Software Systems

(Course code: 202500340)
Module 2 of the TCS curriculum
University of Twente
2025-2026 Academic year

Module team:
Fernando Castor (*Module Coordinator*)
Tom van Dijk (*Programming*)
Marieke Huisman (*Programming*)
Alexander Stekelenburg (*Programming*)
Petra van den Bos (*Design*)
Marcus Gerhold (*Design*)
Vadim Zaytsev (*Design*)
Dipti Sarmah (*Security*)
Georgiana Caltais (*Skills*)
Mahboobeh Zangiabady (*Skills*)
Rico van Lingen (*Mathematics*)
Tina Holtkamp (*Module Support*)

17th November 2025

Contents

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). You get acquainted with the waterfall software development process. You also 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 the *testing* of 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 a number of *career 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. The Mathematics course is 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 programmes of the University.

Study Units In terms of content, this module consists of the following threads:

- *Design*: methods and techniques for the high-level design of software systems.
- *Programming*: methods and techniques for the programming and testing of software systems.
- *Skills*: techniques to effectively manage a software development project within a team.
- *Mathematics*: the Calculus 1B course, which covers the theory of mathematical functions and integrals.

The Design thread is contained in the Software Design and Modeling (202500341) study unit. The Programming thread is split into two study units, Introduction to Programming (202500342) and Advanced Programming (202500343). Skills is part of the Software Systems Implementation Project study unit (202500344). The Mathematics thread is contained in the study unit Calculus 1B (202001197). In this module, you will work on a project that puts into practice most of what you learn in Design, Programming, and Skills. As the name indicates, that is part of the Software Systems Implementation Project study unit (202500344). The project is performed in pairs.

Learning Objectives

After successfully finishing the Software Design and Modeling study unit, you should be 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.
- Analyzing relations among entities within a model, among different models and between each model and software code.
- 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.
- Designing programs so as to increase cohesion and reduce coupling.
- Applying software design patterns to avoid tight coupling between different components of software, such as use of interfaces, the strategy pattern and the observer pattern.

After successfully finishing Introduction to Programming, you should be capable of

- Applying the core concepts of imperative programming, such as variables, data types, structured programming statements, recursion, arrays, methods, parameters.
- Applying the core concepts of object-oriented programming, such as encapsulation, abstraction, inheritance and polymorphism.
- Documenting software using Javadoc.
- Collaborating with other students according to the pair programming method.
- Designing and implementing unit tests
- Designing programs so as to increase cohesion and reduce coupling.

After successfully finishing Advanced Programming, you should be capable of

- Implementing programs that use lists, sets, or maps and select the appropriate data structure according to the requirements.
- Defining new exception types and appropriately handling exceptions.
- Writing programs that read from files and write to files.
- Specifying software by defining preconditions and postconditions.
- Explaining problems with concurrent threads (race conditions) and applying basic synchronization mechanisms to eliminate data races, as well as application of the wait-notify pattern.
- Applying the basic concepts and techniques of security engineering to address the challenges of producing secure software.
- Applying software design patterns to avoid tight coupling between different components of software, such as use of interfaces, the strategy pattern and the observer pattern.

Software Systems Implementation Project has technical and skills-related intended learning outcomes. After finishing this study unit, you should be capable of

- Implementing software of 10–20 classes in Java by using the core concepts of object-orientated programming.
- Defining and performing a test plan for software of 10–20 classes with appropriate test coverage.
- Implementing client-server programs using networked communication.
- Defining your role and responsibilities within a given project team.
- Managing time effectively, by applying tools and methodologies for project scheduling, and deciding and prioritizing personal tasks within the project.

Modes of Instruction

The following modes of instruction are used in this module:

Self-study	You study the material, which consists of your lecture notes of the programming and design lectures, the programming and design topics (short videos) on Canvas, 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.
Lectorials	The Lectorials are a mixture of lecture, tutorial, quiz and Q&A. You may get small exercises to do during the lectorial, to which answers will be provided. There may be a quiz. You can ask questions to the teachers. These sessions are recommended.
Practicals	You work **in pairs** on the practical exercises, supervised by teachers and teaching assistants. Some exercises are mandatory and will be signed off during the practicals.
Project supervised	You work on the project, and an instructor (teacher or teaching assistant) is present to answer your questions.
Workshop	These are activities where you may be expected to prepare some topics in advance and then dedicate the work time to a related activity so that you put into practice what you learned previously.
Diagnostic test	You perform exercises to assess how well you are acquainted with the material. The diagnostic test environment also prepares you for the real test. This happens in week 3.

The module schedule with the actual sessions (lectures, tutorials, etc.) can be found in https://www.timeedit.net/nl_utwente and on Canvas. In case these systems show contradictory information, the information found in https://www.timeedit.net/nl_utwente prevails.

Project Groups and Practical Groups

All exercises and the project are performed in pairs. You are free to form pairs yourself. Within the course, we have first-year students, minor students, premaster students, AM-TCS double-degree students, and resit students. Please only form pairs with others in the same group of students. For example, first-year students may only pair with other first-year students. **It is NOT allowed to change partners later, except in extraordinary circumstances such as a partner quitting the study.**

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 practicals.

Communication

For communication of information, we use several systems.

- On **Canvas** you can find official announcements, materials, slides, topic videos, files for the practicals, links to documents where you can ask questions to be answered in the lectorials, and your grades.
- We use **Horus** during the practicals, where you can find a record of your sign-offs, and where you can indicate to teaching assistants that you have a question or want to sign off exercises.

Help I have a Question

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

- The **teachers** are available during the Lectorials.
- The **teaching assistants** are available during the Practicals.
- 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.

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

What can I expect from Teaching Assistants?

During the practicals, teaching assistants will be available 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 practicals. Simultaneously, you can expect that they sometimes cannot answer every question. Some teaching assistants are a teaching assistant for the first time.
- Teaching assistants will expect that you have studied the materials.
- 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 check one sign-off point at the time. If you want to sign off more you have to rejoin the queue.
- Teaching assistants are not allowed to sign off exercises outside practical hours.
- Teaching assistants have no obligation to answer questions asked in a private chat.
- Asking questions on Canvas is expected, however you should not expect a response from a teaching assistant outside working hours.
- You can expect **queues** during practicals. 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 and regularly to the practicals and queue for signing off as soon as you finish a sign-off point. **Go to the early morning practicals!!**

What can I expect from my Partner?

During the module you will pair up with another student for the entire module. Together you will work on the exercises and the project. We strongly recommend that you pair up with the same students for all the study units of the module. Although it is theoretically possible to, for example, pair up with one student for Introduction to Programming, a second one for Advanced Programming, a third one for the Software Systems Implementation Project, and a fourth for Software Design and Modeling, that should not happen in practice. Why not? It would be confusing, as the units are not completely separated in time, it would make scheduling work together harder, as there will be more agendas to juggle, and it could result in conflicting expectations about results and commitment, since a student taking just one study unit may be willing to dedicate much more time to it than one who has to handle the entire module. Furthermore, for students taking the entire module, pairing up with the same student for all the study units highlights potential problems in the work dynamics within the pair early on, which leads to early solutions. This is why pairs must be formed within the same group, e.g., first-year students with first-year students. You can expect the following things from your partner:

- The ambition and motivation to pass Software Systems.
- Your partner should respond within 1 working day.
- Your partner should be available to work with you on the exercises during the practicals.
- In the event that your partner is ill or is unavailable for some reason, your partner should inform you as soon as possible.

It is important to realize that your partner can also expect these things 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:

- [`L0T`*x*]: introduction to Design

- [**L1T***x*]: introduction to software engineering, history, terminology, software (development) lifecycle, software modelling
- [**L2T***x*]: the use of models (UML and otherwise) in different phases of software development
- [**L3T***x*]: structural modelling
- [**L4T***x*]: behavioural modelling
- [**L5T***x*]: supplemental modelling
- [**L6T***x*]: software analytics

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

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

David J. Eck. *Introduction to Programming Using Java*. Version 9.0, May 2022. 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 project and exercises in Software Systems. It does *not* contain information about the Mathematics thread.

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;
- Mandatory exercises;

The project is described in a separate chapter.

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

It is important to study the book and the topic videos on Canvas, prior to doing the exercises. You can also study these materials during the practicals.

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).
- UTML (https://utml.utwente.nl/), which is a general tool for drawing UML diagrams. UTML is maintained by the University of Twente and is used as a web application.
- IntelliJML (https://gitlab.utwente.nl/fmt/intellijml/-/releases), which is a plugin for IntelliJ that supports syntax checking of software specifications written in JML.
- 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.
- GitLab (https://gitlab.utwente.nl), which is a version control system which is often used for large programming projects.

Consult Canvas for a guide on installing these tools on your computer. On the Monday of week 1 there is an installation session scheduled to install everything before you start with the practicals. You are expected to have a working installation of all tools by the end of this session.

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.

To successfully complete Software Design and Modeling, you have to:

- Participate in all mandatory activities and sign off all mandatory exercises:
  - All mandatory Design exercises must be signed off (**week 4**).
- Get a minimum of 5.5 for the Software Design and Modeling test

To successfully complete Introduction to Programming and Advanced Programming, you have to:

- Participate in all mandatory activities and sign off all mandatory exercises:
  - All mandatory Programming exercises must be signed off (**week 8**).
- Get a minimum of 5.5 for the test for each study unit (**week 7** for Introduction to Programming and **week 10** for Advanced Programming)

To successfully complete Software Systems Implementation Project, you have to:

- Successfully sign off all parts of the project: initial design (**Week 8**), code quality and testing, functionality, and reflection (**Week 10**).
- Participate in the Tournament (**week 10**).
- Pass all mandatory Skills assignments (**weeks 2 and 7**).

The **project is pass-fail** and does not have a grade in the 1-10 scale, differently from the other study units. The deadlines the Programming project can be found on the schedule. If a pair misses any of the project sign off deadlines, they can perform a late sign off on Friday morning, the last official day of the module. **There is no repair for the project**.

Mandatory participation If personal (severe) circumstances have made you incapable of attending one of the mandatory activities, you should contact the module coordinator to discuss the situation and agree on an alternative. This is only for severe personal circumstances, and not for "early holiday" or "I had to go to work" circumstances.

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 two Programming tests 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 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 will be reported to the Examination Board and 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, in the Chromebooks. You will **not** be able to use IntelliJ IDEA. Instead, you will write code as text (with syntax highlighting) using Remindo. Therefore, you should get used to not relying on autocompletion features when practicing for the test. For the Software Design and Modeling test, depending on the type of question you will have access to the tool UTML to help you draw your diagrams. Any updates to these rules will be announced on Canvas.

Signing off Mandatory Exercises

Many exercises in the practicals 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 or if you do not demonstrate the knowledge about the solution that would be expected from someone who has developed it, you will be asked to improve your design. If you come to the practicals 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 practicals. Pending exercises can be signed off at the beginning of a next practical, but do not spend the practicals working on exercises from past sessions. During each practical, you spend the first 15 minutes on a specific exercise. After these 15 minutes, a TA will explain (a part of) the solution with the whole room. After this moment you work further on your assignments with your partner and sign-off the other exercises.
There are only so many practicals. The deadline for Design exercises is in week 4.

Programming You do all exercises *in pairs*. There are explicit *sign off points* in the manual. When you reach these, you should ask a TA to sign off your exercises. In order to remain on schedule, all exercises for a week 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 practicals**.

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 practicals.
The Programming practicals 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 practicals. The deadline for Programming exercises is in week 8. This applies to both Introduction to Programming and Advanced Programming.

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

You are expected to sign off *all* sign-off points within the allocated time. If you fail to do so, you fail the course.

Not all exercises are mandatory, only the ones related to the sign-off points. However, for the great majority of students, we advise doing all exercises. You can ask teaching assistants to give feedback on your solutions.

Policy on Generative AI

Engaging and **struggling** with the exercises is crucial for learning. If you cheat on this by relying on AI-generated solutions, even if you just glance at the answer, then it will hinder your learning process. If you have not learned enough, you will fail the exams. During the exams, AI tools and even the weaker kind of autocompletion afforded by tools such as IntelliJ IDEA will **not be available**. Your ability to write programs should not depend on the help provided by such tools.

Bear in mind that sign offs for both exercises and projects will assess whether students understand what they are signing off. Generated solutions do not promote that kind of understanding. Furthermore, during the tests, no such capability is available. Students who produced solutions using generative AI tools are more likely to have bad results.

For similar reasons, avoid using generative AI tools to help in the project. During the project sign offs, you will be required to show that you know how it works, why it was built the way it was, and how the process that led to the construction of the system worked out, among other things. Relying on generative AI

enables students to (partially) bypass thinking about these issues without actually needing to learn anything, or at least learning significantly less than if they had to labor intensively in the development of the project. Again: **Engaging and struggling with the exercises is crucial for learning**.

When studying new topics, specially if you are a programming novice, use the recommended textbook, as its explanations are accurate and didactic. The only scenario for the use of generative AI tools that we consider somewhat acceptable, though still recommended against, is to ask specific questions, such as to clarify doubts and explore concepts or different approaches. However, exercise critical judgment, as some details may be incorrect. Do *not* use ChatGPT or similar tools to generate answers or debug code. Furthermore, most AI models, when asked coding questions, tend to produce complete code solutions even if explicitly told not to, which may tempt well-intentioned students. If you have a programming question, ask a TA or a lecturer.

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 an intermediate module evaluation sessions, *probably* in week 8 during the Tuesday lunch break. This is organised by CEEP. You will be invited to attend this session via Canvas.

Acknowledgements

Many people have contributed to this manual since its first version in 2013-2014. We acknowledge the valuable contribution of Kevin Alberts, Christoph Bockisch, Twan Coenraad, Frans van Dijk, Martijn Hoogesteger, Sophie Lathouwers, Renate van Luijk, Laurens Rouw, Jip Spel, Leon de Vries, Wim Kamerman, Remco Abraham, Rick de Vries, Andrei Popa, Silas de Graaf, Daniël Floor, Joris Kuiper, Alexander Stekelenburg, Brianna Dringa, Anissa Donkers, Irvine Verio, Valeria Veverita, Zhang Fay, Laurens van der Wal, Naum Tomov, Denis Asenov, Jelle Hulter, Kishan Thakurani. 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 practicals, 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

We commonly notice different working styles in the pairs for exercises and project work:

1. "Single-player mode", where one of the students usually performs the tasks, while the other student mostly watches.
2. Individual, where students work on exercises independently with minimal interaction.
3. Individual and help out, where students primarily work solo but assist each other when necessary.
4. Individual and discuss, where students operate separately then come together to review and discuss their results.
5. Traditional pair programming, where the driver concentrates on coding details, and the navigator focuses on the bigger picture and checking the driver's work for mistakes.
6. Strong-style pair programming, where the navigator instructs the driver, who attentively engages with the instructions, and with roles regularly alternating to promote balanced skill development.

Working together is fundamentally different from merely splitting tasks or working alone in the same room. In essence, working together means actively engaging with one another to solve problems and learn from each other's insights and viewpoints. It becomes counterproductive when a team (often unintentionally) shifts into a pattern where only one person actively participates while the others become passive observers. This often happens when students initially aim to do all the work together, but gradually regress into one person doing most of the work while the others watch.

Relying heavily on a teammate to navigate through the coding challenges can be tempting as it avoids personal struggle. However, this approach often means missing out on difficult, but necessary, personal learning experiences. In other words, if you are just watching someone else (or generative AI) doing the exercises, then you're not learning. It's common to see students fall even further behind in such scenarios.

Effective collaboration, on the other hand, with active participation from all members, promotes a more balanced and enriching learning environment. It helps prevent the formation of unproductive habits that often emerge in less interactive group dynamics.

With the exception of strong-style pair programming, all of the above styles run a risk of unequal participation, where one of the students takes a back seat in the learning process. While strong-style pair programming can seem difficult at first, it is a skill that is practiced by doing.

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. 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 engaged, thinking critically.

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 takes 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.

Motivation for 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 for a struggling student to watch the other student finish the exercises.
- It 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 practicals.
- Students are required to sign off in pairs.
- 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.

Week 1

Week 1

1.1 Overview

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

Design The activities in this week cover the following topics

- Level 1 Topics: Q&A session on [**L1T***x*].
- Level 2 Topics: Q&A session on [**L2T***x*].
- Practicals: S/P/E systems, their stakeholders and requirements, see Section 1.2.1.
- Practicals: Git

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 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.

1.1.2 Materials for This Week

Tool installation: See Canvas ("Course Materials").

Design:
Watch video [**L0T1**] to get acquainted with the course structure.
Watch videos [**L1T1**], [**L1T2**], [**L1T3**] to get started.
Watch videos [**L2T1**], [**L2T2**], [**L2T3**], [**L2T4**].

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

Practicals: The following files are provided on Canvas.

- `design/week1/use_case_template.utml`
- `src/ss/week1/BrokenFibonacci.java`
- `test/ss/week1/BrokenFibonacciTest.java`
- `test/ss/week1/FibonacciTest.java`

1.1.3 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 Practical exercises on modelling

All practical exercises, 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** Choose at least 4 systems from the following list:

- An automatic sudoku solver.
- An interactive sudoku playing app.
- A student calendar app with scheduled lectures.
- Untappd (a social network that tracks which beer you drank and shares it to your friends).
- Airbnb
- A PDF reader.
- An augmented reality set.
- A library system.
- Crowd size indication system for public transport.

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

From the Toucan hotel system -> Predictive model : effort estimation predict how much time will be needed to successfully design the system

1.2.2 Practical exercises on user stories

☞ **D-1.2** Read the following case description on "Inquiry Process at Event Horizon" and complete the provided UTML diagram `use_case_template.utml` from the practical files, by filling in missing use case elements, i.e. fill in the text within the use cases, actors and the extend/include relationships.

Case Description: Inquiry Process at Event Horizon At Event Horizon, events are separated into commercial- (e.g. music festivals) and non-commercial events (e.g. weddings, birthday parties). Through Event Horizon's inquiry system, clients are able to make detailed plans for their envisioned events. In order for a client to hold an event they need to access the Event Horizon website, after which they have to fill in an inquiry form. Depending on the event's nature (commercial or non-commercial), clients have to provide different information in the next step. For non-commercial clients, Event Horizon offers

different predetermined packages that a client has to pick from (packages are shown below). As for commercial clients, Event Horizon has to offer more flexibility in providing data, so a commercial client is required to fill in their preferences (e.g. event type, catering type, etc.) and upload several commercial & legal documents in no particular order. After a form has been completely filled in, and any additional information is provided, a client is able to send the inquiry to the administrator.

Once an administrator receives the form, they are required to process it. While processing the form, the inquiry can either be accepted or rejected. Upon rejection, the inquiry will be canceled, requiring the client to place a new one in order to reapply. In the case an inquiry gets accepted, an administrator will provide potential planning options that accommodate the client's wishes (e.g. venue, catering, etc.). After receiving a list with all potential options a client is able to choose their preference. In the event that none of the proposed options are preferable, the client will inform the administrator who will then suggest new options. After a satisfactory option has been selected, a client is required to make a deposit to Event Horizon. The deposit is 10% of the estimated cost of the event. Upon receiving the deposit, the administrator will confirm the payment and finally register the event. During registration an event manager is selected who will be responsible for executing the event.

☞ **D-1.3**

1. Create a use case diagram based on below *two* descriptions on scope and the role of Event Managers. First identify actors, then make a list of use cases, and finally integrate both actors and use cases in use case diagram(s).
2. Extend your previous diagram(s) to include the information from the case description on client interaction and payment.

Case Description: Platform scope The Event Horizon Platform (EHP) is at the base of everything that Event Horizon does. For years it enabled clients, event managers, and all other internal departments to create, coordinate, and manage events efficiently. As the company grew, the complexity brought by events and their demands for customization increased, especially in the case of commercial events.

Event Horizon uses EHP to simplify key aspects of event planning and to store important information, but the company's CEO acknowledges that maintaining flexibility is key. For this reason, he thinks that keeping all the communication and payment handling strictly on the platform would be quite cumbersome in a dynamic field like event management.

Case Description: The role of Event Managers Event managers are responsible for all aspects of event planning, ensuring that every event goes according to plan. They review client inquiries via the EHP platform, where they have an overview of all events together with their current state. There is usually a briefing every morning with all the staff in which important matters are discussed and new inquiries are assigned to event managers based on their experience and preferences.

Once an event manager reviews an inquiry, they usually contact the manager of the agreed-upon venue to confirm the date. This is done either by phone or email. Afterwards, depending on preferences and whether the event is private or commercial, the event manager has to contact various vendors through similar means.

The event managers have access to Event Horizon's database of vendors, which consists of a lengthy Excel spreadsheet that contains important information pertaining to all vendors, such as company name, telephone number, email, and the physical address of where they can be found. After a vendor sends confirmation, the event manager creates an "arrangement" for an event on the EHP. Arrangements are entries in the EHP that contain details about a vendor's services for a specific event.

After all of the arrangements are done, the event manager thoroughly reviews the information and prepares a final quote that is then submitted to the client for approval. The final quote consists of a summary of all the provided services, including the costs, detailed descriptions, and any additional remarks.

Case Description: Client interaction and payment After the event manager sends the final quote on EHP, the client has to review its contents. Once reviewed, they are presented with the option to either accept or decline the final quote. If a client disagrees with any of the proposed services or their cost, they are presented with the option to either suggest changes by providing feedback directly through EHP or cancel their inquiry. This will automatically trigger a refund request. Although event cancellations are rare, when they do occur, the event manager must handle them by carefully assessing the situation and refunding the deposit minus the planning costs up to that point.

If the client accepts the final quote, then the Finance Department is automatically notified on EHP. Once a quote is evaluated, an employee will contact the client via email and present them with different payment plans. After which the client can either pay for the event in whole or in installments up until the start of the event. Once the Finance Department receives the client's preference, they submit the invoice on EHP, together with a link that redirects the client to a payment platform that Event Horizon has partnered with.

1.2.3 Practical exercises on version control

This practical 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 the practicals 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 practical 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 practical 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-1.4 Go through the online GIT tutorial at `https://try.github.io`. This will acquaint you with some of the basic concepts and commands of GIT used on the command line.

D-1.5 What does GIT stand for?

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

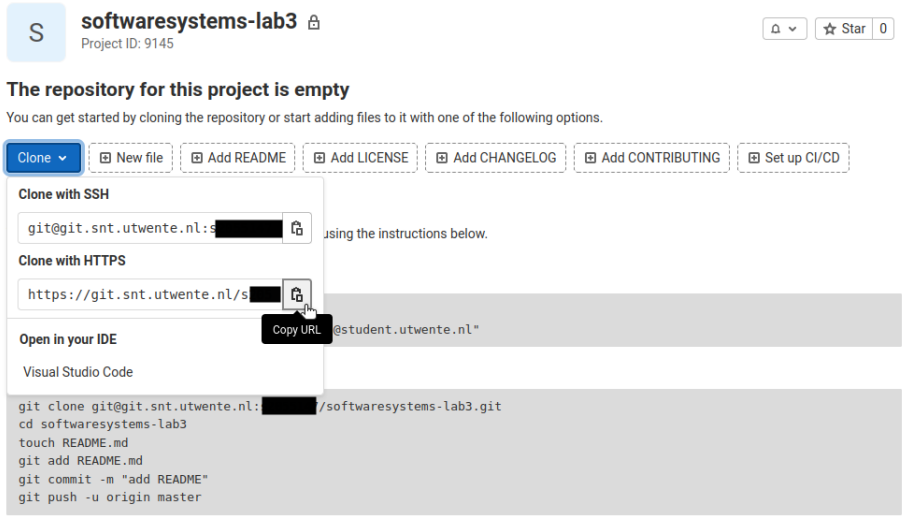
Central and local repositories At this point, it's time to really get going and create a repository of your own. This and the next exercises are spelled out on a quite fine-grained level. Do not just dumbly follow the steps: think actively on what is happening, and ask assistance if you feel that you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

D-1.6 (Your first repository)

1. 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.
2. Create a new (bare, initially empty) remote repository on that account.
   (a) Click NEW PROJECT or go to the new repository page directly: `https://gitlab.utwente.nl/projects/new`.
   (b) Select CREATE BLANK PROJECT
   (c) Give the repository a fitting name in the PROJECT NAME field, like `softwaresystems-lab3`.

(d) 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.
(e) Click CREATE PROJECT.

3. Create a new INTELLIJ project, create a local repository and link the local repository to the remote repository.
   (a) In INTELLIJ, create a new empty Java project and add a trivial, single class `Hello` in it.
   (b) Select `VCS → Enable Version Control Integration...`
   (c) Keep GIT selected and click on `OK` (In the toolbar `VCS` should now be replaced with `Git`)
   (d) Select `Git → Manage Remotes...`
   (e) Click on the +-symbol
   (f) 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 it should look something like `https://gitlab.utwente.nl/<username>/<repositoryname>.git`.[3]



   (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

[1] The origin is sometimes also called the *upstream* repository.
[2] Setting up and using an SSH key is somewhat out of the scope of this exercise. If you already have an SSH key setup and know how to use it, it's fine to use the SSH link as well.
[3] This is analogous to the `git remote add origin <url>` command.
[4] Checking the boxes next to files is analogous to running the `git add <filename>` command.
[5] Adding a message and clicking "Commit" is analogous to running the `git commit -m "<message>"` command.

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-1.9 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-1.7 (Rolling back)

1. Add a method to your `Hello` class (from Exercise D-1.6) and commit it. Note that you have to enter a *commit message* every time you do this.
2. Add another class to your project, and commit it as well. 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.

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-1.8 (Pushing and pulling)

1. Now it's time to see how Git handles multiple users, and how changes by multiple users are merged together.
   (a) On the web page of your repository on GitLab, add your partner as a collaborator to the project. Go to 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.
   (b) On your partner's laptop, clone the central repository: In the toolbar, select GIT → CLONE… or VSC → GET FROM VERSION CONTROL…
   (c) Paste the URL to the repository in the URL field. This is the same URL that you used in Exercise D-1.6 when you added the remote repository to your local repository.

   (d) 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.
   (e) Click on CLONE
2. Make some edits to one of the two local repositories you have now, commit them, and push them to the central repository.
3. 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.
4. 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?
5. If the push failed in the last step, try to pull the repository first, and then push again. What happens?
6. Make sure both repositories are up-to-date with the upstream by pulling both of them.
7. 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.
8. 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-1.9** (Branching and merging)

1. 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.
2. Create a new class in your project, and add a method to another class. Commit and push. These edits are now part of the new branch, but are *not* visible in the master branch.
3. Switch to the `master` branch (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 exercise 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.

1.3 Introduction to Programming

1.3.1 Practical exercises

Hello World

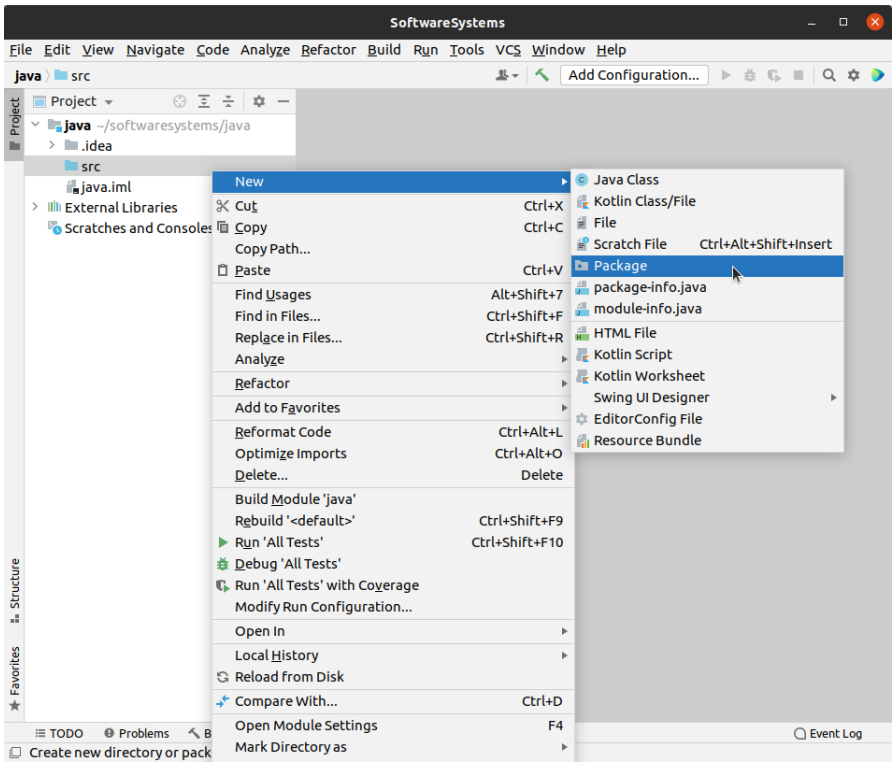
The following exercises are intended to familiarise you with the process of compiling and running a JAVA program.

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.

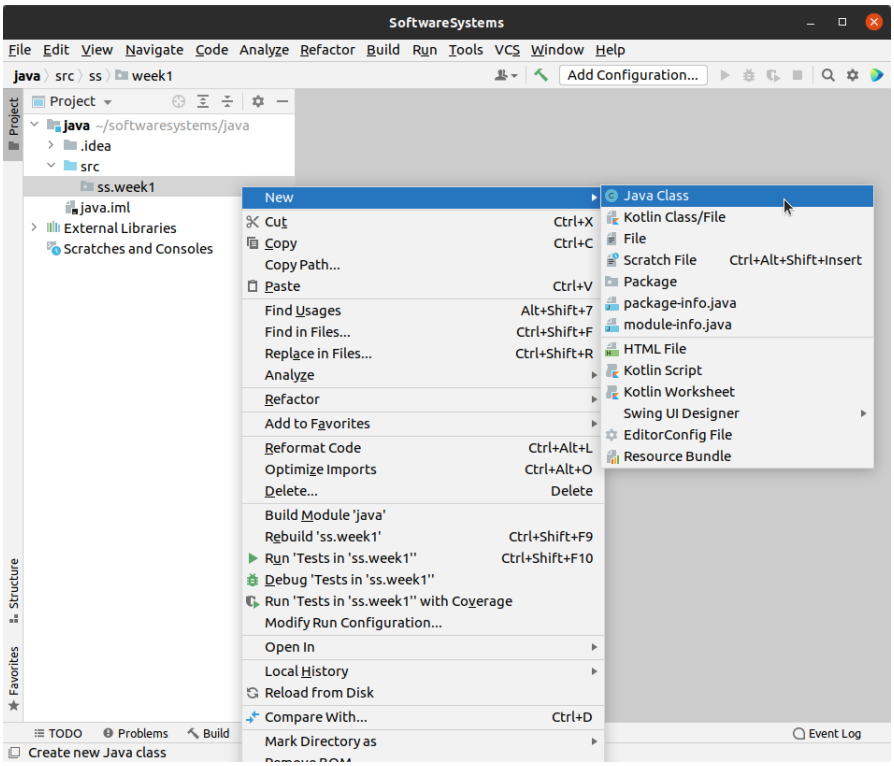
We start with the simplest program imaginable, which does nothing but print a welcome message on your screen.

IP-1.1 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`



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

- You can compile the class by going to the Build menu and choosing the option "Recompile Hello.java"
- You now see in your Project view (the left side of the IntelliJ window) that a new directory "out" has been created. If you open this directory and its subdirectories, you will find a file called "Hello.class".
- Open your Files Explorer and browse to the root directory of your project.
- Look inside the "src/ss/week1" subdirectory and notice the presence of the file "Hello.java".
- Now look inside the "out" directory and find the "Hello.class" file. Open it in a text editor like Notepad.
  You will see a lot of unreadable characters and some text that you might recognize. This is called **bytecode** and it is suitable for a computer to process. Bytecode is the result of the step called **compilation**. You can in fact compile a file manually without a development environment like IntelliJ if you want, but we will not do that now.
  Normally, when you change your source code, you would just click the Build icon (the hammer) or use the Build Project option from the Build menu to compile your entire project. This often happens automatically when you choose to run a program of your project (see below).
- 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.

IP-1.2 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.

Study This!

Make sure you have studied Chapters 1 and 2 of Eck, as well as the related topics on Canvas. Although Eck discusses a `TextIO` class, we will not use this class and instead will use a `Scanner`. To learn more about the `Scanner`, read Section 2.4.6 of Eck, and https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Scanner.html.

IP-1.3 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.

IP-1.4 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.

IP-1.5 (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*144 + 3*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.

IP-1.6 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.

IP-1.7 The area of a regular polygon is

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

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
```

IP-1.8 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. Don't forget to divide percentages by 100 to obtain the corresponding fraction.

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`**.

Study This!

Make sure you have studied Chapter 3 (except 3.9) of ECK, as well as the related topics on Canvas.

IP-1.9 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:

- Up to 35472 euro: 9.42%
- Between 35472 and up to 69398 euro: 37.07%
- Above 69398 euro: 49.5%

This translates to the following rules:

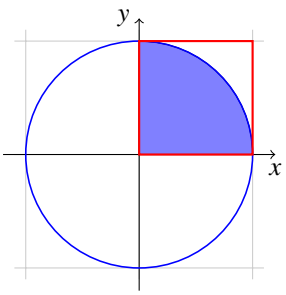
- If your income I is at most 35472, then the tax is $0.0942 \times I$.
- If your income I is more than 35472 and at most 69398, then the tax is $3341 + 0.3707 \times (I - 35472)$.
- If your income I is more than 69398, then the tax is $15917 + 0.4950 \times (I - 69398)$.

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
```

IP-1.10 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.

IP-1.11 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
```

Study This!

Make sure you have studied Chapter 4 of ECK, as well as the related topics on Canvas, including Documentation.

Procedures

In the following exercises, you will be asked to create **methods**.

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 therefore not **`static`**), and not isolated subroutines (which don't use any object state, and are therefore **`static`**). The topic of objects will come next week, and from next week on, you will most of the time make methods that are not **`static`**!

Javadoc

When creating methods (and, starting next week, classes), you are required to *document* your methods and classes with Javadoc. Javadoc for a class is placed immediately before the **`class`** declaration; 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.

In the rest of the course, you are **required** to document all methods and classes with Javadoc.

IP-1.12 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 files from Canvas

Files for the practical exercises, 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.

IP-1.13 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.

IP-1.14 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.

IP-1.15 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.

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

Example run:

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

Arrays

Study This!

Make sure you have studied Chapter 3.8 of ECK about arrays.

In the next exercises you will work with arrays in Java. Arrays in Java are fixed-size containers that hold multiple values of the same type. They are often used to store and process collections of numbers, measurements, or other data.

To create a new array, you can use a **new expression**. For example, to create an array of 10 integers:

```
int[] data = new int[10];
```

This creates an array filled with zeroes, indexed from `0` to `9`. You can assign and read values as follows:

```
data[0] = 42;
System.out.println(data[0]);
```

Alternatively, if you already know the values, you can use an array literal to initialize a variable:

```
int[] values = {4, 7, 2, 9, 5};
```

To process all elements of an array, use a **for**-loop or **while**-loop with the value of `values.length`:

```
for (int i=0; i<values.length; i++) {
    System.out.println(values[i]);
}
```

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 larger array, and copy all the information from the original array to the new array. Java provides an efficient copy method for this:
`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`.

IP-1.16 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 **int**s, first "split" the `String` into fragments, then convert each fragment into **int**s (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
```

IP-1.17 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 97 107 113 149 157 167 179 199 311 337 347 359 389
```

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**.

Sign off point IP-1.A

This sign off point will be about all preceding exercises and in particular about exercises IP-1.14, IP-1.15, IP-1.16 and IP-1.17. You can be asked to explain your approach to these exercises to the TA and they can give you feedback. The TA can also ask about earlier questions and can expect that you have done all exercises and studied the materials. Make sure you can also show the following:

- Your code for exercise IP-1.14
- Your code for exercise IP-1.15
- Your code for exercise IP-1.16
- Your code for exercise IP-1.17

The following exercises are meant to further exercise your ability to work with arrays. In your solutions, create a method that implements the asked computation, and use your `main` method to display the results for some test data, such as the given example input data or any input you come up with. Alternatively, you could make JUNIT tests similar to the `FibonacciTest`.

IP-1.18 Imagine that you are processing data from a speed camera. Given an array of measured speeds (in km/h), compute the total amount of fines. Assume the speed limit is 80 km/h, and exceeding it costs:

- €50 for at most 10 km/h over,
- €100 for at most 20 km/h over,
- €200 for more than 20 km/h over.

Print the total fines for all given speeds. Here's an example array of input data:

```
int[] speeds = {
        78, 85, 93, 102, 81, 77, 110, 120, 99, 83, 75, 88,
        97, 115, 100, 79, 80, 90, 104, 112, 70, 82, 86, 130, 84
};
```

IP-1.19 Given an array of daily temperatures, determine the length of the longest increasing streak. For example, given the array `{ 13, 14, 15, 12, 13, 14, 15, 16, 10, 11, 12 }`, the longest increasing streak has length 5. Here's another example array of input data, with the longest streak having length 7:

```
int[] temps = {
        12, 13, 14, 15, 13, 14, 15, 16, 17, 12, 13, 14, 10, 11, 12, 13,
        12, 13, 14, 15, 16, 17, 18, 12, 11, 10, 9, 8, 7, 8, 9, 10, 11, 12
};
```

IP-1.20 Given an array of integers, return a new array containing only the unique values, in the order of their first appearance. For example, given `{ 4, 7, 4, 3, 7, 9, 3, 2, 9, 1, 2 }`, the result should be `4, 7, 3, 9, 2, 1 }`. Here's another example array of input data:

```
int[] values = {
        4, 7, 4, 3, 7, 9, 3, 2, 9, 1, 2, 11,
        14, 10, 7, 12, 4, 13, 2, 1, 13, 14, 15, 15, 9
};
```

Sign off point IP-1.B

This sign off point will be about the preceding exercises on arrays. Make sure you have done all exercises.

Recursion

Recursion happens when a method calls itself to solve a smaller version of the same problem. You have already seen one example of a recursive function: the implementation of Fibonacci in exercise IP-1.13 A recursive method usually has:

- a **base case**: a simple situation that can be solved directly, and
- a **recursive step**: a call to itself with a smaller or simpler input.

When designing a recursive solution, always ask:

- "What is the smallest case I can solve directly?"
- "How can I make the problem slightly smaller and call myself again?"

IP-1.21 Write a recursive method that, given a positive integer, returns the highest digit in that number. For example, for `48271` the result is `8`. Think of the base case (a single-digit number) and how to make the problem smaller (removing the last digit).

IP-1.22 Write a recursive method that checks whether an array of integers is a palindrome. For example:

```
{1, 2, 3, 2, 1} == true
{5, 4, 4, 5}    == true
{1, 2, 3, 4, 5} == false
```

Your method should take parameters for the array and two indices (the start and end position currently being compared).

IP-1.23 Given a **sorted** array of integers, implement a **recursive** binary search method that returns the index of a given target value, or `-1` if the value is not found.

Your method should take parameters for the array, the target value, and the current start and end indices.

For example, given the array

```
int[] data = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
```

the search results should be:

```
search(data, 10, 0, data.length - 1) == 4
search(data, 2, 0, data.length - 1)  == 0
search(data, 15, 0, data.length - 1) == -1
```

Think carefully about the base case (when the range is empty) and how to make the range smaller at each recursive call.

Sign off point IP-1.C

This sign off point will be about the preceding exercises on recursion. Make sure you have done all exercises.

1.3.2 Recommended exercises

IP-1.24 Make the following exercises from ECK:

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

Week 2

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 [**L3T***x*].
- Level 4 Topics: Q&A session on [**L4T***x*].
- Practicals: class and sequence diagrams, see Section 2.2.1.

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

2.1.3 Materials for this Week

Design

Watch videos [**L3T1**], [**L3T2**], [**L3T3**].
Watch videos [**L4T1**], [**L4T2**], [**L4T3**].

Programming

Materials ECK, Chapter 5.1–5.4, the week 2 topic videos
Practicals The following predefined files are provided on Canvas:

2.2 Design

2.2.1 Practical exercises on class diagrams

☞ **D-2.1** Complete the provided class diagram, see this week's file `class_diagram_template.utml`, by filling in the missing multiplicities, attributes, and the missing association names. Base your answer on the case description given below.

Case Description: Event Horizon Platform Event Horizon is dedicated to customer satisfaction, building their business around a customer-first approach that addresses all their clients' event planning needs. Through their platform, customers can easily book events by providing basic information, selecting an event type, specifying their preferred date, and optionally indicating their own venue choice. If no venue is specified, the event managers will handle this aspect of the planning. Event Horizon values customer loyalty, offering special reduced pricing for returning customers who have booked more than two events with them, as a way to recognize and reward their continued trust in Event Horizon.

When a booking request is received and approved by the administration team, a dedicated event manager to oversee all aspects of the event planning process is assigned. To maintain their high standards of service quality, each event manager is limited to coordinating no more than three events simultaneously. The system maintains detailed records of all events that each manager has coordinated throughout their tenure with Event Horizon.

The company maintains a comprehensive database of service types tailored to different event categories. Event managers select appropriate services and negotiate with vendors over email or phone to secure the best arrangements for each event. Once terms are agreed upon, the system records the service delivery schedule and any price variations from the standard rates. These services encompass everything from catering and venue arrangements to music and photography, depending on the event's specific requirements. To ensure clear accountability and avoid conflicts, each service for an event is assigned to a single vendor, though multiple vendors may work on the same event providing different services. This system ensures that no two vendors provide identical services for the same event, maintaining organization and efficiency in Event Horizon's service delivery.

☞ **D-2.2**

1. Create a class diagram based on below description for commercial non-commercial event management.
2. Extend your previous diagram to include the information from the case description on event promotion and marketing.

Case Description: Commercial and non-commercial events Events at Event Horizon fall into two categories: non-commercial and commercial. Commercial events are those open to public attendance, typically through ticket purchases. These events are organized into sessions, with ticket availability carefully managed to remain below the venue's capacity. Tickets can be purchased either through Event Horizon's online platform or at on-site ticket stands. In line with their privacy policy, Event Horizon Platform does not collect personal data during ticket purchases. However, they do provide a platform where ticket holders can enter their ticket numbers to rate their attended sessions and provide feedback. This feedback system helps Event Horizon continuously improve their services and, when necessary, process reimbursements for valid complaints.

Case Description: Event promotion and marketing To ensure effective event promotion, each event requires strategic marketing efforts. Event managers assign one dedicated marketing manager per event, who is responsible for creating and implementing appropriate promotional campaigns. Upon assignment, the marketing manager develops a comprehensive resource plan, focusing primarily on the budget needed to effectively reach the target audience interested in the event.

The process begins with the marketing manager submitting a funding request that outlines three key elements: the proposed campaign type, expected outcomes, and required budget. This request undergoes review by the event manager, who either approves it or rejects it based on budget constraints and financial feasibility. While each event is limited to one marketing campaign for simplified management, marketing

managers have the flexibility to create multiple advertisements within that campaign once the budget is approved. This approach ensures focused, efficient marketing efforts while maintaining budget control and clear accountability.

☞ **D-2.3** Now that your class diagram for the Event Horizon Platform (EHP) is complete, evaluate the **internal design quality** of your model by reflecting on two key object-oriented design principles: *cohesion* and *coupling*. High cohesion and low coupling make a design easier to maintain, extend, and reuse.

1. **Cohesion:** Review three key classes (e.g., `EventManager`, `Service`, `MarketingCampaign`) and explain whether each has a single, focused responsibility or mixes several. If not cohesive, suggest one improvement.
2. **Coupling:** Identify two or three class pairs (e.g., `Event–Service`, `Service–Vendor`) that are strongly connected. Discuss if this coupling is necessary and (if not) suggest one way to reduce it.

2.2.2 Practical exercises on sequence diagrams

☞ **D-2.4** Use below case description on adding an advertisement to complete the provided sequence diagram, see file `sequence_diagram_template.utml`. Fill in the missing object names, messages, and operator types within the combined fragments.

Case Description: Advertisement Addition A Campaign Manager is working to add a new advertisement to one of the campaigns under their care. The first step is to get acquainted with the event they are working with, starting by learning the event details. Once that's done, the manager retrieves the campaign associated with the given event. After retrieving the relevant campaign to focus on, the manager turns their attention to the advertisements already part of it, taking a closer look at what's currently running. With everything reviewed, the manager works on their ideas and creates a brand-new advertisement, subsequently adding it to the chosen campaign.

☞ **D-2.5** Create your own sequence diagram from the Event Planning case description given below. Take the following steps:

1. Identify the relevant actor(s) and objects. Check that all are required for the sequence diagram to be functionally correct. Write down all actors and objects in a table.
2. Identify the relevant messages sent between actor(s) and objects. Check that all are required for the sequence diagram to be functionally correct. Write down the messages as a table where for each message you denote the message label, sender, and receiver.
3. Finally, utilizing the actor(s), objects, and messages you identified create a sequence diagram. Determine the correct type for your messages: synchronous, asynchronous, or return. Use sequence diagram elements such as loops, alts, and opts where applicable.

Case Description: Event Planning Event Horizon's event planning process begins when an Event Manager initiates the planning phase in the system. For each event, multiple services need to be arranged, such as catering, venue, and entertainment. After an event manager has retrieved the event information, the Event Manager starts by searching for appropriate vendors through the system for each required service. The system returns a list of potential vendors who can provide the specific service.

The Event Manager then enters into a negotiation cycle with vendors. For each vendor, the Event Manager sends a service request detailing the requirements. The vendor responds by providing a quote for their services. If the quote falls within the allocated budget, the Event Manager adds the service into the system, confirms the service availability for the event, and sends a formal confirmation to the vendor. However, if the quote exceeds the budget, the Event Manager requests the vendor to revise their quote. The vendor may then send a revised quote, and this negotiation continues until either an acceptable quote is received or the vendor is ruled out. This vendor selection process repeats for each service required for the event.

Once all services are arranged, the Event Manager sends the complete event plan to the customer for review. The customer has the option to request modifications to the plan. If the customer requests any changes, the Event Manager updates the event plan in the system accordingly. This ensures that the final event plan aligns perfectly with the customer's expectations.

2.2.3 Recommended exercises on class diagrams

☞ **D-2.6** 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 Aquitane 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-2.7** Extend the class diagram of D-2.6 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).

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

D-2.8 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 tim 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-2.9 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.

2.2.4 Recommended exercises on sequence diagrams

☞ **D-2.10** 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 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-2.11) 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.

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.

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.

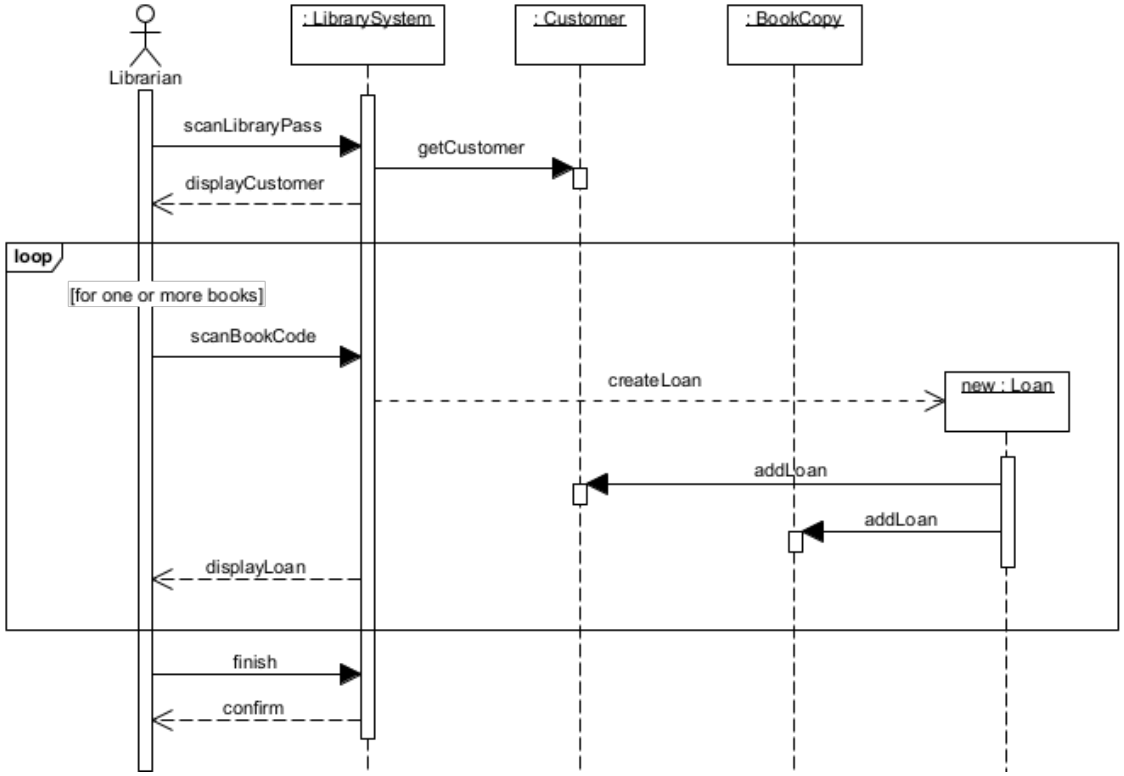


Figure 2.1: Lending library books (Exercise D-2.10)

Changing a service plan
When a nurse starts this function, they first choose the client for whom the service plan is to be changed. A service plan states all the information of a patient and their prescriptions. A prescription is the information of one specific medicine, it contains information about the dosage and at which time the patient should take it. When this client has been chosen, their service plan is deactivated (only if it was active at the time). Subsequently, the following changes can be made;

- change the number of times per day and/or the dose of a particular medicine;
- indicate that a medicine is no longer taken by the client;
- indicate that a new medicine has been prescribed to the client;
- change the times at which a client is visited for medication.

Elderly people often use multiple medicines for multiple illnesses and conditions, so each of these changes can be applied more than once when the service plan is adapted. There is no prescribed order, a nurse can do the steps in any order they like.
Finally, the service is activated.

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*.

☞ **D-2.11** Implement for *Change services* the reference box shown in the top-level design in Figure 2.4, which shows that the nurse already retrieved the relevant patient information from the system. From this point onwards, the nurse will only interact with the service plan.

D-2.12 Make a sequence diagram for booking a tour with Outdoor Holiday Tours (see Section 2.2.1 for the case description). To limit the amount of drawing, you don't have to include a control object (and, as a consequence, you don't have to draw return messages).

Making a booking involves the following steps:

- Selecting a tour;

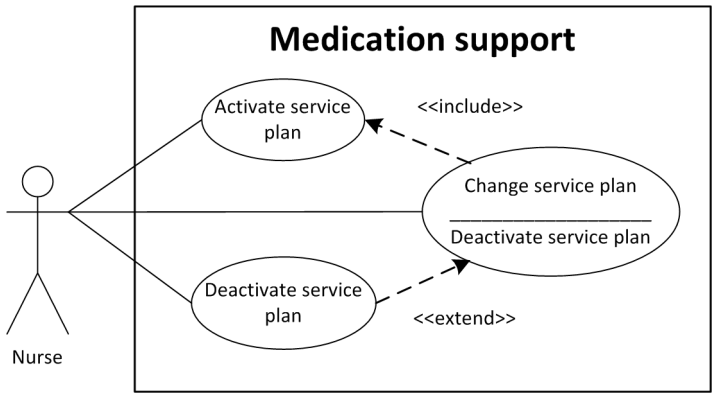


Figure 2.2: Partial Use Case Diagram for medication support (Exercise D-2.11)

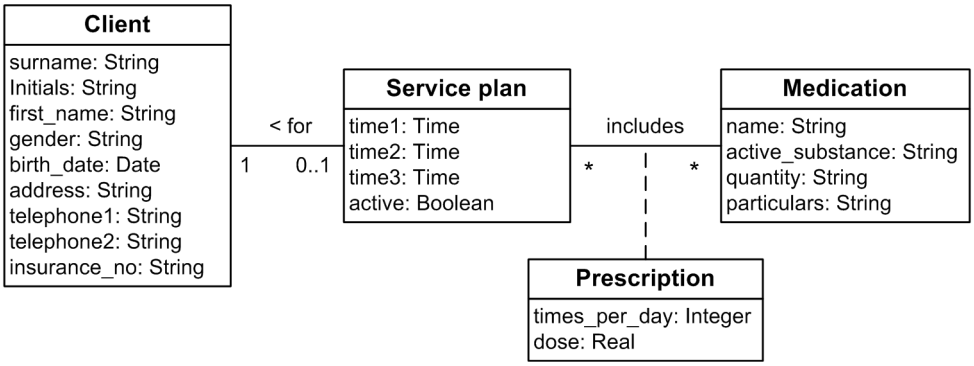


Figure 2.3: Partial Class Diagram for medication support (Exercise D-2.11)

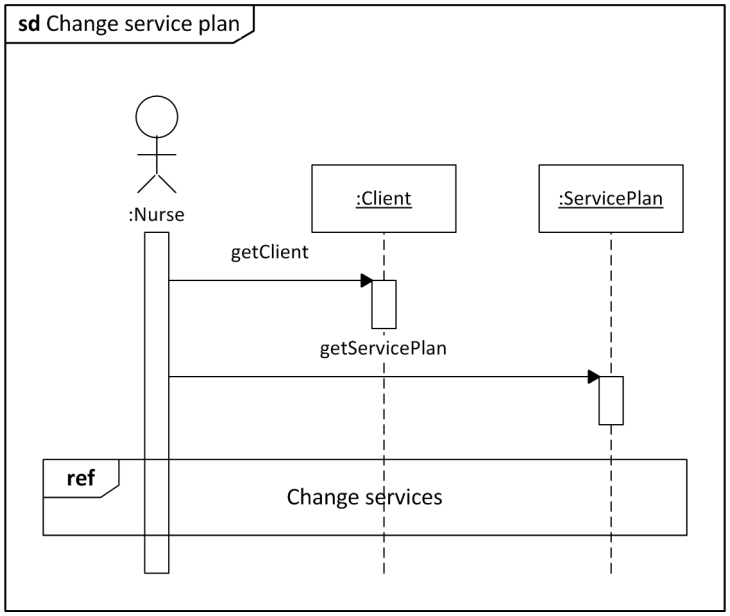


Figure 2.4: Sequence Diagram for *Change serviceplan* (without control object) (Exercise D-2.11)

- 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-2.13 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*).

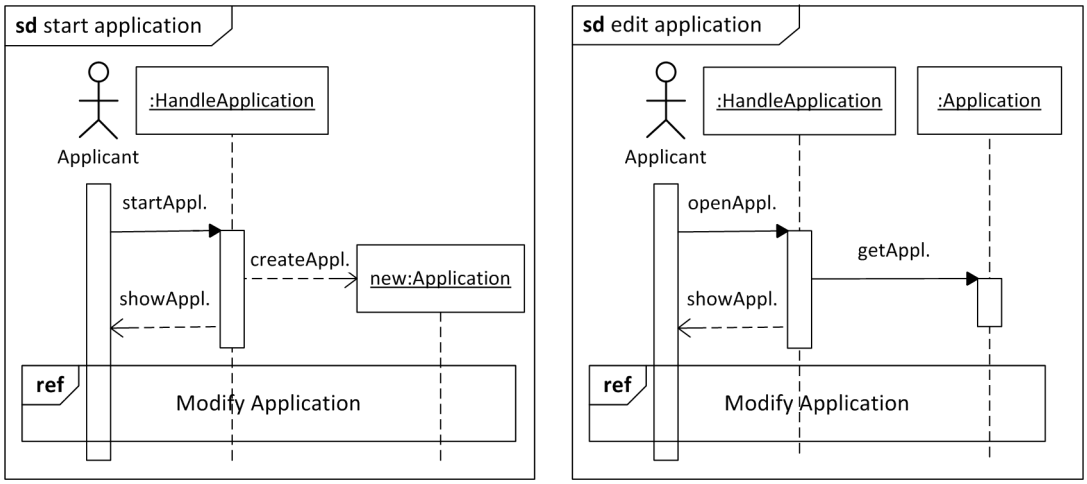


Figure 2.5: Creating/editing an application for university admission (Exercise D-2.13)

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 Introduction to Programming

2.3.1 Practical exercises

Import the contents of this week's files from Canvas to your IntelliJ project by following the instructions mentioned on page 26.

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 practicals this week. After completing the assignment for class `DollarsAndCentsCounter` in Exercise IP-2.1, for instance, the compilation errors in `DollarsAndCentsCounterTest` should be gone.

Objects and Classes

Study This!

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

This week, we start applying the *object-oriented programming paradigm* to our programs. Object-oriented programming allows us to *structure* a program into smaller, more manageable components. Using *abstraction* and *encapsulation*, we can focus on individual parts of the software without needing to understand every technical detail of other parts.

Ideally, our programs are designed to minimize the *ripple effect* of changes, ensuring that modifications in one part of the system have minimal impact on others. This is achieved through encapsulation: each class hides its internal details while providing a set of public methods that other classes can interact with, allowing components to communicate without exposing unnecessary complexity.

There are several ways to structure a program. One is to organize classes based on data—for example, classes that represent entities such as a *Customer*, *Product*, or different *Animals* in a zoo, or *Rooms* in a hotel with *Guests*. Another approach is to structure classes based on functionality, where classes represent specific behaviors or actions, such as a *PaymentProcessor* or *DataValidator*.

The *single responsibility principle* is essential in both approaches: each class should focus on a single task or responsibility, making the software easier to maintain, debug, and extend. In a good design, classes exhibit *high cohesion*, meaning that all the methods and attributes within a class are closely related to its core purpose. Additionally, the system should strive for *low coupling*, where classes are as independent as possible, reducing dependencies between them and making the program more flexible and easier to modify without introducing unintended side effects.

In later weeks, we will explore additional aspects of object-oriented programming, such as *inheritance* and *polymorphism*, which enable the creation of more flexible, reusable code.

IP-2.1 In this exercise, you implement and test a class that counts an amount of money in a wallet 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.

For example, if you call the `reset()` method followed by `add(1, 105)`, the method `getDollars()` should return 2 and the method `getCents()` should return 5.

Pay special attention to the Javadoc specification of the `ss.week2.DollarsAndCentsCounter` class. In the imported files of this week 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.

Command Query Separation

Command Query Separation (CQS) is a design principle that distinguishes between methods that perform actions (commands) and those that retrieve data (queries). Commands change the state of an object and (usually) do not return a value, while queries return data but do not modify the object's state. This separation promotes clearer, more predictable code by ensuring that methods have a single responsibility—either to perform an action or to provide information, but not both.

In particular, software developers typically expect that they can use query methods without altering the state of an object. This expectation ensures that queries are safe to call multiple times without side effects, making the code more reliable and easier to reason about.

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*.

Study This!

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`

IP-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?

IP-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. You also need to make the enumerated type `ThreeWayLamp.LampSetting` (*inside* the `ThreeWayLamp` class) otherwise you have no way of representing the current or desired setting.

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.

IP-2.4 Write unit tests for the `ThreeWayLamp` class based on the documented stub you defined in Exercise IP-2.3. Create a class `ThreeWayLampTest` in package `ss.week2` under the *test sources root* 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.

IP-2.5 Implement the methods that you specified in Exercise IP-2.3 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 IP-2.4.

Text interface for the lamp 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`.

IP-2.6 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.

IP-2.7 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;
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
```

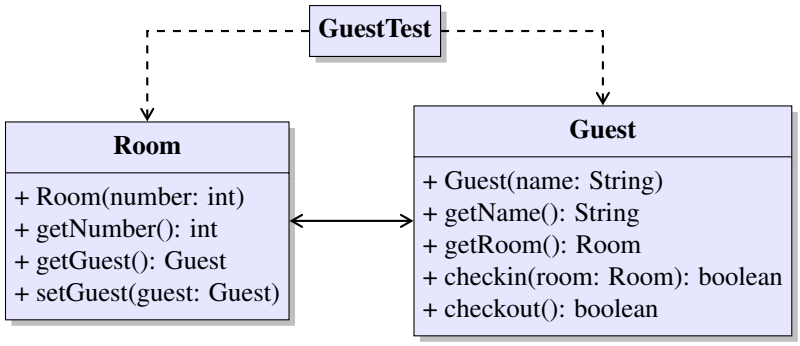


Figure 2.6: Class Diagram for Hotel Application

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

Study This!

Study 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?

Sign off point IP-2.A

To sign off, present the following:

- Your code for all preceding exercises.
- Your `ThreeWayLampTUI` should work perfectly.

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.

IP-2.8 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.

IP-2.9 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.

IP-2.10 Add Javadoc 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.

IP-2.11 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.

IP-2.12 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.

IP-2.13 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()`.

Study This!

Study ECK Section 5.3.2 to learn about the `Object` class.

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.

IP-2.14 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.

IP-2.15 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`.

IP-2.16 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 `hotel.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.

IP-2.17 Specify (with Javadoc) 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.

Also extend the class diagram of Exercise IP-2.8 (Figure 2.6) with the new classes; the class diagram should show all classes in the package `ss.hotel`. Feel free to draw the diagram on paper if you prefer that.

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

IP-2.18 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.

IP-2.19 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 **`if`**s you are asked to use the **`switch`** statement to check the input values and act accordingly.

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

Study This!

Study ECK Section 4.8.3 to learn 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.

Sign off point IP-2.B

To sign off, present the following:

- Your code for exercises IP-2.8 to IP-2.19, especially IP-2.18 (Implementation + Debugging) and IP-2.19 (Full working TUI)
- You need to do all hotel-related questions for this sign off point.

2.3.2 Recommended exercises

IP-2.20 Make the following exercises from ECK:

- Exercises 5.1, 5.2 and 5.3.

Week 3

Week 3

3.1 Overview

3.1.1 Contents of This Week

Design The activities in this week cover the following topics:

- Level 5 Topics: Q&A session on [**L5T***x*].
- Level 6 Topics: Q&A session on [**L6T***x*].
- Practicals: activity and state machine diagrams, and cyclometic complexity see Section 3.2.1.

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; 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**].

Programming

Lectures ECK, Sections 5.5–5.8, the week 3 topic videos
Practicals The following files are provided on Canvas:

3.2 Design

3.2.1 Practical exercises on activity diagrams

☞ **D-3.1** Reread the case description on "Inquiry Process at Event Horizon" from week 1 (lab on Use Case Diagrams) and complete the activity diagram of provided file `activity_diagram_tenplate.utml`, by filling in missing activities.

☞ **D-3.2**

1. Create an activity diagram based on the provided description for non-commercial event management.
2. Make a copy of your previous activity diagram, and extend it to cover the case description on commercial event management processes.
3. Make a copy of your previous activity diagram, and further extend it to cover the case description on the payment management process.

Case Description: Non-commercial Events At Event Horizon, after an event has been registered during the inquiry process, the event planning process will start. To properly plan an event, an Event Manager will need to review the inquiry and additionally reserve the selected venue for a particular timeslot in EHP. After reserving a venue, the Event Manager is responsible for creating a possible plan which is then suggested to the client on the platform.

For non-commercial clients, Event Horizon uses a package system. Where each increase in tier (Bronze $\rightarrow$ Silver $\rightarrow$ Gold) will result in more services being arranged. The plan for an event is chosen by the client in the event inquiry phase, which is before the event planning takes place.

In case packages 2 or 3 (Silver or Gold) are selected, the event manager will need to arrange additional services, namely quoting the price for catering and/or scouting a photographer. In case a client has chosen the Gold Package, the company's protocol states that the event manager will only start scouting for a photographer after finishing with all of the catering arrangements. This step-by-step process is done to ensure that the event manager can come up with a clear timeline for the event that can be presented to the potential photographer(s). After all services have been arranged, a final quote is calculated and sent to the client on the EHP, after which the client is also notified by the event manager via other means of communication (email, phone, etc.). Upon reviewing the arrangements, a client can either choose to accept or reject the arrangements. If accepted, the client will send a confirmation, otherwise they will reject the proposal.

Case Description: Commercial Events Event Horizon also provides services for commercial events, for which some more variety and flexibility is added in the inquiry and planning process in comparison to the non-commercial events. In the case of a commercial event, the event manager addresses all of the customer's requests simultaneously once the venue has been reserved since these requests are usually more complex and take longer to conclude. The three main areas of services that Event Horizon provides for commercial events through partners are commercial catering, renting and preparing technical equipment, and providing various entertainment services. For each commercial event, the event manager usually has to produce a quote for catering services, propose technical equipment and organize potential entertainment services. Depending on the desired services from the client, some or all of these arrangements can be skipped. After all services have been planned out, the Event Manager will calculate and add the final quote in EHP, and then notify the client.

After reviewing arrangements, a client has the options to either accept or reject them. Once a client has accepted they will send a confirmation. In the event a client rejects the arrangements, they can either reject the plans permanently and potentially receive a refund, or ask for another quote after sending some suggestions for improvement in EHP. Once the event manager has reviewed the feedback, they will reiterate through the whole process once again and send the customer a new quote. **Note**: The refund process is not part of the Event Management process, so it is not necessary to add information about it.

Case Description: Payment Process At Event Horizon, after a client has accepted the final quote sent by the event manager, the Financial Department will evaluate the quoted cost of the event. After the evaluation, the Financial Department will compile a set of payment plans and send them to the client on the EHP. The client then has to select a payment plan, after which the Financial Department will send out an invoice. Upon receiving the invoice, a client will need to pay the invoice. In the event there is still an unpaid installment, the Finance Department will wait one month before sending a new invoice. After all installments have been paid, the payment is automatically marked as completed. **Note**: The payment plans differ in the number of installments necessary to cover the full payment for the services provided by Event Horizon. The option for a single full payment is always available, but the clients might choose the option to pay for the services in multiple installments (2, 3, 4, 6, etc.) based on the date of the event and other financial aspects.

3.2.2 Practical exercises on state machine diagrams & Cyclomatic Complexity

☞ **D-3.3** Use below case description on event management to complete the provided state machine diagram `state_machine_template.utml`. Fill in the missing states, transitions, and guards.

Case Description: Event Management At Event Horizon, managing event services through the Event Horizon Platform (EHP) follows a structured and efficient process. The event manager initiates the service arrangement phase by selecting the type of service required for the event, such as catering, entertainment, or equipment rental. Once the service type is determined, the event manager proceeds to identify and select potential vendors. After identifying suitable vendors, quotes are requested and received. These quotes are then thoroughly reviewed by the event manager to ensure they meet the client's specifications and align with the event's estimated budget. If a quote falls within the budgetary limits, it is accepted; if not, the event manager either rejects the quote or requests a revision from the vendor. Once the quote is confirmed, the service is finalized, and the process continues when all required event services have been confirmed.

The budget process begins with the event manager providing an initial cost estimate, which is based on their professional experience and an understanding of the event's requirements. This preliminary budget estimate is then submitted to the financial department for review and approval.

While the budget is under review, the event manager is not required to wait for final approval to begin making arrangements with vendors. As soon as the budget estimate is submitted for approval, the event manager is free to initiate negotiations with vendors. However, the client is not notified of the finalized event plan until all services have been confirmed and the budget has been formally approved.

☞ **D-3.4** Now consider previous case description on event management, and the two (!) below case descriptions on event preparation and the full event life cycle.

1. How should the state machine diagram that includes all three case descriptions look like? Make a quick sketch of the main structure of the state diagram, and indicate where the previous diagram should logically be placed.
2. Extend your diagram of exercise 1 with the details from the event preparation case description.
3. Complete your diagram of 2 with the case description on the full event cycle.

Case Description: Event Preparation The process of event preparation begins when an event manager self-assigns a specific event. At this point, the event manager conducts an assessment of the event's requirements, which includes identifying the necessary services and estimating the associated costs. Based on this assessment, an initial budget is drafted, which serves as a preliminary financial outline for the event. Concurrently, the event manager initiates arrangements for the required services, such as catering, entertainment, or equipment rental, while awaiting approval of the budget estimation from the financial department. Once the budget is approved and the service arrangements are made, the event manager compiles the event plan and submits it to the client for review. The client then has the option to either confirm the event plan or request modifications. If modifications are requested, the event manager reassesses the event requirements and revises the plan accordingly. The process of refining the event plan continues until the client is satisfied with the proposed arrangements. The client retains the ability to cancel the event at any point in the preparation process unless they have provided final confirmation of the event plan. This allows the client flexibility in the planning stages, ensuring that the event is aligned with their needs and expectations before final confirmation.

Case Description: Full Event Life Cycle An event is initiated when a customer submits an inquiry, prompting the event manager to self-assign responsibility for planning the event. The event manager then proceeds to make the necessary arrangements, which includes coordinating services and drafting an initial budget. These preparations occur prior to receiving formal confirmation from the client. Once the client formally confirms the event plan, the event is added to the schedule and is set to be executed on the designated event date. After the event is completed, the event manager prepares an audit report, which is submitted to the internal management team. This report is used to evaluate the event's execution, identify areas for improvement, and inform future event planning decisions. Following the completion of the audit report, the event is considered finalized.

☞ **D-3.5** Create a fresh INTELLIJ project and copy the package ss.tictactoe.model to this project (the package is included in the files for this week). From the class Board consider the method toString:

```
public String toString() {
    String s = "";
    for (int i = 0; i < DIM; i++) {
        String row = "";
        for (int j = 0; j < DIM; j++) {
            row += "␣" + getField(i, j).toString().substring(0, 1)
                    .replace("E", "␣") + "␣";
            if (j < DIM - 1) {
                row = row + "|";
            }
        }
        s = s + row + DELIM + NUMBERING[i * 2];
        if (i < DIM - 1) {
            s = s + "\n" + LINE + DELIM + NUMBERING[i * 2 + 1] + "\n";
        }
    }
    return s;
}
```

1. Draw (on paper) the flow graph that represents the cyclomatic complexity of the method Board.toString.
2. What should be the result of the McCabe cyclomatic complexity (VG) of this method according to the calculations discussed during the lecture? What is the value calculated by the Metrics plugin?
3. Refactor the method by moving lines 4—10 from toString to a separate method rowToString, which is then called from within toString. INTELIJ can do the refactoring for you: select the lines in question and then select REFACTOR → EXTRACT METHOD.... What is the cyclomatic complexity of the refactored toString and of your extracted method?

3.2.3 Recommended exercises on activity diagrams

D-3.6 In the files for this week on Canvas, you will find the file `Dlab-3a-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.
Hint: Make sure to use the right notation, decision nodes as shown in Figure 3.2 are not acceptable.
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.

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

[1] *Disclaimer*: The texts below are based on a case study in one of the regional police forces in the Netherlands several years ago. Procedures could be different now.

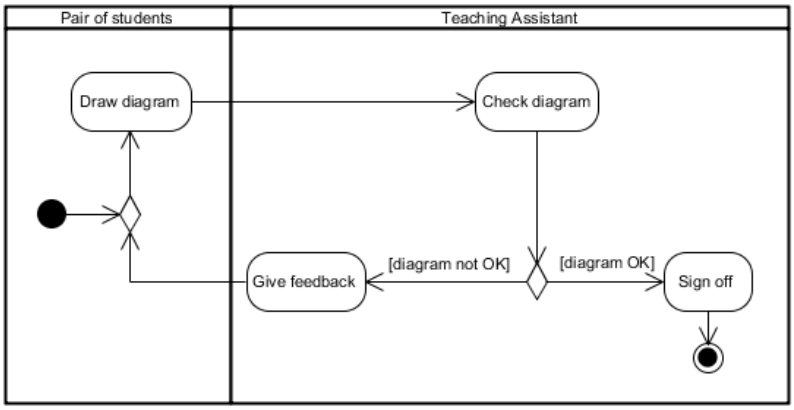


Figure 3.1: Activity Diagram for introductory exercise

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

The search functions in PBS could possibly be improved to retrieve more information, since the system is more than twenty years old. But before a project is initiated to improve things, it 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-3.7 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-3.8** 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-3.9** Extend the activity diagram by taking the following into account:

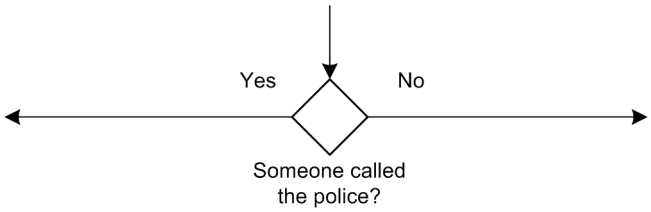


Figure 3.2: Alternative branch notation—*not acceptable*

- 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-3.10 (*Optional*) If you have finished Exercise D-3.9 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-3.9 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.

D-3.11 Make an activity diagram for X-rays in the hospital, according to the case history described below.

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

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

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

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

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

- A medical specialist in the Polderland hospital usually requests a radiographic examination during a consultation with the patient. From within the patient's electronic record, the specialist needs can open a request form with a single mouse click. The patient's essential data will be included automatically. The specialist adds a reason for the request.

- After the request has been filed by the medical specialist, the patient should make an appointment for the examination. A patient can visit or phone the Radiology secretariat for an appointment. A secretary will record an appointment in the system for a time that suits the patient. (*We ignore special cases where the request is urgent and perhaps the patient is incapacitated, then the medical staff will make an immediate appointment on their behalf. You don't have to model that.*)

- General practitioners in the region can also make a request for a radiographic examination of a patient. In this case the request will be made electronically as well, but chances are that the patient data are not as complete as the hospital would like to have them.
  Consequently, if a patient contacts the Radiology secretariat, the secretary should do two things: make an appointment for the patient and check whether the patient data are complete. If not, ask the patient for the missing data and enter them into the system.

- At the appointed time the patient checks in at the Radiology secretariat. A secretary places a patient on the work list for that day. Usually it takes 10–15 minutes until it is the patient's turn, in exceptional cases it could take a bit longer.
  Unfortunately, it happens that patients do not turn up. If a patient hasn't checked in one hour after the appointed time, a letter is generated automatically, asking the patient to make a new appointment. Later that day a secretary prints the letter and sends it by (physical) mail to the patient.

- The X-rays are made by a radiology assistant. The assistant positions the patient so that the right body part will be photographed from the right angle and then hides 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.

3.2.4 Recommended exercises on state machine diagrams

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

In this case study, we disregard most of their services and focus only on the repair service.

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

RedHot knows what to do with it when it arrives.

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

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

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

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

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

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

☞ **D-3.12** Make a state machine for the state of an article.
Hints:

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

Repair service of an internet shop (extended version)
Some details, left out in the first version, still need to be included.

- Warranty. Most articles come with a warranty period of a few years. When an article breaks down during the warranty period, repairs are free of charge. Therefore, if warranty applies, the customer does not have to give permission and there is no need to make a cost estimate.

- Repair by the supplier. In some cases, RedHot does not carry out the repair itself, but sends it back to the supplier of the article. In the process as it is presented to the customer, it does not make any difference. RedHot makes an estimate of the costs. When the customer gives permission for repair, the article is sent to the supplier (which of course is registered in RedHot's administration). When the repaired article arrives back from the supplier it is registered as repaired and then sent back to the client as usual.

☞ **D-3.13** Extend the state machine with the new details.
The extended state machine should have a transition "warranty expires". Please be aware that this can

happen at any moment in time, also during a repair. (However, the repair is still free of charge if the article was reported broken before the warranty expired).

D-3.14 Draw a state machine diagram for the state of a pet, according to the case description below.

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

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

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

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

- Everyone who owns a pet can have it registered by a veterinarian. The vet (or an assistant) implants the RFID in the pet, gives the owner a registration certificate (on paper) with all the information and registers the pet with ADN. The RFID code consists of a unique 15-digit number. The composition of this number has been standardized across the EU. The first three digits identify the country ('528' for the Netherlands), the next three digits identify the RFID producer, the remaining nine digits identify the animal.
- A change in address can be entered into the system by the owners themselves. Changing the ownership of an animal can be reported by the old owner on the ADN website. To make sure that this is done properly, ADN will contact the new owner for a confirmation.
- Occasionally it happens that pets run away. Also, certain types of animals can get stolen (e.g. koi carps, who do get RFIDs for this very reason). If an animal is missing, the owner can report this by means of a web form on the ADN website.
- When people find a run-away pet, they can bring it to the nearest animal shelter. Depending on the circumstances, one can also call the animal ambulance to pick it up. An employee of the animal shelter scans the pet's RFID code, registers it as "found" with ADN, and gets the information about the owner from ADN. ADN registers the date on which the animal was found and the animal shelter that made the registration. If an e-mail address is known, the owner automatically gets a notification where the pet can be picked up. In addition, the animal shelter always phones the owner. When the pet is picked up, an employee of the animal shelter records in the ADN database that this has happened.
  With found pets, the following special cases can be distinguished:
  - It could be dead, or so badly wounded that it has to be killed.
  - Sometimes an animal is found that carries no RFID. The animal shelter always implants an RFID (unless the animal has to be killed) and registers it with the ADN. An owner may turn up later, in which case the animal shelter updates the registration.
  - It also happens that a pet with RFID is found, which has not been reported missing. In that case, the animal shelter contacts the owner. The fact that the animal was not missing could be an indication of suspicious circumstances. The animal shelter may contact the Animal Protection Society, which could further investigate the case. If the owner cannot be traced or

refuses to pick up the pet, the animal shelter contacts the Animal Protection Society, who can dispossess the owner.
  - Animals without owners stay in the animal shelter until a new owner can be found. Fortunately, there are a lot of people who get a new pet from the shelter. If this happens, the animal shelter updates the registration. If the shelter's capacity if fully used, animals for which no new owner can be found have to be killed.
- If a pet is still missing after three months, ADN will change its status from "missing" to "lost forever". The owner gets a letter in which it is explained that, unfortunately, their pet has not been found and, after so much time, it is unlikely that this will ever happen. Nevertheless, once in a while, a permanently lost pet is found. In that case, the procedure is the same as for a missing pet.
- If an animal dies, any of the concerned parties (owner, vet (assistant), animal shelter employee) can register this with ADN. Many owners forget to do this, creating some data pollution in the ADN database.

3.3 Introduction to Programming

3.3.1 Practical exercises

Import the contents of this week's files from Canvas to your IntelliJ project by following the instructions mentioned on page 26.

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.

IP-3.1 Study the documentation of the `BasicPassword` class. The documentation is included in the 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.

IP-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.

IP-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.

Study This!

Study ECK Section 5.7 to learn about interfaces.

IP-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`.

Write useful documentation (Javadoc) for the `Checker` class and its methods.

IP-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 IP-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 `Checker`s 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`.

IP-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()`. Make sure that the `Password` class uses the `Checker` to determine whether the password is acceptable.

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.

Sign off point IP-3.A

To sign off, present the following:

- Your results and code for exercises IP-3.1 to IP-3.6

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. `Bill`s 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

IP-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 `BillPrinter.format` method a `default` implementation, which calls `String.format` to format the `text` and `price`, such that the statements

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

(where `p` is a `BillPrinter`-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.

IP-3.8 Program the following `BillPrinter`-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 `BillPrinter` 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*.

Study This!

Study ECK Section 5.8 to learn 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.

IP-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`.

IP-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()`, annoted `@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 `contains(CharSequence s)` method available on String objects in conjunction with something like `assertTrue()`
- To test **`double`**s 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.

Sign off point IP-3.B

To sign off, present the following:

- Your results/code for exercises IP-3.7 through IP-3.10

Password-Protected Hotel Safe

Now that you have a `Bill` and you can print it, you can implement `Item`s 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.

IP-3.11 Specify a class `ss.hotel.PricedSafe` with Javadoc 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

Study This!

Study ECK Section 8.4.1 to learn about assertions.

For the `open` and `activate` methods mentioned below and for the constructor of the new class, add **`assert`** statements that tests that the input variables have valid values, such as, that the given password is not **`null`** or empty, that the price is not negative, etc. You will use them in Exercise IP-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 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`.

IP-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.

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

IP-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.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 IP-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.

IP-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?

IP-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`.

IP-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.BillPrinter` for the bill. The method should return an instance of `Bill`, which was created in Exercise IP-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.

IP-3.18 Make a copy of 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.

Sign off point IP-3.C

To sign off, present the following:

- The tests of exercise IP-3.12

- The results of exercise IP-3.14
- Your code for exercise IP-3.18

3.3.2 Recommended exercises

Hotel bill improvement

IP-3.19 Improve your hotel bill class in Exercise IP-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.

IP-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.

IP-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

IP-3.22 In Exercise IP-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.

IP-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

IP-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).

IP-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 should have the query `canMoveTo(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.

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

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

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

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

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

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

Write a program or JUNIT tests to test your implementation.

Week 4

Week 4

4.1 Overview

4.1.1 Contents of This Week

Design The activities in this week cover the following topics:

- Topics: Q&A session on anything related to the Design part.
- Practicals: design patterns, see Section 3.2.2.

Programming This week the following topics will be discussed:

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

4.1.2 Mandatory Presence

There are no mandatory activities this week.

4.1.3 Expected Self-Study and Project Work

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

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

4.1.4 Materials for this Week

Design Watch video [`L7T1`].

Programming

Lecture ECK, Sections 7.1–7.3, 9.1–9.2 and 10.1–10.4, the week 4 topic videos
Practicals The following files are provided on Canvas:
Tools Use of the "Style Problem" inspections

4.2 Design

4.2.1 Practical exercises on design patterns

☞ **D-4.1** Remember the design patterns that were discussed in the videos. Below are five systems that EventHorizon is investigating to add to their EventHorizon Platform (EHP). For each of them decide which design pattern you would use and briefly motivate your choice. No need to actually flesh out the design of any of these.

1. EHP venue environment sensor hub pushing live readings to an on-site kiosk, attendee mobile app, and LED wall.
2. EHP stage media player for guided audio tours with Stopped → Playing → Paused modes controlling allowed actions during shows.
3. EHP branding and accessibility themes (light/dark/high-contrast) applied across all UI components.
4. EHP event schedule editor where event managers edit event entries (title, time, room, speaker) via a form; changes should appear in both a schedule table and an event detail view.
5. EHP checkout supporting card, PayPal, and bank transfer with distinct validation/charge flows.

☞ **D-4.2** For each system from exercise 1, sketch a small UML class diagram (3–4 classes, no attributes or methods needed, we focus on the associations here). Use only the titles provided and add associations as you see fit to match the patterns you identified in Exercise 1.

- **EHP venue environment sensor hub**
  Classes: `SensorHub`, `KioskDisplay`, `MobileApp`, `LedWallController`.
- **EHP stage media player for guided audio tours**
  Classes: `MediaPlayer`, `StoppedMode`, `PlayingMode`, `PausedMode`.
- **EHP branding and accessibility themes**
  Classes: `ThemeService`, `LightTheme`, `DarkTheme`, `HighContrastTheme`.
- **EHP session scheduling console**
  Classes: `ScheduleData`, `ScheduleTable`, `EventDetailView`, `EditForm`.
- **EHP checkout: card, PayPal, bank transfer**
  Classes: `BankTransfer`, `Checkout`, `PaymentMethod`, `Card`, `PayPal`.

☞ **D-4.3** Read the case description below (Route Planning) and produce a UML class diagram that enables interchangeable routing behaviors and runtime switching while keeping the planner independent of concrete algorithms and external providers. Model only the types and relationships that are necessary. Include the main planner, an abstraction for routing behavior, at least three distinct behaviors, and any helper abstractions needed to hide external providers. You can keep UI and persistence out of scope.

Case Description: Route Planning EventHorizon's logistics team plans point-to-point routes between venues, hotels, and suppliers. Depending on event goals, the planner wants different optimization behaviors: minimize travel time for tight schedules; minimize monetary cost when budgets are strict; minimize estimated CO_2 for green events; combine criteria for premium events. The current planner module is a single class with hard-coded conditionals for each optimization, making changes risky and blocking parallel work. The UI and downstream systems only need a route result with checkpoints and a score, and must not depend on how the route was computed. The business wants to add new optimization behaviors later without touching existing planner code. Switching the active behavior must be possible at runtime from the planner UI. External providers (map APIs, emissions calculators) may vary per optimization and must not leak into the rest of the system.

4.3 Advanced Programming

4.3.1 Practical exercises

Import the contents of this week's files from Canvas to your IntelliJ project by following the instructions mentioned on page 26.

Code style

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

AP-4.1 This class violates the configured "Style Problem" coding conventions in several ways. Check the class for those inspections and write down the violations. (There may be other inspections that are not "Style Problems" only write down the inspections marked in the colour you set during the installation) Now modify the class implementation such that the functionality of the class implementation does not change, but the coding conventions are satisfied.

Note: If the style problems are not being highlighted make sure you've set them up according to the installation instructions. Furthermore, you may have to right click the file, go to Analyze -> Inspect Code.. to manually rerun the inspections.

Tip: IDEs like IntelliJ also offer features to automatically fix the layout of your source code.

Preconditions and postconditions with JML

Study This!

Make sure you have studied Appendix B (page 111) about JML specifications and the related topics on Canvas.

This week, in addition to writing Javadoc, you will be asked to *specify* methods of classes, that is, to write preconditions, postconditions, and sometimes invariants. See Appendix B (page 111) 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 exercises, this means adding specifications in the form of invariants, preconditions and postconditions in JML.

AP-4.2 In this exercise, you are asked to provide a *formal specification* for a method that you worked earlier. Examine your method `fibonacci` from Week 1. In order to write a JML specification for that class, we need to first informally describe what this method expects from its input and what it guarantees about its output. Then, we use JML syntax to formalize this specification. Let's apply this in practice:

1. Think of at least one **precondition** and at least one **postcondition** for the `fibonacci` method. What does it **require** from the input to run correctly and what does it **guarantee** about the output? Note these down as a comment above the method, to show the TA when you're signing off.
2. Formalize the statements you have come up with using JML pre- and postconditions. Write them above your method with appropriate JML syntax.

Note: you can use the implies operator `==>` to reason about different cases, that is, use `expr1 ==> expr2` to reason that if `expr1` is true, then `expr2` must be true. For example, `expr1` can be a Boolean expression on the parameters and `expr2` can be a Boolean expression on the result.

Sign off point AP-4.A

To sign off, present the following:

- The result of exercise AP-4.1
- Your specification of exercise AP-4.2

Sorting

In Module 1, you developed implementations of various sorting algorithms, such as bubble sort and merge sort. See, for example, `http://y2u.be/EeQ8pwjQxTM` for an explanation of merge sort.

Study This!

Study ECK Sections 9.1, 10.2.1 and 10.2.2 to learn about lists.

AP-4.3 Reimplement the merge sort algorithm from Module 1 in Java for `Lists` of `E`, where `E` is a generic type such that `E extends Comparable<E>`. If you have not done Module 1 you can find the description of the merge sort algorithm at https://en.wikipedia.org/wiki/Merge_sort. Use the provided stub code in class `MergeSort` for your implementation, and `MergeSortTest` to test it.
Also write JML preconditions and postconditions for the method, in particular to specify that the result is sorted. For example, you could use the keyword \forall. Hint: See Appendix B (page 111) to see how to use JML keywords. Below you can find the pseudo-code for the merge sort algorithm.

```
1:  input: List data
2:  output: Sorted copy of list data
3:  if data has 0 or 1 element(s) then
4:      return data;
5:  else
6:      List fst ← mergeSort(first half of data);
7:      List snd ← mergeSort(second half of data);
8:      List result ← new empty list;
9:      fi, si ← 0;
10:     while fi and si are valid indexes for fst and snd do
11:         if fst[fi] < snd[si] then
12:             Append fst[fi] to result;
13:             Increase fi by one;
14:         else
15:             Append snd[si] to result;
16:             Increase si by one;
17:         end if
18:     end while
19:     if fi is a valid index in fst then
20:         Append fst to result, starting from fi;
21:     end if
22:     if si is a valid index in snd then
23:         Append snd to result, starting from si;
24:     end if
25:     return result;
26: end if
```

AP-4.4 Instead of writing your own sorting algorithms for Lists, you can use the sort method Java provides through the Collections framework.
See also https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collections.html
Write a test in `MergeSortTest` that creates a list with 20.000 random integers, copy it into another list, and checks that both lists are the same when one is sorted with `MergeSort.mergeSort()` and the other with `Collections.sort()`.

Sign off point AP-4.B

To sign off, present the following:

- The JML specification and the code for exercise AP-4.3
- The code for exercise AP-4.4

Linked Lists

Next you will implement linked lists, as discussed during the lectures. However, there is one difference: this linked list implementation uses a **sentinel node**. A sentinel node acts as the beginning and the end (also known as the head and the tail) of the list. We don't store any data in this node, it only acts as our reference to the chain of nodes. Because this sentinel node always exists you don't need to do anything special when the list is empty. The "next" field of the sentinel node always points at the first element of the list and the "previous" field always points at the last element. If the list is empty, then these references are `null`. If unsure how to implement operations, it may be helpful to draw a diagram.

Study This!

Study ECK Section 9.2 to learn about linked lists.

AP-4.5 Implement the methods `add(int index, E element)` and `remove(int index)` of the `DoublyLinkedList` class provided this week. Use the JUNIT test `DoublyLinkedListTest` to test your implementation.

Hint: Make use of the provided `getNode(int i)` method.
Hint: Try to draw the nodes in a doubly linked list and how they are connected. Then figure out which references need to be changed when adding a node before another node, or removing a node between two other nodes.

Tic Tac Toe

In the following exercises you will implement the immensely popular game Tic Tac Toe (see https://en.wikipedia.org/wiki/Tic-tac-toe). Tic Tac Toe is a turn-based game for two players X and O, who take turns marking the spaces in a 3x3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal line wins the game.

The program design of the game consists on the high level of the following interfaces:

- A `Game` interface representing a turn-based game in progress. It has several important methods:
  - `Player getTurn()` to see whose turn it is.
  - `List<Move> getValidMoves()` to get all possible moves.
  - `boolean isValidMove(Move move)` to check if a particular move is allowed.
  - `doMove(Move move)` to play a particular move.
  - `boolean isGameover()` to check if the game is over.
  - `Player getWinner()` to see who won the game.
- A `Move` interface representing a move. It has no methods, as any details are game-specific. The purpose of the interface is to mark objects of type `Move` as objects representing moves of the game.
- A `Player` interface representing a player. It also has no methods and serves to mark objects representing players of the game.

These interfaces are implemented with specific Tic Tac Toe classes:

- A `TicTacToeGame` class, which uses a `Mark` enumerated type to represent the X and O marks on the board, and which uses a `Board` class to represent which marks are placed where.
- A `TicTacToeMove` class, representing the move to place a specific mark on a specific place.

For players, we introduce an abstract class, which will have several subclasses:

- An `AbstractPlayer` which records the name of the player and allows choosing a move via an abstract method `Move determineMove(Game game)` that depends on the kind of player, e.g., a human player or a computer player.

In order to play a game, we also need a user interface. The user interface should be separated as much as possible from the rest of the program. Maybe later you want to develop a graphical user interface, or a web interface. In those cases, we want to change as few classes as possible. We define the following UI classes for our TUI:

- A `TicTacToeTUI` class that can start games and that asks the player whose turn it is to determine their next move until the game is over.
- A `HumanPlayer` class implementing the `determineMove` method by asking for the move on the console.

Of the above classes, you will need to complete the implementation of `Board`, write the full implementation of `TicTacToeGame`, `TicTacToeMove`, `TicTacToeTUI` and `HumanPlayer`.

In our version of the Tic Tac Toe game, the board is represented by a one-dimensional array of `3*3=9` `Mark`s. The relation between the array and the board is shown below:

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

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

AP-4.6 Implement the missing methods of the class `ss.tictactoe.model.Board`. Make sure your implementation is valid according to their preconditions and postconditions. In your implementation you should use constants instead of hard-coded values. Who knows, in the future you want to play Tic Tac Toe on boards of 5x5? Use the `BoardTest` JUnit tests from Canvas to test your implementation.

AP-4.7 Define **JML invariants** for the Board class which formalize the following two properties:

- The length of the `fields` array is bounded by the `DIM` constant.
- The number of occurrences of any player mark is less than or equal to 5.
- The number of `Mark.XX` is always greater than or equal to the number of `Mark.OO`. (Given that `Mark.XX` is the starting player symbol.)

Hint: See Appendix B (page 111) for explanation of the `\num_of` keyword.

AP-4.8 Create classes `TicTacToeMove` and `TicTacToeGame` implementing the interfaces `Move` and `Game` respectively. The `TicTacToeMove` class must represent a move in the game, i.e., store the mark to place and the location to place it. The `TicTacToeGame` class should encapsulate a `Board` as well as keep track of who the players are and whose turn it is. The class should also have a `toString()` method that returns a String representing the current state of the game, i.e., the board and whose turn it is.

For generating valid moves, consider all empty fields of the board and add appropriate objects to a `List<TicTacToeMove>` to be returned. If the game is over, of course, an empty list is returned. To check whether a move is valid, consider whether the game is not yet over, check if the mark being placed is of the current player, and check if the location to place the mark is currently empty.

You are now ready to implement the TUI to play the game. Similar to the TUIs you made in previous weeks, make a `TicTacToeTUI` class in the package `ss.tictactoe.ui` with a minimal `main` method and the functionality in a `run()` method.

When the program starts, it should ask for the names of two players. Then the TUI should make appropriate `Player` objects for them; for this, make a `ss.tictactoe.ui.HumanPlayer` class extending `AbstractPlayer` and implement the `determineMove` method to ask the player for a valid move. Keep asking until a valid move is given by the player, then return that move.

After creating the two `HumanPlayer` objects, your TUI should repeatedly play a game as follows:

- Create a new `TicTacToeGame` object.
- While the game is not over:
  - Display the current state of the game.
  - Ask the current player to determine a move (by calling `determineMove`).
  - Play the move (by calling `doMove`).
- If the game is over, display the winner (unless there is a draw).
- Ask the user if they want to play another game.

AP-4.9 Implement the classes `ss.tictactoe.ui.TicTacToeTUI` and `ss.tictactoe.ui.HumanPlayer`. Test your system by playing some games.

Sign off point AP-4.C

To sign off, present the following:

- The code for exercises AP-4.6 to AP-4.9.

Function Properties

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

Study This!

Study ECK Section 10.3 to learn about the Map interface.

For example, given a function $f : \mathbb{N} \to \mathbb{N} : x \mapsto x + 1$, we can model this as a map `mapF`, and if 1 is in the `keySet()` of `f`, then `mapF.get(1)` should return 2. For more information, see https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Map.html.

In this exercise, you will implement a class `MapUtil` that defines several auxiliary functions to define commonly used properties and definitions of mathematical functions:

- a function $f : X \to Y$ is *injective* or *one-on-one* if (see Figure 4.1):

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

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

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

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

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

- the *composition* of two functions $f : X \to Y$ and $g : Y \to Z$ is a function $h : X \to Z$ such that (see Figure 4.2):

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

AP-4.10 Implement the static method `isOneOnOne` in the provided `MapUtil` class that checks whether a `java.util.Map<K,V> f` passed as a parameter is an injection, i.e., `isOneOnOne` returns true if for all `v` in the value set of the map `f`, there exists exactly one key `k` in the map's key set, such that `v == f.get(k)`.

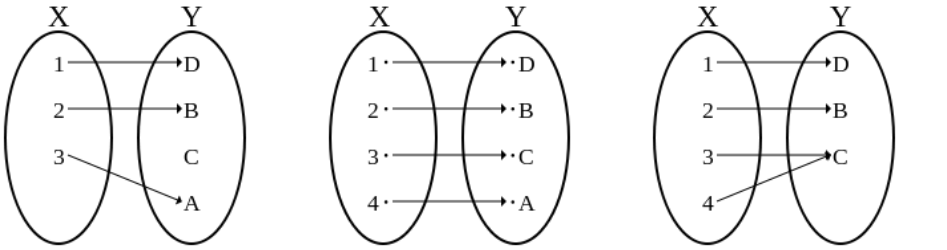
Write preconditions and postconditions (JML) where applicable for this method that at least specify what this method returns. Test your implementation using the provided `IsOneOnOneTest` JUNIT tests.

Not all mathematical properties can be implemented directly. To check whether a method is surjective, *i.e.*, for all values in the range of the function, there is a value in the domain that maps to it, we have to pass the intended range as an explicit argument.

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

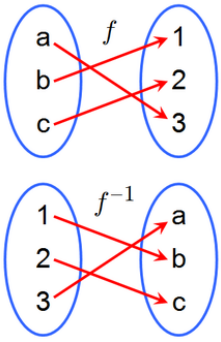
Write preconditions and postconditions where applicable for this method that at least specify what this method returns. Test your implementation using the provided `IsSurjectiveOnRangeTest` JUNIT tests.

Next you will implement the inverse of a function. There are two alternatives, namely one that defines the inverse of an arbitrary function, and one that defines the inverse of a bijection, *i.e.*, of a function that is one-on-one and surjective.

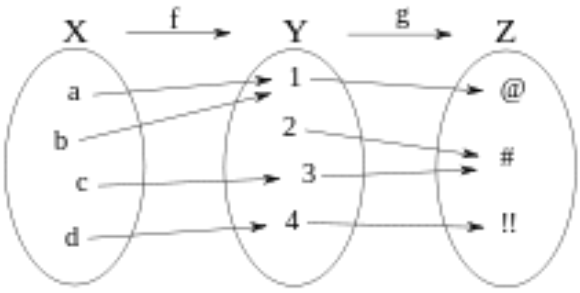


Injective but not surjective | Injective and surjective | Surjective but not injective

Figure 4.1: Examples of injectivity and surjectivity



A function and its inverse



Function Composition

Figure 4.2: Examples of inverse and function composition

AP-4.12 Implement the static method `inverse` that given a `Map<K,V>` object returns a map of type `Map<V, Set<K>>`. Explain why you need the result type to be a `Map<V, Set<K>>` instead of `Map<V,K>` in this case.

Additionally, implement the static method `inverseBijection` that returns a map of type `Map<V, K>` if the function is injective and surjective.

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

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

AP-4.13 Implement the static method `compatible` that checks whether the two maps passed as parameter can be composed, *i.e.*, whether all values in the value set of the first map are in the key set of the second map.

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

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

Sign off point AP-4.D

To sign off, present the following:

- The code for exercises AP-4.10 until AP-4.13

The Stack-based Calculator

In the following exercises, you will develop a simple stack-based calculator. The idea of this calculator is that it stores values on a "stack". For an example of how a stack works, see Figure 4.3).

You will implement the following elementary operations on this stack-based calculator:

- `push`: put a value on the stack.
- `pop`: remove a value from the stack and return it.
- `add`: Take two values from the stack, add them and put the result on the stack.
- Similarly, `sub` (subtraction), `mult` (multiplication) and `div` (division).

AP-4.14 Inspect the `Calculator` interface in the `calculator` package. Inspect the `CalculatorTest` class in the test subdirectory. Notice that the test class is currently `@Disabled`. The `setup` method is empty, but it should set the `calculator` field to an instance of the `Calculator` interface.

- Create a new suitable package for your implementation, inside the `calculator` package. In all following steps, your classes should live in this package.

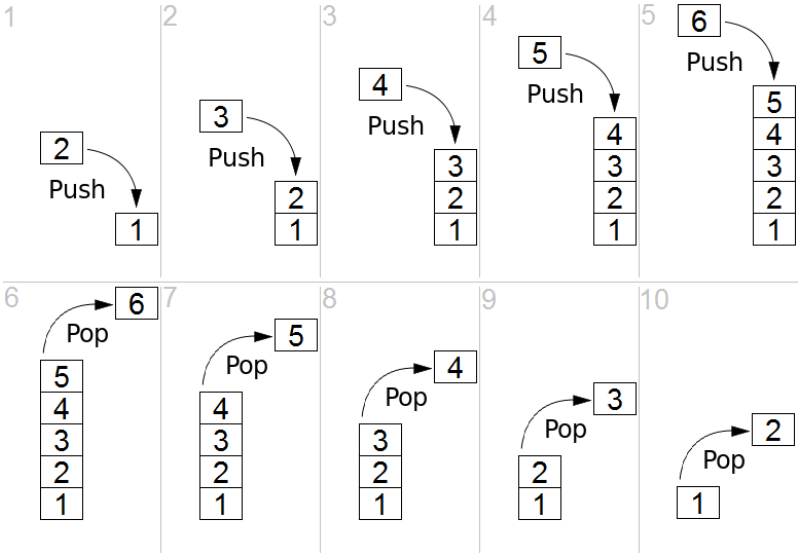


Figure 4.3: Example of a stack (source: https://en.wikipedia.org/wiki/File:Lifo_stack.png)

- Create a class that implements the `Calculator` interface. Implement all methods in your class with stubs.
- Modify `CalculatorTest.setup()` to make a calculator.
- Remove the `@Disabled` annotation from the test class.
- Run the tests and verify that the `CalculatorTest` tests fail.
- Start implementing your `Calculator` until all tests pass.
  Hint: first implement `push` and `pop` until `testPushPop` works, then continue with the four operations `add`, `sub`, `mult` and `div`. You can use your `push` and `pop` methods to implement those.

AP-4.15 Imagine we want to have extra functionality. Add the "dup" operation that duplicates the value on top of the stack: for example, if the value on top of the stack is 123, push another 123 on top of the stack. Add the "mod" operation that computes the remainder after division. In Java, this is the % operator, as in `a % b` computes a modulo b. Then perform the following operations in this order:

- Add two appropriate methods to the `Calculator` interface. Add the appropriate Javadoc to these two methods.
- Add appropriate tests in `CalculatorTest`, including normal behaviour and error handling.
- Implement the methods in the `Calculator` and ensure your tests pass.

Sign off point AP-4.E

To sign off, present the following:

- The code for exercises AP-4.14 and AP-4.15

4.3.2 Recommended exercises

AP-4.16 Make the following exercises from ECK:

- Exercises 7.1, 7.2, 7.3 and 7.5.

4.4 Mathematics: Relevance for Computer Science

The goal of the Mathematics case session this week is to show you the relevance of Mathematics for TCS.

During the first half of the session there will be "speed dates" with several researchers who are from different departments of Computer Science.

You will be divided in groups. These groups will be discussing with the participating researchers to find out how they use mathematics for research. The exact schedule will appear on Canvas. The purpose of the discussion is that you find out the following:

- What mathematics does the researcher use?
- How does the researcher apply this mathematics?
- Can you find applications of this mathematics in every day life?
- What are the computer science applications of this mathematics?
- Is the mathematics mainly applied, or is new theory being developed?
- What mathematical knowledge are you still missing related to this research?

Further details about the session will be shared on Canvas.

Week 5

Week 5

5.1 Overview

5.1.1 Contents of This Week

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

- The Test. For a small sample test, see Section 5.2.1.

Programming This week the following topics will be discussed:

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

5.1.2 Deadline

The deadline for team report assignment is **Fri 23:59**

5.1.3 Expected Self-Study and Project Work

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

- 3 hours self-study for final preparations for the Design test;
- 2 hours self-study for the Programming thread.

5.1.4 Materials for this Week

Programming

Lecture ECK, Sections 3.7 and 11.1 and 11.2, Chapter 8
Chapter 5 of "Security Engineering", Sections 5.3–5.3.1.2 and 5.6–5.6.2
The week 5 topic videos

Practicals The following files are provided on Canvas:

5.2 Design

5.2.1 Test

The questions in this test have comparable difficulty to the practicals, recommended exercises, and in particular the questions on the final exam. Unlike the practical, you will submit your solutions through the Remindo system. All material of all the topics can be included in the test questions, except "`git` *in a Nutshell*" [**L5T3**].

The test is an open slides test. See the section *Tests and Grades* (p. 6) in the Introduction for what materials you can use during the test.

How to prepare for the test

All you need to know, is covered by the topics (videos and slides). The best way to prepare for the test is by rewatching the topic videos attentively and by doing the recommended exercises and the example tests. You should look at the solutions *after* you have given it a serious try—and consult the slides if you have difficulties understanding the solutions.

5.3 Advanced Programming

5.3.1 Practical exercises

Study This!

Go to `https://refactoring.guru/design-patterns/strategy` and read about the **Strategy** pattern.

Strategy

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

AP-5.1 Document (with Javadoc) and specify (with JML) an interface `ss.tictactoe.ai.Strategy` to determine the next move for the Tic-Tac-Toe game. This interface must have the following two methods:

- **`public`** `String getName()` that returns the name of the strategy;
- **`public`** `Move determineMove(Game game)` that returns a next legal move, given the current state of the game.

AP-5.2 First you have to develop a naive strategy (called `"Naive"`), in which method `determineMove()` returns an arbitrary move.

Write the class `ss.tictactoe.ai.NaiveStrategy` that implements this naive strategy by implementing the `Strategy` interface.

Hint: you can use method `Math.random()` to choose a random valid move.

AP-5.3 Implement a class `ss.tictactoe.ai.ComputerPlayer` that extends class `AbstractPlayer`. A `ComputerPlayer` should be initialized with a strategy and it should have a name. As the name, you could choose a default name of the strategy, followed by a hyphen (`"-"`), followed by a representation of the player's mark. You could also choose to let your computer player have any other name as specified in the constructor.

Class `ComputerPlayer` should have the constructor:

- **`public`** `ComputerPlayer(Mark mark, Strategy strategy)` that constructs a computer player using the given mark and strategy.

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

AP-5.4 Modify `TicTacToeTUI` such that the game can be played with a naive computer player. For example, by adding an option to make a computer player instead of a normal player. Test the program by playing some games against the naive computer player. It should also be possible to let two computer players play against each other.

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

AP-5.5 Implement a class `ss.tictactoe.ai.SmartStrategy` that implements the `Strategy` interface. Method `determineMove()` returns a move using the following strategy:

- If there is a move that guarantees a direct win, this move is returned.
- If there is a move after which the opponent has a winning move, then this move is not allowed. If no move is allowed after this, then return a random move.
- If none of the cases above applies, a random allowed is returned.

Hint: It is advisable to make a copy of the board in the implementation of `determineMove()` to calculate what happens next when a move is played. To do this, you could implement a method `deepCopy` for the `TicTacToeGame` class. You can then call `doMove` on a copy of the game.

Hint: implement a method `Move findWinningMove(Game game)` that returns a move that wins the game immediately, or **`null`** if no such move exists. Use this to find a move that guarantees a direct win, and to check if the opponent can win in one move. You can use `Game.getWinner()` to see if a game is won after playing a move.

Hint: make a local variable `List<Move> allowedMoves` to keep track of moves that do not let the opponent win immediately.

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

AP-5.6 Update the class `TicTacToeTUI` in such a way that also the smart computer player can play the game.

Test the program by playing some games against the smart computer player. Also play some games where the naive computer player plays against the smart computer player.

Sign off point AP-5.A

To sign off, present the following:

- The results and code for exercises AP-5.1 to AP-5.6.

Encoding

When dealing with security-related subjects such as cryptographic hashes, MACs, etc., one often needs to handle binary data (i.e., raw bytes). To be able to represent these binary data correctly in plain, normal (UTF-8) text you can use encoding methods[1], such as Hex and Base64. See also `http://en.wikipedia.org/wiki/Hexadecimal` and `http://en.wikipedia.org/wiki/Base64`.

There is no need to implement these encoding schemes yourself, as you can use existing libraries that implement them. In the next assignments you will work with one of these libraries, namely the Apache Commons Codec library (see `https://commons.apache.org/proper/commons-codec/`), using **`static`** methods of the classes `org.apache.commons.codec.binary.Hex` and `org.apache.commons.codec.binary.Base64`.

AP-5.7 To get you started, you are given the class `ss.week5.EncodingTest`. This class uses the Apache Commons Codec library, which can be obtained from `https://commons.apache.org/proper/commons-codec/download_codec.cgi`. Download file `commons-codec-1.15-bin.zip` and extract its contents. The bytecode of the library is in a Java archive (JAR) file called `commons-codec-1.15.jar`.

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

Now you have to add this library to your Java project in order to run the `EncodingTest` program. There are multiple ways to achieve this but below we set out the route we prefer you take. Copy the jar-file with the library (`commons-codec-1.15.jar` in our case) to the `lib` directory of the project. You probably first have to create this directory. Then go to FILE → PROJECT STRUCTURE... → LIBRARIES. Click on the +-symbol and select Java. Navigate to the previously mentioned lib folder and select the `commons-codec-1.15.jar` file. Click OK and Apply. Now INTELLIJ is aware of your new toy as well!

The class `EncodingTest` prints out the hexadecimal encoding of the ASCII string "Hello World" by first getting the raw bytes representation of the string using the `getBytes` method of the `String` class. These raw bytes are given to the `encodeHexString` method of the `org.apache.commons.codec.binary.Hex` class. The provided code prints out `48656c6c6f20576f726c64` as the hexadecimal representation of the ASCII string `"Hello␣World"`. Now change the input string to `"Hello␣Big␣World"`. How does the hexadecimal output change?

AP-5.8 This same library also supports the conversion of an hexadecimal representation back to bytes. Add a few lines of code to the `EncodingTest` class so that it converts the hex string `4d6f64756c652032` to bytes (a byte array) and then prints the result when these bytes are turned into a `String`. Check the documentation for the Apache Commons Codec library at https://commons.apache.org/proper/commons-codec/archives/1.15/apidocs/index.html to determine which method to use. Use `new String(bytearray)` to create a string from a byte array.

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

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

- First, based on your experience with the `Hex` class, encode the string `"Hello␣World"` in Base64.
- Next, take the hex string `010203040506`, decode it to a byte array, encode this byte array with Base64, and print it. What is the output? What is the length of the Base64 representation? Can you see an advantage of Base64 over Hex encoding?
- Then, decode the Base64 string `U29mdHdhcmUgU3lzdGVtcw==` and print it.
- Finally, produce the Base64 encoding for each of the following strings: `"a"`, `"aa"`, `"aaa"`, `"aaaa"`, `"aaaaa"`, ..., `"aaaaaaaaaa"`. What do you notice?

Exceptions

Study This!

If you have not done so yet, study ECK Chapter 8 about robust programming, especially Section 8.3.

On Canvas, you can find the class `Zipper`, which implements functionality to "zip" two Strings into one, i.e., to create a new String by concatenating the characters alternating between both Strings.

The method `zip()` has two preconditions and the `main()` method tests the provided command line arguments to see whether they fulfill these preconditions. If this is not the case, an error message is printed; the `zip()` method is only called when the preconditions are met.

AP-5.10 Define three classes:

- `ss.week5.WrongArgumentException`, which extends `Exception`[2],
- `ss.week5.TooFewArgumentsException`, which extends `WrongArgumentException`, and
- `ss.week5.ArgumentLengthsDifferException`, which extends `WrongArgumentException`.

Define a new method `zip2()` that performs the same functionality as `zip()`, but in addition checks whether the provided arguments satisfy the preconditions. If the first precondition is violated, a `TooFewArgumentsException` must be thrown, if the second one is violated, an `ArgumentLengthsDifferException` must be thrown. The message of a thrown exception (which can be queried from

[2]Note that the built-in `IllegalArgumentException` has a very similar purpose. For the project, you should use that one instead, since it is undesirable to duplicate standard API components. Create your own exceptions for this exercise, however, to practise with defining them.

an exception object using `getMessage()`) must correspond to the error message that is printed by the provided `main()` method.

Now also change the `main()` method such that:

- The new method `zip2()` is called.
- The `main()` method does not check the preconditions itself anymore.
- The functionality performed by the `main()` method does not change.

Use JUNIT tests `ss.week5.ArgumentExceptionTest` and `ss.week5.ZipperTest` from Canvas to test your implementations.

Sign off point AP-5.B

To sign off, present the following:

- The results and code for exercises AP-5.7 to AP-5.10.

The Stack-based Calculator

Study This!

Study ECK Sections 11.1 and 11.2 about streams and files. Don't forget the topic videos on Canvas.

During week 4's exercises, you have made a simple stack-based calculator. You extended the functionality of the calculator by adding two new operations to the interface, adding unit tests for the operations and implementing the new operations.
In the next exercise, you will implement a class that can be used to read a file containing instructions for the calculator, and write the result.
Recall what you have learned about *streams*. There is a difference between streams that operate on `bytes` (the internal encoding of bits and bytes in a computer) and streams that operate on `chars`.

- Classes that operate on `bytes` are `InputStream`, `OutputStream` and related classes.
- Classes that operate on `chars` are `Reader`, `Writer` and related classes.

In this exercise, we only use `chars`.

AP-5.11 Inspect the `StreamCalculator` interface and the `StreamCalculatorTest` test. Look at the file `calculatorinstructions`. Notice how the `fileAndStringTest` method opens the file for reading and how the output of your `StreamCalculator` is written to a `String` via the `StringWriter`.

- Create a class that implements the `StreamCalculator` interface.
- Modify `StreamCalculatorTest.setup()` to use your new class and remove the `@Disabled` annotation from the test class.
- Implement your `StreamCalculator` class until all tests pass. Ensure that your implementation is sufficiently documented through Javadoc and comments. Write Javadoc and comments while programming, not after programming. If there is an error, for example a division by zero, your implementation should write a line starting with "error: " to the `Writer`.
  Tip: you can use a `BufferedReader` to read lines from the `Reader` and a `PrintWriter` to write lines to the `Writer`. Study the Javadoc of these classes. To process a line like "push 5" you can use the `String.split` method.
- If you don't "flush" the output, the tests do not succeed. Explain why.
- Explain how your `process()` method detects that there is no more input from the Reader.
- Explain if and how you could detect that the `Writer` is closed.
- Most of the tests in `StreamCalculatorTest` do not test normal behavior, but error handling. Explain why this is important.

AP-5.12 Implement a class with a `main` method that uses your `StreamCalculator` to read from `System.in` and write to `System.out`. Run the program and try to compute things, see what happens if you provide invalid input, etc.

Sign off point AP-5.C

To sign off, present the following:

- The results and code for exercises AP-5.11 and AP-5.12.

5.3.2 Recommended exercises

AP-5.13 Make the following exercises from ECK:

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

Tic-Tac-Toe Perfect Strategy

AP-5.14 In AP-5.5 you developed a smart strategy for the Tic-Tac-Toe game. However, this strategy is not always satisfactory, as it thinks only one move ahead. However, it is possible to implement a *perfect strategy*, using a recursive algorithm. If there is a possibility to win the game, this strategy will lead to victory. If there is a possibility for a draw, the strategy will never lead to a loss. First we define some terminology, before describing the algorithm:

- A move is *winning* (for the player whose turn it is) if after the move one of the following situations applies:
  - The player has won;
  - If the opponent can still make a move, then all possible moves of the opponent will make the opponent *lose*.
- A move is *losing* (for the player whose turn it is) if after the move the opponent can make a move such that the opponent *wins*.
- In all other cases, the move is *neutral*.

Using this terminology, we describe an algorithm that given a game situation and the knowledge whose turn it it, returns the best move for this player, together with an estimate of the *quality* of this move (winning, losing, or neutral):

- The best move so far, and its quality is stored using auxiliary variables, for example `bestMove` and `bestQual`;
- For every possible move, the following happens:
  1. The player whose turn it is, does the move (as a tryout)
  2. Next the quality `qual` of the move will be determined
     (a) If the player whose turn it is, now has won, the `qual` will be *winning*.
     (b) If the opponent now has not won, the best move of the *opponent* will be determined by a recursive call to the algorithm. Call the result of this call `oppQual`.
     - If `oppQual` is "winning", then `qual` will be "losing".
     - `oppQual` is "losing", then `qual` will be "winning".
     - Otherwise, `qual` will be "neutral".
  3. If `qual` is better than the current value of `bestQual`, then `bestQual` will be replaced by `qual`, and the value of `bestMove` will be replaced by the current move.
  4. Finally, the tried move will be removed from the list of possible moves, so the remaining ones can be tried.
- The result of the algorithm is the values of `bestMove` and `bestQual`.

The algorithm can be made more *non-deterministic* by making a random choice in step 3 whether to replace the `bestMove` by another move of the same quality, or not.

Figure 5.1 gives an overview of all possible moves of the X-player that can lead to a 'perfect' strategy. In the figure, the X-player first chooses the left upper corner, and the steps can be followed by (recursively) zooming in on the position chosen by the O-player.

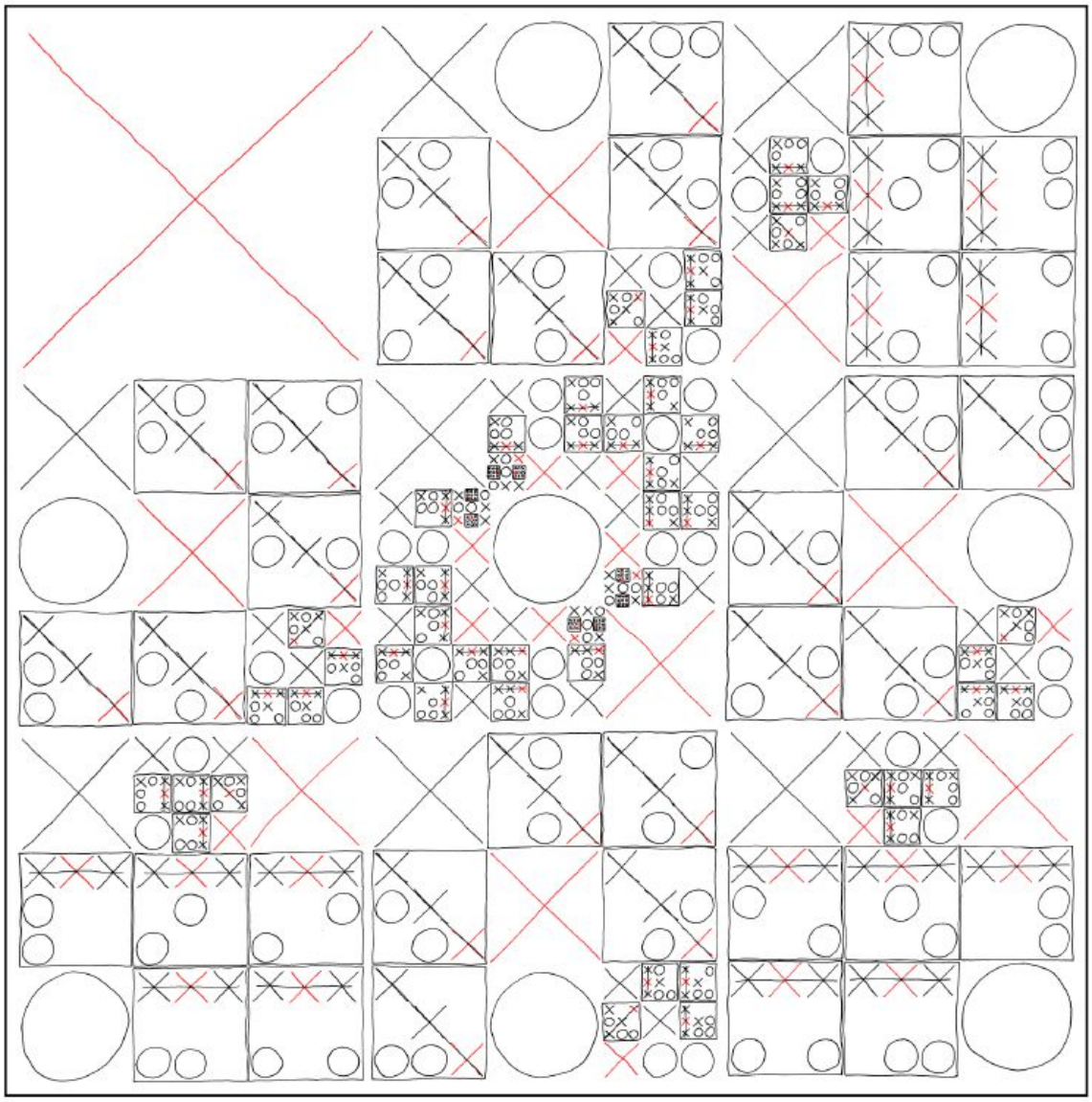


Figure 5.1: Possible perfect moves for player X.

Specify and implement a class `ss.tictactoe.ai.PerfectStrategy` that implements the `Strategy` interface. Method `determineMove` should use the algorithm described above. The name of this strategy is "`Perfect`".

Would this algorithm also work for a game of chess? Why, or why not?

Week 6

Week 6

6.1 Overview

6.1.1 Contents of This Week

Design This week contains no session on design.

Programming This week the following topics will be discussed:

- Principles of programming with threads:
  - Thread creation and thread life cycle
  - Race conditions
  - Synchronisation between threads; locks
  - The wait-notify mechanism
- Kick-off for the programming project

6.1.2 Mandatory Presence

There are no mandatory activities this week.

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

6.1.4 Materials for this Week

Programming

Lecture
- ECK, Sections 12.1–12.3
- The week 6 topic videos

Practicals The following files are provided on Canvas:

6.2 Advanced Programming

6.2.1 Practical exercises

Password dictionary attack

Suppose that the fictitious company BigSimpleCorp has a simple online service that authenticates its users through passwords. Somehow, the password file for their online service was leaked to pastebin (http://pastebin.com/h0etcvvS). Although the passwords are not stored as plain text, the only protection is that the passwords appear to be hashed. A snippet of the password file:

```
alice: c0af77cf8294ff93a5cdb2963ca9f038
allen: c6009f08fc5fc6385f1ea1f5840e179f
```

As stated above, the passwords are stored using a cryptographic hash function. Cryptographic hash functions are one-way: it is easy to compute the hash for a certain input, but obtaining the original input given only a hash is (by design) very (very) hard. The reason for using hash functions for storing passwords is straightforward: it is easy to check whether a password is correct (by hashing a password and comparing it with the hash that is stored), but in case an unauthorized person gets access to the list, the passwords are not as easy to obtain. However, *dictionary attacks* are still possible: given the hashes of a large number of common words (e.g., from a dictionary), one can check whether the hashes in a password file match any of the hashes of words in the dictionary. In the following few exercises, you will build such a dictionary attack step-by-step. In the process, you will use some of the container structure from the `java.util.Collection` hierarchy (in particular the Map). You are provided with a incomplete file for this assignment called `ss.week6.dictionaryattack.DictionaryAttack`. You will complete it during the exercises.

AP-6.1 Fill the body of the `readPasswords()` method of the `DictionaryAttack` class. The JavaDoc for this method describes what it should do.

Hint: keep the parsing of the password file simple. You could, for example, simply use the `String.split()` method for each line (with which argument?) to obtain the username and password hash.

From the length of the string that represents the hashed password we can make a guess as for which hash algorithm has been used. The hex-encoded string is 32 characters long, which means it represents 16 bytes ($16 \times 8 = 128$ bits). This matches the output of the MD5 algorithm, which is 128 bits[1].

AP-6.2 Implement the `getPasswordHash()` method of the `DictionaryAttack` class. This method should take a password (a String), compute the MD5 hash of it, convert the resulting byte-array to a Hex-encoded string and return this. The JavaDoc for the method contains more information. Java provides support for cryptographic hashes in the class `java.security.MessageDigest`. See Oracle's documentation on `MessageDigest` at https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/security/MessageDigest.html. Check your implementation with `ss.week6.DictionaryAttackTest`. The hexadecimal encoding of the MD5 hash of the password "password" is `5f4dcc3b5aa765d61d8327deb882cf99`. Based on visual inspection of the password file, which users appear to have chosen "password" as their password? Note that the `MessageDigest` methods can throw exceptions. Decide for yourself how to handle those.

AP-6.3 Implement the `checkPassword()` method. See the Javadoc for details, and use the `getPasswordHash()` method you have implemented. Check your implementation with the `DictionaryAttackTest` JUNIT tests that can be found on Canvas.

AP-6.4 Next, compute a dictionary of password hashes by implementing the `addToHashDictionary()` method of the `DictionaryAttack` class. As described in the JavaDoc documentation, this method should read a file and compute the (MD5) hash of each line (use your `getPasswordHash()` method for this). It should then fill a dictionary (a `Map` interface implementation) that can map a password hash back to the original password. Search on the Internet for "most common passwords" and you will find, for example, a list of the 25 most common passwords. Use this list to populate a small dictionary to start your tests.

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

AP-6.5 With the building blocks in place, now implement the dictionary attack in the `doDictionaryAttack()` method of the `DictionaryAttack` class. This method should use the two `Map` instance variables. Whenever a password is found, print out both the username and the password. With the dictionary file you created based on the most common passwords, you should be already able to find several passwords. To find even more passwords, you can use a larger word list. For example, you can use the wordlist at `http://www.cs.duke.edu/~ola/ap/linuxwords` to make your dictionary larger.

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

AP-6.6 Someone tells you that the user Alice has used a simple four letter lowercase word as her password. How many tries would it take on average to find her password when using a brute-force approach?

Although the following assignment is marked optional, do read through it and the text that follows it.

AP-6.7 (Optional) Write a program that finds Alice's four letter password by trying all possible combinations (even though you may have found the password already using your dictionary attack). Measure how much time it takes on average per attempt (use `System.currentTimeMillis()`). What other passwords can you find? Try increasing the password length. How does that change the running-time? How many passwords are you able to recover?

Exercise AP-6.7 illustrates how relatively easy it is to recover passwords from such a password file. Suppose it takes on average 0.000315 milliseconds to try one password. Extrapolating this to 8-letter lowercase passwords (that is, 104413532288 possible passwords) means that even with a naive implementation, such a password could be recovered within 9 hours.

Based on this, it should be clear that the approach used at BigSimpleCorp is not the way to go. Even more so considering GPUs and dedicated hardware are capable of brute-forcing such MD5 hashes orders of magnitudes faster.[2] Instead, it is better to use a salt[3] and to use hashing functions specifically designed for the purpose of storing passwords. Such functions are designed such that they are fast enough for normal usage but too slow to effectively brute force. Examples of such hashing functions are PBKDF2, bcrypt, and scrypt.

Sign off point AP-6.A

To sign off, present the following:

- The results and code for exercises AP-6.1 to AP-6.6.

Synchronisation

Study This!

This is a great point to read ECK Section 12.1 about threads, race conditions, and synchronizing. Also make sure you watch the related videos from Canvas.

In the classes and methods seen so far, we did not bother about the functioning of a method when methods on a single object are executed via several threads simultaneously. However, if we design classes to function correctly when accessed by multiple threads at once, then we should make them *thread safe*.

The main problem that we get with multiple threads is called a **race condition** and occurs when multiple threads access the same object simultaneously in a thread-unsafe way, i.e., without protection. While the classical example is that of a bank account, we will consider a pepperoni pizza.

AP-6.8 Study the classes `ss.week6.pizza.Pizza` and `ss.week6.pizza.PizzaChef`. Notice how the `PizzaChef` class is a `Runnable`. That means that a `PizzaChef` object can be used to start a `Thread`.

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

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

1. Create a class `OnePizzaPizzeria` with a single `main` method. This pizzeria is dedicated to making a single awesome pizza. Let the `main` method create a `Pizza` object, as well as two (or more) dedicated `PizzaChef` chefs.
2. For every chef, create a separate `Thread` using each `PizzaChef` object as the parameter of the constructor. Then run `Thread#start()` on all threads. Afterwards, print the pizza to the standard output to see the amount of pepperoni on the pizza.
3. Run the program. You will likely see the output "pizza with 0 pepperoni". The reason is that after creating two new threads, the original thread simply continues printing the current amount of pepperoni toppings before the new threads have had a chance to add pepperoni.
4. Right before the print statement and after the two start statements, run `Thread#join()` on the created threads. Read the Javadoc of `Thread#join`. (If you hold your mouse over the method name, you can see the Javadoc.)
5. You will need to catch the `InterruptedException`. In the ECK book you have read what this means. What is the reason this exception could be thrown and what is the best way to deal with this exception?
6. The output you now get should be a pizza with any amount of toppings between 0 and 10000. Run the program a few times to get different results. Explain in detail how this is possible, write your answer down in detail.

You can use the **`synchronized`** keyword to let a thread synchronize on an object. JAVA guarantees that only one thread can synchronize on that object at the same time. The syntax is:

```
synchronized(obj) {
    ...
}
```

The code inside the block is called the **critical section**. If you include the keyword **`synchronized`** in the method signature, this is syntactic sugar for wrapping the entire method in `synchronized(this) {...}`.

AP-6.9 Make the `Pizza#addPepperoni` method **thread-safe** with the **`synchronized`** keyword. Run the program again. Is this the result you expected?

Consider a program with multiple `Pizza` objects. Can two different threads update two different `Pizza` objects simultaneously, or only one at a time?

The `java.util.concurrent.locks` library declares an interface `Lock` with methods `lock()` and `unlock()` for synchronisation. A synchronised block can be replaced by declaring a `Lock`, preceding the block by a call to `lock()`, and finishing the block by a call to `unlock()`.

AP-6.10 Study the `Lock` interface from `java.util.concurrent`. The most commonly used implementation of this class is `ReentrantLock`. Answer the following questions:

1. What does it mean for a lock to be reentrant?
2. Is this behaviour different from the `synchronized` statement?
3. What would be advantages of using a `ReentrantLock`?
4. And what would be disadvantages?

AP-6.11 Implement a method `Pizza#addPepperoniLocked` by using a `Lock` to protect the critical section from being used simultaneously by multiple threads.

Change `PizzaChef#run` to use this method instead of `Pizza#addPepperoni`. Run your program again a few times to verify that it works.

Sign off point AP-6.B

To sign off, present the following:

- The results and code for exercises AP-6.8 to AP-6.11.

Study This!

Study ECK Section 12.3.5 about using wait and notify. Also make sure you have watched the video on Monitors from Canvas.

Producer-Consumer with wait and notify

For a proper collaboration of parallel processes, synchronisation alone is sometimes not sufficient. In many cases, processes should wait until it is their *turn* or there are other restrictions on the order in which certain methods may be called.

As an example, we look at a typical *Producer-Consumer pattern*. Producers and consumers execute in parallel, and they communicate via a shared object. Multiple producers and consumers may be operating simultaneously.

AP-6.12 Study the classes `ss.week6.pizza.PizzaCounter` and `ss.week6.pizza.Pizzeria`. Run the `Pizzeria#main` program a few times. Explain what you see.

AP-6.13 You can fix the problem using `wait` and `notify` or `notifyAll` in the `PizzaCounter` class. **As long as** there is no pizza on the counter, the `takePizza` method should **wait**. When a pizza is added to the counter, the waiting thread(s) should be **notified**. These waits and notifies must happen in a critical section with the same lock, that is, in a **`synchronized`**-block on the same object, and the `wait`/`notify`/`notifyAll` methods must be called on that object. Which method is preferable? `notify()` or `notifyAll()`? Why?

Run the program again. The error you got before should not appear again. But now you have another problem! Explain the problem.

AP-6.14 Solve the problem you found in the previous question by using `wait` and `notify` or `notifyAll` again.

In your current implementation, there are many *spurious* notifies. If you have many deliverers, then they might keep waking each other even though no pizza is ready on the counter. If you have many chefs, they might keep waking each other even though there is no space yet on the counter.

Therefore, it can be better to have a more fine-grained notification mechanism, and to wake up only the threads that are waiting for the condition that has just been achieved.

The `java.util.concurrent.locks` library declares a `Condition` interface for this purpose. Each implementation of the `Lock` interface can be associated with multiple conditions. Instead of calling `wait()` on an object, a thread can now `await()` on a condition. When this particular condition is reached, the call to `signal()` on this `Condition` will awake only the threads waiting on this condition.

AP-6.15 Study the documentation of interface `Condition` and implement a class `FinePizzaCounter` that implements a pizza counter that uses a `Lock` and multiple `Condition`s, instead of the `synchronized` keyword and methods `wait` and `notify()` or `notifyAll()`. **Be careful not to use `wait` when you mean to use `await`!** The documentation of interface `Condition` gives a producer-consumer example to illustrate the usage of `Condition`.

Don't forget to test your implementation.

Sign off point AP-6.C

To sign off, present the following:

- The results and code for exercises AP-6.12 to AP-6.15.

AP-6.16 Specify the `PizzaCounter` class with JML:

- An invariant that reasons about the number of pizzas in the `pizzas` list.
- Postconditions for `addPizza`. Ignoring multithreading: all pizzas that were there before will still be there after the method call; if there is room, the given pizza will be on the counter.
- Postconditions for `takePizza`. Ignoring multithreading, this would be something about the return value, something about the number of pizzas before/after, and that all pizzas that are on the counter afterwards were all on the counter before
- What happens to these postconditions when you take multithreading into account?

Sign off point AP-6.D

To sign off, present the following:

- The specifications for exercise AP-6.16

AP-6.17 Study the `ss.week6.ListAdder` class. Run the `main` method several times.

- What do you observe? Explain this observation in detail.
- Adding **`synchronized`** to the `addNumber` method signature does not solve the issue. Why not?
- What is the correct way to solve this problem?

Sign off point AP-6.E

To sign off, present the following:

- Your observations of exercise AP-6.17

6.2.2 Recommended exercises

Concurrency Theory

AP-6.18 Consider the following program fragment:

```
package ss.week6;

public class ConcatThread extends Thread {
    private static String text = ""; // global variable
    private String toe;

    public ConcatThread(String toeArg) {
        this.toe = toeArg;
    }

    public void run() {
        text = text.concat(toe);
    }

    public static void main(String[] args) {
        (new ConcatThread("one;")).start();
        (new ConcatThread("two;")).start();
    }
}
```

1. Which lines of `ConcatThread` are a critical section, and why?
2. What are the possible values of `text` after executing `ConcatThread`? Explain how these results can be achieved.
3. Adapt the method `run` of `ConcatThread` in order for the program to always terminate, ensuring that `text` has either the value `"one;two;"` or `"two;one;"`.
4. Adapt the method `run` of `ConcatThread` in such a way that `text` always has the value `"one;two;"`.

AP-6.19 In this exercise, we develop a multithreaded version of the well-known quick sort algorithm. Consider the class `QuickSort`.

```
package ss.week6;


public class QuickSort {
    public static void qsort(int[] a) {
        qsort(a, 0, a.length - 1);
    }
    public static void qsort(int[] a, int first, int last) {
        if (first < last) {
```

```
            int position = partition(a, first, last);
            qsort(a, first, position - 1);
            qsort(a, position + 1, last);
        }
    }
    public static int partition(int[] a, int first, int last) {

        int mid = (first + last) / 2;
        int pivot = a[mid];
        swap(a, mid, last); // put pivot at the end of the array
        int pi = first;
        int i = first;
        while (i != last) {
            if (a[i] < pivot) {
                swap(a, pi, i);
                pi++;
            }
            i++;
        }
        swap(a, pi, last); // put pivot in its place "in the middle"
        return pi;
    }
    public static void swap(int[] a, int i, int j) {
        int tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
```

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

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

Week 7

Week 7

7.1 Overview

7.1.1 Contents of This Week

Design This week contains no session on design.

Programming This week the following topics will be discussed:

- Network programming:
  - Use of sockets for communication over a network.
  - Networking and multithreading.
  - Framework of a client/server network application, including protocol.

You also make a planning for the programming project and discuss it with a teaching assistant.

7.1.2 Mandatory Presence

During the following activities, your presence is mandatory.

- Programming diagnostic test

7.1.3 Expected Self-Study and Project Work

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

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

7.1.4 Materials for this Week

Programming

Lecture
- ECK, Section 11.4.
- The week 7 topic videos

Practicals The following files are provided on Canvas:

7.2 Advanced Programming

7.2.1 Practical exercises

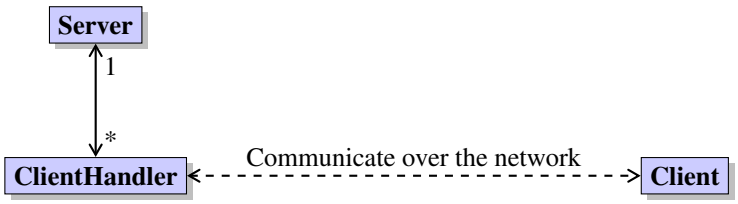
Client-server architecture

Many Java programs are not designed to run standalone on a computer, but as part of a larger networked ecosystem. This section briefly outlines the client-server architecture, which enables Java applications to communicate across the internet using so-called sockets. JAVA implements sockets using the `Socket` class.

The underlying technology used to implement connections on the internet is the TCP/IP protocol, which is one of the standard technologies behind the Internet. In TCP/IP, computers have an IP address and connect through TCP ports. Each TCP/IP connection between two programs is uniquely identified by the IP addresses and TCP ports of the two sides. One of the two parties acts as the server, a program that accepts connections on a TCP port. The other party is the client, a program that initiates a connection to the server. Usually the server is designed to serve many clients.

A basic JAVA client-server architecture consists of at least the following three classes:

- the `Server`, which is responsible for accepting connections and creating a `ClientHandler` for each new connection.
- the `ClientHandler`, which is responsible for a single connection with a client. Access to the underlying connection and the input/output streams is encapsulated by the `ClientHandler`. Usually a thread is started by the `ClientHandler` that continuously reads data from the socket. The `ClientHandler` also implements methods to send data to the client.
- the `Client`, which is responsible for a single connection with a server. The `Client` object also encapsulates the underlying connection.



Depending on the requirements of the application, there can be many more classes. For example, how will a `ClientHandler` process the received data? In the simplest designs, a message received from a client is simply handled by the `ClientHandler` itself, calling appropriate methods of the shared `Server` object if needed. However, this may violate separation of concerns, as the `Server` object now has the responsibility to accept connections and to provide other functionality to the `ClientHandler`s. Maybe the functionality of the server further divided over different classes? Similarly, the `ClientHandler` itself might have multiple responsibilities that could be spread over multiple classes to improve the coherence and separation of concern, such as dealing with encryption, decoding/parsing and encoding/generating the communication protocol, etc. Thus, multiple classes might be needed there as well to adhere to the single responsibility principle.

To wait for connections in JAVA, a server uses a special `ServerSocket` class. Each `ServerSocket` is associated with a TCP port which is identified by a number between 0 and 65535. The server waits for connections to that port by invoking the `accept` method of a `ServerSocket`. If the port is already in use, this results in an `IOException`. Each time `accept` is used, JAVA waits for one connection to the server and returns the `Socket` that represents the new connection. The server then creates a new `ClientHandler` object to handle the new connection. By repeatedly calling the `accept` method and creating `ClientHandler` objects, the server can handle multiple connections.

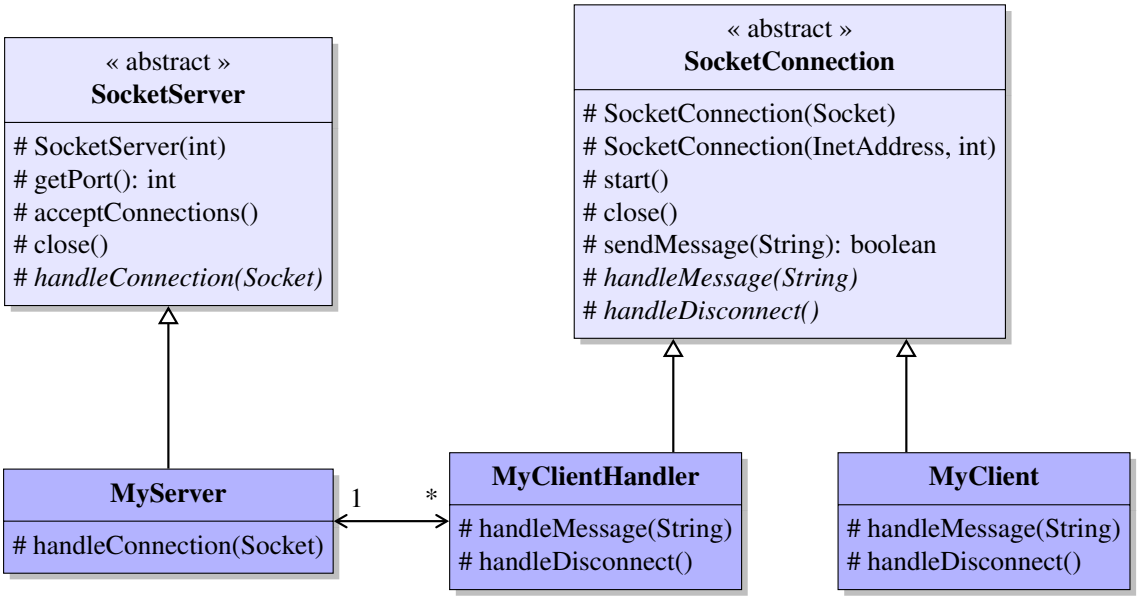
A client connects to the server by constructing a `Socket` object with the IP address of the server and the correct TCP port number. For connections to the same computer, one can use the address of localhost, which in Java is obtained with `InetAddress.getLocalHost()`.

For example: `new Socket(InetAddress.getLocalHost(), 12345)` connects to TCP port 12345 of the same computer. Similarly, `InetAddress.getByName("utwente.nl")` can used to get the IP address of a hostname.

Once a connection is established, a `Socket` object has an `InputStream` and an `OutputStream`. Bytes sent to the `OutputStream` arrive (after flushing) at the `InputStream` on the other side of the connection. When the `Socket` object is closed via a call to `close()`, the two associated streams are closed as well.

Module 2 Helper Classes

While using the `Socket` object directly is not necessarily complicated, there are a few technical details that might cause difficulties for beginner programmers. Thus, we provide two predefined helper classes, `SocketServer` and `SocketConnection`:



In the above image, we see five classes:

- The predefined **`abstract`** class `SocketServer` is constructed with a port as parameter. If 0, any free port is used. The actual port number can be obtained by calling `getPort()`. To start accepting new connections, call the `acceptConnections()` method. This method will forever wait for new connections, until `close()` is called from a different thread. The `handleConnection` method is called when there is a new connection. This method must be implemented by subclasses.
- The predefined **`abstract`** class `SocketConnection` is constructed with either the `Socket` (server) or the IP address and TCP port (client). After construction, the `start()` method starts a thread in the background that reads messages from the connection. Whenever a message is received, the `handleMessage(String)` method is called. When the connection ends, the `handleDisconnect()` method is called. These two methods must be implemented by a subclass. The `close()` method ends the connection and `sendMessage(String)` allows sending a message over the connection. This method returns **`true`** if successful, and **`false`** if the connection was lost.
  Every message sent and received by the `SocketConnection` is a line of text.
- The example class `MyServer` implements the `handleConnection` method. All this method does is create a `MyClientHandler` object for the given `Socket` instance.
- The example class `MyClientHandler` implements `handleMessage` and `handleDisconnect`. Furthermore, at some point `SocketConnection#start` must be called to start receiving. This is either done in the `MyClientHandler` constructor or by `MyServer#handleConnection`.
- Similarly, the example class `MyClient` implements `handleMessage` and `handleDisconnect`.

All methods of `SocketServer` and `SocketConnection` are **`protected`** by default; they can be made **`public`** in the subclasses by overriding them and changing the access modifier to **`public`**. This allows the subclasses to control which functionality is exposed to other classes.

Chat Application

For the following exercises you are going to implement a networked chat application. This application will consist of a server and of a client with a TUI. For the communication between the server and the client, you need a communication protocol. This protocol explains the structure and order of the messages that will be sent between the client and the server. The protocol for this exercise is quite straightforward:

- Every message between the client and the server is a line of text. The predefined `SocketConnection` class takes care of this.

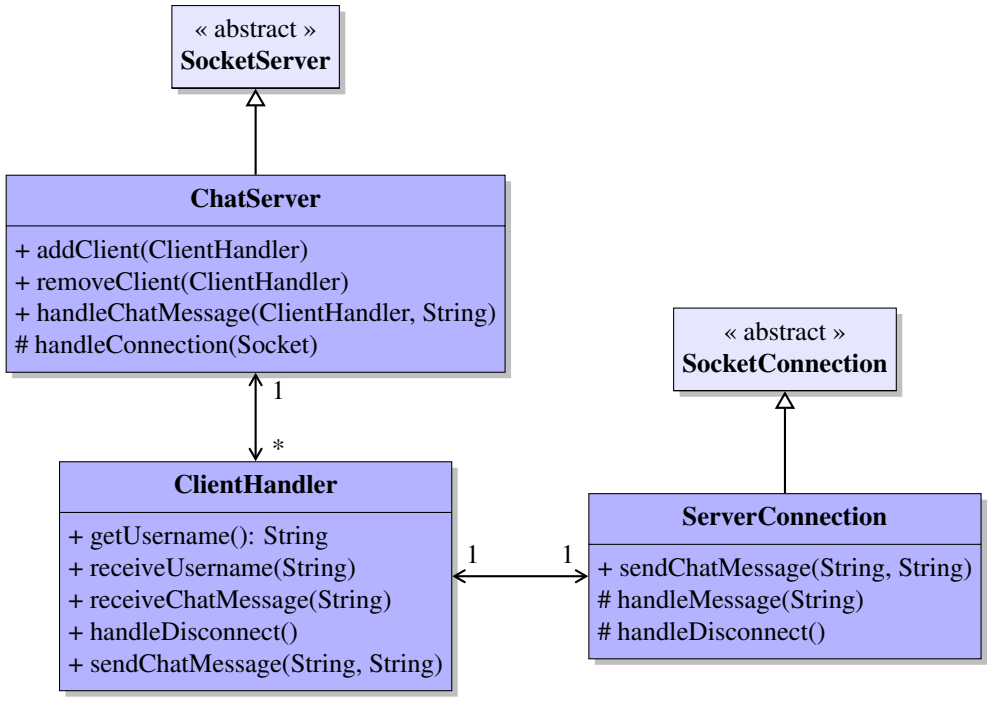


Figure 7.1: Class diagram of the chat server with the most relevant methods.

- The first message from the client to the server is in the format `USER~<name>` and communicates the username of the client. This username may not include the tilde ~ character because the ~ has special meaning in the protocol. Afterwards, the client may not send `USER` messages again.
- All following messages from the client to the server are in the format `SAY~<message>`. Again, to keep the protocol simple, messages may not include a ~.
- All messages from the server to a client are in the format `FROM~<name>~<message>`.
- Any invalid messages are ignored by the server and the client.

Depending on the requirements for your server, you could implement a protocol in different ways. For example if you plan to support (or in the future support) different protocols, you should separate protocol handling code from the rest of your program. You could do that by letting the protocol handling code directly handle incoming and outgoing messages with the streams, and have your protocol-independent code sit on top of that, i.e., your protocol-specific class reads from the stream, decodes the message, and then forwards it to another class that is not protocol-specific; similarly, the rest of the program tells your protocol-specific class about messages that need to be sent and the protocol-specific class handles sending the message. The exact details and abstractions chosen and the interfaces that you define for this depend on how you design your software.

For the chat server and chat client in this exercise, we will separate the protocol specific part from most of the other code.

The general client-server architecture explained before only specifies the minimal responsibilities of the `Server`, `ClientHandler` and `Client` classes. However, this chat server will have to do more than just accept connections. You will need to track online users and forward received messages to online users. Maybe you can envision other responsibilities in the future. In this exercise, we let the main `Server` class handle all responsibilities, but in larger projects you should consider moving different responsibilities to different classes so that your `Server` stays simple. You could also apply design patterns here, such as the Chain-of-responsibility pattern, to handle received messages in the `ClientHandler`.

AP-7.1 In this exercise you will implement the chat server.

First inspect the code that is already available: the `ss.week7.chat.server.ChatServer` class, the `ss.week7.chat.Protocol` class, and the `ss.week7.chat.ServerTest` class.

The `ss.week7.chat.server` package will hold the server-related classes of this program. The classes that you will implement in this exercise will look roughly as in Fig. 7.1.

- The `ChatServer` class has several responsibilities: it waits for connections on a `ServerSocket` and creates a `ServerConnection` object for each new connection. It also keeps track of all chat clients and also forwards received chat messages to all clients.
- The `ServerConnection` class has several responsibilities: it maintains/encapsulates the connection to the client, handling the input/output streams (because it subclasses `SocketConnection`). It also decodes/parses messages from the client according to the protocol, and encodes/generates messages to the client according to the protocol.
- The `ClientHandler` has a clearly defined responsibility: it simply handles the (decoded) messages from the client and forwards chat messages from the `ChatServer`.

Of course, these concerns could be separated with future extensions in mind. In this exercise we keep things relatively simple and only separate protocol and stream handling from the rest of client handling (as opposed to a single `ClientHandler` class extending `SocketConnection`. We will now build the server step by step.

1. First, we need to make sure our server can accept connections and receive messages. For now, create the `ServerConnection` class, and implement the two handlers with simple print statements (`System.out.print...`). Then implement `ChatServer#handleConnection`, which for now should just create a `ServerConnection` object and invoke the `start()` method.
2. Add a `main` method to the server, which asks for a port number and then starts the server on the given port by calling `acceptConnections()`. It might be helpful to print the actual port number to the screen, obtained via `getPort()`, especially if the user chooses port 0.
3. Now you should test your server! An easy way to do this is by installing netcat, which you can find at https://www.nmap.org/download. Then use the command `ncat localhost <port number>` with the port number that your server is running on. You should be able to type whatever you want and see this appear on the console of your server. When you terminate netcat, your server should also display a message from `handleDisconnect()`.
4. The next step is adding the `ClientHandler`. Create the class. For now, just focus on the basics: when `ChatServer` creates a `ServerConnection`, it should also create a `ClientHandler` and the two new objects should have the appropriate fields that reference each other. Make sure everything still compiles.
5. We shall now work on parsing the protocol. Create the two methods `receiveUsername(String)` and `receiveChatMessage(String)`. Then modify `ServerConnection#handleMessage` to parse the message. Use the predefined `Protocol` class and remember to ensure that your program does not crash from invalid messages. Ensure that the right method in the `ClientHandler` is called. Also add `handleDisconnect` to `ClientHandler` and modify the implementation in `ServerConnection`. Remove the temporary print statements from `ServerConnection` and create print statements in the `ClientHandler` methods. Then run the server and netcat again, and check the following things: 1) username messages are properly parsed; 2) chat messages are properly parsed; 3) invalid messages do not result in crashes. Your implementation should also ensure that every `SAY` message before a `USER` message is ignored, and that every `USER` message after an earlier `USER` message is ignored.
6. The next step is connecting the `ClientHandler` to the `ChatServer`. For this, you need to add a reference to the server in the client handler. After the connection with the client is established, you should invoke the `addClient` method of the server. If the client is disconnected, the `removeClient` method should be called. If a chat message is received (by a client that already has a username), then the `handleChatMessage` method should be called. Let the server call `sendChatMessage` of all connected clients. Remove all temporary print statements from `ClientHandler` and instead add them to `ChatServer` when clients are added/removed and messages are received.
7. Finally, we need to encode/generate protocol messages from the server to the client. Implement the `sendChatMessage` method of `ClientHandler` by delegating to `ServerConnection`, and then implement `sendChatMessage` of `ServerConnection` by calling the `sendMessage` method of the superclass with an appropriately encoded `String`. Don't forget to use the constants defined in the `Protocol` class.

At this point, your server should be fully functional. Check this by connecting multiple netcat clients to your server and sending some chat messages.

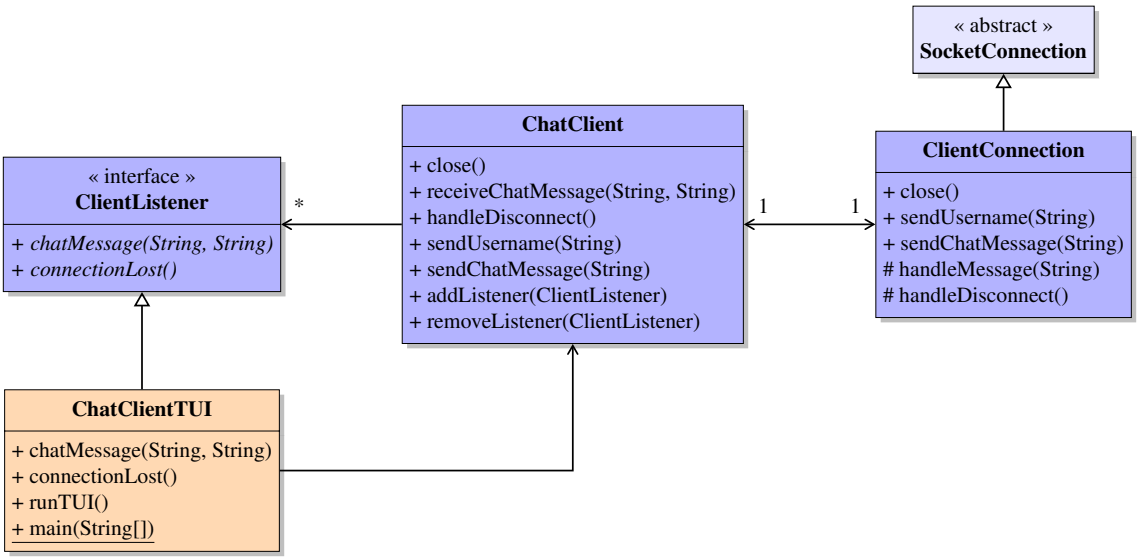


Figure 7.2: Class diagram of the chat client with the most relevant methods. The non-UI classes have no dependencies on the UI class.

You can now also run the test class, `ServerTest` inside the `chat` package for the server and check whether it passes all the tests.

AP-7.2 Unfortunately, your new server has a race condition. Identify the race condition and write a concrete scenario of something that can go wrong. Change your program so it is thread-safe, that is, free of race conditions.

Sign off point AP-7.A

To sign off, present the following:

- The code for exercises AP-7.1 and AP-7.2

Study This!

Go to `https://refactoring.guru/design-patterns/observer` and read about the **Observer** pattern.

AP-7.3 In the following exercise, you will design and implement the client side of the chat application. The client implementation will be separated into two parts: **UI** (User Interface) and **networking**. We apply a *light-weight* version of the Model-View-Controller (MVC) design pattern. The MVC pattern is actually a combination of several design patterns: Observer, Strategy and Composite. In our light-weight version, we only care about separating the M from the VC by using the Observer pattern[1]. Thus, all UI-related code resides in UI classes, while all non-UI classes have no dependencies on the UI classes. In other words, the networking part should not be aware of the existence of a specific user interface. How can you ensure that when the UI is changed, the remaining program does not need to be changed?

AP-7.4 We shall now implement the client! See Fig. 7.2.

1. First, inside the `chat` package, create a new package called `client`.
2. Similar to how you implemented the server, start by creating a class `ClientConnection` and a class `ChatClient`. The `ClientConnection` should be a subclass of `SocketConnection` and offer a constructor that takes an address and a port number. It should implement methods `sendChatMessage` and `sendUsername` and its `handleMessage` and `handleDisconnect` methods should call appropriate methods of the `ChatClient` object.

[1] Normally, the UI would be further separated into components such as View-Client or View-Presenter, etc., but for simplicity, we are keeping it unified here. Normally, the View is internally organized using the **Composite** pattern and the Controller is the **Strategy** of the View.

3. The next step is creating *most* of `ChatClient`. Implement a constructor that takes an address and a port number, and which creates a `ClientConnection` object. Implement `close`, `sendUsername` and `sendChatMessage` by delegating to `ClientConnection`.
4. Before we apply the Listener pattern (as explained below), we first write the parts of `ChatClientTUI` needed to make a connection, so that we know that that part works. Create the class and implement a `main` method by creating a new `ChatClientTUI` object and just calling its `runTUI` method, which you now also create. Implement `runTUI` by asking the user for a server and a port, then creating a suitable `ChatClient` object. Then, the program should ask the user for a username, which should be sent to the server. Then, the program should ask the user for messages to send. When the user enters the message "quit", the program should first terminate the connection (with `close`) and then exit.
   Don't forget to start the `ClientConnection` in the client TUI.
5. Test your client using the server you made earlier. You are now able to send chat messages to the server. Make sure to test your client for robustness against user input. You can also use netcat to act as a server: `ncat -l 1234` to listen on port 1234.
6. The remaining task is to be notified when the client receives a message from the server, which can be achieved using the Listener pattern. In this pattern, classes implementing the Listener interface are "notified" of events by the "publisher" class, `ChatClient`. The publisher maintains a collection of listeners and provides methods `addListener` and `removeListener` to modify this collection. When an event occurs, all registered listeners are notified.
   (a) Create the `ClientListener` interface.
   (b) Add an appropriate field for the listeners to `ChatClient` and implement the `addListener` and `removeListener` methods.
   (c) Implement `receiveChatMessage` and `handleDisconnect` by notifying *all* listeners.
   (d) Now make `ChatClientTUI` implement `ClientListener`.
   (e) Implement the two methods appropriately: if a message is received, it should be printed to the screen. If the connection is lost, the program should terminate.

   Now your UI part of the client should be complete. Run the server and the client and test the Chat Application together with your partner. In fact, you should be able to chat with all other students on a shared server! Give it a try!

Sign off point AP-7.B

To sign off, present the following:

- The code for exercises AP-7.3 and AP-7.4

AP-7.5 Maybe you want to keep logs of your chats? Add another class `LogListener` that implements `ClientListener` and writes all received messages to a file. Modify `ChatClientTUI` to automatically create a `LogListener` which writes all received chat messages to a file using the syntax:

```
<username> said: "<message>"
```

Remember to close the file when the connection is lost.

AP-7.6 Now it is time to add a System Test for our chat application. To do that, write a testing report using the template provided in Appendix A.2. In this report you should:

- Assign a name to the test and offer a succinct description of the test's scope or the feature under test.
- Provide detailed, step-by-step instructions that a tester should follow to execute the test.
- Describe the expected behavior.
- Give your testing results obtained by executing your own instructions.
- Note: you should not write code.

Sign off point AP-7.C

To sign off, present the following:

- The code for exercise AP-7.5 and the system test for exercise AP-7.6

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

7.2.2 Recommended exercises

Confusing confidentiality and integrity

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

AP-7.7 A common security mistake is to confuse confidentiality and integrity. In other words, some (real-world) implementations assume that encryption is enough to also provide integrity. There have been systems for example that used simple unauthenticated encryption to protect (browser) cookies containing session information. In the following exercise you will experience some of the problems with such an approach.

You are provided a `BadCookieCrypto` class. This class has two methods:

- `public String createCookie()`
  This method creates an encrypted cookie (encoded in Base64) containing (among others) the authorizations of the user in the browser session.
- `public boolean isAdmin(String cookie)`
  This method accepts a string containing a base64 encoded ciphertext representing the cookie. The method decrypts the cookie and determines whether the user presenting the cookie has admin rights. If so, it returns true, otherwise false.

This is what you should know about the encryption scheme: the `BadCookieCrypto` class uses CTR mode (see wikipedia) with a random IV together with an AES blockcipher and a random (64 bit) nonce and 64 bit counter. As wikipedia will tell you, in CTR mode, encryption boils down to XORing a plaintext block with the encryption of the nonce concatenated with the block counter. *So in essence, the ciphertext byte array is the result of a byte-wise XOR operation of a cipherstream and the plaintext.*

In addition, you know that the plaintext is of variable length, but it always ends with the string ";admin=N".

You now have enough information to be able to manipulate the ciphertext in such a way that the cookie token will allow a user presenting the cookie will have admin access. All without editing the `BadCookieCrypto` class!

Consider the following code and remember that the XOR operator in Java is "^".

```
byte c = 0x60 ^ 0x65;
byte b = c ^ 0x60;
```

What is the value of `b`?

Write a program that instantiates the `BadCookieCrypto` class, calls the `createCookie()` method to produce an encrypted cookie, and manipulates the the ciphertext in such a way that the `BadCookieCrypto` class is tricked into accepting the manipulated ciphertext as a valid admin cookie.
Hint: to turn a character into a byte you can use typecasting (e.g., the result of `(byte)'a'` will be a byte of value 0x60).

Appendix A

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 inferface 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 unaccessible 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 IP-3.12. Important things to test are functionalities that are high-risk or essential, such has handling passwords correctly. Most functions that contain more than a few lines or that contain branches, other method calls, or loops, are sufficiently complex to require testing. Testing is done not to convince the programmer that the implementation is correct, but to convince others (including their future self) that the implementation is correct.

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

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

A.3 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 system test should have a title, a description, a step-by-step concrete list of instructions for the tester, desired correct results, potential incorrect results, and a test report with the actual results.

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 is 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 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 correctly[1]. 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.

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

- *Loop invariants*
  Loop invariants are properties that specify properties that hold whenever a loop (i.e. a while- or a 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.

Appendix B

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.

- 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.

```
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`

- 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 ∀ and ∃ are allowed in JML specifications, using keywords `\forall` and `\exists`.

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

```
//@ 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).

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

Specialized JML expressions

- **\max** and **\min**
  The `\max` and `\min` predicates have the same form as `\forall` and `\exists`, except that the value term is now numeric, so that maximum and minimum are meaningful concepts. You might ask, for example, for the maximum and minimum values of an array:
  `(\max int i; 0 <= i < a.length; a[i])`
  The value of the `\max` expression is a number that is at least as large as all the elements being considered and is equal to at least one of them. E.g. if `a = [1,2,10]` this expression will evaluate to 10.
  The range provided **must not** be empty.

- **\sum** and **\product**
  The `\sum` quantifier adds up all the values of the value term for which the range term is true. For example, the sum of all the elements in an array a would be expressed as:
  `(\sum int i; 0 <= i < a.length; a[i])`
  This expression returns the numerical sum of the array `a`. E.g. if `a = [1,2,3,4]` this expression will evaluate to 10.
  `(\product int i; 0 <= i < a.length; a[i])`
  This expression returns the numerical product of the array `a`. E.g. if `a = [1,2,3,4]` this expression will evaluate to 24. If the range predicate is empty (always false) then the sum is 0 and the product is 1.

- **\num_of**
  A final quantified expression is `\num_of` which counts the number of times the boolean value term is true when the range term is also true. For example:
  `\(num_of int i; 0 <= i < a.length; a[i] == 0)`
  counts the number of elements of the array that are 0. E.g. if `a = [0,1,2,0,3,4]` this expression will evaluate to 2, because there are two elements which equal zero.
  If the range is empty the value of the expression is 0.

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 invariant[2] is a predicate over the object state that holds in all *visible* states of an object. A visible state of an object is defined to be any state in which a method call to the object either starts or terminates. Thus, an invariant I is implicitly added as a precondition and a postcondition to every method in the class. In addition, also the post-states of the constructor are visible states, thus any constructor has to ensure that the invariant is established.

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

Example 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.*

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.

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

```
package ss.jml;

public class StudentWithInv {

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

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

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

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

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

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

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

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


}
```

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

```
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

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 <= i && i <= 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 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