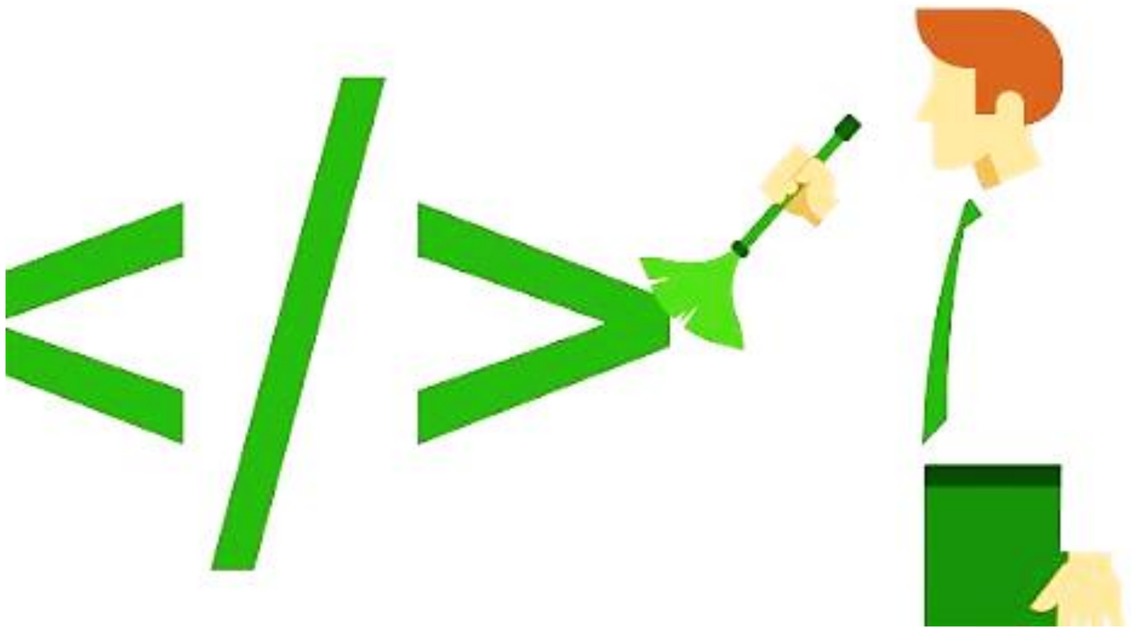
Course: DPR

Date: 20 December 2019

Design Patterns



Class: E-S41

Lecturer: Mariëlle Fransen

Student: Nadya Cheperkova

Nidhi Sharma

INTRODUCTION

Design pattern is a general reusable solution to a commonly occurring problem in software design. It is not a finished design that can be transformed directly into code. However, it is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns show the relationships and interactions between classes and objects, without specifying the final application classes or objects that are involved.

CREATIONAL PATTERNS

Creational design patterns provide various object creation mechanisms, which increase the flexibility and reusability of an existing code. Creational design patterns help in the production of a system that is independent of the creation, composition and representation of its objects. It is an important pattern because systems evolve to depend more on object composition than class inheritance. Hence, the hardcoded fixed set of behaviors is replaced with defined set of smaller fundamental behaviors that can be composed into many complex ones. Therefore, creating objects with particular behaviors requires more than just instantiating a class. The pattern is divided into two sub-patterns – class creational pattern which uses inheritance to vary the class that is instantiated and object creational pattern which will delegate instantiation to another object.

There are six different creational patterns - **Abstract Factory**, **Builder**, **Factory Method**, **Object Pool**, **Prototype**, **Singleton**.

ABSTRACT FACTORY

Abstract factory pattern is a creational design pattern that lets you produce families of related products without specifying their concrete classes. You use the abstract factory design pattern when your code needs to work with various families of related products, but you do not want it to depend on the concrete classes of those objects which is also known as loose coupling. They are either known beforehand or simply allowed for future extensibility.

APPLICATION

The application represents a clothing factory with three different brands (sub-factories) – Bershka, Zara and Stradivarius. The three of the sub-factories are creating T-shirts, Pants and Jacket. However, each of them offer different clothing style – Bershka factory has basic clothes including basic T-shirt, basic pants and basic jacket, Zara factory has formal clothes including formal T-shirt, formal pants and formal jacket, Stradivarius factory has sport clothes including sport T-shirt, sport pants and sport jacket. We created this application with the use **Abstract Factory** design pattern, which helped for the creation of one reusable solution. Hence, we do not need to repeat the code in all the concrete products because they just inherit from the abstract products interfaces. The main clothing factory is used to defer the creation of product objects. Normally, the concrete factories Zara, Bershka and Stradivarius are created at run-time and they create the different product objects.

The participants in the application are as follows:

- **Abstract Factory** *(IClothingFactory)* – declares an interface for the methods that create the abstract product objects

- **Concrete Factories** *(BershkaFactory, ZaraFactory, StradivariusFactory)* – implements the methods to create concrete product objects

- **Abstract Products** *(ITshirt, IPants, IJacket)* – declares an interface for the type of product objects

- **Concrete Products** *(BasicTshirt, BasicPants, BasicJacket, FormalTshirt, FormalPants, FormalJacket, SportTshirt, SportPants, SportJacket)* – implements the abstract product interface and defines a product objects to be created by the corresponding concrete factory

- **Client** *(Form1)* – uses only the interfaces declared by the abstract factories and abstract product classes

MAINTAINABILITY

The application can be easily maintained because it is created in such a way that it can be used in the real world. Abstract factory offers a maintainable and clean approach in order to make it easy for new developer to add or remove new products. The class of the concrete factories (Bershka, Zara, Stradivarius) appears only once in the application. Hence, it is easy to change the concrete factories and by changing it, it can use different product configurations. That is because the abstract factory (IClothingFactory) creates a complete family of products and the whole family can change at once. In our case we can change Bershka to H&M by switching the corresponding factory objects and recreating the interface that makes it easier for exchanging product family. Furthermore, because of the separation of the different classes (factories and products) and the single responsible principle, the code can be easily tested as we demonstrated in our application.

REUSABILITY

There are two reasons why we can say that this solution is reusable. The first one is because we create once the abstract products (T-shirt, Pants, Jacket) and the different types such as basic T-shit, formal pants and sport jacket use the methods from T-shirt, Pants and Jacket. The same rule applies to the factories, where we create once the interface IClothingFactory with the methods for the creation of different objects and then all the sub-factories Bershka, Zara, Stradivarius reuse all the methods.
Secondly, we can say that this solution is reusable because in this case it is used for clothing factory, but it can easily be reused for a car factory, food factory or any other kind of factory.

EXTENSIBILITY

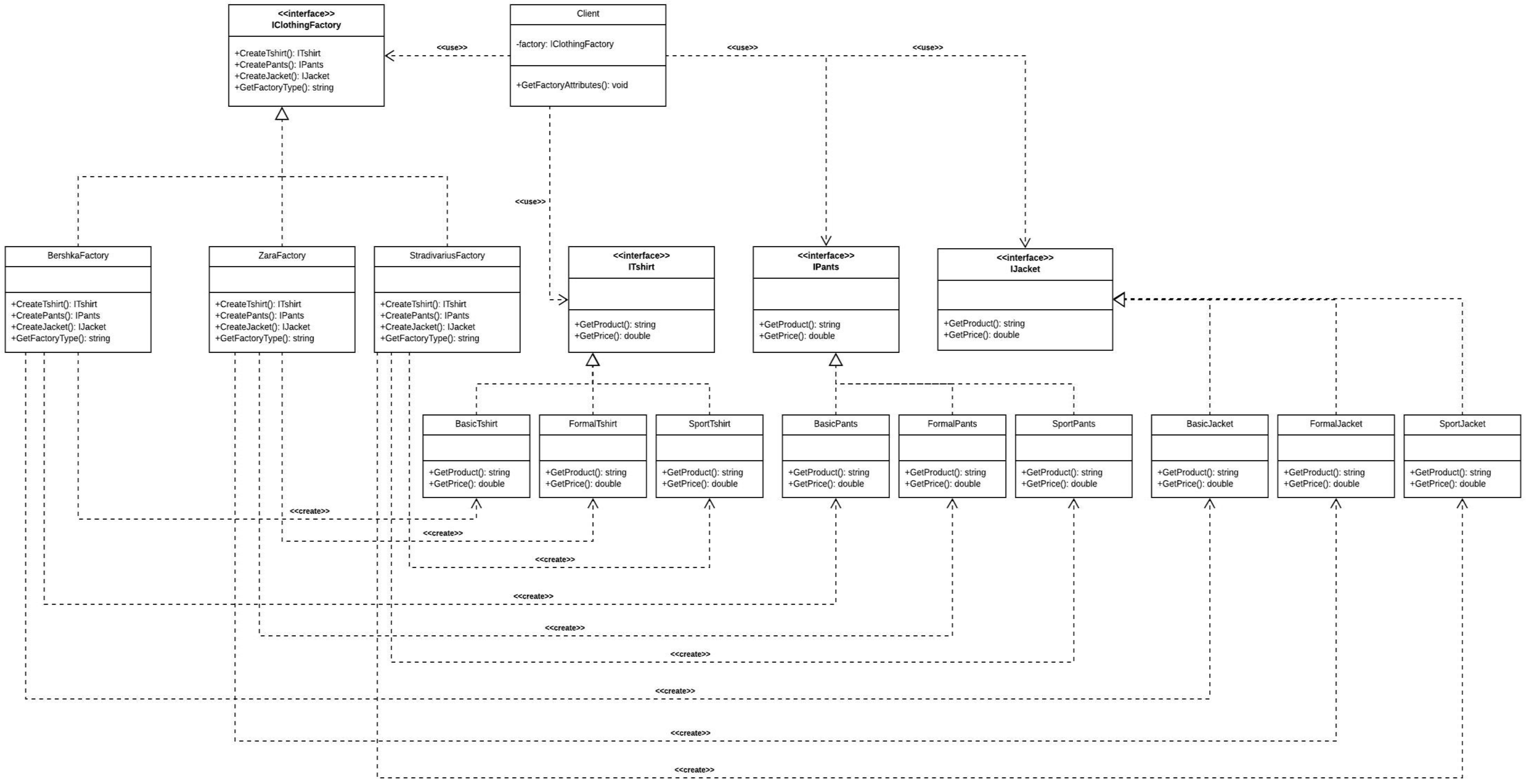
Supporting new kinds of products is not easy when the application is getting bigger because the abstract factory interface (IClothingFactory) specify the products that can be created and supporting new kinds of products require extending the factory interface and changing the sub-factories. In order to fix this drawback, we can add only one method for creating a product and just add a parameter to it to specify the kind of object to be created. However, with such an implementation the client will not be able to differentiate the classes of the product because

all the products are returned with the same abstract interface as given. Luckily, in the cases where the application is not that big, supporting new kinds of products will not be a problem. Therefore, we can easily add IShoes or IBikini to our application by adding a create methods (CreateShoes and CreateBikini) to the IClothingFactory and implement them in the different factories.
Moreover, supporting new factories is not problematic because we can simply add H&M sub-factory and inherit from IClothingFactory a complete family of products. Additionally, adding a new concrete product such as high-waisted pants or cotton jacket is also straightforward because it just inherits from the corresponding abstract product IPants and IJacket.

CONCLUSION

To conclude, as all design patterns **Abstract Factory** has some advantages and disadvantages. Even though it is a more complex solution because of the many interfaces and classes, it has huge applicability in today's world enterprises. The products from the factory are compatible with each other and tight coupling can be avoided between the concrete products and the client code. Abstract factory is one of the most commonly used design patterns and many times it is used in combination with other patterns.

UML DIAGRAM



<<interface>>
IClothingFactory
+CreateTshirt(): ITshirt
+CreatePants(): IPants
+CreateJacket(): IJacket
+GetFactoryType(): string
Client
-factory: IClothingFactory
+GetFactoryAttributes(): void
<<use>>
<<use>>
<<use>>
<<use>>
BershkaFactory
+CreateTshirt(): ITshirt
+CreatePants(): IPants
+CreateJacket(): IJacket
+GetFactoryType(): string
ZaraFactory
+CreateTshirt(): ITshirt
+CreatePants(): IPants
+CreateJacket(): IJacket
+GetFactoryType(): string
StradivariusFactory
+CreateTshirt(): ITshirt
+CreatePants(): IPants
+CreateJacket(): IJacket
+GetFactoryType(): string
<<interface>>
ITshirt
+GetProduct(): string
+GetPrice(): double
<<interface>>
IPants
+GetProduct(): string
+GetPrice(): double
<<interface>>
IJacket
+GetProduct(): string
+GetPrice(): double
BasicTshirt
+GetProduct(): string
+GetPrice(): double
FormalTshirt
+GetProduct(): string
+GetPrice(): double
SportTshirt
+GetProduct(): string
+GetPrice(): double
BasicPants
+GetProduct(): string
+GetPrice(): double
FormalPants
+GetProduct(): string
+GetPrice(): double
SportPants
+GetProduct(): string
+GetPrice(): double
BasicJacket
+GetProduct(): string
+GetPrice(): double
FormalJacket
+GetProduct(): string
+GetPrice(): double
SportJacket
+GetProduct(): string
+GetPrice(): double
<<create>>
<<create>>
<<create>>
<<create>>
<<create>>
<<create>>
<<create>>
<<create>>
<<create>>