System design document (SDD) for Achtung, die Klonen (Group 7)

Table of Contents

Version: 3

Date: 2013-05-26

Authors:
Jakob Csörgei Gustavsson, Niklas Helmertz, Lucas Persson, Joel Torstensson

This version overrides all previous versions.

1 Introduction

Achtung, die Klonen is based on a classic game from 1995 called *Achtung, die Kurve*. It is a game where multiple players compete against each other by controlling their own snake in such a way that it doesn't collide into a wall or a part of a snake's, including oneselves, body.

1.1 Design goals

We set out to make the application modular, testable and easy to understand.

1.2 Definitions, acronyms and abbreviations

Achtung, die Kurve – an online multiplayer game developed in 1995 wherein multiple users using the same computer competes by controlling its snake so that it doesn't collide into something. At start, the snakes are only a point, but as the head moves a trail, referred to as the body, is left behind in every point that the snake visits.

Power-up – an object that can be picked up by a player and causes an effect on something in the game. For example a speed boost which makes the user move at a faster pace than normal.

Lightweight Java Game Library (LWJGL) – a game library for Java that amongst other things gives developers the ability to utilize the GPU using OpenGL.

Open graphics Library (OpenGL) – a very common API for utilizing the GPU that has been implemented in most modern graphics hardware.

Graphics processing unit (GPU) – a hardware circuit specialized for graphics rendering.

Nifty-GUI – a library for easily constructing menus and graphical user interfaces for Java games.

2. System design

The application is designed with a type of MVC.

2.1 Overview

2.1.1 The model functionality

The most top-level model class is Game, which only handles what is not related to a single round, such as before the game starts or when the game is over. The next primary class is Round, which holds much of the state of the visible game world. It handles much of the game's logic, such as point distribution. Some responsibility of the Round is delegated to a class called Map, which for instance stores the size of the world amongst other things.

The Player class represents the separate users which play a game together. It stores information that does not change in between rounds, such as points and color.

The more complex physical state of a player in a round is stored by a Body class, which a Player has a reference to. At every new round a new Body is set for each of the players. The Body class stores important variables like speed and rotation. This class then delegates some responsibility to the classes Head and BodySegment. Head simply acts as a data structure for the current position of the Body, while BodySegment represents a part of the visible Body. In other words, the Body's segments represent the history of the Body, a trail left after the head.

2.1.2 Power-up system

One important goal was to design a power-up system in such a way that new power-ups could easily and seamlessly be added. A power-up is an effect that once gets applied to a Body or a Round, which will after a set time be removed. The way that it has been implemented is not perfect, but it is modular enough, and most importantly easy to use. To add a new power-up one needs simply to implement an interface with six methods (PowerUpEffect), and add the new power-up to a list in a factory class.

The two primary methods define what happens when the effect is applied and removed. Both of them take as argument an instance of a model object that the effect is intended to be applied to, such as a Body object. The apply method applies the effect by altering one or more properties of the received object. In the case of the Body, a power-up can for example double the width of the Body. The remove method should if possible represent an inverse to the operation performed in the apply method, such as halving the width if it had been doubled before.

If an inverse can be found as shown in the Body example, it is possible that multiple effects of the same type can stack on one another, which for example means the body can become be four times as wide if the effect is applied twice. The power-up signals that this is possible by returning true in its isStackable method, which is one of the five interface methods. If a power-up cannot stack and a Body already has one of that type, the timer is instead reset.

The three remaining methods are getDuration, getName and getWeight. The duration defines how long the effects lasts, the name is used for connecting the power-up to an image and the weight is a floating point value that determine how common the effect is, the higher value, the more likely it is that it will appear.

There are several different power-up related classes with different responsibilities. The PowerUpEffect interface is what defines what the power-up actually does when it is activated. The class PowerUpEntity, which has a reference to a PowerUpEffect and represents an object in the game world, with position and size. A third class is called PowerUp and represents a power-up in an activated state, it too has a reference to a PowerUpEffect. It is primarily responsible of keeping track of when the effect should be removed from the object which the effect has been applied to.

2.1.3 Graphics

The application relies on GPU-accelerated graphics by using OpenGL. The primary graphics library used is LWJGL (Lightweight Java Game Library), which has much of the functionality that was needed.

The primary reasons for choosing an OpenGL-based library was that it is well suited for the type of game we are making, and that there are good, high performing, java implementations. Additionally we wanted to gain some experience with using OpenGL. LWJGL in specific was chosen because it is one of the most used and best documented Java implementations. One issue is that it is not very object-oriented, the functionality is mostly accessed using global static methods. We choose it since we were confident that we could do the necessary abstraction for an object-oriented MVC-application ourselves.

A more high-level object-oriented library that builds upon LWJGL is Slick2D. One of the reasons why we didn’t choose to use it, which in other regards is very good for constructing 2D Java games, is because you more locked in to the way Slick2D handles rendering and game logic. We wanted the ability to structure our application without running into problems with how things must be done with Slick2D. We didn't want to get in a situation where we needed to work around the limitations of the library.

The way LWJGL graphics are integrated is by making a rendering service, defined by interfaces, which has methods for drawing various shapes, as well as for drawing images. In addition to that, it handles things like scaling of the window and automatically adjusts the size of what is shown, without aspect ratio and scaling needing to be taken into consideration by most of the view classes closer to the model.

2.1.4 Event handling
There are a fair amount of events constantly being passed around the system. Primarily, model objects are the ones sending events to view objects listening to it, who in turn respond with appropriate actions when something needs to happen. Because the model changes at a constant rate, the view objects do not only "repaint" themselves at events, but rather as often as possible.

The keyboard and mouse inputs are also provided by LWJGL, and there is a separate service for input which is structured in much the same way as the rendering service. Using the input service is done by objects being added as listeners to it, and events being sent on user input.

2.1.5 Audio
The reading and playback of audio files has been done using the library *Slick-Util*, a smaller part of the larger library *Slick2D*. Slick-Util consists only of smaller individual components that could easily be integrated independently of the application's structure. There is also support for image and text rendering in the library, but only the audio components are used in Achtung, die Klonen.

There is a service for sound, defined by an interface, that works by being a listener to model objects, similar to the view classes, and it responds to certain events by playing a sound. There are also methods called from the outside which toggles music playback.

2.1.6 The game loop
As in many games there is a game loop in the background that handles when things are to happen. The application basically consists of one thread that passes through the game loop until the game is over. The goal with the loop was to make it so the updates of the game logic were completely independant of the rate the rendering could be done.

The choice made was to use a loop with ”Constant Game Speed independent of Variable FPS”, which means that there will be an even rate of game updates, independent of render rate (Koonsolo, 2009). To achieve smooth rendering, you then need to not only draw the model objects where they actually are, but someplace in between where they were and will be, a form of interpolation (Koonsolo, 2009).

The way the calculation for where an object is drawn, used in Achtung, die Klonen, can be called ”forward extrapolation”. That means that you estimate, when rendering, where an object should be at the exact moment, without considering exceptional events like collision (Diffenderfer, 2013). On the exceptional events, like collision, objects can be drawn too far ahead of what's actually allowed. But the effect should be hardly noticeable with a somewhat high rate of game updates (Koonsolo, 2009).

Because of issues in Java, no sleep is done regularly in the game loop. The way it could be done, using Thread.sleep(), causes problems because of its irregularity and unreliability, that can cause some stuttering. We chose to instead only do sleeping in thread when the game is out of focus, when those issues doesn't really matter.

2.1.7 Menu system

The menu system is done by using a library called Nifty-GUI, which comes with common menu elements and integration with the graphics library used. We chose to use a library for the menu system, bearing in mind how much time and effort would be required to make a proper menu system, when it only is but a fairly small part of the application.

The menus are defined by XML files with a simple structure using layers and panels, which for example contains elements like buttons and labels. Having them be XML-based allows for clean and simple nesting of elements, which is common in menus. This in contrast to if it were to be implemented in Java code.

There are special controllers intended to be used with a Nifty menu. The interaction between the view and controller can be done in several ways, either by the controller registering methods to be supplied events (using annotations), or by defining controller methods to be called on "interaction" in the XML file. The latter way is how it has primarily been done in the application.

2.2 Software decomposition

2.2.1 General

The package structure can be seen in figure 1. These are all the applications packages, and their responsibility:

- Model; contains all the model functionality, it represents the state of the game. The model part of MVC. No model object knows anything of the application's other packages.
- View; contains classes that interprets the model objects and visualizes it. The view part of MVC. The classes use the methods of the service in the rendering package, which are fairly primitive and not adjusted for the model classes.
- Rendering; contains classes for the rendering service which used by and more technical than the classes in the view package. The view part of MVC.
- Controller; contains most of the controller classes. The controller part of MVC. The classes that keeps the application alive and causes changes in the model.

- Input; contains classes relating to the key and mouse input service. The controller part of MVC. It's responsible for detecting and forwarding user input.
- Sound; classes relating to the sound service.
- Menu; classes related to the Nifty-GUI system. The controller part of MVC. The view part of the Nifty-GUI system is defined by XML files that represents the menus.

2.2.2 Decomposition into subsystems

We have separate systems for handling input and rendering. Mainly the other packages and systems only need to interact with an interface representing the service. This is done so that we can easily switch input- and/or rendering-service simply by implementing a new class of a new library of some sort.

In addition to this, we have adopted an MVC-like structure with models, views and controllers. They are all in their own subsystems/packages. As per convention, views only know about models, controllers know about both, and models know of none of them, only of other models.

2.2.3 Layering

Can be seen in figure 4, see Appendix. The top class, being Game, is at the top of the diagram.

2.2.4 Dependency analysis

Can be seen in figure 1. What might be notable is that there currently are few references between the view and the input package. That is because the input handling is currently global and not connected to a particular view.

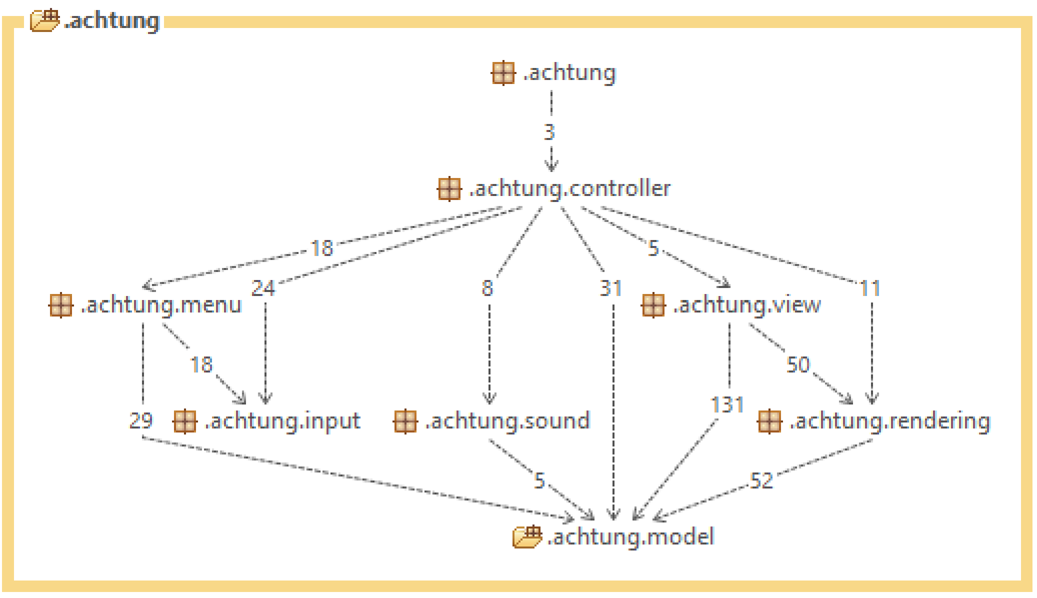

Figure 1: Package overview

2.3 Concurrency issues
The design of the application involves only one thread. There should be no concurrency issues.

2.4 Persistent data management
Some options are saved to file, such as game settings (player speed etc.) and whether sound is enabled or not. To do this the class Settings is used. It is a singleton which loads the settings from a file when first invoked. To retrieve or set settings, simple getters and setters are used. For the settings to be stored to the configuration file ("achtung.conf") the method "save()" has to be called. The file formatting and saving is done through java.util.Properties.

2.5 Access control and security
NA

2.6 Boundary conditions
The application is intended to be started by scripts for the current platform. Exiting the application can be done from the menu inside the application, but also as normal desktop applications.

3 References
Diffenderfer, Philip (2013), *Generic Game Loop*, http://www.gameprogblog.com/generic-game-loop/

Koonsolo (2009), *deWiTTERS Game Loop*, http://www.koonsolo.com/news/dewitters-gameloop/

APPENDIX

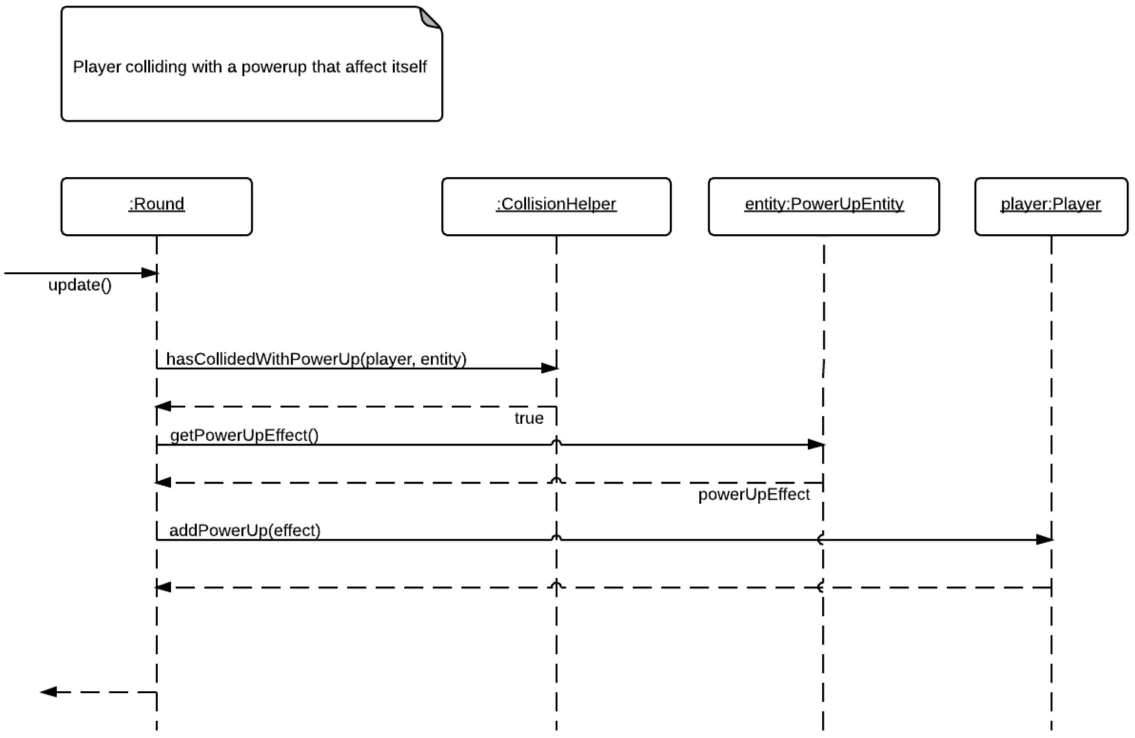

Figure 2: Sequence diagram for colliding with a power-up that affects self.

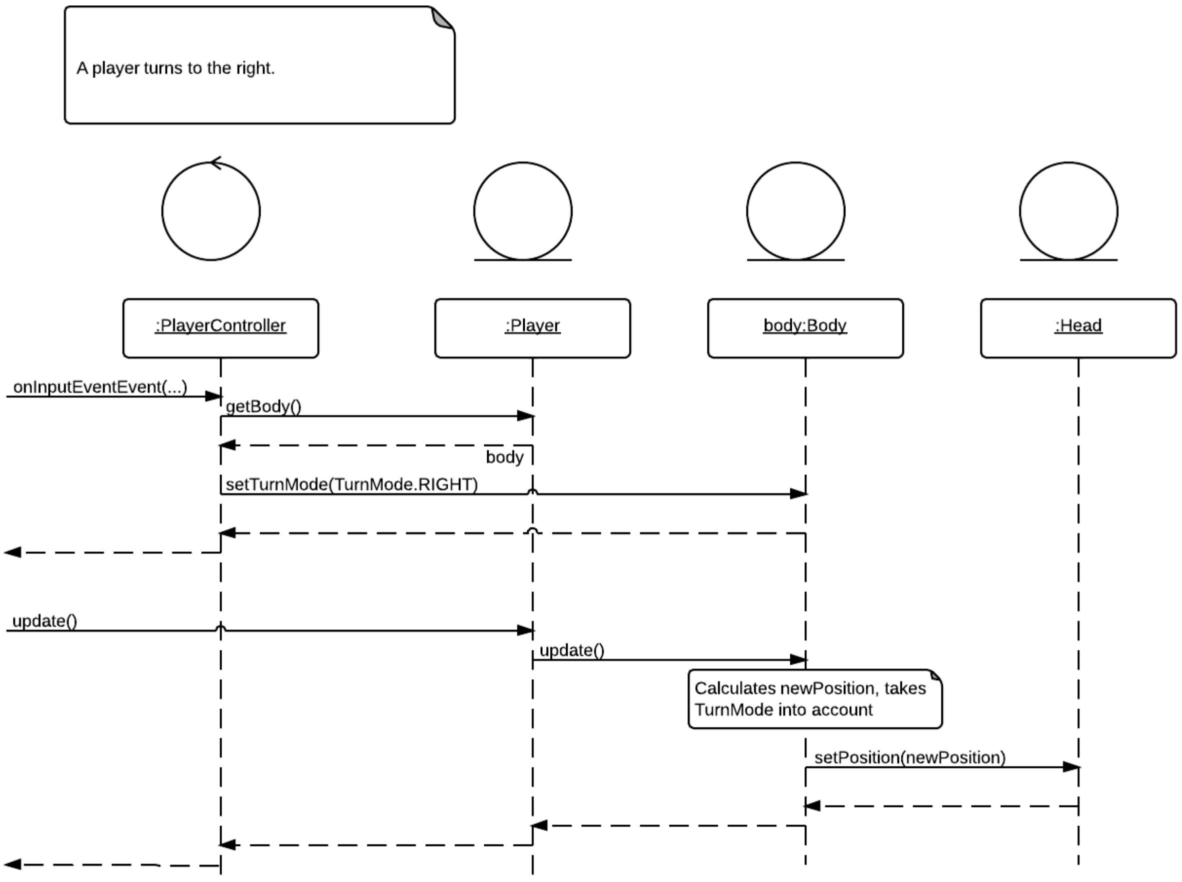

Figure 3: Sequence diagram for a player turning.

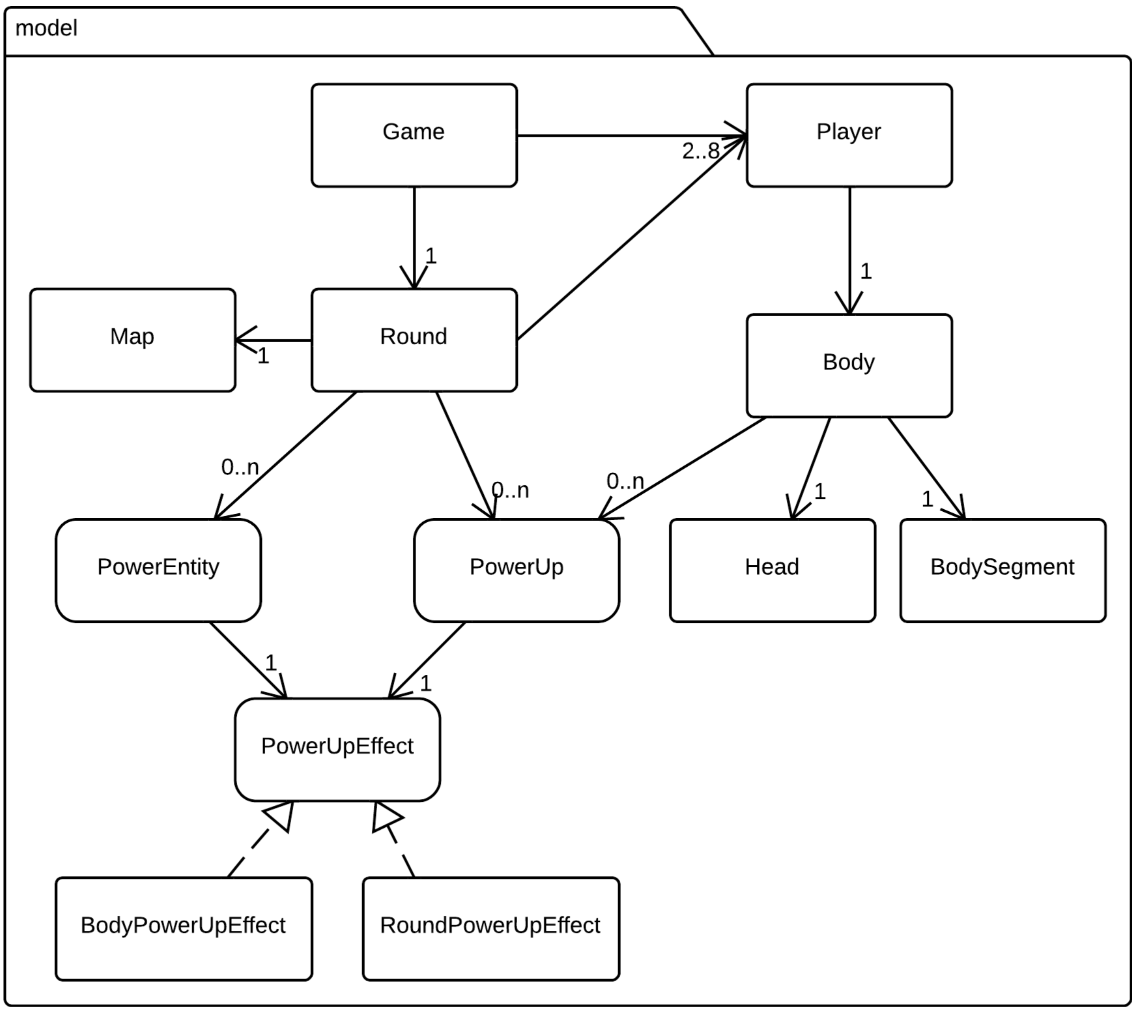

Figure 4: Simplified class diagram for model.