Robo Rally Software Design Document

31 maj, 2015

Henrik Nilson, Axel Aringskog, Pär Löfqvist, Fredrik Kindström[1]

Version 1.0

Handledare: *Andreas Widén*

Objektorienterat Programmeringsprojekt TDA367

[1] Informationsteknik TKITE-1 Chalmers Tekniska Högskola

Table of contents

1 Introduction

1.1 Design Goals

One high priority goal is to have as few connections between classes and packages as possible, therefore reduce dependencies to a minimum. This helps to gravitate towards an abstract structure where it is easy to apply new functionality such as adding new tiles, adding new cards and adding new boards etc. The should result in packages with high cohesion and high modularity, and with minimal coupling between packages

1.2 Definitions, acronyms, abbreviations

GUI - Graphical User Interface, the graphical representation of the application.
MVC - Model View Controller, a programming design pattern.
Package by Feature - A package should be able to be removed without ruining the application.

2 Architecture

2.1 Overview

The application is made using the MVC design pattern, and is divided into three main packages. Model, view and controller. The model package is then subdivided into smaller packages to separate different functionality. These include model.card, model.gameactions, model.maps, model.robot and model.tiles which all holds classes related to the package name. The controller package holds the controllers and the view package holds the classes related to the GUI.

We choose not to use any external libraries or a already existing framework but to create our own platform. We did this because we wanted to extend the knowledge we'd learned throughout the year and to have 100% control over our code. The following text is the result.

2.1.1 Interfaces and abstract classes

Interfaces and abstract classes is used to achieve a "package by feature" standard in our application. This system greatly simplifies the process of adding new features. All you need to do is implement the interface and fit the required methods to a new specification.

2.1.2 Singleton pattern

The project contains classes which due to safety procedures only need to be created one time. The application therefore use the singleton design pattern to achieve this. Classes that use this are all the factories and the utility classes.

2.1.3 Single responsibility principle

Single responsibility principle states that all classes should be in charge of one isolated part of the functionality.

2.2 Software Decomposition

As shown in appendix 3.1 Package Overview, there are no circular dependencies between the three main packages.

2.2.1 Model

model.roborally

The top level model class which holds the status of the game.
This class mainly sets the ground stones for the game and holds the card deck.

model.round

Round illustrates a part of the game which deals cards and initiates the turn.

model.turn

Turn holds lots of the logic for what should happen when actions affects players.

model.player

Object class for a player in the game. This is the main object that is manipulated by the other classes in the model. Player is given a position, health, a card hand etc.

2.2.1.1 GameBoard (model.maps)

This model represents the gameboard and holds all tiles and their actions for each specific map. Here we use a factory to create the maps. (See appendix 3.2 Maps Package)

2.2.1.2 Tiles (model.tiles)

This package consists of a tile-factory which creates all of the tiles and assigns them different attributes, a tile can have multiple attributes. This makes it really easy to change the behavior and look of the tile. (See appendix 3.3 Tiles Package and 3.4 Attribute Package)

2.2.1.3 Cards (model.cards)

Similar to model.tiles, contains the cards used in the game. Instead of using a factory like model.tiles the package has an abstract class, "*RegisterCard",* which all card-classes extends instead. Subclasses to *RegisterCard* uses the "*setAction*"-method to differ from each other. (See appendix 3.6 Card)

2.2.1.4 Game Actions (model.gameactions)

This package hold all the actions a player (or robot) can execute. "*MovePlayer", "RotatePlayer"* and "*KillPlayer"* are some examples of actions. These actions are used by both cards and tiles. This is a really useful system since we can use them in different places of the program. (See appendix 3.5 Game Actions)

2.2.1.5 Robot (model.robot)

This package is used to create a robot (a graphical representation of a player) for each user playing the game. It does this by using another factory, "*RobotFactory*". In the factory, each robot is given a name, which is used to draw each robot's icon, and a color, which is used to create a unique representation of the robots prints in the console. This concept makes it really easy to adjust the graphical representation of a player.

2.2.2 View

The view package contains the GUI, the graphical representation of the application. Its a collection of classes extending swing components such as JPanels and JLabels. The exchange of panels is controlled by the top “*GUI-class*” which is registered as an event listener and displays the right panel depending on which event that is fired. The *GUI-class* is the only class visible to strangers, which means that all outside communication must pass through this class or not talk to the package at all. This is a really good way to make things secure.

2.2.3 Controller

The controller package holds the controllers which listens to events and delegate orders to both the model and the view. “*AppController*” is the main controller. It is directly in charge of starting the program and is the class that is called by the Java main-method. The “*GameController*” delegates orders to the classes holding the game logic and therefore contains almost all the code present in the package.

The idea behind the AppController class was that it should control things not concerned with game logic like adjusting sound and saving/loading games etc. These features were never implemented for this project but the AppController is still there to be extended with code if they ever are some day. It also makes sense to have the main-method call the application control instead of starting a game directly.

2.2.4 Utilities

The utilities package contains a collection of classes thats not a part of neither gamelogic nor the GUI but stilled used throughout the program. These classes are:

utilities.constans
	Containing all constants used throughout the program.
utilities.position
	An object class representing a point in a 2D space. Consists of an x and y value.

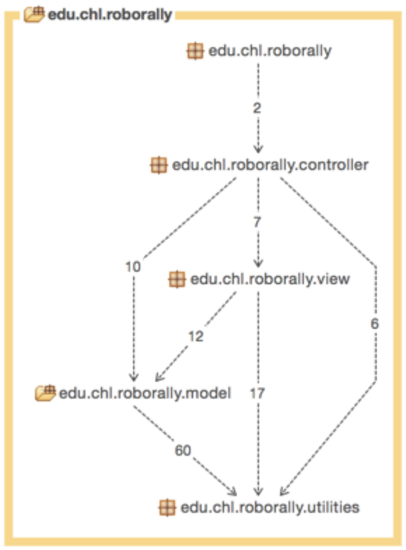
This package also contains the class and interface of the event system covered in the next section.

2.2.4.1 EventTram

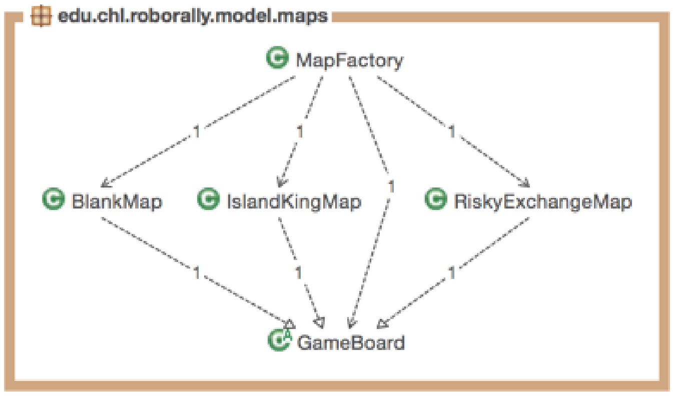
The “*EventTram”* (originally “*EventBus*”) handles all the communication between the packages (see Appendix 3.7 EventTram). This ensures that there is no circular dependencies between the three main packages.

3 Appendix

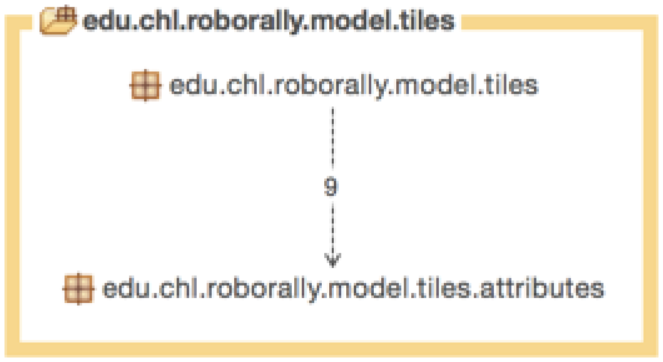
3.1 Package Overview



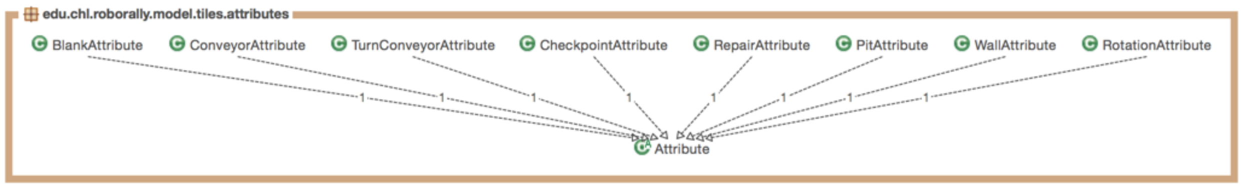
3.2 Maps Package



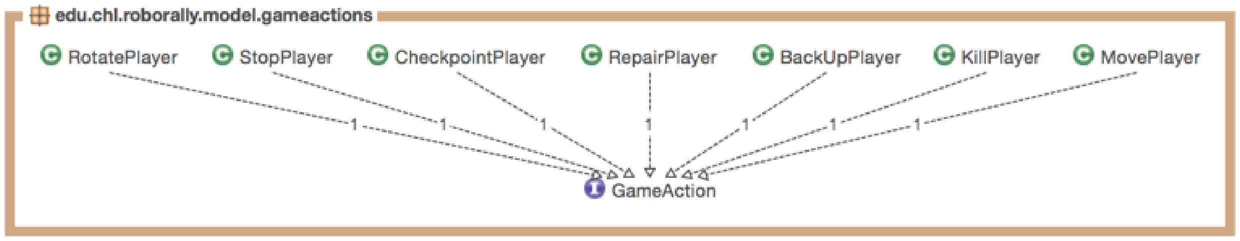
3.3 Tiles Package


edu.chl.roborally.model.tiles
edu.chl.roborally.model.tiles
9
edu.chl.roborally.model.tiles.attributes

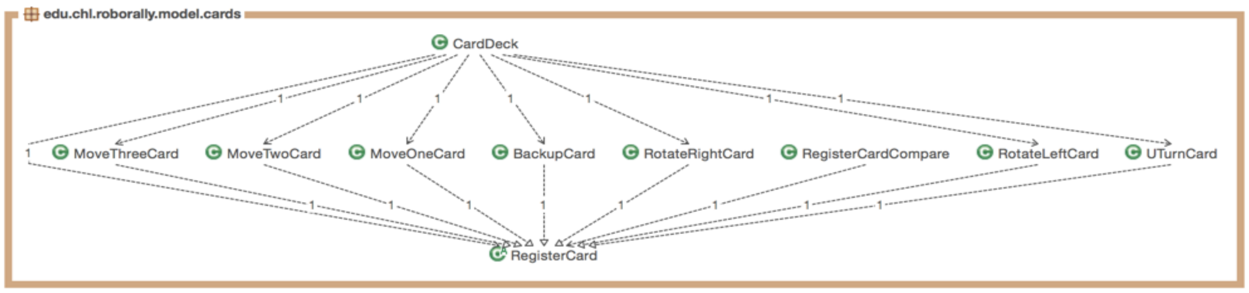
3.4 Attribute Package



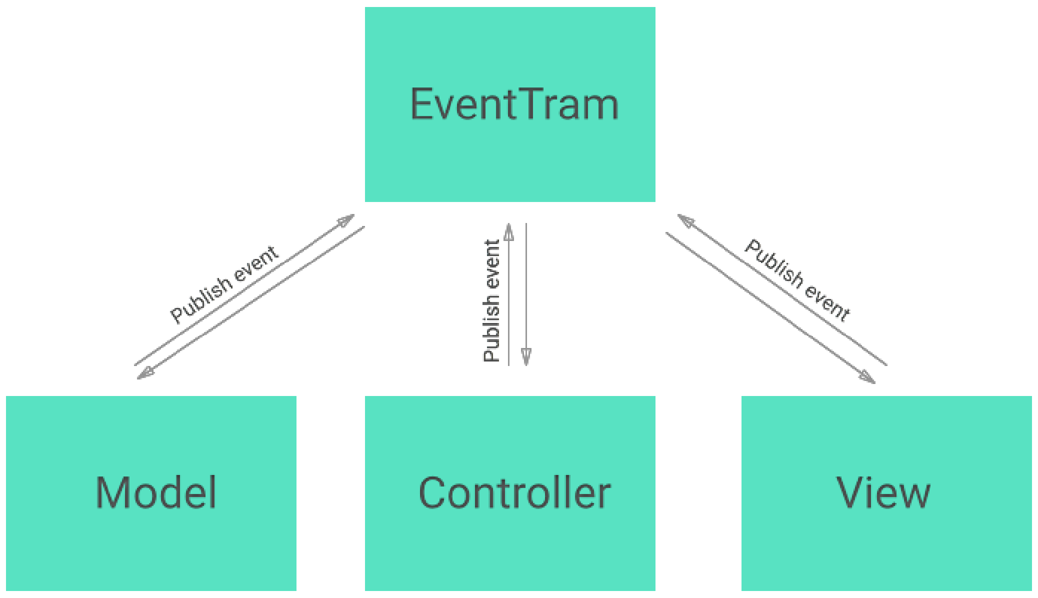
3.5 Game Actions



3.6 Card



3.7 EventTram


EventTram
Publish event
Publish event
Publish event
Model
Controller
View