Projektarbete
TDA367 Objektorienterat programmeringsprojekt
7,5 hp, VT 2016

System design document for get_rect()

May 26th. Version 1.1.

This version overrides all previous versions.

Felix Jansson, Jesper Lindström, Samuel Håkansson, Simon Sundström

Chalmers Tekniska Högskola

Contents

1 Introduction

1.1 Design goals

The design goals of this game is to be as loosely coupled to the underlying game framework as possible. The models may not have any direct framework dependencies, but we have also decided to strive for complete framework decoupling for controllers and views as well. Model View Controller pattern will be obeyed throughout the game.

The game content should be component based and easily extensible. We strive to keep a clear separation between the game content and the reusable framework classes.

1.2 Definitions, acronyms and abbreviations

- **Model-View-Controller (MVC):** A commonly used design pattern where business logic and presentation are clearly separated.
- **Strategy pattern:** Let a class ask for an interface instead of a concrete class, allowing more modular functionality without changing the class itself.
- **Adapter pattern:** Wrap another class and implement an interface, allowing the other class to be treated as the interface, via the adapter.
- **Role-Playing-Game (RPG):** A game where the player plays as a character in the game. Often includes features such as player items and quests.
- **Entity:** A game object, such as a player, weapon or door.

2 System design

2.1 Overview

Adapters and strategy classes are are used to add an abstraction layer between LibGDX classes and the game.

The game is structured as Model-View-Controller in order to clearly separate business logic, graphical representation and the controlling code. Views and controllers have a reference only to the model, but not the other way around.

See *figure 0* for a brief overview of the *get_ rect()* package structure. The top level package consists of sub modules, adapters and the main *game* package. The *game* package contains all game content, such as quests and game entities.

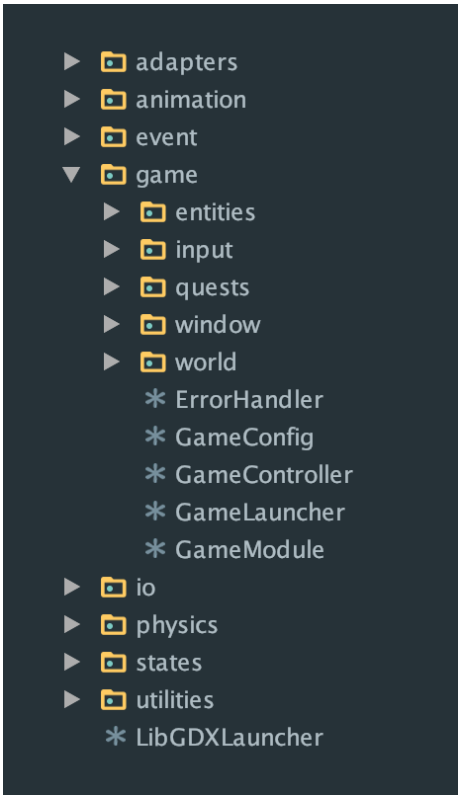

Figure 0: Overview of the *get_ rect* package

2.2 LibGDX

LibGDX is the game framework used by this game application. We never use the framework directly from the game code, and instead always interact

via adapters. The LibGDX components currently used are camera, graphics, input, sound, asset manager and geometry (rectangle).

2.3 Dependency injection

Dependency injection is used to provide game classes with correct concrete classes, allowing easier use of strategy pattern and resulting in better decoupling and modularity. We specify the class bindings in two levels; adapter classes in LibGDXModule and game classes in GameModule. Google's Guice package is used to handle injection.

2.4 Strategy for design patterns

Each game entity group (such as enemies, or world objects) has it's own set of a clearly defined package structure, consiting of factory, models, views, optional controllers, and optional repository.

Factory pattern The entity factory handles creation of the group sub entities. The return type for entity factories are always *IEntity*, which contains a model and a view. The factory is responsible for creating the model, view and providing the requested dependencies, which the factory can obtain via dependency injection in an easy manner.

Repository pattern The repository is used to, using the *IOFacade*, load details about for instance which entities should be present in the world. The repository uses the corresponding factory to create the entities, and then returns a list of *IEntity*.

Observer pattern Observable models are used via the *EventSource* class for when the models need to notify other components, such as the view or quest manager, of changes, just one time and not continously. For continous state changes, such as whether an enemy is currently flying or not, should rather be provided as a get-method, which the view can check on every *draw* call.

Model abstraction Abstract subclasses to *IModel* are frequently used throughout the game, to reduce code duplication and allow new game content to be easily added with as little code as possible. Protected methods are provided, to give the concrete subclasses with an easy way to perform frequent tasks, such as showing a text dialog or handling quest completion.

2.5 Software decomposition

2.5.1 General

The *get_ rect* package contains sub packages for both *LibGDX* framework adapters, reusable game components (such as *state*, *physics* and *animation*), but also the game content itself. The *game* package contains everything that is considered game content or RPG specific features. Due to the large number of game entities, we group MVC by feature – meaning, each entity package contains it's own models, views and controllers. The reason behind this is discussed in the following section.

2.5.2 Decomposition into subsystems

Several subsystems are used throughout the game application. These are defined in the top level package and are meant to be reusable for other games as well. The game content itself is a more integrated package, grouped by feature rather than functionality. As a result, it is very easy to add and remove entities, since all the entity code is bundled in their own package. In the game content package we consider game entity modularity to outweigh decoupling of specific functionality.

The *animation* package is a package that is entirely separated from the game content, using only the graphics adapter, which in turn uses the *LibGDX* graphics class. This package handles frame based animation, grouped by sequences and is used as a convenience class for views in the *game* package.

The *io* package is a package that provides an easy-to-use facade for reading and writing JSON data. It also handles mapping of JSON to simple Java data classes, using Google's GSON library.

The *physics* package is a lightweight and reusable physics engine, depending only on the rectangle adapter to calculate bounding box intersection. Models in the game content package implement the physics package interface *IPhysicsObject* (via the game package interface *IPhysicsModel*) and are added to the physics engine, to be tracked. Gravity, velocity and collision checks are handled for every model that is tracked.

The *states* package is a simple generic state machine, allowing one active state at a time. The *StateManager* also notifies the states when they become active or inactive.

2.5.3 Layering

The game application is divided into four distinct layers. On the very top, the game content itself, is kept in the *game* package. Beneath that lies the reusable components such as *io* and *animation*. The game and most of the reusable components then use the framework adapters, such as the Graphics adapter, which in turn translates the actions to the lowest layer, that is the *LibGDX* game framework.

2.5.4 Dependency analysis

Top level package dependencies are shown in *Figure 1*. No circular dependencies are found on a package level, as of the latest game version. See appendix A.1 and A.2 for dependency diagrams of the *game* and *game.entities* packages. Third party dependencies are listed in the appendix.

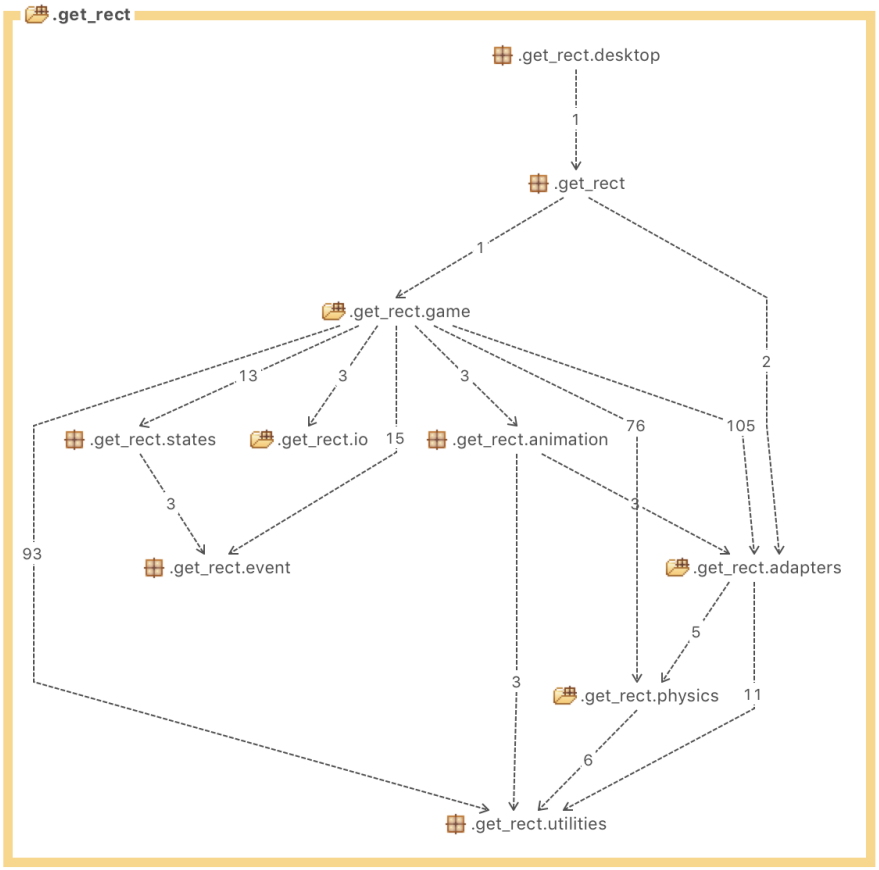

Figure 1: STAN dependency diagram of the *get_ rect* package

2.6 Concurrency issues

The game does not utilize multi-threading due to it's relatively small size. The *LibGDX* framework does not actively endorse use of threading in their guides, and running the game on thread has proved sufficient as of the latest game version.

2.7 Persistent data management

The game uses JSON for persistent data store, both for read-only game content and to save the player progress. The read-only game content contains information about which world objects, enemies and NPCs should be present in which world. It also contains details about quests and items. The player save data contains items, health and quest progress.
The data management is handled by repositories, such as the *PlayerRepository* or *EnemyRepository*. The repository utilizes the *get_ rect.io* package, which supports reading and writing JSON data, using Google's GSON library, and mapping the data to *Plain Old Java Objects*. These objects are named *DataStore* to differentiate from classes that are used in the game and the ones only used for saving and loading data. The *DataStore* is then used by the repository to, using the factory, create and return a new real game object, such as an *IEntity*.

For example, the *EnemyRepository* has a method that takes the world name (the room or location which the player can visit) and returns a list of ready-made *IEntity* objects containing various pairs of enemy models and views, that according to the JSON data should be present in the particular world. The repository tells the *IOFacade* to load the JSON file as a list of *EnemyDataStore* (containing the type and position), then uses the *EnemyFactory* to create *IEntity* for each data in the *EnemyDataStore* list.

2.8 Access control and security

NA

2.9 Boundary conditions

NA. Application is launched and exited like a normal desktop application.

3 References

3.1 APPENDIX

3.1.1 Diagrams

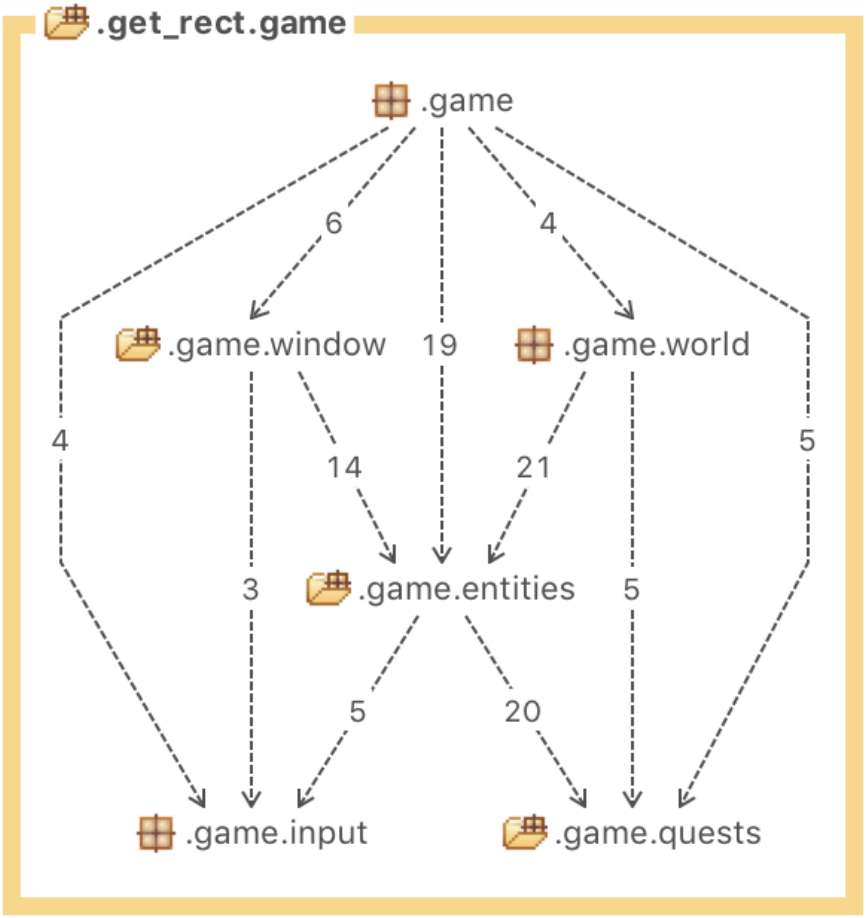

Figure A.1: STAN dependency diagram of the ‘get_rect.game‘ package

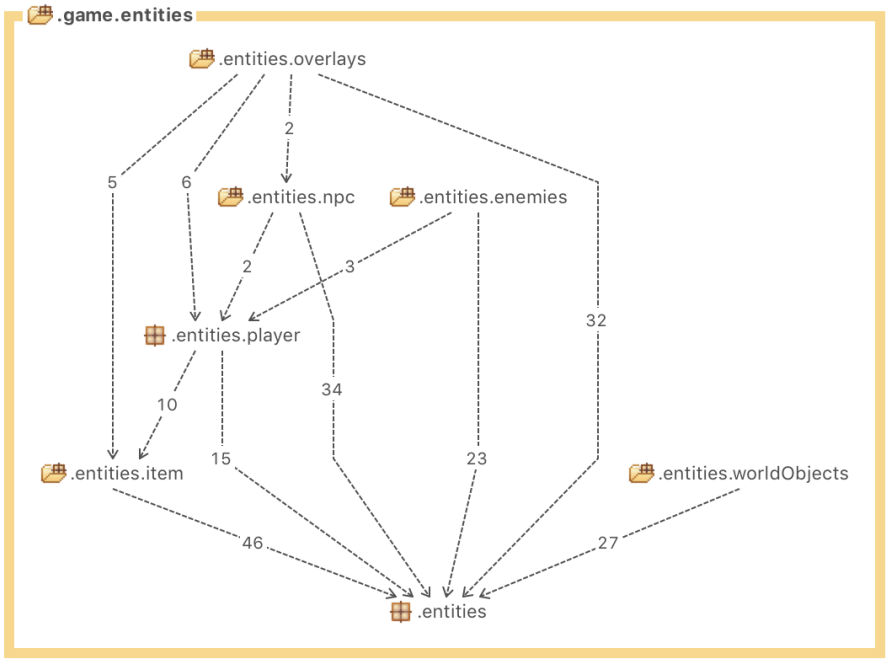

Figure A.2: STAN dependency diagram of the 'get_rect.game.entities' package

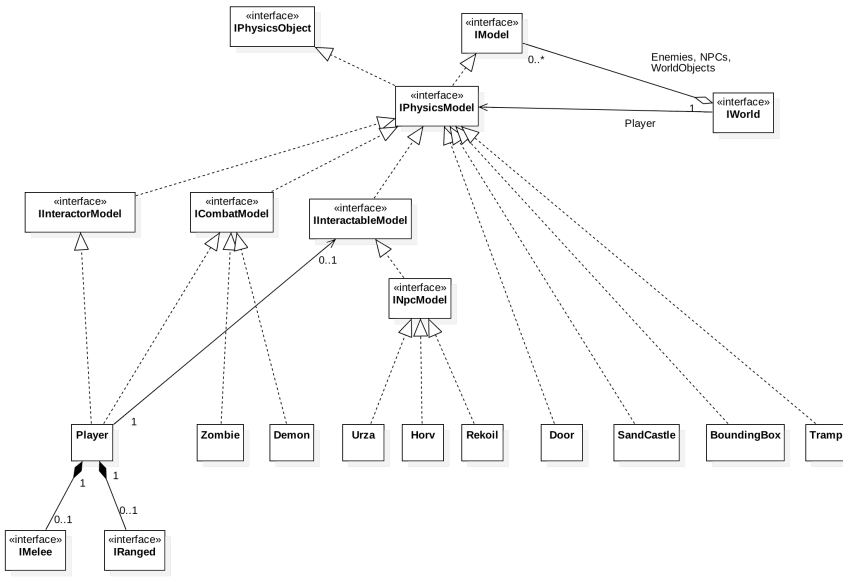

Figure A.3: Domain model based UML

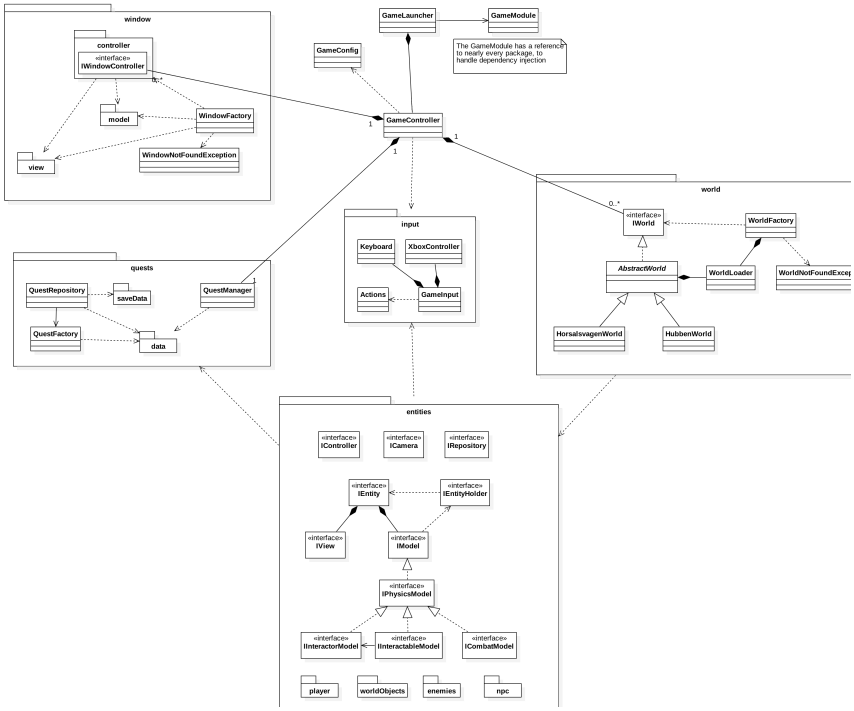

Figure A.4: Basic *get_rect.game* UML

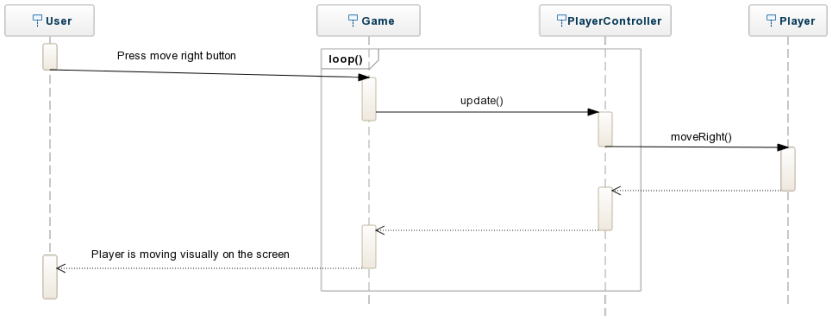

Figure A.5: Sequence diagram: Use Case 1 Move

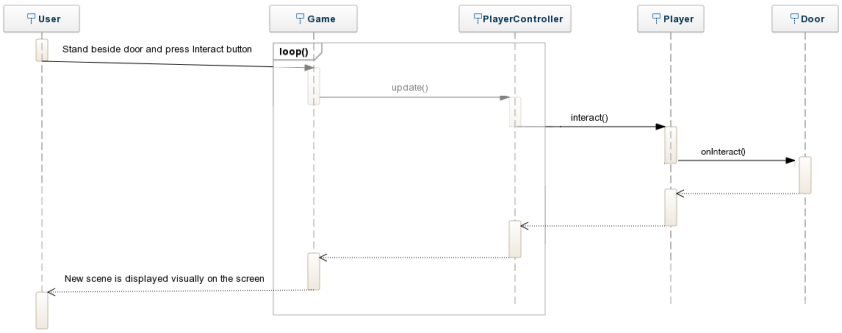

Figure A.6: Sequence diagram: Use Case 6 Interact with door

3.1.2 Software dependencies

- LibGDX
- LibGDX Controllers Desktop
- LibGDX Controllers Platform
- Guice
- GSON
- JUnit
- Mockito