Laser Maze - Goblin Edition

Final Report

Group 15

Authors

- Linus Juni, ID: s225224
  Email: s225224@student.dtu.com
- Casper Lauge Nørup Koch, ID: s225169
  Email: s225169@student.dtu.com
- Mathias Markvardsen, ID: s225182
  Email: s225182@student.dtu.com
- Felipe Bahamondes, ID: s221516
  Email: s221516@student.dtu.com
- Morten Henriksen, ID: s225183
  Email: s225183@student.dtu.com
- Oscar Johan Høeg Wohlfahrt, ID: s225117
  Email: s225117@student.dtu.com

Institution: Technical University of Denmark
Course: 02160 Agile Object Oriented Software Development
Screencast: `https://www.youtube.com/watch?v=afP_-Q_go4M&ab_channel=LinusJuni`
Gitlab Repository: `https://gitlab.gbar.dtu.dk/02160-f24/group-15`

1 Introduction & Project Description

We were given the task of developing a small video game in Java. We decided to make the game: Laser Maze - Goblin Edition, a puzzle game designed to test the player's creativity and problem-solving skills. The game's main objective is to place and rotate pieces on a board in order to guide a laser beam to one or more targets. The pieces have unique properties, such as being reflective, blocking or splitting, and are used in different ways so that the player can solve the level. The game draws inspiration from the original Laser Maze game, but introduces additional features and game modes.

We recommend watching the screen cast of a walk through of the game, as well as reading section 7, a user manual, before continuing to read the report, in order to get a full understanding.

Our development process heavily followed the Behavior Driven Development (BDD) approach, where we used User Stories to guide us in our implementation. This will be covered in detail in section 2. In the following we will briefly discuss the games features, that were developed through this process.

1.1 Mandatory features

Game and Rules: The game captures the essence of the original Laser Maze game by allowing players to interact with a board using pieces while adhering to the original rules.
User Interface and Interaction: The application has a GUI that allows players to interact dynamically with the game, for example, through a drag-and-drop functionality for placing pieces on the board. When designing the GUI, we had considerations for UX-concepts and MVC-architecture.
Game Modes: We have implemented the mandatory support for Random Level and Campaign Mode. For Random Level, the player is prompted to choose a difficulty in which they want to receive the level. For Campaign Mode, the player can choose from a set of campaigns to play. In each campaign, the player has 5 lives initially, and each time they miss the target, they are subtracted a life. If they loose all their lives before finishing the campaign, they lose the campaign.

1.2 Optional features

Multiplayer Mode: We decided to make an online feature enabling multiple players to compete with one another. Two or more players can connect to the same session, where they will be presented with a level simultaneously, and compete against one another at finishing the level first. When one player finishes the level everybody else is notified. The server has been dockerized and pushed to Azure cloud, so that you can

play with your friends no matter where you are, without the hazzle of hosting a local server. (please notify *s225169@student.dtu.dk* when the serer can be shut down so unnecessary credit is not wasted).
Persistency Layer: The game allows players to save their progress and resume the game later. This functionality uses a JSON file to store the game data in one of four slots, which can be reloaded by the player at another time.
Sandbox mode: We implemented a mode, where the player is not restricted by cells nor in which direction to shoot the laser. In this mode a piece can be placed *anywhere* and the laser can be shot in any direction. This mode also includes new custom pieces, such as the `Flicker`, which bends the laser, and a `WallPiece` that stops the laser without making you fail the level.
Improved graphical representation: The GUI received a makeover with improved visual effects to create a more engaging experience for the player. It was important to us that the graphics were not overshadowing, such that we maintained an intuitive interaction with the game.

Throughout the project's development we worked with CI/CD, testing and user feedback. We organized the project through a Kanban board. In the following sections we will discuss more in detail how we worked with BDD, SOLID principles and project management.

2 User Stories

Before starting our coding journey we made sure to have a robust set of user stories set in place. Initially we had the following user stories

- As a player I want to be able to move pieces to the board so that I can configure the board to represent the solution.
- As a player I want to be able to shoot the laser so that I can see if I have solved the puzzle
- As a player I want to be able to choose a game mode, so that the game can be played

With these user stories in place we believed we could implement the core features of the game and decided we would later create user stories for extra features after we had started the implementation of the core game. This was, in part, done to get an initial feeling of the project and to establish a foundation on which the rest would be built.

We tried to make the scenarios in each feature as atomic as possible and create them such that they all tested a different part of their corresponding feature. For example we had the following feature describing the responsibilities of our `Board` class.

```
Scenario: Moving a piece from the inventory to a valid board position
  Given an empty board
  And a "TargetAndMirror" piece in the inventory
  When player tries to place the "TargetAndMirror" piece at board position x = 0, y = 0
  Then the board has the "TargetAndMirror" piece at position x = 0, y = 0
```

Figure 1: A sample scenario with a description of the expected behavior of the Board class.

By using a `PieceFactory` in our step code to convert the string input to a `Piece` object we made each line act as an independent component effectively making it reusable, such that we did not have to repeat the implementation of the step in another scenario.

When we were closing in on finishing the user stories constituting the core game, we decided on user stories for our extra features. One of our extra features, making a beautiful UI, could of course not be unit tested, and therefore it has no corresponding user story. We finally decided to go with the following user stories for the extra features:

- As a player I want to be able to play multiplayer mode, so that I can play with my friends.

- As a player I want to be able to save the current level so that I can close the game and reload it later.
- As a player I want to be able to place pieces freely on a board, so that I can construct more dynamic board configurations.

Since the core features had already been implemented, we could focus on implementing and integrating the new features into the existing game.

The selected user stories were chosen based on their potential to engage users and offer interesting challenges from a software development perspective. The save feature required that we had to venture into I/O side of Java while the multiplayer feature forced us to experiment with the STOMP-protocol and the Spring Boot Framework.

3 UML, Design Patterns, MVC & S.O.L.I.D. principles

3.1 Classes organization and architecture

Most of the heavy lifting in the game is done by the `BoardVerifier`. The sole purpose of this class is to verify the configuration of the board and return a meaningful `VerificationResult` that gives one insight into what went wrong if the configuration is invalid and also return a `Laser` which can be used to draw the path of the laser in the UI. The `BoardVerifier`'s only public method is the `VerifyConfiguration` which is called by another class to verify the provided `Board` up against the amount of target pieces which must be hit. The `BoardVerifier` has a list of `LaserShot` objects which traverse the board. All the `LaserShot` objects traverse in a breadth first manner each moving a fixed amount in each step. If a `LaserShot` encounters a `Piece`, it is the `Piece` that decides what will happen in the `hitByLaser()` method. It returns an array of `Orientation` objects which the `BoardVerifier` uses to change the direction of the given `LaserShot`. In each step the `BoardVerifier` also checks if any of the `LaserShot` objects violate any properties such as being out of bounds or having crashed with a piece. When all `LaserShot` objects have stopped moving, which only happens when they hit a target on a target piece, the `VerifyConfiguration` terminates and returns a `VerificationResult`. Figure 2 shows a UML diagram describing the `BoardVerifier` and relevant associated classes and methods are shown.

We also put a lot of thought into the class structure of our pieces. In figure 3 on the next page the UML diagram of the classes related to the pieces is shown. The idea is that each piece originates from an abstract `Piece` class which contains attributes which all pieces share. All pieces places on the board must be able to interact with the laser, and since we have implemented two board types, all pieces must also be able to interact with the laser in both scenarios, which is accomplished by the `hitByLaser` method. Furthermore different classes have different functions. Some are reflective, others are not considered tokens e.g. they do not require being hit by the laser, while other pieces behave like an area. The last two properties does not change how the piece itself behaves, but changes how other objects interact with the pieces. That is why the two properties are embodied by implementing two empty interfaces, so one can do type checking when interacting with the piece. Finally, pieces in sandbox mode consists of different `ShapeTypes` which in turn consists of `SegmentTypes`. For example a `TargetAndMirror` piece is formed like a shape, has a side that reflects, another one that makes a laser crash and yet another one that acts as a target. The `TargetAndMirror` piece is also the only piece with added functionality, since they might require the laser hitting it's target side.

3.2 Design Patterns

Our program uses various design patterns, each serving a specific creational, structural or behavioral purpose. Below are some of the most important design patterns.

- `Builder` : In our `LevelBuilder` and our `BoardBuilder`. We used this pattern since the `Level` and `Board` class are rather complex classes and are tedious to construct programmatically.

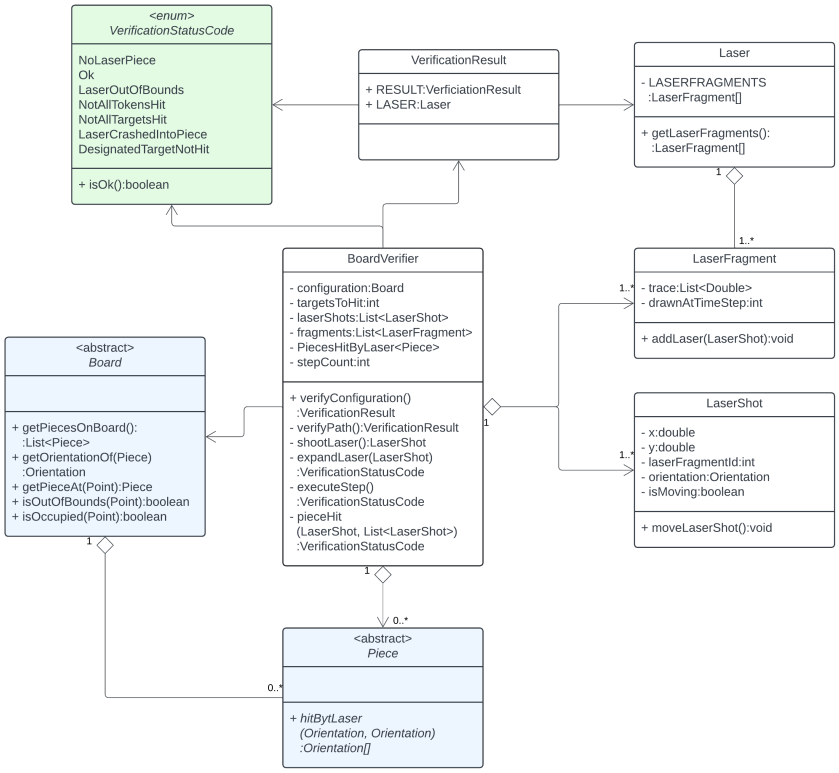

Figure 2: UML diagram describing the `BoardVerifier` and relevant associated classes and methods

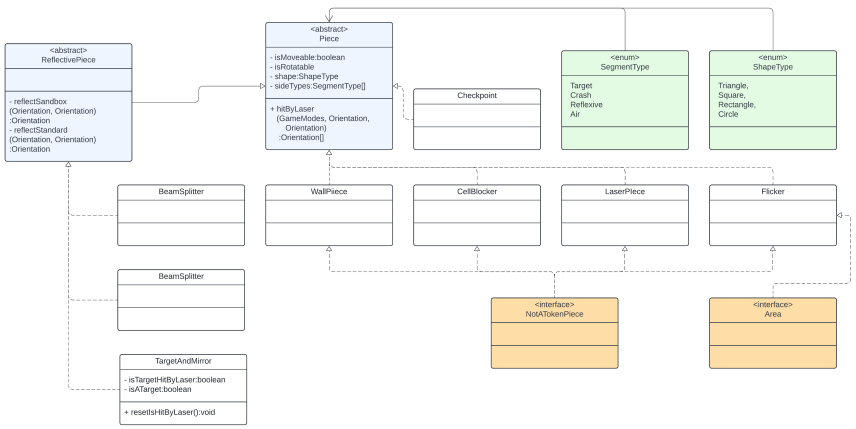

Figure 3: UML diagram showing the piece structure

To simplify this process the two `Builder` classes were made to minimize the need for boilerplate code.

- `Factory` : We have multiple `Factory` classes which are mostly used for cucumber files where complex objects such as `Board` objects and `Piece` objects are passed as strings and a factory method is required to convert these strings objects. We used a similar approach to generate the appropriate image for a specific `Piece` in our front-end.
- `Observer` : The `Observer` pattern is used in our implementation of our back-end server that

enables multiplayer mode. A `Publisher` class is used to notify its `Subscribers` whenever it receives an updated version of the game being played from the server. We chose to use this pattern to decouple the logic handling the connecting with the server and the logic controlling the UI. By using the `Observer` pattern the UI only had to focus on acting on updates from the server and sending simple messages to the server depending based on the user input.

- `Singleton` : We use this pattern in our `OnlineGameController` since the `OnlineGameController` is used to connect to a physical server. It would not make sense if the same client could connect to a server multiple times, which effectively raises the need for a singleton to limit each client to a single connection each. We also use the pattern in the `PlayerState` class, to make the current player globally accessible, making it easy to access it from all over the UI.

3.3 SOLID Principles

Early on in the development we discussed what classes to make and what purpose they should have in order to adhere to SRP. For example we talked about that the `Board` class should be aware of all positions and orientations. We tried to stick to our initial responsibilities as much as possible, however some times we had to bend the responsibility in order to ease the process of some other implementation. For example once we started writing code for the `BoardVerifier` we realised that it would be very convenient if the `LaserShot` objects also know their own position. This motivated us to reimagine the purpose of the `Board` class to only keep track of `Piece` objects. A `LaserShot` traversing a `Board` should keep track of it's own position to decouple it from the logic in the `Board` class.

An example of OCP is when the sandbox board was implemented. We already had a board class, that had a lot of the same functionality that the sandbox-board could have. Instead of creating a new class parallel to the standard board class with almost identical content, we created an abstract super class `Board` that had all the same methods. This allowed us simply to rename the current board class to `StandardBoard` and let it extend the abstract class. On top of this, we could easily let our new `SandboxBoard` extend the same class, as many of the methods were identical. If one would like to add another type of board in the future, he would only have to implement the abstract `Board` class.

The design choice of making the board class abstract to enable other types of boards, also aligns well with LSP too. When implementing the sandbox board, the `Board` class was used by many other classes, that would not be able to use a sandbox board, for example the `BoardVerifer` . By abstracting the functionalities of the `Board` away, and enabling polymorphism we made it easy for other classes to adapt for different type of boards allowing our code to be more flexible.

We adhered to the concept of ISP when designing the piece class. The piece class uses the abstract class `ReflectivePiece` and the two interfaces `AreaPiece` and `NotATokenPiece` to accommodate for the different functionalities of pieces; some can reflect the laser, some are mandatory to be hit, and other occupies an area which affects the laser in some way. This helps our piece class to have less coupling between components and prevents pieces from being burdened with methods they do not need. For example the `ReflectivePiece` allow the pieces that actually reflect the laser to have a reflect method, while excluding non-reflective pieces to have that method.

We structured around DIP in the way we made the `BoardVerifier` . As seen in the UML-diagram, it does not depend on a direct implementation of `Board` . Instead, it depends on an abstraction of `Board` , such that it does not care about the specifics. This approach allows `BoardVerifier` to interact with any class that conforms to `Board` abstraction, without making changes to the `BoardVerifier` . The same approach goes for `Piece` .

4 Testing

Throughout the project, BDD was implemented by initially wording the features that was deemed necessary for core functionalities, after which they were written in a cucumber framework where each feature consist of a set of scenario. The scenarios should be seen as partitions of the entire "feature set". Here a challenge is to prove to yourself that you cover all cases in a feature. Initially this hindered the development speed

severely since one had to get used to the concept of feature files and the cucumber framework, but as the group got used to it the project quickly evolved. Halfway through the project there was tendency of leaning away from BDD and TDD. Later it was discovered that this was a mistake, since unexpected behavior rose from the Model when testing the UI of the game. Since there was little familiarity with JavaFX, the UI required a lot of debugging not realizing that the error was caused by the implementation of the game logic and not the UI framework itself. Many long hours spent finding bugs taught us the hard way, the value of TDD. We went back to our feature files and made sure that our tests covered all the different test scenarios we could think of. Here code coverage became a crucial tool to determine which parts of the code the tests did not cover. The untested code then guided one in creating new scenarios in the feature files. It was here it became evident why the the cucumber framework can be so powerful. It only requires a small amount of step definitions to cover a lot of different scenarios, meaning we could cover more of our code just by adding more scenarios to our cucumber files.

The tests based on the cucumber framework solely consists of unit tests and all testing of UI was carried out manually. Quickly it became clear that testing UI can be a long and cumbersome process, especially if the scene you are testing can only be accessed by navigating the game with buttons and actions to reach the desired scene. But lack of better option and a fear of learning to write tests for UI in the JavaFX framework forced us to do tests manually. Luckily, the rather small size of the game and a rather big team in comparison enabled us to test the UI of our game thoroughly.

While code coverage is a useful tool to help find which lines of code are run, we have found it to not be worthwhile to aim for 100% code coverage. Instead we have found it more rewarding to go for around 90%. This is due to a couple of different reasons:

- **Time Optimization**: While we found that taking the time to write quality code and understand how it interacts with the rest of the code base to be highly important, we also had to prioritize the most crucial work and time.
- **Test Quality over Test Quantity**: We found it to be tedious to aim for 100% code coverage, because it would mean that our test environment would become verbose, which would be counterproductive. So instead of excessive test cases, we focused on writing fewer high quality tests.

An example of the balance between test quality and practicality is important to have in mind in a large agile project, this can be seen in the way we chose not to write tests for every method, like all `toString()` . This of course brings our code coverage down, but it is an active choice resting on the above mentioned reasons.

5 Project Management Detail

To organize our project and keep track of features to implement we used a kanban board. The kanban board contained entire user stories and smaller tasks such as implementing a button or styling a view in the UI, and it made it possible for us to maintain an overview of our overall process.

In our development process we took a lot of advantage of pair programming. It enabled us to write better code and it made it easier for the UI team to integrate the view with the model that the back-end team had created, since they could be guided by the other back-end person who had written the code and knew how best to use it. It also made it easier to spot mistakes and put some pressure off of the driver.

Initially we maintained a strict TDD development where we were never allowed to implement features before tests had been written, but we experienced that this strict approach hindered the speed of our development if we had little to no clue about how a feature should be implemented. Therefore, as the project progressed we relaxed on our strict testing standards and started to experience more with the code and different frameworks such as JavaFX and Spring Boot to get a feeling of how to implement new advanced features. After some time spent experimenting with implementation approaches, we would finally complete our cucumber files and write tests for the new implemented features. This worked really well for us, since it is in practice very hard to test something, if you have no clue how it actually works.

Talking about our commit strategy for the project, we faced several challenges, primarily due to the fact that we had too many branches. In our Git repository, we created many branches for different features.

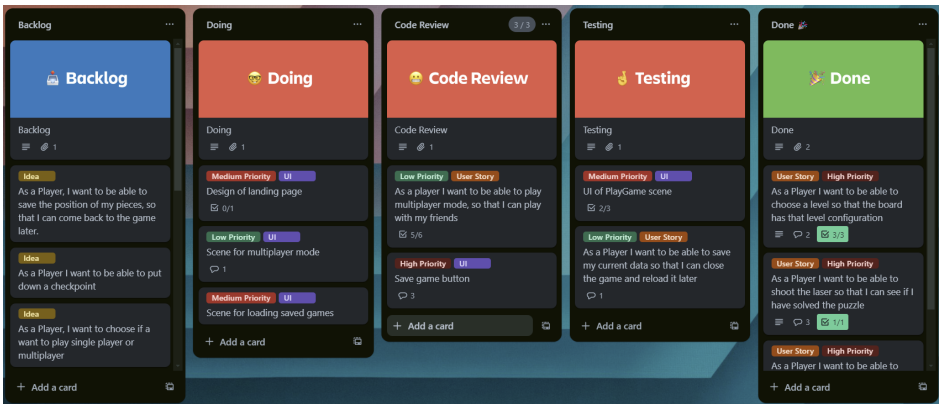

Figure 4: Our kanban board at a point in our project

In theory this approach is great, since it separates the development efforts. However, it quickly became overwhelming, which led to complex merging tasks. This was a significant issue in the integration of Sandbox Mode. Nonetheless, we managed to fix all misalignments, but in hindsight we could have put more effort towards our commit strategy to decrease the complexity. By ensuring more atomic commits and keeping each branch up to date with main daily.

6 Work Distribution

The work distribution was split equally among all team members.

	s225169	s225224	s225182	s221516	s225183	s225117
Workload	1.0	1.0	1.0	1.0	1.0	1.0

Each team member contributed to coding, feature implementation, and debugging. Looking at the 'Contributor Statistics' in the Git repository, there are some discrepancies to this statement. This is due to a couple of different reasons.

- **Pair Programming**: Some team members frequently used the agile development strategy of pair programming, which helped us improve our overall software quality. This had the effect that the person taking the role as the 'navigator' did not get any commits logged.
- **Frequency of Commits**: The frequency of commits among team members varied a lot. Some committed each small change to the codebase, which is a great approach for troubleshooting. Others committed less often, which gave us some more complex merges.
- **Deliberate Committing**: Some team members took a more cautious approach - if they were not entirely happy with the implementation of their solution, they would delay committing it to a branch until they they had discussed with the team and ensured the code to be good.
- **Code Review Practices**: Some team members spent time reviewing the code of other developers, thus explaining the fewer commits.
- **Agile Tasks and User Stories**: Some team members also put in many hours with user stories and other agile tasks.

In summary, the workload was distributed equally among all members, and we had a good spirit on the team, always encouraging each other to deliver high quality software, following the concepts and principles from the course.

7 User Manual

7.0.1 Playing the Game

Our application consists of two parts. An application and a server. The application is simply started with the command `mvn clean javafx:run` . The application works independently from the server with online-mode being the exception.

For your convenience and to enable true multiplayer functionality, the Java Spring Boot server has been dockerized and is running in Azure Cloud. The game is configured to connect to the server running in the cloud so you can play with your friends while using different computers. However, the server may experience problems if it crashes unexpectedly, and therefore require a restart. If you experience problems with the server you can either reach out to *s225169@student.dtu.dk* so the server can be restarted or you can go to the `OnlineGameController` at *src/main/java/Main/Multiplayer/OnlineGameController.java* and change the URL attribute to connect to a server running locally on your PC. If you go for the second option you must start the server locally by running `mvn spring-boot:run` at the source of the project, and you must run two instances of the game on you computer to test the multiplayer features. Please notify *s225169@student.dtu.dk* when you are done testing the program so he can close the container instance and not spend all his credit.

The controls of the game depends on the board type. If you are playing a game on the standard map e.g. the 5x5 map we know from the real world board game, you place pieces by dragging them from the inventory to the board. You rotate the pieces by clicking on them and if you want to return the pieces to your inventory, you must press the *"Return pieces!"*. If you are playing a game on a sandbox map you drag and drop the pieces and scroll your mouse wheel to rotate the pieces. On either board type, you can test your solution by firing the laser by pressing *"Shoot Laser!"*. You can watch the demonstration, link in the beginning of document, for further information.

8 References

8.0.1 Sound

Soundtrack: `https://www.youtube.com/watch?v=hrgzWEgCCFg&ab_channel=FreeMusic`
Laser Shot Sound Effect: `https://www.youtube.com/watch?v=wmlXPgsyxU4&ab_channel=Seaguli`
Button Click Sound Effect: `https://pixabay.com/sound-effects/search/button-click/`

8.0.2 Images

Goblins: `https://gencraft.com/generate`
Pieces: `https://www.thinkfun.com/wp-content/uploads/2013/09/Laser-1014-Instructions.pdf`
Buttons: `https://kenney.nl/assets/ui-pack`