Universidade do Minho

Mestrado em Engenharia Informática

Arquiteturas Aplicacionais

Trabalho II

Design Patterns

Ana Murta (PG50184)

Beatriz Oliveira (PG50942)

Gonçalo Soares (PG50393)

Joana Alves (PG50457)

Vicente Moreira (PG50799)

Março 2023

1 Introdução

Neste relatório apresentamos o resultado da pesquisa e implementação de dois *design patterns* de **criações** distintas, mais especificamente, o *Factory Method* e o *Abstract Factory*. Estes dois padrões são muitas das vezes confundidos devido à sua semelhança, sendo, por isso, o objetivo deste relatório a implementação de ambos para o mesmo contexto/problema, de modo a distingui-los.

Assim, o **tema** escolhido retrata a seleção de vários tipos de janelas (do ecrã) e de botões para um determinado sistema operativo, podendo dessa forma usufruir ao máximo destes *design patterns*, uma vez que estes enquadram-se nos contextos de herança, de composição e de generalização de código.

2 *Abstract Factory*

Este padrão tem como objetivo fornecer uma interface para a criação de famílias de objetos sem conhecer a sua implementação concreta, minimizando as dependências entre os objetos. Resumidamente, uma classe delega a responsabilidade da instanciação do objeto a outro objeto por **composição**. Desta forma, foram criadas três interfaces no total:

- **AbstractWindow -** interface de generalização dos tipos de janelas;
- **AbstractButton -** interface de generalização dos tipos de botões;
- **AbstractFactory -** interface de generalização dos tipos de sistemas operativos;

De seguida, implementamos as classes concretas de cada objeto, sendo estas: os botões e as janelas para ambos os sistemas operativos. Finalmente, foram implementadas as classes de gestão de cada sistema operativo que definem os tipos de janelas e de botões a utilizar em cada caso, obtendo, assim, o seguinte diagrama:

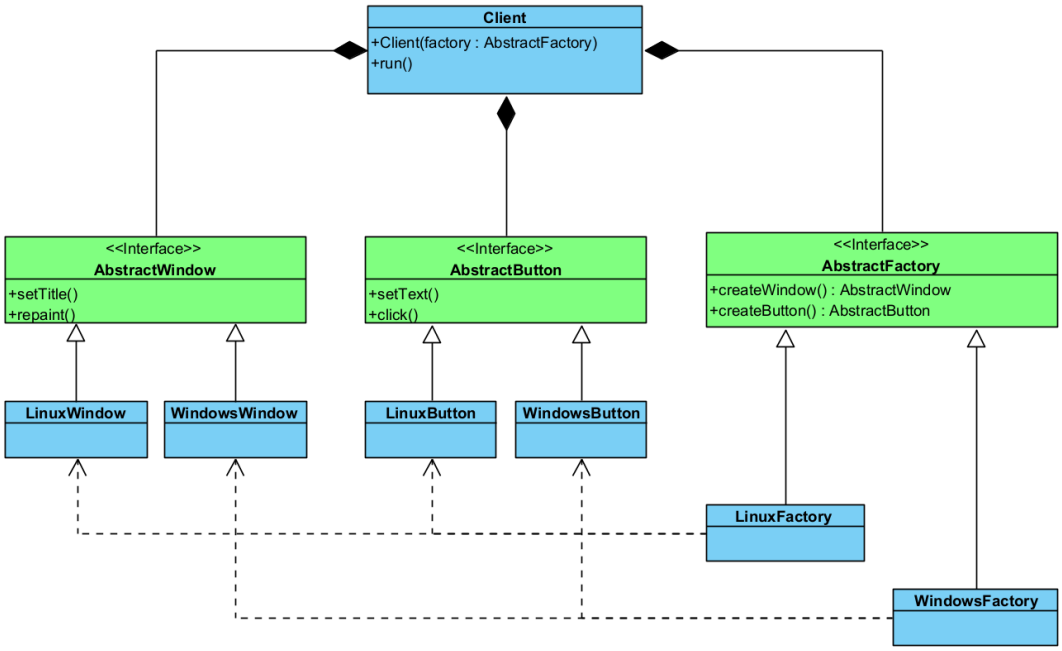

Figura 1: Diagrama de Classes - *Abstract Factory*

3 *Factory Method*

O padrão de *Factory Method*, tal como o nome indica, é caracterizado por um método e não por uma classe propriamente dita, isto é, utiliza a **herança** e depende de uma subclasse para lidar com a instanciação do objeto desejado.

Desta forma, o objeto de generalização dos sistemas operativos (*AbstractFactory*), que pelo *design pattern* anterior se descrevia como uma interface, neste foi instanciado como uma classe abstrata, delegando assim a responsabilidade de instanciação para as suas sub-classes. Para além disto, foram retirados os objetos correspondentes ao botões, desde a sua interface às suas classes de especialização concretas, com o objetivo de facilitar a perceção da implementação.

Deste modo, obtivemos o seguinte diagrama de classes final:

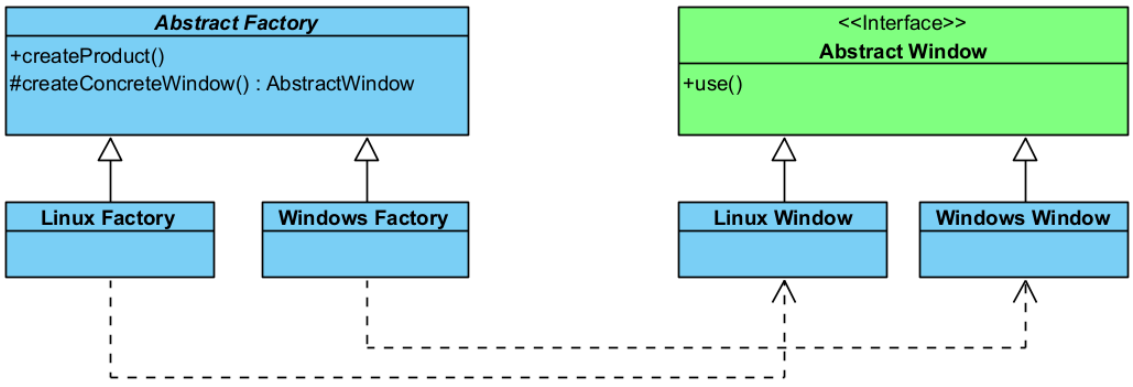

Figura 2: Diagrama de Classes - *Factory Method*

4 Implementação

De modo a facilitar a leitura do código desenvolvido, iremos apresentar as diferenças existentes nas secções críticas de cada um dos *design patterns*.

4.1 *Abstract Factory*

No que diz respeito ao **Abstract Factory**, podemos denotar que a interface apenas descreve o comportamento, tendo cada uma das classes que a implementam de escolher explicitamente o tipo de objeto que gostariam de instanciar.

```
class LinuxFactory implements AbstractFactory {
    @Override
    public Window createWindow() {
        return new LinuxWindow();
    }

    @Override
    public Button createButton() {
        return new LinuxButton();
    }
}
```

```
class WindowsFactory implements AbstractFactory {
    @Override
    public Window createWindow() {
        return new WindowsWindow();
    }

    @Override
    public Button createButton() {
        return new WindowsButton();
    }
}
```

Desta forma, o lado do Cliente recebe um objeto do tipo *AbstractFactory* por composição no seu construtor, sendo apenas necessário chamar os métodos necessários para a sua execução:

```
public Client(AbstractFactory factory) {
    this.factory = factory;
    this.window = factory.createWindow();
    this.button = factory.createButton();
}
```

4.2 *Factory Method*

Relativamente ao **Factory Method**, este diferencia-se, tal como foi referido anteriormente, ao declarar o objeto *AbstractFactory* como uma classe abstrata, ao invés de uma interface. Assim sendo, as suas sub-classes ficaram responsáveis pela atribuição dos objetos corretos.

NOTA: O método *factory*, neste caso, corresponde ao *createConcreteWindow*.

```
public abstract class AbstractFactory {
    public AbstractWindow createWindow() {
        AbstractWindow window = createConcreteWindow();
        window.use();
        return window;
    }
    protected abstract AbstractWindow createConcreteWindow();
}
```

```
public class LinuxFactory extends AbstractFactory {
    @Override
    protected AbstractWindow createConcreteWindow() {
        return new LinuxWindow();
    }
}
```

```
public class WindowsFactory extends AbstractFactory {
    @Override
    protected AbstractWindow createConcreteWindow() {
        return new WindowsWindow();
    }
}
```

Por fim, do lado do Cliente será apenas preciso instanciar o tipo de *Factory* que se pretende ao instanciar, através do respetivo construtor. De seguida, é invocado o método da *createWindow* que, por sua vez, utiliza o método *createConcreteWindow* definido pela própria classe.

```
class Client {
    AbstractFactory factory = new LinuxFactory();
    AbstractWindow window = factory.createWindow();
}
```