System Design Document for Tank Wars

Version: 0.4

Date: 2018-5-08

Authors: Patricia Zabecka, Thomas Jinton, Adam Kjäll, Carl Lundborg.

This version overrides all previous versions.

1.Introduction

This System Design Document has been created to outline the system design for the new game version of the artillery game named Tank Wars that was originally created by Kenneth Morse in 1990. This version of Tank Wars is intended to keep the original version' s gameplay but with improved visual features.

The purpose of this System Design Document is to provide a detailed description of how the game is constructed. Descriptions of the software, system architecture and security will be presented in order to provide the reader with an overall understanding of the game's system design.

1.1 Design Goals

The project should follow the concept of object-oriented programming and its structure should be categorized in packages that follows the MVC-model. In addition all functionality is separated in independent interchangeable packages in order to make the project as modular as possible. Another goal is to achieve loose coupling and other general object-oriented principles among the packages. Furthermore the project should result in a working game whose functionality resembles the original Tank Wars game.

1.2 Definitions, acronyms, abbreviations

- **Model view controller(MVC),** a way to partition an application with a GUI into distinct parts avoiding a mixture of GUI-code, application code and data spread all over. Also makes it possible to test the model very easy and to be able to use this model for other frameworks as well.
- **Java Runtime Environment(JRE),** a requirement in order to run Java programs.
- **libGDX,** a free and open-source game-development application framework.
- **JUnit,** a unit testing framework for Java programming language.

2. System architecture

2.1 System Overview

The system is only being runned at one computer and there is no communication between other devices. The system design follows the MVC pattern and contains a model, view and a

controller package. The model contains all of the logical decision making for the system, the view contains all output in forms of rendering images and sound and the controller is responsible to handle all input from the use in the form of mouse clicks and keyboard input. Services, utilities and all the test classes have also their own packages.

2.2 Software Dependencies

The application uses the LibGDX framework for graphics and sound effects. All usage of the framework is separated from the classes included in the model package. This way the framework could be easily changed.

The project has followed a test-driven development, all the testing has been done with the JUnit framework.

A single desktop computer is required for running the application, provided that the JRE 8 version is installed. The *TankWars* jar file must also be installed to deploy the application including its classes and associated resources. The application is compatible for both Mac and PC.

2.3 General Principles and Patterns

Single Responsibility Principle

Classes that are working together to achieve a specific goal are kept in the same package. Most of the classes have single responsibilities so that only one functionality is handled by each class.

Open Closed Principle

The project's design is structured so that classes and methods are open for extensions but closed for modifications.

Factory Pattern

The factory pattern is used to create new objects in the model.

2.4 System flow description (How to stop/start system)

The flow of the program is supposed to be fairly easy to comprehend. Nothing extra complicated has been added to confuse the user.

In short starting and stopping can be described with the following steps:

- The system is started when the user runs the desktop application through the required jar file.
- The system takes input from the user when keys are pressed on the keyboard.
- The view starts the application.
- The Controller communicates with the Model through input from the user.
- View has event listeners that handle the displaying of resources.
- Both Controller and View have a reference to the Model.
- View uses the Model to know what to display on the screen.

- The render class in the view takes references to resources from assets that is a part of the used framework. Every view class contains a reference to the renderer.
- The system stops when the user shuts down the desktop application.
- No data is saved when the user exits the application.

3. Subsystem decomposition

The composition of the system consists of the following packages:

- **Core**
    - **assets**, contains resources such as images, fonts, audio etc.
    - **src**
        - **main**, contains the MVC-packages….
            - **ctrl** - the controller classes for the screens.
            - **events**
            - **model** - the OO-model for the game with only logic for the application.
                - **factorys**, the factory classes that contains all object creation in the model.
            - **services,** the Asset class that contains all the essential resources and data for our application.
            - **utils** the Hud and the Bar, necessary classes for the Playscreen. The Hud holds the game information like score and angle. The Bar sets up the drawable bars in the game info.
            - **view** - contains all the GUI related classes for the different screens.
        - **test**, same file structure as the model and contains all the test classes for the model use cases.
- **Desktop**
    - **src**, contains the DesktopLauncher class which launches the application.

The dependencies between the different level-packages included in the main package are shown in *Figure 1* in the Appendix section. The red arrows indicate circular dependencies.

3.1 MVC implementation

The application has three packages which in turn contains the view, ctrl and model. The view package has all of the view classes which are used to show each new screen for the user. In the view there are also classes used for rendering the images, animations and sounds. The view also uses event listeners in the model to decide which sound to play. It also keeps a reference of the model in order to decide what should be rendered and animated on the screen.

When values have been initiated in the view, the view initialize the model which uses these values to create the correct number of objects and also creates the world. The model only contains the logic of the application which decides how the game behaves and how input from the user changes the parameters.

The view also initiates the controller which uses input from the user. Each input from the user is handled by the controller which sets the variables in the model according to what input it receives.

3.2 Design Model Description

The design model is described by the model-package. This package includes all classes that are based on the domain model presented in the RAD. Each object for the model is created in a factory class to ensure that all object creation can be found in one place. The model has a top class called TankWars which uses all the objects from the factory classes and then decides which classes to use and what they are used for. Each class that the class TankWars uses has its own logic in terms what its purpose is. Each class also has a reference to a class under it in terms of the “has a ” relationship. How the classes are dependent of each other is shown in *Figure 3* in the Appendix section.

3.3 Use case flow

One of the actions that the user can do with the tank is to shoot. This is a central use case whose flow is summarized with the following steps:

1. The user presses the space-key on the keyboard.
2. The Controller takes input from the keyboard.
    a. If it’s the user’s turn the method fire() is called with the argument windSpeed. The method is called through the class TankWars that accesses the method from the class TankGun.
    b. The ShotFactory is created.
    c. Then a sound event is published to our event bus.
    d. A new Shot object is initialized by the ShotFactory class. Depending on what state TankGun has, a specific type of Shot object is created.
    e. The new object added to an array list containing the shots.
    f. The class PlayScreen updates TankWars which updates the shot’s position.
    g. Then the PlayScreen tells the render to draw the newly created.
    h. TankWars checks if the Shot object is alive. A Shot object is alive if it is in the air.
    i. Thereafter it looks after collisions with the help of the methods: hasCollidedWithTank() and hasCollidedWithWorld().
        i. As a result of the collision a piece of the terrain is removed.
        ii. If a tank is a part of the terrain that got removed it will take damage which results in a decrease of health points.
        iii. If the Shot object has also collided with a tank, the tank’s health points decreases.
            1. If the tank’s health points are equal to zero the tank is killed and an explosion animation is drawn.

For other use cases and their sequence diagram, see the RAD of iteration 1.

3.4 Quality

The core package contains a test package where all the tests can be found. Each test class is made for a specific class from the model, where methods included in each class are tested.

3.5 Known issues

- Two circular dependency (see *Figure 1*).
- When the user moves the tank and fires at the same time, the tank will continue moving when the user no longer presses the key that invokes movement.
- The game doesn't clearly display whose turn it is. All tanks look the same which could be confusing, especially when there are many tanks on the field. This could be solved by for example assigning a different color to the active tank.
- There are two buttons on the playscreen that overlap each other so that only one of the buttons is visible, however both of them are clickable.
- *Deleting 8 Buffer(s)* - somewhere in the program some resources are not properly disposed.
- The screen classes have lots of duplicated code, for instance the render methods' code is almost identical. This also applies to the creation of the button style and the layout tables.
- There are some methods and some logical aspects of the game that are not tested.

4. Persistent data management

We have used AssetsManager which stores all the data of our application, for example audio, pictures and fonts. For this we have a service package in our application and in the class Assets we have loaded all of our relevant assets so that they can be used for the game. Whenever an event in the game needs an image or animation or sound we use the AssetsManager. In the class were we need the resource we simply get the asset from its path. All of our assets are stored in the assets folder in our application. When we no longer need the assets we use the dispose method for releasing the assets.

5. Access control and security

There are no differences between the users. All users have the same access to all information in this software. Everything in the application can be accessed by anyone who uses it.

6. References

- Lecture slides from the course *TDA367*: http://www.cse.chalmers.se/edu/course/TDA367/#lectures

7. Appendix

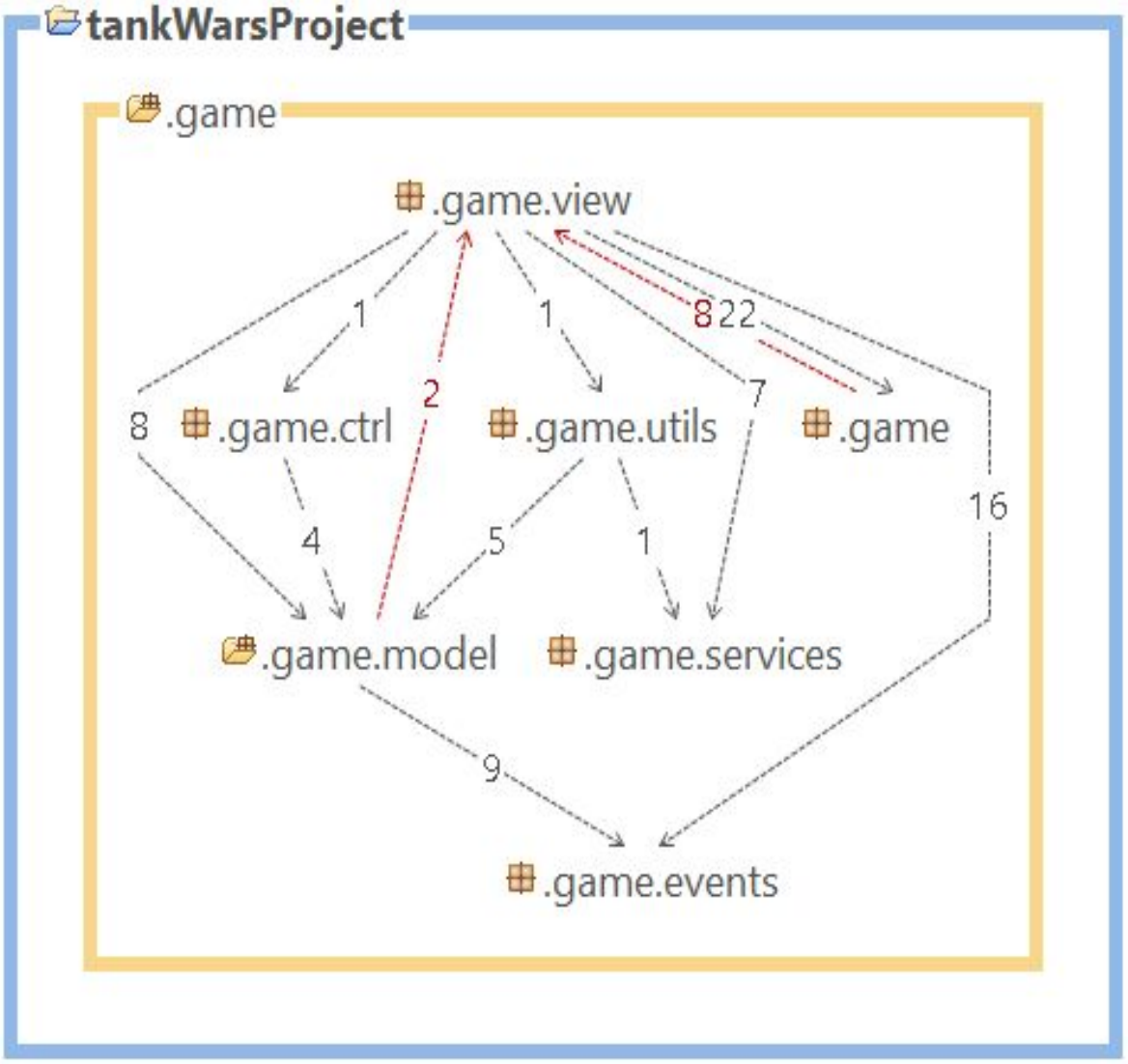

Figure.1. STAN dependency diagram of the main package. The red arrows indicate circular dependencies.

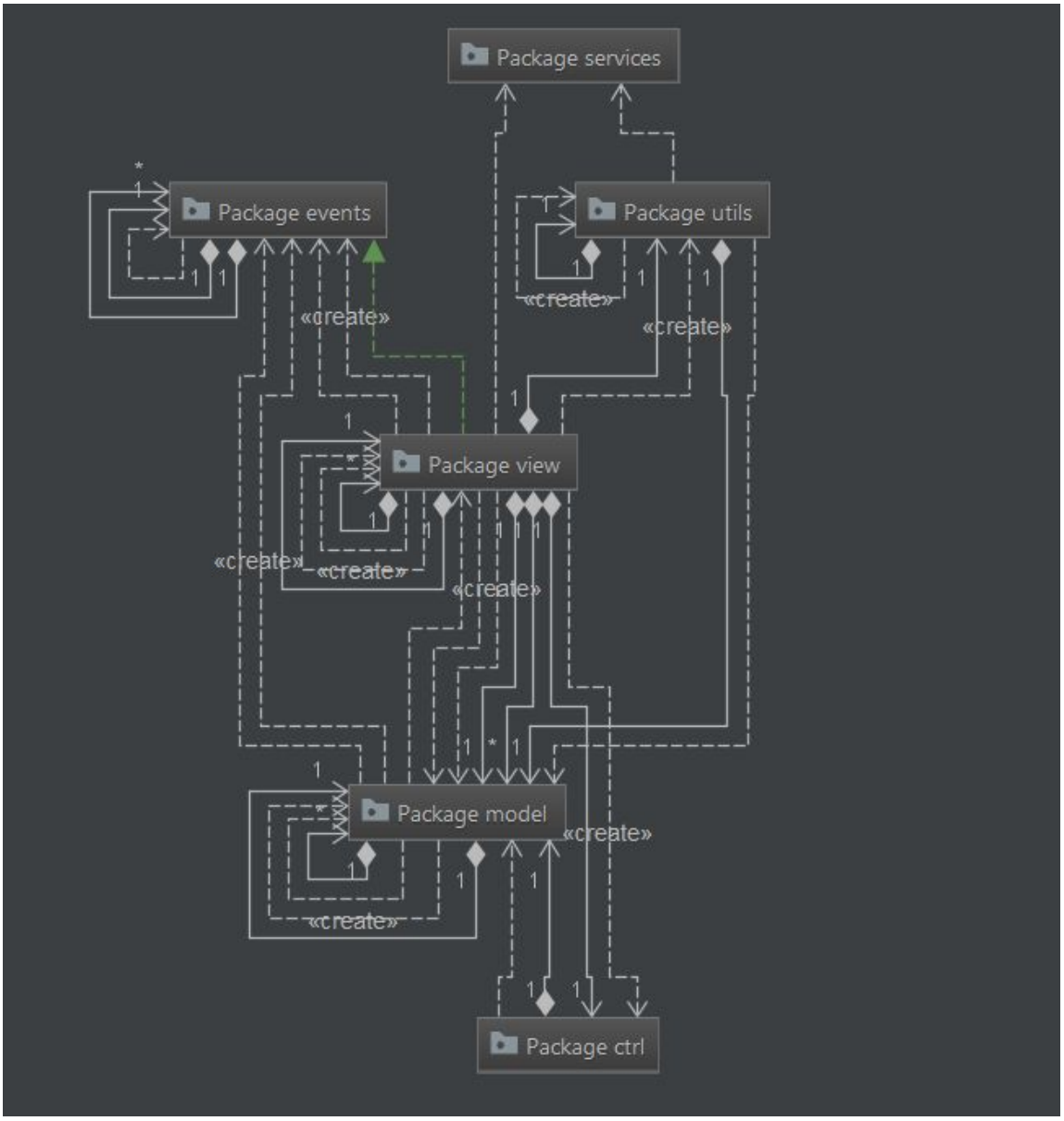

Figure 2. Dependencies among the packages of the main-package

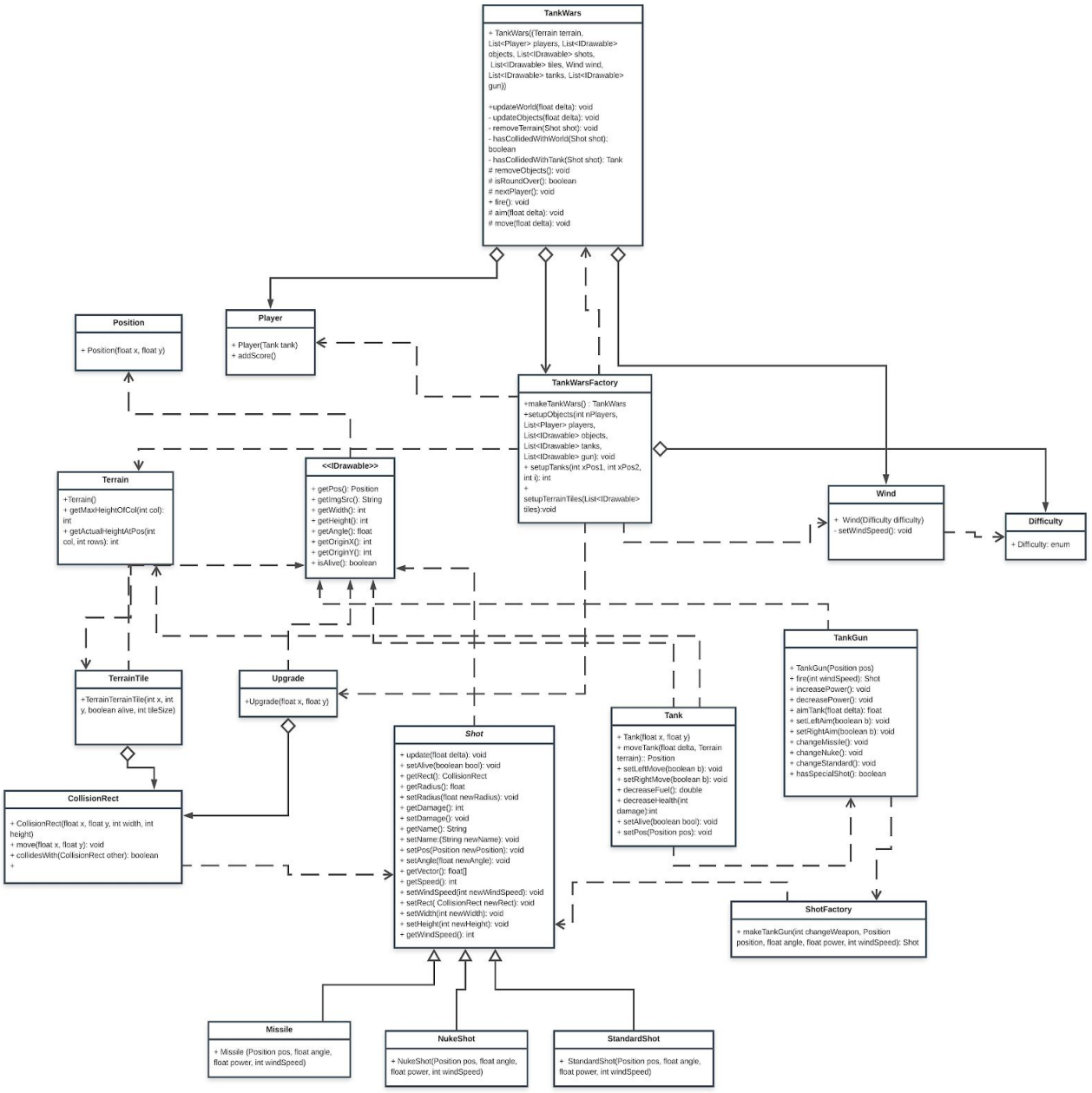

Figure 3. Uml diagram for the classes of the model-package.