# Übung Maven

## Ziel

1. Erstellen einer Java-FX-Anwendung mit Maven
2. Erstellen eines Executables

## Anleitung

1. Legen Sie ein neues Maven-Projekt an.

<groupId>**IHRE.GROUP.ID**</groupId> 🡸 entspricht dem Paketnamen

<artifactId>**IHR-ARTEFACT-NAME**</artifactId> 🡸 entspricht dem Projektnamen

<version>1.0-SNAPSHOT</version>

1. Verwenden Sie Java 11.

Setzen Sie dazu die Properties maven.compiler.source und maven.compiler.target in der pom.xml.

**pom.xml**

<properties>

**<maven.compiler.source>11</maven.compiler.source>**

**<maven.compiler.target>11</maven.compiler.target>**

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

1. Fügen Sie die Java-FX-Dependency in der Version 11 ein.

* Die benötigte Dependency ist das javafx-controls.
* Finden Sie die Koordinaten im zentralen Maven-Repository: <https://mvnrepository.com/>
* Fügen Sie die Dependency in die pom.xml ein.

**pom.xml**

<properties>

<maven.compiler.source>11</maven.compiler.source>

<maven.compiler.target>11</maven.compiler.target>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

**<dependencies>**

**<dependency>**

**<groupId>org.openjfx</groupId>**

**<artifactId>javafx-controls</artifactId>**

**<version>11</version>**

**</dependency>**

**</dependencies>**

1. Erstellen Sie das Hauptpaket im src/main/java-Folder

**Project Root 🡪 src 🡪 main 🡪 java 🡪 IHRE 🡪 GROUP 🡪 ID**

1. Erstellen Sie die Einstiegsklasse (Main).

* Diese muss von Application erben.
* Sie benötigt eine Main-Methode, die die Launch-Methode ruft, die durch die Vererbung von Application verfügbar ist.
* Die lauch-Methode ruft die Start-Methode, was den eigentlichen Einstiegspunkt darstellt.
* Die start-Methode soll so implementiert werden, dass ein Fenster auf dem Bildschirm angezeigt wird, auf dem eine Hallo-Welt-Nachricht ausgegeben wird (zum Beispiel als Label).

**Main.java**

public class Main extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage stage) {

String message = "Hello World";

Label messageLabel = new Label(message);

HBox container = new HBox(messageLabel);

container.setPadding(new Insets(100));

Scene scene = new Scene(container);

stage.setScene(scene);

stage.show();

}

}

1. Java-FX benötigt eine Launcher-Klasse:

* Die Ursache liegt in einer Java-Basisklasse (sun.launcher.LauncherHelper), die immer, wenn die Main-Klasse von Application erbt und eine main-Methode hat nach der Komponente javafx.graphics sucht. Diese muss als benanntes JPMS-Modul vorliegen. Ist dies nicht der Fall, terminiert die Applikation. Dies ist hier auch der Fall, da keine JPMS-Module genutzt werden.
* Die Lösung ist die Launcher-Klasse, die eine Main-Methode hat, die an die Main-Methode der eigentlichen, von Application erbenden Main-Klasse delegiert.
* Implementieren Sie eine entsprechende Launcher-Klasse

**Launcher.java**

public class Launcher {

public static void main(String[] args) {

Main.main(args);

}

1. Kompilieren Sie das Projekt und führen Sie es aus.
2. Importieren Sie das maven-shade-plugin

* Um ein ausführbares JAR anlegen zu können, gibt es mehrere Möglichkeiten. Hier soll das maven-shade-plugin genutzt werden, um alle JPMS-Abhängigkeiten aufzulösen und in ein sogenanntes Uber-JAR zu verpacken.
* Das maven-shade-plugin finden Sie im zentralen Repository.
* Fügen Sie es in die Build 🡪 Plugin-Sektion der pom.xml ein.
* Binden Sie das Goal Shade des Plugins an die Phase Package. Das geschieht in der Execution-Sektion des Plugins. Tätigen Sie dort auch die nötige Configuration des Transformers. Binden Sie diesen an die Transformer-Implementierung org.apache.maven.plugins.shade.resource.ManifestResourceTransformer und benennen Sie Ihre Launcher-Klasse als Main-Class.

**pom.xml**

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<version>3.5.2</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<transformers>

<transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">

<mainClass>de.hwglu.atdit\_2024.Launcher</mainClass>

</transformer>

</transformers>

</configuration>

</execution>

</executions>

</plugin>

1. Führen Sie die Packaging-Phase aus

Dies erstellt die ausführbare Jar-Datei im Ordner Target ihres Projekts.

**Project Root 🡪 target 🡪 IHR-ARTEFACT-NAME-1.0-SNAPSHOT.jar**

1. Führen Sie die Jar-Datei aus.

In der Console: **javaw -jar IHR-ARTEFACT-NAME-1.0-SNAPSHOT.jar**

# Übung Git

## Ziel

1. Ein Projekt mit Git anlegen
2. Ein Projekt auf GitHub veröffentlichen
3. Änderungen vornehmen
4. Parallele Änderungen abmischen

## Anleitung

1. Legen Sie ein neues Maven-Projekt an mit Git-Repository
2. Führen Sie ein Initialcommit aus

* Wählen Sie alle geänderten Dateien aus
* Fügen Sie eine Commit-Message ein
* Führen Sie den Commit aus

1. Veröffentlichen Sie das Projekt auf GitHub

* Suchen Sie das entsprechende Kommando im Suchfeld oder finden Sie es unter dem Menüpunkt Git 🡪 GitHub und folgen Sie dem UI

1. Führen Sie die folgenden Änderungen in einem neuen Branch durch.

* Legen Sie einen neuen Branch auf basis des Master-Branches an und stellen Sie sicher, dass er ausgecheckt wurde.
* Main-Klasse und Main-Methode erstellen. Anschließend Commit.

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

**}**

* In der Main-Methode soll zunächst per Konsole die Welt gegrüßt und ihr danach ein schöner Tag gewünscht werden. Implementieren Sie dieses Verhalten und führen Sie anschließend ein Commit durch.

public static void main(String[] args) {

**System.out.println("Hallo, Welt");**

**System.out.println("Hab einen schönen Tag");**

}

* Beschreiben Sie das Methodenverhalten in einem entsprechenden Javadoc-Kommentar gefolgt von einem Commit.

**/\*\***

**\* Dies ist der Einstiegspunkt in unsere Applikation.<br/>**

**\* Die Methode macht folgendes:**

**\* <ol>**

**\* <li>Hallo Welt schreiben</li>**

**\* <li>Einen schönen Tag wünschen</li></ol>**

**\***

**\* \* @param args Kommandozeilenparameter**

**\*/**

public static void main(String[] args) {

System.out.println("Hallo, Welt");

System.out.println("Hab einen schönen Tag");

}

1. Übernehmen Sie die Änderungen des neuen Branches in den Master-Branch (Merge)

* Checken Sie den Master-Branch aus.
* Führen Sie ein Fetch auf dem Master Branch aus. Es sollte keine Änderungen geben.
* Führen Sie den Merge aus (entweder über das Menu Git 🡪 Merge oder das Suchfeld).
  + Wählen Sie Ihren Branch, der in den Master eingemischt werden soll.
  + Wählen Sie die Zusatzoption –squash
  + Führen Sie den Merge durch
* Committen Sie den Merge.
* Zusätzlich können Sie nun über das Kontextmenü Ihres Branches noch einmal mergen, um den Branch auch im Graph mit dem Master-Branch zu vereinen.
* Führen Sie ein Push nach GitHub durch.

1. Schließen Sie das Projekt.

**Es folgen zwei parallele Korrekturen und das Auflösen eines Abmischkonflikts.**

Aktuell grüßt das Programm die Welt wie folgt:

Hallo, Welt

Hab einen schönen Tag

Es gibt zwei Tickets:

1. Die Welt möchte gesiezt werden (“Haben Sie einen schönen Tag”)
2. Der Gruß sollte überschwänglicher sein (“Hab einen wunderschönen Tag”)

Beide Änderungen werden parallel durchgeführt!

1. Erstellen Sie dazu zunächst ein neues Projekt aus der Versionsverwaltung von GitHub. Klonen Sie Ihr gerade geschlossenes Projekt und nennen Sie es “Ticket1”.
2. Erstellen Sie einen neuen Branch “bugfix1” auf Basis des Master-Branches
3. Führen Sie folgende Codeänderung durch und committen Sie diese.

**Main.java**

/\*\*

\* Dies ist der Einstiegspunkt in unsere Applikation.<br/>

\* Die Methode macht folgendes:

\* <ol>

\* <li>Hallo Welt schreiben</li>

\* <li>Einen schönen Tag wünschen</li></ol>

**\* Die Welt muss natürlich gesiezt werden!**

\*

\* @param args Kommandozeilenparameter; werden nicht genutzt

\*/

public static void main(String[] args) {

System.out.println("Hallo, Welt");

**~~System.out.println("Hab einen schönen Tag");~~**

**System.out.println("Haben Sie einen schönen Tag");**

}

1. Erstellen Sie ein weiteres neues Projekt als Klon des zuvor geschlossenen und nennen Sie es “Ticket2”.
2. Erstellen Sie einen neuen Branch “bugfix2” auf Basis des Master-Branches
3. Führen Sie folgende Codeänderung durch und committen Sie diese.

/\*\*

\* Dies ist der Einstiegspunkt in unsere Applikation.<br/>

\* Die Methode macht folgendes:

\* <ol>

\* <li>Hallo Welt schreiben</li>

**~~\* <li>Einen schönen Tag wünschen</li></ol>~~**

**\* <li>Einen wunderschönen Tag wünschen</li></ol>**

\*

\* @param args Kommandozeilenparameter; werden nicht genutzt

\*/

public static void main(String[] args) {

System.out.println("Hallo, Welt");

**~~System.out.println("Hab einen schönen Tag");~~**

**System.out.println("Hab einen wunderschönen Tag");**

}

1. Mergen Sie den Branch bugfix2 im Projekt Ticket2 in den Master-Branch und pushen Sie die Änderung ins Remoterepository auf GitHub.
   1. Branch Master auschecken
   2. Fetch (Master)
   3. Branch bugfix2 nach Master mergen
   4. Push (Master)
2. Mergen Sie den Branch bugfix1 im Projekt Ticket1 in den Master-Branch und pushen Sie die Änderung ins Remoterepository auf GitHub. Hier sollte ein Mergekonflikt entstehen.
   1. Branch Master auschecken
   2. Fetch (hier wird die Änderung von Projekt Ticket2 importiert)
   3. Mergen Sie bugfix1 nach Master (das öffnet des Konfliktfenster)
   4. Konflikte manuell beseitigen und merge durchführen
   5. Push (Master)

# Übung Unit-Test

## Ziel

1. Implementierung der klassischen Caesar-Verschlüsselung
2. Einen Unit-Test mit Junit schreiben

## Anleitung

1. Legen Sie ein neues Maven-Projekt an.
2. Binden Sie das JUnit Jupiter-Aggregate als Test-Dependency ein.

**pom.xml**

<dependencies>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

<version>5.10.2</version>

<scope>test</scope>

</dependency>

</dependencies>

1. Aktualisieren Sie das Maven-Surefire-Plugin.

**pom.xml**

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.2.5</version>

</plugin>

</plugins>

</build>

1. Legen Sie eine Schnittstelle für das Verschlüsselungsmodul an. Es soll ein String entgegengenommen und verschlüsselt zurückgeliefert werden.

**Encryption.java**

public interface Encryption {

String encrypt(String plain);

}

1. Legen Sie eine Implementierung gemäß des klassischen Caesar-Chiffrieralgorithmuses an. Achtung: Die Lösung enthält einen Fehler; das ist gewollt.

**Caesar3Encryption.java**

public class Caesar3Encryption implements Encryption {

@Override

public String encrypt(String plain) {

StringBuilder result = new StringBuilder();

var plainCharacters = plain.toCharArray();

for (char plainChar : plainCharacters) {

int plainCharNumber = plainChar - 96; //makes a = 1; z = 26

int encryptedCharNumber = plainCharNumber + 5; //makes a = 4; z = 29

if (encryptedCharNumber > 26) //avoid overflow; after z comes a

encryptedCharNumber -= 26; //z changes from 29 to 3

char encryptedChar = (char) (encryptedCharNumber + 96); //convert back to character

result.append(encryptedChar);

}

return result.toString();

}

}

1. Legen Sie einen Unit-Test an.

* Im Ordner Project Root 🡪 src 🡪 test 🡪 java
* Benutzen Sie dasselbe Paket wie im Produktivcode
* Bennenen Sie die Testklasse wie die zu testende Klasse
* Fügen Sie einen Testfall ein

**Caesar3EncryptionTest.java**

public class Caesar3EncryptionTest {

@Test

public void IHR\_METHODEN\_NAME() {

// assemble

// act

// assert

}

}

1. Implementieren Sie den Testfall wie folgt und halten Sie sich dabei an die Drei As.

* Ein beliebig gewählter String soll der Input für die klassische Caesar-Chiffrierung sein.
* Leiten Sie einen Erwartungswert ab.
* Prüfen Sie, dass der Erwartungswert und das Ergebnis einer Verschlüsselung des Eingabewerts mit der klassischen Caesar-Verschlüsselung (Verschiebung um drei Buchstaben) übereinstimmen. Falls dies nicht der Fall ist, muss der Unit-Test scheitern.
* Geben Sie der Methode einen Sinnvollen Namen.

**Caesar3EncryptionTest.java**

public class Caesar3EncryptionTest {

@Test

public void HelloMustBecomeKhoor() {

// assemble

String input = "hello";

String expectedResult = "khoor";

var cut = new Caesar3Encryption();

// act

var actualResult = cut.encrypt(input);

// assert

Assertions.assertEquals(expectedResult, actualResult);

}

}

1. Lassen Sie den Testfall laufen. Er muss auf einen Fehler laufen, wenn Sie sich an der Lösung orientiert haben.
2. Finden Sie den Fehler im Chiffrierungsalgorithmus durch Debuggen des Unit-Tests und korrigieren Sie ihn.

**Caesar3Encryption.java**

public class Caesar3Encryption implements Encryption {

@Override

public String encrypt(String plain) {

StringBuilder result = new StringBuilder();

var plainCharacters = plain.toCharArray();

for (char plainChar : plainCharacters) {

int plainCharNumber = plainChar - 96; //makes a = 1; z = 26

**~~int encryptedCharNumber = plainCharNumber + 5; //makes a = 4; z = 29~~**

**int encryptedCharNumber = plainCharNumber + 3; //makes a = 4; z = 29**

if (encryptedCharNumber > 26) //avoid overflow; after z comes a

encryptedCharNumber -= 26; //z changes from 29 to 3

char encryptedChar = (char) (encryptedCharNumber + 96); //convert back to character

result.append(encryptedChar);

}

return result.toString();

}

}

1. Lassen Sie den Testfall erneut laufen. Der Test sollte bestanden werden.
2. Lassen Sie den Testfall mit Abdeckungsmessung laufen. Wenn Sie sich an der Lösung orientiert haben, sollten einige Statements nicht erreicht worden sein. Erstellen Sie einen weiteren Testfall, damit auch das letzte Statement mindestens einmal erreicht wurde.

**Caesar3EncryptionTest.java**

public class Caesar3EncryptionTest {

@Test

**public void wholeAlphabetMustBeShiftedThreePlaces () {**

// assemble

**String input = "abcdefghijklmnopqrstuvwxyz";**

**String expectedResult = "defghijklmnopqrstuvwxyzabc";**

var cut = new Caesar3Encryption();

// act

var actualResult = cut.encrypt(input);

// assert

Assertions.assertEquals(expectedResult, actualResult);

}

}

1. Übung Dependency Injection
   1. Ziel
2. Implementieren eines SecureCredentialStringBuilders
3. Manuelles Auflösung einer Abhängigkeit in Unit-Tests
4. Auflösung einer Abhängigkeit in Unit-Tests mit Mockito
   1. Anleitung
5. Installieren Sie das Caesar-Modul aus Übung 3 ins lokale Repository (Maven Install-Phase).
6. Legen Sie ein neues Maven-Projekt an.
7. Binden Sie das JUnit Jupiter-Aggregate als Test-Dependency ein, sowie das Caesar-Modul mit Default Scope (compile).

**pom.xml**

*<dependencies>*

*<dependency>*

*<groupId>IHRE\_GROUP\_ID</groupId>*

*<artifactId>IHR\_CAESAR\_ARTEFAKT\_NAME</artifactId>*

*<version>IHRE\_CAESAR\_ARTEFAKT\_VERSION</version>*

*</dependency>*

*<dependency>*

*<groupId>org.junit.jupiter</groupId>*

*<artifactId>junit-jupiter</artifactId>*

*<version>5.10.2</version>*

*<scope>test</scope>*

*</dependency>*

*</dependencies>*

1. Aktualisieren Sie das Maven-Surefire-Plugin.

**pom.xml**

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.2.5</version>

</plugin>

</plugins>

</build>

1. Legen Sie eine Schnittstelle für den CredentialStringBuilder an. Dieser soll einen Usernamen und ein Passwort entgegennehmen (jeweils Strings) und einen kombinierten String zurückliefern im Format <user>:<passwort>.

**CredentialStringBuilder.java**

public interface CredentialStringBuilder {

String makeCredentialString(String user, String password);

}

1. Erstellen Sie eine entsprechende Implementierung (SecureCredentialStringBuilder), welche sowohl den User als auch das Passwort per Caesarchiffre verschlüsselt. Ignorieren Sie dabei die Möglichkeit, die Chiffrierungsmethode per Dependency Injection austauschen zu können, erstellen Sie also eine Instanz für die Chiffrierung in dieser Implementierung.

**SecureCredentialStringBuilder.java**

public class SecureCredentialStringBuilder implements CredentialStringBuilder {

@Override

public String makeCredentialString(String user, String password) {

StringBuilder sb = new StringBuilder();

Encryption encryptor = new Caesar3Encryption();

String encryptedUser = encryptor.encrypt(user);

String encryptedPassword = encryptor.encrypt(password);

sb.append(encryptedUser)

.append(":")

.append(encryptedPassword);

return sb.toString();

}

}

1. Erstellen Sie einen Unit-Test, der prüft, dass SecureCredentialStringBuilder den User “myuser” und das Passwort “mypassword” entsprechend der Caesar-Verschlüsselung verschlüsselt und im Format <verschlüsselter user>:<verschlüsseltes passwort> ausgibt.

Sorgen Sie dafür, dass der Unit-Test bestanden wird.

**SecureCredentialStringBuilderTest.java**

public class SecureCredentialStringBuilderTest {

@Test

public void userAndPasswordMustBeSeparatedByColon() {

String inputUser = "myuser";

String inputPassword = "mypassword";

String expected = "pbxvhu:pbsdvvzrug";

var cut = new SecureCredentialStringBuilder();

var actual = cut.makeCredentialString(inputUser, inputPassword);

Assertions.assertEquals(expected, actual);

}

}

1. Öffnen Sie das Caesar-Projekt erneut und führen Sie eine Korrektur durch, die die Verschlüsselung von +3 auf +5 verschiebt. Installieren Sie diese Änderungen anschließend ins lokale Repository unter Überspringen der Unit-Tests.

**Caesar3Encryption.java**

public class Caesar3Encryption implements Encryption {

@Override

public String encrypt(String plain) {

StringBuilder result = new StringBuilder();

var plainCharacters = plain.toCharArray();

for (char plainChar : plainCharacters) {

int plainCharNumber = plainChar - 96; //makes a = 1; z = 26

**~~int encryptedCharNumber = plainCharNumber + 3; //makes a = 4; z = 29~~**

**int encryptedCharNumber = plainCharNumber + 5; //makes a = 4; z = 29**

if (encryptedCharNumber > 26) //avoid overflow; after z comes a

encryptedCharNumber -= 26; //z changes from 29 to 3

char encryptedChar = (char) (encryptedCharNumber + 96); //convert back to character

result.append(encryptedChar);

}

return result.toString();

}

}

1. Kompilieren Sie das aktuelle Maven-Projekt neu; stellen Sie zuvor sicher, dass in den Einstellungen “always update Snapshots” angeschaltet ist. Lassen Sie im Anschluss den Unit-Tests laufen. Dieser schlägt nun fehl, da sich die Abhängigkeit geändert hat.
2. Abhängigkeit muss aufgelöst werden, indem der Unit-Test einen Encryptor definiert und dem SecureCredentialStringBuilder per Dependency Injection unterjubelt. Dazu muss die lokale Instanz aus der Schnittstellenmethode zu einem Attribut der Klasse werden, das per Konstruktor gesetzt wird. Implementieren Sie diese Änderung.

**SecureCredentialStringBuilder.java**

public class SecureCredentialStringBuilder implements CredentialStringBuilder {

**private final Encryption encryptor;**

**public SecureCredentialStringBuilderImplementation(Encryption encryptor) {**

**this.encryptor = encryptor;**

**}**

@Override

public String makeCredentialString(String user, String password) {

StringBuilder sb = new StringBuilder();

**~~Encryption encrypter = new Caesar3Encryption();~~**

String encryptedUser = encryptor.encrypt(user); //encryptor zeigt auf this.encryptor

String encryptedPassword = encryptor.encrypt(password);

sb.append(encryptedUser)

.append(":")

.append(encryptedPassword);

return sb.toString();

}

}

1. Jetzt können Sie im Unit-Test händisch einen eigenen Encryptor implementieren. Damit weiß der Unit-Test ganz genau, was bei welcher Eingabe als verschlüsseltes Ergebnis herauskommen soll. Am einfachsten ist es, die Verschlüsselung komplett zu ignorieren. Implementieren sie daher einen Encryptor, der einfach die Eingabe zurückliefert. Es ist aber Sinnvoll, zu prüfen, ob die erwartete Eingabe beim Encryptor ankommt, denn dann steht fest, dass der Datenfluss innerhalb des SecureCredentialStringBuilder korrekt ist. Lassen Sie den Encryptor den Unit-Test zum fehlschlagen bringen, falls er eine andere Eingabe als den erwarteten Benutzer oder das erwartete Passwort erhält [mit Assertions#fail()].

Ändern Sie den Unit-Test entsprechend ab.

**SecureCredentialStringBuilderTest.java**

public class SecureCredentialStringBuilderTest {

@Test

public void userAndPasswordMustBeSeparatedByColon() {

**Encryption encryptorMock = plain -> {**

**if (!Objects.equals(plain, "myuser") && !Objects.equals(plain, "mypassword"))**

**Assertions.fail("unexpected input");**

**return plain;**

**};**

String inputUser = "myuser";

String inputPassword = "mypassword";

**~~String expected = "pbxvhu:pbsdvvzrug";~~**

**String expected = "myuser:mypassword";**

**~~var cut = new SecureCredentialStringBuilder();~~**

**var cut = new SecureCredentialStringBuilder(encryptorMock);**

var actual = cut.makeCredentialString(inputUser, inputPassword);

Assertions.assertEquals(expected, actual);

}

}

1. Lassen Sie die Unit-Tests laufen und vergewissern Sie sich, dass sie erfolgreich sind und warum (am besten per Debugger den Programmfluss verfolgen).
2. Abhängigkeiten können auch mit der Bibliothek Mockito implementiert werden. Mit Mockito lassen sich komplexe Implementierungen sicher generieren; die Gefahr, dass der Mock einen Programmfehler enthält, wird abgemildert. Weiterhin bietet Mockito Tools, um etwa den Datenfluss zu prüfen, was gerade ebenfalls händisch gemacht werden musste.

Um Mockito zu nutzen, fügen Sie die JUnit-Jupiter-spezifische Test-Dependency ein (mockito-junit-jupiter).

**pom.xml**

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-junit-jupiter</artifactId>

<version>5.11.0</version>

<scope>test</scope>

</dependency>

1. Erstellen Sie einen weiteren Unit-Test. Generieren Sie einen Encryptor mit Mockito, der bei Eingabe des erwarteten Benutzernamens eben diesen zurückgibt; analog für das erwartete Passwort. Nutzen Sie weiterhin die Mockito#Verify-Methode, um den Datenfluss zu kontrollieren.

**SecureCredentialStringBuilderTest.java**

public class SecureCredentialStringBuilderTest {

@Test

public void userAndPasswordMustBeSeparatedByColonMockito() {

//assemble

String inputUser = "myuser";

String inputPassword = "mypassword";

String expected = "myuser:mypassword";

**//mock erstellen**

**Encryption encryptorMock = Mockito.mock(Encryption.class);**

**//wenn mock inputUser erhält, diesen unverändert gerade zurückgeben; selbiges bei inputPassword**

**//alle anderen Eingaben liefern automatisch NULL**

**Mockito.when(encryptorMock.encrypt(inputUser)).thenReturn(inputUser);**

**Mockito.when(encryptorMock.encrypt(inputPassword)).thenReturn(inputPassword);**

var cut = new SecureCredentialStringBuilderImplementation(encryptorMock);

//act

var actual = cut.makeCredentialString(inputUser, inputPassword);

//assert

**//Datenfluss prüfen. Es wird erwartet, dass die encrypt-Methode genau einmal gerufen wurde mit inputUser**

**//und ebenso genau einmal mit inputPassword**

**Mockito.verify(encryptorMock, Mockito.times(1)).encrypt(inputUser);**

**Mockito.verify(encryptorMock, Mockito.times(1)).encrypt(inputPassword);**

Assertions.assertEquals(expected, actual);

}

}

1. Lassen Sie die Unit-Tests laufen und vergewissern Sie sich, dass sie erfolgreich sind und warum (am besten per Debugger den Programmfluss verfolgen).