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

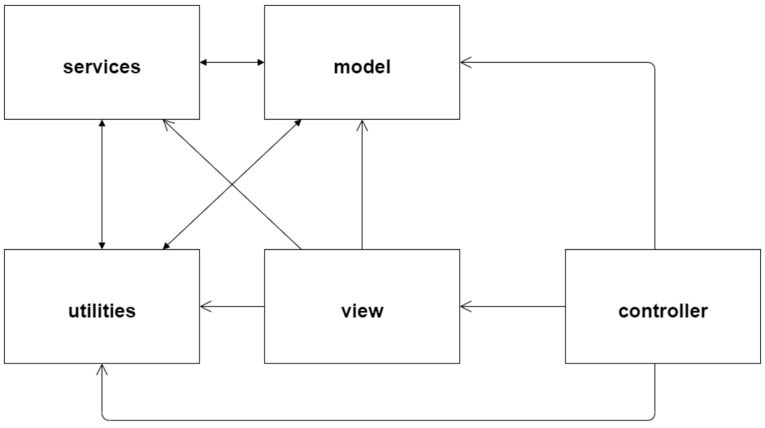

Figure 1: UML of packages (high level).

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

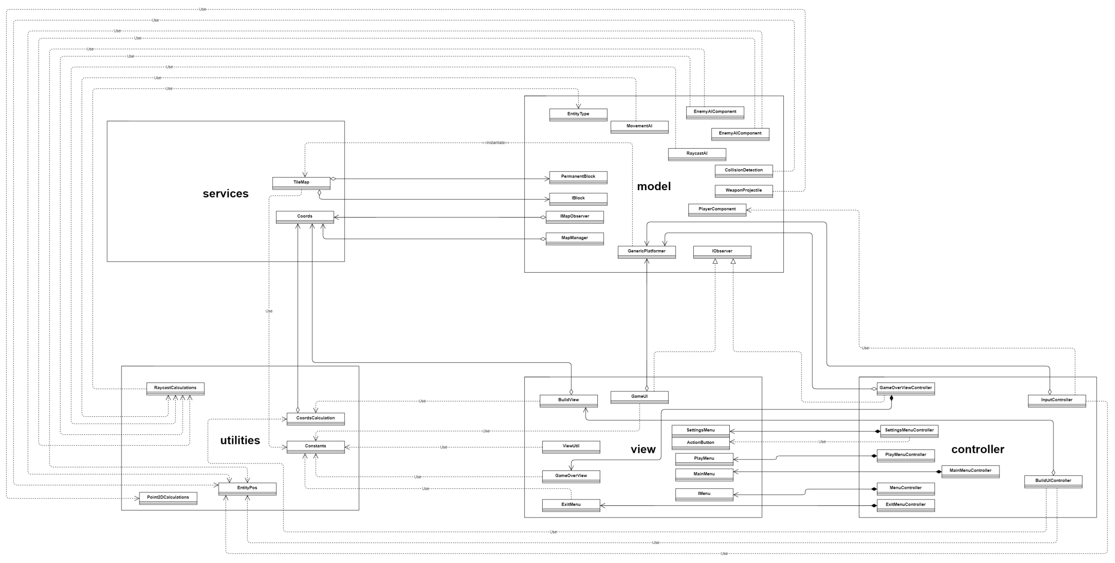

Figure 2: UML of packages dependencies.

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

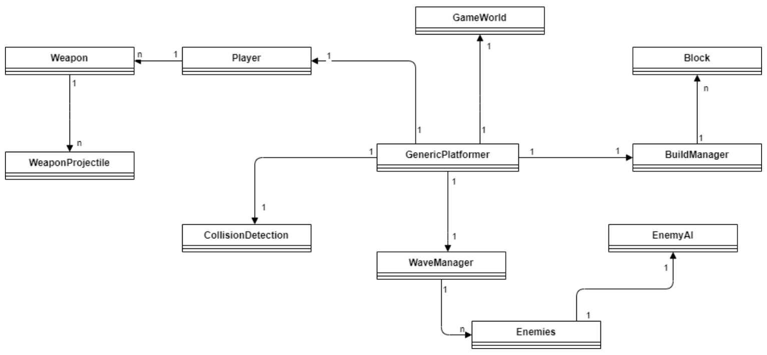

Figure 3: Domain model.

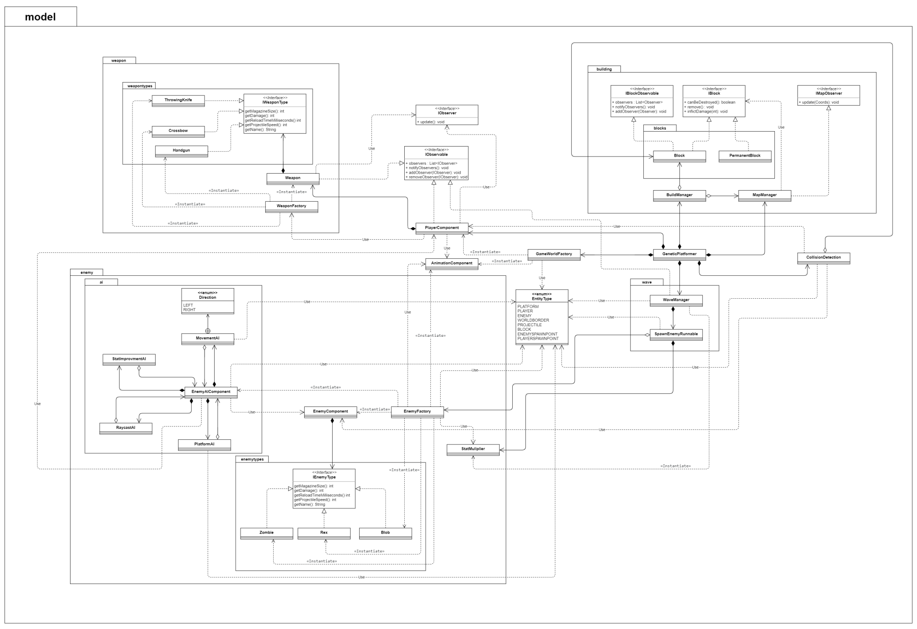

Figure 4: Design model.

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

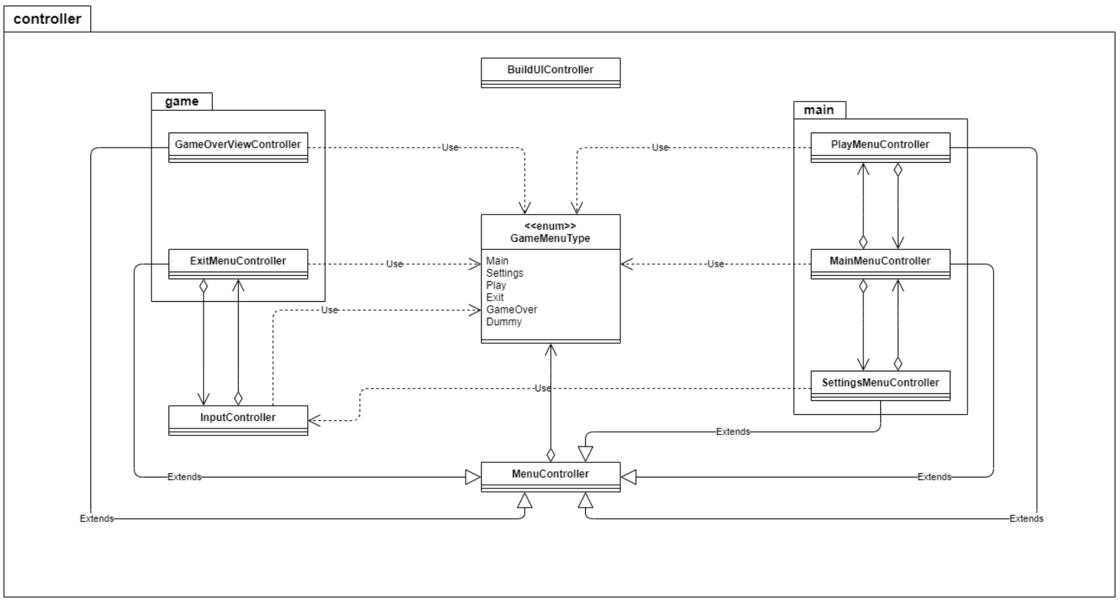

Figure 5: UML of controller package.

3.3.3 View

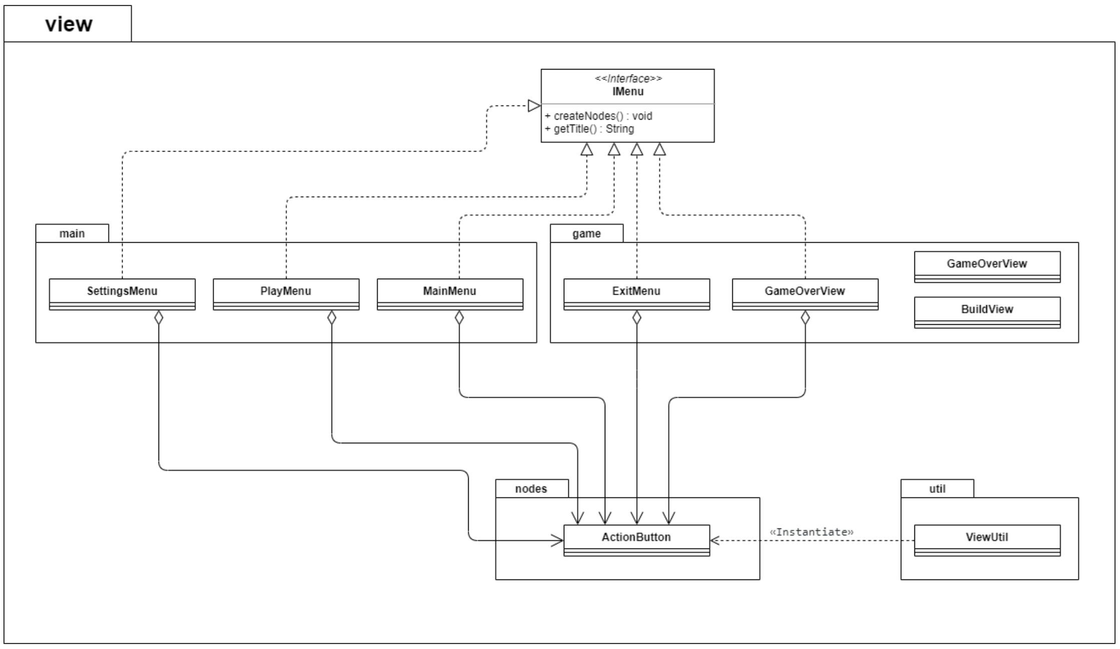

Figure 6: UML of view package.

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 GameWorld-Factory'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).

Requirements and Analysis 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

The aim of this project is to create a 2D platformer 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

Wordlist:

- User story: Description of a software feature from an-end user perspective. A user story describes the type of user, what they want and why. User stories help to create a simplified description of a requirement.
- 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.
- Levels: Different worlds/environments the player can choose between to play on.
- 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.
- Player: The character you play as.
- Block: An object the player can place down on the level.
- Weapon: An object used by the player to kill enemies.
- Entity: An object in the game world with physics properties.
- Hitbox: An invisible box that surrounds entities. When this box touches other entities it detects a collision with the entity.
- User: A person using the application.

2 Requirements

2.1 User Stories

Story Identifier: GPG001

Implemented: Yes

Story Name: Basic World

Description: As a user I need to see a world in the game so that I have something to explore and interact with. A game world would let me see how I can interact when I'm in the game for example seeing where I can go.

Confirmation:
Functional:

- The player needs to be able to stand on some kind of ground.
- The game world should have 2-3 platforms.
- Platforms should have a collidable hitbox.

Non-functional:

- The game world's terrain should be built up using a grid.
- The game world will have three different texture tiles.

Story Identifier: GPG002

Implemented: Yes

Story Name: Player and Movement

Description: As a user I want to be able to see my character and to control my character by moving and jumping in order to make progress in the game.

Confirmation:
Functional:

- Have a movable player.
- The player moves right when the user presses the D key.
- The player moves left when the user presses the A key.
- The player jumps when the user presses the W key.

Non-functional:

Story Identifier: GPG003

Implemented: Yes

Story Name: Enemies

Description: As a user I want different types of enemies to appear in the game world and follow the player. Three different types will create a bigger variation for my gameplay and with enemies following me I will get a reason to move.

Confirmation:
Functional:

- Have three different types of enemies.
- Enemies will run towards players.
- If an enemy reaches the player, it will stop.
- The enemy will primarily follow the player in the x-direction.

Non-functional:

- Enemies should look different in order to tell them apart.

Story Identifier: GPG004

Implemented: Yes

Story Name: Player Shooting

Description: As a user I want to be able to shoot possible threats e.g enemies, in order to defend myself.

Confirmation:
Functional:

- The user aims with the mouse.
- The user shoots by clicking the Mouse1 button.
- The weapon can be reloaded.
- The player can only shoot when the weapon has reloaded.

Non-functional:

Story Identifier: GPG005

Implemented: Yes

Story Name: Weapon Projectiles

Description: As a user I want the projectiles fired from my weapon to move in the direction I am aiming to prevent enemies from coming too close to me.

Confirmation:
Functional:

- The projectile travels towards the designated target selected by the user.
- The projectile travels forward in the angle that has been calculated based on user input.
- After the projectile has traveled for more than 3 seconds it is destroyed.
- The projectile collides with enemies and pushes them away.
- The projectile travels at a constant speed.

Non-functional:

Story Identifier: GPG006

Implemented: Yes

Story Name: Jump Collision Reset

Description: As a user I want my character to regain the ability to jump when touching the ground so that I can jump more than one time during the game and not jump while I am in the air.

Confirmation:
Functional:

- Regain the ability to jump when touching the ground/platforms.
- Stop the player from being able to jump when in the air.
- Prevent the player to jump up and sit on the walls.
- Do not allow the player to climb walls when the user is spamming the jump button.

Non-functional:

Story Identifier: GPG007

Implemented: Yes

Story Name: Collision Detection

Description: As a user I want the game to react when e.g my character shoots an enemy or when an enemy hits me, so that an enemy and a player can take damage/lose health.

Confirmation:
Functional:

- Collisions between the player and enemies should lower players' health.
- Collisions between enemies and projectiles should lower the enemy's health.
- Collisions between enemies and projectiles should remove projectiles.
- Collisions between the player and platforms should reset the player's amount of jumps.

Non-functional:

Story Identifier: GPG008

Implemented: No

Story Name: Basic Wave Manager

Description: As a user I want to be able to fight waves of enemies so that there is a challenge and goal in the game.

Confirmation
Functional:

- A new wave will spawn x amount of different enemies (depending on which wave it is).
- Enemies will spawn at the edges of the map.
- The higher the wave, the more enemies will spawn.
- When the timer for each wave runs out a new wave will spawn.
- When all enemies in a wave are dead a new wave will spawn.

Non-functional:

Story Identifier: GPG009

Implemented: Yes

Story Name: Main Menu

Description: As a user I want to be able to see a main menu so that I can get an overview of the game before it starts.

Confirmation
Functional:

- When you press the play button, the game will open up the play menu.
- The player will be able to select a level in the play menu.
- When you press the settings button, the settings menu will open.
- When you press the exit button, the game will exit.

Non-functional:

Story Identifier: GPG010

Implemented: Yes

Story Name: Enemy Pathfinding

Description: As a user I want enemies to be able to traverse the map in order to reach the player and follow it if it moves from one place to another.

Confirmation
Functional:

- Enemies should walk towards the player and stop when the player is reached.
- Enemies shouldn't get stuck on obstacles.
- Enemies shouldn't walk through the map or obstacles to reach the player.
- Enemies shouldn't fly or jump uncontrollably.
- Enemies should take into account the players X and Y coordinates.

Non-functional:

Story Identifier: GPG011

Implemented: Yes

Story Name: Player Building

Description: As a user I want to be able to place blocks that can defend me from enemies in order to enhance my chances to complete the next wave.

Confirmation
Functional:

- The player can only build within a certain amount of tiles (a building range).
- You can only place a block on a tile that is connected to the ground or another building block.
- Blocks can only be placed on one of the tiles inside the map.
- The blocks can be destroyed by enemies when collided with.

Non-functional:

Story Identifier: GPG012

Implemented: Yes

Story Name: Build View

Description: As a player I want to be able to see where I can build in the form of a grid.

Confirmation
Functional:

- The grid the user can see is a transparent overlay covering certain tiles.
- The tiles the grid covers is the player's buildrange.
- This grid is constantly updated, following the player around.

Non-functional:

Story Identifier: GPG013

Implemented: Yes

Story Name: Weapon Selection

Description: As a user I want to have different kinds of weapons with more damage

Confirmation
Functional:

- Have at least 3 types of different weapons
- Different weapons have different damage.
- Different weapons have different projectiles.
- A player can only hold x amount of weapons at one time and change between them.

Non-functional:

Story Identifier: GPG014

Implemented: Yes

Story Name: UI

Description: As a user I want to be able to see the necessary information about my player as well as my progress in the game while I am playing. In order to get a better understanding of how the game is going for me.

Confirmation
Functional:

- The health bar changes when the player gains/loses HP.
- The wave indicator is incremented whenever the user completes a wave.
- A counter displays the amount of ammunition the player has left.
- A text indicates which weapon is selected by the user.

Non-functional:

Story Identifier: GPG015

Implemented: Yes

Story Name: Enemy Tile Destroy

Description: As a user I want enemies to attack and destroy tiles I have placed to make the game more challenging.

Confirmation

Functional:

- Enemies can destroy tiles placed by the user when they collide with blocks.
- The predefined tiles in the level varies depending on level.

Non-functional:

Story Identifier: GPG016

Implemented: Yes

Story Name: Various Worlds

Description: As a user I want to be able to be in various worlds (levels) in the game, with different tiles and backgrounds.

Confirmation

Functional:

- The environment of the game world changes every time the user switches between a level.
- Different enemies destroy tiles with different speeds (depending on their attack).

Non-functional:

- The game world's terrain should be built up with the use of different tiles (e.g. grass tiles).

Story Identifier: GPG017

Implemented: Yes

Story Name: Exit Menu

Description: As a user I want to be able to pause the current game by pressing the Escape button. After that, I can decide whether or not to return to the main menu.

Confirmation

Functional:

- The user can press the Escape button to pause the game. After that the user will be given an option to return to the main menu.

- The user can resume the game by pressing the Escape button again.

Non-functional:

- Game is paused when the exit menu is opened.

Story Identifier: GPG018

Implemented: Yes

Story Name: Movement Animation

Description: As a user I want to see an animation when my character and the enemies are running or jumping, in order to get the visual information of which action the player or the enemies are performing currently.

Confirmation
Functional:

- Have a sprite of the player running when it is running.
- Have a sprite of the player jumping when it's jumping.
- Have a sprite of the enemy running when it is running.
- Have a sprite of the enemy jumping when it is jumping.

Non-functional:

Story Identifier: GPG019

Implemented: Yes

Story Name: Game Settings

Description: As a user I want to be able to change the characteristics of the game through a settings menu. As a user this will make it easier to see all my options and alternatives I have in the game and make me have more control over my experience.

Confirmation
Functional:

- The user can rebind a key when the user presses a control button in the settings menu.

Non-functional:

Story Identifier: GPG020

Implemented: No

Story Name: Audio

Description: As a user I want to be able to hear sounds from various sources in the game while playing it.

Confirmation
Functional:

- Enemies make sounds.
- Jumping makes a sound.
- Placing blocks makes a sound.
- Firing a projectile makes a sound.
- Taking damage makes a sound.

Non-functional:

Story Identifier: GPG021

Implemented: No

Story Name: Player Pickup

Description: As a user I want to be able to pick up weapons and building blocks from the ground.

Confirmation
Functional:

- The player can pick up weapons and building blocks by walking over them (when they are on the ground as a dropped entity).
- The player can not pick up more weapons than the maximum carry amount.
- The player can not pick up building blocks if the stack is full.

Non-functional:

Story Identifier: GPG022

Implemented: No

Story Name: Player XP System

Description: As a user I want to be able to level up after I kill enough enemies so that I can get stronger and have an incentive to play the game.

Confirmation
Functional:

- The player gets XP by killing enemies.
- The player levels up after enough XP has been reached.

- When the player levels up, it's stats (attack, speed, hp) increases.
- On each level-up it gets progressively more difficult to level up (more and more xp is required to level up again).

Non-functional:

2.2 Definition of Done

For a user story to be defined as done it has to meet all of its own acceptance criteria. All involved code and classes related to the user stories must be documented with java docs, in order to be understood by everyone associated with the project. The user stories acceptances will be tested through concrete test classes, human code evaluation, and user acceptance testing.

2.3 User interface

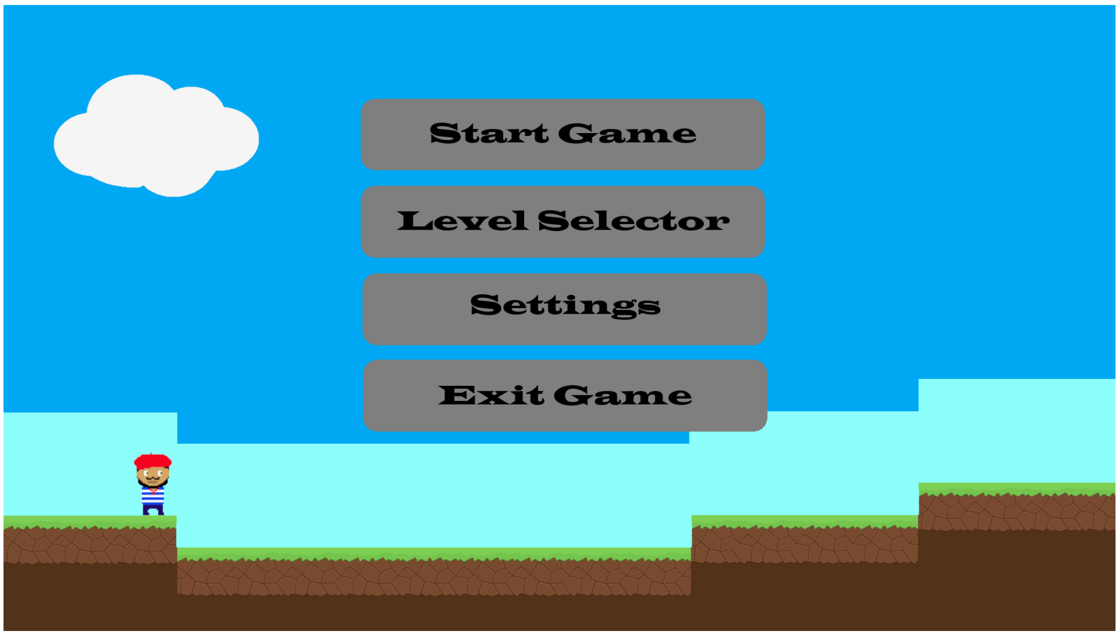

Figure 1: Sketch of main menu.

When the application first starts the user is directed to the main menu of the game. In the main menu there are buttons for selecting a level, changing settings and exiting the game.

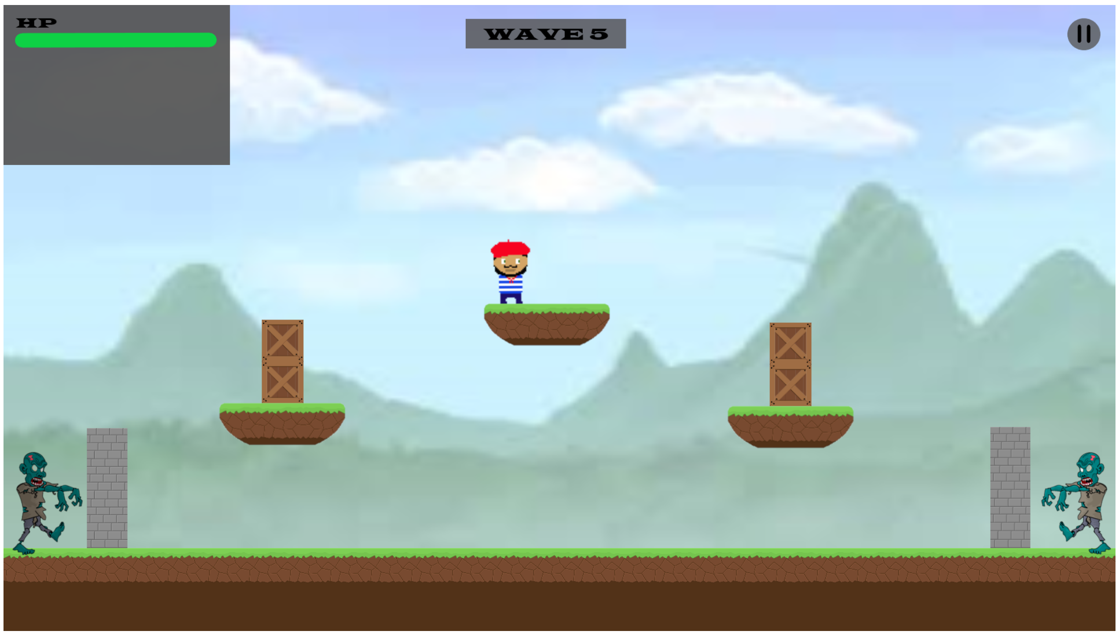

Figure 2: Sketch of in game view.

The gameplay scene is shown after the user selects a level to play. When playing a level the user is for instance able to see the wave the player is on, the player, player's hp and enemies in the game.

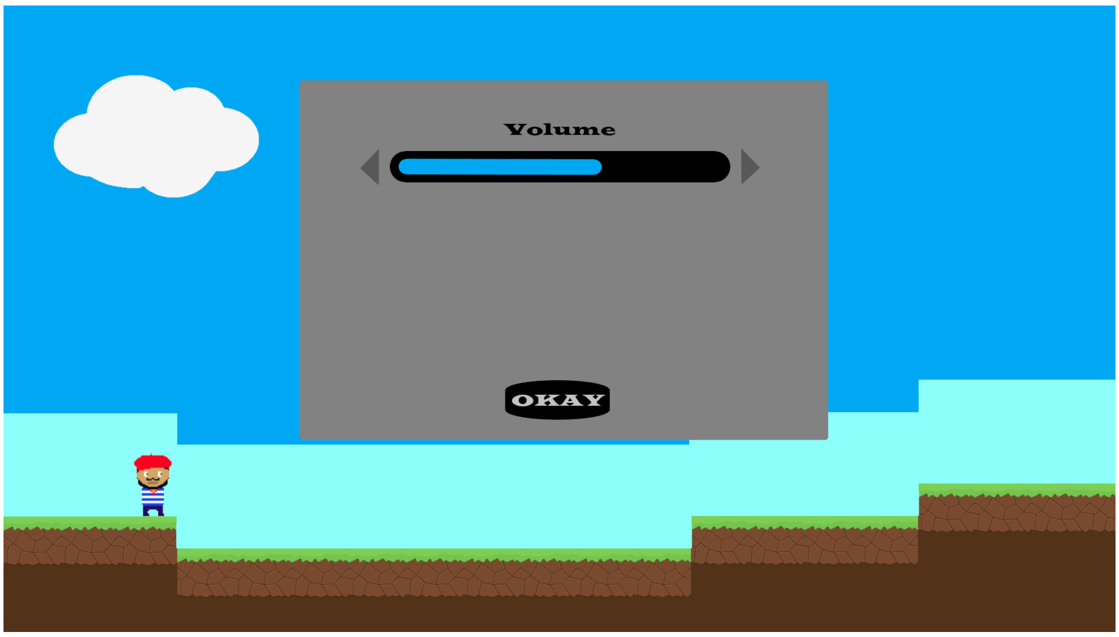

Figure 3: Sketch of settings menu.

Settings scene is shown when the user clicks on the settings button in the main menu. In the settings scene the user is able to change keybinds.

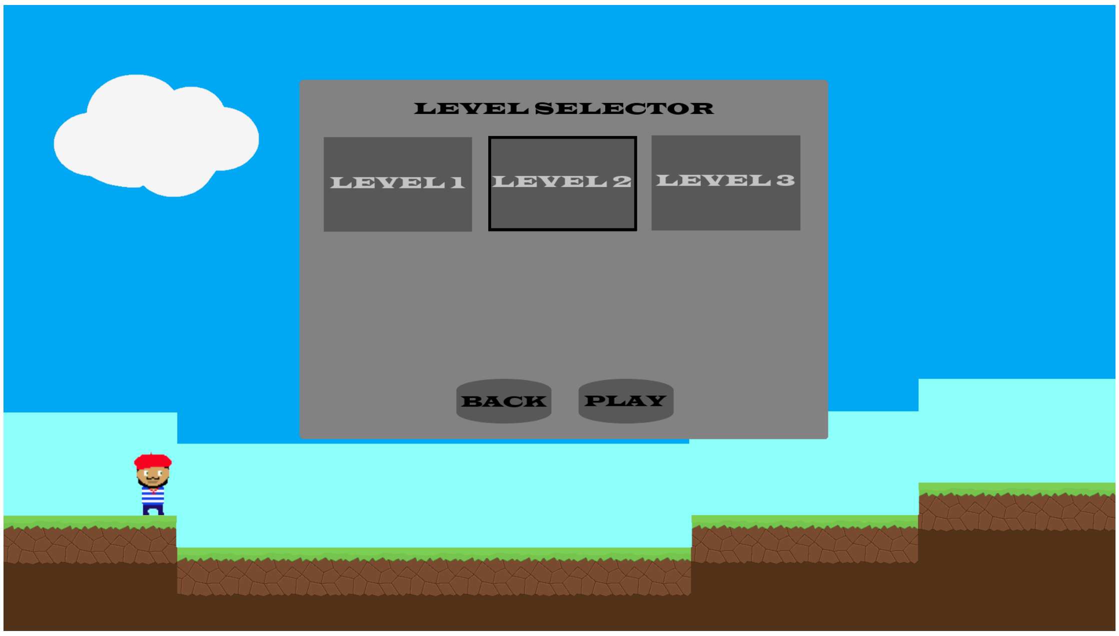

Figure 4: Sketch of level selector view.

The level selector scene is shown when the user clicks on the select level button in the main menu. In the selector scene the user is able to select the level to play.

3 Domain model

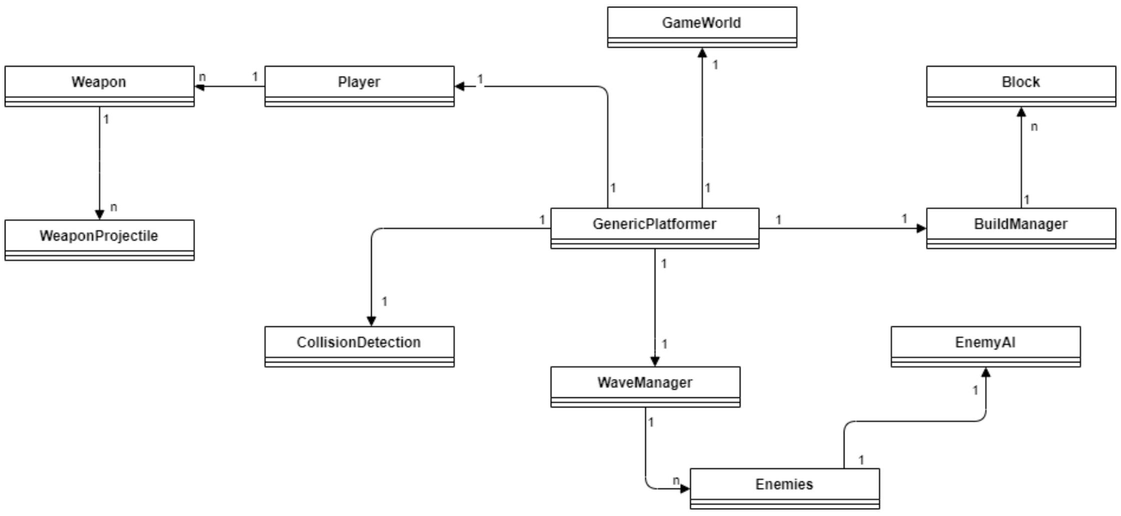

Figure 5: UML of the domain model.

3.1 Class responsibilities

GenericPlatform:
Is used as a root class for the game and the model package. Every data and information passing from the model to either the view or controller should pass through GenericPlatformer in order to achieve low coupling.

BuildManager:
Handles player being able to spawn and place blocks. A block is able to spawn if the player can reach it, the tile to spawn it on is empty, the tile to spawn it on is connected to another block and the block to spawn isn't on the player.

Block:
This class creates a collidable block entity which can take damage and be destroyed.

CollisionDetection:
Manages all collisions between different entities in the game and is responsible for calling all necessary methods used to resolve/react to said collision.

WaveManager:
The class handles waves in the game. It calculates which types of enemies should spawn and creates new waves of enemies. The amount of enemies that spawns depends on which wave it is (the higher the wave the more enemies). It will generate a new wave when all enemies are dead and/or when the timer for the wave runs out.

Enemies:
The class responsible for creating a component with the ability to give an entity all common

attributes and behaviours of an enemy.

EnemyAI:
Manages and contains all functionality for the Enemy AI. Functionality includes moving to the player while he is moving or standing still, jumping over obstacles when path to player is interrupted, traversing floating platforms to reach player, etc.

GameWorld:
This class contains all methods for spawning the game world entities, such as the player and platforms.

Player:
The class responsible for creating a component the ability to give an entity all attributes and behaviours of a player to a specific entity.

Weapon:
The class that is responsible for creating instances of WeaponProjectile as well as managing a "weapon's" ammunition counter and the attributes of different types of "weapons".

WeaponProjectile:
The class that is responsible for creating a projectile entity and calculate the movement direction of this entity.

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).

Peer Review of group 16 “Päron”
Made by group 5 “Hmmm”

Do the design and implementation follow design principles?

They follow Command Query Responsibility Segregation, each method either returns data or performs an action (a change in-state). SRP is followed where each module/class/method has one responsibility.
A principle that is not fully being followed is the Open Closed Principle. The HeaderController class uses a switch statement to underline each header depending on which page you are on. For example, if you are on the Account page, the header text Account is underlined. OCP is not followed here since if you were to add a completely new page to the project you’d have to manually go into the switch statement and add a new case. In this new case, you’d then have to tell it that when you are on this new page it’s the new header that should be underlined.

Does the project use a consistent coding style?

For the most part, the project uses a consistent coding style. The only difference in the coding style that was found was the use of curly brackets. Almost all classes except for Pilot, License, FlyingClub, and Airplane have their opening curly brackets begin after a space. The classes not following this style instead uses no space.

Is the code reusable?

Sure. The model doesn’t seem to be connected to the view, which allows for different kinds of implementations of the model.

Is it easy to maintain?

Yes. The project is well separated into multiple distinct classes which allow for easy maintainability. The use of many controllers also helps.

Can we easily add/remove functionality?

There are currently two empty interfaces in the code (iBorrower & iBookable), if these are implemented correctly. It would mean that adding new bookable objects or borrower objects could be achieved easier by implementing these interfaces on the objects. Therefore, making it easy to add functionality in that area.

Adding a new page to the project, however, is difficult (as stated under OCP in the design principles).

Are design patterns used?

Yes, one design pattern that is used is the factory pattern. The service package in the project has a ServiceFactory where you can create all services that exist. Singleton Pattern is used for FlightBuddy.

Is the code documented?

The code’s documentation is lacking. Around 40-50% of the public methods in the project have Javadoc comments.
There is a big difference in code documentation between the classes. For example, the big class AccountWizardController with 200 lines of code is well documented where all its public methods

have Javadoc comments. Most of the smaller classes on the other hand, such as Flight, License and MyClubController, etc. have no comments whatsoever.

Are proper names used?
Yes, the names make sense and are descriptive enough. They also fit well with their given variable types.

Is the design modular? Are there any unnecessary dependencies?
There are some classes in the project that have unexplainable dependencies on each other. One example being the class “MyClubListItem” that has a dependency on the “MyClubController” for no good reason. Similar unnecessary dependencies occur throughout the project here and there, but this seems like a problem that would otherwise be ironed out towards the completion of the project.

Does the code use proper abstractions?
No abstractions were found.

Is the code well tested?
The code overall is not well tested. Model is the only package that currently has any tests with 5/9 of its classes containing any tests whatsoever. However, the few methods that have been tested are well written.

Are there any security problems, are there any performance issues?
The password to be filled in for the new account, in step 3 of the registration process, is shown as plaintext instead of being hidden. Hide the password in the same way that was done on the login page, but let the user also show the password so that he or she can make sure that the password is properly entered.
It is hard to find performance issues in such a minimalistic application, but one thing that can be noted is that exceptions are being thrown whenever you attempt to navigate through the different pages on the start page. This will undeniably cause slowdowns, albeit somewhat negligible at this stage.

Is the code easy to understand? Does it have an MVC structure, and is the model isolated from the other parts?
The code is relatively easy to understand and follow. MVC structure looks great with the model not depending on any view or controller. However, the coupling between packages in the model using a private variable and creating instances of model classes. Instead, you could just depend on one root class and take advantage of chain calling and the facade pattern. For example with a call like flightBuddy.createPilot() instead of creating a new pilot in AccountWizardController and get more dependency.

Can the design or code be improved? Are there better solutions?
The way that the packages (modules) are structured in the project could be vastly improved. The packages that already exist could be divided into sub-packages in order to avoid having overly complex packages. The project, the way that it is as of now, gives the developers reviewing the project a false perception of high cohesion in the individual packages. An example being the classes “Flight” and “License”, in the package “model”. They certainly both belong to the model of the project, but they have little reason to be in the same package and they could instead be separated into sub-packages.