DESIGN PATTERNS IN SPACE INVADERS

SPRINT 5

by

A. Smesseim, S. Baraç, A. el Khalki, A.W.L. Oostmeyer, M. Maton

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,

to be presented on 23 October 2015.

Supervisor: Dr. A. Bacchelli

Delft
University of
Technology

Contents

1

20-Time, Revolutions

Requirements

Functional Requirements

For the game Space Invaders, the requirements regarding functionality and service of the multiplayer functionality are grouped under the Functional Requirements. Within these functional requirements, four categories can be identified using the MoSCoW[1] model for prioritizing requirements.

Must Haves

1. The game must show a "START LOCAL MULTIPLAYER" button on the main menu.
2. When the player presses this button, the 60 enemies and barricades will be displayed on the screen (as if the player had pressed "START GAME" to start a regular singleplayer game). The game must also show two ships.
3. The ships must be able to move independently from each other (i.e. one keyboard key must not be able to move both ships).
4. The ships must be able to shoot independently from each other (i.e. one keyboard key must not be able to order both ships to shoot).
5. The game must show the store after all enemies have died.
6. After the player presses "Continue" in the store, the 60 enemies and barricades will be displayed on the screen (as if the player had pressed "Continue" to continue a regular singleplayer game). The game must also show two ships.

Should Haves

7. The two ships should be visually distinguishable by a non-colorblind player (i.e the colors of the ships should be different).

Could Haves

8. The game could let multiple players control different ships in the same game.
   (a) The game could show a "START SERVER" button on the main menu, that starts an instance of the game with 60 enemies, four barricades (as if the player had pressed "START GAME" to start a regular singleplayer game) and two ships.
   (b) The game could show a "CONNECT TO SERVER" button on the main menu. If the player presses this button, then the player sees the game that the server has started on IP address "127.0.0.1". The player (client) could manipulate the game by using its own keyboard inputs.
   (c) The player could change the IP address of the game the player wants to join by passing the IP address as a command line argument.
   (d) If the player presses on "CONNECT TO SERVER", and there is no server started at the specified port, then the text changes to "CONNECT TO SERVER (failed)".

WOULD/WON'T HAVES

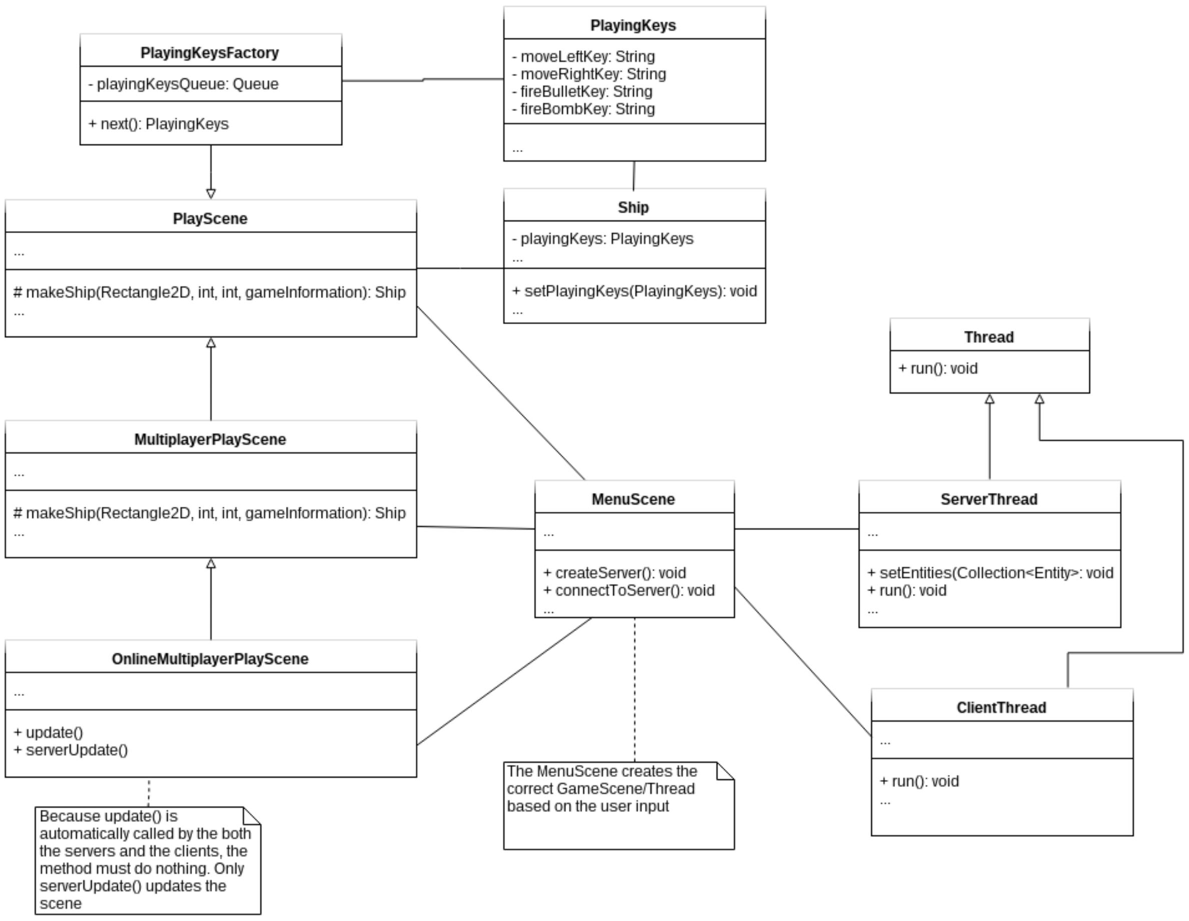
9. The game won't support having three or more ships in the game.

NON-FUNCTIONAL REQUIREMENTS

In addition to the functional requirements specified in chapter 1, the game should also adhere to several non-functional requirements, outlined in this chapter. These requirements will not affect the functionality of the game, but will ensure that the game is gradable by the instructors of the course TI2206. The non-functional requirements of the game are:

10. The game must be playable on Windows (7 or higher), Mac OS X (10.8 and higher) and Linux.
11. The game must be implemented in Java.
12. Scrum methodology must be applied.
13. The source code must be hosted on GitHub as a public repository.
14. The version control used for developing this game must be Git.
15. The tests must be automated using JUnit.
16. This project must use Maven for build automation.
17. This project must use Travis CI for continuous integration.
18. This project must employ the following static analysis tools: Checkstyle, PMD and FindBugs.

UML


PlayingKeysFactory
- playingKeysQueue: Queue
+ next(): PlayingKeys
PlayingKeys
- moveLeftKey: String
- moveRightKey: String
- fireBulletKey: String
- fireBombKey: String
...
Ship
- playingKeys: PlayingKeys
...
+ setPlayingKeys(PlayingKeys): void
...
PlayScene
...
# makeShip(Rectangle2D, int, int, gameInformation): Ship
...
Thread
+ run(): void
MultiplayerPlayScene
...
# makeShip(Rectangle2D, int, int, gameInformation): Ship
...
MenuScene
...
+ createServer(): void
+ connectToServer(): void
...
ServerThread
...
+ setEntities(Collection<Entity>: void
+ run(): void
...
OnlineMultiplayerPlayScene
...
+ update()
+ serverUpdate()
ClientThread
...
+ run(): void
...
The MenuScene creates the correct GameScene/Thread based on the user input
Because update() is automatically called by the both the servers and the clients, the method must do nothing. Only serverUpdate() updates the scene

CRC CARDS

PlayScene	
Subclasses:	MultiplayerPlayScene & OnlineMultiplayerPlayScene
Superclasses:	GameScene
Responsibilities:	Collaborators:
Creating game entities	Entity
Creating the store	StoreScene
Switch scene to the store	Engine
Creating the server	ServerThread
Creating the client	ClientThread
Displaying the lives, bombs and points	GameInformation

MenuScene	
Subclasses:	None
Superclasses:	GameScene
Responsibilities:	Collaborators:
Showing labels and performing right action upon click	LabelEntity
Creating the playing field	PlayScene
Switch scene to the play scene	Engine
Initiate loading	GameInformation

ServerThread	
Subclasses:	None
Superclasses:	Thread
Responsibilities:	Collaborators:
Parsing the current elements in the scene	Entity
Sending a representation of the scene	Sprite

ClientThread	
Subclasses:	None
Superclasses:	Thread
Responsibilities:	Collaborators:
Receiving a representation of the scene	Sprite
Creating entities from that information	Entity

2
Design Patterns

2.1. Natural Language Description

2.1.1. Singleton

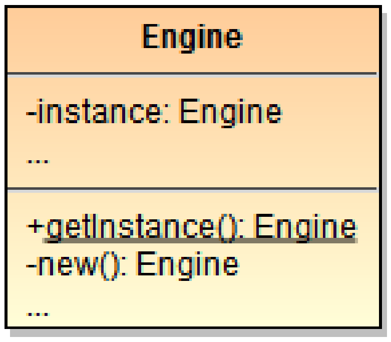
We implemented a singleton pattern for the Engine class. The problem we faced was that we wanted to be able to let Entities change the current scene. We implemented that by creating a private instance variable in Engine that holds a reference to the single instance of Engine. A public method was added that creates an instance if non exists, stores it and returns the instance.

2.1.2. Factory Method

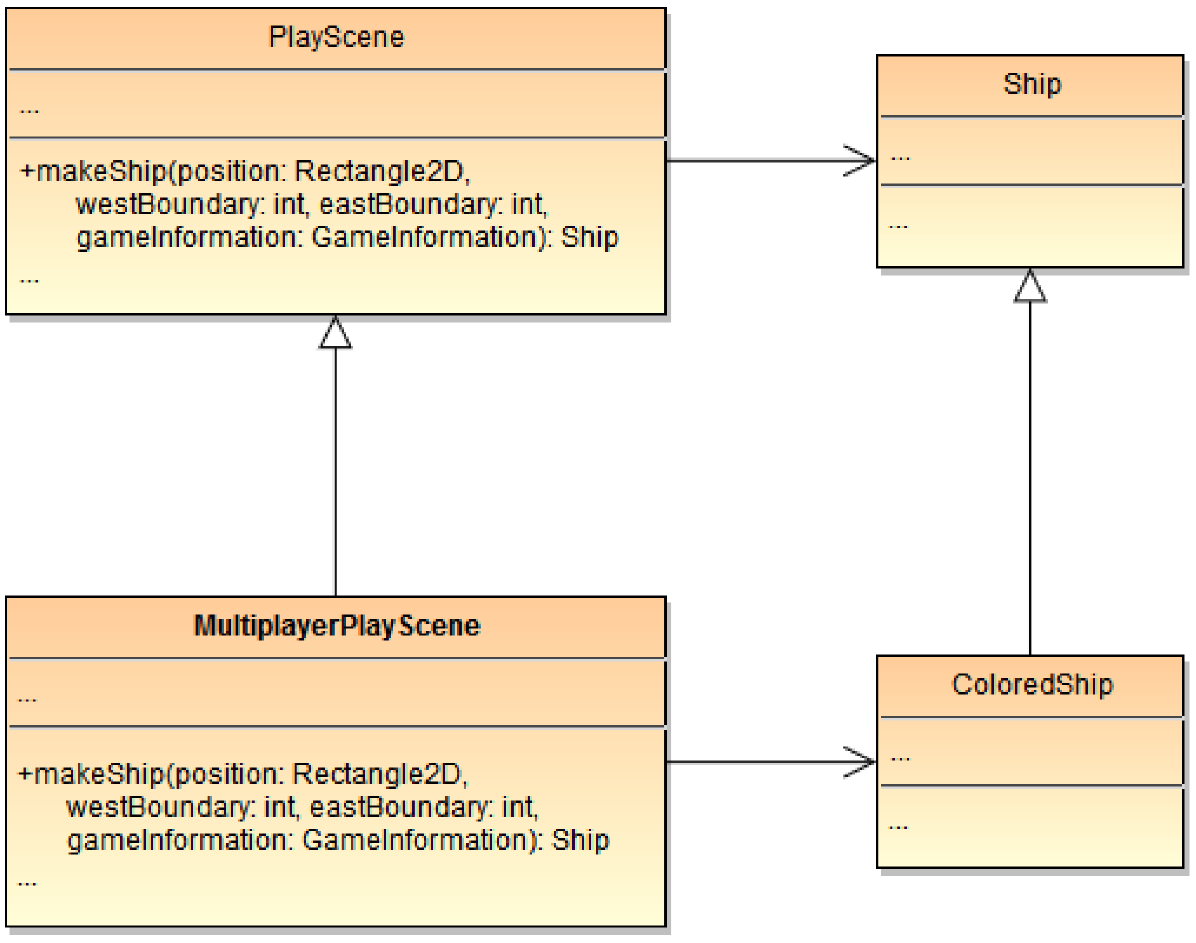
During the implementation of multiplayer we encountered the problem that it was hard to distinguish the ships from each other so we invented colored ships. Because we didn't want switches based on whether the current game is multiplayer or not we added a method that could be overridden in the multiplayer version of the PlayScene. This saves other classes the trouble of determining what kind of ship they should create.

2.2. Class Diagrams

2.2.1. Singleton

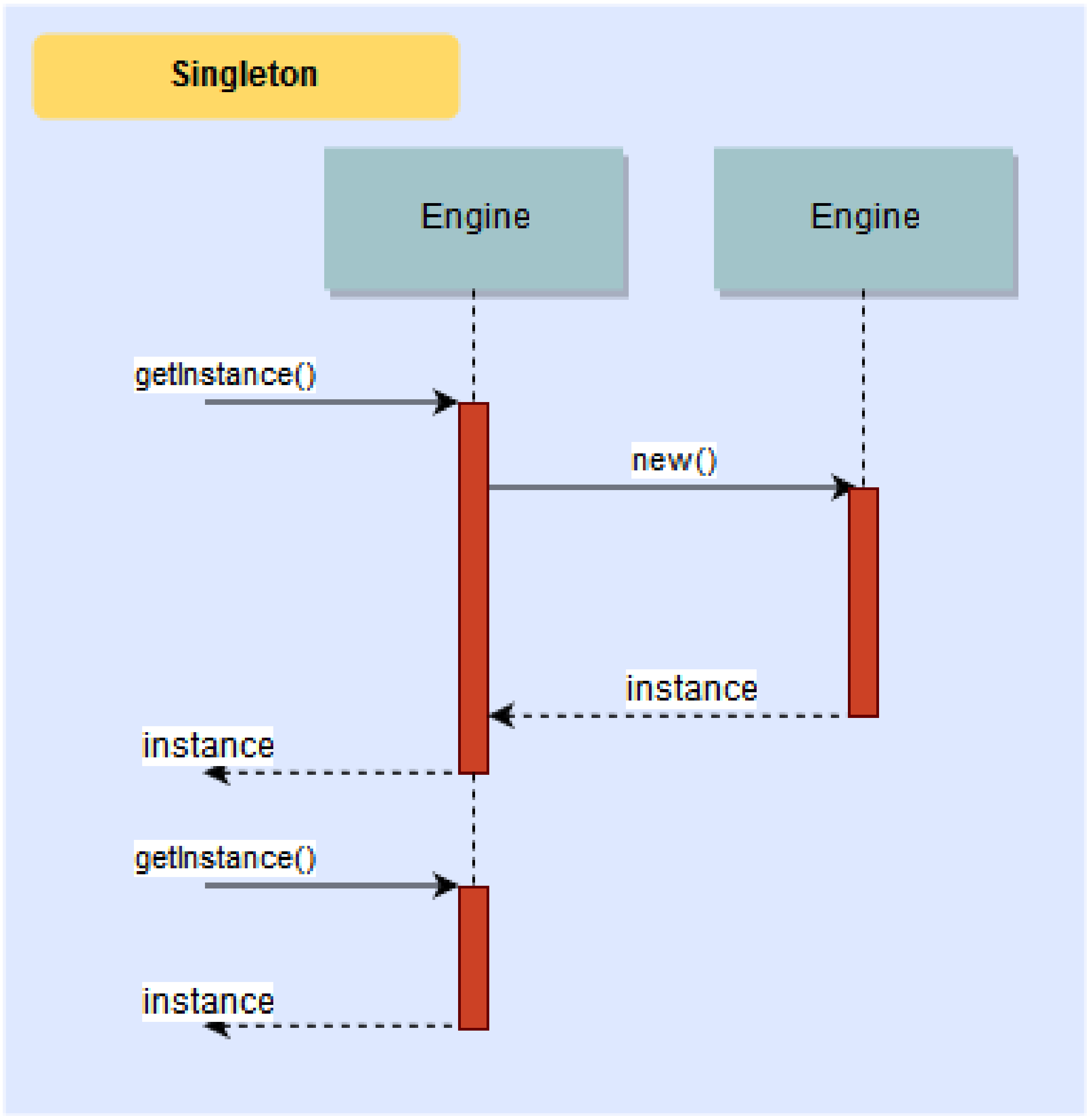


2.2.2. FACTORY METHOD


PlayScene
...
+makeShip(position: Rectangle2D,
westBoundary: int, eastBoundary: int,
gameInformation: GameInformation): Ship
...
Ship
...
...
MultiplayerPlayScene
...
+makeShip(position: Rectangle2D,
westBoundary: int, eastBoundary: int,
gameInformation: GameInformation): Ship
...
ColoredShip
...
...

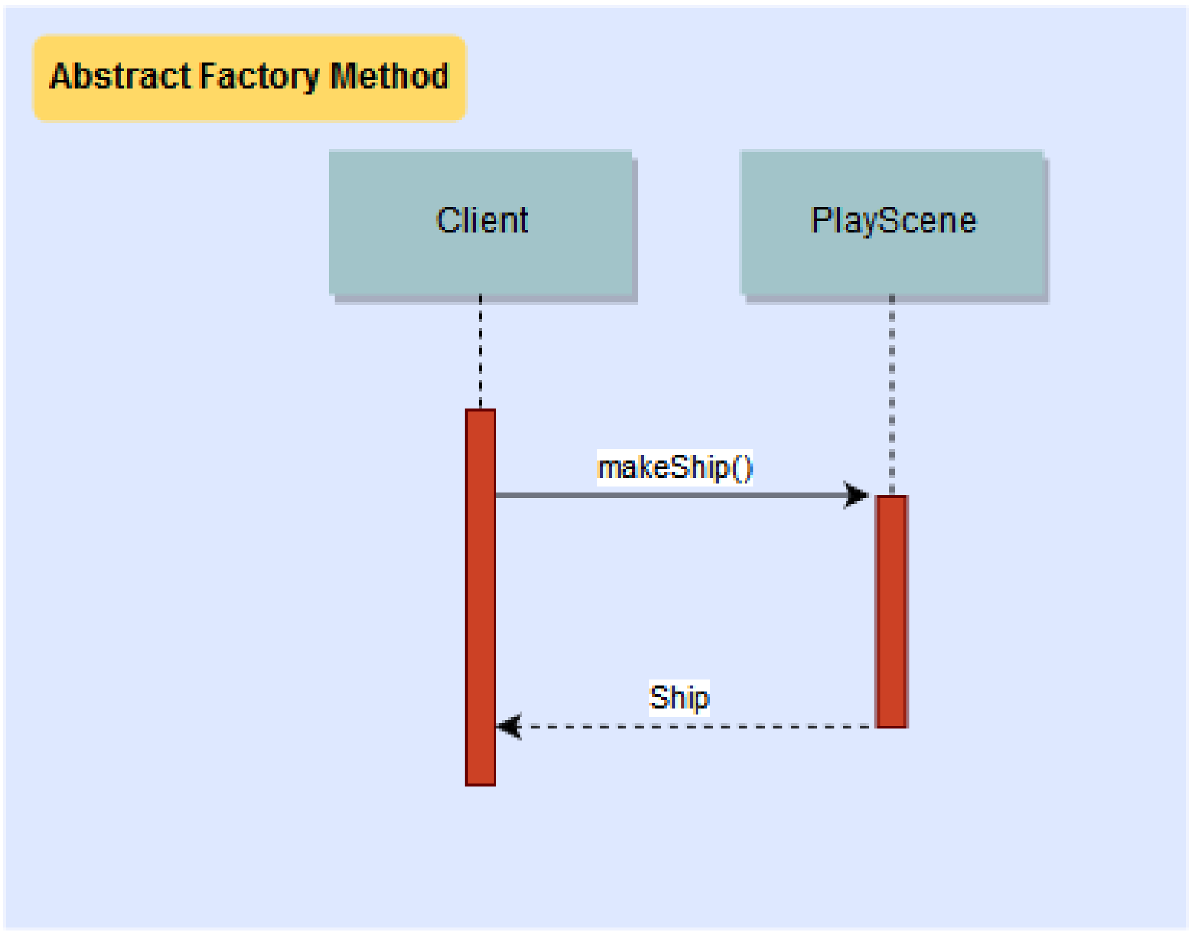
2.3. SEQUENCE DIAGRAMS

2.3.1. SINGLETON



2.3.2. FACTORY METHOD

NOTE: Abstract Factory Method should just be Factory Method


Abstract Factory Method
Client
PlayScene
makeShip()
Ship

3

Wrap up — Reflection

Introduction

To assemble this essay a couple of factors have to be taken into account. First of all an evaluation of both the initial and final version will be done. In these parts we will not go into specific parts of code, but talk rather globally about how the code came to be that way. Afterwards a comparison of the initial version and the, at the time being almost, final version will be given. Furthermore a reflection on the team will show the cooperation between the (soon to be) software engineers on this project. In this part of the essay a reflection on what the team has learned during this project will be made.

Initial Version

As anyone can expect, the initial version of anything will not be perfect. That is why it is called an 'initial' version. However this does not mean that the initial version is a bad release. On the contrary our initial version implemented all must- and should-haves we described in our requirements. A lot of software engineering principles were not implement yet though. Luckily enough this was not the main goal in this version. To apply scrum-based development to our project, we needed a base. This base is made in the initial version. It contained almost everything mentioned in the rubrics of this particular assignment. We are going to evaluate the initial version with the rubrics of it.

- **Requirements**
  - *Specification*
    The requirements cover all the core concepts of the game (both functional and non-functional). The requirements are not contradictory nor ambiguous. Furthermore they are verifiable and are prioritized using the MoSCoW-method.
  - *Implementation*
    As said before the must- and should-haves of our initial version were completely implemented. This results in a very playable game. It is simple and convenient.
- **Code readability**
  - *Formatting*
    The code of the game is well formatted. This results in readable code. Some conventions of java were not implemented, but these divergences were motivated.
  - *Naming*
    Some variables in the code are not properly named. This creates confusions. However all other variables, methods and classes are very clear. It is easy to understand the function of a variable, method or class just by looking to its name.
  - *Comments*
    All of our code is clearly commented using JavaDoc. This is very useful in a project with multiple engineers.

- **Continuous integration**
  - *Building*

    As soon as a commit failed, we fixed the particular commit. However the tests are not as good as we would like them to be at this point.
  - *Testing*

    As mentioned above the tests were not that good. When running Corbura on the initial version we get a total test coverage of 37%. However the engine has test coverage of 51%.
- **Pull-based development model**

  We used an excellent pull-based development model. All of our new features, bug fixes etc. are done in a separate branch and integrated through pull-requests. However pull-requests were not as thoroughly reviewed as we would like them to be reviewed.
- **Tooling**

  All the tools that were described in the requirements are used. However it was a bit difficult getting the POM file ready to use Maven.

(Almost) Final Version

- **Completeness**

  The final version of our game is a very nice and playable game. All of the requirements were met. We even added multiplayer over lan. This is an addition to the requirements made for the working version.
- **Source code quality**

  Our source code is of high quality. This is because we implemented a lot of design patterns. We also made use of the software engineering principles learned in the course. Furthermore every method has a decent length. This improves the readability of the code.
- **Software architecture**

  The system has a clear over-arching software architecture. This means the software architecture is of a good quality.
- **Testing**

  Our tests are not covering a lot of code. It has a test coverage of approximately 70%. This is an estimation, because at the time of writing this essay, we are still implementing tests.
- **Code readability**
  - Formatting

    The code of the game is well formatted. This results in readable code. Some conventions of java were not implemented, but these divergences were motivated.
  - Naming

    Some variables in the code are not properly named. This creates confusions. However all other variables, methods and classes are very clear. It is easy to understand the function of a variable, method or class just by looking to its name.
  - Comments

    All of our code is clearly commented using JavaDoc. This is very useful in a project with multiple engineers.

Comparison: Initial versus Final

As you have read above the evaluation of both the initial and final version look a lot a like. However the progress we made is huge. We implemented a lot of design patterns and applied very useful software principles. In the iterations we can see we have had a lot of progress. Not only as a team, but also as programmers.

Reflection on what we have learned

In the section below we will recall everything we have learned from this course and its lab and thus what we will apply to our future software projects. We can recall and list all the presented software engineering methods, which are designed and used to build maintainable and evolvable software systems. We can use presented software visualization techniques and software metrics to obtain an overview of an existing software system and to detect anomalies in the structure of the code. We can also follow agile methodologies to develop of a new software project. We can create a new software project, by applying the most appropriate softwareengineering practices, given the context of development.

Bibliography

[1] S. Ash, *Moscow prioritisation briefing paper,* DSDM Consortium (2007).