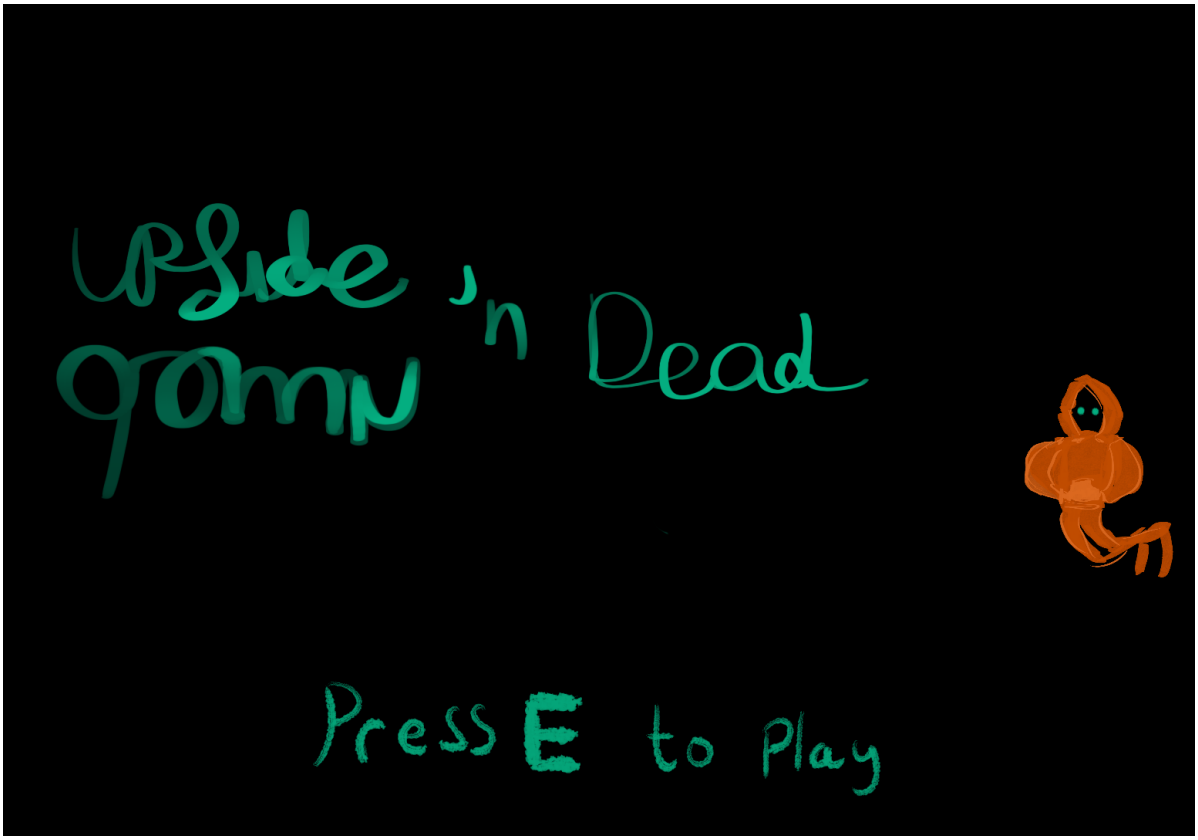
System design document for Upside Down & Dead

Amanda Dehlén, Linnea Johansson,
Johannes Kvernes, Anna Nilsson, Samuel Widén

2019
0.1



Contents

1 Introduction

Upside Down & Dead is a puzzle game presented as an application for desktop users. The game starts with the user finding out that they are dead and the world is upside down - hence the name. The user's goal is to flip the world back to normal through solving puzzles. In this document the design of the code behind the game will be discussed and presented.

1.1 Definitions, acronyms, and abbreviations

- NPC: Non player character. A character that players do not control, but can often interact with.
- Puzzle: Tasks for the player to do to complete a level, for example open the door with a key or cut down a bush with some scissors.
- Item: Items are things that exists in the game that the player can use to solve the puzzles, for example a key that can open the door to the next level or some scissors that can cut down a bush.
- Level: The game is divided into different levels that each are completed when all the puzzles in the level are completed and the door is opened to the next level.
- Trap: A trap is something that deceives the player into using it and makes the room more challenging for the player or slows the player down in their quest to complete the level.
- Dependency: In this document, dependencies refer to dependencies between classes or packages, which are formed when one module cannot exist without another.
- Model: The part of the code that is the domain specific part. The Model should do all the calculations and represent the brain of the program.
- View: The part of the code that shows the user the program.
- Controller: The part of the code that responds to the user interacting with the program. The controller communicates with the Model and View and 'tells' them what to do.
- Abstract class: A class with abstract methods to be implemented by its sub classes.

2 System architecture and System design

When the user starts the game, there is a creation of the main controller, that creates an instance of the model and the view. The game then starts, and initially shows the Start Menu View.

2.1 Design model

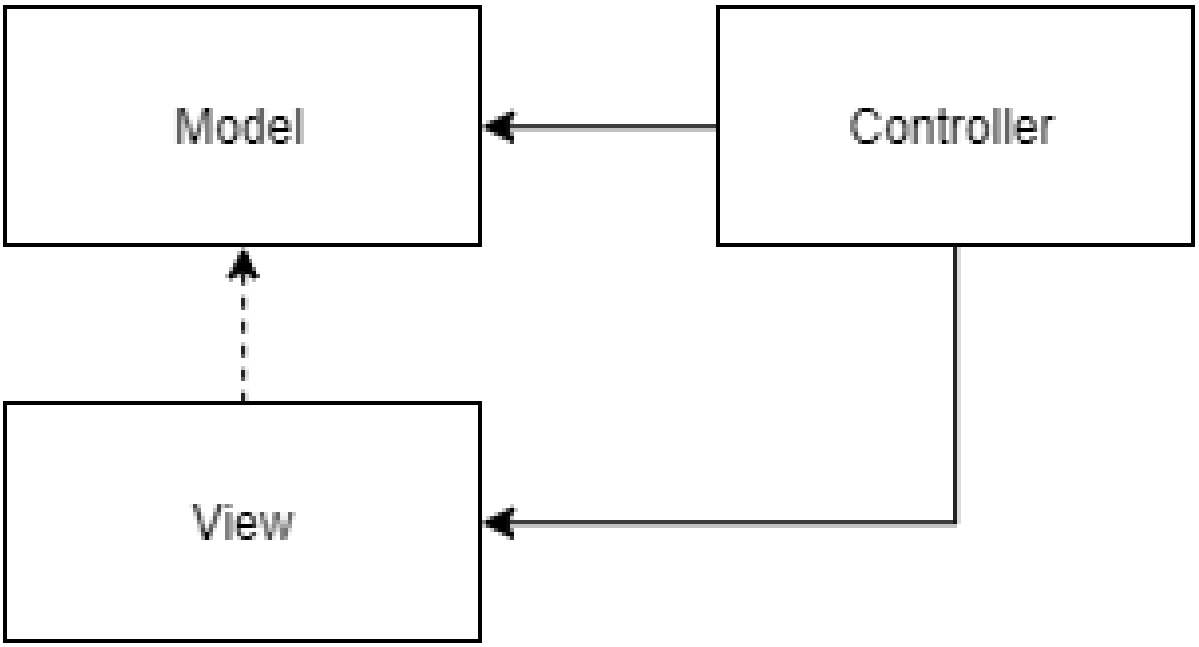


Figure 1: Top level UML between packages

Figure 1 shows a model that represents the connections between the top packages. This illustrates how we used the Model View Controller design model. Controller and View are highly dependent on Model. Model has no external dependencies. View has TextObservers that observe the Model. Controller has instances of classes from Model and View.

2.1.1 Model

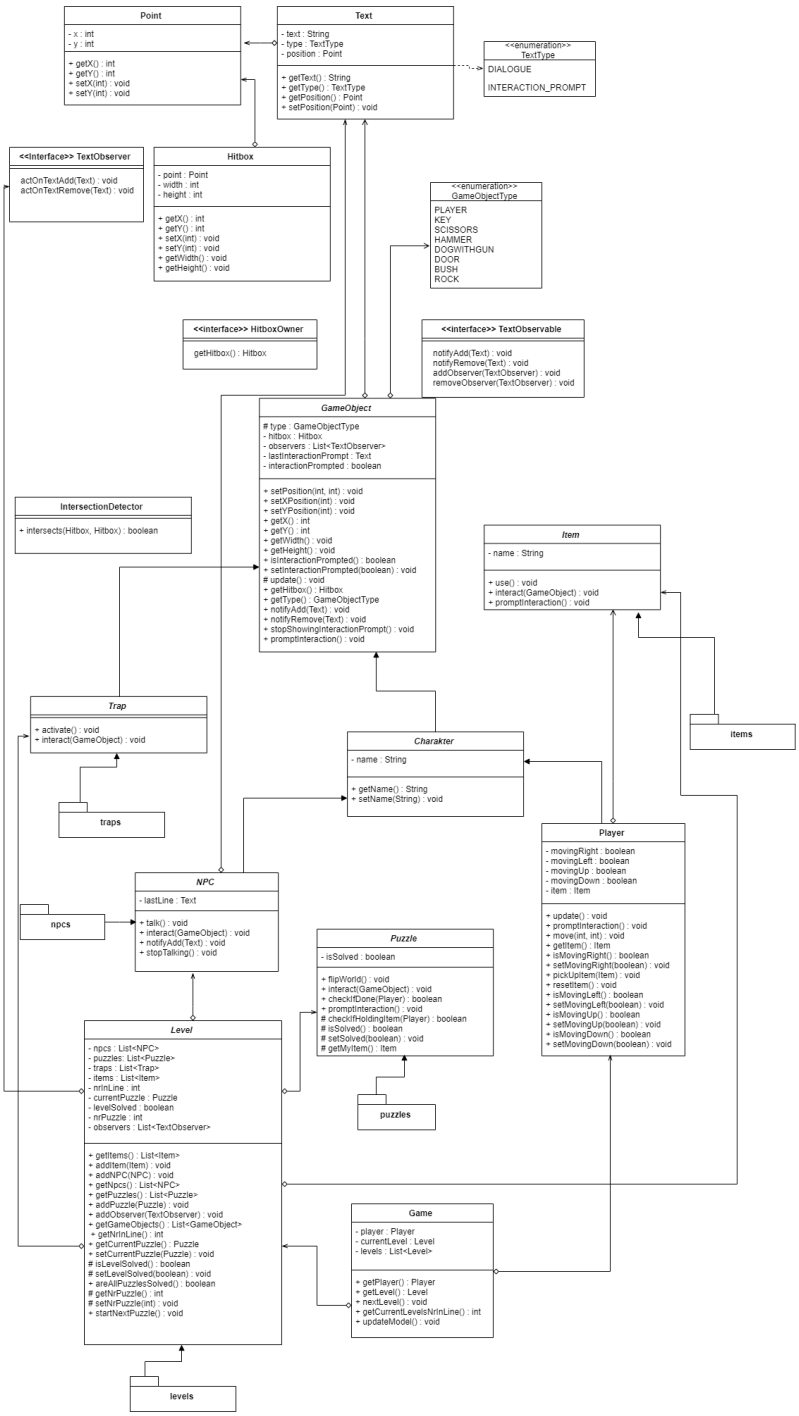


Figure 2: UML of the model package

2.1.1.1 Domain model and design model

Figure 2 shows the design model of the model package. This structure evolved from the domain model, which can be found in the RAD document. The implementation uses the modules/objects that are found there, and has the same relations between them; a level has puzzles, npc's and items etc. There are ofcourse more classes and interfaces in the design model so that the code runs and functions as it should. The following is a brief summary of the most important classes in the design model:

- Charakter: The abstract class for any character in the game that The class is intentionally misspelled to not interfere with java's own Character class.
- Game: The class represents an instance of a game
- GameObject: The abstract class represents a object that can exist in the game
- GameObjectType: An enum that holds the different types of objects that can exist in the game
- Hitbox: The class represents the area that the gameobject are located at and could intersect with others on
- HitboxOwner: A interface that returns the hitbox
- IntersectionDetector: The class used to check if objects intersect
- Item: An abstract class that represents items found in the game
- Level: An abstract class that represents a level in the game
- NPC: An abstract class that represents a character the player cannot control
- Player: Represents the player that the user uses to play the game and can control
- Point: The class represent an position on both the x and y axis
- Puzzle: An abstract class for any puzzle in the game
- Trap: An abstract class for any trap in the game

2.1.2 View

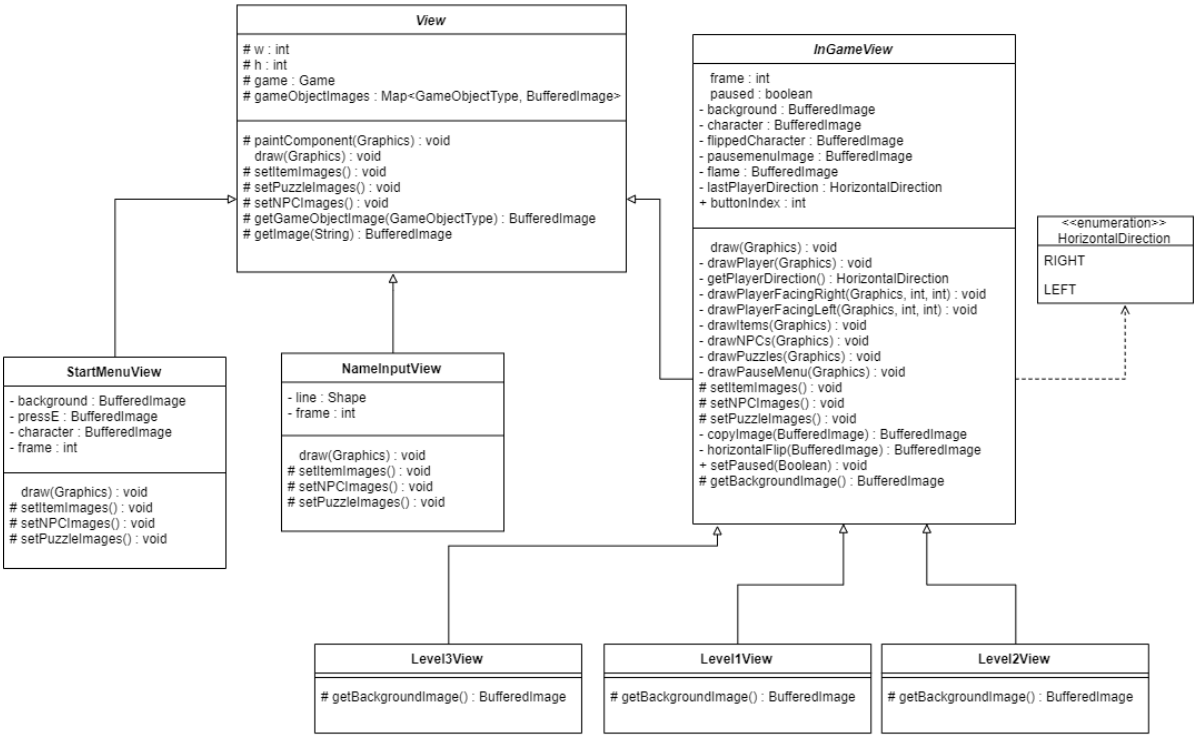


Figure 3: UML of the view package

Figure 3 shows the UML of the View. The LevelXView (where X represents a number) are the classes where the view of the specific level is located. There also exists a View for when the game is in the start menu as well as for when the user chooses a name. The InGameView is an abstract class that extends the abstract class View, to abstract the common factors found in the different Views.

- View: An abstract class for any view in the game
- InGameView: An abstract class for the view while in the game
- Level1View: Represents the view for level 1
- Level2View: Represents the view for level 2
- Level3View: Represents the view for level 3
- NameInputView: Represents the view for the name input
- StartMenuView: Represents the view for the start menu

2.1.3 Controller

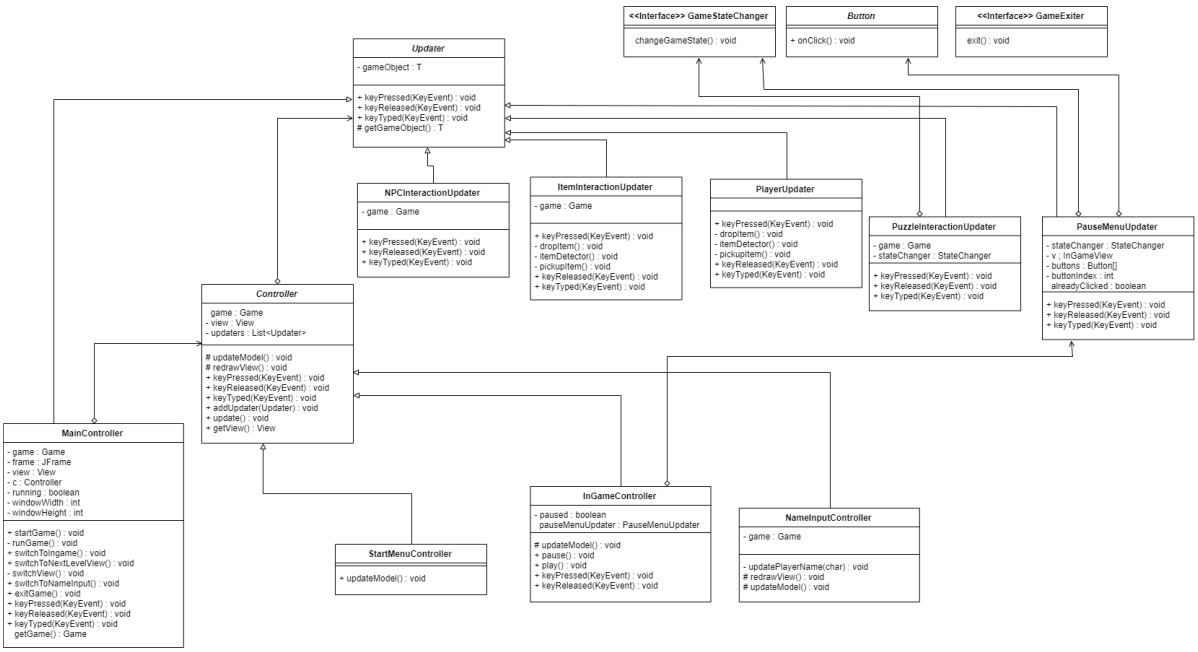


Figure 4: UML of the controller package

Figure 3 shows the UML of the Controller. There is an observer, the abstract class Updater, which many of the controllers extend. Similarly to the View package, there is a different Controller for the different Views.

- Button: Abstract class that is used for buttons
- Controller: Abstract class for all controllers
- GameExiter: Interface with method for exiting the game
- GameStateChanger: Interface with method for changing the game state
- InGameController: The in game controller that updates the model while playing.
- ItemInteractionController: Updater for all interactions with items
- MainController: The master of controllers, responsible for switching which one is active, as well as which view is shown.
- NameInputController: Public class that handles the name input
- NPCInteractionUpdater: Public class that handles interactions with NPC's

- PauseMenuUpdater: Public class that updates the pause menu
- PlayerMovementUpdater: Public class that updates the players movement
- PuzzleInteractionUpdater: A class that updates the level when you solve a puzzle
- StartMenuController: Public class that controls the start menu
- Updater: Abstract class that handles key input

2.2 Design patterns and principles

- Single responsibility principle: SRP is followed as every class does only one thing.
- Open-Closed principle: As there is much abstraction in the code, the project is open for extension but closed for modification.
- Liskov substitution principle: LSP is followed. An example of this is that you can switch any Item for any other, and the same goes for npc's.
- Interface segregation principle: An instance of the ISP is the GameExiter interface. Here, the MainController sends its exitGame() method to those that need it and take a GameExiter as an argument, for example the InGameController. This means that when the InGameController gets the GameExiter, it has only the exit() method, meaning it depends only on that method and not the rest of the MainController.
- Dependency inversion principle: Implementing the Observer design pattern leads to the Observable classes depending on the Observer interface instead of the specific implementation of an Observer, which adheres to the Dependency inversion principle since there are dependency on abstractions and not implementations.
- MVC architecture: The code is divided into three core packages: mode, view and controller. This structure aids the development of the project in the long run by introducing constraints on which dependencies are allowed to exist.
- Observer pattern: There are many instances of the observer pattern in the code. Some examples are: Controller (observes KeyEvents), Updater (observes Controller), and TextObserver (observs TextObservable, which is currently implemented as GameObject). In most cases, this pattern is used to avoid circular dependencies, but it also makes it easier to depend on abstractions, as mentioned in DIP.
- Template method pattern: This pattern is used for instance in the NPC class, where its interact() method uses the abstract method talk().

- Module pattern: In the code, the different modules Model, View and Controller are divided into different packages. The Model package also has sub-packages for items, levels, NPC's, puzzles and traps.

  Not yet implemented patterns:

- Facade pattern and Factory pattern: The usage of the Module pattern sets up for the implementation Facade and Factory patterns. This was planned, but not implemented.

2.3 Plans for implementations

If in the future it is decided that functionality for saving the progress of the game should be implemented, this should use some kind of database or similar. However, this has not yet been done.

A way to save levels (so that a new class for each level does not have to be implemented) has also been discussed. One thought about how to do this was to save levels as .json-files containing information about the level's items, puzzles, background image, etc. This would also allow for the creation of levels through a visual interface, a 'level editor'. This has not yet been implemented.

3 Persistent data management

The application does not make use of persistent data.

4 Quality

We test the code we write using JUnit, in a test package that can be found in ”src”.

- Known issues:
  - Switching to the next level results in index out of bounds, should be fixed by implementing a factory for Levels.
  - Switching levels is done in PuzzleInteractionUpdater, should be done in Door.
  - New InGameView subclasses have to be created for every level, which is not extensible. Can be solved by making InGameView more generic.
  - New Level subclasses have to be created for every level. Can be fixed by implementing a class to read levels from files (such as .json files).
  - The pause menu is directly implemented in the InGameController. It should instead be a state or similar that the InGameController can use.
  - Documentation: The code does not have the correct documentation everywhere.
  - Code Smells: In our code we have some code smells. For example there is some inheritance that should not happen. Classes that inherit GameObject has to implement update() but not all subclasses use this method. This results in unused code which is a sign of bad code. This should be resolved by abstracting this method perhaps into an interface and have only the classes that use update() implement this interface.
  - The pause menu-view’s graphics were designed with the intention of the option ”quit” being a quit and save the game-option. This has yet to be implemented but are one of the user stories that should be considered implementing in the future.
  - The graphics in the level are not yet finished. Therefore are some of the objects displayed using placeholders for the real images that are yet to be created.
- Analytical tools used for testing quality of code:
  - We tried to get STAN (stan4j) to work, but to no avail.
  - PMD has been used to eliminate some bad practices.

4.1 Access control and security

The application does not have any kind of access control and security