System design document for the Monopoly project (SDD)

Contents

Version: 3.5

Date 2011-03-26

Author hajo

This version overrides all previous versions.

1 Introduction

This section gives a brief overview of the project.

1.1 Design goals

The design must be loosely coupled to make it possible to switch GUI and/or partition the application into a client-server architecture. The design must be testable i.e. it should be possible to isolate parts (modules, classes) for test. Furthermore, the design must take into considerations the future directions, see 2.5 RAD. For usability see RAD

1.2 Definitions, acronyms and abbreviations

All definitions and terms regarding the core Monopoly game are as defined in the references section.

- GUI, graphical user interface.
- Java, platform independent programming language.
- JRE, the Java Run time Environment. Additional software needed to run an Java application.
- Host, a computer where the game will run.
- Round, one complete game ending in a winner or possible canceled.
- Turn, the turn for each player. The player can only act during his or her turn (roll dices, buy, sell, etc.). Thou, the player can be affected during other players turns (i.e. pay to actual player, etc.)
- Resources (for players), the total value of the properties, buildings and cash of a single player. A player is bankruptcy when he or she has no more resources.
- MVC, a way to partition an application with a GUI into distinct parts avoiding a mixture of GUI-code, application code and data spread all over.

1.3 References

MVC, see http://en.wikipedia.org/wiki/Model-view-controller

2 Proposed system architecture

In this section we propose a high level architecture.

2.1 Overview

The application will use a modified MVC model without a separate C-part. This is because the model will be fat i.e. most of the logic will be in the model. This will affect testing. Model objects should be extensively tested. Possible a need for mock-up objects or special setups (because many model objects will reference each other).

(ALTERNATIVE: The model will be anemic i.e. no logic will be in the model. Very little testing of the model. Testing will take place in the control part.

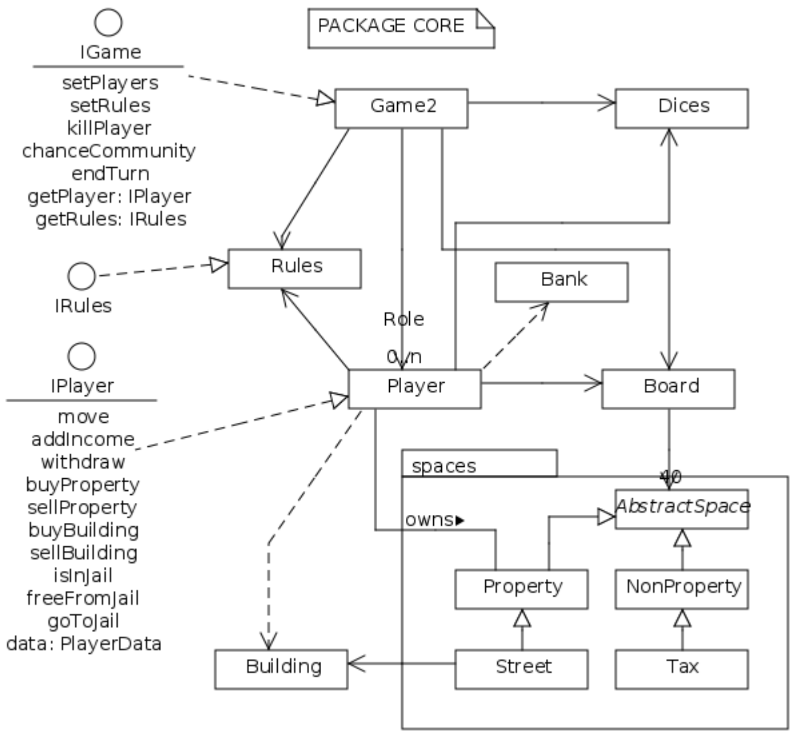

Figure 1: Model and functionality (interfaces)

2.1.1 Rules

The rules of the game could vary. This could be handled by different implementations of affected classes (subclasses). Yet, here we have chosen a different approach. All rules have been re-factored to a Rules class. Model classes delegates the rules-dependent parts to the Rules class.

In this way the rules also can easily be used to enable/disable components in the GUI.

2.1.2 The model functionality

Because we have a fat model the functionality of the model must be exposed (to be used in GUI and possible other parts). The models functionality will be exposed by the interface IGame. Run time an unique implementation of the interface will be globally accessible to the rest of the application. To avoid a very large and diverse interface the functionality will be split into (sub) interfaces. IGame will be the top level interface acting as an entry to other interfaces (IPlayer and IRules) see Figure.

2.1.3 Value objects

The (fat) model classes expose functionality. This is expressed as interfaces. If the application needs the functionality it call methods in the interface. This is quite different from when the application just need the core data of the object (for example to display in GUI). The design separates the functionality aspect and the data aspect. The model class Player has a matching (close to) immutable data class PlayerData containing the pure data (a value object). PlayerData objects are used when sending data between GUI and model.

2.1.4 Unique identifiers, global look-ups

We will not use any globally unique identifiers for any entity. There will be no look ups from anywhere in the application (objects will be directly connected or accessible without an identifier). Example: The spaces-objects will not be stored in any global accessible data structure to be looked up. They will be directly connected to interacting objects (GUI, etc.). See also Spaces.

2.1.5 Spaces

To have a uniform handling of spaces (possible configurable), all kinds of spaces are kept in a single list. The ordering of the spaces is determined by the ordering of the list. This will make it easy to create different views of the spaces (just traverse list and create a view for each space).

2.1.6 Event handling

Many events, state changing or not, can happen during the play (new player, dices equal, go to jail, etc.). A need for a flexible event handling is evident. If this is done at an "individual" level i.e. each receiver and sender connects, we possible end up with a hard to understand web of connections (also possible many receivers for one event/sender). How and when should connections be set? Also, during testing of the model we possible would like to disable the event handling.

To solve the above we decide to develop an "event-bus". All connections of senders/receivers and transmitting of evens is handled by the event-bus.

The connections could be setup at application start for static parts. Dynamic parts must have means to connect to the bus at any time.

Because some connection is between a unique sender and receiver (unique objects) we haven't found a way to use any existing framework.

2.1.7 Internal representation of text

All texts should be localizable. Therefore internally all objects will use a language independent key for the actual text. Using the key the object can retrieve the actual text.

2.2 Software decomposition

2.2.1 General

The application is decomposed into the following modules, see Figure 2.

- view, is the top level package for all GUI related classes including the main window.
- view.details, details pop-ups for the spaces (showing detailed info about some specific space).
- view.dialog, dialogs shown as response to some event.
- Main is the application entry class.
- event, classes related to event handling.
- build, classes related to the building of the model.
- core, the OO-model
- core.space, all spaces on the OO-model
- io, file handling
- util, general utilities.
- constants, application global constants.

2.2.2 Tiers

No tiers for now. If the application should be split into client/server architecture the view-package could form the (thin) client. Remaining packages will be the server. The event package could be either client (fat) or server side. The event handling must be adapted (networked).

2.2.3 Communication

NA

2.2.4 Decomposition into subsystems

The only subsystem is the file handling in package io (not a unified subsystem, just classes handling io).

2.2.5 Layering

The layering is as indicated in Figure . Higher layers are at the top of the figure.

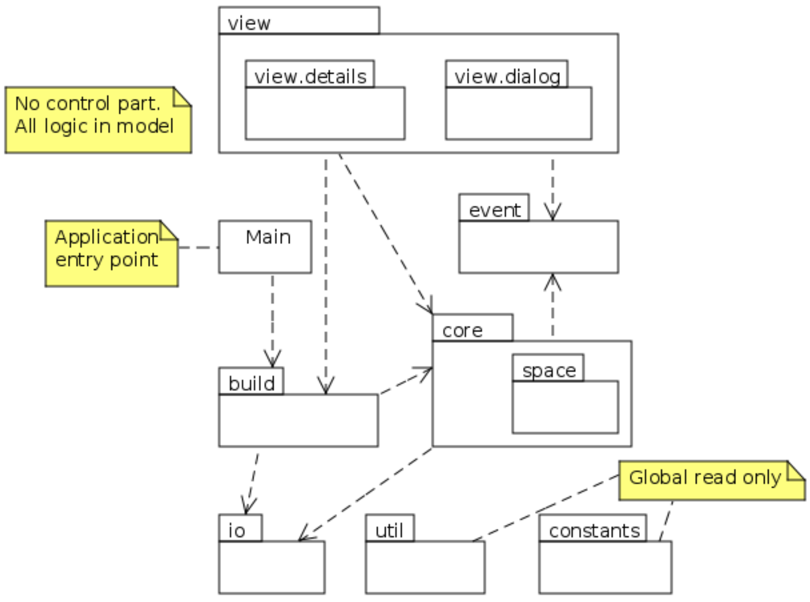

Figure 2: High level design

2.2.6 Dependency analysis

Dependencies are as shown in Figure. There are no circular dependencies except between core and core.spaces (no problem since this is just a administrative separation of classes representing spaces).

2.3 Concurrency issues

NA. This is a single threaded application. Everything will be handled by the Swing event thread. For possible increased response there could be background threads. This will not raise any concurrency issues.

2.4 Persistent data management

All persistent data will be stored in flat test files (format, see APPENDIX). The files will be;

- A file for spaces. The ordering of the spaces is used as the internal (implicit) ordering for the spaces-objects. This ordering will be directly reflected in the GUI. See further directions in RAD.
- Localization files containing entries (texts) for the text keys in the application.

2.5 Access control and security

NA

2.6 Boundary conditions

NA. Application launched and exited as normal desktop application (scripts).

2.7 References

Monopoly game: http://en.wikipedia.org/wiki/Monopoly_game

APPENDIX

(missing, U do...)