each question. For each of those, propose a user story [**L2T5**] or two, to test the envisioned requirements [**L2T1**].

Example:

Coffee machine

1. As a user, I want to be able to see the list of available drinks from the machine, so I can decide which drink I want
2. As a user, I want to be able to make cashless payments, so I do not need to bring cash with me

 **D-1.6**

Choose 3 of your favourite designs from the answers to the previous exercises. For each of them, sketch two different test policies [**L2T5**] that could be in place in a big governmental organisation and a small private startup, implementing the system. Compare the test strategies that the engineering project for

these software systems could have, based on different test policies.

Example:

Coffee machine

Big organisation - “Have a test coverage of 99% and complete integration and unit testing before deploying new version every Friday”

Small organisation - “Ensure unit testing is done on every new updated class that is pushed in the repository”

1.3 Programming

1.3.1 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. To make a JAVA program, you really only need a simple plain text editor.

To run a JAVA program, the Java source code is first *compiled* to so-called bytecode. Bytecode is a version of the program that a Java Virtual Machine (JVM) can run, regardless of whether you use Windows, Linux, or some other operating system. Bytecode is not meant for humans to read; it is meant for computer programs such as a JVM to read and run. While source code in JAVA ends with the file extension `.java`, bytecode files typically end with the file extension `.class`. During compilation, the JAVA compiler checks for various errors in your code, and if there are no compile errors, you can run your program.

After learning how to manually compile and run a simple program, you will learn how to do this using INTELLIJ as an Integrated Development Environment (IDE).

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

P-1.1 For this assignment we will use the `c:\softwaresystems\java` directory created in the Installation Session. This directory will function as your *working directory* for now.

- Within the `src` folder in your working directory, create a subdirectory `ss`, and inside this directory another subdirectory `week1`.
- Open a simple text editor. In Windows, this could be Notepad or Notepad++ (recommended). In MacOS, this could be TextMate or Sublime Text.
- 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!");
    }
}
```

- Save this file inside the `week1` subdirectory with the filename `Hello.java`. (The name of the file is the same as the name of the class defined in the file. Classes are a topic of later weeks.)
- Confirm using your Files Explorer that there is now a file with the name `Hello.java` in the `week1` directory. Be aware that in some operating systems, such as Windows, file extensions such as `.java` are not always visible.

- Open a terminal shell or command-line window and change your current working directory to the `src` directory. In Windows, you can also do this by navigating to the `src` directory in the Windows Explorer and typing the command `wt` or the command `cmd` or the command `powershell` in the Address Bar and pressing *Enter*.
- *Compile* your program by typing¹

```
path\to\jdk\bin\javac ss\week1\Hello.java
```

where you replace `path\to\jdk` with the location of your JDK. IntelliJ installs JDKs in the `C:\Users\[username]\.jdk` directory, for example `C:\Users\Tom\.jdk\temurin-11.0.13`. If you installed a JDK independently of IntelliJ, you can also use the path where you installed that JDK.

- Inspect the subdirectory `src\ss\week1` again. What happened?

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. See also ECK Sections 2.6.6 and 4.5.2 about Java packages.

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

- In a command line window inside the directory `src`, type the command:

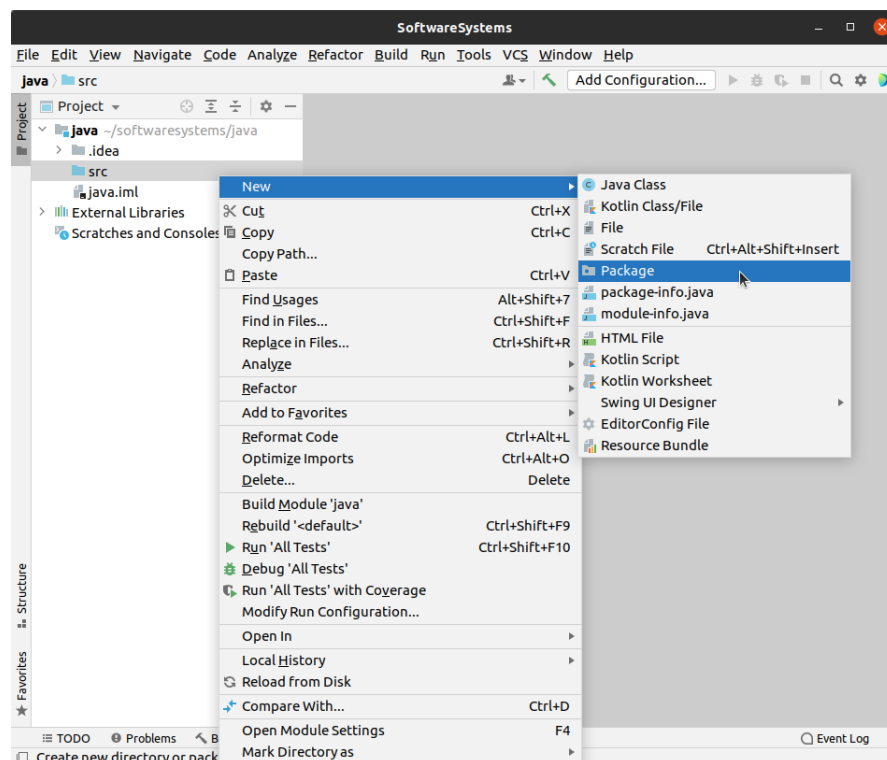
```
path\to\jdk\bin\java ss.week1.Hello
```

where you replace `path\to\jdk` again with the location of your JDK. 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?
- Delete the `Hello.class` file, and try the above `java` command again. What happened?

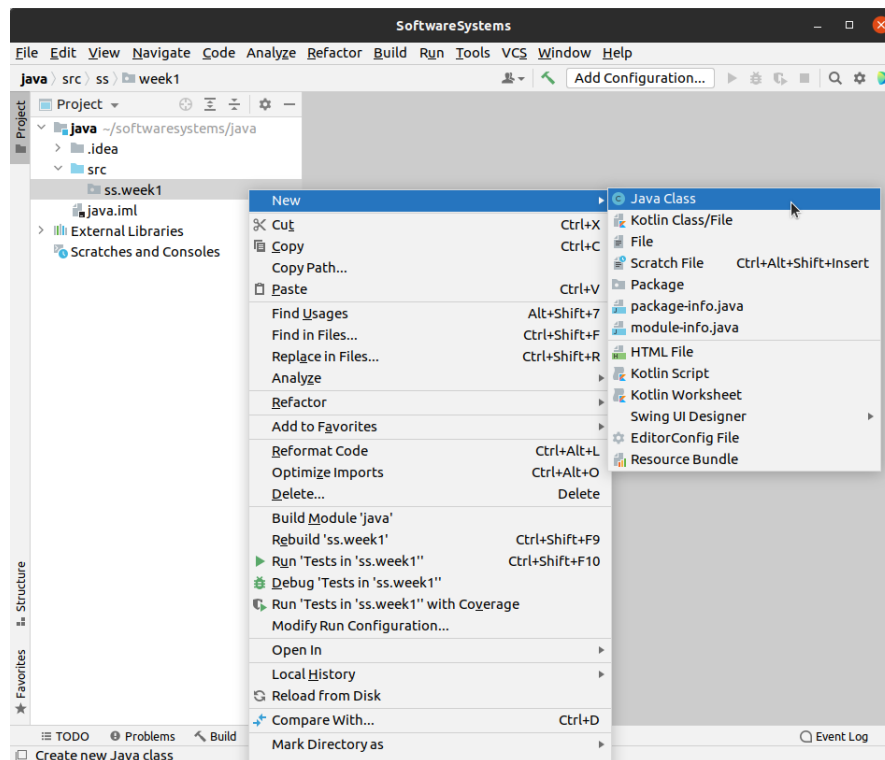
P-1.3 Start INTELLIJ and take the following steps:

- On the left of the window, you should see the PROJECT TOOL WINDOW. Right-click on the `src` folder and navigate to `New -> Package`. Give this new package the name `ss.week1`. This creates a new package called `ss` and within it another new package called `week1`



¹On 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`.

- Right-click on the newly created `week1` package and navigate to `New -> Java Class`. Give this new Java class the name `Hello`.



- Copy the following code into the new `Hello` file

```
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!");
    }
}
```

- Now you will run the class. There are several ways to run a program. For example, you can click the green play icon next to either the start of the class in the editor or the start of the `main` method. You can right-click the class name in the Project window and choose the option to Run the program. You can also use the Run menu in INTELLIJ to start running or debugging a program. Here you can also edit so-called run configurations if you want more control over how your program will be run. In the top right of the INTELLIJ window you can see what run configuration is currently selected. A run configuration can be run by pressing the green arrow next to it. This can be also be done with the shortcut `SHIFT+F10`
- The output of the program execution will be printed in the INTELLIJ CONSOLE view on the bottom of the window.

P-1.4 Try to introduce some small errors in your `Hello`. For example, change the package name in the first line of the file 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)?

Values and Variables

In the following exercises, you will practice using simple values and variables, as well as reading input from the user and writing output to the screen.

Make sure you have studied Chapters 1 and 2 of ECK, as well as the related topics on Canvas. Be aware that we are not going to use the `TextIO` class in these exercises. Instead, you will use a `Scanner`. To learn more about this, read Section 2.4.6 of ECK, and <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html>.

P-1.5 In this exercise, you will change the `Hello` program to ask the user for their name and age.

You have seen that you can write output using `System.out.println`. To read input from the user, you can create a `Scanner` using the following line:

```
Scanner input = new Scanner(System.in);
```

Add this line inside the `main` method of your `Hello` class. INTELLIJ will now ask you if you want to import the `Scanner` class. Choose the suggested class `java.util.Scanner`.

As you will learn in later weeks, this creates a variable with the name `input` of the `Scanner` type. The variable points to an *object* of the `Scanner` class. For now, you do not need to understand all the details. What matters is that you can use this `Scanner` object to obtain input from the user.

When you start a new line below the line you just added, and type `input.` (including the dot), INTELLIJ will give some helpful suggestions of things to choose from. Here are a few examples:

- `input.nextLine()` reads the next line (ends when you press *Enter*) and returns it to the caller.
- `input.nextInt()` reads the next value and converts it to type `int`.
- `input.nextDouble()` reads the next value and converts it to type `double`.

Modify `Hello` to accomplish the following:


1. Print to screen asking the user for their name
2. Read a line from input, which (hopefully) contains their name
3. Print to screen asking the user for their age
4. Read the age from input
5. Print a hello message to the screen, including their name and age

After modifying `Hello`, try it out!

Different types of errors

We can distinguish different kinds of programming errors in Java:

- Compile-time errors occur when compiling the JAVA code to bytecode. Examples are syntax errors, for example a missing semicolon or bracket, and type errors, for example trying to assign a `String` value to an `int` variable.
- Runtime errors occur when running the bytecode on the JVM. For example, when the `Hello` program asks for a user's age, and the user types something that is not a number, then the program will crash by reporting a so-called exception and a stack trace describing where the crash happened. You will encounter many examples of runtime errors in the coming weeks.
- Logic errors are mistakes when a program does not behave as intended, and does not crash. For example computing and printing a calculation with a mistake.

 **P-1.6** Try it out! Enter something that is not a number when your `Hello` program asks for your age.

In a later week, you will learn how to properly deal with these so-called exceptions.

P-1.7 (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` that asks the user how many eggs there are, and then prints to screen how many gross and dozens of eggs that is, and how many eggs are left over. A gross of eggs is 12 dozens.

For example, if the user says that there are 1342 eggs, then your program should respond with

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


since 1342 is equal to $9 \cdot 144 + 3 \cdot 12 + 10$.

Debugging

One of the most powerful tools when trying to understand the behavior of your program is called debugging. Especially when your program does not behave the way you intended or expected, it is helpful to run your program in the debugger. In Debug mode, you can

- Run the program step by step.
- Set a *breakpoint* and run the program until that point.
- Look at the current value of all variables in the program (called the “current state” of the program).

See also the Debugging topic on Canvas.

 **P-1.8** Start the program in Debug mode. Similar to running a program, you can click the green play icon, and choose “Debug” instead of “Run”. If you don’t set any breakpoints, then the program runs as normal. However, you can set breakpoints on lines with instructions to make the debugger halt before executing that instruction. In INTELLIJ you can set a breakpoint by clicking next to the line number of a line, after which a large red dot indicates the breakpoint.

- Set a breakpoint in your `GrossAndDozens` program.
- Start debugging your program.
- Use “step over” (hotkey: F8) to run the program step by step.
- Use the debugger to see the updates to the values of the different variables.

Documentation of the Math API

Using a web browser, access the Java API documentation for the *Math* class at <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html>. See also Section 2.3.1 of ECK. Pay attention to the difference between *degrees* and *radians* for methods like `Math.sin`. 360 degrees equals 2π radians.

P-1.9 The area of a regular polygon is

$$\text{Area} = \frac{1}{4}ns^2 \cot \frac{\pi}{n} = \frac{ns^2}{4 \tan \frac{\pi}{n}}$$

where n is the number of sides and s is the length of each side.

Write a program (in the `ss.week1` package) that asks the user for the number of sides and the length of each side, and then prints the area of the n -sided regular polygon.

Example run:

```
Enter the number of sides: 7
Enter the length of each side: 3,5
The area is: 44.51542743901947
```

P-1.10 The fixed monthly payment for a fixed rate mortgage is calculated using an annuity formula. The monthly payment c is computed by the formula

$$c = \frac{r}{1 - (1 + r)^{-N}} P$$

where r is the *monthly interest rate* (the yearly rate divided by 12), P is the amount borrowed, also called the *principal*, and N is the number of monthly payments (the number of years multiplied by 12). If $r = 0$ (no interest), then $c = P/N$.

Write a program (in the `ss.week1` package) that asks the user for the amount borrowed, the yearly interest rate as a percentage, and the number of years, and then prints the monthly payment, rounded to the nearest integer. To keep things simple, you may assume that the user never inputs a yearly interest rate of 0.

Example run:

```
What is the amount borrowed? 290000
What is the yearly interest rate (in %)? 2,18
What is the number of years? 20
The monthly payment is 1492
```

Control structures

In the following exercises, you will practice different *control structures*: **if**, **else**, **while**, **for**, **switch**, **continue**, **break**. Make sure you have studied Chapter 3 (except 3.9) of ECK, as well as the related topics on Canvas.

- 🔊 **P-1.11** Many countries tax annual income using tax brackets, in which taxation increases as the individual's income grows. For example, the tax brackets in The Netherlands in 2022 are:

- Below 35472 euro: 9.42%
- Between 35472 and 69398 euro: 37.07%
- Above 69398 euro: 49.5%

This translates to the following rules:

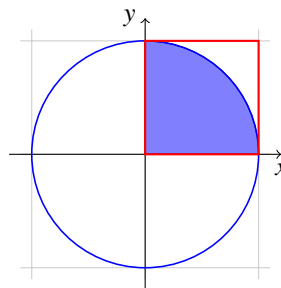
- If your income I is at most 35472, then the tax is $0.0942 \times I$.
- If your income I is more than 35472 and at most 69398, then the tax is $3341 + 0.3707 \times (I - 35472)$.
- If your income I is more than 69398, then the tax is $15917 + 0.4950 \times (I - 69398)$.

Using **if** and **else** control flow statements, write a program in package `ss.week1` that asks the user for their income, and writes their income tax to screen. Example run:

```
What is your income? 30000
Your income tax is 2826.0
```

- P-1.12** Write a program `NumberGuesser` in package `ss.week1` that lets the user guess a random number. Your program should first generate a random number from 1 to 100 (Hint: use `Math.random()`). Then the program should ask the user to guess the number. As long as the user gives the wrong number, your program should tell the user that their guess is too high or too low, and ask again. When the user gives the right number, the program should congratulate the user and stop.

- P-1.13** One method to estimate the value of π uses *random sampling*.



The above picture is a circle of radius 1. The area of such a circle is given by πr^2 and given that $r = 1$, we know that the area of this circle is π . The blue filled quarter has area $\frac{\pi}{4}$, while the area of the red rectangle is 1. Thus, the chance that a random point inside the rectangle is in fact inside the blue area equals $\frac{\pi}{4}$. That is, if the fraction f of random samples lies inside the blue area, we can estimate π as being $4f$. Furthermore, we know using the Pythagorean theorem that a point (x, y) lies inside the circle of radius 1, if and only if $x^2 + y^2 \leq 1$.

Imagine we are using N samples, for example, $N = 10000$. That means that N times, we compute random numbers x and y between 0 and 1, and if $x^2 + y^2 \leq 1$, then we increase a number n . Afterwards, we estimate $\pi = \frac{4n}{N}$ and give that as the answer.


Write a program that asks the user for a number of iterations N , estimates π using this random sampling technique, and writes the result to the screen.

Example run:

```
Please enter the number of iterations: 10000000
Estimation of pi: 3.142126
```

Procedures

In the following exercises, you will be asked to create methods. Make sure you have studied Chapter 4 of ECK, as well as the related topics on Canvas, including Documentation. Since we are not dealing with classes yet, you can create **static** methods. Keep in mind that **static** methods are the exception, as methods are most of the time related to objects, and not isolated subroutines.

 **P-1.14** A *prime number* is a number greater than 1 that is only divisible by itself and 1.

Write a program that asks the user for a number N of primes. Then output the first N primes to the screen.

Example run:

```
How many primes? 20
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
```

To simplify things, create a method **public static boolean isPrime(int number)** that returns **true** if and only if the given number is a prime number. This simplifies the program. You can then use this method in a loop to obtain the first N primes.

To do this properly, use the following procedure:

1. Create the `isPrime` method.
2. Add a description in Javadoc above the method describing what the method does. See also the topic on Documentation.
3. Write comments inside the method, describing the algorithm in *pseudocode*.
4. Finally, write code below each comment.

Importing lab files from Canvas

Lab files, such as auxiliary classes and tests, are available every week on Canvas. The instructions below can be used to import these files into INTELLIJ:

1. Download the ZIP file from Canvas.
2. Extract the contents of the ZIP file.
3. Navigate with your file explorer to the location where the `ss` folder is located that you extracted.
4. Drag and drop the `ss` folder from your file explorer to your `src` folder in IntelliJ.

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 `FibonacciTest` file in `ss/week1/test`. This file is needed in the following exercise.

Side note: If you right click a directory in the Project view of INTELLIJ, a context menu opens and you can find the submenu “Mark directory as”. If there is a package that you don’t want to compile, you can *exclude* the package. Typically, there is a directory marked as “sources root”, this is the `src` directory. In many projects, test sources go into a parallel directory structure called the “test sources root”.

Unit tests

In the next exercise you will use the class `FibonacciTest`. This is a so-called JUNIT *test class*: all methods preceded with the tag `@Test` are so-called *test methods*. In INTELLIJ, you can run this in a similar way as an application class. However, the result will be different: instead of output in the console view, you will get a new JUNIT view listing all tests and their results. If there are errors, you can find out where in the code these errors occurred by selecting them.


P-1.15 The Fibonacci numbers form a famous sequence of numbers, where each number is the sum of the previous two numbers. Typically the numbers are defined as follows:

- $fib(0) = 0$
- $fib(1) = 1$
- $fib(n) = fib(n-1) + fib(n-2)$

Write a program `ss.week1.Fibonacci` that asks the user to input an `int` value, which represents the index of the number, and prints the Fibonacci number. The program has to contain two methods:

1. A recursive method called `fibonacci` which takes as a parameter an `int` value and returns a `long` value, the Fibonacci number.
2. A main method, which asks the user for an integer n and prints the result of `fib(n)`.

After the implementations are finished, run the test in `ss.week1.FibonacciTest`. When signing off, show the TA that the test runs successfully.

 **P-1.16** Solving bugs is a common task of a programmer. In this exercise, you will fix the bugs in the `BrokenFibonacci` class using the debugger. This is an alternative implementation of Fibonacci using an array instead of a recursive function.

First you need to remove the `@Disabled` annotation above the `fibonacciTest` test method in the class `ss.week1.BrokenFibonacciTest` and run the test to see that it goes wrong.

Now run the test in the debugger, setting a breakpoint at the first line that reports an error and using *Step Into* (hotkey F7) to step into the `fibonacci` method. Now step through the program and fix any errors you find. Keep fixing problems until the program is healed.

Documentation of the String API

Using a web browser, access the Java API documentation for the `String` class at <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html>. See also Section 2.2.4 and Section 2.3.3 of ECK. Pay attention to the `String.format` method and the explanation in Section 2.4.1 of ECK.

P-1.17 A very long time ago, humans used a mapping between the alphabet and digits for phone numbers, also called phonewords. See https://en.wikipedia.org/wiki/Telephone_keypad and <https://en.wikipedia.org/wiki/E.161>.



Write a program that asks the user for a word, then translates the word to digits using this mapping. Your program must translate both lowercase and uppercase letters, and ignore nonalphabetical input.

Hint: use something like `String.charAt` or `String.toCharArray` and a **switch**-statement for your implementation.

Example run:

```
Please enter a word: Software
76389273
```

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.lang.System#arraycopy(src, srcPos, dest, destPos, length)`.

Documentation of the Arrays API


Using a web browser, access the Java API documentation for the *Arrays* class at <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html>. See also Section 2.2.4 and Section 2.3.3 of ECK. Pay special attention to `Arrays.sort`.

P-1.18 Write a program `SplitNumbers` that reads a line of numbers separated by whitespace, then print the numbers sorted from low to high.

Hint: To go from a `String` of numbers separated by spaces to an array of `ints`, first “split” the `String` into fragments, then convert each fragment into `ints` (use `Integer.parseInt` for that). Typically, one uses `String[] split = input.split("\\s+");` to split a `String` on one or more *whitespace characters*.

Example run:

```
Please enter some numbers: 5 0 -6 9 3
-6 0 3 5 9
```

 **P-1.19** An *emirp* is a prime number that results in a different prime when its decimal digits are reversed. See also <https://en.wikipedia.org/wiki/Emirp>.

Write a program that asks the user for a number *N* of emirps. Then output the first *N* emirps to the screen.

Example run:

```
How many emirps? 20
13 17 31 37 71 73 79 97 107 113 149 157 163 167 169 179 199 311 337 347
```

Now this exercise is a bit more complicated than previous exercises. Therefore, use procedures to simplify the program design.

- A method **public static boolean** `isPrime(int number)` that returns **true** if and only if the given number is a prime number.
- A method **public static int** `reverse(int number)` that reverses the digits of a number and returns the result.
- A method **public static boolean** `isEmirp(int number)` that returns **true** if and only if the given number is an emirp, using the methods `isPrime` and `reverse`.

To do this right, use the following procedure:

1. Create a method.
2. Add Javadoc above the method describing what the method does.
3. Write comments inside the method, containing *pseudocode* of the algorithm.
4. Finally, write code below each comment.

When signing off, the TA expects to see Javadoc and comments, so don't delete them!

Hint: to reverse a number, use `Integer.toString` and `String.toCharArray`, then create a `String` with the same characters reversed, and convert that back to `int`.

1.3.2 Recommended exercises

P-1.20 Make the following exercises from ECK:

- Exercises 2.4, 2.6 and 2.7.
- Exercises 3.2, 3.4 and 3.7.

Week 2

2.1 Overview

2.1.1 Contents of This Week

Design The design activities in this week cover the following topics

- Level 3 Topics: Q&A session on [L3Tx].
- Level 4 Topics: Q&A session on [L4Tx].
- Project: interview, showcasing, activity diagrams, sequence diagrams, state machine diagrams, see Section 2.2.1.
- Lab: class diagrams, see Section 2.2.2.
- Lab: sequence diagrams, see Section 2.2.3.

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 Expected Self-Study and Project Work

In addition to the scheduled activities (which include watching topic videos), we estimate that you need approximately:

- 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.

2.1.3 Materials for this Week

Design

Watch videos [L3T1], [L3T2].

Watch videos [L4T1], [L4T2], [L4T3], [L4T4].

Programming

Materials ECK, Chapter 4–5.4, the week 2 topic videos

Laboratory The following predefined files are provided on Canvas:

- `design/week2/Dlab-2b-lendBooks.vpp`

- docs/src/ss/hotel/Guest.html
- docs/src/stylesheet.css
- src/ss/hotel/Room.java
- src/ss/week2/DollarsAndCentsCounter.java
- test/ss/hotel/GuestTest.java
- test/ss/hotel/HotelTest.java
- test/ss/hotel/RoomTest.java
- test/ss/week2/DollarsAndCentsCounterTest.java


2.2 Design

2.2.1 Project

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

- Conduct an interview [**L2T1**] with a relevant stakeholder of the ... system, in order to elicit requirements for the management functions of the system. On Canvas, you can find a list of contact persons according to your group. The interview can be conducted some time this week, so make your appointment as soon as possible, if you have not made it already in Week 1.
- Think and discuss how the input of the interview (the elicited requirements) influenced your understanding of the system. Adjust the list of requirements that you have prepared last week, accordingly—perform requirements validation [**L2T1**].
- Discuss in your team what (possibly partial) structural models can be useful in your project. Start by drawing some object diagrams [**L3T1**] to showcase important functions of the system that you identified last week. Proceed to generalise them to class diagrams [**L3T2**].
- Discuss in your team what behavioural models will be useful in your project. Which processes are better suitable to be modelled by activity diagrams [**L4T1**], which by sequence diagrams [**L4T2**], which by state machine diagrams [**L4T3**]? Start drawing the diagrams that you agree to be useful and appropriate.

2.2.2 Laboratory exercises on class diagrams

 **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 received more and more participants, and the program is now expanding. This calls for a more professional organization. One of the things OHT needs is a proper booking system. You are asked to help make 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 tours 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 in which the difficulty of a tour is indicated by the 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); 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 hires local 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 Aquitaine Tour includes canoeing on the rivers Lot and Dordogne; the canoes are rented locally with different companies. 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, their name, address, gender and SSN (social security number) are known. 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 tour, 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).

2.2.3 Laboratory exercises on sequence diagrams

D-2.3 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 cannot be taken home. An attempt to lend out a book that is not lendable will fail as the system will display that it is not lendable.
- If books are returned too late, a small fee is due. Preferably this is paid immediately when the books are returned. But sometimes customers have no money with them and in this 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-2.4**) 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 medications. In addition, if 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.

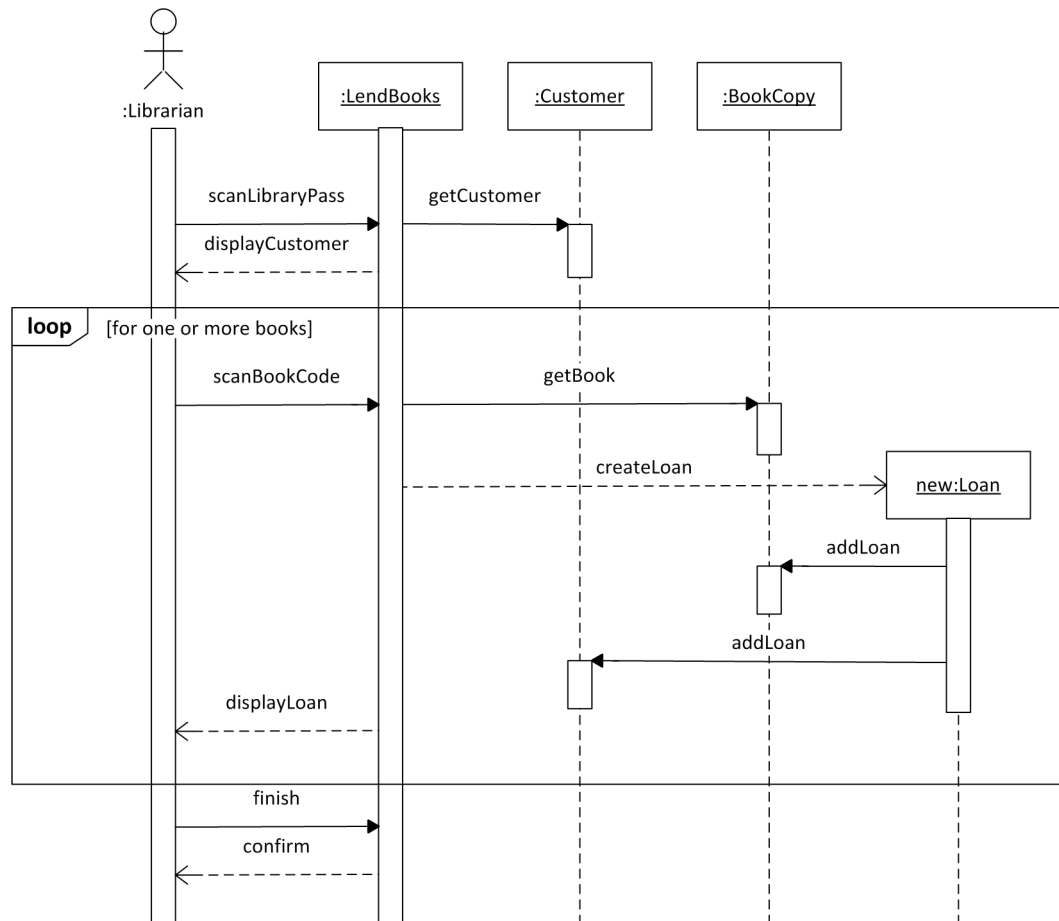


Figure 2.1: Lending library books (Exercise D-2.3)

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 3.2.6 (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.*

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 which 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 when 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;

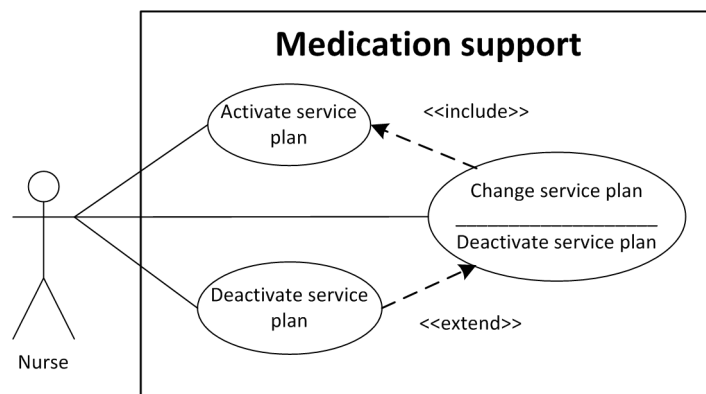


Figure 2.2: Partial Use Case Diagram for medication support (Exercise D-2.4)

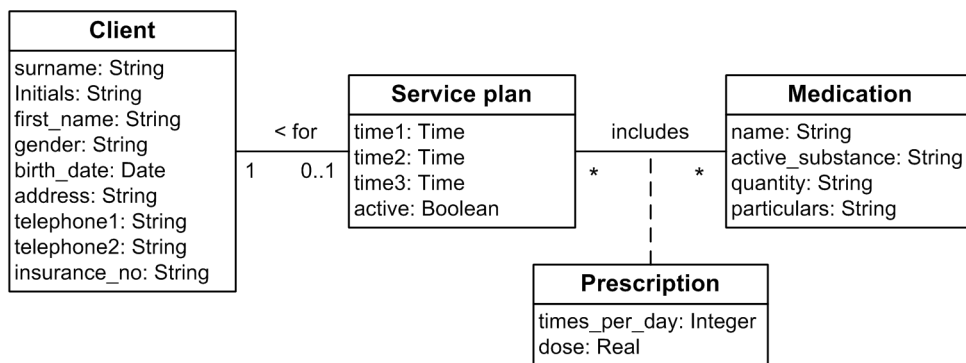


Figure 2.3: Partial Class Diagram for medication support (Exercise D-2.4)

- 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.

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*.

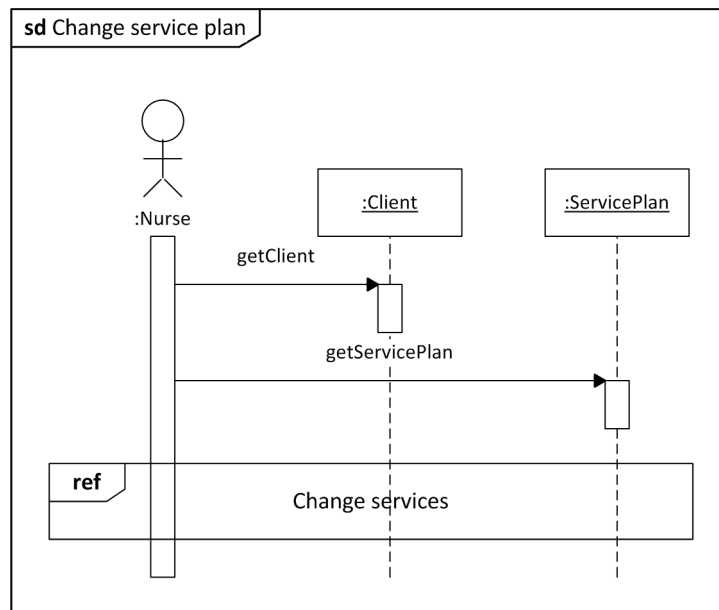



Figure 2.4: Sequence Diagram for *Change serviceplan* (without control object) (Exercise D-2.4)

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

2.2.4 Recommended exercises on class diagrams

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

D-2.5 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 1980-s 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 a 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 the 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-2.6 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 owned by Ship Rentals itself, others are chartered by Ship Rentals from their respective 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 an 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 license number). The contract also states the date that the contract was 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. If 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 paid 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 be stored. In most cases, this is the end date of the contract, but due to bad weather or other reasons, it could happen that the ship is returned later. 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.

2.2.5 Recommended exercises on sequence diagrams

D-2.7 Make a sequence diagram for booking a tour with Outdoor Holiday Tours (see Section 2.2.2 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 description.

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 applicant fulfills the prerequisites, based on the evidence that the applicants 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 makes 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 both cases, starting/opening the application will need some interaction with the web server to authenticate the user, which we disregard for convenience*).

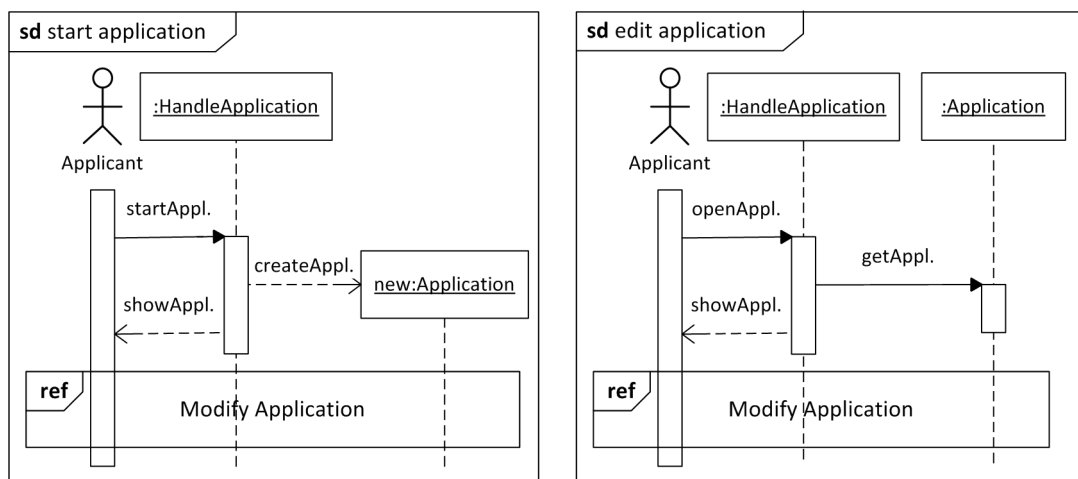


Figure 2.5: Creating/editing an application for university admission (Exercise D-2.8)

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.

2.3 Programming

2.3.1 Laboratory exercises

Import the contents of this week's lab files from Canvas to your INTELLIJ project by following the instructions mentioned on page 34.

Important: INTELLIJ will show compilation errors in the classes that you have just imported. This is because some imported classes make references to classes that you will develop during the lab sessions this week. After completing the assignment for class `DollarsAndCentsCounter` in Exercise P-2.1, for instance, the compilation errors in `DollarsAndCentsCounterTest` should be gone.

Objects and Classes

Make sure you have studied ECK Chapter 5 and the related topics on Canvas.

Javadoc

This week, you will be required to *document* your classes with Javadoc. Javadoc for a class is placed immediately before the **class** declaration, similar to how Javadoc for a method is placed immediately before the method. Typical conventions for Javadoc in Java projects include rules like:


- All classes and public constants and methods must have Javadoc
- Any side-effects (changes to the object) should be clear from the Javadoc
- Use `@param` to describe every parameter, even if trivial
- Use `@returns` to describe the return value, even if trivial
- Do not use `@author` and `@version`
- Consider whether parameters can be **null**

See also <https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html> and <https://www.oracle.com/java/technologies/javase/api-specifications.html>.

Preconditions and postconditions

This week, you will be required to *specify* methods of classes, that is, to write preconditions and postconditions. See Appendix B (page 85) and the related topics on Canvas. Preconditions and postconditions are placed between the Javadoc and the method declaration. Invariants are typically placed after the fields and before the methods.

Whenever we ask you to *specify* in lab exercises, this means adding specifications in the form of invariants, preconditions and postconditions in JML.

 **P-2.1** In this exercise, you implement and test a class that counts an amount of money in dollars and cents. The counter should provide the following functionality:

- The method **public int** `getDollars()` returns the amount of dollars.
- The method **public int** `getCents()` returns the amount of cents.
- The method **public void** `add(int dollars, int cents)` adds the specified amount of dollars and cents to the counter.
- The method **public void** `reset()` sets the amount of dollars and cents to 0.

Pay special attention to the Javadoc and JML specification of the `ss.week2.DollarsAndCentsCounter` class. In the imported lab files you will find the test class `ss.week2.DollarsAndCentsCounterTest`. Improve the implementation until it passes all test cases.

Hints:

- Consider the internal representation: what field(s) do you need, what type of field(s)?
- Do you need a constructor? If so, add an appropriate one.
- Make sure your class fulfills the postconditions (the `ensures` expressions).

Three-Way Lamp

In the following exercises, you will implement a three-way lamp. This is a light switch with four different settings: *off*, *low*, *medium*, *high*. The ideal way to represent these settings is with an *enumerated type*. In case you have forgotten about **enum**, consult Section 2.3.5 of ECK. The lamp will be accompanied by a *textual user interface*, which should give the user several options:

- Change the current setting to a specified setting
- Print the current setting of the lamp to the screen
- Switch to the next setting, observing the order *off* → *low* → *medium* → *high* → *off*

All in all, you will design and implement:

- The class `ThreeWayLamp`
- The enumerated type `ThreeWayLamp.LampSetting`
- The class `ThreeWayLampTUI`

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

Write Javadoc documentation before the class and before every method.

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 or an enumerated type, 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.

P-2.4 Complete the specification of the `ThreeWayLamp` class by JML for it. This means defining invariants at the class level, and preconditions and postconditions for all methods. See also Appendix B.

In this specification you have to explicitly define that after OFF the lamp goes to LOW, after LOW it goes to MEDIUM, etc. Every method with a return value and/or side-effects should have an `ensures` post-condition specifying the return value and/or the side-effect. Also consider statements that are always true of the fields, which you can specify as invariants.

Hint: make use of `\old` to reason how values changed after method calls.

P-2.5 Write unit tests for the `ThreeWayLamp` class based on the conditions you defined in Exercise P-2.4. Create a class `ThreeWayLampTest` in package `ss.week2` that implements your tests. See also Appendix A. The test 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.
- If setting the lamp to a specific setting works properly.

The `ThreeWayLampTest` class should have the following elements:

- A (private) field of type `ThreeWayLamp` to hold the object to be tested.
- A `setUp` method that creates the `ThreeWayLamp` object to be tested. This method should get the `@BeforeEach` annotation, so that every unit test starts with a fresh object.
- A method for each of the test cases above; these methods should get the `@Test` annotation.

Use assertions such as `assertEquals` to test your methods. After creating the test, run it to see that all tests fail.

 **P-2.6** Implement the methods that you specified in Exercise P-2.4 by replacing the method body of 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.

Now that you have a correct implementation of a three-way lamp, it is time to make a textual user interface (TUI) for the lamp. The idea of a TUI is to repeatedly ask the user for input, and then take the appropriate action. This input takes the form of commands. The TUI can also ask the user for additional input.

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

- **OFF**: Set the lamp to OFF
- **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

In the previous exercises, the `main` method implemented dealing with user input. In the object-oriented paradigm, the `main` method should be minimal. It only creates the objects necessary to start the program, and then calls methods of these objects that implement the desired functionality. The code that runs the TUI should be in a separate method that will be invoked by `main` after constructing the `ThreeWayLampTUI` object. This method could be named `run`.

Your `ThreeWayLampTUI` also needs a `ThreeWayLamp` to be a user interface for. You should add this `ThreeWayLamp` as a field of `ThreeWayLampTUI`.

P-2.7 Create a new class `ss.week2.ThreeWayLampTUI` with a `main` function. *Tip*: if you type `main` in IntelliJ, it suggests to automatically create a `main` prototype. Create a field for the `ThreeWayLamp` object, a `run()` command, and a constructor.

The field needs to be initialized with an actual `ThreeWayLamp` object in the constructor. There are two ways to do this:

- Let the constructor of `ThreeWayLampTUI` create a `ThreeWayLamp` object.
- Create the `ThreeWayLamp` object first, then give it as a parameter to the constructor of the `ThreeWayLampTUI`.

This is a *program design decision*. Make a decision and explain why. Then implement the constructor and `main` accordingly, while keeping `run` a stub.

 **P-2.8** The expected structure of the `run()` method body is represented by the following pseudo code:

```

1: Print the help menu;
2: exit ← false;
3: while ¬exit do
4:   input ← input string from stdin
5:   if input = OFF then
6:     Set light to off;
```

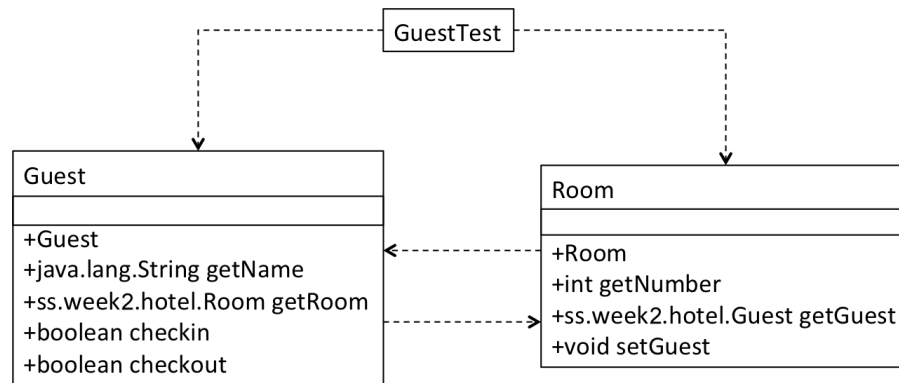


Figure 2.6: Class Diagram for Hotel Application

```

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:     Switch light to next value;
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
  
```

Instead of nested **ifs** you are asked to use the **switch** statement to check the input values and act accordingly. See also ECK Section 3.6.2 for examples of input menus and the **switch** to handle options.

Hint: Create a separate method that prints the help menu. What is the advantage of this?

The Hotel


In the following exercises, you will develop a hotel management program. Because this program will be extended in week 3, your program will live in the `ss.hotel` package.

P-2.9 Create a class `Guest` in package `ss.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-2.10 In INTELLIJ, run the `ss.hotel.GuestTest` JUNIT tests that you downloaded from Canvas in order to test your implementation. Explain what happens.

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

P-2.11 Add Javadoc and JML 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.

-  **P-2.12** Generate Javadoc documentation for your project. How you can generate Javadoc for your code is explained on the “Tools installation session” page on Canvas. Once the documentation is generated, you can open the file `javadoc/index.html` to start browsing the documentation. Which information is included in the Javadoc documentation?

Explain when Javadoc can be useful.

- P-2.13** Implement your `Guest` class 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 `GuestTest` test from Canvas.


Before your program passed all tests, it probably 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-2.14** 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.hotel.Room@5cdd8682> but was:<null>
```

The `ss.hotel.Room@5cdd8682` part is automatically generated and represents a unique “reference” to the object. This is the default behavior for classes in Java; the reason why will come when inheritance from `Object` is discussed. JAVA automatically obtains `String` representations of an object by looking for a method `public String toString()`. See also ECK Section 5.3.2.

The current representation is not very informative. 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 `Guest` or `Room` objects.

-  **P-2.15** 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-2.16** 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.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-2.17** 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.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-2.18 Specify (using JML) a class `ss.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.

Additionally, the hotel should have an instance variable `name` with an appropriate query. This instance variable is set when the object is initialised.

Complete your specification by adding class invariants, preconditions and postconditions i.e. write JML. Extend the class diagram of Exercise **P-2.9** with the new classes; the class diagram should show all classes in the package `ss.hotel`.

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

-  **P-2.19** Implement the class `Hotel` as you specified above. Improve your implementation until it passes tests of `ss.hotel.HotelTest` without errors. While implementing `Hotel`, use the debugger to catch and fix any mistakes. You can use the debugger by running `HotelTest` in debug mode. The debugger gives you insight by going over your code line-by-line.

When signing off, show the TA the following actions:

- Open `HotelTest`.
- Set a breakpoint on the first line of the method called `testCheckoutOccupiedRoom()`. The important lines here are the method calls `hotel.checkIn(GUEST_NAME_1)` and `hotel.checkOut(GUEST_NAME_1)`. The other lines can be ignored.
- Run the test using the debugger.
- Step into `hotel.checkIn(GUEST_NAME_1)`.
- Step over the lines in the method and show/explain the state of the hotel, its rooms and guests.
- After `hotel.checkIn(GUEST_NAME_1)` is done executing, step over to the line `hotel.checkOut(GUEST_NAME_1)`.
- Step into `hotel.checkOut(GUEST_NAME_1)`.
- Step over the lines in the method and show/explain the state of the hotel, its rooms and guests.
- After `hotel.checkIn(GUEST_NAME_1)` is done executing, let the program continue.

Hotel TUI Integration

Similar to the textual user interface for the Three Way Lamp, one can define a textual user interface for the hotel.

- 🔗 **P-2.20** Develop a class `ss.hotel.HotelTUI` implementing a textual user interface for the hotel. The TUI must support checking guests in and out, requesting the current room of a guest, and activating the safe.

An example execution would be the following.

```
Welcome to the Hotel management system of the "Hotel Twente"
Commands:
in name ..... check in guest with name
out name ..... check out guest with name
room name ..... request room of guest with name
activate name ..... activate safe of guest with name
help ..... help (this menu)
print ..... print state
exit ..... exit

Command: in Richard
Guest Richard gets room 101
Command: room Richard
Guest Richard has room 101
Command: activate Richard
Safe of guest Richard is activated
Command: print
Hotel Hotel Twente:
  Room 101:
    rented by: Guest Richard
    safe active: true
  Room 102:
    rented by: null
    safe active: false
Command: out Richard
Command: room Richard
Guest Richard doesn't have a room
Command: exit
```

The expected structure of the main TUI loop is represented by the following pseudo code:

```
1: Print menu;
2: exit ← false
3: while ¬exit do
4:   line ← line from stdin;
5:   split[] ← line split into words;
6:   command ← split[0];
7:   param ← null
8:   if split.length > 1 then
9:     param ← split[1];
10:  end if
11:  if command = IN then
12:    if param = null then
13:      Print error message;
14:    else
15:      Try to check in the guest;
16:      if no available room then
17:        Print error message;
18:      else
19:        Print success message;
20:      end if
21:    end if
22:  else if command = OUT then
```



```

23:         if param = null then
24:             Print error message;
25:         else if guest was not checked in then
26:             Print error message;
27:         else
28:             Check out the guest;
29:             Print success message;
30:         end if
31:     else if command = ROOM then
32:         if param = null then
33:             Print error message;
34:         else
35:             Get room of guest;
36:             if Room of guest is null then
37:                 Print error message;
38:             else
39:                 Print room information;
40:             end if
41:         end if
42:     else if command = ACTIVATE then
43:         if param = null then
44:             Print error message;
45:         else
46:             Get room of guest;
47:             if Room of guest is null then
48:                 Print error message;
49:             else
50:                 Activate the safe;
51:                 Print success message;
52:             end if
53:         end if
54:     else if command = PRINT then
55:         Print hotel information;
56:     else if command = HELP then
57:         Print menu;
58:     else if command = EXIT then
59:         exit ← true
60:     else
61:         Print error message;
62:         Print menu;
63:     end if
64: end while

```

Instead of nested **ifs** 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. Typically, one uses `String[] split = input.split("\\s+");` to split the input line on one or more *whitespace characters*.

Also take the following guidelines into account to implement this program:

- Place the main TUI loop in a method **void** `run()`.
- Your `HotelTUI` needs a **main** method. This method should be the only **static** method in your entire program. The **main** method should create a new `Hotel` and then create a `HotelTUI` for this `Hotel` and run the TUI using the `run` method of the created `HotelTUI` object.
- Use constants to define the command strings (`in`, `out`, `room`, etc.) of your program. Constants can be defined by declaring them with **static final**. For example, a constant to represent the *checkin* command with value `"in"` would be defined as **static final** `String IN = "in";`. By convention, constants are defined in uppercase. See also ECK Section 4.8.3 about named constants.
- Apart from the constants, your `HotelTUI` class only needs a single field, for the `Hotel` object.

- Create helper methods such as **void** `doIn(String name)` that you can call from `run()` to implement the commands, instead of putting the implementations directly inside the **switch**.

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.

2.3.2 Recommended exercises

P-2.21 Make the following exercises from ECK:

- Exercises 5.1, 5.2 and 5.3.

Week 3

3.1 Overview

3.1.1 Contents of This Week

Design The activities in this week cover the following topics:

- Level 5 Topics: Q&A session on [**L5Tx**].
- Level 6 Topics: Q&A session on [**L6Tx**].
- Project: use case diagrams, metrics, conventions, consistency, see Section 3.2.1.
- Lab: activity diagrams, see Section 3.2.3.
- Lab: use case diagrams, see Section 3.2.4.

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.

- Design diagnostic test

3.1.3 Expected Self-Study and Project Work

In addition to the scheduled activities (which include watching topic videos), we estimate that you need approximately:

- 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

Design

Watch videos [L5T1], [L5T2], [L5T3].

Watch videos [L6T1], [L6T2], [L6T3], [L6T4], [L6T5].

Programming

Lectures ECK, Sections 5.5–5.8, the week 3 topic videos

Laboratory The following predefined files are provided on Canvas:

- design/week3/Dlab-1a-intro.vpp
- design/week3/Dlab-1b-incident-reporting.vpp
- design/week3/Dlab-1b-TheatreTickets/ActorsList.rtf
- design/week3/Dlab-1b-TheatreTickets/Glossary.rtf
- design/week3/Dlab-1b-TheatreTickets/RequirementsAndUseCases.rtf
- design/week3/Dlab-1b-TheatreTickets/UseCaseDescriptions-brief.rtf
- design/week3/Dlab-1b-TheatreTickets/UseCaseDescriptions-extended.rtf
- docs/src/ss/hotel/bill/Bill.html
- docs/src/ss/hotel/bill/Bill.Item.html
- docs/src/ss/hotel/password/BasicPassword.html
- docs/src/stylesheet.css
- test/ss/hotel/HotelTest.java
- test/ss/hotel/password/BasicPasswordTest.java
- test/ss/hotel/PricedRoomTest.java
- test/ss/hotel/PricedSafeTest.java

3.2 Design

3.2.1 Project

D-P.3 For the project, you should work on the following tasks:

- Re-read the project description and your list of requirements. Draw use case diagrams [L5T1] for core functions of the system. Revise your list of stakeholders if you notice some missing or excessive actors.
- Find connections between the use case diagrams, activity diagrams [L4T1] and sequence diagrams [L4T2] that you have drawn last week. What's missing? Revise and remove/add diagrams as you deem appropriate.

3.2.2 Laboratory exercises on version control

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 will 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 do not 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 INTELLIJ. 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-3.1 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.

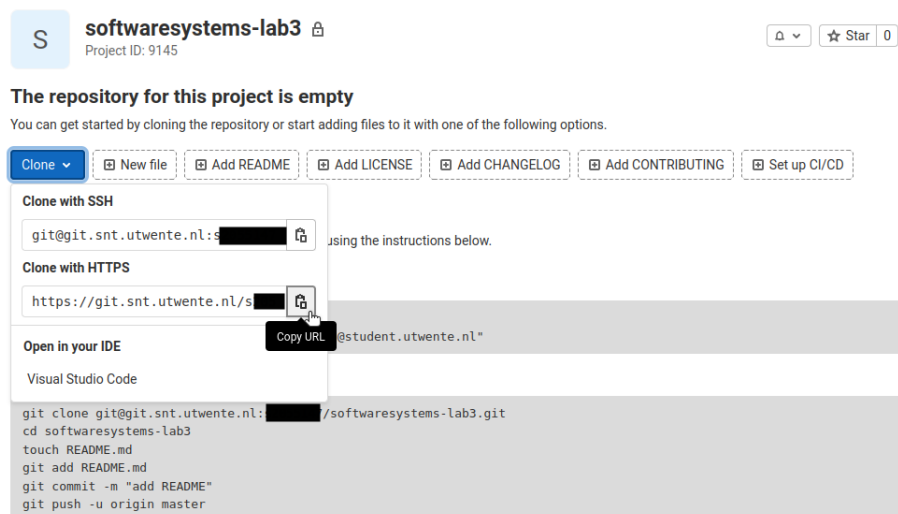
D-3.2 What does GIT stand for?

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-3.3 (Your first repository)

- For this module, and in module 4, the GitLab server of the UT is used. All students already have access to it with their student account. So, go to <https://gitlab.utwente.nl/> and login via Microsoft Single Sign On (UT) with your student email address and password.
- Create a new (bare, initially empty) remote repository on that account.
 - Click NEW PROJECT or go to the new repository page directly: <https://gitlab.utwente.nl/projects/new>.
 - Select CREATE BLANK PROJECT
 - Give the repository a fitting name in the PROJECT NAME field, like `softwaresystems-lab3`.
 - Ensure that the *Visibility Level* is set to PRIVATE. This is to ensure that your project is not visible to all other users on the UT.
 - Click CREATE PROJECT.
- Create a new INTELLIJ project, create a local repository and link the local repository to the remote repository.
 - In INTELLIJ, create a new empty Java project and add a trivial, single class `Hello` in it.
 - Select VCS → Enable Version Control Integration...
 - Keep GIT selected and click on OK (In the toolbar VCS should now be replaced with Git)
 - Select Git → Manage Remotes...
 - Click on the +-symbol
 - Use `origin`¹ as the remote name. For the URL, use the link to your remote repository. You can find this link on the web page of your repository. Use the HTTPS link for now and not the SSH one.² It should look something like `https://gitlab.utwente.nl/<username>/<repositoryname>.git`.³



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

²Setting 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.

³This is analogous to the `git remote add origin <url>` command.

- (g) Log in with the same credentials you used to log into the GitLab
- 4. Now we will actually use the repository. We will first have to put our `Hello` file under version control, before GIT actually cares about changes made in that file
 - (a) In the toolbar select GIT → COMMIT.... This will open the GIT panel
 - (b) Our `Hello` file is hiding under UNVERSIONED FILES. Fold this list open and check the box next to the `Hello` file.
 - (c) Always write an appropriate message to go along with changes you make. This ensures that your teammates and your future self can understand the intentions behind these changes.
 - (d) Click COMMIT to commit the file to the local repository
- 5. Now 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.
 - (b) Open the GIT panel again.
 - (c) Now we have to tell GIT that we want to make our changes permanent. This is done by *staging*, and then *committing* the file. We can select files whose changes we want to add to the commit. By default all files that have been changed will be *staged*⁴
 - (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.⁵
 - (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 synchronizes all of your local changes with the remote repository. Push your local repository to the origin now. In the toolbar select GIT → PUSH.... Double check if all commits you expect are present and click on PUSH
- 6. 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. Notes: When making a GitLab repository, the branch is named "main" by default by GitLab. However, when we make a local repository via IntelliJ, the repository is named "master" by default. This has caused GitLab to have two branches by default. This does not cause any conflict for now, but it is good to know that there are two branches by default when doing the exercise. In order to fix this, the two branches could be easily merged together as taught later in Exercise D-3.6 about branching and merging.

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-3.4 (Rolling back)

1. Add a method to your `Hello` class (from Exercise D-3.3) 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. Do not change anything in the `Hello` class. This change is clearly independent from the previous one.
3. At the bottom of INTELLIJ select the Git history panel (ALT+9 ALSO DOES THE TRICK). Here all your recent commits are displayed with the newest at the top. Your commit messages form a guide as to what has happened (Just the more reason to use meaningful commit-messages!)
4. Select the one but last commit in the history view, and select REVERT COMMIT from the context menu. This opens up a window showcasing what changes the revert will cause. Press COMMIT. 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.
6. Finish up by pushing all local commits to the remote repository.

⁴Checking the boxes next to files is analogous to running the `git add <filename>` command.

⁵Adding a message and clicking “Commit” is analogous to running the `git commit -m "<message>"` command.

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-3.5 (Pushing and pulling)

- Now it’s time to see how Git handles multiple users, and how changes by multiple users are merged together.
 - On the web page of your repository on GitLab, add your partner as a collaborator to the project. Go to the PROJECT INFORMATION tab in the left bar and select MEMBERS, search for your partner’s student number, name or email and add them to the project with the MAINTAINER role.
 - On your partner’s laptop, clone the central repository: In the toolbar, select GIT → CLONE... or VSC → GET FROM VERSION CONTROL...
 - Paste the URL to the repository in the URL field. This is the same URL that you used in Exercise D-3.3 when you added the remote repository to your local repository.
 - Pick a directory to clone the repository to. This can be anywhere you want, but the default location IntelliJ chooses for you should be fine.
 - Click on CLONE
- Make some edits to one of the two local repositories you have now, commit them, and push them to the central repository.
- In the *other* local repository, execute a *pull* by clicking on GIT → UPDATE PROJECT... in the toolbar. This should result in an update such that the states of these two projects are now the same.
- 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?
- If the push failed in the last step, try to pull the repository first, and then push again. What happens?
- Make sure both repositories are up-to-date with the upstream by pulling both of them.
- 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.
- Look at the repository history again. 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-3.6 (Branching and merging)

- In the toolbar select GIT → NEW BRANCH... Call the new branch `try` or some other clever name. Note that you can see which branch your project is on in the bottom right.
- Create a new class in your project, and add a method to another class. Commit and push. These edits are now part of the new branch, but are *not* visible in the master branch.

3. Switch to the `master` branch (Toolbar: GIT → BRANCHES... → MASTER (UNDER LOCAL BRANCHES) → CHECKOUT). 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 GIT → 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 indispensable 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.

3.2.3 Laboratory exercises on activity diagrams

D-3.7 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 an incomplete version of the activity diagram shown in Figure 3.1. 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 such that your diagram is equivalent to Figure 3.1.

Please note: VP offers the option to add a condition to the decision node (so that you could annotate the outgoing control flows with “yes” and “no”). We use the standard UML notation where the conditions are given as annotations to outgoing arrows.

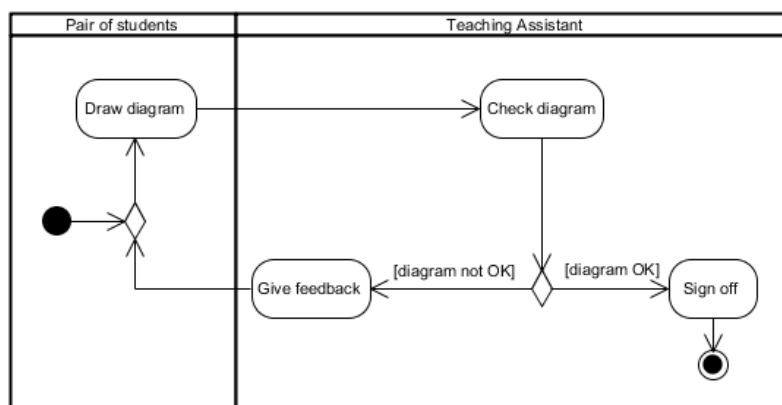


Figure 3.1: Activity Diagram for introductory exercise

In the following series of exercises we will make an activity diagram of the information gathering process of the police, described below⁶.

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

⁶*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.

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 makes 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; offenses 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-3.8 Create an (initial) activity diagram for 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 consists 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-3.9** Extend the activity diagram by incorporating the following:

- Sometimes a police officer on duty observes an offense (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-3.10** 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-3.11 (Optional) If you have finished Exercise **D-3.10** 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-3.10** it is assumed that police officers always comply when they are asked to improve a mutation, but some police officers are stubborn. The improvements are sometimes okay, sometimes still poor quality, and sometimes not done at all. Therefore, if a police officer has been asked to improve a mutation, data managers will carry out another inspection two weeks later. However, if it is still not up to standard, Data Management does not have the authority to sanction the police officer. What can be done though, is notify the manager of the police officer.

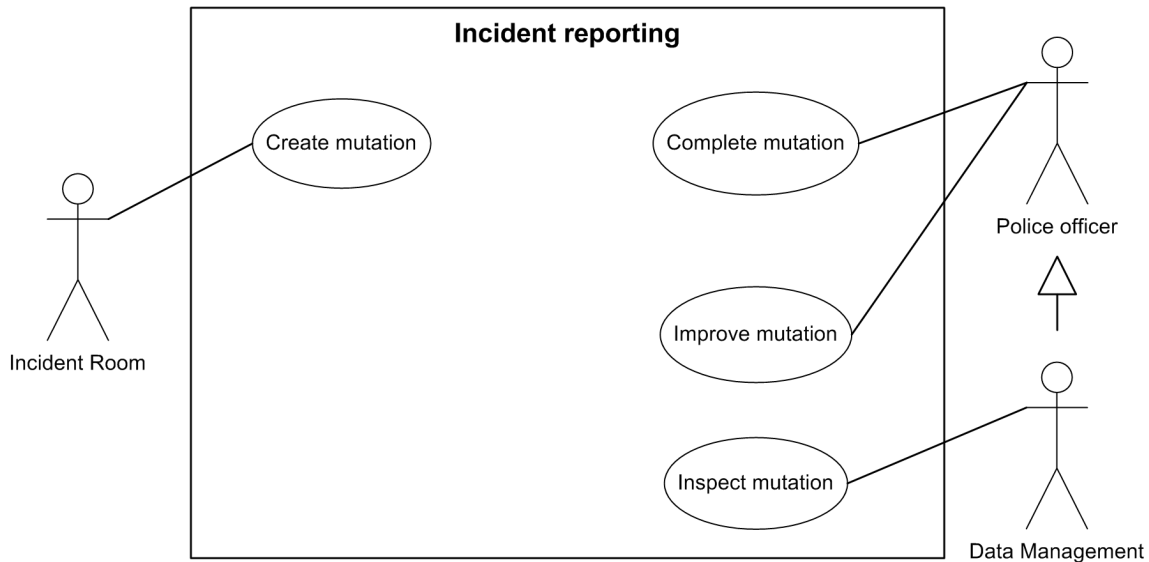



Figure 3.2: Use Case Diagram for incident reporting (Exercise D-3.12)

3.2.4 Laboratory exercises on use case diagrams

 **D-3.12** Figure 3.2 gives a very simple use case diagram for a part of the Police case study from Section 3.2.3. 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-3.13 – D-3.19 all relate to a case study about handling theatre tickets.

Theatre tickets: Introduction

State subsidies for cultural activities have repeatedly decreased over the last few years. As a consequence, the cultural sector in the Polderland region 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 is needed.

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 it easier to attract sponsors. They are mentioned as sponsors 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 a 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 while tickets for the following performances have a regular price. The system with different prices for different seating areas was 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. But 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 and start with modelling two ways of buying tickets: by means of a paper order form and through the 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 before the 1st of September, the payment will 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 uses a graphical interface, showing all seats that are still available for a performance, to make the booking.

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

D-3.13 Figure 3.3 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 can find the document in this week’s lab files.

-  **D-3.14** Make a use case diagram that includes all the use cases identified in D-3.13. For your convenience, an actor list has been provided in Figure 3.4

	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 3.3: Requirements for the Polderland Theatre system (so far) (Exercise D-3.13)

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 3.4: Actors for the Polderland Theatre system (so far) (Exercise D-3.14)

For Exercises D-3.15 to D-3.19, 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 people 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 buy 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?

	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 3.5: Template for brief use case descriptions (Exercise D-3.18)

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 as 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 how it works again. 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-3.15 Extend the list of actors in Figure 3.4 so that it covers the whole case description.


You can find the document in this week's lab files.

D-3.16 Extend the list with requirements and use cases (see D-3.13) so that it covers the whole case description.

 **D-3.17** Extend the use case diagram with all use cases identified in D-3.16.

D-3.18 Make a complete listing of brief use case descriptions for the use cases in D-3.16 / D-3.17.

Figure 3.5 gives a template. You can find the document in this week's lab files.

 **D-3.19** Make an extended use case description for *Process order form* by completing the description given in Figure 3.6. You can 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, extended use case descriptions add further design details which are not present in the use case diagram.)*

3.2.5 Recommended exercises on activity diagrams

D-3.20 Make an activity diagram for X-rays in the hospital, according to the case history described below.

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 performance data entered so far identify more than one performance		
...	...		

Figure 3.6: Incomplete extended use case description (Exercise D-3.19)

Radiology administration

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

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

A few introductory remarks:

- Different medical examinations may have different procedures. To keep things concrete and simple, the case description is limited to making X-ray photographs.
- Also, we ignore the fact that different parts of the procedures described below will be embedded in different systems which are already present in the hospital. The purpose here is to clarify the process steps themselves, not the systems in which these steps will be embedded.

Radiographic examination

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

- A medical specialist in the Polderland hospital usually requests a radiographic examination during a consultation with the patient. From within the patient's electronic record, the specialist needs can open a request form with a single mouse click. The patient's essential data will be included automatically. The specialist adds a reason for the request.
- After the request has been filed by the medical specialist, the patient should make an appointment for the examination. A patient can visit or phone the Radiology secretariat for an appointment. A secretary will record an appointment in the system for a time that suits the patient. (*We ignore special cases where the request is urgent and perhaps the patient is incapacitated, then the medical staff will make an immediate appointment on their behalf. You don't have to model that.*)
- General practitioners in the region can also make a request for a radiographic examination of a patient. In this case the request will be made electronically as well, but chances are that the patient data are not as complete as the hospital would like to have them. Consequently, if a patient contacts the Radiology secretariat, the secretary should do two things: make an appointment for the patient and check whether the patient data are complete. If not, ask the patient for the missing data and enter them into the system.

- At the appointed time the patient checks in at the Radiology secretariat. A secretary places a patient on the work list for that day. Usually it takes 10-15 minutes until it is the patient's turn, in exceptional cases it could take a bit longer.
Unfortunately, it happens that patients do not turn up. If a patient hasn't checked in one hour after the appointed time, a letter is generated automatically, asking the patient to make a new appointment. Later that day a secretary prints the letter and sends it by (physical) mail to the patient.
- The X-rays are made by a radiology assistant. The assistant positions the patient so that the right body part will be photographed from the right angle and then hides behind a protective cover for making the X-ray. When all requested 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.

3.2.6 Recommended exercises on use case diagrams

Exercise **D-3.21** is 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 medications. When, in addition, their memory is not 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. Medications are 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 medications from the cupboard, and sees to it that the medications are 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 will 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 the nursing staff makes sure that the pills are actually taken. However, there is a large 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, who will then visit 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 for when the 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 a choice of several *service patterns* with fixed values for m , k , and n . (A service pattern for heart problems is different from a service pattern for rheumatic conditions. For the latter, timely medication is much less important, so there are more repeats with longer intervals.)

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

- When a nurse enters a (new) service plan, the service plan is activated automatically.

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

- Occasionally the medication of a client is changed. If this happens, a nurse adapts the service plan.

When a service plan is changed, the current service is automatically deactivated (if it was active at that moment). When the changed service plan is stored, the (changed) service will be automatically reactivated.

- An obvious condition is that the medication is physically present in the dispenser. From time to time, a nurse visits the client's apartment and refills the dispenser. The dispenser contains a series of packets. Every time the client needs to take medication, the next packet is released. (So the nurses should take proper care filling the dispenser when different medication is taken at different times. The dispenser has no knowledge of the contents of the packets.)

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

An actor list for the central information system has already been compiled, it is shown in Figure 3.7.

D-3.21 Make a use case diagram for the central information system for medication support.

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 medication 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 3.7: Actor list for the medication dispenser service (Exercise D-3.21)

3.3 Programming

3.3.1 Laboratory exercises

Import the contents of this week's lab files from Canvas to your INTELLIJ project by following the instructions mentioned on page 34.

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 lab files from Canvas (in the package `ss.hotel.password`).

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`?
3. A `BasicPassword` also has a field to store the password. Why is this field not visible in the documentation?

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.hotel.password`, respecting its documentation. Make sure the stub implementation compiles. Run the `BasicPasswordTest` JUnit test and confirm that the tests fail.

 **P-3.3** Complete the implementation of `BasicPassword` and make sure that the `BasicPasswordTest` JUnit tests succeed. 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.

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 Section 5.7 about interfaces.

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

- **boolean** `acceptable(String)`, which should return **true** if the parameter is an acceptable `String`.
- `String generatePassword()`, which should return (an example of) an acceptable `String`.

Define preconditions and postconditions, as well as useful documentation for the `Checker` class and its methods.


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

- A class `BasicChecker` 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**;
- 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.

It is fine for both `Checkers` to return a constant `String` in `generatePassword()`. After all, as long as the returned `String` is acceptable, the specification is adhered to.

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 the class `ss.hotel.password.BasicPassword` to `ss.hotel.password.Password`, and then give `Password` an additional instance variable `checker` (of type `Checker`) and an instance variable `initPass` (of type `String`). The purpose of the `initPass` 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. Expose the instance variables via the queries `getInitPass()` and `getChecker()`.

Class `Password` should have two constructors:

- The first constructor receives a `Checker` as parameter and sets `checker` and `initPass` 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.

Bill 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 `BillPrinter` interface, and after that you will implement a `Bill` class. Bills can be printed using implementations of the `BillPrinter` interface. In order to do this, you need to use the static method `format` from class `String`, whose purpose is to create a formatted `String` from a number of input objects.

Tip: check the Javadoc for the `Formatter` online: <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html>

P-3.7 Define an interface `ss.hotel.bill.BillPrinter` 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.println("Text1", 1.0);
p.println("Other_text", -12.1212);
p.println("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.

P-3.8 Program the following Printer-implementations:

1. A class `ss.hotel.bill.SysoutBillPrinter`, whose `printLine` method directly prints to the standard output. Give `SysoutBillPrinter` a main method that calls `printLine` a couple of times with examples to show how this method works.
2. A class `ss.hotel.bill.StringBillPrinter`, whose `printLine` method collects all lines in a single `String` by continuously adding lines to the same variable (initialised as an empty `String`). Nothing should be printed directly to the standard output. `StringBillPrinter` 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 represents a bill. To make this as general as possible, a bill will consist of *items*, where each item has a *description*, and a *price* 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 Section 5.8 about 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 `BillPrinter` object when it is constructed. If we want to print to the standard output (console) we can pass the `SysoutBillPrinter` to this object, otherwise, we can use a `StringBillPrinter`. Using the `StringBillPrinter` we can query the result after adding items.

P-3.9 Implement a class `ss.hotel.bill.Bill` that has a nested interface `Item`. The files imported from Canvas contain the specification of this class and of the interface (`ss/hotel/bill/Bill.html` and `ss/hotel/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.hotel.bill.BillTest` for your implementation of `Bill`. To create a JUNIT class in INTELLIJ:

- Create a new Java class (naming convention is: [name of class to be tested]Test, leaving out the square brackets)
- Make JUNIT components available by importing it `import org.junit.jupiter.api.*`

By importing JUNIT components you can annotate classes with certain directives telling JUNIT what methods have what purpose in testing. Annotations are prefixed with an `@`-symbol and put right above a method.

- 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 three methods. One method called `setUp()`, annotated `@BeforeEach`, that creates a new printer and bill before each test. Another method testing the begin state of a `Bill` (Contains no items) and another method testing whether items are inserted correctly 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` `price`; its `toString` method should return the `text`, and its `getPrice` method should return the `price`.
- The constructor of `Bill` takes a `BillPrinter` object. For the purpose of `BillTest`, a `StringBillPrinter` is the best choice, since it collects the printed items in a `String` that can be tested afterwards.
- To test the correctness of the `String` of the printed items you can use the `strObj.contains(CharSequence s)` method available on `String` objects in conjunction with something like `assertTrue()`
- 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.hotel.Safe` and `ss.hotel.Room`, instead of modifying the classes to include this new functionality.

P-3.11 Specify a class `ss.hotel.PricedSafe` (i.e. provide pre- and postconditions but not yet the method bodies) that extends the `ss.hotel.Safe` class and implements the `ss.hotel.bill.Bill.Item` interface. To add an extra level of certainty that your code works you have to make use of **assert** statements here. These statements can be explicitly enabled to catch certain edge-cases. You can read more about them here: <https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html> See also ECK Section 8.4.1 about assertions. For the `open` and `active` methods mentioned below add an **assert** statement that tests for the precondition. You will use them in Exercise **P-3.15**.

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 be activated with the correct password. Use an instance variable of type `ss.hotel.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`.

P-3.12 File `ss.hotel.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 `getPrice` works properly;
- Test if method `toString` works properly;
- Test that a deactivated safe can be activated with the correct password and is activated and closed after that;
- Test 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;
- Test 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. To include all relevant classes to be checked for coverage you have to edit the RUN CONFIGURATION of you test class. By default only the package containing the test class is included.

- Go to RUN → EDIT CONFIGURATIONS... and select the configuration of the `PricedSafeTest` class.
- In the CODE COVERAGE section, click the +-symbol and select the `ss.week3.hotel` package.
- To now run the test with coverage enabled, select RUN → RUN 'PRICEDSAFETEST' WITH COVERAGE.

The COVERAGE view of INTELLIJ 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?
- Does a high test coverage (in general) indicate that the test itself is of high quality?

If you followed the instructions properly from Exercise **P-3.11**, your implementation of `PricedSafe` contains a couple of **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.

P-3.15 First, 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. Second, enable assertions and execute the program:

- Create a run-configuration for the `PricedSafe` class
- Go to RUN → EDIT CONFIGURATIONS... and go the the configuration for the `PricedSafe` class
- Use MODIFY OPTIONS and choose ADD VM OPTIONS and add `-ea`. This tells the VM to care about assert statements in the `PricedSafe` class
- Run the `PricedSafe` class

Which error do you see? In which cases are assertions useful?


P-3.16 Implement a class `PricedRoom` that extends `ss.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.hotel.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** Create a class `ss.hotel.PricedHotel` that extends `ss.hotel.Hotel`. Change the constructor so that the first room in the hotel is a `PricedRoom`. To do this, you will need to change some fields of `Hotel` from **private** to **protected**. Why?

The `PricedHotel` inherits all functionality from the base class. You can also add functionality.

Add a method `getBill` to this class that receives as parameters the name of a guest, the number of nights the guest spent in the hotel and a `ss.hotel.bill.Printer` for the bill. The method should return an instance of `Bill`, which was created in Exercise P-3.9. 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.hotel.PricedHotelTest` JUNIT tests from the Canvas files, and improve this implementation until it passes the tests without errors.

 **P-3.18** Make a copy the class `ss.hotel.HotelTUI` to `ss.hotel.PricedHotelTUI`. Add the following functionality:

- A new command *bill name nights* that prints the bill for a `Guest` with name `name` for a number of nights. Use the method `PricedHotel.getBill()` and print the bill to the standard output using a `SysoutBillPrinter`.
- 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 Hotel Twente
Commands:
in name ..... checkin guest with name
out name ..... checkout guest with name
room name ..... request room of guest
activate name password .. activate safe, password required for PricedSafe
bill name nights..... print bill for guest (name) and number of nights
help ..... help (this menu)
print ..... print state of the hotel
exit ..... exit

in Richard
Guest Richard is checked into room 101

activate Richard
Wrong params at activation (password required)

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

```

print
Hotel Twente:
  Room 101 (100.0/night):
    rented by: Guest Richard
    safe active: true
  Room 102:
    rented by: null
    safe active: false

bill 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

out Richard
Guest Richard successfully checked out.

print
Hotel Twente:
  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 `bill` and add it as a new **case** in the **switch** statement of the menu.

3.3.2 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 an 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.

Testing and Design By Contract

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

Software engineers generally distinguish four different kind of tests they perform:

1. Unit testing

The focus is on functionality of a single software unit: a method, a function, a class, etc. Multiple unit tests usually cover one unit, and when this is done properly, it makes any maintenance activity performed on such a unit, a quick and pleasant experience: all the relevant unit tests are run first to ensure that they succeed, then the change is brought into the system (and can concern production code or test code or both), then all the relevant unit tests are run again to ensure that the change did not break anything essential in this unit.

2. Integration testing

With some level of confidence in correctness of each individual unit, we can start combining different units within one program, to see how well they work *together*. The focus of integration tests is on finding interface problems among units. There are many desired properties that are simply invisible or just unmeasurable on the individual unit level, and they only come into play when integration tests are written.

3. System testing

The application can also be tested as a whole to ensure that not only its components work properly by themselves or together, but also the entire system operates as expected. If user stories [L2T5] were not used on lower levels, they come into the picture now. System testing is the first level that can ensure that the software system meets the technical, functional and business requirements initially elicited from the stakeholders.

4. Acceptance testing

Finally, the application that had its components tested as individual units, as a cooperating system of units and as a whole, is given to the intended users to determine whether the new version is ready

for release. If the software system passes this level of testing, the next release is made and put in production. A difference with system testing is that system testing is mostly done by developers while acceptance testing is done by stakeholders.

For example, if a software system in question is a *webshop*, then unit testing can be about classes like *Book* or *Product* functioning; integration testing can involve activating fake special events to see if the price of each product is changing automatically to reflect the discount in place; system testing will be running entire sandboxed scenarios including logging into the system, choosing a product to buy and checking out; and acceptance testing will be similar but outside of a sandbox on real data. Acceptance testing can be performed by a randomly chosen fraction of all the users (this is called a *pilot release*) or in a closed group of specially trained or at least warned users (and sometimes is split further into *alpha testing* when done in a trusted group of people who may even have knowledge about internal structure of the system, and *beta testing* when done with strangers outside the developing organisation).

Testing is performed automatically whenever possible. Unit tests are usually run by developers on a regular basis after each change they make in the code. Integration tests are either run similarly or linked to commits or pushes when the code is exposed to the code management system for versioning. System tests may involve additional frameworks for mocking components that are inaccessible in a normal way, for simulating user clicks, etc. Automating the testing process makes it cheaply repeatable.

Any failing tests uncovered at any level, are called *regressions*, so sometimes the process of rerunning tests “just to be sure nothing is broken”, is referred to as *regression testing*.

The later a defect is found, the more expensive it gets: it is not uncommon to fix a failing unit test within minutes by a developer or a pair of developers working together; fixing a failing integration test may involve many more steps of navigating through the code and figuring out which combination of statements is to blame. Fixing a failing system test is even more demanding in required knowledge, in time consumption, in a number of steps, etc. A failing acceptance test is colloquially known as “*works on my machine*”, and can take considerable effort to fix, up to an including a complete redesign of certain system components. As a direct consequence, if anything can be expressed as a unit test, it should be—alternatives are simply more expensive.

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 *software 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.

For example, using code metrics like cyclomatic complexity [L6T1] or structures like system dependence graphs [L6T2], to strengthen the test suite, are white-box techniques, while random input value generation [L2T5] or using a test strategy relying on a state machine [L4T3] produced by a third party, are black-box techniques.

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* [L2T5] 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 as 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: <i>The name or description of Functional Requirement 5</i>
Expected behaviour: <i>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.</i>
Testing result <ul style="list-style-type: none"> • <i>Step 1</i> • <i>Step 2</i> • <i>Step 3</i> <i>Some screenshots, etc</i>

A.3 JUnit 5

In this course, we use JUNIT 5 for testing. JUNIT 5 is supported by development environments such as 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.

One of the topic videos on Programming, also includes a screencast of a short program being developed and tested in TDD style with JUNIT 5.

A.4 System testing

In the programming project you are asked to write system tests for your software. System tests have a clearly defined *scope*: something that you are testing, such as that a certain feature works correctly, or that certain events are handled by the program, that your client can handle loss of connection to the server, or invalid user input, etc. Essentially this consists of a list of clear and specific instructions for a developer to do, as well as predefined inputs to use, and the expected result for the test to succeed. This could be instructions to start the software, perform certain actions and then check if the output is the correct output. Systems tests should be such that another developer can perform the system test and get the same outcome. Thus for every test you should include: a name or number that identifies the test, a test scope or description, specific instructions with test data, and the expected result to assess whether or not the test is successful.

A.5 Design by Contract

When we would like to describe the intended behaviour of a program in a precise way, we can document this in the form of method contracts. Method contracts consist of the following ingredients.

- *Preconditions*
A precondition specifies which conditions have to hold *before* a method is called. We typically have two kinds of preconditions: (1) a condition on the method arguments, and (2) a requirement that the object on which the method is called is in particular state. When we specify a precondition, the convention in this course (and in many tools) is that it is preceded by the keyword `requires`. A precondition is a (side-effect-free) Java expression of type `Boolean`. To improve the expressiveness, also logical connectives like `implies` (`==>`) and quantifiers `\forall` and `\exists` may be used. Calls to query methods can be used in preconditions, as long as the query method does not update the state.
- *Postconditions*
A postcondition specifies which conditions are established by a method, i.e. which properties hold *after* the method finishes. We typically have the following two kinds of postconditions: (1) a property about the return value of the method, for example that the return value is always positive, and (2) a condition that specifies how the internal state of the object on which the method was called is updated. When we specify a postcondition, the convention in this course (and in many tools) is that it is preceded by the keyword `ensures`. A postcondition is a (side-effect-free) Java expression of type `Boolean`. As for preconditions, calls to query methods and extra logical connectives like implication and quantification may be used. Moreover, we use a special keyword `\result` to denote the return value of a method, and we use a keyword `\old` (*expr*) to indicate that the expression *expr* should be evaluated in the pre-state of a method, i.e. in the state in which the method was called.
- *Invariants*
A (class) invariant is a condition that should hold for every state that an object can be in. Class invariants may also specify relations between attributes, and therefore we allow to temporarily break an invariant property *inside* a method body. However, at any state in which a method is called or exited from, the invariant property should hold. We can have two kinds of invariant properties: (1) private invariants that describe properties that should always hold for the private attributes of a class, and (2) public invariants that document publicly visible invariant properties of a class. When we specify a class invariant, the convention in this course (and in many tools) is that it is preceded by the keyword `invariant`. Again, an invariant is a (side-effect-free) Java expression of type `Boolean`, and it may contain calls to query methods and extra logical connectives like implication and quantification.

Method contracts can be considered as a contract between the caller and the method. The intended meaning is the following: “if the precondition is true when the method is called, *then* the method implementation guarantees that the postcondition will be true when the method finishes”. Method contracts force the programmer to be clear and unambiguous about the conditions under which a method may be called, as well as the (intended) semantics (logical meaning) of the implementation.

Preconditions, postconditions and invariants can be used for testing, but they also make it possible to obtain stronger guarantees about the behaviour of a program, namely that it is correct for all possible executions of a program, by using *program verification*. The main difference between testing and verification is that testing provides you a guarantee about one particular execution of the program - the execution that you tested - , while verification provides you a guarantee about all possible executions of a program. However, as the guarantees are stronger, typically also the effort to formally verify a program is bigger.

When verifying programs, often additional annotations need to be used, such as:

- *Assertions*
Assertions specify properties that have to hold whenever control reaches the assertion, i.e. it describes a property that holds at that particular point in the execution of a program. JAVA provides an `assert` instruction. These have to be explicitly enabled, for example, by using `java -ea` or by configuring the IDE correctly¹. When assertions are enabled, they can be used as “runtime checks” in the code, for example to see if a program works in the way to programmer intended it to work. A common use of assertions is 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.
- *Loop invariants*
Loop invariants are properties that specify properties that hold whenever a loop (i.e. a while- or a

¹Please refer to the tool installation guide on Canvas to see how you can enable assertion checking for your full project.

for-loop) is executed. Loop invariants must be true before and after each iteration of the loop. In program verification, 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.

Java Modeling Language

In this module, we use a language called Java Modeling Language (or JML, for short), which is a widely-used annotation language for Java. One nice advantage of using JML is that there is tool support available. During this module, we will only use type checking facilities of the tool support, but in some of the challenge exercises, you can also experiment with more advanced tool support of OpenJML.

JML is a large language, with many different specification constructs. There is an online reference manual available via <http://www.jmlspecs.org>. On this webpage, you can also find references to tools and papers about JML. The remainder of this appendix describes the basic ingredients of JML (method specifications, class invariants, and loop invariants) that we will use in the Software Systems module.

B.1 JML Method Contracts

Ingredients of a Method Contract So, what exactly is a method contract? A method contract consists of two things: it describes what is expected from the code that calls the method, and it provides guarantees about what the method will actually do.

The expectations on the caller are called the *precondition* of the method. Typically, these will be conditions on the method's parameters, *e.g.*, the argument should be a positive integer, or a non-null pointer, but the precondition can also describe that the method can only be called when the object is in a particular state. In JML, every precondition expression is preceded by the keyword `requires`.

The guarantees provided by the method are called the *postcondition* of the method. They describe how the object's state is changed by the method, or what the expected return value of the method is. A method only guarantees its postcondition to hold whenever it is called in a state that respects the precondition. If it is called in a state that does not satisfy the precondition, then no guarantee is made at all. In JML, every postcondition expression is preceded by the keyword `ensures`.

JML specifications are written as special comments in the Java code, starting with `/*@` or `//@`. The `@` sign allows the JML parser to recognise that the comment contains a JML specification. Sometimes, JML specifications are also called *annotations*. The preconditions and postconditions are basically just Java expressions (of Boolean type). Typically the JML specifications of a method come after the Javadoc. They cannot be mixed: the `@` sign must directly follow the comment syntax, without any whitespace.

Example 2.1 Figure B.1 contains an example of a basic JML specification. The specification should be understood independent of the implementation, therefore we just provide a stub implementation here. The specification contains contracts for the methods in a class `Student`, representing a typical UT student.

We discuss the different aspects of this example in full detail.

```
package ss.jml;

public class Student {

    public static final int BACHELOR = 0;
    public static final int MASTER = 1;

    /*@ pure */ public String getName() {
        return null;
    }

    /*@ ensures \result == BACHELOR || \result == MASTER;
    /*@ pure */ public int getStatus() {
        return 0;
    }

    /*@ ensures \result >= 0;
    /*@ pure */ public int getCredits() {
        return 0;
    }

    /*@ ensures getName().equals(n);
    public void setName(String n) {
    }

    /*@ requires c >= 0;
        ensures getCredits() == \old(getCredits()) + c;
    */
    public void addCredits(int c) {
    }

    /*@ requires getCredits() >= 180;
        requires getStatus() == BACHELOR;
        ensures getCredits() == \old(getCredits());
        ensures getStatus() == MASTER;
    */
    public void changeStatus() {
    }

}
```

Figure B.1: First JML example specification Student

- For method `getName`, we specify that it is a pure method, *i.e.*, it may not change the state in any way (in other words: it may not have any (visible) side effects). Only pure methods may be used in specifications, because these do not change the state.
- Method `getStatus` is also pure. In addition, we specify that its result may only be one of two values: `BACHELOR` or `MASTER`. To denote the return value of the method, the reserved JML-keyword `\result` is used.
- For method `getCredits` we also specify that it is pure, and in addition we specify that its return value must be non-negative; a student thus never can have a negative amount of credits.
- Method `setName` is non-pure, *i.e.*, it may change the state. Its postcondition is expressed in terms of the pure methods `getName` and `equals`: it ensures that after termination the result of `getName` is equal to the parameter `n`.
- Method `addCredits`'s precondition states a condition on the method parameters, namely that only a positive number of credits can be added. The postcondition specifies how the credits change. Again, this postcondition is expressed in terms of a pure method, namely `getCredits`. Notice the use of the keyword `\old`. An expression `\old(E)` in the postcondition actually denotes the value of expression *E* in the state where the method call started, the *pre-state* of the method. Thus the postcondition of `addCredits` expresses that the number of credits only increases: after evaluation of the method, the value of `getCredits` is equal to the old value of `getCredits`, *i.e.*, before the method was called, plus the parameter `c`.
- Method `changeStatus`'s precondition specifies that this method only may be called when the student is in a particular state, namely he has obtained a sufficient amount of credits to pass from the Bachelor status to the Master status. Moreover, the method may only be called when the student is still having a Bachelor status. The postcondition expresses that the number of credits is not changed by this operation, but the status is. Notice that the two preconditions and the two postconditions of `changeStatus` are written as separate `requires` and `ensures` clauses, respectively. Implicitly, these are assumed to be joined by conjunction, thus the specification is equivalent to the following specification:

```
/*@ requires getCredits() >= 180 && getStatus() == BACHELOR;
    ensures getCredits() == \old(getCredits()) && getStatus() == MASTER;
*/
public void changeStatus() { ... }
```

Specifications and Implementations Notice that the method specifications are independent of possible implementations. One could imagine different implementations of this class, as long as they respect the specification. One obvious implementation is using a field `credits` that keeps track of the number of credits earned by the student. However, an alternative implementation is to keep track of a list of courses as well as the credits earned for each course and to compute the total number of credits as the sum of the credits of the individual courses.

Method specifications do not always have to specify the exact behaviour of a method; they give minimal requirements that the implementation should respect.

Example 2.2 Consider the specification in Figure B.1 again. The method specification for `changeStatus` prescribes that the credits may not be changed by this method. However, method `addCredits` is free to update the status of the student. So for example, an implementation that silently updates the status from Bachelor to Master whenever appropriate is within this specification.

```
/*@ requires c >= 0;
    @ ensures getCredits() == \old(getCredits()) + c;
    @*/
public void addCredits(int c) {
    credits = credits + c;
    if (credits >= 180) { status = MASTER; }
}
```

Notice also that both `addCredits` and `changeStatus` would be free to change the name of the student, according to the specification, even though we would typically not expect this to happen. A way to avoid

this, is to add explicitly conditions `getName().equals(\old(getName()))` to all postconditions. JML provides way to capture this more concisely, but we will not go into more details here (if you would like to know more, you could look at the reference manual).

Default Specifications You might have wondered why not all specifications in `Student` have a pre- and a postcondition. Implicitly though, they have. For every specification clause, there is a default. For pre- and postconditions this is the predicate `true`, *i.e.*, no constraints are placed on the caller of the method, or on the method's implementation.

Example 2.3 *Thus for example the specification of method `getStatus` actually is the following:*

```
/*@ requires true;
    ensures status == BACHELOR || status == MASTER;
*/
public int getStatus() {
    return status;
}
```

However, there is one exception to this. In JML all reference values are implicitly assumed to be non-null, except when explicitly annotated otherwise (using the keyword `nullable`).

Example 2.4 *This means that the methods `getName` and `setName` have implicit pre- and postconditions about the non-nullity of the parameter and the result. Explicitly, their specifications are as follows:*

```
/*@ requires true;
    ensures \result != null;
*/
/*@ pure */ public String getName() {return "";}

/*@ requires n != null;
    ensures getName().equals(n);
*/
public void setName(String n) {}
```

Notice that the non-null by default also can have some unwanted effects, as illustrated by the following example.

Example 2.5 *Consider the following declaration of a `LinkedList`.*

```
public class LinkedList {
    private Object elem;
    private LinkedList next;
    ....
}
```

Because of the non-null by default behaviour of JML, this means that all elements in the list are non-null. Thus the list must be cyclic, or infinite. This is usually not the intended behaviour, and thus the next reference should be explicitly annotated as `/@ nullable @*/`.*

Specification Expressions Above, we have already seen that standard Java expressions can be used as predicates in the specifications. These expressions have to be side-effect-free, thus for example assignments are not allowed. As also mentioned above, these predicates may contain method calls to pure methods.

In addition, JML defines several specification-specific constructs. The use of the `\result` and `\old` keywords has already been demonstrated in Figure B.1, and the official language specification contains a few more of these. Besides the standard logical operators, such as conjunction ¹ `&`, disjunction `|` and negation `!`, also extra logical operators are allowed in JML specifications, *e.g.*, implication `==>`, and logical equivalence `<==>`. Also the standard quantifiers \forall and \exists are allowed in JML specifications, using keywords `\forallall` and `\existsists`.

Example 2.6 *Using these, we can specify for example that an array argument should be sorted.*

¹Since expressions are not supposed to have side effects or terminate exceptionally, in JML the difference between logical operators `&` and `&&`, and `|` and `||` is not important.

```
//@ requires (\forall int i, j; 0 <= i & i < j & j < a.length; a[i] <= a[j]);
public ... manipulateArray(int [] a) { ...
```

The first argument (`int i, j`) is the declaration of the variables over which the quantification ranges. The (optional) second argument (`0 <= i & i < j & j < a.length`) defines the range of the values for this variable, and the third argument is the actually universally quantified predicate (`a[i] <= a[j]` in this case).

Validating Method Contracts A way to validate your specifications is by inserting assertions at appropriate positions in your program. Validating a postcondition means that one validates one's own implementation. Therefore, it should not be necessary to always have the postcondition check enabled.

As said, there is a wide range of tools available for JML. In particular, many of these tools provide support for run-time checking. This means that they automatically transform the code to validate the pre- and postconditions during the execution. This can be very useful in the testing phase. The tool IntelliJML that we use in the module only provides support for syntax checking and autocompletion, however you can also use a tool like OpenJML to do so-called run-time checking, meaning while the program is running.

B.2 Class Invariants

Consider again the specification of `Student` in Figure B.1. If we look carefully at the specifications and the description that we give about the student's credits, we notice that implicitly we assume some properties about the value of `getCredits` that hold throughout. For example, we wrote above:

“a student thus never can have a negative amount of credits”

and also

“the number of credits only increases.”

But if we would like to make explicit that we assume that these properties always hold, we would have to add this to *all* the specifications in `Student`, and thus in particular also to all methods that do not relate at all to the number of credits. Thus for example, we would get the following specification:

```
/*@ requires getCredits() >= 0;
   @ ensures \result == BACHELOR || \result == MASTER;
   @ ensures getCredits() >= \old(getCredits());
   */
/*@ pure */ public int getStatus() { return 0; }
```

Clearly, this is not desired, because specifications would get very large, and besides describing the intended behaviour of that particular method, they also describe properties that are related to how an object can evolve over time.

Therefore, JML provides class invariants to restrict how the internal state of an object can change during the object's lifetime.

An object invariant² is a predicate over the object state that holds in all *visible* states of an object. A visible state of an object is defined to be any state in which a method call to the object either starts or terminates. Thus, an invariant *I* is implicitly added as a precondition and a postcondition to every method in the class. In addition, also the post-states of the constructor are visible states, thus any constructor has to ensure that the invariant is established.

Note that we only show how the first property is specified as an object invariant. The second property, namely that the credits can only increase, requires using the `\old` keyword, which is not supported by the `invariant` keyword in JML. There are other ways to specify invariants in JML, which also allow using `\old` expression, however, this is not discussed in this document.

Example 2.7 Figure B.2 shows three possible invariants that can be added to interface `Student` (and removes method specifications that now have become superfluous). These specify that credits are never non-negative; a student's status is always either Bachelor or Master, and nothing else; and if a student's status is Master, he or she has earned more than 180 credits.

²Not to be confused with loop invariants, as discussed below.

```

package ss.jml;

public class StudentWithInv {

    public static final int BACHELOR = 0;
    public static final int MASTER = 1;

    //@ invariant getCredits() >= 0;
    //@ invariant getStatus() == BACHELOR || getStatus() == MASTER;
    //@ invariant getStatus() == MASTER ==> getCredits() >= 180;

    /*@ pure */ public String getName() {
        return null;
    }

    /*@ pure */ public int getStatus() {
        return 0;
    }

    /*@ pure */ public int getCredits() {
        return 0;
    }

    //@ ensures getName().equals(10);
    public void setName(String n) {
    }

    /*@ requires c >= 0;
        ensures getCredits() == \old(getCredits()) + c;
    */
    public void addCredits(int c) {
    }

    /*@ requires getCredits() >= 180;
        requires getStatus() == BACHELOR;
        ensures getCredits() == \old(getCredits());
        ensures getStatus() == MASTER;
    */
    public void changeStatus() {
    }

}

```

Figure B.2: Interface Student with class-level specifications

Of course, instead of specifying invariants, one could also add these specifications to all pre- and postconditions explicitly. However, this means that if you add a method to a class, you have to remember to add these pre- and postconditions yourself. Moreover, invariants are also inherited by subclasses (and by implementations of interfaces). Thus any method that overrides a method from a superclass still has to respect the invariants. And any method that one adds in the subclass also has to respect the invariants from the superclass. Specifying the invariants centrally as class invariants, thus leads to a very nice separation of concerns.

An important point to realise is that invariants have to hold only in all *visible object states*, i.e., in all states in which a method is called or terminates. Thus, inside the method, the invariants may be temporarily broken.

Example 2.8 *The following possible implementation of `addCredits` is correct, even though it breaks the invariant that a student can only be studying for a Master if he or she has earned more than 180 points inside the method: if `credits + c` is sufficiently high, the status is changed to Master. After this assignment the invariant does not hold, but because of the next assignment, the invariant is re-established before the method terminates.*

```
/*@ requires c >= 0;
   @ ensures getCredits() == \old(getCredits()) + c;
   @*/
public void addCredits(int c) {
    if (credits + c >= 180) {status = MASTER;} // invariant broken!
    credits = credits + c;
}
```

Validating Class Invariants If one wishes to validate the specified class invariants, assertions should be added at the beginning and end of every method call. Clearly, this is not a task that you want to do manually, but where tool support is necessary. Again, tools like OpenJML provides support for doing this.

B.3 Loop invariants

JML also provides support for loop invariants.

Example 2.9 *Figure B.3 shows an example of a non-trivial loop invariant. Method `thirdPower` computes n^3 without actually using the power function. Its loop invariant describes the intermediate values for all local variables.*

Loop invariants also are very common for methods that iterate over arrays.

Example 2.10 *Method `search` in Figure B.4 checks whether a given value occurs in an array. The loop invariant expresses that `found` is true if and only if the value was among the elements in the array that have been examined so far.*

Loop invariants as for the method `search` show a very common loop invariant pattern for methods iterating over an array. All the elements that have been examined so far respect a certain property, and since the loop terminates when all the elements in the array have been examined, from this loop invariant and the negation of the loop condition we can conclude a property for all the elements in the array. The loop invariant restricting the range of the value of the loop variable `i` ($0 \leq i \ \&\& \ i \leq a.length$) is typically always needed, but not sufficient alone.

Notice that a loop invariant could contain an `\old(E)` expression. This refers to the value of the expression `E` before the method started, *not* to the value of `E` at the previous iteration of the loop.

```

package ss.jml;

public class Power {

    /*@ requires n >= 0;
       ensures \result == n * n * n;
    */
    public int thirdPower(int n) {
        int u = 0;
        int v = 1;
        int w = 6;
        int k = 0;
        //@ loop_invariant 0 <= k && k <= n;
        //@ loop_invariant u == k * k * k;
        //@ loop_invariant v == 3 * k * k + 3 * k + 1;
        //@ loop_invariant w == 6 * (k + 1);
        while (k < n) {
            u = u + v;
            v = v + w;
            w = w + 6;
            k = k + 1;
        }
        return u;
    }
}

```

Figure B.3: Method `thirdPower` with loop invariant

```

package ss.jml;

public class Search {

    /*@ requires a != null;
       ensures \result ==
           (\exists int i; 0 <= i && i < a.length; a[i] == val);
    */
    public boolean search(int[] a, int val) {
        boolean found = false;
        int i = 0;
        //@ loop_invariant
           found == (\exists int j; 0 <= j && j < i; a[j] == val);
        loop_invariant 0 <= i && i <= a.length;
        loop_invariant a != null;
        //@
        while (i < a.length & !found) {
            if (a[i] == val) {
                found = true;
            }
            i++;
        }
        return found;
    }
}

```

Figure B.4: Method `search` with loop invariant over array

Appendix C

Design Project Rubric

On the following pages, you can find the rubrics that we use to grade the design project.

For each row, if multiple cells are applicable, then the *leftmost* applicable cell is selected.

General Grading Rubric - Design Project

For the full points (excellent), you must additionally fulfil **ALL** the requirements of the 'good' column

Partial grade= points /60 * 10

0

Information

	Excellent (6)	Good (4)	Insufficient (2)	Bad (0)	Scored points
Overall Quality of the Main Report & Final Submission	The diagrams are explained and placed in a meaningful way within the main text (not all the diagrams are placed in the appendices). The writing errors are very minimal.	The report is mostly well-written. There are some writing errors, but overall, the report is comprehensible, easy to navigate, and has a consistent writing style. It contains a title, the authors and a table of contents. The report has a good introduction and an overview of individual contributions. Note that the final submission must be formatted/packaged as requested.	One of the following is applicable: The report is incomprehensible and contains an unacceptable amount of writing errors. The report is missing two of the three components: Title / Authors / Table of contents. Inconsistencies in writing (different styles in different sections) and/or layout. Missing introduction. Missing the overview of individual contributions. The final submission is not formatted/packaged as requested (See "submission and grading" in the manual).	Two or more of the following are applicable: The report is incomprehensible and contains an unacceptable amount of writing errors. The report is missing two of the three components: Title / Authors / Table of contents. Inconsistencies in writing (different styles in different sections) and/or layout. Missing introduction. Missing the overview of individual contributions. The final submission is not formatted/packaged as requested (See "submission and grading" in the manual).	
	Excellent (6)	Good (4)	Insufficient (2)	Bad (0)	Scored points
Interview Report and feedback from interviewee	The interview is summarized and quoted in the report. The conversation was realistic and well-organized (intro, questions, outro). The members maintained a professional tone during the conversation and were focused on the interviewee. After the interview, an (additional) list of requirements is given. The questions covered more in depth topics about the project.	Either the summary of the interview or the questions/ answers are present. The report is detailed and contains important insights regarding the project. The team was on time. The team had sufficient knowledge of the task and prepared relevant questions. The meeting was done properly (intro, questions, outro). The summary report was sent on time after the interview.	One of the following is applicable: No interview report or the report is very brief. Very little information was acquired during the interview. The interview was cancelled on short notice or without explanation, or the team did not appear on time. The summary report was not sent after the interview. Little knowledge of the task at hand, ill-prepared.	Two or more of the following are applicable: No interview report or the report is very brief. Very little information was acquired during the interview. The interview was cancelled on short notice or without explanation, or the team did not appear on time. The summary report was not sent after the interview. Little knowledge of the task at hand, ill-prepared.	
	Excellent (6)	Good (4)	Insufficient (2)	Bad (0)	Scored points

Requirements and Stakeholders/actors	<p>There is a very extensive requirements list</p> <p>The requirements include error reporting</p> <p>There should be at least one requirement for each actor identified.</p> <p>There is a role description for each stakeholder/ actor identified</p> <p>There is a systematic introduction for each category.</p>	<p>There is an extensive requirements list which depicts the system.</p> <p>The requirements list resembles the use cases in the use case diagrams.</p> <p>Both functional and non-functional requirements are identified.</p> <p>The requirements are ordered and categorized (e.g. non-functional are grouped based on the 7 categories presented in the lecture L2T1; functional requirements are categorized based on the main functionalities of the system).</p> <p>All crucial stakeholders/ actors are identified.</p>	<p>One of the following is applicable:</p> <p>The requirements list does not have any resemblance with the use cases in the use case diagram</p> <p>The requirements are not systematically ordered and/or the identified requirements only consists of functional or they only identified non-functional requirement.</p> <p>Some crucial stakeholders/ actors are missing.</p> <p>Requirements are in conflict with other parts of the report</p>	<p>Two or more of the following are applicable:</p> <p>The requirements list does not have any resemblance with the use cases in the use case diagram</p> <p>The requirements are not systematically ordered and/or the identified requirements only consists of functional or they only identified non-functional requirement.</p> <p>Some crucial stakeholders/ actors are missing.</p> <p>Requirements are in conflict with other parts of the report</p>	
	Excellent (6)	Good (4)	Insufficient (2)	Bad (0)	Scored points
User Stories L2T5	<p>The user story cases are described in great detail. They are placed on a well-organized list. Each extended user story is depicted/illustrated/exemplified by an object diagram</p>	<p>There is an extensive user stories list</p> <p>The user stories are consistent with the requirements</p> <p>The user stories are consistent with the use case diagrams (all use cases in the list should be used in one of the use cases diagrams)</p> <p>The user stories are extensive and complete *</p>	<p>One of the following is applicable:</p> <p>The user stories are not consistent with the requirements list</p> <p>The user stories are not consistent with the use case diagram</p> <p>The user stories are not extensive</p> <p>User stories follow no particular pattern (e.g., L5T1)</p>	<p>Two or more of the following are applicable:</p> <p>The user stories are not consistent with the requirements list</p> <p>The user stories are not consistent with the use case diagram</p> <p>The user stories are not extensive</p> <p>User stories follow no particular pattern (e.g., L5T1)</p>	
	Excellent (6)	Good (4)	Insufficient (2)	Bad (0)	Scored points
Testing and Metrics	<p>Clear overview (possibly tabularly) with test type, description, test step, expected results and verdicts. The most crucial items of the system to be tested are mentioned</p> <p>Meaningful metrics and detailed reasoning (with specific examples) as to why they are meaningful and useful in the context.</p>	<p>The group specified a test strategy of sufficient quality (incl.: the objective of testing, the scope of testing, the test levels and the testing methodology)</p> <p>A list of metrics with a short and abstract explanation of how they relate to the system</p>	<p>One of the following is applicable (testing):</p> <p>Test plan is missing or, it is not specific and contains only superficial terms which are not related to the system.</p> <p>Unstructured way of testing</p> <p>Just mentioning the metrics presented in lecture, without relating them to the actual system</p>	<p>Two or more of the following are applicable (testing):</p> <p>Test plan is missing or, it is not specific and contains only superficial terms which are not related to the system.</p> <p>Unstructured way of testing</p> <p>Just mentioning the metrics presented in lecture, without relating them to the actual system</p>	

* Regular User Story: structure is <actor><verb> or <actor><verb><noun> (L5T1) e.g.: “As a student, I want to borrow books”

Extended User Story: add more details to the regular user story, e.g.: “The student, FirstName LastName, borrows two books, *Jane Eyre* by Charlotte Bronte, and *Les Miserables* by Victor Hugo, on August 10th, 2022, from Vrijhof library.

Diagrams

	Excellent (6)	Good (4)	Insufficient (2)	Bad (0)	Scored points
Class Diagram	The diagram is easy to read and navigate through. The class diagram has more meaningful classes than just the basic ones. The generalisations and associations are used properly. The diagram shows the proper usage of composition and aggregation.	The class diagram complies with the UML syntax**. The diagram relates to the system. It consists of all the obvious classes. All classes are given a singular noun name and connected via generalisations or associations. The association names are clear and denoted with multiplicity.	The class diagram consists of incorrect relations and/or multiplicities. Associations are consistently badly named. The diagram misses some essential elements and is very hard to read. Classes are not correctly named.	The class diagram models activities through associations and/or has elements that do not store data. The diagram shows multiple modelling mistakes and incorrect interpretations of the information which could have been prevented.	
	Excellent (6)	Good (4)	Insufficient (2)	Bad (0)	Scored points
Activity Diagrams	The diagrams are easy to read and navigate through. There are more than two activity diagrams of decent quality. At least one diagram is modelled in very much detail. The chosen topics which are modelled are relevant.	The activity diagrams comply with the UML syntax**. The diagrams relate to the system. There are at least two activity diagrams. Naming is done properly as well as the control statements such as forks/rejoins. Activities are placed in the swimlane of the correct actor.	The diagram could be syntactically correct however it has the wrong semantics. The diagrams miss some minor important activities.	There are not enough diagrams presented. The diagrams are missing key activities.	
	Excellent (6)	Good (4)	Insufficient (2)	Bad (0)	Scored points
Sequence Diagrams	The diagrams are easy to read and navigate through. There are more than two sequence diagrams of decent quality. At least one diagram is modelled in very much detail. The chosen topics which are modelled are quite complicated.	The sequence diagrams comply with the UML syntax**. The diagrams relate to the system. There are at least two sequence diagrams. Creating, activating and ending a lifeline are used properly.	The diagram could be syntactically correct however it has the wrong semantics. The diagrams miss some minor events. One of the topics is quite easy. No diagram shows conditions, for loops, nor opt clauses.	The diagrams use incorrect syntax to a level that makes them incomprehensible. There are not enough diagrams presented. Diagrams are missing key events. The chosen topics are quite easy.	
	Excellent (6)	Good (4)	Insufficient (2)	Bad (0)	Scored points

State Machine Diagrams	The diagram is easy to read and navigate through. There are more than two state machine diagrams of decent quality. The diagrams have at least 27 transitions in total. A diagram includes a composite state.	The state machine diagrams comply with the UML syntax**. The diagram relates to the system. There are at least two state machine diagrams. Transitions between states are clearly defined events or performed actions.	The diagrams show less than 20 transitions in total. There are syntactic errors	There are not enough diagrams presented. Diagrams are missing key states. The chosen topics are quite easy.	
	Excellent (6)	Good (4)	Insufficient (2)	Bad (0)	Scored points
Use Case Diagrams	The diagrams are easy to read and navigate through. The diagrams are complete and very detailed.	The use case diagrams comply with the UML syntax**. The diagrams relate to the system. The diagrams show only things that take place within the system. The use cases are described in imperative form. The diagrams are complete and clear.	Use cases are not written in the correct form. Extensions are placed where none should be. The diagram misses some minor parts of the system.	The diagram shows part of the system as an actor. The diagram misses crucial parts of the system interaction.	

** see lecture 😊 or the specification on <https://www.visual-paradigm.com/>

Programming Project Rubric

On the following pages, you can find the rubric that we use to grade the programming project. We have a rubric for the Functionality, for the Software and for the Report aspects of the project.

For each row, if multiple cells are applicable, then the *leftmost* applicable cell is selected. For each row, a percentage for the weight of that row is given. The grade for that row corresponds to the cell that is selected for that row. These grades are indicated at the top of the rubric.

The grades for Functionality, Software and Report are first calculated as a weighted average of the grades for the different rows and rounded to 1 decimal precision, and are then combined to produce the final project grade, rounded to 1 decimal precision.

Desk reject

The project is rejected if source code is missing, if the report is missing, if required parts (game logic, server/client) are clearly unfinished.

Functionality (20% of project grade)

Criterion	1 (poor / missing)	4 (subpar)	6 (sufficient)	8 (good)	10 (excellent)
Game Logic (25%)	Common execution flows in the game logic are incorrect. Examples: bad implementation of game rules or gameover checks, etc.	One of the following applies: <ul style="list-style-type: none">• Game logic is almost correct except for some uncommon corner cases.• Implementation supports only a few games or only one game simultaneously.	<ul style="list-style-type: none">• All game rules are correctly implemented.	As sufficient , and the full game state and logic is independent of other components (client/server/protocol).	As good , and game logic methods do not crash with unchecked exceptions from incorrect usage (invalid moves, negative indices, null pointers, etc.)
Networking (25%)	There are major protocol violations. Examples: bad parsing of messages, sending invalid messages, sending messages that are not in the protocol.	One of the following applies: <ul style="list-style-type: none">• Connection loss results in a crash or stack trace in networking code.• Connection loss is not reported to the client/server classes.• Protocol messages are not generated by networking code (but e.g. by client/server code).	<ul style="list-style-type: none">• The protocol is implemented correctly or almost correctly.• Connection loss does not result in a crash or stack trace.• Connection loss is reported to the client/server classes.• Protocol messages are generated in the networking code.	As sufficient , and: <ul style="list-style-type: none">• The protocol is implemented perfectly.• Networking code is thread-safe, there are no race conditions.	As good , and: <ul style="list-style-type: none">• A malicious attacker cannot make the software crash via the network connection, for example, bad messages.
Client (25%)	One of the following applies: <ul style="list-style-type: none">• It is not possible to play a game without looking at the source code or knowing the protocol or internal game details.• The client asks to type protocol messages that are then sent to the	Three or four of the points of sufficient .	<ul style="list-style-type: none">• The client is usable enough to play several games without reconnecting.• The client asks for the hostname and port of the server.• The client gracefully handles connection loss.	As sufficient , and: <ul style="list-style-type: none">• The client is user-friendly and intuitive. It is always clear what the client expects from the user. Anyone can play using the UI, even if they don't know the game	As good and: <ul style="list-style-type: none">• At any time, the user can ask for a list of allowed commands, and use non-game commands such as requesting the list of online players.• The client supports

	server. • At most two of the points of sufficient .		<ul style="list-style-type: none"> • The user can ask the AI to play a full game • The user can ask the AI to suggest a valid move. 	yet. • The client does not crash on invalid input.	colors or is graphical. • The client has sound effects.
Server (25%)	At most two of the points of sufficient .	Only three of the points of sufficient .	<ul style="list-style-type: none"> • The game asks for a port to listen to on startup. • Multiple games can be played simultaneously. • A client can play multiple games after each other. • The server ends a game when one of the players loses connection to the server, by informing the other client according to the protocol. 	As sufficient , and <ul style="list-style-type: none"> • The server is thread-safe, there are no race conditions. • The server is properly multithreaded, that is, games progress independent of each other. 	As good and the following: <ul style="list-style-type: none"> • When players disconnect and games end, references to related objects are correctly removed.

Software (40% of project grade)

Criterion	1 (poor / missing)	4 (subpar)	6 (sufficient)	8 (good)	10 (excellent)
Packaging (10%)	Both errors from subpar are applicable.	One of the following applies: <ul style="list-style-type: none"> • There are compilation errors or failed tests. • Required libraries (other than JUnit) are missing. • Exported Javadoc is missing. 	<ul style="list-style-type: none"> • The project builds without errors and passes all tests. • Any required libraries other than JUnit are included. • Javadoc is correctly exported 	As sufficient , and one of the following: <ul style="list-style-type: none"> • Each program (server, client) is exported as an executable jar file. • There is a proper README file with a description, building requirements and instructions, testing instructions and how to run the software. 	As good , and both points apply.
CODE QUALITY AND TESTING					
Clean code (10%)	We grade this by considering six items. If an item is fulfilled, it counts as 2 points. If an item is <i>mostly</i> fulfilled (violated only incidentally or in a minor way), it counts as 1 point.				

	<ul style="list-style-type: none"> • No Checkstyle violations. • Constants are used where appropriate. • No obvious duplicate or repetitive code, i.e., good code reuse. • Methods are simple and serve a clear goal. • Variables and methods have descriptive names. • No dead code (commented code or unused methods or fields). 				
	5 points or fewer	6-7 points	8-9 points	10-11 points	12 points
Tests (25%)	One of the following applies: <ul style="list-style-type: none"> • Gameover conditions are not checked. • Common execution flows are not checked. • Tests are trivial (can't detect actual bugs) 	<ul style="list-style-type: none"> • There are non-trivial game logic tests that check common execution flows and gameover conditions. 	<ul style="list-style-type: none"> • Tests cover all edge cases of game logic, i.e., at least 90% test coverage of methods related to moves and gameover conditions. • There is a test that plays a full game from beginning to end with random legal moves. • Tests are not trivial, i.e., can detect actual bugs. 	As sufficient , and: <ul style="list-style-type: none"> • All tests are documented (with Javadoc and comments). • Each test is simple and has a clear goal. 	As good , but the server is also tested: <ul style="list-style-type: none"> • Automated tests that check whether the protocol is handled correctly. • An automated test that plays several games on the server.
SOFTWARE DESIGN					
Encapsulation (10%)	Classes directly access fields of other classes.	Some fields are not private but they are not actually accessed by other classes.	All fields except constants are private.	As sufficient , and methods are only public when intended to be used from other classes.	As good , and also other classes cannot gain indirect access to fields that are managed by the object, for example via getters and setters.
Structure (10%)	All items from subpar apply.	One of the following applies: <ul style="list-style-type: none"> • There is no clear separation of concerns between classes. Many classes have multiple responsibilities or no clear purpose. • There is only a single component. 	<ul style="list-style-type: none"> • There is a clear separation of concerns between most classes. Some classes may have either no clear purpose or multiple responsibilities. • There are clearly defined components, reflected in the package structure. 	As sufficient and <ul style="list-style-type: none"> • No classes carry multiple responsibilities that should have been placed elsewhere. 	As good and <ul style="list-style-type: none"> • Interfaces are applied where appropriate to ensure low coupling between components. • The network protocol can be replaced without affecting the rest of the code.
Design Patterns (10%)	This is graded based on how many design patterns are correctly applied. <ul style="list-style-type: none"> • The MVC pattern is applied such that the UI code is separated from non-UI code. • The Listener pattern is applied. 				

	<ul style="list-style-type: none"> • The Strategy pattern is applied. • A creational pattern such as the Factory or Builder pattern is applied. 				
	0 patterns applied	1 pattern applied	2 patterns applied	3 patterns applied	4 patterns applied
Exceptions (5%)	None of the points of sufficient applies, i.e., <ul style="list-style-type: none"> • No sensible custom exceptions are defined and used appropriately • Exceptions are caught and ignored without a good explanation; or checked exceptions that can be expected to occur, print a stack trace. • Exceptions that are not thrown (checked or unchecked) are caught. 	Only one point of sufficient applies.	Two of the following: <ul style="list-style-type: none"> • One or more sensible custom exceptions are defined and used appropriately. • All checked exceptions are properly handled. • All caught unchecked exceptions can be expected to be thrown. 	All of the points of sufficient apply.	As good , and: <ul style="list-style-type: none"> • Most expected unchecked exceptions are properly handled. • All thrown exceptions are documented with appropriate Javadoc at the method-level.
CODE DOCUMENTATION					
Javadoc (10%)	An insignificant portion of the public classes and methods of the game logic have been documented with reasonable Javadoc, or no Javadoc (of quality) is present at all.	One of the following applies: <ul style="list-style-type: none"> • Class-level Javadoc is missing on most classes • Javadoc is missing on some public methods of the game logic 	<ul style="list-style-type: none"> • Most classes and interfaces have reasonable class-level Javadoc • All public methods of the game logic have reasonable Javadoc 	<ul style="list-style-type: none"> • All classes and interfaces have reasonable class-level Javadoc. • Most methods have reasonable Javadoc. 	There is reasonable Javadoc on all classes, on all methods, and on all packages.
	Reasonable Javadoc is: a concise and complete description for other programmers what a class is for or what a method does (in relation to the parameters), and possible return values/exceptions.				
Pre-/postconditions (5%)	At most one item of sufficient applies.	Only two items of sufficient apply.	<ul style="list-style-type: none"> • Most methods of game logic have reasonable nontrivial pre- and postconditions. • There are sensible pre- and postconditions beyond merely stating that objects are not null. • The <code>void</code> keyword is used appropriately wherever 	<ul style="list-style-type: none"> • All methods of game logic have reasonable nontrivial pre- and postconditions. • There is sensible use of <code>forall</code> or <code>lexists</code> Full modeling of the game logic is not required.	As good , but also sensible class invariants are defined for the fields.

			applicable.		
Comments (5%)	Few or no informative comments are present.	Many unnecessary comments are present that overload the reader with superfluous information.	<ul style="list-style-type: none"> Comments are present in most places where they are necessary. Most comments are informative to the reader. However, there are numerous places where comments are either missing or do not provide (enough) clarity to the reader. 	<ul style="list-style-type: none"> Comments are present in most places where they are necessary. All comments are informative and have a clear meaning 	<ul style="list-style-type: none"> Comments are used to aid other programmers in the understanding of the code, and document all non-trivial or implicit implementation details, without adding too much verbosity. Complicated code blocks are clarified by the comments.

Report (40% of project grade)

Criterion	1 (poor / missing)	4 (subpar)	6 (sufficient)	8 (good)	10 (excellent)
SOFTWARE DESIGN					
Explanation realized program design (25%) <i>(Why is this a good design? What decisions led to this?)</i>	Two or more items of subpar apply.	One of the following applies: <ul style="list-style-type: none"> A list of responsibilities and/or classes and their relation is missing. The class structure is not explained or is incomprehensible There are no class diagrams or the class diagrams are not explained in the text. There are no sequence diagrams or the sequence diagrams are not explained in the text. The diagrams are unreadable. 	<ul style="list-style-type: none"> There is a reasonably complete list of responsibilities. There is a list of all classes, a brief description of each class and how the class is related to the responsibilities. The class structure of the software into components and packages is explained with a justification that is not terrible. There is at least one class diagram of the largest component. 	As sufficient and: <ul style="list-style-type: none"> All responsibilities are identified and explained in relation to the project requirements. The justification of the class structure is convincing. 	As good and: <ul style="list-style-type: none"> There is a class diagram of appropriate detail level for every component The description of each class also includes a justification of the other classes each class depends on. At least two nontrivial and well-chosen execution flows are explained by means of a sequence diagram.

			<ul style="list-style-type: none"> • There is at least one sequence diagram of an execution in multiple components. 		
Concurrency mechanism (15%)	<p>An explanation of the concurrency design is missing, is incomprehensible or numerous issues are unidentified.</p> <p>Writing “we use synchronized” without explaining where/how and why it solves what problem. in detail</p>		<ul style="list-style-type: none"> • There is a list of threads and their purpose. • There is a list of shared objects and why each object is shared and by which threads. • There is an explanation why the application is thread-safe, but the explanation is unclear or does not identify all issues or the solutions are unnecessarily complex. 		As sufficient , but the explanation why the application is thread-safe is clear. The mechanisms to ensure safety are explained well and provide a safe way to prevent any concurrency issues, without introducing unnecessary complexity. Conversely, if safety is not guaranteed, the issues are identified and adequate solutions are proposed.
<p>Reflection on Initial Design (5%)</p> <p><i>(How could the design be made even better? In which way was the initial design worse than the realised one?)</i></p>	<p>One of the following applies:</p> <ul style="list-style-type: none"> • There is no initial design • There is no reflection on the initial design or on the final design • The reflection does not identify any major design flaws, despite their obvious presence. • None of the proposed changes to the design (process) would reasonably bring improvements. 		<p>There is reflection on the initial design and on the final design, and, one of the following applies:</p> <ul style="list-style-type: none"> • Not all of the important pros and cons were identified. • Some design mistakes were not recognized or not elaborated upon. • The improvements to some issues do not convincingly resolve all relevant problems or introduce new unidentified problems. 		There is a good reflection on the initial and the final design. The pros and cons of the initial design are listed, as well as possible improvements to both the final design and the design process itself.
<p>Reflection on Final Design (5%)</p> <p><i>(How could the design be made even better? In which way was the initial design worse than the realised one?)</i></p>	<p>One of the following applies:</p> <ul style="list-style-type: none"> • There is no initial design • There is no reflection on the initial design or on the final design • The reflection does not 		<p>There is reflection on the initial design and on the final design, and, one of the following applies:</p> <ul style="list-style-type: none"> • Not all of the important pros and cons were identified. 		There is a good reflection on the initial and the final design. The pros and cons of the initial design are listed, as well as possible improvements to both the final design and the design

	<p>identify any major design flaws, despite their obvious presence.</p> <ul style="list-style-type: none"> None of the proposed changes to the design (process) would reasonably bring improvements. 		<ul style="list-style-type: none"> Some design mistakes were not recognized or not elaborated upon. The improvements to some issues do not convincingly resolve all relevant problems or introduce new unidentified problems. 		process itself.
TESTING					
System tests (15%)	<p>This is graded depending on how many of the following points apply:</p> <ul style="list-style-type: none"> The tests are described in a systematic way and are reproducible, with clear and specific instructions and expected outcomes. All tests have a clear scope, i.e. test a particular feature. There are system tests that test correct handling of malformed inputs (both client and server). Network I/O is tested for handling sudden disconnects (both server and client) There is at least a test that tests features of the server using the client and vice versa. 				
	0 or 1 points	2 points	3 points	4 points	All 5 points
Overall testing strategy (15%)	<p>This is graded depending on how many of the following points apply:</p> <ul style="list-style-type: none"> Incorporation of coverage metrics in explanation, split into different code sections where appropriate. Incorporation of complexity metrics to argue that the tests focus on the riskiest parts of your application. Explanation of how unit tests cover each other's gaps (edge cases and such) Explanation of how system tests and automated unit/integration tests complement each other Explanation of what the least-tested components of your system are and a justification of how much confidence you have in those components. 				
	0 or 1 points	2 points	3 points	4 points	All 5 points
DEVELOPMENT PROCESS					
Reflection on process (10%)	<p>There is no reflection by individual group members on the planning or the collaboration or the reflection does not point out any meaningful successes or shortcomings.</p>		<ul style="list-style-type: none"> All group members provide an individual reflection on both the initial planning and the collaboration in the project. The reflection identifies both successes and shortcomings, but it is unclear what lessons are learned and how this will 		<p>All group members provide an individual reflection on both the initial planning and the collaboration in the project. The reflection identifies both successes and shortcomings and suggests concrete points of improvements for future projects.</p>

			improve future projects. <ul style="list-style-type: none"> • The reflection summarizes how the work was done, i.e., who did what. 		
PRESENTATION					
Report quality (10%)	The report is incomprehensible, poorly structured and/or contains an unacceptable amount of errors, giving the report an unfinished appearance.		<ul style="list-style-type: none"> • The report is mostly well written. Some sentences are odd or clearly needed proofreading, and some grammatical and/or spelling errors are present, but the report is still decently structured and comprehensible. • The report has a cover page that includes the group name, and the names and student numbers of the authors. 		The report is well written and clearly structured. There are no grammatical or spelling errors, apart from some minor mistakes which do not influence the readability of the report.