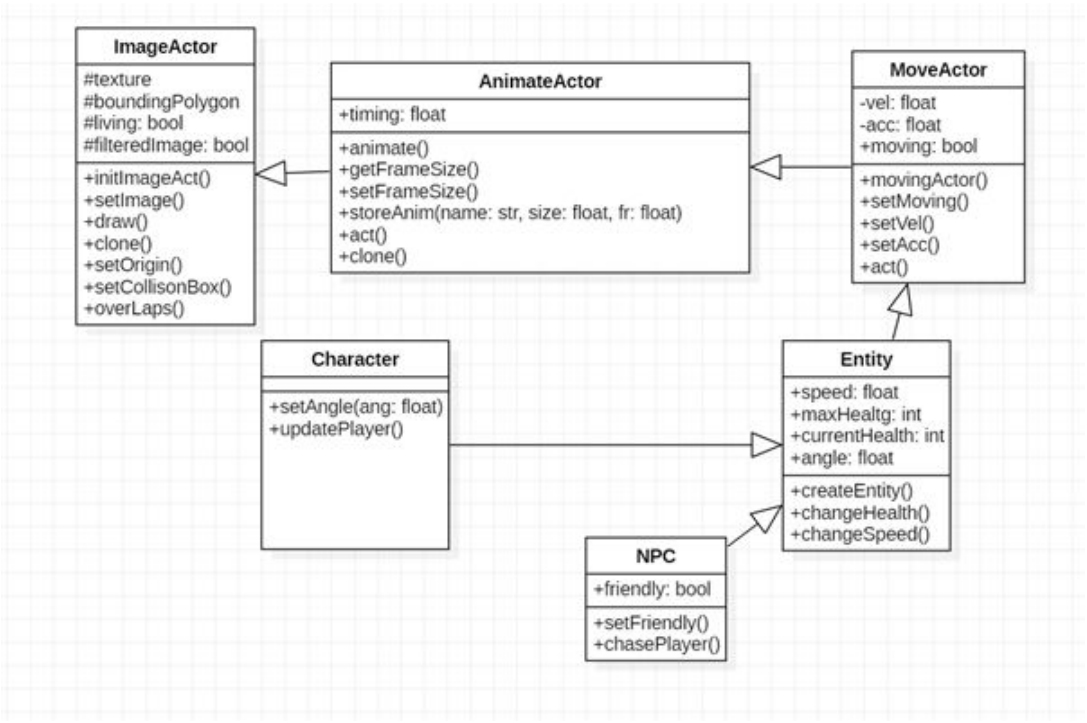
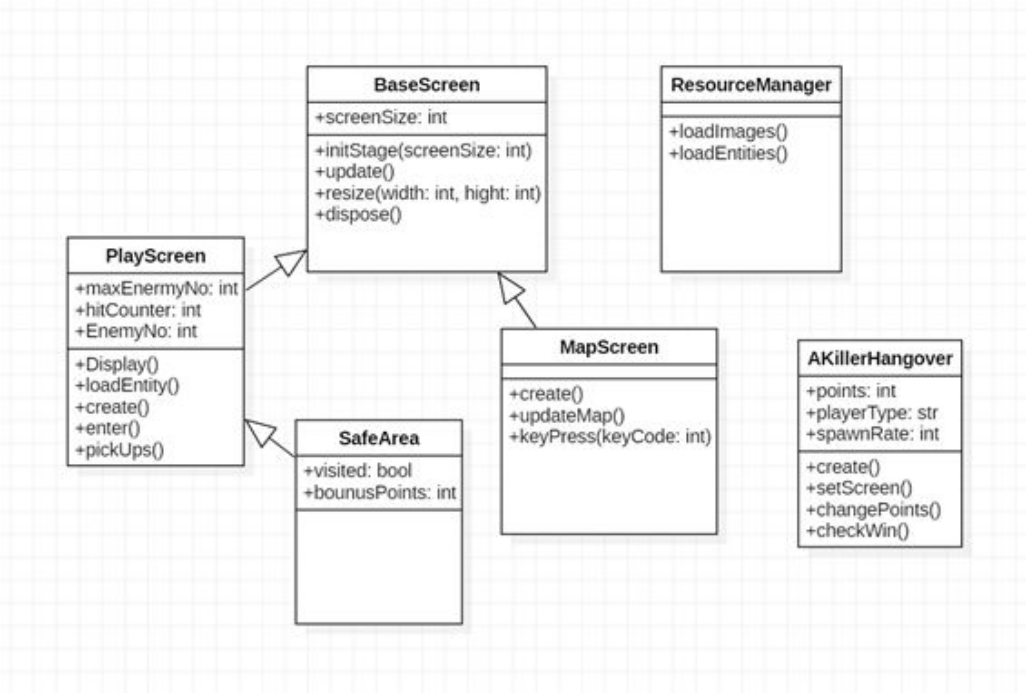
Architecture

From the collected requirements specified in the given brief, a high-level class diagram was produced. This was generated to highlight key aspects of the whole program, which is being implemented in a Java-based library LibGDX.

We constructed our architecture for the game in a UML (Unified Modelling Language) diagram, through using StarUML. This UML program allows the user to sketch out their UML diagram in a simple but effective graphical environment. What makes it effective is that it is convenient for the user to pick up the tools, simplifies editing and assembles a good representation of their project, with no programming necessary; it can be all be generated graphically.





Main changes

There have been a variety of changes from the proposed original architecture to the new one. You will immediately see a lot of classes have been removed. This is due to them going beyond the specification for this brief or we have found different ways to implement the class with benefits provided through the game engines. In addition to the changes from the first UML diagram, we have simplified the new diagram by only putting major classes in the document. Also, we have simplified some of the variables and operations, for example, the keypress method is a selection of the different buttons you can press but is also a placeholder for a handful of methods which deal with each individual input.

The first major change is with the Entities within the game i.e the player and zombie. All three now have consecutive parent classes; these are classes that are involved in moving the entity, loading textures and animating them. These were added for their functionality and positioned so that it stops some code duplication to be used by all entity classes. Therefore, both the NPCs and the player can use the same movement function. Furthermore, we overlooked the animation class originally in the first design and did not account for it until the construction of the software started. This is similar to the textures assets for the game, which can be traced back to not fully understanding how the game engine works.

Secondly, we didn't know that we would need an extensive resource manager to handle all our assets; this was added to manage the loading and storing the relevant assets for the game. With this, there can be fast access to each of the images during run time requests.

With the "A Killer Hangover" class, it has a lot of the main variables, e.g. points, which were initially in the player class but was moved to this class as it was easier to implement with multiple classes. Additionally, the graphics with the health bar and points were easier to implement. Instead of the enemy class giving points and damage values to the player and then to the graphics module, it is all managed here to reduce complicated computation, creating a much simpler implementation.

Architecture

The below table goes into some detail on each component of the UML diagram.

Class	Explanation
BaseScreen	This is used for initializing a window and to set up a variety of stages. Each stage contains the capability for graphics on 3 separate layers (backStage, entityStage and uiStage) and also allows the user to provide input to the system. Every interval stage was added to the architecture to allow ease of displaying each object to the user. Else without this, we would have to put more resources into how and when to print to the screen. But with this it simplifies everything.
PlayScreen	Abstract subclass of BaseScreen which implements the polling for collision detection, assigning keys to movement and shooting, spawning zombies, moving the zombies, etc. Essentially provides all of the functionality used in the game in order to make it easier to create areas for the game. Furthermore, it was able to manage the main game mechanics. With it, in one place it was easier to maintain separate mechanics to find bugs or to update/change them. Also, this class stores the effect that the player will receive for picking up drops from the enemies.
SafeArea	A subclass of PlayScreen which grants the user points if it is the first time the user has been in the area.

MapScreen	Lists some of the locations in the game, where each item on the list can be clicked to navigate to the corresponding area. This is to keep other screen classes simplified as possible so it will manage the stated functionality.
ImageActor	A subclass of the in-built libGDX Actor class which draws an internally stored texture when the Actor's draw function is called. Draw is called by a stage that contains the actor. Also contains the code for checking for overlap with other ImageActors. You will notice that this is before the Animate Actor class this was architecturally chosen because the Animate class needs the image textures to perform its function without the images it will not achieve this.
AnimatedActor	A subclass of ImageActor which instead of drawing a texture draws an animation. Also contains a store of animations in order to allow for the actor having multiple different animations. It was its own class along with the Image actor class as it can be cloned when running the program with multiple instances of it being needed.
MovingActor	A subclass of AnimatedActor that implements all of the variables and a modification to the act function required to calculate simple movement based on a horizontal velocity, vertical velocity, horizontal acceleration, vertical acceleration and angular velocity. This was kept separate from the entity class so that it is more assessable to check that the functionality of the class works to its full potential. In addition, it was done to reduce the length of the entity class so that other parts of the method could be checked separately.
Entity	A subclass of MovingActor which implements a type system which allows for the Entity easily being cloned from the list of entities in the ResourceManager, and also implements the attributes used to calculate how fast the entity should move and how much health it should have and does have. You can see that this has been chosen as a parent class to the Character and NPC class as they can both share the same attributes.
Character	A subclass of Entity which contains the code for making the player face the mouse cursor based on an x and y coordinate in such a way that it does not affect which direction the player moves in.
NPC	A subclass of Entity which contains the code for making the NPC face the player and follow the player based on the player's x and y coordinates.
ResourceManager	A class which avoids the need to load image files and animation files multiple times by loading them in once and then storing them as ImageActors for images or entities for animations.
AKillerHangover	A subclass of the in-built libGDX Game class which contains the variables that should be accessible from every class in the BaseScreen family of classes.

Architectural requirements

ID (IDs are from requirements section)	Description	Architecture
Func.UI	The game requires a user interface.	This is mainly covered by the PlayScreen class but is helped by the classes: BaseScreen, MapScreen and SafeArea. These classes were designed to manage all graphical implementation.
Func.input	The user needs to be able to interact with the system with mouse and keyboard.	The method PlayScreen will be responsible for taking the input and acting accordingly to the user's actions. Like moving the player.
Func.Points	Players will be able to accumulate point when playing the game.	The KillerHangover class manages all of the points by keeping track of the total and points gain through versus actions like defeating an enemy in the game.
Func.Char	The user will be able to select different characters before starting the game.	With the Entity class being the parent class of the Character class it is easy to change the player's character between different settings associated with each player type.
Func.Area/Safe	Players will be able to traverse a variety of areas.	The selection of the Screen classes and the SafeArea class we have used this to allow the player to move through different areas in the game.
Func.Vary	The game will provide a selection of enemies throughout the game.	With the Entity class, we are able to accommodate this as like the distinctive players the player can use.
Func.Powers	The software will need to facilitate a selection of powerups/downs.	The PlayScreen class will aid in the implementation of this. It will both manage and display the relevant power up/down as required.