System design document for

Sunset Ninjas

Version: 2.5 – Final version
Date: 2016-05-27
Author: Lina Blomkvist, Andreas Carlsson, Felix Engelbrektsson, Robin Lilius-Lundmark

This version overrides all previous versions.

1 Introduction

This document will describe how the application was designed, which design decisions we had to take and why we choose specific ways of solving design issues.

1.1 Design goals

Our main design goal with the project was to make a desktop Java application with MVC design pattern. We've tried to keep the code clean and easy to read. It should be easy to understand everything without the need of extensive documentation and comments in the code.

We also made a serious effort to keep the amount of external dependencies to a minimum. More about that in section 2.2.4.

1.2 Definitions, acronyms and abbreviations

MVC - Model View Controller. A software architectural pattern. Separates the logic from the presentation.

Game loop - A continuous loop that runs for as long as the application runs. This makes sure the time is pushed forward. Without a game loop the program would terminate after the first frame. This is a common way of programming an application that should response to things in time. You could compare this with event-driven programming where the system responds to events. The events in this case is the "ticks" och the system clock.

AI - Artificial Intelligence. What it sounds like, simulating intelligence. In our case the AI isn't very intelligent however but it's good enough for our purposes.

2 System design

2.1 Overview

We've built the application around the Model-View-Controller design principle. This is a well-known software architecture design principle for separating the concerns of business logic from its presentation. Classes are roughly divided into Models, Views or Controllers. In our case we also have some classes that lies outside of the scope of these, for example utilities and asset containers.

2.1.1 Application flow

The application is started from a class named Main, placed in the top level package edu.chalmers.vaporwave. This class extends the JavaFX Application-class and overrides the start()-method. In the start method we create a Group-class and a Scene-class.

We then set up the stage to show the scene and pass in our Group called root. This root-object is then passed on the to application which adds content to it. The physical game window properties is also set up here with a width, height, title and so on.

After the window is set up the ListenerController is initialized. This is a singleton used throughout the game and handles all input from the user, both from keyboard and optional gamecontrollers.

Now everything is ready for starting to game which is done by creating a new MainController-object and passing the root-object to it.

The first thing that happens in MainController is that loader is set up and starts to load in all the assets needed in the game. This happens in a thread separate from our JavaFX thread. Meanwhile the loaderView shows how far we've come in our loading process. Total time for loading is somewhere around four to five seconds on a normal computer.

When all the loading is complete, the loading-thread terminated and the methods for initializing the game loop and application is run. For the game loop a new AnimationTimer is created as an anonymous object. The code in the handle-method here is run approximately 60 times every second. Depending on if the user is in game mode or menu mode the updates from the game loop method is sent either to GameController or MenuController.

We chose to put these controllers in different classes since they work so differently. The MenuController has responsibility for everything happening in the menu and GameController for all game related code. However both of these implements the ContentController interface with the timerUpdate() method. In the MainController we use the state pattern to activate either the MenuController or GameController.

From here on all the game logic takes place inside the respective controller. The code inside the timer update is run once for every iteration in the game loop. There is quite a lot of code being run for every iteration but since it's so lightweight there is no trouble keeping up with the desired update frequency of 60 frames per second.

2.1.2 Asset Containers

To gather all external file dependencies and the administration of them in one place, the package assetcontainer was created, with the central class Container. In it, every image, sound and file is loaded and cached into the application. In addition, it also sets up every Sprite, font and character info, to be fetched later on. All this to ease up the rest of the application.

The Container class works somewhat like a factory class. It contains every other resource container, like ImageContainer or SoundContainer, and it is only the Container that is public; all other container classes is package private. In this way, Container delegates every call for resources, such as Container.getImage(ImageID). Every resource also have an respective ID enum class, to prevent illegal calls.

When loading everything in every container class, each container counts every task done (e.i resource loaded) up to a constant total. It is not ideal to have the total amount hard-coded, but this solution works well in the extent we need it.

2.1.3 XML Loader

XML, Extensible Markup Language, is markup language where you define your own set of tags to store information in an organized way.

To avoid a situation where the data is stored in classes we decided to store as much as possible in XML-files. These are then parsed when the game is loaded. The java.io library is used to load the file into the application and then javax.xml.parsers is used to do the parsing.

2.1.4 Map Loader

Map files are a little special and does not fit very well for storing in XML-files. We had the ambition to make these human readable and editable. We landed in storing these in text-files and building an own class for parsing these, MapFileReader.

A map file consists of predefined characters that maps to a MapObject enum. For example a X maps to MapObject.INDESTRUCTIBLE_WALL. When the file is loaded we first make sure that the file is correctly formatted. Then we decode the file och set create a two-dimensional array consisting of MapObjects. This is used as indata to the ArenaMap constructor.

2.1.5 Utilities

To handle common things like cloning arrays, storing pairs and converting values we introduced a Utility-package. This package also stores most of the enums in the project and constants..

Here follow a brief overview of some of the most used classes in the package.

ClonerUtility
Contains only static methods for cloning different kinds of lists, collections and maps.

Constants
Setting game constants. Since many of these values is used in lots of different places in the code it makes sense to collect them all in one class. It's also a design principle to avoid using "magic numbers" inside the code.

LongValue
Wrapper for a long since we need to send in a mutable object and not a primitive to the AnimationTimer. We've made a conscious decision to expose this class inner representation.

Pair
Generic class to simulate tuples in Java.

SoundPlayer
Uses JavaFX MediaPlayer-class to play sounds. For each sound that plays, a new thread is run.

Util
A classic utility class with only static methods that handles all conversions between grid and canvas positions, image rescaling, key-to-direction mapping and a lot of other small things.

2.1.6 Sprites

A sprite in computer graphics refers to a two-dimensional pre-rendered picture, which then can be used to build a graphic representation of an application. They are often used specifically in games, where lots of moving graphics needs to be rendered simultaneously.

Our solution consists of a class Sprite, which contains an Image-reference and several attributes that connects to the graphic, such as its scale when rendering on screen, screen position and position offset values. A Sprite can also make use of a spritesheet, which is a big image with many different sprites, from which it cuts its specific representation.

A natural subclass to Sprite is AnimatedSprite, which extends Sprite with the functionality to display a series of pictures after each other to create animation. This includes attributes such as how many frames the animation consists of, how long each frame should be shown on screen and whether to loop the animation when it ends, and in that case, how many loops, etcetera, etcetera.

All our game objects has a sprite representation in every respective view, and every frame the view is called to update its graphics with a specific timestamp that originates from the main AnimationTimer. Every Sprite has a method render() with timestamp parameters, and if it is a AnimatedSprite, the current frame is calculated from this timestamp.

2.1.7 AI

Artificial intelligence, or an algorithm for how the computer controlled enemies and players are supposed to move. AIHeuristics is a class that evaluates the board depending on where enemies, players, powerups and walls are in its current state. Thereafter it puts an appropriate value assigned to each tile of the gameboard. The enemy and/or CPU then chooses its next move according to which of the closest tiles has the highest value through an elimination algorithm. It also takes bombs that are about to explode and their blasts into consideration, and prevents the bots from walking there unless they really have to. To avoid the bots from walking backwards and forward we have prevented such a pattern by blocking the opposite of the previous direction. This however isn't fully optimized yet, so it still do happen occasionally.

2.1.8 DoubleTile

The arena is built of a grid of StaticTile objects, one for each grid position. However, it is sometimes necessary to keep more than one StaticTile in a grid position e.g several blast tiles or a powerup tile under a destroyable wall tile. Instead of changing from a two-dimensional array to a more complex structure, it was decided to use a recursive solution.

DoubleTile extends StaticTile and is a container of sorts, for other StaticTiles, using a version of decorator pattern. Every DoubleTile can hold an upper tile and a lower tile, so when rendering it is clear which is supposed to be placed over which. A DoubleTile can also hold other DoubleTiles, since DoubleTile extends StaticTile. In this way, DoubleTile enables an infinite number of StaticTiles in one grid position, in the arena.

One of the neater parts of DoubleTile is how it recursively can search for a StaticTile of a specific class, and if needed return it. For example, a DoubleTile can check both it's upper and lower tiles, and if any of them is a DoubleTile, it recursively calls the same search method in next DoubleTime.

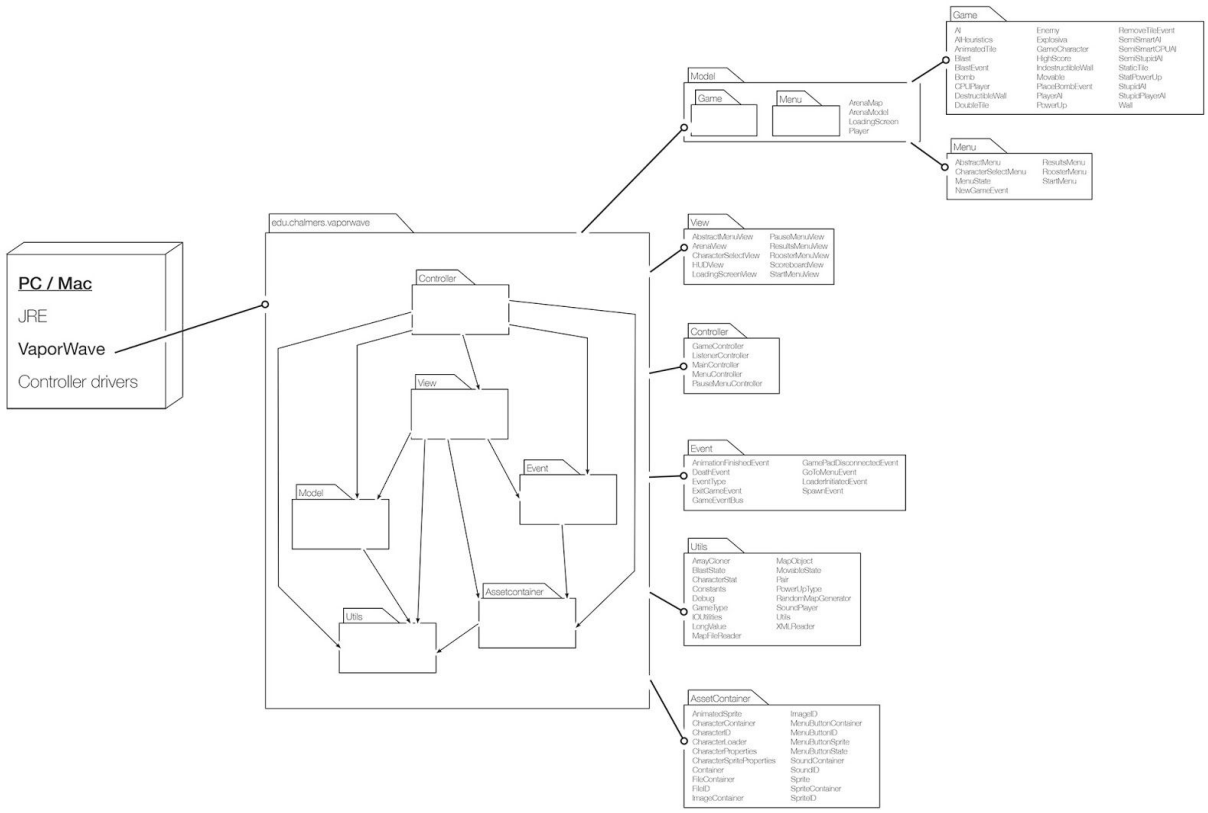
2.1.9 GameEventBus

It was early established that this application needed some kind of eventbus-solution. After trying to build our own, it was decided to use Google's java-eventbus. The reason to this lies in the simplicity of it: a target class need only to subscribe to the eventbus class, and a posting method need only to grab the eventbus class and send an event object to it.

The only setback with the EventBus is that you need to pass around the eventbus instance. A workaround for this was to simply create a singleton subclass to Google's EventBus. In our case we created the GameEventBus class, and call for it from wherever in the application structure it was needed. This worked very smooth, although it introduced some cyclic dependencies to and from the GameEventBus.

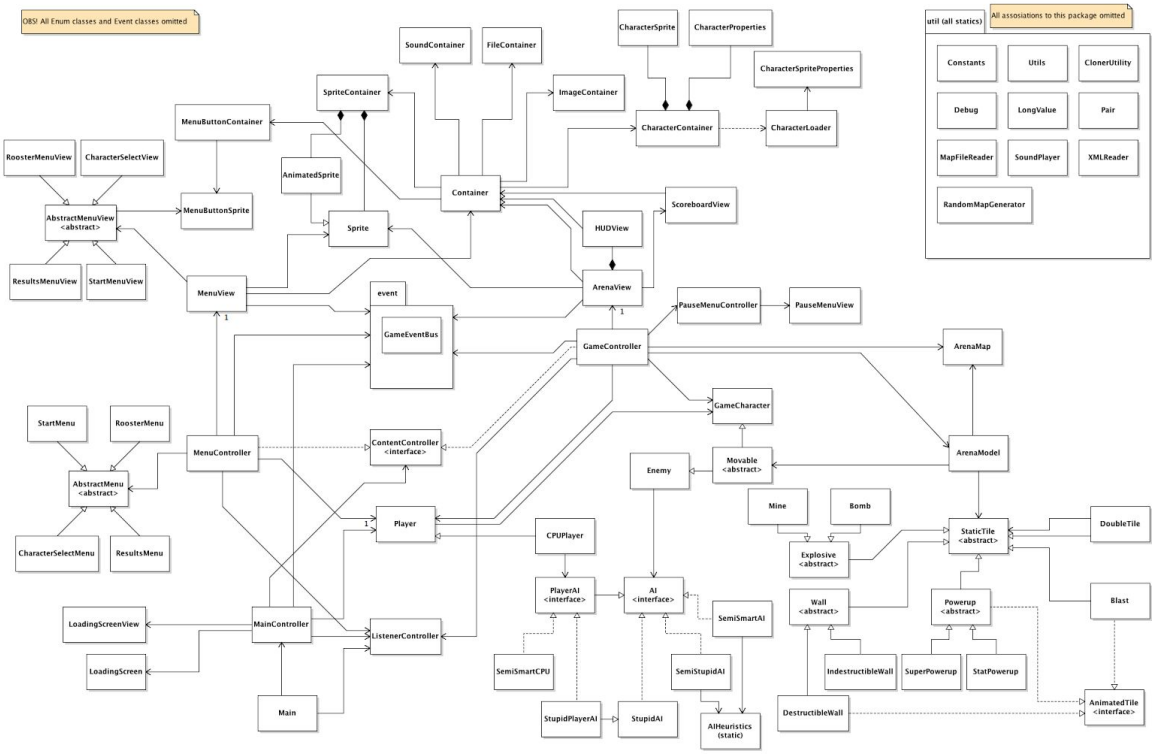
2.2 Software decomposition

See attached pdf: **Software_Decomposition_group15.pdf**



2.2.1 General

See attached image: **UML_group15.png**



2.2.2 Decomposition into subsystems

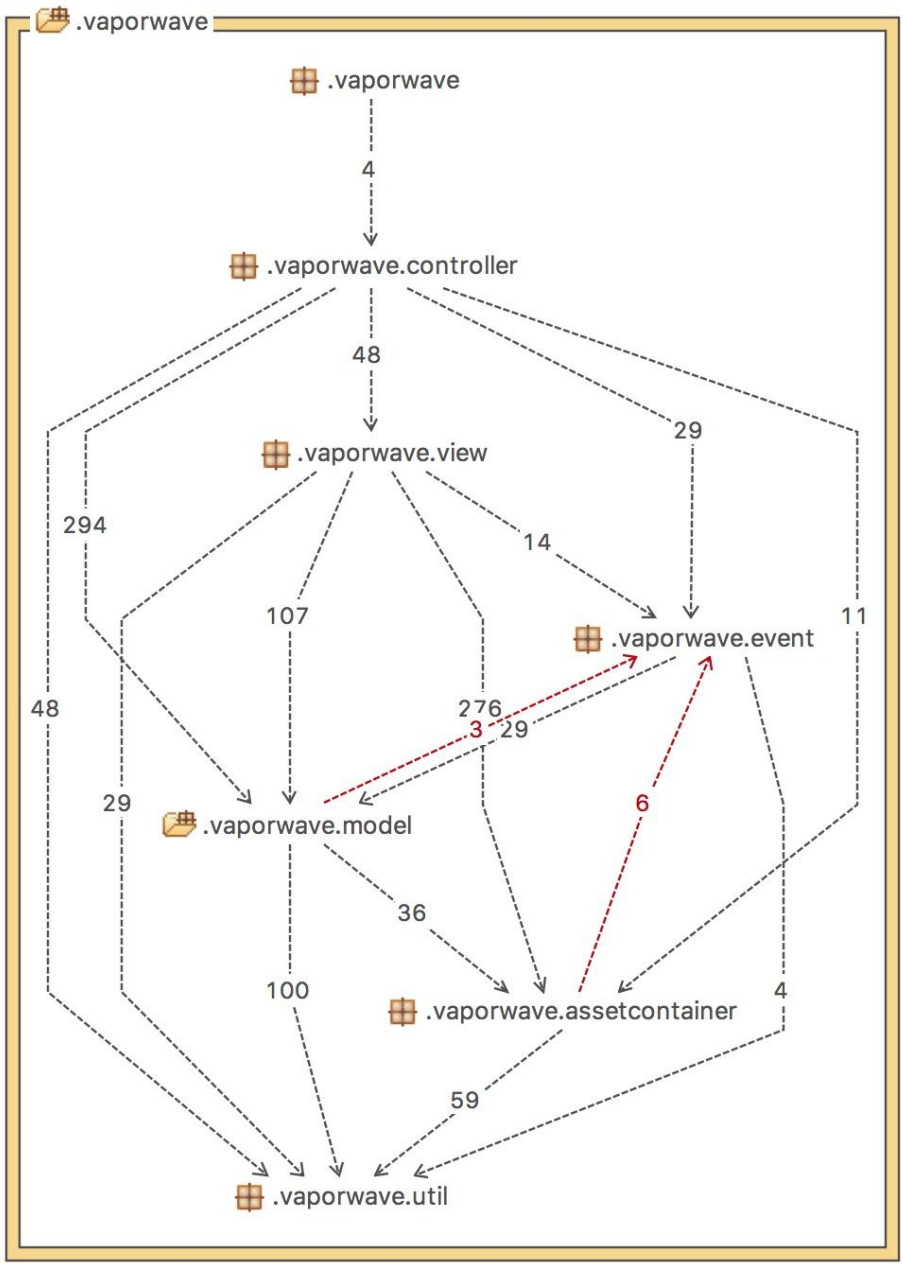
We do have a couple subsystems in our application. These handle things that aren't directly connected to the game but nevertheless needed to make the application work. The loading of assets from files to the game is probably the most significant. More about this can be read about in section 2.1.2 - Asset Containers.

Another subsystem is the XML Loader which reads data from an XML-file and populate the objects in the game with the correct data.

2.2.2 Dependencies to external libraries

Throughout the project we've been trying to keep the amount dependencies to a minimum. This is partly because we like to understand the project to the largest possible extent. We think you gain a deeper understanding of the project if you write as much code as possible in the group. Also it's partly because we like the concept of a minimal amount of dependencies.

2.2.4 Dependency analysis



2.3 Concurrency issues

When the applications starts up, a JavaFX-thread is initiated and run. The JavaFX-library has a built in animation timer to handle updates.

When our application grew, it became apparent that a loading section was needed, to show the user how long s/he would have to wait for the game to start. We tried to solve this with multiple threads. One that loads all the content of the game, and one that checked how much was loaded and printed the progress on screen.

Our first approach was to create a new thread for printing the progress and let the initial JavaFX-thread start loading content. This, however, was not possible, because when using JavaFX, the FX-thread must be the one that handles FX-elements. We therefore switched the threads in the solution, so that our JavaFX AnimationTimer printed the progress, and a new thread was created to load content and then instantly destroyed.

2.4 Persistent data management

There is no persistent data throughout the game. When a new game is initialized all the data for characters, powerups and so on is loaded from an XML-file.

2.5 Access control and security

This was no issue in the project since the application is only run on one computer and have no network capabilities. If anyone would hack to application they couldn’t do much damage either since it doesn’t have any access to system files.

2.6 Boundary conditions

Not applicable.

APPENDIX

Design model

See **Software_Decomposition_group15.pdf** in git project folder Documentation

UML Diagrams

See **UML_group15** in git project folder Documentation

Software dependencies

Google Guava EventBus - For posting and listening for events
JInput - This is a subproject of Java Games
JUnit - Used for handling unit tests.