Samenvatting van de cursus Advanced Software Development I

Robbe Magerman

06/01/2025

Inhoudstafel

1 Hoofdstuk 0: Opstart Project

1.1 Maken van een Maven project met nodige dependencys

In de `pom.xml` file moet je volgende inhoud plakken onder de `<version>`:

```
    <properties>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- Lombok-->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.34</version>
            <scope>provided</scope>
        </dependency>

        <!-- JUnit-->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.11.0</version>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-params</artifactId>
            <version>5.11.0</version>
        </dependency>

        <!-- Mockito-->
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-core</artifactId>
            <version>5.13.0</version>
        </dependency>
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-junit-jupiter</artifactId>
            <version>5.13.0</version>
        </dependency>
    </dependencies>
```

Daarna rechtermuisknop op de pom.xml –> Maven –> Update Project –> Yes

2 Hoofdstuk 1: Java

2.1 Collections

2.1.1 List

- Een List is een geordende collectie van elementen, die duplicaten kan bevatten. Elk element heeft een index, en je kunt toegang krijgen tot elementen via hun positie in de lijst.
- Dit zijn de verschillende soorten lijsten:
  - **ArrayList**
    - Een dynamische array die groeit naarmate er elementen worden toegevoegd. Voordeel is snelle willekeurige toegang.
  - **LinkedList**
    - Gebaseerd op een dubbel-gekoppelde lijst, waardoor het sneller is voor het invoegen/verwijderen van elementen, maar langzamer voor willekeurige toegang.
  - **Stel dat je enkel maar één type in de lijst wil steken**

```
List checkedRawList = Collections.checkedList(rawList, Integer.class);
```

- Belangrijkste methodes:
  - **add(E element)**:
    - Voeg een element toe aan het einde van de lijst.
  - **get(int index)**:
    - Haal het element op een bepaalde index op.
  - **remove(int index)**:
    - Verwijder het element op een bepaalde index.
  - **size()**:
    - Geeft het aantal elementen in de lijst terug.
  - **contains(Object o)**:
    - Controleert of de lijst een bepaald element bevat.
  - **isEmpty()**:
    - Controleert of de lijst leeg is.

2.1.2 Set

- Een Set is een collectie die geen duplicaten bevat. Een Set heeft geen volgorde voor de elementen, tenzij je een specifieke implementatie zoals TreeSet gebruikt.
- Dit zijn de verschillende soorten lijsten:
  - **HashSet**:
    - Gebaseerd op een hash-tabel, biedt snelle toegang tot elementen, maar zonder enige volgorde.
  - **LinkedHashSet**:
    - Zoals HashSet, maar behoudt de volgorde waarin elementen worden toegevoegd.
  - **TreeSet**:
    - Een Set die de elementen in gesorteerde volgorde opslaat (natuurlijke of op basis van een Comparator).
- Belangrijkste methoden:
  - **add(E element)**:
    - Voeg een element toe aan de set (voegt alleen toe als het element nog niet bestaat).
  - **remove(Object o)**:
    - Verwijder een element uit de set.
  - **contains(Object o)**:
    - Controleer of een element aanwezig is in de set.
  - **size()**:
    - Geeft het aantal elementen in de set terug.
  - **isEmpty()**:
    - Controleert of de set leeg is.

2.1.3 Queue

- Een Queue is een collectie die elementen op een bepaalde volgorde opslaat, meestal volgens het "First-In-First-Out" (FIFO)-principe. Sommige implementaties ondersteunen andere volgordes zoals prioriteitsvolgorde.
- Belangrijkste methoden:
  - **offer(E element)**:
    - Voeg een element toe aan de queue.
  - **poll()**:
    - Haal en verwijder het element aan de voorkant van de queue (of retourneer null als de queue leeg is).
  - **peek()**:
    - Haal het element aan de voorkant van de queue op (of retourneer null als de queue leeg is).
  - **isEmpty()**:
    - Controleer of de queue leeg is.
  - **size()**:
    - Geeft het aantal elementen in de queue terug.

2.1.4 Deque (Double-Ended Queue)

- Een Deque is een uitbreiding van Queue waarbij je elementen aan beide zijden (begin en eind) kunt toevoegen en verwijderen. Dit biedt meer flexibiliteit dan een gewone Queue.
- Dit zijn de verschillende soorten lijsten:
  - **ArrayDeque**:
    - Een zeer efficiënte implementatie van een Deque zonder capaciteitslimiet
- Dit zijn de belangrijkste methoden:
  - **addFirst(E element)**:
    - Voeg een element toe aan het begin van de deque.
  - **addLast(E element)**:
    - Voeg een element toe aan het einde van de deque.
  - **removeFirst()**:
    - Verwijder het eerste element.
  - **removeLast()**:
    - Verwijder het laatste element.
  - **getFirst()**:
    - Haal het eerste element zonder te verwijderen.
  - **getLast()**:
    - Haal het laatste element zonder te verwijderen.

2.1.5 Stack

- Een Stack is een LIFO (Last-In-First-Out) collectie. Je voegt elementen toe en haalt ze af van de bovenkant van de stack.
- Dit zijn de verschillende soorten lijsten:
  - **push(E element)**:
    - Voeg een element toe aan de bovenkant van de stack.
  - **pop()**:
    - Verwijder en retourneer het bovenste element van de stack.
  - **peek()**:
    - Bekijk het bovenste element zonder het te verwijderen.
  - **isEmpty()**:
    - Controleer of de stack leeg is.
  - **size()**:
    - Geeft het aantal elementen in de stack terug.

2.1.6 Map

- Een Map slaat paren van sleutels en waarden op. Elke sleutel is uniek, en je kunt de waarde ophalen via de sleutel.
- Dit zijn de verschillende soorten lijsten:

- **HashMap<K, V>**:
  - De meest gebruikte implementatie, gebaseerd op hashing, biedt snelle toegang tot elementen.
- **LinkedHashMap<K, V>**:
  - Zoals HashMap, maar behoudt de volgorde waarin de elementen zijn toegevoegd.
- **TreeMap<K, V>**:
  - Slaat de sleutel-waardeparen in gesorteerde volgorde op, gebaseerd op de natuurlijke volgorde van de sleutels of een Comparator.

- Dit zijn de belangrijkste methoden:
  - **put(K key, V value)**:
    - Voeg een sleutel-waarde paar toe.
  - **get(Object key)**:
    - Haal de waarde op bij de opgegeven sleutel.
  - **remove(Object key)**:
    - Verwijder het paar met de opgegeven sleutel.
  - **containsKey(Object key)**:
    - Controleer of de map een bepaalde sleutel bevat.
  - **size()**:
    - Geeft het aantal sleutel-waardeparen in de map terug.
- Overlopen van een map:

```
String result = map.entrySet()
                .stream()
                .map(entry -> "Key: " + entry.getKey() + ", Value: " + entry.
                    getValue())
                    //.map(entry -> "key: %s, value: %s".formatted(entry.getKey(),
                        entry.getValue()))
                .collect(Collectors.joining("\n"))
```

2.1.7 Verschil met Iterator en ListIterator

- Iterator is enkel om enkel te overlopen/verwijderen
- ListIterator kan je de lijst aanpassen via bv. `set()`
- Je kan via een Iterator ook een lijst omkeren:

```
    public void keerLijstOm(List<String> lijst) {
        ListIterator<String> iterator = lijst.listIterator(lijst.size());

        System.out.println("\nReversed List");

        while (iterator.hasPrevious()){
            System.out.printf("%d: %s%n", iterator.nextIndex(), iterator.previous()
                );
        }
    }
```

2.1.8 Mogelijke examenvraag: wanneer gebruik je een linkedlist en wanneer een arraylist

- LinkedList: Handig als je moet inserten en deleten
- ArrayList: Handig als je moet opvragen

2.1.9 Samenvatting Collections Framework

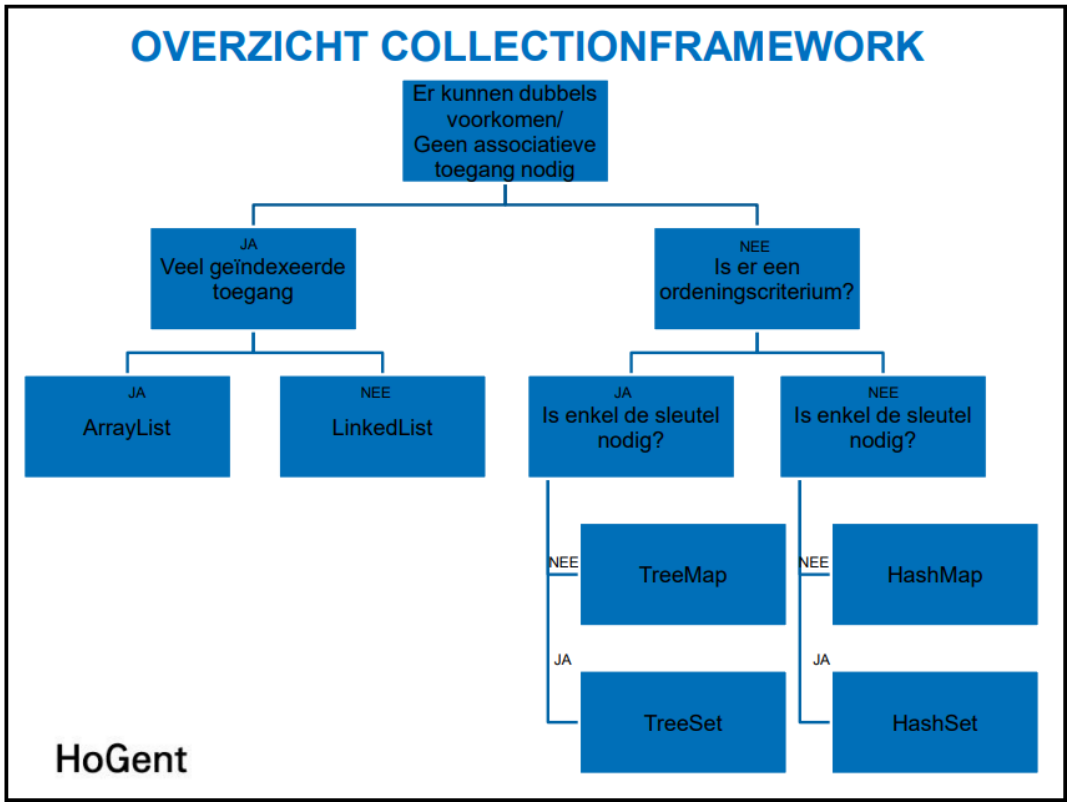


Figure 1: Schema om te kiezen welke Collectie je moet pakken

2.2 Streams

2.2.1 Immutibla vs mutable

- Een immutable collectie kan niet worden aangepast nadat deze is gemaakt. Dit betekent dat je geen elementen kunt toevoegen, verwijderen of wijzigen in de lijst.
  - Je bereikt dit door aan het einde van je stream het volgende toe te voegen:

```
lijst.stream().collect(Collectors.toList());
```

  - er zijn ook andere mogelijk, zoals `toSet()`
- Mutable: Een mutable lijst kan na het maken nog worden aangepast. Je kunt elementen toevoegen, verwijderen of wijzigen.
  - Je bereikt dit door aan het einde van je stream het volgende toe te voegen:

```
lijst.stream().toList()
```

2.2.2 Oplossen van een encapsulatielek / zorgen dat een lijst onveranderbaar is

```
return Collections.unmodifiableCollection(sportersLijst);
```

2.2.3 DOA (Data Access Object)

- DAO is een ontwerppatroon dat wordt gebruikt om toegang te krijgen tot gegevens vanuit een database of een andere opslagbron. Het idee is om de toegang tot de gegevensbron te scheiden van de business logica.

Een DAO is verantwoordelijk voor het uitvoeren van CRUD-operaties (Create, Read, Update, Delete) op de database.

2.2.4 JPA (Java Persistence API)

- JPA is een specificatie binnen de Java EE/ Jakarta EE-omgeving die helpt bij het mappen van Java-objecten op relationele databases. Het is geen bibliotheek of framework op zich, maar een reeks richtlijnen die worden geïmplementeerd door frameworks zoals Hibernate, EclipseLink of OpenJPA.

2.3 Generics

Wat zijn generieke methoden

- Dit zijn methoden waarbij meerdere datatypes mogelijk zijn
- Het maakt dus niet uit welke types hier worden ingestoken

2.3.1 Generieke methodes

Voorbeeld

- Je hebt 3 arrays die maar allemaal met een verschillend en je wil die afprinten
- Het probleem is dat je drie verschillende methoden moet schrijven waarbij enkel het type van de parameter verschillend is:

```
public static void printArray(Character[] /* Integer, String */ inputArray ){
    Stream.of(inputArray).forEach(element -> System.out.printf("%s ", element));
}
```

- Dit is op te lossen met een generieke methode waardoor meerdere datatypes in deze lijst kunnen zijn

```
public static <E> void printArray(E[] /* Integer, String */ inputArray ){
    Stream.of(inputArray).forEach(element -> System.out.printf("%s ", element));
}
```

Belangrijk

- Meerdere parameters kunnen hier hetzelfde generieke type (E) krijgen
- Om bv. een generieke maximumMethode te implementeren kunnen we gebruik maken van `Comparable`:

```
public static < T extends Comparable< T > > T maximum( T x, T y, T z ){
    T max = x;
    if ( y.compareTo( max ) > 0 )
        max = y;
    if ( z.compareTo( max ) > 0 )
        max = z;

    // of via streams
    // T max = Arrays.asList(x,y,z).stream().max(T::compareTo).get();
    return max;
}
```

2.3.2 Generieke klassen

Hier is een voorbeeld van een generieke klasse door een eigen `Stack`-implementatie te bouwen:

```
public class Stack<E>{
    private final int SIZE;
    private int top;
    private E[] elements;
```

```
    public Stack(){
        this( 10 );
    }

    public Stack( int s ){
        SIZE = s > 0 ? s : 10;
        top = -1;
        elements = (E[]) new Object[SIZE]; // creatie van de array
    }

    public void push(E pushValue) {
        if ( top == SIZE - 1 )
            throw new FullStackException( String.format("Stack is full, cannot push
                %s ", pushValue ));
        elements[ ++top ] = pushValue; // plaats op de stack
    }

    public E pop() {
        if ( top == -1 )
            throw new EmptyStackException( "Stack is empty, cannot pop" );
        return elements[ top-- ]; // verwijder en geef het top element terug
    }
}
```

2.3.3 Wildcards

Wildcards worden gebruikt om flexibiliteit te introduceren bij het werken met generics in methodes of klassen.

- **Upperbound wildcard**
  - Een **upperbound** wildcard wordt gebruikt als je wilt beperken dat een type maximaal een bepaald supertype mag zijn:

```
public static double sum(Collection<? extends Number> list) {
  return list.stream().mapToDouble(Number::doubleValue).sum();
}
```

Uitleg

- `<? extends Number>` betekent dat list elk type kan bevatten dat Number is of een subtype daarvan (bijvoorbeeld `Integer, Double`, enz.).
- Hierdoor kun je een lijst van verschillende `Number`-subtypes doorgeven en veilig werken met hun methodes.
- **LowerBound wildcard** Een lowerbound wildcard wordt gebruikt als je wilt beperken dat een type minimaal een bepaald subtype mag zijn:

```
public static void addIntegers(List<? super Integer> list) {
    list.add(10);
    list.add(20);
}
```

Uitleg

- `<? super Integer>` betekent dat `list` elk type kan bevatten dat `Integer` is of een supertype daarvan (bijvoorbeeld `Number, Object`).
- Dit is nuttig als je elementen wilt toevoegen, omdat Java anders niet weet welk type hij moet accepteren.

2.3.4 Belangrijke termen bij wildcards

- `?` **(unbounded wildcard)**:
  - Staat voor "ik weet niet wat het type is".
  - Wordt gebruikt als je een generieke methode flexibel wilt maken, zonder restricties:

```
public static void printList(List<?> list) {
  for (Object elem : list) {
      System.out.println(elem);
  }
}
```

- **Upperbound wildcard (`extends`):**
  - Gebruik `? extends T` als je gegevens **alleen wilt lezen** en **niet wilt schrijven**.
  - Dit zorgt ervoor dat je veilig toegang hebt tot de methodes van T.
- **Lowerbound wildcard (`super`)**:
  - Gebruik `? super T` als je gegevens **alleen wilt schrijven** en **niet wilt lezen**.
  - Dit zorgt ervoor dat je veilig elementen kunt toevoegen aan een collectie.

2.3.5 Andere belangrijke punten over generics

1. **Type-erasure**:
   - Tijdens compilatie vervangt Java alle generieke types door hun "raw type" (zoals `Object`), en voegt indien nodig casts toe.
   - Dit betekent dat generics puur syntactische controle tijdens compileertijd bieden.
2. **Generics en primitive types:**
   - Generics werken niet met primitive types zoals int of double. Gebruik in plaats daarvan hun wrapper-klassen (Integer, Double, enz.).
3. **Generics in method chaining:**
   - Generics maken method chaining mogelijk, zoals in builder-patronen, door hetzelfde generieke type terug te geven:

```
public class Builder<T> {
 private T value;

 public Builder<T> setValue(T value) {
     this.value = value;
     return this;
 }

 public T build() {
     return value;
 }
}
```

4. **Niet-generieke klassen kunnen erven van generieke klassen:**

Een niet-generieke klasse kan erven van een generieke klasse door een type vast te zetten:

```
public class StringStack extends Stack<String> { }
```

2.4 Observer Pattern

2.5 Netwerken

2.5.1 Inleiding

- Netwerkprogrammeren draait om communicatie tussen computers **via het internet of een lokaal netwerk**.
- Twee **communicatieprincipes**:
  - **Stream-based** communicatie (bijvoorbeeld TCP): Gegevens worden continu als een stroom verzonden.

  - **Packet-based** communicatie (bijvoorbeeld UDP): Gegevens worden in afzonderlijke pakketten verzonden.
- **Client/Server-model**:
  - **Client**: Stelt een verzoek in via een netwerk.
  - **Server**: Behandelt het verzoek en stuurt een antwoord terug.
  - Dit model volgt het **request-response-principe** (zoals webbrowsers en webservers).

2.5.2 Sockets

- **Sockets** zijn softwareobjecten die eindpunten van een netwerkverbinding vertegenwoordigen.
- **Stream Sockets** gebruiken TCP voor een **verbinding-georiënteerde service**.
- Sockets lijken op file I/O: een server of client opent een connectie, stuurt/ontvangt gegevens, en sluit de connectie.
- De server bindt zich aan een poort waarop hij luistert naar inkomende connecties.
- Geldige poortnummers: 1024 - 65535 (poorten < 1024 zijn gereserveerd voor systeemservices).

2.5.2.1 Eenvoudige Server Bouwen van een heel eenvoudige server

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class SocketServer {

    /*  Stappen van de absolute basis van een server
        1. Maak een ServerSocket in de try catch (poort 5555 of 12345)
        2. Zet alles in een while true
        3. Maak een Socket die de ServerSocket accepteert
        4. Maak de ObjectOutputStream via socket.getOutputStream() en flush het
        5. Maak de ObjectInputStream via socket.getInputStream() -- optioneel
        6. Stuur via de output een berichtje naar de client
        7. Flush opnieuw
        8. Sluit de socket
    */

    public static void main(String[] args) {
        try (ServerSocket server = new ServerSocket(5555, 10)) {
            while (true) {
                Socket socket = server.accept();

                ObjectOutputStream output = new ObjectOutputStream(socket.
                   getOutputStream());
                output.flush();

                ObjectInputStream input = new ObjectInputStream(socket.
                   getInputStream());

                output.writeObject("Hallo! Dit heb ik gestuurd vanuit de server");
                output.writeObject("Een tweede bericht");

                output.flush();
                socket.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
    }
}
```

2.5.2.2 Eenvoudige Client Bouwen van een heel eenvoudige Client

```
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetAddress;
import java.net.Socket;

public class SocketClient {

    /*  Stappen van de absolute basis van een server
        1. Maak een Socket in de try catch voeg jezelf (localhost) toe en de poort
           van de server
        4. Maak de ObjectOutputStream via socket.getOutPutStream() en flush het
        5. Maak de ObjectInputStream via socket.getInputStream()
        6. Stuur via de output een berichtje naar de client
        7. Flush opnieuw
        8. Sluit de socket
     */

    public static void main(String[] args) {
        while (true) {
            try (Socket socket = new Socket(InetAddress.getLocalHost(), 5555)) {
                System.out.println("Server is actief!");

                // Zorgt voor communicatie met de server
                ObjectOutputStream output = new ObjectOutputStream(socket.
                   getOutputStream());
                output.flush();

                // Haalt de input van de Server op
                ObjectInputStream input = new ObjectInputStream(socket.
                   getInputStream());

                // Lees bericht van de server
                String message = (String) input.readObject();
                System.out.println("Bericht ontvangen: " + message);

                break; // Nadat alles is ontvangen, sluit de server

            } catch (Exception e) {
                System.out.println("Server nog niet gestart...");
                try {
                    Thread.sleep(5000); // 5 Seconden wachten indien het ophalen
                       van de actieve server faalt
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
            }
        }
    }
}
```

2.5.2.3 Probleem met de eenvoudige server en client

- Het probleem is dat bovenstaande socket wel werkt, maar je kan maar één bericht sturen
- Met onderstaande server en client kan je er meerdere sturen

2.5.2.4 Complexere Client

```
import java.io.*;
import java.net.InetAddress;
import java.net.Socket;
import java.util.Scanner;

public class SocketClient {

    public static void main(String[] args) {
        try {
            System.out.println("Client gestart. Verbinden met server op poort
                5555...");

            // 1. Maken van de socket die verbinding maakt met de Server
            try (Socket socket = new Socket(InetAddress.getLocalHost(), 5555);

                // 2. Maak de ObjectInput- en ObjectOutputStream  via de socket.
                    getInput- en getOutputStream
                ObjectOutputStream output = new ObjectOutputStream(socket.
                    getOutputStream());
                ObjectInputStream input = new ObjectInputStream(socket.
                    getInputStream())) {

                System.out.println("Client is geconnecteerd met de server");

                // 3. Maak een scanner zodat je constant berichten kan blijven
                    sturen
                Scanner scanner = new Scanner(System.in);

                // 4. De server is actief dus je kan beginnen met berichten te
                    sturen
                while (true) {
                    System.out.print("Typ je bericht ('STOP' om te stoppen): ");
                    String message = scanner.nextLine().trim();

                    if ("STOP".equalsIgnoreCase(message)) {
                        System.out.println("Client afgesloten.");
                        break;
                    }

                    // 5. Eenmaal je bericht getypt stuur het door via de output en
                        flush
                    output.writeObject(message);
                    output.flush();

                    // 6. De server zal laten weten via de input.readObject of het
                        berichtje goed is toegekomen
                    String response = (String) input.readObject();
                    System.out.println("Reactie van server: " + response);
                }
            }
        } catch (IOException | ClassNotFoundException e) {
            System.out.println("Kan niet verbinden met de server. Zorg ervoor dat
                de server actief is.");
        }
```

```
    }
}
```

2.5.2.5 Complexere Server

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;

public class SocketServer {

    private static boolean isServerRunning = true;

    public static void main(String[] args) {
        // 1. Maken van een Thread
        Thread serverThread = new Thread(() -> {

            // 2. Maken van de serverSocket
            try (ServerSocket server = new ServerSocket(5555)) {

                System.out.println("Server gestart op poort 5555...");

                // 3. Zet alles in één grote while
                while (true) {

                    if (isServerRunning == false) {
                        Thread.sleep(1000);
                        continue;
                    }

                    // 4. Accepteer de ServerSocket
                    Socket socket = server.accept();

                    // 5. Maak de ObjectInput- en ObjectOutputStream
                    // via de socket.getInput- en getOutputStream
                    ObjectInputStream input = new ObjectInputStream(socket.
                        getInputStream());
                    ObjectOutputStream output = new ObjectOutputStream(socket.
                        getOutputStream());
                    output.flush();

                    // 6. Lees het bericht binnen van de client
                    String clientMessage = (String) input.readObject();
                    System.out.println("Ontvangen: " + '\'' + clientMessage+ '\'');

                    // 7. Stuur een bevestiging (optioneel) - flush output indien
                        je dit doet
                    output.writeObject("Server heeft je bericht ontvangen: " + '\''
                        + clientMessage+ '\'');
                    output.flush();

                    // 8. Sluit de Socket
                    socket.close();
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
```

```
        // 9. Start de server
        serverThread.start();

        @SuppressWarnings("resource")
        Scanner scanner = new Scanner(System.in);

        // 10. Bouw een scanner dat ervoor zorgt dat je de server kan stoppen en
           starten
        while (true) {
            String command = scanner.nextLine().trim();
            if ("STOPSERVER".equalsIgnoreCase(command)) {
                isServerRunning = false;
                System.out.println("Server gepauzeerd...");
            } else if ("STARTSERVER".equalsIgnoreCase(command)) {
                isServerRunning = true;
                System.out.println("Server opnieuw gestart...");
            }
        }
    }
}
```

3 Hoofdstuk 2: Test Driven Development

3.1 JUnit

Test Triven Development

- Je schrijft testcode nog voor je de echte code gaat schrijven
- Maar om rode lijntjes te vermijden is het de bedoeling dat je alsnog de **file aanmaakt** en **alle methoden** zonder uitwerking

3.1.1 Een test maken die slaagt

- Je voert de functie uit als gewoon
- Via de `assertEquals` wordt er gekeken of de waarde geldig is

```
public void Waterfles_vulDeFlesBij_deWaterFlesWordtGevuld(){
    int hoeveelheidBijTeVullen = 500;
    fles.vulBij(hoeveelheidBijTeVullen);
    Assertions.assertEquals(hoeveelheidBijTeVullen, fles.getHoeveelheid());
}
```

3.1.2 Een test maken die hoort te falen

- Deze test bekijkt of de juiste exception wordt geworpen

```
public void Waterfles_vulDeFlesBij_vultDeFlesMetEenNegatieveWaarde_WerptException()
    {
    int hoeveelheidBijTeVullen = -50;
    Assertions.assertThrows(IllegalArgumentException.class, () ->
            fles.vulBij(hoeveelheidBijTeVullen));
}
```

3.1.3 @ParameterizedTest

- Om testen te vermijden met bijna exact dezelfde code gebruiken **ParameterizedTest**
- Zo kunnen we verschillende waarden testen in dezelfde test zonder hem meerdere malen te testen #### ValueSource
- Een valuesource zijn allemaal waarden die 1 zelfde test zullen doorgaan

```
@ParameterizedTest
@ValueSource(ints = {1, 2, 3, 4, 5})
```

- ints kan ook de waarde hebben als:
  - ints: longs: doubles:shorts: bytes: chars: strings

3.1.3.1 @CsvSource

- Dit wordt gebruikt voor het leveren van meerdere invoerwaarden aan een parameterized test in CSV-formaat (Comma-Separated Values).

```
@ParameterizedTest
    @CsvSource({
        "add, 1, 2, 3",
        "subtract, 5, 3, 2",
        "multiply, 2, 3, 6"
    })
```

```
    void testWithMultipleParams(String operation, int a, int b, int expectedResult)
        {
        int result = 0;
        switch (operation) {
            case "add": result = a + b; break;
            case "subtract": result = a - b; break;
            case "multiply": result = a * b; break;
        }
        assertEquals(expectedResult, result);  // Controleert of het resultaat
           correct is
    }
```

3.1.3.2 @EnumSource

- Wordt gebruikt om waarden van een enum door te geven aan een parameterized test.

```
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

@ParameterizedTest
@EnumSource(Day.class)
```

3.1.3.3 @MethodSource

- Maakt het mogelijk om waarden dynamisch te genereren via een methode. De methode moet een Stream, Iterable, of array van waarden retourneren.

```
static Stream<String> stringProvider() {
        return Stream.of("JUnit", "MethodSource", "Parameterized");
}
```

of

```
    private static Stream<Arguments> juisteAangiftes() {
        return Stream.of(
            Arguments.of(Boekhouding.BELASTINGSCHAAL_1, new double[] {100.00,
               200.25}),
            Arguments.of(Boekhouding.BELASTINGSCHAAL_0, new double[] {100.00,
               50.25}),
            Arguments.of(Boekhouding.BELASTINGSCHAAL_0, new double[] {0.0, 0.0}),
            Arguments.of(Boekhouding.BELASTINGSCHAAL_0, new double[] {199.99, 0.0})
               ,
            Arguments.of(Boekhouding.BELASTINGSCHAAL_0, new double[] {100.00,
               99.99}),
            Arguments.of(Boekhouding.BELASTINGSCHAAL_1, new double[] {100.00,
               100.00}),
            Arguments.of(Boekhouding.BELASTINGSCHAAL_1, new double[] {0.0, 200.00})
               ,
            Arguments.of(Boekhouding.BELASTINGSCHAAL_1, new double[] {150.00,
               200.25})
        );
    }
@ParameterizedTest
@MethodSource("stringProvider")
```

3.1.3.4 @NullAndEmptySource

- Combineert de functionaliteit van @NullSource en @EmptySource. Dit betekent dat je een test kunt uitvoeren met zowel null als een lege string (of lege collecties).

```
@ParameterizedTest
@NullAndEmptySource
void testWithNullAndEmptySource(String input) {
    assertTrue(input == null || input.isEmpty());
}
```

3.1.3.5 @NullSource

- Levert de waarde null aan een parameterized test. Dit is handig om te testen hoe methoden reageren op null input.

```
@ParameterizedTest
@NullSource
void testWithNullSource(String input) {
    assertNull(input);  // De input moet null zijn
}
```

3.1.3.6 @EmptySource

- levert een lege waarde aan de test, zoals een lege string "", een lege collectie, of een lege array.

```
@ParameterizedTest
@EmptySource
void testWithEmptySource(String input) {
    assertTrue(input.isEmpty());  // Controleert of de invoer leeg is
}
```

3.2 Mockito Framework

- Mockito is een Java-framework dat wordt gebruikt om **mock-objecten** te maken, waarmee we bepaalde onderdelen van de code kunnen testen zonder afhankelijk te zijn van de werkelijke implementatie van andere klassen.
- Dit is handig voor unit testing.

Integreren van Mocktio Om Mockito te gebruiken in combinatie met JUnit 5, moet je de `@ExtendWith(MockitoExtension.class)`-annotatie toevoegen aan je testklasse. Deze annotatie zorgt ervoor dat Mockito de annotaties zoals `@Mock` en `@InjectMocks` kan verwerken.

```
@ExtendWith(MockitoExtension.class)
class BoekhoudingTest {

}
```

3.2.1 Mock-objecten maken

Mocks zijn gesimuleerde objecten die specifiek gedrag kunnen nabootsen. Met Mockito kun je aangeven hoe een mock-object zich moet gedragen in bepaalde situaties.

Mock-object Met `@Mock` maak je een gesimuleerd (mock) object van een klasse die in je code wordt gebruikt. Dit zorgt ervoor dat je afhankelijkheden niet echt hoeft te implementeren.

- InjectsMock dient om de klasse te mocken dat een associatie heeft met FactuurMap

```
    @Mock
    private FactuurMap factuurMap;
```

InjectMocks Met `@InjectMocks` kun je aangeven dat een instantie van de klasse die je test (in dit geval Boekhouding) automatisch wordt geïnjecteerd met de gemockte afhankelijkheden. Dit is handig als je klasse afhankelijk is van andere objecten.

```
    @InjectMocks
    private Boekhouding b;
```

- Mock trainen zonder lenient: we krijgen een extra controle: de methode van de mock moet minstens één keer opgeroepen worden.
- Mock trainen met lenient: de extra controle ongedaan maken

3.2.2 Mock-object trainen

Mocken betekent dat je bepaalt hoe een mock zich gedraagt wanneer een bepaalde methode wordt aangeroepen. Dit doe je met `Mockito.when()`. Je kunt bijvoorbeeld een mock trainen om een specifieke waarde terug te geven

```
Mockito.when(factuurMap.geefBedragen(CORRECT_ONDERNEMINGSNUMMER)).thenReturn(new
    double[] {100.00, 200.00});
```

Lenient mocks

- Standaard controleert Mockito of de getrainde methode (`when()`) ook daadwerkelijk wordt aangeroepen. Als dat niet gebeurt, krijg je een foutmelding.
- Met `Mockito.lenient().when()` schakel je deze controle uit. Dit is handig als een methode mogelijk niet altijd wordt aangeroepen tijdens de test

3.2.3 Verifiëren van interacties dat Mockito heeft uitgevoerd

Je kunt met Mockito controleren of een bepaalde methode van een mock-object correct is aangeroepen, en hoe vaak dat is gebeurd. Dit doe je met `Mockito.verify()`:

```
Mockito.verify(factuurMap, Mockito.times(1)).geefBedragen(
    CORRECT_ONDERNEMINGSNUMMER);
```

- `times(1)`: Controleert dat de methode precies één keer is aangeroepen.
- Andere opties:
  - `never()`: Controleert dat de methode nooit is aangeroepen.
  - `atLeast(n)`: Controleert dat de methode minstens n keer is aangeroepen.

3.2.4 Testdate genereren

Met `@MethodSource` kun je testdata leveren vanuit een statische methode in dezelfde klasse of een andere klasse. Dit is handig voor het testen van meerdere scenario's in één testmethode.

```
private static Stream<Arguments> juisteAangiftes() {
    return Stream.of(
        Arguments.of(Boekhouding.BELASTINGSCHAAL_1, new double[] {100.00, 200.25}),
        Arguments.of(Boekhouding.BELASTINGSCHAAL_0, new double[] {100.00, 50.25})
    );
}
```

De `@MethodSource-annotatie` wijst naar de methode die de testdata levert.

```
@ParameterizedTest
@MethodSource("juisteAangiftes")
void testGoedeGevallenMetJuisteAangifte(double schaal, double[] waardes) {
    Mockito.when(factuurMap.geefBedragen(CORRECT_ONDERNEMINGSNUMMER)).thenReturn(
        waardes);

    Aangifte expected = new Aangifte(schaal, waardes);
    Aangifte real = b.genereerAangifte(CORRECT_ONDERNEMINGSNUMMER);

    assertEquals(expected.belastingSchaal(), real.belastingSchaal());
    assertArrayEquals(expected.bedragen(), real.bedragen());

    Mockito.verify(factuurMap, Mockito.times(1)).geefBedragen(
        CORRECT_ONDERNEMINGSNUMMER);
}
```

3.2.5 Volledige testbeschrijving

Test voor goede aangiftes:

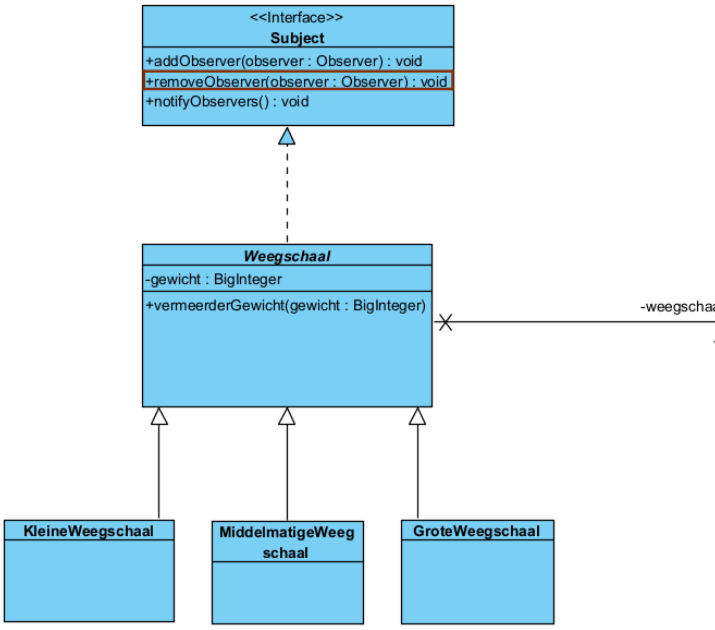
- Mocked gedrag: Wanneer `factuurMap.geefBedragen()` wordt aangeroepen met een correct ondernemingsnummer, retourneert de mock de opgegeven bedragen.
- Asserties:
  - Controleer dat de gegenereerde `Aangifte` hetzelfde is als verwacht.
  - Verifieer dat de mock correct is aangeroepen. **Test voor foutieve ondernemingsnummers**: Je kunt ook testen of je klasse correct omgaat met foutieve input, bijvoorbeeld door een uitzondering te gooien:

```
@ParameterizedTest
@NullAndEmptySource
@ValueSource(strings = {"     ", "1234567890", "FR1234567890"})
void testSlechteOndernemingsNummers_werpException(String ondernemingsnummer) {
    Mockito.lenient().when(factuurMap.geefBedragen(ondernemingsnummer)).thenReturn(
        new double[] {0.0, 0.0});

    assertThrows(IllegalArgumentException.class, () -> b.genereerAangifte(
        ondernemingsnummer));

    Mockito.verify(factuurMap, Mockito.times(1)).geefBedragen(ondernemingsnummer);
}
```

4 Hoofdstuk 3: Design Patterns

4.1 Voorbereiding

- Voor we kunnen starten aan design patterns moeten we ook met VP kunnen werken
- Voornamelijk de manier hoe we met pijlen moeten werken



- Dit diagram is enkel ter illustratie en dit bestaat dus ook niet
- **Associatie**: een relatie die vertrekt van de **klasse** naar de klasse toe die de eerste **klasse** moet hebben als attribuut
- **Generalisatie**: een releatie die vertrekt van de **te-erven-klasse** die naar **ervende klasse** gaat
- **Realisatie**: een releatie die vertrekt **van de klasse** naar de **interface** die de klasse moet implementeren

4.2 Strategy Pattern

Wat is dit Design Pattern?

- Wanneer objecten met overerving (Eend -> RubberenEend, HoutenEend …) bepaalde eigenschappen hebben, hoeft niet iedereen exact dezelfde waarden van die eigenschappen over te erven.
- Een normale eend kan bv. vliegen en kwaken terwijl een rubbereneend beide niet kan.
- Dit probleem kunnen we dus oplossen met het gebruik van interfaces.

Ontwerpprincipes:

- Bepaal de aspecten van je applicatie die variëren en scheid deze van de aspecten die hetzelfde blijven
- Programmeer naar een interface, niet naar een implementatie.
  - M.a.w. we gaan voor ieder gedrag een interface gebruiken, bv. FlyBehavior en QuackBehavior, en iedere implementatie van een gedrag implementeert één van deze interfaces.

Implementatie

- Hier hebben onze eenden twee over te erven attributen
  - Vlieg
  - Kwaak
- We maken dus twee interfaces voor deze attributen:

```
public interface VliegGedrag {
    public String vlieg();
}
```

- Eenmaal de hoofdinterface is gemaakt, maken we nog een maken we nog normale klassen aan die de effectieve waarden zullen bevatten van de atrributen

- Voor VliegGedrag is dat dan:
  - VliegWel
  - VliegNiet

```
public class VliegWel implements VliegGedrag{

  @Override
  public void vlieg() {
      Systen.out.print("Ik kan vliegen");
  }
}
```

- Eenmaal we al onze attribuutklassen hebben kunnen we de domeinklasse ervoor aanmaken:

```
public abstract class Eend {
    private QuackGedrag kwaakGedrag;
    private VliegGedrag vliegGedrag;
    private SoortEend soortEend;

    public Eend(QuackGedrag kwaakGedrag, VliegGedrag vliegGedrag, SoortEend
        soortEend) {
        this.kwaakGedrag = kwaakGedrag;
        this.vliegGedrag = vliegGedrag;
        this.soortEend = soortEend;
    }

    public void kwaak() {
        kwaakGedrag.kwaak();
    }

    public void vlieg() {
        vliegGedrag.vlieg();
    }
}
```

- We kunnen de applicatie starten met volgende code:

```
public void overloopEenden() {
    for (Eend eend : eenden) {
        eend.kwaak();
        System.out.println();
       eend.vlieg();
        System.out.println();
        eend.getSoortEend();
        System.out.println();
    }
}
```

Dit is een algemeen schema voor hoe State eruit ziet:

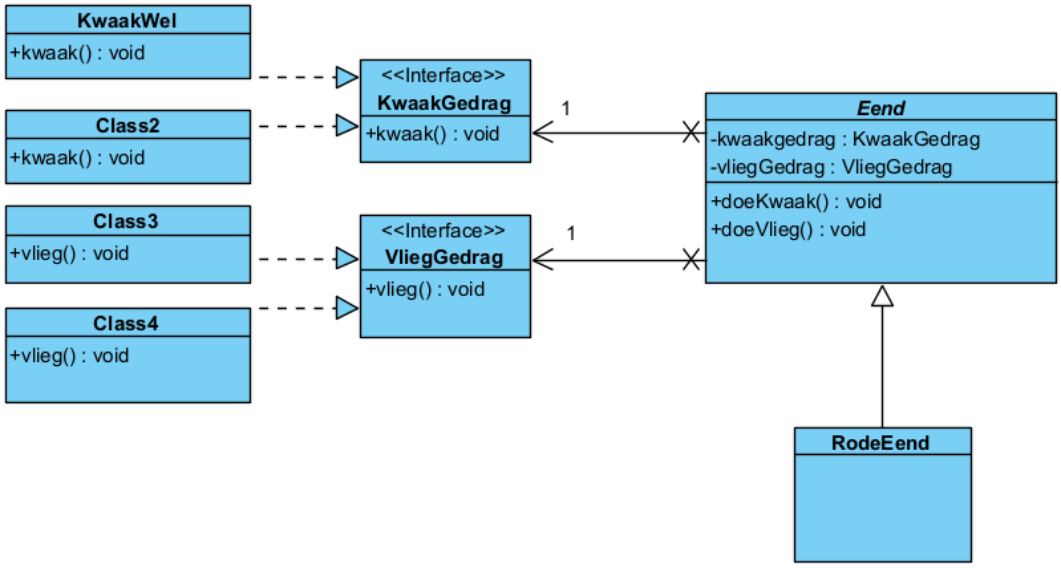


Figure 2: Strategy schema

4.3 Simple Factory

Wat is dit Design Pattern?

- In dit design pattern handelen we keuzes af
- Wanneer we bv. keuzemenuutjes moeten aanpassen is het de bedoeling dat we van dat menu een Factory-klasse maken
- In die factory is de default-keuze ook automatisch een klasse

Implementatie

- Stel we hebben een shop waar mensen moeten kiezen uit een menu en dit menu kan veranderen dan hebben we dit design pattern nodig
- We beginnen met het maken van onze pizza-object

```
public abstract class Pizza {

    public void prepare() {
        System.out.println("Preparing " + getClass().getSimpleName());
    }

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

}
```

- Waarbij we ook een paar soorten pizza's zullen aanmaken:
- Deze kan leeg zijn vanwege de manier hoe ik Pizza heb opgebouwd

```
public class KaasPizza extends Pizza{

}
```

- Daarna implementeren we onze shop

```
public class PizzaStore {
    private PizzaFactory factory = new PizzaFactory();
```

```
    public Pizza bestelPizza(String teBestellenPizza) {
        Pizza pizza;

        pizza = factory.orderPizza(teBestellenPizza);

        pizza.prepare();
        pizza.deliver();

        return pizza;
    }
}
```

- Het bestellen van de pizza (via dat menuutje) gebeurt dan via de PizzaFactory

```
public class PizzaFactory {
    public Pizza orderPizza(String typePizza) {
        Pizza pizza = null;

        switch (typePizza) {
            case "barbeque":
                pizza = new BarbequePizza();
                break;
            case "kaas":
                pizza = new KaasPizza();
                break;
            case "margaritta":
                pizza = new Margaritta();
                break;
            default:
                pizza = new NonExistentPizza(typePizza);
        }

        return pizza;
    }
}
```

- Het is ook de bedoeling dat als de opgegeven keuze (hier een pizza) niet bestaat dat we een object maken voor dit soort resultaten

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

    public NonExistentPizza(String opgegvenPizza) {
        this.opgegvenPizza = opgegvenPizza;
    }
    @Override
    public void prepare() {
        System.out.println(opgegvenPizza + " does not exist in our store");
    }

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

- Laten we wat pizza's bestellen!

```
public class PizzaApplicatie {
    private PizzaStore pizzaStore;

    public PizzaApplicatie() {
        pizzaStore = new PizzaStore();
        this.initialiseerMethodes();
    }

    private void initialiseerMethodes() {
        this.bestelPizzas();
    }

    private void bestelPizzas() {
        // pizzaStore.bestelPizza("barbeque");
        pizzaStore.bestelPizza("kaas");
        pizzaStore.bestelPizza("nietBestaandePizza");

    }
}
```

Dit is een algemeen schema voor hoe Factory eruit ziet:

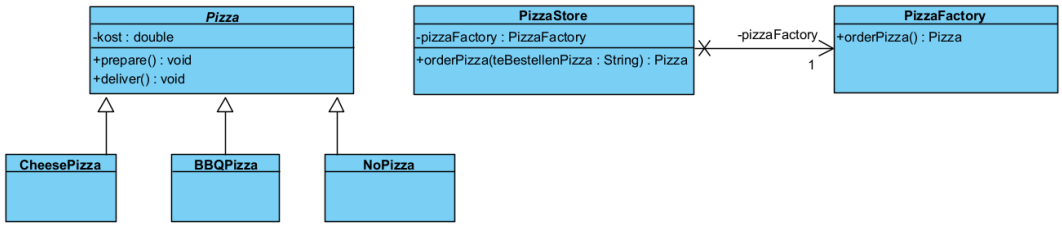


Figure 3: Factory

4.4 Observer Pattern

Wat is dit Design Pattern?

- Dit pattern streeft ernaar om eenvoudig en dynamisch tekst op schermen te laten verschijnen en ze te laten updaten zonder problemen
- Streef naar ontwerpen met een zwakke koppeling tussen de objecten die samenwerken..

Implementatie

- Maak twee interfaces:
- Subject

```
public interface Subject {
    public void addObserver(Observer observer);
    public void removeObserver(Observer observer);
}
```

- Observer

```
public interface Observer {
    /* variabelen die moeten geupdatetet worden  */
    public void update(double temperatuur, double luchtdruk);
}
```

- De klasse die hier **Subject** implementeert krijgt hier volgende attributen:

```
private Set<Observer> observers;  // Lijst van observers
    private double temperatuur;
    private double luchtdruk;

    public WeerStation() {
        observers = new HashSet<>();
    }

    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
       observers.forEach(observer -> observer.update(temperatuur, luchtdruk));
    }

    public void setMetingen(double temperatuur, double luchtdruk) {
        this.temperatuur = temperatuur;
        this.luchtdruk = luchtdruk;
        notifyObservers();  // Laat alle observers weten dat er nieuwe data is
    }
```

- De observerklasse ziet er dan ongeveer zo uit

```
public class VolledigScherm implements Observer {
    private double temperatuur;
    private double luchtdruk;

    @Override
    public void update(double temperatuur, double luchtdruk) {
        this.temperatuur = temperatuur;
        this.luchtdruk = luchtdruk;
        toon();
    }

    public void toon() {
        System.out.println("VolledigScherm: De huidige temperatuur is " +
           temperatuur + "°C en de luchtdruk is " + luchtdruk + " hPa");
    }
}
```

- Als je de applicatie dan runt zal het ongeveer zo moeten uitzien om alle klassen in orde te brengen

```
    public static void main(String[] args) {
        // Maak het weerstation aan
        WeerStation weerStation = new WeerStation();

        // Maak de observers aan
        TemperatuurScherm temperatuurScherm = new TemperatuurScherm();
```

```
        VolledigScherm volledigScherm = new VolledigScherm();

        // Koppel de observers aan het weerstation
        weerStation.addObserver(temperatuurScherm);
        weerStation.addObserver(volledigScherm);

        // Update de metingen
        System.out.println("Nieuwe metingen:");
        weerStation.setMetingen(22.5, 1013.1);

        System.out.println("\nNog een update:");
        weerStation.setMetingen(24.0, 1009.8);

        // Verwijder een observer en update opnieuw
        System.out.println("\nVerwijder TemperatuurScherm en update:");
        weerStation.removeObserver(temperatuurScherm);
        weerStation.setMetingen(19.5, 1012.3);
    }
```

Dit is een algemeen schema voor hoe Observer eruit ziet:

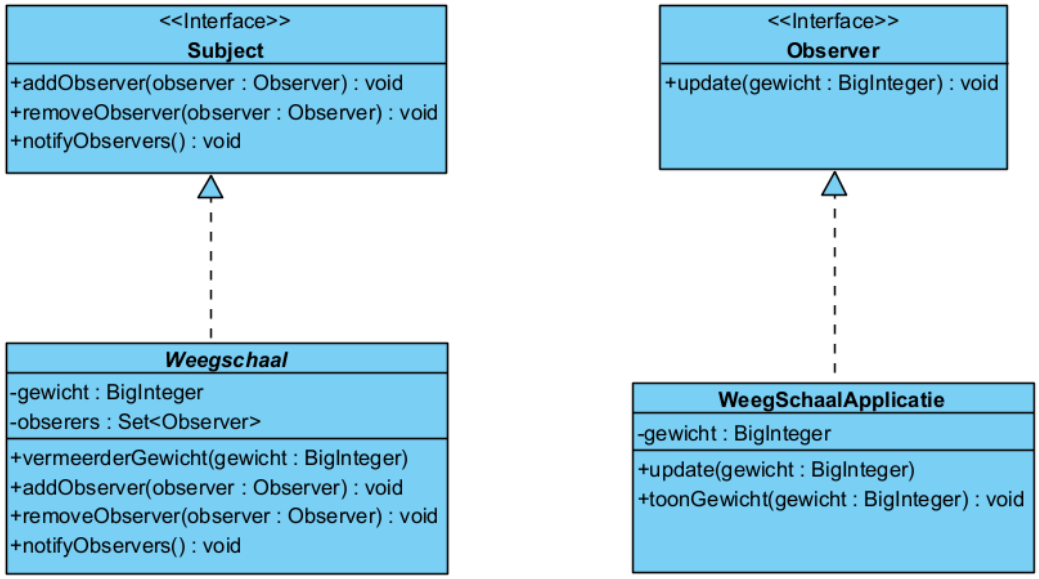


Figure 4: Observer

4.5 Decorator Pattern

Wat is dit Design Pattern?

- Het doel is om klassen eenvoudig te kunnen uitbreieden om nieuw gedrag te incorporeren zonder de bestaande code te wijzigen
- Dit gaan we doen door allerlei booleans toe te voegen aan onze hoofdklasse waarvan iedereen gaat erven

Implementatie

- We beginnen met het maken van onze abstracte hoofdklasse
- We maken een beschrijving die we later nog gaan aanvullen met de echte waarde ervan en maken hier getters en setters voor

```
public abstract class Drank {
    protected String description = "Unknown Beverage";
```

```
    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public abstract double cost();
}
```

- Dan beginnen we met de overerving en maken we drankjes aan voor onze koffieshop

```
public class Espresso extends Drank {
    public Espresso() {
        setDescription("Espresso");
    }

    @Override
    public double cost() {
        return 1.99;
    }
}
```

- Daarna maken we een klasse dat hier de 'toppings' beheert, hier is dit dan onze Decorator
- Dit is een abstracte klasse dat een variabele bevat die we gaan 'decoreren'

```
public abstract class DrankDecorator extends Drank {
    protected Drank drank;

    public DrankDecorator(Drank drank) {
        this.drank = drank;
    }

    @Override
    public abstract String getDescription();
}
```

- Nu dit is gemaakt kunnen we een toppings/decorators maken

```
public class Melk extends DrankDecorator{

    public Melk(Drank drank) {
        super(drank);
    }

    @Override
    public String getDescription() {
        return drank.getDescription() + " " + getClass().getSimpleName().
            toLowerCase();
    }

    @Override
    public double cost() {
        return drank.cost() + 0.10;
    }
```

```
}
```

- Voila onze decorator is klaar en nu kunnen we drankjes bestellen

```
    private void bestelKoffies() {
         drank = new Espresso();
         System.out.println(drank.cost() + " -> " + drank.getDescription());
         System.out.println();

         drank = new Melk(drank);
         System.out.println(drank.cost() + " -> " + drank.getDescription());
         System.out.println();

         drank2 = new Karamel(new Melk(drank2)) ;
         System.out.println(drank2.cost() + " -> " + drank2.getDescription());
         System.out.println();

    }
```

Dit is een algemeen schema voor hoe Decorator eruit ziet:

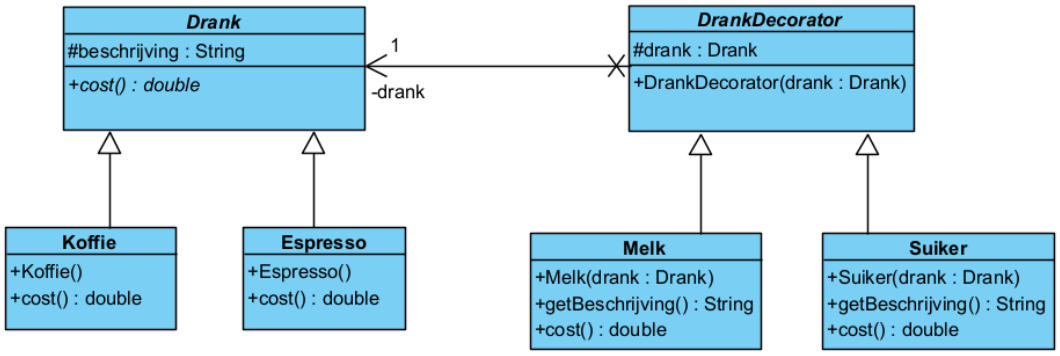


Figure 5: Decorator

4.6 State Pattern

Wat is dit Design Pattern?

Implementatie

- Begin met het maken van al je mogelijk staten in één interface
- Bij alle methodes moet je hier goed nadenken “in welke status kan mijn nachtlampje allemaal zijn?”
- Je begint met alle methodes wat een koffiemachine kan doen en implementeert zo statussen ervan
  - De lamp kan aan staan
  - De lamp kan uitstaan
  - De lamp kan ook kapot gaan
- Nu moet je nadenken, hoe kan ik die staten bereiken?
  - methode zetAan
  - methode zetUit
  - De state ‘kapot’ wordt bereikt via een if-statement in NachtLamp

```
public abstract class NachtLampState {

    protected final NachtLamp lamp;

    public String aan() {
        return "default invulling (aan)";
```

```
    };

    public String uit() {
        return "default invulling (uit)";
    }

    public String kapot() {
        return "default invulling (kapot)";
    }

    @Override
    public String toString() {
        return "default invulling";
    }
}
```

Nu is het de bedoeling dat je al deze klassen gaat aanmaken en het gedrag implemnteert

```
public class LampAan extends NachtLampState{

    public LampAan(NachtLamp lamp) {
        super(lamp);
    }

    @Override
    public String uit() {
        lamp.setCurrentState(new LampUit(lamp));
        return "het lampje word uitgezet, je bespaart goede stroom hiermee, huidige
            toestand: " + lamp.getElektriciteit();
    }

    @Override
    public String kapot() {
        lamp.setCurrentState(new LampKapot(lamp));
        return "";
    }

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

```
public class LampKapot extends NachtLampState{

    public LampKapot(NachtLamp lamp) {
        super(lamp);
    }

    @Override
    public String aan() {
        return "de lamp kan niet aan want hij is kapot";
    }

    @Override
    public String uit() {
        return "de lamp kan niet uit want hij is kapot";
```

```
    }

    @Override
    public String kapot() {
        return "De lamp is al kapot.";
    }


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

```
public class LampUit extends NachtLampState {

    public LampUit(NachtLamp lamp) {
        super(lamp);
    }

    @Override
    public String aan() {
        lamp.setElektriciteit(lamp.getElektriciteit() - 1);
        lamp.setCurrentState(new LampAan(lamp));
        return "Het nachtlampje staat aan, elektriciteit verminderd met 1. Huidig
            niveau: " + lamp.getElektriciteit();
    }

    @Override
    public String kapot() {
        lamp.setCurrentState(new LampKapot(lamp));
        return "De lamp is kapot gegaan.";
    }

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

Na alle states kunnen we onze hoofdklasse implementeren die deze statussen zal kunnen bevatten

```
public class NachtLamp {
    @Getter @Setter private NachtLampState currentState;

    @Getter @Setter private int elektriciteit;

    public NachtLamp(int elektriciteit) {
        setElektriciteit(elektriciteit);
        if (elektriciteit >=10) {
            setCurrentState(new LampAan(this));
        } else {
            setCurrentState(new LampKapot(this));
        }
    }

    public void zetLampAan() {
```

```
        currentState.aan();
    }

    public void zetLampUit() {
        currentState.uit();
    }

    public void zetLampOpKapot() {
        currentState.kapot();
    }

    @Override
    public String toString() {
        return String.format("%s met state %s met elektriciteitsniveau %d",
           getClass().getSimpleName(), this.currentState, this.elektriciteit);
    }
}
```

Nu hebben we een volledig functionerend nachtlampje en kunnen we ermee spelen:

```
public class NachtLampApplicatie {

    private NachtLamp lamp;

    public NachtLampApplicatie() {
        this.lamp = new NachtLamp(11);
        System.out.println(lamp);

        lamp.zetLampAan();
        System.out.println(lamp);

        lamp.zetLampUit();
        System.out.println(lamp);

        lamp.zetLampOpKapot();
        System.out.println(lamp);
    }
}
```

Dit is een algemeen schema voor hoe State eruit ziet:

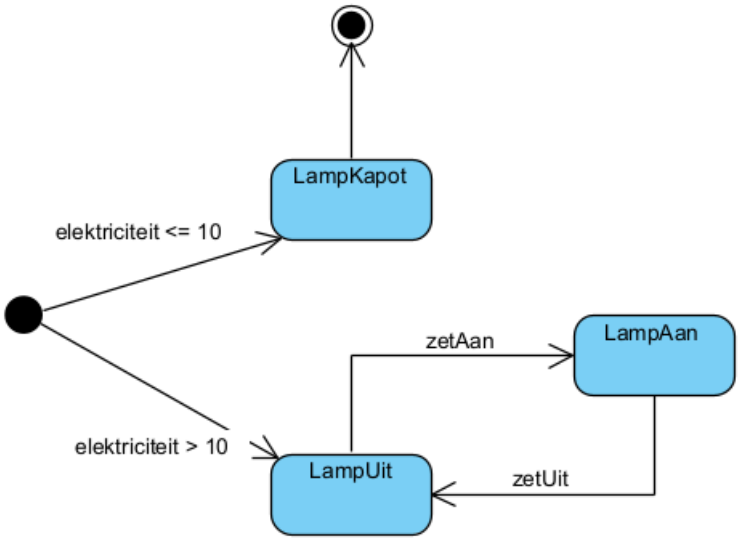


Figure 6: State

- De blauwe kaders zijn de states (klassen)
- De woorden die zweven zijn de methodes die kunnen uitgevoerd worden

4.7 facade Pattern

Wat is dit Design Pattern?

- Dit werkt zoals de DomeinController dat we hebben gezien in jaar één
- We houden als onze klassen bij in één grote klasse
- Praat alleen met je directe vrienden. Hoe minder je weet hoe beter.

Implementatie

- We beginnen met ons object

```
public class Game {
    private String naam;
    private double prijs;
    private int rating;

    public Game(String naam, double prijs, int rating) {
        setNaam(naam);
        setPrijs(prijs);
        setRating(rating);;
    }

    public String getNaam() {
        return naam;
    }

    private void setNaam(String naam) {
        this.naam = naam;
    }

    public double getPrijs() {
        return prijs;
    }

    private void setPrijs(double prijs) {
```

```
        this.prijs = prijs;
    }

    public int getRating() {
        return rating;
    }

    private void setRating(int rating) {
        this.rating = rating;
    }

    public double verhoogPrijs(double percentage) {
        return (this.prijs * (percentage * 100)) / 100;
    }

}
```

- Gevolgt door onze facade

```
public class GameFacade {
    private List<Game> games;

    public GameFacade() {
        this.games = new ArrayList<>();
        this.maakGames();
    }

    public void maakGames() {
        games.add(new Game("Elden Ring", 69.99, 4));
        games.add(new Game("Call of Duty Black Ops 6", 79.99, 4));
        games.add(new Game("Horizon Forbidden West", 59.99, 5));

    }

    public List<Game> geefGames(){
        return this.games;
    }

    public void verhoogPrijs(double percentage) {
        games.forEach(game -> game.verhoogPrijs(percentage));
    }
}
```

- Uiteindelijk kunnen we ze tonen in de applicatie

```
public class GameApplicatie {

    private GameFacade gameFacade;

    public GameApplicatie(GameFacade gameFacade) {
        this.gameFacade = gameFacade;
        this.toonGamesInclusiefInflatie();
    }

    public void geefAlleGames() {
        for (Game game : gameFacade.geefGames()) {
            System.out.printf("Naam: %s - Prijs: %.2f - Rating - %d%n", game.
                getNaam(), game.getPrijs(), game.getRating());
```

```
        }
    }

    public void toonGamesInclusiefInflatie() {
        this.geefAlleGames();
        gameFacade.verhoogPrijs(10.00);
        System.out.println();
        this.geefAlleGames();

    }
}
```

4.8 MVC (Geen Design Pattern maar zit alsnog in dit hoofdstuk)

Wat is dit Design Pattern? Dit kun je vergelijken met het vak front-end development Je steekt alle **domeinlogica** apart en alle **UI-onderdelen** ook

Implementatie Het model bevat alle domeinlogica

```
/**
 * Backend van het rekenmachine
 */
public class Model {

    private double resultaat;

    public Model() {

    }

    public void telOp(double getal1, double getal2) {
        this.resultaat = getal1 + getal2;
    }

    public double getResultaat() {
        return this.resultaat;
    }
}
```

De View zorgt voor de UI-onderdelen

```
/**
 * Frontend van het rekenmachine
 */
public class View {

    private final static Scanner invoer = new Scanner(System.in);

    public View() {

    }

    public int vraagNummer(String prompt) {
        System.out.print(prompt);
        return invoer.nextInt();
    }

    public void GeefResultaat(double result) {
```

```
        System.out.println("Result: " + result);
    }
}
```

De Controller verbind het model en de view:

```
/**
 * Verbind het model (backend) en de view (frontend)
 */
public class Controller {
    private Model model;
    private View view;

    public Controller(Model model, View view) {
        this.model = model;
        this.view = view;
    }

    public void startApp() {
        double getal1 = view.vraagNummer("Geef getal één: ");
        double getal2 = view.vraagNummer("Vraag getal twee: ");

        model.telOp(getal1, getal2);
        double resultaat = model.getResultaat();
        view.GeefResultaat(resultaat);
    }
}
```

We kunnen nu de applicatie runnen door simpelweg `startApp()` aan te roepen