Högskolan I Halmstad
Advanced Object Oriented Programming
DT4014

Final Project

A Game Framework for ***Sokoban***

Michael Forschlé - 19890731-4515

Supervisors:
Wojciech Mostovski
Sina Entekhabi

Halmstad, Saturday 10th July, 2021

Abstract

The build and use of frameworks is one technique of efficient software application development in an object oriented approach. A good framework provides a comprehensive and qualitative set of structures to give a frame for a software project that is both well-defined but still flexible. As many applications are based on the same or similar concepts, it is unnecessary to frequently "re-invent the wheel" while at the same time lack the safety and robustness of a highly-tested, well-proven and intensively debugged framework. Games are no exception to be often highly suitable to use a (game) framework that already provides templates for many universal requirements like a menu structure, a canvas for the game visualization and control structures, all of which are adjustable for the individual case.
This report shall give a compact overview of a created game framework in the course of *Advanced Object Oriented Programming*. The framework was then used to implement the classical game *Sokoban* to give the option of testing and demonstrating the functionalities of the game framework in a motivational and entertaining way. The code design and some features, as well as other learnings are summarized and shall accompany and illustrate the project code that can be downloaded from the public GitHub repository at Framework for Sokoban.

Table of Contents

1. Introduction

To gain a deeper understanding of OOP (Object Oriented Programming), Halmstad University created for its computer science students and related programs the course *Advanced Object Oriented Programming*. Additionally to the lectures and labs, the students were instructed to prove their skills in object oriented code development in a final task by programming a game framework and implement the game *Sokoban* or a game of comparable format that uses this framework. And what is better to motivate students then to program a game?!
Given a maximum degree of freedom for the way how to implement game and framework, the initial decision was made to program a game framework that was related to a framework concept shown in a paper by [Christensen and Caspersen, 2002] that was introduced in a course assignment. This framework should at least provide a structure to fit in and visualize a level design and to process user inputs to navigate through a game level to successfully finish it. As the project work had to take place exclusively on weekends due to a travel and labour intensive full-time employment, the personal requirements were kept intentionally low to carry out code development, presentation and project report in a reasonable amount of time. Depending on the progress and remaining time available, more and more framework functionalities and game content should be added. A valuable online resource for the learning and development process of a framework in *Java*, especially during the implementation of more and more optional functionalities were the teaching notes of a professor of the NTU in programming (see [NTU, 2021]). The following chapter 2 presents the fundamental design of the final game framework and the *Sokoban* implementation and chapter 4 digs a bit deeper by showing some of the challenges during the framework and game development and the implemented solutions.
After the basic structures of the game framework were set, the code was continuously tested in parallel to the integration of the *Sokoban* game code. The tests followed a prioritization approach that was also the base of the development process itself and can be read further in chapter 3.
While the final range of functionalities of both the game framework as well as the *Sokoban* game could surpass the minimum expectations by a significant amount, at least equally important was the question of actually applied OOP concepts according to the teaching objective of the course. While the described diagrams in chapter 2 and 4 try to illustrate the object oriented approach, this goal is also discussed and evaluated in chapter 5 together with a total summary of the project work, achievements and potential for further improvement through changes or extensions.
Finally, the last chapter 6 adds some personal experiences with version control in Git and GitHub which was a side task for the project but had no influence on the code development from a content specific perspective and has therefore more the character of an appendix.

2. Design

This chapter clarifies the design structure of the project by presenting the central elements of the game framework first, visualizing the hierarchies and connections of components and functionalities as well as showing some source code parts related to the illustrations. After introducing the game framework as a foundation, the very essential additions of the *Sokoban* game, which was implemented to test the game framework, are described together with some more illustrations.

2.1 Game Framework

This section presents the key components of the game framework from a source code perspective. The described components contain or call all major elements that build the structure of the framework and can or must be modified to certain degrees for any implemented game but shall provide the very same basic functionality and a basic structure to use for every programmer. The central component is the *GameMain* class which contains the declarations of the general framework variables and hosts all other framework classes and their sub-functions. Next follows a presentation of the *GameCanvas* class which provides the central visualization and administrates the necessary customization for the visualization, but also for the player input processing. The last part explains the setup of the *GameMenu* bar which improves the user friendliness by adding some very common features.

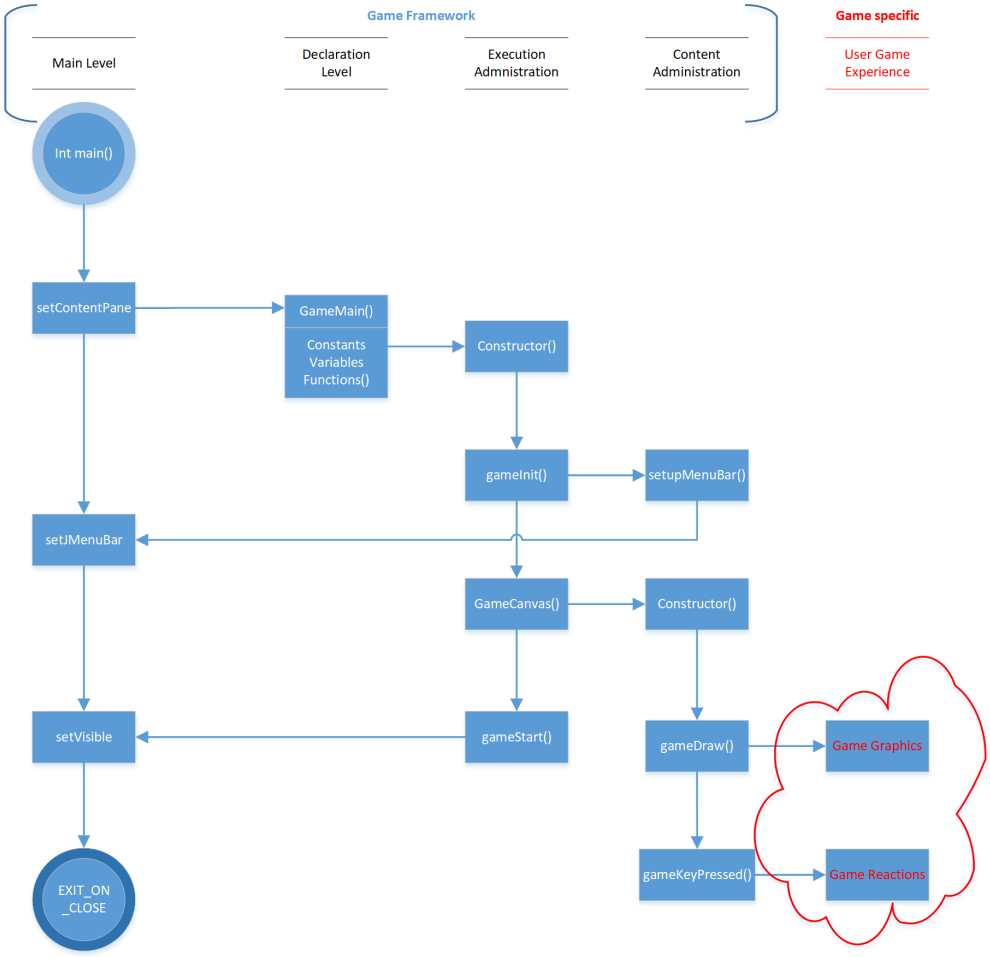

Figure 2.1: Framework Overview

2.1.1 GameMain

The GameMain class is, as the name implies, the central component of the game framework. It extends the *JPanel* class and can therefore be added as content to a *JFrame* by the actual *main* function. As a host for all framework related code, it provides the frame for the project. The hierarchy of the framework classes and functions that are embedded in GameMain is illustrated in diagram 2.1. Two of them, *GameCanvas* and *GameMenu* (initialized by *setupMenuBar()*), will be presented separately afterwards.

Furthermore, GameMain declares the top-level variables, parameters and enumerations of the game framework right at the top. These are separated in

- game constants, defined as *static final* < *type* >
- game variables, private but (over)written during game execution
- special types like *static enum* or special components/classes

E.g. declares GameMain the enumerations *GameState* and *Direction* which give the game developer the possibility to organize the game flow in a state machine and to easily apply the four basic directions of a 2D environment. A special enumeration case is the *static enum SoundEffect* which has a nested structure, including its own sub-variables, sub-functions and a constructor. The only part that requires customization by the game programmer is the enumeration part in the beginning of it, defining the names and file paths of the individual game sounds. For the game *Sokoban*, this looks like follows:

```
// Enumeration for sound effects
 static enum SoundEffect {
    // customizable list of sounds for each game
    PUSH("d:/workspace/GameProject/freeze.wav"),
    STEP("d:/workspace/GameProject/koopa-stomp.wav"),
    SUCCESS("d:/workspace/GameProject/itemget.wav"),
    FINISH("d:/workspace/GameProject/itemreel.wav"),
    WALL("d:/workspace/GameProject/hit.wav");
```

The GameMain constructor then takes care, that

1. the initialization function *gameInit()* is called
2. the *GameCanvas* gets constructed in the size defined during initialization and placed in the center
3. the game itself is started by the trigger function *gameStart()*

It turned out that the game is running more stable if run in a thread, started by *gameStart()*. The while-loop for the game execution is then outsourced in the function *gameLoop()*, run by the thread. The GameMain constructor and the *gameStart* function for the thread/loop control are shown below:

```
public GameMain() {
   // Initialize the game objects
   gameInit();
```

```
   // UI components
   canvas = new GameCanvas();
   canvas.setPreferredSize(new Dimension(w, h));
   add(canvas, BorderLayout.CENTER);   // center of default BorderLayout

   // Start the game.
   gameStart();
}
```

Source Code: Constructor of GameMain

```
// To start and re-start the game.
public void gameStart() {
   // Create a new thread
   Thread gameThread =  new Thread() {
      // Override run() to provide the running behavior of this thread.
      @Override
      public void run() {
         gameLoop();
      }
   };
   // Start the thread. start() calls run(), which in turn calls gameLoop().
   gameThread.start();
}
```

Source Code: Thread control of *gameStart*

2.1.2 GameCanvas

The GameCanvas is a sub-class of the *GameMain* class and used to fill the central space with the game graphics after the initialization phase. The implementation of the GameCanvas class itself is fully independent from any game implementation by the programmer, but calls extern functions that can be customized by the game developer to personalize the look and feel of the game.

Game Visualization

The *repaintComponent()* function of GameCanvas manages the visual appearence of the canvas. To enable a game-dependent visualization, it calls an extern *gameDraw()* function that provides a switch-case loop which has to be filled by the game developer to draw the intended visualizations dependant of the possible game states. The size of the canvas is controlled in *GameMain* after the constructor of GameCanvas is called and is determined by the *gameInit()* function (see previous section *GameMain*) which ensures that the canvas size fits to the number of elements of the game. The GameCanvas does not have to display just the game all the time. The decision to fill it with menu screens, etc. can be made by the game developer later by using the state machine of *gameDraw*. However, the top-level visualization function of GameCanvas is called itself by the *repaint()* function in every iteration of *gameLoop()* in *GameMain*.

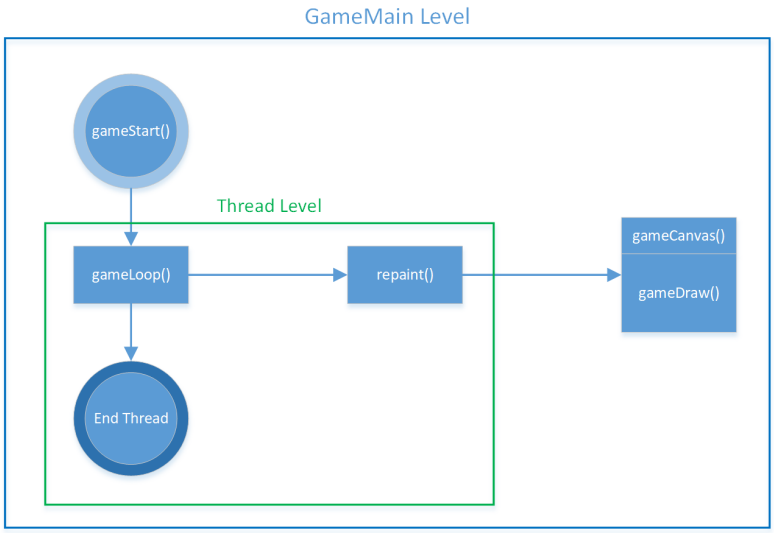

Figure 2.2: Visualization Structure

Game Inputs

For the player inputs later, the GameCanvas adds a *KeyListener* that forwards any received keyboard input to an extern *gameKeyPressed()* functions, which like *gameDraw* has to be filled by the game developer. The game developer can then decide, which keys shall be supported by the game and how the game should react if any of them is pressed. To ensure that key inputs are detected by the *KeyListener*, the GameCanvas constructor ensures that the application sets the focus on the canvas. As key inputs to control the game mechanics are usually only necessary during the *PLAYING* state and might be desired to be usable differently otherwise, it is recommendable to separate also between states in *gameKeyPressed*. However, the implemented game framework leaves this choice to the game programmer. As an example, the implemented version of *Sokoban* leaves a backdoor to the attentive player to "cheat": during the pause screen, the worker in the game can still be moved "blindly" and even push boxes on the target points. Anyways, as there are currently no advantages related to speed, this gives no real advantage yet.

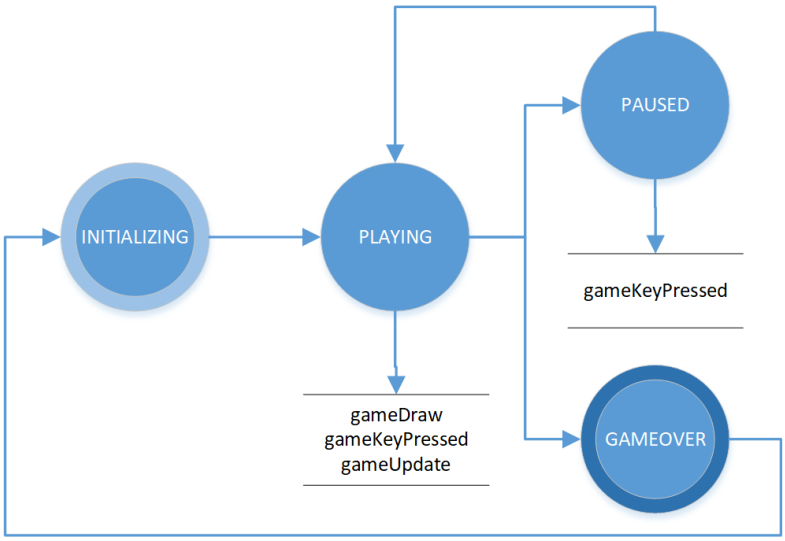

Figure 2.3: Overview of Game State Executions

Besides displaying only the pause screen, acoustical feedback and goal tracking is also deactivated during *PAUSED* state (see figure 2.3, so that most people won´t notice this rather "hidden" game mechanic anyways, and the levels can also only be finished in *PLAYING* state.

2.1.3 Game Menu

The Game Menu is like the sound effects in the first *GameMain* section an extra feature that is provided by the game framework to make the game appearance more attractive, but also more convenient. It consists of two sections, one to generally influence the game flow and one to receive customized information about the specific game implementation. The menu itself is declared as *JMenuBar* component variable by *GameMain* and added to the overall frame in the frameworks *main* function. However, all of its content is defined in the specific function *setupMenuBar()* of the *GameMain* class. In more detail, the menu bar is set-up by two (sub) menus "Game" and "Help" which are themselves again set-up each by two menu items ("New" / "Pause" and " Help Contents" / "About").

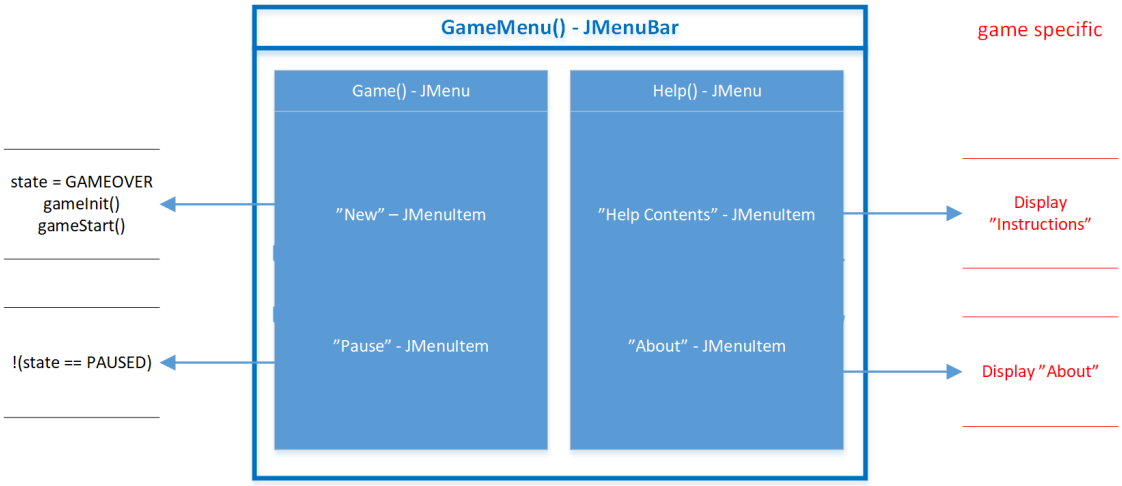

Figure 2.4: Available Game Menus

While the first menu "Game" is ready-to-use without any more programming, the game developer is required to customize the "Help". The latter is intended to give the player some information about the purpose, rules and control of the game which are highly game specific. In contrast, the other one effects the game directly by enabling the player to reset a current level or whole game (dependent of the game implementation) or to pause it. The navigation through the menu(s) is possible via mouse, but every menu item has also an added *KeyListener* so that all of them can also be selected by the corresponding key inputs, ideally to be displayed also by the "Help Contents" section. As they are bound to the menu bar, the key inputs are only working while browsing through the menus.

2.2 Sokoban Add-Ins/-Ons

The second half of the *Design* chapter describes the *Sokoban* specific components of the source code. The section starts with a presentation of the *GameObject* class which serves as a template for the *Sokoban* objects and is therefore used by all of them. Afterwards, the game specific functions for the movement in a *Sokoban* environment are described, finished by the tracking function for the specific *Sokoban* goal requirements to win the current round (and eventually the game).

2.2.1 GameObject

The GameObject class serves as a template for the variety of *Sokoban* elements. It is implemented as class object and not as an interface because its key properties are the same for every game element that is using it. There was also no obvious advantage to implement it as an abstract class, but rather in the classical way as "standard" class that gets extended by the final objects to add a few individual properties where needed. The key properties of the GameObject class are

- **x and y:** *private int* variables that define the object position in the canvas
- **image:** the visualization icon of the element in the format *private Image*
- **SPACE:** a *final int* variable that defines the size of required space for the image in the canvas to fit properly
- **getter and setter methods** for the x, y and image variables to maintain a good coding standard by keeping the variables themselves private
- a **constructor** for the class, setting the initial x- and y-coordinates
- **collision check methods** that are used during the game to check if moving the game element would place it at the location of another game element of interest to determine the correct consequence

The last methods of collision checks are theoretically not necessary for every type of GameObject but prevented repetitive lines of code in some of the individual game elements that use the class. It was a weighing between the amount of functions, number of game elements that use the methods and those that don´t. As the memory overhead is minimal, the choice was made as is. In case of the function *public void move()*, the decision was made the other way around so that it can only be found for the game elements that can actually be moved. This shall also prevent accidentally moving of static game elements where in contrast the existence of needlessness collision check methods cannot cause negative effects as they do not change any properties of elements.

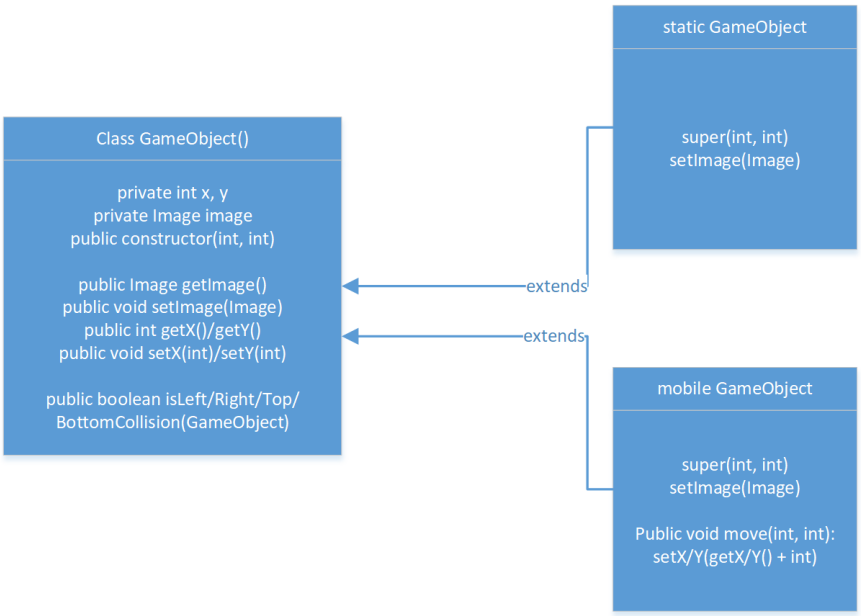

Figure 2.5: Structure and Use of Class *GameObject*

2.2.2 Movement: Collision Checks

One way to summarize the concept of the game *Sokoban* is to "move specifics objects to defined locations while avoiding collisions with specific other objects". In consequence, the game specific key methods are those to check for all sorts of collisions. The following hierarchy of collision checks is performed when a keyboard input by the player to move the worker is perceived:

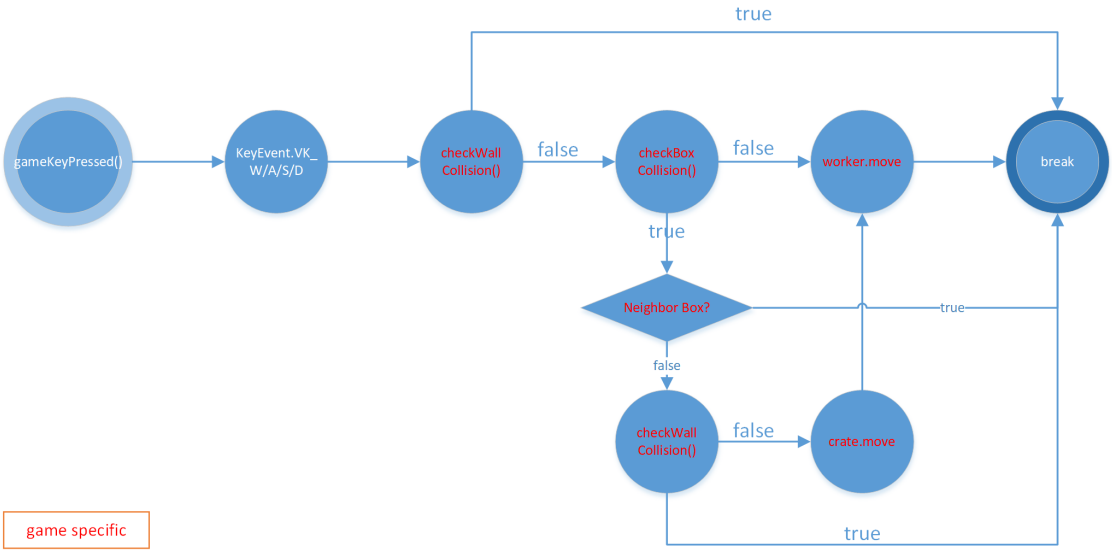

Figure 2.6: Move Request Procedure

As can be seen, the two functions *checkWallCollision* and *checkBoxCollision* make excessive use of the collision methods of the *GameObject* class. The first check for all mobile objects is always, if a movement in the requested direction would let them collide with a wall element. Therefore, *checkWallCollision* applies a for-loop to carry out an individual collision check for every single wall element with the object-to-be-moved. If a potential collision is detected, the move request gets rejected. The second test *checkBoxCollision* applies for the worker when a wall collision could already be excluded. This function imitates the basic concept of *checkWallCollision* but is a little more complex. In the first step, it checks for a potential collision of the worker with one of the boxes. Two rules apply for the move request of a box:

- the worker is no Hulk - only one box can be moved at a time
- walls have the last word - as always: if there´s a wall in the way, you shall not pass!

So, if a collision with a box is detected in the first step, the next collision check is performed for the box itself. Only in case of a free path, the move can be performed: the box will be moved first, then the worker follows. As this happens in micro seconds, the movement appears as one. The described methods fully satisfy the *Sokoban* rules. Anyways, they offer the possibility to change the game mechanics e.g. for specific modes: collect a power-up to become the Hulk after all and be able to move two or more boxes in a row, destroy boxes when smashing them into a wall and receive less points in consequence after clear up the remaining boxes and so on.

2.2.3 Goal Tracking

The goal in *Sokoban* is to push all boxes on their target areas. The function *isCompleted()* checks this condition with every update in the game loop as long as the application is in game state

PLAYING. The function itself is relatively simple: the number of boxes is counted and for every one is then checked if the box is already at a target location or not (see code below). If that´s the case, the goal is fulfilled, the game plays a celebration sound (which varies, according to the level) and sets the whole application in *GAMEOVER* state. Theoretically, the number of boxes could be also counted once during the initialization phase instead of every round. The advantage of counting it every round is to be on the safe side that we only run the check for as many times as the box-array is long. In case that e.g. a possibility to change the number of boxes during the game is implemented, the *isCompleted* function would be prepared to not cause an error. In case that the number of target areas would not be synchronized, a small change would still be necessary.

```
int nOfBoxes = crates.size();
   int finishedBoxes = 0;

   for (int i = 0; i < nOfBoxes; i++) {
      Crate crate = crates.get(i);
      for (int j = 0; j < nOfBoxes; j++) {
         Target target =  targets.get(j);
         if (crate.getX() == target.getX() && crate.getY() == target.getY()) {
            finishedBoxes += 1;
         }
      }
 }
```

Source code: goal tracking algorithm

3. Testing

In order to develop a maximum bug-free game framework and *Sokoban* implementation, the projected needed to be tested continuously during the development process. The testing itself was categorized in several steps that followed each other from general requirements to specific ones and from basic requirements of high importance to minor requirements of individual features. The order of sequence of the tested stages was as follows:

1. **Canvas/Graphics** - Are the visualizations working properly?
2. **Inputs/Movement** - Do the mobile elements react as expected to user inputs and to each other?
3. **Goal Tracking** - Does the goal evaluation work properly?
4. **Game Menu** - Are all added features of the game menu selectable and working?
5. **Advanced Game States** - Does the transition of game states between levels work as intended?
6. **Sounds** - Are all sounds playing correct?

During test phase 1, the visual appearance of the application was tested. Only if the game was displayed correctly would it be possible (and enjoyable) for the user to play the game. In the beginning, the sizes of the game elements were not adjusted correctly and needed to be set up correctly so that there were no more overlappings or invisible objects. With the ongoing development of the framework and game, parts of test category 1 were later repeated between the ongoing tests phases.
Test phase 2 covered the processing of keyboard inputs to control the worker in the game and the correct behaviour of the worker himself and the other game elements in reaction to the moving worker. This included, if stopping rules (walls, two or more boxes in a row) were active in any kind of situation and if all areas that should be accessible could be entered from any direction. This test phase re-activated the first test phase, as it was recognized that the visualization of elements in connection to start positions was faulty (see *Object Visualization* in chapter 4).
The shortest test phase was checking the goal tracking to finish the levels successfully according to the rules. Not only was it tested, if the *isCompleted* function detected it correctly when all boxes were pushed on their target areas but also the robustness of the function by pushing boxes in and out of the target areas. Test phase 3 re-activated again the visualization tests, as there was a bug with boxes disappearing when they reached a target location and re-appearing when moved out again (see again chapter 4).
Test phase 4 covered the additional features that were brought by the game menu(es) and the navigation through the game menu itself. This included the correct detection of game menu related key inputs and the correct behaviour of the game to menu features that affected the game. The game menu was extended during the final development phase, so that some sub-features required additional testing in the end.
When the transition of game states became more complex, test phase 5 was introduced which addressed the correct change of game states between levels or after manipulation through the game menu. Additionally to the correct game state in every situation, the correct change of variable values and the precise call of required functions (e.g. level re-initialization) had to be checked.
The last implemented feature of the game framework were sound effects that were made available for the game play but also all other phases of the application. Therefore, the test phase was linked to almost all other phases (Movements, Menu, Game States).
The numbers represent not only the sequence of the tests, but also the importance of the test category. Only exceptions are number 4 and 5, as the correct game state behaviour in every situation belongs to the fundamental project requirements while the game menu included partly features of minor importance. Regarding the limited time scope, all tests were manually by execution and observation without a form of automation like *javascript* or *json* test scripts.

4. Highlighted Solutions

Even though the game framework itself is not very spectacular, some challenges appeared during the further development and integration of the game *Sokoban* in the framework that required sometimes noteworthy solutions. This chapter shall present the following three:

1. Level Design
2. Prevent unintended Game States
3. Correct Sound Effects and Object Visualizations

They represent challenges during the project work, both regarding some aspects of the *Sokoban* game and the development of the game framework itself. As can be assumed, the chapter begins with the implementation of the level design which led to a solution that makes especially the creation of levels relatively easy and is then followed by the solutions for some problems that appeared during the testing regarding the handling of and switching between the game states. The last part illustrates, how the identified conflicts to play the correct sound effects and to make the right elements in every place of the *GameCanvas* visible were solved.

4.1 Level Design

The primary goal for the level design was that it should be easy and fast to create levels with it. The implemented level design fulfills this goal by being easy to interpret even in the form of source code and therefore also easy to create levels without thinking too much in abstractions. The way it works is that every type of game element that extends the *GameObject* class is represented by another ASCII symbol so that they can be added to a level variable of type *String*. The following code representation was chosen:

- # - a wall element, walls set the boundaries where mobile objects can move
- $ - a box object, boxes can be freely moved around according to the collision check rules in chapter 2
- * - a target location, they are static (currently) and need to be covered by the boxes
- @ - the worker, defining the starting place of the worker to go for the boxes

Two more *String* symbols support the level initialization. To set not only the 'x'- but also the 'y'-positions of the elements correctly, each "\n" (or EOF) symbol will add the value that represents the height of one line to the common 'y'-value. At last, the "floor" should be represented, too, to have a nicer looking play area. The first version of the presented *Sokoban* implementation simply represented all floor areas with a space (" ") character. While the ASCII character kept its representation among the game elements, additional implementations where required for a completely clean level initialization which is described in the last section *Visualization and Sound Effect Conflicts* of this chapter. An example of the introduction/test level variable can be seen below:

```
private String level1
//* Level 1
= "                    \n"
+ "                ### \n"
+ "                  # \n"
+ "     #######      # \n"
+ "     #     #      # \n"
```

```
+ "      #* $ @#       # \n"
+ "      #     #         \n"
+ "      #######         \n"
+ "                      \n"
+ "                      \n"
+ "                      \n";
```

Source Code: Representation of Level 1 as *String* variable

It is important to notice that the current implementation requires every level variable to have a size of 20 character per line (+ EOF) and eleven lines in total for the canvas to be set up correctly, even if the individual required level size is smaller. Every level variable is kept in *String* format until the corresponding level is initialized to be played. Only then will the required *String* variable be decoded, their positions calculated and the resulting elements will be added into one array for each type. These element arrays will then be used for the further processing of the current level until the level is finished or otherwise changed. The concept of the level design was, to the authors knowledge, first introduced by Jan Bodnar, also known as *Zetcode* on his similar named webpage [Bodnar, 2020]. The code in his *GitHub* repository (`https://github.com/janbodnar`) served also for the majority of the further *Sokoban* code elements as base that was then further adjusted and extended with increased functionality according to the framework structure and the authors personal taste.

4.2 Advanced Game State Control

A simple game state concept contains at least the following three states:

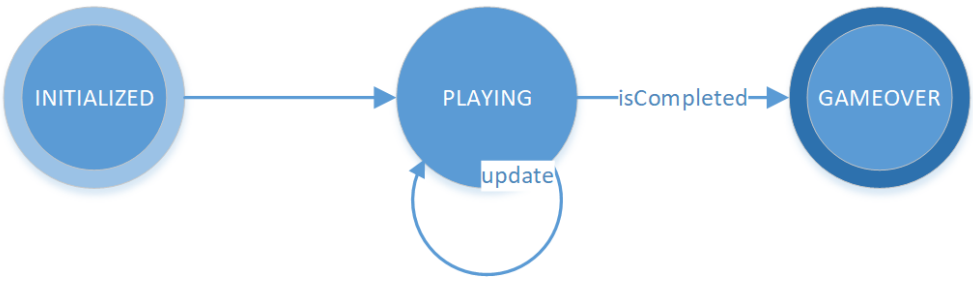

Figure 4.1: Basic State Structure

The game framework provides in total four game states for the support of a controlled game flow: *INITIALIZED, PLAYING, PAUSED* and *GAMEOVER*. The possible transitions between the states are shown in the following diagram:

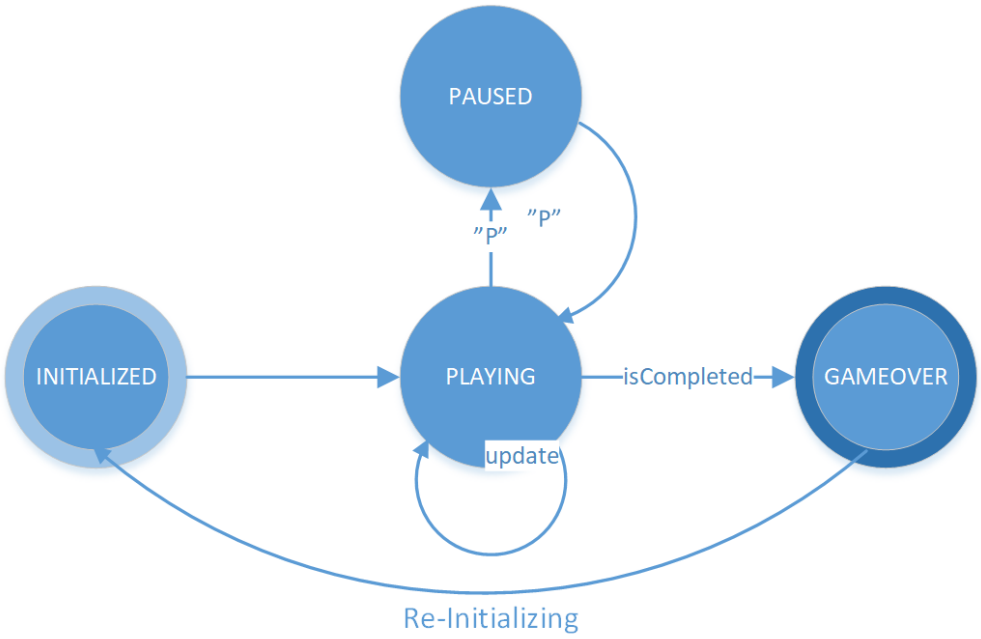

Figure 4.2: Extended State Structure

Two aspects increase the complexity of the game state control of the implemented game framework in contrast to the initially shown "basic" version:

1. the game can be paused which leads to an additional state in which the game is in standby but still running
2. start a new level or restart the current one after finishing a level successfully

There are two methods in the game framework which constantly monitor the game state as their services vary according to the current game state. One is the function *gameDraw* that draws either a dynamic in-game screen while the game is in an active *PLAYING* state or static "info" screens in the states outside of the game. Basically the same is valid for the second function *gameKeyPressed* which evaluates key inputs differently according to the game state. The pause function of the game framework requires a careful handling of this key monitoring. While the game can be interrupted without risk at any time by setting it into *PAUSED* state, it needs to be made sure that it can only be set into *PLAYING* state as long as there is a game world in standby that can be proceeded (e.g. going from a "paused" *GAMEOVER* state into a *PLAYING* state would lead to an error). To make sure that we were in a *PLAYING* state before when we want to un-pause the game, a transition from *PLAYING* to *PAUSED* activates a flag *legalPause*. Only with an active *legalPause* flag, the game can be brought back from *PAUSED* to *PLAYING* state.

```
public void actionPerformed(ActionEvent e) {
     if (state == GameState.PLAYING) {
     state = GameState.PAUSED;
     legalPause = true;
     } else if (legalPause) {
     state = GameState.PLAYING;
     legalPause = false;
     }
```

```
    }
```

Source Code: example of *PAUSED* state control

The possibility to start and restart levels from the *GAMEOVER* state was the second challenge. The early decision to extend the framework functionality so that a game does not have to be necessarily over when a level is finished led to the decision to put a level re-initialization trigger together with the *GAMEOVER* state. Especially since one of the causing events is to finish a level which leads to the *GAMEOVER* state anyways made it the logical choice. While *gameDraw* is a passive method that is just reacting, the *gameKeyPressed* method is playing the central role again. If the player is pressing the right key in the *GAMEOVER* state, the method takes care that the right level according to the play input and previous level gets initialized and started.
No rule without exception: as mentioned in the presentation of the game menu in chapter 2, the menu adds a functionality to restart the current level of the player if he or she got stuck. While the request for a restart sets the game state to *GAMEOVER*, too, it triggers directly the re-initialization and start of the current level. This is because the *GAMEOVER* state is asking the player for his/her wish to proceed which is not necessary here because the intention of the player was already made clear.

```
if (state == GameState.GAMEOVER || state == GameState.INITIALIZED) {
  // different key handling outside of the game
  if (keyCode == KeyEvent.VK_Y) {
     if (level == level1) {
      level = level2;
     } else if (level == level2) {
      level = level3;
     }
     gameInit();
     gameStart();
  }
  if (keyCode == KeyEvent.VK_R) {
   level = level1;
   gameInit();
   gameStart();
  }
      return;
 }
```

Source Code: *GAMEOVER* game flow control

4.3 Visualization and Sound Effect Conflicts

In the simplest of all worlds, every field of the game canvas is occupied by exactly one element with its associated image and every different action causes its unique sound. Anyways, even in the world of Sokoban, things are not (always) that simple. In the following section, we will see how this results in conflicts of low complexity but high effect and how these conflicts were fixed.

4.3.1 Object Visualization

In the section *Level Design* at the beginning of this chapter, we learned that every "tile" of the play-field receives an "identity" during the initialization phase to determine it as a floor, wall,

box or any other possible element. However, this is a very simplified interpretation of the world, even for our *Sokoban* world. What would be obvious in a 3D world, is only detected in a 2D environment when objects move their place: the element below them. This is less relevant for static elements like walls as we don´t care about their underground as long as they keep intact. But even for the target marker, this becomes relevant when a box enters their place - unless a target area does not act as a huge trashcan or a hungry monster that can swallow everything that we throw in there, we don´t want the box to disappear. Additionally, the start positions of the boxes and our (hopefully) diligent worker shall not evolve to black holes as soon as the objects their change place for the first time because they loose their "worker-" or "box-identity". The solution requires two things:

- every box or worker location needs to - as in real life - be also a floor location, which requires the initialization phase to not only add the "regular" floor fields to the list of floor elements, but add the box and worker fields twice: first with their symbol-coded identity, and then as floor elements
- as it is now obvious, that we can have fields in our game canvas which have several "identities", the drawing of the fields has to be implemented in a specific order so that the relevant object in each field gets displayed

For the second point, it is just important to remember which object will be seen by the player.. exactly, the one on top. The following diagram visualizes this hierarchy:

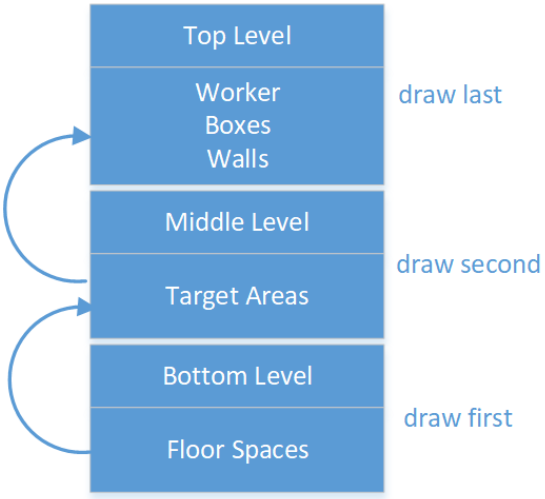

Figure 4.3: Visualization Hierarchy

4.3.2 Sound Effects

A game feels much more exciting and interactive when the actions of the player cause acoustic feedback. Therefore, the game framework provides the possibility to integrate sound effects that can be played at any given time during playing the game and even outside of the *PLAYING* state. *Sokoban* received sound effects for e.g. the boots of the worker on the floor, running into a wall or when a level is cleared. Now what if two things take place simultaneously which theoretically cause both their individual noise. As in code, none of them appears *really* simultaneously as the code is executed line by line so that the sounds could be played after each other as the simplest solution. However, this is not always practical. In *Sokoban*, the most common parallel action is the worker walking over the floor while pushing a box. We can assume, that the foot steps of the

worker would get easily overheard in reality by the loud noise of the box that is dragged along. So in consequence, the worker´s boots on the floor are only heard when walking around alone. This is implemented by the use of the boolean variable *defaultSound*. It is checked at each move of the worker; if it is *true*, his footsteps are heard. Each triggered move of the worker initiates some checks before to see if the intended move is possible. One of them is the check, if there´s a box in front of the worker and if this box can be moved in the same direction. A successful check causes not only a move of the box and plays the corresponding sound but also sets *defaultSound* to *false* so that no sound of the footsteps is heard. Instead of playing it, the variable is set *true* again so that the next "free" move can be heard again.

```
private boolean checkCrateCollision() {
(...)
   defaultSound = false;
   crate.move(-SPACE, 0);
   SoundEffect.PUSH.play();
(...)
}
```

Source Code: sound control in *checkCrateCollision*

```
if (checkCrateCollision()) {
   return;
}
worker.move(-SPACE, 0);
if (defaultSound) SoundEffect.STEP.play();
else defaultSound = true;
```

Source Code: sound control in *gameKeyPressed*

5. Discussion

This chapter reflects the development work of this project, throwing a look at both the implemented game itself and the game framework around/behind it. The objectives achieved shall be investigated first, before the other side of what could have been done differently or better and what could follow shall be mentioned, too. After the evaluation of results, a conclusion sums up the authors impression of the project work.

5.1 Evaluation of Results

From the perspective of the developer, the resulting application code should fulfill primarily two objectives:

- maximum bug-free implementation of basic game (*Sokoban*) functionalities
- flexible, easy to use game framework with numerous usable capabilities

This section is split in two sections: the first part evaluates the achievement of game specific and framework functionalities from the perspective "What is there and how well is it working?". The second part turns the focus on the code structure of the project, to evaluate how well the code is organized and understandable and how flexible and modular is it when it comes to modifications and potential extensions.

5.1.1 Evaluation of Functionality

While the quality of the game framework might have been the most important aspect of the project, the obvious usability test of the framework from a user perspective was how well the implemented *Sokoban* game was playable on the framework and what other functionalities were offered around the basic game mechanism. It can be claimed, that the resulting *framework - game* combination is fully working in the following categories:

Basic functionalities

- **Graphical Appearance:** the level graphics are all displayed correctly, menu/selection screens are "boring simple" but also fully working
- **In-Game Control Mechanics:** the in-game keyboard inputs are correctly processed and the requirements set of allowed game actions is fully tested and working

Advanced functionalities

- **Multi-level support:** the game offers a series of levels with increasing degree of difficulty which can be accessed after successful completion one after another
- **Sound Effects:** actions in the *Sokoban* game trigger sounds to improve the experience and successes are rewarded with special melodies
- **Game Menu:** a game menu completes the game experience offered by the framework. It adds very different features, like pause/restart options and a help menu

From the perspective of functionalities, both the game framework and the specific *Sokoban* functions seem to cooperate well to provide a game implementation that can be played without any more (detected) serious issues. A restart function prevents the player from being stuck in a level due to irreversible actions that deny finishing the level and the game sounds give the game an

improved feeling of interaction. Besides the restart function, the game menu adds rather trivial functions so far that served for the developer to experiment a bit more with the framework mechanics.
One advanced feature that might be also interesting but is currently missing is e.g. a scoring system that evaluates the game-play of the user and compares it in a ranking to increase the motivation. An even more advanced feature could be a set of additional game modes, e.g. randomly appearing power-ups/effects that increase the worker strength or change his speed among other things. Anyways, regarding the limited available time, the range of functions of the current implementation is seen high.

5.1.2 Evaluation of Structure

Among others, the code structure of an application decides about two key factors: how efficiently are resources handled/used (memory, computational speed) and how easy can the code be understood, customized and extended. While computational resources are not playing an overly important role in a software project of this size, the modularity of the code structure does - not only for potential further developments by other programmers but also all the way through the initial development as it decides about the ease or unease of coding in the game framework from the beginning. Furthermore, the course *Advanced Object Oriented Programming* to which this project work belongs had a special focus on applying useful structures and methods that are provided by objected oriented programming in general and *Java* as the programming language of the project. To start the structure evaluation, this part will shortly recap the topic of patterns as they are offered by *Java* and discussed during large parts of the course lectures. Its purpose is to state out which patterns were considered and what led to the decisions of the developer which methods were applied for the final project state of the framework and game implementation. The second part evaluates then the implemented code structures on the base of the previous part, regarding their ability to be modified and to add more and new features.

A Word about Patterns

Patterns are a complex topic that shall not be explained itself at this point. In short, patterns or design patterns to use a more specific term, serve to solve specific problems or tasks by providing a well-proved solution in *Java* [JavaTPoint, 2021]. During the course, we discussed especially *interface* patterns, namely the composite, visitor, strategy and the observer pattern. *Interface* patterns are not instancing an object by themselves but providing a set of one or more functions that needs to be defined by any class that implements this pattern, thereby separating class statements from the operation-set itself. The current game framework as well as the *Sokoban* implementation rather focus on extending template classes and class patterns instead of implementing interface patterns. Therefore, the only applied *interface* pattern of the mentioned four is the *observer* pattern in form of *KeyListeners* and *ActionListeners*. One reason is, that the patterns used contain often a set of functions that are the same for all users which only want to add some individual functions. Instead, the *Sokoban* implementation makes use of e.g. the *Iterator* pattern and the game framework uses e.g. the *Singleton* pattern, not only for classes but also in form of an *enumeration* for the sound effects to encapsulate the list of sounds in direct combination with the desired functions.

Modifications and Extensibility

Initially, the game framework was basically intended as a combination of a state machine, implemented inside both a drawing and a key-input function, both hosted by a game canvas, thereby managing the game flow. All the actual implementations of what to draw in which state and what actions to map to which key was seen as task of the game developer and that´s it. Anyways, with the time, the objectives grew proportionally. The additional game control features of the game menu affect now the game state control as well. The menu itself is basically set-up in a generic

structure that can be relatively easy customized by a game developer to match the individual game requirements. Main attention has to be given probably to the manipulation of the game states as the code sections where they are affected or affect other things are a little bit distributed among several places. The game framework as a whole ended up in a sort of "hybrid" version where the game implementation is in parts directly integrated in the universal framework. A good example is the initialization function where the frontiers between framework and game specific implementation seem to blur. However, it is also a question of definition. In the beginning, only the call of the init-functions for the other parts of the framework (menu bar, sound effects) and the game state change to *INITIALIZED* in the end would have been counted to the framework. Still, the concept of the *Sokoban* game to initialize all game objects by decoding *Strings* is not necessarily game specific but could be applied to all sorts of "tile-based" games. Consequently, with a little more time, I would modify the framework slightly to directly offer this "game world" initialization as a framework service to the game programmer whose only task it would be to define the linked classes of the game specific elements that shall be placed. The *Sokoban* implementation is centered around the two game specific collision detection functions and the *GameObject* template class. All *Sokoban* elements are implemented as children of this function while extending it for the individual parts. Other game elements like the level *Strings* or the sound effects of *Sokoban* are embedded in the framework but can be still easily accessed, changed and more can be added. Overall, I´d say that the game project in its final state is modular to a medium degree. Most changes in the current implementation were easy enough and most interfaces are clear. New functionalities or improvements/replacements of existing ones should be possible at a moderate effort required. However, there is still some room to increase the amount of encapsulation and intensify the use of patterns.

5.2 Conclusion

Looking back at the course lectures after the project work, it can be said that the project itself was a good match to the topic of advanced object oriented programming. Also the choice of *Java* as the course´s programming language was pleasant, as the typical programming languages during my studies in the last years were almost exclusively *C/C++* and *Python*. Therefore, most of the content in this "advanced" programming course was actually indeed new to me so that I could learn a lot of new, more or less *Java* specific, techniques and approaches. While the lecture and assignment part of the course took place during my regular study phase, the project phase fell in a phase of personal on-boarding in a full-time system engineers job, so that the project work fell exclusively on weekends, which shattered the work a little bit. Anyways, this was purely due to the early acceptance of a job offer before the end of the academic year and not at all the fault of the course planners. In the contrast, it was nice to be able to work on such a practical project as the final course work before finishing the final thesis. Even though some more course contents could have been integrated in the project with a bit more time and concentration on that, I´m still highly satisfied with the outcome myself. Overall, I personally see this experience as very beneficial as it increased my knowledge about methods in advanced OOP, especially regarding *Java*, in a significant way.

6. Version Control

During the last two years of my studies in Sweden, I used Git on GitHub mainly for two reasons:

1. Share and update code that I wrote individually as part of contract work for a customer.
2. Working on project code in a team, so that everyone could easily participate and changes of individual team mates could be easily tracked.

The first point showed the advantages of, on the one hand making the *code development transparent* to someone else that had no direct part in the development himself and, on the other hand the combined ability of GitHub as an online platform to make the results *easier available* to the "customer". On the base of my GitHub commits, I could write the working packages and state the invested working hours to get paid for.
The second point basically demonstrates the core advantages of the combination of a version control tool and a sharing platform to have as a development team the possibility to collaborate "risk-free" on common project code, as every change could be reverted, while also having an easier job to identify and interpret the changes of teammates in the code. For long time projects, it helped also to better understand one´s own changes that were made some time ago. Additionally, all that could be further improved by good commenting of individual commits.
Regarding the game framework project, the version control was only of minor importance for me this time, mainly because I was doing it on my own and the project was not that big that I had to fear incomprehensible code changes to cause any more than minor damage. Anyways, it was still a good feeling to always have a backup in case of some unlikely, but critical incident.
Since my start at *SAAB*, I got to know the tool *SVN* (Subversion) to work with version control in an even bigger scale. In this case, projects have grown so big, that it is impractical for an individual developer to work on the software project as a whole. *SVN* administrates the exclusive work on a sub-part of the project by downloading it, modifying it and prompt *SVN* to merge the modified sub-part again into the total project while managing a huge amount of parallel changes of other developers impressively well. In general, I think that version control is always useful and would have prevented many critical development incidents, especially for hobby developers and students, if it would have been properly implemented. Even nothing more than a simple, local *git init* can already be a game changer sometimes.

Bibliography

[Bodnar, 2020] Bodnar, J. (2020). ZetCode Tutorials. https://zetcode.com/. Last access: 2021-07-05. 12

[Christensen and Caspersen, 2002] Christensen, H. B. and Caspersen, M. E. (2002). Frameworks in cs1: A different way of introducing event-driven programming. *SIGCSE Bull.*, 34(3):75–79. 1

[JavaTPoint, 2021] JavaTPoint (2021). Design Patterns in Java. https://www.javatpoint.com/design-patterns-in-java. Last access: 2021-07-04. 18

[NTU, 2021] NTU, C. H.-C. (2021). yet another insignificant programming notes. https://www3.ntu.edu.sg/home/ehchua/programming/index.html#HowTo. Last access: 2021-07-05. 1