Samenvatting DEA

Julian van Kuijk

December 24, 2020

Contents

1 Herhaling OOPD

DEA is een vervolgvak op oopd en het object georienteerd programeren. Hierdoor is het belangrijk dat je weet hoe je OO programeerd. Daarom wordt in dit hoofdstuk OO herhaald. Voor het gemak doen we dit in Java.
Object georienteerd programeren werkt met *objecten*. Objecten zijn delen van een programma met hun eigen functie. Deze objecten staan in *Klasse*. Deze klasse werken samen om een werkend programma te maken. Ook zijn er nog andere begrippen(piliaren):

- Overerving
- Encapsulatie
- Polymorfie

1.1 Klasse

In java is elke klasse een .java document. In een java klasse staat de Package waarin het bestand staat, welke bestanden je import en de inhoud van de klasse zelf.

```
package App;

import java.util.ArrayList;

public class Karakter {
   private int levens;
   private String naam;

   public Karakter(String naam){
      this.naam = naam;
      this.levens = 100;
   }
}
```

De klasse begint met public class Karakter. public is een keyword in het OO, dit bepaalt hoe de initialisatie die volgt kan communiceren met andere klasse. Vervolgens worden de variabele gedeclareerd, hier zie je private bij staan. public betekend dat een variabele met andere klasse mag communiceren, private betekent dat het niet met andere klasse mag comuniceren. Als laatste staat er public Karakter(String naam). dit is de constructor. Die wordt aangeroepen als een klasse wordt aangemaak. Eventuele parameters worden dan gebruikt. this.naam zorgt er voor dat de variabele goed wordt geplaatst in het geheugen. Hier zie je hoe de constructor wordt aangeroepen bij het maken van een object

```
Karakter karakter = new Karakter("Speler");
```

1.2 Polymorfie

Polymorfie is de gelijkvormigheid tussen klasse en objecten met een verschillende implementatie. Een interface is hier bijvoorbeeld handig voor.

1.2.1 Overerving

Overerving lost de kleine verschillen tussen 2 klassen op. In een klas zitten leerlingen en docenten. Maar het zijn allemaal personen.

1.3 Encapsulatie

Encapsulatie zorgt voor flexibele systemen. Klassen hoefen niet van elkaar te weten hoe ze werken. Als ze de informatie maar goed krijgen. Dit wordt bv gedaan door getters en setters.

1.4 Static

functies en variabelen kunnen worden verklaard als static. Als iets static is behoort het tot de klasse, niet tot het object.

1.4.1 Static variabelen

Door in een klasse een static int te maken en in de constructor die te verhogen, wordt iederekeer als een Counter wordt gemaakt de static var verhoogd.

```
public class Counter {
   public static int COUNT = 0;
   Counter() {
      COUNT++;
   }
}
```

Als je dan in een andere klasse maak je 2 Counters aan:

```
public class MyClass {
   public static void main(String[] args) {
      Counter c1 = new Counter();
      Counter c2 = new Counter();
      System.out.println(Counter.COUNT);
   }
}
// Outputs "2"
```

1.4.2 Static methoden

Methoden kunnen ook static zijn. Dan hoef je weer niet een object te maken om een functie uit te voeren. Bij static methoden zijn wat meer resricties dan

bij de variabelen. Ze kunnen niet non-static variabelen gebruiken. Ook kunnen ze niet this en super gebruiken.

```
public class Counter {
   public static int COUNT = 0;
   Counter() {
      COUNT++;
   }

   public static void increment(){
      COUNT++;
   }
}
```

Vervolgens gebruik je de increment functie.

```
public class MyClass {
   public static void main(String[] args) {
      Counter.increment();
      Counter.increment();
      System.out.println(Counter.COUNT);
   }
}
// Outputs "2"
```

Voor extra dea opgaven

2 Maven

Maven is een build programma dat samen werkt met Java. Het wordt gebruikt voor het vervangen van de *IDE's* zoals Eclipse en Intelij. Hierdoor heb je meer vrijheid in het programmeren. Maven maakt gebruik van Archetypes en van de pom.xml file. Je begint een Maven in een IDE of in de command prompt: *mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false*

2.1 Archetypes

Archetypes zijn een soort templates waarvan Maven projects worden gemaakt. Door zelf een custom archetype te maken hoef je minder werk te doen met de folder instellen. *Maven Introduction* kan helpen met je eigen archetype maken, en extra verduidelijking van archetypes

2.2 pom.xml

De pom.xml file is de file die het gedrag van maven bepaalt. hier kun je versies in veranderen en plugins/dependecies toevoegen.

Je begint het document met je properties. Meestal staat dit al goed ingesteld, eventueel kun hier versies aanpassen.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
       http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>
<artifactId>app-name</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>

<properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <maven.compiler.source>12</maven.compiler.source>
   <maven.compiler.target>12</maven.compiler.target>
</properties>
```

Vervolgens regel je de build variabelen. Hierin staan je plugins, sommige plugins hebben een configuration.

```
<build>
   <plugins>
      <plugin>
         <artifactId>maven-surefire-plugin</artifactId>
         <version>2.22.1</version>
      </plugin>
      <plugin>
         <artifactId>maven-war-plugin</artifactId>
         <version>2.4</version>
         <configuration>
            <failOnMissingWebXml>false</failOnMissingWebXml>
         </configuration>
      </plugin>
   </plugins>
</build>
```

Als laatste heb je de dependecies:

```
<dependencies>
   <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.4.0</version>
      <scope>test</scope>
   </dependency>
```

```
    <dependency>
      <groupId>javax</groupId>
      <artifactId>javaee-api</artifactId>
      <version>8.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
```

Een goede site voor Maven dependecies en plugins is *mvnrepository*
Een alternatief voor Maven zou Gradle zijn.

3 JUnit

In samenwerking met *Maven* kun je *JUnit* gebruiken. JUnit is een tool die wordt gebruikt bij het testdriven development.

3.1 Unit

Een belangrijk begrip voor het schrijven van test's zijn Unit's. Een unit is een klein deel van je programma, bijvoorbeeld een functie. Door de Unit's te testen zorg je dat een onderdeel het doet. Zo kun je sneller debuggen en problemen vinden. Als je een onderdeel goed test weet je dat daar het probleem niet ligt.

3.2 Setup

Om JUnit goed te gebruiken moet je in je pom.xml files de goede dependecies en plugins gebruiken:

```
   <properties>
     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
     <maven.compiler.source>13</maven.compiler.source>
     <maven.compiler.target>13</maven.compiler.target>
  </properties>
```

Bij dependecies kies je de goede versie van JUnit.

```
  <dependencies>
    <dependency>
    <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-launcher</artifactId>
      <version>1.7.0</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.7.0</version>
      <scope>test</scope>
```

```
    </dependency>
    <dependency>
      <groupId>org.junit.vintage</groupId>
      <artifactId>junit-vintage-engine</artifactId>
      <version>5.7.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

Bij de plugins regel je de plugins zodat maven met JUnit kan communiceren.

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <source>13</source>
          <target>13</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.22.1</version>
      </plugin>
    </plugins>
  </build>
```

Let hierbij op dat je de goede versies voor je project gebruikt.

3.3 Gebruik

Als je een goed maven project hebt gemaakt heb je in je folder structuur een *java* folder en een *test* folder. Het schrijven van test doe je in de test folder. Het is aan te raden dat je de structuur van je test folder net maakt als je java folder. Maar dan met test overal achter. Vervolgens maak je de testFunctie met @Test als annotatie. Declareer in de klasse je sut(system under test) en eventueel dependencies. Bij de //Arrange vul je je verwachte gegevens in, bij de //Act vul je een var met je testgegevens en in de //Assert vergelijk je je testgegevens met je verwachte gegevens.

```
    @Test
public void greetTestEnglish() {
  //Arrange
  var expected = "Hello, World in general and 'Don Quixote'
      specifically!";
  var name = "Don Quixote";
```

```
    var greeting = "Hello, World in general and '%s' specifically!";
    EnglishGreeter greeter = new EnglishGreeter();
    //Act
    var actualString = greeter.greet(greeting, name);
    //Assert
    assertEquals(expected, actualString);
}
```

Vervolgens run je de test.

4 Test Driven Development

Test Driven Development(TDD) is een vorm van programmeren waarbij je eerst de test schrijft met de verwachte uitkomst. En aan de hand van de test schrijf je de echte functie. Hierdoor kan je programma erg modulair worden. Een voordeel van zo programmeren is dat Unit tests schrijven geen probleem is.

5 Exceptions

Exceptions zijn erg belangrijk voor het afhandelen van fouten. Het doel van exceptions is dat je gecontroleerd omgaat met fouten. Door zelf Exceptions te maken kun je makkelijker debuggen en andere developers laten weten wat er fout gaan. Er bestaan 2 hoofdsoorten exceptions

1. Unchecked
2. Checked

beide stammen af van Throwable.

5.1 Unchecked

Unchecked exceptions zijn Runtime Exceptions. Dit zijn system errors zoals null pointer exceptions. Deze kun je niet goed afhandelen anders dan je programma af te sluiten.

5.2 Checked

Checked exceptions zijn de rest van de exceptions. Deze kun je afhandelen in try/catch blokken.

```
    public int divide(int a, int b) throws EmptyDivideExceptions {
        if(b == 0){
            throw new EmptyDivideExceptions("Oh nee je hebt " + a + "/" + b
                + " ingevoerd");
        }else{
            return a/b;
```

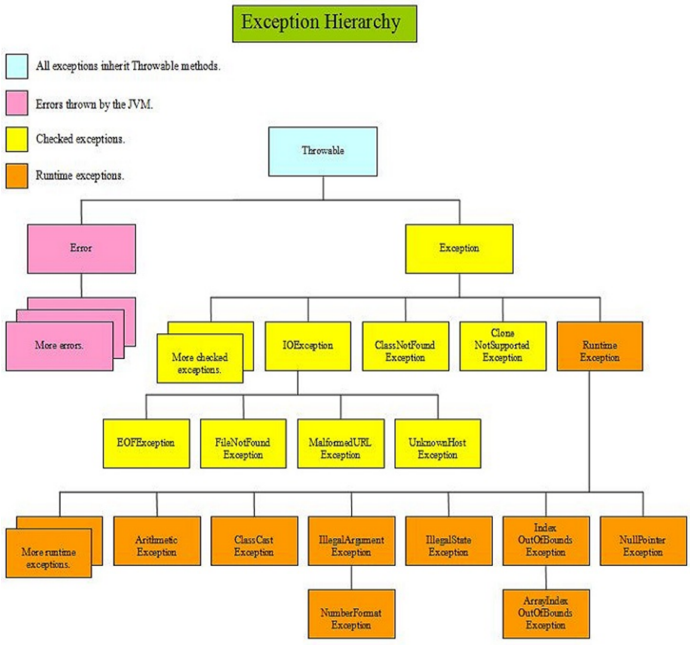

Figure 1:

```
    }
  }

  public int getInfo() throws EmptyInfoException {
    try{
      return divide(1, 0)
    }catch(EmptyDivideExceptions e){
      e.printStackTrace();
      throw new EmptyInfoException("Er ging iets mis");
    }
  }
```

Deze functies geven een error. De exception wordt gegooid(Throw) en opgevangen(Try/Catch). Op deze manier is het duidelijker wat er fout gaat.

```
  public class EmptyDivideExceptions extends Exception {

    public EmptyDivideExceptions(String message) {
      super(message);
    }
  }
```

---

6 Refactoring

Bij het schrijven van grotere programmas wordt het steeds belangrijker om je code te refactoren. Refactoren is het herschrijven van code om van Code Smells af te komen. Dit kun je op verschillende manieren doen, bijvoorbeeld dubbele code naar een functie te verplaatsen en die te gebruiken met parameters. Intelij biedt hier uitgebreide mogelijkheden voor.

7 Java Stream API

Met de Stream API kun je een 'stroom' van java commands uitvoeren op een lijst(ArrayList¡¿, List¡¿, etc). Hiermee kun je bijvoorbeeld for loops vervangen.

```
public List<String> addHobbitsToFellowship(List<String> fellowship){
   List<String> hobbits = new ArrayList<>();

   hobbits.add("Frodo");
   hobbits.add("Sam");
   hobbits.add("Merry");
   hobbits.add("Pippin");

   hobbits.stream().forEach(hobbit -> fellowship.add(hobbit));

   return fellowship;
}
```

Er zijn 2 soorten Operations op een stream: *Terminerende* en *Niet-Terminerende*.

7.1 Terminerende

Operaties die geen Stream als returnwaarde hebben. Dit zijn de Terminerende operaties. De Terminerende operations returnen soms wel elementen uit de lijst. Deze zijn vaak optional om de nullPointerExceptions te voorkomen.

- forEach()
  Voer voor ieder element uit de Stream de als parameter meegegeven Lambda expressie.
- min()
  Retourneer het kleinste element uit de Stream. Als parameter moet een Comparator worden meegegeven die gebruikt wordt voor het bepalen van het minimum.
- max()
  Retourneer het grootste element uit de Stream. Als parameter moet een Comparator worden meegegeven die gebruikt wordt voor het bepalen van het maximum.

- reduce()
  Voeg alle elementen uit de Stream samen tot één element.

7.2 Niet-Terminerende

Operaties die zelf ook weer een Stream als returnwaarde hebben. Dit zijn Niet-terminerende operaties.
Hieronder staat een lijst met wat Niet-Terminerende Operaties

- map()
  Converteer ieder element uit de Stream naar een ander Object.
  (map() heeft een Lambda expression als parameter, dit wordt in het volgende hoofdstuk behandeld)

- filter()
  Creëer een nieuwe Stream met enkel elementen die voldoen aan het predicaat dat als parameter aan filter wordt meegegeven.

- distinct()
  Creëer een nieuwe Stream met enkel de unieke Objecten uit de Stream.

- limit()
  Creëer een nieuwe Stream met het aantal elementen dat als parameter aan limit() wordt meegegeven.

7.3 Stream stringen

Het is ook mogelijk om meerdere stream operations achter elkaar uit te voeren. De volgende code klopt niet voledig, maar laat wel zien wat het idee is.

```
public List<String> filterOnFirstLetter(String letter, List<String>
    fellowship){
   List<String> hobbits = new ArrayList<>();

   hobbits.add("Frodo");
   hobbits.add("Sam");
   hobbits.add("Merry");
   hobbits.add("Pippin");

   hobbits.stream()
   .filter(s -> s.startsWith(letter)))
   .map(String::toUpperCase)
   .sorted()
   .forEach(hobbit -> fellowship.add(hobbit));


   return fellowship;
}
```

8 Lambda expressions

Een Lamda expressie is een implementatie van een interface. Maar niet een hele interface, alleen 1 functie. Het kan worden gebruikt om code overzichterlijk te maken. // Hieronder staat hoe je het zou doen zonder Lamda expressie.

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

   Comparator<String> stringComparator = new Comarator<String>(){
      @Override
      public int compare(String o1, String o2){
         return o1.compareTo(02);
      }
   };

   int comparison = stringComparator.compare("AnonymousClass", "Lambda");

   System.out.println("ComparatorValue: " + comparison);
}
```

Zo maak je een lambda functie:

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

      Comparator<String> stringComparator = (o1, o2) -> o1.compareTo(o2);

      int comparison = stringComparator.compare("AnonymousClass",
          "Lambda");

      System.out.println("ComparatorValue: " + comparison);
   }
```

De parameters voor de pijl hoeven geen type te hebben. Op de plek van de implementatie kun je ook zelf functies maken

9 Multithreading

Computer systemen draaien meerdere processen. Elk proces heeft zijn eigen stukje geheugen. Dit kan niet worden gedeeld tussen processen. Processen gebruiken *Threads* deze kunnen wel geheugen verdelen. Ze zijn lichter dan processen en gebruiken minder geheugen. Op deze manier kun je processen versnellen. Bijvoorbeeld een video kun je renderen in een uur met 1 thread, maar met 2 threads kun je het doen in een halfuur etc etc.
Om een nieuwe Thread te maken moet je de interface java.lang.Runnable en de klasse java.lang.Thread gebruiken. Runnable is een interface met maar 1 methode: run(); Thread is een klasse met een constructor die een Runnable als parameter heeft. In deze klasse zit ook nog de functie start(), als deze wordt

aangeroepen wordt run() ook aangeroepen.

```
Runnable runnable = () -> System.out.println("running");
var thread = new Thread(runnable);
thread.start();
```

Voor het maken van een nieuwe thread is het makkelijk om Lambda expressies te gebruiken. Je moet zelf de invulling geven aan de runnable. Zodra de runnable klaar is met zijn functie stopt de thread.
Zodra je meerdere threads gebruikt heet het multi-threading. Maar multi-threading kan een probleem opleveren bv: als 4 threads teglijk een static variabele verhogen, kan dit fout gaan in de Heap. Want elke thread maakt een nieuwe Stack. Hiervoor is een oplossing bedacht: Synchronisatie. Door bij functies waarbij dit fout kan gaan synchronized te zetten hierdoor kan maar 1 thread teglijk de methode uitvoeren. Dit wordt dan een atomaire functie.

10 HTTP

HTTP betekent Hypertext Transfer Protocol. Via dit protocol wordt data via het web verstuurd. Je browser leest de HTTP en is dus een HTTP Client. Standaard ondersteunen HTTP Clients 2 soorten methoden: GET en POST. Er zijn meerdere methoden maar deze werken alleen via JavaScript.

11 TomEE plus

Tomcat + Java EE
TomEE is de Java Enterprise Edition van Tomcat. Het is een verzameling van een aantal JavaEE projects. Het wordt gebruikt voor web profielen. Het volgende voorbeeld zal worden uitgelegd in dit hoofdstuk.

```
@Path("/voorbeeld")
public class VoorbeeldResource {

   @POST
   @Path("/{name}")
   @Produces(MediaType.APPLICATION_JSON)
   public Response setName(@PathParam("name") String name){
      return Response.ok("Hello " + name + " good day").build();
   }

   @GET
   @Produces(MediaType.TEXT_HTML)
   public String checkHealth(){
      return "<h1>Voorbeeld</h1>";
   }
}
```

12 API

API betekent application programming interface, een api is een set routines, protocolen en tools om software mee te schrijven. Een API is een verbindstuk tussen 2 delen software, bijvoorbeeld back-end en database.

13 War

War is een package structuur die java-based web components bevat. Het is hetzelfde als jar maar dan meer. Je moet de packaging in de pom.xml file aanpassen. En een dependency toevoegen.

```
<groupId>jvk.voobleed</groupId>
<artifactId>voorbeeld</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>

<!--Dependency-->
<dependency>
   <groupId>javax</groupId>
   <artifactId>javaee-api</artifactId>
   <version>8.0.1</version>
   <scope>provided</scope>
</dependency>
```

14 HTTP protocollen

Voor de HTTP protocollen gebruiken we de RESTful API. Hierdoor kunnen we makkelijk HTTPrequest opsturen en ontvangen.

14.1 GET

Met een get request haal je informatie van een HTTP. Een get request kan met of zonder TomEE worden gemaakt. Je kan op de volgende manier een Request maken.

```
public void getSite() throws IOException, InterruptedException {
   var httpClient = HttpClient.newHttpClient();

   var request =
       HttpRequest.newBuilder().uri(URI.create("https://www.nu.nl")).build();

   httpClient.sendAsync(request,
       HttpResponse.BodyHandlers.ofString()).thenAccept(response -> {
      System.out.println(response.statusCocde());
      System.out.println(response.headers());
```

```
        System.out.println(response.body());
      }
   );
}
```

of

```
   @GET
   @Produces(MediaType.TEXT_HTML)
   public String checkHealth(){
      return "<h1>Voorbeeld</h1>";
   }
```

De belangrijkste onbekende delen zijn:

- HttpClient
  De client die gebruikt wordt voor het versturen van een HttpRequest
- HttpRequest
  Het request, geëncapsuleert door Java-objecten. Default is een GET!
- HttpRequest.Builder
  De builder die gebruikt wordt voor het maken van een HttpRequest
- HttpResponse
  De response, geëncapsuleert door Java-objecten
- HttpResponse.BodyHandler
  Een functionele interface voor het verwerken van de body van het response

Je kan ook een callback functie gebruiken

14.2 POST

Om van een GET een POST te maken hoef je alleen maar een .POST er bij te zetten.

```
   var request = HttpRequest.newBuilder()
   .uri(URI.create("https://www.nu.nl"))
   .POST(HttpRequest.BodyPublishers.ofString("message to Post"))
   .build();
```

14.3 PUT en DELETE

Put en Delete worden niet veel gebruikt. Dit komt omdat het niet veilig is om direct gegevens vanuit de applicatie naar database te versturen. Hiervoor worden API gebruikt. Als gegevens uit een database worden verwijdert worden hier request voor geschreven. Deze worden dan aangeroepen met een POST. Vandaar dat ik hier inet verder op inga.

14.4 Response codes

Als een resource wordt behandeld wordt een response code terug gestuurt. Deze hebben allemaal een eigen betekenis. **Succes**

- 200 OK
  Standaard response van een HTTP request. De response hangt af van het type request:
  - Get
    De response is corrosponderend aan de aangevraagde resource.
  - Post
    De response is een entity die het resultaat van de actie beschrijft.
- 201 Created
  De request is geslaagd, een nieuwe resource is aangemaakt.
- 202 Accepted
  De request is geaccepteerd maar nog niet verwerkt.

Client errors

- 400 Bad Request
  De server kan de request niet afhandelen vanwege een client error.
- 401 Unauthorized
  Voor de request is authenticatie nodig, deze ontbreekt of is fout.
- 403 Forbidden
  De request heeft niet de goede authoriteit.
- 404 Not Found
  De request is niet gevonden.
- 405 Method Not Allowed
  De request resource wordt met de verkeerde functie gebruikt.
- 409 Conflict
  De request kan niet worden uitgevoerd omdat er een fout zit in de resource.
- 417 Expectation Failed
  De server voldoet niet aan de requirements van de request.

Server errors

- 500 Internal Server Error
  De standaard error als er iets misgaat op de server en niks anders bekend is.

- 501 Not Implemented
  De server herkend de request methode niet. OF de server kan de request niet uitvoeren omdat dat niet is geimplementeerd.
- 502 Bad Gateway
  De server deed zich voor als een gateway of proxy en kreeg een involedige response van een upstream server.

14.5 @Path(param)

Doormiddel van Tomee is het mogelijk om informatie en variablen uit een url te halen. Boven elke klasse die als resource wordt behandeld moet @Path("/-sublink-") staan. Dit is een toevoeging aan het domein. Functies binnen een resource kunnen ook weer een @Path hebben. De @Path kan ook accolades bevatten met daartussen een variabele naam: (id-nummer). Door in de functie met de @Path, @PathParam("id-nummer") int id-nummer als parameter mee te geven kan de informatie uit de url als parameter worden gebruikt.

```
@Path("/{name}")
public String setName(@PathParam("name") String name){
   return "Hello " + name + " good day";
}
```

Ook is het mogelijk om dit niet als speciaal path te hebben maar als:
https://www.voorbeeldLink.nl/profielen?id=21
variabelen die als ?var= staan beschreven kunnen ook worden uitgelezen.

15 Layer pattern

Voor grote systemen is het aan te raden om lagen in je software aan te brengen. In deze lagen kun je ook weer lagen aanbrengen. In een restfull API kun je de volgenede lagen verwachten:

1. Rest
   Roept de REST request aan en stuurt/ontvangt Responses.
2. Service
   Handelt de Opgehaalde gegevens af.
3. Persistance
   Handelt de Database connecties af.
4. Util
   Heeft alle onderdelen die andere lagen nodig hebben.

Het aantal lagen verschilt voor de opdracht die je maakt. Ook is het aan te raden om voor elke laag een Exception te maken.

16 Dependency Injection

Bij OO programeren heb je vaak andere klasse nodig. Je doel is om de koppeling tussen klassen zo laag mogelijk te maken. Om dit te realiseren maak je voor alles een Interface. vervolgens Inject je deze Interface.

```
public class Clock {
   private IWijzer = wijzer

   @Inject
   public void setWijzer(IWijzer wijzer){
      this.wijzer = wijzer;
   }
}
```

17 Mockito

Mockito is een tool waarmee je uigebreider kan testen. Je mocked de klasse die worden inject. Hiermee roep je dus niet echt die klasse aan. Want voor een unit test wil je maar een unit testen, je gaat ervan uit dat de functies die in de unit worden aangeroepen werken. Je mocked de injected klasse en zegt dat ie een bepaald resultaat moet retourneren als je die aanroept.

```
UserDTO userDTO = new UserDTO("test@email.com", "testUser",
    "password");

UserDAO userDAOfixture;

UserService sutSpy;

@BeforeEach
void setUp() {
   userDAOfixture = Mockito.mock(UserDAO.class);

   sutSpy = Mockito.spy(UserService.class);

   sutSpy.setUserDAO(userDAOfixture);
}

 @Test
void authenticateUserDAOTest() throws ServiceException {
   Mockito.when(userDAOfixture.getPasswordByUser(Mockito.anyString(),
       Mockito.anyString()))
      .thenReturn(userDTO.getUser());

   sutSpy.authenticate(userDTO);
```

```
      assertEquals("testUser",
          userDAOfixture.getPasswordByUser(Mockito.anyString(),
          Mockito.anyString()));
   }
```

Ook kun je met mockito void methoden testen. Hierbij test je of funcites in de void worden aangeroepen.

```
      UserDTO userDTO = new UserDTO("test@email.com", "testUser",
          "password");

   UserDAO userDAOfixture;

   UserService sutSpy;

   @BeforeEach
   void setUp() {
      userDAOfixture = Mockito.mock(UserDAO.class);

      sutSpy = Mockito.spy(UserService.class);

      sutSpy.setUserDAO(userDAOfixture);
   }

    @Test
   void sentVerificationMailSetVerifyUserTest() throws ServiceException,
       MessagingException, IOException {
      Mockito.doNothing().when(userDAOfixture).setNewVerifyableUser(Mockito.anyString(),
          Mockito.anyString());

      sutSpy.sentVerificationMail(userDTO);
      Mockito.verify(userDAOfixture).setNewVerifyableUser(Mockito.anyString(),
          Mockito.anyString());
}
```