|  |
| --- |
| Fontys ICT |
| Abstract Factory pattern |
| Design patterns |

|  |
| --- |
| Jan-Niklas Schneider, Georgiana Manolache  9-28-2016 |

Contents

[1 Introduction 3](#_Toc462782112)

[2 Factory pattern 3](#_Toc462782113)

[3 Implementation 5](#_Toc462782114)

[3.1 Explanation of classes 5](#_Toc462782115)

[3.2 Features 6](#_Toc462782116)

[4 Design choices 6](#_Toc462782117)

[5 Graphical User Interface 6](#_Toc462782118)

[6 Unit tests 7](#_Toc462782119)

[7 References 7](#_Toc462782120)

# Introduction

The goal of this document is to give an overview of the abstract factory pattern by giving an example implementation which displays a simplified car factory service. Furthermore, reusability, extensibility, and maintainability of this pattern are elaborated. Also, the implementation, its unit test and graphical user interface (GUI) are reviewed.

# Factory pattern

The Abstract Factory pattern is a software design pattern which separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.

The pattern allows building individual ‘factories’ with common functionalities without referring to concrete classes. A factory is the location of a concrete class in the code at which [objects are constructed](https://en.wikipedia.org/wiki/Object_creation). (Abstract factory pattern, n.d.).

The figure below depicts an UML diagram of the abstract factory pattern.

Firstly, an interface *AbstractFactory* which contains operations to create abstract products. One or many concrete factories inherit from the *AbstractFactory*, in this example *ConcreteFactory1*, *ConcreteFactory2.* They implement the operations to create concrete products.

An *AbstractProduct* is an interface for each type of product, in this case *AbstrcatProduct1*, *AbstrcatProduct2*. Each concrete product defines a product object to be created by the corresponding concrete factory.

Lastly, the *Client* uses interfaces declared by *AbstractFactory* and *AbstractProduct* classes

.

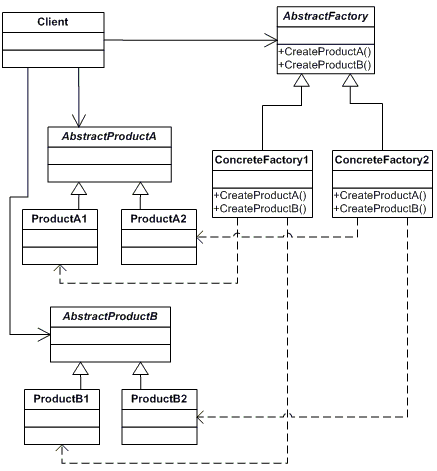


Figure 2‑1: UML diagram of abstract factory pattern (Data & Object Factory, LLC, n.d.)

# Implementation

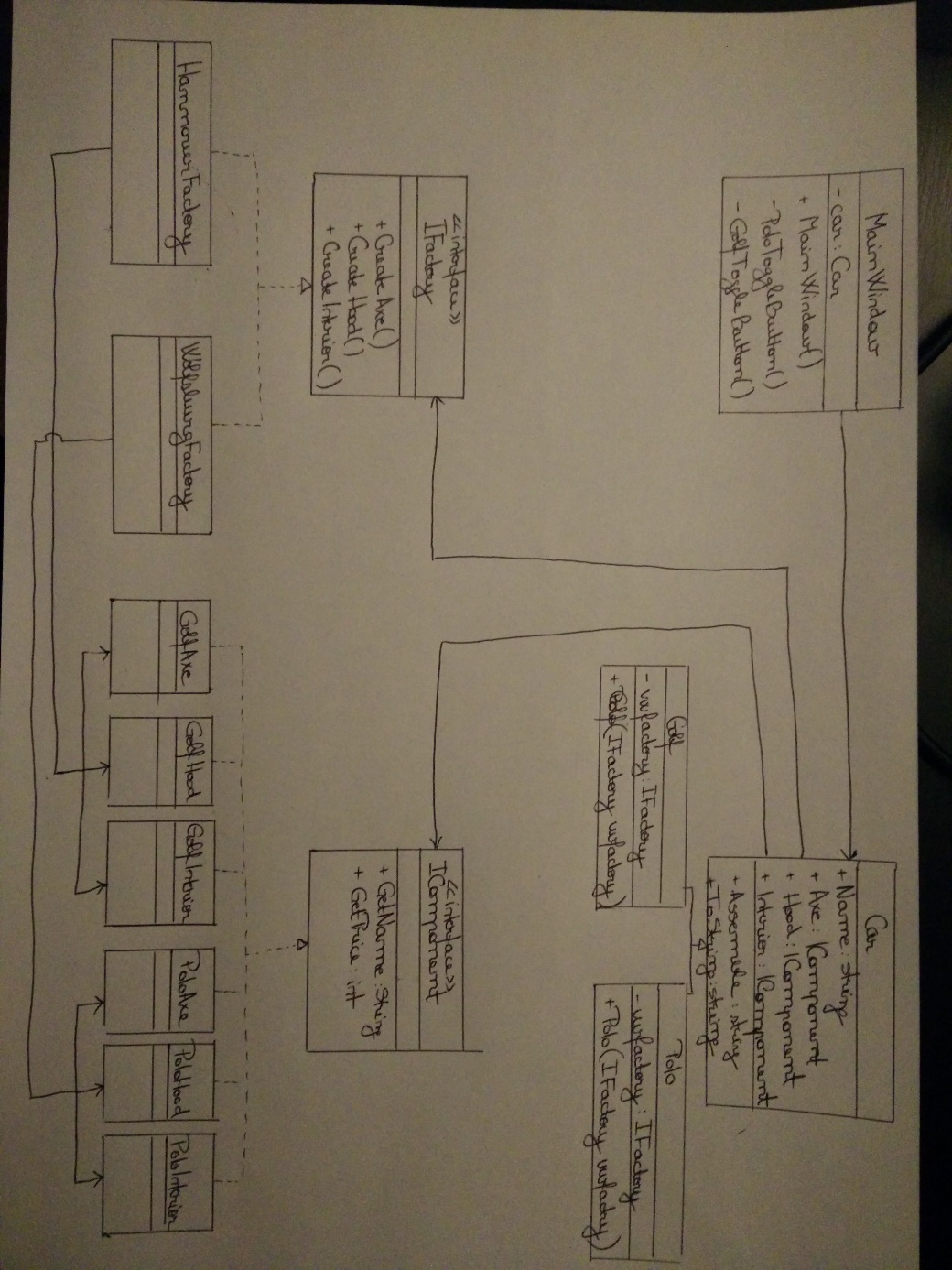
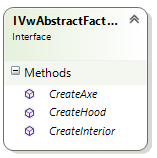


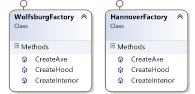
Figure 3‑1: UML diagram of the Abstract Factory pattern

Explanation of classes

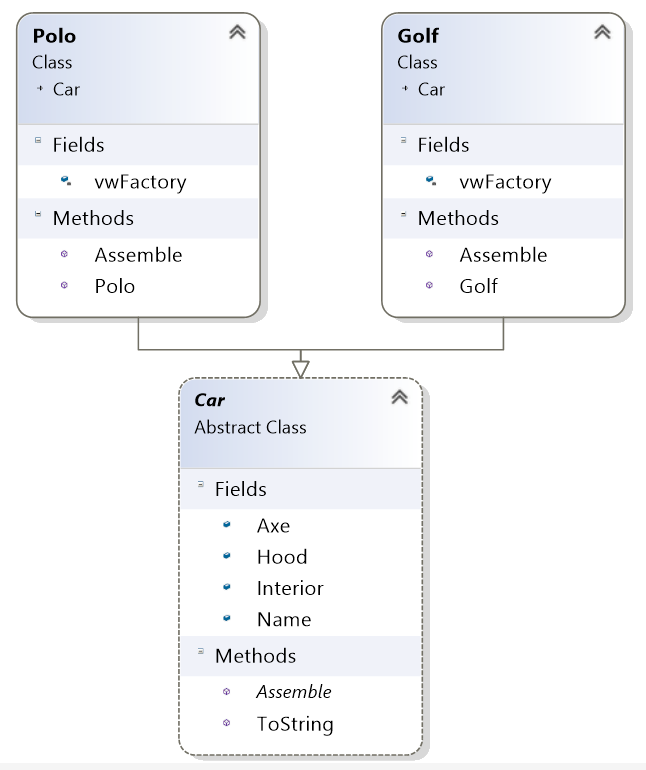
1. **IFactory** is an interface with three methods *CreateHood(), CreatAxe(),* and C*reateInteror()*. These are the common functionalities to all concrete factories.



1. two concrete components implement interface **IFactory**: **HannoverFactory**, **WolfsburgFactory**



1. Each factory products are implemented by is an interface: **IHood**, **IAxe**, **IInterior**
2. The concrete products implements their corresponding interfaces.The concrete products are based on the factory they belong to: **PoloHood**, **PoloAxe**, **PoloIntrior** are products specific to **WolfsburgFactory** whereas **GolfHood**, **GolfAxe** and **GolfInterior** are part of the **HannoverFactory**.
3. The **Car** class is an abstract class which assembles all car parts.



1. Concrete classes **Polo**, **Golf** implement abstract class **Car**.

## Features

The application has a simple and straightforward user interface. User can select a store and a car of choice. Selection can be cleared up.

# Design choices

The implementation of the assignment has been done with regard to reusability, extensibility, and maintainability.

The **reusability** of the abstract factory pattern is rather low. The one factory class *IVwAbstrcatFactory* is an interface that is specific to one application and cannot be reused without changes. However It could be reused in applications which concern car factories, given its functionality: *createHood*(), *createAxe*(), *createInterior*() are parts that exists in all cars. Since **reusability** of this pattern of this pattern is low the implementation of the *ConcreteProduct* interface has been done using an abstract class *Car*. By making use of an abstract class redundant and repetitive code is reduced.

In terms of **maintainability** the pattern is easy to maintain since logic is separated into independent units, such as *ConcreteFabrics* and *ConcreteProducts.* Also *if-else* statements which are notoriously difficult to maintain, are also not present.

The decorator pattern excels at **extensibility**. Functionality can be added by creating a new implementation of the *ConcreteFabric* or *ConcreteProduct* class which allows effortless integration of new functionality. Nevertheless, the pattern shows flaws when different kind of *ConcreteFabrics* or *ConcreteProducts* need to be introduced.

In order to reduce redundant code the abstract class *Car* is introduced which handles assembling the car for all derived classes.

# Graphical User Interface

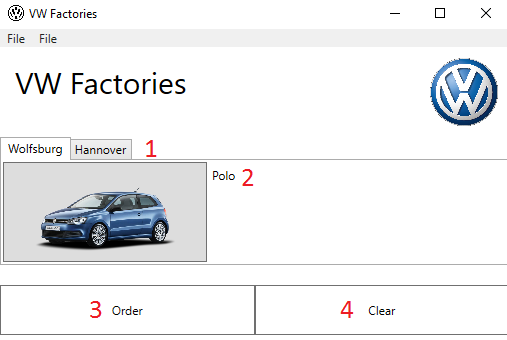


Figure 5‑1: Graphical user interface

The figure above depicts the user interface where red numbers indicate functionality or controls. More precisely these are:

1. Factories from where a car can be purchased (assembled).
2. Choose from one of the cars using car specific icon.
3. Order car.
4. Clear order.

# Unit tests

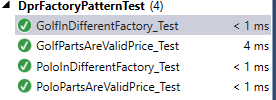
For each implemented Component unit tests have been defined to assert correct behavior. The test validate correct price of the components created in the factory, further, products have been created in different factories. Consequently, all test ran successfully.

Figure ‑: Unit test results

# References

*Abstract factory pattern*. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Abstract\_factory\_pattern

Data & Object Factory, LLC. (n.d.). *Abstrcat factory design pattern*. Retrieved from DoFactory: http://www.dofactory.com/net/abstract-factory-design-pattern