Samenvatting van de cursus Advanced Software Development I

Robbe Magerman

11/02/2025

Inhoudstafel

1 Design Patterns

1.1 Factory Method

1.1.1 Hoe herken je dit?

- Je kan dit herkennen door het volgende voorbeeld:
  - 1 familie, compleet object
    - Iedereen van de familie mag dus meedoen
    - **SimpleFactory**
  - 1 familie, compleet object, niet hele familie, keuze op voorhand
    - Niet iedereen van de familie mag meedoen
    - **Factory Method**
  - 1 familie, stukjes uitkiezen
    - Niet alle familieleden mogen iets doen
    - **Abstract Factory**
  - Je familie wordt groter maar ze mogen niets meer doen
- Het aanmaken van objecten kan dus veranderen aan de hand van het verhaal

1.1.2 Theorie

Theorie - slides:

- Het **Factory Method Pattern** definieert een interface voor het creëren van een object, maar laat de subklassen beslissen welke klasse er geïnstantieerd wordt. De Factory Method draagt de instanties over aan de subklassen
- Wees afhankelijk van abstracties. Wees niet afhankelijk van concrete klassen
- Hiermee wordt bedoelt dat je programmeert naar een Interface
- Het principe suggereert dat onze highlevelcomponenten niet afhankelijk mogen zijn van onze low-levelcomponenten.
- Beiden zouden moeten afhangen van abstracties.

Notities uit de les:

- Het voorbeeld hier rond in **ASDI** was hier een **PizzaFactory**
- Zie hiervoor mijn vorige samenvatting van ASDI
- Nu gaan we dit uitbreieden, stel dat nu meerdere winkels hebben die verschillende pizza's verkopen, hoe doen we dit dan?
- We maken van `PizzaStore` een abstract methode en laten hier **PizzaStore-kinderen** van erven
- De `createPizza`-methode is vanaf nu **abstract** en **protected**

1.1.3 UML

1.1.4 Implementatie

- We maken onze **abstracte** klasse `domein/Pizza`:
- We weten dat we meerdere **Stores** hebben dus we maken alvast Pizza's volgens al onze verschillende Stores:

1.2 Abstract Factory

1.2.1 Hoe herken je dit?

- Meerdere families en je maakt telkens iets van een bepaald iets (kleine stukjes)
- Dus je hebt meerdere keuzes iets en moet één bepaald iets ervan kiezen van die x-aantal keuzes

1.2.2 Theorie

Theorie - slides:

- Een abstract Factory levert een interface voor een reeks producten. In ons geval alle dingen die nodig zijn om een pizza te maken: deeg, saus, kaas, vleeswaren en groenten.

- We schrijven onze code zodanig dat deze de fabriek gebruikt voor het maken van producten. Door een verscheidenheid aan fabrieken krijgen we een verscheidenheid aan implementaties voor de producten. Maar onze clientcode blijft hetzelfde.
- Pizza gemaakt van de ingredienten vervaardigd door een concrete fabriek.
- Het **Abstract Factory Pattern** levert een interface voor de vervaardiging van reeksen gerelateerde of afhankelijke objecten zonder hun concrete klassen te specificeren.
- Werkwijze volgens de 5-stapsmethode:
  - 1Maak per gedrag een **interface**
    - Deeg, Saus, Kaas, Groenten ...
  - Maak voor elke gedrag/familie een concrete **klasse** aan
    - Deeg: DunneKorst, DikkeKorst
    - Saus: MarinareSaus, SpaghettiSaus ...
  - Maak een **abstract factory (altijd een interface)**. Voeg een create method toe per soort object
    - createDeeg() : Deeg - returnt Deeg
    - createSaus() : Saus - returnt Saus
  - Maak **per familie** factory **een concrete factory**
    - PizzaIngredientFactory
      - GentPizzaIngredientFactory
      - OostendePizzaIngredientFactory
  - Injecteer de factory in de client
    - We **verbinden hoofdklasse Pizza met de PizzaIngredientFactory**

Notities uit de les:

- We breiden onze vorige applicatie nog 1x uit waarop nu de **ingredienten** zullen veranderen
- bv. ingredient Kaas
  - Mozzarella
  - Cheddar ...
- Dit moeten we uitbreiden via een `IngredientenFactory`
- Deze factory heeft bv. `createSaus` - `createKaas` - `createKorst` waarvan 2 klassen erven **GentPizzaIngredientenFactory** en **OostendePizzaIngredientenFactory** ### UML

1.2.3 Implementatie

- Het startpunt van dit design pattern is via het project dat ik in mijn vorige cursus heb gemaakt, zie https://github.com/Robbe04/samenvattingen/blob/main/Samenvattingen_Semester_1_2024_2025/Cursus_Advanced_Sof
- !LET OP! Verwijder de klasse `domein/Margaritta` zodat we het onszelf niet te moeilijk maken !LET OP!
- Nu we dat hebben gaan we van start met ons project
- We beseffen dat we ook **regionale PizzaFactorys** willen omdat **Pizza's kunnen verschillen**
- Daarom maken we van `domein/PizzaStore` nu een **abstracte** klasse met een **protected** methode `createPizza` :

```
public abstract class PizzaStore {
    public Pizza bestelPizza(String teBestellenPizza) {
        Pizza pizza;
        pizza = createPizza(teBestellenPizza);
        pizza.prepare();
        pizza.deliver();
        return pizza;
    }

    protected abstract Pizza createPizza(String teBestellenPizza);
}
```

- Na dit bouwen we onze 2 nieuwe stores:
- `domein/PizzaStoreGent` :

```
public class PizzaStoreGent extends PizzaStore {
```

```
    @Override
    protected Pizza createPizza(String teBestellenPizza) {
        Pizza pizza;
        switch (teBestellenPizza) {
        case "barbeque":
            pizza = new BarbequePizzaGent();
            break;
        case "kaas":
            pizza = new KaasPizzaGent();
            break;
        default:
            pizza = new NonExistentPizza(teBestellenPizza);
        }
        return pizza;
    }

}
```

- `domein/PizzaStoreOostende` :

```
public class PizzaStoreOostende extends PizzaStore {

    @Override
    protected Pizza createPizza(String teBestellenPizza) {
        Pizza pizza;
        switch (teBestellenPizza) {
        case "barbeque":
            pizza = new BarbequePizzaOostende();
            break;
        case "kaas":
            pizza = new KaasPizzaOostende();
            break;
        default:
            pizza = new NonExistentPizza(teBestellenPizza);
        }
        return pizza;
    }

}
```

- Je merkt dat je nu alle soorten pizza's gaat moeten aanpassen naar een specifieke locatie zoals `BarbequePizza -> BarbequePizzaGent/BarbequePizzaOostende`
- Hetzelfde geld voor de `KaasPizza -> KaasPizzaGent/KaasPizzaOostende`
- Nu kunnen we dit al uittesten via `cui/PizzaApplicatie` :

```
import domein.Pizza;
import domein.PizzaStoreGent;
import domein.PizzaStoreOostende;

public class PizzaApplicatie {

    public PizzaApplicatie() {
        this.bestelPizzas();
    }

    private void bestelPizzas() {
        PizzaStoreGent pizzaStoreGent = new PizzaStoreGent();
```

```
        PizzaStoreOostende pizzaStoreOostende = new PizzaStoreOostende();

        Pizza gentPizza = pizzaStoreGent.bestelPizza("kaas");
        System.out.println();
        Pizza oostendePizza = pizzaStoreOostende.bestelPizza("kaas");
    }
}
```

- Voila we hebben onze pizza's afgewerkt en hebben daarmee het `Factory method` -pattern voltooid
- Nu gaan we dit uitbreiden via het `Abstract factory` -pattern waarbij elke PizzaStore dezelfde pizza's kan verkopen, maar met verschillende **toppics**
- We beginnen met het maken van een interface voor de `domein/PizzaIngredientFactory` :

```
public interface PizzaIngredientFactory {
    public Saus createSaus();

    public Korst createKorst();

    public Kaas createKaas();
}
```

- Nu maken we deze ingredienten ook echt aan door te beginnen met een **interface** van elke klasse:
- Dit ook voor `Kaas` en `Korst`

```
public interface Saus {
    public String toString();
}
```

- Nu maken we wat verschillende types voor onze `Saus` , `Korst` en `Saus` :
- `domein/SausLook`

```
public class SausLook implements Saus {
    @Override
    public String toString() {
        return "Looksaus";
    }
}
```

- `domein/SausTomaas` :

```
public class SausTomaat implements Saus {
    @Override
    public String toString() {
        return "Tomatensaus";
    }
}
```

- Doe nu hetzelfde voor `domein/KaasCheddar` , `KaasParmezaan` , `KorstDun` , `KorstDik`
- Nu kunnen we eindelijk beginnen aan onze **PizzaIngredientFactorys**
- `domein/PizzaIngredientFactoryGent` :

```
public class PizzaIngredientFactoryGent implements PizzaIngredientFactory {

    @Override
    public Saus createSaus() {
        return new SausTomaat();
```

```
    }

    @Override
    public Korst createKorst() {
        return new KorstDun();
    }

    @Override
    public Kaas createKaas() {
        return new KaasCheddar();
    }

}
```

- Hetzelfde voor `domein/PizzaIngredientFactoryOostende`:

```
public class PizzaIngredientFactoryOostende implements PizzaIngredientFactory {

    @Override
    public Saus createSaus() {
        return new SausLook();
    }

    @Override
    public Korst createKorst() {
        return new KorstDik();
    }

    @Override
    public Kaas createKaas() {
        return new KaasParmezaan();
    }

}
```

- Nu gaan we onze **Pizza**-klassen moeten aanpassen om met al deze fabrieken te werken:
- De methode prepare wordt vanaf nu abstract
- Ook geven we een **PizzaIngredientFactory** mee aan onze Pizza

```
public abstract class Pizza {
    private Kaas kaas;
    private Saus saus;
    private Korst korst;
    private PizzaIngredientFactory pizzaIngredientFactory;

    public Pizza(PizzaIngredientFactory pizzaIngredientFactory) {
        this.pizzaIngredientFactory = pizzaIngredientFactory;
    }
    // System.out.println("Preparing " + getClass().getSimpleName());

    public abstract void prepare();

    public void deliver() {
        System.out.println("Your " + getClass().getSimpleName() + " is ready");
    }

    protected Kaas getKaas() {
        return kaas;
```

```
    }

    protected void setKaas(Kaas kaas) {
        this.kaas = kaas;
    }

    protected Saus getSaus() {
        return saus;
    }

    protected void setSaus(Saus saus) {
        this.saus = saus;
    }

    protected Korst getKorst() {
        return korst;
    }

    protected void setKorst(Korst korst) {
        this.korst = korst;
    }

    protected PizzaIngredientFactory getPizzaIngredientFactory() {
        return pizzaIngredientFactory;
    }

    protected void setPizzaIngredientFactory(PizzaIngredientFactory
       pizzaIngredientFactory) {
        this.pizzaIngredientFactory = pizzaIngredientFactory;
    }

}
```

- Nu passen we onze subklassen dus ook aan:
- We vragen aan onze interface om iets te voor te bereiden, en dat wordt gedaan door onze factory's
- `domein/BarbequePizzaGent`

```
public class BarbequePizzaGent extends Pizza {

    public BarbequePizzaGent(PizzaIngredientFactory pizzaIngredientFactory) {
        super(pizzaIngredientFactory);
    }

    @Override
    public void prepare() {
        System.out.printf("Preparing %s", this.getClass().getSimpleName());
        this.setKaas(getPizzaIngredientFactory().createKaas());
        this.setKorst(getPizzaIngredientFactory().createKorst());
        this.setSaus(getPizzaIngredientFactory().createSaus());
    }

}
```

- `domein/KaasPizzaGent` :

```
public class KaasPizzaGent extends Pizza {

    public KaasPizzaGent(PizzaIngredientFactory pizzaIngredientFactory) {
```

```
        super(pizzaIngredientFactory);
    }

    @Override
    public void prepare() {
        System.out.printf("Preparing %s", this.getClass().getSimpleName());
        this.setKaas(getPizzaIngredientFactory().createKaas());
        this.setKorst(getPizzaIngredientFactory().createKorst());
        this.setSaus(getPizzaIngredientFactory().createSaus());
    }

}
```

- De klasse `domein/NonExistentPizza` moet ook ietsjes worden aangepast nu:

```
public class NonExistentPizza extends Pizza {
    private String opgegevenPizza;

    public NonExistentPizza(String opgegevenPizza, PizzaIngredientFactory
       ingredientFactory) {
        super(ingredientFactory);
        this.opgegevenPizza = opgegevenPizza;
    }

    @Override
    public void prepare() {
        System.out.println(opgegevenPizza + " does not exist in our store");
    }

    @Override
    public void deliver() {
        System.out.println(opgegevenPizza + " does not exist in our store");
    }
}
```

- Dit moet je exact hetzelfde doen voor `domein/BarbequePizzaOostende` en `KaasPizzaOostende`
- Nu kunnen we onze `PizzaStores` gaan bijwerken:
- `domein/PizzaStoreGent`

```
public class PizzaStoreGent extends PizzaStore {

    @Override
    protected Pizza createPizza(String teBestellenPizza) {
        Pizza pizza;
        PizzaIngredientFactory pizzaIngredientFactory = new
           PizzaIngredientFactoryGent();

        switch (teBestellenPizza.toLowerCase()) {
        case "barbeque":
            pizza = new BarbequePizzaGent(pizzaIngredientFactory);
            break;
        case "kaas":
            pizza = new KaasPizzaGent(pizzaIngredientFactory);
            break;
        default:
            pizza = new NonExistentPizza(teBestellenPizza,
               pizzaIngredientFactory);
```

```
        }
        return pizza;
    }

}
```

- `domein/PizzaStoreOostende`:

```
public class PizzaStoreOostende extends PizzaStore {

    @Override
    protected Pizza createPizza(String teBestellenPizza) {
        Pizza pizza;
        PizzaIngredientFactory pizzaIngredientFactory = new
           PizzaIngredientFactoryOostende();
        switch (teBestellenPizza) {
        case "barbeque":
            pizza = new BarbequePizzaOostende(pizzaIngredientFactory);
            break;
        case "kaas":
            pizza = new KaasPizzaOostende(pizzaIngredientFactory);
            break;
        default:
            pizza = new NonExistentPizza(teBestellenPizza,
               pizzaIngredientFactory);
        }
        return pizza;
    }

}
```

- Vanaf nu kunnen we dan effectief pizza's bestellen in `cui/PizzaApplicatie`:

```
package cui;

import domein.Pizza;
import domein.PizzaStoreGent;
import domein.PizzaStoreOostende;

public class PizzaApplicatie {

    public PizzaApplicatie() {
        this.bestelPizzas();
    }

    private void bestelPizzas() {
        PizzaStoreGent pizzaStoreGent = new PizzaStoreGent();
        PizzaStoreOostende pizzaStoreOostende = new PizzaStoreOostende();

        Pizza gentPizza = pizzaStoreGent.bestelPizza("kaas");
        toonPizzaDetails(gentPizza);

        System.out.println();

        Pizza oostendePizza = pizzaStoreOostende.bestelPizza("kaas");
        toonPizzaDetails(oostendePizza);
    }
```

```
    private void toonPizzaDetails(Pizza pizza) {
        System.out.println("Ingredienten van " + pizza.getClass().getSimpleName
            () + ":");
        System.out.println("Kaas: " + pizza.getKaas());
        System.out.println("Korst: " + pizza.getKorst());
        System.out.println("Saus: " + pizza.getSaus());
    }
}
```

f

1.3 Sjabloon - Pattern

1.3.1 Hoe herken je dit?

1.3.2 Theorie

Theorie - slides:

Notities uit de les:

1.3.3 UML

1.3.4 Implementatie

2 Java