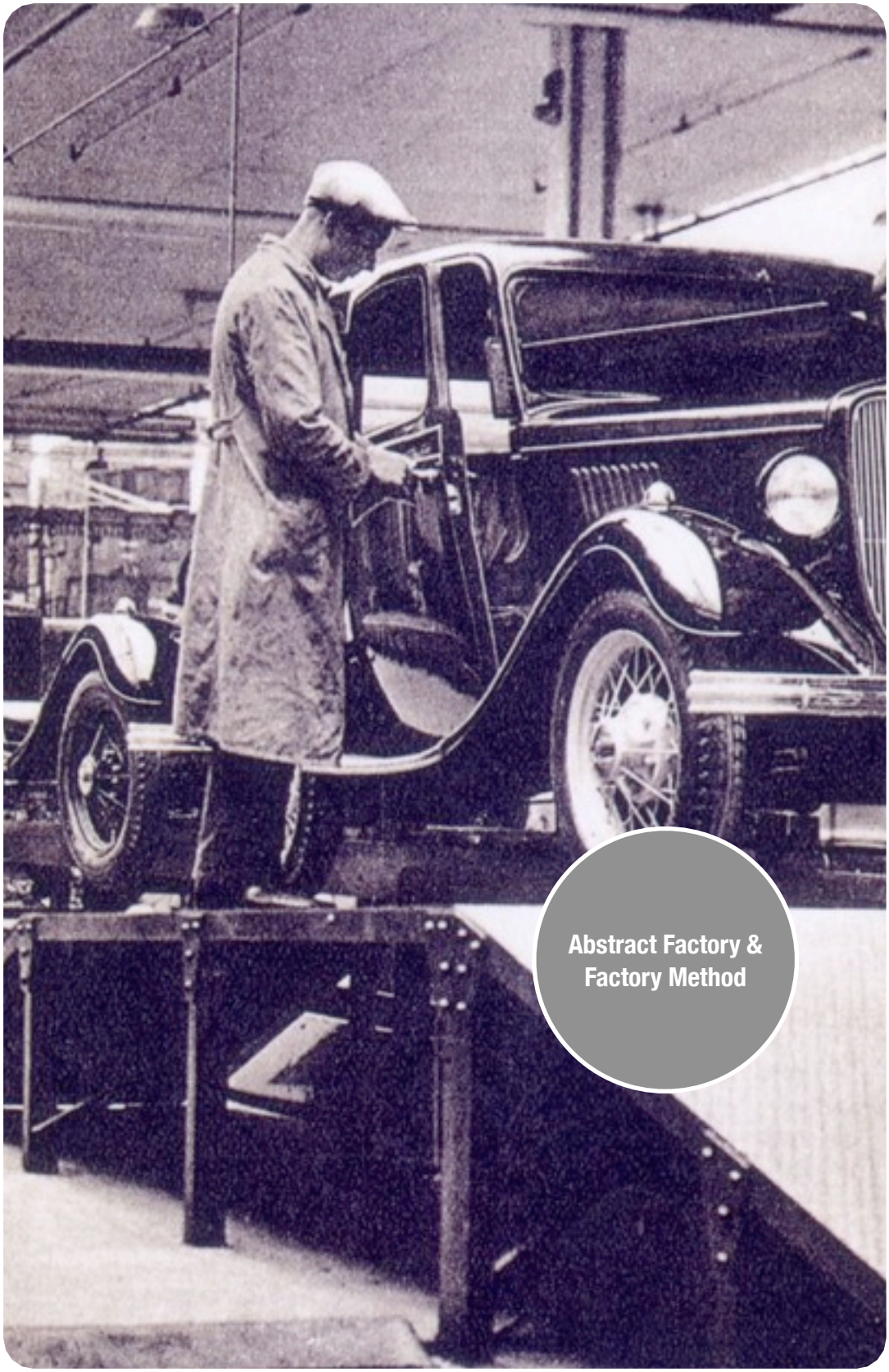
November

DESIGN PATTERN 2011



Abstract Factory & Factory Method

INHALT

BEARBEITET VON

Christoph Süsens

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1. Allgemeines

Design Patterns sind ein wichtiger Bestandteil der Softwareentwicklung und dienen dazu dem Programmierer einer Anwendung standardisierte Lösungsmittel für immer wieder kehrende Problemstellungen zur Verfügung zu stellen. Hauptziel des Einsatzes solcher Muster ist unter anderem die Sicherstellung der Flexibilität, Wartbarkeit, Zuverlässigkeit und Wiederverwendbarkeit der Anwendung für zukünftige Bedürfnisse. In dieser Dokumentation wird auf das Abstract Factory Muster, sowie das Factory Method Muster, welche beide zur Familie der Erzeugungsmuster zählen, eingegangen.[1]

2. Abstract Factory

2.1 Intent (Ziel)

Das Abstract Factory Design Pattern gehört zu der Familie der Erzeugungsmuster (Creational Patterns) und dient somit der Erzeugung von Objekten (Tabelle 1.0).

Die Objekterzeugung wird gekapselt und ausgelagert, um den Kontext unabhängig von der konkreten Implementierung zu halten, gemäß dem Grundsatz: „Programmiere auf die Schnittstelle, nicht auf die Implementierung!“.[2]

Nachfolgend wird von der Erzeugung von Produkten, die zu einer Familie gehören gesprochen. Die abstrakte Fabrik (Abstract Factory) als Produkthersteller kann dabei durch konkrete Fabriken erweitert werden, die jede für sich unterschiedliche Produkte unterschiedlichen Typs und in verschiedenen Kombinationen erzeugen kann.

---

[1] Judith Bishop: „C# 3.0 Entwurfsmuster“. O‘REILLY 1. Auflage 2008

[2] http://de.wikipedia.org/wiki/Erzeugungsmuster 04.10.11

Dem Client (Anwendung) bleibt dabei die Produktdefinition und ihre Klassennamen unbekannt, diesen Schritt übernimmt die Fabrik. Damit lässt sich die Produktfamilie leicht austauschen / anpassen, ohne die Struktur des Clients zu beeinflussen.[3]

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Tabelle 1.0 - Organisation der Design Patterns (modifiziert) [4]

Definition Abstract Factory[4]:

Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.

[3] Judith Bishop: „C# 3.0 Entwurfsmuster“. O‘REILLY 1. Auflage 2008 Seite 135

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software". Addison-Wesley. 1. Auflage 1996 Seite 107

2.2 Motivation (Beweggrund)

Betrachtet wird ein User Interface, das mehrere Erscheinungsstandards (look-and-feel standards), wie es der Presentation Manager (Abbildung 1.0) bereitstellt, beinhaltet. Unter Look-and-Feel wird dabei das Aussehen und die Handhabung einer grafischen Benutzeroberfläche verstanden.

Einige Beispiele hierfür sind die Wahl der Farben, Schriften, das Layout von grafischen Elementen, sowie ihre Reaktion auf Benutzereingaben.[5] Um das Erscheinungsbild flexibel zu halten, sollte der Code daher nicht auf eine bestimmte Look-and-Feel-Instanz festgelegt werden.

Um dieses Problem zu lösen wird eine abstrakte Look-and-Feel-Fabrik Klasse definiert, eine Schnittstelle, die die grundlegenden Erscheinungsstandards beinhaltet. Unterklassen, die von der abstrakten Look-and-Feel-Fabrik Klasse erben definieren dabei ein bestimmtes Look-and-Feel.

Die abstrakte Look-and-Feel-Fabrik (Abstract Factory) beinhaltet Methoden, die neue Instanzen des gewünschten Look-and-Feel mittels der Übergabe des entsprechenden Wertes erzeugen. Die Anwendung (Client) ruft damit nur die gewünschte Methode, die die entsprechende Instanz initiiert auf. Welche konkrete Klasse die Anwendung nun verwendet bleibt unbekannt. Damit bleibt die Anwendung unabhängig vom Look-and-Feel.[6]

Abbildung 1.0 - Beispiel Look-and-Feel[7]

---

[5] http://de.wikipedia.org/wiki/Look_and_Feel 04.10.11

[6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Elements of Reusable Object-Oriented Software". Addison-Wesley. 1. Auflage 1996 - Seite 99

[7] http://www.ergosign.de/images/uploads/look_feel_examples.jpg 04.10.11

2.3 Applicability (Einsetzbarkeit)

Das Abstract Factory Pattern findet Anwendung, wenn:

- ein System unabhängig von der Art der Erzeugung seiner Produkte arbeiten soll,
- ein System mit einer oder mehreren Produktfamilien konfiguriert werden soll,
- eine Gruppe von Produkten erzeugt und gemeinsam genutzt werden soll, oder
- in einer Klassenbibliothek die Schnittstellen von Produkten ohne deren Implementierung bereitgestellt werden sollen.

Eine typische Anwendung ist die Erstellung einer grafischen Benutzeroberfläche mit unterschiedlichen Themes (Design). Eine Abstrakte Fabrik vereinigt die Verantwortungen „Zusammenfassung der Objektgenerierung an einer Stelle“ und „Möglichkeit zu abstrakten Konstruktoren“[8]

2.4 Structure (Aufbau)

Den strukturellen Aufbau des UML-Diagramm für das Abstrakte Fabrik Muster (Abbildung 1.1) bilden eine Reihe von Teilnehmern. Anhand des Beispiels der abstrakten Look-and-Feel Factory werden in der nachfolgenden Tabelle (Tabelle 1.1) die Teilnehmer inhaltlich eingeordnet.

Teilnehmer	Definition
IFactory (AbstractFactory)	Interface mit Create-Operationen für jedes der abstrakten Look-and-Feel Produkte
Factory1, Factory2	Konkrete Look-and-Feel Fabriken z. B. MacOSXFactory, Implementieren alle Erzeugungsoperationen von der abstrakten Look-and-Feel Fabrik
IProductA (AbstractProduct) IProductB (AbstractProduct)	Interface, z. B. IScrollBarA für eine bestimmte Produktart mit seinen eigenen Operationen
ProductA1, ProductA2, ProductB1, ProductB2	Klassen, die das Interface von IScrollBarA implementieren und konkrete Produktobjekte definieren, die von den entsprechenden Fabriken (MacOSXFactory oder WindowsFactory) erzeugt werden

[8] http://de.wikipedia.org/wiki/Abstrakte_Fabrik 04.10.11

Teilnehmer	Definition
Client	Anwendung, die nur auf die Interfaces IFactory (AbstractFactory) und IProduct (AbstractProduct) zugreift

Tabelle 1.1 - Einordnung Teilnehmer Abstract Factory Pattern (modifiziert)[9]

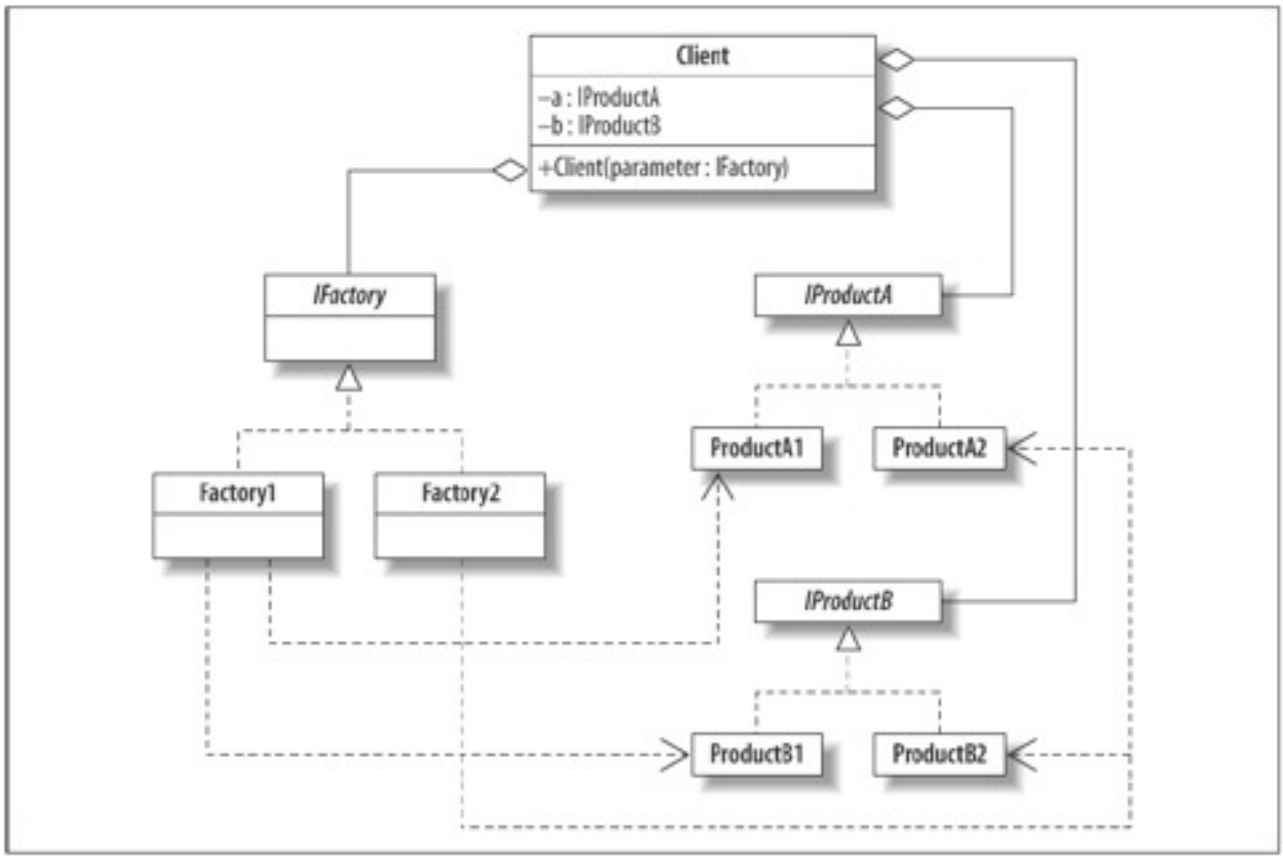


Abbildung 1.1 - UML-Diagramm für das Abstrakte-Fabrik-Muster[10]

[9] Judith Bishop: „C# 3.0 Entwurfsmuster“. O‘REILLY 1. Auflage 2008 Seite 136/137

2.5 Consequences (Tragweite)

Der Einsatz des Abstract Factory Pattern beinhaltet eine Reihe von Vorteilen, jedoch auch Nachteile, auf die nachfolgend näher eingegangen werden soll.

1. Isolation konkreter Klassen

Das Abstract Factory Pattern sorgt durch die Koordination der Klassen, die Objekte in Anwendungen erzeugen können, für einen allgemein gültigen Code. Dabei wird, die Anwendung nicht auf eine spezielle Klasse festgelegt oder gar die Verantwortung zur Erzeugung neuer Instanzen vereint.

2. Erleichterung des Austausches von Produktfamilien

Der Austausch von Produktfamilien kann problemlos erfolgen, da sich die Anwendung nur auf die Interfaces AbstractFactory (IFactory) und AbstractProduct (IProduct) stützt. Damit bleibt der Code stabil und gegenüber späteren Änderungen flexibel.

3. Konsistenz zwischen Objekten

Die Zusammengehörigkeit von Produktobjekten muss sichergestellt werden, es dürfen nur Objekte gleicher Produktfamilien verwendet werden. Dies kann mittels Abstract Factory Pattern erreicht werden.

4. Integrierung neuer Produkte schwierig

Die Erweiterung neuer Produkte wird durch das Abstract Factory Interface erschwert, da es genau definiert, welche Produkte erzeugt werden können. Sollen zu einem späteren Zeitpunkt neue Produkte integriert werden, ist ein erheblicher Code-Anpassungsaufwand der Abstract Factory Klasse und ihrer Unterklassen notwendig. Daher sollte vor der Implementierung eine genaue Analyse stattfinden. [10] [11]

---

[10] http://www.philipphauer.de/study/se/design-pattern/abstract-factory.php#vorteile 07.10.11

[11] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Elements of Reusable Object-Oriented Software". Addison-Wesley. 1. Auflage 1996 - Seite 102

2.6 Implementation (Einbettung)

Von der Theorie in die Praxis, wie genau wird ein Abstract Factory Pattern in C# implementiert? Anhand der Konsolenanwendung (ImplementationAbstractFactoryPattern) soll der Einsatz des Abstract Factory Design Pattern dargestellt werden. Die zu erzeugenden Teilnehmer-Klassen (Abbildung 1.2) befinden sich im Abschnitt Controller.

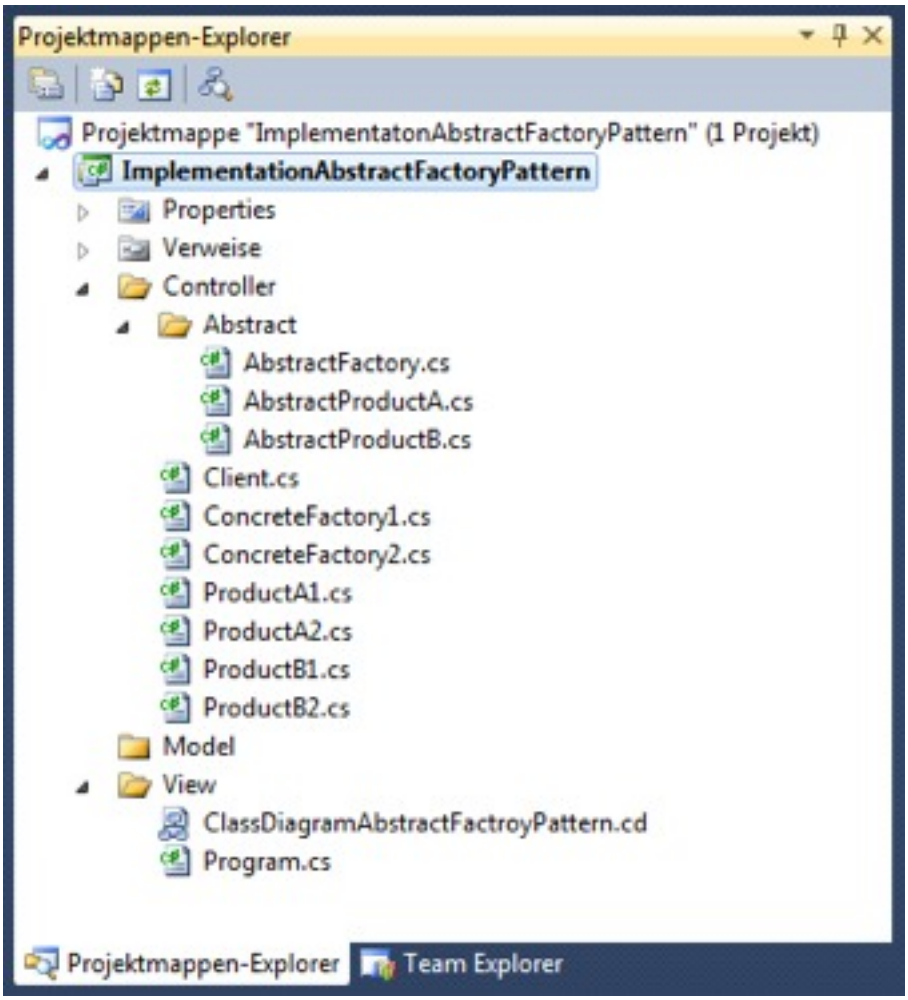


Abbildung 1.2 - Übersicht Teilnehmer-Klassen Abstract Factory Pattern in Visual Studio

Quellcode[12]:

```
namespace ImplementationAbstractFactoryPattern.Controller
{
    /// <summary>
    /// The AbstractFactory abstract class
    /// </summary>

    public abstract class AbstractFactory
```

[12] http://www.dofactory.com/Patterns/PatternAbstract.aspx#_self1 08.10.11

```
    {
        /// <summary>
        /// abstract methods which extends the real factory
        /// </summary>
        /// <returns></returns>
        public abstract AbstractProductA createProductA();
        public abstract AbstractProductB createProductB();
    }
}


namespace ImplementationAbstractFactoryPattern.Controller
{
    /// <summary>
    /// The AbstractProductA abstract class
    /// </summary>
    public abstract class AbstractProductA
    {
        //AbstractProductA has no methods to extend
    }
}


namespace ImplementationAbstractFactoryPattern.Controller
{
    /// <summary>
    /// The AbstractProductB abstract class
    /// </summary>

    public abstract class AbstractProductB
    {
        /// <summary>
        /// abstract method to show interaction with AbstractProductA
        /// returns object type
        /// </summary>
        /// <param name="a"></param>
        public abstract void interact(AbstractProductA a);
    }
}


namespace ImplementationAbstractFactoryPattern.Controller
{
    /// <summary>
    /// The ConcreteFactory1 class extends AbstractFactory
    /// </summary>

    public class ConcreteFactory1 : AbstractFactory
    {
        /// <summary>
        /// override the abstract method createProductA from datatype
AbstractProductA
```

```
        /// </summary>
        /// <returns>ProductA1</returns>
        public override AbstractProductA createProductA()
        {
            return new ProductA1();
        }
        /// <summary>
        /// override the abstract method createProductB from datatype
AbstractProductB
        /// </summary>
        /// <returns>ProductB1</returns>
        public override AbstractProductB createProductB()
        {
            return new ProductB1();
        }
    }
}


namespace ImplementationAbstractFactoryPattern.Controller
{
    /// <summary>
    /// The ConcreteFactory2 class extends AbstractFactory
    /// </summary>

    public class ConcreteFactory2 : AbstractFactory
    {
        /// <summary>
        /// override the abstract method createProductA from datatype
AbstractProductA
        /// </summary>
        /// <returns>ProductA2</returns>
        public override AbstractProductA createProductA()
        {
            return new ProductA2();
        }
        /// <summary>
        /// override the abstract method createProductB from datatype
AbstractProductB
        /// </summary>
        /// <returns>ProductB2</returns>
        public override AbstractProductB createProductB()
        {
            return new ProductB2();
        }
    }
}


namespace ImplementationAbstractFactoryPattern.Controller
{
    /// <summary>
```

```
    /// The ProductA1 class extends AbstractProductA class
    /// </summary>
    public class ProductA1 : AbstractProductA
    {
        //AbstractProductA has no methods to extend
    }
}


namespace ImplementationAbstractFactoryPattern.Controller
{
    /// <summary>
    /// The ProductA2 class extends AbstractProductA class
    /// </summary>
    public class ProductA2 : AbstractProductA
    {
        //AbstractProductA has no methods to extend
    }
}

using System;

namespace ImplementationAbstractFactoryPattern.Controller
{
    /// <summary>
    /// The ProductB1 class extends AbstractProductB class
    /// </summary>
    public class ProductB1 : AbstractProductB
    {
        /// <summary>
        /// override the method of AbstractProductB class
        /// shows the interaction to AbstractProductA
        /// returns type object
        /// </summary>
        /// <param name="a"></param>
        public override void interact(AbstractProductA a)
        {
            Console.WriteLine(this.GetType().Name + " interacts with " +
a.GetType().Name);
        }
    }
}

using System;

namespace ImplementationAbstractFactoryPattern.Controller
{
    /// <summary>
    /// The ProductB1 class extends AbstractProductB class
    /// </summary>
    public class ProductB2 : AbstractProductB
    {
```

```
        /// <summary>
        /// override the method of AbstractProductB class
        /// shows the interaction to AbstractProductA
        /// returns type object
        /// </summary>
        /// <param name="a"></param>
        public override void interact(AbstractProductA a)
        {
            Console.WriteLine(this.GetType().Name + " interacts with " +
a.GetType().Name);
        }
    }
}


namespace ImplementationAbstractFactoryPattern.Controller
{
    /// <summary>
    /// The Client class. Interaction environment for the products.
    /// </summary>
    public class Client
    {
        //variable of abstractProductA and abstractProductB
        private AbstractProductA abstractProductA;
        private AbstractProductB abstractProductB;

        //Constructor
        public Client(AbstractFactory factory)
        {
            //declare the constructor for the real factory with the
method
            //to create the product
            abstractProductA = factory.createProductA();
            abstractProductB = factory.createProductB();
        }

        //method to show interaction between the products
        public void run()
        {
            //productB(x) interacts with productA(x)
            abstractProductB.interact(abstractProductA);
        }

    }
}

using System;

namespace ImplementationAbstractFactoryPattern.Controller
{
    /// <summary>
    /// Program startup class for Structural
```

```
    /// Abstract Factory Design Pattern.
    /// </summary>
    public class Program
    {
        static void Main()
        {
            // Abstract factory #1
            AbstractFactory factory1 = new ConcreteFactory1();
            Client client1 = new Client(factory1);
            client1.run();

            // Abstract factory #2
            AbstractFactory factory2 = new ConcreteFactory2();
            Client client2 = new Client(factory2);
            client2.run();

        }
    }
}
```

Abbildung 1.3 - Ausgabeergebnis ImplementationAbstractFactoryPattern

2.7 Sample Code (Beispiel programmieren)

Am Beispiel des Look-And-Feels (siehe Punkt 2.2) soll nun verinnerlicht werden, wie das Abstract Factory Design Pattern in einer Windows Forms Anwendung umgesetzt werden kann. Benötigt werden hierzu die in der Abbildung 1.5 dargestellten Klassen mit ihren Methoden.

Das Textfeld Ergebnis soll ausgeben welcher Konstruktor (Fabrik) aufgerufen wurde. Im Textfeld Hintergrundaktivitäten soll dargestellt werden, welche Methoden aufgerufen wurden.

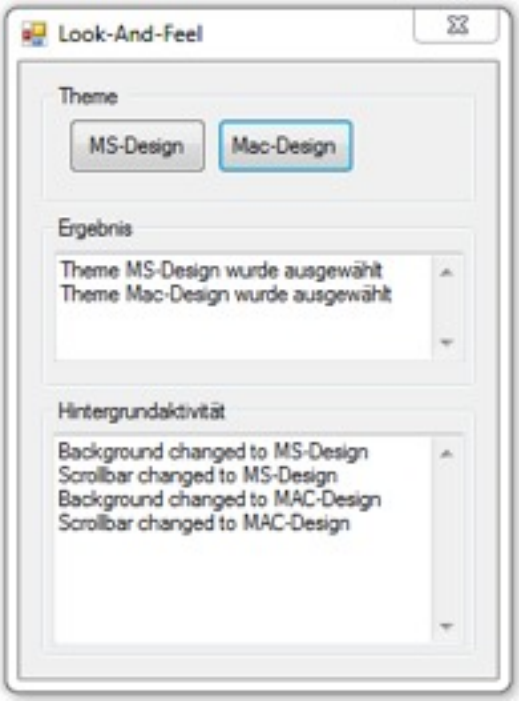


Abbildung 1.4 - Projekt „Look-And-Feel“ Ausgabeergebnis

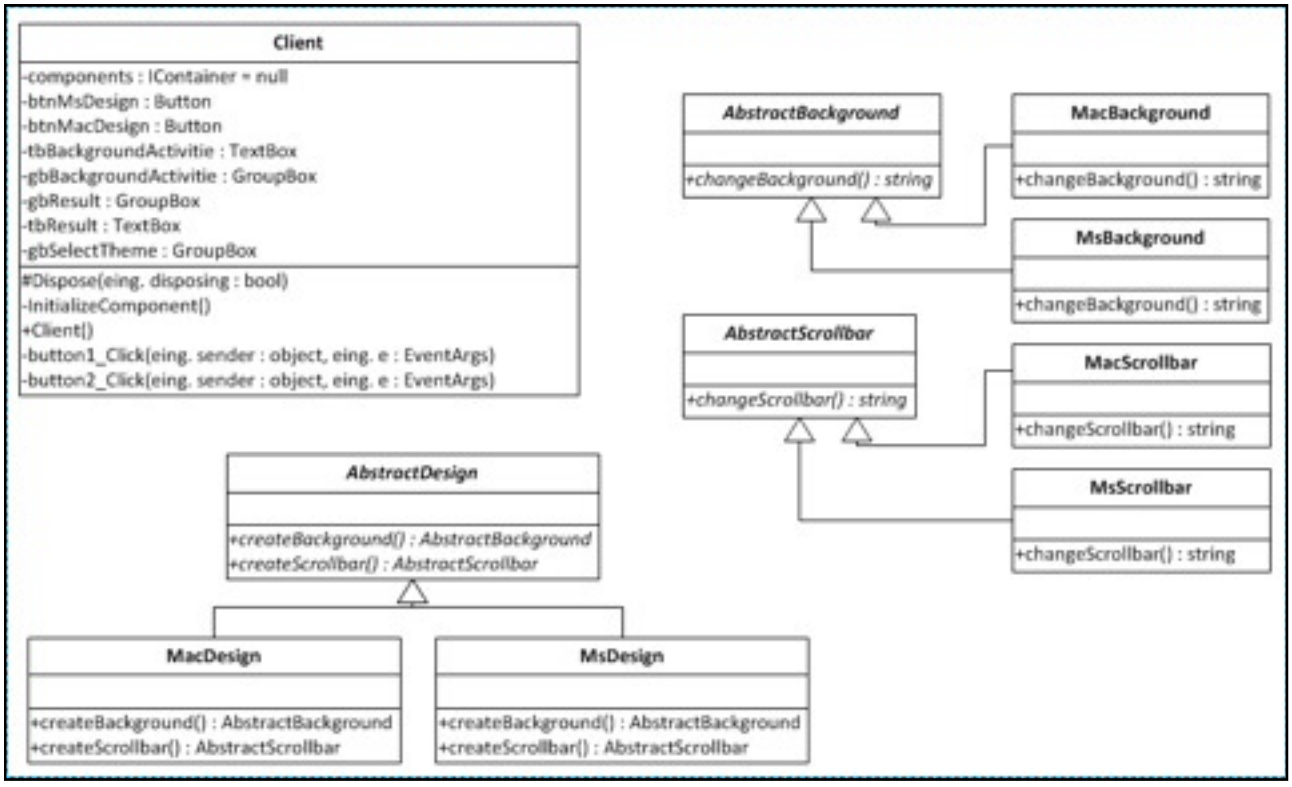


Abbildung 1.5 - UML-Diagramm Projekt „Look-And-Feel“

3. Factory Method

Ebenfalls zur Familie der Erzeugungsmuster (Creational Pattern) gehört das Factory Method Design Pattern (Tabelle 2.0). Es ist ein klassenbasiertes Erzeugungsmuster und behandelt damit die Beziehungen zwischen den Klassen, die zur Übersetzungszeit festgelegt werden.[13]

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Tabelle 2.0 - Organisation der Design Patterns (modifiziert)[14]

3.1 Intent (Ziel)

Aufgabe der Factory Method ist es Objekte zu erzeugen, jedoch die Unterklassen entscheiden zu lassen, welche Klasse genau instanziiert wird. Dabei kann es mehrere Unterklassen geben, die das Interface implementieren. Mittels der übergebenen

[13] http://www.uni-siegen.de/fb12/ws/lehre/lehre11/ei2_2011/material/v06.pdf 10.10.11

[14] http://wdict.net/img/creational+pattern.jpg 04.10.11

Informationen entscheidet die Factory Method, welche Subklasse vom Client (Anwendung) erzeugt wird.[15]

Definition Factory Method[16]:

> *Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse die Erzeugung von Objekten an Unterklassen zu delegieren.*

3.2 Motivation (Beweggrund)

Betrachtet wird ein ShopSystem, das vorrangig Microsoft Office (sog. Factory) vertreibt. Kauft ein Kunde diese Applikation, so wird die gewünschte Software ausgewählt, instanziiert und zurückgeben. Dieser Entwurf zeigt jedoch Schwachstellen, wie eine geringe Wiederverwendbarkeit, und zudem wird gegen das Offen/Geschlossen-Prinzip verstoßen. Bei Erweiterungen muss der bestehende Code angepasst werden.

Daher ist es notwendig einige Verbesserungen im Entwurf vorzunehmen. Berücksichtigt werden dabei das Offen/Geschlossen-Prinzip (Entwürfe sollten für Erweiterungen offen, aber für Veränderungen geschlossen sein) und das Prinzip des Kapseln (Identifiziere jene Aspekte, die sich ändern, und trenne sie von jenen, die konstant bleiben.). Erweiterungen z. B. neue Office-Programme, sollen ohne Änderungen am bestehenden Code integriert werden können.

Um das Kapseln-Prinzip anzuwenden, wird der veränderliche Teil, das Instanziieren von Applikationen, in eine separate Klasse, Creator genannt, ausgelagert. Dieser Neuentwurf zeigt nun eine Reihe von Stärken, wie Kohäsionsgewinnung durch Trennung der Objekterstellung von der Objektverarbeitung, Wiederverwendbarkeit der Factory und eine zentrale Stelle für Wartung und Erweiterung.

---

[15] Judith Bishop: „C# 3.0 Entwurfsmuster“. O‘REILLY 1. Auflage 2008 Seite 121

[16] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software". Addison-Wesley. 1. Auflage 1996

Leider ist auch dieser Entwurf noch nicht ganz vollkommen. So können zwar neue Sortimente (Factorys), wie z. B. Apple IWork (AppleIWorkFactory) oder Sun OpenOffice (SunOpenOfficeFactory), integriert werden, jedoch sind die Objektverarbeitung und die Objekterstellung nun zu stark entkoppelt. Um die Verarbeitung von der Herstellung zu trennen und gleichzeitig zu koppeln, wird die **Factory Method** in der Klasse Creator angewandt.

Das Shop System wird nun um die factoryMethod() Methode erweitert. Damit wird genau die Definition des Factory Method Design Pattern erfüllt. Das Shop System (Client) enthält zwar die Operationen zur Erzeugung von Objekten, doch allein die Subklasse (Creator) entscheidet, welches das zu erzeugende Objekt ist. Durch die Implementierung der factoryMethod() Methode.

Kurz Zusammenfassung:

Der Client (Anwendung) instanziiert die Klasse Creator, welche die Methode factoryMethod() beinhaltet und holt sich über diese Methode das gewünschte Programm (Factory). Es instanziiert damit die entsprechende Factory. Es kennt dabei nur die Schnittstelle (IOfficeProgram), aber nicht die konkreten Produkte. Die kennen jedoch die Subklassen und erzeugen diese.[17]

3.3 Applicability (Einsetzbarkeit)

Das Factory Method Pattern findet Anwendung, wenn:

- die Klasse neuer Objekte bei der Objekterzeugung unbekannt ist,
- Unterklassen sollen Klassen neuer Objekte bestimmen,
- Klassen delegieren Verantwortlichkeiten an Unterklassen, das Wissen um Unterklassen soll dabei aber lokal bleiben.[18]

[17] http://www.philipphauer.de/study/se/design-pattern/factory-method.php 11.10.11

[18] www.complang.tuwien.ac.at/franz/objektorientiert/oop06-10.pdf 16.10.11

3.4 Structure (Aufbau)

Den strukturellen Aufbau des UML-Diagramm für das Factory Method Design Pattern (Abbildung 2.0) bilden eine Reihe von Teilnehmern. Anhand des Beispiels des Shop Systems werden in der nachfolgenden Tabelle (Tabelle 2.1) die Teilnehmer inhaltlich eingeordnet.

Teilnehmer	Definition
IProduct	Interface für Produkte z. B. IOfficeProgram
ProductA & ProductB	Klassen die IProduct implementieren z. B. MicrosoftOfficeFactory, AppleIWorkFactory
Creator	Stellt die FactoryMethod bereit z. B. factoryMethod()
FactoryMethod (Client)	Entscheidet welche Klasse instanziiert werden soll z. B. holeApp()

Tabelle 2.1 - Einordnung Teilnehmer Abstract Factory Pattern (modifiziert)[19]

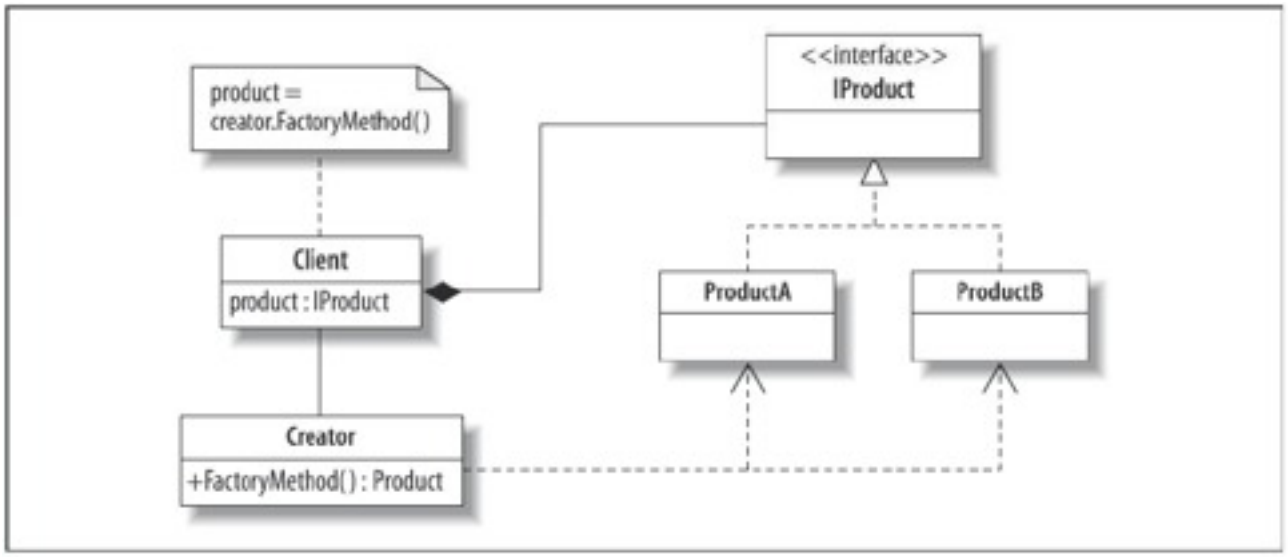


Abbildung 2.0 - UML-Diagramm für das Factory-Method-Muster [20]

[19] Judith Bishop: „C# 3.0 Entwurfsmuster“. O‘REILLY 1. Auflage 2008 Seite 122

[20] Judith Bishop: „C# 3.0 Entwurfsmuster“. O‘REILLY 1. Auflage 2008 Seite 122

3.5 Consequences (Tragweite)

Der Einsatz des Factory Method Pattern beinhaltet eine Reihe von Konsequenzen, auf die näher eingegangen werden soll.

1. Erreichung der Unabhängigkeit von anwendungsspezifischen Klassen

In dem der Client gegen die Schnittstelle (IProduct) programmiert, verliert das Factory Pattern die Bindung an anwendungsspezifische Klassen. Daher ist es flexibler und kann mit jeder benutzerdefinierten konkreten Produktklasse (ConcreteProduct classes) kommunizieren.[21]

2. Verbindung paralleler Klassenhierarchien

Als Vorteil sei zu erwähnen, dass das Factory Method Pattern parallele Klassenhierarchien verbinden kann. Voraussetzung dafür ist jedoch, dass jede Hierarchie eine FactoryMethod enthält, womit sich Instanzen von Subklassen erzeugen lassen.[22]

3.6 Implementation (Einbettung)

Nachfolgend wird gezeigt, wie genau das Factory Method Design Pattern in der Programmiersprache C# implementiert wird. Dazu wird eine Konsolenanwendung mit dem Titel „ImplementationFactoryMethodPattern“ erstellt. Die zu erzeugenden Teilnehmer-Klassen können der Abbildung 2.1 entnommen werden.[23]

---

[21] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Elements of Reusable Object-Oriented Software". Addison-Wesley. 1. Auflage 1996 - Seite 123

[22] Judith Bishop: „C# 3.0 Entwurfsmuster“. O‘REILLY 1. Auflage 2008 Seite 125

[23] http://www.dofactory.com/Patterns/PatternFactory.aspx#_self1 16.10.11

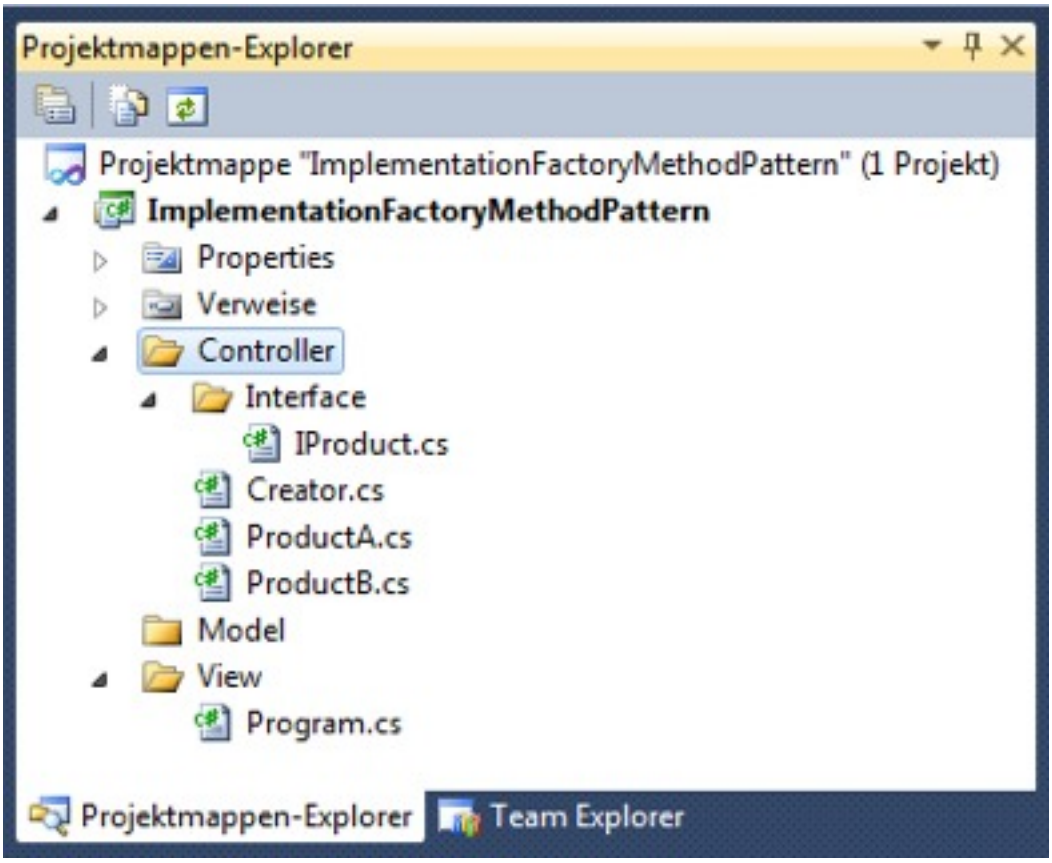


Abbildung 2.1 - Übersicht Teilnehmer-Klassen Factory Method Pattern in Visual Studio

Quellcode[24]:

```
namespace ImplementationFactoryMethodPattern
{
    /// <summary>
    /// The interface for Products
    /// </summary>
    public interface IProduct
    {
        string product();
    }
}

namespace ImplementationFactoryMethodPattern
{
    /// <summary>
    /// Declares the factory method
    /// </summary>
    public class Creator
    {
        /// <summary>
        /// Decide which class to instantiate
```

[24] Judith Bishop: „C# 3.0 Entwurfsmuster“. O‘REILLY 1. Auflage 2008 Seite 123 (modifiziert)

```
        /// </summary>
        public IProduct factoryMethod(char achar)
        {
            IProduct aIProduct = null;
            //The factory method decides on the basis
            //of 'achar' which product class to instatiates.
            switch (achar)
            {
                //In case of 'achar' = a -> creates an instance of
ProductA
                case 'a' :
                    aIProduct = new ProductA();
                    break;
                //In case of 'achar' = b -> creates an instance of
ProductB
                case 'b' :
                    aIProduct = new ProductB();
                    break;
            }
            return aIProduct;
        }
    }
}

namespace ImplementationFactoryMethodPattern
{
    /// <summary>
    /// Class ProductA, implement IProduct
    /// </summary>
    public class ProductA : IProduct
    {
        public string product()
        {
            return "Class ProductA - Method Product";
        }
    }
}

namespace ImplementationFactoryMethodPattern
{
    /// <summary>
    /// ClassProductB, implement IProduct
    /// </summary>
    public class ProductB : IProduct
    {
        public string product()
        {
            return "Class ProductB - Method Product";
        }
    }
}
```

```
using System;

namespace ImplementationFactoryMethodPattern
{
    /// <summary>
    /// The client tests the implementation of the
    /// factory method pattern
    /// </summary>
    public class Client
    {
        public static void Main(string[] args)
        {
            Creator c = new Creator();
            IProduct aIProduct = null;

            //Calls the FactoryMethod in the Creator class
            //The FactoryMethod creates a new instance of ProductA
            //and saves it in product.
            aIProduct = c.factoryMethod('a');
            //Calls the method Product in ProductA class
            //and print it on the console.
            Console.WriteLine(aIProduct.product());

            //Calls the FactoryMethod in the Creator class
            //The FactoryMethod creates a new instance of ProductB
            //and saves it in product.
            aIProduct = c.factoryMethod('b');
            //Calls the method Product in ProductB class
            //and print it on the console.
            Console.WriteLine(aIProduct.product());
        }
    }
}
```

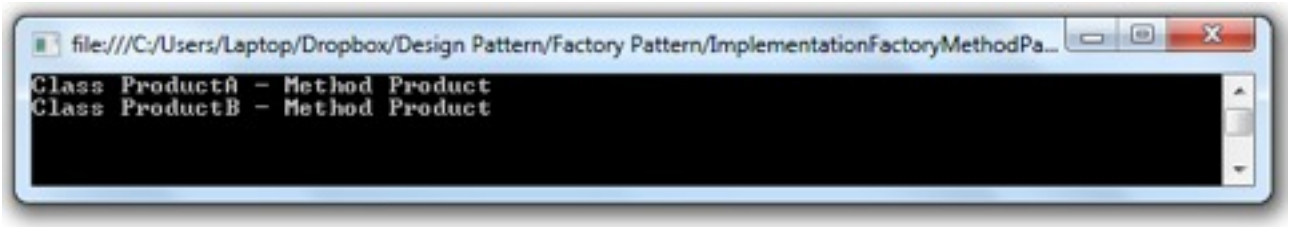


Abbildung 2.2 - Ausgabeergebnis ImplementationFactoryMethodPattern

3.7 Sample Code (Beispiel programmieren)

Am Beispiel des Shop Systems (siehe Punkt 3.2) soll nun verinnerlicht werden wie das Factory Method Design Pattern in einer Windows Forms Anwendung umgesetzt werden kann. Benötigt werden hierzu die in der Abbildung 2.3 dargestellten Klassen mit ihren Methoden.

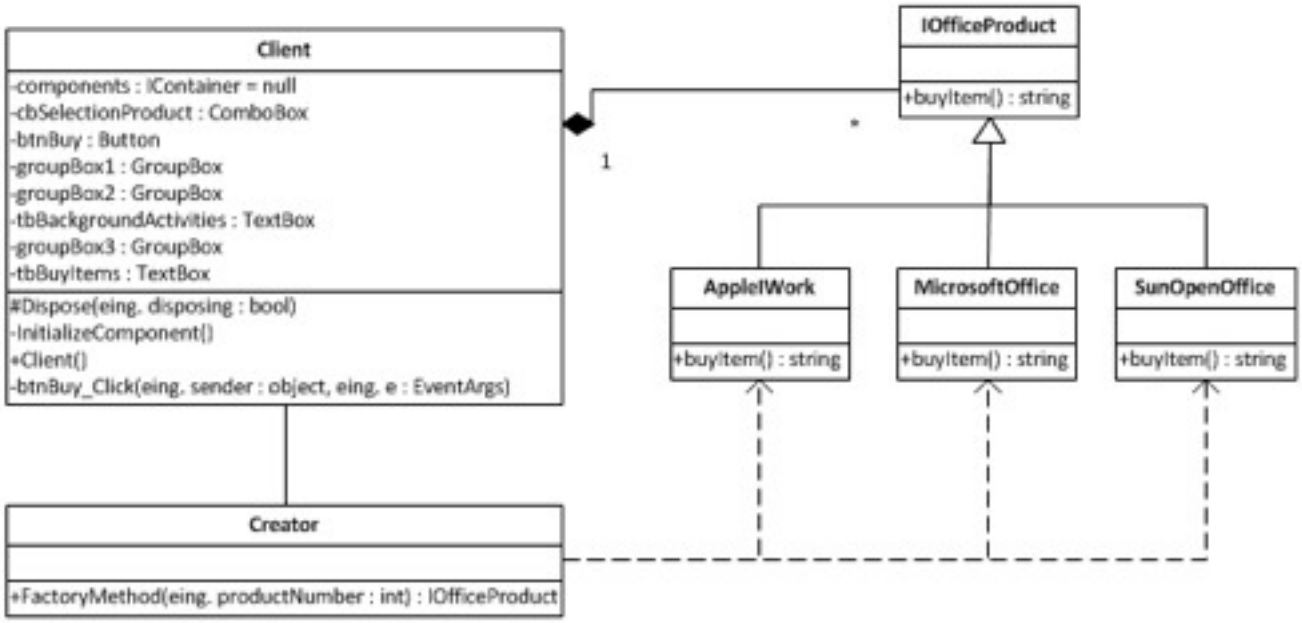


Abbildung 2.3 - UML-Diagramm Projekt „ShopSystem“

Die Methode buyItem() soll als Ergebnis das entsprechende Produkt zurückgeben und dieses im Textfeld des Warenkorb ausgeben. Im Textfeld Hintergrundaktivitäten soll dargestellt werden, welcher Konstruktor aufgerufen wurde.

Abbildung 2.4 - Projekt „ShopSystem“ Ausgabeergebnis

4. Literaturverzeichnis

- Judith Bishop: „C# 3.0 Entwurfsmuster“. O‘REILLY 1. Auflage 2008
- Erich Gamma: "Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software". Addison-Wesley. 1. Auflage 1996
- http://de.wikipedia.org/wiki/Erzeugungsmuster 04.10.11
- http://de.wikipedia.org/wiki/Look_and_Feel 04.10.11
- http://www.ergosign.de/images/uploads/look_feel_examples.jpg 04.10.11
- http://de.wikipedia.org/wiki/Abstrakte_Fabrik 04.10.11
- http://www.philipphauer.de/study/se/design-pattern/abstract-factory.php#vorteile 07.10.11
- http://www.dofactory.com/Patterns/PatternAbstract.aspx#_self1 08.10.11
- http://www.uni-siegen.de/fb12/ws/lehre/lehre11/ei2_2011/material/v06.pdf 10.10.11
- http://wdict.net/img/creational+pattern.jpg 04.10.11
- http://www.philipphauer.de/study/se/design-pattern/factory-method.php 11.10.11
- www.complang.tuwien.ac.at/franz/objektorientiert/oop06-10.pdf 16.10.11
- http://www.dofactory.com/Patterns/PatternFactory.aspx#_self1 16.10.11
- http://www.dofactory.com/Patterns/PatternAbstract.aspx#_self1 08.10.11