# Bridge

## 1)

Die einzelnen Pizzaklassen unterscheiden sich nur geringfügig voneinander, dennoch muss jede einzelne Pizzaklasse einzeln implementiert werden, um den Bridge Pattern gerecht zu werden.

## 2)



## 3)

## Pizza

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public abstract class Pizza

{

public virtual Pizzeria Pizzeria

{

get;

set;

}

public abstract string[] rezept();

public abstract int kosten();

public Pizza(Pizzeria laden)

{

this.Pizzeria = laden;

}

}

## Magherita

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public class Margherita : Pizza

{

public override int kosten()

{

return this.Pizzeria.teigPreis() + this.Pizzeria.tomatensoßePreis() + this.Pizzeria.käsePreis();

}

public override string[] rezept()

{

string[] result = { this.Pizzeria.teig(), this.Pizzeria.tomatensoße(), this.Pizzeria.käse() };

return result;

}

public Margherita(Pizzeria laden)

: base(laden)

{

}

}

## Pizzeria

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public interface Pizzeria

{

string teig();

int teigPreis();

string tomatensoße();

int tomatensoßePreis();

string käse();

string salami();

int käsePreis();

int salamiPreis();

string schinken();

int schinkenPreis();

string paprika();

int paprikaPreis();

string sardellen();

int sardellenPreis();

}

## Tonis

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public class Tonis : Pizzeria

{

public virtual string teig()

{

return "Schön dünner Teig für besonders knusprige Pizza";

}

public virtual int teigPreis()

{

return 1;

}

public virtual string tomatensoße()

{

return "Traditionelle italienische Tomatensoße mit Oregano";

}

public virtual int tomatensoßePreis()

{

return 2;

}

public virtual string käse()

{

return "Etwas Käse";

}

public virtual int käsePreis()

{

return 1;

}

public virtual string salami()

{

return "Feinste Toskanasalami";

}

public virtual int salamiPreis()

{

return 4;

}

public virtual string schinken()

{

return "Prosciutto Cotto";

}

public virtual int schinkenPreis()

{

return 4;

}

public virtual string paprika()

{

return "Etwas Paprika";

}

public virtual int paprikaPreis()

{

return 1;

}

public virtual string sardellen()

{

return "Etwas Fisch";

}

public virtual int sardellenPreis()

{

return 3;

}

}

# Visitor

## 1)

Wenn eine neue Pizzaklasse (concrete element) hinzugefügt wird, dann muss jeder existierende Koch (visitor) erweitert werden, damit dieser auf die neue Pizzaklasse reagieren kann. Das führt dazu, dass mit jeder neuen Pizzaklassen der Umfang der Kochklassen steigt.

## 2)



## 3)

## KochRezeptVisitor

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public interface KochRezeptVisitor

{

string[] visitRezept(Pizza pizza);

}

## Mauro

using System;

using System.Collections;

using System.Linq;

using System.Text;

public class Mauro : Koch

{

public virtual string[] visitRezept(Pizza pizza)

{

ArrayList rezept = new ArrayList();

if (pizza is Margherita)

{

rezept.Add(pizza.Pizzeria.teig());

rezept.Add(pizza.Pizzeria.tomatensoße());

rezept.Add(pizza.Pizzeria.tomatensoße());

rezept.Add(pizza.Pizzeria.käse());

}

else if (pizza is Napoli)

{

rezept.Add(pizza.Pizzeria.teig());

rezept.Add(pizza.Pizzeria.tomatensoße());

rezept.Add(pizza.Pizzeria.käse());

rezept.Add(pizza.Pizzeria.sardellen());

rezept.Add(pizza.Pizzeria.sardellen());

}

else

rezept.Add("Es exestiert kein Rezept");

return (string[])rezept.ToArray();

}

}

## Pizza

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public abstract class Pizza

{

public virtual Pizzeria Pizzeria

{

get;

set;

}

public abstract string[] rezept();

public abstract int kosten();

public virtual Pizza (Pizzeria laden)

{

this.Pizzeria = laden;

}

public virtual string[] accept(KochRezeptVisitor visitor)

{

throw new System.NotImplementedException();

}

public abstract int durchmesser();

}

## Magherita

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public class Margherita : Pizza

{

public override int kosten()

{

return this.Pizzeria.teigPreis() + this.Pizzeria.tomatensoßePreis() + this.Pizzeria.käsePreis();

}

public override string[] rezept()

{

string[] result = { this.Pizzeria.teig(), this.Pizzeria.tomatensoße(), this.Pizzeria.käse() };

return result;

}

public Margherita(Pizzeria laden)

: base(laden)

{

}

public override string[] accept(KochRezeptVisitor visitor)

{

return visitor.visitRezept(this);

}

public override int durchmesser()

{

return 3;

}

}

# Composite und Decorator

## 1)

## 2)



## 3)



## 4)

## Pizza

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public abstract class Pizza

{

public virtual Belag Belag

{

get;

set;

}

public abstract int preis();

}

## Piccolo

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public class Piccolo : Pizza

{

public override int preis()

{

return 1 + this.Belag.preis();

}

}

## Standard

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public class Standard : Pizza

{

public override int preis()

{

return 2 + this.Belag.preis();

}

}

## Belag

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public abstract class Belag

{

public abstract int preis();

}

## Käse

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public class Käse : Belag

{

public override int preis()

{

return 1;

}

}

## Salami

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public class Salami : Belag

{

public override int preis()

{

return 3;

}

}

## 5)

## Pizza

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public interface Pizza

{

int preis();

}

Piccolo

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public sealed class Piccolo : Pizza

{

public int preis()

{

return 1;

}

}

Standard

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public sealed class Standard : Pizza

{

public int preis()

{

return 2;

}

## }

## Belag

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public abstract class Belag : Pizza

{

public Pizza Pizza

{

get;

set;

}

public Belag(Pizza auf)

{

this.Pizza = auf;

}

}

## Käse

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public sealed class Käse : Belag

{

public int preis()

{

return 1 + this.Pizza.preis();

}

public Käse(Pizza auf) : base(auf)

{

}

## }

## Pilze

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public sealed class Pilze : Belag

{

public Pilze(Pizza auf) : base(auf)

{

}

public int preis()

{

return 2 + this.Pizza.preis();

}

## }

# Diskussion (Erwin Stamm)

Entwurfsmuster bieten eine Problem unspezifische Struktur, die von anderen Entwicklern erkennbar ist und ermöglichen die Wiederverwendung von bestehendem Quellcode. Dennoch kann der übermäßige oder unpassende Einsatz von Design Pattern zur erhöhten Komplexität des Quellcodes führen. Im Folgenden wird die korrekte Arbeitsweise und die Code-and-Fix Arbeitsweise verglichen, mit dem Fokus auf dem Einsatz von Entwurfsmuster.

Entwurfsmuster können durch Benennung und Aufbau von einem Entwickler schnell erkannt werden. Durch die Wiedererkennung des Patterns muss der Entwickler sich nicht den gesamten Quellcode durch gehen, sondern nur den Teil des Quellcodes, für den sich der Entwickler interessiert. Dadurch kann der Entwickler den Quellcode in geringer Zeit anpassen oder erweitern. Bei der Erweiterung kann z.B. das Adapterpattern benutzt werden um bestehende Klassen zu erweitern oder anzupassen. Wenn sich nun der Entwickler auch bei seiner Änderung an das Entwicklungsmuster hält, dann Bleibt der Quellcode auch für folgende Entwickler in einem wiedererkennbaren Zustand.

Dieses Beispiel zeigt das Entwicklungspattern das gezielte Ändern einer bestehenden Implementierung erleichtern und den Zeitaufwand verringern können. Trotz dessen werden Entwurfsmuster nicht von allen Entwicklern eingesetzt. Dies geschieht nicht Grundlos und kann auf mehrere Probleme zurückgeführt werden. Im Folgenden möchte ich die Probleme benennen und beschreiben die mir im Besonderen aufgefallen sind.

Eines der Probleme meiner Meinung nach ist die übliche zeitliche Positionierung der Pattern Auswahl. Ein normaler Entwickler wird meist ein Problem geschildert, das dieser bewältigen soll. Nun arbeitet sich der Entwickler in das Problem ein und überlägt sich mögliche Implantationen wie er das Problem lösen könnte. Als nächstes gibt es zwei Möglichkeiten was der Entwickler als nächstes machen könnte. Entweder er Programmiert seine Implementierung oder er überlegt sich ein passendes Entwicklungsmuster. Ersteres wird oft als Code-and-Fix bezeichnet und letzteres als die korrekte Arbeitsweise.

Die korrekte Arbeitsweise hat das Problem, das ein weitere Problemlösung (Auswahl des Entwurfsmusters), zwischen die Überbelegungen des Hauptproblems und dessen Umsetzung gelegt wird. Das bewirkt zwei Kontextwechsel beim Programmierer: von dem Hauptproblem zur Auswahl des Entwurfsmusters und zurück. Dies führt zu einem zeitlichen Verlust des Entwicklers. Weiterhin könnte der Entwickler Überlegungen vergessen die er vor der Auswahl des Entwurfsmusters für das Hauptproblem getroffen hat und muss sich erneut in das Hauptproblem einarbeiten. Weiterhin besteht die Möglichkeit, dass bei der Auswahl des Entwurfsmusters ein unpassendes Entwurfsmuster gewählt wurde und sich dies erst während der Implementierung zeigt. Entweder könnte es hingenommen werden und die Implementierung an dem unpassenden Entwurfsmuster angepasst werden, was die Lesbarkeit des Quellcodes verringert oder es wird ein passendes Entwurfsmuster ausgewählt, dafür muss aber das bestehende Entwurfsmuster und die Implementierung verworfen werden. Der letztere Fall führt dabei zu einem enormen zeitlichen Verlust.

Die Code-and-Fix Lösung, die im Folgenden als Protoyping bezeichnet wird, hat keinen Kontextwechsel zwischen der Analyse des Hauptproblems und dessen Umsetzung, hat aber das Problem das kein Entwurfsmuster zum Zeitpunkt der ersten Implementation vorliegt, dies erschwert die Einarbeitung in den Quellcode für Zukünftige Nutzer. Um dieses Problem zu beseitigen muss nach der ersten Implementierung (Prototype), überlegt werden ob der Einsatz eines Entwurfsmusters zur Strukturierung des Quellcode eingesetzt werden soll. Der im Protoype erstellte Quellcode kann dann beim Einsatz eines Entwurfsmusters auf diesen Abstrahiert werden, da der Entwickler noch keine feste Quellcodestruktur zum Zeitpunkt des Protoypes erstellt haben sollte und nur Funktionen in Methoden kapseln sollte. Das Hauptproblem bei dieser Vorgehensweise ist, das die nötige Zeit zur Entwurfsmusterauswahl nach der Prototype-Implementation einberechnet werden muss. Zeitlicher druck wird dazu führen, dass das Problem frühzeitig als beendet erklärt wird und der Prototype als fertige Lösung übergeben wird. Dies führt dazu, dass Quellcode unstrukturiert liegen gelassen wird und dieser für zukünftige Entwickler schwerer zu verstehen ist und diese somit eine längere Einarbeitungszeit haben. Im Gegensatz dazu kann die Lösung bei der korrekten Arbeitweise erst übergeben werden, wenn die Struktur schon eingeplant ist, da diese vor der Implementierung umgesetzt wird. Dabei ist auch zusehen, das bei der korrekten Arbeitsweise die Implementierung an das Entwurfsmuster angepasst wird und beim Prototyping das Entwurfsmuster an der Implementierung.

Die Endgültige Auswahl liegt beim Entwickler, dieser kann entscheiden welche Vorgehensweise er für sich bevorzugt.