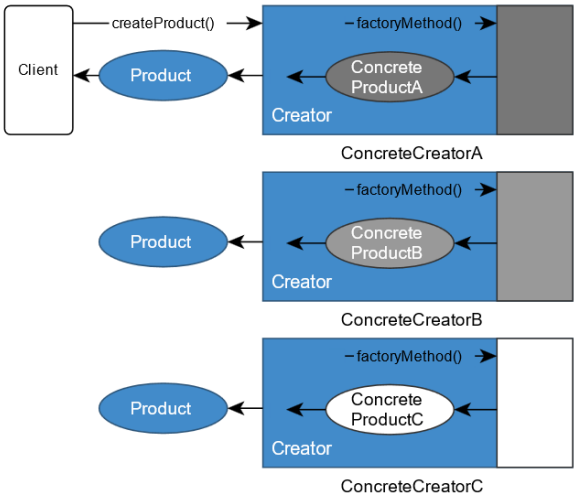
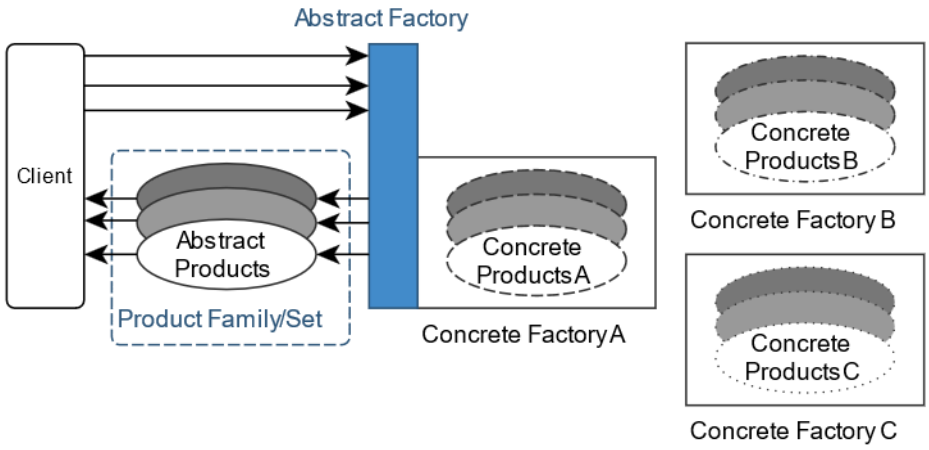
Factory Method Entwurfsmuster



Das Factory Method Entwurfsmuster dient der Entkopplung des Clients von der konkreten Instanziierung einer Klasse. Das erstellte Objekt kann elegant ausgetauscht werden.



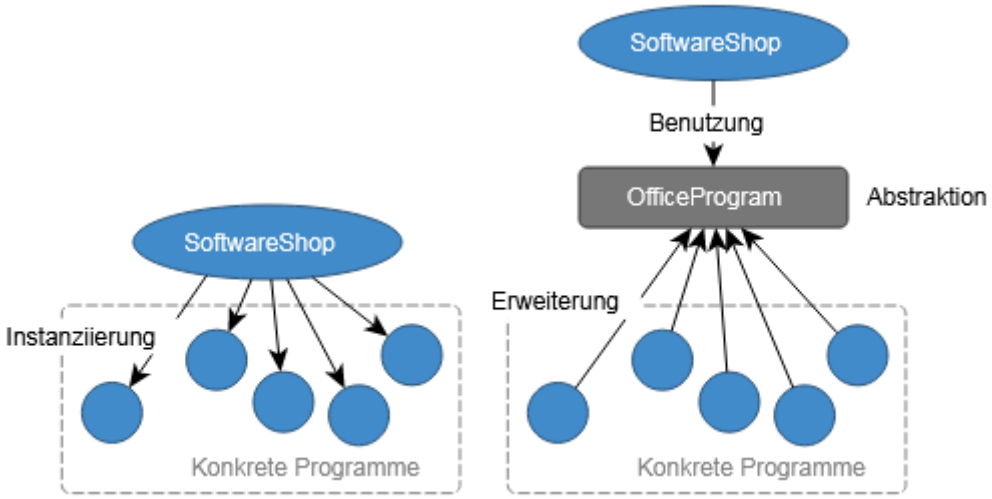
Das Abstract Factory Design Pattern dient der Definition einer zusammenhängenden Familie aus Produkten. Die Familien können elegant ausgetauscht werden

Exkurs: Dependency Inversion Prinzips

Das Factory Method Pattern ermöglicht die Einhaltung des Dependency Inversion Prinzips, dem Prinzip der Abhängigkeitsumkehrung. Zugrunde liegt folgendes OO-Entwurfsprinzip:

Stütze dich nie auf eine konkrete Klasse, sondern immer auf Abstraktion - und das beidseitig.

Bei diesem Prinzip handelt es sich um eine konsequente Weiterentwicklung unseres "Programmiere immer auf eine Schnittstelle»-Entwurfsprinzips. Es gilt immer mit Abstraktionen zu arbeiten und das nicht von den hochschichtigen Komponenten zur tiefschichtigen Komponenten, sondern auch anders rum: von tiefschichtigen Komponenten zu hochschichtigen Komponenten. Denken wir an unserer Software Shop:



Im finalen Entwurf kennt der SoftwareShop nur die Abstraktion OfficeProgramm und ist damit von den konkreten Klassen entkoppelt.

Gleiches gilt für die konkreten Programme: Sie haben zwar an Abhängigkeit gewonnen, sind aber nur von der Abstraktion OfficeProgramm abhängig, da sie diese Klasse erweitern. Und genau an diesem Punkt hat sich die Abhängigkeit umgekehrt.

Alle Komponenten - sowohl die der hohen Schicht (SoftwareShop), als auch die der tiefen (Konkrete Programme) - stützen sich gleichermaßen auf eine Abstraktion. Dank dieser Schnittstelle ist die hohe Komponente unabhängig von den tiefen Komponenten und umgekehrt.

Einführung

Ein Softwareshop verkauft vorranging Microsoft Office Applikationen, wie Word, Excel und Powerpoint. Wenn ein Kunde eine solche Applikation ordert, so sieht der entsprechende Code in der Verwaltungssoftware so aus:

```
public class SoftwareShop {

    public OfficeProgramm holeApp(String zuHolendesProg) {
        OfficeProgramm programm = null;
        //Auswahl der benötigten Applikation
        if (zuHolendesProg.equals("Word")) {
            programm = new Word();
        }
        else if (zuHolendesProg.equals("Powerpoint")) {
            programm = new Powerpoint();
        }
        else if (zuHolendesProg.equals("Excel")) {
            programm = new Excel();
        }
        else {
            System.err.println("Ungültig!");
        }

        //Weitere Verarbeitung
        programm.einpacken();
        programm.etikettieren();

        return programm;
    }
}
```

Schnittstelle OfficeProgramm und die Implementierungen Word, Powerpoint und Excel:

```
abstract class OfficeProgramm {
    public void einpacken() {}
    public void etikettieren() {}
    public abstract void starten();
}

class Word extends OfficeProgramm {
    public void starten() {
        System.out.println("Word startet");
    }
}
class Powerpoint extends OfficeProgramm {
    public void starten() {
        System.out.println("Powerpoint startet");
    }
}
class Excel extends OfficeProgramm {
    public void starten() {
        System.out.println("Excel startet");
    }
}
```

Es wird die gewünschte Software ausgewählt, instanziiert, weiter bearbeitet (*einpacken()*, *etikettieren()*) und schließlich zurückgegeben. Dieser Entwurf hat allerdings eine Reihe von Nachteilen:

- Geringe Kohäsion. Die Programminstanziierung und die weitere Verarbeitung werden zusammen durchgeführt.
- Geringe Wiederverwendbarkeit. Was ist, wenn neue Klassen andere OfficeProgramme instanziieren wollen, den Verarbeitungscode allerdings wiederverwenden möchten?
- Unser Offen/Geschlossen-Prinzip ist verletzt. Für Erweiterungen am Programmportfolio muss der bestehende Code geändert werden.

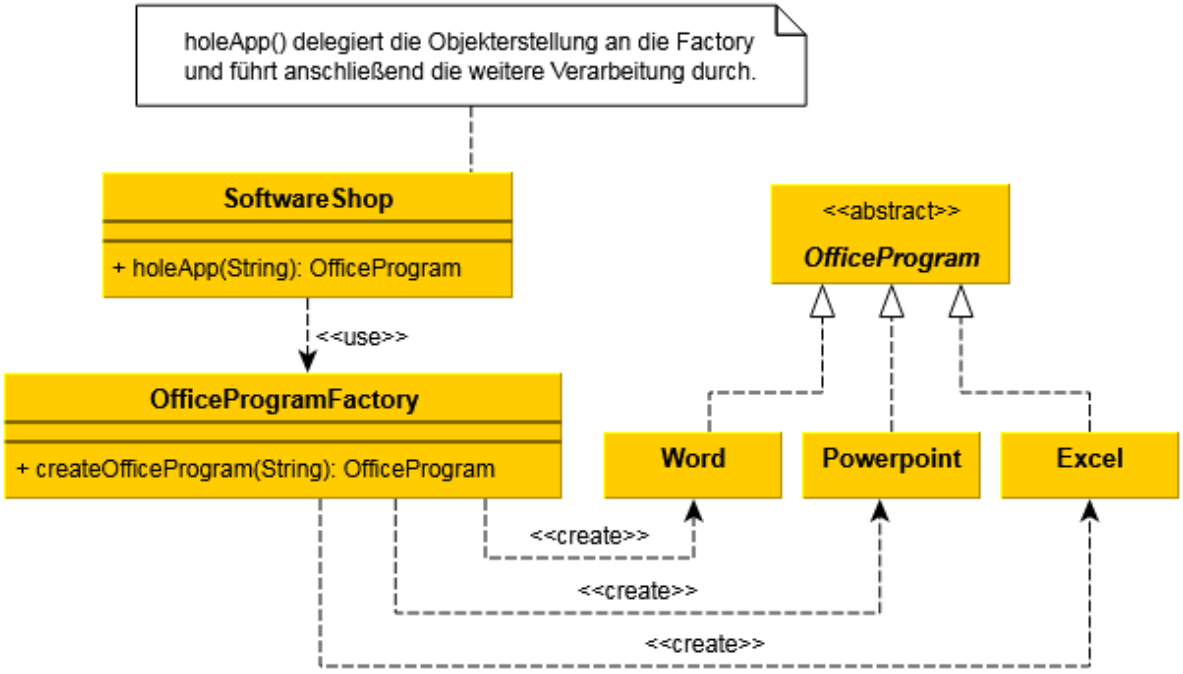
Um unseren Entwurf zu verbessern, rufen wir uns zwei alt bekannte OO-Entwurfsprinzipien ins Gedächtnis:

Offen/Geschlossen-Prinzip (Open/Closed):
Entwürfe sollten für Erweiterungen offen, aber für Veränderungen geschlossen sein.

Gemeint ist, dass Erweiterungen (neue OfficeProgramme etc.) ohne Änderungen an bestehenden Code in das System integriert werden können. Dies im Hinterkopf behaltend erinnern wir uns an ein weiteres stets aktuelles OO-Prinzip:

Identifiziere jene Aspekte, die sich ändern und trenne sie von jenen, die konstant bleiben.

Veränderlich ist die Instanziierung von OfficeProgrammen. Konstant bleibt der weitere Verarbeitungsprozess danach (einpacken() und etikettieren()). Also liegt es nahe, die Logik zur OfficeProgrammerstellung in einer separaten Klasse zu kapseln - einer Factory.



```
class SoftwareShop {
    OfficeProgramFactory officeProgFactory = new OfficeProgramFactory();

    public OfficeProgram holeApp(String zuHolendesProg) {
        //Logik zur Auswahl des gewünschten Programms in Factory gekapselt.
        OfficeProgram program = officeProgFactory.createOfficeProgram(zuHolendesProg)

        //weitere verarbeitung
        program.einpacken();
        program.etikettieren();

        return program;
    }
}
```

OfficeProgramFactory:

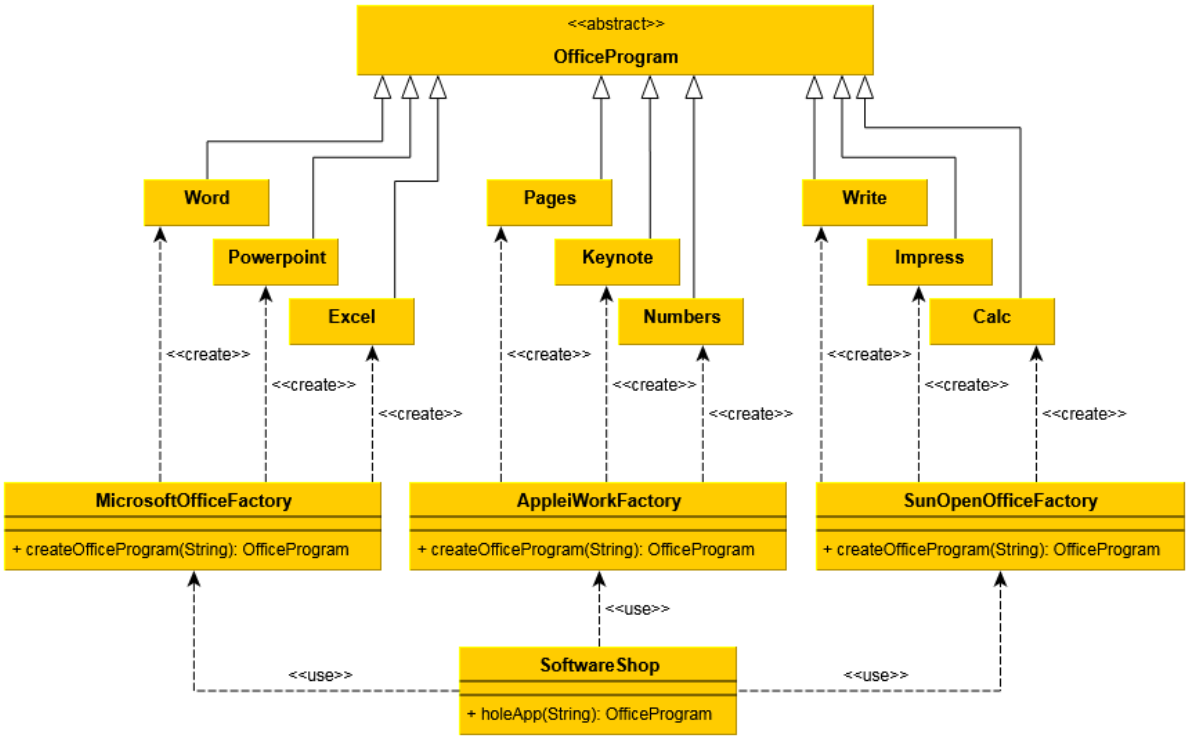
```
class OfficeProgramFactory {
    public OfficeProgram createOfficeProgram(String zuHolendesProg) {
        if (zuHolendesProg.equals("Word")) {
            return new Word();
        }
        else if (zuHolendesProg.equals("Powerpoint")) {
            return new Powerpoint();
        }
        else if (zuHolendesProg.equals("Excel")) {
            return new Excel();
        }
        else {
            System.err.println("Ungültig!");
            return null;
        }
    }
}
```

Die Factory - üblicherweise mit statischen Methoden - macht alleine aber noch kein Design Pattern. So nützlich sie auch ist, es handelt sich lediglich um ein Idiom. Nichtsdestotrotz haben wir nun eine Reihe von Missständen im alten Entwurf abgestellt.

- Kohäsionsgewinn durch Trennung der Objekterstellung von der Objektverarbeitung.
- Wiederverwendbarkeit der Factory.
- Zentrale Stelle für Wartung und Erweiterung.

Doch was ist mit unserem Offen/Geschlossen-Prinzip? Sind wir nun für Erweiterungen offen? Was ist, wenn unser SoftwareShop nun noch weitere Office Suites neben Microsoft Office im Sortiment führen soll, beispielsweise Apple iWork oder Sun OpenOffice?

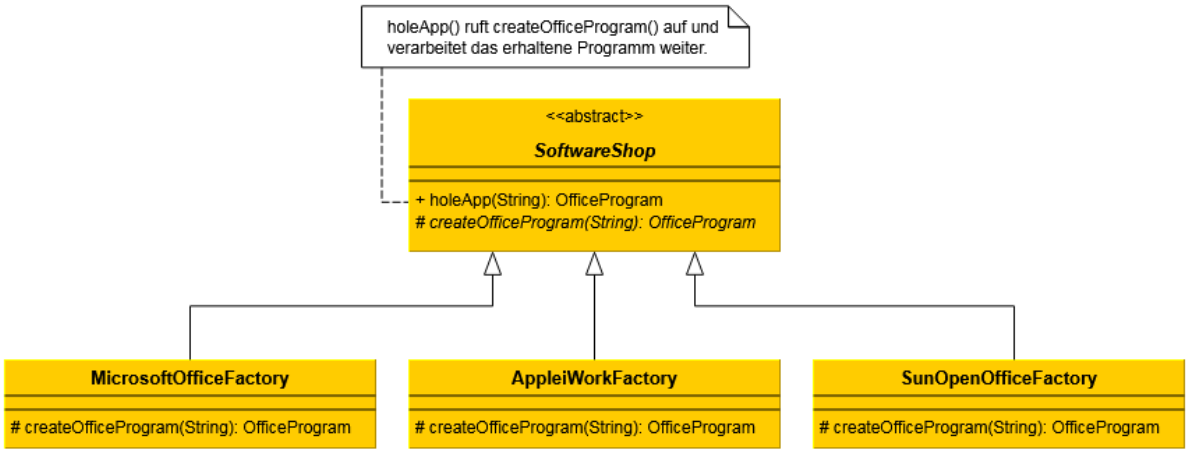
Unser Praktikant schlägt einen *ungünstigen* Entwurf vor: Dieser sieht für jede Suite eine eigene Factory vor, die durch den SoftwareShop genutzt wird.



Schon nicht schlecht, allerdings wird nun ein entscheidener Nachteil unseres Entwurfs deutlich: Die Objekteverarbeitung und die Objekterstellung sind nun **zu stark entkoppelt**! Denn nun ist es möglich die (public) Methode createOfficeProgramm() von der MicrosoftOffice-Factory (oder einer anderen Factory) direkt aufzurufen, und damit die notwendigen Verarbeitungsschritte der holeApp()-Methode des SoftwareShops zu umgehen. So kann Software unverpackt und unetikettiert an die Umwelt gelangen. Maximale Entkopplung ist hier nicht sinnvoll.

Wie können wir auf der einen Seite die Verarbeitung von der Herstellung trennen, sie aber trotzdem aneinander koppeln? Die Lösung liegt in der Vererbung und der Verwendung einer abstrakten Methode, die eine Subklasse implementieren **muss**. Diese abstrakte Methode ist die namensgebende **Factory Method**.

Dabei fungiert unserer SoftwareShop als abstrakte Superklasse. Sie enthält nun zwei Methoden: Zum einen die bekannte holeApp() und zum anderen definiert nun *sie* (nicht die Factorys) die Methode createOfficeProgram(), allerdings abstrakt. Diese Methode wird mit den Zugriffsmodifier protected der Umwelt unzugänglich gemacht.




```
public abstract class SoftwareShop {

    public OfficeProgramm holeApp(String zuHolendesProg) {
        //Delegation der Objekterstellung an Subklasse
        OfficeProgramm programm = createOfficeProgram(zuHolendesProg);

        //weitere verarbeitung
        programm.einpacken();
        programm.etikettieren();

        return programm;
    }

    //Definition der Factory Method
    protected abstract OfficeProgramm createOfficeProgram(String zuHolendesProg);

}
```

Der SoftwareShop hat *keine Ahnung*, welches konkrete Programm er erhält. Dies entscheidet allein die Subklasse, denn diese muss die Factory Method createOfficeProgram() implementieren. Der SoftwareShop delegiert die Objektinstanziierung an seine Subklasse. Von außen ist diese Methode unsichtbar (Zugriff protected). Somit wird gewährleistet, dass der Verarbeitungsprozess immer durchgeführt wird.

Die Subklassen erweitern SoftwareShop und implementieren die Factory Method, wobei sie natürlich ihre speziellen Programme je nach Hersteller zurückgeben.

Die Subklassen MicrosoftOfficeFactory, AppleiWorkFactory, SunOpenOfficeFactory instanziieren ihre Programme und geben sie zurück:

```
class MicrosoftOfficeFactory extends SoftwareShop{
    @Override
    protected OfficeProgramm createOfficeProgram(String zuHolendesProg) {
        OfficeProgramm programm = null;
        if (zuHolendesProg.equals("Textverarbeitung")) {
            programm = new Word();
        }
        else if (zuHolendesProg.equals("Präsentation")) {
            programm = new Powerpoint();
        }
        else if (zuHolendesProg.equals("Tabellenkalkulation")) {
            programm = new Excel();
        }
        else {
            System.err.println("Ungültig!");
        }
        return programm;
    }
}
```

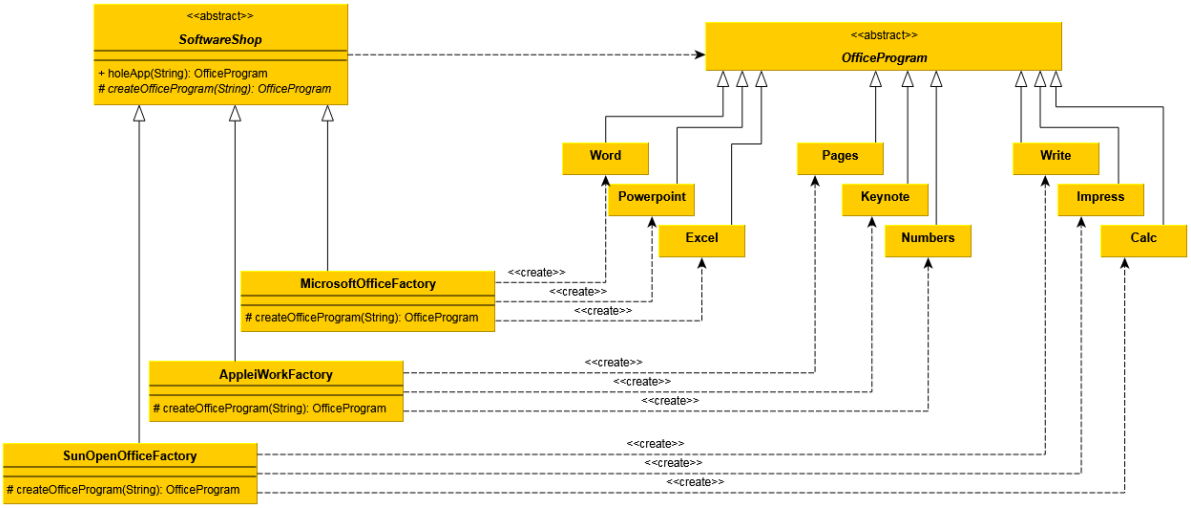
```
class AppleiWorkFactory extends SoftwareShop{
    @Override
    protected OfficeProgramm createOfficeProgram(String zuHolendesProg) {
        OfficeProgramm programm = null;
        if (zuHolendesProg.equals("Textverarbeitung")) {
            programm = new Pages();
        }
        else if (zuHolendesProg.equals("Präsentation")) {
            programm = new Keynode();
        }
        else if (zuHolendesProg.equals("Tabellenkalkulation")) {
            programm = new Numbers();
        }
        else {
            System.err.println("Ungültig!");
        }
        return programm;
    }
}
```

```
class SunOpenOfficeFactory extends SoftwareShop{
    @Override
    protected OfficeProgramm createOfficeProgram(String zuHolendesProg) {
        OfficeProgramm programm = null;
        if (zuHolendesProg.equals("Textverarbeitung")) {
            programm = new Write();
        }
        else if (zuHolendesProg.equals("Präsentation")) {
            programm = new Impress();
        }
        else if (zuHolendesProg.equals("Tabellenkalkulation")) {
            programm = new Calc();
        }
        else {
            System.err.println("Ungültig!");
        }
        return programm;
    }
}
```

```
abstract class OfficeProgramm {
    public void einpacken() {}
    public void etikettieren() {}
    public abstract void starten();
}

class Word extends OfficeProgramm {
    public void starten() {
        System.out.println("Word startet");
    }
}
class Powerpoint extends OfficeProgramm {
    public void starten() {
        System.out.println("Powerpoint startet");
    }
}
class Excel extends OfficeProgramm {
    public void starten() {
        System.out.println("Excel startet");
    }
}
class Pages extends OfficeProgramm {
    public void starten() {
        System.out.println("Pages startet");
    }
}
class Keynode extends OfficeProgramm {
    public void starten() {
        System.out.println("Keynode startet");
```

Der Client instanziiert die gewünschte HerstellerFactory und kann sich von dieser fortan Programme über die Methode der Superklasse holen. Er arbeitet nur gegen die Schnittstellen (SoftwareShop, OfficeProgramm) und weiß damit nicht, mit welchen konkreten Produkten er arbeitet.



Zu den Vorteilen, die uns der Einsatz der "normalen" Factory brachte, kommen nun dank Factoy Method Pattern weitere:

- **Wiederverwendbarkeit** durch Entkopplung der Objektverarbeitung einer konkreten Implementierung. Die Verarbeitung kennt nur die Schnittstelle. Sie kann damit jedes beliebige Programm verarbeiten. Neue Programme sind schnell integriert, da der Verarbeitungscode wiederverwendet werden kann.
- **Konsistenz** durch die Zentralisierung der Verarbeitung und die Unumgänglichkeit dieses Schrittes. Der Verarbeitungsprozess kann nun an einer zentralen Stelle gewartet und erweitert werden. Weiterhin kann kein unverarbeitetes Programm mehr an den Client gehen.
- **Erweiterbarkeit**. Neue OfficeSuites können ins System integriert werden ohne bestehenden Code verändern zu müssen. Zur Erweiterung bedarf es lediglich der Ableitung des SoftwareShops. Das Offen/Geschlossen-Prinzip wurde realisiert.

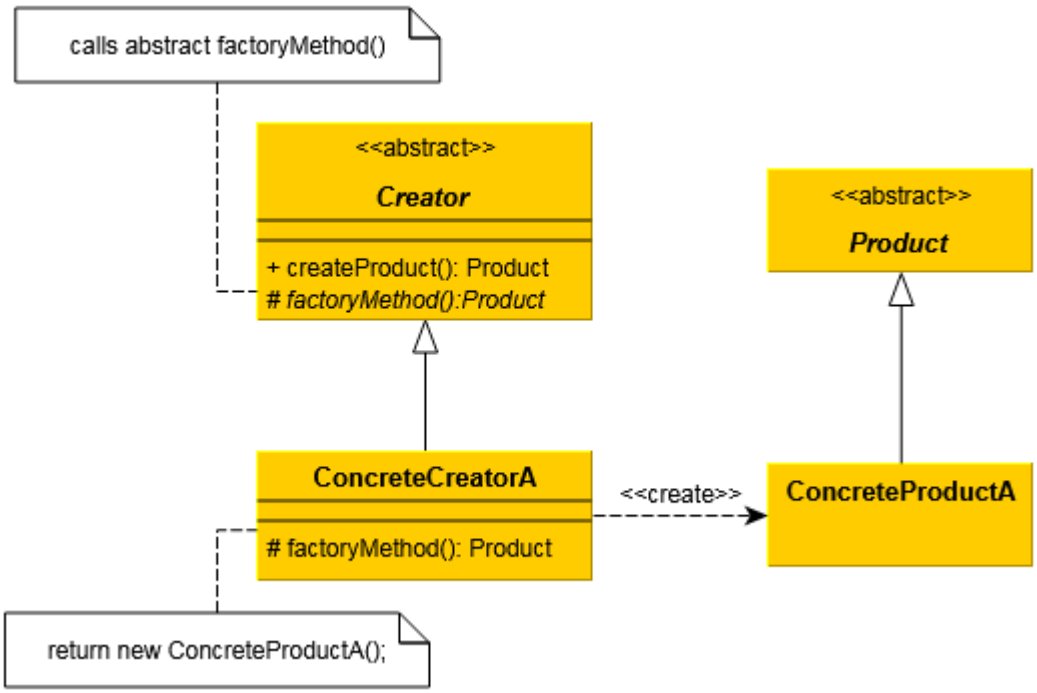
Es zeigt sich, dass durch den Einsatz des Factory Method Patterns, ein hohes Maß an Flexibilität und Allgemeingültigkeit gewonnen wird, während zeitgleich die Wartung erleichtert wurde und Erweiterungen schnell und unkompliziert möglich sind.

Nach dieser Einführung wird im folgenden Abschnitt das Factory Method Design Pattern formalisiert, näher analysiert und diskutiert.

factoryMethod()), die es in seiner createProduct()-Methode aufruft und die seine Unterklassen implementieren müssen. Unterklassen (ConcreateCreators) implementieren diese Methode und geben ein ConcreteProduct zurück.

Nachdem die Unterklasse des Creators das konkrete Product an den Creator zurückgegeben hat, kann der Creator noch allgemeinen Herstellungscode enthalten, die auf jedes Product angewandt werden muss, bevor es an den Client geliefert wird.

Der Creator kann beliebig erweitert werden (ConcreteCreatorB, ConcreateCreatorC) und somit verschiedene Products liefern.



Quellen:

https://www.tutorialspoint.com/design_pattern/

https://www.philipphauer.de/study/se/design-pattern.php