System design document for MonOOPPoly

Authors:
Hampus Rhedin Stam
Jon Emilsson
Vilhelm Hedquist
William Johnston

October 24, 2021
version 3.0

1 Introduction

The application is a game that is inspired by Monopoly and which tiles are named after places around Chalmers campus. This document explains the system design for the document.

1.1 Definitions, acronyms, and abbreviations

- **MVC**, Model-view-controller, is a structural design pattern used to divide and organise the code into three sections: Model, View and Controller.
- **UML**, Unified Modelling Language, is used to visually represent the structure of a program.

2 System architecture

Our application uses the classical MVC pattern for top level architecture. The group felt most familiar with this pattern and it's use felt perfect when dealing with very clear visual models separated from the model and a controller to deal with inputs.

The model in our application is responsible for all calculations, like for example if things are purchased or sold. It's also responsible for keeping track of the game-state, who's turn it is, if someone is bankrupt and make sure special events take place. The View is responsible for all the visual elements. However as scenebuilder has been used, a lot of the graphical components are located under a resource folder. Most of the dynamic changes should happen in the view class, although this is not fully implemented yet. The controller handles user input, it sends the information to the model which acts accordingly.

The flow of the game, simply starts with the game initialising the view and controller, there isn't much model behaviour in the first two scenes. It stores some visual info, like how many players there are and their behaviour. When the final start game button is pressed Game is created and the model begins to work. Now all three components need to work together to achieve a playable game.

3 System design

There are 3 designpatterns used in the program, Observer, Visitor and MVC.

The observer pattern is used to reduce the amount of dependencies and to avoid circle dependencies. Because we have a class responsible for handling the board. This boardcontroller needs to know how it's controller affect the game. For example if a space is selected and then make sure necessary changes are made. Therefore in it's current implementation it's mostly used for boardcontroller to keep track of what it's others controller are doing.

The visitor pattern is used to connect different locales with different views. To make use of it we use an abstract class PropertyRentView which is used to gather all RentViews in a hashmap and the visitor pattern makes sure that the depending on which subclass of Property is used it will use the right RentView. This pattern can be used in more cases of the program but this has yet to be implemented.

MVC pattern is also used. The application is divided into 3 primary packages, view, controller and model. The main goal of this usage is to separate model from the rest. So that the model doesn't depend on any parts of the program. There are parts of the controller and view which intersect meaning they are not completely separated. But the model should be completely independent.

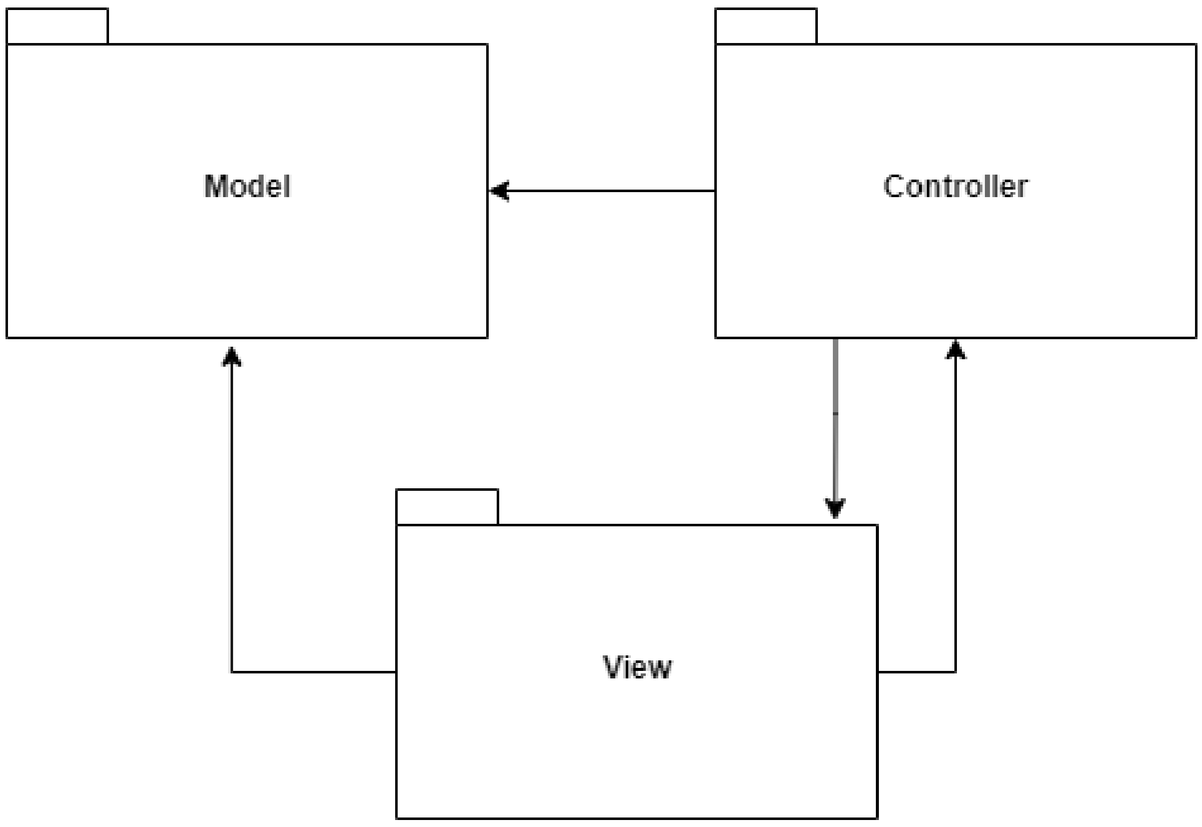


Figure 1: Package-diagram

We weren't able to compress our picture to a feasible format to be included in this report. Use the following link to see the UML. `https://github.com/williamProgrammerar/MonOOPPoly/blob/0b1ee7924e38bfd64ed5228e8b7b5682405d71d1/Documents/UML%20and%20Domain%20Model/UML%20final.drawio.png`

4 Persistent data management

MonOOPPoly has no user profiles or other persistent data, the only real resources used are small icons used to represent the players which are stored in a "resources" folder in the src-directory.

5 Quality

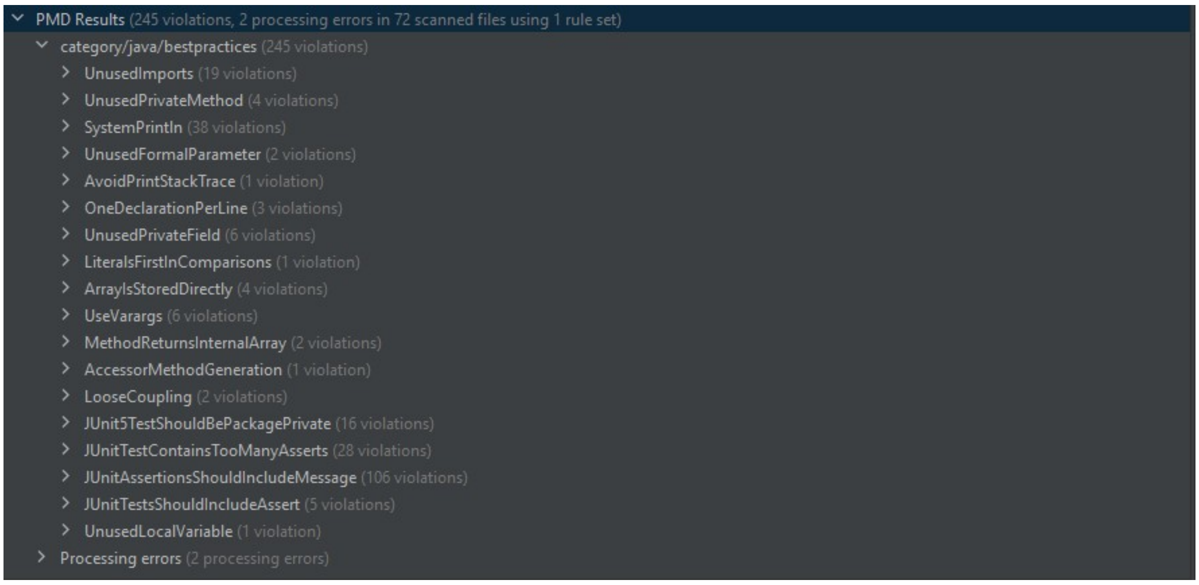
To obtain a high quality of the application it is necessary to test the program and check that the different classes and methods act like they are supposed to do. The tests of

MonOOPPoly can be found the in the test-directory inside of the src-directory. The tests are done with JUnit by asserting for different requirements. For example, there is a test for the dices and pieces to check that the pieces move the right amount of steps following the dices. This is done by asserting that the sum of the dice and the steps moved are the same. If it's true, then the test has passed. There are no known big issues for now.

Down below we list some known issues, most of these issues are how we want the program to be structured in the future and certain functionality.

- Bankruptcy should not be applied as soon as you run out of money. Bankruptcy also needs more fixing as players property aren't sold when you lose, meaning that you can still gain money which also means you can get back in the game.
- There is a lot of casting of objects and certain moments where we use instance off.We should have more polymorphism and an even more object oriented design.
- We break the single responsibility principle many times. The class Game, for example, does too much work, it's necessary to put all parts together in some form of class. But right now game does a lot of heavy work on it's own and there should be steps in the middle. An idea would be to add "ManagerClasses" some form of class which handles a certain part of game and then you instance that object, instead of creating an instance of all small components of the game.
- Mortgage is not fully implemented yet it still has problems regarding mortgaging properties other than those of the Locale class. You also can't select other properties or spaces then Locales.
- The game also exits the program as soon as someone has won. This is not preferable and a victory screen should appear instead before exiting.
- If you start a trade when somebody doesn't own a property it throws an exception, this doesn't crash the program but it's an issue.

Following picture describes the Quality reported from PMD running - java best practice rule.



```
PMD Results (245 violations, 2 processing errors in 72 scanned files using 1 rule set)
  category/java/bestpractices (245 violations)
    UnusedImports (19 violations)
    UnusedPrivateMethod (4 violations)
    SystemPrintln (38 violations)
    UnusedFormalParameter (2 violations)
    AvoidPrintStackTrace (1 violation)
    OneDeclarationPerLine (3 violations)
    UnusedPrivateField (6 violations)
    LiteralsFirstInComparisons (1 violation)
    ArrayIsStoredDirectly (4 violations)
    UseVarargs (6 violations)
    MethodReturnsInternalArray (2 violations)
    AccessorMethodGeneration (1 violation)
    LooseCoupling (2 violations)
    JUnit5TestShouldBePackagePrivate (16 violations)
    JUnitTestContainsTooManyAsserts (28 violations)
    JUnitAssertionsShouldIncludeMessage (106 violations)
    JUnitTestsShouldIncludeAssert (5 violations)
    UnusedLocalVariable (1 violation)
  Processing errors (2 processing errors)
```

Figure 2: Java best practice report

We weren't able to run a dependency analysis tool on our program we therefore refer to our UML diagram if information is needed regarding the dependency.

5.1 Access control and security

MonOOPPoly uses no login or similar system, there is no need for security or different user roles.

6 References

List all references to external tools, platforms, libraries, papers, etc. The purpose is that the reader can find additional information quickly and use this to understand how your application works.

For the program we have used JavaFX for building up everything that is graphically visible. This together with an external tool called Scenebuilder. We have also used classic Monopoly-rules for the game and some cultural words and places for Chalmers University of Technology for naming different things.

JavaFX: https://gluonhq.com/products/javafx/

Scenebuilder: https://gluonhq.com/products/scene-builder/

Monopoly: https://www.hasbro.com/common/instruct/00009.pdf

Chalmers: https://www.chalmers.se/sv/Sidor/default.aspx