CAMPUS 02 GRAZ
FACHHOCHSCHULE DER WIRTSCHAFT

SOFTWARE DESIGN FOLIENSATZ 4

Factory Pattern

Bernhard Fuchs

ZAM - WS 2025/26

TERMINE

0004VD2005 SOFTWARE DESIGN (29,75UE IL, WS 2025/26)

Gruppe

	Tag	Datum	von	bis	Ort	Ereignis	Termintyp	Lerneinheit	Vortragende*r	Anmerkung
	Standardgruppe									
	Do	22.01.2026	08:15	11:45	CZ106	Abhaltung	fix	**1 Organisation + Strategy Pattern**		
	Do	29.01.2026	08:15	11:45	CZ106	Abhaltung	fix	**2 Decorator Pattern + Singleton**		
	Do	05.02.2026	12:30	16:00	CZ106	Abhaltung	fix	**3 Observer**		
Heute	Fr	06.02.2026	08:15	12:15	CZ106	Abhaltung	fix	**4 Factory**		
	Do	12.02.2026	08:15	13:00	CZ106	Abhaltung	fix	**5 Command + Adapter**		
	Fr	13.02.2026	08:15	12:15	CZ106	Abhaltung	fix	**6 Facade, Template, Iterator** mit Christian Hofer!		
	Do	19.02.2026	08:15	11:45	CZ106	Abhaltung	fix	**Wiederholung**		
	Fr	20.02.2026	08:15	12:15	CZ106	Prüfungstermin	fix	**Prüfung**		

PIZZARESTAURANT

Quelle: https://unsplash.com/photos/MQUqbmszGGM

FERTIGER FINALER CODE –
WAS KÖNNEN WIR ERKENNEN?

```
1   package at.campus02.swd;
2
3   public class Main {
4       public static void main(String[] args) {
5           PizzaStore gStore = new GrazPizzaStore();
6           PizzaStore wStore = new WienPizzaStore();
7
8           Pizza pizza1 = gStore.orderPizza("mushroom");
9           System.out.println("Ich bestelle eine " + pizza1.getName());
10
11          Pizza pizza2 = wStore.orderPizza("veggie");
12          System.out.println("Ich bestelle eine " + pizza2.getName());
13
14      }
15
16  }
```

Aber: beginnen wir von vorne.

Es folgen zwei Schritte:

Schritt 1) unelegante Lösung

Schritt 2) bessere Lösung: Factory Pattern.

PIZZARESTAURANT

```
Pizza orderPizza() {
    // soll ein Interface sein damit
    // wir flexibel bleiben
    Pizza pizza = new Pizza();


    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

UNTERSCHIEDLICHE PIZZEN

```
Pizza orderPizza(String type) {
    Pizza pizza;
    if(type.equals("cheese")){
        pizza = new CheesePizza();
    }else if(type.equals("greek")){
        pizza = new GreekPizza();
    }else if(type.equals("pepperoni")){
        pizza = new PepperoniPizza();
    }
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

OBERE TEIL VERÄNDERT SICH -> AB IN DIE FACTORY

Quelle: https://unsplash.com/photos/XaSr29oo5vo

SIMPLE FACTORY (NOCH NICHT DIE RICHTIGE LÖSUNG)

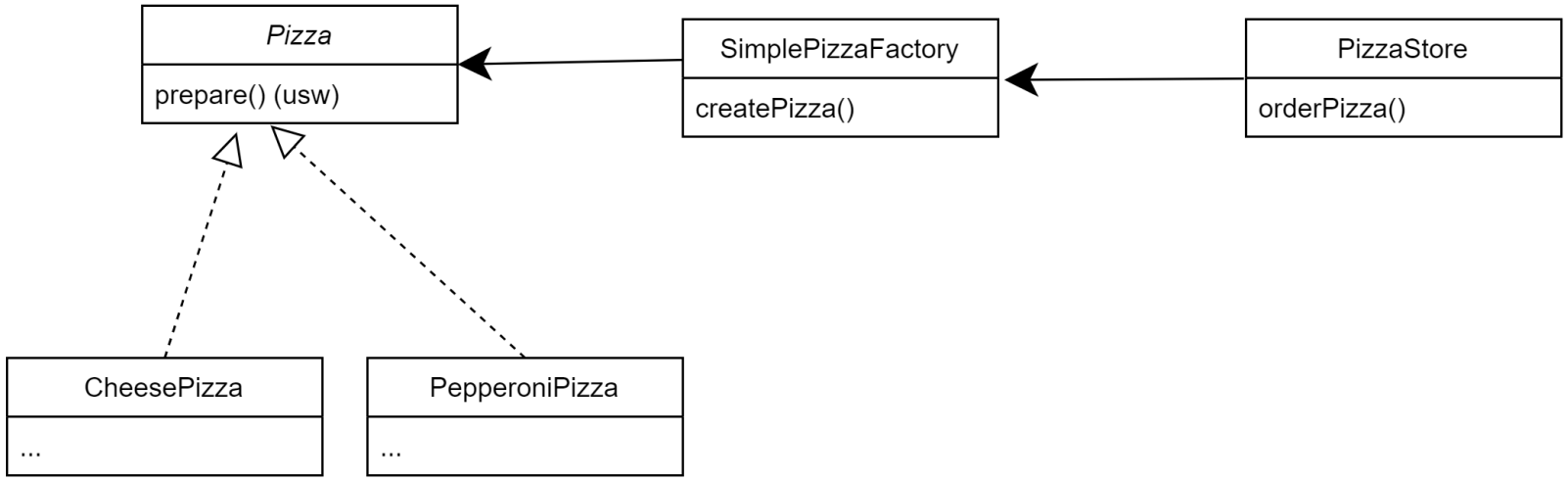
```
SimplePizzaFactory factory;

Pizza orderPizza(String type) {
    Pizza pizza;

    // Kapselung der Pizzaerzeugung
    // keine konkrete Objekterzeugung hier
    // (kein new yaayy)
    pizza = factory.createPizza(type);

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

SIMPLE FACTORY (NOCH NICHT DIE RICHTIGE LÖSUNG)

Produkt der Fabrik

Factory sollte einzige Ort sein der konkrete Pizza Klassen verwendet

Klient der factory der sich Pizzen erzeugen lässt



Konkrete Produkte die Interface umsetzen

ERÖFFNEN FRANCHISE FILIALEN

- Wir möchten regional unterschiedliches Angebot
  - Brauche weitere Factory Klassen

FRANCHISE PARTNER SPAREN AN JEDEM ENDE

- Billige Verpackung
- Pizzen werden nicht geschnitten

- Verwenden nicht unsere mit Liebe entwickelte PizzaStore Funktionalität

FACTORY PATTERN

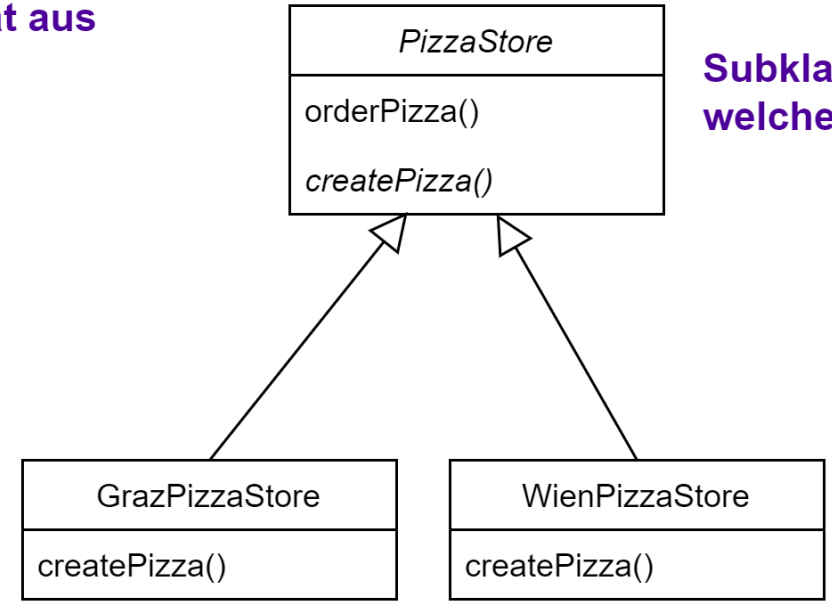
```
public abstract class PizzaStore {// Abstrakte Klasse
    Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    // Subklassen sollen implementieren
    abstract Pizza createPizza(String type);
    }
```

FACTORY PATTERN: CREATOR KLASSE

Subklassen müssen Methode überschreiben, aber zugleich verwenden alle Subklassen Funktionalität aus PizzaStore

orderPizza ruft createPizza auf

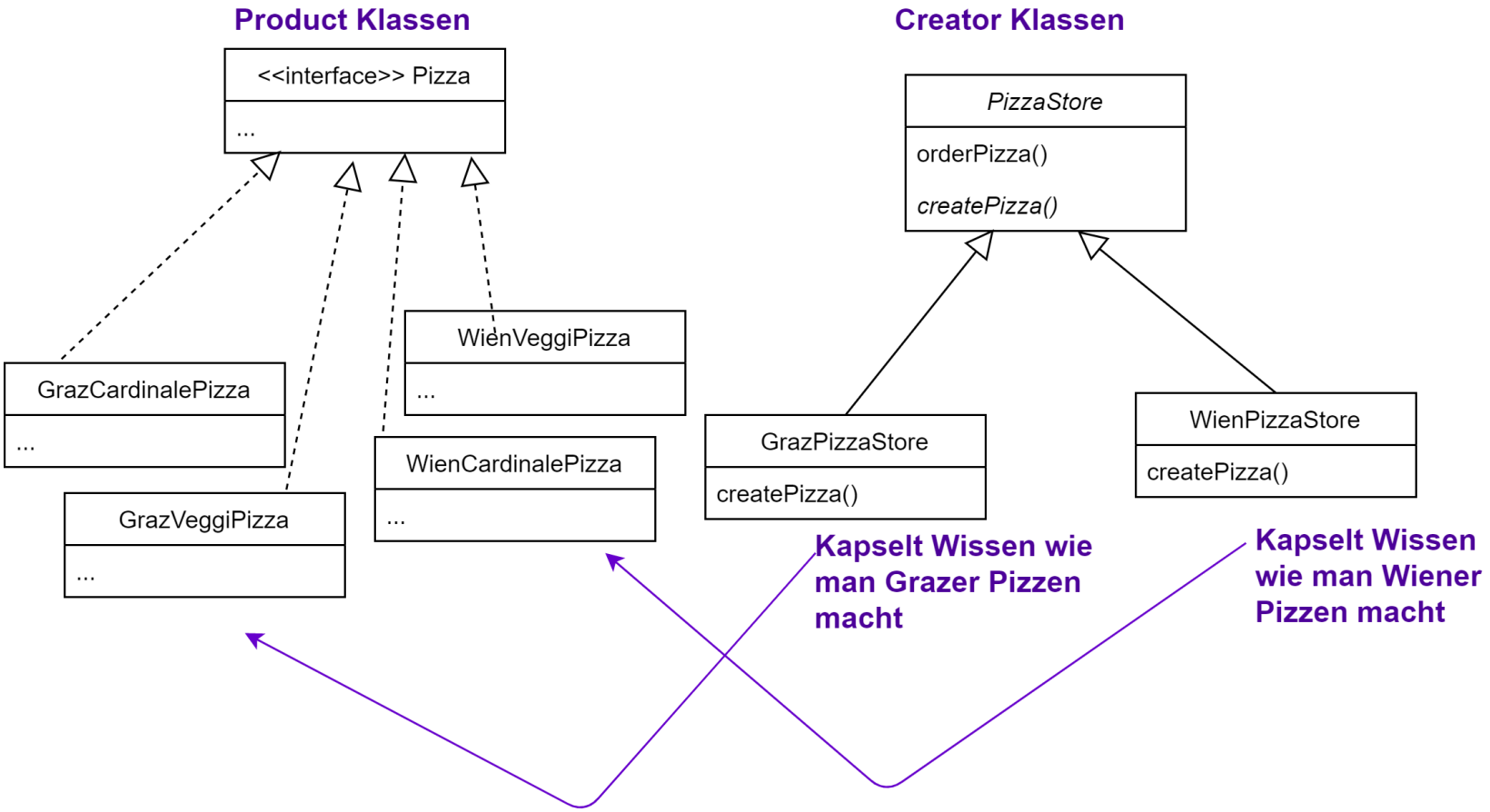
Subklasse entscheidet welche Pizza



Dünne italienische Pizzen (Graz ist fast schon in Italien :))

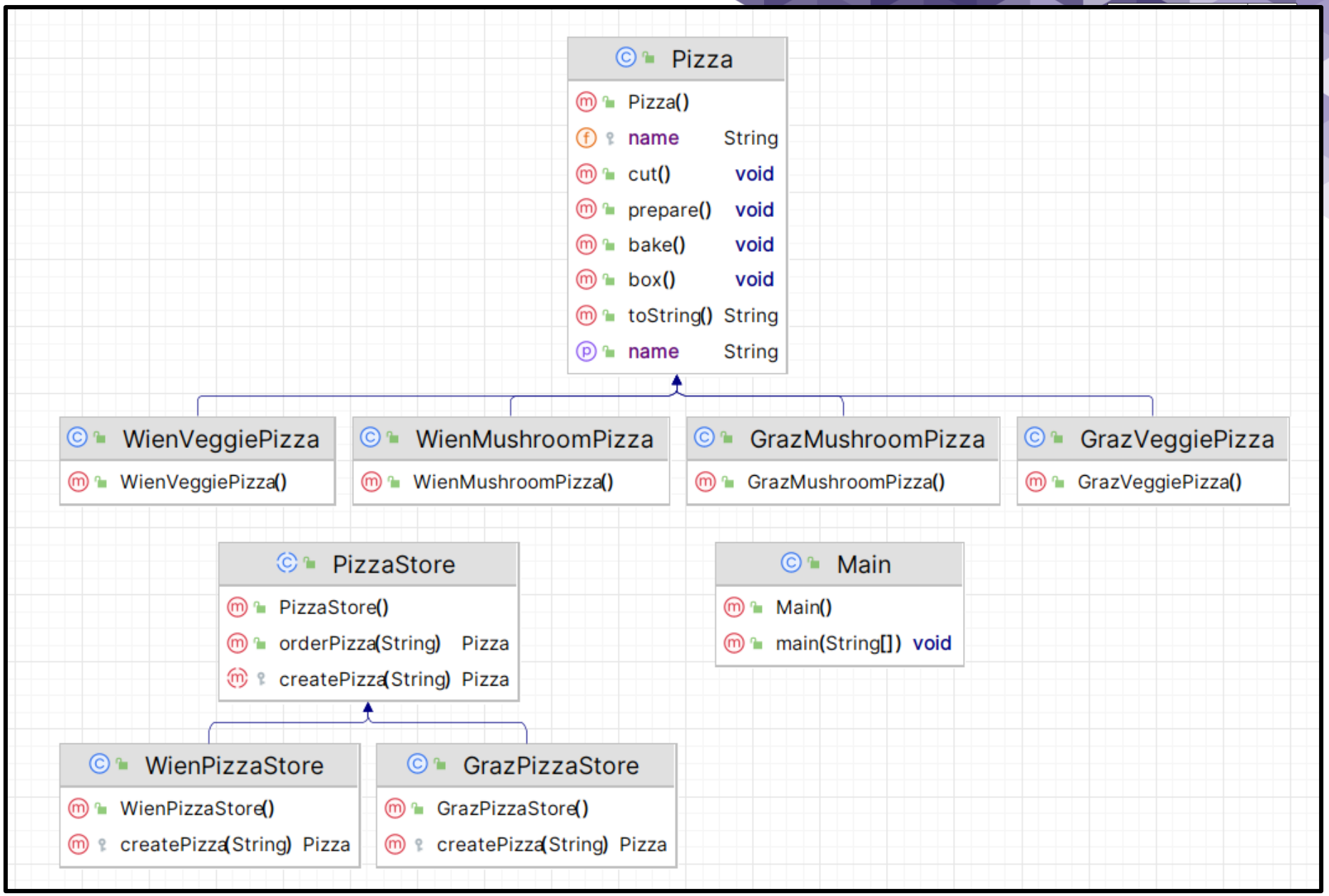
Standardpizzen

FACTORY PATTERN



FACTORY PATTERN

- Let‘s code :)

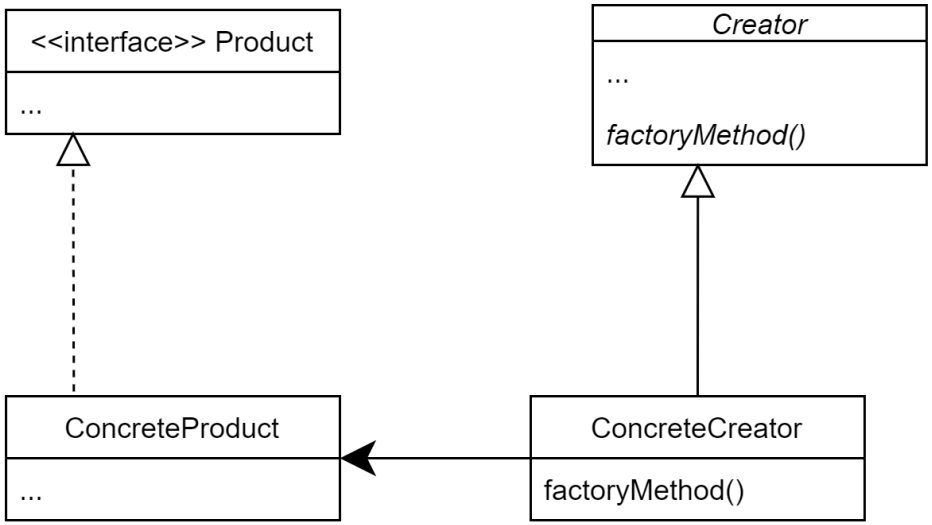

Pizza
Pizza()
name String
cut() void
prepare() void
bake() void
box() void
toString() String
name String
WienVeggiePizza
WienVeggiePizza()
WienMushroomPizza
WienMushroomPizza()
GrazMushroomPizza
GrazMushroomPizza()
GrazVeggiePizza
GrazVeggiePizza()
PizzaStore
PizzaStore()
orderPizza(String) Pizza
createPizza(String) Pizza
Main
Main()
main(String[]) void
WienPizzaStore
WienPizzaStore()
createPizza(String) Pizza
GrazPizzaStore
GrazPizzaStore()
createPizza(String) Pizza

FACTORY PATTERN

Definiert eine Schnittstelle um Objekte zu erzeugen, aber **lässt Subklassen entscheiden**, welche Klasse zu erzeugen ist. Factory Methode ermöglicht es die **Erzeugung** von Objekten an Subklassen zu **delegieren.**

FACTORY PATTERN GENERISCH

Alle Produkte implementieren das Interface, damit benutzende Klassen nicht die konkreten Klassen verwenden müssen



Enthält Implementierung für alle möglichen Methoden die Produkt verändern - mit Ausnahme der Factory

Erzeugt eines oder mehrere Produkte - nur diese Klasse weiß wie man das Produkt erzeugt

DEPENDENCY INVERSION PRINCIPLE

- Abhängigkeit: Klasse A ist abhängig von Klasse B wenn eine Änderung in B zur Folge hat, dass auch Klasse A sich ändern muss
- Pizzastore ist **abhängig** von Pizza
- Klasse Pizza ist **abhängig** von konkreten Pizzen

DEPENDENCY INVERSION PRINCIPLE

- Hänge von Abstraktionen und nicht von konkreten Klassen ab. (Kurze Version)

FACTORY BEISPIEL

CAMPUS 02 GRAZ
FACHHOCHSCHULE DER WIRTSCHAFT

1. Der Autofabrikant „Krisentrotz“ stellt verschiedene Auto-Typen her: PKW, LKW, Sportwagen.
2. Krisentrotz besitzt Fabriken in Deutschland und England. An beiden Standorten werden diese verschiedenen Auto-Typen auf unterschiedliche Weise hergestellt.
3. (Beachten Sie auch die mögliche Expansion von Krisentrotz-Fabriken in weitere Länder, z.B. Mexiko.)
4. Modellieren Sie die Herstellung der Autos an beiden Standorten und implementieren Sie diese.
5. Ein *enum* spezifiziert welche Art von Fahrzeug hergestellt werden soll.
6. Die englische Fabrik spezialisiert sich auf Fahrzeuge größer 3,5 Tonnen, während in Deutschland der Rest produziert wird.
7. Jedes Fahrzeug soll eine hupen und waschen Methode haben und diese sollen nach der Produktion zum Funktionstest ausgeführt werden. Simulieren Sie die Logik der Methoden mit Konsolenausgaben.