2014

Software-Engineering II

TIAI3006.1: Objektorientiertes Software-Engineering

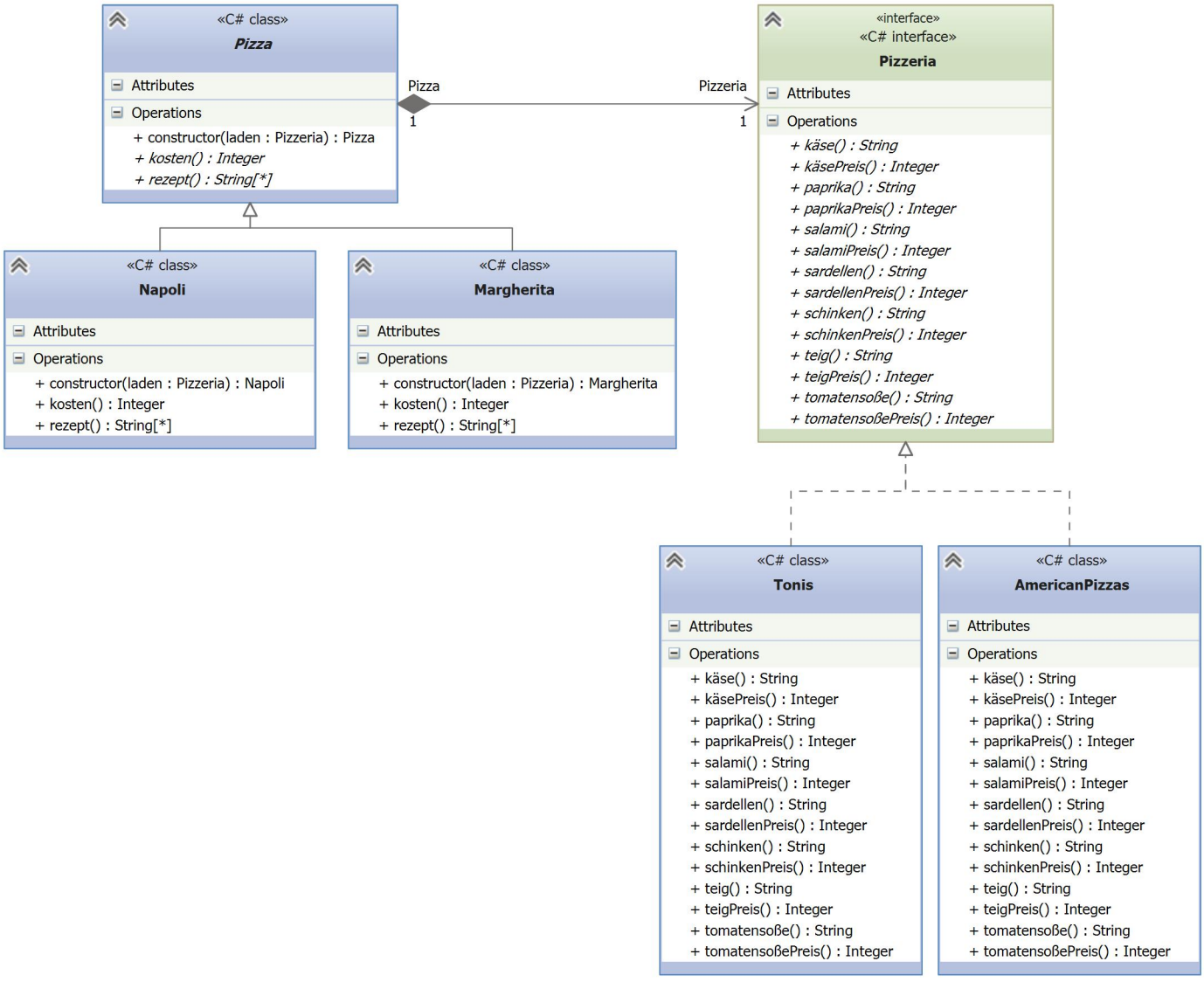
Patrick Robinson & Erwin Stamm

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.Pizze-
ria.käsePreis();
	}

	public override string[] rezept()
	{
	 string[] result = { this.Pizzeria.teig(), this.Pizzeria.tomatensoße(), this.Pizze-
ria.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

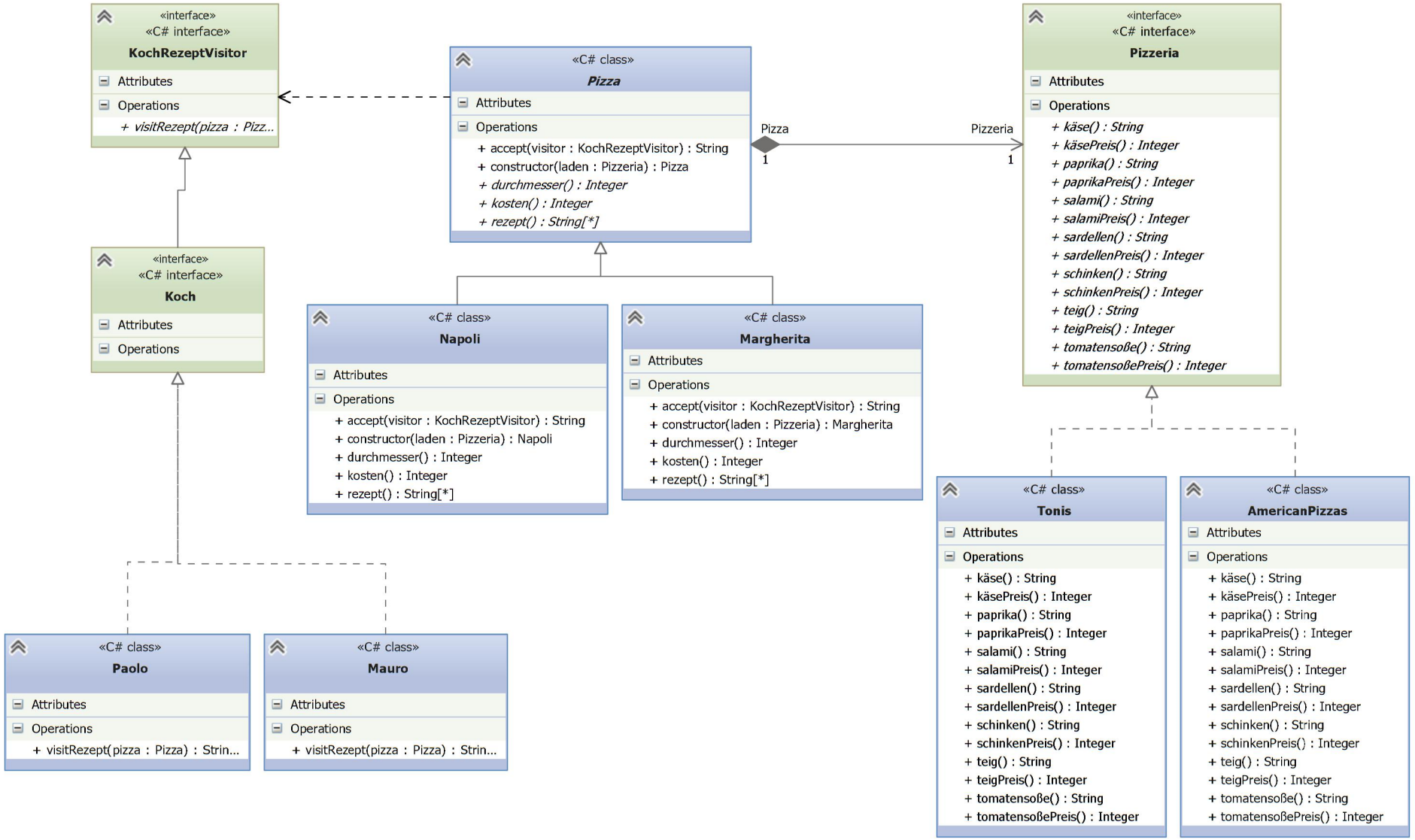
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

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 Pizzaklasse der Umfang der Kochklassen steigt.


«interface»
«C# interface»
KochRezeptVisitor
Attributes
Operations
+ visitRezept(pizza : Pizz...
«C# class»
Pizza
Attributes
Operations
+ accept(visitor : KochRezeptVisitor) : String
+ constructor(laden : Pizzeria) : Pizza
+ durchmesser() : Integer
+ kosten() : Integer
+ rezept() : String[*]
Pizza
1
Pizzeria
1
«interface»
«C# interface»
Pizzeria
Attributes
Operations
+ käse() : String
+ käsePreis() : Integer
+ paprika() : String
+ paprikaPreis() : Integer
+ salami() : String
+ salamiPreis() : Integer
+ sardellen() : String
+ sardellenPreis() : Integer
+ schinken() : String
+ schinkenPreis() : Integer
+ teig() : String
+ teigPreis() : Integer
+ tomatensoße() : String
+ tomatensoßePreis() : Integer
«interface»
«C# interface»
Koch
Attributes
Operations
«C# class»
Napoli
Attributes
Operations
+ accept(visitor : KochRezeptVisitor) : String
+ constructor(laden : Pizzeria) : Napoli
+ durchmesser() : Integer
+ kosten() : Integer
+ rezept() : String[*]
«C# class»
Margherita
Attributes
Operations
+ accept(visitor : KochRezeptVisitor) : String
+ constructor(laden : Pizzeria) : Margherita
+ durchmesser() : Integer
+ kosten() : Integer
+ rezept() : String[*]
«C# class»
Tonis
Attributes
Operations
+ käse() : String
+ käsePreis() : Integer
+ paprika() : String
+ paprikaPreis() : Integer
+ salami() : String
+ salamiPreis() : Integer
+ sardellen() : String
+ sardellenPreis() : Integer
+ schinken() : String
+ schinkenPreis() : Integer
+ teig() : String
+ teigPreis() : Integer
+ tomatensoße() : String
+ tomatensoßePreis() : Integer
«C# class»
AmericanPizzas
Attributes
Operations
+ käse() : String
+ käsePreis() : Integer
+ paprika() : String
+ paprikaPreis() : Integer
+ salami() : String
+ salamiPreis() : Integer
+ sardellen() : String
+ sardellenPreis() : Integer
+ schinken() : String
+ schinkenPreis() : Integer
+ teig() : String
+ teigPreis() : Integer
+ tomatensoße() : String
+ tomatensoßePreis() : Integer
«C# class»
Paolo
Attributes
Operations
+ visitRezept(pizza : Pizza) : Strin...
«C# class»
Mauro
Attributes
Operations
+ visitRezept(pizza : Pizza) : Strin...

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.Pizze-
ria.käsePreis();
    }

    public override string[] rezept()
    {
        string[] result = { this.Pizzeria.teig(), this.Pizzeria.tomatensoße(), this.Pizze-
ria.käse() };
        return result;
    }

    public Margherita(Pizzeria laden)
        : base(laden)
    {
    }

       public override string[] accept(KochRezeptVisitor visitor)
       {
        return visitor.visitRezept(this);
    }
```

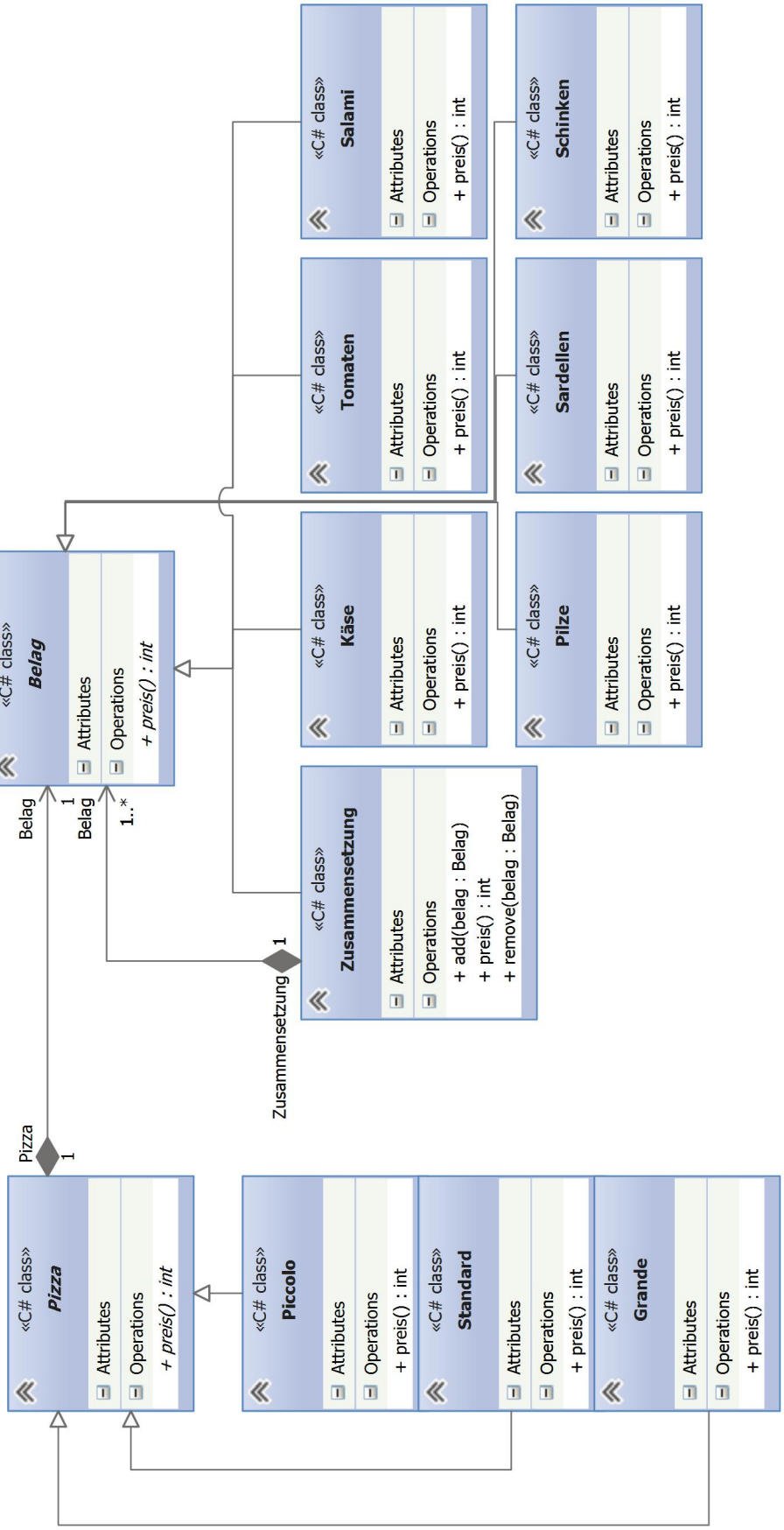
```
        public override int durchmesser()
        {
         return 3;
        }

}
```

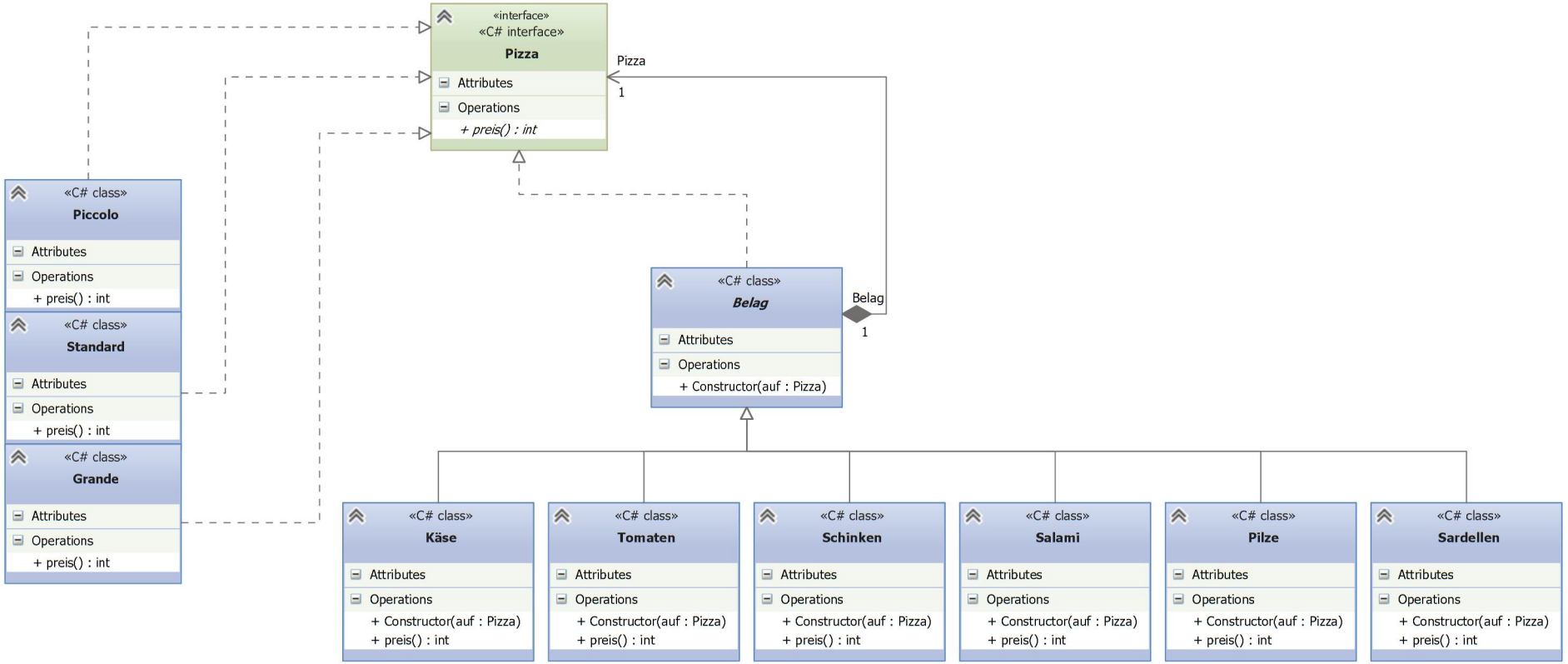
Composite und Decorator

1)

Grundlegend anders an dieser Aufgabe ist vor allem, dass die Pizzen hier zur Laufzeit definiert werden, also ihr Belag zur Bestellzeit gewählt wird und nicht nur eine Auswahl aus vordefinierten Pizzen bereitgestellt wird.
Da eine Pizza nur einen Teigboden und damit nur eine Größe hat, ist dies jeweils die Basis, die dann z.B. dekoriert wird.
Der Fokus der Entwurfsmuster muss auf dem Zusammenstellen von Belag liegen und in diesem Punkt unterscheiden sich die beiden auch am meisten.
Während beim Decorator Pattern die Pizza nach und nach mit den verschiedenen Belagsdekoratoren umschlossen wird, wird beim Composite Pattern erst ein Belag aus den Komponenten zusammengestellt und mit dem Teigboden verbunden.

2)


«C# class»
Pizza
Attributes
Operations
+ preis() : int
«C# class»
Piccolo
Attributes
Operations
+ preis() : int
«C# class»
Standard
Attributes
Operations
+ preis() : int
«C# class»
Grande
Attributes
Operations
+ preis() : int
Pizza
1
Belag
1..*
«C# class»
Belag
Attributes
Operations
+ preis() : int
«C# class»
Salami
Attributes
Operations
+ preis() : int
«C# class»
Tomaten
Attributes
Operations
+ preis() : int
«C# class»
Käse
Attributes
Operations
+ preis() : int
«C# class»
Pilze
Attributes
Operations
+ preis() : int
«C# class»
Sardellen
Attributes
Operations
+ preis() : int
«C# class»
Schinken
Attributes
Operations
+ preis() : int
Zusammensetzung
1
«C# class»
Zusammensetzung
Attributes
Operations
+ add(belag : Belag)
+ preis() : int
+ remove(belag : Belag)


«interface»
«C# interface»
Pizza
Attributes
Operations
+ preis() : int
Pizza
1
«C# class»
Piccolo
Attributes
Operations
+ preis() : int
«C# class»
Standard
Attributes
Operations
+ preis() : int
«C# class»
Grande
Attributes
Operations
+ preis() : int
«C# class»
Belag
Attributes
Operations
+ Constructor(auf : Pizza)
Belag
1
«C# class»
Käse
Attributes
Operations
+ Constructor(auf : Pizza)
+ preis() : int
«C# class»
Tomaten
Attributes
Operations
+ Constructor(auf : Pizza)
+ preis() : int
«C# class»
Schinken
Attributes
Operations
+ Constructor(auf : Pizza)
+ preis() : int
«C# class»
Salami
Attributes
Operations
+ Constructor(auf : Pizza)
+ preis() : int
«C# class»
Pilze
Attributes
Operations
+ Constructor(auf : Pizza)
+ preis() : int
«C# class»
Sardellen
Attributes
Operations
+ Constructor(auf : Pizza)
+ preis() : int

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

Entwurfsmuster bieten eine vom Problem unabhängige 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, besonders wenn auch UML-Diagramme angefertigt wurden. Durch die Wiedererkennung des Patterns muss der Entwickler nicht den gesamten Quellcode durch gehen um die Funktionsweise zu verstehen, 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 Patternauswahl. Ein normaler Entwickler wird meist ein Problem geschildert, das dieser bewältigen soll. Nun arbeitet sich der Entwickler in das Problem ein und überlegt sich mögliche Wege wie er das Problem lösen könnte. Als nächstes gibt es zwei Möglichkeiten was der Entwickler 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, dass eine 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 Arbeitsweise erst übergeben werden, wenn die Struktur schon eingeplant ist, da diese vor der Implementierung umgesetzt wird. Dabei ist auch zu sehen, dass bei der korrekten Arbeitsweise die

Implementierung an das Entwurfsmuster angepasst wird und beim Prototyping das Entwurfsmuster an die Implementierung.

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

Im Allgemeinen muss man natürlich auch die Größe des Problems betrachten. Je umfangreicher das zu implementierende Problem oder das am Problem arbeitende Team ist, desto wichtiger wird eine klare Unterteilung, während der Overhead sich bei sehr kleinen Problemen vielleicht nicht lohnt bzw. eine zu extrem kleingliedrige Aufteilung auch die Lesbarkeit, Wartbarkeit, etc. wieder verringert. Das hat sich in diesem Projekt stark gezeigt, da das Problem sehr simpel gehalten war, ist aber auch bei komplexeren Problemen zu beachten.

Manchmal kann das Fokussieren auf ausgefallene Entwurfsmuster auch von einer einfachen Lösung ablenken. Nehmen wir als Beispiel den Fall 3. „Der Preis setzt sich additiv aus den Preisen für die Größe der Pizza und den Preisen der gewählten Zutaten zusammen."

Sowohl Decorator- als auch Composite-Pattern lassen sich darauf anwenden, aber es sind auch viel simplere Lösungen möglich:

Alternative Implementierung

```
using System;
using System.Collections.Generic;

enum size { Piccolo, Std, Grande };
enum belag { cheese, salami }; // ...

public static class Pizza {
    public static Dictionary<size, int> sizepay = new Dictionary<size,int>(){ {Piccolo, 1}, 
{Std, 2}, {Grande, 3} };
    public static Dictionary<belag, int> belagpay = new Dictionary<belag,int>(){ {cheese, 
1}, {salami, 2} };

    public static int preis(size g, List<belag> b) {
        return sizepay[g] + b.Sum((bel) => belagpay[bel]);
    }
}
```

Diese Implementierung ist in keinster Weise weniger flexible, performant, lesbar oder verständlich. Sie ist auch durchaus aus einer Planung bzw. einem Konzept entstanden. Es ist für einen Entwickler direkt ersichtlich wie sich der Preis errechnet und Änderungen, z.B. am Preis von Salami können einfach und zentral und auch während der Ausführung des Programms durchgeführt werden.

Es wäre sogar recht einfach denkbar dies noch zu erweitern und zum Beispiel die Preise aus Datenbanken auslesen, etc., ohne dass es dadurch als Implementierung unübersichtlich würde.

Im Allgemeinen sollte man darauf achten sauberen und wartbaren Code zu schreiben und dabei können Entwurfsmuster hilfreich sein, aber man sollte nicht um jeden Preis versuchen die Implementierung in ein Entwurfsmuster zu pressen.