

SDD

System design document for Help me this is not a joke

Version: 1.0

2017-05-25

Author: Olle Persson, Jacob Torrång, Klas Ludvigsson, Jesper Persson

1 Introduction	4
1.1 Definitions, abbreviations and synonyms:	4
2. System architecture	4
This section gives an overview of the structure of the project, followed by the subsystem decomposition section where we will take a closer look at the implementation of the subpackages and how they are implemented.	4
2.1 Overview	4
2.1.1 Game engine	5
2.1.2 Memory management	5
2.1.3 Exception handling	6
2.1.4 Encapsulation	6
2.2 Dependency analysis	6
2.3 Model structure	7
2.3.1 Facade pattern	8
2.3.2 Factory pattern	9
2.3.3 Visitor pattern	9
2.3.4 Observer pattern	9
2.3.2 Test driven development	10
3. Subsystem decomposition	11
3.1 Game Package	11
3.1.2 SaveLoad Package	11
3.2 Engine Package	11
3.2.1 Core package	11
3.2.2 Renderer package	12
3.3 Utils	12
3.3.1 Functions	12
3.3.2 Interfaces	12
3.3.3 MathI	12
3.3.4 Maybe	12
3.3.5 Tuple	13
4. Persistent data management	13
5. Access control and security	13
6. References	13

1 Introduction

Help me this is not a joke is a dungeon crawler game made to be run on a desktop. It is run on a custom made engine and a game framework which were created for this project.

The project is not a complete game. The goals that were set in the beginning of the project was not all met. The reason for this was an underestimation of the time it takes to implement a game engine and have it render a model. With this said the engine and the model are both working, but the information needed to play the game is not all rendered.

Below you will find a technical overview and documentation regarding programming design choices made during the course of this project.

1.1 Definitions, abbreviations and synonyms:

- Player = the user of the application
- NPC = non-playable character
- Tile = small squares that make up the floor, like tiles in a bathroom
- Dungeon crawler = **from wikipedia** "A dungeon crawl is a type of scenario in fantasy role-playing games in which heroes navigate a labyrinthine environment (a "dungeon"), battling various monsters, and looting any treasure they may find."
- FPS = frames per second
- Alignment = hostile and friendly are alignments
- Friendly = companion to the player
- Hostile = antagonist of the player; wants to attack player
- Item = an item that a player can interact with

2. System architecture

This section gives an overview of the structure of the project, followed by the subsystem decomposition section where we will take a closer look at the implementation of the subpackages and how they are implemented.

2.1 Overview

The system uses a passive Model-View-Controller structure. The game is a single player game with no online features, so there is only one machine involved.

The application was programming using Java, following Oracle's Java code conventions¹.

2.1.1 Game engine

For this project we decided to implement everything ourselves so we created our own lightweight game engine and framework. To communicate with the graphics card we used OpenGL, in this case wrapped for Java by the library LWJGL².

The game engine provides access to many services, for example; rendering 2d and 3d objects, loading and parsing shaders, a camera entity, object transforms and keyboard input handling.

At the base of the engine there is a game loop with sequentially handles the update timing and order of the events of the game to be updated. We decided to use a semi-fixed timestep.³ This means that whether or not the running computer manages to render the game in the minimum frequency (in our case 60 FPS) the game logic is still updated in *at least* the minimum frequency.

An implementation of a scene graph is used to update the objects to be drawn on the screen with the correct transform data (position, rotation, scale).

The game engine in its entirety can be used by itself in another game and has no dependencies tied from our game specifically. The only dependency that goes from the engine is to the math library.

2.1.2 Memory management

OpenGL is typically used in combination with C++ so all the OpenGL functions are originally written for C++. Since Java is used with OpenGL for this project the LWJGL library wrapper methods use a lot of buffer data types. Java usually handles garbage collection automatically so the programmer usually does not have to manually delete objects. This is not the case with buffer types, such as FloatBuffers. A couple of memory leaks were discovered in the application (and are now fixed). This could have been prevented if we wrapped the buffers in our own data type to manage memory easier. Although due to time restrictions we decided against it.

¹ <http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>, Accessed 2017-05-25

² <https://www.lwjgl.org/>, Accessed 2017-05-25

³ <http://gafferongames.com/game-physics/fix-your-timestep/>, Accessed 2017-05-25

2.1.3 Exception handling

When an exception is thrown in our application we always catch the specific type of exception thrown. We have created our own exceptions where third party libraries throw a regular Exception. This enables the programmer to see exactly where the exception was thrown and why it was thrown.

2.1.4 Encapsulation

The objects in our application's access levels are kept to a minimum. Getters and setters are used when accessing private variables.

2.2 Dependency analysis

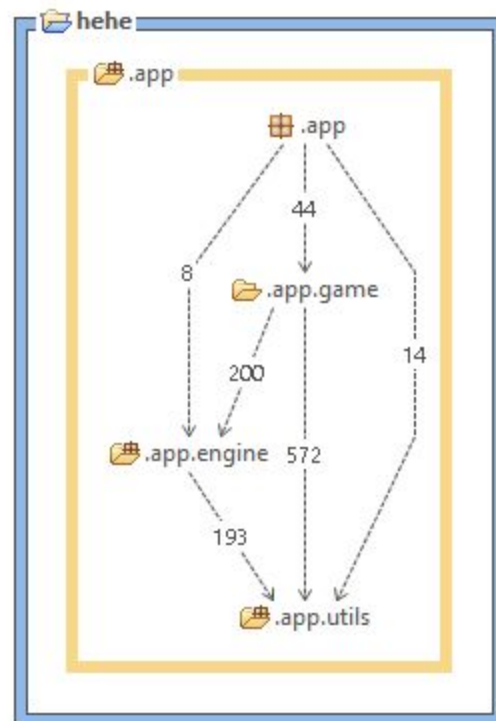


Figure 1: STAN diagram for our project

As seen, in figure 1 our project has no circular dependencies. Throughout the development process we have been using different techniques to avoid circular dependencies. We have used dependency injections⁴ in many different places. The interfaces also makes use of the facade pattern⁵.

⁴ https://en.wikipedia.org/wiki/Dependency_injection

⁵ https://en.wikipedia.org/wiki/Facade_pattern

2.3 Model structure

The model has been developed independently of the view, the engine and the controller. The idea has been that the model's functionality be tested with tests that are developed alongside it (more about this in 2.3.2). It's been developed in a way to avoid circular and unnecessary dependencies. The dependencies can be seen through the STAN analysis in Figure 2.

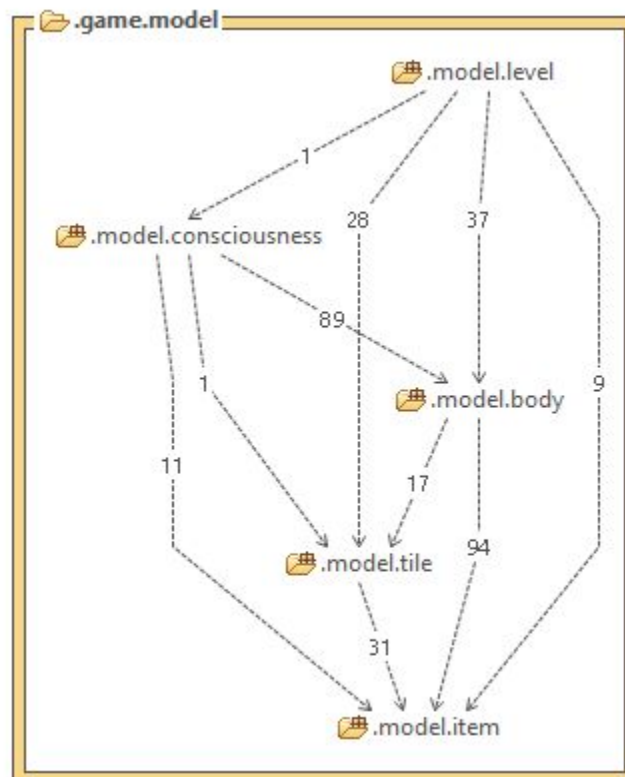


Figure 2: STAN diagram of the model package

2.3.1 Facade pattern

The model has been developed with the idea that it should be possible to swap out the concrete implementation of different parts without having too many dependency problems. As such we've been working extensively with implementing the facade pattern⁵ in the code. The packages are structured with interfaces on the top level with a subpackage containing the concrete implementation of those interfaces. STAN has been used to verify that the implementation has been correct and that there are no circular dependencies.

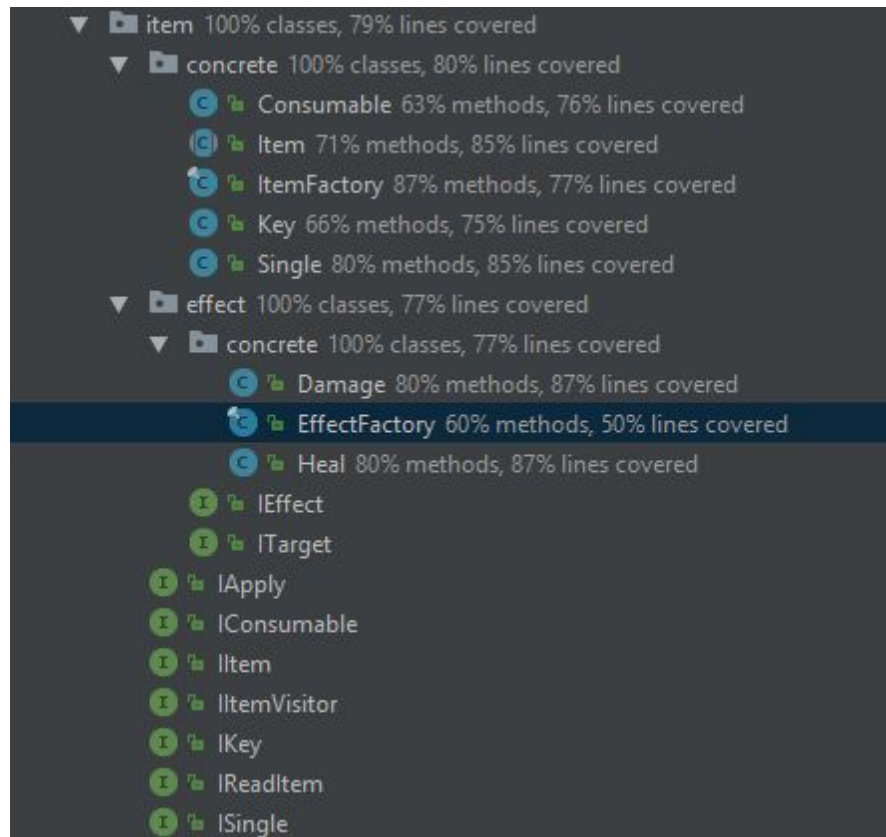


Figure 3: Example of splitting up interfaces and their concrete implementation

Not only has this made it possible to change the concrete implementation of different classes, but having interfaces for every class has made it possible for us to do mock testing⁶ (more about this in 2.3.2).

⁶ https://en.wikipedia.org/wiki/Mock_object, Accessed: 2017-05-28

2.3.2 Factory pattern

To further reduce dependencies on concrete implementations most packages in the model make use of the factory pattern⁷ to create concrete instances of classes. The reduction of dependencies have not only been moved from outside the package, but sometimes also from inside of the packages. One of the bigger benefits has been that a lot of generation code (to ease with creating objects) have been moved from inside of a class to the factory. This has made classes more cleaner but also removed those dependencies on concrete classes that they had before. The best example of this is about ~49 lines of code in the class LevelFactory that used to clutter up the class Level.

2.3.3 Visitor pattern

There are two uses of visitor patterns⁸ in the code, with the classes Item and Edge where the type. These are instances where we, the developers, thought that the types of classes would remain static during the development and provably have remained so. This decision was made early in the development and has paid off, as it's been relatively easy to add different visitors as these classes needed to be manipulated. It has also allowed the classes to be a lot cleaner in terms of their content. Instead of having general methods that don't apply to most of the classes, it is possible to isolate those applications with visitors.

2.3.4 Observer pattern

Observer patterns have been implemented in several classes, most notably body, to indicate whenever a change has happened and allow other classes to listen in and react to it. This is mostly done so that the controller can react to changes and update the view accordingly. The reason why observer patterns have been used is to avoid unnecessary and circular dependencies.

It's interesting to note that the class Consciousness is both observable and an observer. It is implemented in the way that it listens to its own instance of a Body and also passes along messages, both from itself and the Body. It's not intuitive that those would send different messages, but they do. For example, the Consciousness is not aware of when it is attacked since that is something that affects the Body, but using observers it is possible for it to be aware of when it happens. Likewise the Consciousness tests if it's able to move through an edge, and if not unlock it. To avoid having the body notifying that it was both blocked and succeeded to unlock the Consciousness itself tests for both of them and picks one to pass on.

⁷ [https://en.wikipedia.org/wiki/Factory_\(object-oriented_programming\)](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)), Accessed: 2017-05-28

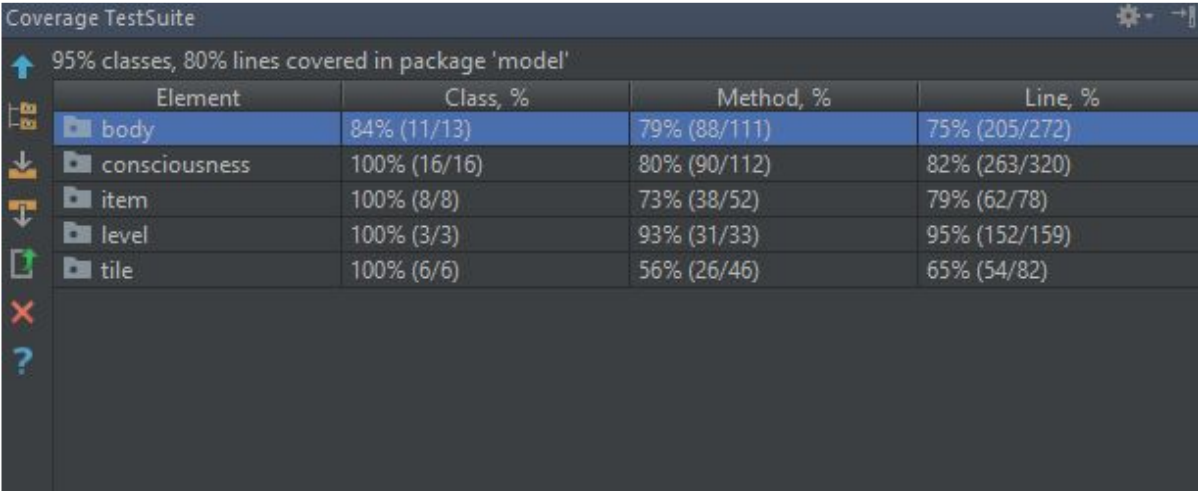
⁸ https://en.wikipedia.org/wiki/Visitor_pattern, Accessed: 2017-05-28

⁹ https://en.wikipedia.org/wiki/Observer_pattern, Accessed: 2017-05-28

2.3.2 Test driven development

During the development of the model tests have been implemented alongside methods, classes and packages. The way that it has been done is that each individual programmer has been making tests for the code that they have produced. This has been helpful at several instances where mostly minor, but sometimes major, faults have been caught very early in the development process. In contrast there were times where we, the developers, have gotten sloppy and only implemented the tests later. In one instance in particular this lack of testing resulted in lots of restructuring later on, which could've been done much earlier.

The testing has development of tests have evolved during the course of the project. The tests early in the development tested a lot of features simultaneously, but that made it harder to determine what had gone wrong if a test failed. As development went on the testing matured and it's now making use of mock objects whenever possible. This has been helped by the fact that the approach during development had been to implement interfaces for every class in the model.



The screenshot shows a 'Coverage TestSuite' window with a summary bar at the top indicating '95% classes, 80% lines covered in package 'model''. Below this is a table with four columns: 'Element', 'Class, %', 'Method, %', and 'Line, %'. The table lists five elements: 'body', 'consciousness', 'item', 'level', and 'tile'. Each element row is highlighted in blue. To the left of the table is a vertical toolbar with icons for expand, collapse, and search. Below the table, there are icons for a red 'X' and a blue '?', likely representing failed or unknown test results.

Element	Class, %	Method, %	Line, %
body	84% (11/13)	79% (88/111)	75% (205/272)
consciousness	100% (16/16)	80% (90/112)	82% (263/320)
item	100% (8/8)	73% (38/52)	79% (62/78)
level	100% (3/3)	93% (31/33)	95% (152/159)
tile	100% (6/6)	56% (26/46)	65% (54/82)

Figure 4: Code covered by tests in the model package

3. Subsystem decomposition

This section will document our different subsystems and how they are implemented. The project have a lot packages and in the interest of time this report will focus only on a couple of the interesting ones. The following section will describe how the system handles data.

The system consists of 4 main packages: game, engine, saveload and utils. In the game package we find the model, view and controller. Separate from this we have a test package for testing the model, plus some additional classes.

3.1 Game Package

The game package is split up in four distinct subpackages; model, view, controller and savegame. We choose to located these here because we wanted to separate the game specific packages and the more general packages. One might, for example, argue that the engine package is part of the view, but we wanted this located outside because it does not depend on the game in any way. Our view package, on the other hand, is created to render and handle input for *this* game, not for general use.

3.1.2 SaveLoad Package

The saveload package are, as the name suggests, used to save and load the state of the game. It uses the JAXB¹⁰ library to parse the different classes into xml code. The xml document is structured as a tree, and the root of the tree is the SaveRoot class. SaveRoot takes a ILevel, IBody(the player), and IConsciousness(the enemies) as arguments and saves them as nodes in the tree. For each class that should be able to be saved to xml, we create a wrapper which is used to only save relevant information for the class in question. Each wrapper implements an interface ILoadable<T> which consists of a <T> getObject() function, so each wrapper can recreate the original class from the wrapper data.

To interact with the package you use the SaveLoad interface which has a saveGame() and a loadGame() function, so the package could easily be replaced.

3.2 Engine Package

This package handles everything associated with the engine for example; camera calculations, transform calculations, game loop and rendering.

3.2.1 Core package

This is, as the name suggests, the engine's core. Here you will find the main game loop mentioned in *section 2.1.1*, this package also handles window and rendering context instantiation.

¹⁰ <http://www.oracle.com/technetwork/articles/javase/index-140168.html>, Accessed: 2017-05-25, Publisher: Oracle

3.2.2 Renderer package

The renderer handles all the logic that is rendering. It generates meshes, loads and binds textures and shaders. The Shader class is implemented as an abstract class that enables the programmer to create more custom shaders. The concrete shaders are implemented as singletons since there are no object specific variables in the class and they should still be able to be used in polymorphism.

3.3 Utils

This package consists of a variety of utilities that are used consistently all over the whole system.

3.3.1 Functions

The functions package contains a variety of functional interfaces which in turn are used to simplify our code by either passing anonymous functions or function references.

3.3.2 Interfaces

These are some interfaces for different functionality that are used in multiple packages. We use, for example, IObservable to implement the observer pattern in different places around the program. So instead of having an interfaces for each implementation, we put the IObservable interface in utils so it can be used freely around the project.

3.3.3 Mathl

At the start of this project our plan was to create our own linear algebra library, which we then called mathl. This proved to be a tedious task and we frequently discovered missing functionality, especially when implementing the game engine. We then decided to import an external math library, JOML¹¹, and implement the adapter pattern to convert mathl to JOML inside the engine. We choose JOML because it was created to work well with our graphics library. We still use our own library in the model, for example we use our Vector2f class to store coordinates and directions.

3.3.4 Maybe

This package is inspired by Haskell's¹² Data.Maybe¹³ and is used in the same way to handle the situations where a method wants to return a negative result. The package contains three classes, Maybe (abstract), Just and Nothing. If the Maybe contains a value, it is a Just and if it doesn't then it's a Nothing. During the development of the project quite a few null exceptions occurred as a result of several methods returning *null* when they "failed". While it isn't necessarily wrong, it isn't visible when calling a method if there is such a behaviour. Applying this package allowed increased transparency when it comes to returning negative

¹¹ <https://github.com/JOML-CI/JOML>, Accessed: 2017-05-25

¹² <https://www.haskell.org/>, Accessed: 2017-05-25

¹³ <https://hackage.haskell.org/package/base-4.9.1.0/docs/Data-Maybe.html>, Accessed: 2017-05-25

results as well as introducing methods for the Maybe class to ease handling of such situations.

The package could've been accomplished with only a single class (Maybe) that is instantiated with either a value or a null. However this approach of splitting up the classes have increased readability, intention and as a simple correction check during the development.

New Nothing() instead of Maybe(null) (more readable, and shows intention)

New Just(value) will give an exception as its instantiated (correction check, and shows intention)

Maybe.wrap(value) for values that are potentially null (shows intention)

3.3.5 Tuple

The tuple package contains two classes: Tuple2 and Tuple3. These are used to easy be able to send pairs or triplets of different types. We discovered early on in the project that this would be a very useful tool to have, and although there surely are plenty of libraries which offer this functionality, it was easy enough to make ourselves. So we did.

4. Persistent data management

As mentioned in *section 3.1.2* the system uses JAXB to save the state of the game to xml. Apart from this the system does not store any data.

5. Access control and security

Not applicable. There are no differences between users.

6. References

<http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>, Accessed 2017-05-25

<https://www.lwjgl.org/>, Accessed 2017-05-25

<http://gafferongames.com/game-physics/fix-your-timestep/>, Accessed 2017-05-25

https://en.wikipedia.org/wiki/Dependency_injection, Accessed 2017-05-25

https://en.wikipedia.org/wiki/Facade_pattern, Accessed 2017-05-25

https://en.wikipedia.org/wiki/Mock_object, Accessed: 2017-05-28

[https://en.wikipedia.org/wiki/Factory_\(object-oriented_programming\)](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)), Accessed: 2017-05-28

https://en.wikipedia.org/wiki/Visitor_pattern, Accessed: 2017-05-28

https://en.wikipedia.org/wiki/Observer_pattern, Accessed: 2017-05-28

<http://www.oracle.com/technetwork/articles/javase/index-140168.html>, Accessed: 2017-05-25, Publisher: Oracle

<https://github.com/JOML-CI/JOML>, Accessed: 2017-05-25

<https://www.haskell.org/>, Accessed: 2017-05-25

<https://hackage.haskell.org/package/base-4.9.1.0/docs/Data-Maybe.html>, Accessed: 2017-05-25

Appendix

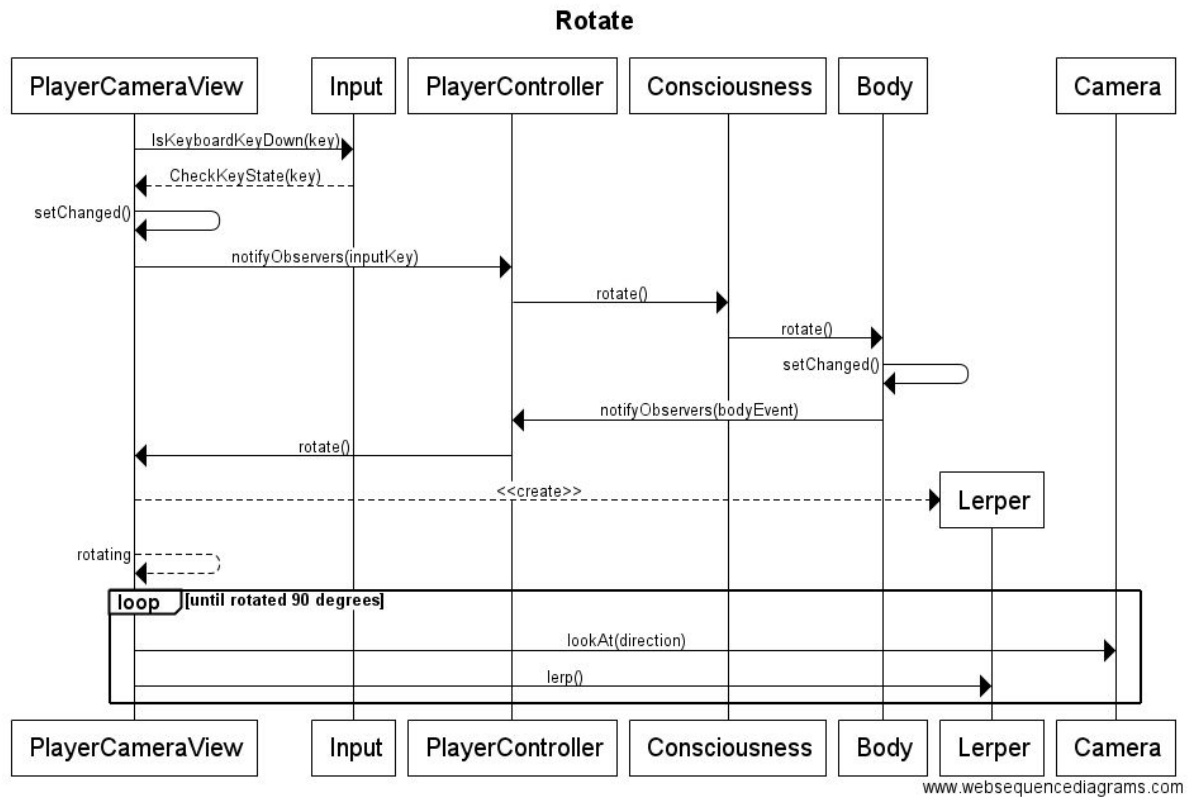


Figure x: Sequence diagram for the Rotate usecase. In the actually implementation you rotate a direction, this is simplified to work for every direction