University of York

Department of Computer Science

SEPR - Assessment 2

Architecture

Team Craig

Thomas Burroughs

Huw Christianson

Joseph Frankish

Isaac Lowe

Beatrix Vincze

Suleman Zaki

Concrete Architecture

For our game we decided to use the LibGDX Engine which provides a useful framework and suitable methods for implementing our game. As stated in our Assessment document the team is required to code in Java, an object-oriented programming language, which all team members were familiar with. The LibGDX framework supports the use of Java and so proved an obvious choice. Our abstract architecture diagram from Assessment 1 was based on this framework, but as will be discussed in this document, large parts of the LibGDX framework was misunderstood and represented wrong due to improper research and poor documentation.

UML Class Diagrams

We chose to represent our architecture with UML class diagrams, as it is intuitive, and a widely accepted industry standard for object-oriented programming. The diagram only represents the current version of the game. Functions that are to be implemented for the full product brief, for example the mini game, have been excluded from the diagram to avoid confusion. Unit tests are also not included as they are not important nor relevant for the architecture.

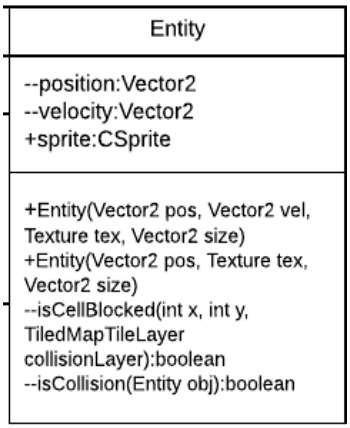
Full UML Diagram: https://teamcraigzombie.github.io/assets/downloads/FullUMLDiagram.pdf

We found that our full concrete architecture diagrams were difficult to read and comprehend because of the complexity of the software, and so we also created a simplified UML diagram. This simplified version does not contain any class attributes or class behaviour and so has less document conventions and less complex notations. This makes class relations and conceptual ideas easier to observe especially for non-team members or for team members who are less technically skilled.

Simplified UML Diagram: https://teamcraigzombie.github.io/assets/downloads/SimplifiedUMLDiagram.pdf

Diagram conventions and notations

Below we have stated our notation conventions for all of our UML diagrams. This helps to provide consistency, which in turn, should improve readability and maintainability for all our UML diagrams. This should also help to achieve requirement **M1** as seen in our updated requirement document.



The heading of the box represents the name of the class. Between the class name heading and the next line are the attributes of the class with their appropriate access modifiers, followed by all the class methods below the next line.

The access modifiers for both the attributes and methods are noted as follows: + for public, - for private, --for protected.

To show the clear distinction between different relationships between classes we have used two coloured arrows. Inheritance between two classes is represented as a black arrow and aggregation is represented by a blue arrow.

Diagram Tools

For the creation of our UML diagrams we used a free online tool called LucidChart. A few members of the group were familiar with this tool from previous projects so was an obvious choice. The tool boasts many useful features, such as a large shape and operator library and the ability to export the diagrams as numerous file types. For example, the tool allowed us to download the file as an image file which was useful when formatting documentation. A key priority when finding a suitable UML diagram tool was to find a tool with a good balance between usability and complexity. We needed a tool which all team members could use but provided enough detail in order to specifically describe our architecture. Lucid chart provided us with this.

Justification for Architecture

From Abstract to Concrete

Our initial abstract view made several assumptions as to the nature of our implementation framework (libGDX) that eventually turned out false or had been oversimplified. In our concrete implementation, we decided that for ease of implementation, attempting to utilise libGDX outside of how it was designed to be used would be counter-productive to producing a working product. Our response to this decision, was to read through documentation, and decided that the best option available to us would be to split up game objects in to several groups of interrelated and functionally-grouped sets of related classes. For example, the different types of screens needed to meet requirements, specifically **REQ ID: F5, F14, F15**, were grouped together as can be seen in our architecture as well as our documented code. A further example would be the grouping of object types (power-ups **REQ ID: F3**, ground items **REQ ID- F13**) that exists within the 'Game Play' layer as described in our initial abstract architecture.

Much of the initial separations between what we had initially described as 'implementation layers' were proven to be good decisions when we began implementation, most of which were implemented and remained as initially planned. However, our initial view of how interactions between our objects would occur was proven to be severely flawed and has received a great deal of revision. An example of this would be how we thought that an in-game location could be implemented as an object at an equal level to player entities. In reality, we discovered that in order to create an elegant implementation that would be easily expandable to meet further client requirements, we would have to treat gameplay, menus, and winning states as different kinds of 'screen' objects. Doing this allowed us to utilise more of the libGDX framework to implement our work, in the process cutting down development time and improving overall stability. This became on overall theme to our changes to architecture as we made many suitable changes to our architecture to maximise the use of the LibGDX framework. We also believe that these changes should help us to produce a more understandable architecture ensuring that any team who chooses our game for Assessment 3 can make a seamless transition (**REQ ID: M1**).

Below we have listed the components (classes) for our game structure. Under each heading we have listed the requirements which that class either satisfies and attempts to satisfies along with a description of the classes purpose and relationship to other classes.

CraigGame

- Requirement(s) related to this class: F14, F15

This class is responsible for opening the loading screen on the execution of the game. This class also uses a switch case to manage and switch between screens as and when player behaviours are detected. For example, on unlocking the 'Golden lock' the switch case is called switching the game to the EndScreen.

EndScreen

- Requirement(s) related to this class: F7, F15
- Related Abstract architecture class(s): Menu System

This class implements the actions if a player wins the game by unlocking the 'Golden key' the end screen will be called using a switch case. The end screen displays an 'Exit' button with a button listen which uses a switch case to set the screen to the MenuScreen.

LoadingScreen

- Related Abstract architecture class(s): Game Engine, Menu System

When player starts the game, the LoadingScreen class will call the MenuScreen which displays the main menu. The player can make their first interaction with the game at this point.

MainScreen

- Requirement(s) related to this class: F4
- Related Abstract architecture class(s): Location

The main screen is the frame on which most of the user interaction is focused. It is needed in order to maintain interactions between other objects that make up the gameplay. Its attributes hold some game assets, instances of the other objects in the game and important values needed for certain functionality such camera movement and locking the player within a certain area. Its update method is the main runtime method of the game which calls the update methods of other classes, checks for user inputs and collisions

MenuScreen

- Requirement(s) related to this class: F5, UI1
- Related Abstract architecture class(s): Menu System

MenuScreen gives the option for the player to either start the game or to exit. A character type can be chosen from a drop-down menu. All buttons are placed within a table which has been staged using actors, to which listeners have been added. A menu was not stated in the assessment 2 requirements but included to improve usability.

PauseScreen

- Requirement(s) related to this class: F14
- Related Abstract architecture class(s): Menu System

While playing the game, the escape key gives the option to the player of pausing the game and transitioning to and from the static pause screen. This PauseScreen class does not dispose of the previous screen so the game can be resumed from the point of pausing.

MaxHealth, Coffee and Rapid-Fire

- Requirement(s) related to this class: F3
- Related Abstract architecture class(s): Location

These classes override the methods from the Powerup class. Each implements different functionality inside the methods, so the different classes are needed to do so. Coffee affects player speed, MaxHealth affects player health, and Rapid-Fire affect how many bullets are fired. Rapid-Fire also implements a timer system and contains a method which indicates the timer status so that the power ups affect can end.

Player

- Requirement(s) related to this class: F1, F9, F10
- Related Abstract architecture class(s): Player

This class handles all functionality related to the player and characters. Most of its methods are used in relation to user inputs. Its attributes hold important information about the current state of the player such as the speed and health. The update method that is regularly called by MainScreen along with other methods such as rotate, and the move methods are needed to change the values of the attributes and keep them up to date.

Projectile

- Requirement(s) related to this class: F9

This class is responsible for handling the creation, motion and termination of an instance of a bullet. When the player shoots an instance of this class is created and added to a list in MainScreen. The methods of this class are needed in order to continuously update bullet positions and to check if any have collided with a wall or left the map so that they can be removed.

Entity Classes:

Entity

- Requirement(s) related to this class: F3

This abstract class' two methods check for collision and if cell is blocked. All types of entities will in some way interact with other entities and they can't be in an area that is blocked so this class implements and handles these aspects. All subclasses then call on these methods.

Key

- Requirement(s) related to this class: F7, F13

Overrides and implements the checkCollision() method from GroundItem class. This class was needed so that it could hold information about if it had been found or not and so that we could implement the functionality to change the information when a collision occurred with a player.

GroundItem

- Requirement(s) related to this class: F3

This abstract class is responsible for setting up the structure for all items on the ground that the player can interact with(power-ups/keys). Its method is used to decide if player has collided with an instance of this object. This method is overridden by subclasses which implement its functionality. All power ups and keys are of the same size and can interact with the player so rather than the classes setting up and handling these aspects independently, this class does that instead to prevent repetitive code.

Powerup

- Requirement(s) related to this class: F3

This abstract class' constructor finds and sets a random position for a power up to be in. All power ups are placed in random locations so this class implements that rather than the subclasses doing so independently as there would be no difference.