RUHR-UNIVERSITÄT BOCHUM
Lehrstuhl für Informatik im Bauwesen

RUB

Programmierung und Programmiersprachen

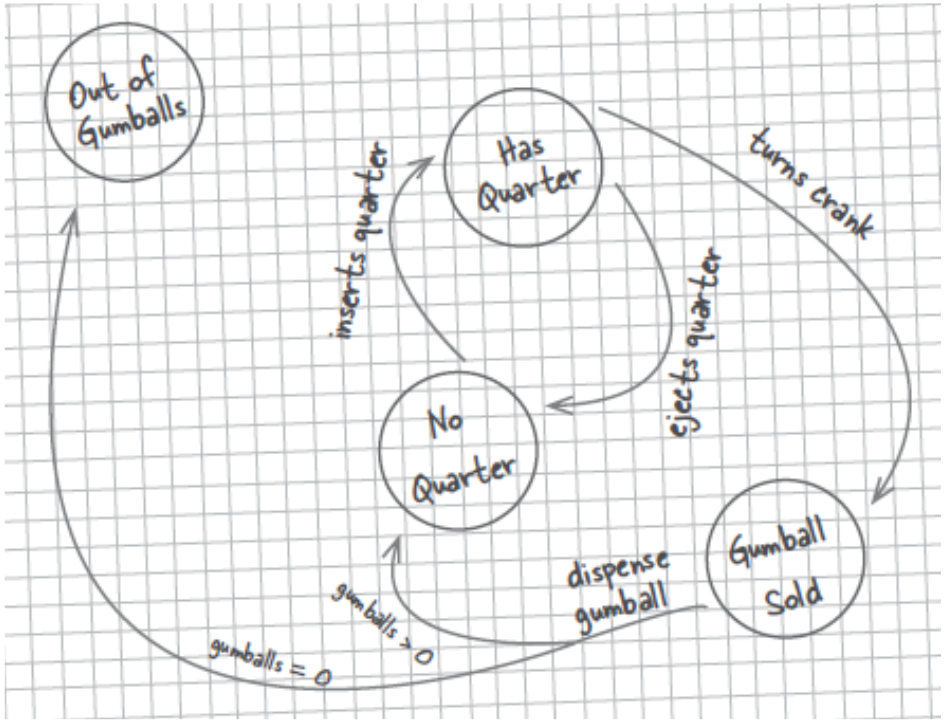
Sommersemester 2023

State-Muster

State-Muster

Zustandsautomaten

Es soll eine Steuerung für einen Kaugummiautomaten entwickelt werden. Folgende Informationen über die Steuerung stehen zur Verfügung:



Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

State-Muster

Zustandsautomaten

Aus den Angaben können einige Methoden identifiziert werden. Diese Methoden überführen einen Zustand des Kaugummiautomaten in einen anderen. Jedoch ist die Ausführung einer Methode abhängig vom vorhandenen Zustand.

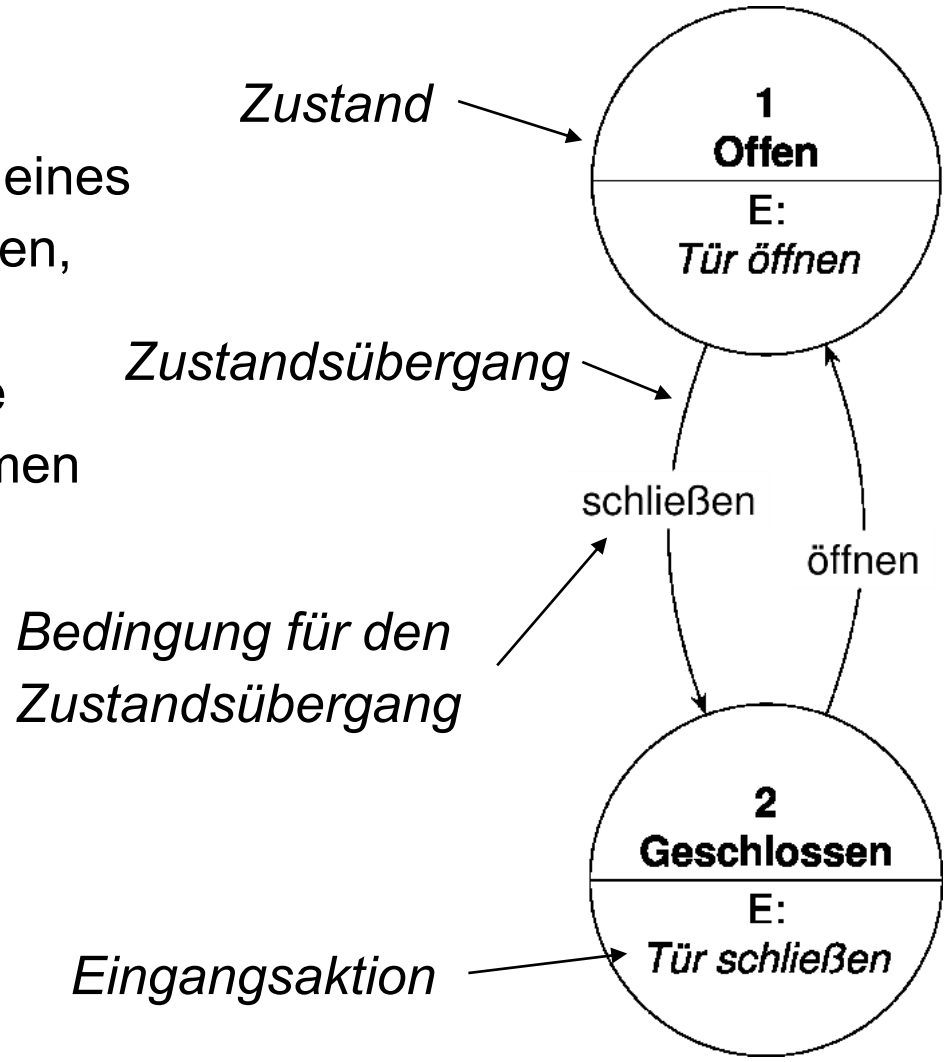
GumballMachine
– count: int
+ GumballMachine(count: int) + insertQuarter(): void + ejectQuarter(): void + tumCrank(): void + dispense(): void + refill(count: int): void

Wie kann sichergestellt werden, dass eine Methode nur bei einem bestimmten Zustand aufgerufen wird?

State-Muster

Zustandsautomaten

Ein Zustandsautomat ist ein Modell eines Verhaltens, bestehend aus Zuständen, Zustandsübergängen und Aktionen.
Ein Automat heißt endlich, wenn die Menge der Zustände, die er annehmen kann, endlich ist

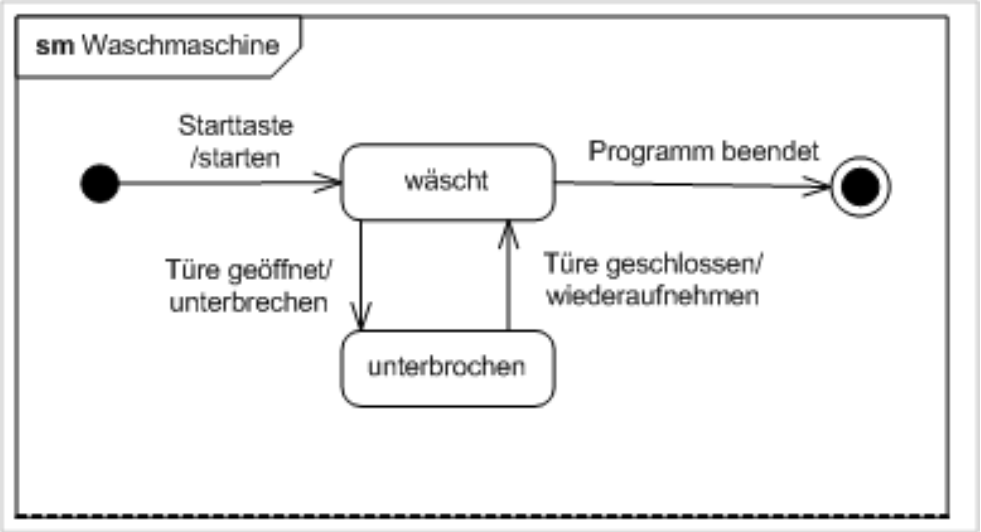


State-Muster

Zustandsautomaten

Zustandsautomaten können mit Hilfe von Zustandsdiagrammen (State Diagram) beschrieben werden. Ein Zustandsdiagramm stellt einen endlichen Automaten in einer UML-Sonderform grafisch dar und wird benutzt, um entweder das Verhalten eines Systems oder die zulässige Nutzung der Schnittstelle eines Systems zu spezifizieren.

Zustände werden durch abgerundete Rechtecke, Übergänge durch Pfeile und Ereignisse durch eine Beschriftung der Pfeile definiert.



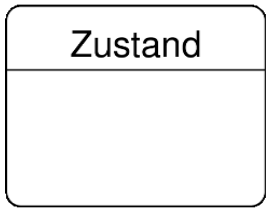
State-Muster

Zustandsautomaten

Des Weiteren gibt es verschiedene sogenannte Pseudozustände (nicht vollständige Auflistung):

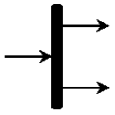
- Startzustand und Endzustand
- Vereinigung
  Zusammenführung des Kontrollflusses von mehreren parallelen Zuständen
- Gabelung
  Aufspaltung des Kontrollflüssen in mehrere parallele Zustände
- Kreuzung oder Entscheidung
  Knoten von dem mehrere alternative Transitionen ausgehen können

Startzustand

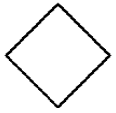


Zustand

Endzustand



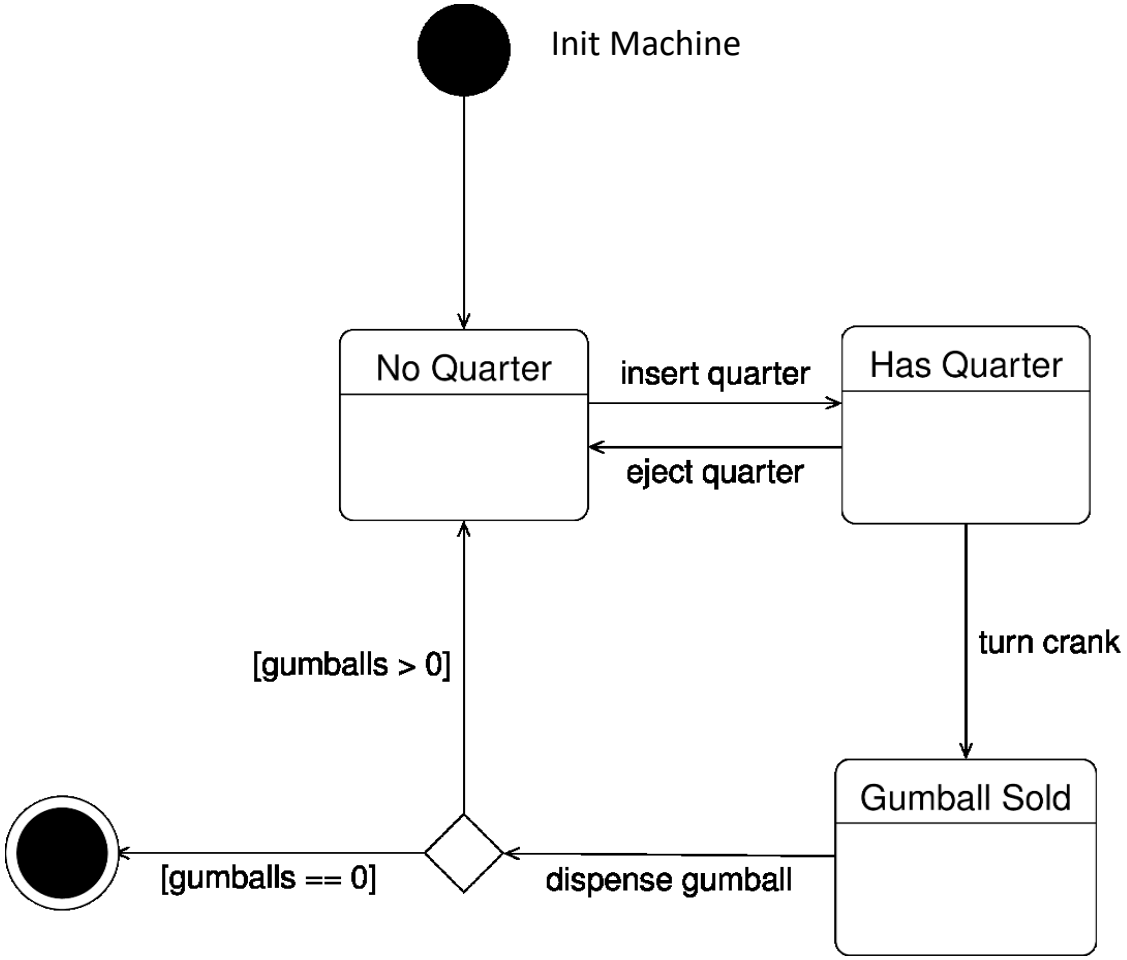
Gabelung oder Vereinigung



Kreuzung oder Entscheidung

State-Muster

Beispiel - Kaugummiautomat



State-Muster

Implementierung von Zuständen

Häufig werden Zustände mit Hilfe von Konstanten und einem Attribut für den aktuellen Zustand umgesetzt. Dabei kann zum Beispiel eine Ganzzahl mit einer Bestimmten Kodierung verwendet werden.

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

int state = SOLD_OUT;
```

State-Muster

Implementierung von Transitionen

Auf Basis der Zustände können jetzt die einzelnen Transitionen (hier Methoden) implementiert werden.

```
public void insertQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("You can't insert another quarter");
    } else if (state == NO_QUARTER) {
        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't insert a quarter, the machine is sold out");
    } else if (state == SOLD) {
        System.out.println("Please wait, we're already giving you a gumball");
    }
}
```

State-Muster

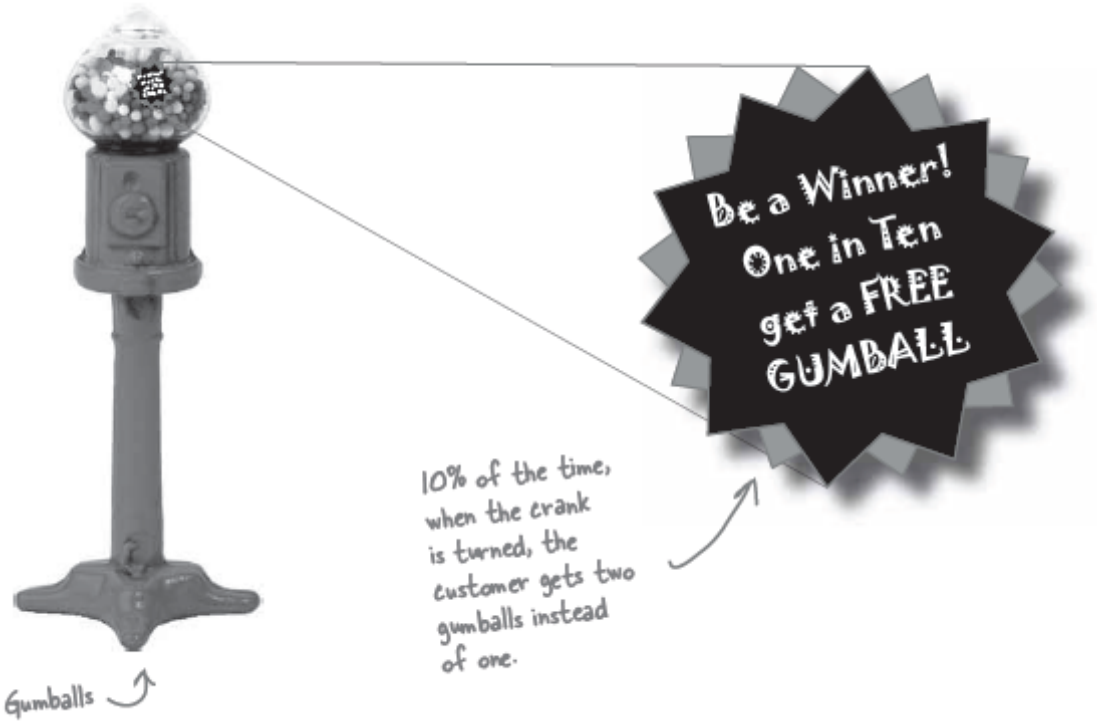
Implementierung von Transitionen

```
public void dispense() {
    if (state == SOLD) {
        System.out.println("A gumball comes rolling out the slot");
        count = count - 1;
        if (count == 0) {
            System.out.println("Oops, out of gumballs!");
            state = SOLD_OUT;
        } else {
            state = NO_QUARTER;
        }
    } else if (state == NO_QUARTER) {
        System.out.println("You need to pay first");
    } else if (state == SOLD_OUT) {
        System.out.println("No gumball dispensed");
    } else if (state == HAS_QUARTER) {
        System.out.println("No gumball dispensed");
    }
}
```

State-Muster

Erweiterbarkeit von Zuständen

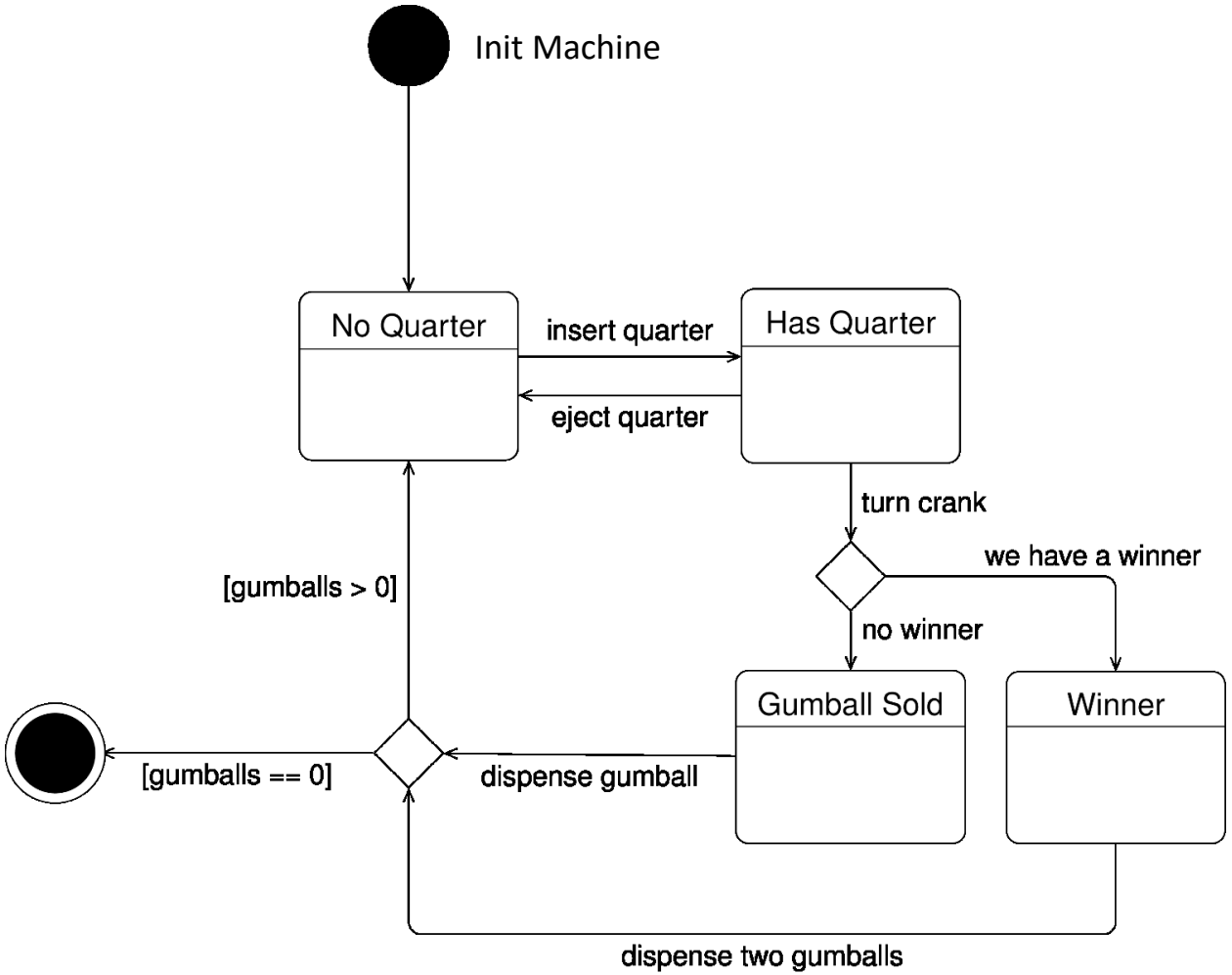
In den Kaugummiautomaten soll ein Gewinnspiel integriert werden. In 10% der Fälle soll der Kunde beim Drehen des Griffs zwei Kaugummikugeln erhalten.



Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

State-Muster

Beispiel - Kaugummiautomat



State-Muster

Erweiterbarkeit von Zuständen

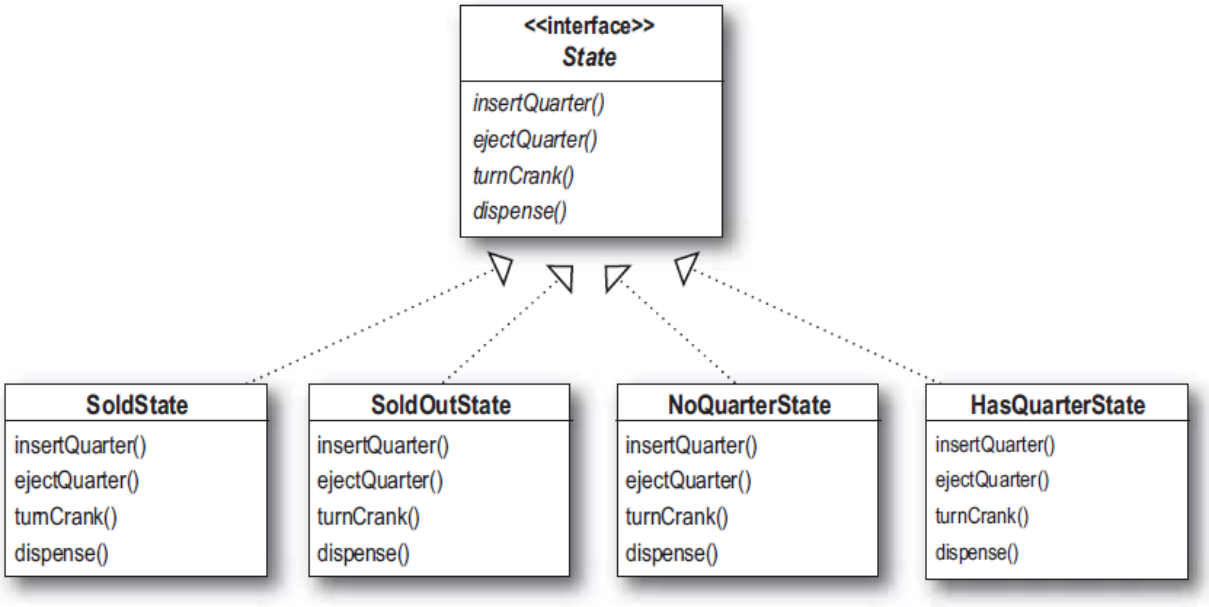
In diesem Beispiel müsste ein neuer Zustand hinzugefügt und alle Methoden angepasst werden. Welche Prinzipien wurden verletzt?

- Prinzip der Objektorientierung
- Kopplung und Bindung
- Offen-Geschlossen-Prinzip
- Kapselung

State-Muster

Zustandsobjekte

Zustandsobjekte werden in eigenen Klassen gekapselt. Transaktionen werden an die jeweiligen Zustände delegiert. Hierzu wird im ersten Schritt ein Interface für die Zustände definiert. Es umfasst alle Aktionen, die im Zustandsautomaten ablaufen können (erst einmal ohne Gewinn)

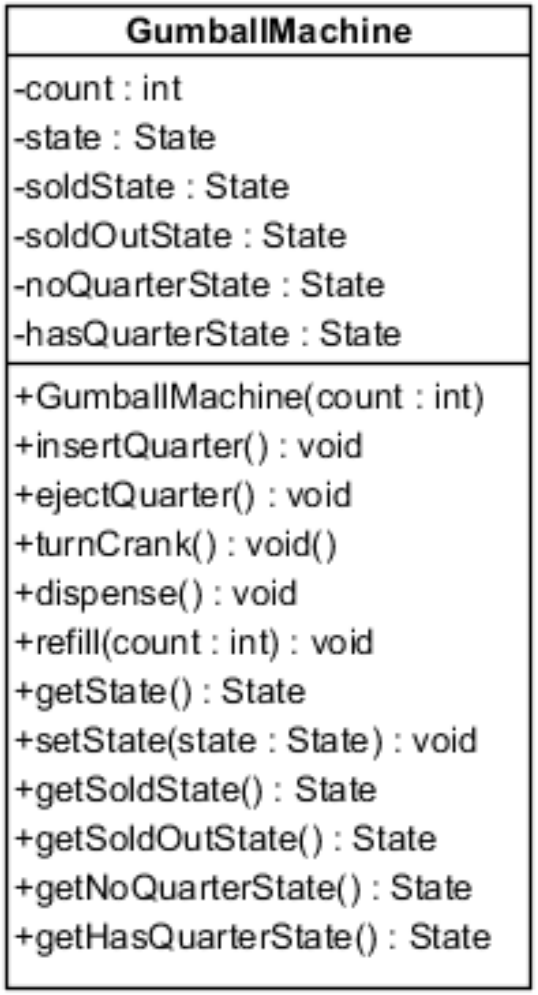


Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

State-Muster

Zustandsobjekte (Kontext)

Für jede Zustandsklasse werden alle Aktionen implementiert. Jeder Zustand muss Zugriff auf den Kontext des Zustandsautomaten haben, um eventuell den aktuellen Zustand ändern zu können. Daher werden `get`-Methoden für die Zustände und eine `set`-Methode zum Setzen des aktuellen Zustandes hinzugefügt.



State-Muster

Implementierung der Zustandsklassen

```
public class NoQuarterState implements State {
    GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter");
    }

    public void turnCrank() {
        System.out.println("You turned, but there's no quarter");
    }

    public void dispense() {
        System.out.println("You need to pay first");
    }
}
```

Jeder Zustand sollte den Kontext kennen

Zustandsübergang

State-Muster

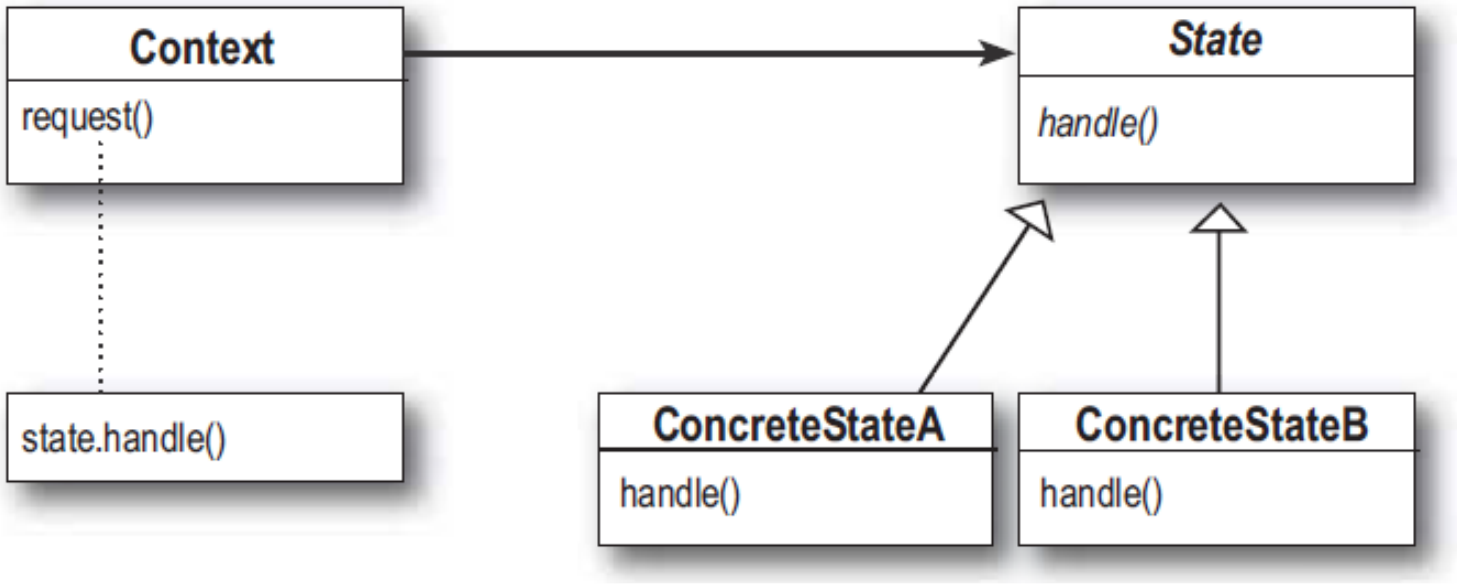
Implementierung der Zustandsklassen

```
public class SoldState implements State {

    GumballMachine gumballMachine;

    public SoldState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }
    public void insertQuarter() {
        System.out.println("Please wait, we're already giving you a gumball");
    }
    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank");
    }
    public void turnCrank() {
        System.out.println("Turning twice doesn't get you another gumball!");
    }
    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}
```

State-Muster

State Muster

Das State Muster ermöglicht einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Das Muster kapselt den Zustand in separaten Klassen. Welche Aktionen möglich sind, wird abhängig von den Zuständen implementiert.



State Muster vs. Strategy Muster

Die beiden Muster unterscheiden sich in der Sichtweise des Objektes, welches flexible bestimmte Methoden einsetzen möchte.

Strategy	Das Client-Objekt gibt vor, mit welchem Strategy-Objekt gearbeitet werden soll. Die Strategy wird dabei eher selten zur Laufzeit verändert.
State	Es wird ein Satz von Verhaltensweisen, die an Zustände gebunden sind definiert. Die Zustände variieren im Laufe der Zeit. Das Client-Objekt weiß in der Regel wenig oder gar nichts über die Zustandsobjekte.

State-Muster

Der Gewinn-Zustand

Es wird eine neue Zustandsklasse implementiert. Die Klasse für den Kaugummiautomat muss nur um diesen Zustand ergänzt werden (sehr kleine Änderung). Der neue Zustand hat Auswirkungen auf andere Zustände (hier nur `HasQuarterState`). Somit müssen auch nur sehr wenige andere Klassen geändert werden.

State-Muster

Implementierung der Zustandsklassen

```
public class WinnerState implements State {
    GumballMachine gumballMachine;
    public WinnerState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }
    ...
    public void turnCrank() {
        System.out.println("Turning again doesn't get you another gumball!");
    }
    public void dispense() {
        System.out.println("YOU'RE A WINNER! You get two gumballs for your quarter");
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() == 0) {
            gumballMachine.setState(gumballMachine.getSoldOutState());
        } else {
            gumballMachine.releaseBall();
            if (gumballMachine.getCount() > 0) {
                gumballMachine.setState(gumballMachine.getNoQuarterState());
            } else {
                System.out.println("Oops, out of gumballs!");
                gumballMachine.setState(gumballMachine.getSoldOutState());
            }
        }
    }
}
```

State-Muster

Implementierung der Zustandsklassen

```
public class HasQuarterState implements State {
    Random randomWinner = new Random(System.currentTimeMillis());
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }
    ...
    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        int winner = randomWinner.nextInt(10);
        if ((winner == 0) && (gumballMachine.getCount() > 1)) {
            gumballMachine.setState(gumballMachine.getWinnerState());
        } else {
            gumballMachine.setState(gumballMachine.getSoldState());
        }
    }
}
```

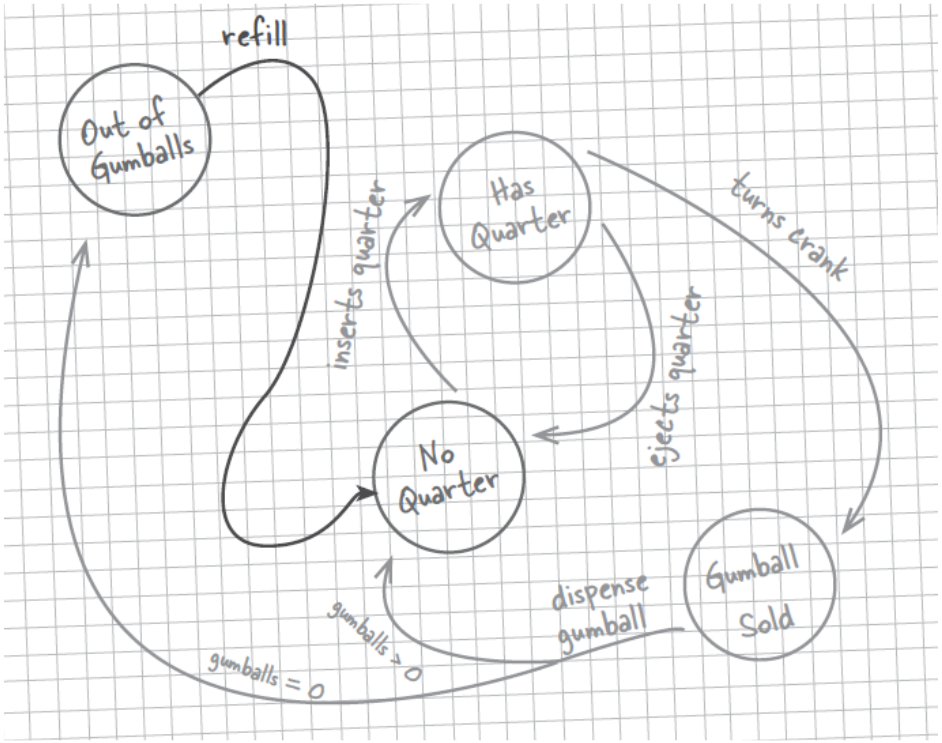
State-Muster

Automatisches Auffüllen

Es soll eine Möglichkeit zum automatischen Auffüllen des Kaugummiautomaten umgesetzt werden.

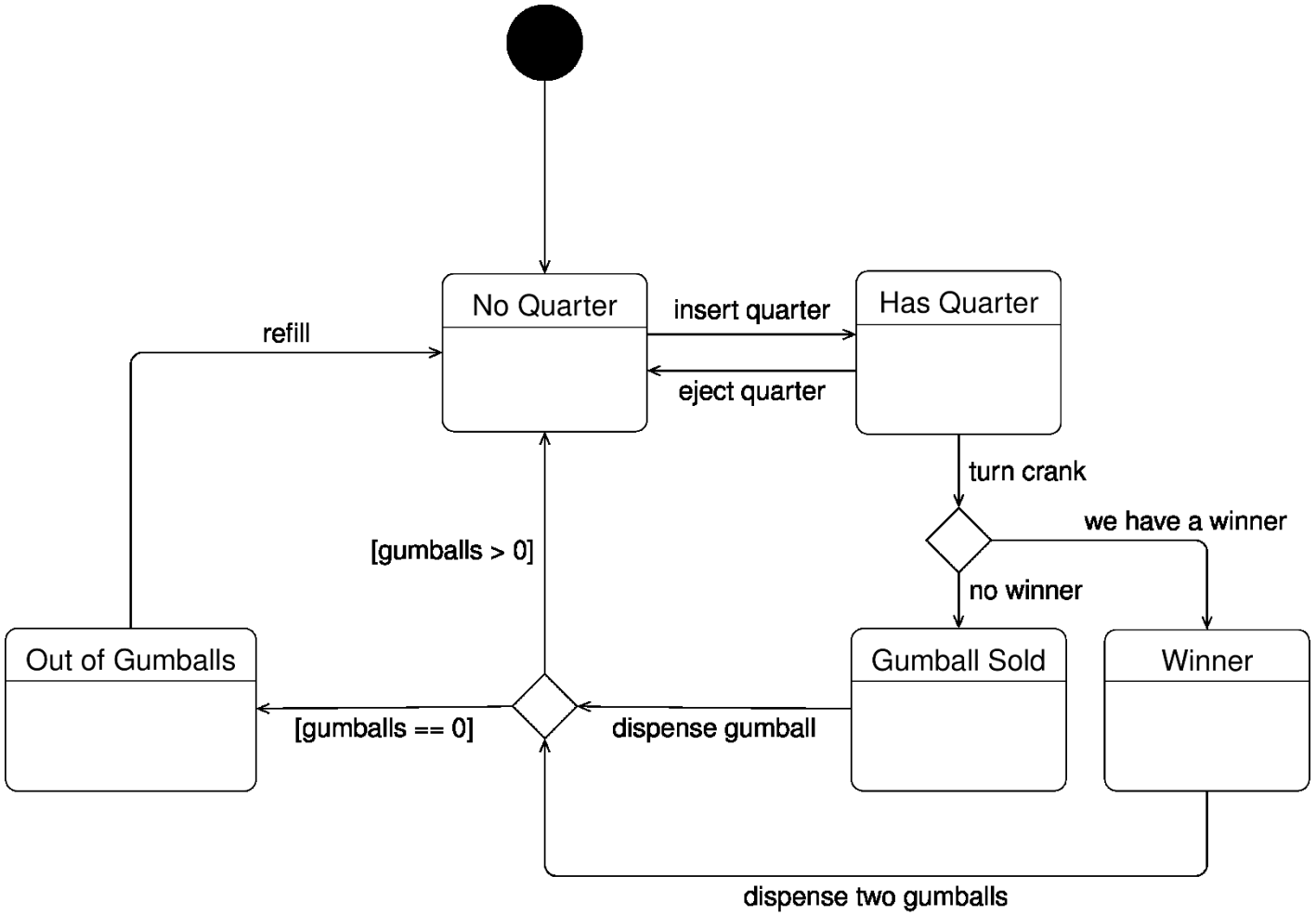
Wie sieht das Zustandsdiagramm aus?

Was muss alles geändert werden?



State-Muster

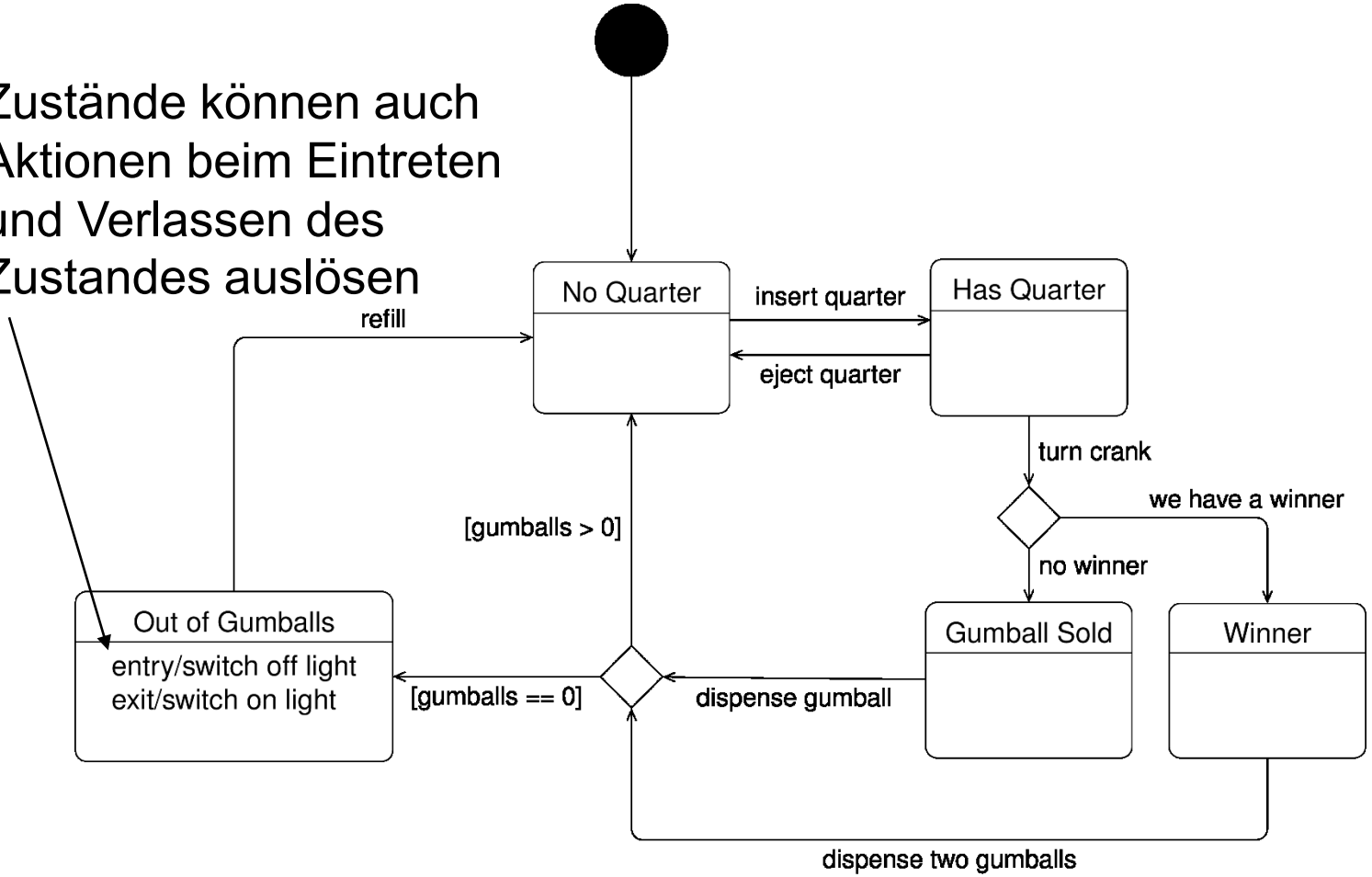
Beispiel - Kaugummiautomat



State-Muster

State Entry & Exit

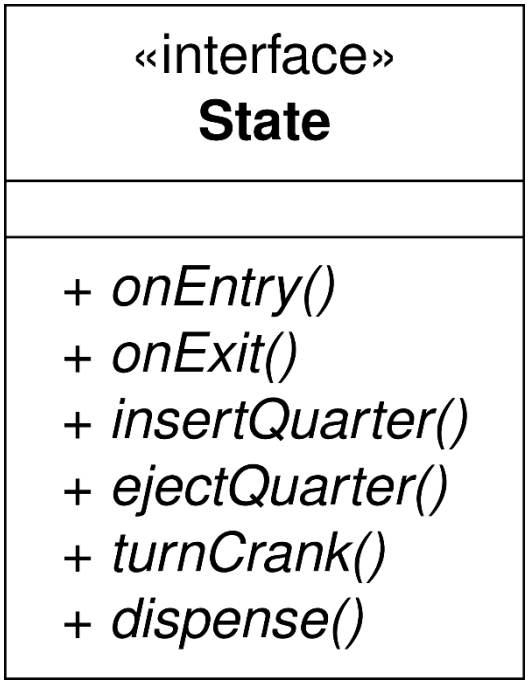
Zustände können auch Aktionen beim Eintreten und Verlassen des Zustandes auslösen



State-Muster

State Entry & Exit

```
public class GumballMachine {
    ...
   public void setState(State state) {
      if(!this.state.equals(state)) {
         this.state.onExit();
         this.state = state;
         state.onEntry();
      }
   }
}
```



RUHR-UNIVERSITÄT BOCHUM
Lehrstuhl für Informatik im Bauwesen

RUB

Programmierung und Programmiersprachen

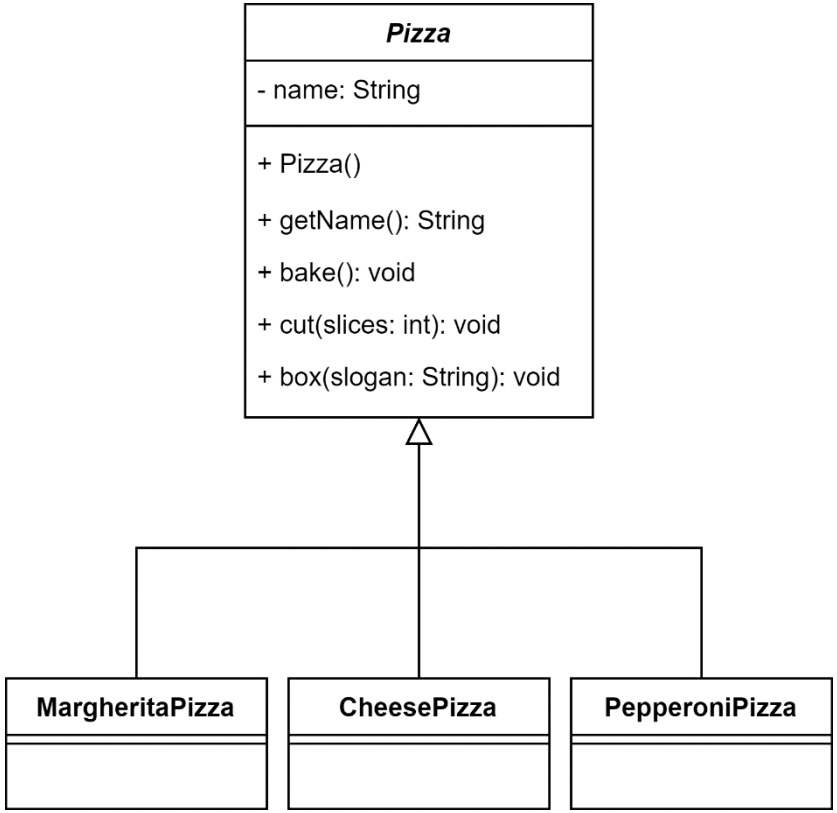
Sommersemester 2023

Factory-Muster

Factory-Muster

Kontrollierte Instanziierung

Gegeben sind folgende Klassen
für das Erstellen von Pizzen.



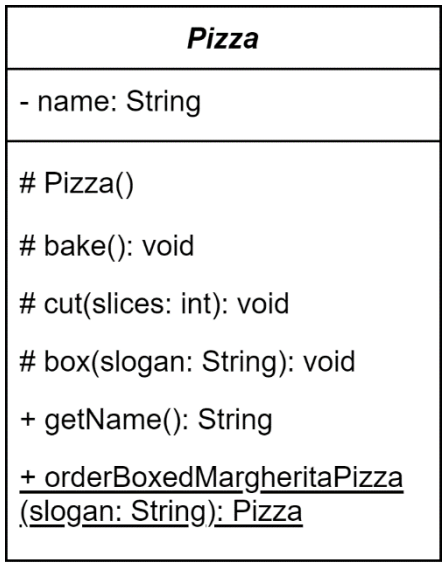
Wie können folgende Anforderungen
sichergestellt werden?

- Nur gebackene Pizzen dürfen ausgeliefert werden
- Pizzen müssen in 8 Teile geschnitten werden
- Box soll den Namen der Pizzeria tragen

Factory-Muster

Fabrikmethode

Instanziieren von Pizzen nur über Fabrikmethoden




```
public abstract class Pizza {
    protected Pizza() {...}

    protected void bake() {...}

    protected void cut(int slices) {...}

    protected void box(String slogan) {...}

    public static Pizza orderBoxedMargheritaPizza(String slogan) {
        Pizza pizza = new MargheritaPizza();
        pizza.bake();
        pizza.cut(8);
        pizza.box(slogan);
        return pizza;
    }
}
```

Fabrikmethode überschattet Konstruktor

Factory-Muster

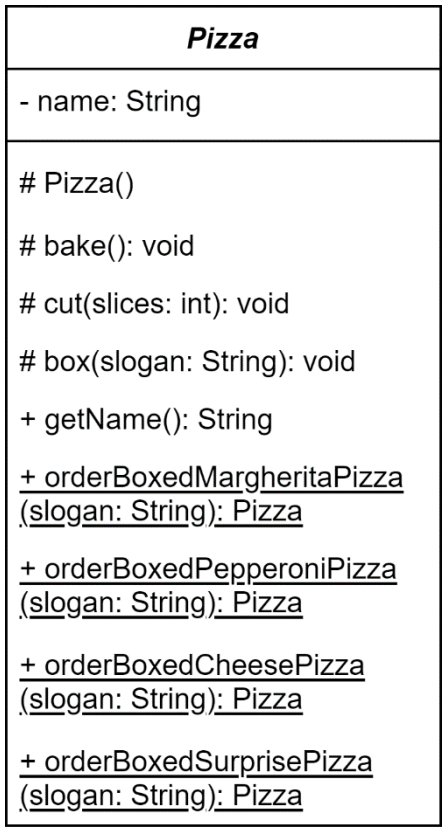
Fabrikmethode

Vorteile:

- Mehrere Fabrikmethoden mit gleichen Parametern möglich
- Späte Entscheidung (*late binding*) der Unterklasse möglich (z.B. Überraschungspizza)

Nachteile:

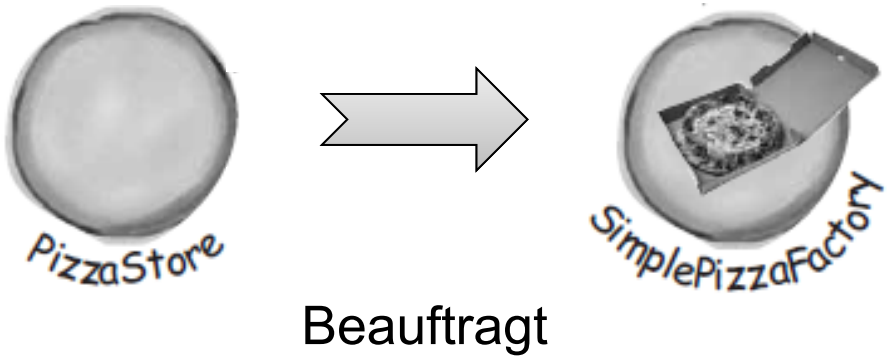
- Eingeschränkte Erweiterbarkeit
- Keine Bindung an die Pizzeria (Slogan kann immer noch frei gewählt werden)



Factory-Muster

Fabrikklassen

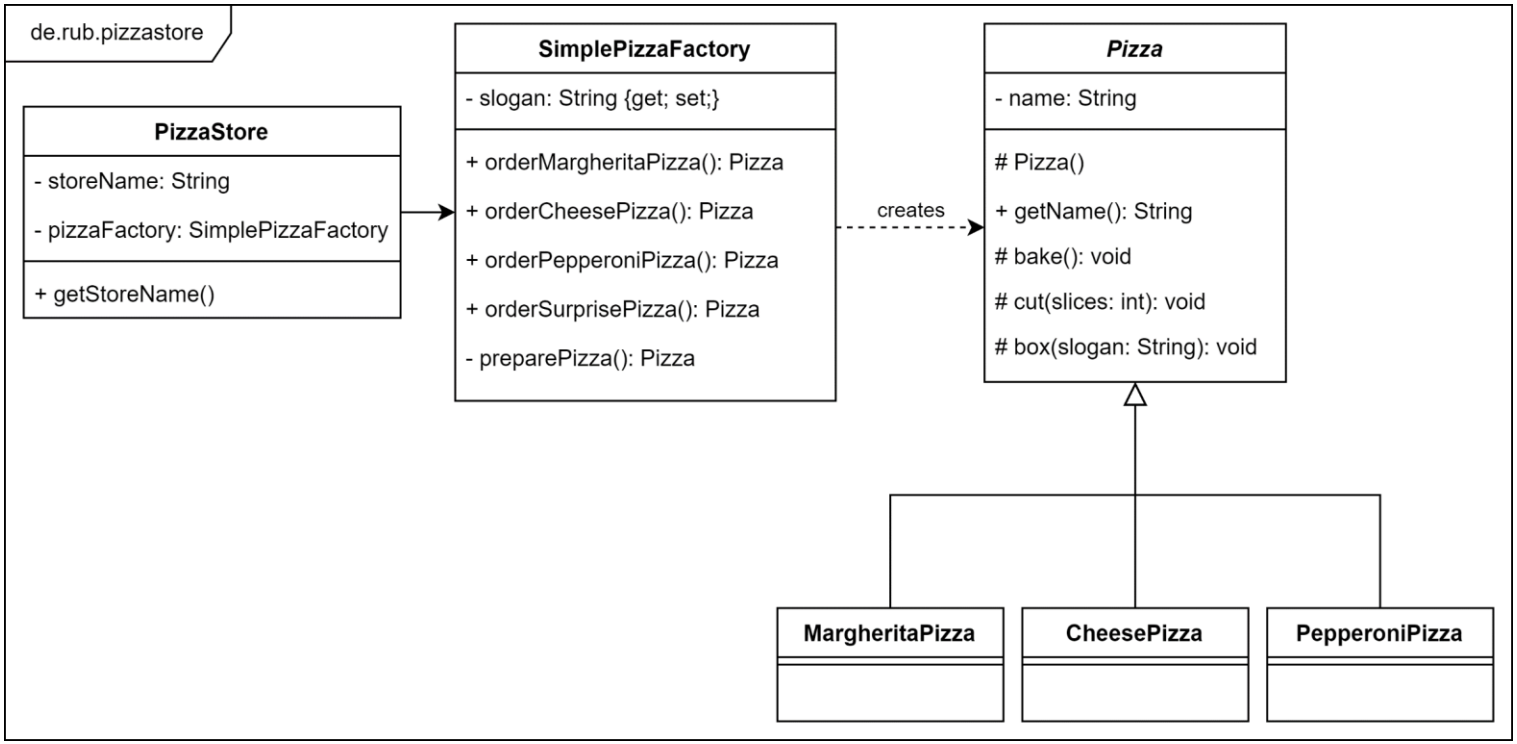
Jedes Mal, wenn eine Pizza benötigt wird, wird die Pizzafabrik (oder Koch) aufgefordert, eine Pizza vom gewünschten Typ zu bereitzustellen.



Factory-Muster

Einfache Fabriken

Auslagern der Instanziierung in eine Factory-Klasse:



Factory-Muster

Einfache Fabriken

```
package de.rub.pizzastore;

public abstract class Pizza {
    protected Pizza() {  }
}

public class SimplePizzaFactory {
    private String slogan;

    public Pizza orderMargheritaPizza() {
        Pizza pizza = new MargheritaPizza();
        preparePizza(pizza);
        return pizza;
    }
    ...
    private void preparePizza(Pizza pizza) {
        pizza.bake();
        pizza.cut(8);
        pizza.box(slogan);
    }
}
```

Konstruktor kann nur innerhalb des Pakets aufgerufen werden

SimplePizzaFactory ist die Fabrikklasse
PizzaStore ist der Client

Fabrikmethode

Anforderungen an das Objekt können sichergestellt werden

Factory-Muster

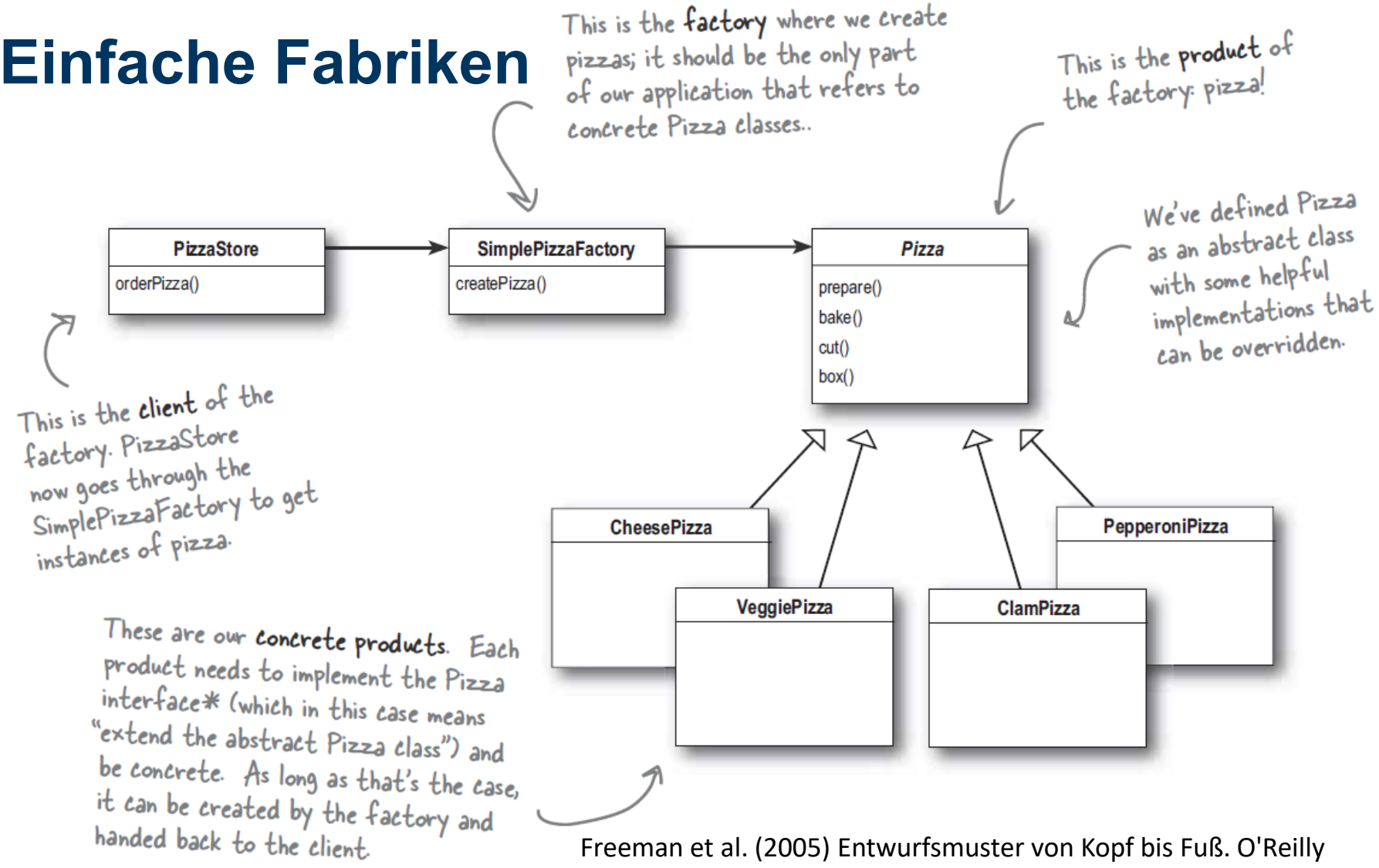
Einfache Fabriken

```
public class SimplePizzaFactory {
    ...
    public Pizza orderSurprisePizza() {
        Random r = new Random();
        int pizzaIndex = r.nextInt(3);

        Pizza p = null;
        if(pizzaIndex == 0) {
            p = new MargheritaPizza();
        } else if(pizzaIndex == 1) {
            p = new CheesePizza();
        } else {
            p = new PepperoniPizza();
        }

        preparePizza(p);
        return p;
    }
}
```

Fabriken erlauben eine späte Entscheidung (late binding) der Unterklasse.

Factory-Muster

Einfache Fabriken



Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

Factory-Muster

Kettenmethoden

Kettenmethoden erlauben eine schnelle Konfiguration eines Objektes:

```
public class SimplePizzaFactory {
    ...

    public SimplePizzaFactory setSlogan(String slogan) {
        this.slogan = slogan;
        return this;
    }
}
```

Eine Kettenmethode gibt das eigene Objekt (`this`) zurück.

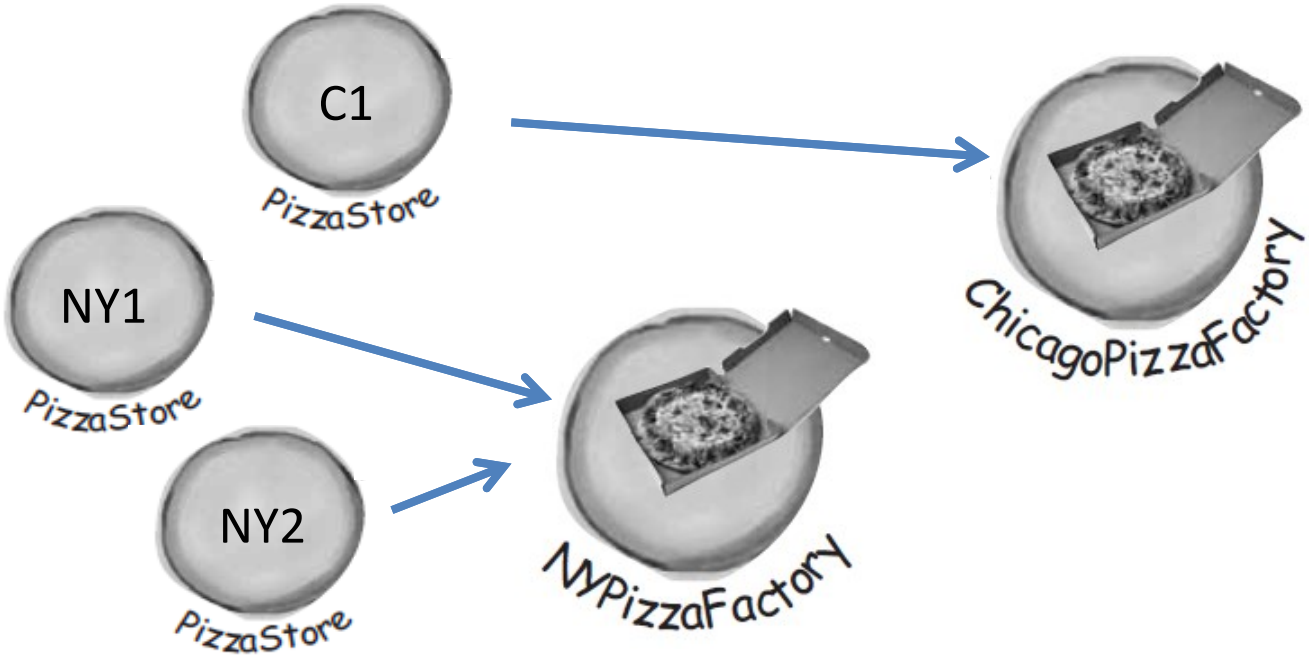
```
public static void main(String[] args) {
    SimplePizzaFactory factory = new SimplePizzaFactory();
    Pizza p = factory.setSlogan("Pizza Time!")
                .setSlices(6)
                .orderMargherita();
}
```

Factory-Objekt kann „verkettet“ konfiguriert werden.

Factory-Muster

Abstrakte Fabriken

Im nächsten Schritt soll die Anwendung so erweitert werden, dass mehrere Zweigstellen einer Pizzeria eröffnet werden können. Dabei soll je nach Zweigstelle ein anderes Angebot angeboten werden.



Factory-Muster

Abstrakte Fabriken

Implementierung

```
public abstract class AbstractPizzaFactory {
    private String slogan;
    private int slices;

    public abstract Pizza orderPizza(PizzaType type);

    ...
}
```

Pizzerien können unterschiedliche Angebote haben

```
public enum PizzaType {
    MARGHERITA_PIZZA, PEPPERONI_PIZZA, CHEESE_PIZZA,
    VEGGIE_PIZZA, MEAT_PIZZA
}
```

Mögliche Bestellungen können durch ein Enum aufgelistet werden

Factory-Muster

Abstrakte Fabriken

Implementierung

```
public class NYPizzaFactory extends AbstractPizzaFactory {
    @Override
    public Pizza orderPizza(PizzaType type) {
        Pizza pizza = null;
        if(type == PizzaType.MARGHERITA_PIZZA) {
            pizza = new MargheritaPizza();
        } else if (type == PizzaType.CHEESE_PIZZA) {
            pizza = new CheesePizza();
        } else if (type == PizzaType.PEPPERONI_PIZZA) {
            pizza = new PepperoniPizza();
        } else if (type == PizzaType.VEGGIE_PIZZA) {
            pizza = new VeggiePizza();
        } else {
            return null;  // or raise an exception
        }

        preparePizza(pizza);
        return pizza;
    }
}
```

Factory-Muster

Abstrakte Fabriken

Implementierung

```
public class ChicagoPizzaFactory extends AbstractPizzaFactory {
    @Override
    public Pizza orderPizza(PizzaType type) {
        Pizza pizza = null;
        if(type == PizzaType.MARGHERITA_PIZZA) {
            pizza = new MargheritaPizza();
        } else if (type == PizzaType.CHEESE_PIZZA) {
            pizza = new CheesePizza();
        } else if (type == PizzaType.PEPPERONI_PIZZA) {
            pizza = new PepperoniPizza();
        } else if (type == PizzaType.MEAT_PIZZA) {
            pizza = new MeatPizza();
        } else {
            return null;  // or raise an exception
        }

        preparePizza(pizza);
        return pizza;
    }
}
```

Factory-Muster

Abstrakte Fabriken

Implementierung

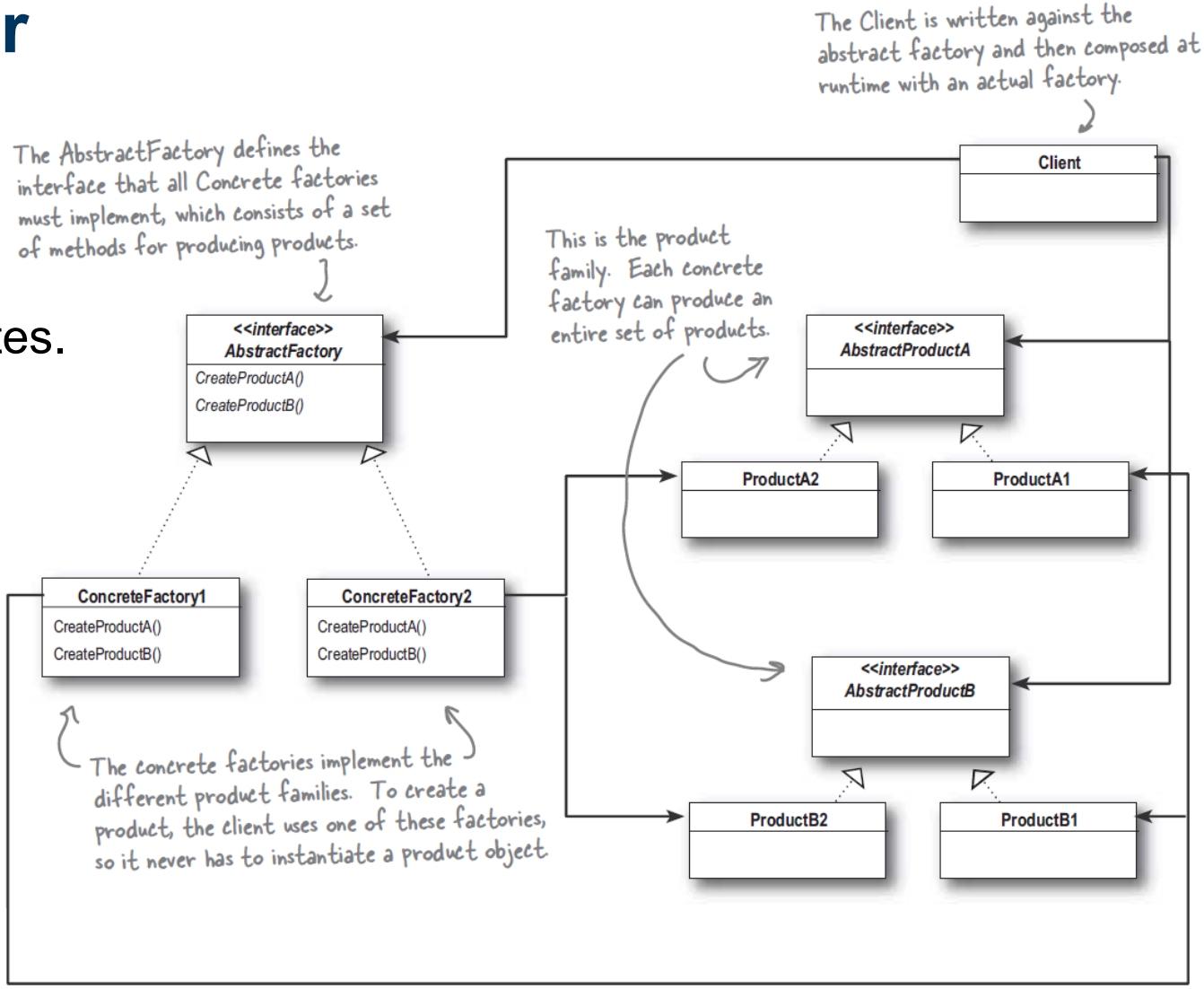
```
public class PizzaStoreTest {
    public static void main(String[] args) {

        NYPizzaFactory nyFactory = new NYPizzaFactory();

        PizzaStore ny1 = new PizzaStore("NY1", nyFactory);
        PizzaStore ny2 = new PizzaStore("NY2", nyFactory);

        ChicagoPizzaFactory cFactory = new ChicagoPizzaFactory();

        PizzaStore c1 = new PizzaStore("C1", cFactory);

        Pizza pizza1 = ny1.orderPizza(PizzaType.VEGGIE_PIZZA);
        System.out.println("Pizza 1: " + pizza1);
        Pizza pizza2 = ny2.orderPizza(PizzaType.MEAT_PIZZA);
        System.out.println("Pizza 2: " + pizza2);
        Pizza pizza3 = c1.orderPizza(PizzaType.MEAT_PIZZA);
        System.out.println("Pizza 3: " + pizza3);
    }
}
```

Factory-Muster

Factory-Muster

Das Factory-Muster definiert eine Schnittstelle zur Erstellung eines Objektes.

Die Unterklassen entscheiden, welche Klassen instanziiert werden sollen.



Factory-Muster

Variation der Zutaten

Gegeben sind die folgenden Speisekarten. Wie können die Pizzen je nach Pizzafabrik individualisiert werden?

Freeman et al. (2005) Entwurfsmuster von Kopf bis Fuß. O'Reilly

Factory-Muster

Fazit

Eine Factory-Klasse kapselt die Erstellung von konkreten Objekten. Der Zweck einer Factory ist es somit, Familien verwandter Objekte zu erstellen, ohne von konkreten Klassen abhängig zu werden.

Das Factory-Muster hat drei Komplexitätsstufen:

Factory-Methode, Factory-Klasse und Abstract Factory

Bisherige OO-Prinzipen

- *Kapseln, was variiert*
- *Komposition vor Vererbung*
- *Programmierung auf eine Schnittstelle*
- *Lockere Bindung von Objekten*
- *Klassen sollen für Erweiterung offen, aber für Veränderung geschlossen sein*