SWEN30006 Software Modelling and Design
Workshop 9: GoF Patterns (Part 3)
Facade, Observer and Decorator

School of Computing and Information Systems
University of Melbourne
Semester 2, 2020

Completion: You *should* complete the exercises ***as a team***. To receive a workshop mark, you **need** to demonstrate your active participation during the workshop. Your tutor will observe your participation for you to be eligible to receive a mark for this workshop. The active participation include (but not limit to) present your solutions, help your teammates, and actively engage with the team. Attending the workshop with only passive participation will *not* be given a mark. See the Workshop Overview under Subject Information on the LMS for more details.

Requisite Knowledge and Tools

To complete this workshop you will need to be familiar with the following terms and concepts:

- Class Diagram
- GRASP principles
- GoF Facade, Observer and Decorator.

Introduction

This week we will be focusing on applying three GoF patterns, **Facade**, **Observer**, and **Decorator**, to a particular problem. The exercises begin with designing a system using provided information, then analysing how the GoF patterns can be used to solve design problems. Finally, we will finish by implementing the system. However, the decorator pattern will only involve design but not implementation. These exercises should allow you to gain an understanding of particular design problems through a concrete example, and demonstrate how to apply design patterns to solve the problems.

GoF Patterns: Facade, Observer and Decorator

Below is a brief summary of three GoF patterns that we will be using in this week's exercises.

- **Facade:** The facade pattern is a structural pattern. This pattern is often used in real-world applications. The facade pattern is ideal when working with a large number of interdependent classes which are complicated to use. Hence, the facade pattern is used to define a simplified interface of such a complex subsystem. By using the facade pattern, the client class that uses this complex system

does not need to know all the classes and methods of this complex subsystem. The client class just can simply call few functions from the facade object.

- **Decorator:** This is a structural pattern that allows users to add new responsibilities to an existing object without modifying its structure or subclassing. For instance, adding a class to decorate the border of a given rectangle (shape) without changing how the rectangle is currently being drawn.
- **Observer:** The observer pattern is a behavioral pattern. The observer pattern is used to allow an object, known as the subject, to publish changes to any objects that are interested (i.e., observers) in these changes. Other observer objects that depend upon the subject can subscribe to the subject object. When changes are published from the subject object, the observer objects will be automatically notified and immediately process the changes if needed. This observer pattern provides loose coupling between the subject class and its observers. We can change any observer object without making changes to the subject object.

The System Under Discussion: (Regular) Monopoly

This week you will be *extending* the Larman's Case Study system (i.e., a simulation of Monopoly). Figure 1 shows a partial domain model of the current Monopoly system. See partial design models from the textbook in the separate file **Monopoly_DesignModel.pdf**.

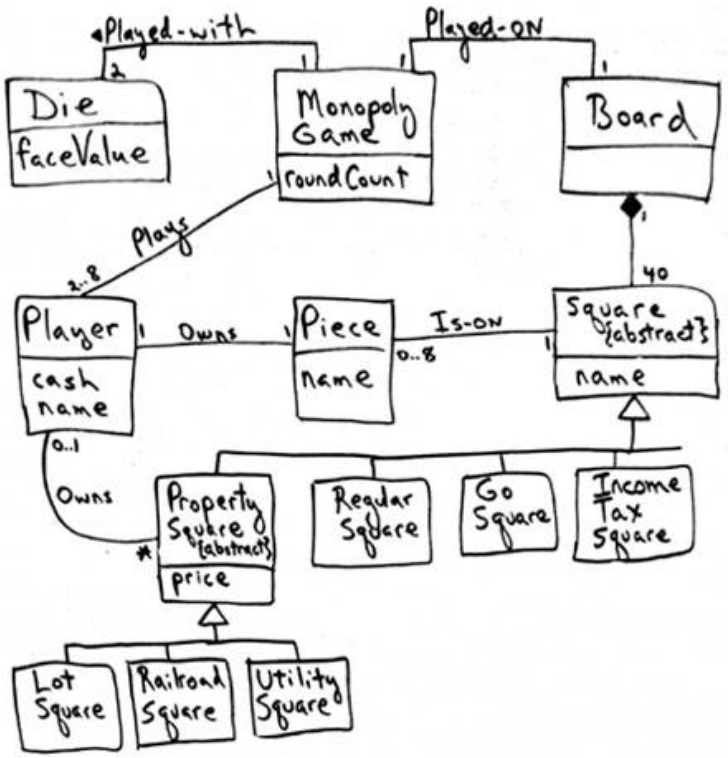

Figure 1: Domain Model of Monopoly

A brief description of the current system is as follow. The system gets the number of players as an input from the user and initialises the board and the players. Then, the system runs for 20 rounds and stops. In this system, the board has 40 squares:

- 1 × Go square
- 1 × IncomeTax square
- 1 × Jail square
- 3 × Go to Jail squares

- 3 × Rail Road squares
- 2 × Utility squares
- 3 × Lot squares
- 26 × Regular squares

The current rules for these squares are:

- If a player lands on a **Rail Road, Utility, or Lot** square, the player will attempt to purchase the square. If they successfully purchase a square, other players have to pay the rent to the owner when they land on that square.
- If a player lands on the **Go to Jail** square, the player has to move to the **Jail** square.
- If a player lands on the **IncomeTax** square, the player has to pay 10% of their net worth (or $200 at maximum) as tax.
- If a player lands on the **Go** square, the player gets $200 cash.
- If a player lands on the **Jail** or **Regular** squares, the system does nothing.

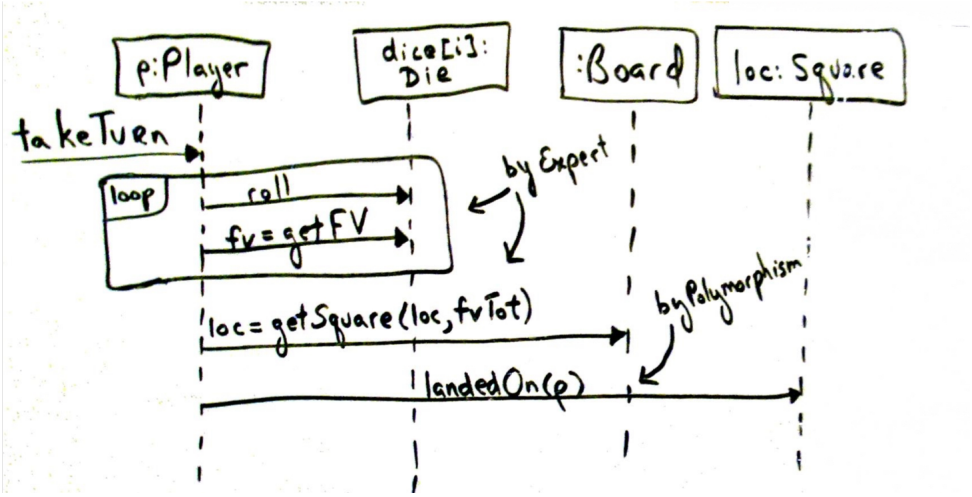

Figure 2: System Sequence Diagram: Dynamic Behaviour when a player takes turn.

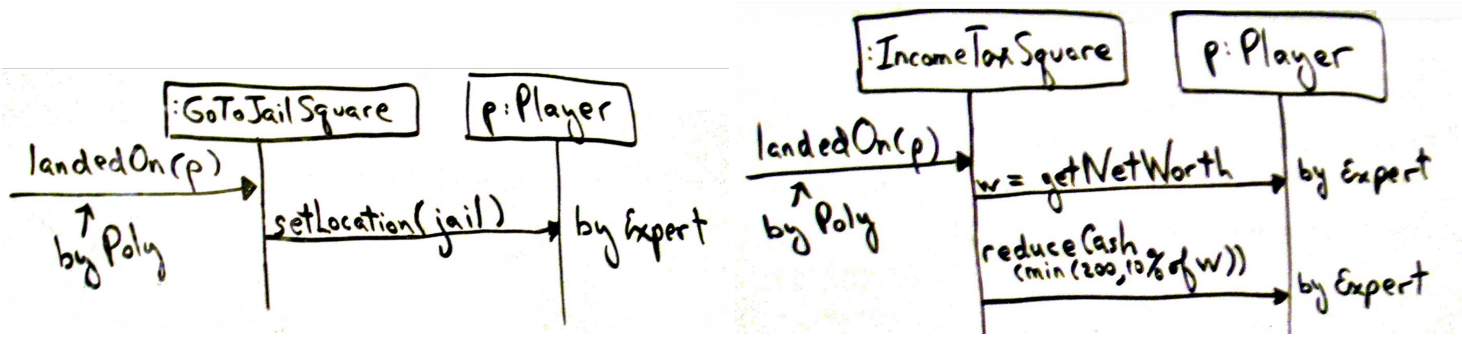

Figure 3: System Sequence Diagram. (Left) Go to Jail case, (Right) Income Tax case

Part 1 Adding LuckCards

Now, you will be adding one more feature – LuckCards – to this system. LuckCards are used only when a player lands on the **Go to Jail** square. The player will randomly draw 1 card from the card deck and follow the instruction described in the card. There are 5 cards in the deck:

- 1 × JailExcemptionCard: If the player gets this card, the player stays on the Go to Jail square. No need to move the Jail square.
- 2 × GoToJailCard: If the player gets this card, the player moves to the Jail square.
- 2 × PayJailFeeCard: If the player get this card, the player pays $500 to exempt from Jail. If the player's cash is not sufficient, the player has to move to the Jail square.

Requirement: If you look at the SSD in the figures above, you should be able to identify that this feature should be added into the `GotoJailSquare` class. However, when adding this feature, the current GotoJailSquare class should have very **low impact**. Ideally, only one statement should be added into the existing class of the Monopoly system. Based on this low impact requirement, the **Facade** pattern should be used to implement this feature. Also consider other applicable patterns.

Task 1.1 Modelling

Using the description above, create a design class diagram for classes to show how can we use the Facade pattern for this additional feature.

Task 1.2 Implementation

Using your design from the previous sub-task, implement this feature into the current Monopoly system. Download the `Monopoly_Workshop.zip` file in LMS and import the project (See Workshop 1 or 7 for the import instructions). A Java package for luck cards (`com.unimelb.swen30006.monopoly.card`) is provided in the zip file. However, you are welcomed to implement your own luck cards.

Part 2 Building a House

In the original behaviour, nothing happens when a player lands on a Regular square. So, let's spice things up! Now, when a player lands on an empty Regular square (i.e., a Regular square without a house), that player can build **a house** and own the square with a house. After that, if the owner (i.e., the same player who built a house) lands on his/her regular square with a house, the owner builds **a garden**.

A regular square which has a house (and a garden) generates income to the owner, which is collected every time as the owner lands on that square. Income is generated at the following rate:

- A Regular square with a house generates $20 to the owner
- If a Regular square has a garden, it also generates ($5×#Visits) to the owner
  Note: The counter of visits starts when a garden is built and it counts every time when any player lands on that sqaure.

Requirement: Instead of modifying the existing code for Regular square to use an if-else statement, we would like to **modify its behaviour at runtime**. Hence, the **Decorator** pattern should be used to implement this feature.

```
Player 6: dice total = 8 move to Square 16 with a house and a garden
Square 16 with a house generates 20 of income to Player 6
Square 16 with a house and a garden generates income of 25 to Player 6
```

Figure 4: An example of log for a regular square with a house and a garden, where Player 6 is the owner of Square 16 and Square 16 with a house and a garden has been visited 5 times by any player (i.e., $5×5=$25

Task 2.1 Modelling

Using the description above, extend your a design class diagram for classes to show how can we use the Decorator pattern for this additional feature.

Task 2.2 Implementation

Using your design from the previous sub-task, implement this feature into the current Monopoly system.

Minimum Expectation: The system is already complex. It is acceptable if you cannot follow some of the GRASP principles. At least, you should try to apply the Decorator pattern.

Part 3 Logging Player's Assets (Extra)

Now, you will be adding another feature – a logging engine – to this system. This logging engine records assets (i.e., cash and owned squares) of each player in a `txt` file. More specifically, while the system is running, it produces one **CashTransaction_{player name}.txt** file and one **OwnedSquares_{player name}.txt** file for each player. Table 1 shows an example of these files. Below are the details of how the logging engine should record this information.

Table 1: An example of logged assets files.

CashTransaction_Player 1.txt	OwnedSquares_Player 1.txt

`Amount, Balance`
`Init, 1000.0`
`- 300, 700.0`
`+ 200, 900.0`
	`Lot x 2`
	`Railroad x 1`

Cash Transaction Logger

When this cashTransaction logger is instantiated, it first writes the column titles (i.e., `Amount, Balance`) and the initial cash of a player (i.e., `Init, 1000.0`) into the `CashTransaction_{player name}.txt` file. Every time the `addCash` or `reduceCash` methods from the player class are called, the logger will automatically append a line that shows the amount of cash that is added or reduced and the player's new balance. For example, if the player had $700 and gained $200, the logger should append the line '`+ 200, 900.0`'. If the player had $1000 and lost $300, the logger should append the line '`- 300, 700.0`'.

Owned Squares Logger

The `OwnedSquares_{player name}.txt` file should show the squares that are owned by a player. Every time the player successfully purchases a square, the logger should automatically rewrite the whole file to show the list of square types and the number of each type of square that are owned by the player. See

the related Class Diagrams in the current Monopoly system in the separated `Monopoly_DesignModel.pdf` file.

Requirement: Ideally, this logging engine should have low coupling with the current Monopoly system. For example, in the future, the logging engine might record the data in a format other than a text file. Based on these requirements, the **observer** pattern should be used to implement this logging engine.

Task 3.1 Modelling

Draw a design class diagram for your subject and observer classes and any other classes (if any) related to this additional feature.

Task 3.2 Implementation

Using your design in the previous sub-task, implement this feature into your current Monopoly system. Below is example code for writing to files in Java.

```
// For Cash Transaction Logger:  Create a new file
public void writeFile() {
    try {
        FileWriter outStream  = new FileWriter(fileName);
        outStream.write("Write Something\n");
        outStream.close();
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(1);
    }
}

// For Cash Transaction Logger: Append text into a file
public void writeFile() {
    try {
        FileWriter outStream  = new FileWriter(fileName, true);
        outStream.write("Append a new line\n");
        outStream.close();
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(1);
    }
}

// For Owned Squares Logger: Count frequency and write into a file
public void writeFile(ArrayList<String> list) {
    try {
            FileWriter outStream = new FileWriter(fileName);
            Map<Object, Long> counts = list.stream().collect(Collectors.grouping
            for(Entry<Object, Long> entry: counts.entrySet()) {
                outStream.write(entry.getKey()+" x "+entry.getValue()+"\n");
            }
            outStream.close();
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
```

```
            }
    }
```