Title: **Design Patterns: Builder & Observer**

Subject: **Padrões e Desenho de Software**

Author: **João Miguel Abrantes Dias**

Date: **06/05/2020**

Index

Introductory Note:

The purpose of this project is to apply the programming concepts learned in *PDS* class. The focus of this essay is on two specific design patterns, the **Builder**, and the **Observer**. Essentially, a design pattern is a general repeatable solution to a commonly occurring problem in software design. It is not a finished design that can be transformed directly into code, but rather a description (or template) for how to solve a problem that can be used in many different situations. In order to get a better understanding of these concepts, I am going to apply them to real-life examples, show the whole process, and conclude by stating the pros and cons of each pattern based on the final product.

1. Builder Pattern

1.1 Introduction

Let us start by understanding what this pattern does, the **builder pattern** belongs to the **creational patterns'** family, meaning that it deals with object creation mechanisms. The builder pattern allows the separation of a complex object from its representation by using a step by step approach, so that the same construction process can create different representations.

In the following example, we are going to use the builder pattern to help us solve a problem about figurines. Figurines fall under several categories, in this example we are going to focus on 3 of them: *Monster* (Figure 1), *Humanoid* (Figure 2), and *Chibi* (Figure 3). All three types of figurines have shared components, they all have legs, arms, head, etc., but there are differences between them, for example, *Chibi* figures are characterised by having a big head relatively to their body, whereas *Monster* figurines are the total opposite.

Figure 1 - Monster figurine example

Figure 2 - Humanoid figurine example

Figure 3 - Chibi figurine example

1.2 Problem

Knowing this, our figurine object is going to be very complex with lots of fields and nested objects. This type of initialization would end up being inside a gigantic constructor with lots of parameters, or scattered all over the client code, which by itself, is an even worse implementation.

1.3 Solution

To solve this problem, we implement the **builder pattern**, meaning that we extract the construction code out of the figurine class and move it to separate objects. These new objects are going to be our builders, responsible for each figurine type. Let us start by building our *UML* diagram:

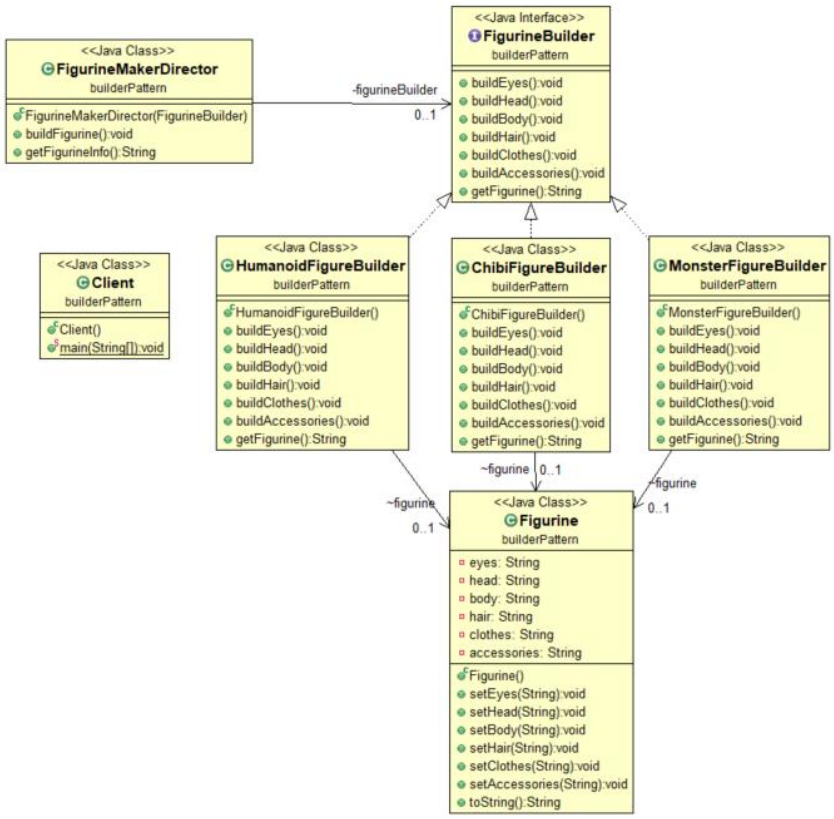

Figure 4 - Figurine *UML* solution

Since our builders implement the same methods, we create a "general" **Figurine Builder** interface where we store all methods. As explained above, we create the figurine class, but all its construction code is implemented in the builders. Our **FigurineMakerDirector** is going to be the one responsible for executing the building steps whereas each builder (**HumanoidFigureBuilder**, **ChibiFigureBuilder** and **MonsterFigureBuilder**) tells him how to do so. All of this is then used by our **Client**.

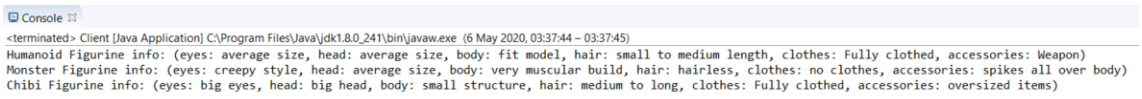

Figure 5 - Output Example

1.4 Advantages

- Immutable objects can be built without using a lot of complex logic in the building process.
- Helps minimizing the number of parameters in the constructor so there is no need to pass a null as an optional parameter in the constructor.
- Complex construction code is isolated from the business logic.
- Allows for control oversteps of construction process.

1.5 Disadvantages

- We need to create a separate builder for each type of figurine, so it would probably not be the best option if we had to make a builder for every type of figurines that exist.
- Dependency injection can be less supported.
- Builder classes need to be mutable.

1.6 References

- https://refactoring.guru/design-patterns/builder
- https://en.wikipedia.org/wiki/Builder_pattern
- https://www.tutorialspoint.com/design_pattern/builder_pattern.htm
- https://sourcemaking.com/design_patterns/builder
- https://www.geeksforgeeks.org/builder-design-pattern/
- https://medium.com/@andreaspoyias/design-patterns-a-quick-guide-to-builder-pattern-a834d7cacead

2. Observer Pattern

2.1 Introduction

Unlike the builder, the **observer pattern** belongs to the **behavior patterns** family, meaning that its concern is the assignment of responsibilities between objects and how they can decouple senders and receivers. The **observer** focuses on the subscription method, it allows for multiple objects to get notified whenever an event or change happens to the object they are observing.

One of the many examples where this pattern would come in handy is for sports betting houses. These usually allow for the users to not only bet before, but also during the match itself. For that, the match information needs to constantly be updated. Usually the game information is fetched from one only entity that contains the updated game information.

2.2 Problem

Knowing the above, for a betting house to know, for example, when a team scores a goal, it needs to keep constantly checking until a goal is scored. However, every time they checked in which no goals were scored, would be a waste of their time. On the other hand, if the entity itself was to send this information to all the betting houses, it would save a lot of them from useless checks, but it would also be sending information that a lot of other betting houses weren't interested in.

2.3 Solution

To solve this problem, we will implement the **observer pattern**, we create an observer entity that implements a subscription mechanism allowing betting houses to subscribe or unsubscribe the entity that provides the continuous stream of information. To understand it better, let us look at our structure first.

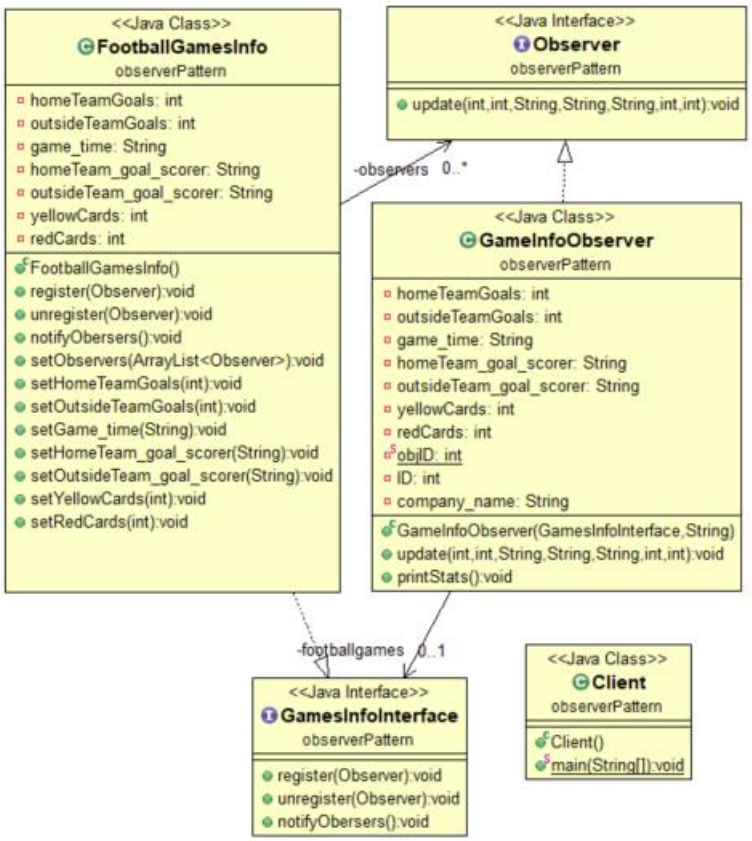

Figure 6 - observer *UML* diagram

The **FootballGamesInfo** class is the subject that our observers are going to subscribe to, it contains an array list with all its observers. The **GamesInfoInterface** is where the methods, for subscription to be possible, are defined. The **Observer** interface is where the method for updating all observers is defined, the **GameInfoObserver** class is the one created for each betting house that wants to receive the game updates. The following example will help us really understand how it works.

We start by creating a new subject and a new observer "*betclick*" (Figure 7). This new subject enters the observers array as soon as it is created by passing the subject in its constructor argument, we can see in the output that the subject was added to the array. After this, we receive a change in the game, a red card occurred so the gameInfo updates its status. As we can see in the output, if we check the observer stats, we can see that the information was indeed updated. After the first update, the subject decides

that it no longer needs more information about the game and it's removed from the observer's array list. Even though the game information is later updated, when we check the information of the subject, it remains unaltered.

```
FootballGamesInfo gameInfo = new FootballGamesInfo();
GameInfoObserver betclick = new GameInfoObserver(gameInfo, "betclick");

gameInfo.setGame_time("57:00");
gameInfo.setOutsideTeamGoals(1);
gameInfo.setOutsideTeam_goal_scorer("Quaresma");
betclick.printStats();

gameInfo.unregister(betclick);

gameInfo.setGame_time("89:00");
gameInfo.setYellowCards(1);
betclick.printStats();
```

Figure 7 - Client Example

```
Console
<terminated> Client (2) [Java Application] C:\Program Files\Java\jdk1.8.0_241\bin\javaw.exe (13 May 2020, 15:51:01 – 15:51:06)
### Observer Design Pattern Example ###
New Observer: betclick, ID: 1
An observer added!
Company: betclick [Home Score: 0, Outside Team Score: 1, Recent home Scorer: null, Recent outside Team Scorer: Quaresma, Red Cards: 0, Yellow Cards: 0, Game Time: 57:00]
An observer removed!
Company: betclick [Home Score: 0, Outside Team Score: 1, Recent home Scorer: null, Recent outside Team Scorer: Quaresma, Red Cards: 0, Yellow Cards: 0, Game Time: 57:00]
```

Figure 8 - Output Example

2.4 Advantages

- The Subject (*FootballGamesInfo*) and Observer classes can be used independently of each other as they are loosely coupled.
- The Subject (*FootballGamesInfo*) does not need to be modified in order to add or remove subscribers.
- Relations between objects can be established at runtime.

2.5 Disadvantages

- A memory leakage known as *Lapsed listener problem* can occur.
- Notifications can be unreliable.
- Subjects are notified in a random order.

2.6 References

- https://www.tutorialspoint.com/design_pattern/observer_pattern.htm
- https://sourcemaking.com/design_patterns/observer
- https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/
- https://refactoring.guru/design-patterns/observer
- https://en.wikipedia.org/wiki/Observer_pattern