Enrico Marchionni
enrico.marchionni@studio.unibo.it

Francesco Cipollone
francesco.cipollone@studio.unibo.it

Raffaele Marrazzo
raffaele.marrazzo@studio.unibo.it

June 14, 2023

Abstract

Darwin's Quest [Dar59] is a multi-level structured video game set in a post apocalyptic world, shaken by climate change. The goal is to survive the natural selection by completing numerous battles, after choosing your genetically modified Banions[1].

[1] *Banion*, meaning Battle Companion, is the monsters' name.

Contents

Chapter 1

Analysis

1.1 Requirements

Functional

- The player can choose between two game modes: Normal, Hard;
- At the beginning of the game, the player must be able to select four starter fight companions from a number of multiple choices;
- The game interface will consist of a board of levels in which the player moves. Each level will prompt a battle;
- The player's movement will be managed by tiles. Each tile is a battle tile. The movement will be determined by the use of a die. Said die value corresponds to the number of tiles that the player can move;
- The companions are able to evolve based on a linear evolution. Each time an evolution is triggered, the companions' statistics will increase;
- The player has to sequentially complete a series of levels, with increasing difficulty until meeting the final boss. The battles are engaged in a 1v1 turn-based combat fashion, with the ability to switch between companions or perform moves.

Non-Functional

- The battle needs to be challenging for the player and will require a careful strategic plan;
- The game performances must be acceptable;

- The game has to be portable and compatible with Windows, macOS, and Linux systems.

1.2 Domain Model

The player can move inside the board map. The map contains the player, the enemies, and battle tiles. Landing on the latter will prompt the beginning of a battle and losing the fight will trigger the game over. The player owns four Banions. Each Banion has an intrinsic elemental type, such as fire, water, grass, rock, air, electro, and is associated to a set of moves, which are divided into groups based on the Banions' elemental type. A certain Banion with a certain type only retains moves of its same elemental type plus neutral moves, e.g. A fire Banion can only use fire moves and neutral moves, and cannot perform other types' moves. Banions in our game are the following:

Banion	Element
Hefty	Air
Aotori	Air
Buzzwings	Air
Jolbat	Electro
Sparkleap	Electro
Zapameleon	Electro
Infernhog	Fire
Blazechick	Fire
Scorchspore	Fire
Florastump	Grass
Herbroot	Grass
Flingleaf	Grass
Stonemaul	Rock
Spebble	Rock
Stonohorn	Rock
Aquamuck	Water
Hydrashell	Water
Quacktide	Water

Table 1.1: Banions

The available moves for the Banions are:

Move	Element	Damage
Tornado	Air	3
Aero Blast	Air	3
Breeze	Air	3
Monsoon	Air	4
Aero Slash	Air	4
Spark	Electro	3
Lightning	Electro	3
Thunderbolt	Electro	3
Volt Blitz	Electro	4
Surge Wave	Electro	4
Fireball	Fire	3
Flame	Fire	3
Torch	Fire	3
Magma Burst	Fire	4
Brazier	Fire	4
Meadow	Grass	3
Foliage Strike	Grass	3
Seed Barrage	Grass	3
Verdant Beam	Grass	4
Spores	Grass	4
Earthquake	Rock	3
Landslide	Rock	3
Bedrock Bash	Rock	3
Boulder Burst	Rock	4
Pebble Pummel	Rock	4
Tsunami	Water	3
Whirlpool	Water	3
Aqua Blast	Water	3
Coral Spear	Water	4
Waterfall	Water	4

Table 1.2: Moves

Some Banions will also have neutral-type moves:

Move	Element	Damage
Claw	Neutral	2
Roar	Neutral	2
Bite	Neutral	2
Impale	Neutral	2
Kick	Neutral	2
Strike	Neutral	2
Blast	Neutral	2
Slash	Neutral	2
Swipe	Neutral	2
Pulse	Neutral	2
Beam	Neutral	2
Slam	Neutral	2
Hurl	Neutral	2
Punch	Neutral	2
Poke	Neutral	2

Table 1.3: Neutral Moves

Elemental reactions, that are bounded to moves, are defined as follows:

Player \Enemy	Air	Electro	Fire	Grass	Rock	Water
Air	-	*	*	-	+	+
Electro	*	+	*	-	*	*
Fire	*	*	*	+	*	-
Grass	+	+	-	*	-	*
Rock	-	*	*	+	*	*
Water	-	*	+	*	*	*

Table 1.4: Elements

Each companion in the game will have a personal set of statistics, such as attack (ATK), defense (DEF), health points (HP). The values make up the base statistics of a certain companion.

In every level, there are two player entities: the player and the opponent (NPC[1]). These entities deploy their companions, which will fight 1v1. A level

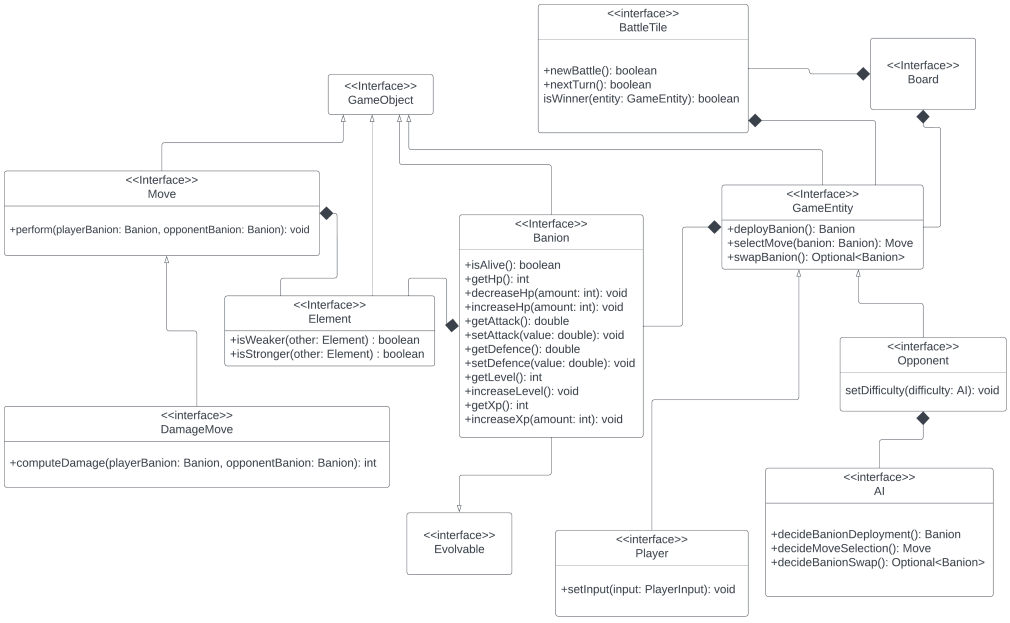
[1]**NPC: Non-Playable Character**

consists of a battle phase, and leveling phase:

- **Battle phase**: during the player's fight turn, it is possible to switch their active companion to a different one, or the player can perform a move. Upon the player's active companion's defeat, they will be forced to switch to another one. If the player is out of eligible companions the fight is over.
- **Leveling phase**: upon the player's victory, Experience Points will be assigned to the Banions. If a Banion obtains enough XP[2] it will evolve.

The fight mechanics are considered a project challenge for their complexity 1.1, from the elemental reaction logic to the moves' management.

Figure 1.1: Model UML



[2]**XP**: Experience Points

Chapter 2

Design

2.1 Architecture

This project is developed in MVC (Model, View and Controller) architecture. In our architecture the **Controller** is the entry point of the application. It instantiates the **View** that instantiates the **Controller** itself. The latter represents the main class, and it coordinates the interaction between the **View** and the **Model**, which is instantiated at a proper time by the **Controller**, following the MVC architectural pattern.

In our architecture the main classes are **JavaFXView** (View), **ControllerImpl** (Controller) and **EngineImpl** (Model). The first one is responsible for the collection of input and shows the output, the third one contains game main components and the second one is the link between them.

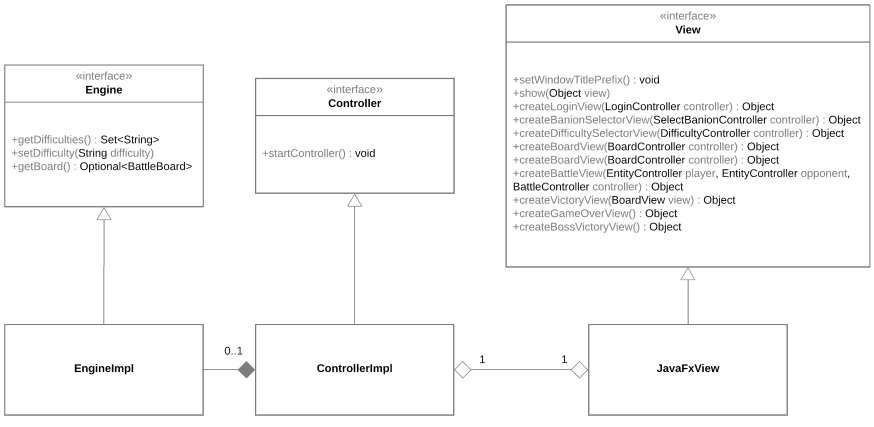


Figure 2.1: UML Model-View-Controller

2.2 Details

Enrico Marchionni

Game Difficulty

Problem Giving the possibility to choose between different thought games. Each difficulty should determine for example the number of tiles for the *Board*, the moving *Strategy* in it, the number and *Element* of the *Opponent*'s *Banion*s, the *AI* for each *Opponent*... A requirement is that this problem has to be resolved in an extendible way. So once the system is built, adding a difficulty must be easy and fast according to *OCP* (open closed principle).

Solution The element *Board* is the main entity of this game because it determines how much levels have to be completed to win the game. So the main *Model* class that is responsible for the difficulties is the *Engine* class. It retrieves the *Board* and everything else go by itself only by selecting the *Difficulty*. The difficulty has the goal to decide the *Board Strategy* of movement, the number of levels and the *AI* associated with the *Opponent*. Each *Opponent* will be created by the *Board* at the right moment through an *OpponentFactory* that was created by the *Difficulty* itself.

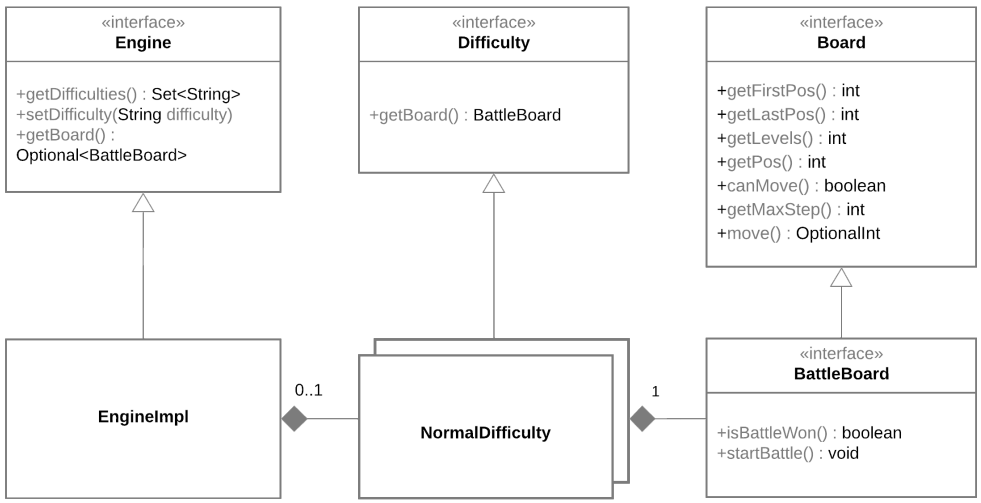


Figure 2.2: UML difficulty

Pattern The *Difficulty* decides the *Board* step of movement by passing a *PositiveIntSupplier* to *Board* constructor via *Strategy*. Furthermore the *Difficulty* decides also the *OpponentFactory* implementation to pass to the *Board* via *Strategy*. The *OpponentFactory* can be considered as *Abstract Factory* because every single implementation of *Difficulty* could pass a different implementation of *OpponentFactory* to the relative *BattleBoard*.

Banion identification

Problem We wanted to make two *Banion*s with same name and statistics be considered different at *Domain Model* level.

Solution After evaluating different options such as not overriding *Banion*'s equals, we decided to use *java.util.UUID* class to identify every single *Banion* instance. In this way it is possible for a player for example to have in the inventory two *Banion*s with the same name twice, because they are different.

Banion modified

Problem In the *View* each *Banion* represented could be modified and should be seen always in most updated state. Furthermore potentially there could be more that one *View* representing the same *Banion*.

Solution I decided to make the Banion observable for its changed state.

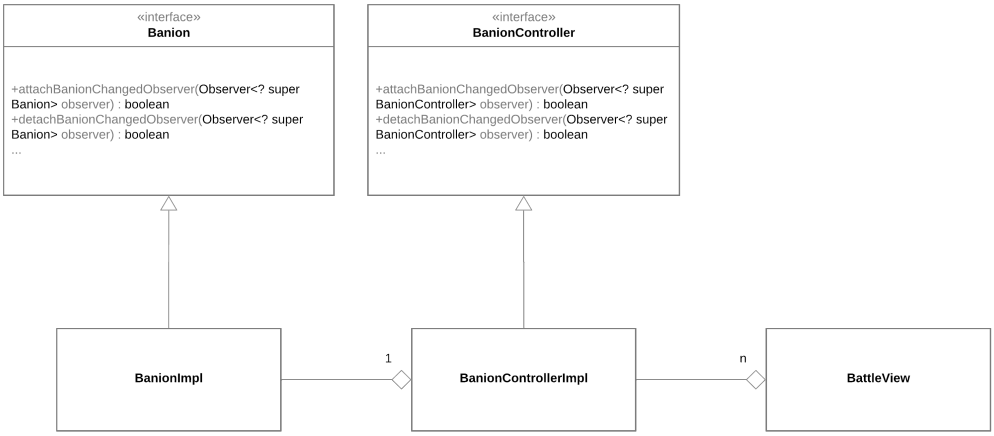


Figure 2.3: UML Banion Observer

Pattern The *Observer* pattern was adopted to notify changes of state from the specific *Banion*. *BanionImpl* is the subject (the observable object) while *BanionControllerImpl* and then *BattleView* are the Observers. In specific *BanionImpl* is observed from *BanionControllerImpl* while *BanionControllerImpl* is observed from *BattleView*. *BanionControllerImpl* is a "read only" *Banion* wrapper that decouples *Model* from *View*.

*Element*s, *Move*s and *Banion*s Generation

Problem Generating game entities: *Element*, *Move* and *Banion*. Our game needs a consistent number of *Banion*s 1.1 to be playable. It would better if they could be the same every time the application starts: same names, same sprites, same moves... Independently from the user progresses it would be nice for a user to choose between the same set of Banions at the beginning of every new the game.

Solution The problem is that somehow these entities has to be created, so how I decided to do that is the solution. At the beginning I started from the generation of *Element*s 1.4 by coding each one in Java classes. The two main problems I observed were the expansion of classes and the inability to easily and quickly add new ones. After that I took time to think at a more extendible and cleaner solution, at the end I decided to go for the deserialization. I decided to create in the MVC *Controller* a system to deserialize entities from file. In specific I chose *.json*, using Google Gson library.

I also wanted to make sure to avoid duplication of information. We decided that every single entity must be identifiable by name. So I created three files, one for each entity to generate, following this structure:

- *Element* →independent from the other entities;
- *Move* →dependent from *Element* entities (each *Move* is associated with one *Element*);
- *Banion* →dependent from *Element* and *Move* entities (each *Banion* is associated with one *Element* and 4 *Move*s);

I also decoupled file reading from entities generation. I put in the *darwinsquest.config* package classes to read from file while in *darwinsquest.config* I put the Factories according to *SRP* (single responsibility principle).

Pattern The entities are created by using *Abstract Factory* pattern. *ElementFactory*, *MoveFactory* and *BanionFactory* retrieves the deserialized data with the shape of a set.

Comments Following this scheme in future it could be easily add the functionality of writing a system to save user progress. This would consist essentially on storing user *Banion*s data in a folder (ex. *.darwinsquest* in the user folder) and reload them after the login operation. For what concerns *User*

and *Board* progresses serialization it should be thought in organization, and it would be integrated with the serialization/deserialization of entities.

Board Opponents

Problem How to represent *BattleTile*s in the *Board*, and when to create *Opponents* for each level.

Solution The *BattleBoard* is a specific board that contains the concepts of battle and levels. Each movement in the *Board* brings the player over a new *BattleTile*. The *BattleBoard* was specifically created by the selected *Difficulty* with a particular *OpponentsFactory* implementation. Once the player is to start a battle its specific opponent is created at that moment.

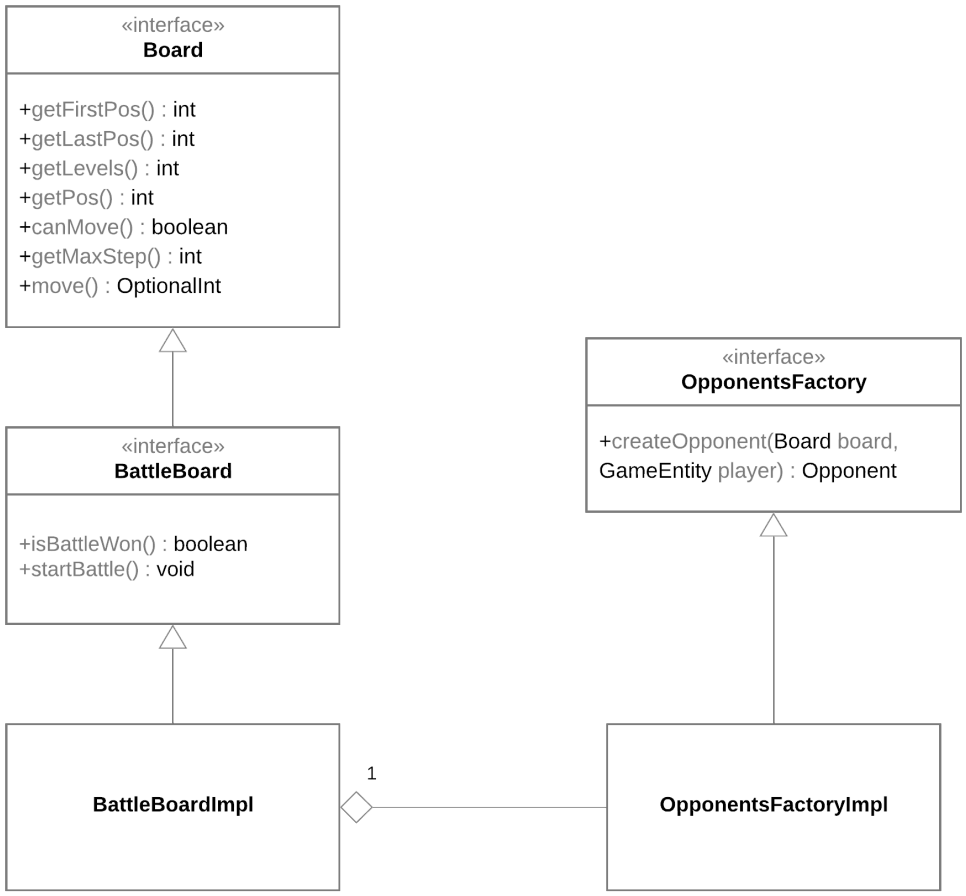


Figure 2.4: UML Board Opponents

Pattern The *OpponentsFactory* is passed via*Strategy* at the specific *BattleBoard*.

*Sprite*s

Problem Each *Banion* should be represented by a *Sprite* in the *View*.

Solution I considered that each *Banion* is uniquely identified by its name. So I decided to put the necessary data in the same *.json* file that contained also the *Banion*s information. By doing that I created a *BanionsSpriteFactory* class in the *View* that has to read *Sprite* record (it contains *Banion* .png information). Each banion will be associated in the *View* to its *Sprite* by its name (this can be seen in *ChooseBanionMenuView* constructor).

View

Problem How to separate concerns in the chosen *JavaFX* library in MVC. How to decouple style from showing controls, for example how to specify each *Scene* font in a *DRY* (don't repeat yourself) way.

Solution I started by using *.css* files to specify style and *.fxml* to specify layouts for each Container. I created a Controller for each Container. I created a *View* interface that is a generic interface with the aim to create and show different views. In this way the *View* interface remains generic and could potentially change without modifying its interfaces.

Francesco Cipollone

Game Entity - Player & Opponent

Problem A player and opponent need to be generalised into a broader concept, such as a game entity.

Solution The interface `GameEntity` was created to represent both the player entity and the opponent entity. It extends `GameObject`, which represents the fundamental entities of the game.
To encapsulate all the common aspects of any game entity an abstract class called `AbstractGameEntity` was created. Its main purpose is to handle the inventory, leaving all the entity decisions (such as deploying a Banion, selecting a move, swapping a Banion, etc.) to the concrete entities.
The `Player` and `Opponent` can now handle those decisions with the input and the AI respectively. In `PlayerImpl` specifically, a static boolean method was added to check the validity of a username, by using a *regular expression*. A valid username is defined by these guidelines:

- A nickname must:
  - Not be `null` nor blank;
  - Start with an alphabetical character;
  - Not start with one or more digits;
  - Not start nor end with one or more symbols;
  - Not start nor end with one or more underscores;
  - Not include any white spaces;
  - Not include Unicode characters.
- A nickname may include:
  - One or more underscores between the first and last characters;
  - Digits after the first character.

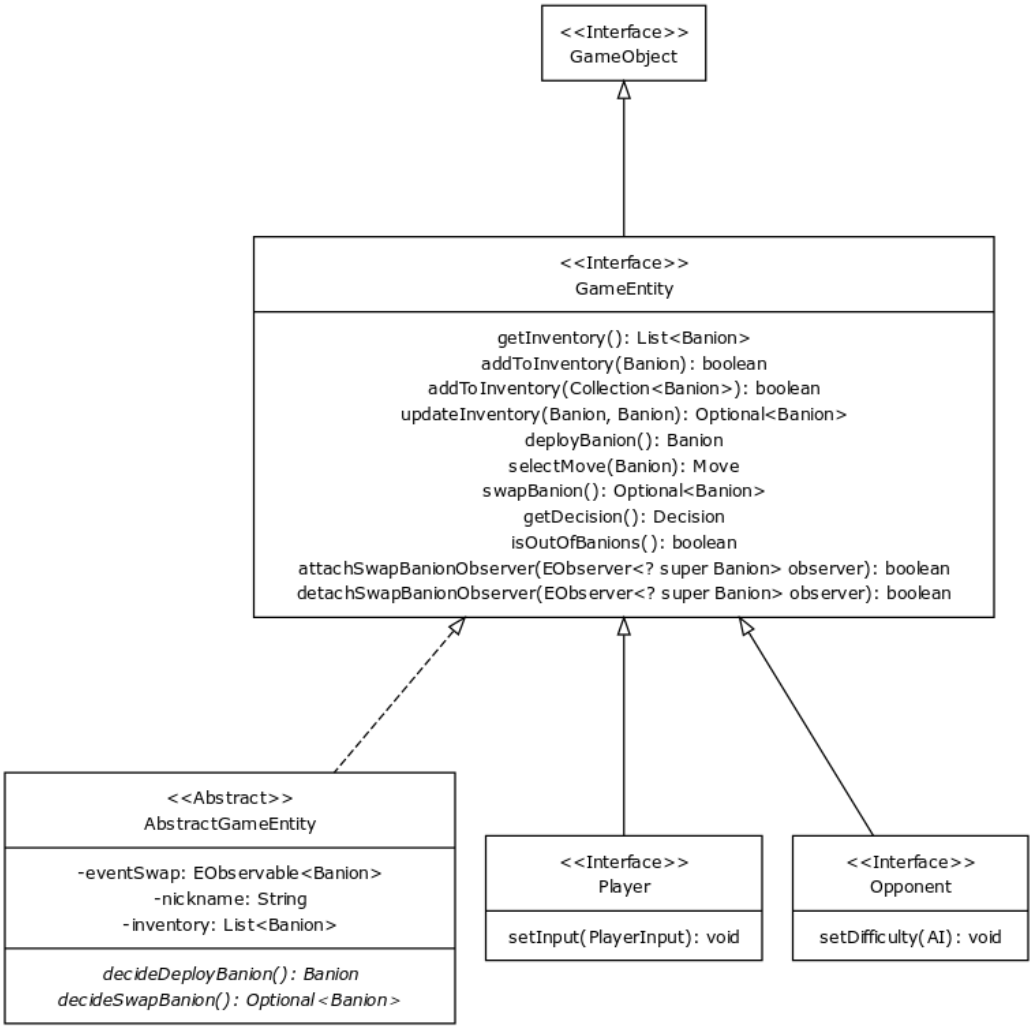
Example 2.2.1. Nickname examples:

✓ **Valid**: `Alice_hash7`, `Bob`

× **Invalid**: `!Alice`, `Amélie`

The game entities make use of an observer to notify when they decided to swap a Banion during a battle.

Figure 2.5: GameEntity UML diagram.



Inventory

Problem Every game entity needs to possess an inventory to store their Banions.

Solution We firstly conceived the inventory to be an object that could hold both Banions and other items, however since the items were scrapped for the lack of time, the concept of the inventory regressed to a simple list held in

each game entity instance.
The inventory permits add and update operations. The latter removes a given Banion in favour of another one.
A crucial characteristic of the inventory is that it does not allow the insertion of a Banion that is already present. Specifically, it checks equality on the Banion's UUID: Banions that are *alike* are allowed, *identical* ones are not.

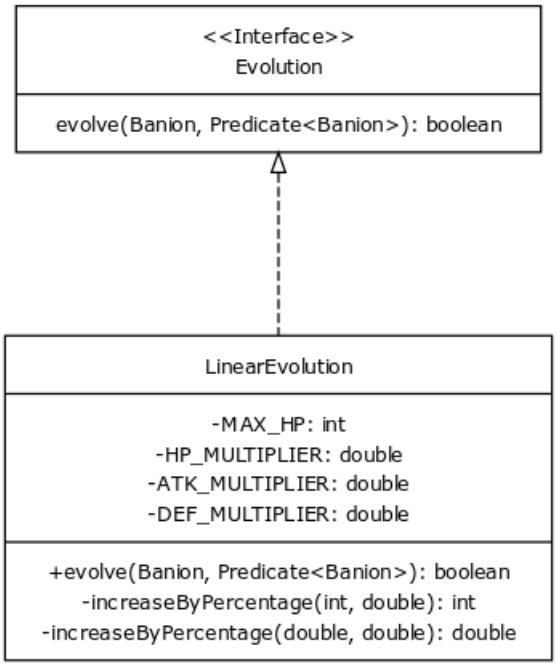
Evolution

Problem Each Banion must be able to evolve by gaining experience points and boosting its statistics.

Solution An `Evolvable` interface is introduced: this interface contains the methods signatures that allows Banions to evolve.
To prompt an evolution, the Banion class has to use an `Evolution`. The `Evolution` interface, which can also be implemented functionally, allows the given Banion to evolve, while basing said evolution on a given condition.
While the use of a `Predicate<Banion>` does not affect the Player's Banions, which are evolved by an experience points system[1], it allows a diverse and customizable evolution for the Opponent's Banions by permitting different evolution approaches.
The `Evolution` interface allows the creation of different kind of evolutions, such as tree evolutions, linear evolutions, etc.
This version uses a `LinearEvolution`, meaning that the player cannot choose what and how to evolve their Banions: each statistics will be jointly evolved.
The aforementioned approach additionally incorporates a maximum hp ceiling, which means that if the current hp statistic reaches the maximum allowable value, further evolutions will not increase said statistic.

[1] An evolution is prompted when the maximum XP value is reached. The current XP will be reset to zero, or to any exceeding amount left from reaching said ceiling.

Figure 2.6: Statistic UML diagram.



Statistic

Problem The Banion's statistics need to be easily manipulated and have to accommodate different numerical data types to allow reusability.

Solution To encapsulate the representation of a statistic with a numerical primitive type (such as `int`, `double` and `float`) a generic `Statistic` interface was used.
This interface generic type `T` extends the abstract class `Number`: this allows the use of primitive types to back a statistic object.
A statistic is defined by a:

- **Type**: Attack, Defence, Hp;
- **Value**: the actual primitive value.

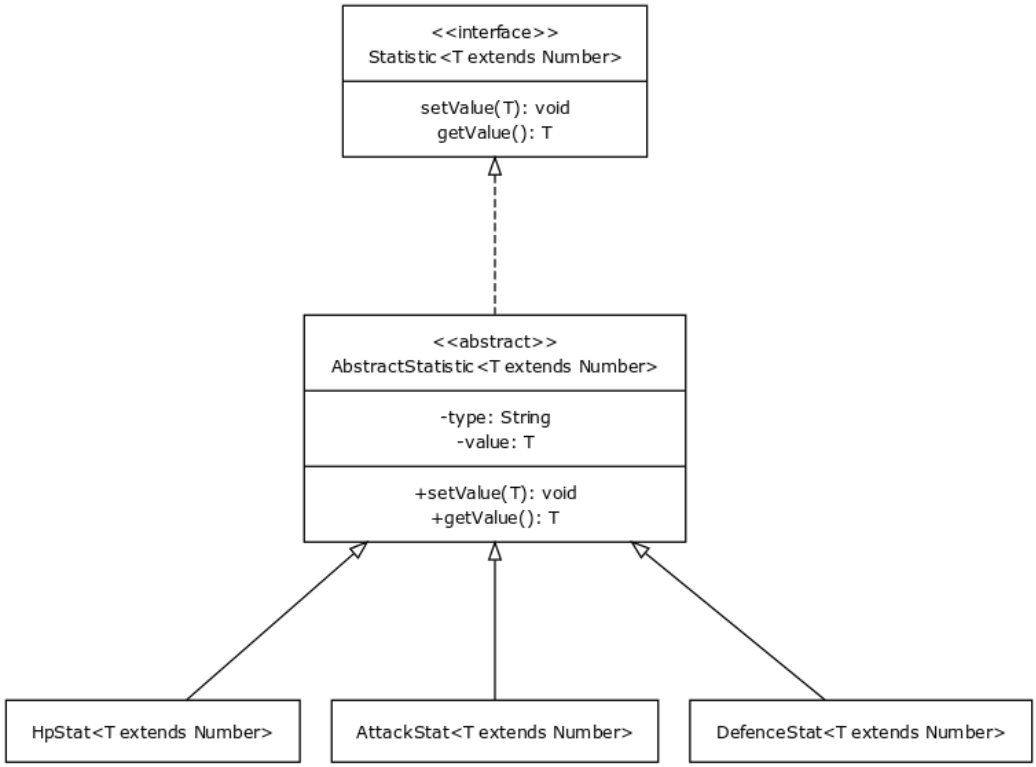
To unify these common characteristics an abstract class was used. The `AbstractStatistic` constructor takes a `String` to identify its type, and a `T` for its value.
Keeping a `type` field is necessary to distinguish different statistics that may

have the same numerical value, but differ in category. It was also deemed acceptable to use a `String` for storing each statistic type since an abstract class cannot be instantiated and has to be extended by a concrete one, therefore keeping the `String` creation inside the concrete constructor, and exposing only a `T value` as its argument.

Listing 2.1: Example of a concrete stat constructor.

```
1 public ConcreteStat(final T value) {
2     super("type", value);
3 }
```

Figure 2.7: Statistic UML diagram.



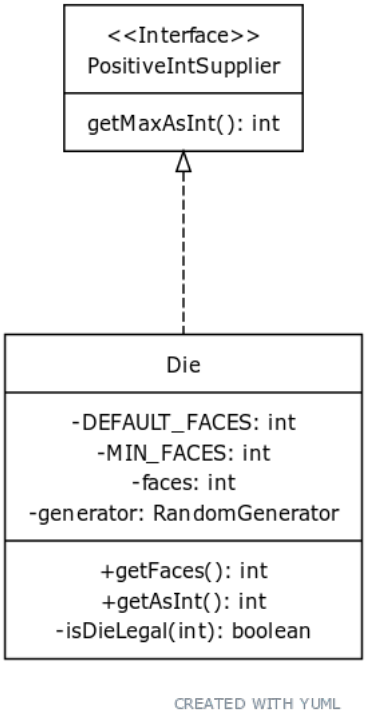
Die

Problem The player movement in the game board is determined by the roll of a die.

Solution The `Die` class extends a specialization of an `IntSupplier`, called `PositiveIntSupplier`. This `Die` represents a Polyhedral die, meaning a

three-dimensional die with polygonal faces, straight edges and sharp vertices.
Polyhedral dice include the *classic* cubical die, as well as non-cubical ones, such as tetrahedra, octahedra, dodecahedra and so on and so forth.
There are many other rare variations of polyhedral dice, with various number of faces, but for the sake of simplicity a `Die` instance was considered legal when its number of faces is even, and it is greater or equal to the minimum faces number.

Figure 2.8: Die UML diagram.



SpriteAnimation

Problem Each Banion has an associated sprite sheet that has to be properly shown to the end user as an animated sprite.

Solution The class `SpriteAnimation` was created to animate sprite sheets [Fro23]. `SpriteAnimation` [Hei12] extends JavaFX's `Transition` abstract class and was adapted for our intended use and needs.

Figure 2.9: The idle sprite sheet for **Infernhog**.

This class *slices* the sprite sheet into single sprites (given the dimensions of a single sprite) and sets the provided `ImageView`'s viewport to said sprite. The aforementioned process is repeated for each individual sprite, creating the illusion of a single moving image for the player.
A newer addition to the original `SpriteAnimation` is the possibility to mirror the sprite horizontally. This enables the usage of the same sprite sheet for both the player and opponent's Banions, since they face each other in the battle GUI.
Another new feature is the usage of an upscale multiplier for the sprite sheet image. To prevent the sprite to look too small and grainy a new `Image` object with the requested dimensions (multiplied by the upscale factor), and the `smooth` argument set to `false` is created. This approach allows the sprites to be displayed crisply on screen.

GameSoundSystem

Problem The game needs a simple way to play background music and sounds effects.

Solution `GameSoundSystem` is a helper class to play BGM[2] and SFX[3].
This audio playback helper class was conceived with these key components in mind:

- **Modularity** and **Encapsulation**: promotes modular design, code organization, easy maintenance and updating;
- **Reusability**: allows reusage across multiple classes without duplicating code;
- **Abstraction**: grants complex operations' encapsulation, reducing application complexity and ensuring efficient usage;
- **Extensibility**: enables the addition of new features, effects, and libraries without affecting other application components.

[2]**BGM**: acronym for background music.
[3]**SFX**: acronym for sound effect.

JavaFX's *Media* module was used as the audio playback library. Different `MediaPlayer`s were used to play BGM and SFX. This distinction allows the playback of both BGM and SFX simultaneously.
Sounds will be loaded from their location into a `Map` that links a `String` containing the file name to a `Pair` that contains the `Media` object itself and an `Optional<Duration>` that allows to trim the audio clip, if desired.
The next paragraph will discuss the problems that were encountered while loading the audio resources from a JAR file.

Loading audio resources from JAR. Loading audio resources at runtime from the jar file revealed itself to be a delicate matter: after an extensive research it was ascertained that loading sounds at runtime while running the application from an IDE/build tool and from the jar file required two different approaches. The latter required the use of the `JarFile` class specified in the package `java.util.jar`.
To address this issue a solution [use12] was found and incorporated after being properly adapted and refactored to follow our needs.

Soundtracks by SketchyLogic [Ske14] were used for BGM, audio clips by OmegaPixelArt [Ome19] and ColorAlpha [Col22] were used for SFX.

Raffaele Marrazzo

Battle realization

Figure 2.10: UML representation of the turns' architecture

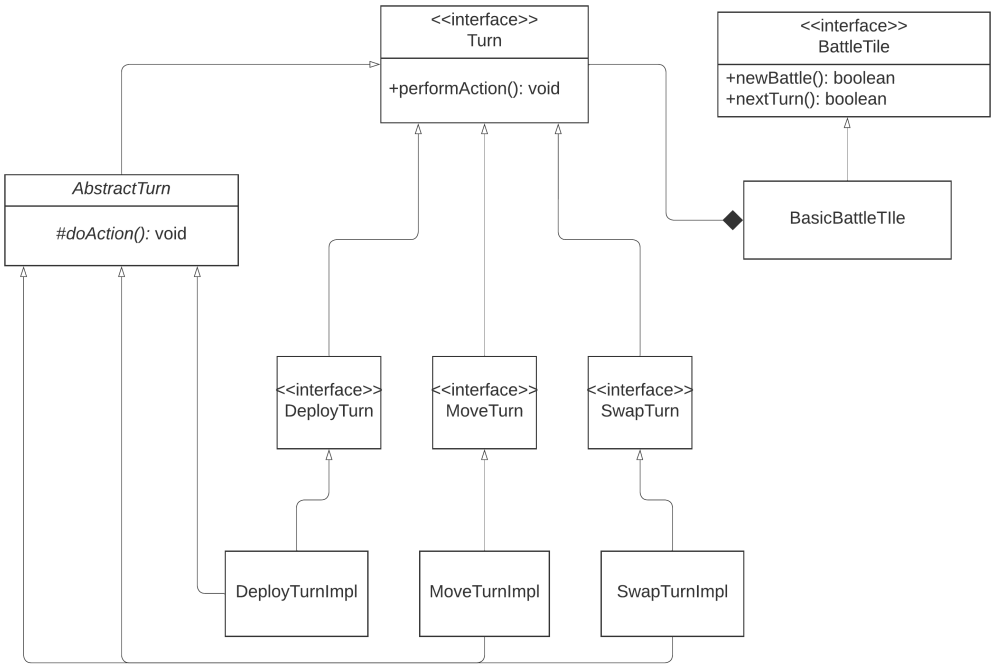
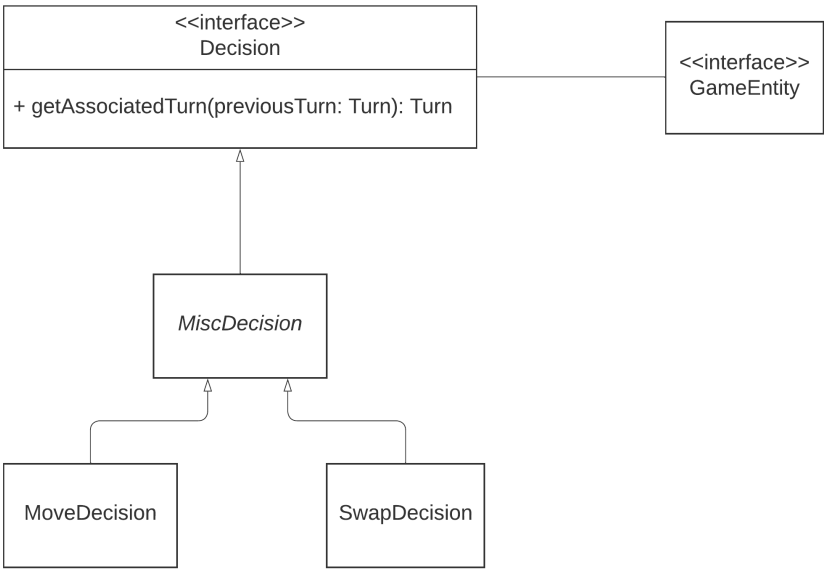


Figure 2.11: Decision system



Problem Developing the battle to perform "step by step", rather than in "one shot", and make it extendible.

Solution At the beginning I conceived the battle as an entity that had to be performed in "one shot", but I had a lot of problems testing it because it was difficult to tell what happened during the battle's development. So I adopted a different approach, described below.

I conceived the battle as a sequence of turns. Each turn allows to perform one action only. The battle has only references to the interface `Turn`, in order to make it extendible: if in the future someone wants to introduce a new kind of action that can be performed in a turn, the battle won't need to be changed. The turn logic also helps to test the battle, because in this way it is possible to test what happens in each turn, and not only at the end of the battle.

The decision system also helps the battle to be extendible: at a certain point in the battle, the `GameEntity` on turn is asked to make their decision. At this moment, thanks to the method `getAssociatedTurn()` of the interface `Decision`, the turn associated with that specific choice is created, and so the battle does not have to worry about the instantiation of a specific `Turn`, or about the presence of a new decision.

Problem During the development of the turns, I noticed that all the `Turn` implementations had a lot of things in common, leading to code duplication.

Solution I developed the abstract class `AbstractTurn`, which factorizes all the things that the Turn implementations have in common. In particular, in the above-mentioned class, I used the pattern *Template method*, in order to avoid code duplication. The template method is `performTurn()`, inherited by the interface Turn, which calls the protected and abstract method `doAction()`, which is implemented in different ways in all the Turn's subclasses, which extend `AbstractTurn`.

Problem Managing with the same strategy the two `GameEntity`'s turns.

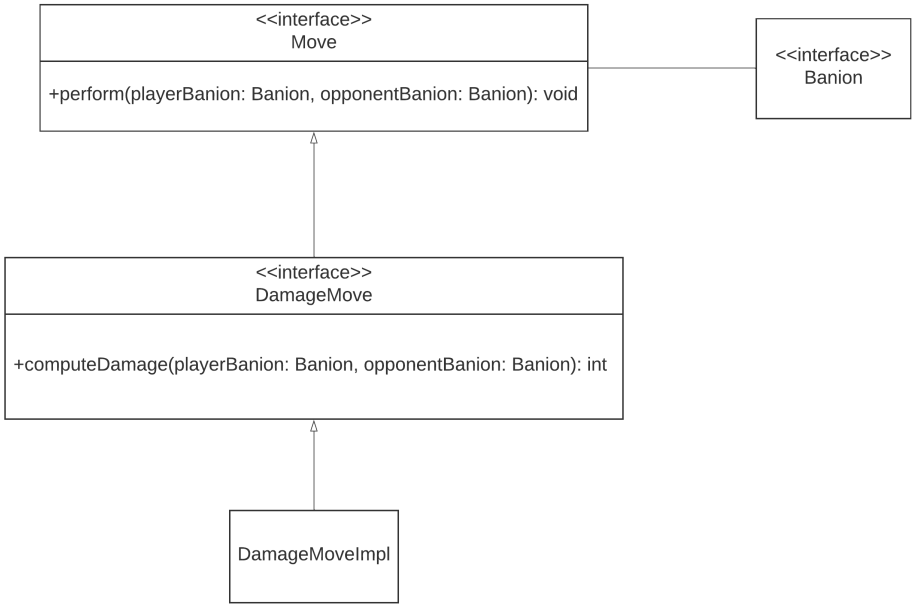
Solution At the beginning, the two entities that fight in the battle were managed with the same strategy. But, when the user input had been introduced, we had problems with the management of the `Player`'s turn, because the battle forced them to swap their currently deployed Banion if it was dead. On the other hand, for the `Opponent`, it was necessary to force the swap in case of death, because its AI is managed randomly. So, even if it is not the best solution, I decided to manage the turns of the two subclasses of `GameEntity` in the battle with different strategies, by not forcing the player to swap its currently deployed Banion directly from the battle.

Problem Each `Turn` needs to be different and identifiable from each other turn.

Solution After different evaluations, for instance identifying a turn by using the `GameEntity` on turn and the `GameEntity` not on turn (this solution was considered not adequate because in this way a battle could have had some equal turns), I decided to identify each `Turn` through a unique identifier (*java.util.UUID*).

Moves

Figure 2.12: Moves' architecture



Problem Manage the existence of different kind of moves.

Solution I decided to create the interface `Move`, which represents all the possible kinds of battle moves. In this way, a `Banion` can have references only to `Move` and so being capable of managing different types of moves (e.g. moves that can change the stats of a Banion, that, at the beginning, we wanted to develop but, for the sake of time we decided to put aside this idea).

Chapter 3

Deployment

3.1 Automatized testing

We used *JUnit* to test our *Model* classes.

Application components that underwent unit testing:

- Battle, Decision and Turn;
- AI;
- Difficulty;
- Evolution;
- Banion;
- Player & Opponent;
- Moves;
- Die;
- Board;
- Engine;
- Banion deserialization.

The aforementioned components were regarded as key application's features, hence the decision of unit testing them.

3.2 Work strategy

During development, we adopted the git-flow approach by creating a `develop` branch and a new branch for each new `feature`.

We agreed upon the Model API interfaces, and we subsequently proceeded as follows:

Enrico Marchionni

- Engine and Board;
- Game difficulties;
- Entities generation;
- JavaFX *View* controllers of start menu, login, difficulties selector, board, choose banions selector, and JavaFXView class;
- Main controller class.

Francesco Cipollone

- GameEntity - Player & Opponent;
- EntityController;
- Inventory;
- Evolution;
- Statistic;
- Die;
- SpriteAnimation;
- GameSoundSystem.

Raffaele Marrazzo

- Opponent AI;
- Battle logic;
- Moves;

- Battle controller;
- Move controller;
- FXML scenes of victory and game over, and relative view controllers.

Game components developed in conjunction

- `BattleView` and relative `fxml`;
  - Enrico Marchionni: sprites rendering;
  - Francesco Cipollone: initialising a random background and music. Added SFX. Created the FXML;
  - Raffaele Marrazzo: interaction with `BattleController`, switch to Victory screen and Game Over screen.
- `ChooseBanionMenuView`;
  - Enrico Marchionni: Event handlers, constructor;
  - Raffaele Marrazzo: FXML.
- `BanionImpl`;
  - Enrico Marchionni: the Banion implementation;
  - Francesco Cipollone: included the evolution.

3.3 Development notes

Each one of us used these advanced aspects of the Java language:

Enrico Marchionni

Bounded type parameters

Used in a few occasions.

Example at `https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/util/EObserver.java#L15`.

Generic class

Used frequently.

Example at https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/config/CustomDeserializer.java#L47.

Annotation *darwinsquest.annotation.Description*

I used this custom annotations in 2 different scopes.

Example 1 at https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/core/EngineImpl.java#L38.

Example 2 at https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/view/JavaFXView.java#L56.

Reflection

Used in different situations.

Example at https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/core/EngineImpl.java#L65.

Lambda expressions

Used very frequently.

Example at https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/core/gameobject/banion/BanionImpl.java#L57.

Optional

Used very frequently.

Example at https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/core/Engine.java#L31.

Stream

Used very frequently.

Example at https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/view/ChooseBanionMenuView.java#L69.

Immutable ordered set

Used only at https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/core/EngineImpl.java#L46.

Library JavaFX

Used for *View* management.

Example at *fxml* with *.css*, example at https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/view/JavaFXView.java#L53.

Library Apache Commons Lang 3

Used frequently.

Example at https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/view/graphics/BanionsSpriteFactory.java#L27.

Library Google Gson

Used for deserialization.

Example at https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/util/JsonUtils.java#L81.

Library Java Faker

Used only at https://github.com/DarwinsQuest/DarwinsQuest/blob/3ec97da07e9622ab8ec0333a774619be3332e88e/src/main/java/darwinsquest/core/difficulty/OpponentsFactoryImpl.java#L114.

Francesco Cipollone

Use of `regex`

Used in `PlayerImpl`.
Example at: https://github.com/DarwinsQuest/DarwinsQuest/blob/cb

0341532cf8fc87d07e9c13e2ae7ea104b54500/src/main/java/darwinsquest/core/gameobject/entity/PlayerImpl.java#L103

Use of `Optional`

Used in different occasions.
Example at: https://github.com/DarwinsQuest/DarwinsQuest/blob/64f84140f503f2e40d97e1dc69d0824c86c322e6/src/main/java/darwinsquest/core/gameobject/entity/AbstractGameEntity.java#L69-L75

Generics

Used in the `Statistic` interface and its subclasses.
Example in `AbstractStatistic`: https://github.com/DarwinsQuest/DarwinsQuest/blob/64f84140f503f2e40d97e1dc69d0824c86c322e6/src/main/java/darwinsquest/core/statistic/AbstractStatistic.java#L9

`Stream` and lambda expressions

Used in various sections.
Example at: https://github.com/DarwinsQuest/DarwinsQuest/blob/64f84140f503f2e40d97e1dc69d0824c86c322e6/src/main/java/darwinsquest/core/gameobject/entity/AbstractGameEntity.java#L61

Creation of a `GameSoundSystem` with the JavaFX media module

Used in `GameSoundSystem`.
Example at: https://github.com/DarwinsQuest/DarwinsQuest/blob/64f84140f503f2e40d97e1dc69d0824c86c322e6/src/main/java/darwinsquest/view/sound/GameSoundSystem.java#L30

Use of `MultiValuedMap` from Apache Commons Collections 4

Used in `Evolvable` interface, Look at 3.2 for implementation details.
Example at: https://github.com/DarwinsQuest/DarwinsQuest/blob/64f84140f503f2e40d97e1dc69d0824c86c322e6/src/main/java/darwinsquest/core/evolution/Evolvable.java#L58

Raffaele Marrazzo

Use of the library Apache Commons lang3

In the turn logic, I used the tuples of Apache Commons lang3. Example at `https://github.com/DarwinsQuest/DarwinsQuest/blob/1dc14f924d95071e1f3222124a6f996737a0d1f6/src/main/java/darwinsquest/core/battle/turn/AbstractTurn.java#L17-L18`.

Use of lambda expressions

Used a few times. Example at `https://github.com/DarwinsQuest/DarwinsQuest/blob/0c620b69b0bfd0acc345c5dcdb62f7a0c2a3aeda/src/main/java/darwinsquest/core/battle/BasicBattleTile.java#L118`.

Use of `Optional`

Used a lot of times. Example at `https://github.com/DarwinsQuest/DarwinsQuest/blob/0c620b69b0bfd0acc345c5dcdb62f7a0c2a3aeda/src/main/java/darwinsquest/core/battle/turn/AbstractTurn.java#L152`.

Use of `Stream`

Used a lot of times. Example at `https://github.com/DarwinsQuest/DarwinsQuest/blob/0c620b69b0bfd0acc345c5dcdb62f7a0c2a3aeda/src/main/java/darwinsquest/core/battle/BasicBattleTile.java#L213-L214`.

Use of the library `JavaFX`

Used for the FXML controller of the victory scene and game over scene. Example at `https://github.com/DarwinsQuest/DarwinsQuest/blob/0c620b69b0bfd0acc345c5dcdb62f7a0c2a3aeda/src/main/java/darwinsquest/view/VictoryView.java#L51`.

Chapter 4

Final comments

Tiles Tile concept was never used in our battle *Board* that consists in a series of battles that can be skipped by die jump. It would be better to introduce different *Tile*s and to be forced to complete *BattleTile*s, while the others can be skipped.

AI The Opponent's AI makes decisions randomly. A more intelligent AI was not developed for lack of time.

Memory leak and profiling During the late state of the project, a memory leak was discovered. This bug created serious performance issues, that caused the game to completely freeze during the battles.
We, Francesco Cipollone and Raffaele Marrazzo, tried to perform a hot fix by using *VisualVM* for profiling. The root of the problem was a faulty observers' management: each `Observable` has a `Collection` of observers that is always getting populated by new observers, but never freed of them, even when they are not needed anymore. This caused the `Collection` to grow uncontrollably, leading to massive CPU and memory usage.
For time concerns and the complexity (that was not debated during the course's lectures) of the problem, we were not able to fix it.
To achieve acceptable performances during the game execution we decided to not alter the original architecture, but to decrease the possibility of a serious memory leak to happen by cutting the overall Banion's health points, making the battle shorter.

Figure 4.1: CPU and Memory usage snapshot

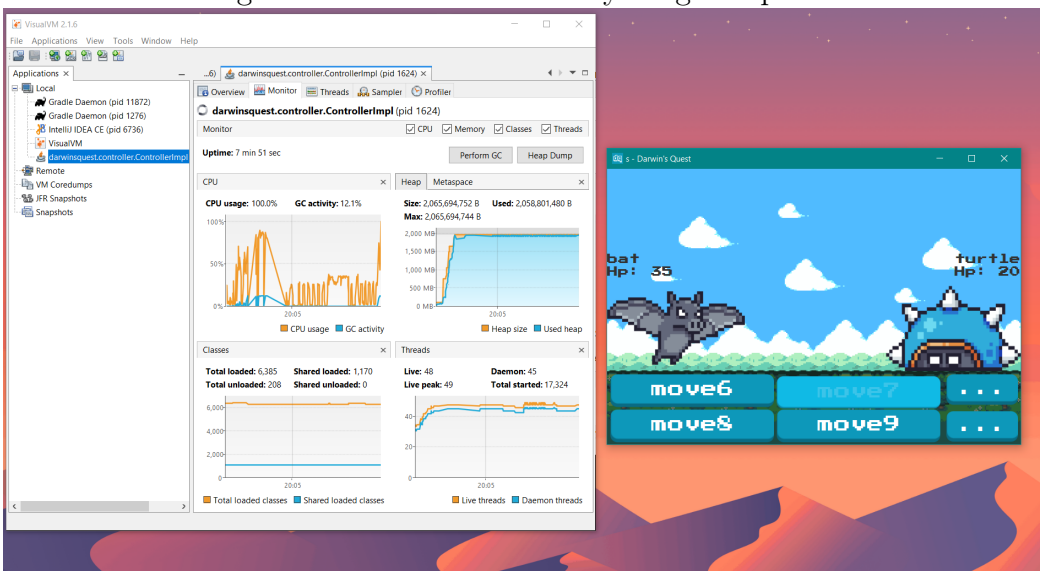
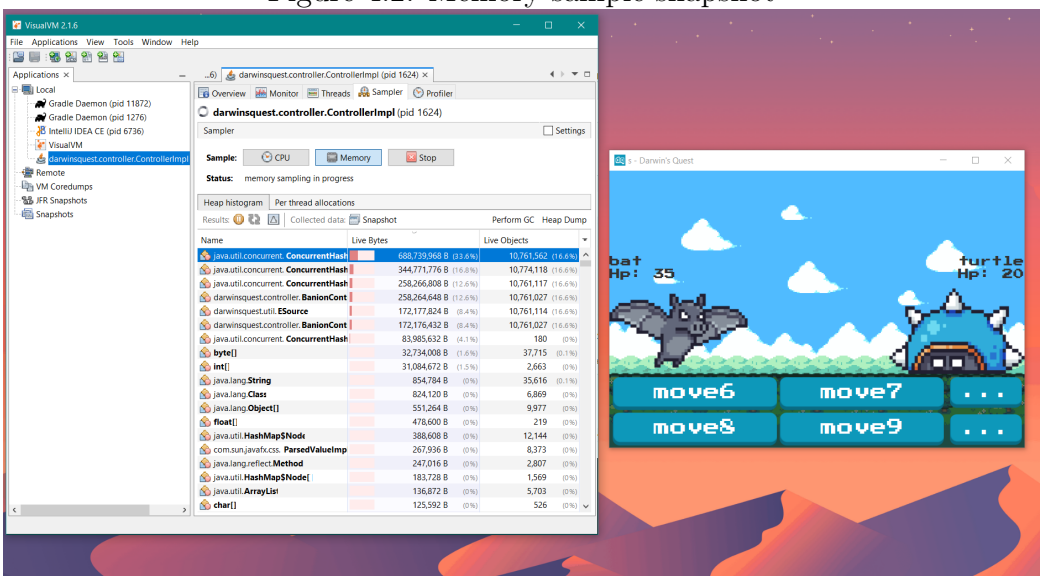


Figure 4.2: Memory sample snapshot



4.1 Self-evaluation and future improvements

Francesco Cipollone

I am quite pleased on how this project turned out even if we did not manage to reach the level of complexity in game mechanics that we agreed upon.
This was my first *real* experience working with a small team of developers, and it strengthened my belief in how much important collaboration is.
I have acquired a stronger sense of team effort, dedication and team management. I believe that the aforementioned skills are vital for teamwork and made me grow professionally.

Raffaele Marrazzo

I'm really happy with the final result, especially taken into consideration the multiple difficulties that we ran into in the last week. I'm proud for developing an essential component of the model: the battle, which took a lot of time but at least I managed to make it extendible. This project gave me a great opportunity: be part of a software development team, even if it was small, and it made me realise that this is what I want to undertake in my life.

4.2 Difficulties and comments to teachers

Francesco Cipollone

In contrast, I am thoroughly dissatisfied with how the interaction between certain team members in various circumstances was turbulent and unprofessional, frequently resulting in unproductive quarrels that only steered the project roadmap to the wrong direction, eventually resulting in low team morale. Poor communication frequently lead to commit wrong choices, and to partially inadequate priority judgement.
There is a lot to ponder, and this experience will undoubtedly help us improve.

Raffaele Marrazzo

I realised how fundamental time management is while working on this project: as beginners it was hard to evaluate in advance how much time we were going to invest on developing this game.

To avoid getting ahead of ourselves, I feel that further instruction was required before confirming the project request.

Appendix A

User guide

A.1 Title Screen

After running the game the user will be presented with a title screen with two buttons: “Start” and “Quit”. To continue, the player should press the “Start” button.

A.2 Username Selection

After hitting “Start”, the player will be asked to enter a username. Enter and confirm the appropriate username in the supplied input field.

A.3 Banion Selection

After confirming their username, the player will be taken to a menu where they can select four Banions for their team. Select four Banions in the menu and then hit “Confirm”.

A.4 Game Difficulty Selection

The player will be asked to select a game difficulty: “Normal” or “Hard”. Select the desired level of difficulty by clicking on the corresponding option.

A.5 Game board and Battles

After selecting the game difficulty, the player will be taken to the game board screen. The game board represents the amount of battles to overcome before reaching the game's final boss. Click the "Fight" button to begin a battle. Once a battle is won, press the "Move" button to progress further on the game board.

A.6 Battle Mechanics

During battles, the player will have several options available:

- **Move Buttons**: These four buttons represent the available moves for the Banions in the player's inventory. Click on the desired move to execute it during the battle;
- **Swap Button**: This button allows the player to swap the active Banion with another one from their inventory. To swap the current Banion with the next one that is alive, click the Swap button;
- **End Button**: This button prematurely ends the battle, resulting in a loss.

A.7 Additional Feature

In every screen, except in battles, the player can access the "Options" menu by pressing the `ESCAPE` key. The "Options" menu allows the player to adjust the master volume. Click "Close" or press any button to return to the previous screen.
Dive into *Darwin's Quest* and have a great time!

Appendix B

Laboratory

B.1 enrico.marchionni@studio.unibo.it

- Lab 04: https://virtuale.unibo.it/mod/forum/discuss.php?d=113869#p169173
- Lab 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p169723
- Lab 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=115548#p171159
- Lab 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p173058
- Lab 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=117852#p174127
- Lab 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p175326
- Lab 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=119938#p176522
- Lab 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=121130#p177368
- Lab 12: https://virtuale.unibo.it/mod/forum/discuss.php?d=121885#p178425

B.2 francesco.cipollone@studio.unibo.it

- Lab 03: https://virtuale.unibo.it/mod/forum/discuss.php?d=112846#p167954
- Lab 04: https://virtuale.unibo.it/mod/forum/discuss.php?d=113869#p169123
- Lab 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p169538
- Lab 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=115548#p171206
- Lab 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p172980
- Lab 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=117852#p174113
- Lab 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p174912
- Lab 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=119938#p176211
- Lab 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=121130#p177361
- Lab 12: https://virtuale.unibo.it/mod/forum/discuss.php?d=121885#p178396

Bibliography

[Dar59] Charles Robert Darwin. *On the Origin of Species*. John Murray, 1859.

[Hei12] Michael Heinrichs. *Creating a Sprite Animation with JavaFX*. Blog post. 2012. URL: https://netopyr.com/2012/03/09/creating-a-sprite-animation-with-javafx/.

[use12] user1079877. *How can I access a folder inside of a resource folder from inside my jar File?* StackOverflow. 2012. URL: https://stackoverflow.com/a/20073154/17169764.

[Ske14] SketchyLogic. *NES Shooter Music 5 tracks, 3 jingles*. OpenGameArt. 2014. URL: https://opengameart.org/content/nes-shooter-music-5-tracks-3-jingles.

[Ome19] OmegaPixelArt. *GAMEBOY SFX PACK 1*. Itch.io. 2019. URL: https://omegaosg.itch.io/gameboy-sfx-pack.

[Col22] ColorAlpha. *50 Menu Interface SFX*. Itch.io. 2022. URL: https://coloralpha.itch.io/50-menu-interface-sfx.

[Fro23] Pixel Frog. *Pixel Adventure 2*. Itch.io. 2023. URL: https://pixelfrog-assets.itch.io/pixel-adventure-2.