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

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());
    }
}
```

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

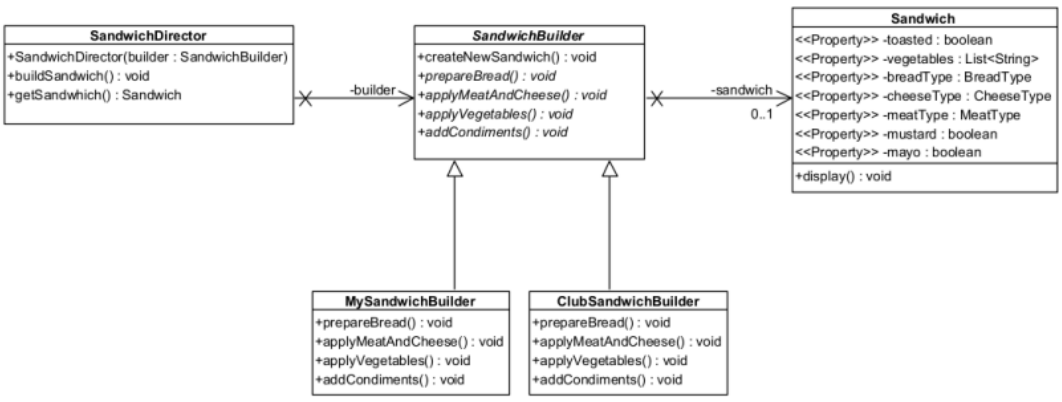

Figure 1: UML Builder Pattern

1.3.4 Implementatie

- We starten met het maken van de domeinklasse waar de builder voor zal dienen:
- Er is geen constructor aangezien de Builder al het werk gaat doen

```java
package domein;

import java.util.List;

import domein.interfaces.BreadType;
import lombok.AccessLevel;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter(AccessLevel.PROTECTED)
@NoArgConstructor
public class Sandwich {

    private BreadType breadType;
    private boolean isToasted;
    private List<String> vegetables;
    private List<String> condiments;

    @Override
    public String toString() {
        return String.format("Sandwich [breadType=%s, isToasted=%s, vegetables
           =%s, condiments=%s]", breadType,
                isToasted, vegetables, condiments);
    }
}
```

- 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

```java
package domein;

import lombok.Getter;

public abstract class SandwichBuilder {

    @Getter
    private Sandwich sandwich;

    public void createNewSandwich() {
        sandwich = new Sandwich();
    }

    public abstract void buildBreadType();

    public abstract void buildToasted();

    public abstract void buildVegetables();

    public abstract void buildCondiments();

}
```

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

```java
package domein;
```

```java
import java.util.List;

import domein.interfaces.BreadType;

public class MartinoSandwich extends SandwichBuilder {

    @Override
    public void buildBreadType() {
        getSandwich().setBreadType(BreadType.BROWN);
    }

    @Override
    public void buildToasted() {
        getSandwich().setToasted(true);
    }

    @Override
    public void buildVegetables() {
        getSandwich().setVegetables(List.of("Lettuce", "Tomato"));
    }

    @Override
    public void buildCondiments() {
        getSandwich().setCondiments(List.of("Prepare", "Martino Sauce"));
    }

}
```

- 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

```java
package domein;

import lombok.AllArgsConstructor;

@AllArgsConstructor
public class SandwichDirector {

    private SandwichBuilder builder;

    public void buildSandwich() {
        builder.createNewSandwich();
        builder.buildBreadType();
        builder.buildToasted();
        builder.buildCondiments();
        builder.buildVegetables();
    }

    public Sandwich getSandwich() {
        return builder.getSandwich();
    }

}
```

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

```
package cui;

import domein.ChickenSandwich;
import domein.MartinoSandwich;
import domein.SandwichDirector;

public class SandwichApplicatie {

    public SandwichApplicatie() {
        maakSandwiches();
    }

    private void maakSandwiches() {
        SandwichDirector chickenDirector = new SandwichDirector(new
           ChickenSandwich());
        SandwichDirector martinoDirector = new SandwichDirector(new
           MartinoSandwich());

        chickenDirector.buildSandwich();
        chickenDirector.getSandwich().toString();

        martinoDirector.buildSandwich();
        martinoDirector.getSandwich().toString();
    }
}
```

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 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

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() {
        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

1.6.4 Implementatie

- We beginnen met het maken van de interfaces

```
public interface Eend {
    void kwak();
    void vlieg();
}
```

```
public interface Kalkoen {
    void gok();
    void vliegKorteAfstand();
}
```

- Nu maken we onze domeinklassen die de interfaces implementeren:

```
public class WildeKalkoen implements Kalkoen {
    public void gok() {
        System.out.println("Gok gok");
    }

    public void vliegKorteAfstand() {
        System.out.println("Ik vlieg een korte afstand");
    }
}
```

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

```
public class KalkoenAdapter implements Eend {
    Kalkoen kalkoen;

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

    public void kwak() {
        kalkoen.gok();
    }

    public void vlieg() {
        for (int i = 0; i < 5; i++) {
            kalkoen.vliegKorteAfstand();
        }
    }
}
```

- 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 Composite

1.7.1 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 ...)

1.7.2 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

1.7.3 UML

1.7.4 Implementatie

1.8 Iterator (Eigen 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
- 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

1.8.3 UML

1.8.4 Implementatie

- Eerst moeten we onze eigen Iterator-interface aanmaken

```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

- Gevolgd door de domeinklassen

```
public class Gerecht {
    private String naam;
    private double prijs;

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

    public String getNaam() {
        return naam;
    }

    public double getPrijs() {
        return prijs;
    }
}
```

```
public class Menu {
    private Gerecht[] gerechten;
    private int aantal = 0;

    public Menu() {
        gerechten = new Gerecht[10];
        voegToe("Pasta", 12.5);
        voegToe("Pizza", 10.0);
        voegToe("Salade", 8.5);
    }

    public void voegToe(String naam, double prijs) {
        if (aantal < gerechten.length) {
            gerechten[aantal++] = new Gerecht(naam, prijs);
        }
    }

    public Iterator createIterator() {
        return new MenuIterator(gerechten);
    }
}
```

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

```
public class MenuIterator implements Iterator {
    private Gerecht[] items;
    private int positie = 0;

    public MenuIterator(Gerecht[] items) {
        this.items = items;
    }
```

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

    @Override
    public Object next() {
        return items[positie++];
    }
}
```

- Nu dit klaar is kunnen we de applicatie uittesten:

```
public class ItteratorApplicatie {
    public static void main(String[] args) {
        Menu menu = new Menu();
        Iterator iterator = menu.createIterator();

        while (iterator.hasNext()) {
            Gerecht g = (Gerecht) iterator.next();
            System.out.println(g.getNaam() + " - " + g.getPrijs() + " euro");
        }
    }
}
```

1.9 Iterator (Java Iterator)

1.9.1 Hoe herken je dit?

- Je wil de objecten overlopen

1.9.2 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

1.9.3 UML

1.9.4 Implementatie

- Eerst moeten we onze eigen Iterator-interface aanmaken

```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

- Gevolgd door de domeinklassen

```
public class Gerecht {
    private String naam;
    private double prijs;
```

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

    public String getNaam() {
        return naam;
    }

    public double getPrijs() {
        return prijs;
    }
}
```

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

public class Menu implements Iterable<Gerecht> {
    private List<Gerecht> gerechten = new ArrayList<>();

    public Menu() {
        voegToe("Pasta", 12.5);
        voegToe("Pizza", 10.0);
        voegToe("Salade", 8.5);
    }

    public void voegToe(String naam, double prijs) {
        gerechten.add(new Gerecht(naam, prijs));
    }

    @Override
    public Iterator<Gerecht> iterator() {
        return gerechten.iterator();
    }
}
```

- Nu dit klaar is kunnen we de applicatie uittesten:

```
public class ItteratorApplicatie {
    public static void main(String[] args) {
        Menu menu = new Menu();

        for (Gerecht g : menu) {
            System.out.println(g.getNaam() + " - " + g.getPrijs() + " euro ");
        }
    }
}
```

1.10 Protection Proxy

1.10.1 Hoe herken je dit?

1.11 Command

1.11.1 Hoe herken je dit?

1.11.2 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

1.11.3 UML

1.11.4 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();
    }
}
```

1.12 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

1.12.4 Implementatie

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

1.13.4 Implementatie

1.13.5 UML

1.13.6 Implementatie

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

1.14.4 Implementatie

1.15 Remote Proxy

1.15.1 Hoe herken je dit?

1.15.2 Theorie

Theorie - slides:

Notities uit de les:

1.15.3 UML

1.15.4 Implementatie

1.16 Template Method

1.16.1 Hoe herken je dit?

1.16.2 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**

1.16.3 UML

1.16.4 Implementatie

1.17 Sjabloon - Pattern

1.17.1 Hoe herken je dit?

1.17.2 Theorie

Theorie - slides:

Notities uit de les:

1.17.3 UML

1.17.4 Implementatie

1.18 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+