System design document for the Monopoly project

Version: 0.4

Date: 2017-02-21

Author: Joachim von Hacht, hajo@chalmer.se

This version overrides all previous versions.

1 Introduction

(Note: This is more of a sketch, not finished ….)

1.1 Design goals

This document describes the construction of the Monopoly application specified in the requirements and analysis document.

Some 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.

- 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.

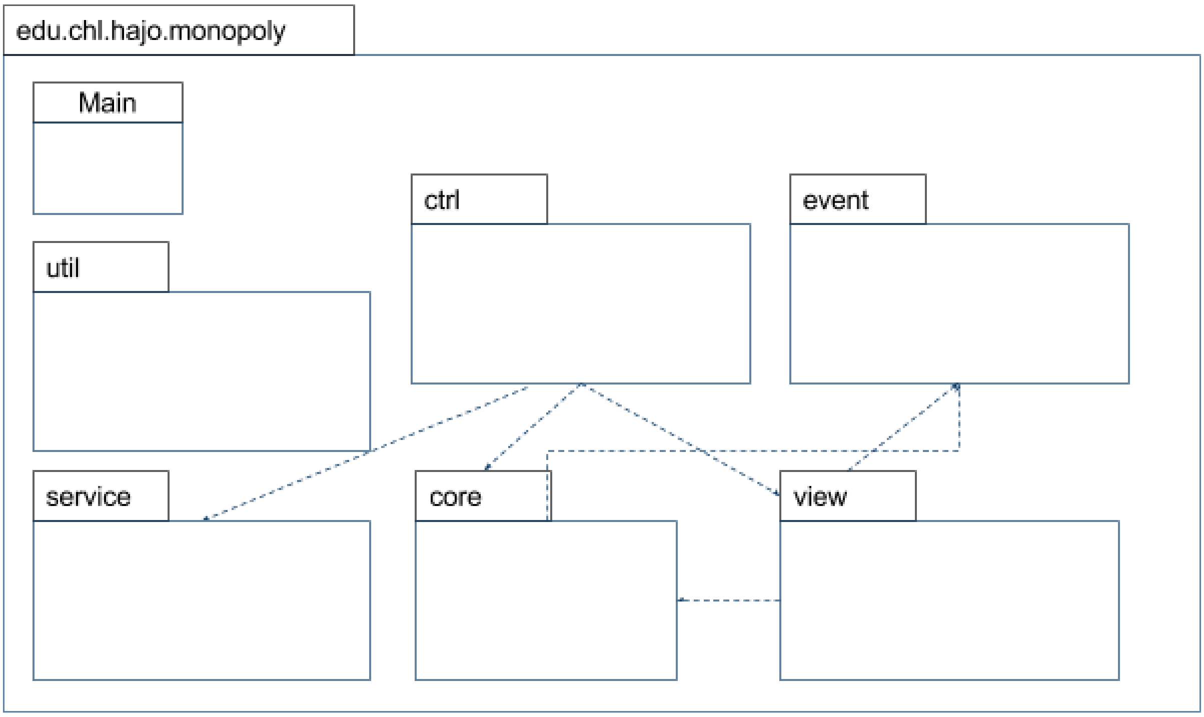
See RAD document for more definitions etc.

2 System Architecture

The application will use a Eventbus based MVC model. More complex uses cases will be handled by separate controllers in the control layer.

The application run on a single desktop computer.

The application is decomposed into the following top level packages (arrows for dependencies)



- Main is the application entry class.
- event, contains classes for the eventbus
- view, is the top level package for all GUI related classes including the main window.
- core, the OO-model
- ctrl are the use case controllers
- service is top level package for services like file handling
- util, general utilities. Possibly reusable

2.1 General observations

Aggregates

The application will consist of a single class aggregate with root class Monopoly. All calls from other parts of application to model will pass through the Monopoly class

The model functionality

The model's functionality (API) will be exposed by the Monopoly class. To avoid a very large and diverse interface to the class there are methods to access other classes in the aggregate (sub API).

Spaces

Spaces are handled uniformly by the board class and the GUI parts as a list. This will make it easy to create different views of the spaces (just traverse list and create a view for each space). The ordering of the spaces is determined by the ordering of the list.

To allow the application to recognize the specific properties of different spaces (street, tax, etc) there are subclasses implementing interfaces like: IOwnable, IBuildable, etc.

```
// Contract for ownable space (like streets)
interface IOwnable {
    public Player getOwner();
    public void setOwner(Player owner);
    public int getRent();
}
```

Unique identifiers, global look-ups

Space names and player names (String) must be unique, will be treated as id's

State notification

User input will be handled by standard Java event handling mechanisms (Swing). Call chains are like: GUI -> Controller class -> Monopoly (aggregate root)

The observer parts of the model will use an eventbus to notify the GUI.

- All updates of attributes in model will be done by setters
- Setters will propagate the state change via the eventbus

This will keep most of the model clean from event handling code.

```
// The contract for all classes able to receive events from eventbus
public interface IEventHandler {
    public void onEvent( Event evt);
}
```

Internal representation of text

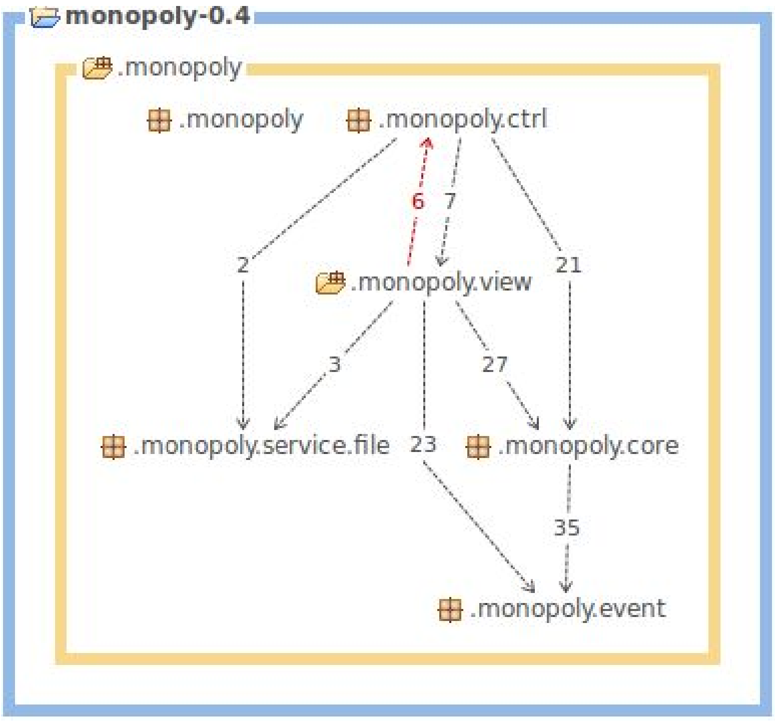
All texts should be localizable. Therefore internally all objects will use language independent keys for the actual text. Using the key the object can retrieve the actual text.

Lookups

For now references to the model are explicitly pass on to GUI and controls. Should be replaced by some lookup mechanism.

Dependency analysis

Dependencies are as shown. There is an issue between ctrl and view. Should be fixed.

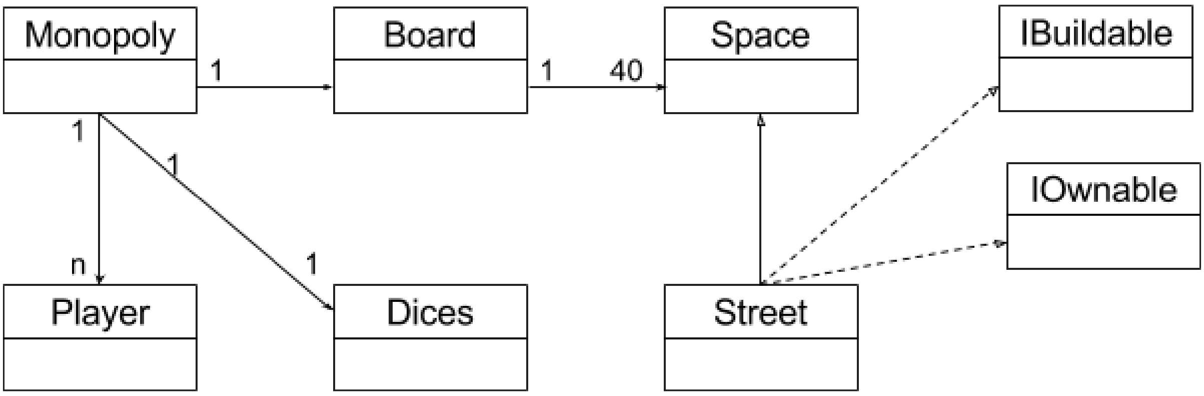


2 Subsystems decomposition

The only service (implemented as a subsystem) is the file handling responsible for getting icons and images

2.1 Core

The core package contains the design model.



(more descriptions here)

2.n Service

The service package contains of a file handling API for resources and … see persistent data managment

```
// Filehandling subsystem interface
public interface IFileHandler {
    public Image getIcon(String fileName) throws IOException;
    public List<String> getDirectoryFileList(String directory)
                                            throws IOException;
    public Image getImage(String pathAndfileName) throws IOException;
}
```

(more descriptions here)

4 Persistent data management

All persistent data will be stored in flat test files. 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.

(exact description of fileformat missing here)

5 Access control and security

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

6 References

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