System design document for the Monopoly project (SDD)

Contents

Version: Monopoly .. last iteration...

Date ...some date...

Author hajo

This version overrides all previous versions.

1 Introduction

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. 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.

2 System design

2.1 Overview

The application will use a modified MVC model.

2.1.1 The model functionality

The models functionality will be exposed by the interface IMonopoly. To avoid a very large and diverse interface the functionality will be split into interfaces for Player and Board. IMonopoly will be the top level interface acting as an entry to other interfaces IPlayer and IBoard, see Figure.

2.1.2 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 will be 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.

Figure 1: Model and functionality (interfaces)

2.1.3 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.

2.1.4 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).

2.1.5 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.

2.1.6 Internal representation of text

All texts should be localizeable. Therefore internally all objects will use language independent keys 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.

- dialogs, GUI dialogs to interact with users. View parts for MVC.
- view, main GUI for application. View parts for MVC.
- eventbus, classes for the eventbus.
- ctrl, is the control classes for the MVC model
- adapter, contains an EventAdapter for the model. Broadcasting events for model state changes
- model, is the core object model of the game. Model part of MVC.
- Main is class holding main-method, application entry point.
- io, is for file handling.

2.2.2 Decomposition into subsystems

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

2.2.3 Layering

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

2.2.4 Dependency analysis

Dependencies are as shown in Figure. There are no circular dependencies.

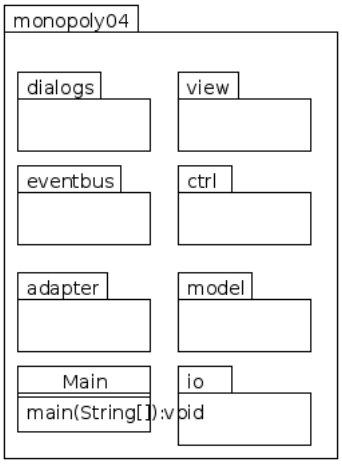


Figure 2: High level design

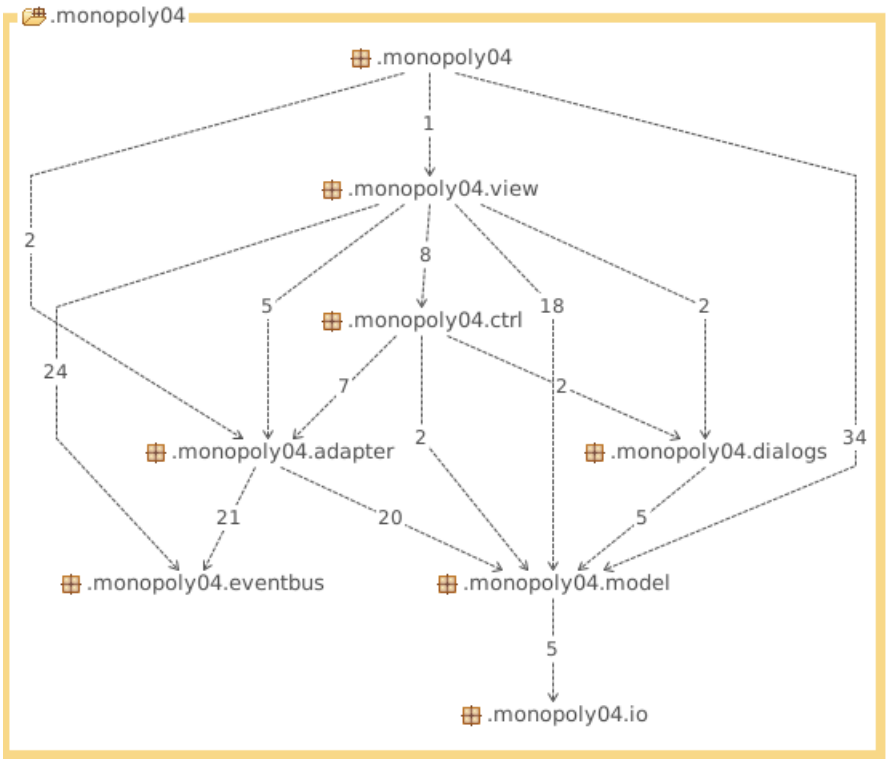


Figure 3: Layering and Dependency analysis

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

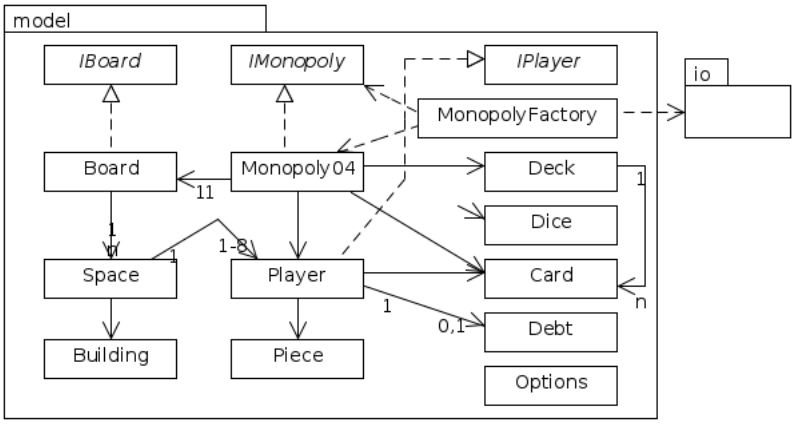
NA. Application launched and exited as normal desktop application (scripts).

3 References

1. Monopoly game: http://en.wikipedia.org/wiki/Monopoly_game
2. MVC, see http://en.wikipedia.org/wiki/Model-view-controller

APPENDIX

Class diagrams for packages (just one package, for the model, U do more if needed)



- Board is a container for Space's
- Monopoly is the controlling class for functionality in model, will coordinate and perform all operations as a response to user actions. If more user interaction needed class will keep state for next interaction.
- MonopolyFactory is responsible for building the complete model. Uses file handling from io to read Spaces.
- Deck is a container for Cards
- Space is the model for Streets, etc. A Space can have visitors (Players positioned at the space)
- Building, is houses or a hotel at a Space
- Piece is what the player moves on the Board
- Dice are a pair of dices
- Card is some Card picked by the player entering Chance or other.
- Debt is a class recording a players debt, if landing on some other players Space, to be paid before ending turn.
- Option is parameters for the model.

File formats (and more, missing, U do...)