System design document for Group 16

Group 16

21st October 2022

Contents

1 Introduction

This is a design document explaining the inner workings of Feyrune, a Pokémon-style RPG, but with the ability to be easily extended with new functionality to differentiate itself.

1.1 Definitions, acronyms, and abbreviations

Feyrune:	The name of the application
DoD:	Definition of Done
Player:	The user controlled element of the game used to interact with the rest of the world.
Tile:	A square of 16x16 pixels.
Tile set:	An image containing multiple different tiles.
Map:	A collection of tiles in a grid.
Libgdx:	A graphical library and game engine
Tiled:	A program for drawing maps from tile sets

2 System architecture

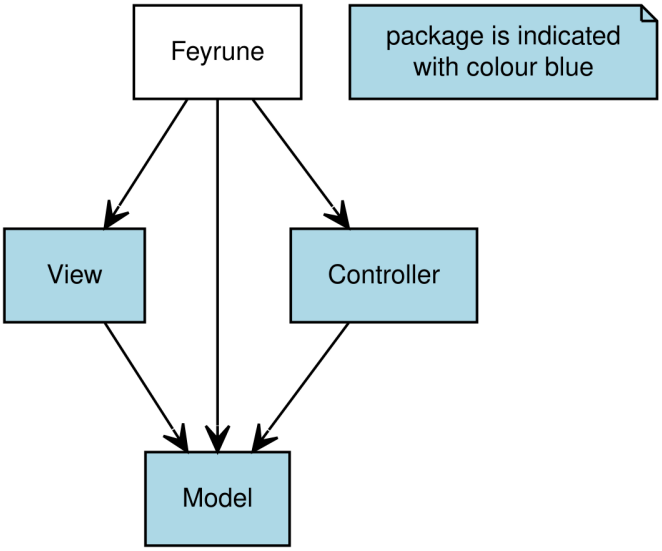
Feyrune was mainly created using libgdx , a java game engine, to render and run the game [1], and Tiled to create and design the maps [2].

When the application is started you as a player is thrown into a world that can be explored. When the application is closed nothing is saved and if you start the application you will be at the same staring point as the first time you started the application.
The application flow can be described by the following game loop:

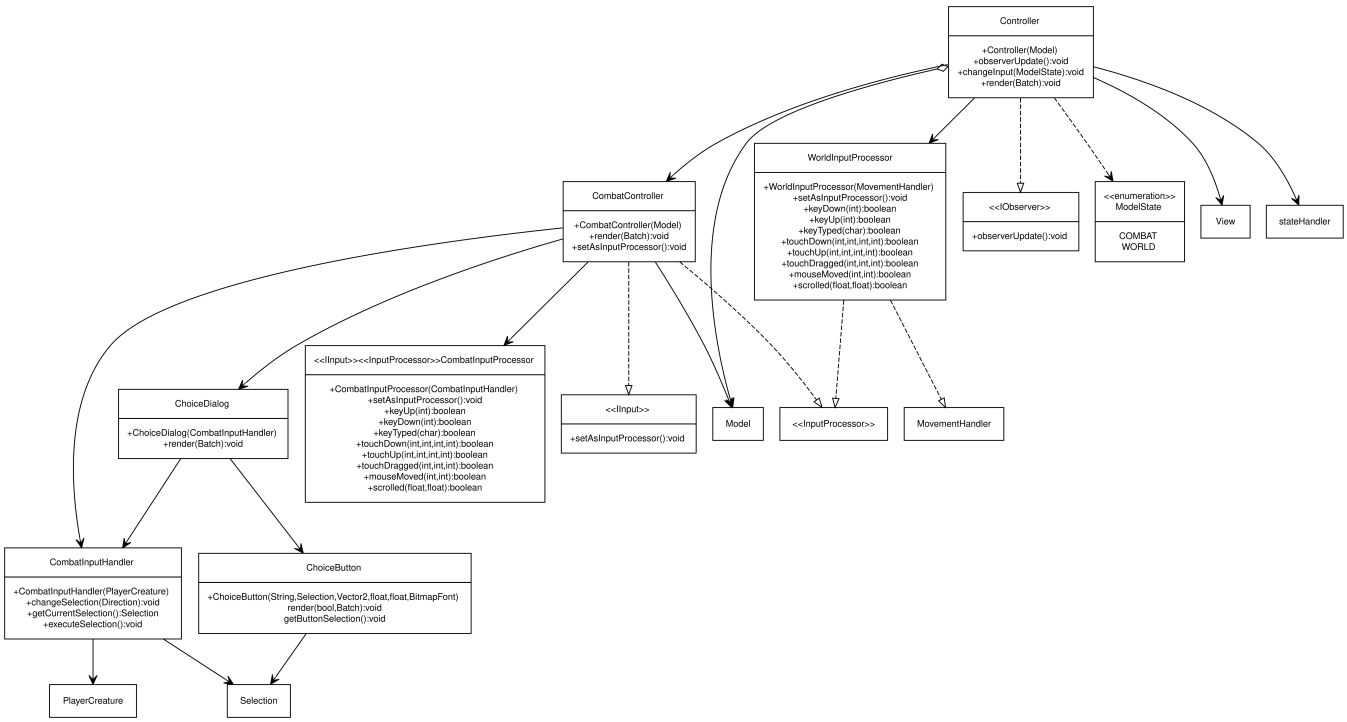
1. The player explores the map.
2. The player encounters an enemy.
3. The player defeats said enemy and gets stronger or is defeated and have to get stronger to defeat said enemy and have to fight easier monsters to get stronger before tackling the stronger monster again.
4. The player continues exploring.

3 System design

3.1 Packages



3.1.1 Controller



The controller is a bit different from the view seeing how it mirrors the model in its separation of combat and overworld. It also depends on the view, as it uses the view's batch to render its components.

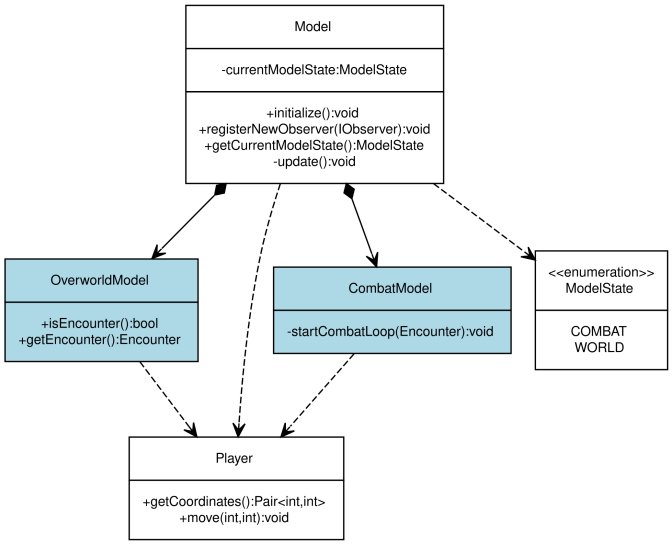
It uses the interface IInput, that extends the libgdx interface InputProcessor. An input processor overrides all button events and you can only have one active at a time, meaning that as the model changes state, the input processor also has to change. When input later is received, the input processor is the component

responsible for processing this and sending the correct information to the model.

The MapInputProcessor can take input from the WASD-keys and then has a timer that controls if it should send this along as movement or not. If it should, it sends the movement direction to the model.

The CombatInputProcessor contains four buttons rendered in the controller. This input processor takes input from both the WASD-keys as well as the enter key; the WASD-keys to register which of the four buttons is currently active, and the enter key to know whether that button's connected functionality should be executed or not.

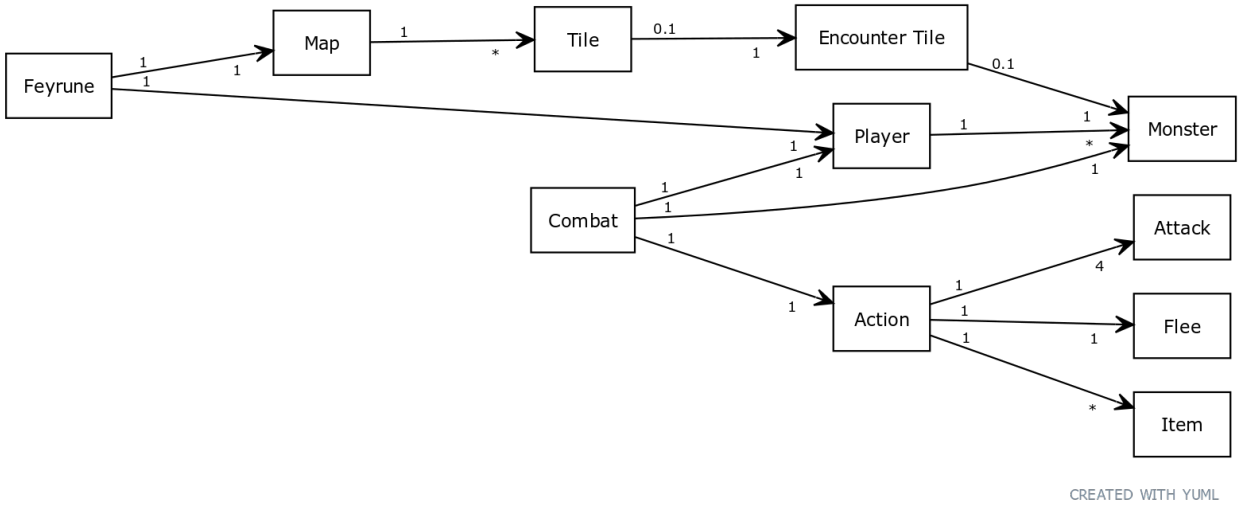
3.1.2 Model



The model is split into two separate models; one which models the overworld, and one which models the combat. The main model class is only used to control which one of those should be considered active and to oversee their relation and information sharing between themselves.

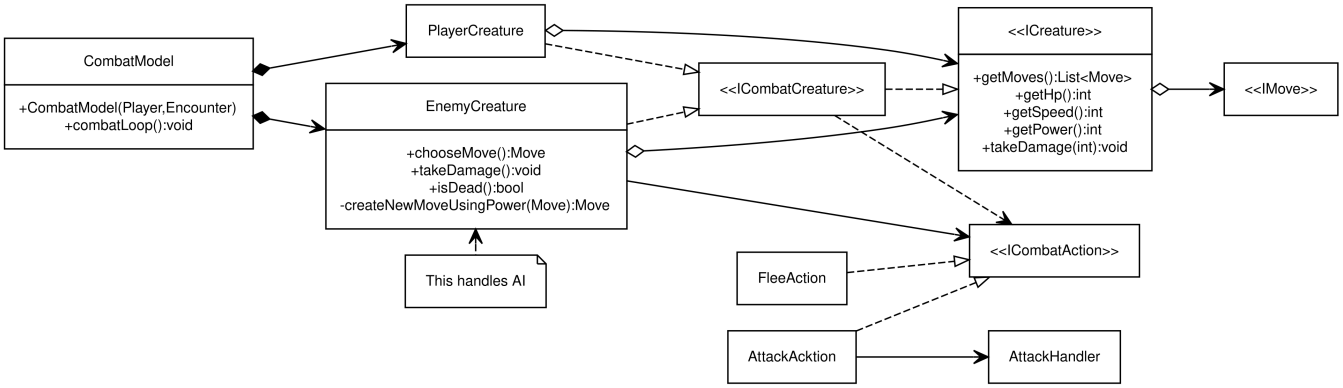
As can be seen in the domain model, both the overworld- and combat-model share the same player, which is stored in the main model class.

The domain model of the application



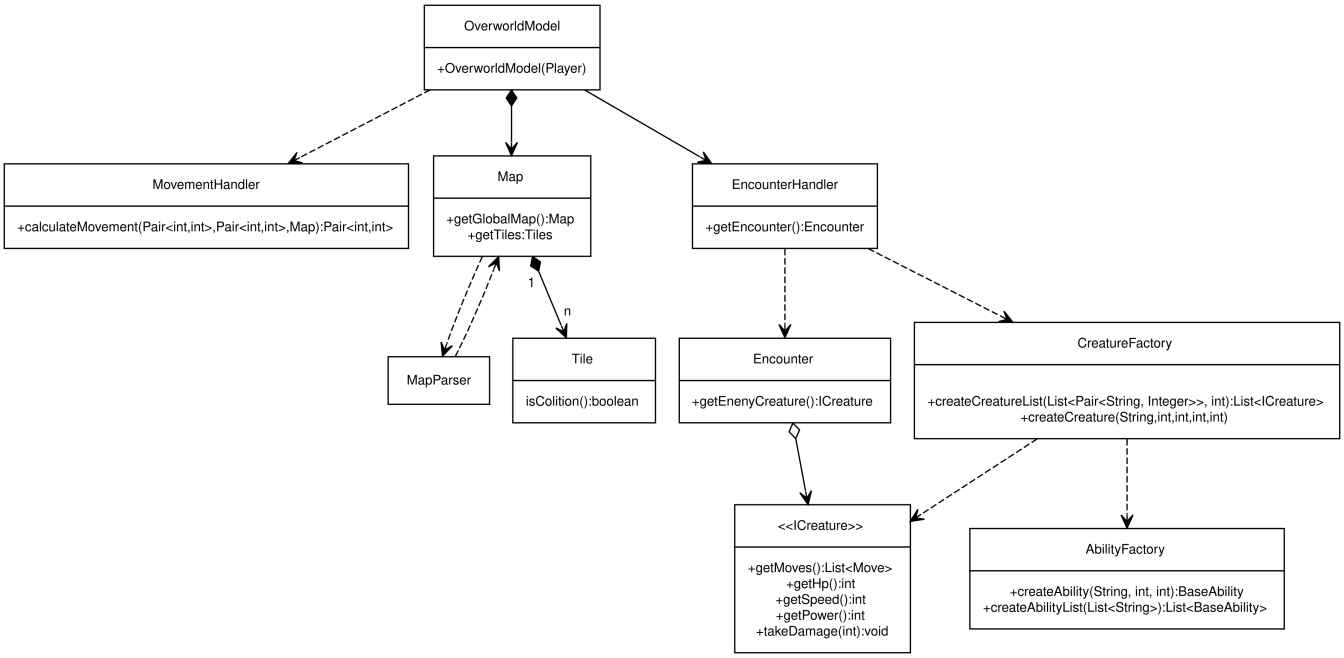
In the domain model, you also see that there are two entry points to the model; one named feyrune, which has a direct relation to map, and another one named combat. This is a separation we have created with our two separate models as their relation practically is just the player and some monsters, but seeing how the monsters aren't constant as long as they aren't owned by the player, the connection there is not as strong.

3.1.3 CombatModel



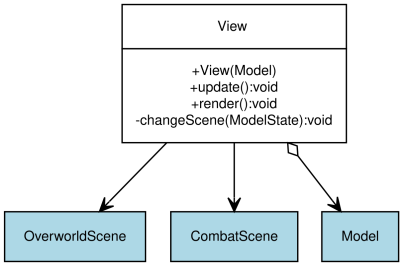
When the CombatModel is created it only holds the player as a reference and then instantiates new encounters or combats as the main model dictates.

3.1.4 OverworldModel



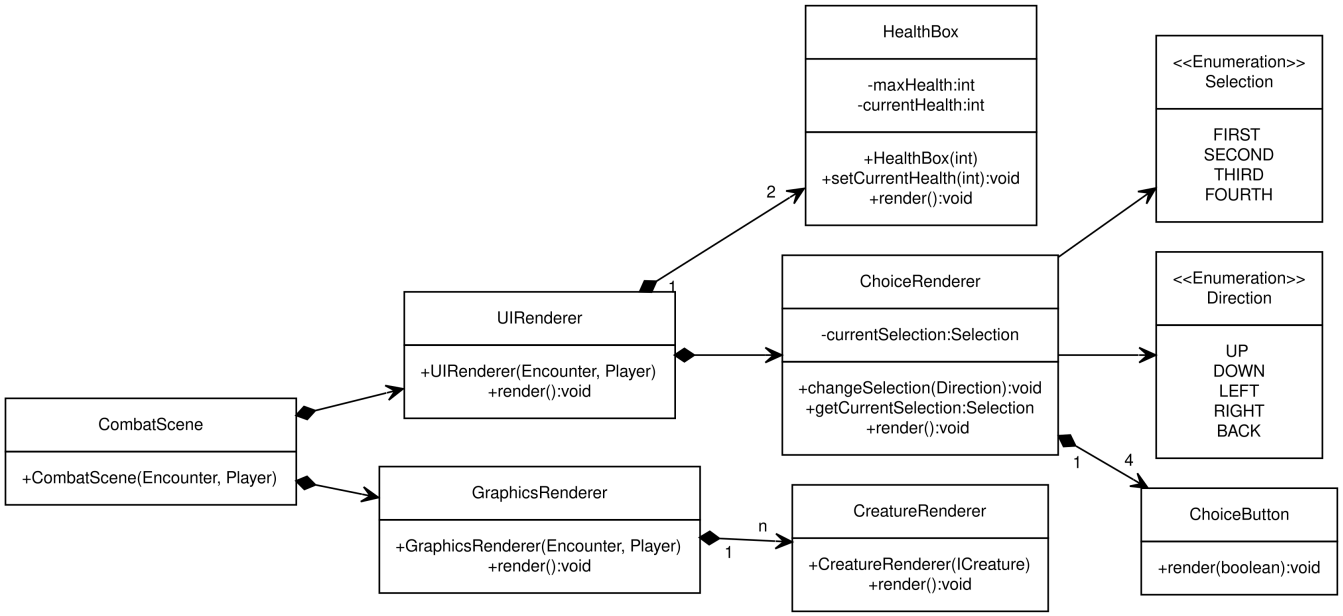
The OverworldModel handles the relation between player and map. It knows where the player is and what attributes a tile can have, if you can move there or if it can create an encounter. As a new encounter is created it is sent to CombatModel and handled there.

3.1.5 View



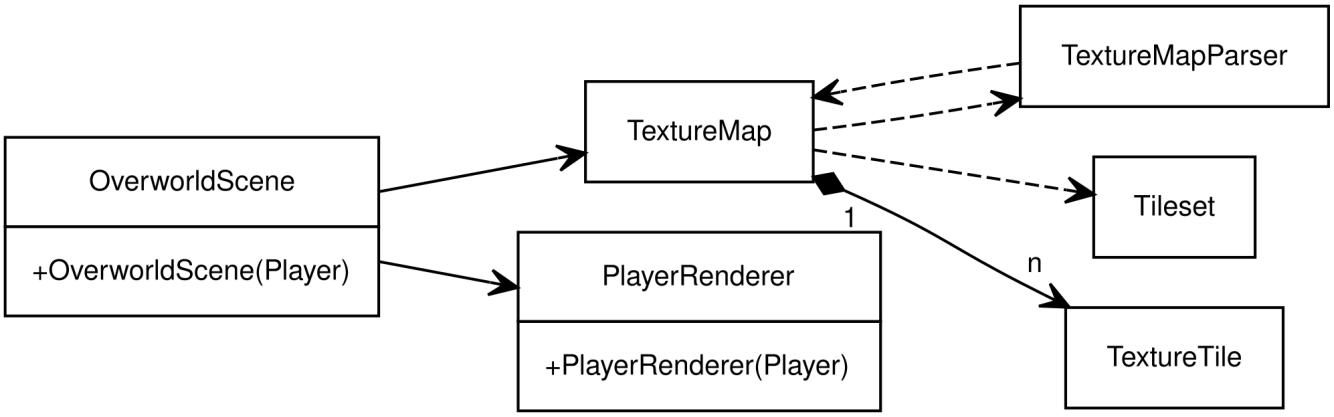
The view mirrors the model in that it is separated into two different parts, overworld and combat. And mirrors the model in what tees two are responsible for. It also has a dependencies to the model as it reads most of its data from there.

3.1.6 CombatScene



As the model state changes from overworld to combat, CombatScene reads all the information that graphically needs to be represented about the encounter from the model. It then renders all that information, such as health and which monsters are active in combat. It does not render any of the interactive UI elements as that is the controller's responsibility.

3.1.7 OverworldScene



The overworld scene is responsible for rendering the map. The data it needs to render this is taken from the same data file as the model uses, but crucially it is parsed and stored separately from the model's information. OverworldScene is responsible for drawing and animating the player at the correct position, as well as moving the camera accordingly.

3.2 Design Patterns

Throughout the code, Group 16 has tried to implement design patterns wherever feasible. Examples of design patterns used are:

3.2.1 Model-View-Controller

The entire project is written with the MVC pattern in mind. The model exists in its own vacuum without any references to either the view or the controller while the view and controller contains references to the model. The controller also has a reference to to the view, this is to implement the sprite Batch that is used to draw things on the screen. This is to allow the controller to render its own buttons.

3.2.2 Factory pattern

We use a factory to create monsters, their abilities, and encounters, this is to make it easier to exchange e.g. our monster class, BaseCreature with a new one, if someone wants to extend the code with specialized monsters that maybe are able to have special abilities or different types.

3.2.3 State pattern

State pattern is used in maps to hide complexity so that other classes don't know which exact map is used. We chose to use the state pattern to easily change between maps, without having to use observers everywhere and update references throughout our codebase. There is also a potential to add one more state pattern which would handle the world state. This is not implemented but would be a good replacement that we have now that is quite hard to extend.

3.2.4 Observer pattern

Even if the code is running continuously and is polling, an observer is implemented in most places where we don't expect a change every frame. We have created several different versions of observers, depending on where they are used, this is because in some cases we only need to ping a class when it's updated to let it know what it should do, while in some cases we might have to send through some data, like a button press. We chose to separate this because without sending data we would have to include references to all objects the update could come from, and if we always sent all data that could be required we would run in to problems like the map having to send keyboard presses when it's getting changed.

3.2.5 Façade pattern

The Façade is implement in the parsing of the map. The facade is there to hide all functions about parsing and only one static function exists in class called map handler that returns a map from a relevant path. This is good because as a user of this functionality that is by all means what you want to do. You do not want to manually sap parse this and apply it to these objects and so on. The only thing a user could do wrong now is sending a faulty path and we catch that error. This reduces the potential error by quite a bit.

4 Persistent data management

No data is stored and all images are stored using tileset .png files in the assets folder. The maps are stored in a similar vein by using .tmx files that are also stored in the assets folder. The .tmx files contain *all* relevant data for the given map.

5 Quality

5.1 Application quality

To ensure the application is of great quality, every update of it goes through several levels of testing, beginning with a personal review from the code writer. When the writer feels their contribution is good enough, the code is tested using JUnit[3] to make sure it functions logically correct, even in common corner cases.

After the JUnit testing, a pull request is created, forcing at least one other contributor to analyze it and test the code themselves before it being added to the source code.

All tests written can be found in the "core/src/tests"-folder.

5.2 Issues

Following is a list of all known issues:

- What ability to use in combat is chosen for you.
  - This would have been solved by creating another version of the ChoiceDialog where the button texts are exchanged for action names and used instead to choose an action.
- The classes aren't documented, only their methods.
  - This is because the classes are not meant to be used by the user, but only by the game engine.
- There are unused methods and imports.
- Overuse of package hierarchy leads to a bit too many public methods.
  - e.g. createCreatureList().
  - We could include a facade in each package to control access, but it probably would require extensive restructuring of our packages.
- Encounterhandler isn't really a handler, it should be called EncounterFactory, as that is what it is.
  - We could rename the class to EncounterHandler.
- AttackHandler is an unnecessary class.
  - The method evasiveManoeuvre() should be placed either inside the creature class (intertwined with takeDamage and dealDamage) or inside the CombatModel.
  - The method handleAttack() should be placed inside the CombatModel instead.
- We have two packages: creature and creatures, which should be merged into one.
  - Move all classes in the creature-package into the creatures-package.
- When using a transporter, you can see how the player flies across the map.
  - We should separate the player movement based on transportation and regular movement, and only use the animation for the regular movement.
- The health bar changes to use the current hp as maximum hp when entering combat.
  - Create methods in the creature classes to fetch max hp separate from current hp.

6 References

References

[1] libGDX, *Libgdx*, 10th Oct. 2022. [Online]. Available: `https://libgdx.com/`.

[2] T. Lindeijer. "Tiled." (10th Oct. 2022), [Online]. Available: `https://www.mapeditor.org/`.

[3] T. J. Team, *Junit 5*, 13th Oct. 2022. [Online]. Available: `https://junit.org/junit5/`.