**1Toets/programmeeropdracht:**

* Creeer een nieuw Java project met Maven (via de command-line of met IntelliJ)
* Importeer het project in je IDE
* Voeg een of meerdere Maven dependencies toe
* Bouw volgens specificatie een of twee classes met enkele methoden waarin input gevalideerd moet worden gebruik makend van Java Exceptions.
* Bouw JUnit testcases voor een selectie van de methoden. De testcases moeten zorgen voor 100% test coverage van de gegeven methoden.
* Voer de main-methode van de applicatie uit via Maven en run de test cases via Maven.
* Verbouw de applicatie zodat deze gebruik maakt van de Thread API door gebruik van de class Thread, interface Runnable en het keyword synchronized.
* Verbouw de applicatie zodat het een client-server applicatie is die gebruik maakt van het RMI protocol: de classes UnicastRemoteObject, Registry en RemoteException en de interfaces Remote en Serializable.

**User story 1: Maven**

**Beschrijving:**

*As a student, I need to learn Maven so I can build, test and run applications with a script outside an IDE to prepare for working in a continuous integration environment.*

**Apache Maven** is een [softwaregereedschap](https://nl.wikipedia.org/wiki/Software) voor Java-projectmanagement en geautomatiseerde softwarebouw. Het is gelijk in functionaliteit aan het gereedschap [Apache Ant](https://nl.wikipedia.org/wiki/Apache_Ant) (en iets minder aan [PHP](https://nl.wikipedia.org/wiki/PHP)'s PEAR en [Perl](https://nl.wikipedia.org/wiki/Perl_(programmeertaal))'s CPAN), maar heeft een simpelere bouwconfiguratie, gebaseerd op de taal [XML](https://nl.wikipedia.org/wiki/Extensible_Markup_Language). Maven wordt gefaciliteerd door de [Apache Software Foundation](https://nl.wikipedia.org/wiki/Apache_Software_Foundation), waar het voorheen een onderdeel was van het Jakarta Project.

Maven gebruikt een "Project Object Model" (POM) om het softwareontwikkeltraject te sturen. In de POM staan verder de afhankelijkheden met andere modules en componenten, waaruit de volgorde van bouwen bepaald wordt. In de POM kunnen naast de gebruikelijke stappen als compileren en samenvoegen voor distributie, extra acties gedefinieerd worden die het ontwikkelproces kunnen ondersteunen. Voorbeelden hiervan zijn automatisch testen, (statische) codeverificatie en analyse van "Code Coverage" door de testen.

Een belangrijk aspect van Maven is de zogenaamde "repository" waarin verschillende versies van componenten opgeslagen zijn. Dit kunnen componenten zijn waarvan de te bouwen software rechtstreeks afhankelijk is. Ook kunnen dit componenten zijn die het bouwproces zelf ondersteunen. Maven biedt ondersteuning om de repository automatisch te vullen met versies die op het internet aangeboden worden, via Apache en andere organisaties.

<https://maven.apache.org/download.cgi>

https://maven.apache.org/install.html

In de powerpoint van school staat ook veel info (Week 1b; Maven).

**GroupID** = projectnaam

**Artifactid** = naam van jaar zonder versienummer.

**Version** = versienummer.

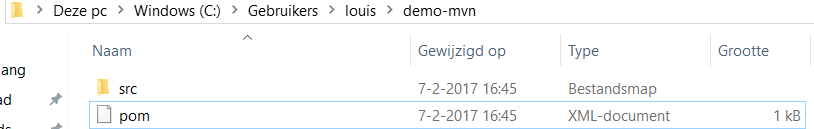
**Acceptatiecriteria:**

* Kan externe libraries toevoegen als Maven dependency.

Uitleg/uitwerking:

Wanneer je een externe library wil toevoegen aan je maven project ga je naar je

Ga je naar je door maven gecreerde map. In dit geval is dat <demo-mvn>.



Open je Pom.xml bestand in bijvoorbeeld kladblok.



Hierin staat <dependencies> daarin staan alle dependencies waarvan jou project afhankelijk is. We willen aan deze een nieuwe <dependency> toevoegen.

<dependency>

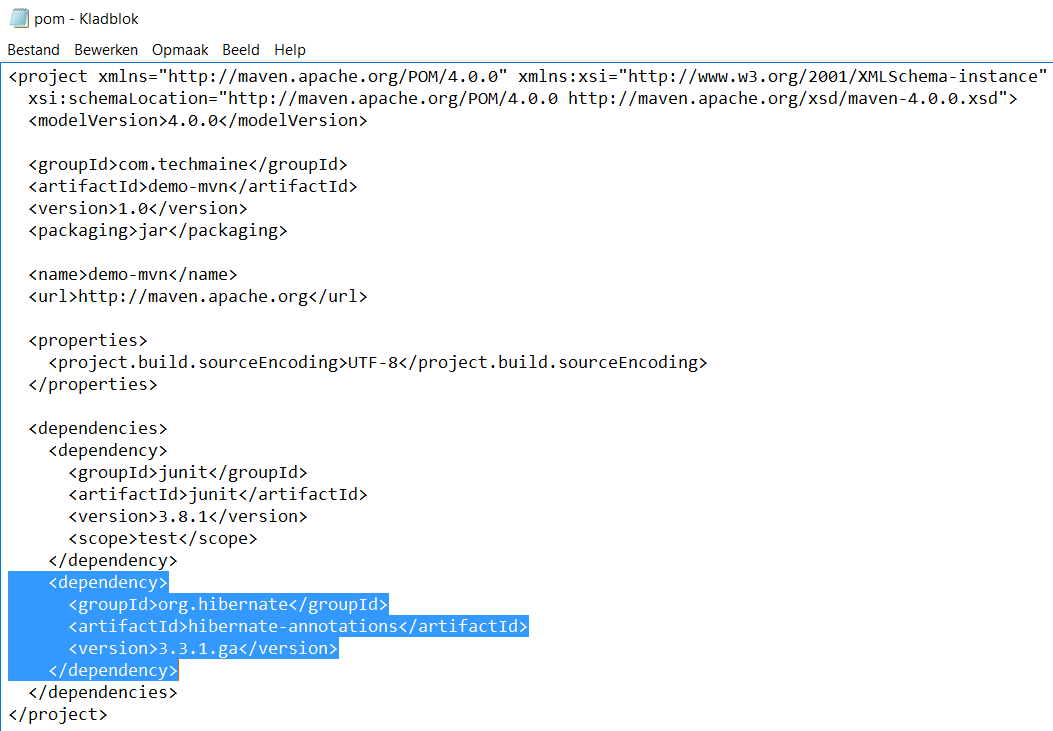
<groupId>org.hibernate</groupId>

<artifactId>hibernate-annotations</artifactId>

<version>3.3.1.ga</version>

</dependency>

Dit voeg je toe aan je POM.xml



Je kan aan je dependencies verschillende <scopes> toevoegen.

**<scope>Compile</scope>**

This is the default scope, used if none is specified. Compile dependencies are available in all classpaths of a project. Furthermore, those dependencies are propagated to dependent projects.

**provided**

This is much like compile, but indicates you expect the JDK or a container to provide the dependency at runtime. For example, when building a web application for the Java Enterprise Edition, you would set the dependency on the Servlet API and related Java EE APIs to scope provided because the web container provides those classes. This scope is only available on the compilation and test classpath, and is not transitive.

**runtime**

This scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.

**test**

This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases. This scope is not transitive.

**system**

This scope is similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.

**import** *(only available in Maven 2.0.9 or later)*

This scope is only supported on a dependency of type pom in the <dependencyManagement> section. It indicates the dependency to be replaced with the effective list of dependencies in the specified POM's <dependencyManagement> section. Since they are replaced, dependencies with a scope of import do not actually participate in limiting the transitivity of a dependency.

* Kan versies van dependencies aanpassen.

Uitleg/uitwerking:

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-annotations</artifactId>

<version>3.3.1.ga</version>

</dependency>

In deze dependency is de versie van deze library 3.3.1.ga . Stel deze library word geupdate en er word een nieuwe versie uitgebracht moet je dus in je POM.xml bestand aangeven dat je een nieuwe versie wil gaan gebruiken.

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-annotations</artifactId>

<version>3.3.2.ga</version>

</dependency>

* Kan Maven command-line gebruiken voor het runnen en unit-testen van een applicatie.

Uitleg/uitwerking:

Eerst moet een Maven project aangemaakt worden; doe dit door command line: mvn archetype:generate uit te voeren.

Hierop volgend dien je het aangemaakte project te importeren in IntelliJ/Eclipse. Dit doe je door in het hoofdmenu te kiezen voor Import Project. Vervolgens kies je de map waar het Maven project staat en kies je voor ‘open’.

Klik een aantal keer door en het project wordt geïmporteerd.

Nadat het project geïmporteerd is is het tijd om een aantal namen aan te passen. Hernoem de map binnen map ‘Java’ (waarschijnlijk com.techmaine oid) naar ‘com.oose’.

Plak vervolgens de volgende code binnen de tag <plugin> in het pom.xml bestand:

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.0.2</version>

<configuration>

<source>1.6</source>

<target>1.6</target>

</configuration>

</plugin>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<version>1.1-beta-1</version>

<artifactId>exec-maven-plugin</artifactId>

<configuration>

<mainClass>com.oose.App</mainClass>

</configuration>

</plugin>

In de bovenstaande code staat de tag <mainClass>. Hernoem dit naar de mapstructuur + naam van je main klasse.

Ten slotte kan de applicatie uitgevoerd worden door ‘mvn package’ en vervolgens ‘mvn exec:java’ uit te voeren in de command line.

**Unit testing**

Neem de volgende regels code op boven een testclass:

import junit.framework.Assert;

import org.junit.Test;

In de klasse zelf dien je een tag te geven en daaronder een methode die een test uitvoert. Zie het volgende voorbeeld:

@Test

public void testPrintHelloWorld() {

Assert.*assertEquals*(App.*getHelloWorld*(), "Hello World");

}

Onthoudt ook de naam van de testklasse; deze kun je gebruiken in Maven om een specifieke test uit te voeren.

In Maven kun je m.b.v. de command line één of meerdere unit tests uitvoeren. Met de volgende code kun je een specifieke test uitvoeren; mvn -Dtest=classname test .Hier hoort classname de naam van je klasse te zijn.

Wanneer je alle testen in je project wilt uitvoeren dien je de ‘-Dtest=’ weg te laten, waarna je dus mvn test overhoudt.

* Kent de lifecycle van Maven met minimaal de goals *clean*, *compile*, *test* en *package*.

Uitleg/uitwerking:

* validate - validate the project is correct and all necessary information is available
* compile - compile the source code of the project
* test - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
* package - take the compiled code and package it in its distributable format, such as a JAR.
* verify - run any checks on results of integration tests to ensure quality criteria are met
* install - install the package into the local repository, for use as a dependency in other projects locally
* deploy - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

Default lifecycle: validate, compile, package, install. Alleen de laatste hoeft uitgevoerd te worden; Maven voert automatisch de vorige stappen uit de build lifecycle uit.

Clean: Verwijderd eerst alles in de target directory, en begint vervolgens met het opvolgende commando ( install, deploy).

Voor gedetailleerde uitleg en onderdelen van elk lifecycle onderdeel zie: <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

Volgens de opdracht dien je minstens clean, compile, test en package te kennen:

* Clean: leegmaken van de target directory.
* Compile: het compileren van de source code.
* Test: uitvoeren van (unit) tests.
* Package: gecompileerde code verpakken in bijvoorbeeld een .JAR.

* Hoeft geen rekening te houden met multi-modules

**Maven Archetypes**

Maven kent verschillende archetypes. Archetypes zijn als het ware een template voor een project. De archetypes zijn hier te vinden:

<https://maven.apache.org/guides/introduction/introduction-to-archetypes.html>

**User story 2: Exceptions**

Referentie xd

|  |
| --- |
| **User story** |
| Als slechtziende wil ik de table of contents kunnen gebruiken zodat ik snel naar het gewenste onderdeel van een boek kan navigeren |
| **Acceptatiecriteria** |
| * Gegeven dat de table of contents links bevat, als ik op een link klik, dan wordt ik naar de bijbehorende pagina geleid. * Gegeven dat de table of contents links naar subhoofdstukken bevat, als ik op een link naar een subhoofdstuk klik, dan wordt ik naar het juiste deel van het hoofdstuk geleid. * Gegeven dat er geen table of contents beschikbaar is, dan wordt dit weergegeven. * Gegeven dat de slechtziende geen link aanklikt, als de slechtziende terug gaat naar het boek, is het boek op de zelfde plaats als het van te voren was |

**Beschrijving:**

*As a student, I need to learn checked and unchecked exceptions so I can handle runtime errors properly.*

**Acceptatiecriteria:**

* Kan gebruik maken van *try* en *catch* om een exception af te vangen

Uitleg/uitwerking:

* Kent de verschillen tussen checked en unchecked exceptions

Uitleg/uitwerking:

* Kan zelf nieuwe exceptions definieren als subclasses van checked of unchecked exceptions in Java.

Uitleg/uitwerking:

* Kan zelfstandig nieuwe exceptions instantieren en "gooien" met de keywords *throw* en *throws*.

Uitleg/uitwerking:

* Kan in unit tests checken of de juiste exception op het juiste moment komt.

Uitleg/uitwerking:

* Hoeft exceptions niet te loggen met een Logging library, System.out.println is voldoende.

**User story 3: Unit testing**

**Beschrijving:**

*As a student, I need to learn unit testing so I can prove that my software matches the requirements.*

**Acceptatiecriteria:**

* Kent de TDD red-green-refactor-cycle.

Uitleg/uitwerking:

Rood test faalt

Groen test gaat goed

Je gaat code herschrijven om de code netjes te houden of iets toevoegen,

Hierna herhaal je dit process weer

* Kan op de TDD wijze testcases definieren die zorgen voor 100% testcoverage van een of twee classes bestaande uit enkele methoden. Hierbij worden niet alleen "happy day"-scenario's getest maar ook randgevallen en uitzonderingen.

Uitleg/uitwerking:

* Kan JUnit als framework gebruiken om testcases te schrijven en uit te voeren

Uitleg/uitwerking:

* Kan unit tests uitvoeren in IntelliJ waarbij coverage rapportage gegenereerd wordt.

Uitleg/uitwerking:

* Kan unit tests uitvoeren met Maven op de command-line.

Uitleg/uitwerking:

* Hoeft geen unit tests te maken voor samenwerkende classes waarbij mocking of stubbing noodzakelijk is.

**User story 4: Threading**

**Beschrijving:**

*As a student, I need to learn the Java Thread API so I can make applications run smoothly.*

**Acceptatiecriteria:**

* Kan met de API-onderdelen *Thread* en *Runnable* een class multi-threaded maken

Uitleg/uitwerking:

* Kan een thread correct starten en stoppen en kent de beperkingen van de API bij het stoppen.

Uitleg/uitwerking:

* Kan het keyword *synchronized* correct inzetten om te voorkomen dat threads tegelijk een gedeelde resource benaderen.

Uitleg/uitwerking:

* Weet het verschil tussen multi-threading en multi-processing en welke rol de JVM, een besturingssysteem en de processor met zijn cores hierin spelen.

Uitleg/uitwerking:

* Hoeft geef rekening te houden met threads die op elkaar moeten wachten (concurrency) waarbij methoden als *wait*, *notify* en *notifyAll*nodig zijn.

**User story 5: RMI**

**Beschrijving:**

*As a student, I need to learn the RMI-protocol so I can build distributed applications and know the concepts of distributed computing.*

**Acceptatiecriteria:**

* Wat is RMI?

Uitleg/uitwerking:

Remote Method Invocation is een vorm van Remote Procedure Call (RPC) en een standaar onderdeel van Java. RMI laat toe om applicaties te schrijven die methodes op andere Java virtuele machines (JVMs) kunnen uitvoeren.

Hiervoor is een referentie nodig naar het remote object. Dit kan mbv een naam-dienst (bootstrap) or een methode aan te roepen die een object referentie oplevert.

Om aanroepen van methoden tussen programma's mogelijk te maken is het volgende nodig:

- server moet bekend maken bij welke objecten methoden kunnen worden aangeroepen

- server moet bekend maken welke methoden dat zijn aanroepen en returnwaarden moeten in een serie bits vertaald kunnen worden (voor verzending) en vice versa

- concrete instanties van objecten die aan de serverkant aangesproken kunnen worden moeten zich bekend maken

Voorbeeld:

**public interface *ObjIf* extends Remote**

**{**

**public *returnwaarde methode*(*parameters*)**

**throws RemoteException;**

**// meer methoden**

**}**

* Kan een interface definieren voor een remote class gebruik makend van de API-onderdelen Remote en RemoteException

Uitleg/uitwerking:

* Kan een class remote benaderbaar maken gebruik makend van de API-onderdelen UnicastRemoteObject en Java interfaces.

Uitleg/uitwerking:

* Kan een server voor een remote class schrijven die via het RMI-protocol en de Registry aanroepen van een client kan ontvangen.

Uitleg/uitwerking:

* Kan een client schrijven die via het RMI-protocol en de Registry methoden op een remote class aanroept.

Uitleg/uitwerking:

* Weet het verschil tussen pass-by-value en pass-by-reference en kan het API-onderdeel Serializable inzetten voor pass-by-value.

Uitleg/uitwerking:

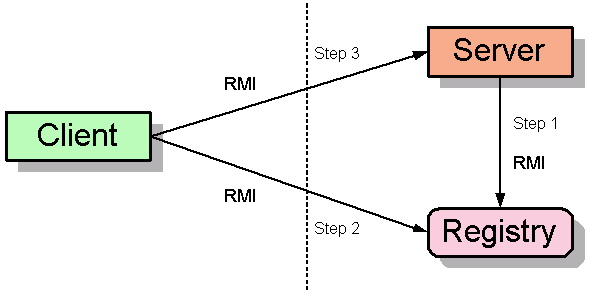
* Kent het concept van client-registry-server.

Uitleg/uitwerking:

* Hoeft geen rekening te houden met security (SecurityManager), codebases en classpaths.

RMI introductie.

RMI(Remote method invocation) is het aanroepen van een functie van een methode op een andere JVM(Java Virtual Machine) door gebruik van de registry(van Java niet te verwarren met de Windows Registry). Hierdoor kan een JVM dus methodes gebruiken van een andere JVM. De registry dient hierbij als wegwijzer. Dit kan lokaal op de zelfde computer en via het internet op een andere computer.



In het voorbeeld is een stappenplan te zien van RMI. Eerst start de server de registry en blijft ermee in contact. Dan pingt de client de registry voor de locatie van de server, dan geeft de client zijn command (waarschijnlijk met een methode aanroep op de server). De server voert deze methode door de remote methode aanroep van de client.