Cursus Robbe Magerman Advanced Software Developement II

Robbe Magerman

08/06/2024

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
    - Geeft complete objecten weer en behoren allemaal tot dezelfde familie
    - Iedereen van de familie mag dus meedoen
    - **SimpleFactory**
  - 1 familie, compleet object, niet hele familie, keuze op voorhand
    - Geef complete objecten weer en behoren allemaal tot dezelfde familie maar er wordt maar een deel meegegeven
    - 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

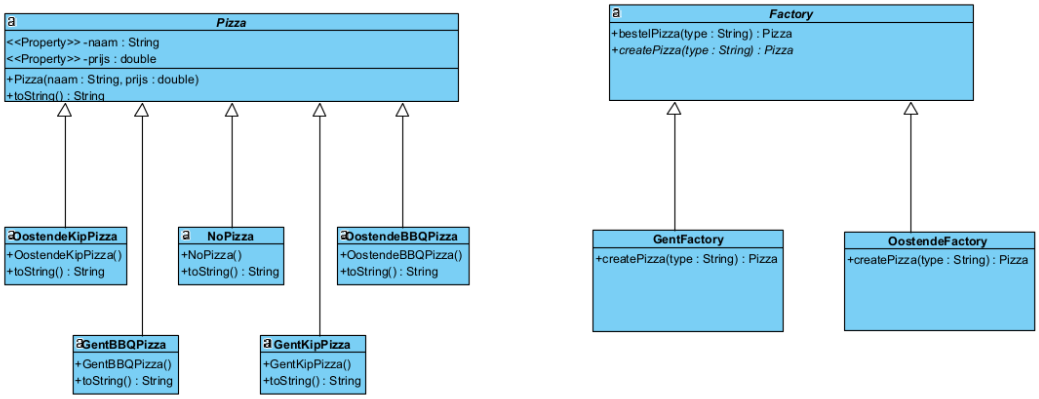

Figure 1: alt text

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:

```
package domein;

public abstract class Pizza {
    private String naam;
    private double prijs;

    public Pizza(String naam, double prijs) {
        this.naam = naam;
        this.prijs = prijs;
    }

    public String getNaam() {
        return naam;
    }

    public double getPrijs() {
        return prijs;
    }

    @Override
    public String toString() {
        return "Pizza{" + "naam='" + naam + '\'' + ", prijs=" + prijs + '}';
    }
}
```

- Nu we dat hebben kunnen we pizza's gaan maken voor onze verschillende factory's:

```
package domein;

public class OostendeKipPizza extends Pizza {

    public OostendeKipPizza() {
        super("Oostende Kip Pizza", 13.99);
    }
```

```
    @Override
    public String toString() {
        return "OostendeKipPizza{" + "naam='" + getNaam() + '\'' + ", prijs=" +
            getPrijs() + '}';
    }

}
```

```
package domein;

public class OostendeBBQPizza extends Pizza {

    public OostendeBBQPizza() {
        super("Oostende BBQ Pizza", 12.99);
    }

    @Override
    public String toString() {
        return "OostendeBBQPizza{" + "naam='" + getNaam() + '\'' + ", prijs=" +
            getPrijs() + '}';
    }

}
```

```
package domein;

public class GentKipPizza extends Pizza {

    public GentKipPizza() {
        super("Gent Kip Pizza", 11.99);
    }

    @Override
    public String toString() {
        return "GentKipPizza{" + "naam='" + getNaam() + '\'' + ", prijs=" +
            getPrijs() + '}';
    }

}
```

```
package domein;

public class GentBBQPizza extends Pizza {

    public GentBBQPizza() {
        super("Gent BBQ Pizza", 10.99);
    }

    @Override
    public String toString() {
        return "GentBBQPizza{" + "naam='" + getNaam() + '\'' + ", prijs=" +
            getPrijs() + '}';
    }

}
```

```
package domein;

public class NoPizza extends Pizza {

    public NoPizza() {
        super("Pizza bestaat niet", 0.0);
    }

    @Override
    public String toString() {
        return "NoPizza{" + "naam='" + getNaam() + '\'' + ", prijs=" + getPrijs
            () + '}';
    }

}
```

- We hebben onze producten, nu is het tijd om een abstract pizzafactory aan te maken.
- Het is abstract zodat we gemakkelijk nieuwe factory's kunnen toevoegen in de toekomst:

```
package domein.factorys;

import domein.Pizza;
import domein.*;

public abstract class Factory {

    public Pizza bestelPizza(String type) {
        Pizza pizza = createPizza(type);

        System.out.println("Bestelde pizza: " + pizza.getNaam());
        return pizza;
    }

    public abstract Pizza createPizza(String type);
}
```

- Waarvan nu enkele concrete implementaties:

```
package domein.factorys;

import domein.GentBBQPizza;
import domein.GentKipPizza;
import domein.NoPizza;
import domein.Pizza;
import domein.*;

public class GentFactory extends Factory {

    @Override
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        switch (type) {
        case "bbq":
            pizza = new GentBBQPizza();
            break;
        case "kip":
            pizza = new GentKipPizza();
```

```
            break;
        default:
            pizza = new NoPizza();
            break;
        }
        return pizza;
    }

    @Override
    public String toString() {
        return "GentFactory{}";
    }

}
```

```
package domein.factorys;

import domein.NoPizza;
import domein.OostendeBBQPizza;
import domein.OostendeKipPizza;
import domein.Pizza;
import domein.*;

public class OostendeFactory extends Factory {

    @Override
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        switch (type) {
        case "bbq":
            pizza = new OostendeBBQPizza();
            break;
        case "kip":
            pizza = new OostendeKipPizza();
            break;
        default:
            pizza = new NoPizza();
            break;
        }
        return pizza;
    }

    @Override
    public String toString() {
        return "OostendeFactory{}";
    }

}
```

- Laten we pizza's gaan bestellen:

```
package cui;

import domein.factorys.GentFactory;
import domein.factorys.OostendeFactory;

public class PizzaApplicatie {
```

```
    public static void main(String[] args) {
        GentFactory gentFactory = new GentFactory();
        OostendeFactory oostendeFactory = new OostendeFactory();

        gentFactory.bestelPizza("bbq");
        oostendeFactory.bestelPizza("kip");
    }
}
```

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**

1.2.3 UML

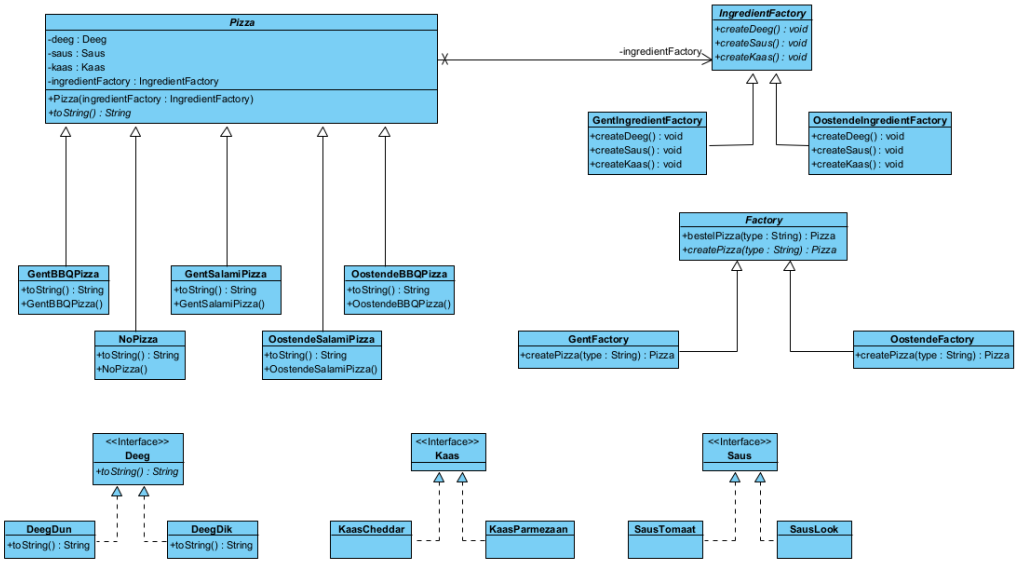

Figure 2: alt text

1.2.4 Implementatie

- We starten het het creeëren van interfaces van de attributen van onze pizza:

```
package domein.interfaces;

/**
 * Stap 1 van het Abstract Factory Pattern: maken van de gedragen hun
    interfaces
 */
public interface Deeg {
    @Override
    String toString();
}
```

```
package domein.interfaces;

/**
 * Stap 1 van het Abstract Factory Pattern: maken van de gedragen hun
    interfaces
 */
public interface Deeg {
    @Override
    String toString();
}
```

```
package domein.interfaces;

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

```java
package domein.interfaces;

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

- Daarna maken we concrete implementaties van onze interfaces:

```java
package domein.gedrag;

import domein.interfaces.Deeg;

/**
 * Stap 2 van het Abstract Factory Pattern: implementatie van de gedragen
 */
public class DeegDik implements Deeg {

    @Override
    public String toString() {
        return "Dik deeg";
    }

}
```

```java
package domein.gedrag;

import domein.interfaces.Deeg;

public class DeegDun implements Deeg {

    @Override
    public String toString() {
        return "Dun deeg";
    }
}
```

```java
package domein.gedrag;

import domein.interfaces.Kaas;

public class KaasCheddar implements Kaas {

    @Override
    public String toString() {
        return "Cheddar kaas";
    }

}
```

```java
package domein.gedrag;

import domein.interfaces.Kaas;

public class KaasParmezaan implements Kaas {

    @Override
```

```
    public String toString() {
        return "Parmezaanse kaas";
    }

}
```

```
package domein.gedrag;

import domein.interfaces.Saus;

public class SausLook implements Saus {

    @Override
    public String toString() {
        return "Looksaus";
    }

}
```

```
package domein.gedrag;

import domein.interfaces.Saus;

public class SausTomaat implements Saus {

    @Override
    public String toString() {
        return "Tomatensaus";
    }

}
```

- We hebben al onze ingredienten waardoor we kunnen beginnen met onze abstracte ingredienfactory:

```
package domein.ingredientfactory;

import domein.interfaces.Deeg;
import domein.interfaces.Kaas;
import domein.interfaces.Saus;

/**
 * Stap 3 van het Abstract Factory Pattern: maken van de abstracte klasse die
    de
 * methodes definieert voor het maken van de verschillende ingredienten
 */
public abstract class PizzaIngredientFactory {

    public abstract Deeg maakDeeg();

    public abstract Saus maakSaus();

    public abstract Kaas maakKaas();
}
```

- Die wederom hier weer concrete implementaties heeft volgens hun concrete factory (volgt later):

```
package domein.ingredientfactory;
```

```java
import domein.gedrag.DeegDik;
import domein.gedrag.KaasParmezaan;
import domein.gedrag.SausTomaat;
import domein.interfaces.Deeg;
import domein.interfaces.Kaas;
import domein.interfaces.Saus;

/**
 * Stap 4 van het Abstract Factory Pattern: implementatie van de concrete
 * ingredient factory
 */
public class GentIngredientFactory extends PizzaIngredientFactory {

    @Override
    public Deeg maakDeeg() {
        return new DeegDik();
    }

    @Override
    public Saus maakSaus() {
        return new SausTomaat();
    }

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

}
```

```java
package domein.ingredientfactory;

import domein.gedrag.DeegDun;
import domein.gedrag.KaasParmezaan;
import domein.gedrag.SausLook;
import domein.interfaces.Deeg;
import domein.interfaces.Kaas;
import domein.interfaces.Saus;

public class OostendeIngredientFactory extends PizzaIngredientFactory {

    @Override
    public Deeg maakDeeg() {
        return new DeegDun();
    }

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

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

```
}
```

- Nu onze setup in orde is, kunnen we eindelijk beginnen aan onze pizza's.
- Eerst beginnen we met de abstracte pizza:

```
package domein.pizza;

import domein.ingredientfactory.PizzaIngredientFactory;
import domein.interfaces.Deeg;
import domein.interfaces.Kaas;
import domein.interfaces.Saus;
import lombok.AccessLevel;
import lombok.Getter;
import lombok.Setter;

/**
 * Stap 5: Maken van de concrete pizza klassen
 */
@Getter(value = AccessLevel.PROTECTED)
@Setter(value = AccessLevel.PROTECTED)
public abstract class Pizza {
    private Deeg deeg;
    private Saus saus;
    private Kaas kaas;
    private PizzaIngredientFactory pizzaIngredientFactory;

    public Pizza(PizzaIngredientFactory pizzaIngredientFactory) {
        this.pizzaIngredientFactory = pizzaIngredientFactory;
        this.deeg = pizzaIngredientFactory.maakDeeg();
        this.saus = pizzaIngredientFactory.maakSaus();
        this.kaas = pizzaIngredientFactory.maakKaas();
    }

    public abstract String toString();

}
```

- Waarvan we nu concrete implementaties gaan maken:

```
package domein.pizza;

import domein.ingredientfactory.PizzaIngredientFactory;

public class OostendeBBQPizza extends Pizza {

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

    @Override
    public String toString() {
        return "Oostende BBQ Pizza met " + getDeeg() + ", " + getSaus() + " en
            " + getKaas();
    }

}
```

```java
package domein.pizza;

import domein.ingredientfactory.PizzaIngredientFactory;

public class OostendeSalamiPizza extends Pizza {

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

    @Override
    public String toString() {
        return "Oostende Salami Pizza met " + getDeeg() + ", " + getSaus() + "
            en " + getKaas();
    }

}
```

```java
package domein.pizza;

import domein.ingredientfactory.PizzaIngredientFactory;

public class GentBBQPizza extends Pizza {

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

    @Override
    public String toString() {
        return "Gent BBQ Pizza met " + getDeeg() + ", " + getSaus() + " en " +
            getKaas();
    }

}
```

```java
package domein.pizza;

import domein.ingredientfactory.PizzaIngredientFactory;

public class GentSalamiPizza extends Pizza {

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

    @Override
    public String toString() {
        return "Gent Salami Pizza met " + getDeeg() + ", " + getSaus() + " en "
             + getKaas();
    }

}
```

```java
package domein.pizza;
```

```
import domein.ingredientfactory.PizzaIngredientFactory;

public class NoPizza extends Pizza {

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

    @Override
    public String toString() {
        return "We don't have that pizza!";
    }

}
```

- Nu alles in verband met onze productie klaar is, kunnen we beginnen met de factorys:

```
package domein.factory;

import domein.pizza.Pizza;

/**
 * Stap 5: Maken van de concrete pizza klassen en de factorys om te bestellen
 */
public abstract class PizzaFactory {
    protected abstract Pizza createPizza(String item);

    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);

        System.out.println("--- Making a " + pizza.getClass().getSimpleName());
        return pizza;
    }
}
```

```
package domein.factory;

import domein.ingredientfactory.OostendeIngredientFactory;
import domein.pizza.NoPizza;
import domein.pizza.OostendeBBQPizza;
import domein.pizza.OostendeSalamiPizza;
import domein.pizza.Pizza;

public class OostendePizzaFactory extends PizzaFactory {

    @Override
    protected Pizza createPizza(String item) {
        Pizza pizza = null;

        switch (item.toLowerCase()) {
        case "bbq":
            pizza = new OostendeBBQPizza(new OostendeIngredientFactory());
            break;
        case "salami":
            pizza = new OostendeSalamiPizza(new OostendeIngredientFactory());
            break;
        default:
```

```
            pizza = new NoPizza(new OostendeIngredientFactory());
            break;
        }

        return pizza;
    }

}
```

```
package domein.factory;

import domein.ingredientfactory.GentIngredientFactory;
import domein.pizza.GentBBQPizza;
import domein.pizza.NoPizza;
import domein.pizza.Pizza;

public class GentPizzaFactory extends PizzaFactory {

    @Override
    protected Pizza createPizza(String item) {
        Pizza pizza = null;

        switch (item.toLowerCase()) {
        case "bbq":
            pizza = new GentBBQPizza(new GentIngredientFactory());
            break;
        case "salami":
            pizza = new GentBBQPizza(new GentIngredientFactory());
            break;
        default:
            pizza = new NoPizza(new GentIngredientFactory());
            break;
        }

        return pizza;
    }

}
```

- Laten we pizza's gaan bestellen:

```
package cui;

import domein.factory.GentPizzaFactory;
import domein.factory.OostendePizzaFactory;
import domein.factory.PizzaFactory;
import domein.pizza.Pizza;

public class PizzaApplicatie {
    public static void main(String[] args) {
        PizzaFactory oostendeFactory = new OostendePizzaFactory();
        Pizza oostendePizza = oostendeFactory.orderPizza("bbq");
        System.out.println(oostendePizza);

        PizzaFactory gentFactory = new GentPizzaFactory();
        Pizza gentPizza = gentFactory.orderPizza("salami");
        System.out.println(gentPizza);
```

```
    }
}
```

1.3 Builder

1.3.1 Hoe herken je dit?

- Je wil objecten maken waarin de volgorde uitmaakt

1.3.2 Theorie

Theorie - slides:

- Gebruik het Builder pattern om de constructie van een product af te schermen en zorg dat je het in stappen kan construeren
- Kent het proces om een sandwich te maken, ongeacht het type sandwich. Dit laat hij over aan de builder

Notities uit de les:

- Dit gebruik je als je een specifieke volgorde in het bouwen van een bepaald iets wil afdwingen
- We maken een specifieke klasse voor deze volgorde af te dwingen waardoor niemand anders van buitenaf hier rekening mee moet houden
- Elke stap in deze klasse is een aparte methode
- Director zegt aan de builder "voer de stappen in deze volgorde uit"
- Een variant hier op is bv. via het bouwen van een rechthoek. Deze gebruikt een innerclass Builder. De hoofdklasse (rechthoek) heeft dan een **private constructor**

1.3.3 UML

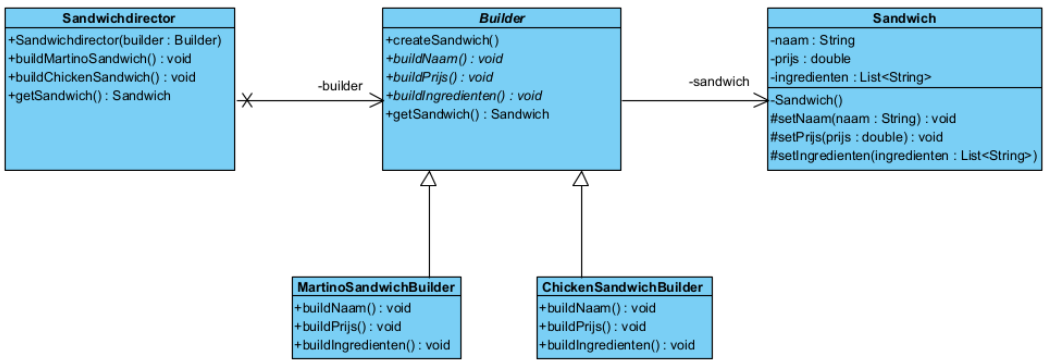

Figure 3: alt text

1.3.4 Implementatie

- We starten met het maken van de domeinklasse waar de builder voor zal dienen:
- Er is no arguments constructor met protected setters (NOOIT PUBLIEKE)

```
package domein;

import java.util.List;

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

/**
 * Stap 1: het maken van de domeinklasse dat gebuilt moet worden in een
 * specifieke volgorde
```

```
 */
@Setter(value = lombok.AccessLevel.PROTECTED)
@Getter
@NoArgsConstructor
public class Sandwitch {
    private String naam;
    private double prijs;
    private List<String> ingredienten;

    @Override
    public String toString() {
        return "Sandwitch{" + "naam='" + naam + '\'' + ", prijs=" + prijs + ",
            ingredienten=" + ingredienten + '}';
    }
}
```

- Hier maken we effectief de **abstracte** Builder wat enkele abstracte methoden heeft zodat nieuwe soorten SandwichBuilder gemakkelijk kunnen aangemaakt met elk hun eigen invullen van de methoden
- Vanaf hieruit is het de bedoeling dat we verschillende soorten Builders maken via deze

```
package domein;

import lombok.Getter;

public abstract class Builder {

    @Getter
    private Sandwitch sandwitch;

    public void createSandwitch() {
        sandwitch = new Sandwitch();
    }

    public abstract void buildNaam();

    public abstract void buildPrijs();

    public abstract void buildIngredienten();
}
```

- Laten we een **Broodje Martino** maken:

```
package domein;

import java.util.List;

/**
 * Stap 3: het maken van de specifieke builder die de domeinklasse in een
 * specifieke volgorde bouwt
 */
public class MartinoSandwitchBuilder extends Builder {

    @Override
    public void buildNaam() {
        getSandwitch().setNaam("Martino");
    }

    @Override
```

```
    public void buildPrijs() {
        getSandwitch().setPrijs(3.5);
    }

    @Override
    public void buildIngredienten() {
        getSandwitch().setIngredienten(List.of("Kip", "Mayonaise", "Pikante
            saus"));
    }

}
```

- De Builder doet vanaf nu zijn ding, maar we mogen deze Builder niet rechtstreeks oproepen in onze code
- Hiervoor is een **Director** nodig zodat hij elke Builder dan ook kan oproepen
- Dit heeft ook een **getBroodje**-methode
- Stel dat je nu nog een `ChickenSandwichBuilder` maakt kan je die heel gemakkelijk implementeren in de methode

```
package domein;

/**
 * Stap 4: het maken van de director die eenderwelke builder aanstuurt
 */
public class SandwichDirector {
    private Builder builder;

    public SandwichDirector(Builder builder) {
        this.builder = builder;
    }

    public void createSandwitch() {
        builder.createSandwitch();
        builder.buildNaam();
        builder.buildPrijs();
        builder.buildIngredienten();
    }

    public Sandwitch getSandwich() {
        return builder.getSandwitch();
    }
}
```

- En voila, we kunnen broodjes bestellen via onze applicatie:

```
package cui;

import domein.MartinoSandwitchBuilder;
import domein.SandwichDirector;
import domein.Sandwitch;

public class SandwichApplicatie {
    public static void main(String[] args) {
        domein.Builder builder = new MartinoSandwitchBuilder();
        SandwichDirector director = new SandwichDirector(builder);
        director.createSandwitch();
        Sandwitch sandwitch = director.getSandwich();
        System.out.println(sandwitch);
```

```
        domein.Builder chickenBuilder = new domein.ChickenSandwichBuilder();
        SandwichDirector chickenDirector = new SandwichDirector(chickenBuilder)
            ;
        chickenDirector.createSandwitch();
        Sandwitch chickenSandwitch = chickenDirector.getSandwich();
        System.out.println(chickenSandwitch);
    }
}
```

1.4 BuilderVariant

1.4.1 Hoe kerken je dit?

- Er is ook nog de **BuilderVariant**. Het grote verschil met deze Builder is dat hier de volgorde niet van belang is.
- Hier ga ik wat vlugger over gezien het praktisch gezien heel erg hetzelfde.

1.4.2 UML

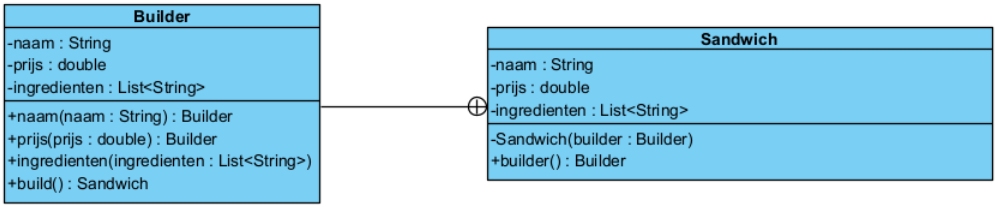

Figure 4: alt text

1.4.3 Implementatie

- De BuilderVariant staat altijd in de domeinklasse zelf van het object dat kan aangemaakt worden.
- De attributen van de klasse `Song` zijn **final** aangezien de Builder deze gaat instellen voor ons
- Er is ook een **private constructor** want objecten zullen niet worden gemaakt op deze manier.
- Je kan ook wat validatie toevoegen via enums die onder deze klasse zullen staan

```
package domein;

import java.time.LocalDate;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import domein.enums.SongType;
import domein.enums.ValidationError;

public class Song {
    private final String name;
    private final List<String> artist;
    private final SongType songType;
    private final LocalDate releasedAt;

    private Song(Builder builder) {
        this.name = builder.name;
        this.artist = builder.artist;
        this.songType = builder.songType;
```

```
        this.releasedAt = builder.releasedAt;
    }

    public static Builder builder() {
        return new Builder();
    }

    @Override
    public String toString() {
        return "Song{" + "name='" + name + '\'' + ", artist=" + artist + ",
           songType=" + songType + ", releasedAt="
                + releasedAt + '}';
    }

    public static class Builder {
        private String name;
        private List<String> artist;
        private SongType songType;
        private LocalDate releasedAt;

        public Builder name(String name) {
            this.name = name;
            return this;
        }

        public Builder artist(List<String> artist) {
            this.artist = artist;
            return this;
        }

        public Builder songType(SongType songType) {
            this.songType = songType;
            return this;
        }

        public Builder releasedAt(LocalDate releasedAt) {
            this.releasedAt = releasedAt;
            return this;
        }

        public Song build() {
            Set<ValidationError> errors = new HashSet<>();

            if (name == null || name.isBlank()) {
                errors.add(ValidationError.MISSING_NAME);
            }
            if (artist == null || artist.isEmpty()) {
                errors.add(ValidationError.MISSING_ARTIST);
            }

            if (songType == null) {
                errors.add(ValidationError.INVALID_SONG_TYPE);
            }

            if (!errors.isEmpty()) {
                throw new IllegalStateException("Kan geen Song maken,
                   validatiefouten: " + errors);
            }
```

```
            return new Song(this);
        }
    }
}
```

- Enums:

```
package domein.enums;

public enum SongType {
    NORMAL, LIVEEDIT, REMIX, SET
}
```

```
package domein.enums;

public enum ValidationError {
    MISSING_NAME, MISSING_ARTIST, MISSING_RELEASE_DATE, INVALID_SONG_TYPE
}
```

- Nadien kunnen we heel gemakkelijk welk object dan ook maken in welke volgorde dan ook

```
package cui;

import java.time.LocalDate;
import java.util.List;

import domein.Song;
import domein.enums.SongType;

public class SongApplicatie {
    public SongApplicatie() {
        makeSongs();
    }

    private void makeSongs() {
        Song promises = Song.builder().artist(List.of("Dual Damage")).songType(
            SongType.NORMAL)
                .releasedAt(LocalDate.now()).name("Promises").build();

        Song voices = Song.builder().artist(List.of("D-Sturb", "Da Tweekaz")).
            name("Voices").songType(SongType.NORMAL)
                .build();

        System.out.println(promises);
        System.out.println(voices);

    }
}
```

1.5 Singleton

1.5.1 Hoe herken je dit?

- Je wil een object aanmaken dat maar één instantie heeft zodat meerdere objecten steeds met dezelfde gegevens.

1.5.2 Theorie

Theorie - slides:

- Het Singleton Pattern garandeert dat een klasse slechts één instantie heeft en biedt een globaal toegangspunt ernaartoe.

Notities uit de les:

- Het heeft een private constructur
- Stappenplan:
  - Bouw een **private constructor** van een bepaalde klasse (Zanger)
  - Bouw een **public static** methode `getInstance() : Zanger`

1.5.3 UML

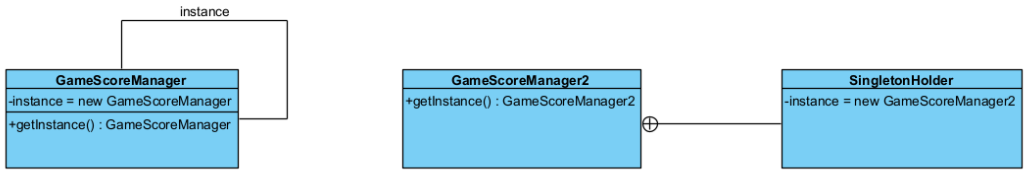

Figure 5: alt text

1.5.4 Implementatie

- De logica voor het bouwen gebeurt in de domeinklasse zelf en kan op 2 manieren gebeuren:
- De 1e manier is niet Multithreading vriendelijk:

```
package domein;

public class GameScoreSingleton {

    private final static GameScoreSingleton instance = new GameScoreSingleton()
        ;
    private int score;

    private GameScoreSingleton() {
        score = 0;
    }

    // 1e methode: Singleton zonder een klasse - slecht voor MultiThreading
    public static GameScoreSingleton getInstance() {
        if (instance == null) {
            return new GameScoreSingleton();
        }
        return instance;
    }

    public synchronized void addPoints(int points) {
        score += points;
    }

    public synchronized void resetScore() {
        score = 0;
    }

    public synchronized int getScore() {
```

```
        return score;
    }

}
```

- De 2e is wel goed voor Multithreading aangezien dit met een **Enum** zal werken wat in java standaard een Singleton is

```
package domein;

public class GameScoreSingleton {

    private int score;

    private GameScoreSingleton() {
        score = 0;
    }

    // 2e methode: wegens oplossing MultiThreading met lazy loading
    private static class SingletonHolder {
        private final static GameScoreSingleton INSTANCE = new
           GameScoreSingleton();
    }

    public static GameScoreSingleton getInstance() {
        if (Singleton.INSTANCE == null) {
            return new GameScoreSingleton();
        }
        return SingletonHolder.INSTANCE;
    }

    public synchronized void addPoints(int points) {
        score += points;
    }

    public synchronized void resetScore() {
        score = 0;
    }

    public synchronized int getScore() {
        return score;
    }

}
```

- Nu kunnen we zoveel initialisaties maken van onze GameScoreSingleton als we maar willen en het zal altijd dezelfde gegevens bevatten:

```
package cui;

import java.util.Scanner;

import domein.GameScoreSingleton;

public class GameScoreApplicatie {

    public GameScoreApplicatie() {
        Scanner scanner = new Scanner(System.in);
```

```
        GameScoreSingleton scoreManager = GameScoreSingleton.getInstance();
        GameScoreSingleton scoreManager2 = GameScoreSingleton.getInstance();

        System.out.println("Welkom bij het Game Score Systeem!");
        boolean running = true;

        while (running) {
            System.out.println("\nKies een optie:");
            System.out.println("1. Voeg punten toe");
            System.out.println("2. Bekijk huidige score");
            System.out.println("3. Reset score");
            System.out.println("4. Stoppen");

            System.out.print("> ");
            String input = scanner.nextLine();

            switch (input) {
            case "1":
                System.out.print("Aantal punten om toe te voegen: ");
                int points = Integer.parseInt(scanner.nextLine());
                scoreManager.addPoints(points);
                System.out.println(points + " punten toegevoegd!");
                break;
            case "2":
                System.out.println("Huidige score: " + scoreManager2.getScore()
                    );
                break;
            case "3":
                scoreManager.resetScore();
                System.out.println("Score gereset!");
                break;
            case "4":
                running = false;
                System.out.println("Tot de volgende keer!");
                break;
            default:
                System.out.println("Ongeldige optie, probeer opnieuw.");

        }

        scanner.close();
    }

}
```

1.6 Adapter

1.6.1 Hoe herken je dit?

- Je wilt dat 2 of meerdere objecten hetzelfde gedrag vertonen **via Interfaces**

1.6.2 Theorie

Theorie - slides:

- Als het loopt als een eend en kwaakt als een eend, moet/kan het een eend/kalkoen zijn, die in een eendenpak verpakt is ....

Notities uit de les:

- Je hebt een aantal objecten die bepaalde methoden gebruiken (Ducks)
- Opeens uit het niets krijg je een nieuwe jar-file binnen met nieuwe objecten (Turkeys), maar die niet dezelfde methoden hebben als de orginele
- Toch wil je dat de nieuwe objecten (Turkyes) ook dezelfde methoden hebben als de oude (Ducks)
- Dan maken we een **ObjectAdapter**(Turkey) wat de interface bevat van de oude objecten (Ducks)
- Op deze manier kunnen we nieuwe objecten hetzelfde gedrag geven als oude en kan je applicatie weer doorlopen als normaal

1.6.3 UML

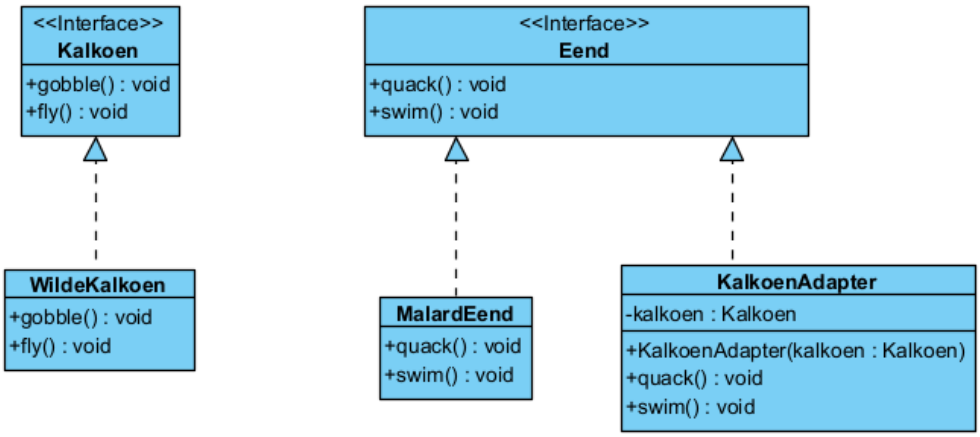

Figure 6: alt text

1.6.4 Implementatie

- We beginnen met het maken van de interfaces

```
package domein;

public interface Eend {
    void quack();

    void swim();
}
```

```
package domein;

public interface Kalkoen {
    void gobble();

    void fly();
}
```

- Nu maken we onze domeinklassen die de interfaces implementeren:

```
package domein;

public class MalardEend implements Eend {

    @Override
    public void quack() {
        System.out.println("Quack quack");
```

```
    }

    @Override
    public void swim() {
        System.out.println("Swimming in a lake");
    }

}
```

```
package domein;

public class WildeKalkoen implements Kalkoen {

    @Override
    public void gobble() {
        System.out.println("gobble gobble");
    }

    @Override
    public void fly() {
        System.out.println("flying a short distance");
    }

}
```

- Nu implementeren we onze adapter zodat een **Kalkoen** ook een **Eend** kan zijn:

```
package domein;

public class KalkoenAdapter implements Eend {

    private Kalkoen kalkoen;

    public KalkoenAdapter(Kalkoen kalkoen) {
        this.kalkoen = kalkoen;
    }

    @Override
    public void quack() {
        kalkoen.gobble();
    }

    @Override
    public void swim() {
        kalkoen.fly();
    }

}
```

- Nu kunnen we een Kalkoen zich laten gedragen als een Eend dankzij de adapter

```
public class KalkoenApplicatie {
    public static void main(String[] args) {
        WildeKalkoen kalkoen = new WildeKalkoen();
        Eend kalkoenAdapter = new KalkoenAdapter(kalkoen);

        System.out.println("De Kalkoen zegt...");
```

```
        kalkoen.gok();
        kalkoen.vliegKorteAfstand();

        System.out.println("\nDe KalkoenAdapter zegt (als Eend)...");
        kalkoenAdapter.kwak();
        kalkoenAdapter.vlieg();
    }
}
```

1.7 Iterator (Eigen Iterator)

1.7.1 Hoe herken je dit?

- Je wil de objecten overlopen

1.7.2 Theorie

Theorie - slides:

Notities uit de les:

- Je kan kiezen tussen twee systemen: de ingebouwde en één die je zelf opbouwt
- Wat je sowieso nodig hebt is een **interface**:
  - next : Object (in de oefeningen is dit meestal een abstracte file)
  - hasNext : boolean
- Je hebt ook een NullIterator die doet alsof hij een CompositeIterator is (AKA AdapterPattern)
- Bijna elke implementatie heeft een ingebakken Adapter

1.7.3 UML

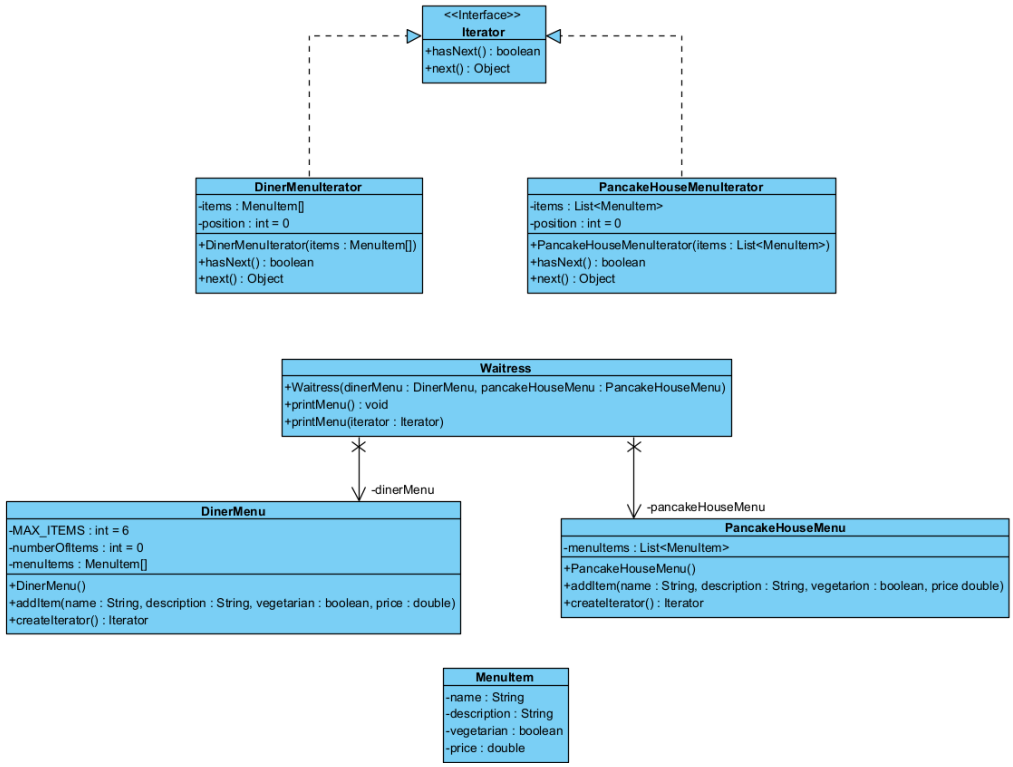

Figure 7: alt text

1.7.4 Implementatie

- Eerst moeten we onze eigen Iterator-interface aanmaken

```
package domein;

/**
 * Stap 1: maak een interface Iterator
 */
public interface Iterator {
    boolean hasNext();

    Object next();
}
```

- Gevolgd door de domeinklasse
- De eerste domeinklasse zal elk `Menu` gebruiken om hun kaart mee aan te maken:

```
package domein;

import lombok.AllArgsConstructor;
import lombok.Getter;

@AllArgsConstructor
@Getter
public class MenuItem {
    private String name;
    private String description;
    private boolean vegetarian;
    private double price;
}
```

- Nu de expliciete menu's zelf aanmaken:
- Puur ter demonstratie zal deze klasse werken met een `normale array` en de andere met een `ArrayList`

```
public class DinerMenu {
    private static final int MAX_ITEMS = 6;
    private int numberOfItems = 0;
    private MenuItem[] menuItems;

    public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];
        addItem("Vegetarische Lasagne", "Een heerlijke vegetarische lasagne met
            veel groenten", true, 8.99);
        addItem("Steak", "Een sappige steak met kruidenboter", false, 14.99);
        addItem("Vis van de dag", "Dagverse vis met een citroenboter saus",
           false, 12.99);
        addItem("Vegetarische Burger", "Een burger gemaakt van bonen en
           groenten", true, 9.99);
    }

    public void addItem(String name, String description, boolean vegetarian,
       double price) {

        if (numberOfItems >= MAX_ITEMS) {
            System.err.println("Menu is vol, kan geen item toevoegen.");
            return;
        }
```

```
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems[numberOfItems++] = menuItem;
    }

    public Iterator createIterator() {
        return new DinerMenuIterator(menuItems);
    }
}
```

```
package domein;

import java.util.ArrayList;

public class PancakeHouseMenu {
    private ArrayList<MenuItem> menu;

    public PancakeHouseMenu() {
        menu = new ArrayList<>();
        addItem("Kleine Pannenkoek", "Een kleine pannenkoek met een beetje
           stroop", true, 1.99);
        addItem("Grote Pannenkoek", "Een grote pannenkoek met veel stroop",
           false, 2.99);
        addItem("Pannenkoek met spek", "Een pannenkoek met spek en stroop",
           false, 3.49);
    }

    private void addItem(String name, String description, boolean vegetarian,
       double price) {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menu.add(menuItem);
    }

    public Iterator createIterator() {
        return new PancakeHouseMenuIterator(menu);
    }
}
```

- Hier bouwen we effectief onze Iterator en implenteren de logica er ook in:

```
package domein;

public class DinerMenuIterator implements Iterator {

    private MenuItem[] items;
    private int position = 0;

    public DinerMenuIterator(MenuItem[] items) {
        this.items = items;
    }

    @Override
    public boolean hasNext() {
        return (position < items.length && items[position] != null);
    }

    @Override
    public Object next() {
```

```
        return items[position++];
    }

}
```

```
package domein;

import java.util.ArrayList;

public class PancakeHouseMenuIterator implements Iterator {

    private ArrayList<MenuItem> menu;
    int position = 0;

    public PancakeHouseMenuIterator(ArrayList<MenuItem> menu) {
        this.menu = menu;
    }

    @Override
    public boolean hasNext() {
        return position < menu.size() && menu.get(position) != null;
    }

    @Override
    public Object next() {
        return menu.get(position++);
    }
}
```

- Uiteindelijk moeten we nog een klasse maken die de Iterators zal aanpsreken
- Dit is een restaurant dus we laten een Waitress het werk doen:

```
package domein;

public class Waitress {
    private PancakeHouseMenu pancakeHouseMenu;
    private DinerMenu dinerMenu;

    public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();

        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);

        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }

    public void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem) iterator.next();
```

```
            System.out.println(menuItem.getName() + ", " + menuItem.getPrice()
               + " -- " + menuItem.getDescription());
        }
    }
}
```

- Nu dit klaar is kunnen we de applicatie uittesten:

```
package cui;

import domein.DinerMenu;
import domein.PancakeHouseMenu;
import domein.Waitress;

public class IteratorApplicatie {
    public static void main(String[] args) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();

        domein.Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu);

        waitress.printMenu(dinerMenu.createIterator());
        System.out.println();
        waitress.printMenu(pancakeHouseMenu.createIterator());
    }
}
```

1.8 Iterator (Java Iterator)

1.8.1 Hoe herken je dit?

- Je wil de objecten overlopen

1.8.2 Theorie

Theorie - slides:

Notities uit de les:

- Je kan kiezen tussen twee systemen: de ingebouwde en één die je zelf opbouwt
- Je hebt ook een NullIterator die doet alsof hij een CompositeIterator is (AKA AdapterPattern)
- Bijna elke implementatie heeft een ingebakken Adapter

1.8.3 UML

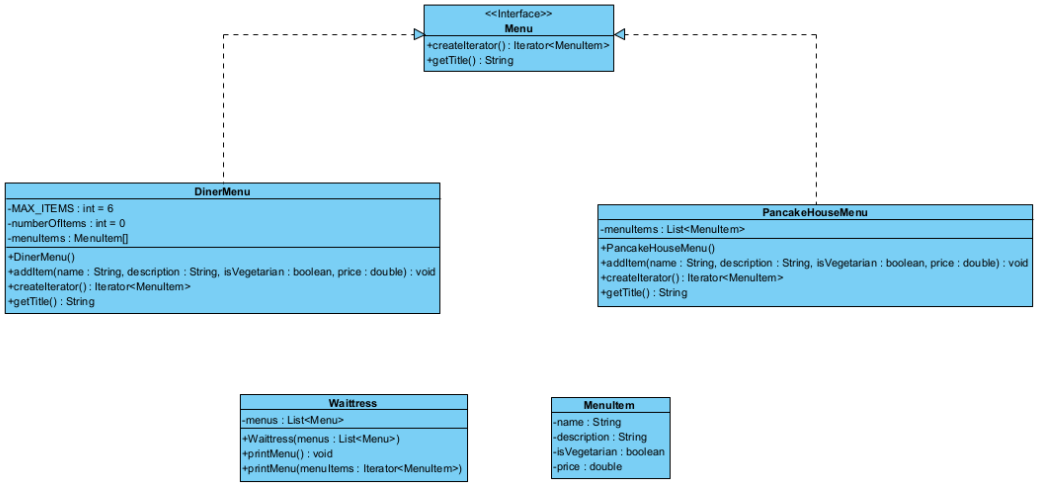

Figure 8: alt text

1.8.4 Implementatie

- Eerst maken we een Menu interface aan:

```
package domein;

import java.util.Iterator;

public interface Menu {
    Iterator<MenuItem> createIterator();

    String getTitle();
}
```

- Gevolgd door de domeinklassen
- De eerste domeinklasse zal elk `Menu` gebruiken om hun kaart mee aan te maken:

```
package domein;

import lombok.AllArgsConstructor;
import lombok.Getter;

@AllArgsConstructor
@Getter
public class MenuItem {
    private String name;
    private String description;
    private boolean vegetarian;
    private double price;
}
```

- Nu de expliciete menu's zelf aanmaken:
- Puur ter demonstratie zal deze klasse werken met een `normale array` en de andere met een `ArrayList`

```
package domein;

import java.util.Arrays;
```

```
import java.util.Iterator;

public class DinerMenu implements Menu {
    private static final int MAX_ITEMS = 6;
    private int numberOfItems = 0;
    private MenuItem[] menuItems;

    public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];
        addItem("Vegetarische Lasagne", "Een heerlijke vegetarische lasagne met
            veel groenten", true, 8.99);
        addItem("Steak", "Een sappige steak met kruidenboter", false, 14.99);
        addItem("Vis van de dag", "Dagverse vis met een citroenboter saus",
           false, 12.99);
        addItem("Vegetarische Burger", "Een burger gemaakt van bonen en
           groenten", true, 9.99);
    }

    public void addItem(String name, String description, boolean vegetarian,
       double price) {

        if (numberOfItems >= MAX_ITEMS) {
            System.err.println("Menu is vol, kan geen item toevoegen.");
            return;
        }
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems[numberOfItems++] = menuItem;
    }

    public Iterator<MenuItem> createIterator() {
        return Arrays.asList(menuItems).iterator();
    }

    @Override
    public String getTitle() {
        return this.getClass().getSimpleName();
    }
}
```

```
package domein;

import java.util.ArrayList;
import java.util.Iterator;

public class PancakeHouseMenu implements Menu {
    private ArrayList<MenuItem> menu;

    public PancakeHouseMenu() {
        menu = new ArrayList<>();
        addItem("Kleine Pannenkoek", "Een kleine pannenkoek met een beetje
           stroop", true, 1.99);
        addItem("Grote Pannenkoek", "Een grote pannenkoek met veel stroop",
           false, 2.99);
        addItem("Pannenkoek met spek", "Een pannenkoek met spek en stroop",
           false, 3.49);
    }
```

```
    private void addItem(String name, String description, boolean vegetarian,
       double price) {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menu.add(menuItem);
    }

    public Iterator<MenuItem> createIterator() {
        return menu.iterator();
    }

    @Override
    public String getTitle() {
        return this.getClass().getSimpleName();
    }
}
```

```
package domein;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class CafeMenu implements Menu {

    private Map<String, MenuItem> menuItems = new HashMap<>();

    public CafeMenu() {
        addItem("Koffie", "Heerlijke verse koffie", true, 2.50);
        addItem("Thee", "Verfrissende thee", false, 2.00);
        addItem("Broodje Gezond", "Gezond broodje met kaas en ham", true, 3.50)
           ;
    }

    private void addItem(String name, String description, boolean isVegetarian,
        double price) {
        MenuItem menuItem = new MenuItem(name, description, isVegetarian, price
           );
        menuItems.put(name, menuItem);
    }

    @Override
    public Iterator<MenuItem> createIterator() {
        return menuItems.values().iterator();
    }

    @Override
    public String getTitle() {
        return this.getClass().getSimpleName();
    }

}
```

- Uiteindelijk moeten we nog een klasse maken die de Iterators zal aanpsreken
- Dit is een restaurant dus we laten een Waitress het werk doen:

```
package domein;
```

```
import java.util.Iterator;
import java.util.List;

public class Waitress {

    private List<Menu> menus;

    public Waitress(List<Menu> menus) {
        this.menus = menus;
    }

    public void printMenu() {
        for (Menu menu : menus) {
            System.out.println("\n" + menu.getTitle());
            System.out.println("---------------------");

            Iterator<MenuItem> menuIterator = menu.createIterator();
            printMenu(menuIterator);
        }
    }

    public void printMenu(Iterator<MenuItem> iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.println(menuItem.getName() + ", " + menuItem.getPrice()
               + " -- " + menuItem.getDescription());
        }
    }
}
```

- Nu dit klaar is kunnen we de applicatie uittesten:

```
package cui;

import java.util.Arrays;
import java.util.List;

import domein.CafeMenu;
import domein.DinerMenu;
import domein.Menu;
import domein.PancakeHouseMenu;
import domein.Waitress;

public class IteratorApplicatie {
    public static void main(String[] args) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        CafeMenu cafeMenu = new CafeMenu();

        List<Menu> menus = Arrays.asList(pancakeHouseMenu, dinerMenu, cafeMenu)
           ;

        Waitress waitress = new Waitress(menus);

        waitress.printMenu();
    }
}
```

1.9 Composite

1.9.1 Hoe herken je dit?

- Een structuur waaruit je uiteindelijk via een boom-achtige structuur gaat uitkomen
- We stellen iets boven elkaar zodat het niet uit maakt waar in de boom dat we zitten (hetzij, een leaf/item . . .)

1.9.2 Theorie

Theorie - slides:

- Het Composite Pattern stelt je in staat om objecten in boomstructuren samen te stellen om partwhole hiërarchiën weer te geven. Composite laat clients de afzonderlijke objecten of samengestelde objecten op uniforme wijze behandelen.

Notities uit de les:

- Hier wordt gebruik gemaakt van een abstracte klasse
- **Examenvraag**, waarom steek je alles in een abstracte klasse:
  - vanuit de clients moet hij deze klasse op dezelfde manier aanspreken
- Je moet kijken naar beide klassen en kijken welke methodes in beide voorkomen en die abstract maken in de nieuwe abstracte klasse
- 3 Stappen:
  - 1 Attributen die in een boomstructuur zitten
  - 2 Methoden die niet overal zijn behandelen met een exceptie in de hoofdklasse
  - 3 Methodes die overal zijn abstract

1.9.3 UML

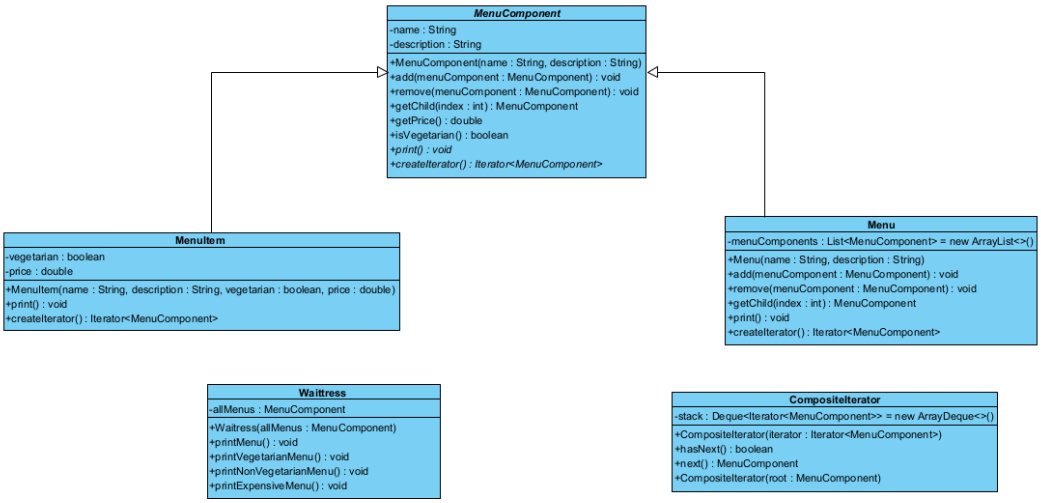

Figure 9: alt text

1.9.4 Implementatie

- We starten met het creeeren van een abstracte klasse dat dient als de 'root' van al onze subklassen/'kinderen'

```
package domein;

import java.util.Iterator;

import lombok.AllArgsConstructor;
import lombok.Getter;

@Getter
```

```
@AllArgsConstructor
public abstract class MenuComponent {
    private String name;
    private String description;

    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }

    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }

    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }

    public double getPrice() {
        throw new UnsupportedOperationException();
    }

    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
    }

    public abstract void print();

    public abstract Iterator<MenuComponent> createIterator();

}
```

- Daarna de implementaties van deze klasse:

```
package domein;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Menu extends MenuComponent {

    private List<MenuComponent> menuComponents = new ArrayList<>();

    public Menu(String name, String description) {
        super(name, description);
    }

    @Override
    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    @Override
    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    @Override
```

```
    public MenuComponent getChild(int i) {
        return menuComponents.get(i);
    }

    @Override
    public void print() {
        System.out.println("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");

        menuComponents.forEach(MenuComponent::print);
    }

    @Override
    public Iterator<MenuComponent> createIterator() {
        return menuComponents.iterator();
    }

}
```

```
package domein;

import java.util.Collections;
import java.util.Iterator;

import lombok.Getter;

@Getter
public class MenuItem extends MenuComponent {
    private boolean vegetarian;
    private double price;

    public MenuItem(String name, String description, boolean vegetarian, double
        price) {
        super(name, description);
        this.vegetarian = vegetarian;
        this.price = price;
    }

    @Override
    public void print() {
        System.out.print(" " + getName());
        if (isVegetarian()) {
            System.out.print("(vegetarian)");
        }
        System.out.println(", " + getPrice());
        System.out.println("     -- " + getDescription());
    }

    @Override
    public Iterator<MenuComponent> createIterator() {
        return Collections.emptyIterator();
    }

}
```

- Nu moeten we over deze 3 klassen itereren, daarom wordt gebruik gemaakt van een CompositeIterator:

```java
package domein;

import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Iterator;
import java.util.List;

public class CompositeIterator implements Iterator<MenuComponent> {
    Deque<Iterator<MenuComponent>> stack = new ArrayDeque<>();

    public CompositeIterator(Iterator<MenuComponent> iterator) {
        stack.push(iterator);
    }

    public CompositeIterator(MenuComponent root) {
        stack.push(List.of(root).iterator());
    }

    @Override
    public boolean hasNext() {
        if (stack.isEmpty()) {
            return false;
        }

        Iterator<MenuComponent> iterator = stack.peek();
        if (!iterator.hasNext()) {
            stack.pop(); // verwijder lege iterator
            return hasNext(); // controleer de volgende in de stack
        }
        return true; // er is een volgende component
    }

    @Override
    public MenuComponent next() {
        if (hasNext()) {
            Iterator<MenuComponent> iterator = stack.peek();
            MenuComponent component = iterator.next();
            System.out.println(component.getName());
            if (component instanceof Menu) {
                stack.push(component.createIterator());
            }
            return component;
        }
        return null; // geen volgende component
    }

}
```

- Uiteindelijk kunnen we weer aan de Waitress allerlei zaken vragen zoals het volledige menu geven (print):
- Maar dankzij de CompositeIterator kunnen we nu ook menu's op een specifieke manier weergeven:
- De Waitress krijgt attribuut `allMenus` omdat in de CUI-laag worden er verschillende menus aangemaakt en één speciale genaamd `allMenus`. Op dit speciale 'menu' voeg je dan elk menu toe dat je maar wil

```java
package domein;

import java.util.Iterator;
```

```
public class Waitress {
    private MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }

    public void printVegetarianMenu() {
        System.out.println("\nVEGETARIAN MENU\n----");
        Iterator<MenuComponent> iterator = new CompositeIterator(allMenus.
            createIterator());
        while (iterator.hasNext()) {
            MenuComponent menuComponent = iterator.next();
            try {
                if (menuComponent.isVegetarian()) {
                    menuComponent.print();
                }
            } catch (UnsupportedOperationException e) {
                // do nothing
            }
        }
    }

    public void printNonVegetarianMenu() {
        System.out.println("\nNONVEGETARIAN MENU\n----");
        Iterator<MenuComponent> iterator = new CompositeIterator(allMenus.
            createIterator());
        while (iterator.hasNext()) {
            MenuComponent menuComponent = iterator.next();
            try {
                if (!menuComponent.isVegetarian()) {
                    menuComponent.print();
                }
            } catch (UnsupportedOperationException e) {
                // do nothing
            }
        }
    }

    public void printExpensiveMenu() {
        System.out.println("\nEXPENSIVE MENU\n----");
        Iterator<MenuComponent> iterator = new CompositeIterator(allMenus.
            createIterator());
        while (iterator.hasNext()) {
            MenuComponent menuComponent = iterator.next();
            try {
                if (menuComponent.getPrice() >= 3.00) {
                    menuComponent.print();
                }
            } catch (UnsupportedOperationException e) {
                // do nothing
            }
        }
    }
```

```
}
```

- Uiteindelijk via deze simpele instelling kunnen we wederom weer menu's overlopen hoe we maar willen:

```
package cui;

import domein.Menu;
import domein.MenuComponent;
import domein.MenuItem;
import domein.Waitress;

public class CompositeApplicatie {

    public static void main(String[] args) {
        MenuComponent pancakeHouseMenu = new Menu("PANCAKE HOUSE MENU", "
            Breakfast");
        MenuComponent dinerMenu = new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu = new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu = new Menu("DESSERT MENU", "Dessert of course
            !");
        MenuComponent coffeeMenu = new Menu("COFFEE MENU", "Stuff to go with
            your afternoon coffee");

        MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        pancakeHouseMenu
                .add(new MenuItem("K&B's Pancake Breakfast", "Pancakes with
                    scrambled eggs, and toast", true, 2.99));
        pancakeHouseMenu
                .add(new MenuItem("Regular Pancake Breakfast", "Pancakes with
                    fried eggs, sausage", false, 2.99));
        pancakeHouseMenu.add(new MenuItem("Blueberry Pancakes",
                "Pancakes made with fresh blueberries, and blueberry syrup",
                    true, 3.49));
        pancakeHouseMenu
                .add(new MenuItem("Waffles", "Waffles, with your choice of
                    blueberries or strawberries", true, 3.59));

        dinerMenu
                .add(new MenuItem("Vegetarian BLT", "(Fakin') Bacon with
                    lettuce & tomato on whole wheat", true, 2.99));
        dinerMenu.add(new MenuItem("BLT", "Bacon with lettuce & tomato on whole
             wheat", false, 2.99));
        dinerMenu.add(new MenuItem("Soup of the day", "A bowl of the soup of
            the day, with a side of potato salad",
                false, 3.29));
        dinerMenu.add(
                new MenuItem("Hotdog", "A hot dog, with saurkraut, relish,
                    onions, topped with cheese", false, 3.05));
        dinerMenu.add(new MenuItem("Steamed Veggies and Brown Rice", "Steamed
            vegetables over brown rice", true, 3.99));

        dinerMenu.add(
                new MenuItem("Pasta", "Spaghetti with Marinara Sauce, and a
```

```
                    slice of sourdough bread", true, 3.89));

        dinerMenu.add(dessertMenu);

        dessertMenu.add(
                new MenuItem("Apple Pie", "Apple pie with a flakey crust,
                    topped with vanilla icecream", true, 1.59));

        dessertMenu.add(
                new MenuItem("Cheesecake", "Creamy New York cheesecake, with a
                    chocolate graham crust", true, 1.99));
        dessertMenu.add(new MenuItem("Sorbet", "A scoop of raspberry and a
            scoop of lime", true, 1.89));

        cafeMenu.add(new MenuItem("Veggie Burger and Air Fries",
                "Veggie burger on a whole wheat bun, lettuce, tomato, and fries
                    ", true, 3.99));
        cafeMenu.add(new MenuItem("Soup of the day", "A cup of the soup of the
            day, with a side salad", false, 3.69));
        cafeMenu.add(new MenuItem("Burrito", "A large burrito, with whole pinto
             beans, salsa, guacamole", true, 4.29));

        cafeMenu.add(coffeeMenu);

        coffeeMenu.add(new MenuItem("Coffee Cake", "Crumbly cake topped with
            cinnamon and walnuts", true, 1.59));
        coffeeMenu
                .add(new MenuItem("Bagel", "Flavors include sesame, poppyseed,
                    cinnamon raisin, pumpkin", false, 0.69));
        coffeeMenu.add(new MenuItem("Biscotti", "Three almond or hazelnut
            biscotti cookies", true, 0.89));

        Waitress waitress = new Waitress(allMenus);

        waitress.printMenu();
        System.out.println();
        waitress.printVegetarianMenu();
        System.out.println();
        waitress.printNonVegetarianMenu();
        System.out.println();
        waitress.printExpensiveMenu();
    }

}
```

1.10 Command

1.10.1 Hoe herken je dit?

1.10.2 Theorie

Theorie - slides:

- Het Command Pattern schermt een aanroep af door middel van een object, waarbij je verschillende aanroepen in verschillende objecten kun opbergen, in een queue kunt zetten of op schijf kunt bewaren; ook undo-operaties kunnen worden ondersteund.

Notities uit de les:

- Er is ergens een soort commando dat je kan doen (ventilatie aanzetten), die meerdere subcommando's hebben (hier zachte, middlematige of harde ventilatie)
- Er zijn Commando's die kunnen worden uitgevoerd en er zijn Receivers die commando's uitvoeren

1.10.3 UML

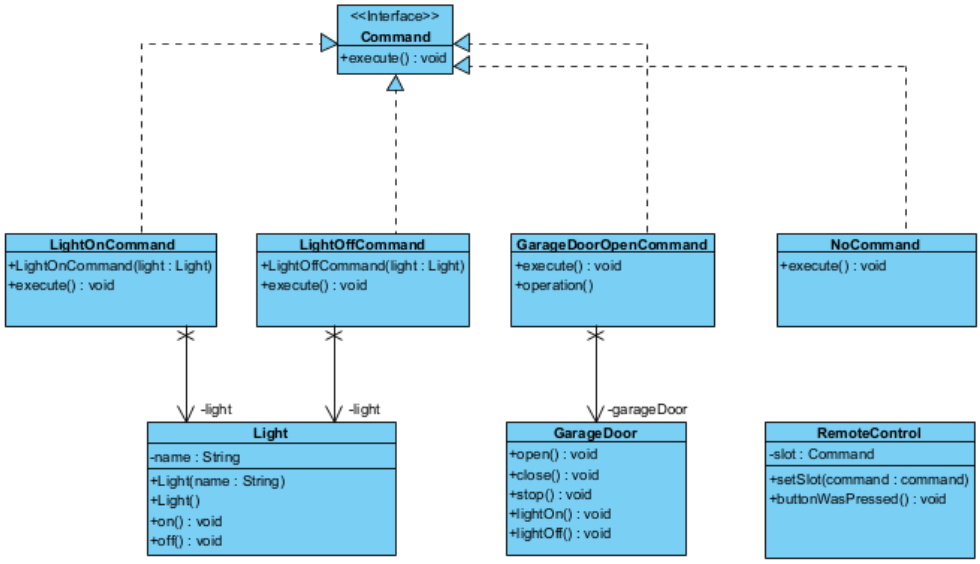

Figure 10: alt text

1.10.4 Implementatie

- We starten met het maken van een **Interface** van alle commands die zullen worden bijgehouden

```
package domein.interfaces;

public interface Bestelling {
    void voerUit();
}
```

- Nadien maken we onze domeinobjecten

```
package domein;

import domein.interfaces.Bestelling;

/**
 * De Serveerster roept de bestelling om.
 *
 * Invoker is losgekoppeld van de Receiver (kok)
 */
public class Serveerster {
    private Bestelling bestelling;

    public void neemBestelling(Bestelling bestelling) {
        this.bestelling = bestelling;
    }

    public void roepBestellingOm() {
        System.out.println("Serveerster roept de bestelling om!");
```

```
        bestelling.voerUit();
    }
}
```

```
package domein;

/**
 * De Kok klasse is verantwoordelijk voor het bereiden van gerechten in het
 * restaurant. Het bevat methoden om een burger en frietjes te maken.
 *
 * Receiver: De Kok klasse is de ontvanger van de commando's.
 */
public class Kok {
    public void maakBurger() {
        System.out.println("Kok maakt een burger klaar!");
    }

    public void maakFrietjes() {
        System.out.println("Kok maakt een portie frietjes klaar!");
    }
}
```

- Nu maken we de klassen die de commando's zullen implementeren:

```
package domein;

import domein.interfaces.Bestelling;

/**
 * De FrietjesBestelling klasse is verantwoordelijk voor het uitvoeren van een
 * bestelling voor frietjes. Het maakt gebruik van de Kok klasse om de frietjes
 * te bereiden.
 *
 * Concrete Command - (bestelling) is losgekoppeld van de uitvoerder (kok)
 */
public class FrietjesBestelling implements Bestelling {
    private Kok kok;

    public FrietjesBestelling(Kok kok) {
        this.kok = kok;
    }

    @Override
    public void voerUit() {
        kok.maakFrietjes();
    }
}
```

```
package domein;

import domein.interfaces.Bestelling;

/**
 * De BurgerBestelling klasse is verantwoordelijk voor het uitvoeren van een
 * bestelling voor een burger. Het maakt gebruik van de Kok klasse om de burger
 * te bereiden.
 *
```

```
 * Concrete Command - (bestelling) is losgekoppeld van de uitvoerder (kok)
 */
public class BurgerBestelling implements Bestelling {
    private Kok kok;

    public BurgerBestelling(Kok kok) {
        this.kok = kok;
    }

    @Override
    public void voerUit() {
        kok.maakBurger();
    }
}
```

- Nu uiteindelijk kunnen we deze applicatie uittesten:

```
package cui;

import domein.BurgerBestelling;
import domein.FrietjesBestelling;
import domein.Kok;
import domein.Serveerster;
import domein.interfaces.Bestelling;

public class RestaurentApplicatie {
    public RestaurentApplicatie() {
        // Ontvanger (de kok)
        Kok kok = new Kok();

        // Concrete Commands
        Bestelling burgerBestelling = new BurgerBestelling(kok);
        Bestelling frietBestelling = new FrietjesBestelling(kok);

        // Invoker (serveerster)
        Serveerster serveerster = new Serveerster();

        // Klant bestelt burger
        serveerster.neemBestelling(burgerBestelling);
        serveerster.roepBestellingOm();

        // Klant bestelt frietjes
        serveerster.neemBestelling(frietBestelling);
        serveerster.roepBestellingOm();
    }
}
```

1.11 Template Method

1.11.1 Hoe herken je dit?

1.11.2 Theorie

Theorie - slides:

Notities uit de les:

- Je wilt ongeveer de zelfde dingen doen in een bepaalde volgorde maar ze verschillen net iets van elkaar (zoals de procesverschillen tussen koffie en thee)

- Dit werkt via **abstracte** klassen/methoden (HotDrink bv)
- De templatemethode bevat dus in die abstracte klasse bevat dus een aantal methodes op volgorde
- De effectie 'maak'-methode is altijd **public** en **final**
- De protectedmethoden in Hotdrink, **MOETEN** overschreven worden
- Er wordt gebruik gemaakt van hooks wat defaultwaarden heeft, zoals needCondiments wat altijd **FALSE** is
- Hoe soorten methoden te herkennen:
  - De enige methode die iets 'aanmaakt': **public en final**
  - Een methode die overal hetzelfde is: **private**
  - Een methode die waarschijnlijk maar bij één object wordt geïmplementeerd (hook die een defaultwaarde nodig heeft): **protected**
  - Een methode die beide objecten nodig heeft, maar alles waarschijnlijk anders implementeerd: **abstract en protected**

1.11.3 UML

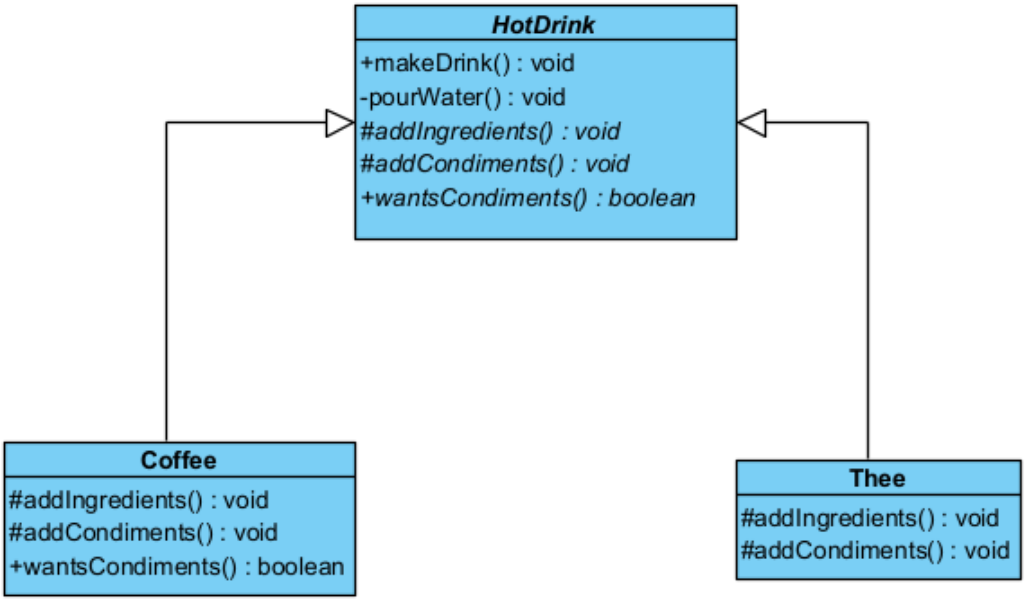

Figure 11: alt text

1.11.4 Implementatie

1.12 Remote Proxy

1.12.1 Hoe herken je dit?

1.12.2 Theorie

Theorie - slides:

Notities uit de les:

- Een soort VPN-achtig
- Dingen die dichter bij u zijn (een soort kopie) ipv van het echt/grote ding.
  - Bv. een kleine dicht Delhaize dicht in de buurt ipv een grote Delhaize iets verder in de buurt
- Gebruikt 2 klassen die elk van één interface erven
- Het is een dus een kopie en het bevat hetgeen wat hij moet gaan voorstellen, maar het is niet echt
- Toegang ergens uitzetten kan makkelijk worden gedaan via het Proxy-pattern

1.12.3 UML

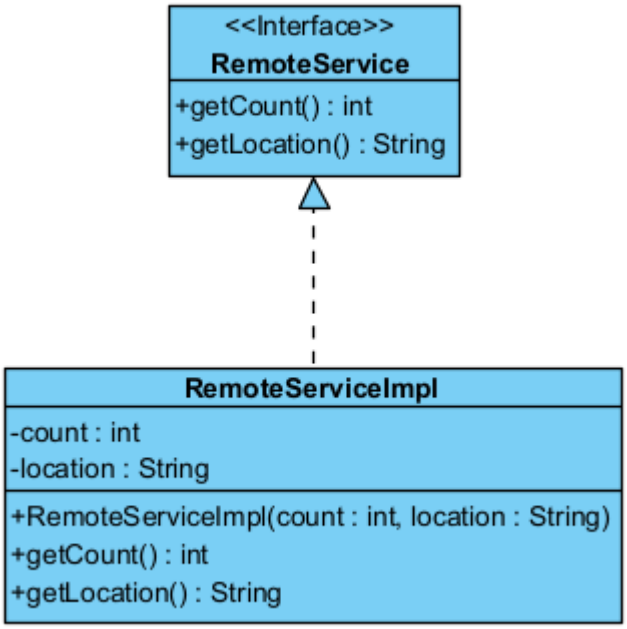

Figure 12: alt text

1.12.4 Implementatie

- We starten met het maken van een Interface dan extend naar `Remote` :

```
package domein;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteService extends Remote {
    String zegHallo(String naam) throws RemoteException;
}
```

- Daarna maken we daar onze implementatie van dat de interface implementeert alsook extend naar `UnicastRemoteObject` :

```
package domein;

import common.RemoteService;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class RemoteServiceImpl extends UnicastRemoteObject implements
   RemoteService {

    public RemoteServiceImpl() throws RemoteException {
        super();
    }

    @Override
```

```
    public String zegHallo(String naam) throws RemoteException {
        return "Hallo, " + naam + " vanop de server!";
    }
}
```

- En uiteindelijk kunnen we de server opstarten:

```
package server;

import common.RemoteService;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class ServerApp {

    public static void main(String[] args) {
        try {
            RemoteService service = new RemoteServiceImpl();

            // Start RMI registry op poort 1099
            Registry registry = LocateRegistry.createRegistry(1099);

            // Registreer het object onder naam
            registry.rebind("HelloService", service);

            System.out.println("Server klaar. Wacht op clients...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- Nu moeten we enkel nog een client bouwen dat met de server communiceert en voila, we kunnen onze RemoteObjecten vanop afstand moniteren:

```
package client;

import common.RemoteService;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class ClientApp {

    public static void main(String[] args) {
        try {
            // Zoek naar RMI registry op localhost
            Registry registry = LocateRegistry.getRegistry("localhost", 1099);

            // Zoek het remote object
            RemoteService service = (RemoteService) registry.lookup("
                HelloService");

            // Roep methode aan op server via proxy
            String response = service.zegHallo("Student");
            System.out.println("Server zegt: " + response);
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
    }
}
```

1.13 Virtual Proxy

1.13.1 Hoe herken je dit?

1.13.2 Theorie

Theorie - slides:

Notities uit de les:

1.13.3 UML

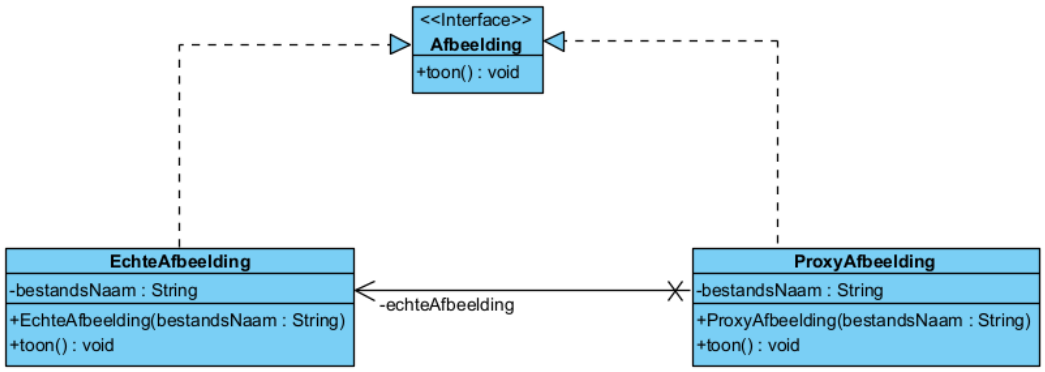

Figure 13: alt text

1.13.4 Implementatie

- We maken eerst een interface Afbeelding

```
package domein;

// Stap 1: maak een interface
public interface Afbeelding {
    void toon();
}
```

- Vervolgens maken we hier implementaties van
- De eerste is de echtAfbeelding:

```
package domein;

public class EchteAfbeelding implements Afbeelding {

    private final String bestandsnaam;

    public EchteAfbeelding(String bestandsnaam) {
        this.bestandsnaam = bestandsnaam;
        laadVanDisk(bestandsnaam);
    }

    private void laadVanDisk(String bestandsnaam) {
        System.out.println("Afbeelding \"" + bestandsnaam + "\" wordt geladen
            van de schijf...");
        try {
```

```
            Thread.sleep(2000); // Simuleer traag laden
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    @Override
    public void toon() {
        System.out.println("Afbeelding \"" + bestandsnaam + "\" wordt getoond."
           );
    }
}
```

- Vervolgens de ProxyAfbeelding
- Het is zijn taak om de huidige foto te vervang terwijl hij nog bezig is met het inladen ervan:

```
package domein;

public class ProxyAfbeelding implements Afbeelding {

    private String bestandsnaam;
    private EchteAfbeelding echteAfbeelding;

    public ProxyAfbeelding(String bestandsnaam) {
        this.bestandsnaam = bestandsnaam;
    }

    @Override
    public void toon() {
        if (echteAfbeelding == null) {
            echteAfbeelding = new EchteAfbeelding(bestandsnaam);
        }
        echteAfbeelding.toon();
    }

}
```

- En voila we kunnen foto's maken

```
package main;

import domein.Afbeelding;
import domein.ProxyAfbeelding;

public class Applicatie {

    public static void main(String[] args) {
        Afbeelding afbeelding1 = new ProxyAfbeelding("zomerfoto.jpg");

        System.out.println("Afbeelding gemaakt, maar nog niet getoond...");

        System.out.println("Eerste keer tonen:");
        afbeelding1.toon(); // attribuut "echteAfbeelding" is nog null in de
           ProxyAfbeelding, maar hier
                            // wordt de echte afbeelding geladen

        System.out.println("Tweede keer tonen:");
```

```
        afbeelding1.toon(); // waardoor dat de volgende keer dat deze methode
            wordt aangeroepen, de echte
                            // afbeelding al in het geheugen zit en direct
                               getoond kan worden
    }
}
```

1.14 Protection Proxy

1.14.1 Hoe herken je dit?

1.14.2 Theorie

Theorie - slides:

Notities uit de les:

1.14.3 UML

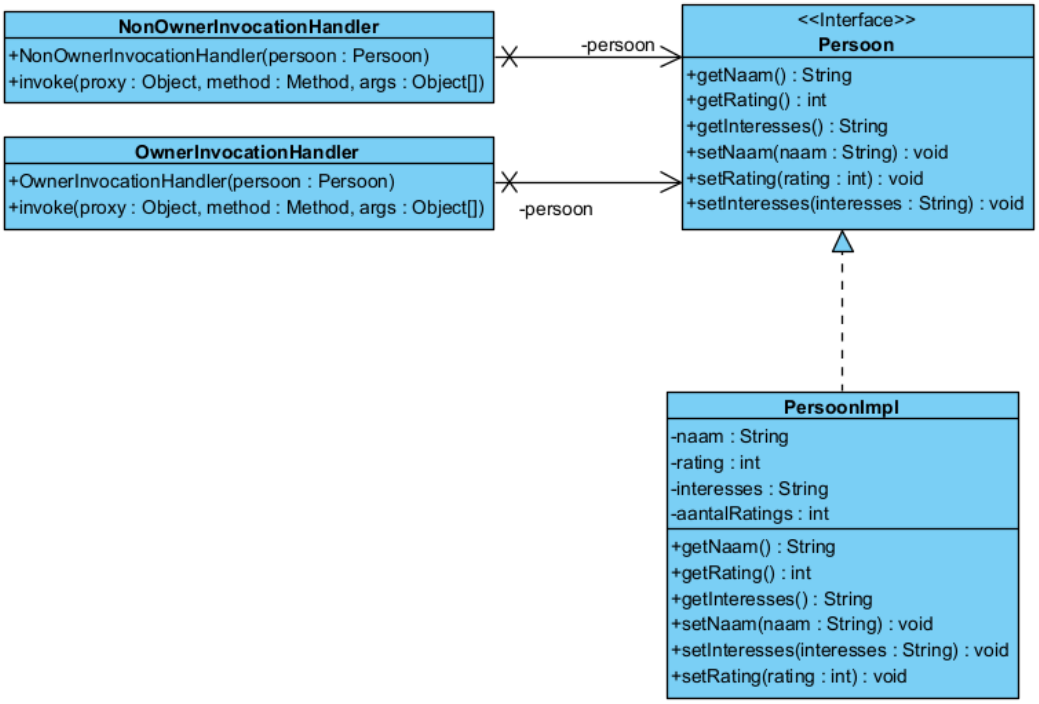

Figure 14: alt text

1.14.4 Implementatie

- Eerst maken we een interface

```
package domein;

public interface Persoon {
    String getNaam();

    String getInteresses();

    int getRating();

    void setNaam(String naam);
```

```
    void setInteresses(String interesses);

    void setRating(int rating);
}
```

- Daar maken we een concrete klasse van:

```
package domein;

public class PersoonImpl implements Persoon {

    private String naam;
    private String interesses;
    private int rating = 0;
    private int ratingAantal = 0;

    @Override
    public String getNaam() {
        return naam;
    }

    @Override
    public void setNaam(String naam) {
        this.naam = naam;
    }

    @Override
    public String getInteresses() {
        return interesses;
    }

    @Override
    public void setInteresses(String interesses) {
        this.interesses = interesses;
    }

    @Override
    public int getRating() {
        if (ratingAantal == 0)
            return 0;
        return rating / ratingAantal;
    }

    @Override
    public void setRating(int rating) {
        this.rating += rating;
        ratingAantal++;
    }
}
```

- Vervolgens maken we onze invokers:

```
package domein;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
```

```
public class NonOwnerInvocationHandler implements InvocationHandler {

    private Persoon persoon;

    public NonOwnerInvocationHandler(Persoon persoon) {
        this.persoon = persoon;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
        if (method.getName().startsWith("set") && !method.getName().equals("
            setRating")) {
            throw new IllegalAccessException("Je mag interesses niet aanpassen"
                );
        } else {
            return method.invoke(persoon, args);
        }
    }
}
```

```
package domein;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class OwnerInvocationHandler implements InvocationHandler {

    private Persoon persoon;

    public OwnerInvocationHandler(Persoon persoon) {
        this.persoon = persoon;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
        if (method.getName().equals("setRating")) {
            throw new IllegalAccessException("Eigen score mag je niet zetten");
        } else {
            return method.invoke(persoon, args);
        }
    }
}
```

- En dit kunnen we in de praktijk gaan omzetten:

```
package cui;

import java.lang.reflect.Proxy;

import domein.NonOwnerInvocationHandler;
import domein.OwnerInvocationHandler;
import domein.Persoon;
import domein.PersoonImpl;

public class ProtectionProxyTestDrive {
```

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

        PersoonImpl jan = new PersoonImpl();
        jan.setNaam("Jan");
        jan.setInteresses("Muziek, films");

        // Proxy voor eigenaar (Jan zelf)
        Persoon janAlsEigenaar = (Persoon) Proxy.newProxyInstance(jan.getClass
            ().getClassLoader(),
                jan.getClass().getInterfaces(), new OwnerInvocationHandler(jan)
                    );

        // Proxy voor iemand anders die Jan bekijkt
        Persoon janAlsBezoeker = (Persoon) Proxy.newProxyInstance(jan.getClass
            ().getClassLoader(),
                jan.getClass().getInterfaces(), new NonOwnerInvocationHandler(
                    jan));

        System.out.println(">>> EIGENAAR PROXY <<<");
        System.out.println(janAlsEigenaar.getNaam());
        janAlsEigenaar.setInteresses("Voetbal, reizen");
        System.out.println("Interesses aangepast!");

        try {
            janAlsEigenaar.setRating(10); // verboden
        } catch (Exception e) {
            System.out.println("Mag geen score geven aan jezelf.");
        }

        System.out.println(">>> BEZOEKER PROXY <<<");
        System.out.println(janAlsBezoeker.getNaam());

        try {
            janAlsBezoeker.setInteresses("Veranderen"); // verboden
        } catch (Exception e) {
            System.out.println("Mag interesses niet aanpassen.");
        }

        janAlsBezoeker.setRating(8);
        System.out.println("Rating gezet: " + janAlsBezoeker.getRating());
    }
}
```

1.15 Tips examen

- Voor het zoeken welk design pattern een oefening is, moet je je de volgende zaken afvragen:
  - Moet je een object maken?
    - **Factory Method**: Volledige objecten met slechts een deel van de familie (GentPizzeria of OostendePizzeria)
    - **Abstract Factory**: Volledige objecten met meerdere families
    - **Builder**: afdwingen van een bepaalde volgorde
    - **Builder Variant**: hier is volgorde niet van belang, wel de stukjes ervan
    - **Singleton**: Slechts 1 instantie
  - Is het een proces dat het moet afgaan
    - Template Method:
    - Command

    - Iterator
  - Geen van beide? Dan is het één van deze:
    - Composite
    - Adapter

2 Java

2.1 Multithreading

2.1.1 Theorie

Theorie - slides:

Notities uit de les:

- Threads zijn wanneer meerdere methodes die tegelijkertijd gebeuren
- Voordat java wordt uitgevoerd worden bepaald regels toegepaste ... zoals de javaVM, maar uiteindelijk of er wel of niet iets speciaals zal gebeuren met de implementatie van meerdere threads zal je OS dit bepalen ondanks het feit dat java OS-onafhankelijk is
- Slides te kennen: 2, 3, 4, 9, 10, 12, 13, 14, 15, 17, 18, 31, 46, 47, 55
- Oefeningen te kennen: zwembaden kennen
- Slides niet te kennen: 49, 59+

2.1.2

2.1.3 Producer maken

```
package domein;

import java.security.SecureRandom;

import lombok.extern.slf4j.Slf4j;

@Slf4j
public class Kok implements Runnable {

    private final Restaurant restaurant;
    private static final SecureRandom random = new SecureRandom(); // voor
       willekeurige pauzes

    public Kok(Restaurant restaurant) {
        this.restaurant = restaurant;
    }

    @Override
    public void run() {
        while (true) { // blijf produceren zolang het programma draait
            try {
                Thread.sleep(random.nextInt(2000)); // wacht willekeurig tot 2
                   seconden
                restaurant.plaatsOrder(new Order()); // plaats een nieuwe order
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

2.1.4 Consumer maken

```
package domein;
```

```
import java.security.SecureRandom;

import lombok.extern.slf4j.Slf4j;

@Slf4j
public class Kelner implements Runnable {

    private final Restaurant restaurant;
    private final String naam;
    private static final SecureRandom random = new SecureRandom();

    public Kelner(Restaurant restaurant, String naam) {
        this.restaurant = restaurant;
        this.naam = naam;
    }

    @Override
    public void run() {
        while (true) { // blijf consumeren zolang het programma draait
            try {
                Thread.sleep(random.nextInt(2000)); // wacht willekeurig
                Order order = restaurant.haalOrderOp(); // haal order op
                System.out.printf("Kelner %s krijgt %s%n", naam, order);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

2.1.5 'Werkplek' maken van de Producer en Consumer:

- Hier maken we gebruik van locks en conditions om ervoor te zorgen dat slechts één Thread er tegelijk aankan

```
package domein;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

// Dit is de "shared resource" tussen kok en kelners.
public class Restaurant {

    private Order order; // bevat de huidige bestelling, maximaal 1 tegelijk

    // Lock en voorwaarden om threadveilig te kunnen werken
    private Lock accessLock = new ReentrantLock();
    private Condition kanPlaatsen = accessLock.newCondition(); //
       wachtvoorwaarde voor kok
    private Condition kanOphalen = accessLock.newCondition(); //
       wachtvoorwaarde voor kelner

    // Methode voor de kok om een order te plaatsen
    public void plaatsOrder(Order o) {
        accessLock.lock(); // probeer toegang te krijgen tot de gedeelde
           variabele
        try {
            // Kok moet wachten zolang er nog een order ligt
```

```
            while (order != null) {
                System.out.println("Kok moet wachten, er staat nog een order
                    ...");
                kanPlaatsen.await(); // kok wacht op signaal om te plaatsen
            }

            // Er is plaats, dus order wordt geplaatst
            this.order = o;
            System.out.println("Kok plaatst: " + o);

            kanOphalen.signal(); // signaleer dat een kelner het mag komen
                ophalen

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            accessLock.unlock(); // vergrendeling loslaten
        }
    }

    // Methode voor de kelner om een order op te halen
    public Order haalOrderOp() {
        accessLock.lock();
        Order toReturn = null;

        try {
            // Wachten tot er effectief een order is
            while (order == null) {
                System.out.println("Kelner moet wachten tot er een order is ...
                    ");
                kanOphalen.await();
            }

            // Order ophalen
            toReturn = this.order;
            this.order = null; // nu is er weer plaats voor een nieuwe order

            System.out.println("Order wordt opgehaald: " + toReturn);

            kanPlaatsen.signal(); // signaleer aan de kok dat hij een nieuwe
                order mag plaatsen

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            accessLock.unlock();
        }

        return toReturn;
    }
}
```

2.1.6 Opstarten van een applicatie

- We maken al onze objecten aan die Threads zullen worden en via de `fixedThreadpool` maken we de Threads effectief aan:

```
        Restaurant restaurant = new Restaurant();
```

```
        Kelner kelner1 = new Kelner(restaurant, "Sofie");
        Kelner kelner2 = new Kelner(restaurant, "Hendrik");
        Kok kok = new Kok(restaurant);

        // 3 threads
        ExecutorService app = Executors.newFixedThreadPool(3);
        app.execute(kok);
        app.execute(kelner1);
        app.execute(kelner2);

        app.shutdown();
```

2.1.7 Zwembad oefening

```
package domein;

import lombok.Getter;

public class Zwembad {

    private final int CAPACITEIT;
    @Getter
    private int inhoud;

    public Zwembad(int cap) {
        CAPACITEIT = cap;
    }

    public synchronized void gietEmmer() {
        inhoud++;
    }

    public boolean vol() {
        return inhoud == CAPACITEIT;
    }

    public boolean leeg() {
        return inhoud == 0;
    }
}
```

```
package domein;

import java.security.SecureRandom;

import lombok.AllArgsConstructor;
import lombok.extern.slf4j.Slf4j;

@AllArgsConstructor
@Slf4j
public class Kind implements Runnable {

    private final Tafel tafel;
    private final Zwembad zwembad;
    private final String naam;
    private static final SecureRandom random = new SecureRandom();
```

```
    @Override
    public void run() {
        while (true) {
            // Stop als zwembad vol is
            if (zwembad.vol()) {
                System.out.printf("%s: zwembad is vol (%d emmers)%n", naam,
                    zwembad.getInhoud());
                break;
            }

            // Stop als vat leeg is en er geen emmers meer zijn
            if (tafel.isVatLeeg() && zwembad.getInhoud() >= 9) {
                System.out.printf("%s: vat is leeg, geen emmers meer %n", naam)
                    ;
                break;
            }

            // Probeer emmer te nemen
            boolean emmer = tafel.pakEmmer();
            if (emmer) {
                System.out.printf("%s neemt een emmer%n", naam);
                try {
                    Thread.sleep(random.nextInt(1500) + 500); // giet wat
                        trager
                } catch (InterruptedException e) {
                    log.error("Thread interrupted", e);
                    Thread.currentThread().interrupt();
                }
                zwembad.gietEmmer();
            }
        }
    }
}
```

```
package domein;

import lombok.AllArgsConstructor;

@AllArgsConstructor
public class Vat implements Runnable {

    private final int inhoud; // totaal aantal emmers die het vat kan vullen
    private final Tafel tafel;

    @Override
    public void run() {
        for (int i = 0; i < inhoud; i++) {
            tafel.vulEmmer(); // 1 emmer vullen
            System.out.println("Vat: emmer is gevuld");
        }

        // Signaal aan de tafel dat het vat leeg is
        tafel.setVatLeeg(true);
        System.out.println("Vat: vat is leeg");
    }
}
```

Eigenschap	Betekenis
`BlockingQueue`	Een thread-safe wachtrij die blokkeert als vol/leeg
`ArrayBlockingQueue`	Een wachtrij met **vaste capaciteit**
`put()`	Wacht als de queue **vol is**
`take()`	Wacht als de queue **leeg is**

```java
package domein;

import java.security.SecureRandom;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

import lombok.Getter;
import lombok.Setter;
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class Tafel {

    private static final SecureRandom random = new SecureRandom();

    @Setter
    @Getter
    private volatile boolean vatLeeg = false; // Volatile: meerdere threads
       kunnen dit goed lezen

    private final BlockingQueue<Boolean> emmerQueue;

    public Tafel(int aantalEmmers) {
        this.emmerQueue = new ArrayBlockingQueue<>(aantalEmmers);
    }

    // Vat vult een emmer
    public void vulEmmer() {
        try {
            emmerQueue.put(true); // blokkeert als vol
        } catch (InterruptedException e) {
            log.error("Thread interrupted", e);
            Thread.currentThread().interrupt();
        }
    }

    // Kind neemt een emmer
    public boolean pakEmmer() {
        try {
            return emmerQueue.take(); // blokkeert als leeg
        } catch (InterruptedException e) {
            log.error("Thread interrupted", e);
            Thread.currentThread().interrupt();
            return false;
        }
    }
}
```

```java
package main;
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.stream.IntStream;

import domein.Kind;
import domein.Tafel;
import domein.Vat;
import domein.Zwembad;

public class ZwembadApplicatie {

    public static void main(String[] args) {
        new ZwembadApplicatie().run();
    }

    public void run() {
        // 1. Maak centrale objecten
        Tafel tafel = new Tafel(2); // max 2 emmers op tafel
        Zwembad zwembad = new Zwembad(6); // zwembad moet 4 emmers hebben
        Vat vat = new Vat(9, tafel); // vat kan 9 emmers leveren

        // 2. Maak kinderen
        Kind[] kinderen = new Kind[3];
        IntStream.range(0, kinderen.length).forEach(i -> kinderen[i] = new Kind
            (tafel, zwembad, "Kind " + (i + 1)));

        // 3. Start alles via threadpool
        ExecutorService pool = Executors.newFixedThreadPool(4);
        pool.execute(vat); // vat vult eerst
        for (Kind kind : kinderen)
            pool.execute(kind);

        // 4. Automatisch afsluiten is niet nodig, programma stopt als alles
            klaar is
        pool.shutdown();
    }
}
```