Software Design Document for KeyboardChaos

1. Introduction

1.1. Design goals

The design of Keyboard Chaos and each module strive to be open for extensions. New spells, maps, goals, bodies etc. is easily added and by using a MVC pattern on the code ensure that a possible port to network games should be possible.

1.2. Definitions, Acronyms and Abbrevations

- Java, the programming language.
- libGDX, the framework currently used by our game, see rapporten(?) for more information.
- Body, the graphical representation of a game object (player, spell, lava...), used by Box2D for physics.
- Spell, an umbrella term for different type of actions a player choose from during the selection stage, and later cast during rounds. This includes, but is not limited to, Fireball.
- Fireball, an object which is created from a class that extends the abstract class OffensiveSpell. This, when cast, is represented by a fixture created in the FixtureManager.
- Iceball, another object that is created from a class that extends the abstract class OffensiveSpell. This, when cast, is represented by a fixture created in the FixtureManager.
- Game Mode, an umbrella term for different ways to play out a round. This includes, but is not limited to, BattleState. // An example of another game mode would be teams, but this is very open to extensions.
- BattleState, the only game mode implemented thus far, every player for himself and last man standing wins the round.
- UIState, the state the program starts in. This is a use case controller for the selection stage, where a user can choose number of player, which map, spells and future game mode can be selected.
- RoundOverState, the state the program get to in between battle rounds.

- Frames Per Second (FPS), how many times per second the program updates per second. KeyboardChaos uses 60, which is recommended by libGDX.

- Cool down, the minimal time that needs to pass between two casts of the same spell.

2. System Design

2.1. Overview

The application uses a passive MVC pattern, meaning the controller does a set amount of updates per second, where the updates will update the model and view.This differs from updating when something changes in the model. The application's controller layer is inspired by "Use Case Controllers" design.

2.1.1 The Model

The model of this application is very thin. The top class also known as KeyboardChaos, consists of 2 different classes, namely Players and Spells. The top class will have 2-4 active players depending on the information gained by the Options service. KeyBoardChaos will during it's construction these players accordingly to information received from the Option service (like the name of a player and spells).

2.1.1.1 Player

A player in the model represents a user's character that the user will control. The player class features mostly data like the name of the player, healthpoints and appropriate methods to affect these values. The class also has positions and a vector representing its position and direction facing in the game world itself.

2.1.1.2 Spells

A spell is a wide term and can be many things in the game. A spell can in short do two different things. It can affect a players health or position. What all spells have in common is that they all have a cool down. Spell is therefore an interface with a method called getCooldown(). Since there are a lot of spells that deal damage in one way or another an abstract class called OffensiveSpell was created containing information and methods that are in common for all offensive spells.

2.1.2. GameStates

libGDX requires the developer to implement an Application Listener(or an equivalent class). The Application Listener is what libGDX will run and update all the time according to the given intervals. KeyBoardChaos as an application can work very differently depending on state of the application itself. The application

should render entirely different things in for example the UI menu than in the game itself. The same can be said for getting input to the program. The mouse should be the primary input device in the menu while in the game itself the mouse is almost irrelevant and the keyboard should instead be the primary input device.

In order to solve these design issues the design pattern State was used. A State context was created in the previous mentioned Application Listener. This State context makes sure that the right state is active. In KeyBoardChaos these states are called GameStates. A GameState is an MVC package in istelf and has corresponding methods to the Application Listener, namely update() and render().

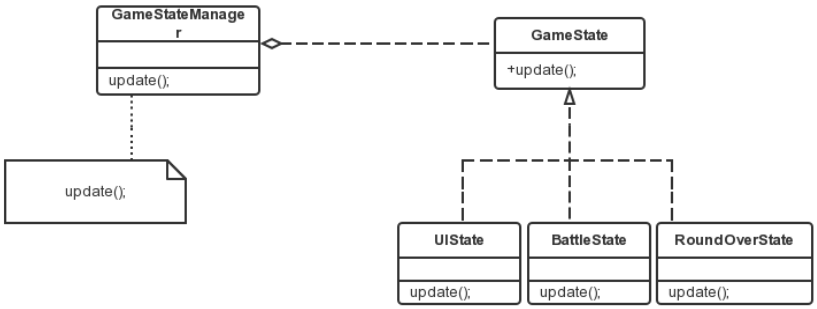


Figure 1, the state-pattern used in KeyboardChaos

2.1.3. PlayerController

The main purpose of the PlayerController class is to be the link between a player object and a body representing that player in the Box2D world. Each time the application updates itself the player and it's corresponding body will be synced. This means that if you change the position of one of them, both will get this new position. PlayerController has various booleans that are true if the controller is getting input to move in a certain direction. A vector is calculated accordingly to these booleans. The body will then receive a force that "pushes it" in the direction of that vector. The body will calculate a new position for itself. The Controller will then sync this position to the player object.

2.1.4. SpellController

A spell itself will mainly be containing various data that is relevant for the spell itself such as how much damage it will cause or for how long the spell will last after being used. To help the spell perform its use case we've constructed a SpellController class. Each version of a spell will have a unique corresponding SpellController that can support it, for an example the spell Fireball have a FireballController class. A SpellController can be seen as a controller for a single spell usage. Each time you use a spell, a SpellController is created that wraps around the spell object and takes care of that and only that usage/use case scenario. That means the next time the spell is used an entirely new SpellController will be created for that usage. The SpellController will be terminated after it has performed the use case. A SpellController contains a spell, the player who casted the spell and appropriate methods/variables to perform the spell's use case. As an

example, FireballController contains a fireball spell and a body that will represent the fireball object in the Box2D world and acts as a class that wraps them both together, another SpellController might work entirely different from this one.

2.1.5. FixtureManager

To create a box2D representation of a player and a spell, the objects are sent to the FixtureManager. The FixtureManager takes care of creating the fixtures, aswell as a safe removal. A safe removal of fixtures, is to make sure that they are removed between, and not during, updates. This is done by adding them to a list that is later looped through before the next update is being done.

2.1.6. ContactListener

The ContactListener is a class used by libGDX to determine the right course of action during a given collision between two box2D-bodies. When two bodies collide, a *Contact* is created. From this contact you can gather information about what fixtures that's in contact, and do customized actions to this sort of contact.

2.1.7. Map

The map is made up from a .tmx-file that we used the program *Tiled* to create. Using a .tmx file, the class MapBodyManager takes care of creating a body around the lava, in order for the ContactListener to be able to notice when a body representing a player is in contact with lava. See MapBodyManager reference.

2.1.8. Eventbus

The eventbus is a singleton used to switch between the game states, but could in theory be used for other things. The bus contains a list of event handlers. You can easily add and remove handlers to the list with the methods subscribe() and unsubscribe(). To notify the eventhandlers of an event, the method publish() is used.

2.1.9.SettingsService

This service is used to save player settings locally to the hard drive. The service uses objects called PlayerSettings which contains settings keybindings and spells that a player has chosen as his/hers settings. These objects are saved to .ser files.

2.1.10. Options

Options is a service that contains four different player settings objects and other relevant information regarding options such as how many active players there are. This service is used to get and set the settings for each player. Options uses the SettingsService to save the player settings objects upon exiting/closing.

2.2. Software Decomposition

2.2.1. General

- desktop-launcher, this class contains the actual main method that starts the application. This is created by doing a libGDX-project, and only takes care

of starting project, aswell as selecting the window properties, such as size, resizable etc.
- controll, contains the controller part of MVC
- model, contains the model part of MVC, the actual game logic and data
- view, the view part of the controller which handles rendering the world and the GUI.
- settingsservice, (read 2.1.9.)
- util, contains an eventbus (read 2.1.8), constant variables and DirectionVector
- main, the main class which, despite not having a main method, for all other purposes would count as the main class. This starts the application.

2.2.2. Decomposition Into Subsystems
The only subsystem is the file handling service in settingsservice package.

2.2.3. Layering
The layers are shown in figure 2 below.

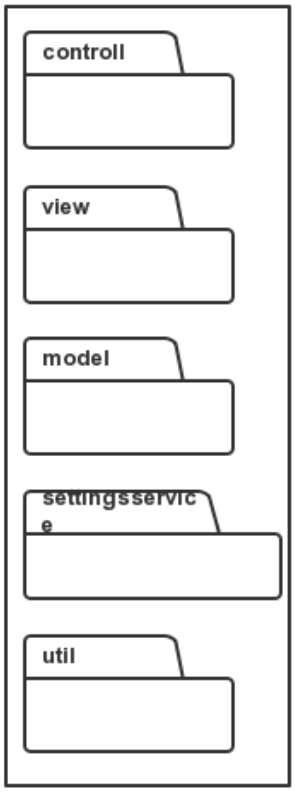


Figure 2, packages in Keyboard Chaos

2.2.4. Dependency Analysis

STAN warns on 4 occasions that circular dependency is present. All 4 includes the singleton Options and we consider this ok due to the fact that it is a service.
See figure 3

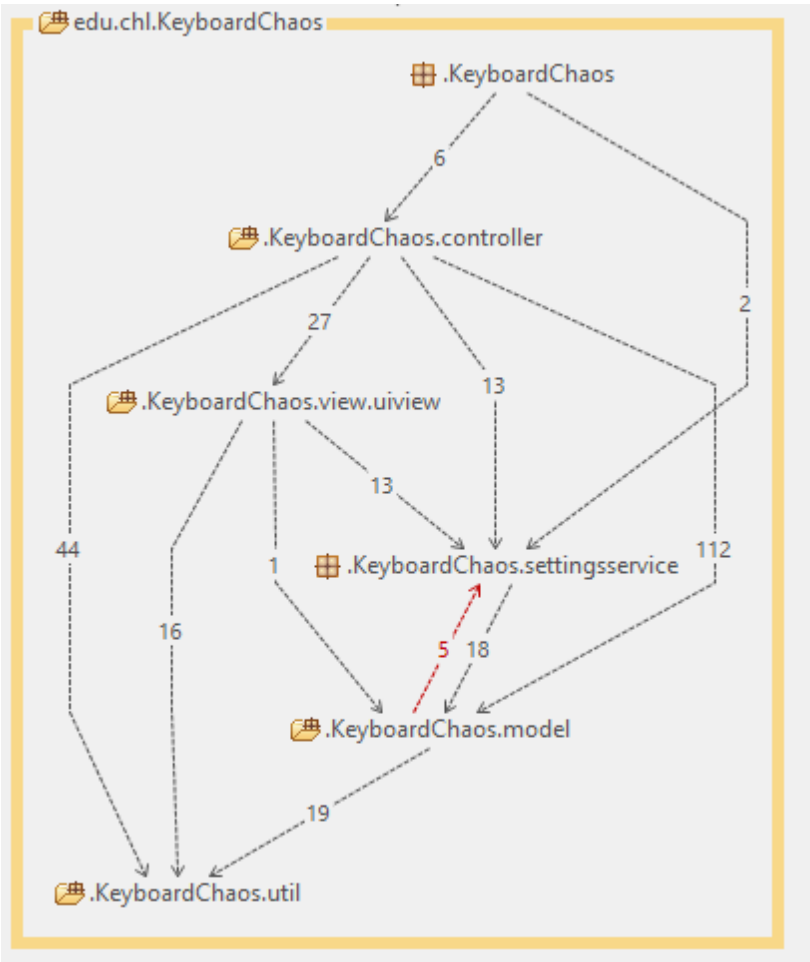


Figure 3, circular dependency between the model and the Options singleton.

2.3. Flowcharts

A pair of example how the code flows through the program on different events are shown in figure 4 and figure 5.

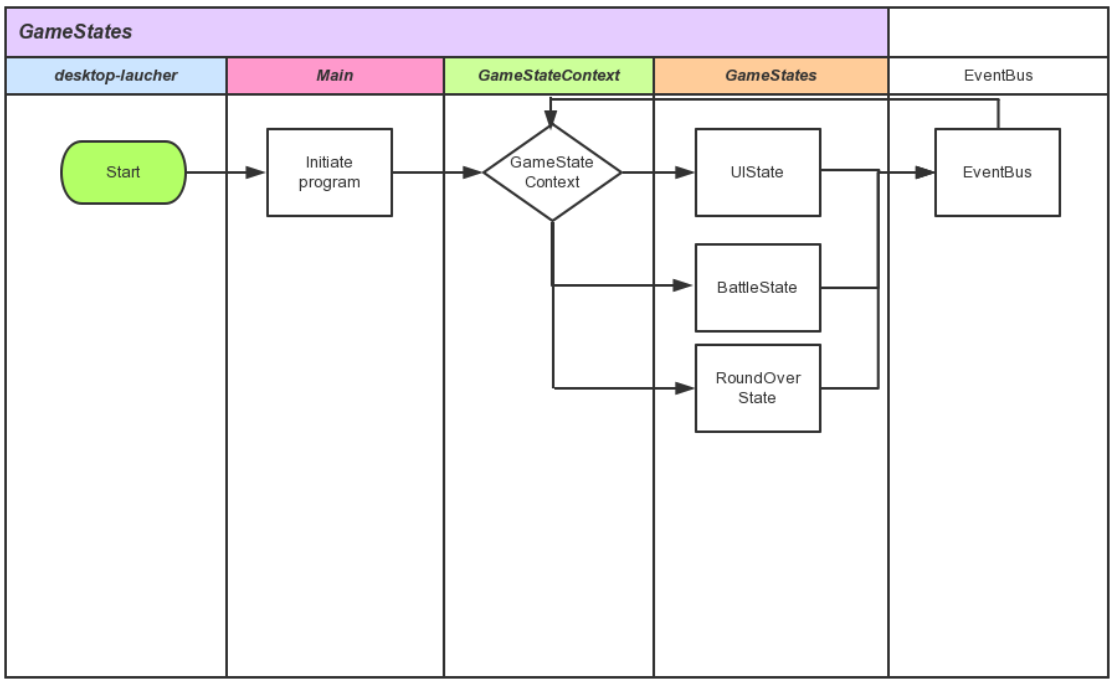


Figure 4, how the code flows when a GameState is changed.

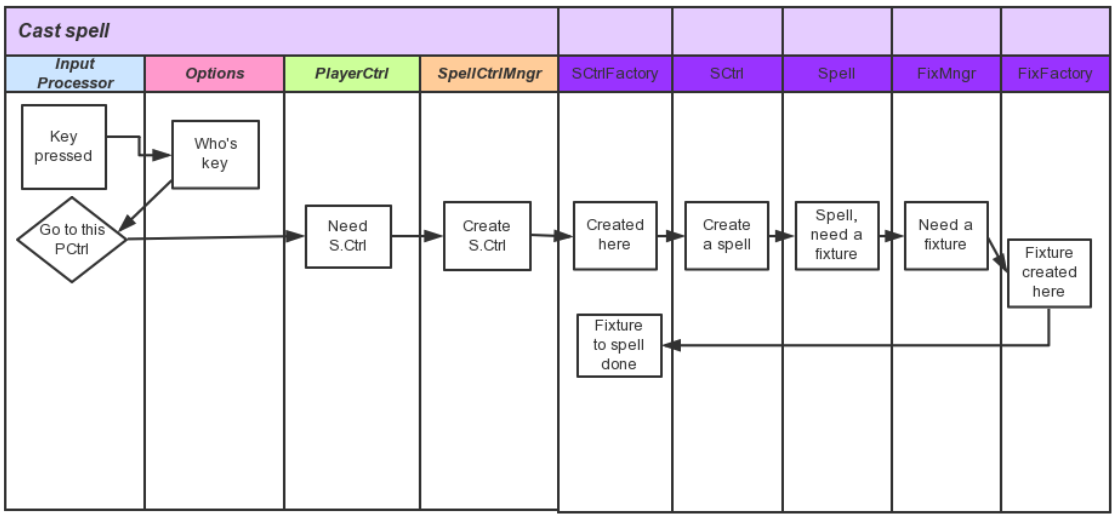


Figure 5, how the code flows when a spell is fired

2.4. Concurrency Issues

Removal of bodies and all references to said body should be done in between updates and never during a step in the world. It leads to strange and unwanted behaviour within libGDX.

2.5. Persistent Data Management

The program saves files containing information about which key bindings and spells that belongs to the correct player.

2.6. Access Control and Security

NA.

2.7. Boundary Conditions

NA. When closing down the desktop application, the last method that's called is the singleton Option's method savePreferences, that saves the current controllers each player use, aswell as what spells they have selected.

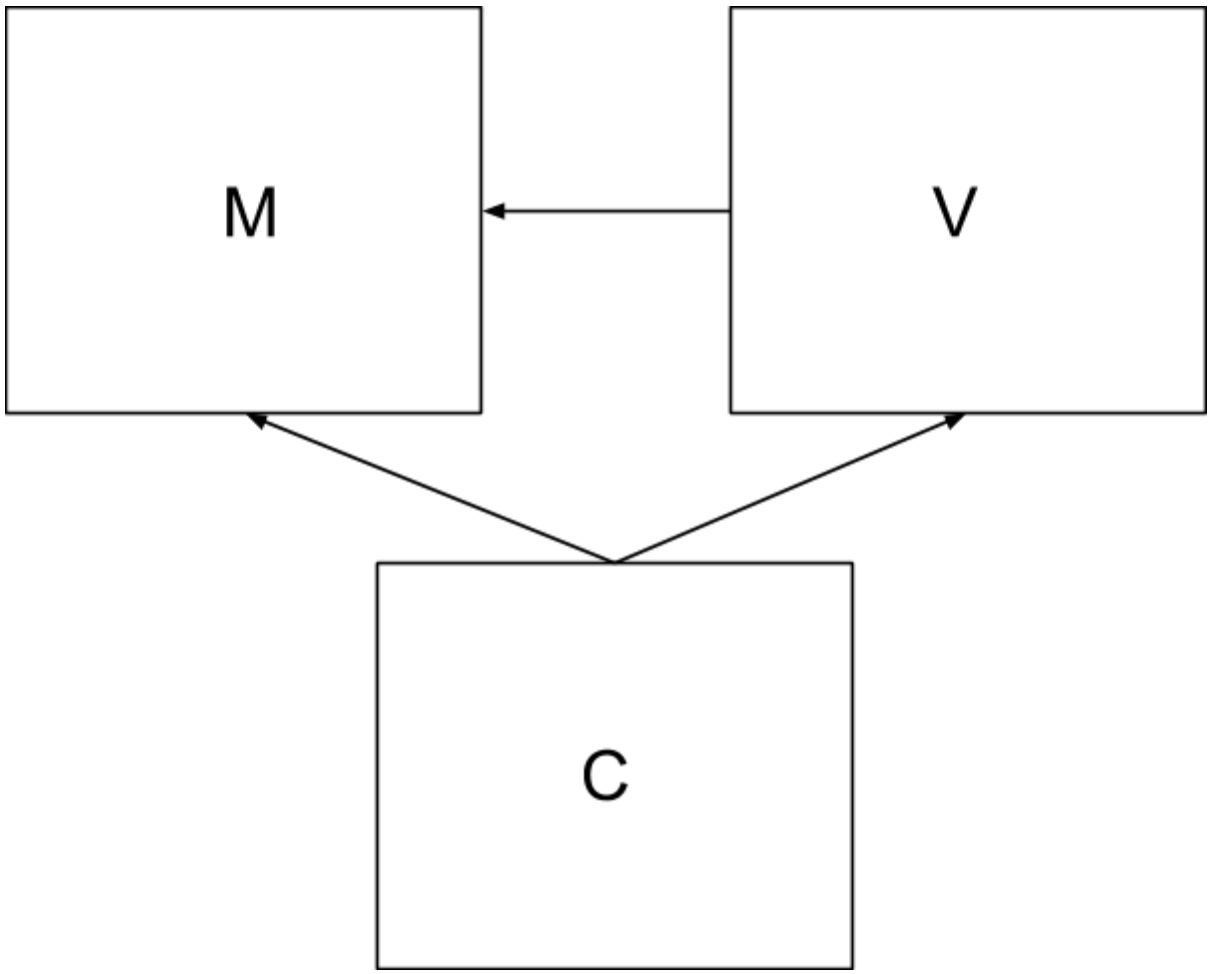
3. References

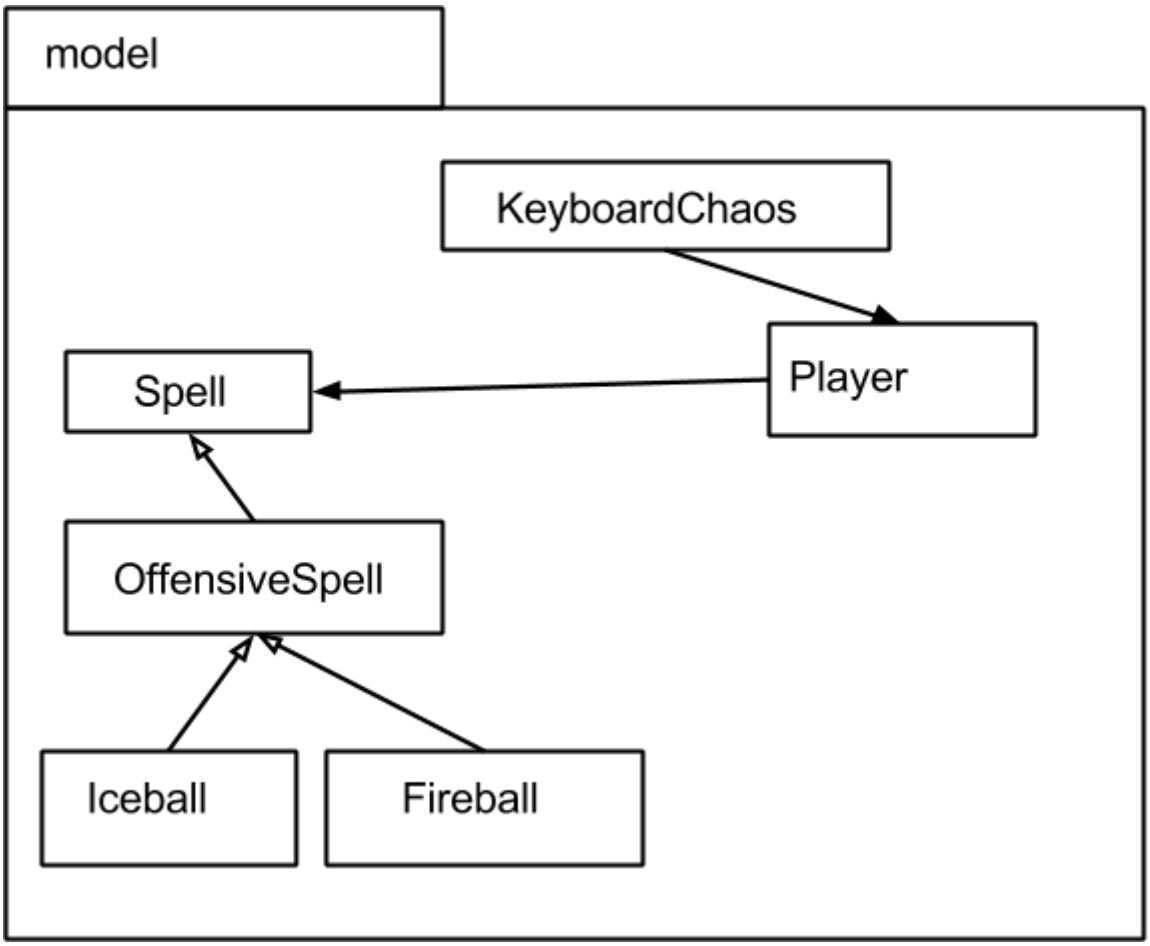
Warlockbrawl, (u. å.)
http://www.warlockbrawl.com

MapBodyManager, David S. Marquez (2013)
http://siondream.com/blog/games/populate-your-box2d-world-using-the-libgdx-maps-api/

Appendix

MVC-pattern




model
KeyboardChaos
Player
Spell
OffensiveSpell
Iceball
Fireball