Poul Klausen

JAVA 20

About the system development process

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 20: ABOUT THE SYSTEM DEVELOPMENT PROCESS

SOFTWARE DEVELOPMENT

Java 20: About the system development process: Software Development
1st edition

ISBN 978-87-403-2669-7
Peer review by Ove Thomsen, EA Dania

CONTENTS

	Foreword	**6**
1	**Introduction**	**8**
2	**System development methods**	**10**
2.1	Code and test	10
2.2	The waterfall model	11
2.3	Unified Process	12
2.4	SCRUM	19
2.5	Extreme Programming	29
3	**Test-Driven Development**	**32**
4	**Design patterns**	**40**
5	**Refactoring and other**	**43**
5.1	Refactoring	44
5.2	Log files	45

6	**Poker**	**47**
6.1	Vision	47
6.2	Analysis of poker	48
6.3	The process	57
6.4	Iteration 1	60
6.5	Iteration 2	64
6.6	Iteration 3	80
6.7	Iteration 4	89
6.8	Iteration 5	97
6.9	Iteration 6	100
6.10	Iteration 7	102
6.11	Iteration 8	104
6.12	Iteration 9	108

FOREWORD

It is the last book in this series of books on software development in Java. The book deals with system development focusing on the process and primarily in relation to major system development projects. In addition, the book gives an introduction to three concrete system development methods: *Unified Process*, *SCRUM* and *Extreme Programming*. In addition, a number of other activities and techniques are described that are part of modern system development, regardless of whether one uses a method. After you have read these 20 books on system development in Java and solved all the exercises and tasks, you may not be able to call you a professional Java developer. This can only lead to by experience, but you have a good foundation and you are well-dressed to join a professional team of software developers. You never graduate as a software developer, and there are always new versions of products and methods that you have to learn, but I hope these books can provide a good basis. Everything is not included either because there was no room or because it was forgotten and something else might even be deprecated, but maybe a future release may remedy some of it, but if it happens, I will end up at the same place, where something did not come along and something else is deprecated. Software development does never stop.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in Java. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples and exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code to the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with one or two larger sample programs, which focus primarily is on process and an explanation of how the program is written. On the other hand appears the code only to a limited extent – if at all – and the reader should instead study the finished program code perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

The programming language is, as mentioned Java, and besides the books use the following products:

1. NetBeans as IDE for application development
2. MySQL to the extent there is a need for a database server (from the book Java 6 onwards)
3. GlassFish as a web server and application server (from the book Java 11 onwards)
4. Android Studio, that is a tool to development of apps to mobil phones

It is products that are free of charge and free to install, and there is even talk about products, where the installation is progressing all by itself and without major efforts and challenges. In addition, there are on the web detailed installation instructions for all the three products. The products are available on Windows and Linux, and it therefore plays no special role if you use Linux or Windows.

All sample programs are developed and tested on machines running Linux. In fact, it plays no major role, as both Java and other products work in exactly the same way whether the platform is one or the other. Some places will be in the books where you could see that the platform is Linux, and this applies primarily commands that concerning the file system. Otherwise it has no meaning to the reader that the programs are developed on a Linux machine, and they can immediately also run under Windows unless a program refers to the file system where it may be necessary to change the name of a file.

Finally a little about what the books are not. It is not “how to write” or for that matter reference manuals in Java, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

All the previous books in this series have been about system development and how to write programs using the Java programming language, and if I look backwards it's probably the last that fills most, and it's actually not fair because if you're are going to be a good software developer, you should strive to free yourself from the tools and instead focus on the methods and principles common to all system development tasks. However, it is not so easy, because in the end, one must necessarily become concrete and learn how to work with the technologies and tools needed to implement solutions practices. In fact, it is a bit of a dilemma in software development, where you have always been emphasized the importance of learning good system development methods and focusing on analysis, design and patterns, and where the technology and programming in some language was secondary. As mentioned above, the intentions of this series of books were to write about software development, but with the emphasis on the process, and perhaps the final examples should have been more important than the case is. Much of the foregoing has focused on Java and details, but maybe it does not matter, because that's what it's all about.

We have gradually developed software for over 50 years, and during that time many new programming languages have been developed, many different development tools, and several system development methods and other guidelines for developing a program has also been developed, so in principle, it should be easier to develop a program today than before. It's also – you simply have better tools than before, but conversely, the programs you are developing today are often much bigger and more complex than before, and also they have a greater diversity. For example, there is a big difference in writing a PC application and a web application, and again it is different to write an app to a mobile phone. In particular, the last means that it's hard to be good all the way around and that it is necessary to specialize as software developer. Finally, new tools (including programming languages) are constantly being marketed to address specific tasks, so there are always new tools to be used to and use in the daily work as a professional software developer.

Fortunately, there is also something that is stable such as algorithms and other solution techniques, and this is where you have to stay focused. In fact, programming languages and other development tools do not mean much. If you have first learned a programming language, it is easy to change to another language if needed. This is often encountered, typically because different development organizations do not use the same tools and because there are always new ones that might be particularly suitable for the type of tasks the current development department is working on. It is far from the fact that, as a developer, you have influence on the choice of tools, as they are often based on basic decisions in the organization, but you are influenced by the need to be critical and test the tools thoroughly before using

them in the specific projects. Those who have elaborated new tools often promise a lot, and although there are also new and good tools on the market, it is important to be aware that it takes time and thus has the cost to take new tools in use. You have to consider the context of whether it is no better to use the time to become even more adept at using the tools that you already do, and adjust and refine the use of the known tools. Often, new tools promise a lot, but when you first dive into them, it's far from the assumption that the news is so big.

The primary tool in these books has been Java and NetBeans, but there are actually many other tools that relate to the Java world, and all have the same purpose to streamline the software development process. Among other things, I have previously mentioned alternatives to NetBeans (Eclipse is an example), while others can interfere with NetBeans. Many of these tools are good and are widely used in practice, but although there is no room for a presentation in these books, I will mention actual project management tools. There are many each with their advantage, but they are tools that support the implementation of large projects with many project participants. They have facilities to manage large projects where multiple developers work closely together, including controlling versions, so that you can always go back to a previous version, so some developers will not override others' code without being sure that you can recreate the code. It's comprehensive tools, that it takes time to learn to use, and partly are either or tools. These are tools that are implemented on the basis of a decision in the development organization and after extensive investigations, which all have to undertake and apply. On the other hand, they are tools used by all major development organizations, and thus tools that you will certainly meet, and it is a must in the development of large and complex IT solutions.

As a software developer, you must know your tools and development methods, and apart from Java, none of the subjects are presented in these books, but this as the last of the books ends with mentioning a few classic system development methods.

2 SYSTEM DEVELOPMENT METHODS

This chapter is a brief review of three system development methods:

1. Unified Process
2. SCRUM
3. Extreme programming

None of the methods will be discussed in detail, and I would recommend that you find literature that explicitly addresses each of the three methods. In particular, SCRUM is a very popular system development method, although the two others certainly also have their followers.

I have previously in the book Java 7 mentioned the waterfall model as a system development method, and although this method is not considered flexible enough today, it definitely has its uses in development of smaller programs where the risks are small. However, I want to start with an even simpler method which, in fact, is not a method at all, but nevertheless describes a method that is often used in practice.

2.1 CODE AND TEST

It is not a system development method, but it is mentioned here because many programs in practice are developed according to this model, and it also has the right to small programs. The method is quite simple where the developer starts to understand what it is for a program to be written, and then the developer programs some code and tests the result, reprograms, and tests again, and continues until the program is written.

You can say that you actually work without using a method, and it is contrary to everything that has been said and written about system development, but yet the method is widespread, and under certain assumptions it also gives good opinion in practice.

Of course, you can not tackle the code and test process before you understand what it is for a program to be written, and the process therefore starts with a form of analysis where you can clarify the requirements of the program with the customer. Then you start programming and testing whether the code you have written works according to the requirements, and so proceed with a large number of small steps until the task is solved.

In order for such a simple approach to lead to a successful outcome, there are two prerequisites that must be met:

1. It must be a small project and a project solved by a single person (such as the final examples in these books).
2. The task must be well-defined and there must be no greater uncertainties about how the task should be solved and what the final result should be.

If these prerequisites are present, there is nothing to solve the task after a code and test model, and in reality there may not be much else that makes sense. Here it is important to note that in the real world there are many tasks that just meet the above two assumptions, so code and test programming has its uses and has been used far more frequently than the many books on system development prescribe.

2.2 THE WATERFALL MODEL

It is a system development method that I have already mentioned (Java 7). It used to be one of the most used system development methods, but since the most dated system development method that has existed, and the latter is not really quite fair. Everything I have previously said about the phases of the waterfall model still applies and is part of any other system development method. The problem with the waterfall method is not so much the content of the method but that it used uncritically as a number of phases performed in a certain order is not flexible enough to solve today's large and complex IT systems. Actually, it is not the method that is something wrong with. It is just not suitable as a solution for large and complex systems, especially for projects with a long time horizon, and where the project group consists of several or many system developers.

I will not go into the content of the method and just refer to the book Java 7, but for smaller and well-defined tasks, the method is still interesting and can be used very well. Several have tried to improve the method, primarily by incorporating ideas from other system development methods and one version of the waterfall model can be found in the literature described as the waterfall model with feedback, which is an attempt to formally integrate iterative system development into the waterfall model, what it in the principle does not support. Whatever you do, the waterfall model for large complex IT systems will never be the right system development method. Here, instead, you have to use one of the agile system development methods described below.

2.3 UNIFIED PROCESS

Unified Process is an object-oriented system development method, but in the literature, the method is often termed the *Rational Unified Process*, which is a further development of the original method. It is an iterative system development method, solving the current task by working iteratively towards the solution through 4 phases:

1. Inception
2. Elaboration
3. Construction
4. Transition

One has the greatest success with unified process as a method for larger projects, where the task is relatively well-defined and where the project is associated with more developers. In many ways, the method can be seen as an alternative to the waterfall model, recognizing that this method is not suitable for the development of large and complex IT solutions. It is an iterative method in which the task is solved through a number of iterations, and one can reasonably perceive each iteration as the partial solution implemented after the waterfall method.

The unified process is also an agile system development method, and thus a method where the goal is quickly to deliver value solutions for the customer, and at the same time after customer's testing of the delivered sub-solutions, it can easily and accurately adjust the requirements according to the customer's wishes. All modern systems development methods today are call agile, and it is perhaps not much more than an observation that today's IT systems are far larger and more complex than before.

I do not want to review the method in detail, but I strongly recommend reading a book or something that presents the method and provides instructions on how the method can be used in practice.

As mentioned, the method comprises 4 phases, and each phase is performed as a number of iterations. How many iterations to be in the individual phases depend on how complex the task is, but the whole idea is, that each iteration should not take too long. It is recommended that an iteration should have a deadline of not more than a few weeks, but conversely, iteration should not be too short and should at least take several weeks. Before starting an iteration, you have to define which tasks are to be solved. These tasks depend on the phase, but in principle, each iteration can include tasks in analysis, design, programming and test, however, such that the weight changes through the course, where an iteration during elaboration largely focuses on analysis and design while the tasks under construction at most emphasis on programming and test.

Inception

This phase will usually consist only of a single iteration. It is a start-up phase where you must analyze what it is for a task to be solved. It is not the goal to prepare a completed requirement specification, but inception is a short phase in which you analyze critical requirements and determine the basic visions of the system. You should not try to make a detailed list of as many system requirements as possible. A risk analysis for the development of the system must be carried out and a decision must be made whether or not to complete the project. The objective is:

- to gain an understanding of what it is for a system to be developed
- describing the most important functions, and often as use cases
- to develop an overall design of the system's architecture
- to develop a development plan for the subsequent phases, including identifying the project's costs and risks

Any system development starts with an idea that is presented to the person(s) to solve the task. During the inception phase there will be relatively few people. The system development project starts with the presentation of the idea, and the phase will provide guidelines for how the system development group comes from idea to the completed solution that the task manager (the customer) can apply.

The first phase is a clearing phase and is a short phase and is typically performed as a single iteration. How long does of course depend entirely on the specific task, and maybe the phase lasts only a few days, and I find it hard to imagine an inception phase that spans more than a few weeks. During the start, the project team must clarify the following:

- What is the task going on?
- What is the technical platform?
- What risk factors are there?
- What is the time horizon?
- What resources are required?
- How is the project plan?
- What are the criteria for the task being solved?

The result of the inception will typically be:

- A task formulation
- A requirement specification
- One or more prototypes
- Attachments
- Conclusion

A task formulation is a text document organized in relation to the relevant of the above headings. The aim is to keep the assignment form brief and accurate, and the task formulation is perceived as a document that is continuously updated after each of the following iterations.

As mentioned, it is not a goal to prepare a final requirement specification during the inception phase, but you will start it. Often you choose one or a few of the issues that the system will solve and start formulating the requirements. This is typically done in the form of use cases, and it is even said that the unified process is use case driven. During start-up, focus will be on the most central issues and the issues with the greatest risks.

The task formulation may be supplemented by one or more prototypes that should indicate to the task manager the important decision regarding the finished project's user interface. Often it is a good idea to have important decisions regarding the user interface in place before continuing with the next phases.

Finally, the inception may include attachments or references to sources that are important for the following system development.

The inception phase concludes with a conclusion where the system developers emphasize the consequences of continuing the project, and possibly highlighting proposals for adjustments. If the result becomes a decision that the project is to proceed, a project plan is prepared, including a planning of the first iterations during elaboration.

Elaboration

In this phase, one or more iterations are performed (and usually there will be more and maybe many), in which it will develop a part of the overall system in an iteration. One can thus think of an iteration as a mini development project, and the result will basically be a program (a product) that the task manager can test. One can think of an iteration as a development process that brings the entire project from one state to another.

An iteration should take shortly in time, but vice versa, it should also be so extensive that the customer experiences progress. An iteration length is determined of the current

project, but the time frame will typically be from 2 to a few weeks. Of course, the number of iterations will also be determined by the scope of the project, and in small projects, there will in principle only be a single iteration, but if you use a unified process, there will usually be more.

As mentioned, an iteration is a development process, and it will consist of

1. analysis
2. design
3. programming
4. test

It is not a fact that you in an individual iteration follow a waterfall model, but more typically you will work iteratively, where you work with analysis, design, programming and then return to some analysis tasks.

The result of an iteration is:

- The project is in a stable state, which can be presented to the task manager or customer. It will usually be a software product that the task testers can test and comment on.
- An updated version of the task formulation with regard to decisions taken in connection with the implementation of this iteration, as well as an extension of a section with important results for the current iteration.
- An updated version of requirement specification based on the analysis performed during the iteration.
- A conclusion that includes the customer's remarks after testing the product and the developers' recommendations regarding the continued system development.
- Adjustment of project plan, including planning of next or the few next iterations.

Looking at the above, it may seem that you work without having a schedule and decides what should happen as you move forward in the project. Such, of course, you can not develop IT systems, because the task manager or customer must know when the system is expected to be delivered and what it will cost, but vice versa, it should be noted that when unified process continuously adjusts the requirements and continuously plans iterations, it is exactly, what's means that it's an agile method. This does not mean that you are working without planning, but it is a recognition that for large and complex systems it is impossible to describe a completed requirement specification and estimate how long the development will take and what it will cost, and the question is, what you then can do it. The idea of unified process is that you continuously work through the requirements specification and

the project plan so that after the completion of elaboration, the requirements specification is complete and including the final price for what the solutions cost and when it can be delivered and applied. Until then, it must all be based on estimates that are continually underpinned and adjusted, and that is exactly the dilemma of system development. However, in the unified process, the goal is that after elaboration the requirements must be in place, as well as the completed project plan and project requirements for resources.

As an addition to the above, I would like to mention that when it is so difficult to estimate a system development project, it is not only because developers are bad planners, but equally because the task manager and customers of large and complex IT solutions can not formulate the task. It requires a process of continuously identifying the requirements and deciding what to do, and what should be delineated, and remember that delineation may also be postponed to later. This is exactly the process that elaboration deals with.

The central concept in unified process is an iteration, which typically has a time span of about one month. As mentioned, an iteration must bring the system from one stable state to another and it is important that the result of an iteration is, that the task manager experiences a progression through a presentation of the result. The planning of the individual iterations starts during the inception phase and continues throughout the elaboration phase, with continuous being expanded with new iterations. For each iteration one must

1. plan what the iteration should include
2. and when the iteration must be completed

With regard to the first, the iteration must include a well-defined work of the entire project process, and the iteration should typically end in a month and should rarely last much longer. More people may work on the same iteration, but there will be few. Another question is what should happen if the iteration is not completed on time. Here it is an imperative requirement in unified process that the iteration is to be terminated on time, and what may be missing has to be postponed to a later iteration.

Construction

It is the longest phase in time. It consists in the same way as elaboration of a number of iterations, where each iteration brings the project from one stable state to another. There is no sharp boundary between elaboration and construction, but the difference is that at the start of construction, the remaining iterations will be planned and under the construction phase, the individual iterations will only contain to a limited extent analysis and, and to a lesser degree, design. One can think of it that when you get to the construction phase,

all requirements are formulated and in place (the requirement specification is complete) and all important decisions and risk assessments are clarified. Therefore, programming and testing are primarily what lacking.

The construction phase is usually extensive in time as many tasks can be deferred to construction. It is also typical that each iterations under construction to a greater extent than during the elaboration can be carried out in parallel, where several developers works on each their iteration.

The result of an iteration is in principle the same as during the elaboration phase, but with the difference that there will typically not be updates of the task formulation and the requirement specification and in worst case only minor adjustments, and in addition to the product, the result is just a short and accurate summary of the current iteration and including the results of the iteration's test.

One can also think of construction as a phase, where the project has now moved into a stable and planned course, through which the project moves forward towards the finished product through a carefully planned course. At that point, the risk is gone and there is only "hard" work left.

Transition

The last phase is also an iterative phase, but typically there are few iterations and often only one. It is the delivery phase and the tasks can vary a lot, but are the same as I previously described in the book Java 7. The planning of the phase's iterations takes place during elaboration.

2.4 SCRUM

Everything that is said in these books about system development relates solely to the developers' specific tasks, including how the developers can go forward to write and test the individual programs. Of course, it is also important, in one way or another it is, what it is all about, but for large projects where a team of developers collaborates to develop the current system, it is necessary with project management, and one of the reasons why IT projects sometimes goes wrong is actually lack of management. Over time, several project management methods have been developed, and one of those who has been very successful is called SCRUM. In fact, the method is not directly aimed at system development of software, but the method has proved particularly suitable for managing IT development projects. The following is a brief presentation of SCRUM.

One of the principles behind SCRUM is the recognition that, due to a number of uncertainties, it is not possible in advance to define all requirements for a system, thus completely the same as what I have mentioned above during the unified process. It is therefore necessary to have a project management method that is flexible enough to handle the development of a system that is characterized by continued change of requirements throughout the development process while ensuring progress in the project. Such system development processes are called agile, and SCRUM is a project management method for agile projects, and is the most widely used project management method in practice.

Now SCRUM is not always the right method, and for SCRUM to makes sense, the project must be of a reasonable size and be a project that should solved by a team of developers. In addition, it usually has to be a project characterized by many uncertainties and complexity, and where there is a need for agile methods, but can a project to some extent be characterized in that way, SCRUM has proven itself as an effective project management method.

The elements in SCRUM

Basically, the principles of SCRUM are to divide a project into a series of small iterations, which are called *sprints*. The starting point is that it has already been decided to develop

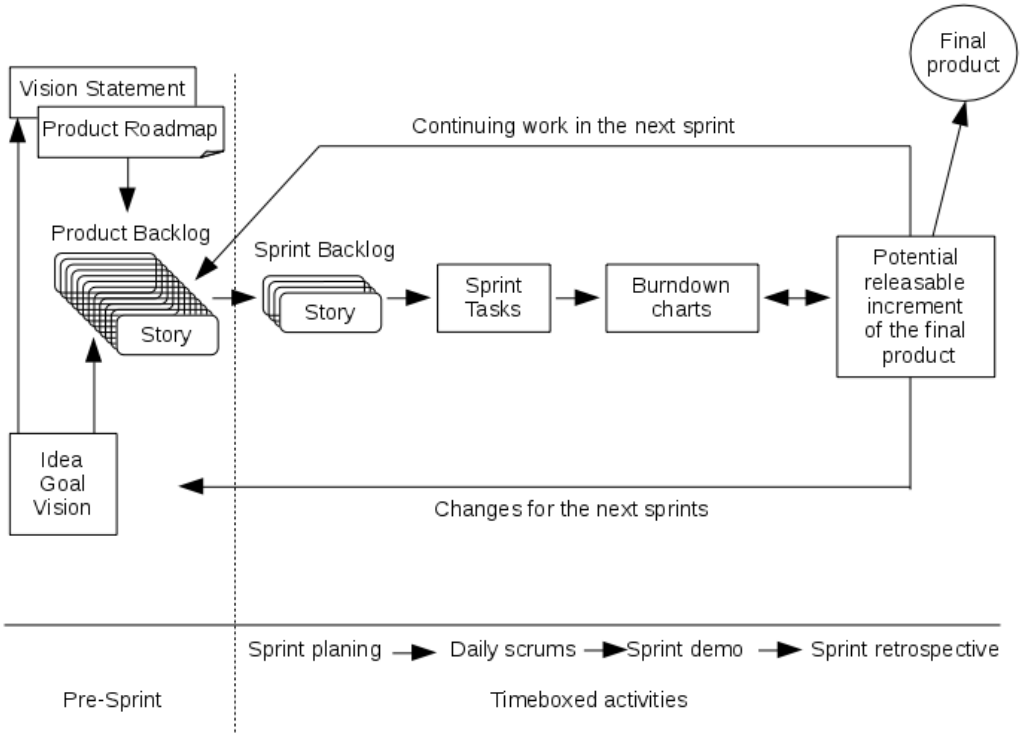
a system (and in this book an application), and the task manager or customer already has a relatively accurate meaning of what it is for a system to be developed. The project starts with the development of two documents:

1. *Vision Statement*, which gives a brief description of the goals of the project, which in the course of the project process will help the project team to focus on what is important seen from the task manager.
2. *Product Map,* which is an introductory schedule for when important sub-products are delivered.

The project process is divided into two phases, called respectively

1. Pre-sprints
2. Timeboxed activities

where you can think of the first one as planning and the other as production, but how far the largest part of time is used in the final phase. In fact, phases are not the right word, as it is not so that you are either in one phase or another, but it all starts with *pre-sprints*, after which you perform a series of sprints and eventually (and often) return to the pre-sprints phase. You usually outline the SCRUM with a figure similar to the following:



During pre-sprint, one focuses on injecting the user requirements and describing them as tasks to be solved. These descriptions are called stories. The result will typically be a large collection of stories, called *Product Backlog*. Typically, you do not complete Product Backlog before starting the individual sprints, and as soon as you find that enough information has been gathered that some stories are well described, you can start the first sprints. The goal is to get something done that end users can relate to and evaluate, and then to return to the pre-sprint phase and possibly update Product Backlog.

Looking at the sprints phase, it includes a sprint planing for every sprint, a meeting where you plan what stories from Product Backlog to be transferred to *Sprint Backlog*, and thus what stories to work on in it a following sprint. A sprint is scheduled to take a certain period of time, which may vary, but typically takes a sprint of about 4 weeks, and it is recommended that a sprint should not take longer than one month. The team breaks down the selected stories for tasks that are then delegated among the team's members. During a sprint, daily meetings, called *Daily Scrum*, are held, where each team member reports briefly the progress (or the opposite) in connection with the current tasks. These Daily Scrums are short meetings, and may take up to 15 minutes. At the end of a sprint, a meeting called *Sprint Demo* (or *Sprint Review*) is held. This meeting is held together with the task manager, where the team demonstrates the result of the current sprint and thus how the individual

stories are resolved. The entire sprint finishes with a *Scrum Retrospective*, that is a meeting that is a review of the sprint, where the team looks back on the course for improvements.

In connection with a SCRUM project, there are three roles (and there may be no other roles), and these roles are completed by the scrum team:

1. *Product Owner*, which is one person who is part of the project on full-time or on part-time, and which represents the business area within the project is anchored.
2. *Scrum Master*, who is one person who is full-time or on part-time on the project and who is a coach and facilitator for the development team.
3. *The Development Team*, which is 3–9 full-time employees (also called specialists) and is part of the development organization as those who perform the individual sprints.

In addition to this, there may be other stakeholders, who typically represent the customer or his organization.

It is assumed that the team meets two basic assumptions that are considered necessary to ensure flexibility, creativity and productivity and are considered necessary for the agile environment:

1. The team is self-organizing and plans its work without explicit leadership from the outside. One or more people in the team can be in charge of management while other participants are solely responsible for special specialist activities, but in SCRUM there is no division of management and specialist functions.
2. The team participants have all the necessary expertise and competencies needed to get the task solved and without the need to get help from outside.

The Product Owner

Any project needs a person familiar with the business area whose task is to maximize the value of the team's work and it is this person that is called *Product Owner*. Generally, the person comes from the development organization. It is important to be aware that the Product Owner must be know the problem area, but the person does not need to have special knowledge about system development. It is the Product Owner that is responsible for the project's Product Backlog, which is a priority list of stories to be solved and it is the central planning tool in SCRUM. Stories are prioritized in terms of their value for the user organization, so that high-quality stories are solved first. It is also the Product Owner who is responsible for that each user story being easy to understand for the Scrum Team and other stakeholders.

It is Product Owner who communicates with the task owner and user organization and uses this information to update the project's Product Backlog. It is also the Product Owner who has an overview of the progress in the project and keeps the project stakeholders updated with regard to the final end date.

The entire organization must respect Product Owner decisions, which is considered a prerequisite for the success of the project. Even Product Owner's chief may not generally correct Product Owner decisions, and no one other than Product Owner may decide what the development team is supposed to deliver. However, it is permitted for the Product Owner to delegate decisions to the Development Team, such as adding items to Product Backlog, as long as the Product Owner has the full overview.

Scrum Master

Scrum Master's task is to assist the Scrum Team by coaching them and ensuring that all principles of SCRUM are respected. Scrum Master must therefore have in-depth knowledge of SCRUM and the principles of the method. It is a management function, but it is not a project manager. Scrum Master will lead the SCRUM process, but not the Scrum Team. In addition to ensuring that the Development Team understands and uses SCRUM correctly, it is also the Scrum Master's task to remove obstacles to the Development Team, ease everyday life and generally coach the participants.

Scrum Master must also support Product Owner to pass the right information to the project's stakeholders and generally provide expert knowledge available to the *Product Owner*.

It is allowed for a person to be both *Scrum Master* and a member of the *Development Team*, but it is not recommended.

The Development Team

Members of the *Development Team* are IT experts and are responsible for delivering solutions to the elements of the backlog. Members generally have knowledge to do everything from A to Z in connection with the development of a task from the backlog, and they must be able to work independently themselves, and be able to solve the challenges that arise. One assignment can be awarded to a particular member through a sprint, but the entire *Development Team* is responsible and accountable for the task and there is thus no single person who owns a particular task.

The Development Team delivers the finished product developed through an iterative process managed by the project's Product Backlog. It is highly recommended that team members work full-time on the current project, and efforts should be made that the Development Team members not being replaced along the way, as it is considered important to ensure that members are focused and agile.

Experience has shown that SCRUM is most effective with a Development Team of 3–9 members, but for large projects it is possible to establish more *Scrum Teams*.

It is important to note that members of the Development Team do not have titles, such as designers, architects, managers, etc. Everyone must have the same role, and the title is only a member of the Development Team. The whole idea behind SCRUM is team work. The reason is that the members will otherwise focus on their own roles and not sufficiently focus on the finished product. Each member is responsible for all project results.

The sprints

Then there is the content of the individual sprints, where the actual system development takes place. A system development project managed by SCRUM solves the task through a number of iterations, called sprints, where a sprint is time-boxed and takes a certain period of time. A sprint contains 4 activities:

1. *Sprint Planning* is the first activity in a sprint, where the Scrum Team plans what will be the result of the sprint and what to deliver.
2. *Daily Scrum*. The Development Team starts the work immediately after Sprint Planning is completed. During the sprint, the Development Team keeps daily meetings, called Daily Scrum. It is a short meeting of approximate 15 minutes where the work for the next 24 hours is coordinated. As part of SCRUM it is recommended that these meetings be held standing.
3. *Sprint Review*. Before the sprint finishes, the Development Team presents the sprint's results for the user organization and retrieves feedback.
4. *Sprint Retrospective*. After Sprint Review and as the last in the sprint, the Development Team holds an internal meeting as a kind of review of the sprint in order to learn from the course.

Time boxing is considered to be central to SCRUM, and is the means to ensure that things are done, and for a project, each sprint should generally have the same fixed length. The length of a sprint is typically about one month, but may be shorter, but hardly for a week. The product is finished after a number of (and possibly many) sprints, and each sprint brings the product a step toward the finished result. If the result of a sprint is a potentially releasable part of the finished product (what to aim for is the result of each sprint), it is said that an *increment* has been reached, and an increment is thus the sum of all the elements in *Product Backlog*, which has been completed so far in the project. One can therefore consider each increment as an update of the previous increment and an extension of the product with new functions and features. An increment may not be applied, but in case of an increment, it should be possible.

Typically, an increment (result of Sprint Review) requires changes, and they will then be added Product Backlog as new stories.

Each story in the Product Backlog should usually be developed in a single sprint, as it makes it much easier to keep an overview. Product Owner and the Development Team selects a number of stories from Product Backlog for each sprint, with the goal of getting them completely completed in the upcoming sprint, so after the sprint the team has a new increment. It is therefore important that from the start you have decided when a story has been completed.

Sprint Planing

Sprint Planing is a time-boxed meeting, where all team members participate as a starting point. If it is a sprint of 4 weeks, the meeting may last 8 hours, but at short sprints the meeting will be shorter. Participants in the Development Team will estimate how many hours they can deliver in the sprint. Prior to that, the Product Owner prioritized the individual elements of the project's Product Backlog and ensured that each story is easy to understand. The Development Team then selects an appropriate number of items from the top of Product Backlog and moves them to Sprint Backlog as the work to be performed in the current sprint. The work for each story is estimated by the Development Team and the total work for Sprint Backlog tasks should as best as possible correspond to the capacity available to the team.

Then, a *Sprint Goal* is written, which is nothing else a short text describing the target of the current sprint, including what are the criteria for the sprint to finishing and what the tasks are solved. Sprint Goal must also contain at least a detailed plan for the first days of the sprint. Sprint Goal is part of Sprint Backlog, and there are no special requirements for documentation or form, and often a board is used, which makes it easy to maintain the plan, and ensure that the plan is visible to everyone. Sprint Backlog is thus made up of

1. *Sprint Goal*
2. The stories selected from Product Backlog for the current sprint.
3. A detailed work plan for the sprint, where the tasks (stories) can be categorized as *To Do*, *Doing* or *Done*.

In principle, a sprint can not be interrupted, but Product Owner may, as a sole exception, decide to interrupt a sprint typically caused by events/changes in the project's environments/ prerequisites.

Daily Scrum

As its name says, it is a daily meeting held throughout the sprint period. At the meeting, only members of the development team participates. The meeting lasts no more than 15 minutes, and at this meeting each team participant must answer three questions:

1. What has been completed since the last meeting?
2. What will be done before next meeting?
3. What obstacles/difficulties are there?

Team members must compare the progress with the Sprint Goal, but should also estimate the likelihood that a task will be completed before the sprint is complete.

Sprint review

It is a meeting to be held at the end of a sprint and the meeting will take for a sprint of 4 weeks max 4 hours, but for a shorter sprint the meeting will usually be shorter. The meeting includes the entire Scrum team but also other stakeholders, and at the meeting the solution of all tasks from the current sprint are presented. The goal is to get feedback and as early as possible to adjust the requirements so Product Owner can update the project's Product Backlog.

It is the Development Team that presents the product, but only for tasks that are *Done* and not for tasks that are "almost" Done. It is the Product Owner who before the meeting must ensure that the products presented also are Done.

At the end of the whole sprint, another meeting called Sprint Retrospective is held, which is a meeting of 2–3 hours. It is a meeting where you look back on the sprint for improvements, and it applies to both the working procedures for the individual team participants, processes and the tools that you use.

More about SCRUM

An activity that in principle always takes place in parallel with the individual sprints is called *Product Backlog Grooming*. It is always about reviewing and adjusting items in Product Backlog, and typically includes adding details, adjusting estimates, and so on. It is Product Owner who is responsible for prioritizing the items in Product Backlog, and it is the Development Team that is responsible for estimating the items.

Although it is not part of SCRUM, it is recommended that you plan a day off or two between each sprint, which you can use to read articles, attend courses or workshops or otherwise orientate yourself to new subjects, or you can simply go for a walk and enjoy the nature. Perhaps this idea may be difficult to sell to a manager, but studies shows that the time is actually well-done, partly in order to make progress in the project and partly to ensure that team participants stay up to date with the technology development. The idea supports SCRUM, where it is absolutely crucial that you are constantly product-oriented rather than activity-oriented.

Another question is what you do at the end of a sprint if you find that some of the work is not done. Here the rules are quite accurate:

1. Unfinished or undone elements (stories) are placed back on the Product Backlog.
2. The Product Owner re-orders the backlog, where typically unfinished elements are placed at the top, but not necessarily.
3. A story that is undone must not be resized to represent only the remaining undone work. The story that was worked on must not be split in and what was done and what was not done. A story is done or not.

In general, it can be said that SCRUM includes the following artifacts:

1. *Product Backlog*, that is an ordered list of all stories that might be needed in the final product
2. *Sprint Backlog*, that are selected stories from Product Backlog to be delivered through a sprint, along with the *Sprint Goal* and sprint plans for all stories in the Sprint Backlog
3. *Increment*, that is the set of all the Product Backlog stories completed so far in the project
4. The definition of *Done*, that is the understanding of what it means that a piece of work to be considered completed
5. *Monitoring Progress* towards a goal, that are the performance measurement and forecast for the whole project
6. *Monitoring Sprint Progress*, that is the performance measurement and forecasts for a single sprint

You should be aware that the literature regarding SCRUM defines the exact content of these artifacts and also typical tools that you can use.

As a last comment on SCRUM, it is important to emphasize that the goal is to provide a framework for how you effectively can organize the development of a larger and complex (IT) system, and the method expresses the experience of professional developers in implementing complex development projects. On the other hand, SCRUM does not say anything about what the individual sprints should contain and how each team participant should perform the daily work, and here you must apply to what else you have learned about system development (for example, something of what is explained in this series of books) or better what is illustrated in the following chapter on extreme programming. In practice, you will combine a variety of systems development and programming techniques.

2.5 EXTREME PROGRAMMING

Extreme Programming, also called XP, is a system development method that, as the name says, unlike the methods mentioned above, focuses on programming. You should not perceive XP as an alternative to, for example, Unified Process or SCRUM, but rather a recommendation for how to successfully complete the programming part of the individual iterations. XP, like the other system development methods, consists of a number of iterations where each iteration brings the project from one state to another. There are the same activities as in other methods, but in XP, the three activities design, program and test together form a single activity.

The method is based on four main principles:

1. *Heavy customer involvement*, where a representative of the user organization is included in the development team and continuously participates and actively contributes to the planning of the individual iterations and the various accept tests and to a large extent also conducts these tests.
2. *Test-driven development*, where developers are required to write unit tests for each new feature before any of the code is written. In this way, a function can not be transferred for testing before all unit tests are performed with success (see the next chapter on test-driven development).
3. *Pair programming*, where all code is written by two programmers, called a *driver* and a *navigator*. Both programmers are at the same computer. The driver writes the code while the navigator monitors the process, comes with suggestions, thinking about design and program quality and so on. Every half hour, they switch the two roles. The idea is quite simple, that two thinks better than one. Pair programming

is of course less effective than two programmers working on each their code, but experience shows that code produced is of better quality and with far fewer errors.

4. *Short iteration cycles and frequent releases*, where you develop software in short iterations that do not extend beyond a few weeks. The goal is frequent releases that can be handed over to and in principle taken in use by the user organization. A release period spans no more than a few months and is performed over few iterations. The idea is that the user organization must experience progress and that something happens.

As with all other system development methods, the goal of XP is doing better, and XP rightly considers the biggest issue in software development as risks. It is evident that schedules that do not hold increases error rates as projects grow bigger and more complex, and misunderstanding the task, which means that the product contains features that the customer does not demand at all, and at worst, one may risk that the project completely must be abandoned. XP tries to minimize risk by controlling four variables:

1. cost
2. time
3. quality
4. features

where the latter might be the most important as a measure of what the program should be capable of, but the four variables are of course not independent and in order to meet the cost and time requirements, it can easily mean that it is necessary to quench either quality (what is always short-term) or features. The only thing new in these variables is that the method from the outset focuses on the four key areas and controls them.

In addition to these control variables, participants in XP projects need to work from multiple core values, where I want to mention *communication*, where all participants commit to spreading valuable knowledge to the entire team. This can be achieved through relatively small development teams, like par programming, and collective ownership of the code also help ensure effective communication.

In addition, extreme programming defines 12 principles for the development process:

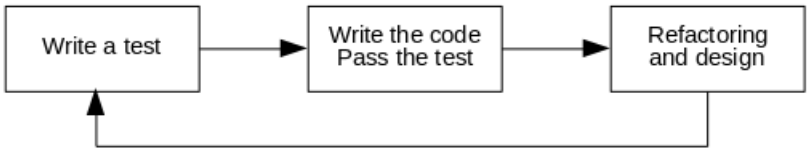
1. Planning
2. Small releases at short intervals
3. System metaphors
4. Simple design
5. Test
6. Frequent refactorization
7. Pair-programming
8. Common ownership of the program code
9. Continuous integration
10. Affordable work rate
11. An overall development team
12. Common code standard

and to be successful with the method, it is considered crucial that these principles are respected.

There is much more about XP, and much are written about XP and gained experience, and it all shows – at least if you ask the method's advocates – that the method is particularly useful when the goal is to develop quality software.

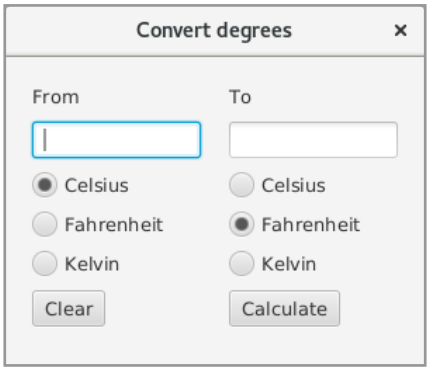
3 TEST-DRIVEN DEVELOPMENT

As mentioned in the previous chapter, *Test-Driven Development* – also shortened TDD – is part of the requirements for *Extreme Programming*. It is very simplified that you start by writing some unit tests. Then you implement the code to be tested and perform the unit tested on the code and finally perform a refactoring. This process is repeated until complete:



TDD is based on the use of unit test and in this book *JUnit* (see possibly the Java 3 book), but as a very simple example to illustrates some of the idea with TDD I will show the development of a program where you can convert between three temperature units:

1. celsius
2. fahrenheit
3. kelvin



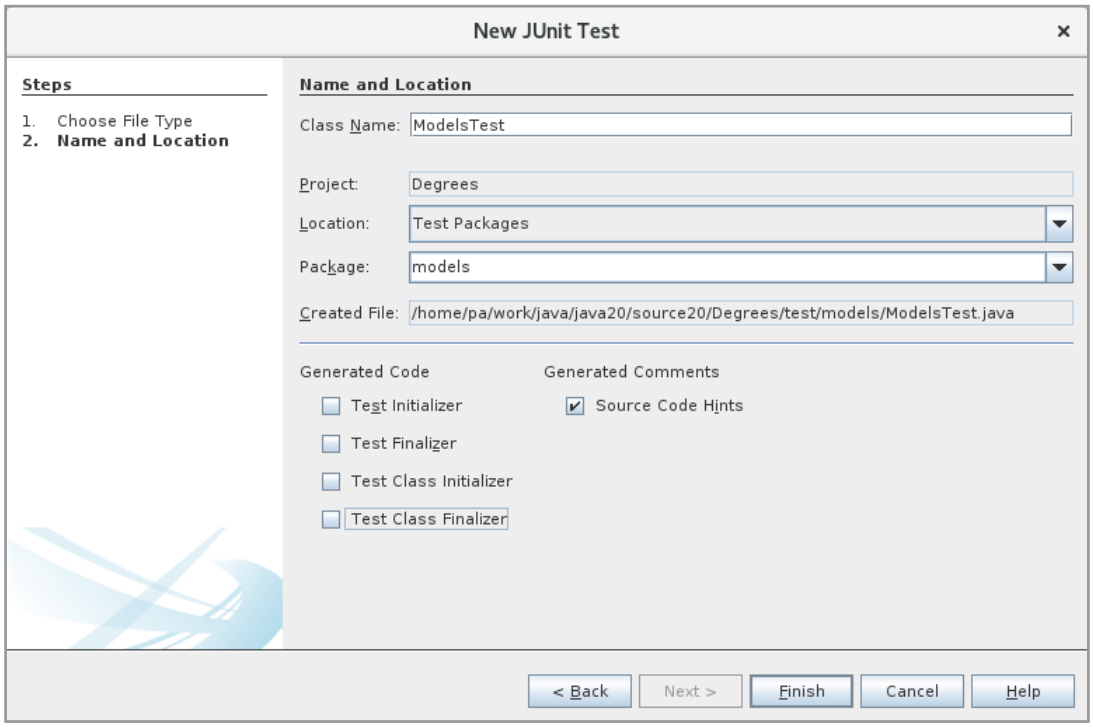
The following shows the procedure.

I start with a JavaFX project, which I have called *Degrees*. In this case, the program must consist of a user interface class – called *Degrees*, created by NetBeans – and a class *Converter* that will implement the necessary conversion methods. I want to think of the class *Converter* as the program's model and to write this class i will start with a test class.

As a first step, under Projects, right-click on the project name and select *New | JUnit Test* (see below). Here I have:

- called the test class for *ModelsTest*
- said that the test class should be added to a package *models* (under *Test Packages*)
- removed all check marks under *Generated Code*

The latter is not necessary, but in this case, I do not apply these initialization methods, but often you can let NetBeans create them as they as default are empty and therefore do not perform anything. When you click *Finish*, NetBeans creates the following test class:




```
package models;

import org.junit.Test;
import static org.junit.Assert.*;

public class ModelsTest
{
 public ModelsTest()
 {
 }

 // TODO add test methods here.
 // The methods must be annotated with annotation @Test. For example:
 //
 // @Test
 // public void hello() {}
}
```

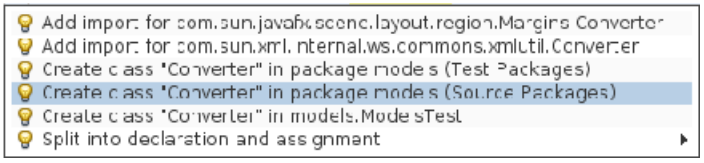
The task now is to add test methods to the test class. I start with the following method:

```
@Test
public void fahrenheitToCelsius()
{
}
```

which is nothing but a simple test method. Here I have added a single statement:

```
@Test
public void fahrenheitToCelsius()
{
 Converter converter = new Converter();
}
```

Obviously, it results in a translation error, since the class *Converter* does not exist, but if you press *ALT* + *Enter*, you get the following pop-up:



If you choose *Create class "Converter" in package models (Source Packages)*, NetBeans creates a package *models*, including a class *Converter*. The class is for now empty. The class must have a method *ftoc()* that can convert a temperature measured in Fahrenheit to Celsius. If *f* stands for degrees measured in Fahrenheit and *c* for degrees measured in Celsius, then the relationship can be expressed by the following formula:

```
c = (f - 32) * 5 / 9
```

In order to test the method *ftoc()*, I have expanded the test class with the static array, which for selected degrees measured in Fahrenheit shows the corresponding degrees in Celsius:

```
private static double[][] FC = {
 {-459.67, -273.15 },
 { -50, -45.56 },
 { -40, -40.00 },
 ...
 { 1000, 537.78 }
};
```

Note that you can create the table either by finding coherent values on the Internet or by calculating the values yourself. With this static array in place, the test method can be completed as follows:

```
@Test
public void fahrenheitToCelsius()
{
 Converter converter = new Converter();
 for (int i = 0; i < FC.length; ++i)
  assertTrue(Math.abs(converter.ftoc(FC[i][0]) - FC[i][1]) < 0.01);
}
```

Again you get a translation error, but you can use ALT + Enter in the same way as above, and by clicking the mouse you can use NetBeans to create the method *ftoc()*:

```
package models;

class Converter
{
 double ftoc(double d)
 {
  throw new UnsupportedOperationException("Not supported yet.");
 }
}
```

If you then try to perform the unit test, it will fail because the method has not yet been implemented. Next step is therefore to implement the method *ftoc()* – write code and pass the test:

```
double ftoc(double d)
{
  return (d - 32) * 5 / 9;
}
```

Then you should be able to test the method (perform the above test method) successfully.

Corresponding to the introduction, the next step is to perform a refactoring. In this case, both the class *Converter* and the method *ftoc()* must be *public*. In addition, -459.67 is the absolute zero point for temperatures measured in Fahrenheit, and the method should therefore raise an exception if the value of the parameter is less than the absolute zero, so the method can be refactored to the following:

```
public double ftoc(double d) throws Exception
{
 if (d < -459.67) throw new Exception("Illegal temperature value");
 return (d - 32) * 5 / 9;
}
```

If you repeat the test, it will fail, as the method can now raise an exception. The test method must therefore be updated:

```
@Test
public void fahrenheitToCelsius()
{
 Converter converter = new Converter();
 try
 {
  for (int i = 0; i < FC.length; ++i)
   assertTrue(Math.abs(converter.ftoc(FC[i][0]) - FC[i][1]) < 0.01);
 }
 catch (Exception ex)
 {
  fail(ex.getMessage());
 }
}
```

and then the test can be performed correctly.

The class *Converter* must also have a method *ctof()* that can convert from Celsius to Fahrenheit, and I therefore start adding a new test method to the test class:

```
@Test
public void celsiusToFahrenheit()
{
 Converter converter = new Converter();
 for (int i = 0; i < FC.length; ++i)
  assertTrue(Math.abs(converter.ctof(FC[i][1]) - FC[i][0]) < 0.01);
}
```

The only difference is that the test method now tests a method *ctof()* and that it is the Celsius value in the table, which is used as a parameter. The method *ctof()* must be created, which I like NetBeans do in the same way as above. Next, the method must be implemented, which consists simply of isolating the variable *f* in the above formula:

```
double ctof(double d)
{
 return d * 9 / 5 + 32;
}
```

Then the unit test is performed again and it should be without error. Again, a refactoring of the method *ctof()* must be made, so it raises an exception if the value of the parameter is below the absolute zero point of Celsius, which is 273.15. The test method must also be updated to treat any possible exception.

The class *Converter* must have four additional methods:

1. *ktoc()* – convert kelvin to celsius
2. *ctok()* – convert celsius to kelvin
3. *ktof()* – convert kelvin to fahrenheit
4. *ftok()* – convert fahrenheit to kelvin

The formula for converting Kelvin to Celsius is:

```
c = k - 273.15
```

where *k* indicates degrees measured in Kelvin. Similarly, the formula for converting Kelvin to Fahrenheit is:

```
f = k * 9 / 5 - 459.67
```

First, the test class is expanded with two new static tables:

```
private static double[][] KC = {
 { 0, -273.15 },
 { 10, -263.15 },
 ...
 { 1000, 726.85 }
};

private static double[][] KF = {
 { 0, -459.67 },
 { 10, -441.67 },
 ...
 { 1000, 1340.33 }
};
```

where the first is a conversion table from Kelvin to Celsius, while the other is a conversion table from Kelvin to Fahrenheit. Next, the test methods are written for each of the four new methods and the new methods are implemented. I do not want to show the code for the four new test methods or the implementation of the four methods, as they are all in principle identical to the above – just other test data and other formulas are used. When all unit tests are performed without errors, you can assumes that the class *Converter* is complete and ready to be used. It happens in the class *Degrees*, which opens a simple window, nor here I will show the code.

The class *Converter* is quite simple, and it's hardly clear what you've achieved by starting typing the test methods, but basically you get two things:

1. When writing a class, you decide through the test methods how the class' methods should work and when the methods are implemented, you can test them immediately using the test methods.
2. If you later update or change a method, you have the necessary test method, so you can immediately test where the method has errors.

Test Driven development is one of the many initiatives in system development that are worth learning and being good at, as the method gives very good results both in small and large projects.

4 DESIGN PATTERNS

At one point, system development began to talk about design patterns, and like everything else new, design patterns received great attention – and perhaps even some interpreted as the solution to this world's problems in software development. It is probably a little over-interpreted, but conversely, knowledge of design patterns is important for software development in practice. It all started with a book

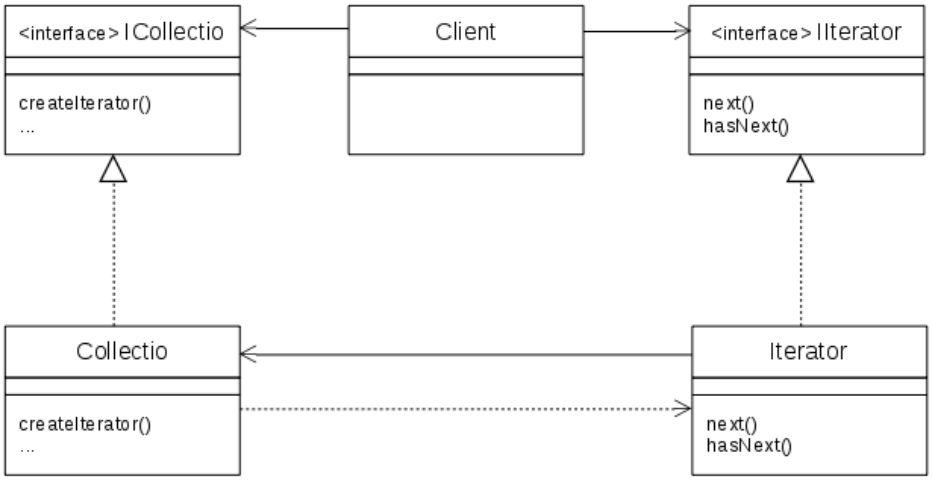
Design Patterns: Elements of Reusable Object-Oriented Software

written by four writers, referred to as the *Gang of Four*. They described 23 patterns, and this book has since been the main source of design patterns. It is certainly recommended to read the book and study the 23 patterns. Since then, others have described other patterns, so today there are many to choose from.

A design pattern is a description of a solution for a particular problem. The starting point is a simple observation that certain issues appear more or less directly over and over again in many different software projects. If that is the case, one can try to describe the problem abstract and subsequently indicate a concrete solution. That way, as a developer, if you have

a problem that can be categorized under a design pattern, you can easily figure out how to solve the problem. This means several important advantages, the most important being that you use the experience of other developers to solve a specific problem, rather than having to find in your own way to do it. This means cheaper and far more stable programs.

As an example of a design pattern, I would mention the *iterator pattern*, as I have previously mentioned and used several times. Given a collection of objects of one kind or another, the problem is that you need to traverse all objects and do something with them. If you use the design principle, to program to an interface, and the collection class *Collection* is defined by an interface *ICollection* so that the *Client* program only knows the collection class through the defining interface. The interface defines a method *createIterator()* that returns an iterator, which is an object that initially refers to the first object in the collection and has two methods, one that returns the object that the iterator points to and step the iterator forward, while the other method tests if the end is reached. The class *Iterator* can also be defined by an interface, and the iterator pattern can thus be illustrated as follows:



If a collection class implements this pattern, it means that the class can be treated in the same way, regardless of whether it is for a collection. For example, think about Java's implementation of collection classes.

I have used other design patterns in the previous books:

- Singleton
- Factory
- Adapter
- Observer

and there are many others, and since *Gang of Four* defined 23, many others have been defined. The latter can be a challenge as many patterns look similar and can be difficult to distinguish, and there has also been a tendency to define patterns for anything, but it does not change that patterns are important and that it pays to teach them.

The patterns of *Gang of Four* and many others concerns concrete issues in programming, but other patterns are more general and are often composed of several patterns. One example is MVC, which is a pattern for the design of a GUI program. It is also a design pattern that expresses others' experience of how to design a GUI program. If you do, others who must maintain the program knows how the program is designed. The pattern is described in these books. It's a complex pattern, and it is a pattern that is found in several variants. It expresses the fact that it is actually difficult to define design patterns if they are to be general and therefore MVC exists in so many variants, which are determined by both software platforms and development tools.

Another important reason for using design patterns is that developers in this way have a family of common concepts. For example, if I say that a class in a program is an adapter, I've told others that it's a class that is used to smooth the differences to other classes, and I can even look up a book about design patterns and see what an adapter is and how it is used.

5 REFACTORING AND OTHER

In order for a development project to be successful, the completed program must meet:

- that the program meets the requirements
- that the program works without errors
- that the program is robust
- that the program is easy to maintain

and if you read the literature you will be presented with a number of other quality factors, but in addition to that, you can add that the project is completed within the estimated resource consumption and that it is completed on time.

5.1 REFACTORING

In order to meet these requirements, you have to work systematically and follow a method, and the four basic activities in system development are, as already mentioned many times:

1. analyse
2. design
3. programming
4. test

and then combined with iterative system development, but in addition to that, there will always be an activity called *refactoring*, where you to some extend has to rewrites the code to make it easier to read and maintain, while you have to retain the code's functionality.

Once you have written the code and tested it and found that it meet the requirements, the code has been through a process where you have developed on the code, tested it, corrected, adjusted, expanded and so on and the result will very often be a code that is characterized by:

- Variables not used and which must therefore be removed.
- Methods not used, which should therefore be removed.
- Two or more methods that almost do the same, and where you can reduce to a single method using a parameter.
- Methods that may be called something different, but do the same where you should only have one method that may be placed in a particular tool class.
- Instance methods that do not depend on the instance, where the method maybe can be defined static or move to a tool class.
- Classes that are almost identical, where classes can be changed to specializations of a common super class and where the methods that are the same are moved to the super class.
- Code commented because the code should not be used. Such a code must be removed.
- Code that solves the task, but makes it inappropriate. Here you should find a better code, which usually is the same as a better algorithm.
- Missing comments, which should be written.
- Classes with very high coupling. If so, you should consider whether you can reduce the coupling in one way or another. An example could be to implement the observer pattern using an interface.
- Classes with very low cohesion. If so, one should consider dividing the class into several classes.
- Import statements that are unnecessary. They should be removed, so you only have the import statements that are necessary.

and many other examples could be mentioned. These relationships mean two things:

1. That the code fills more than necessary is.
2. That the code is far harder to maintain than necessary is.

and here it is absolutely the last, which is the biggest problem. Therefore, the code must be refactored, which can be considered as a cleanup process to clean the code for the above inconvenience.

As mentioned, the aim of a refactoring is to get a simpler code and thus a code that is easier to maintain. Conversely, you should note that refactoring does not develop into direct reprogramming. If that is the case, the cause is usually serious error decisions during the development process, and if necessary, you should postpone a refactoring to a new project where the task is directly a refactoring of an existing program. To avoid getting in that situation, the solution is frequent refactoring, and maybe it should be part of each iteration, and here's the worth to note that in Extreme Programming refactoring is an integral part of each iteration.

5.2 LOG FILES

As a last comment on system development, I want to mention log files, which are typically text files, where a program intermittently stores data relating to the application's use. Now it does not have to be text files, but it can also be a database table, and if necessary, make sure that the program has database access. If you are using text files, make sure that the log file is created in a directory with a write access, and often you want to use the user's home directory, but it is of course not an option if the log is to be used by multiple users. If a program uses log files, it can of course be a little dangerous to let a program write unrestricted and unchecked to a file or database table, and therefore log files should usually be combined with an option to disable/enable this functionality.

There may be several reasons for using log files, and one is to collect statistic data relating to the application's use, perhaps in order to improve the program later. Another and perhaps even more important application is logging of an error message. Once a program has been put into operation and errors occurs, they are reported to developers who then have the correct error. In order for this to be possible, you need some information, and here I would like to say that an error handling, where the program just crass and the screen is filled with errors regarding exceptions, simply can not be used. Firstly, users will not accept such malware, and it creates frustration and unwillingness towards the development organization, and secondly, it is far from the assumption that users are reported to such errors, and at

best, you will be notified that the program is not working. As a developer, you know critical places where a program can go down and here you can encapsulate the code that can fail in a try/catch block and in case of an exception write a message in a log file. As a developer, in case of an error, you can ask to get the log file, and here you can see when and where in the program there have been errors.

Such error logging can be a very big help in troubleshooting, just be careful about where in the code you write in the log file. In principle, the error log must be empty and therefore it may be a nice check of the program to periodically inspect the log file.

It is important to note that error logging is by no means an alternative to testing and it is only something that can be used when the program is finished and tested and ready to deliver to the customer, but conversely, has the program first been put into operation, it can also be an extremely important source of reporting errors.

6 POKER

I want to close this book and thus also this series of books about software development in Java with another example of a program. The goal is similar to the final examples in the other books to show the development process, but the task/project does not directly follow any of the previous system development methods, but uses elements from several of them and what is otherwise mentioned in these books. The reason is partly the size of the program, where it is still a small program in size, and partly that the program is written by a single person. For these reasons, it does not make sense to use, for example, SCRUM or Unified Process, but conversely, elements of the methods will be used. In fact, it also reflects how system development methods are used in practice, where elements of concrete methods are used, and because the following is a slightly bigger program than what I have otherwise presented in these books, but in relation to the development of many IT systems in practice, it is still a very small program.

I have previously shown – in the book Java 4, Problem 2 – a program where you could play poker against three virtual players. The program was somewhat simplified, primarily because it did not contain any betting. The following is also a poker program, but for the purpose of solving some of the shortcomings the program in Java 4.

6.1 VISION

The task is to write a program where a player (user) can play poker on a single computer, a little like what for many is a very well-known Windows program called *Hearts Free*, but the card game here is instead poker. The user is playing against a number of virtual players. It is thus a usual GUI program.

The game must support the following variants of poker:

1. Texas Hold'em
2. Omaha
3. 7 Card Stud
4. 5 Card Draw

and it is the player (user) who decides what kind of game to start.

The program's vision is entertainment and you should not play with real money, but only about chips (jetons). The player must have an account that over time keeping track of how

much the player (the user) has won and lost, and it must be possible to insert and raise the account.

It is an important challenge to implement a virtual player that simulates a real poker player so that the virtual player is acting intelligently. The goal is that a user should find it interesting to play against other virtual players and that it is difficult to figure out how a virtual player acts.

6.2 ANALYSIS OF POKER

The following is an analysis of the game poker, and thus the rules of the game, or more precisely how the game poker is interpreted in this program and what options the program should support.

Poker is played with all 52 cards and without jokers. Basically, the cards rank in ascending order are:

- two, three, four, five, six, seven, eight, nine, ten, jack, queen, king and ace

In some cases, the color also has significance, and the rank is then (in ascending order):

- diamonds, hearts, spades and clubs

A player must in one round form a *hand* consisting of 5 cards and the result of the hand is one of 9 options, arranged according to falling value:

Straight flush

Five consecutive cards of the same color, making it the hand with the highest value. An example could be

- 5 spades: 8, 9, 10, Jack and Queen

If there are two or more players who have straight flush, the highest card determines the rank, and if the cards are the same, the color determines the rank.

Four of a kind

Four cards of same value, and an example could be

- diamonds 4, hearts 4, spades 4, clubs 4 and another card

The last card has no significance for the rank and has two or more players four of a kind, the highest card of the four of a kind determines the rank.

Full house

Three of a kind and one pair, for example

- diamonds 8, spades 8, clubs 8, hearts king, spades king

If two or more players have a full house the highest of the three of a kind determines the rank.

Flush

Five cards in the same color, but not necessarily continuous values, for example

- diamonds 2, diamonds 7, diamonds 8, diamonds 10 and diamonds queen

If two or more players have a flush determines the highest value of the first card the rank, which they are different, and if all cards the same value determines the color the rank.

Straight

Five cohesive cards, but not necessarily in the same color, for example

- diamonds 5, clubs 6, spades 7, heartsr 8 and spades 9

Having multiple players straight determines the highest card the rank and has all cards the same value determines the color of the highest card the rank.

Three of a kind

Three cards of same value, where the last two cards have a different value and are not a pair, for example

- hearts jack, spades jack, clubs jack, diamonds 2 and hearts 5

In the case where two or more players have three of a kind, the rank of the highest card (for the 3 of a kind) determines the rank. The last two cards therefore never have a meaning for the rank.

Two pairs

Two pairs: Two different pairs, where the last card must have another value, for example

- spades 2, clubs 2, hearts king, spades king and clubs ace

If two or more players have two pairs, it is the largest pair that determines the rank. If the largest pair has the same value, the rank is determined as the largest of the smallest pair, and if this pair has the same value, the rank is determined of the last card. If also the last cards have the same value, the rank is determined as the color of the last card.

One pair

Two cards with the same value, where the last three cards all have a different value and do not have the same value as the two cards that make up the pair, for example

- spade ace, clubs ace, hearts 4, hearts 7 and diamonds jack

Have two or more players a pair the biggest pair determines the rank. Do they have two pairs of the same value, the rank is determined as the highest card of the first of the three remaining cards that are different and have all three remaining cards the same value determines the color of the highest of these cards determines the rank.

High card

None of the above hands occurs, for example

- clubs 5, diamods 6, hearts 9, spades 10 and spades ace

If more players have high card, determines the highest value of the first card that is different the rank and have all 5 cards the same value determines the color of the highest card the rank.

A round

A single game is called a round, and for each round the players are switched as dealer. The game is played by players continuously building their hand as the cards are dealt. How it happens and how the hand is formed depends on the current variant (*Texas Hold'em*, *Omaha* and so on). For each variant there are a number of betting rounds where players either deposit money into a pool (a pot) or leave the round (and the money they have already deposited are thus lost). You can win a round in poker in two ways:

1. By having the best hand at *showdown* (when all betting rounds are completed and the round is over).
2. By being the last remaining player in the current round.

For a round, as mentioned above, there are more betting rounds where each remaining player bets, thus either depositing an amount in the pot or leaving the current round (dropping out). You have the following options (slightly dependent of the current variant):

1. FOLD, the player gives up and is no longer a player in the current round.
2. CHECK, the player wishes to continue the round, but without making a deposit (only possible if in the current betting round there is not already a player who has deposit money in the pot).
3. BET, the player deposits an amount in the pot (only possible if there is not already a player who has deposited an amount) or there is another player who has CHECK.
4. CALL, you deposit the same amount as the last BET or RAISE and is thus the least amount that allows you to continue playing.
5. RAISE if there is already a BET, a player can increase the deposit amount and all remaining players who are still there must choose either CALL, RAISE or FOLD.

In connection with BET and RAISE, there will usually be an upper limit for the amount of money you have to deposit.

With regard to the different variants, the current program should support 5 variants:

- Texas Hold'em
- Omaha
- 7 Card Stud
- 5 Card Draw
- 5 Card Stud

Here is there probably not no game as the last version, and at least it's not a game found on the usual poker game pages, but it's a very simple poker game and that's how I've learned to play poker. The game is similar to *7 Card Stud*, and hence the above name.

Chips

In this program, the smallest amount is a chip (or jeton), which in principle is just a unit. When a new game is started, each player gets a number of chips, and one player is released from the game when the player does not have more chips. If the player is the user, the game is over.

At the start of a game, it is determined how much as a number of chips (default is 1), it will cost each player to detach in a round. This value is in this program called a *blind*.

A blind is then the minimum value a player has to deposit in the pot at a BET.

In this program, the following rules must be applied regarding how much players must deposit in the pot and thus the limits of the game.

1. At the start of a new round, all players deposit a blind.
2. A round has a value at any time, and is the last deposit in the pot at any time. That is, at the start of the first betting round the size of the value is equal to a blind.
3. At BET you can deposit the value or 2 times the value, which is then the round's value.
4. At RAISE you can raise to 2 times the value or 4 times the value, which is then the round's value.

In addition, in a single betting round, there should be a limitation on how many times a player must perform a RAISE (for example 2 or 3).

If a player does not have more chips, the player must leave the game.

It has been decided (primarily for the user interface) that there can be 2–6 players regardless of the variant of the game.

Texas Hold'em

In this game each player have two closed cards (which only the player can see). In addition, 5 open cards are placed on the table (cards that everyone can see). Each player must then form the best possible hand with 5 cards based on his own two closed cards and the 5 open cards on the table. It is optional if one or both of the closed cards are included in the hand.

The game has 4 betting rounds, and the game progresses (in this program) as follows:

1. At the start of a game, the user is selected as a dealer. The dealer changes a clockwise position after each round.
2. Before the players get cards, all players must put a blind in the pot. Then all players at the table are given two closed cards.
3. Then the first betting round begins at the player after the dealer.
4. Now three cards all with the face upside are placed at the table – the so-called 'flop'. It is common cards and can be used by all players to create their best poker hand. Then there is another betting round, where it is the first player after the dealer, who is still in the game, that begins.
5. Another open common card is placed on the table. This is called the 'turn' card. Then a third betting round follows with the same pattern as above.
6. The fifth and last common card – the 'river' card – is placed on the table, after which the game's last betting round is performed, exactly as the previous two betting rounds.
7. If there are two or more players who have not yet thrown their cards after the last betting round, the winner will be settled at a showdown. Each player builds the best five-card poker hand out of his own two personal cards and the five common cards on the table (the player does not have to use his own closed cards). The player with the best poker hand wins the pot. If there is only one player left, this player has won the pot.

Omaha

This variant is basically played as *Texas Hold'em*. Each player is given closed cards, followed by five common cards placed on the table as a flop (three cards), one turn card (one more card) and one river card (one last card), with a betting round between each and thus 4 betting rounds as in *Texas Hold'em*. In *Omaha*, however, you get four cards on the hand (4 closed cards) and when you combine your personal cards with the common cards, you need to use exactly two cards from the hand and three from the table.

7 Card Stud

In this variant there are 5 betting rounds. The variation differs from *Texas Hold'em* and *Omaha* in that each player is assigned his own open personal cards – that is cards, which may only be used by one player. There is thus no common cards. In *7 Card Stud*, each player receives from the start two closed cards and one open card, followed by three more open cards one at a time with a betting round between each and finally another closed card. Thus, each player ends finally at the end of the round with seven cards – three closed and four open. The game progresses as follows:

1. Each player pays a blind to the pot.
2. Each player gets two closed cards (face down) and an open card (face up). Then the first betting round follows and the player with the lowest open card starts. If two players open cards have the same value, it is the cards color that determines who starts.
3. Players who have not yet folded their cards now receive a fourth and open card and a new betting round (the 2nd betting round) begins. This time it is the player with the best combination of the open cards that starts.
4. Another card is given with face up and another betting round (the 3rd bidding round) takes place according to the same pattern as the previous betting round.

5. Another card is given with face up and another betting round (the 4th betting round) takes place according to the above pattern.
6. A seventh and last card is given to the remaining players in the round, but with the face down (a closed card) and the last betting round (the 5th betting round) takes place according to the same pattern as the previous one.
7. If there are two or more players who have not yet folded their cards after the last betting round, the winner will be settled at a showdown where each player will make the best five-card poker hand out of his own seven cards. The player with the best hand wins the pot. If there is only one player left, this player has won.

5 Card Draw

It is the original poker game and is simpler than the above variants, and there are only 2 betting rounds. The procedure is as follows:

1. Each player deposits a blind.
2. Each player is given 5 closed cards.
3. The first betting round takes place and starts with the player to the left of the dealer.
4. After the first betting round, the players still in the game must throw between 0–5 cards, after which they receive a corresponding number of new cards given by the dealer.
5. When the players have exchanged cards (if they choose so), the second betting round will come.
6. If there are more than one player left, the winner is founded at showdown and if there is only one player left that player has won.

If there are 6 players, it means that you in principle needs 60 cards, which is not possible. It is decided to ignore this problem as it is perceived as theoretically that all players throw all there cards. Should it happen, it is assumed that the game is ignored, but the pot will remain to the next round.

5 Card Stud

As mentioned, it is a simplified game with four betting rounds and the procedure is as follows:

1. Each player pays a blind to the pot.
2. Each player receives a closed card and an open card. Following is the first betting round where the player with the highest open card starts.

3. Players who have not yet folded their cards now receive a third open card and a new betting round (a 2nd betting round) begins where the player with the best combination of the two open cards starts.
4. A fourth card is given with the face up and another betting round (a 3rd betting round) takes place according to the same pattern as the previous betting round.
5. A fifth and last open card is given to the remaining players in the game and the last betting round takes place according to the same pattern as the previous two.

If there are two or more players who have not yet thrown their cards after the last betting round, the winner will be settled at a showdown, otherwise the remaining player will win the pot.

6.3 THE PROCESS

The development is planned to include the following iterations:

1. iteration
   Development of prototype for *5 Card Stud*. The purpose is primarily to get started, including the preparation of a preliminary design of the user interface. You should not be able to complete a hole round, but you should be able to deal the first two cards so that all the technical regarding showing the cards is in place.

2. iteration
   An algorithm must be developed for how a virtual player should play *5 Card Stud*. The algorithm should include

   - how the virtual player should act in a betting round, for example, how much the player has to bet (BET of BIG BET), when the player has to raise (RAISE or BIG RAISE) and when the player should choose FOLD and possibly bluff
   - who should start a betting round
   - who wins at showdown

   In addition, the iteration must include an algorithm for how the individual betting rounds should proceed depending on whether it is the user or a virtual player who starts the betting round.

3. iteration
   The overall design of the program's classes, which includes important decisions for which classes the program should primarily consist of.

Implementation of the poker variant *5 Card Stud*. The iteration thus includes implementation of all algorithms from iteration 2. After this iteration, you should be able to play poker with the program, but only the simple *5 Card Stud* variant and there should be no user administration.

4. iteration

   The Iteration includes decisions and designs regarding how to save the program's data and what data to save. It is decided that data should be stored in common files, partly because it should be simple to install the program, and partly because data is not critical, which means that the program's vision is merely entertainment. It also includes the design and implementation of the classes that are necessary for users' accounts.

   Finally, the iteration must include an extension of the version from iteration 3 extended with login, including the maintenance of users and their accounts.

   The iteration must also implement the user dialog to start a new game.

5. iteration
   Implementation of the variant *7 Card Stud*. It includes new algorithms corresponding to the algorithms developed for *5 Card Stud*, but it is expected that these algorithms will substantially resemble the algorithms from iteration 2, and part of the algorithms may be used directly. In addition, the iteration will include a change of the user interface corresponding to the fact that each player now has more cards.

6. iteration
   Implementation of *Texas Hold'em*. In the same way as in iteration 5, this iteration includes implementation of new game algorithms, as well as developing a new user interface.

7. iteration
   Implementation of *Omaha*. It is perceived as a simple iteration, which is primarily a matter of modifying the result of iteration 6.

8. iteration
   Implementation of *5 Card Draw*. It is the last variation of the game, and after this iteration, the program is in principle complete. The iteration includes new algorithms corresponding to *5 Card Draw*, including where the virtual players must make decisions as to which cards they wish to exchange. In addition, the variant requires the user interface to change.

9. iteration
   This iteration includes improving the user interface, including colors and fonts. Optionally, a form of settings must also be implemented.

   The iteration also includes a code review and refactoring.

   Finally, the iteration must include the development of an installation script as well as an icon for the program.

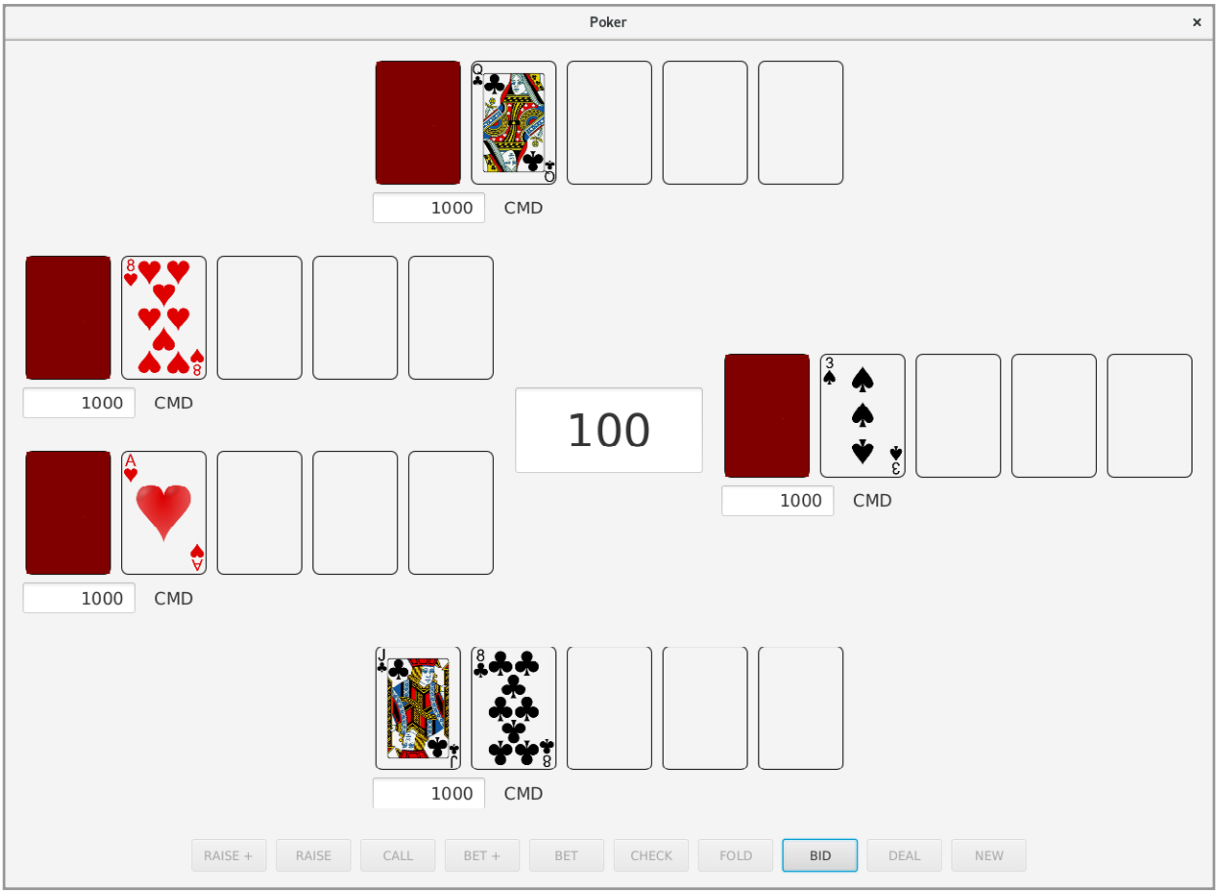
Challenges and risks

In principle, it is a relatively simple program corresponding to the fact that it is a standalone windows program, and it is thus well known how the task should be solved. However, there are two challenges to the program where the solution is not clear and both are crucial to the success of the program and whether there are users who wish to use the program:

1. The user interface must be nice and appealing – whatever it may be. In addition, the user interaction must be easy and intuitive.
2. The virtual player must be implemented with a sensible behavior and as a player you can not immediately win over. It must be difficult (impossible) to figure out how the virtual player acts.

6.4 ITERATION 1

The window below shows the prototype where a *5 Card Stud* game has been started with 5 players and the 5 players are given the first 2 cards:



The bottom player is the user. Below each player there is a *TextField* showing the player's amount of money (the number of chips) and the field in the center is the pot.

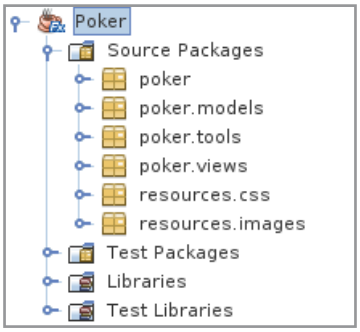
At the bottom, the buttons are the commands that the user must use to play. The meaning is as follows:

- When you start a new game, only NEW is enabled and when the user clicks on it, a new round starts.
- After that, only DEAL is enabled and when clicked, cards is dealt out starting with the player who is after the dealer. In the prototype the user is the dealer. After dealing cards, only the BID button is enabled.
- When you click on BID, a bidding round starts and when it's the turn of the user to bet, the remaining 7 buttons are enabled as they make sense, and it is
  - CHECHK, BET, BET+, FOLD
  - CALL, RAISE, RAISE+, FOLD
- After the player has clicked on one of the buttons they are all disabled and the remaining (virtual) players are betting. Then there are three options:
  1. The betting round has not been completed because a player has raised the deposit. If it is the case, BID will be enabled and the betting round will continue as described above, however, the player may only choose
     - CALL, RAISE, RAISE+, FOLD

2. The betting round is completed and if it is not the last round, DEAL will be enabled and when the user clicks on it the next card is given and BID will then be enabled.
3. The betting round is completed, but it is the last round. Then DEAL is enabled, but the text changes to SHOW, which tells you that the round is over, and when clicking on SHOW, the winner's chips are updated and the pot is cleared. Then NEW is enabled and you are ready for the next round.

You should note that the functionality of these buttons has not yet been implemented.

The project's architecture is preliminary the following, where the package *resources.images* contains images of the playing cards:



resources.css contains a style sheet that for now only contains one style used in this program – in the class *CardView*. *poker.tools* contains a few help classes. One class is called *Images* and is a singleton that makes the images of the cards available. The other class is *CardView* that is a *GridPane* with an *ImageView* and thus a simple component that shows a playing card with a frame around.

With respect to the package *poker.models*, 4 enumerations are defined:

1. *CardColor*, representing the four card colors
2. *CardValue*, representing the 13 card values
3. *PokerGames*, which lists the 5 poker variants
4. *PokerBets*, which indicates the player's bets

and 5 classes. *Card* represents a playing card:

```
public class Card implements Comparable<Card>
{
 private CardColor color;
 private CardValue value;
```

while *Cards* represents a stack of playing cards with 52 cards:

```
public class Cards
{
 public static final Random rand = new Random();
 private Card[] card = new Card[52];
 private int top = 0;
```

These are classes that I have shown before and they work basically the same way as in the program *Poker* from the book Java 4. The following class, which is the beginning of a model for the game itself, is also defined, but it is not yet used to much:

```
public class Game
{
 private ObjectProperty<Integer> size = new SimpleObjectProperty(5);
 private ObjectProperty<Integer> blind = new SimpleObjectProperty(1);
 private IntegerProperty start = new SimpleIntegerProperty(100);
 private ObjectProperty<PokerGames> poker =
  new SimpleObjectProperty(PokerGames.CARD5);
 private Cards cards = new Cards();
```

Finally, there is a class representing the cards that a given player has:

```
public class Hand
{
 private List<Card> cards = new ArrayList();
```

and in addition there is an exception class.

Then there is the *poker.views* package that contains view and presenter classes. Preliminary there is 4:

1. *MainView* and *MainPresenter*, which defines the main window with the buttons and maybe later a toolbar or menu.
2. *HandView* and *HandPresenter*, which shows the individual player's cards and the number of player's chips.
3. *Card5View* and *Card5Presenter*, which defines a pane for the current poker variant.
4. *TableView* and *TablePresenter*, there are basic classes for the above.

The iteration ends with a simple refactoring, which is nothing but a simple cleanup in the code.

6.5 ITERATION 2

The following algorithms have a parameter of the type *Card[]* that represents a player's cards. It is assumed that the cards are sorted in descending order.

The first algorithms are used to determine the value of a player's cards. The algorithms are help algorithms used by the other algorithms.

```
/ Tests if there among the cards are n cards with the same value.
/ The algorithm assumes that the cards not contains more than n
ards with the
/ same value, and if n = 2 that the hand does not contain two pairs.
ard[] sameValue(Card[] c, int n)

put the first card in a list
loop over all other cards
{
 if the card has the same value as the first card in the list
 {
  put the card in the list
  if the list contains n cards return the list
 }
```

```
  else
  {
   clear the list
   put the card in the list
  }
 }
 return the empty list
}

// Tests where the cards have the same color.
boolean sameColor(Card[] c)
{
 for (i = 1; i < c.length; ++i) if (color(c[i]) != color(c[0])) return false;
 return true;
}

// Tests if the cards can be cards for a straigth.
boolean canStraight(Card[] c)
{
 if (two cards have the same value) return false
 test where the difference between the value of the first and the last card
 is less than 5
}

// Tests where the cards is a full house.
// The algorithm assumes that there are 5 cards.
boolean fullHouse(Card[] c)
{
 if (value(c[0]) == value(c[2]) && value(c[3]) == value(c[4]) return true;
 if (value(c[0]) == value(c[1]) && value(c[2]) == value(c[4]) return true;
 return false;
}

// Tests if the cards contains two different pairs.
// The algorithm assumes that there are 4 or 5 cards and the cards
not have
// 3 or 4 of a kind.
boolean twoPair(Card[] c)
{
 if (there are 4 cards)
 {
  if (value(c[0]) == value(c[1]) &&
    value(c[2]) == value(c[3]) &&
    value(c[1]) != value(c[2]) return true
  return false
 }
 else
 {
```

```
  if (value(c[0]) == value(c[1]) &&
   value(c[2]) == value(c[3]) &&
   value(c[1]) != value(c[2]) &&
   value(c[3]) != value(c[4])) return true
  if (value(c[0]) == value(c[1]) &&
   value(c[3]) == value(c[4]) &&
   value(c[1]) != value(c[2]) &&
   value(c[2]) != value(c[3])) return true
  if (value(c[1]) == value(c[2]) &&
   value(c[3]) == value(c[4]) &&
   value(c[2]) != value(c[3]) &&
   value(c[0]) != value(c[1])) return true
  return false
 }
}
```

There will probably be a need for multiple algorithms of similar kind, and as tests of where a sequence of cards may have a certain value, and they will, if appropriate, look like the above and will generally be simple.

In *5 Card Stud*, the player with the best open cards must always start the betting round. There are 4 betting rounds and an algorithm is defined for each betting round, which returns the index for the player to start the betting round. As a parameter, the algorithms have an array of the cards to be tested, and it is assumed that index 0 always indicates the user's hand.

```
// Returns the index of the player to start the first betting round.
int round1(Card[] c)
{
 returns the index of the highest card
}

// Returns the index of the player to start the second betting round.
// Each player has two open cards and the highest value can then be
a pair.
int round2(Card[][] c)
{
 if there is a player with a pair return the index for the player with
 the highest pair
 else return the index of the player with highest card
}

// Returns the index of the player to start the third betting round.
// Each player has three open cards and the highest value can then
be a three
// of a kind, and is it not the case a player can have one pair.
```

```
int round3(Card[][] c)
{
 if there is a player with 3 of a kind return the index for the player with
 the highest cards for 3 of a kind
 else if there is a player with a pair return the index for the player with
 the highest pair
 else return the index of the player with highest card
}

// Returns the index of the player to start the last betting round.
// Each player has four open cards and the highest value can then
be four
// of a kind, and if not three of a kind, and is it not the case
a player
// can have two  pairs and if not so one pair.
int round4(Card[][] c)
{
 if there is a player with 4 of a kind return the index for the player with
 the highest cards for 4 of a kind
 else if there is a player with 3 of a kind return the index for the player with
 the highest cards for 3 of a kind
 else if there is a player with two pairs return the index for the player with
 the highest two pairs
```

```
 else if there is a player with a pair return the index for the player with
 the highest pair
 else return the index of the player with highest card
}
```

At showdown, an algorithm is required that returns the index for the player who has the highest hand:

```
// The parameter is the cards for the payers that have not fold.
The number
// of players is two or more.
int showDown(Card[][] c)
{
 if there is a player with straight flush return the index of the player with
 the highest straight flush.
 else if there is a player with 4 of a kind return the index for the player
 with the highest cards for 4 of a kind
 else if there is a player with full house return the index for the player with
 the highest full house
 else if there is a player with flush return the index of the player with the
 highest flush
 else if there is a player that has a straight return the index of the player
 with the highest straight
 else if there is a player with 3 of a kind return the index for the player with
 the highest cards for 3 of a kind
 else if there is a player with two pairs return the index for the player with
 the highest two pairs
 else if there is a player with a pair return the index for the player with
 the highest pair
 else return the index of the player with highest card
}
```

The last algorithms relates to how the virtual player should act in a betting round, for example, to say how much the player has to bet (BET), and how much when the player raises (RAISE) and where the player should choose CHECK, FOLD and possibly bluff. The player in question will make his choice based on his own cards and the other players' open cards. How the player makes his choice is determined from the current betting round and hence how many cards the player can see and to make it harder to predict the betting, the choice is to some extent random. For each betting round there are two situations:

1. that it is the player who is the first or round's last bet was CHECK, and the player then has four options: BE = BET, B+ = BET double, CH = CHECK, FO = FOLD
2. that a player has already bet, and thus has deposited money in the pot, and in this case there are four options: RA = RAISE, R+ = RAISE double, CA = CALL, FO = FOLD

For each bet, there are two options, such as CH or BE, but such that the first one is chosen with some probability or random. If a situation in the following algorithms determines that the player can choose

10, CH, BE

this means that the player randomly chooses CH in 10% of the cases, while in other cases, the player chooses BE. This means that it can be difficult to predict what the player bets. The following 4 algorithms are all formulated using a five-column table, where the first column indicates a condition and the other columns indicates the player's bet for the two situations. The individual rows should be perceived as if/else so that if a condition is not met then the next one is tested. Note that the virtual player's action can be fine-tuned by introducing more rows in the tables, which will probably also be the case in the implementation, and you should only perceive the following as examples of which conditions the algorithm may contain.

```
// Returns player's bet in the first betting round.
// The value of player's cards kan be one pair or high card.
// The value of others cards kan be high value.
PokerBets round1(Card[] player, Card[][] others)
{
```

Player has one pair				
others have two cards with same value	30	CH, B	10	R, CA
others have one cards with same value	15	CH, B	20	R, CA
others have big cards	10	CH, B	20	R, CA
player has a big pair	30	B+, B	15	R+, R
else	20	B+, B	10	R+, R
Player has high card				
player has two big cards, others have at least 4 cards with the same value	20	CH, B	10	R, CA
player has two big cards, others have cards with the same value	10	CH, B	10	R, CA
player has two big cards	5	CH, B	15	R, CA
player has one big card, others have 3 cards with the same value	10	FO, CH	10	FO, CA

player has one big card, others have 2 cards with the same value	5	FO, CH	5	FO, CA
player has one big card, others have 1 card with the same value	30	CH, B	10	R, CA
player has one big card	15	B+, B	10	R+, R
else	10	B+, B	5	R+, R

```
}
// Returns player's bet in the second betting round.
// The value of player's cards kan be
//    3 of a kind
//    one pair
//    same color
//    can straight
//    high card
// The value of others cards kan be one pair or high value.
PokerBets round2(Card[] player, Card[][] others)
{
```

Player has 3 of a kind				
More than 1 of others have a pair, at least one is bigger than players cards	20	B+, B	10	R+, R
More than 1 of others have a pair	30	B+, B	20	R+, R
else	40	B+, B	30	R+, R
Player has 1 pair				
More than 1 of others have a pair, at least one is bigger than players cards	10	CH, B	10	R, CA
More than 1 of others have a pair	5	CH, B	15	R, CA
1 of others have a pair that is bigger than players cards	10	CH, B	10	R, CA
1 of others have a pair	5	CH, B	5	R, CA
others have the pair card 2 times	10	CH, B	5	FO, CA
others have the pair card 1 time	10	CH, B	10	R, CA
else	10	B+, B	10	R+, R
Player has same color				
more than 1 of others have one pair	20	CH, B	5	R, CA
1 of others have one pair	15	CH, B	5	R, CA
others have at least 5 cards of the color	20	FO, B	10	FO, CA
ohters have at least 4 big cards	15	FO, B	5	FO, CA
else	20	B+, B	10	R+, R
Player can straight				
more than 1 of others have one pair	20	CH, B	5	R, CA
1 of others have one pair	15	CH, B	5	R, CA
ohters have at least 4 big cards	15	FO, B	5	FO, CA
else	20	B+, B	10	R+, R
Player has high card				
more than 1 of others have one pair, one a pair with a higher value than the players cards	20	FO, B	10	FO, CA
more than 1 of others have one pair	10	FO, B	5	FO, CA

1 of others have one pair with a higher value than the players cards	15	FO, B	10	FO, CA
1 of others have one pair	10	FO, B	10	FO, CA
player has big cards, others have the card 3 times	20	CH, B	10	FO, CA
player has big cards, others have the card 2 times	15	CH, B	5	R, CA
player has big cards, others have the card 1 time	10	CH, B	5	R, CA
player has big cards	5	B+, B	5	R+, R
else	20	B+, FO	10	R, FO

```
}

// Returns player's bet in the third betting round.
// The value of player's cards kan be
//    4 of a kind
//    3 of a kind
//    two pairs
//    one pair
//    same color
//    can straight
//    high card
// The value of others cards (they have 3 open cards) kan be
//    3 of a kind
//    one pair
//    high value.
PokerBets round3(Card[] player, Card[][] others)
{
```

Player has 4 of a kind				
More than 1 of others have 3 of a kind, at least one is bigger than players cards	15	B+, B	10	R+, R
More than 1 of others have 3 of a kind	30	B+, B	40	R+, R
1 of others have 3 of a kind, that is bigger than players cards	10	B+, B	10	R+, R
1 of others have 3 of a kind	40	B+, B	40	R+, R
else	30	B+, B	50	R+, R
Player has 3 of a kind				

More than 1 of others have 3 of a kind, at least one is bigger than players cards	20	CH, B	10	R, CA
More than 1 of others have 3 of a kind	10	CH, B	20	R, CA
1 of others have 3 of a kind, that is bigger than players cards	30	CH, B	5	R, CA
1 of others have 3 of a kind	10	B+, B	10	R, CA
More than 1 of others have a pair, at least one is bigger than players cards	15	CH, B	5	R, CA
More than 1 of others have a pair	5	CH, B	10	R+, R
1 of others have a pair, that is bigger than players cards	10	CH, B	10	R, CA
1 of others have a pair	20	B+, B	20	R+, R
else	30	B+, B	30	R+, R
Player has two pairs				

More than 1 of others have 3 of a kind, others have more than one of the players cards	80	CH, B	80	FO, CA
More than 1 of others have 3 of a kind, others have one of the players cards	60	CH, B	60	FO, CA
More than 1 of others have 3 of a kind	40	CH, B	40	FO, CA
1 of others have 3 of a kind, others have more than one of the players cards	70	CH, B	70	FO, CA
1 of others have 3 of a kind, others have one of the players cards	50	CH, B	50	FO, CA
1 of others have 3 of a kind	30	CH, B	30	FO, CA
More than 1 of others have a pair, others have more than one of the players cards	20	CH, B	20	FO, CA
More than 1 of others have a pair, others have one of the players cards	15	CH, B	15	FO, CA
More than 1 of others have a pair	10	CH, B	10	R, CA
1 of others have a pair, others have more than one of the players cards	20	CH, B	20	R, CA
1 of others have a pair, others have one of the players cards	10	CH, B	20	R, CA
1 of others have a pair	5	B+, B	20	R, CA
else	20	B+, B	20	R+, R
Player one pair				
others have 3 of a kind with a bigger value than players	70	FO, B	80	FO, CA
others have 3 of a kind	50	CH, B	60	FO, CA
others have a pair with bigger value than playes	20	CH, B	10	R, CA
others have a pair	10	B+, B	20	R, CA
others have the pair card more than 1 times	20	CH, B	10	FO, CA
others have the pair card 1 time	10	CH, B	10	R, CA
else	20	B+, B	20	R, CA
Player has same color				

others have 3 of a kind	20	CH, B	30	FO, CA
others have a pair and at least 5 cards of the color	15	CH, B	15	FO, CA
others have a pair	10	CH, B	10	FO, CA
others have at least 5 cards of the color	5	CH, B	5	R, CA
else	5	B+, B	5	R, CA
Player can straight				
others have 3 of a kind	20	CH, B	30	FO, CA
others have a pair and 3 of the missing card	15	CH, B	15	FO, CA
others have a pair and 2 of the missing card	10	CH, B	10	FO, CA
others have a pair and 1 of the missing card	5	CH, B	5	R, CA
others have a pair	5	B+, B	5	R, CA
others have 3 of the missing card	15	CH, B	10	FO, CA
others have of the missing card	10	CH, B	5	FO, CA
others have 1 of the missing card	5	CH, B	5	R, CA
else	10	B+, B	15	R, CA
Player has high value				
others have 3 of a kind	100	FO, B	100	FO, CA
others have a pair, others have bigger cards	80	FO, B	100	R, CA
others have a pair	50	FO, B	10	R, CA
others have bigger cards	30	FO, B	10	R, CA
else	20	B+, B	20	R, CA

```
}
// Returns player's bet in the last betting round.
// The value of player's cards kan be
//    straight flush
//    4 of a kind
//    full house
//    flush
//    straight
//    3 of a kind
//    two pairs
//    one pair
//    high card
```

```
// The value of others cards (they have 4 open cards) kan be
//    4 of a kind
//    3 of a kind
//    two paies
//    one pair
//    same color
//    can straight
//    high value.
PokerBets round4(Card[] player, Card[][] others)
{
```

Player has 4 straight flush				
others can have straight flush with a higher value	10	CH, B	10	R, CA
others can have straight flush	30	B+, B	30	R+, R
else	20	B+, B	20	R+, R
Player has 4 of a kind				
others can have straight flush	10	B+, B	0	R, CA

others have 4 of a kind, 1 have bigger cards than player	100	FO, B	100	FO, CA
others have 4 of a kind	40	B+, B	40	R+, R
more than 1 of others have 3 of a kind, at least one is bigger than players cards	15	B+, B	10	R+, R
more than 1 of others have 3 of a kind	30	B+, B	40	R+, R
1 of others have 3 of a kind, that is bigger than players cards	10	B+, B	10	R+, R
1 of others have 3 of a kind	40	B+, B	40	R+, R
else	30	B+, B	50	R+, R
Player has full house				
others can have straight flush	30	CH, B	10	R, CA
others have 4 of a kind	100	FO, B	100	FO, CA
others have 3 of a kind and can have a bigger full house	40	CH, B	15	R, CA
others have 3 of a kind	10	CH, B	30	R, CA
others have two pairs and can have a bigger full fouse	40	CH, B	20	R, CA
others have two pairs	5	B+, B	30	R+, R
else	30	B+, B	40	R+, R
Player has flush				
others can have straight flush and a better straight	50	CH, B	30	FO, CA
others can have straight flush	30	CH, B	10	FO, CA
others have 4 of a kind	100	FO, B	100	FO, CA
others have 3 of a kind	30	CH, B	10	FO, CA
others can have a better flush	20	CH, B	5	R, CA
others can have a flush	10	B+, B	15	R, CA
others have two pairs and can have a full house	20	CH, B	10	R, CA
else	30	B+, B	20	R+, R
Player has straight				

others can have a better straight	30	CH, B	0	FO, CA
others can have a straight	10	B+, B	20	R+, R
others have 4 of a kind	100	FO, B	100	FO, CA
others can have a flush	30	CH, B	10	R, CA
others have 3 of a kind and can have a full house	20	CH, B	10	R, CA
others have 3 of a kind	10	CH, B	10	R, CA
others have two pairs and can have a full house	20	CH, B	10	R, CA
others have two pairs	10	CH, B	10	R, CA
else	30	B+, B	40	R+, R
Player has 3 of a kind				
others can have a straight	20	CH, B	100	R, CA
others have 4 of a kind	100	FO, B	100	FO, CA
others can have a flush	30	CH, B	10	R, CA
others have 3 of a kind and can have a full house	20	CH, B	10	R, CA
others have 3 of a kind	10	CH, B	5	R, CA
others have two pairs and can have a full house	20	CH, B	5	R, CA
others have two pairs and can have a better 3 of a kind	20	CH, B	5	R, CA
More than 1 of others have a pair, at least one is bigger than players cards	15	CH, B	5	R, CA
1 of others have a pair, that is bigger than players cards	10	CH, B	10	R, CA
else	30	B+, B	30	R+, R
Player has two pairs				
others can have straight	20	CH, B	0	FO, CA
others have 4 of a kind	100	FO, B	100	FO, CA
others have 3 of a kind, and player can have bigger cards	30	B+, B	30	R+, R

others have 3 of a kind	100	FO, B	100	FO, CA
others can have a flush	30	CH, B	10	FO, CA
others have a better two pairs	100	FO, B	100	FO, CA
others have two pairs	40	CH, B	10	FO, CA
else	40	B+, B	40	R+, R
Player one pair				
others can have straight or a better pair	20	CH, B	10	R, CA
others can have a straight	10	CH, B	10	R, CA
others can have a flush or a better pair	20	CH, B	10	R, CA
others can have a flush	10	CH, B	10	R, CA
others have 4 of a kind	100	FO, B	100	FO, B
others have 3 of a kind	100	CH, B	100	FO, CA
others have two pairs, player can have 3 of a kind	20	B+, B	30	R+, R

others have two pairs	100	FO, B	100	FO, CA
others have one pair, that is bigger than the players	100	FO, B	100	FO, CA
others have one pair	20	B+, B	10	R, CA
others have the pair card more than 1 times	20	CH, B	10	FO, CA
others have the pair card 1 time	10	CH, B	10	R, CA
else	20	B+, B	20	R, CA
Player has high value				
others have high card, the values less than players cards	10	CH, B	10	FO, CA
others have a pair	50	FO, B	10	R, CA
others have bigger cards	30	FO, B	10	R, CA
else	100	CH, B	100	FO, CA

}

In connection with the implementation and later when testing, there will certainly be changes to the above tables, partly as to which and how many conditions are required, and partly as to the probabilities that will be used. If applicable, the above tables will not be updated. The game's value will largely be determined by how lucky it is with the implementation of these algorithms and it is actually extremely difficult (imposible) to implement the kind of algorithms so that an experienced poker player can not win at all times.

6.6 ITERATION 3

Compared to the prototype from iteration 1, there are no significant changes to the overall design, and the development follows the classic MVP design pattern for JavaFX applications. With regard to the model layer, it is primarily expanded with three new classes. A class *Model* has been added

```
public class Model {
 protected IntegerProperty pot = new SimpleIntegerProperty(0);
 private Game game = new Game();
 protected int value;
 private List<Hand> hands = new ArrayList();
```

The variable *pot* represents the pot on the table and where players deposit amounts (chips) in each betting round. It is defined as a JavaFX property, so the user interface can bind to it. The next variable *game* represents the program's settings as an object of the type *Game*. Initially, there are only 4 properties, but the class may need to be extended at a later time. The variable *value* is the current value of a round that is updated each time a player bet BET or RAISE. The variable thus indicates what it costs to continue in a specific round. Finally, there is the variable *hands*, which is a list of current players and represents their cards.

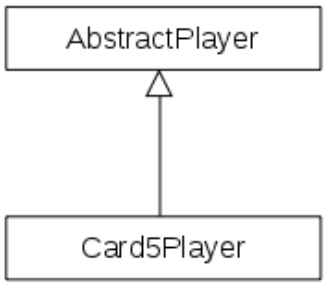
The class's methods are generally simple and consist primarily of *get* and *set* methods. In addition, there are methods for maintaining the variables *pot* and *value*:

```
public int bet()
{
 pot.set(value + getPot());
 return value;
}

public int bigBet()
{
 value *= 2;
 pot.set(value + getPot());
 return value;
}

public int call(int val)
{
 pot.set(value - val + getPot());
 return value;
}

public int raise()
{
 value *= 2;
 pot.set(value + getPot());
 return value;
}

public int bigRaise()
{
 value *= 4;
 pot.set(value + getPot());
 return value;
}
```

The methods are quite simple, but important as these are the methods that the players call in connection with the individual betting rounds. You should especially note the method

call(), which has a parameter. If a player raises and another player has already placed an amount in the pot, it is necessary to subtract this amount, before the pot is updated and that is the goal of this parameter. The above methods thus determines what it costs to play in the individual betting rounds. Similar to the decisions in iteration 1, it is somewhat simplified in relation to "real" poker games, and another purpose of the methods is also that these rules need to be changed by overriding the methods in a derivative class. It is also the reason why the two variables *pot* and *value* are defined protected.

There are also added two classes:



where the class *Card5Player* represents a virtual player who plays *5 Card Stud*. The base class is created because there will be a corresponding class for each of the 4 other poker variants. I will explain both classes below.

Compared to the prototype, the class *Hand* is extended significantly:

```
public class Hand implements Comparable<Hand>
{
 public static CardValue bigCard = CardValue.JACK;
 private List<Card> cards = new ArrayList();
 private List<Boolean> state = new ArrayList();
 private boolean folded = false;
 private int place;
 private int raised = 0;
 private int value = 0;
 private IntegerProperty jetons = new SimpleIntegerProperty(0);
```

The first variable *bigCard* determines what the virtual players should perceive as a big card (JACK, QUEEN, KING and ACE), and the value is used in conjunction with the betting rounds. The value may need to be changed later into an option regarding the program's settings. The next variable *cards* is a list of a player's cards and the variable *state* is used to keep track of whether a card is open (facing up) or closed (facing down). The variable *folded* indicates whether the player has folded and thus dropped out of the round. The variable *place* indicates the player's position at the table, where position 0 is always perceived as the physical player, while the variable *raise* indicates how many times a player has bet RAISE or BIGRAISE in a betting round. The value is used to set a limit for how many times a player must RAISE in the same betting round. Finally, there is the variable *value*, which indicates how many chips the player has already deposited in the pot in the current betting round. The value is used in conjunction with CALL if another player bet RAISE. Finally, there is the variable *jetons*, which indicates the player's amount of chips, and it is defined as a JavaFX property, so that for each player it can bind to the user interface.

Regarding a limit on how many times a player can bet RAISE in a betting round, it is not in agreement with a real poker game, but it is included here as otherwise it is difficult to get the virtual players to act properly. So far, this limit is 2 for the first three betting rounds and 3 for the last betting round, but the limit may need to be changed to an option.

The class has many methods and implements all the algorithms regarding the value of a hand, including the methods that the virtual players must use to determine their bets during a betting round. There are many methods, and many of them are defined as both instance methods and static methods.

For the implementation of the methods in the classes *AbstractPlayer* and *Card5Player* (and also *Card5Presenter*), it is crucial that the methods in the classes *Card*, *Cards* and *Hand* work correctly, and especially the comparison of objects, as the objects in some contexts are compared in ascending order while in other contexts are compared in descending order. Therefore, unit test classes are written for each of the three classes. After these classes have been tested, I can return to the classes *AbstractPlayer* and *Card5Player*.

The base class is so far a simple class:

```
public abstract class AbstractPlayer
{
 public static int MAX = 1;
 protected Model model;
 private int currentValue;
 private int temp;
```

The first constant is the limit of how many times a player must RAISE, and the next variable is the program's model. When the program has to deal out cards, a smaller delay is inserted between each card (so the user can follow the cards being dealt out). The same goes for the bets, where there is a delay between each player's bet – again in order for users to keep track of how the virtual players act (simulates they think). It requires that the algorithms for dealing of cards and the betting rounds are performed in their own threads, which complicates the code a bit to ensure proper user interface update, where the user interface needs to be updated with *Platform.runLater()*. The last two variables in the class *AbstractPlayer* are used in that context.

Otherwise, the class consists primarily of auxiliary methods. The class defines 9 abstract methods:

```
public abstract PokerBets
round1(List<Card> player, List<List<Card>> others, boolean bet, Hand hand);
public abstract PokerBets
round2(List<Card> player, List<List<Card>> others, boolean bet, Hand hand);
public abstract PokerBets
round3(List<Card> player, List<List<Card>> others, boolean bet, Hand hand);
public abstract PokerBets
round4(List<Card> player, List<List<Card>> others, boolean bet, Hand hand);
public abstract int round1(List<Hand> hands);
public abstract int round2(List<Hand> hands);
public abstract int round3(List<Hand> hands);
public abstract int round4(List<Hand> hands);
public abstract int round5(List<Hand> hands);
```

where the first 4 are the methods for the 4 betting rounds. The first parameters are the player's card and the next parameter *others* is the other players' open cards. The third parameter indicates whether it is the first betting round, while the last parameter indicates the current player. The last 5 methods are used to determine who will start a betting round, which is determined by the players' open cards.

The class *AbstractPlayer* will probably change during the course of development, where there will surely be more and other abstract methods.

The class *Card5Player* is an extensive class and consists exclusively of implementing the 9 abstract methods from the base class. Here are especially the first 4 comprehensive corresponding to the algorithms from iteration 2.

In addition to the model classes, there are the classes in the user interface, and here the changes primarily concerns the class *Card5Presenter*. It is also expanded considerably and is until this time the program's most complex class equivalent to the fact that this class contains the entire game logic.

In the same way as in the prototype, the class is derived from

```
public abstract class TablePresenter
{
 protected MainPresenter commands;
 protected Model model;
 protected final TableView view;
 protected int dealer = 0;
 protected int round = 0;
 protected AbstractPlayer player;
```

which now has a reference to an object of the type *AbstractPlayer*. Also note the first variable, which is a reference to the program's *MainPresenter*, and the purpose is that the class *Card5Presenter* should be able to refer to the window's buttons. Which buttons are available depends on who the player is (a virtual player or a physical player). In relation to the prototype, the class is expanded with 7 abstract methods:

```
public abstract   void bet1();
public abstract   void bet2();
public abstract   void raise1();
public abstract   void raise2();
public abstract   void call();
public abstract   void check();
public abstract  void  fold();
```

with one method for each of the 7 possible bets in a betting round.

Then there is the class *Card5Presenter*. The code does not fill much, and the challenges are primarily to make the threads run correctly, and especially how to test the code. For example, RAISE complicates the logic, as RAISE has to go back and let the front sitting players bet again, and as they also can RAISE, there are manys situations to take into account. With regard to tests, it is necessary to test the program and that is to play the game. Here you can find errors and inconveniences, but it can be difficult to track where and how errors occurs. Therefore, I have extended the model layer with an additional class, which represents a form of log file:

```
package poker.models;

import java.io.*;

public class LogFile
{
 private static final String path =
  System.getProperty("user.home") + "/pokerlog/";
```

```
private static LogFile instance;
private BufferedWriter writer;

private LogFile()
{
 try
 {
  writer = new BufferedWriter(new FileWriter(path));
 }
 catch (Exception ex)
 {
  System.out.println("Open Error");
 }
}

public static synchronized LogFile getInstance()
{
 if (instance == null)
 {
  synchronized (LogFile.class)
  {
   if(instance == null) instance = new LogFile();
  }
 }
return instance;
}

public void write(String line)
{
 try
 {
  writer.write(line);
  writer.newLine();
  writer.flush();
 }
 catch (Exception ex)
 {
  System.out.println("Write Error");
 }
}

public void clear()
{
 try
 {
  writer.close();
  writer = new BufferedWriter(new FileWriter(path));
 }
```

```
  catch (Exception ex)
  {
   System.out.println("Clear Error");
  }
 }
}
```

The class is quite simple to understand. You can then let the program write to the log on specific places, which can make it much easier to find out where the program is failing.

Of course, the log file should not be part of the completed program, and the class – and all references in the code – must of course be removed before the application should be used. I have not done so for now. Partly because I can use the class later in the other iterations, and partly because the reader can see how I used the log file.

There is also reasons to warn a little against this technique, where a log file as part of the development process has been used for errors. The risk is that you not remove all references in the code and then the program suddenly fails because you refer to something that no longer exists. In this case, the class *LogFile* is a singleton, and if I delete the class, the compiler will find all references to the class, and I can thus make sure that everything about the log file is removed.

As another thing of the same kind, the program works in that way, that when you select SHOW to see who has won, then all cards will appear – even for those players who have folded. This should not happen and when all cards are shown, it is because some code in the method *deal()* in the class *Card5Presenter* must be replaced by something else:

```
public void deal()
{
 commands.disable(true, true, true, true, true, true, true, true, true, true);
 if (round == 4)
 {
// for (int i = 1; i < model.getHands().size(); ++i)
// {
//  if (model.getHands().get(i).isFolded()) continue;
//  model.getHands().get(i).open(0);
//  view.others[i].views[0].setImage(Images.getInstance().
//    getImage(model.getHands().get(i).getCard(0)));
// }
  for (int i = 0; i < model.getHands().size(); ++i)
  {
   model.getHands().get(i).open(0);
   for (int j = 0; j < model.getHands().get(i).size(); ++j)
    view.others[i].views[j].setImage(Images.getInstance().
      getImage(model.getHands().get(i).getCard(j)));
  }
```

Again, it's a question that I have to remember at one point to remove the bottom *for* loop and replace it with the code being commented out. The purpose is, of course, that in the course of the test I can keep an eye on whether players who fold, also make it on a reasonable decision.

6.7 ITERATION 4

After the end of 3rd iteration, *5 Card Stud* is in principle implemented, at least a user can play this poker variant against 4 other virtual players. In this iteration, the following features must be implemented:

Maintenance of user accounts for each user (player) where you have to register:

- user name, which must uniquely identify the user
- password
- balance and thus how much is in the account
- transactions, when and how much has been added to/used from the account, but not the result of the individual rounds

A user can not play unless there is money on the account.

Login where players at the start of the program must identify themselves as a player with a user name and a password. As part of the feature, it must be possible to create a new account.

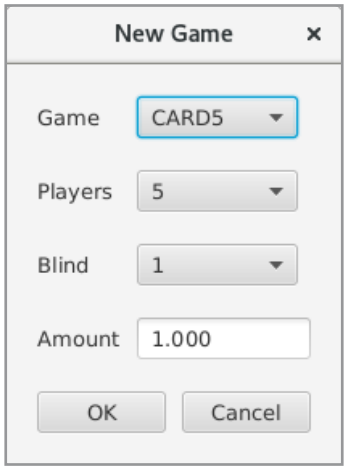
Starting a new game that is a feature that the user can choose at any time. The function opens a dialog box where the user must select:

1. Poker variant (Card 5 Stud, Card 7 Stud, Texas Hold'em, Omaha, Card Draw)
2. Number of players (2–6)
3. Blind (1–10, the number of chips to join a new round, default 1)
4. Start amount (the number of chips for each player, default 1000)

The program must be able to save the result for each round, where the program for each betting round stores the cards and the bets and thus logs the game. It has not been decided what this result should be used for, but it could be used for some form of statistics in order to improve the behavior of the virtual players. The function must be able to be turned on and off and will probably be disabled as default.

Start a new game

This feature is simple to implement and opens the following dialog box:



The dialog opens either from the login window or from a button at the top of the application window. The function can be performed at any time, and you can therefore interrupt an ongoing round.

The implementation consists primarily of adding two classes *GameView* and *GamePresenter* to the views layer and a few changes in the classes *MainView* and *MainPresenter*.

User accounts

It has been decided to completely remove this feature from the program as it does not really make any sense. The argument for the function was that if you play poker for a long time, it might be interesting to follow if you win or lose for a long time, but thinking about the game's idea, it is a facility that will hardly be used. Therefore, the program is reduced with regard to this feature, but also because it is not really trivial to implement the feature if you have to make sure that it also reflects the actual conditions and there are, for example, decisions related to the function as to when to updates the account and the program will not be interrupted without modifications are written back to the disk.

It also means that the login function is dropping away, as the whole user concept is thus uninteresting and thus the user (the player) will always be the current user of the machine.

Due to this decision, this iteration is significantly reduced.

The log file

The log file must be a regular text file, but the content depends on the specific poker variant, and the following concerns *5 Card Stud*. A line must be written in the file in four situations.

Each time a new round is started, a single semicolon separated line containing the date, the game variant, the number of players and the size of a blind and the starting value is written to the file. An example could be:

```
2018:10:05;C5;5;1,1000
```

where the second field is the variant and the value can be:

- C5 = Card 5 Stud
- C7 = Card 7 Stud
- TH = Texas Hold'em
- OM = Omaha
- CD = Card Draw

After each deal, a line is written for each player who are arranged after their position at the table, showing the players' cards sorted in descending order. The colors of the cards are D (Diamonds), H (Hearts), S (Spades) and C (Clubs), and the values of the cards are 2, 3, 4, 5, 6, 7, 8, 9, 0, J, Q, K, A. Is there 5 players the result after the 2nd deal could be:

```
RQ;S7;C2
H4;S3;S2
CA;SA;DA
CK;CQ;CJ
D0;H7;D5
```

For each betting round, one line is written each time a player give a bet, where the player's place at the table is written, the player's bet, how much the player has added to the pot, the game's new value, the player's amount, and the content of the pot. Note that the number of lines in a single betting round may vary, but the result of a single line could be:

```
3;B1;2;4;966;19
```

With regard to players bets, the following terms are used:

- B1 = BET
- B2 = BIGBET
- CH = CHECK

- FO = FOLD
- R1 = RAISE
- R2 = BIGRAISE
- CA = CALL

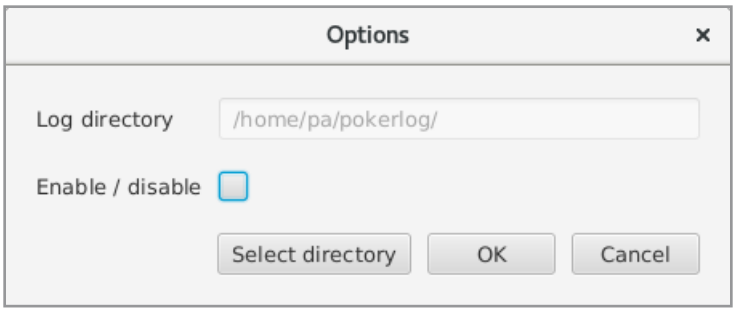
Finally, a line is written at the end of a round showing the result of the player who has won. The line shows the player's place, the hand's value, the player's amount after the game is played and the cards. A result could be:

```
2;FH;1215;C3;S3;D3;SQ;DQ
```

where the following abbreviations are used for the value of a hand:

- SF = Straigth Flush
- C4 = Four of a kind
- FH = Full House
- FL = Flush
- ST = Straight
- C3 = Three of a kind
- TP = Two pairs
- C2 = One pair
- HC = High Card

Since the function should be disabled as default, the program is expanded by the following dialog box:



which maintains the following data structure:

```
package poker.models;

import javafx.beans.property.*;
```

```
public class Options
{
 private BooleanProperty logging = new SimpleBooleanProperty();
 private StringProperty path = new SimpleStringProperty();

 public Options()
 {
  setPath(System.getProperty("user.home") + "/pokerlog/");
 }

 public Options(Options opt)
 {
  setLogging(opt.isLogging());
  setPath(opt.getPath());
 }

 public void copy(Options opt)
 {
  setLogging(opt.isLogging());
  setPath(opt.getPath());
 }
```

```
 public final boolean isLogging()
 {
  return logging.get();
 }

 public final void setLogging(boolean logging)
 {
  this.logging.set(logging);
 }

 public final BooleanProperty loggingProperty()
 {
  return logging;
 }

 public final String getPath()
 {
  return path.get();
 }

 public final void setPath(String path)
 {
  this.path.set(path);
 }

 public final StringProperty pathProperty()
 {
  return path;
 }
}
```

The dialog box itself is defined by two classes *OptionsView* and *OptionsPresenter*. The settings in this dialog should be saved somewhere, and provisionally, the idea is that it should be done by object serialization in the user's home directory. However, this facility is not yet implemented, as it is expected that the program will receive other settings later, and the class *Options* and above dialog box will be expanded accordingly.

It's simple to implement the log file itself using a functionality similar to the class *LogFile* from the previous iteration. This class has now been deleted and replaced by a class *Logfile* that contains a write method for each of the four lines to be written:

```
public class Logfile
{
 private static Logfile instance;
```

```
 private Logfile()
 {
 }

 public static synchronized Logfile getInstance()
 {
  ...
 }

 public void writeLine1(Model model)
 {
  ...
 }

 public void writeLine2(Model model)
 {
  ...
 }

 public void writeLine3(Model model, Hand hand, int value, PokerBets bet)
 {
  ...
 }

 public void writeLine4(Model model, Hand hand)
 {
  ...
 }

 private void write(String line, String path)
 {
  try
  {
   BufferedWriter writer = new BufferedWriter(new FileWriter(path, true));
   writer.write(line);
   writer.close();
  }
  catch (Exception ex)
  {
  }
 }
}
```

Here you should note that the file is opened and closed for each line the program writes to the file. The reason is that function as default is disabled, and idea is that you only use the function if you want to collect information about the program for one reason or another.

Otherwise, the challenge is to get the write operations in question put in the code in the correct places and then of course test.

You should note that to write the cards to the log the class *Hand* is expanded with a single new method, which for a card returns the text to be written in the file.

6.8 ITERATION 5

The next four iterations are similar to each other and are about to implement another poker variant and, in principle, repeat much of the above just for another game. I want to start with *7 Card Stud* simply because this variant looks like *5 Card Stud* a lot. The most important differences are:

1. that each player has 7 cards with three closed and 4 open cards
2. that there are 5 betting rounds
3. that the players at the last two betting rounds have 6 and 7 cards respectively to choose from
4. that it is the player with the lowest card that starts the first betting round

Compared to *5 Card Stud*, the two biggest challenges are:

1. Implementation of the class *Card7Player* as an alternative to *Card5Player*.
2. Implementation of the user interface, since there must now be room for more cards.

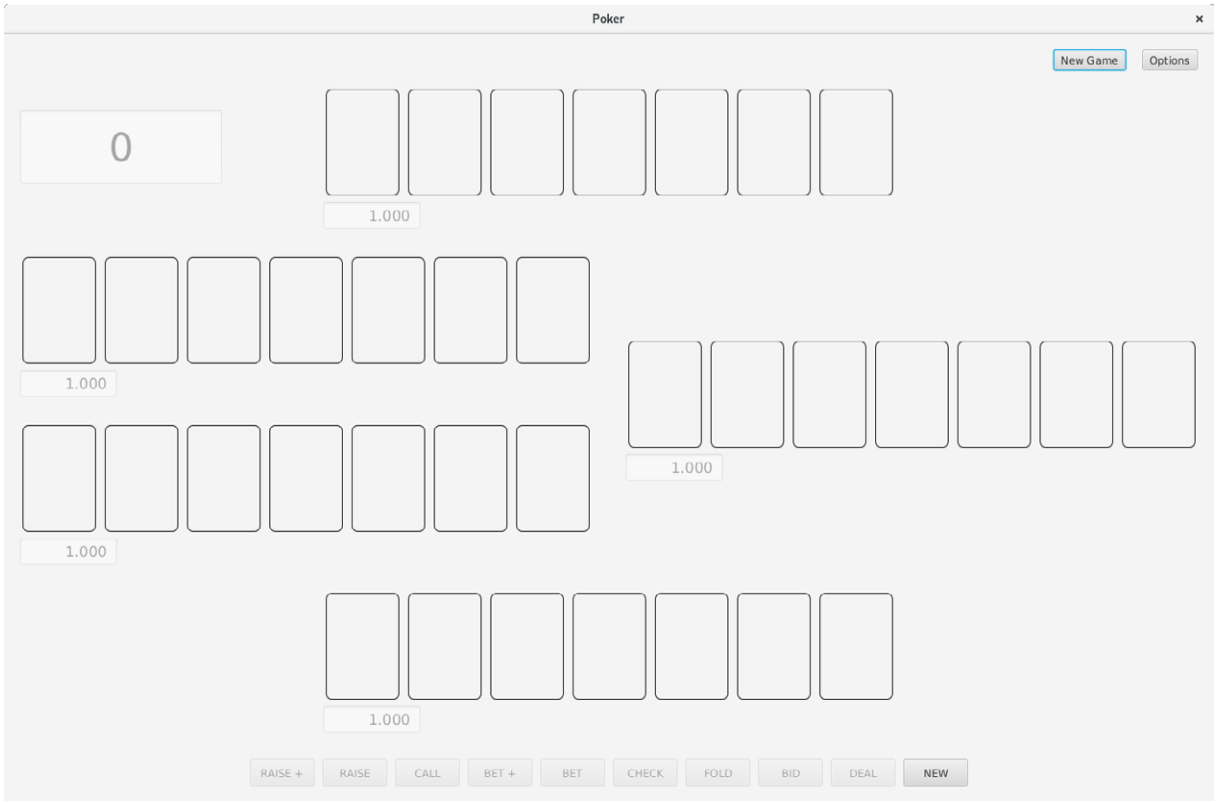
I have started with the last as follows:

- Created two classes *Card7View* and *Card7Presenter* which are copies of *Card5View* and *Card5Presenter*.
- Modified the class *Card7Presenter*, so it instantiates a *Card7View* object instead of a *Card5View* object.
- Changed the class *Game* so it initializes the property *poker* with the value *CARD7*.
- Changed the method *initialize()* in the class *MainPresenter*, so there is now a case entry for *CARD7*.
- In the class *Card7View* I have commented out the line that shows the pot:

```
//    this.setCenter(createCenter());
```

After that, the program is running again, so it uses the new classes *Card7View* and *Card7Presenter* for the user interface, and for each player there is now room for 7 cards (the number of cards to be room for has already been implemented in the base class *TableView*.

I have then modified the class *Card7View*, so the pot moves up in the upper left corner:



Then the user interface is in principle complete.

As a next step, the model has been expanded with a new class *Card7Player*, which so far is just a copy of the class *Card5Player*. In addition, the constructor in the class *Card7Presenter* has been changed, so it instead creates a *Card7Player* object. After that, the program can run again, though it still plays *5 Card Stud*, but the task is now substantially reduced to modify the two classes *Card7Player* and *Card7Presenter*.

As a first step, I have in the class *Card7Presenter* modified methods *deal()* and *deal1()*, so I can now deal 7 cards where the first two and the last are closed.

The class *Card7Player* is extensive, as the algorithms regarding the betting rounds needs to be changed. Since there is now an additional betting round, the base class *AbstractPlayer* is expanded with two abstract methods:

```
public abstract PokerBets
 round5(List<Card> player, List<List<Card>> others, boolean bet, Hand hand);
public abstract int round6(List<Hand> hands);
```

It requires that these methods are also implemented in the class *Card5Player*, where they are simply implemented as trivial methods, for example.

```
public int round6(List<Hand> hands)
{
 throw new UnsupportedOperationException();
}
```

The methods that determines who will start a betting round are largely unchanged, and the changes primarily concerns the five methods that determines the virtual player's bets. In principle, these methods work in the same way as in *Card5Player* and are based on determining a bet from the open cards. This is primarily done by methods in the class *Hand*, but as a player may have more cards (6 or 7) this time, it is necessary to modify some of the methods in the class *Hand* and to add some new ones. These methods are crucial to how the game algorithms works, and it is therefore necessary to test them thoroughly before using them in the algorithms. Therefore, the test classes for the unit test have been expanded and a single new test method has been added. After that, the algorithms in the class *Card7Player* can be implemented according to the same pattern as in *Card5Player*.

The class *Card7Presenter* must of course also be modified. It is actually modest what needs to be changed, and it primarily concerns that the class now has to support 5 betting rounds. In addition, the class is changed, so when showdown alone the 5 cards that the player uses are shown and as always are the cards that give the best hand.

Looking at the classes *Card7Player* and *Card7Presenter* they contains many repeats from the corresponding classes from the previous iteration, and it means that some of the code should later be moved to the base classes. I will postpone that later to a refactoring in iteration 9.

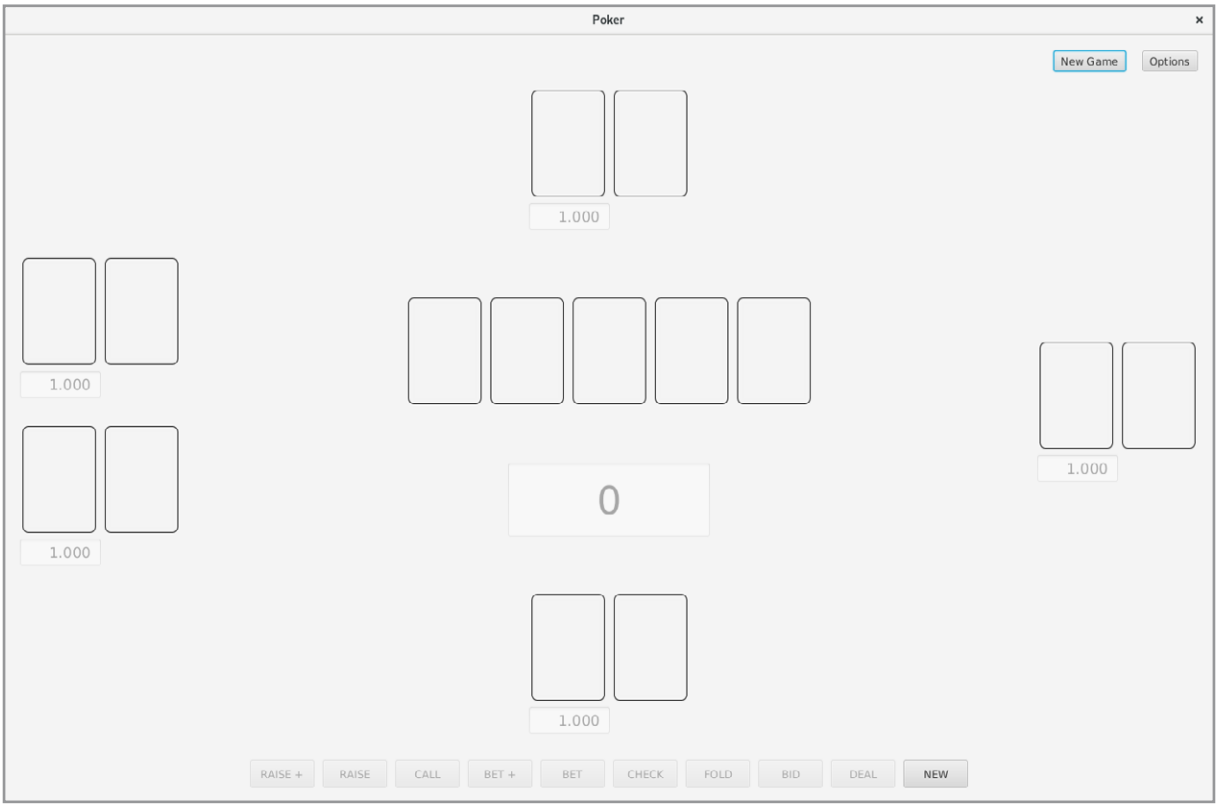
6.9 ITERATION 6

As the next poker variant, I will implement *Texas Hold'em*. Here, in the same way as in *5 Card Stud* there is 4 betting rounds, but the players can, as in *7 Card Stud*, have up to 7 cards, where the two are closed personal cards, while the remaining 5 are open common cards on the table. The challenges in this iteration are a bit like in the previous iteration, but such that the user interface should be somewhat different.

In the same way as in the previous iteration, I have added the following classes:

1. *TexasView*, as a copy of *Card7View*
2. *TexasPresenter*, as a copy of *Card7Presenter*
3. *TexasPlayer*, as a copy of *Card7Player*

Then I have modified the class *Game*, so the default poker variant is *Texas Hold'em*, and I have expanded the method *initialize()* in the class *MainPresenter*. Then the program can run again and use as the default the above classes. As the next step, the class *TexasView* has been changed, so the user interface is as follows:



where each player now has room for 2 cards while there are room for 5 cards on the table.

There is an additional challenge for the user interface, that in one way or another I have to present the cards that are a player's hand after the end of a round. It is solved that way, that after showdown, I replace the above panel with another panel similar to the user interface in *Card7View*. It's a relatively simple solution, which consists simply of adding another two classes:

- *ResultView*
- *ResultPresenter*

and then open the window from the method *deal1()*.

The biggest task is the change of game algorithms in the class *TexasPlayer*. The first algorithms that concern which player should start a betting round should not be used in this variant of poker, so they are all trivial algorithms. The other algorithms that determines a player's

bet are immediately harder to implement appropriately, as a player this time only can see quite a few cards. In the first betting round, the player can only see his own two closed cards, and the rules must therefore be quite simple, quite a few and to a large extent seem random. In the other three rounds, the player can see 5, 6 or 7 cards respectively, where 3, 4 or 5 of the cards are shared with the other players. In the current implementations, the algorithms take into account only which cards the current player has, but the algorithms can be improved by also taking into account which cards the other players can get from the open cards on the table.

6.10 ITERATION 7

The *Omaha* variant is basically similar to *Texas Hold'em*. The implementation includes the following classes:

1. *OmahaPlayer*, as a copy of *TexasPlayer*
2. *OmahaView*, as a copy of *TexasView*
3. *OmahaPlayer*, as a copy of *TexasPresenter*

Next, the following classes are modified corresponding to the other variants:

1. *Game*
2. *MainPresenter*
3. *OmahaPresenter*, where the parameter to *showResult()*, that is called by showdown, must be changed to 9

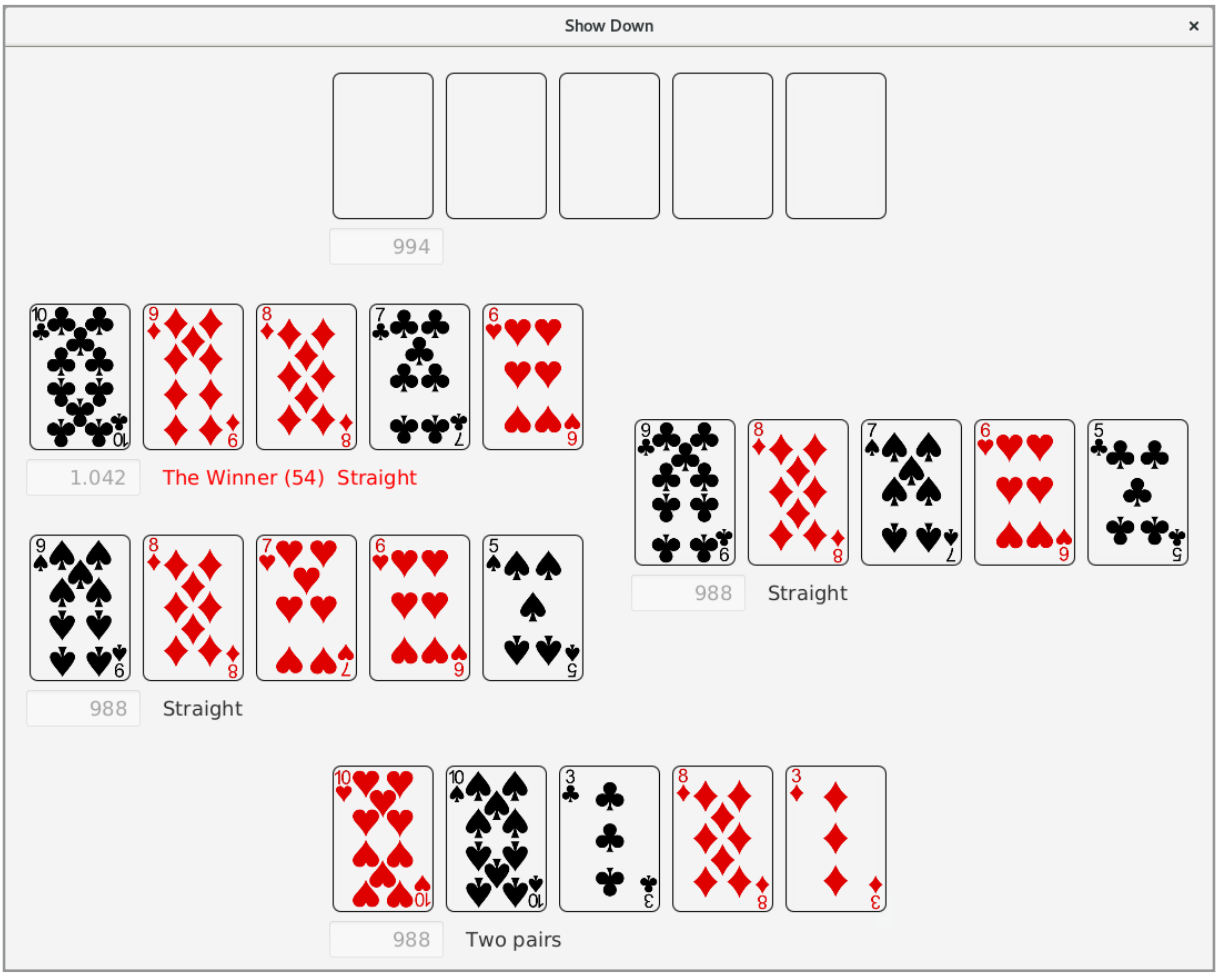
and then the program runs again, but so the program obviously still plays *Texas Hold'em*. However, with regard to the user interface, there is a small problem, because the cards in *ResultView* fill up too much. The biggest task, however, is to change the game rules in the class *OmahaPlayer*. As for who to start the betting rounds there are no changes, as I will apply the same rule as for *Texas Hold'em*, where it is always the player after the dealer who starts the betting. However, it is a little simplified in relation to the real poker world, but I maintain this simple rule as it does not change anything in relation to the current program.

Regarding the algorithms that determines what a player has to bet in a betting round, I have used the same rules as in *Texas Hold'em*, but this time there is an expansion as to which cards may be used.

In the first betting round, only two of the player's four closed cards must be selected, giving 6 options. For each of these 6 options, I choose the two cards that can lead to the

best hand, and on these two cards, the same algorithm is used as in *Texas Hold'em*. In the second betting round, two of the player's closed cards must be selected, which in turn gives 6 options and these two cards are combined with the three open cards on the table, and I choose the combination that results in the highest hand. For this combination, I use the same algorithm as in *Texas Hold'em*. The third betting round goes the same way. There are 2 out of 4 closed cards (6 ways) and then 3 out of 4 open cards (4 ways), giving 24 combinations. Here the best combination is selected and the same algorithm as in *Texas Hold'em* is applied to this combination. Finally, there is the fourth betting round where 3 out of 5 open cards are to be selected, what can be done in 10 ways. In this case, there are 60 combinations, and again the hand that with the highest value is selected.

At showdown players, who have not folded, should show the cards they have chosen and who have won. For each player, it is enough to show 5 out of 9 cards (that's the best hand the player can make) and thus a bit like the *Texas Hold'em* variant, but this time I will show the result in a dialog box:



Perhaps the *Texas Hold'em* variation should be changed to show the result using the same dialog box.

6.11 ITERATION 8

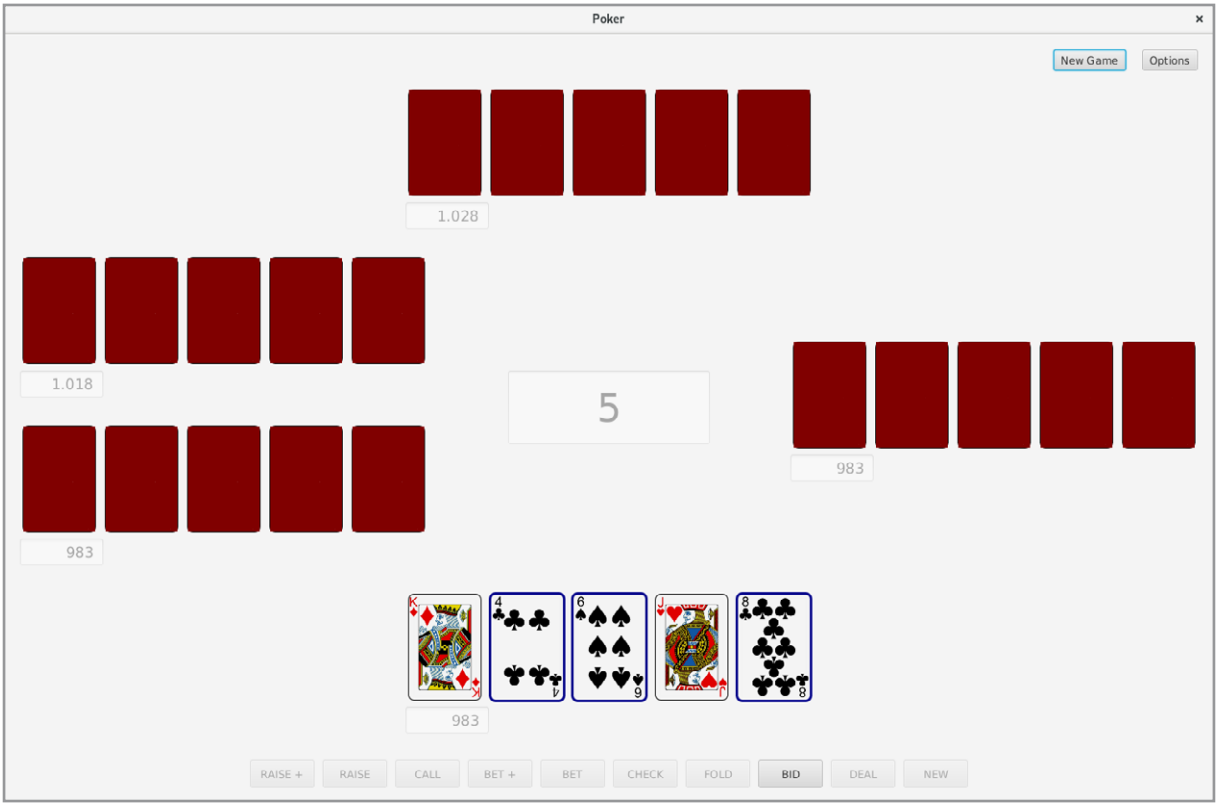
The last variant is *5 Card Draw*, which is also the simplest variant, as there are only two betting rounds, and since the game rules are very simple, a player can always only see his own 5 cards. With regard to bets, I want to use the same algorithm as in *Texas Hold'em*, but with slightly higher probabilities for higher bets. In addition, the first algorithm must be expanded so the players can decide which cards to switch. It is based on a very simple algorithm:

1. If the player has a straight flush, no card is exchanged
2. If the player has 4 of a kind, no card is exchanged
3. If the player has a full house, no card is exchanged
4. If the player has flush, no card is exchanged

5. If the player has straight, no card is exchanged
6. If the player has 3 of a kind two cards are exchanged
7. If the player has two pairs, one card is exchanged
8. If the player has a pair three cards are exchanged
9. If the player has high card, all cards less than a jack are exchanged

These rules are simple and even too simple as it is relatively easy for the user to figure out which cards the virtual players may have. Therefore, the 4 last rules are changed, so the player may exchange a smaller number of cards.

Finally, the physical player must be able to exchange cards and therefore be able to select the cards to be exchanged. This happens when the user clicks on the cards that he wants to exchange:



In order for the user to click on a card, the class *CardView* has been changed so that the frame without the card uses two different styles depending on whether or not the card is clicked:

```
public class CardView extends GridPane
{
 private ImageView view = new ImageView();
 private boolean clickOn = false;
 private boolean clicked = false;
```

```
 public CardView()
 {
  view.setFitHeight(130);
  view.setFitWidth(89);
  add(view, 0, 0);
  getStyleClass().add("cardstyle");
  setMinSize(97, 138);
  setMaxSize(97, 138);
  setPrefSize(97, 138);

  view.addEventHandler(MouseEvent.MOUSE_CLICKED,
new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent event)
    {
     if (clickOn)
     {
      clicked = !clicked;
      if (clicked) getStyleClass().set(0, "markstyle");
      else getStyleClass().set(0, "cardstyle");
     }
    }
  });
 }
```

```
 public void setImage(Image image)
 {
  view.setImage(image);
 }

 public void clear()
 {
  clicked = false;
  getStyleClass().set(0, "cardstyle");
 }

 public void setClickOn()
 {
  clickOn = true;
 }

 public void setClickOff()
 {
  clickOn = false;
 }

 public boolean isClicked()
 {
  return clicked;
 }
}
```

The variable *clicked* is used to indicate whether a card is clicked such I can test for that in the code, while the variable *clickOn* is used to indicate whether this feature is supported. Only the poker variant *5 Card Draw* will support this feature, and only in the first betting round.

In addition, the model class *Hand* is expanded with a variable

```
private boolean[] clicked = new boolean[5];
```

so the user can specify which cards are to be exchanged and the same for the virtual players. In the class *DrawPlayer*, the method *round1()* called by players in the first betting round is expanded so that it initializes the above array with the cards that the player wants to exchange according to the above algorithm and finally the class *DrawPresenter* is expanded to replace the cards as a part of dealing cards in the method *deal()*.

6.12 ITERATION 9

It is the final iteration. In relation to the planning, the iteration has changed a bit, but includes

1. the program's options
2. the user interface
3. code review and refactoring
4. an installation script

It's a relatively big job and I want to start with a code review of the following types:

- The program uses 5 enumerations: *CardColor*, *CardValue*, *PokerValue*, *PokerGames* and *PokerBets*.
- The classes *Card* and *Cards*, that are simple classes but classes that could be used in other applications.
- The class *Game* where there are no changes. In the review of this class, I have also seen at the classes *GameView* and *GamePresenter*, for which there are no changes either.

The program's options

Up to this place there are two options:

1. The directory to be used for the application's log file when logging the individual game rounds.
2. Where this functionality is disabled or enabled.

It has been decided to expand with two new options. A player can RAISE twice in the same betting round and three times in the last betting round. It has been decided that it should be an option so that a player at the first betting rounds can RAISE 1, 2, 3, 4 or 5 times while the player at the last betting round can RAISE one more time. Finally, it has been decided that the program should have the option of error logging, and whether this feature is enabled or disabled must also be an option.

Similarly, the class *Option* is expanded with two new properties:

```
public class Options
{
 private BooleanProperty errors = new SimpleBooleanProperty();
 private BooleanProperty logging = new SimpleBooleanProperty();
 private StringProperty path = new SimpleStringProperty();
 private ObjectProperty<Integer> raises = new SimpleObjectProperty(2);
```

and the methods are changed accordingly. Then the classes *OptionsView* and *OptionsPresenter* are to be adjusted, which is immediately simple. In the view class there is not much to note but the settings must be saved somewhere and it is decided that it should be in a file *poker.init* in the user's home directory and that it should be done by simple object serialization. It has been decided that the settings should be saved each time they are changed. However, it does have a small problem, since JavaFX properties are not immediately serializable. Therefore, a small class has been written that encapsulates these properties so that you can serialize an object of this class.

```
public class PokerOptions implements java.io.Serializable
{
 private boolean errors;
 private boolean logging;
 private String path;
 private int raises;

 public PokerOptions()
 {
 }
```

```
 public PokerOptions(Options opt)
 {
  errors = opt.isErrors();
  logging = opt.isLogging();
  path = opt.getPath();
  raises = opt.getRaises();
 }

 ...

 public Options getOptions()
 {
  Options opt = new Options();
  opt.setErrors(errors);
  opt.setLogging(logging);
  opt.setPath(path);
  opt.setRaises(raises);
  return opt;
 }
}
```

With this class available, it is easy to extend the class *OptionsPresenter* with a method that serializes a *PokerOptions* object with the program's settings when clicking on the *OK* button in *OptionsView*. Similarly, the class *Model* is expanded with a method *deserialize()*, which, if possible, deserialize a *PokerOptions* object and instantiate an *Options* object.

In terms of error logging, it must occur in a text file called *poker.error* in the user's home directory. The functionality is implemented by the following class:

```
package poker.tools;

import java.io.*;

public class Error
{
 private static final String path =
  System.getProperty("user.home") + "/poker.error";
 private static Error instance;

 private Error()
 {
 }

 public static synchronized Error getInstance()
 {
```

```
  if (instance == null)
  {
   synchronized (Error.class)
   {
    if(instance == null) instance = new Error();
   }
  }
  return instance;
 }

 public void write(String message)
 {
  try
  {
   BufferedWriter writer = new BufferedWriter(new FileWriter(path));
   writer.write(Utils.getDate() + ": " + message);
   writer.newLine();
   writer.close();
  }
  catch (Exception ex)
  {
  }
 }
}
```

and back is just to call the method *write()* if the functionality is enabled. An example could be:

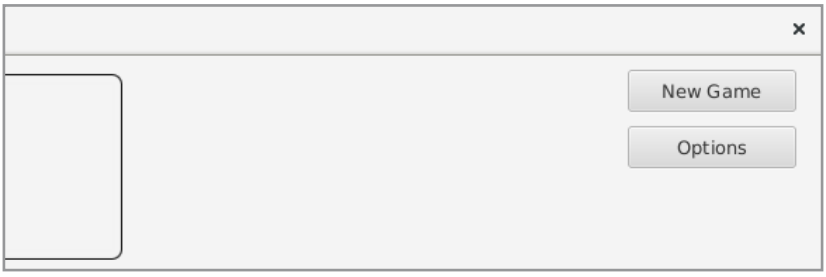
```
catch (Exception ex)
{
 if (model.getOptions().isErrors())
  poker.tools.Error.getInstance().write(ex.getMessage());
}
```

which is an error handling in the class *OptionsPresenter*. Back then, the error handling has to be implemented, where relevant, which happens in connection with the following code review and refactoring.

The user interface

The user interface should be improved in some places (and can certainly be improved in many places). The program, as it is now, runs well on a high resolution display (1920 × 1080), but on a screen with significantly lower resolution, there is no room for all the cards. The following is primarily about making the user interface dependent on the current screen resolution and at the same time depending on the current variation of the game.

The program has two buttons in the top right corner of the window that are used to open the dialog boxes for game selection and the settings. These two buttons have I moved down to the actual game view, where they are located to the right of the top hand.



The goal is to save a little space at the top. On the other hand, the change must be implemented in all 5 game views. For that in the package *poker.tools*, the following component has been added:

```
public class MenuView extends VBox
{
 private Button cmdGame = new Button("New Game");
 private Button cmdPoker = new Button("Options");
```

```
 public MenuView(MainPresenter presenter)
 {
  super(10);
  cmdGame.setOnAction(e -> presenter.newGame(e));
  cmdPoker.setOnAction(e -> presenter.setOptions(e));
  getChildren().addAll(cmdGame, cmdPoker);
  cmdGame.setPrefSize(120, 30);
  cmdPoker.setPrefSize(120, 30);
  setPadding(new Insets(10, 20, 0, 20));
 }
}
```

Here you should first notice the parameter of the constructor, which is a *MainPresenter*, so that the two buttons can call event handlers in this class. In *MainView*, the two original buttons are removed, and the class *MainPresenter* is modified accordingly. The above component must be inserted into the 5 game views. It is resolved to add the following method to the base class *TableView*

```
protected Pane createTop(int n, MainPresenter presenter)
{
 Label label = new Label();
 label.setPrefSize(160, 80);
 BorderPane pane =
  new BorderPane(others[n + 1], null, new MenuView(presenter), null, label);
 return pane;
}
```

and the method is then used to insert the component into *Card5View*, which is a *BorderPane*:

```
setTop(createTop(n, presenter));
```

Then, the same change must take place in *Card7View*, *TexasView*, *OmahaView* and *DrawView*.

Then I have reduced the font size of the *TextField* components (the class *HandView*) that shows the players' amount and the same applies to the pot. The class *Card5View* has a method called *createCenter()* that creates a component for the pot. This method I have moved to *TableView* and called it *createPot()*. Next, it must be changed in *Card7View*, *TexasView*, *OmahaView* and *DrawView*, and the result is that the pot is displayed alike and in the same place (in the middle of the window) in all 5 variants.

Each time the user selects a new game, the method *newGame()* is performed in the class *MainPresenter*. When this method is performed, a new view is created, and you can customize the window size to the current game. What determines the size of a view is primarily the cards, and you can reduce the window size requirements by compressing the cards. For that reason, the class *CardView* is expanded with 4 static variables:

```
public class CardView extends GridPane
{
 public final static double WIDTH = 97;
 public final static double HEIGHT = 138;
 public static double width = 97;
 public static double  height = 138;
 private ImageView view = new ImageView();
 private boolean clickOn = false;
 private boolean clicked = false;

 public static void reset()
 {
  width = WIDTH;
  height = HEIGHT;
 }
```

where the first two are constants which indicates the default size of a card. The other two variables can be changed if you want to scale the image size and they are then used to define the size of a component. They are also used in the class *HandView* to define the size of the component for a hand. As a result, how much the cards now fill on the screen are determined by the two variables *width* and *height*.

The class *CardView* is additionally expanded with the following static method:

```
public static void setSize(double width, double height, double sizeX
double sizeY,
 double marginX, double marginY, int cardsX, int cardsY, Stage stage)
{
 if (width < sizeX || height < sizeY)
 {
  double scale = Math.min(Math.min((width - marginX) / (cardsX * WIDTH),
    (height - marginY) / (cardsY * HEIGHT)), 1);
  CardView.width *= scale;
  CardView.height *= scale;
  stage.setWidth(cardsX * CardView.width + marginX);
  stage.setHeight(cardsY * CardView.height + marginY);
 }
 else
 {
  CardView.width = WIDTH;
  CardView.height = HEIGHT;
  stage.setWidth(sizeX);
  stage.setHeight(sizeY);
 }
}
```

which is used to initialize the two static variables and calculate the window size, where the window is represented by the last *Stage* object. Both parts depend on the current screen resolution and the poker variant to be played. The method's first two parameters indicates the screen resolution, while the two next indicates how much space that view requires. The next two parameters indicates how much space is to be used horizontally and vertically beyond the cards, and thus the amount of space there is no need to scale and finally the next two parameters indicates the number of cards, respectively, horizontal and vertical. The last parameter refers to the window whose size is to be changed.

The class *MainPresenter* has a method *setSize()* which uses the above method to determine the window size:

```
private void setSize()
{
 CardView.reset();
 int size = model.getGame().getSize();
 Rectangle2D bounds = Screen.getPrimary().getVisualBounds();
 double width = bounds.getWidth();
 double height = bounds.getHeight();
 if (size == 2)
 {
```

```
  switch (model.getGame().getPoker())
  {
   case CARD7:
    CardView.setSize(width, height, 1070, 560, 390, 280, 7, 2, parent);
    break;
   case TEXAS:
   case OMAHA:
    CardView.setSize(width, height, 930, 735, 380, 320, 5, 3, parent);
    break;
   default:
    parent.setWidth(930);
    parent.setHeight(560);
  }
 }
 else if (size == 3 || size == 4)
 {
  switch (model.getGame().getPoker())
  {
   case CARD7:
   CardView.setSize(width, height, 1650, 680, 290, 280, 14, 3, parent);
    break;
   case TEXAS:
    CardView.setSize(width, height, 1025, 735, 150, 320, 9, 3, parent);
    break;
   case OMAHA:
    CardView.setSize(width, height, 1430, 735, 170, 320, 13, 3, parent);
    break;
   default:
    CardView.setSize(width, height, 1240, 680, 270, 140, 10, 3, parent);
  }
 }
 else
 {
  switch (model.getGame().getPoker())
  {
   case CARD7:
    CardView.setSize(width, height, 1650, 850, 290, 300, 14, 4, parent);
    break;
   case TEXAS:
    CardView.setSize(width, height, 1025, 850, 150, 300, 9, 4, parent);
    break;
   case OMAHA:
    CardView.setSize(width, height, 1430, 850, 170, 300, 13, 4, parent);
    break;
   default:
    CardView.setSize(width, height, 1240, 850, 270, 300, 10, 4, parent);
  }
 }
 parent.centerOnScreen();
}
```

The method determines the size of the window based on the number of players and thus the amount of what the cards filled up corresponding to how many players should be presented at the table. It is divided into 2 players, 3 or 4 players and 5 or 6 players, and for each division, the size is determined based on the current poker variant.

With the above change, the window adjusts to the current screen resolution. If you play *Omaha*, the program opens a dialog at showdown, and this dialog should be configured in the same way that it also takes into account the screen resolution. It requires that the class *OmahaPresenter* has a *setSize()* method similar to the above, but it is much simpler as the dialog always displays 5 cards for each hand.

If you play *Texas Hold'em*, another view opens at showdown, but it is preferable to use the same dialog box as above. This means that classes *TexasPlayer* and *TexasPresenter* need to be changed. In *TexasPlayer*, the method *round4()* is called at the last bidding round, which should be adjusted according to the corresponding method in *OmahaPlayer*, and in *TexasPresenter*, the dialog must be opened instead of the window *ResultView*. Then the two classes *ResultView* and *ResultPresenter* can be deleted.

As a last change regarding the user interface, in *Tesax Hold'em* and *Omaha* you can not see if a player has folded. To solve this problem, I have added a back side card with a different color, then used it to display the cards for a player who has folded. For the time being, it is only implemented in connection with *5 Card Stud* but it must be implemented for the other variants in connection with the following refactoring.

Code review and refactoring

Next activity is code review combined with refactoring, and that means I'm going through all classes and review and adjust the code. It is an extensive work. There are 46 classes divided into 3 packages, and the following is a brief overview of what has been done. I want to start with the classes in the package *views* and the work will therefore indirectly include a test of the application's user interface.

The four classes *GamePresenter*, *GameView*, *OptionsPresenter, OptionsView* and the related interface *IGameUpdater* have indirectly been through a review in this iteration so it is all in place. The classes *HandView* and *HandPresenter* are simple classes where a review also does not mean changes.

The classes *MainView* and *MainPresenter* are important classes, but reverse relatively simple classes, and the result of a review is nothing but simple cleanup in the code.

Back there are the classes for the 5 views for poker variants, which are at the same time the key classes for the user interface. The 5 view classes *Card5View*, *Card7View*, *TexasView*, *OmahaView* and *DrawView* are all simple and there's nothing else to do than a simple cleanup. The same applies to the base class *TableView*, as methods from the derived classes already earlier in this iteration have been moved to the base class.

Looking at the 5 presenter classes *Card5Presenter*, *Card7Presenter*, *TexasPresenter*, *OmahaPresenter* and *DrawPresenter*, they are very similar that indicates that a significant part of the code can be moved to the base class. Methods that in the 5 classes are the same can and should be immediately moved to the base class, where you may need to change the visibility from *private* to *protected*. Then there are methods that are almost the same and here the choice is not quite so easy, and here it is always important to keep in mind that the primary reason is to increase maintenance. Are methods almost the same, there are basically three options:

1. You can create a method in the base class with the same signature as the current method and which implements the common code. In the sub-classes, you can then implement an override that calls the base class method and also has the code that the is specific for the sub-class.

2. You can move the method to the base class and then add conditions that take into account for the specific classes.
3. You can let the methods be as they are, even if that means that the same code is duplicated.

Each option has its pros and cons, and it's very accurate to decide what's right, but overall I prefer duplication of the code (and thus the last option) rather than complicating the code (what the middle option tends to). In the present case, I have actually applied all three options.

The 5 classes also have methods that are used only in some of the classes. If so, one might consider moving the methods out as static methods in an auxiliary class. The class *Utils* in the package *poker.tools* is an example of such a class.

The result of the above refactoring is that the class *TablePresenter* has grown significantly, while the 5 concrete view classes now do not fill a lot.

Then there is refactoring of the model, where there are 20 classes (types), and here I have already at the start of this iteration carried out a review of 10 of these types. Back there are the classes:

- *Model*
- *Hand*
- *AbstractPlayer*
- *Card5Player*
- *Card7Player*
- *TexasPlayer*
- *OmahaPlayer*
- *DrawPlayer*
- *Logfile*
- *PokerException*

where there is nothing to notice about the last one. The same goes for the class *Model*, which is a central class, but also a very simple class.

The class *Hand* is both comprehensive and important, and for the rest of the program it is crucial that this class works correctly. In connection with code review, nothing but ordinary cleanup has been done, but the class has a design problem similar to a low cohesion. The problem is that the last two variables:

- *selected*
- *clicked*

that does not make sense for all variants of poker. The first one is only used in conjunction with *Texas Hold'em* and *Omaha*, while the latter is only used in conjunction with *CardDraw*. The problems are that you can refer to the variables in a context where they are not initialized or meaningful, and in particular, future maintenance of the program may be more difficult. The problem could be solved by introducing two derived classes, and then using these classes in the current variants. In the present case, I have omitted this change as it will mean changing some other classes and the problem solved solely with a comment in the code.

Regarding the log file, there are also no important changes. However, a single adjustment has been made in the method *writeLine2()* to support *Texas Hold'em* and *Omaha*.

Back there are the 6 player classes

- *AbstractPlayer*
- *Card5Player*
- *Card7Player*
- *TexasPlayer*
- *OmahaPlayer*
- *DrawPlayer*

where the first is the basic class for the rest. A refactoring basically consists of the same as above with the view classes, where methods can be moved from the specific classes to the base class, but there are fewer methods that can be moved this time.

Another thing is the algorithms, where there is room for improvement, but I've seen only on *Texas Hold'em* and *Omaha*. As for the first one, the algorithm looks only on the player's own card, but does not take into account what cards the other players can form from the cards on the table. For example, if there are three cards on the table (what is the case in the second betting round), each of the other players can form a hand using two of 47 other cards (47 = 52 – the cards of the current player – the three cards on the table). The other players have therefore

- 47 * 46/2 = 1081 possibilities

to form a hand that contains the three cards on the table. For example, if there are 4 and 5 cards on the table, respectively, the numbers are respectively

- (46 * 45 / 2) * 4 = 4140 possibilities
- (45 * 44 / 2) * 10 = 9900 possibilities

In order to determine what the player has to bet, instead, you can use a strategy that determines the value of the player's hand (out of 5, 6 or 7 cards) and then determines how many of the other players' options are better or worse than the player's chosen hand. For example, if there are 5 cards on the table and you decide which cards the player should choose, and then you can do the following where *player* is a *List<Card>* with the 7 cards (the player's two closed cards and the 5 on table) in descending order:

```
List<Hand> list = new ArrayList();
for (int i1 = 0; i1 < 2; ++i1)
 for (int i2 = i1 + 1; i2 < 3; ++i2)
  for (int i3 = i2 + 1; i3 < 4; ++i3)
   for (int i4 = i3 + 1; i4 < 5; ++i4)
    for (int i5 = i4 + 1; i5 < 6; ++i5)
     list.add(Utils.createHand(hand.getPlace(),
       player.get(i1), player.get(i2),
       player.get(i3), player.get(i4), player.get(i5)));
Collections.sort(list);
player = list.get(list.size() - 1).cards();
```

and here c*reateHand()* is a simple method that creates a *Hand* object. With the player's card available, the following method counts the frequency of hands that the other players can form from the 5 cards on the table and two other cards (which may not be the 7 cards the player knows):

```
protected double weight(Hand hand, List<Card> player)
{
 List<Card> table = model.getTable().cards();
 List<Card> skip = Utils.merge(hand.cards(), table);
 List<List<Card>> list = new ArrayList();
 Cards cards = model.getGame().getCards();
 for (int i = 0; i < cards.size(); ++i)
 {
  if (skip.contains(cards.getCard(i))) continue;
  for (int j = i + 1; j < cards.size(); ++j)
  {
   if (skip.contains(cards.getCard(j))) continue;
   for (int i1 = 0; i1 < table.size() - 2; ++i1)
    for (int i2 = i1 + 1; i2 < table.size() - 1; ++i2)
     for (int i3 = i2 + 1; i3 < table.size(); ++i3)
     {
       List<Card> l = new ArrayList();
       l.add(table.get(i1));
       l.add(table.get(i2));
       l.add(table.get(i3));
       boolean ok = false;
       for (int k = 0; !ok && k < l.size(); ++k)
         if (cards.getCard(i).compareTo(l.get(k)) > 0)
         {
           l.add(k, cards.getCard(i));
           ok = true;
         }
       if (!ok) l.add(cards.getCard(i));
       ok = false;
       for (int k = 0; !ok && k < l.size(); ++k)
         if (cards.getCard(j).compareTo(l.get(k)) > 0)
         {
           l.add(k, cards.getCard(j));
           ok = true;
         }
       if (!ok) l.add(cards.getCard(j));
       list.add(l);
     }
  }
 }
```

```
 double n = 0;
 for (List<Card> lst : list) if (Hand.compare(lst, player) < 0) ++n;
 return n / list.size();
}
```

The method returns a value between 0 and 1, and a large value (close to 1) indicates that the player has good cards and therefore can bet high. With this method available, the three bet methods can be written differently, where the value is determined based on the return value of the above method.

You can use the same strategy in connection with *Omaha*, just be aware that the hand that a player can form is in another way, and that you get higher frequencies, as *Omaha* generally gives results with a greater value.

The last thing

There are still a few things that are missing. Firstly, there is a test of whether a player has money to deposit in the pool. If this is not the case, the player must leave the game and thus be out of the current game. In principle, it is quite simple to implement this facility, and it is primarily a matter of adding a variable to the Hand class:

```
private boolean lost = false;
...

public boolean isLost()
{
 return lost;
}

public void setLost()
{
 folded = lost = true;
 setJetons(0);
}
```

If a player bet and does not have chips enough, this variable is set to true (the method *setLost()* is called). Then the number of the player's chips is shown as 0, and the player folds the cards. In addition, the player in subsequent rounds must not be assigned cards.

If it is especially the user who has lost his chips, it has been decided that the game is over and you get the following dialog box:

After that, the user can choose to start a new game, but closing the dialog by clicking on the cross will close the program.

Then there is the error logging where code should be written, that write to the log file, and as mentioned earlier, it is important that only important information is stored, and that is methods as you find critical. As an example, below is shown a line from the log file (provoked by division with 0):

```
2018:11:16: CARD5, 5, 1:      class poker.views.Card5Presenter.deal1():
 java.lang.ArithmeticException: / by zero
```

Finally, some deficiencies and forgettings have been corrected and, as the last one, there is the installation script.