System design document for ”Nollans första dag”

Alexander Brunnegård, Julia Böckert, Malte Carlstedt, Steffanie Kristiansson, Clara Simonsson

24 oktober 2021
version 4

1 Introduction

This document describes the background and the construction of the application. The application is described from a perspective of a developer and includes terms and proceedings to come up to the final result, the game.

The game is called ”Nollans första dag” and is about completing different tasks to reach the main goal, create the ”Nollbricka”. The user controls a player, who can walk around through Campus, which is built up by different maps, and complete tasks. The user receives different items when completing the tasks. These items are used to create the ”Nollbricka”, and when they are all collected, the game is finished.

1.1 Definitions, acronyms, and abbreviations

- **Slick2D** - A 2D Java Game Library, set of tools to make development of Java games easier.
- **Non-Playable Character (NPC)** - An idle character.
- **MVC** - Model-View-Controller structure in the application. Its a design pattern for programming.
- **MVVM** - Model-View-ViewModel structure in parts of the the application. Its a design pattern for programming.
- **Nollan** - Newly admitted student at Chalmers.

2 System architecture

2.1 Overview

The application is based on the Model-View-Controller structure which means that the model is not depending on the controller or the view. The controller knows about both the model and the view, but the view knows nothing about the controller. The view is depending on the model only.
This makes it easier to change and extend each part separately without caring about the other parts in the MVC.

2.2 Software decomposition

The game consists of seven packages which are Model, View, Controller, NPC, items, textboxes and tasks. The model, view and controller-packages is the three components which are included in the MVC pattern. The other four are also following the MVC pattern but are separated by their responsibility. MVVM is used for the maps and it handles the loading of the different maps.

The model package handles the game logic and has seperated models for different components. It consist of a playermodel, mapmodel and all the different mapstates. All the logic about collisions with objects are also in the model package.

The View and Controller-packages has two classes each, one for the player and one for the map. There is also a viewmodel in the view package which is used by the mapView. It holds the logic about loading maps in the application.

The task, textBoxes, Item and NPC packages has a model, view and controller as well and they are separated. There are seperated MVC:s for each task. This structure makes it easier to expand and add more tasks because they're not depending on each other.

There are six different classes in the application which doesn't belongs to a package. These are EnterTask, GameMenu, HelpViewMenu, GameDoneView, MainGame and StateSetup. The EnterTask does as the name describes. It checks which of the different tasks are supposed to start depending on which map the player is on.
The Gamemenu and HelpViewMenu are in the startscreen of the game. MainGame holds all the information that should exist in the actual game. StateSetup handles the different states for the views and tasks. GameDoneView is the final view for the game.

2.3 Application flow

When the application is started the StateSetup class adds all states which will be used in the game. The state will always be zero at the beginning because it is connected to the game menu where the user can choose if it wants to start the game or go to the helpview.

When the actual game is started the application will enter state 1. All setups that will be in the game are initialized here and the player, maps, NPCs, tasks, items and textboxes are rendered and shown at the screen.
The mainGame holds an update method which are always updating whats happening in the game. It is depending on the different controllers for the packages and is for example controlling when the player walks, collides with objects or changes map. This is continuing until the criterias for ending the game is completed. The game will then change its state to the GameDoneView and the player will appear in a new view where the user can end the game by clicking on a button.

3 System design

3.1 UML diagram

To see the UML-diagrams go to the appendix (7.2).

3.2 Design Patterns

3.2.1 MVC

The very common pattern for the application is the MVC pattern. In particular the whole game is built up by this pattern. Every component in the game has its own model, view and controller and it makes the game less dependent on small and irrelevant classes. The application is also tolerant for changes in different components because it isn't tightly coupled to many other parts in the application.

3.2.2 MVVM

An implementation of the MVVM pattern is used in the application. This is used because some of the logic in changing and loading maps in the application couldn't be handled in the model because the maps consisted of tmx-files which should be a part of the view. In the MVC pattern the view should be dumb and wasn't supposed to handle the logic about changing maps and that's why the MVVM pattern was implemented.

3.2.3 Factory Pattern

To initialize every separate NPC and TextBox on the screen, two factory classes (one for each type of object) was created. This way, the client code can simply use the factory's method for creating whichever NPC or TextBox it wishes to render, without knowing any of the underlying initialization code. This also makes the code easily extendable, since the developer can simply add a method in the factory class to add a new object with different values, without having to view the object's constructor. Therefor, the code follows the Open-Closed principle.

3.2.4 State Pattern

To switch between the different maps a state pattern is used. All the maps are separate states which has its own class. They are all implementing the same interface and has common methods but they are implemented in different ways.
This makes it possible to easily switch between the maps through the MapModel which only needs to know the current map. This in turn makes it easier to add more maps without having to change in the classes connected to MapModel or the MapModel itself.

3.3 Relation between domain model and design model

The domain model closely follows the division of the code's packages, although the Player and the Map in the domain model are the central parts of the application's Model design, and all logic concerning these two components are stored together in one common Model, View and Controller-package (with separate classes for each model etc). In other words, the Player and Map are separate components of the domain model, but uses the same packages in the design model, which is the biggest difference between the two.

The two controllers in the design model contain a model and a view each, and the views use a model. The model is independent and has no dependencies on its views and controller, but the MapModel contains a MapState, and the class CollisionChecker (located in the model package) contains a MapModel and a MapState.

Not included in the Model package UML are the separate Model classes for each package containing Tasks, NPCs, Items and TextBoxes. NPCs and TextBoxes both use a Factory-class which is used by their respective model to initialize a list with all the desired objects from each package.

The component "HighScore" in the domain model exists as one class within the Task-package in the application's code.

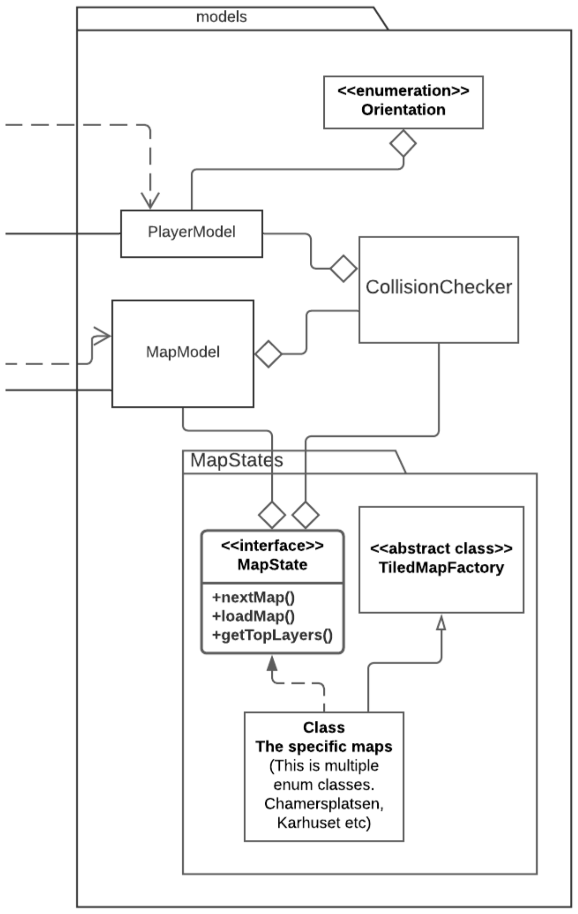

Figure 1: Model package

Description of Model package:

- CollisionChecker, a class used as a mediator between a specific map and the player, checks for collisions.
- Mapmodel, holds the information of the maps, contains CollisionChecker to keep track of collisions on the chosen map.
- ScreenViewModel, an interface to load the correct map
- Playermodel, used for all data-holding of the player, contains Orientation for the location of the player.
- MapState, interface implemented by all map enumerations.

- PlayerController, controls the arrow keys to move in different directions with the player and the different animations for each arrow key. Contains PlayerModel (to get the player), PlayerView and CollisionChecker (to check the collisions between the player and the layers).
- ItemModel, contained by ItemController, used to hold the items (filled and unfilled) for the game story.
- NPCModel, used by NPCView and NPCController, contains logic regarding the painting of the ConcreteNPCs, and uses a MapModel.
- TextBoxModel, used by TextBoxView and TextBoxController, contains logic regarding the painting of the TextBoxes, and uses a MapModel.
- NPCFactory, contained by NPCModel, contains methods to initialize ConcreteNPCs.
- TextBoxFactory, contained by TextBoxModel, contains methods to initialize TextBoxes.
- TaskModel (an abstract name of all the tasks' models, since they work in the same way). Contained by taskView and taskController, contains pictures, attributes and methods for setting up and manage the task-sequence.

4 Persistent data management

The application does not continuously store data in other forms than in .txt-documents containing the high scores of the tasks. New highscores are added to the files every time a new task is finished.

Every component drawn on the maps are images, each divided and stored in their own folder respectively (NPCs, Player-images, Task-components, etc.) under the repository's folder "data". The maps are .tmx-files with .tsx-objects, and the sound-files are .wav. All of them are stored in their own folder as well.

5 Quality

5.1 Tests

The main goal is to have 100 % test coverage for the Model package (including the Models in NPC, TextBoxes and Items). Tests regarding the rest of the project is considered good but not essential. All tests can be found in the folder "test" in our directory. The packages NPC, Items and TextBoxes each contain a test class to be able to test the package private methods, without compromising their encapsulation.

5.1.1 Test-related issues

Unfortunately we were unsuccessful in testing the NPCModel, TextBoxModel and some of ItemModel in the end. This has to do with a seemingly necessary image-initialization in every concrete MapState that appeared after a refactoring of the Map-classes, which was thought necessary to be able to test our MapModel at all. Tests for all model classes exist but some are now non-functional, but did however pass before this refactoring. JUnit cannot test any instances that use an image, and since these previously mentioned Model-classes' tests uses a MapState indirectly, the tests are now invalid. In consultation with a teacher, it was however decided that for now, this code structure was our best solution.

5.2 Known Issues

– When entering a new map the fade-in and fade-out transition is bugging a little bit.

  - On some maps the player experiences some lag when walking.
  - The window is not resizable, but full screen can be used.
  - Sometimes when adding highscore it duplicates the highscore. Probably has to do with the adding of highscore happening in the middle of a game update.

5.3 Quality Checks

- We ran the quality tool PMD on our code. Link can be found in our GitHub repository by the name ”PMDreport.html”. Most warnings were concerning variable names, missing constructors, possible violations of Law of Demeter and unused imports.

5.4 Access control and security

N/A

6 Further work

6.1 Ideas we have regarding development of the code

The code can always be made more efficient, better structured and easier to read. It's a never ending process and is something that we have learned during this project. We have switched up the structure a couple of times but the following are some changes that we would like to implement given more time.

6.1.1 Initialize NPCs later

First we would have made the NPC's initialized first when the correlated map is about to be shown to the player. As it is now, all of the NPCs is initilized at the start of the game but are not drawn until the correct map. If we would have changed up the structure and not initialized the NPCs until they are to be used, we would have a more efficient program that are not wasting resources when not used.

6.1.2 Initialize Tasks later

Plenty tasks are being initialized when the game starts, which is slowing it down pretty much. Instead we would want to fix so they only get initialized when needed, it would make the game more efficient. What we didn't know when deciding to separate tasks and the main game in different tasks is that the tasks are being initialized at the start of the game through the StateSetup class. In retrospect we would like to have the tasks initialized first when we are about to enter a task. We believe that this would free up resources and make the game more efficent.

6.1.3 Not Extending State Based Game

As the structure is now, we are extending State Based Game in several of our classes. This forces us to override the functions: init, render and update. This make a bit awkward mvc structure since we are having an extra step for each update and render functions. Example of this is that all our controllers and updates have their own update and render functions. It seemed like a good idea at the beginning, but for maintaining the code with different developers it might seem a bit odd.

7 References

7.1 Tools

PMD Quality tool: https://pmd.github.io/latest/index.html

7.2 Appendix

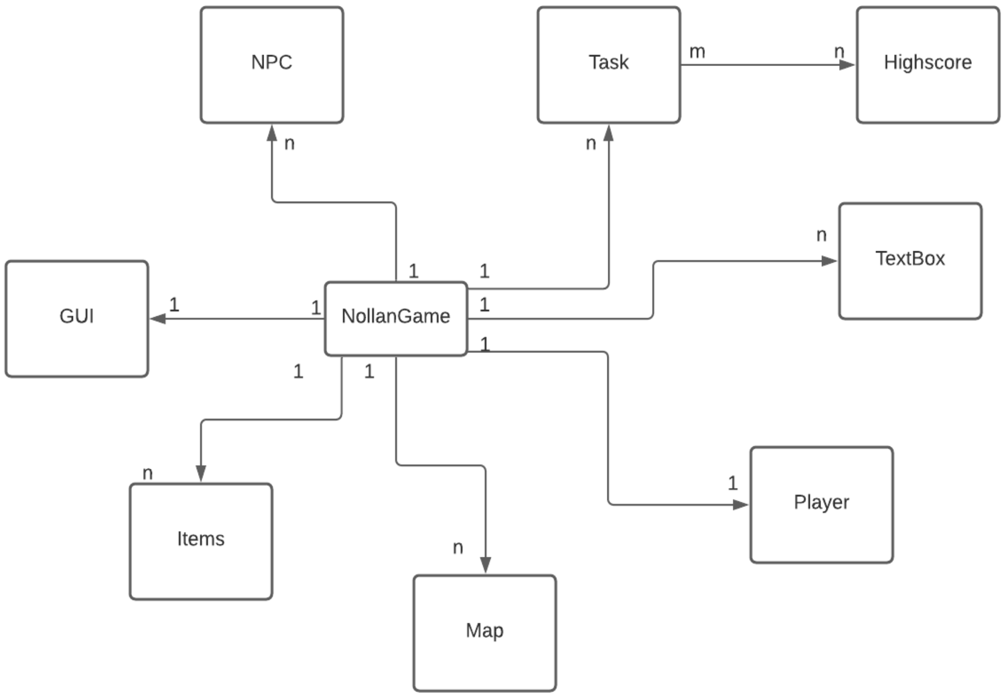

Figure 2: Domain model

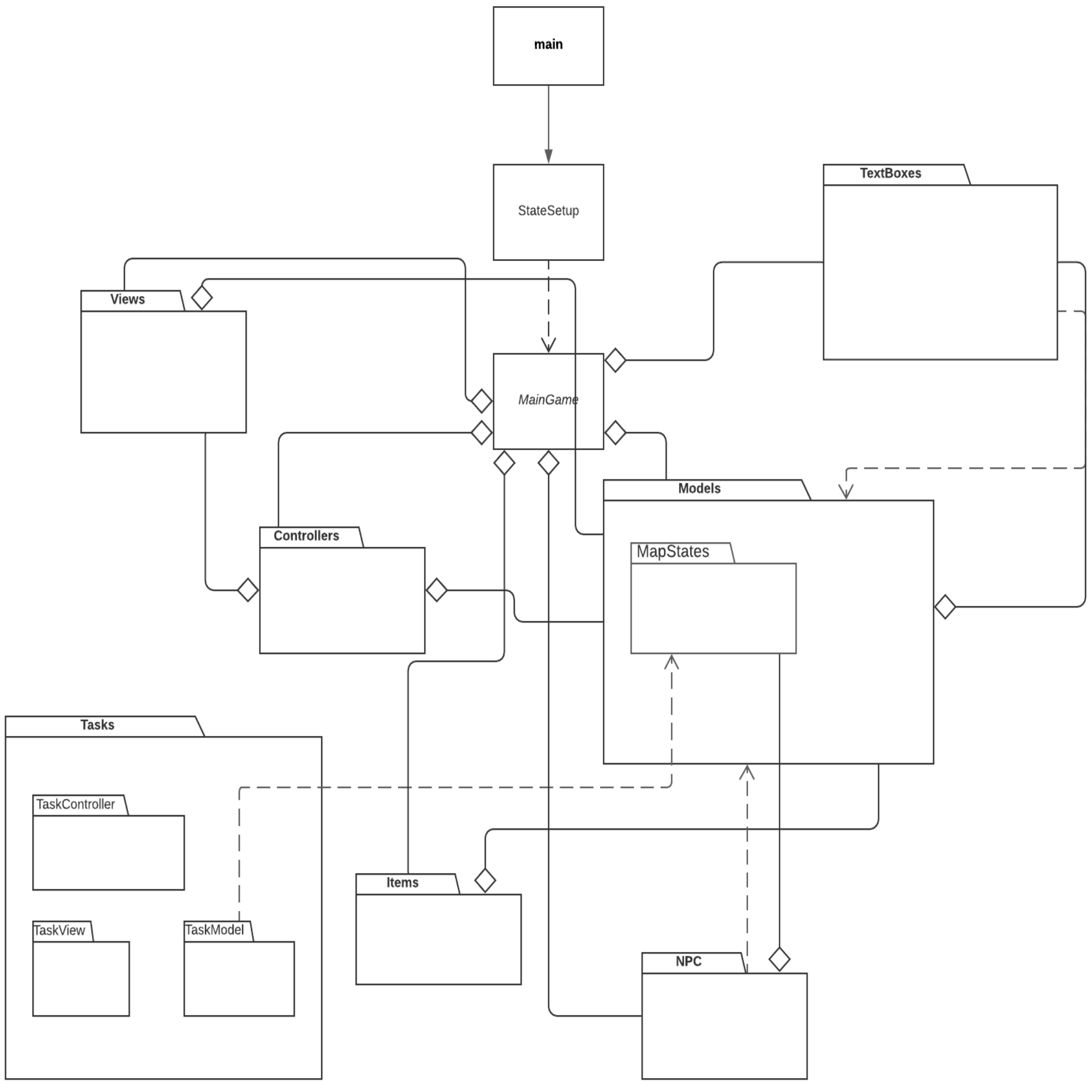

Figure 3: UML diagram of all packages in the application

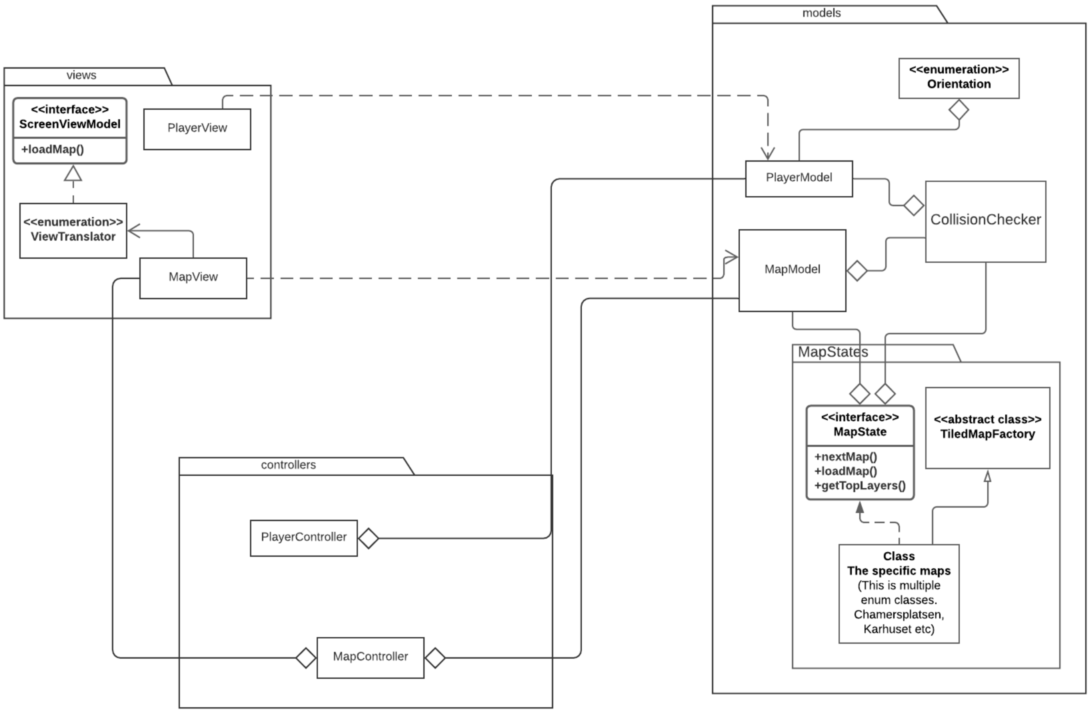

Figure 4: The MVC structure

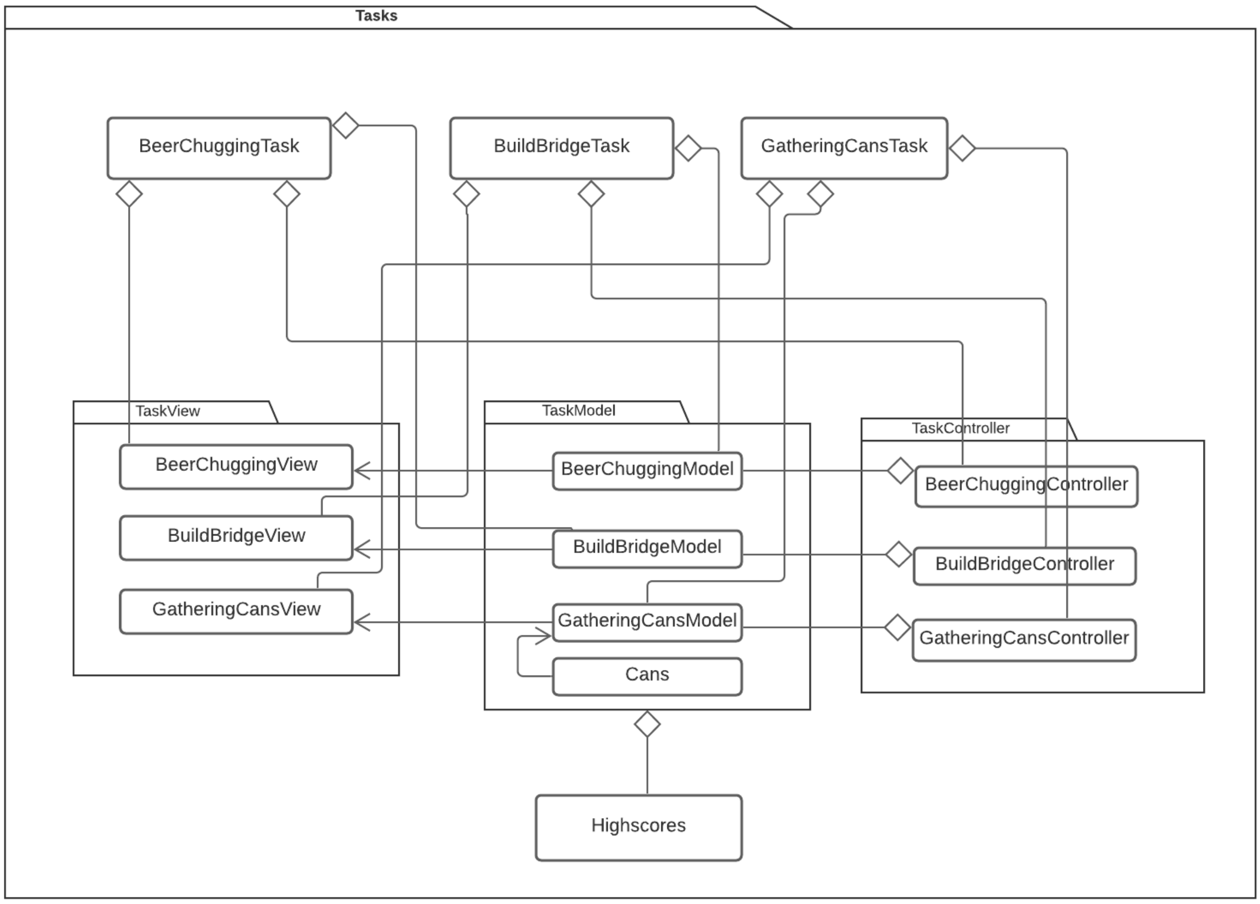

Figure 5: UML diagram representing the tasks

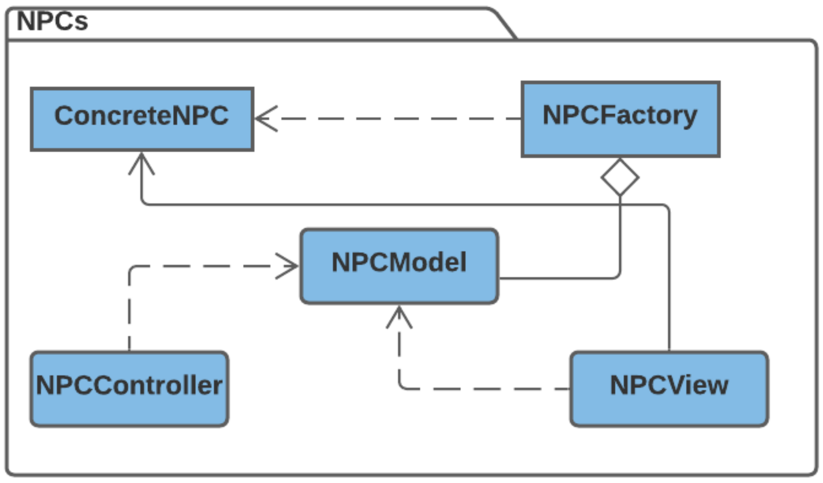

Figure 6: UML diagram of the NPC-package

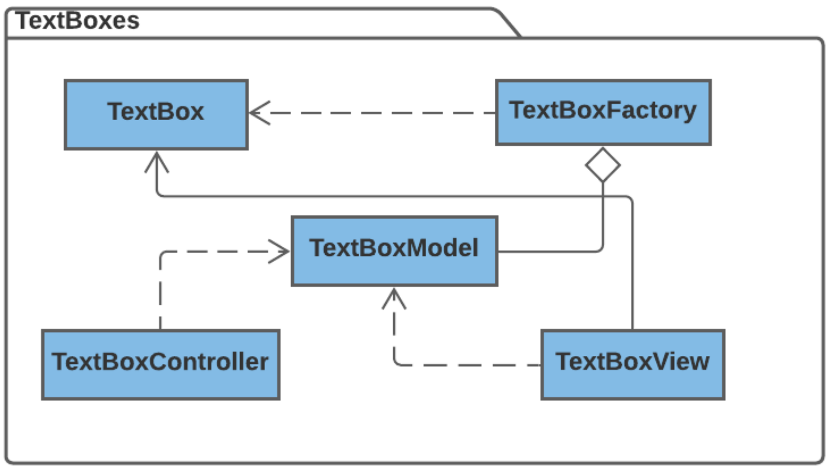

Figure 7: UML diagram of the TextBox-package

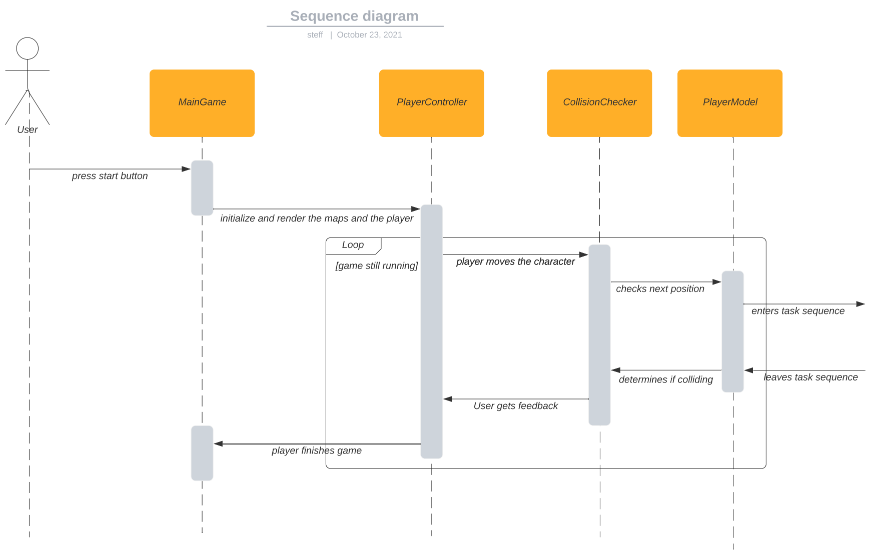

Figure 8: Sequence diagram

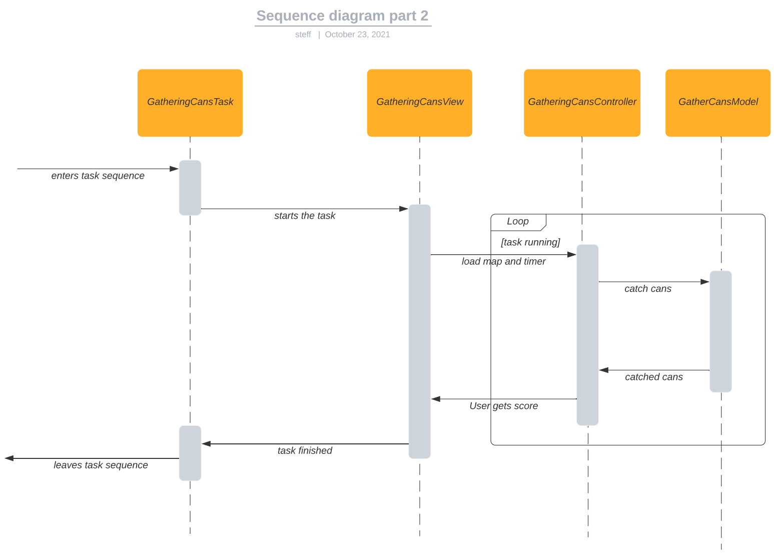

Figure 9: Sequence diagram part 2