Abstract Factory Pattern Aufgabe

Folgendes Codestück für einen Pizza Online Händler ist gegeben:

```
public class PizzaOnlineStore {

	public Pizza orderPizza(String pizzaType) {
		Pizza pizza = null;

		switch (pizzaType) {

		case "Cheese":
			pizza = new CheesePizza();
			break;
		case "Vegetarian":
			pizza = new VegetarianPizza();
			break;
		case "Pepperoni":
			pizza = new PepperoniPizza();
			break;
		}

		pizza.prepare();
		pizza.bake();
		pizza.cut();
		pizza.box();

		return pizza;
	}
}
```

Häufig ergibt sich folgendes Problem (vgl. hierzu auch Kap 3.2 im Skript):

In diesem Codestück hängt der Bestellvorgang einer Pizza von der Erzeugung der unterschiedlichen Pizzen ab. Wenn also ein neuer Pizzatyp hinzukommt, muss dieses Codestück modifiziert und neu compiliert werden.

Diesem Codebeispiel liegt also folgendes UML Diagramm zugrunde:

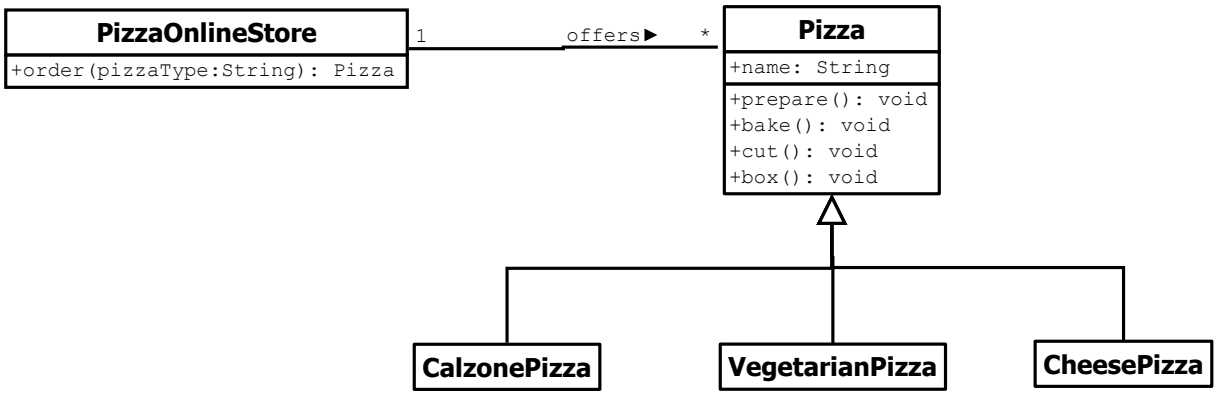

Abb. 1: UML Diagramm PizzaOnlineStore (unvollständig)

Zu dieser Klassenhierarchie gehört folgendes Testprogramm:

```
public class PizzaOnlineStoreTest {

	public static void main(String[] args) {

		PizzaOnlineStore store = new PizzaOnlineStore();

		Pizza pizza = store.orderPizza("Cheese");
		System.out.println("We ordered a " + pizza);

		pizza = store.orderPizza("Vegetarian");
		System.out.println("We ordered a " + pizza);

		pizza = store.orderPizza("Calzone");
		System.out.println("We ordered a " + pizza);
	}
}
```

Wenn man dieses Testprogramm ausführt, werden folgende Ausgaben erzeugt:

```
Preparing 5 min Cheese Pizza
Baking 25 min at 350°Cheese
Cut Cheese
Boxing Cheese
We ordered a ----Cheese----

Preparing 3 min Vegetarian Pizza
Baking 20 min at 350° Vegetarian
Cut Vegetarian
Boxing Vegetarian
We ordered a ----Vegetarian----

Preparing 6 min Calzone Pizza
Baking 30 min at 350° Calzone
No Cut Calzone
Choosing special box for Calzone
We ordered a ----Calzone----
```

Aufgabe:

1. Implementieren Sie alle Klassen `PizzaOnlineStore`, `PizzaOnlineStoreTest`, `Pizza`, `CalzonePizza`, `CheesePizza` und `VegetarianPizza` und zwar so, dass **die genannten Testausgaben exakt wie angegeben erzeugt werden.**

2. a. Um das geschilderte Problem der „Vermischung“ von Erzeugung und Bestellung besser zu lösen, ändern Sie das Beispiel unter **Anwendung des Abstract Factory Patterns** entsprechend ab.
   b. Fügen Sie in diese neue Implementierung eine neue Pizzasorte `PepperoniPizza` ein. Diese Pizza muss 27 min gebacken werden und bedarf einer Vorbereitungszeit von 4 min.
   Machen Sie sich alle notwendigen Änderungen klar.

3. Die bis jetzt implementierte Factory spiegelt die Welt nur sehr unzureichend wieder, da:
   a. es **verschiedene** PizzaBäcker gibt, unser Beispiel aber momentan nur von einem ausgeht.
   b. die verschiedenen PizzaBäcker **verschiedene Rezepturen** für die Pizzen haben:
   Beispielsweise belegt
   der PizzaBäcker `PizzeriaAlberto` eine CheesePizza mit:
   - `MozzarellaCheese`
   - `Tomato sauce`
   - `Oregano`

   der PizzaBäcker `PizzeriaMakkaroni` eine CheesePizza mit:
   - `GoudaCheese`
   - `Diced Tomato sauce`
   - `Pepper`

   Aus diesem Grunde wird die Oberklasse `Pizza` erweitert:

```
public abstract class Pizza {
       String name;
       ArrayList<String> toppings = new ArrayList<String>();

       public void prepare()     { …    }
       public void bake()        { …    }
       public void cut()         { …    }
       public void box()         { …    }

       public String toString()  { …    }

}
```

3.1 Implementieren Sie eine `PizzeriaAlbertoFactory` und `PizzeriaMakkaroniFactory` und führen auch die notwendigen Klassen auf der Pizza Klassenhierarchie ein (z.B. `AlbertoCheesePizza`, `MakkaroniCheesePizza`. Die Pizza Hierarchie implementieren Sie für die Sorten Cheese, Vegetarian und Calzone. Bei den jeweiligen Belägen können Sie kreativ sein (oder unter www.chefkoch.de nachschauen ;-))
3.2 Ändern Sie auch Ihr Testprogramm ab und erzeugen mal Pizzen von Alberto und mal Pizzen von Makkaroni.
3.3 Zeichnen Sie ein UML Klassendiagramm für Ihre Klassenhierarchie(n).
3.4 Machen Sie sich die Konsequenzen dieses Pattern klar.

4. **Optional**: Die Erzeugung von Objekten in unserem Beispiel ist String basiert. Dies kann man verbessern, indem man statt Strings das Reflection API verwendet d.h. z.B. die Klasse `AlbertoCheesePizza.class`.
Ändern Sie die Erzeugung Ihrer Objekte so ab, dass nur noch die Metaklassen statt Strings verwendet werden. Machen Sie sich auch die Vor- und Nachteile einer `String` versus `.class` basierten Implementierung klar.