System design document for Project_D

Version: 1.0
Date: 2017-05-02
Author: Jesper Blidkvist, Edvin Meijer, Hampus Carlsson, Emil Jansson
This version overrides all
previous versions.

1 Introduction

General info. What is this? What does it describe?

1.1 Definitions, acronyms and abbreviation

- Player: The person holding the controller.
- Supervisors: The “good” characters in the game, trying to get the factory to function as usual.
- Saboteur: The “evil” player, is trying to destroy the factory from the inside.
- Worker/NPC: The characters walking around the factory that serves the purpose of being decoy for supervisors/saboteur.
- Machine: The target for the saboteur and the things all supervisors need to maintain and secure. The main objective of the saboteur.
- Spotlight: A spotlight that casts a pillar of light around on the map, when hovering over a sabotaged machine for the first time it will trigger an animation of the machine that is destroyed.
- Honest interaction: Either maintaining a machine or entering the spotlight controller.
- Dishonest interaction: The saboteur is sabotaging a machine.
- Catch: The supervisors has figured out (think that they’ve) who the saboteur is and tries to catch him/her.
- Blackout: A saboteur has the ability to make a 5th of the map around him/her go black and this way escape in the darkness, used to give the saboteur another chance if spotted by the supervisors.
- Strike: The supervisors have tried to catch 4 workers but none of them were the saboteur they will start to strike and the supervisors lose.

Some definitions etc. probably same as in RAD

2 System architecture

Project_D is meant to be played on one single computer with multiple controllers. It runs on Windows, Mac and Linux since it is written in java and the JVM (Java Virtual Machine) is supported in these OS's.

The project implements a model, view, controller pattern and uses some additional custom packages.

A level editor has been created in Blender which is able to export to our map format which can then be imported into the game.

An AI package has also been created that is able to create nodes and assign these values which can then be utilised by entities in the game. First built as a simulation for the AI behaviour but is written in a way that some classes can be reused.

A lot of the functionality for the view will be composed of preexisting functionality included in libGDX.

The most overall, top level, description of the system. Which (how many) machines are involved? What software on each (which versions). Which responsibility for each software? Dependency analysis. If more machines: How do they communicate? Describe the high level overall flow of some use case. How to start/stop system.

An UML deployment diagram, possibly drawings and other explanations.
Possibly UML sequence diagrams for flow. (Persistence and Access
control further down) Any general principles in application? Flow, creations, ...

3. Subsystem

decomposition
For each identified software above (that we have implemented), describe it …

3.1 Model

The model consists of two main branches deriving from the GameObject class, the entities and the static objects.

3.1.1 Static objects

The static objects represent all objects in the game that just stand still and may be walls but also machines that the players are able to interact with. These are all represented by a 3D model in the rendered view.

The OOP design structure is simple. Mainly class inheritance and not a lot of design patterns except from a state pattern used for the machines to represent their current state. They can be either active so that they're able to interact with the players or "destroyed" and therefore not able to interact with the players. The usage of the states is when a machine is "destroyed" the it delegates to an empty method.

3.1.2 Entities

The entities represent all objects in the game that can move around still and may be walls but also machines that the players are able to interact with. Some of these are represented by a 3D model in the rendered view.

We've put more time into designing the Entity branch and therefore the structure is more thought through. To reuse a s much code as possible we've tried to structure the inheritances in more layers. Sometimes a layer is just used to make another method reusable but nothing more. In PlayerCharacters are designed with the template method pattern for the different behaviours corresponding to supervisors and saboteurs. Each PlayerCharacter has an internal state which is used to determine if a function should be executed or not. This is needed since characters controlled by players can "be caught" and therefore they're out of the game.

3.2 View

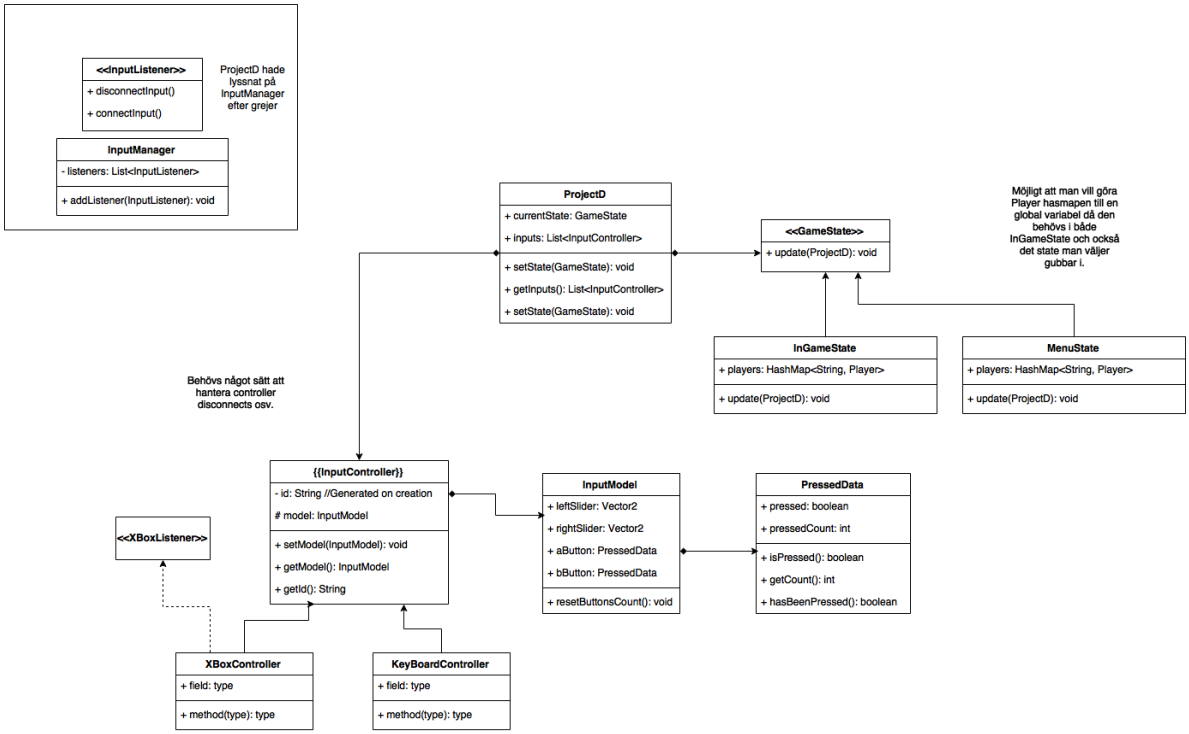
As stated before lot of the functionality for the view will be composed of preexisting functionality included in libGDX.

3.3 Controller

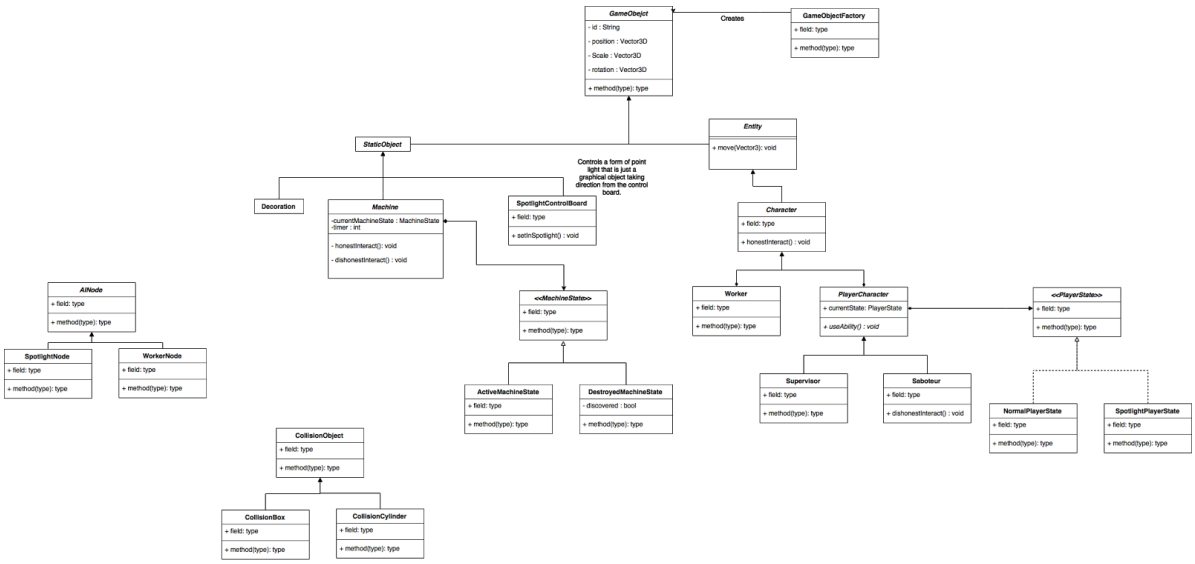
The controller will handle input from a number of Xbox controllers to enable players to interact with the model.

3.3.1 Xbox Controller UML

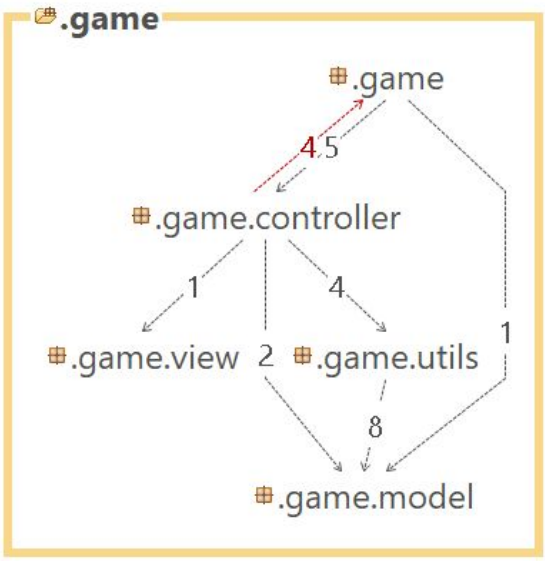
Here is UML for how the Xbox input will be handled.



3.3.1 MVC-Controller UML



3.4 AI



3.1 “...First software to describe” ...

Recap: What is this doing (more detailed)? Divide it into top level subsystems. An UML package diagram

for the top level. Describe responsibilities for each package (subsystem). Describe interface. Describe the flow of some use case inside this software. Try to identify abstraction layers. Dependency analysis
Concurrency issues.
If a standalone application

- -Here you describe how MVC is implemented
- -Here you describe your design model (which should be in one package and build on the domain model)
- -A class diagram for the design model.

else

- -MVC and domain model described at System Architecture

Diagrams

- -Dependencies (STAN or similar)
- -UML sequence diagrams for flow.

Quality

- -List of tests (or description where to find the test)
- -Quality tool reports, like PMD (known issues listed here)

NOTE: Each Java, XML, etc. file should have a header comment: Author, responsibility, used by.., uses ...

3.2 “...next software to describe” ...

As above....

4. Persistent data management

Data is stored in an assets folder. Within the assets folder there are a few subdirectories for storing maps, models and shaders. Data is accessed using classes libGDX provides, for example FileHandle.

How does the application
store data (handle resources, icons, images, audio, ...). When?
How? URLs, pathe’s, ...
data formats... naming..

5. Access control and security

Project_D is a local only game, therefore no communication is involved.

Different roles using the a
pplication (admin, user, ...)? How is this handled?

6. References