System Design Document for Generic Platformer

Oscar Arvidson, Sam Salek, Anwarr Shiervani, Erik Wetter, Malte Åkvist
Group 5: Hmmm

October 23, 2020
version 2.0

Contents

1 Introduction

This report is written with the purpose of presenting the design and system structure for the application named "Generic Platformer".

The aim of this project is to create a 2D platformer wave game. It's a single-player game where the player can use a multitude of weapons to fight off different kinds of enemies. The game progress is built up using multiple levels and wave logic. This means when the current wave of enemies is defeated a new wave will appear.

The purpose of this project is to entertain our users and create stress relief in their everyday life. Generic platformer is supposed to create a fun and relaxing atmosphere, unlike competitive games which can form a very stressful environment at times. To achieve these goals and still build an enjoyable game to play the project used a cartoon-like design rather than a more realistic design.

1.1 Definitions, acronyms and abbreviations

- Java: A programming language.
- FXGL: A Java game library made by Almas Baimagambetov.
- TMX-files: Translation Memory Exchange files, a type of data file.
- UI: User Interface. A way which a user interacts with an application.
- GUI: Graphical User Interface. Allows users to interact with applications through graphical icons.
- MVC: Model-View-Controller, a design pattern used to structure and organize an application, primarily to avoid mixing the applications' code. The code is separated into logic (model), GUI (view), and input (controller).
- Player: The character that the user plays as.
- User: A person using the application.
- Platformer: A platformer is a video game in which the game-play revolves heavily around players controlling a character who runs and jumps onto platforms, floors, ledges, stairs or other objects depicted on a single or scrolling (horizontal or vertical) game screen.
- Enemies: An in game character that follows the player and tries to deal damage to the player.
- Wave: The enemies are placed in the game in groups. Each group has a set amount of enemies in it. An individual group of enemies is referred to as a wave.
- Single-player game: A game where only one player exists and plays the game.
- Thread: A virtual component which divides the computer's physical processor core into multiple virtual cores.
- OOP: Object Oriented Programming.

2 System architecture

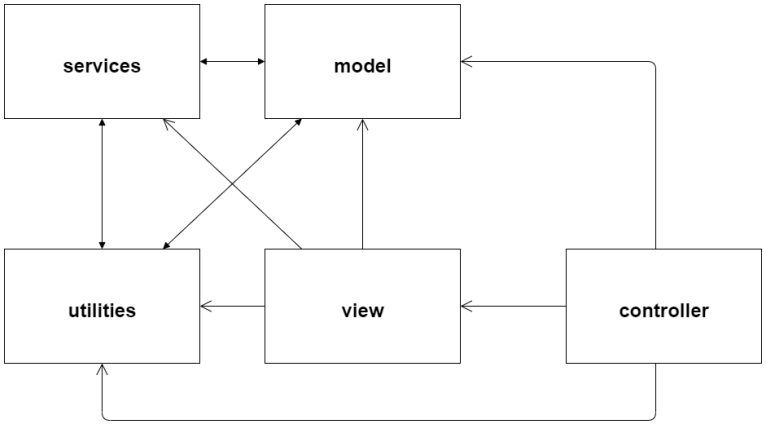

Figure 1: UML of packages (high level).

Generic Platformer is a desktop application with a GUI. The application uses FXGL and JavaFX for the GUI and consists of five different modules, which all have their own area of responsibility. Model is responsible for all the game logic. The view handles the GUI that the user interacts with. Controller package manages communication between user input, GUI and model. Utilities provide helper methods for various calculations (such as getAngle() between two different points). Services gives the model a map array from the tmx-files by reading from the file and then giving it to the model.

2.1 Application flow

The application starts in the Main class with calls to FXGL's library functions initSettings() and initGame(). These functions initialize the game's settings and the game classes respectively. The game classes include GenericPlatformer which acts as the main game application class, and the InputController which initializes the inputs.

This will launch the program and show the main menu for the user. From there the user can press on Play and choose a level to play on. GenericPlatformer will then initialize the classes needed to play the game. These classes include GameWorldFactory, WaveManager, MapManager, BuildManager, and CollisionDetection. The class also creates an instance of the Player which the user controls and plays as in-game.

The user can choose to exit the level and go back to the main menu. This will reset all instances of other classes in GenericPlatformer (Player, WaveManager, etc). All existing entities in the game world are also removed.

Everything is reset and closed down on exit of the application. No values or files are saved.

3 System design

3.1 Package dependency

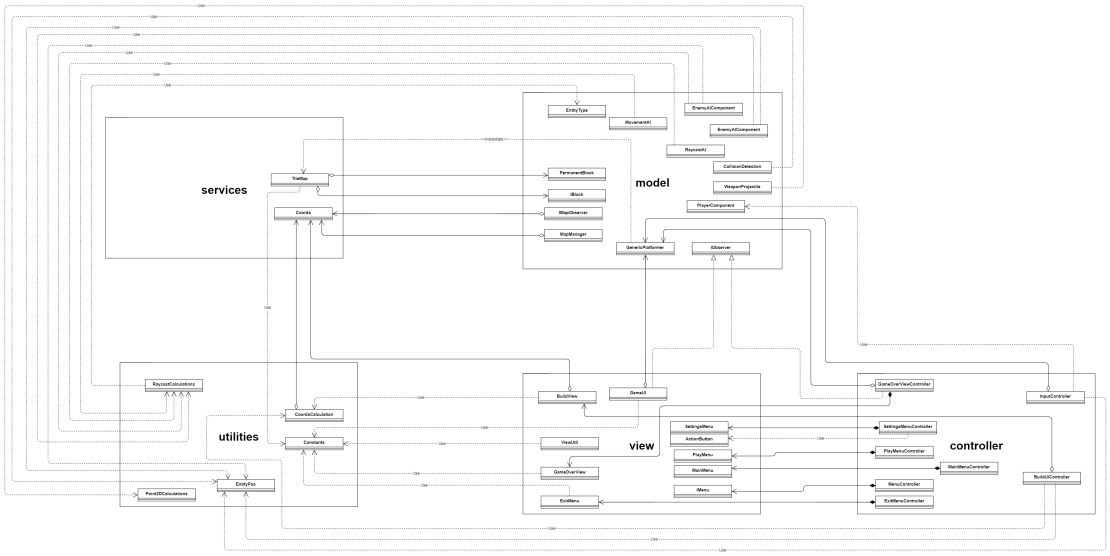

Figure 2: UML of packages dependencies.

Model

Model has a dependency on the utilities and services package in order to use helper methods such as calculations.

Controller

Controller has a dependency on the view, model and utilities. It has a dependency on the view in order to update the UI, for example when the user clicks on a button the controller will tell the view to update. It has a dependency on the model in order to update it when the program receives user input and it is necessary for the model to update. An example of this is if the user clicks 'Mouse1' to shoot the controller tells the model to run the Shoot() method. The dependency on utilities is for calculating and converting between coords and points.

View

View has a dependency on the model, utilities and services. It is dependent on the model to update game related UI, for example when a new wave is generated the UI text for displaying which wave the player is on is updated. The model dependency is done through observer pattern so that the view gets information about any UI related changes in the model. The dependency on both utilities and services is the same as controller, used for calculation related methods.

Services

Services package has a dependency on the model and utilities. It is dependent on the model so that the TileMap class can get the correct blocks from the model in order to create the correct map. It is dependent on utilities for the same reason as view and controller.

Utilities

Has a dependency on services in order to perform Coords calculation (utilities package should have no dependencies since the Coords class should be moved to this package see 5.2 Known Issues).

3.2 MVC design pattern

The application consists of 4 different packages, model, view, controller and application. The application package contains the main class and is used to start the game. Since the application package starts and instantiates the game it has dependencies on both the model, view and controller package.

MVC has been implemented by not letting the model have any dependencies to the other packages (except utilities). By using observer pattern and letting the view observe the model the view is then able to update and draw correct information about the game (such as showing the player hp, ammunition, etc).

Handling user input is done through controllers. For example when the user presses 'A' to walk left the controller then calls the model to run the player method for moveLeft(). The controller also acts as a middleman between user inputs and views by for example updating the view when a button is clicked.

3.3 UML:s

A more in depth look into the applications internal dependencies and structure.

3.3.1 Domain model vs Design model

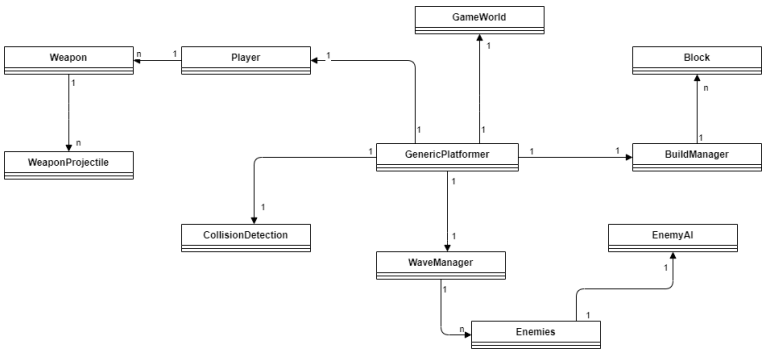

Figure 3: Domain model.

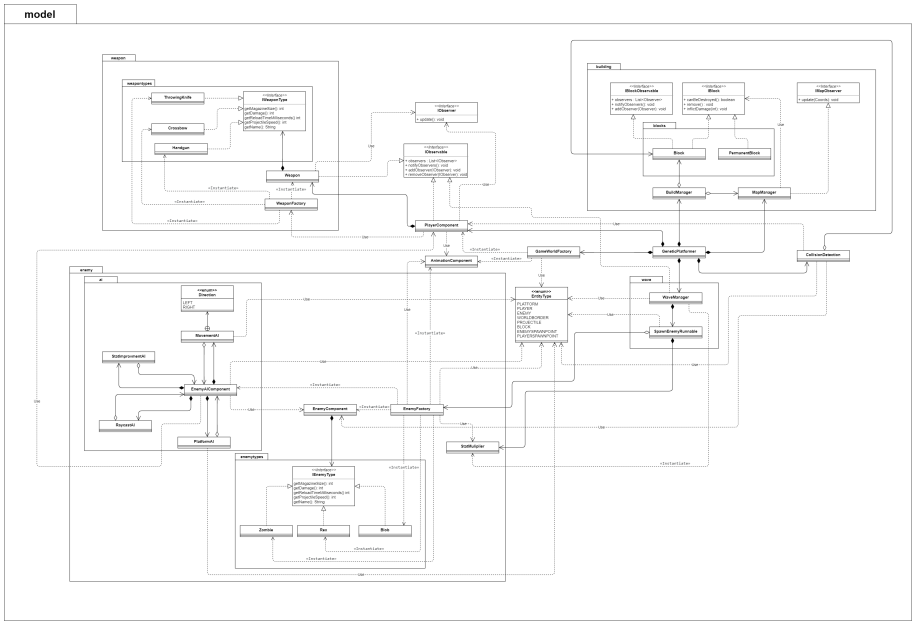

Figure 4: Design model.

Just like in the domain model the design model represents GenericPlatformer as a root class for the model. The different classes branching out from GenericPlatformer in the domain model represent a different package/code structure in the design model, with the exception of Player, GameWorld and CollisionDetection.

WaveManager represents the wave package in the design model where the logic of spawning enemies is located. Enemies include factories for creating enemyComponents and enemy types as well as their entities. EnemyAI manages all logic for how an enemy reacts in game. BuildManager and block illustrates the relation between the build- and mapmanagers creation of blocks in the game. Weapon and WeaponProjectile represent the weapon factory and weapon

type presented in the design model. GameWorld is a GameWorldFactory that creates entities based on information from a tmx-file. CollisionDetection is the exact same and handles all in game collisions between entities. At last Player represents a PlayerComponent that adds player functionality to an entity.

3.3.2 Controller

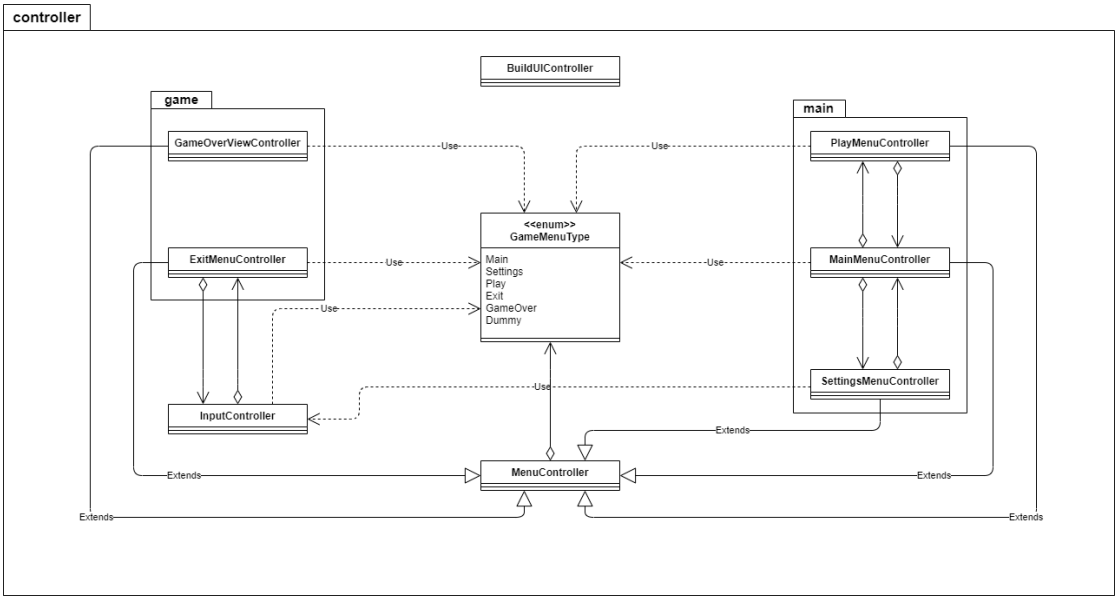

Figure 5: UML of controller package.

3.3.3 View

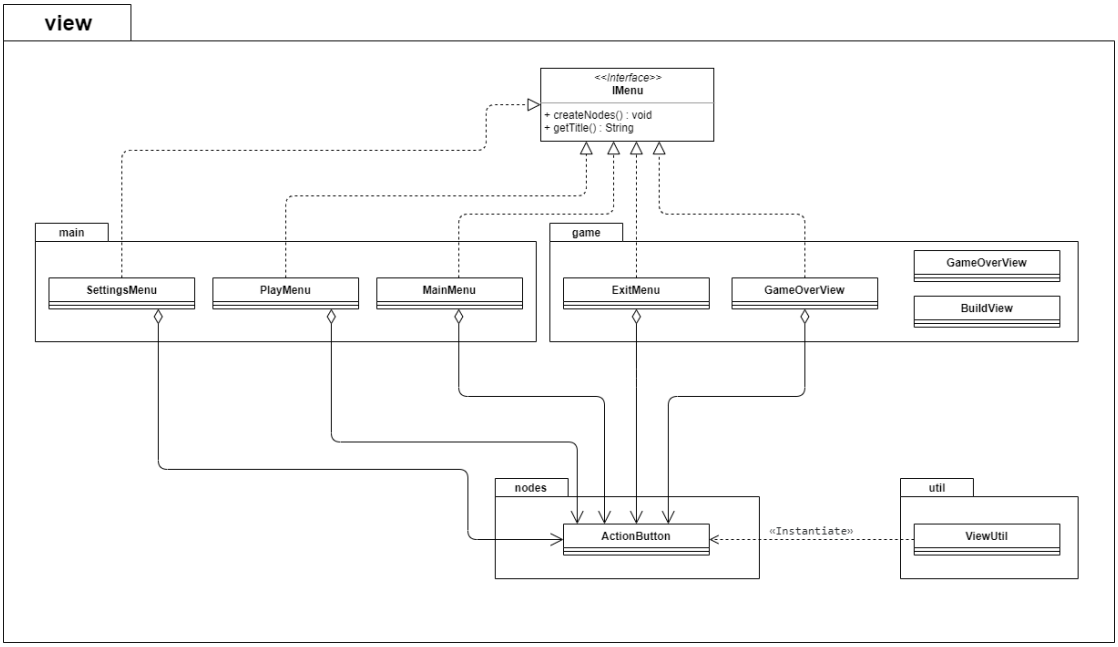

Figure 6: UML of view package.

3.4 Design patterns

Factory pattern:

This pattern is used by the classes EnemyFactory, GameWorldFactory, and WeaponFactory to create enemies, game world entities, and weapons respectively. GameWorldFactory works like an entity factory that is primarily used to create map entities such as platforms and world borders. When a tmx is loaded into the gameWorld it will create all its entity types using GameWorldFactory's methods. The other two factories are meant to create a better code structure for the creation of enemies and weapons with weak dependencies on the rest of the code.

Observer pattern:

This pattern is used between classes Block (IBlockObservable) and MapManager (IMapObserver) by letting the MapManager update the map when a block is destroyed or killed. This way a circular dependency is avoided. Observer pattern is also used between the GameUI and multiple classes in the model in order to update UI nodes such as health bar and ammunition displays etc.

Singleton:

This pattern is used by EnemyFactory and WeaponFactory as they only need to have one instance at a time. The implementation is however not thread safe, but as the model runs on a single thread it shouldn't be an issue.

Facade pattern:

This pattern is used in the GenericPlatformer class. GenericPlatformer works as a root class for the model. It is meant to lower the coupling between packages and improve the MVC structure.

4 Persistent data management

4.1 Storing data

This program does not store any data. The application was originally built with the intent to store data such as sound settings. So far this never came to be, but the possibility for further incorporation of this is layed out.

4.2 Usage of data

Images and tmx-files that are used in the application are stored in an assets directory. The application uses a string-based system from FXGL to fetch the requested file.

Information about levels in the game is stored in tmx-files, one tmx file for each level. These tmx-files consist of images, positions and hitboxes for all blocks in a level. The positions for all blocks are stored as an int array, so by using the service TileMap it is possible to read and get the int array from a tmx-file. This TileMap class takes in a file path for a tmx-file, reads the file and then creates a map over the position of all blocks in a level. The map is then used in the model to check if the player can place blocks on a certain position, it's for example not possible to place a block on a position that already contains another block.

5 Quality

5.1 Testing

The test framework JUnit is used for testing. The aim for tests was that 100% of the model package would be tested and as much as possible for the controller package. Testing for the view package on the other hand was a low priority.

The tests are located under "OOP-Programmeringsprojekt\src\test\java\edu\chalmers".

A problem that was encountered during the first iteration of the tests, for the majority of the classes, was that the Main class had to be instantiated and the application had to be initialized properly. If the proper procedures were not followed, to correctly initialize the application during test execution, then an exception would be thrown every time the tests would try to interact with either the JavaFX or FXGL. Launching the application with the help of the FXGL worked, but this introduces a timing issue. The test helper class has to wait until the application has been properly initialized through JavaFX, otherwise the library would complain about the toolkit not being initialized. The idea here is to avoid deadlocks, such as Thread.sleep(...) encapsulated in indefinite while(true) codeblocks, which would cause the test execution to completely freeze if the application never initializes properly.

It is a much better idea to use a class like CountDownLatch which allows the helper class to set a specific timeout for the initialization of the application. The helper class can then immediately halt the execution of the tests if the initialization did not complete within the requested time frame. The instance of CountDownLatch in the Main class, used in conjunction with the application initialization procedure, was stored in an AtomicReference. The point of an AtomicReference is to allow for different threads on different processor cores to access the same variable and still receive the same (reference-) value. It is said that different threads could perceive a single variable to have different values assigned to itself.

Another problem that had to be dealt with was the fact that everything that interacts with the UI had to be done on the JavaFX UI thread. This was done with the help of Platform.runLater that lets the tests execute code on the UI thread. This method is not synchronous which means that we have to await for the code to fully execute before proceeding with the rest of the test. This was accomplished using the same techniques that were mentioned above, with the CountDownLatch/AtomicReference combination.

5.2 Known issues

Despite all of these test work arounds one class that we did not manage to test were CollisionDetection, because FXGL has collisionHandler methods that we found untestable. Since these methods are from FXGL we decided it would not be necessary to test these methods. However all the methods called from inside the FXGL methods are tested even though CollisionDetection does not have any test coverage.

Another issue that was encountered during the testing of the project had to do with the synchronization for the CountDownLatches not working properly. This meant that the tests for the controllers and the Main class would randomly not function correctly. A lot of analyzing and research was done to study more about the inner mechanisms of CountDownLatch/AtomicReference but no progress was accomplished on that front. The cause for this issue greatly has to do with how

FXGL does not implement asynchronous support for starting/stopping the game, etc.

Instead of using classes like the CountDownLatch, FXGL relies on using delays for waiting until conditions to be met. This introduces heaps of timing issues and timing is of great importance when executing tests, as they tend to execute actions far faster than what a user could. This issue is therefore something that could be blamed on a third-party library, rather than the project itself. Forking Github projects and helping the author fixing issues is a really big and vital part of the open-source community, but there is no way to perform a complete rewrite of the library within the timeframe for this project.

An in game issue we are aware of is a bug which is: if you switch weapons after firing a projectile, the projectile will inflict the damage amount from the selected weapon, instead of the weapon that fired the projectile.

This application has some flaws in its design due to FXGL:s unflexible design. Its intended use does not support the conventional OOP design, it instead uses an entity-component-system. When we realized this the project had already come too far to change and we did our best to keep following the conventional OOP design in our own implementations whilst still using parts of the library. In hindsight this library made our job more difficult rather than simplifying it.

One design flaw we noticed in hindsight is that the Coords class in the services package should be moved into the utilities package. is a simple class used as a complementary to the Point2D to differentiate from tiles and positions in the game. This class is more suited as a utility instead of a service because of its simplicity and since it does no file handling. Placing the class in the utilities package would remove the circular dependency between the utilities and services package.

5.3 Analytical tools

PMD-Report of the code design. Most of the violations presented in the PMD-Report relate to private fields not being final, even though they are only initialized in the declaration or constructor. These violations primarily occurred because we lack the knowledge required to know when to use final variables.

5.4 Access control and security

NA (not applicable).

References

[1] A. Baimagambetov. (2020). Fxgl 11 wiki, [Online]. Available: https://github.com/AlmasB/FXGL/wiki/FXGL-11 (visited on 09/30/2020).

[2] ——, (2020). Fxgl library, [Online]. Available: https://github.com/AlmasB/FXGL (visited on 10/21/2020).

[3] (2020). Tiled map editor, [Online]. Available: https://www.mapeditor.org/ (visited on 10/21/2020).

[4] Z. Alfitra. (2020). Game art 2d, [Online]. Available: https://www.gameart2d.com/ (visited on 10/21/2020).