

System design document for Project D

Version: 1.6

Date: 2017-05-28

Author: Jesper Blidkvist, Edvin Meijer, Hampus Carlsson, Emil Jansson

1 Introduction

1.1 Definitions, acronyms and abbreviation

- **Player:** The person(s) holding the controller(s).
- **Supervisors:** The “good” characters in the game, trying to maintain the machines and keep the factory functioning.
- **Saboteur:** The “evil” player, trying to destroy the factory from the inside by sabotaging the machines.
- **Worker/NPC:** None playable characters maintaining machines and walking around the factory. Can be used by both supervisors/saboteur(s) to hide in the crowd.
- **Character:** Term used to describe both supervisors, saboteurs and workers.
- **Machine:** The target for the saboteur and the things all supervisors need to maintain and secure. The main objective of the saboteur tot destroy.
- **Sabotaged Machine:** A machine sabotaged by a saboteur.
- **Destroyed Machine:** A machine that has been destroyed due to lack of maintenance by workers/supervisors.
- **Spotlight:** A spotlight that casts light on a section of the map revealing its content. When hovering over a sabotaged machine for the first time it will trigger an animation revealing that the machine is sabotaged.
- **Honest interaction:** Either maintaining a machine or changing the state of a player to controlling the spotlight.
- **Dishonest interaction:** The saboteur is sabotaging a machine.
- **Catch/capture:** The supervisors believed that they have figured out (think that they’ve) who the saboteur is and tries to catch/capture him/her.
- **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.
- **LibGDX:** A game framework that we base or game on.
- **Bullet:** A physics library that is integrated into libGdx and is used for collisions and game physics.

2 System architecture

Project D is meant to be played on a 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 by each of these OS’s.

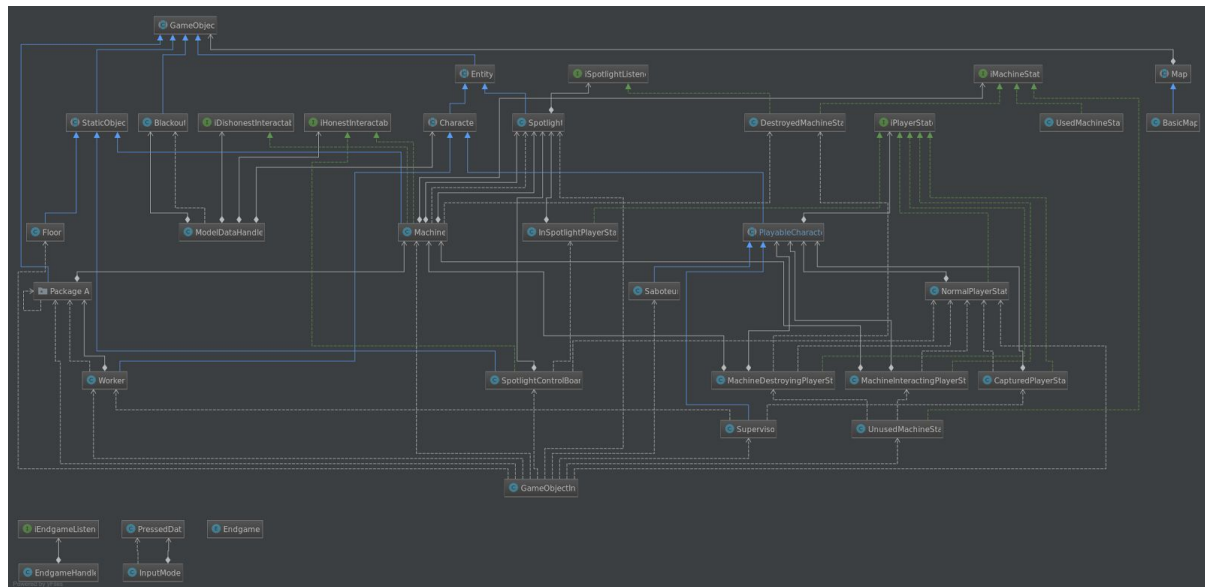
The project implements a model, view, controller pattern (MVC) and uses some additional custom packages/frameworks like libgdx, and by extension LWJGL and OpenGL, for rendering and JBullet for handling collisions between player controlled objects. A lot of the functionality for the view will be composed of this preexisting functionality included in libGDX in the sense that we will almost never interact directly with OpenGL. JBullet was primarily chosen due to its integration with libGdx

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. The map editor utilises a python script in blender that exports an xml-file and then LibGdx own file manager to interpret it into java compatible code.

An AI package has also been created that can be used to create nodes and assign these values which can then be utilised by entities in the game to navigate around the map. A description of which can be found in 3.5. First built as a simulation for the AI behaviour but is written in a way that some classes can be reused.

3. Subsystem

3.1 Model



(Better resolution and with the possibility to zoom can be found here:

<http://imgur.com/Mun0My7>)

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, players and workers to represent their current state. This will be described more in detail in the state pattern section below. 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. Eg. the usage of the states is when a machine is "destroyed" then it delegates to an empty method if a player honest interacts with it.

3.1.2 Entities

The entities represent all objects in the game that can move around on the map. This include for example workers and players. 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 as 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 and extensible. Characters are designed with the template method pattern for the different behaviours corresponding to being caught by the supervisors. Each Character 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

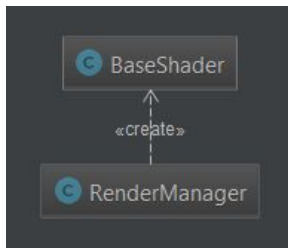
3.1.3 The state pattern

In the game there are a lot of objects that needs to have multiple behaviours depending on the current game situation. The machines can be either active or destroyed, the players can be in a normal player state or in a state in which they control a spotlight and the workers can be either working or in on strike.

Initially there was a discussion on if a decorator pattern or a strategy pattern could be used and if these would have any advantages over a state machine occurred, but in the end the state pattern was the winner. The state pattern was declared the winner since it is feels like an extension of the class since it represents an internal state. Depending on the situation it is easy to change state and through dynamical typing run the code that we want to. It also increases the readability and understandability of the code where it is feels more natural when a Worker is in a captured state rather than it implements a strategy that it can not move.

Compared to strategy pattern which would have been the second alternative the state pattern support that the behaviour can be quite different from state to state whereas the strategy pattern is different methods to achieve a similar or the same behaviour. Because of this and the other profits the strategy pattern therefore weighs up the circular dependencies that it creates between the state and the context.

3.2 View



3.2.1 Render manager

As stated before lot of the functionality for the view will be composed of preexisting functionality included in libGDX. The view mainly consists of a RenderManager that as the name implies handles all the rendering by getting passed a collection of objects that map to a specific graphical instance.

The rendermanager also handles rendering everything once from the camera to generate a shadow map that is later used during the normal rendering to create shadows.

3.2.2 MenuView

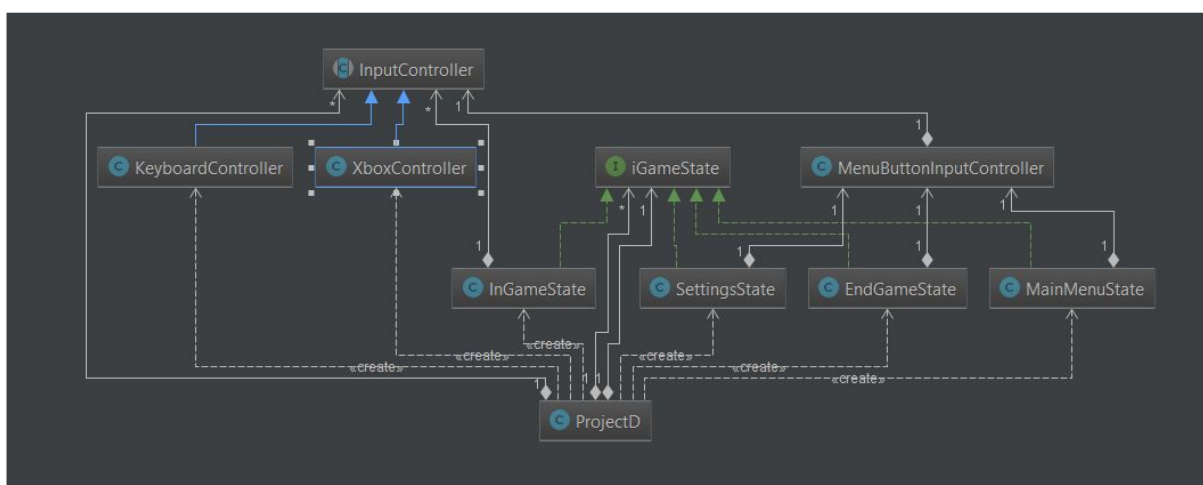
This class is responsible for rendering all menus in the game.

3.2.3 Base Shader

The view also includes a basic shader class used to create a basic rendering pipeline with custom shaders.

3.3 Controller module

The controller module consists of the input handler classes and the different game states that currently have support. The controller also handle input from a number of Xbox controllers or a keyboard to enable interaction with the model.



3.3.1 Game States

The game states are the internal state of our game and determine what the player view. It can either be a menu or a part of the game view. Since it represents the internal states of the

game eg. when the game has ended there will be an ending state that shows an ending screen.

3.3.1.1 iGameState

Interface which defines the methods which all game states have to implement. These consists of an init, start, stop, update, resize and an exit method.

3.3.1.2 InGameState

This state controls the actual gameplay and what the player see in the game view. Ties the MVC together in a state that runs the game.

3.3.1.3 MainMenuState

This state displays the main menu (UI) where the player(s) can either start the game or go to the settings state.

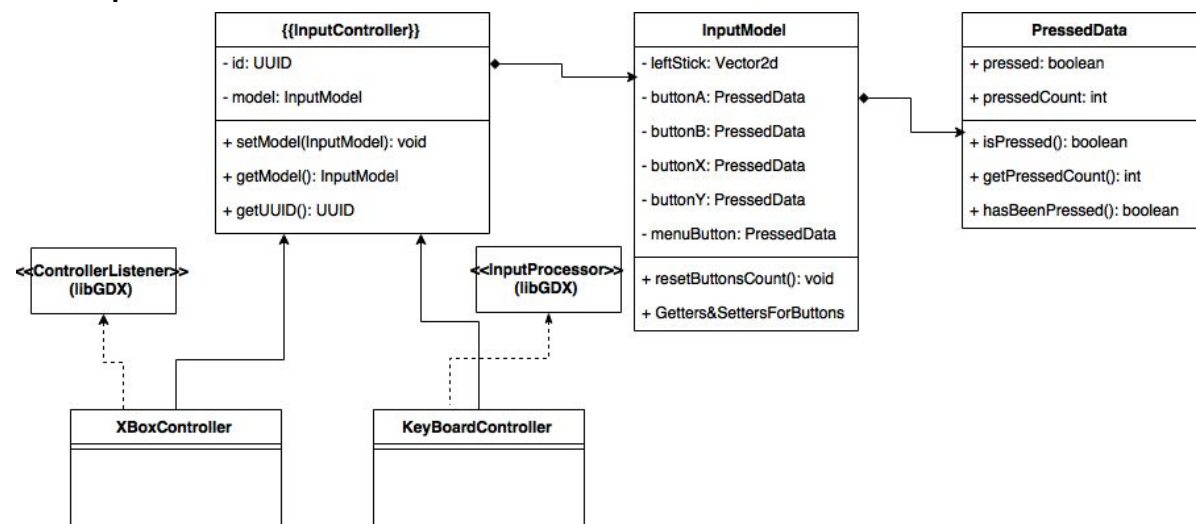
3.3.1.4 SettingsState

This state displays the settings (UI) with various parameters and setting for the game, mainly related to the game view.

3.3.1.5 EndGameState

This state displays a screen telling who won and how while pausing the game and redirecting the players to the main menu.

3.3.2 Input



User inputs are handled by the input controller classes. Each controller contains a **InputModel** which itself contains input data, for example which button is pressed. This makes it possible to easy to support many different types of controllers.

Inputs are later read in all different game states. This makes it possible to respond to a button differently depending on which game state it is in.

3.3.2.1 InputController

This is an abstract class that is a base for all sorts of inputs. It contains an InputModel instance where all input data will be saved. The classes that implement this interface then translates inputs such as keycodes or joysticks.

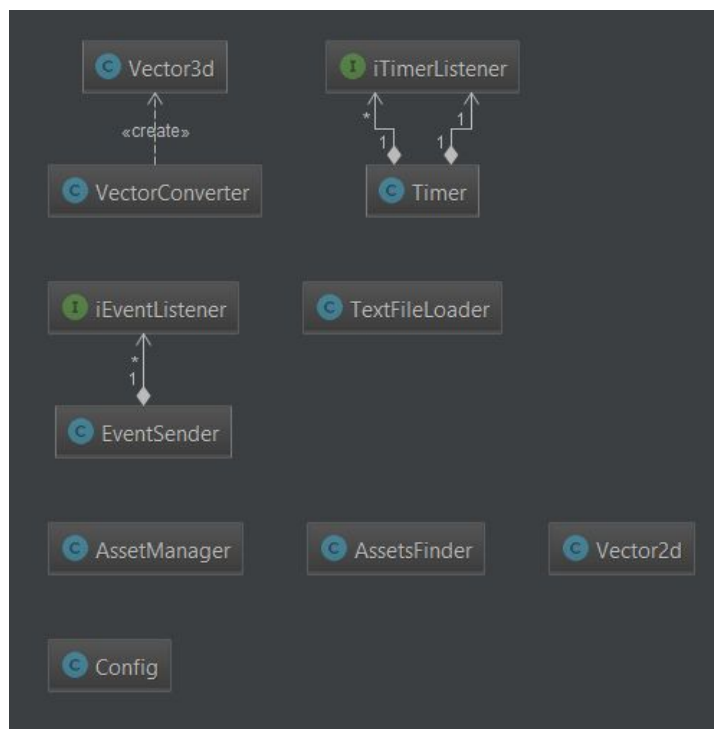
3.3.2.2 KeyboardController

This input controller is responsible for handling input from the keyboard and converting it to a format known to the model.

3.3.2.3 XboxController

This input controller is responsible for handling input from the Xbox controller and converting it to a format known to the model.

3.4 Utils



3.4.1 Asset manager

The asset manager is a facade for the assetmanager provided by libGDX. Has support for loading various assets and changing the texture of a specific modelinstance but is currently only used by GameWorld to get the modelInstance.

3.4.2 Asset Finder

The asset finder ties the ID of our gameObject to a specific model and the possibility of using a texture not defined in the model format.

3.4.3 Config

As the name implies, this class is used to store variables that might be considered global such as a enabling and disabling debug printouts throughout the program or changing the background color of all menus.

3.4.5 Event sender

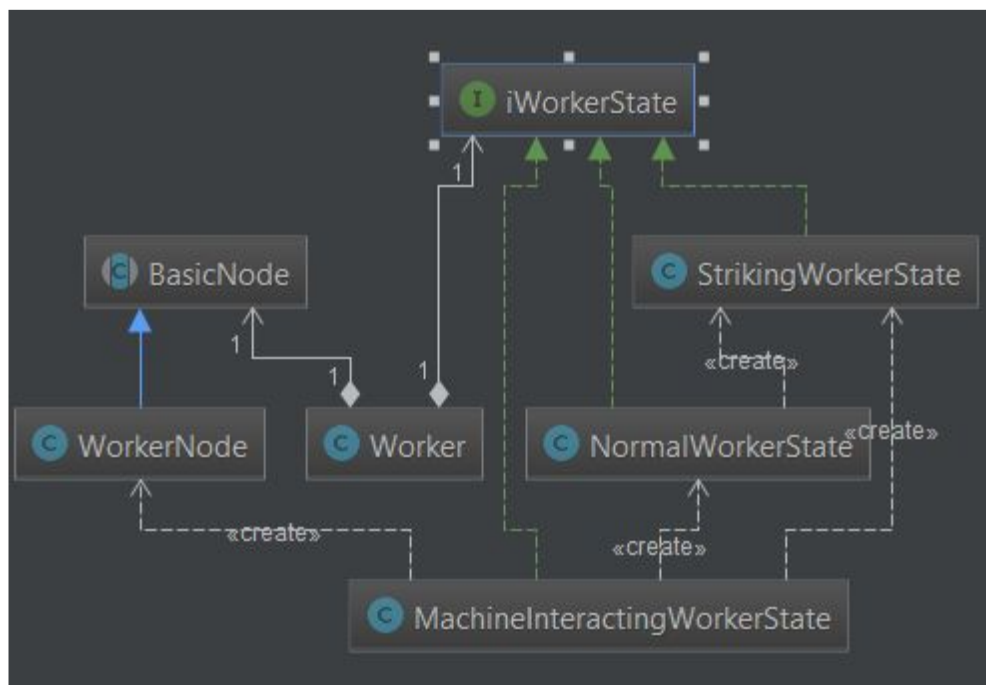
The event sender is a class that utilizes the singleton pattern by creating a single instance of itself and allows this instance to be accessed through a static method. The event sender is able to send event listed in the enum Events to objects that implement the interface `iEventListener`. This allows for indirect communication from one part of a system to another. The event sender is used to allow the model to notify the controller in Project D but is not limited to this usage.

3.4.6 Timer

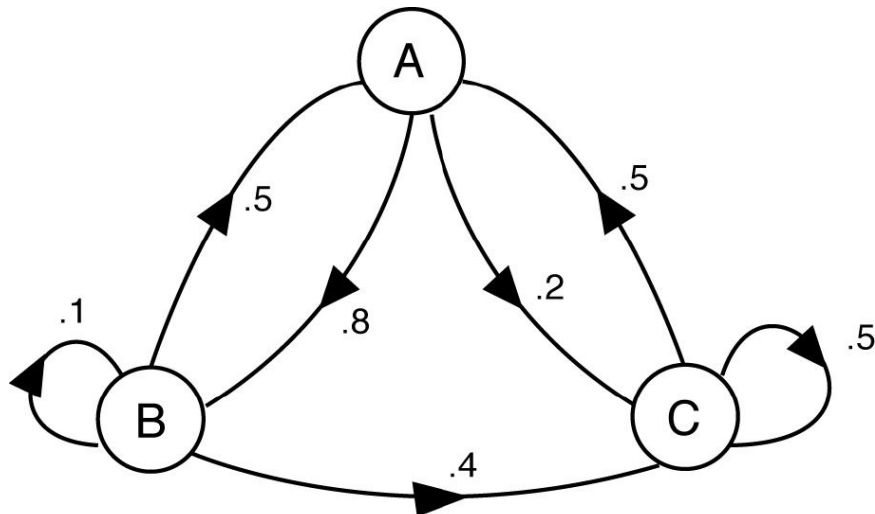
The timer class is used to count down to zero from an arbitrary value and, when done, notify classes implementing the interface `iTimerListener`.

3.5 AI

The AI package consists of a node class which allows creation of graph that can later be used by entities in the game to move around the map, creating the illusion of intelligent behaviour.



The nodes have a list of connections, and an associated strength with each connection allowing fine tuning of how likely entities will be to choose a certain path.



Visually this could be described by the above graph. A for example has connections to B and C with the strengths 0.8, and the strength 0.2 respectively. In essence, this makes an entity 80% more likely to move to B when positioned at A. Due to the fact that every node contains its own list of connections and strengths this allows as shown in the graph a different likelihood of moving from B to A than from A to B. Essentially this is a Markov chain.

3.6 GameWorld, GameObjectContainer and EntityContainer

GameWorld creates all GameObjectContainers and adds them to a bullet physics world.

GameObjectContainer is a container class that contains 3 different types of objects that are all in a way connected to each other. It contains a GameObject, GraphicObject and PhysicsObject. There is also an EntityContainer which extends GameObjectContainer. It is supposed to be used for GameObjects that will be moving around in the world, entities. Therefore it has methods which enable it to move and rotate the GameObject around in the world.

3.7 Physics, collision and jbullet

The game uses jbullet for physics which is a wrapper for the C++ engine bullet. Project D uses jbullet for collisions and movement. It uses the btRigidBody provided by jbullet for all physics objects. These physics objects are added to a btDynamicsWorld which then automatically handles all collisions when force is added to the physics objects. When a collision happens a callback will be made to the inner class CollisionListener which extends ContactListener within InGameState. The callback method onContactAdded provides information about which GameObjects a collision happened with. That information can then be sent further for more functionality.

3.8 LibGdx

LibGdx is a game development framework which provides basic functionality for a number of common features desirable for a game that a developer can build upon and implement as seen fit. It is used in this project to facilitate rendering of objects to the screen. This is done by passing the collection to the render manager, along with a list of all the lights in the scene

and the camera. The render manager then uses the render objects in this list to draw the objects to the screen and the light objects to accurately light the scene.

3.9 Map parser and GameObjectInit

A map for projectd is easily built using our xml map format. This format is read by a mapparser which sends values read from each row and sends them to a GameObjectInit object. GameObjectInit has a lot of default values that can be overridden by values from the map. When the row is finished map parser execute a method from the GameObjectInit instance that creates a GameObject with the values provided from the map.

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.

4.2 Map format

A map is stored using our own custom XML tags and as can be seen below:

```
<Machine x="-30" y="0" z="120" rotationX="0.0" rotationY="34"
rotationZ="0" scaleX="1" scaleY="1" scaleZ="1"></Machine>
```

The map itself is created by exporting objects from the 3D modeling application Blender using a python script.

Each tag is associated with an object in the model, in the above example a machine, and its associated transformation attributes are parsed and used to instantiate the object in the game.

If a value is missing the parser will create an object with predefined default values, for example in the case of positions if the xyz coordinates are not defined the parser will set these as 0.

Currently, when the new game button is pressed in the main menu the inGameState's init method is run which parses the selected map.

5. Access control and security

Project D is a local only game, therefore no communication is involved.

6. References

https://sourcemaking.com/design_patterns/state

<https://github.com/libgdx/libgdx/wiki>

<https://gradle.org/>

Joel Hultin, handledare TDA367

Joakim Von Hacht, föreläsare TDA367

Adam Wendelberg, handledare TDA367