System design document for 3 gamers och Jacob

Table of Contents

Version: 1.0

Date 2014-05-25

Author Daniel Jansson, Erik Pihl, Jacob Genander, David Michaëlsson

This version overrides all previous versions.

1 Introduction

This document describes the system and programming of project Teedee.

1.1 Design goals

The main goal of project Teedee is to create a Tower Defense game. The game should, to some degree, be extensible and make use of the Model-View-Controller pattern.

Goals that are important for the system are its performance and flexibility on different platforms and hardware. The system should act the same way without being dependent on what hardware is used (for example movement and speed of objects in the system shouldn't go faster or slower on different hardware). This is because the user shouldn't have to fulfill any major requirements on her side to be able to use the system the correct way.

The performance of the system is also important. This is because the system should run as smooth as possible and for the user to not experience latency issues.

1.2 Definitions, acronyms and abbreviations

The following list of words gives a definition of the model's parts.

Enemy - Thing that needs to be killed before it reaches the end of the path.

Player - The person that plays the game.

Path - The path that the enemies follow across the map.

Map - A top down representation of a map. Contains a path and a build area.

Game over - The player's game session ends if the player has no more lives. Prompts the user

to play the same map again or to select a new map.

Tower - An automated sentry-like object which shoots and damages enemies. Is bought by the player and can be upgraded by the same person, these actions requires money. Can only be placed in the build area of the map.

Money - Money is used to upgrade or buy towers. The player gets money from defeating enemies.

Status effect - an effect that changes the status of an enemy, e.g. change the speed of the enemy.

2 System design

2.1 Overview

The application is based on the Model-View-Controller (MVC) pattern. To comply with the library handling the graphics, the pattern was somewhat modified. The method that is rendering the view is calling the method that is updating the model. To follow the MVC pattern completely, the render method should ideally be called after an update in the model, not the other way around.

The way towers affect enemies that are shot was decided to be additive, meaning that an enemy may have multiple statuses applied to it at one point in time. However, a status from a particular tower is never applied twice, but the time of influence the status has on the enemy is reset every time the particular tower shoots the enemy. This was decided due to the fact that the tower otherwise would keep on shooting on the enemy, but have no effect on it.

2.2 Software decomposition

2.2.1 General

For UML-diagrams, see apendix.

2.2.2 Decomposition into subsystems

The project is utilising the design pattern MVC, though it doesn’t follow pure MVC design. The view and controller parts have been merged together. The project could therefore be divided into two subsystems: Model and View-Controller.

The model handles data, such as positions, lives and money. It is the model that keeps track of the game state.

The view handles the rendering of the application window, by reading positions and various states from the model and then drawing the visual representation of the model parts.

The controller handles incoming actions from the user, such as clicks and other mouse actions.

In the case of using libGDX as graphics library, the input is handled by the same class that acts as the view, that is why the MVC pattern is modified.

2.2.3 Layering

The project is mainly composed of five packages. These are:

- teedee - contains all the model classes and the packages towers, enemies and test.
- towers - contains the various tower classes used by the model.
- enemies - contains the enemy classes used by the model.
- screens - contains the classes used by the View.
- test - contains classes testing the model.

2.2.4 Dependency analysis

Project Teedee utilises libraries for assisting in testing and drawing graphics. For testing the project, the library JUnit is used. For drawing the game's graphics, the library LibGDX is used.

2.3 Concurrency issues

NA

2.4 Persistent data management

NA

2.5 Access control and security

NA

2.6 Boundary conditions

Because of the fact that we have no data storage or data that needs to be collected, the program doesn't need to initialize anything special on startup. What the program does on start up is to display the game intro to the user and then waits for input.

The system can be terminated at anytime without any risk of losing data or changing its state at next startup because we have no data storage, therefor nothing needs to be saved.

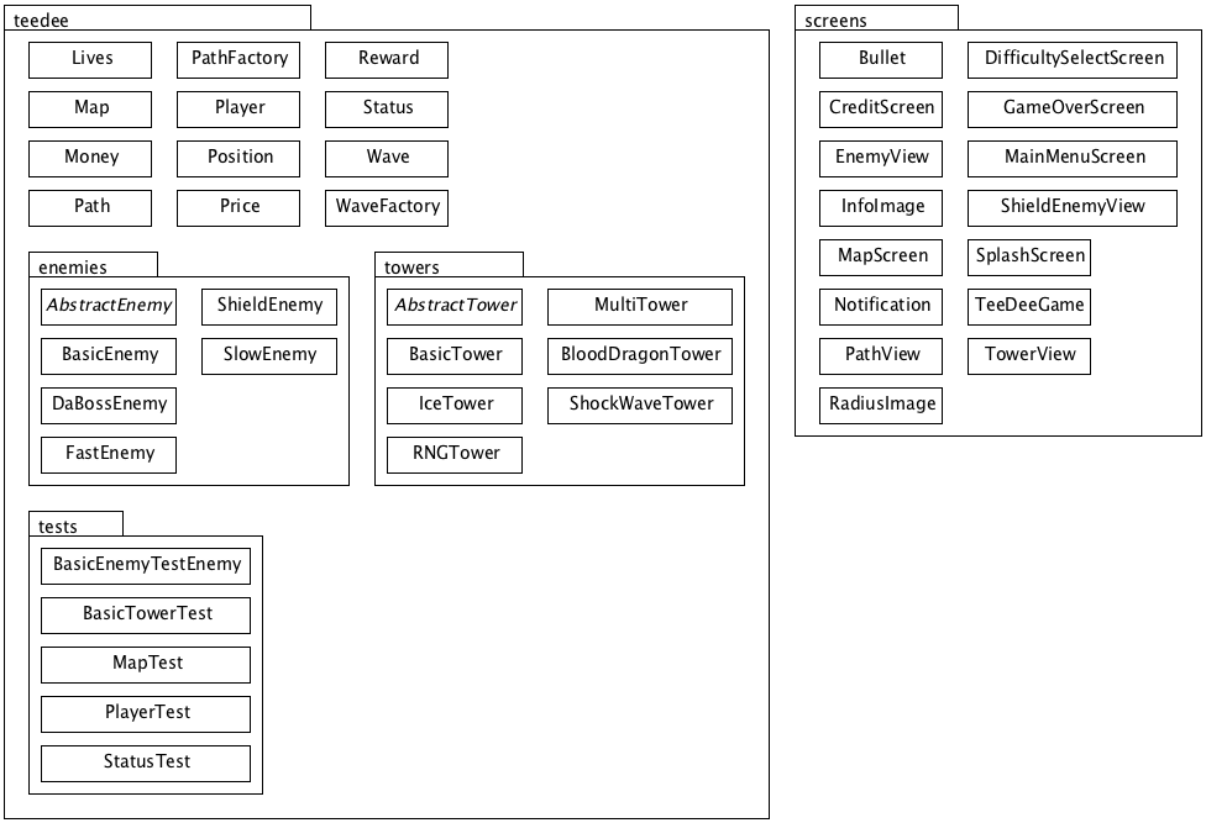
The system doesn't communicate with the internet or any other devices and also isn't depending on any files or other storage that can cause a failure. The only failure that can happen is a crash and then the system just shuts down. There are no back-ups or saving done if this happens.

3 References

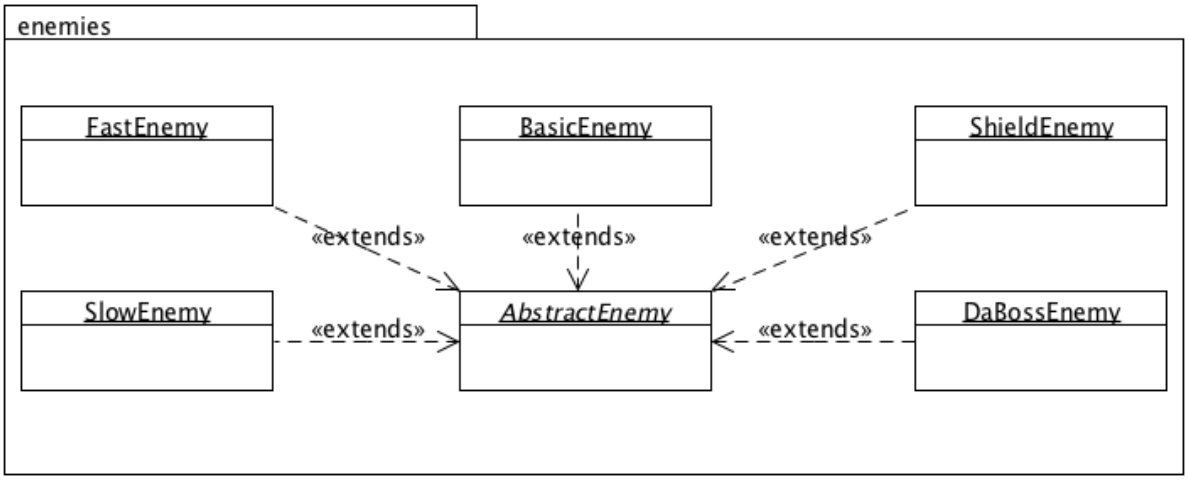
NA

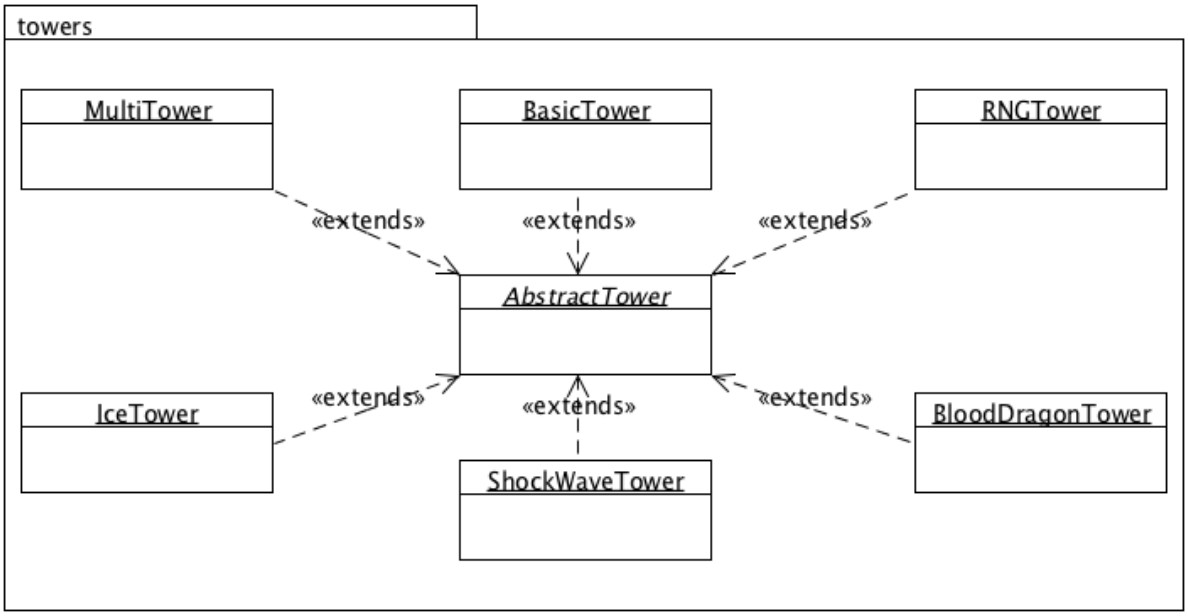
APPENDIX

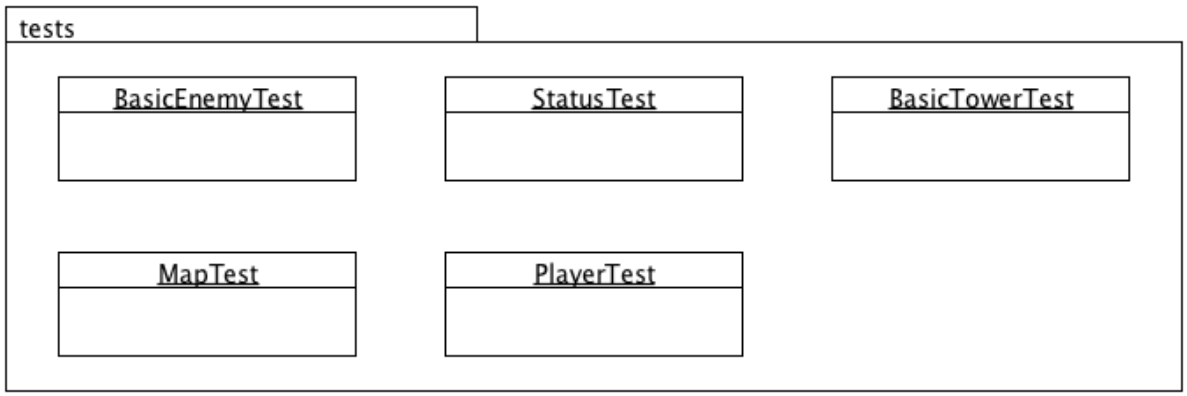
UML-diagram over packages:

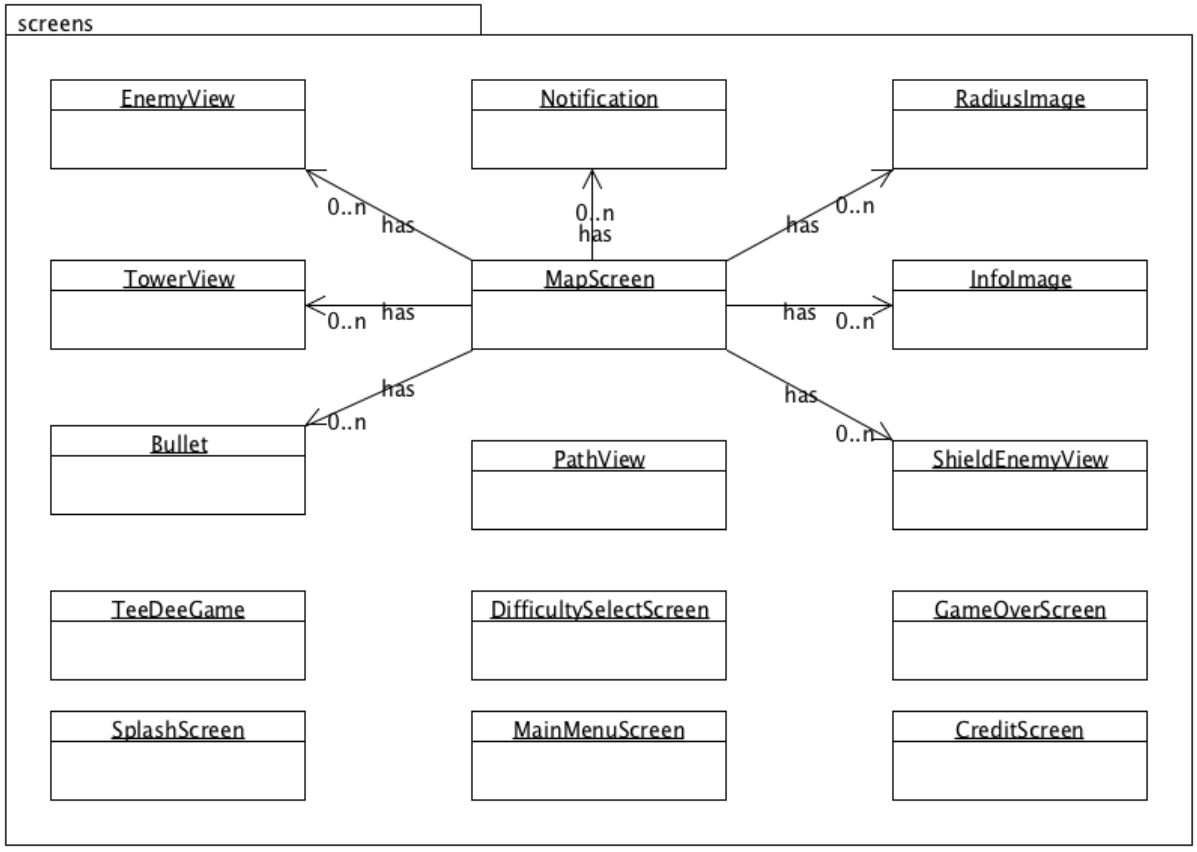


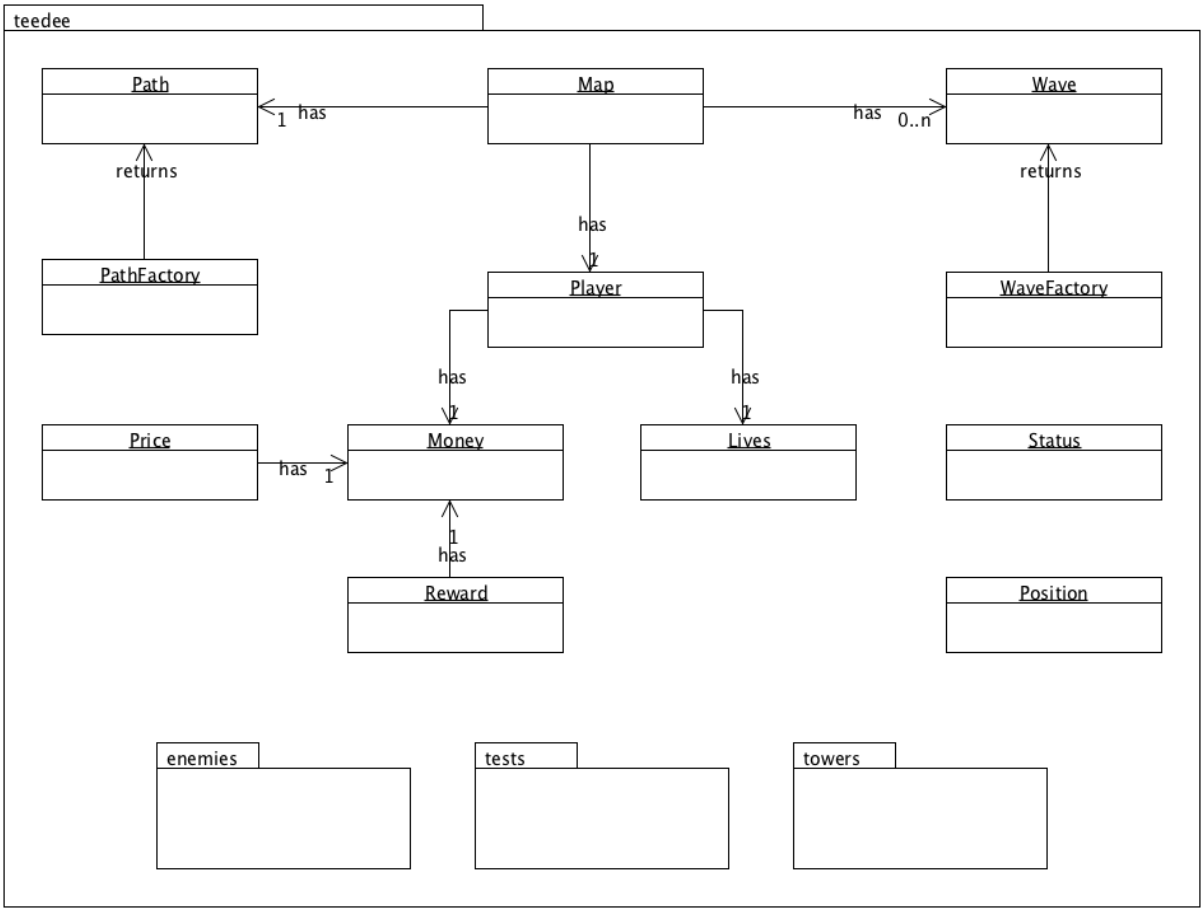
UML-diagram over each package:






tests
BasicEnemyTest
StatusTest
BasicTowerTest
MapTest
PlayerTest


screens
EnemyView
Notification
RadiusImage
TowerView
MapScreen
InfoImage
Bullet
PathView
ShieldEnemyView
TeeDeeGame
DifficultySelectScreen
GameOverScreen
SsplashScreen
MainMenuScreen
CreditScreen
has
0..n


teedee
Path
Map
Wave
1
has
has
0..n
returns
returns
has
1
PathFactory
Player
WaveFactory
has
1
has
1
Price
Money
Lives
Status
has
1
1
has
Reward
Position
enemies
tests
towers