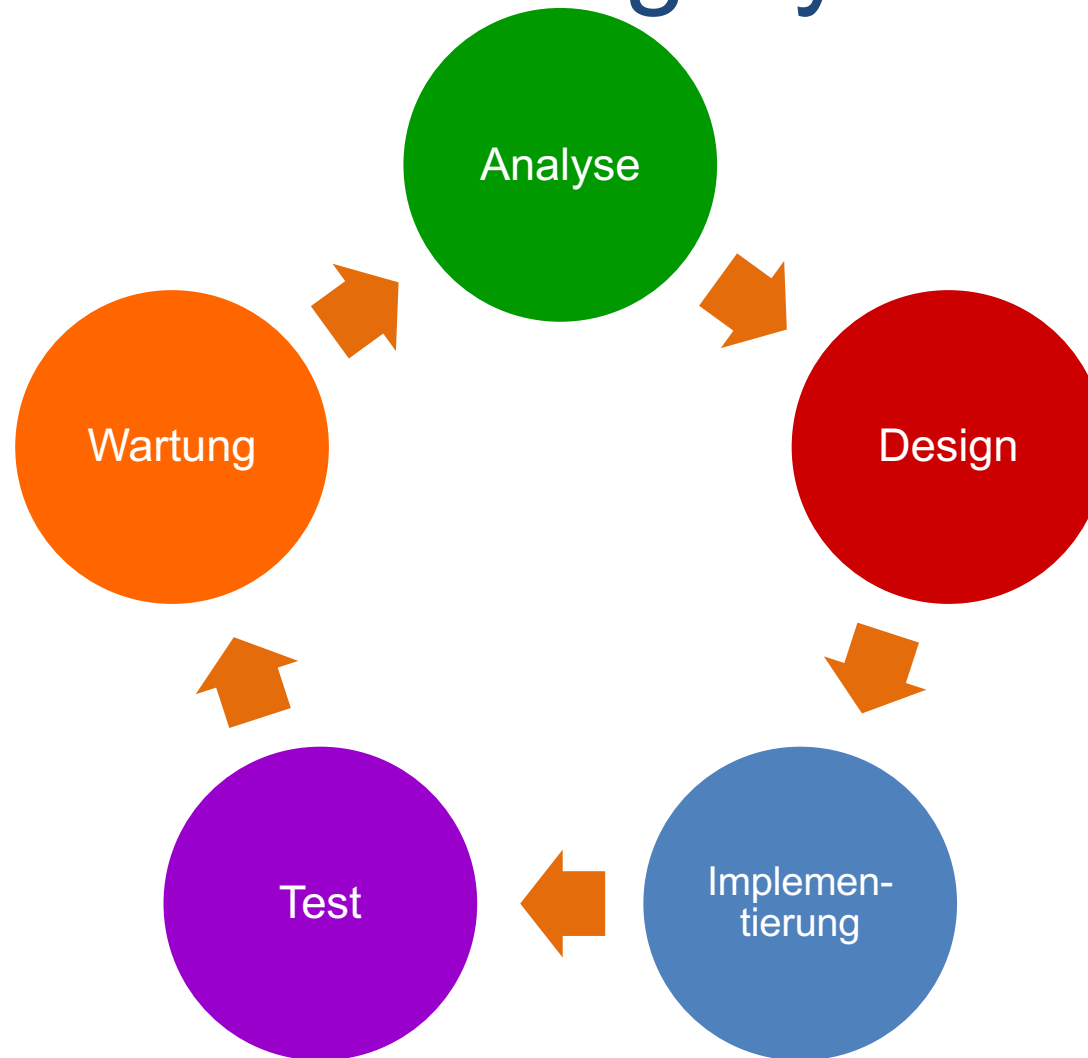


# 7. Programmieren im Großen

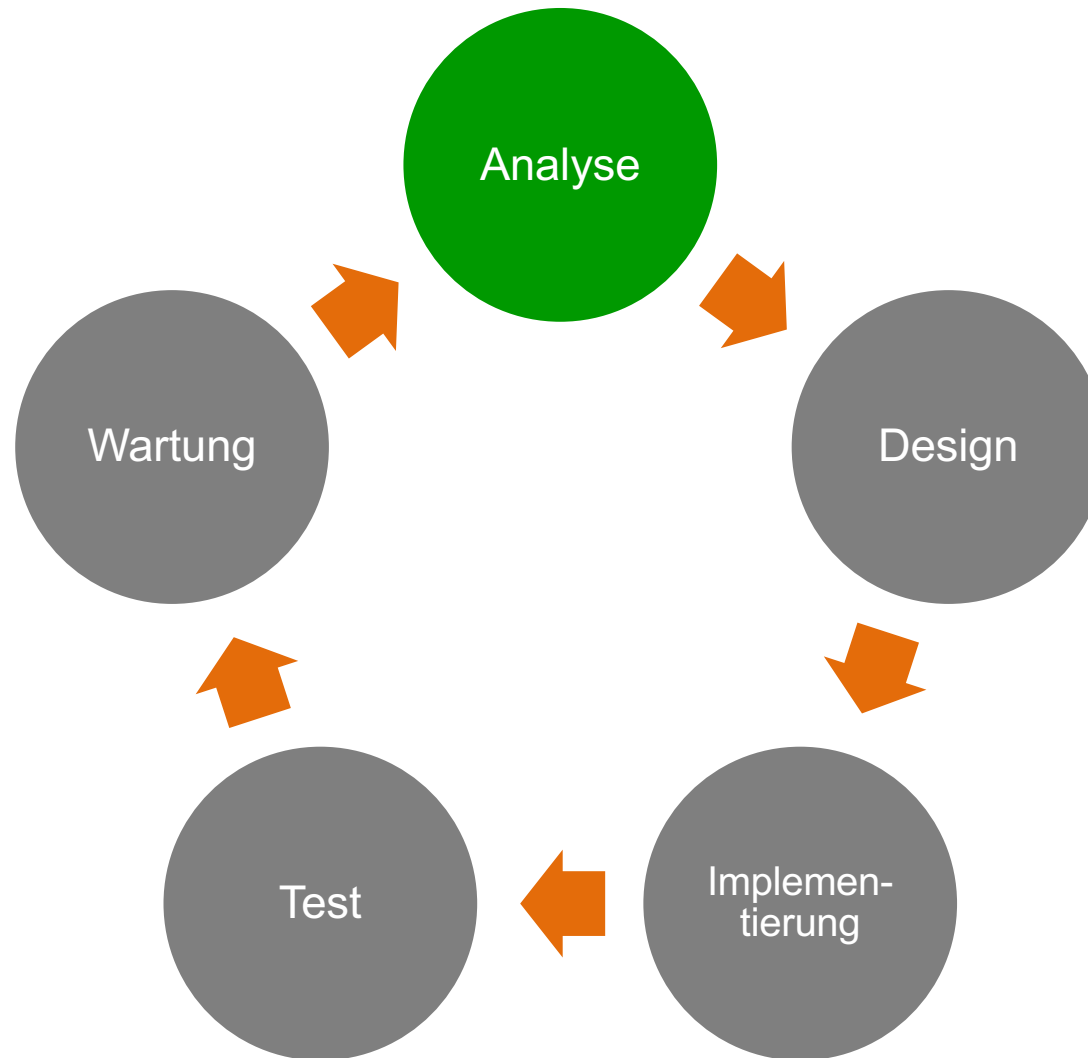
- Entwicklung: IntelliJ IDEA, Bibliotheken, Versionsverwaltung
- Testen: Unit-Tests, Integrationstests
- Fehlerbehebung: Debugger
- Auslieferung: JAR-Dateien



# Software-Entwicklungszyklus



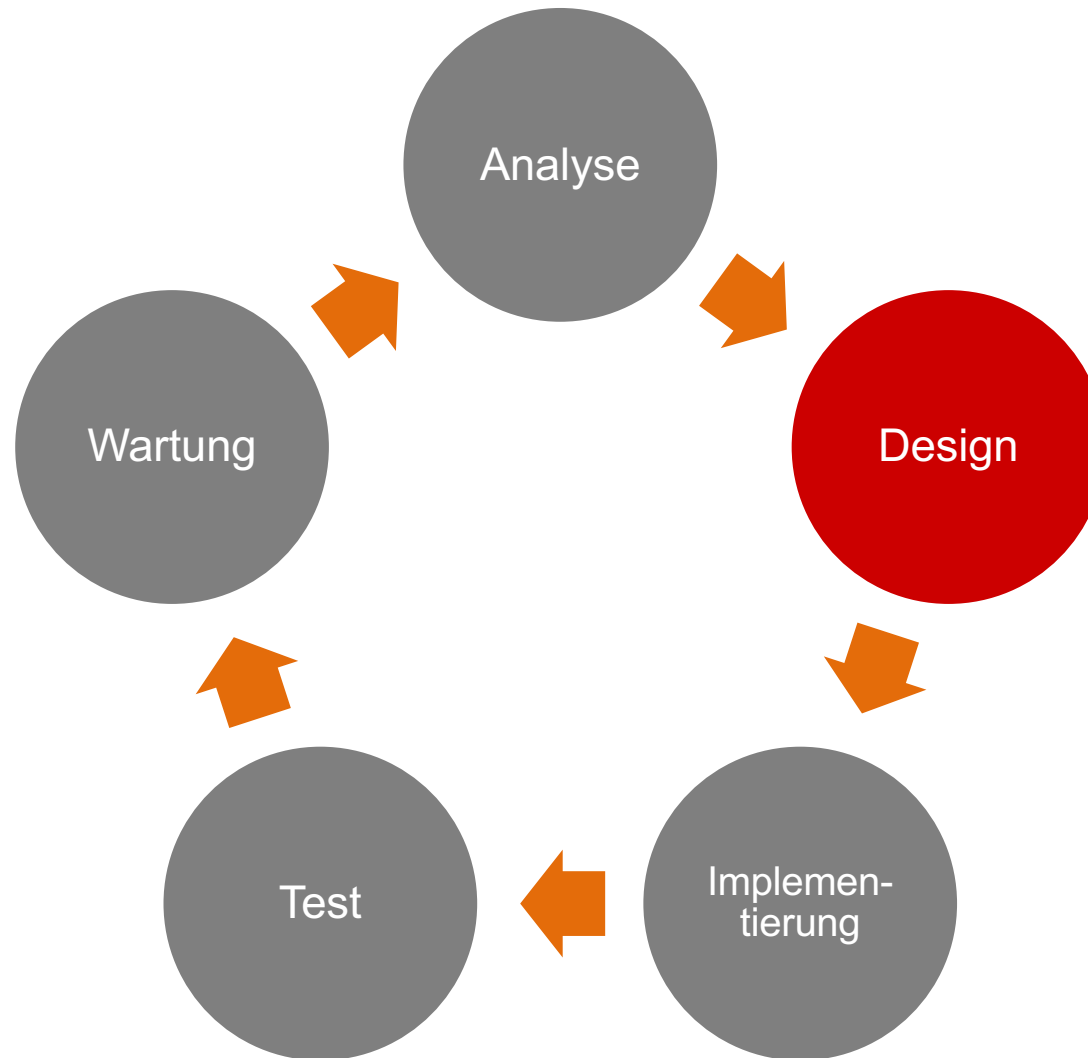
# Übersicht



# Anforderungsanalyse

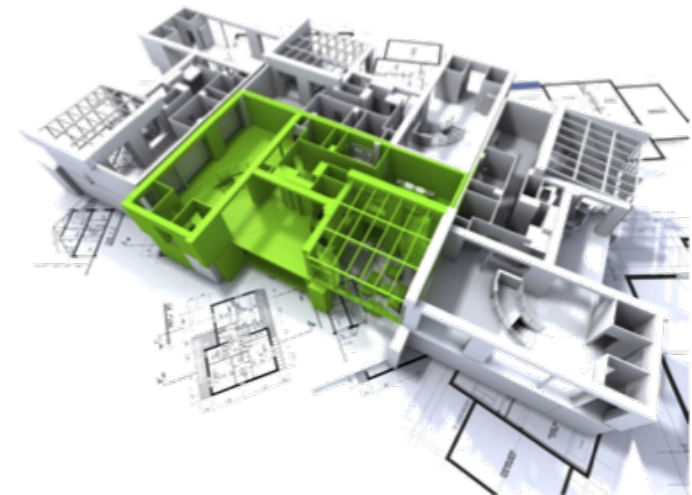
- Was möchte der Auftraggeber?
- Sammlung aller Anforderungen
  - Anwender und Entwickler haben oft nicht die gleiche Sichtweise
- Überprüfung der Anforderungen
  - Machbarkeit, Abhängigkeiten, Konsistenz
- Sorgfältige Analyse ist wichtig, da sich Fehler über den gesamten Entwicklungszyklus erstrecken und enorme Kosten verursachen können.

# Übersicht

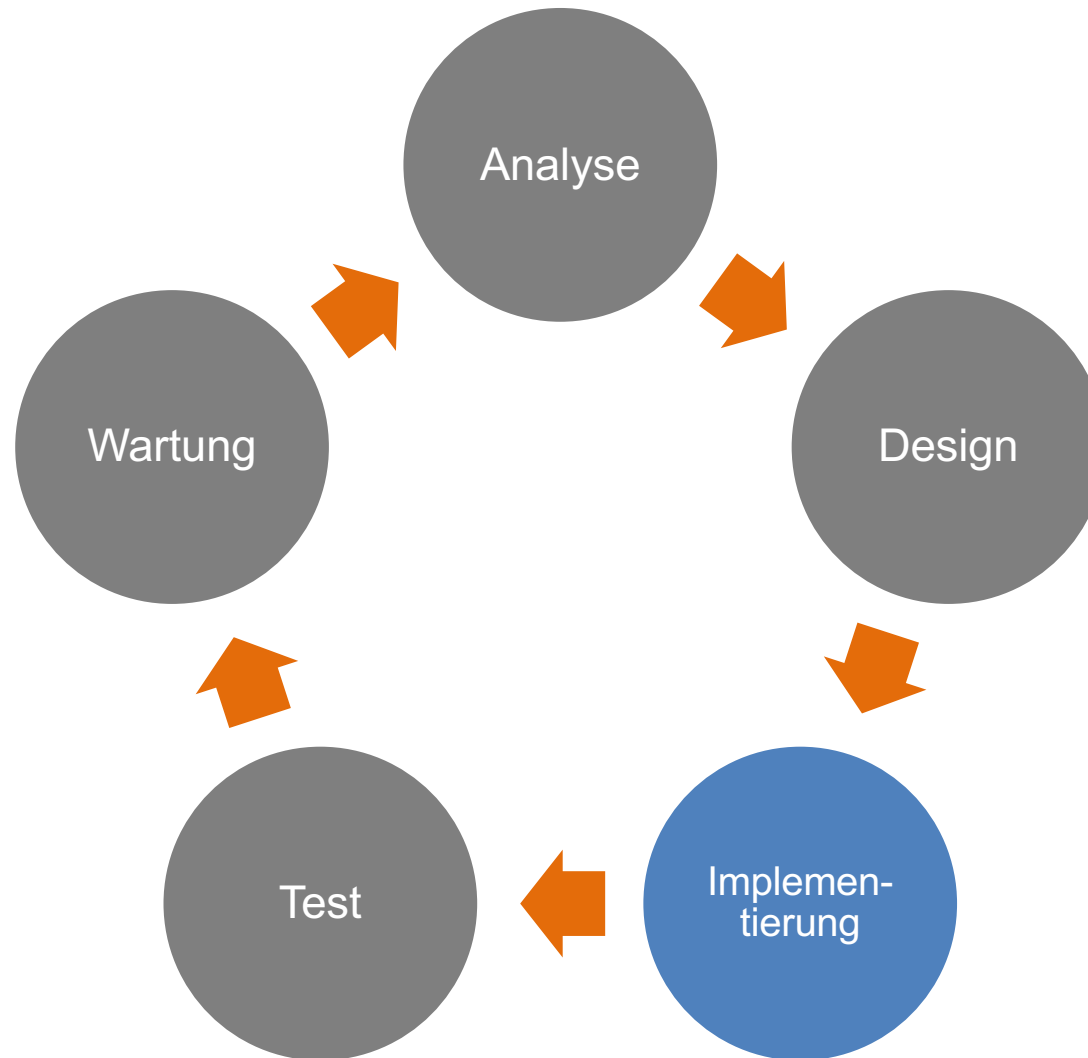


# Softwaredesign

- Programmieren im Kleinen: Implementierung eines Algorithmus in Java, Aufteilung in Methoden
- Programmieren im Großen: Entwurf der Systemarchitektur
  - Bestimmung von Komponenten des Systems (Modularisierung)
  - Definition von **Schnittstellen** zwischen den Komponenten



# Übersicht



# Implementierung

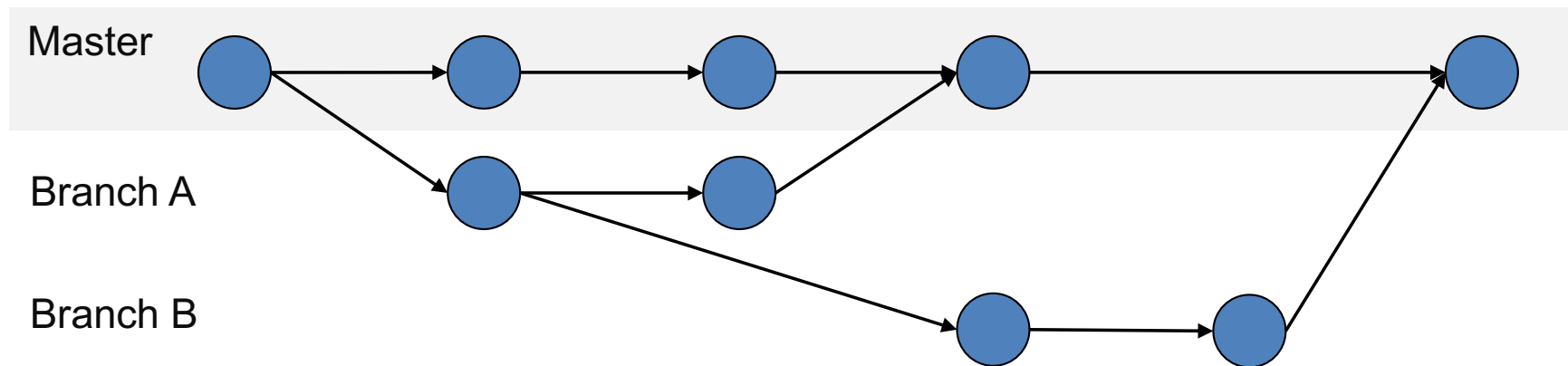
- Nach der Designphase sind die wichtigen Schnittstellen definiert
- Implementierung erfolgt üblicherweise unabhängig voneinander (oft sogar in getrennten Teams)
- Klare Trennung der Komponenten mit Hilfe von Schnittstellen erleichtert die parallele Entwicklung
- Aber: die Komponenten an sich können auch wieder sehr umfangreich sein und mehrere Personen beschäftigen

→ Mehrere Personen arbeiten am gleichen Code



# Versionsverwaltung

- Software wird oft in großen Teams und an verschiedenen Orten gleichzeitig entwickelt
  - gemeinsame Code-Basis und Synchronisierung notwendig, um **Inkonsistenzen** zu vermeiden
- Änderungen der Software sollen nachvollziehbar und ggf. auch reversibel sein



# Versionsverwaltungssysteme

- **Git** ist das mit Abstand am verbreitetste Tool
  - Dezentrales System ohne notwendige Client-Server-Struktur
  - Änderungen werden feingranular lokal versioniert (*commits*) und dann mit Teammitgliedern synchronisiert
  - Konflikte werden über *merges* gelöst
- github.com ist eine populäre Git-Hosting-Plattform
  - Frei nutzbar für Open-Source-Projekte
  - 24 Mio. Benutzer (5 Mio. in Europa)
  - 67 Mio. Repositories (25 Mio. aktiv in 2017)
    - Linux-Kernel mit über 700k Commits, Git selbst
    - Microsoft VSCode mit über 15k Beteiligungen

# Implementierung mit einer IDE

- Bisher:
  - Entwicklung mit Hilfe eines Editors (z.B. Notepad++)
  - Übersetzen mit *javac* auf der Konsole
  - Ausführen mit *java* auf der Konsole
  - Auf Dauer sehr lästige Vorgehensweise!
- Abhilfe: Integrierte Entwicklungsumgebungen (IDE)
  - Umfangreiche Werkzeuge für die Entwicklung von Programmen
  - Bekannte IDEs für Java: **IntelliJ IDEA**, Eclipse, NetBeans

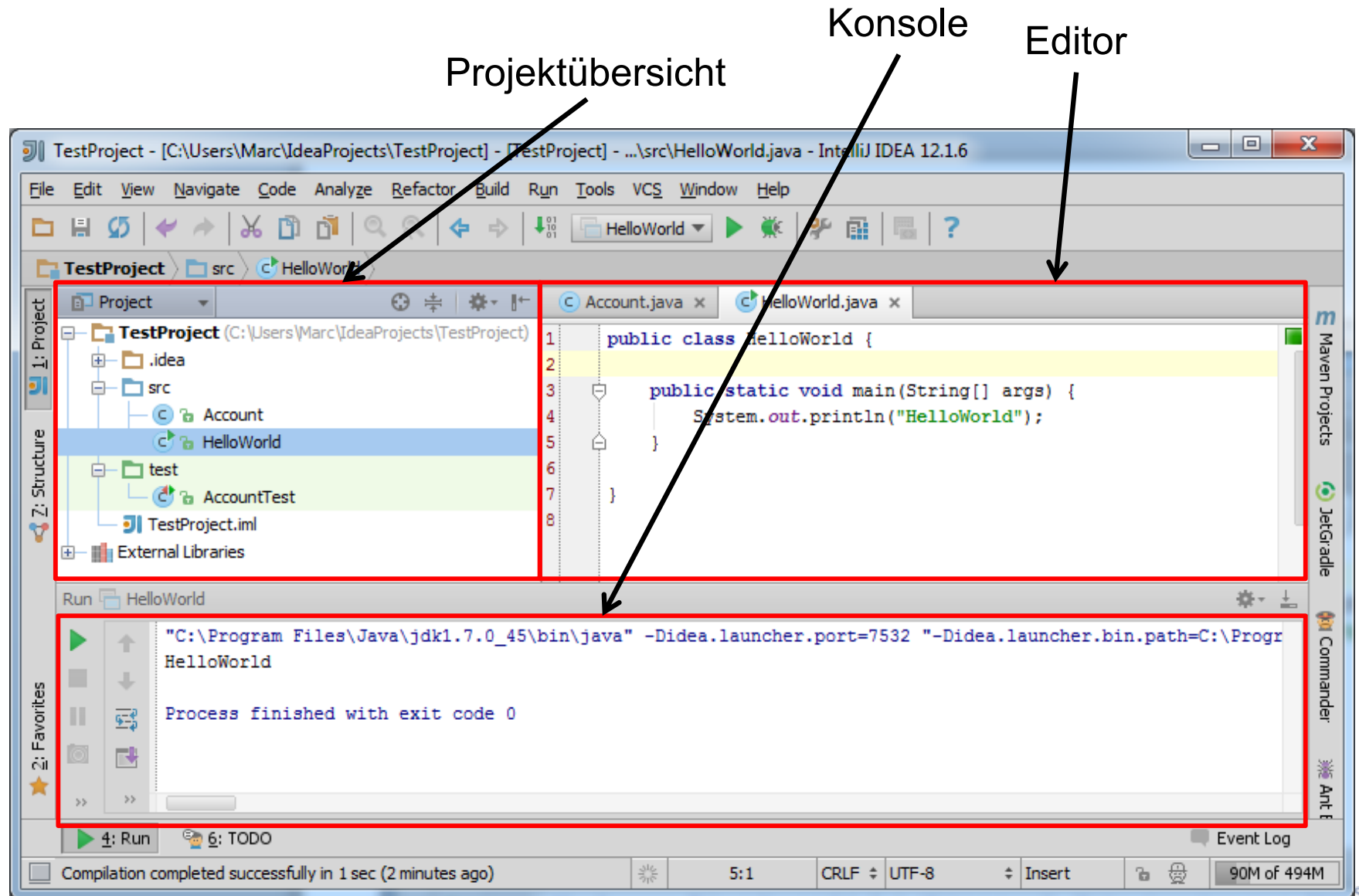


# IntelliJ IDEA

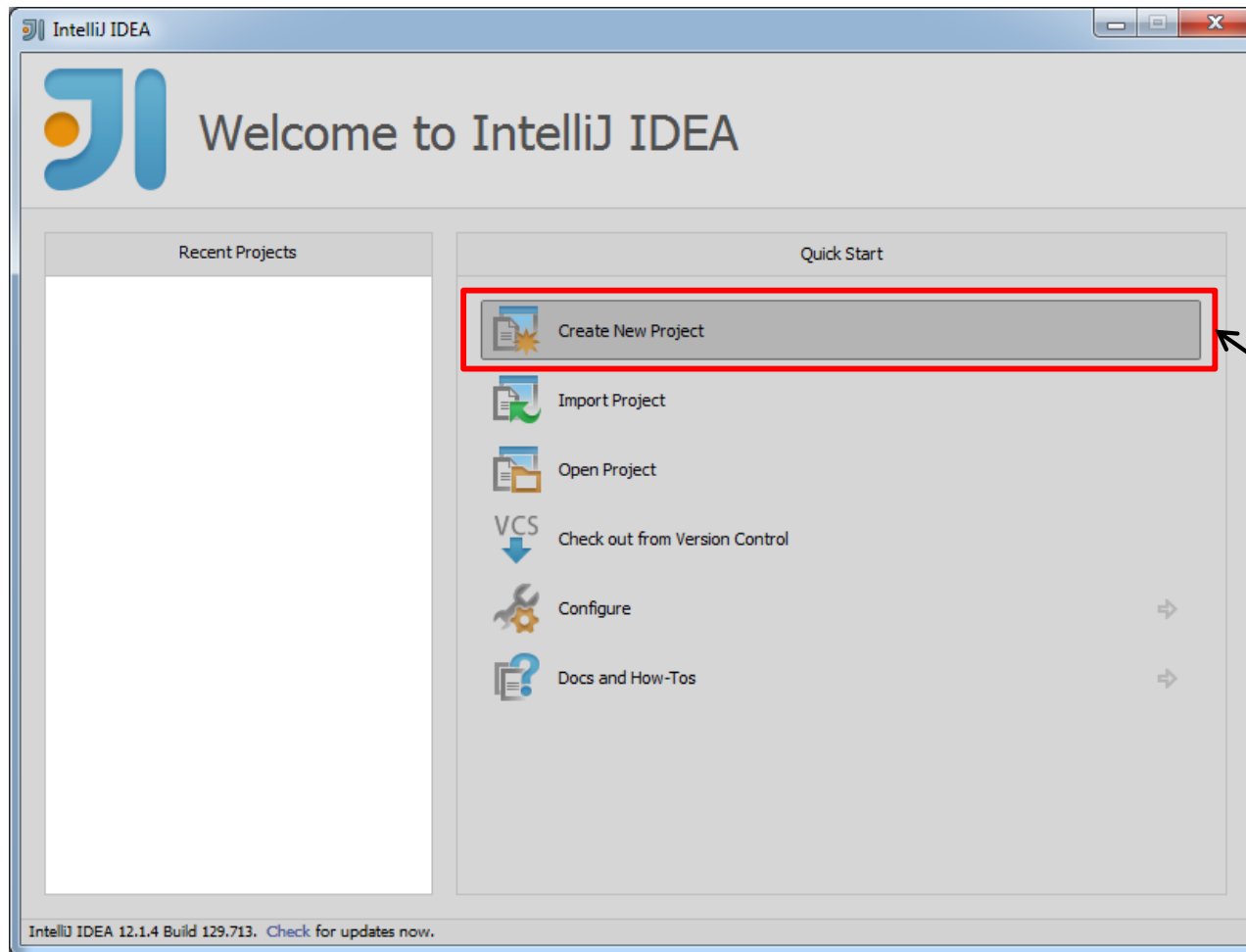
- Wir werden zukünftig die Open-Source Community Edition von IntelliJ IDEA (kurz: IntelliJ) nutzen (<http://www.jetbrains.com/idea/>)
  - Verwaltet Programmcode in Projekten
    - Quellcode liegt im Verzeichnis „src“
  - Zeigt Fehlermeldungen des Compiler direkt im Editor an
  - Übersetzt den Programmcode automatisch
  - Autovervollständigung im Editor
  - Integrierte Konsole
  - ... und weitere nützliche Features

Hilft nicht in der Klausur. Deshalb sollte man sich auf diese Funktionalität nicht verlassen.

# IntelliJ: Übersicht

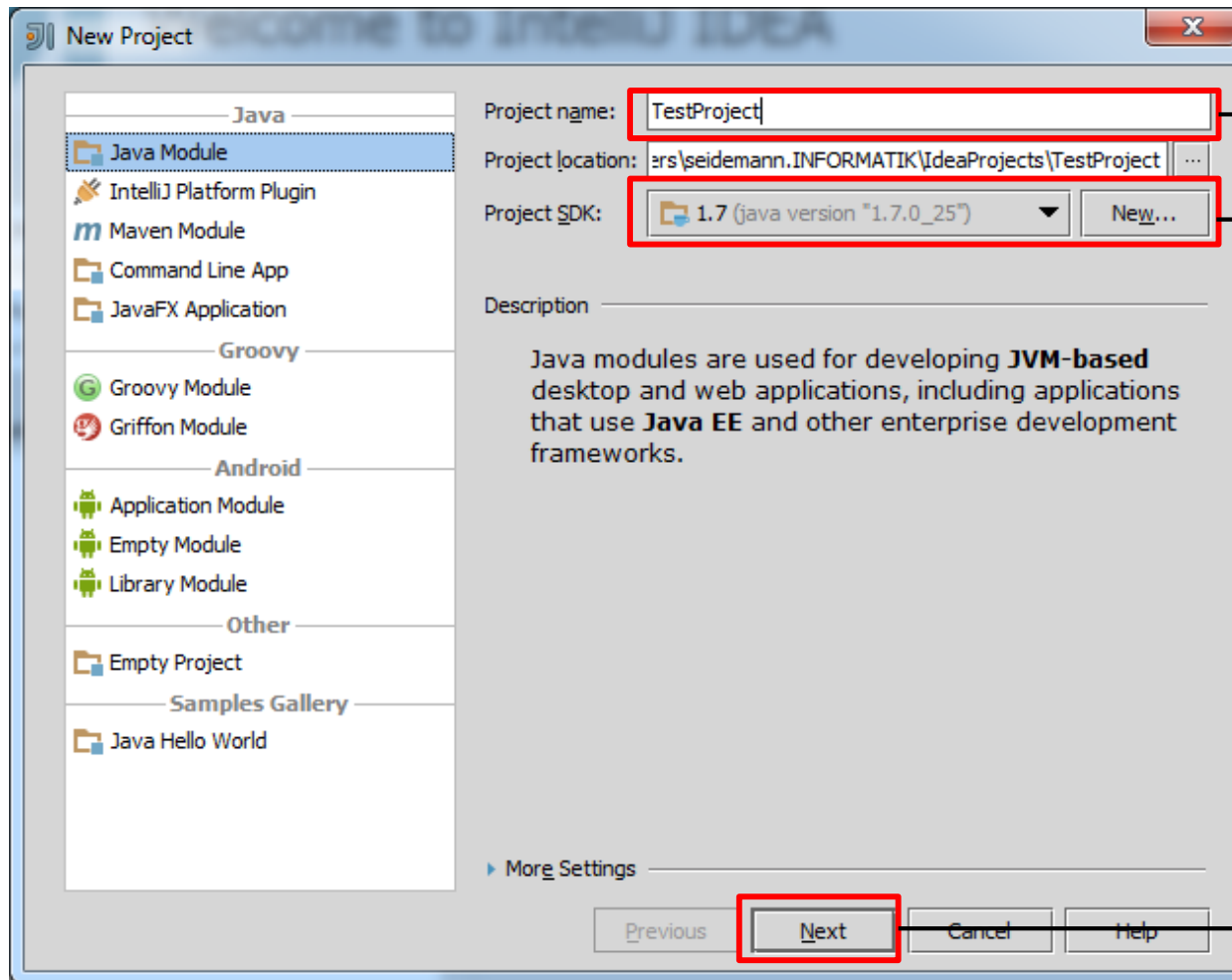


# IntelliJ: Einrichtung



Neues Projekt erzeugen

# IntelliJ: Einrichtung

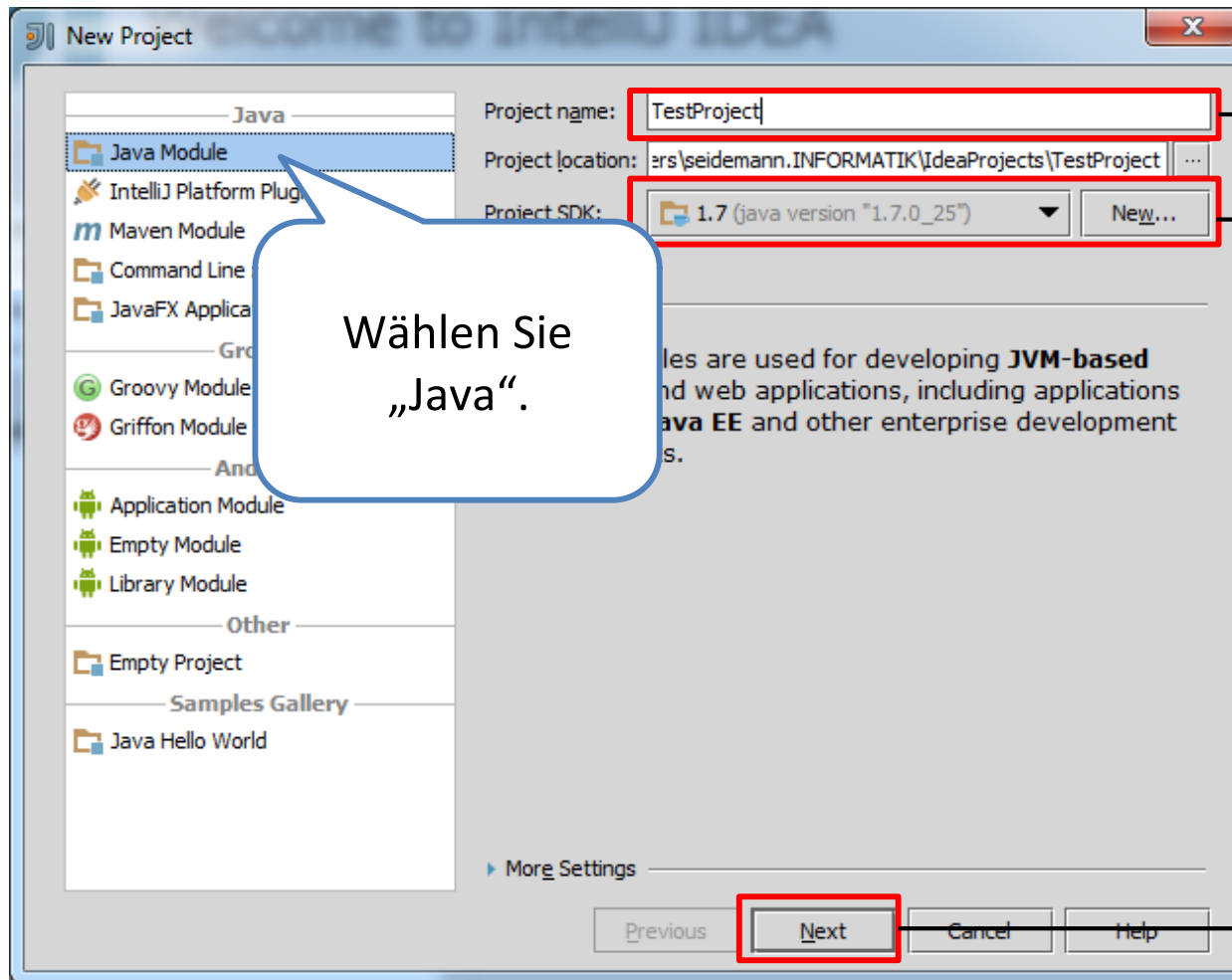


→ Name des Projekts

→ Falls nicht vorhanden:  
über „New...“ den Pfad  
zum JDK-Verzeichnis  
auswählen

→ „Next“ und im  
anschließenden Fenster  
„Finish“ klicken

# IntelliJ: Einrichtung



Wählen Sie  
„Java“.

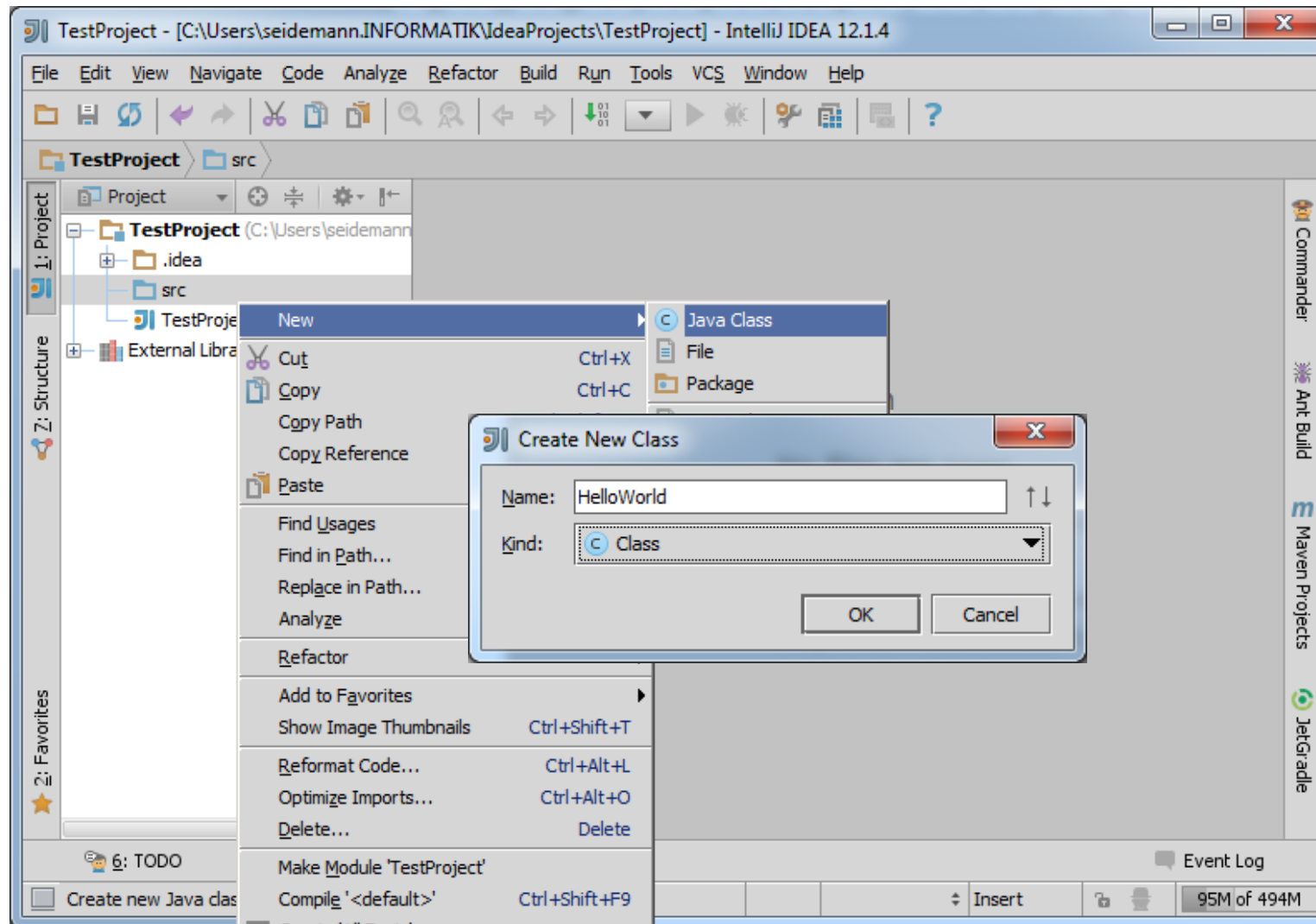
Name des Projekts

Falls nicht vorhanden:  
über „New...“ den Pfad  
zum JDK-Verzeichnis  
auswählen

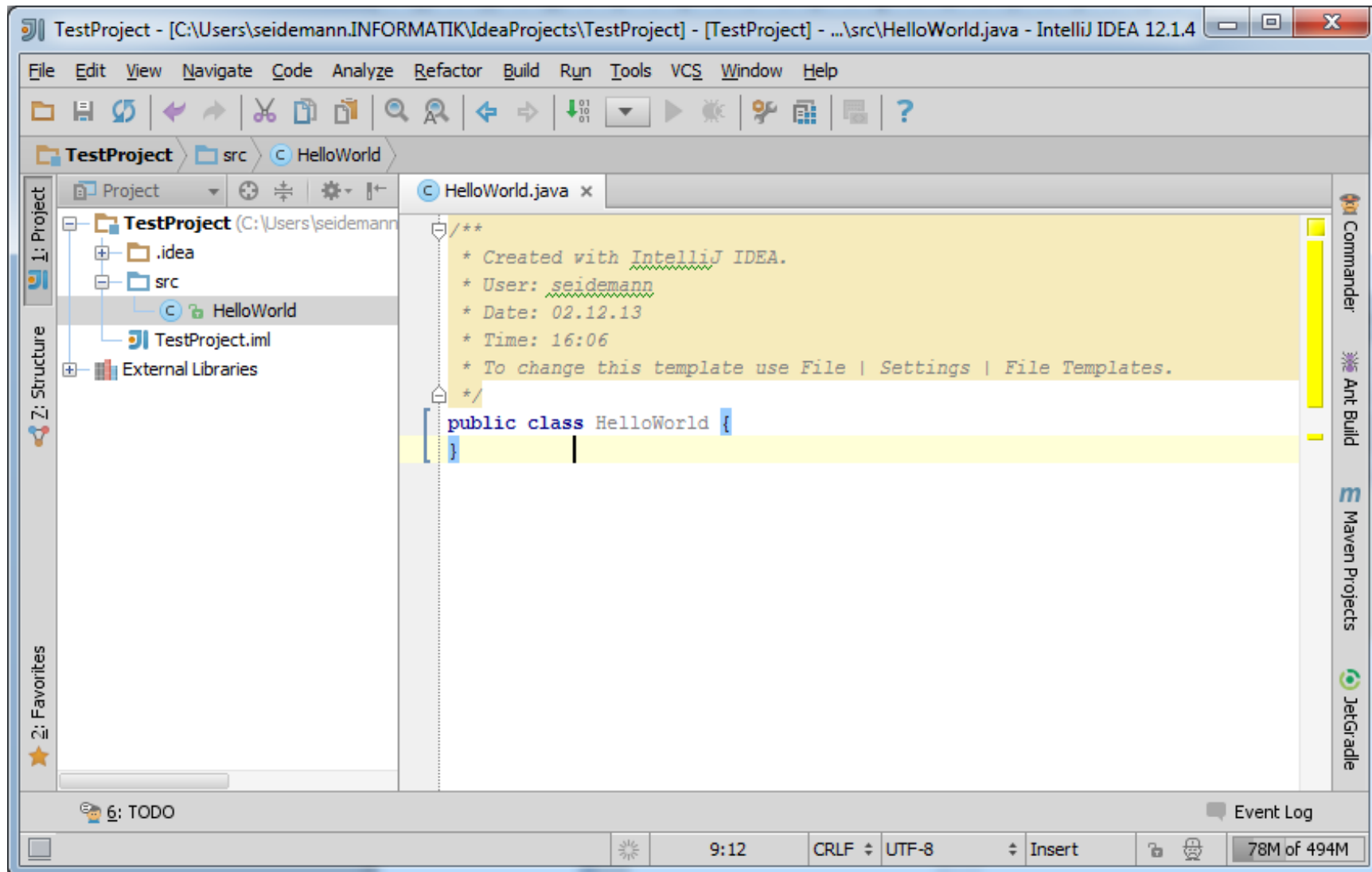
„Next“ und im  
anschließenden Fenster  
„Finish“ klicken



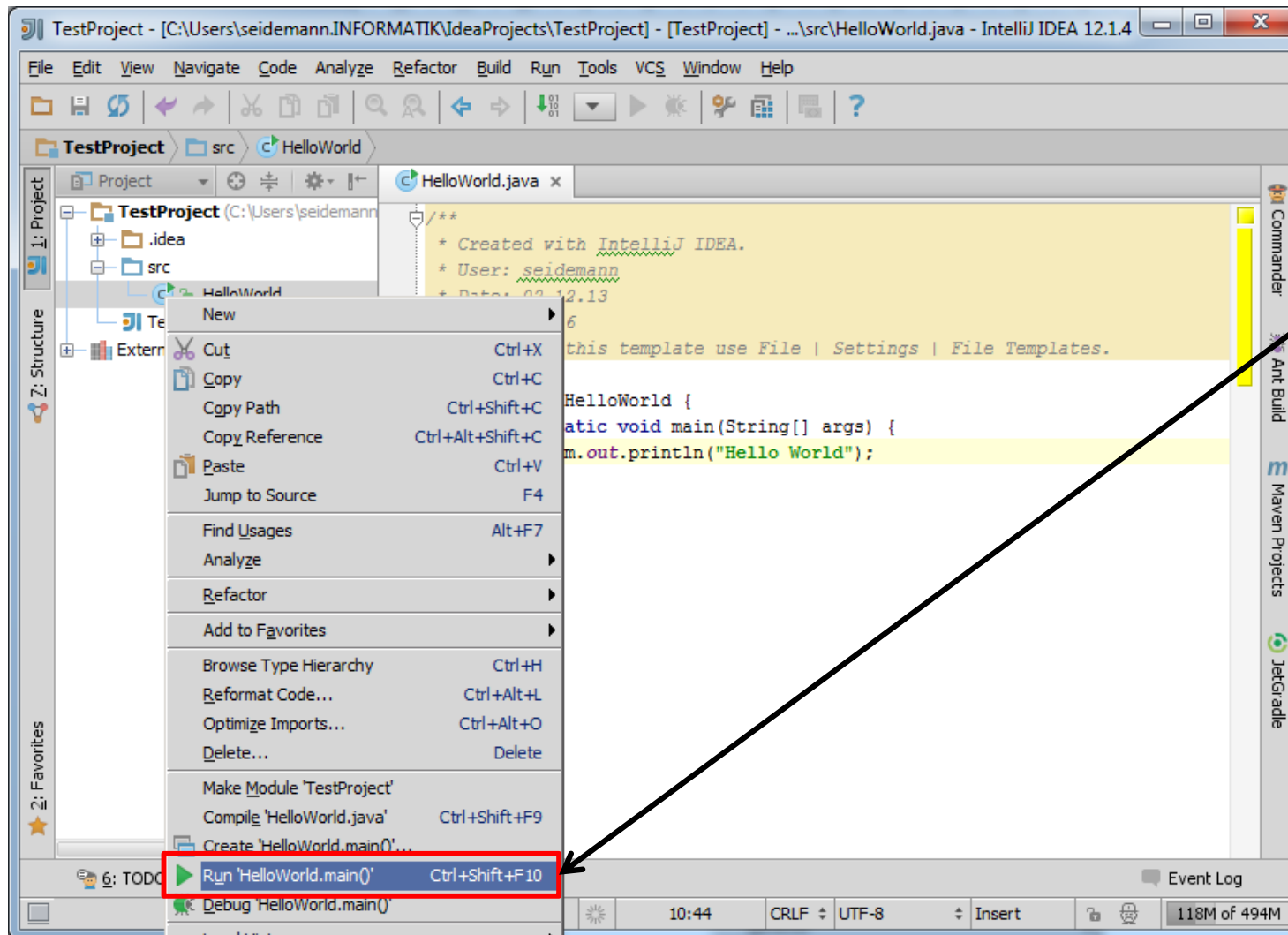
# IntelliJ: HelloWorld



# IntelliJ: HelloWorld

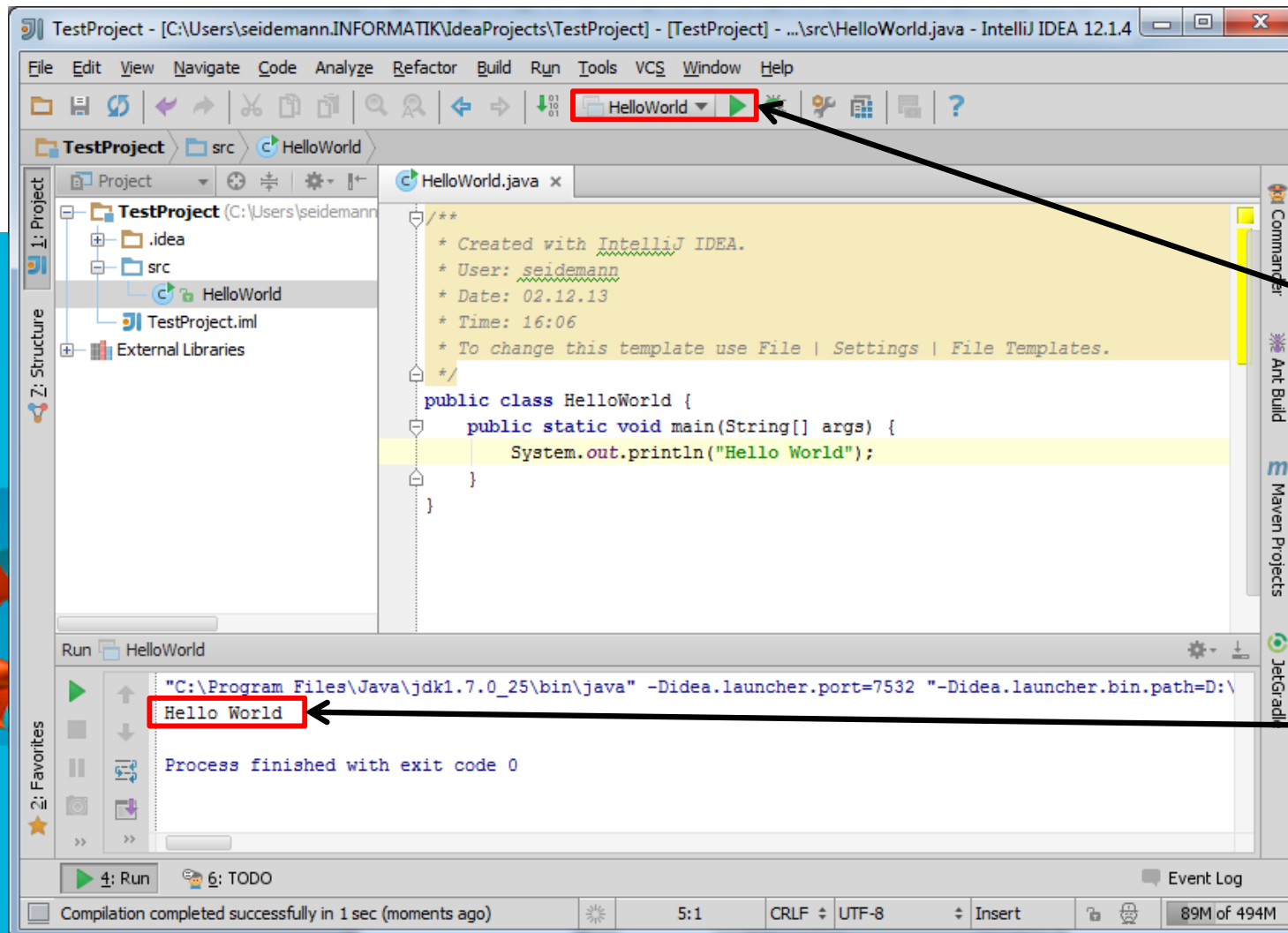


# IntelliJ: Programm ausführen



Programm  
ausführen  
([STRG]+[SHIFT]+[F10])

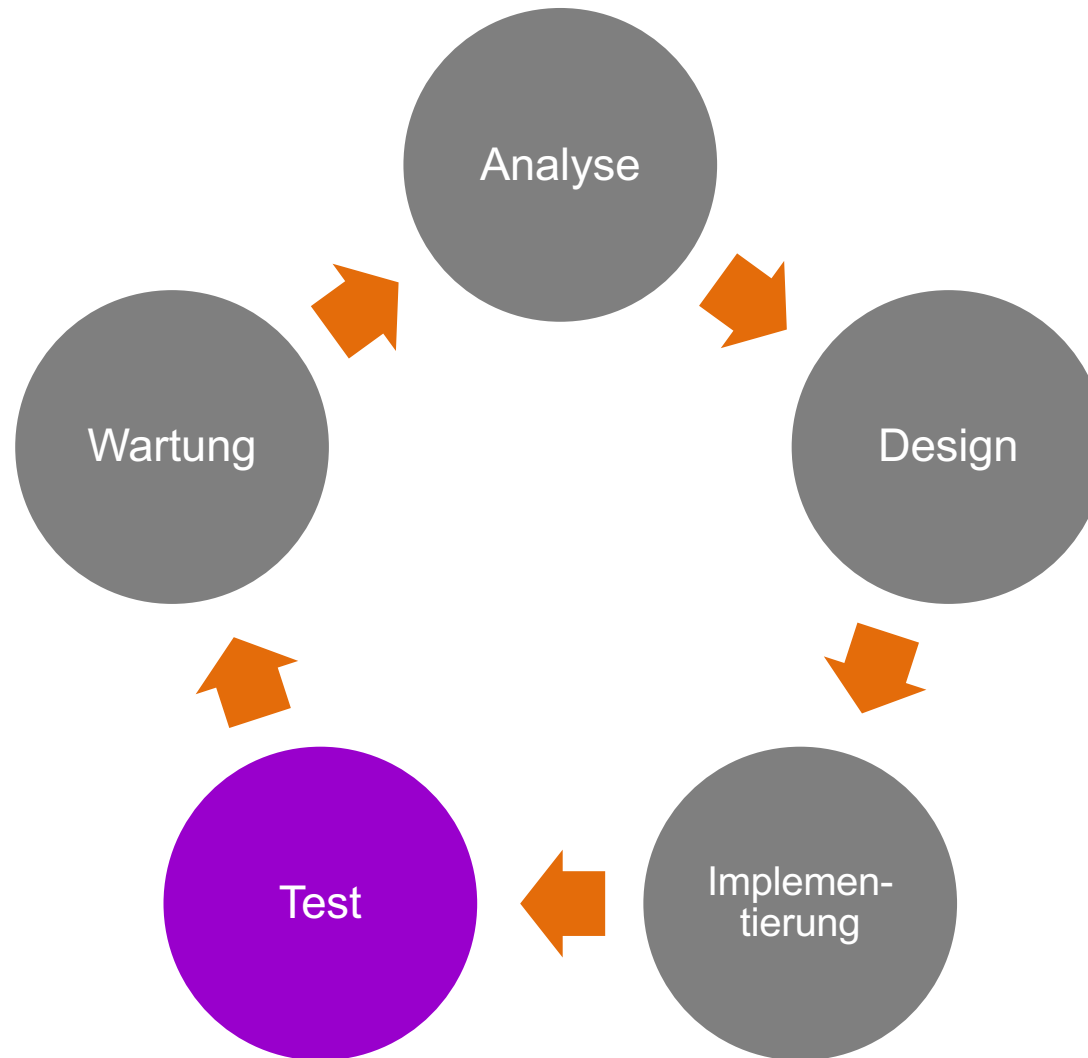
# IntelliJ: Programm ausführen



Programm ausführen  
neue Run-Configuration

Ausgabe auf  
der Konsole

# Übersicht



# Testen

- Auf 1000 Zeilen Programmcode fallen durchschnittlich 1-25 Fehler
- Wird ein Fehler erst nach der Auslieferung der Software gefunden, kostet die Behebung nicht selten das **zehnfache**
- Oder auch mehr...

**Start einer Ariane 5 (1996):**

Fehlerhafte Umwandlung einer 64-bit Gleitkommazahl in einen 16-bit Short-Wert führte zu einem Ausfall der Steuerung

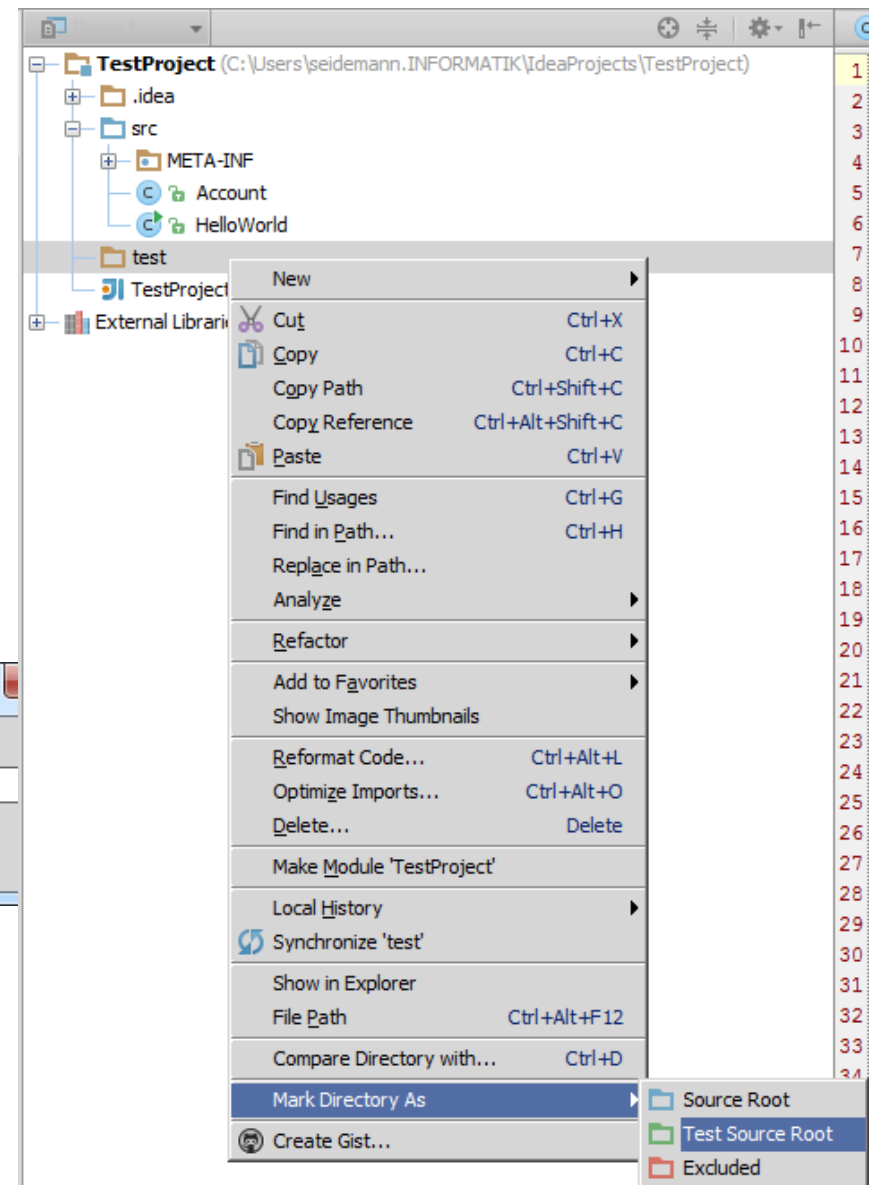
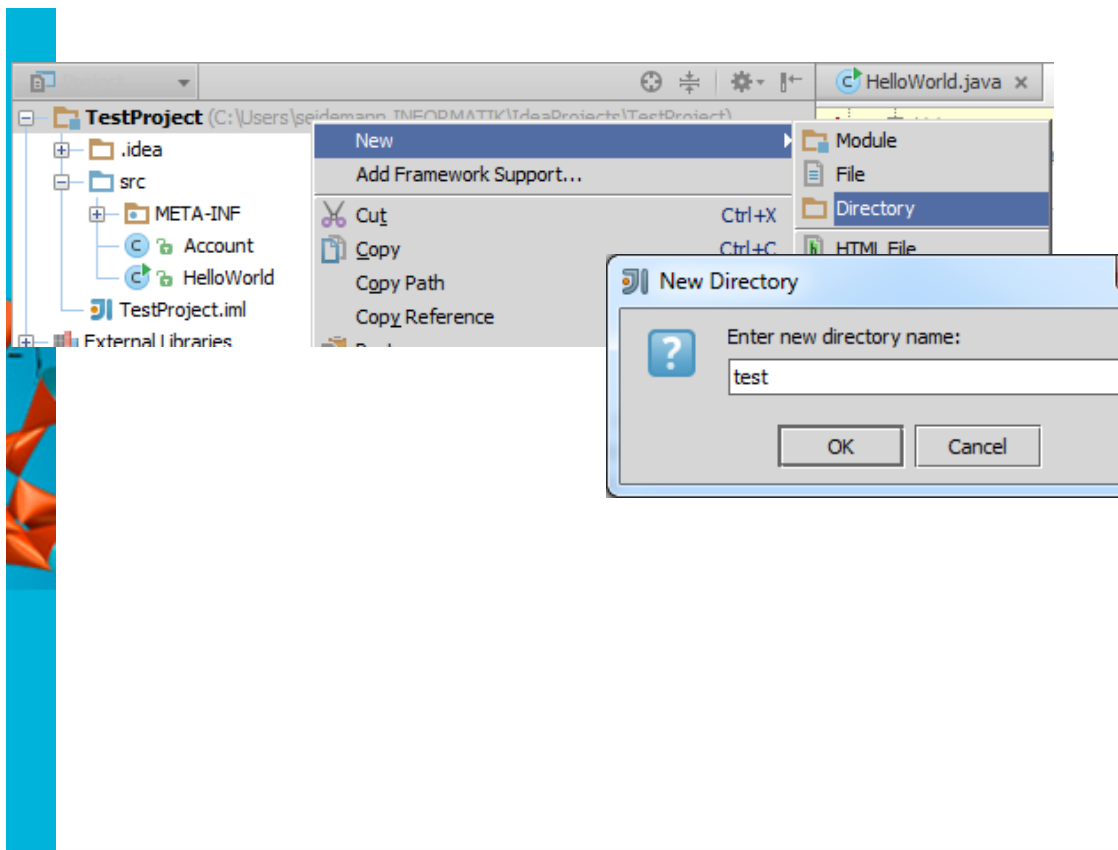


# Unit-Tests

- Test einer kleinen Einheit (z.B. Klasse) für sich
- Für jede Klasse und jede Methode ein Test
  - mit Ausnahme von trivialen Methoden
- Prinzipien
  - Jeder Test spezifiziert ein unabhängiges und wiederholbares Szenario.
  - Vorbedingungen und Nachbedingungen werden in Form von **Assertions** geprüft.
  - Unit-Tests laufen selbstständig ab und besitzen keine Interaktion mit dem Anwender oder anderen Komponenten (idealerweise).

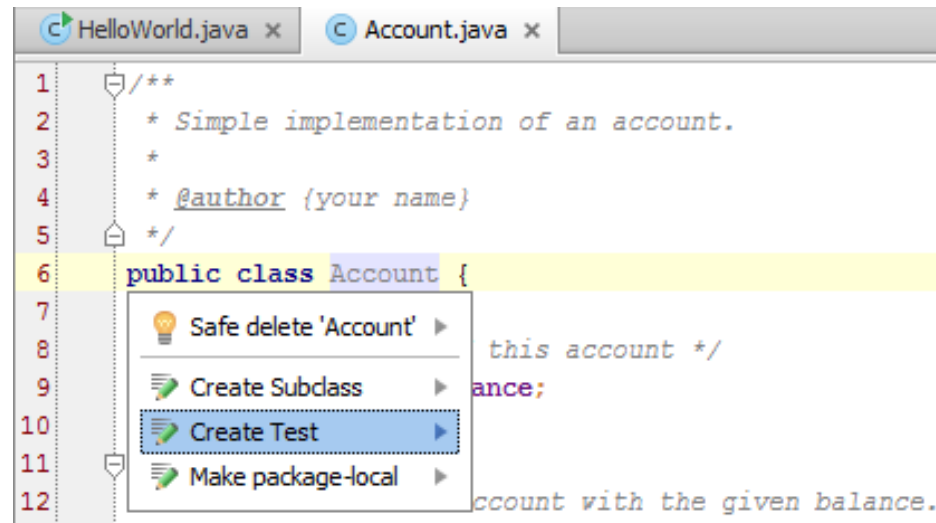
# JUnit

Zuerst:  
Verzeichnis für  
Tests erstellen





# JUnit

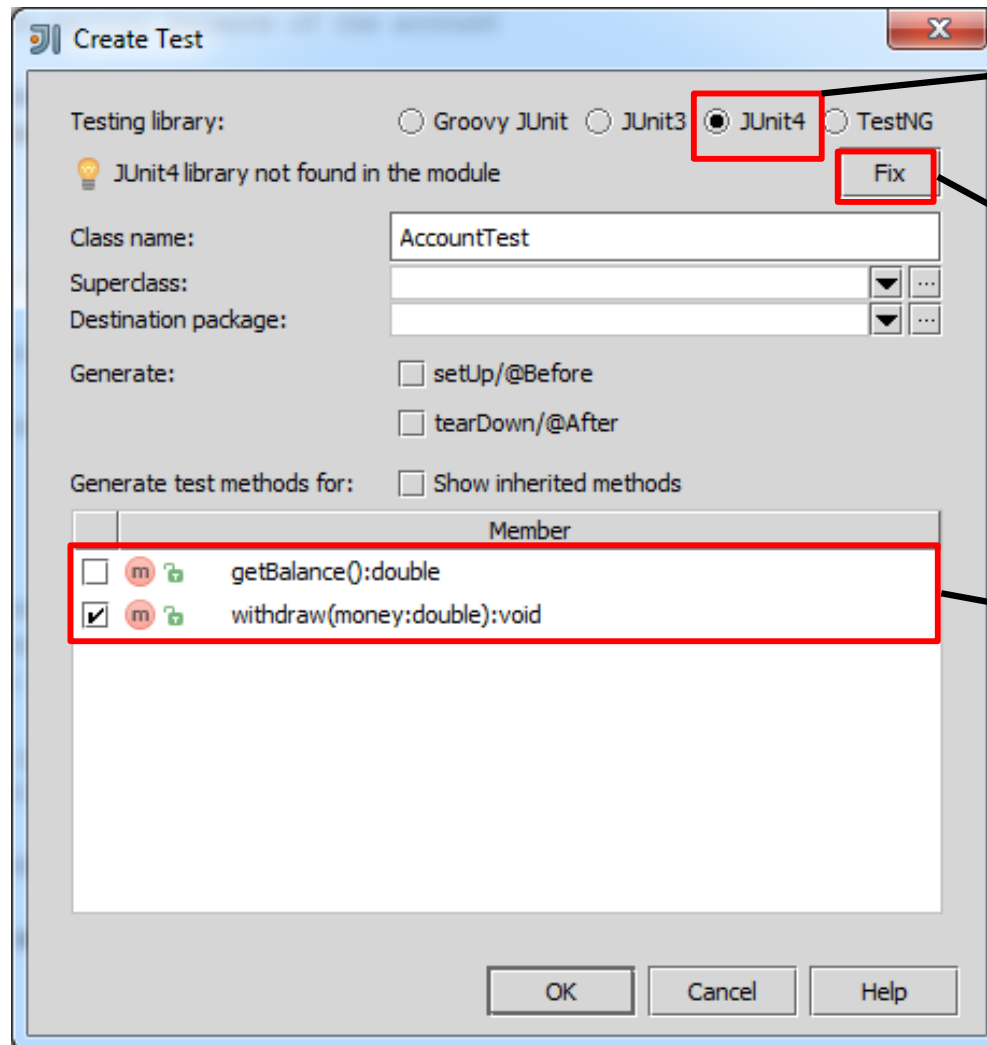


```
1  /**
2   * Simple implementation of an account.
3   *
4   * @author {your name}
5   */
6  public class Account {
7
8      this account */
9
10     ance;
11
12     account with the given balance.
```

## Testklasse erstellen

- Cursor auf Klassennamen
- [ALT]+[ENTER]
- Create Test

# JUnit

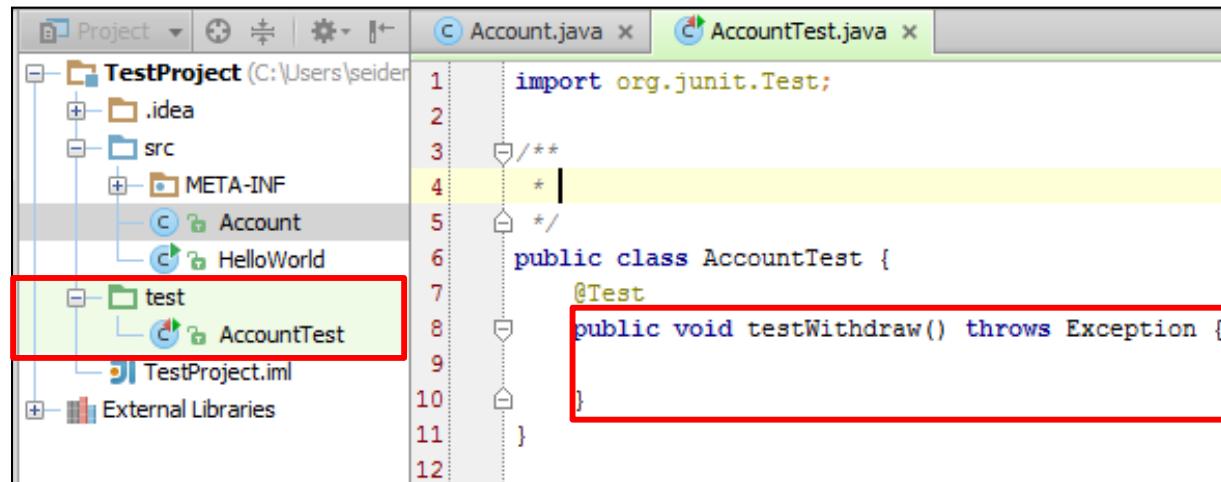


Wir benutzen **JUnit4**

Beim ersten Mal: **Fix** klicken

Auswahl der Methoden, für die eine Testmethode erstellt werden soll

# JUnit

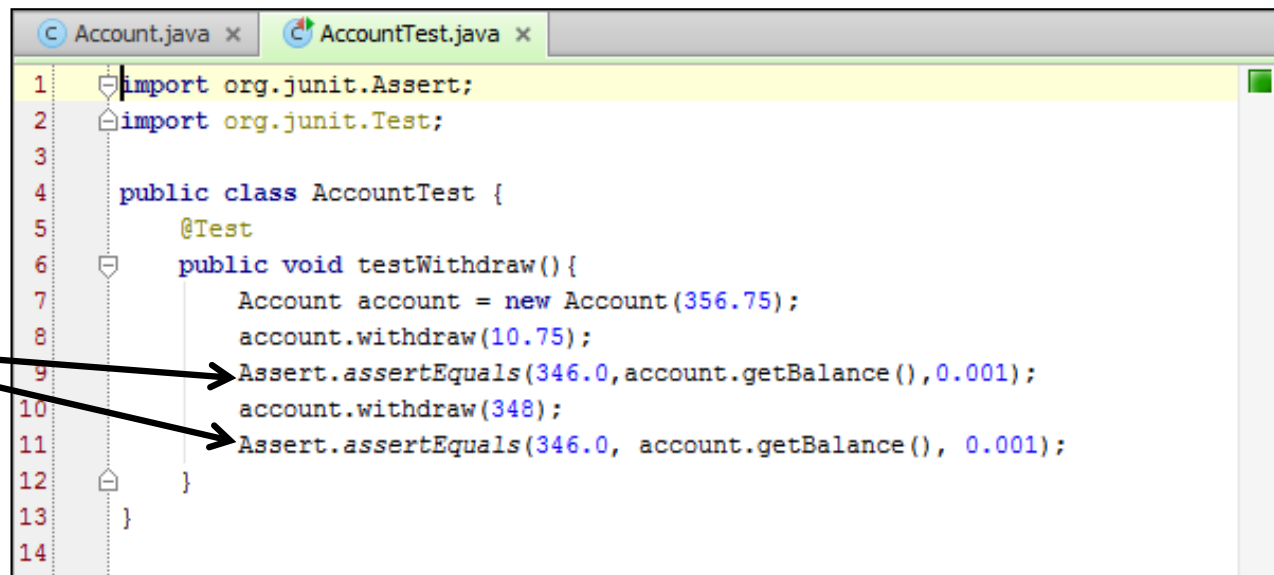


Die Testklasse befindet sich nun im Ordner „test“

# JUnit

- Weitere Assertions
  - Assert.assertFalse(boolean condition)
  - Assert.assertTrue(boolean condition)
  - ...

Überprüfung der  
Nachbedingung



```
1 import org.junit.Assert;
2 import org.junit.Test;
3
4 public class AccountTest {
5     @Test
6     public void testWithdraw() {
7         Account account = new Account(356.75);
8         account.withdraw(10.75);
9         Assert.assertEquals(346.0, account.getBalance(), 0.001);
10        account.withdraw(348);
11        Assert.assertEquals(346.0, account.getBalance(), 0.001);
12    }
13 }
14
```

**Assert.assertEquals**(double expected, double actual, double difference)

# JUnit

- Weitere Assertions
  - Assert.assertFalse(boolean condition)
  - Assert.assertTrue(boolean condition)
  - ...

Überprüfung der  
Nachbedingung

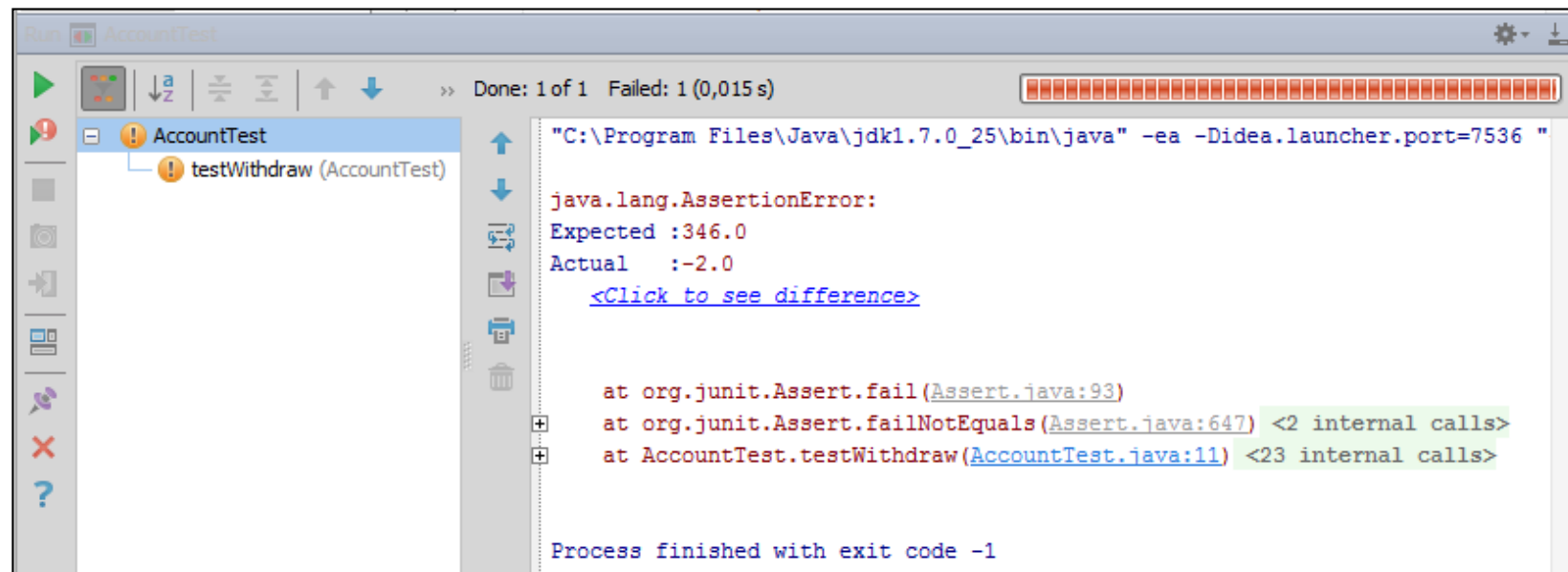
```
1 import org.junit.Assert;
2 import org.junit.Test;
3
4 public class AccountTest {
5     @Test
6     public void testWithdraw() {
7         Account account = new Account(356.75);
8         account.withdraw(10.75);
9         Assert.assertEquals(346.0, account.getBalance(), 0.001);
10        account.withdraw(348);
11        Assert.assertEquals(account.getBalance(), 0.001);
12    }
13 }
```

Die Methode assertEquals und ähnliche  
existiert auch für andere Datentypen.

**Assert.assertEquals**(double expected, double actual, double difference)

# JUnit

- Test ausführen: [STRG]+[SHIFT]+[F10]



```
public void withdraw(double money) {  
    this.balance -= money;  
}
```

Negative Kontostände sind  
nicht erlaubt!

# Integrationstests

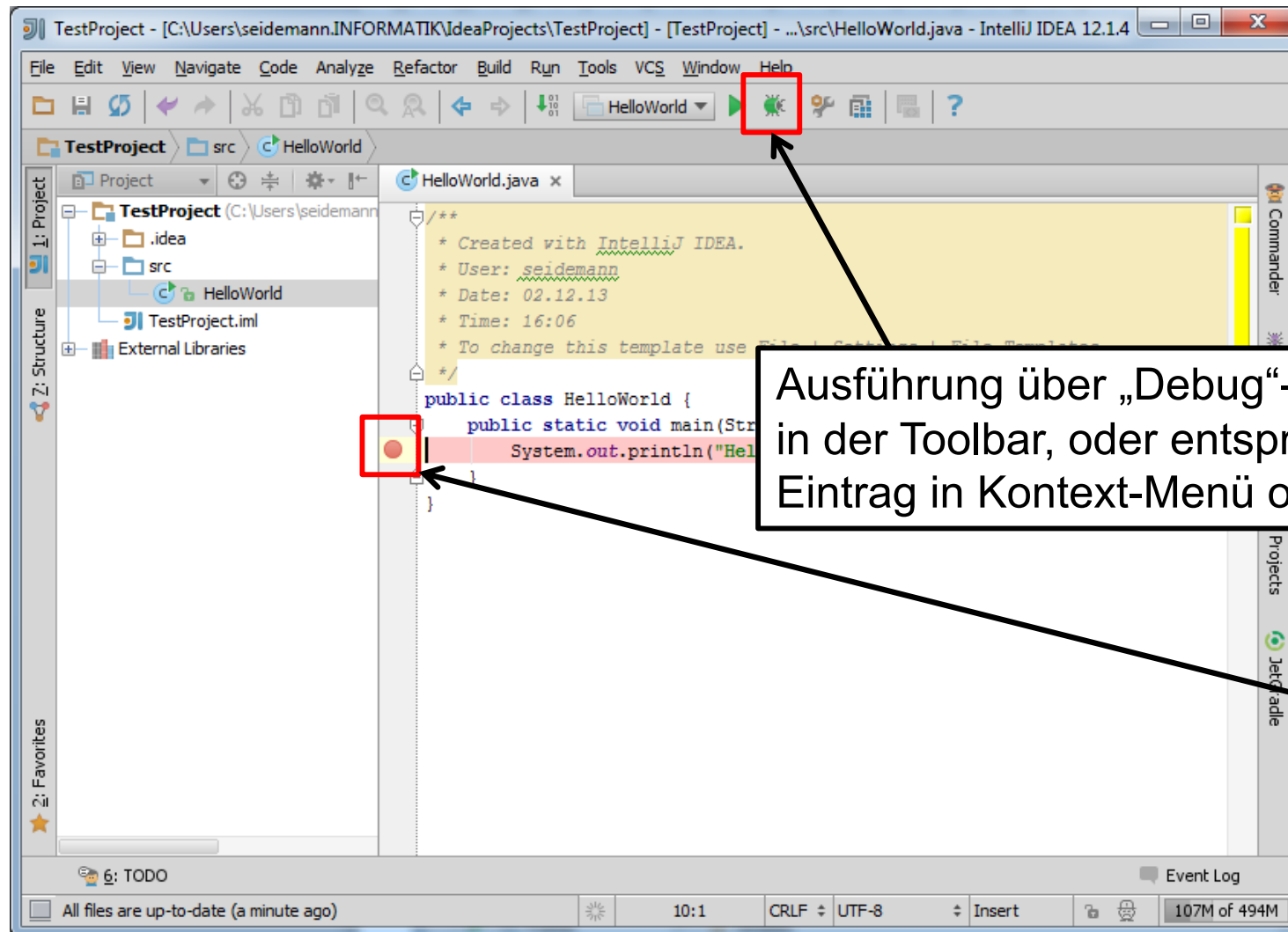
- Unit-Tests alleine genügen nicht!
  - Eine Software muss immer auch im Zusammenspiel mit externen Schnittstellen betrachtet werden (Klassen-übergreifend, Festplatten, Netzwerk, ...)
- Integrationstests dienen dazu, eine Komponente im Zusammenspiel mit anderen Komponenten zu testen
- Andere Komponenten sind durch externe Faktoren beeinflusst
  - Eine Methode möchte eine Datei auf der Festplatte anlegen
  - Festplatte wird auch von anderen Programmen verwendet
  - Zum Beispiel: Test geht 100x gut, beim 101-ten Test ist jedoch die Festplatte voll

# Wenn es schief geht: Debugging

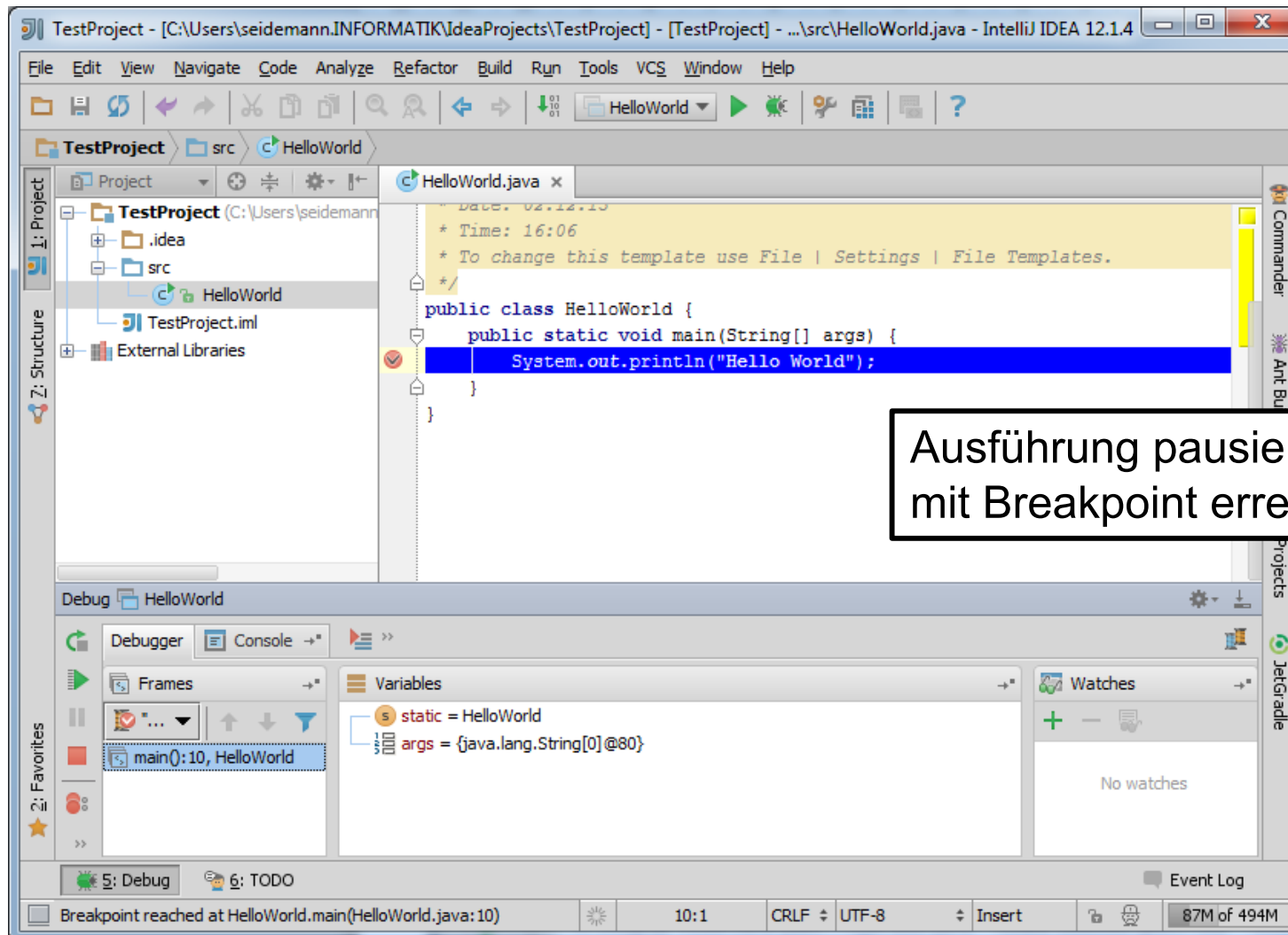
- Schlägt ein Testfall fehl, beginnt die Fehlersuche
- Old School:
  - Ausgabe von Variablen auf der Konsole → sehr umständlich
  - Nicht empfehlenswert: das Programm wird zur Fehlersuche modifiziert!
- Debugger
  - Teil der Entwicklungsumgebung
  - Kann mit Hilfe von **Breakpoints** (Haltepunkten) die Ausführung des Programms gezielt an einer gewünschten Stelle unterbrechen
  - Nachvollziehen des Programmflusses
  - Navigation in der Programmausführung (Hinein- und Herausspringen aus Methoden)
  - Beobachtung und ad-hoc-Veränderung von Variablen während der Laufzeit



# Debugger in IntelliJ

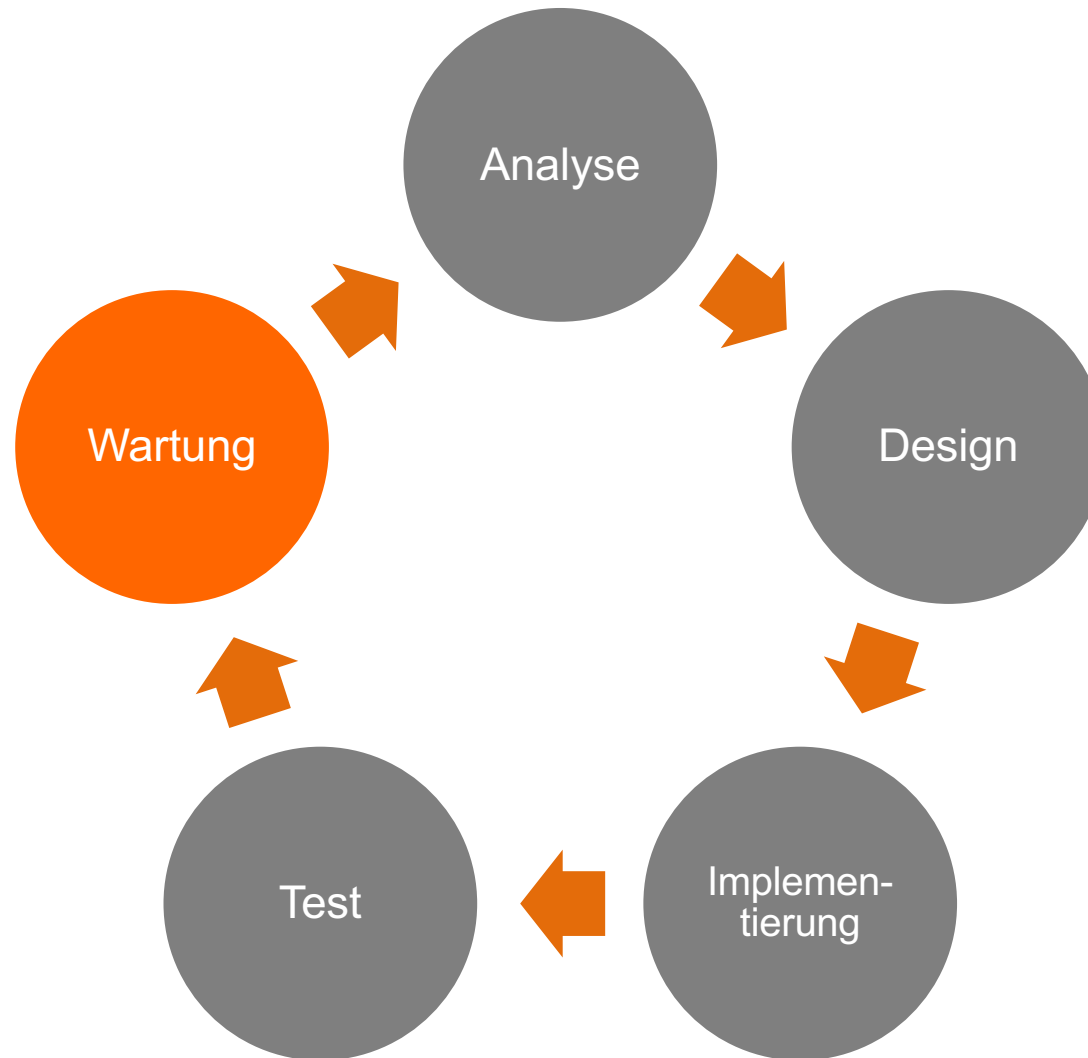


# Debugger in IntelliJ



Ausführung pausiert sobald Zeile mit Breakpoint erreicht wird.

# Übersicht



# Auslieferung: JAR-Dateien

- Nachdem alle Tests erfolgreich bestanden wurden, kann das Programm ausgeliefert werden
- In Java: ausführbare **JAR-Datei** erzeugen
  - JAR-Dateien sind im Wesentlichen ZIP-Dateien, die neben den übersetzten Class-Dateien eine *Manifest-Datei* enthält
  - Manifest-Datei spezifiziert Metadaten, wie z.B. den Classpath und die Klasse, die die **main**-Methode enthält
- Ausführung der JAR-Datei:

```
java -jar MeineJar.jar
```

# JAR-Manifest

- Die Metadaten befinden sich in der Datei „META-INF/MANIFEST.MF“
- Auszug aus einer Manifest-Datei

Manifest-Version: 1.0  
**Main-Class:** HelloWorld  
Class-Path: <Jar1>

enthält die **main**-Methode

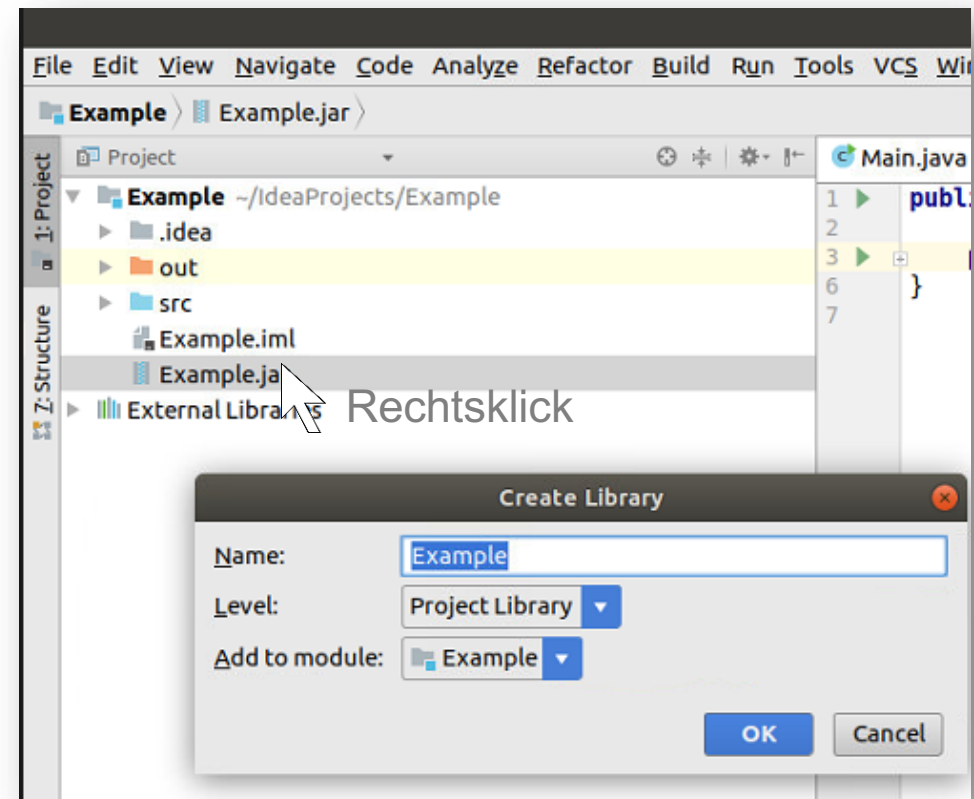
# Bibliotheken

- JAR-Dateien dienen nicht nur als ausführbare Programme, sondern auch als Bibliotheken
  - Idee dahinter: nicht ständig das Rad neu erfinden, sondern bestehenden Code nutzen
  - Klassen werden in JAR-Dateien gebündelt und können in anderen Projekten eingebunden und genutzt werden
- Problem
  - Wir müssen dem Java-Compiler mitteilen, in welchen Bibliotheken (JAR-Dateien) die referenzierten Klassen liegen.
  - CLASSPATH muss gesetzt werden (siehe auch Kapitel 6)

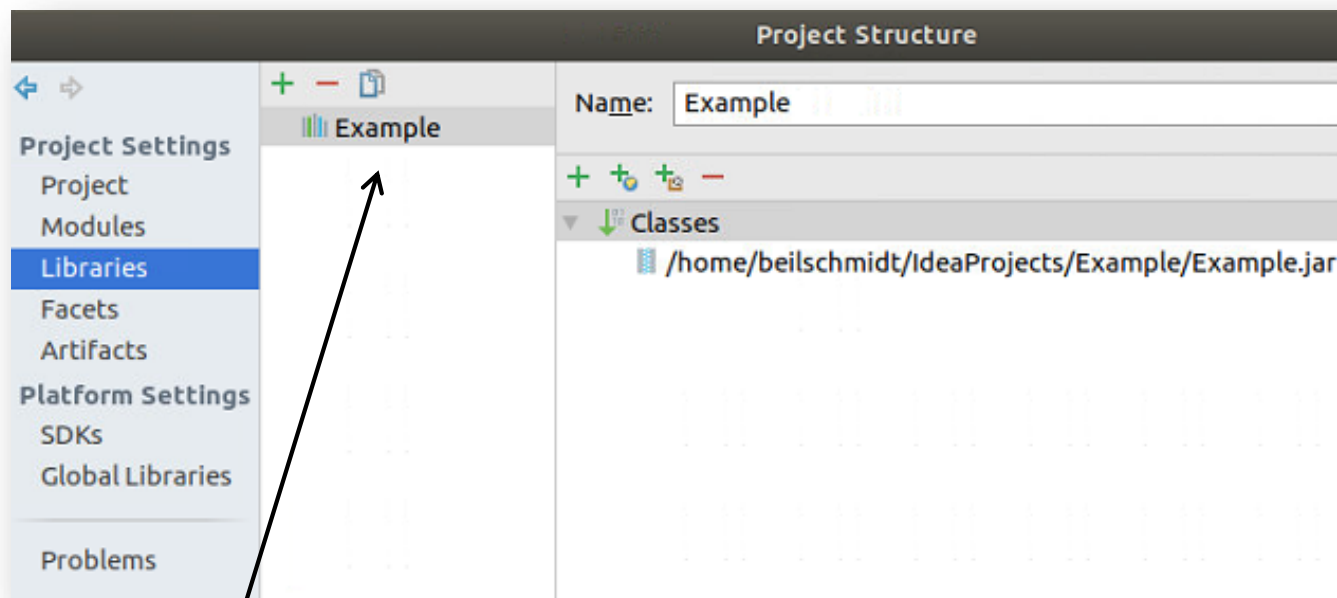


# Classpath

- Compiler muss Programmcode finden
- Classpath definiert die Verzeichnisse, in denen gesucht wird
  - Ausführungsverzeichnis
  - Umgebungsvariable CLASSPATH
  - Optionaler Parameter -classpath
- In IntelliJ:
  - Rechtsklick → Kontextmenü  
→ Add as library ...



# Classpath (2)

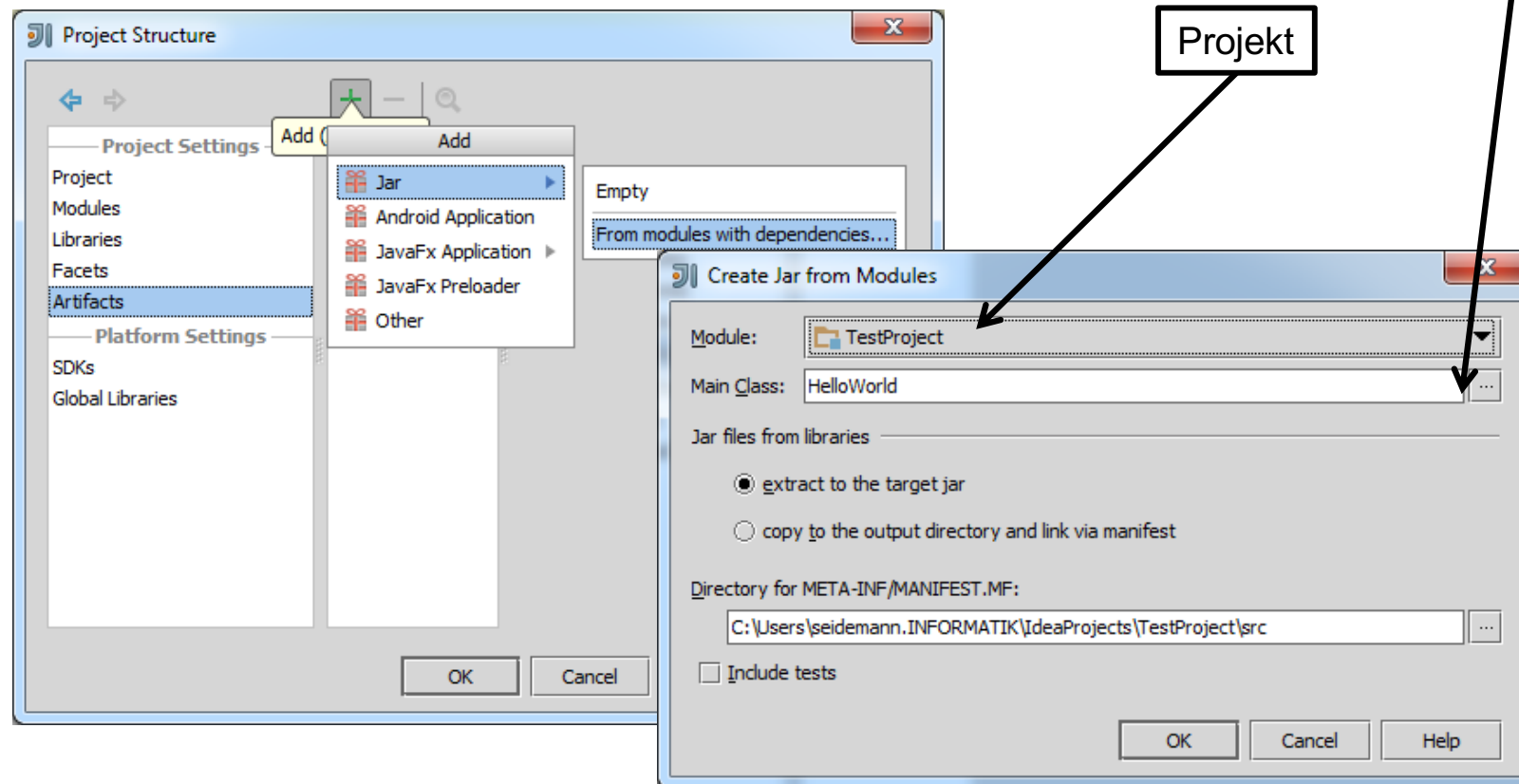


Die unter "Libraries" hinzugefügten Bibliotheken werden automatisch auch dem CLASSPATH hinzugefügt.

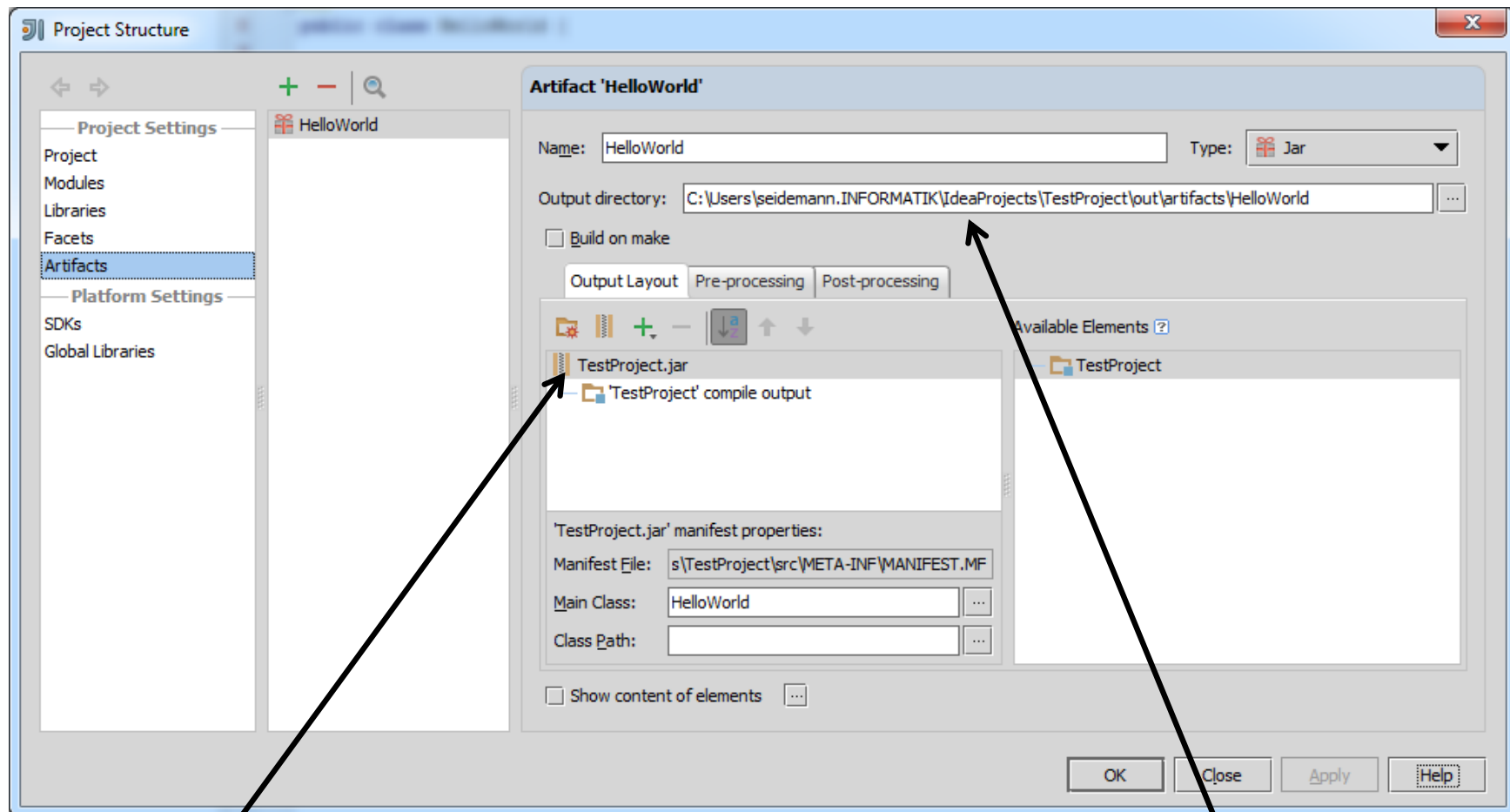


# JAR-Datei mit IntelliJ

- Rechtsklick auf Projekt →  Open Module Settings → Artifacts



# JAR-Datei mit IntelliJ

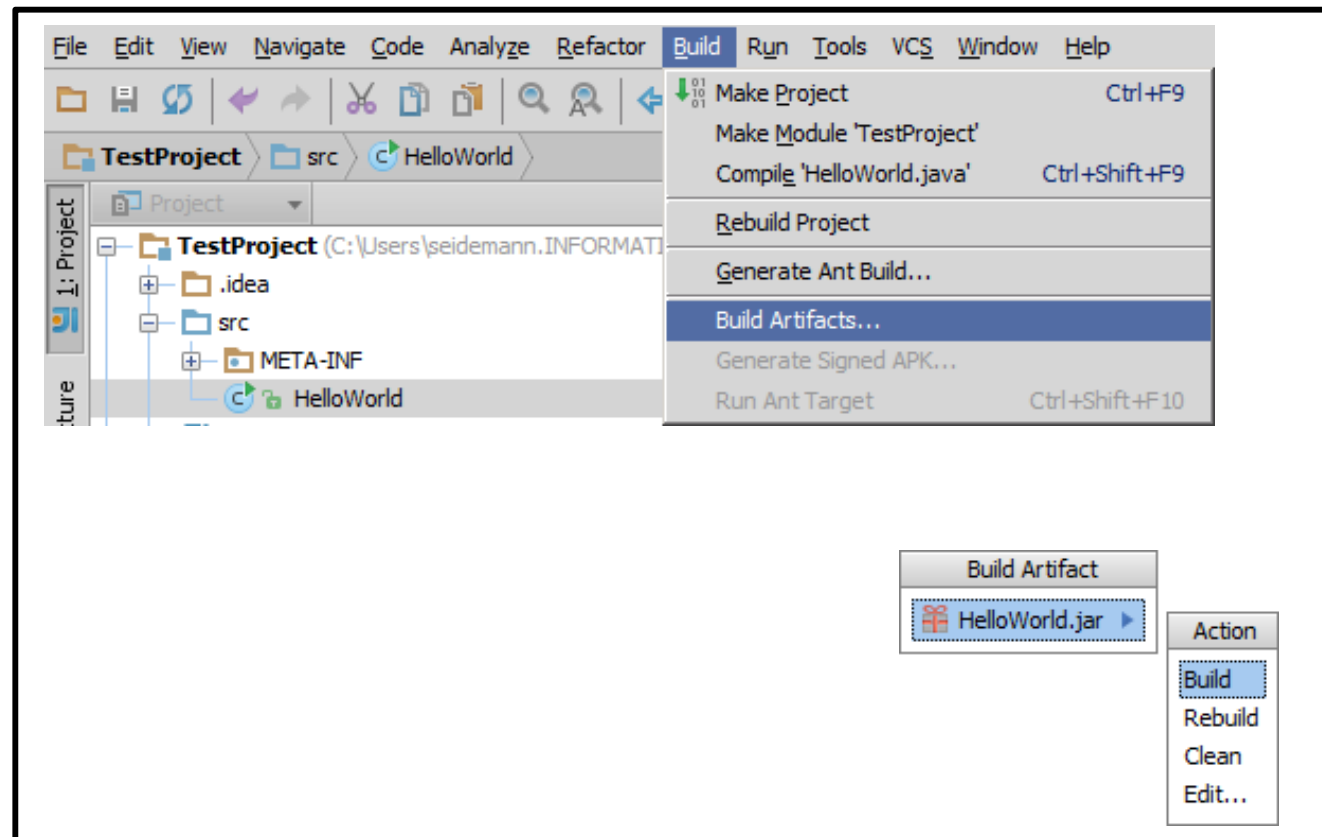


Name der JAR-Datei

Ausgabepfad der JAR-Datei

# JAR-Datei mit IntelliJ

- JAR-Datei muss nach einer Änderung neu erstellt werden



# Besser: Build-System

- Maven, Gradle, etc. bilden Pfad von...
  - Abhängigkeiten (Bibliotheken)
  - Kompilieren
  - Testen
  - Ausliefern (Jar)

Jedoch nicht Inhalt dieser  
Vorlesung.

...ab.

- Spezifikation in einer Datei (XML, Gradle Script DSL)
- Arbeitsfluss ist IDE-unabhängig
- Abhängigkeiten aus Online-Repositories ([search.maven.org](https://search.maven.org))

# Wartung

- Nach der ersten Auslieferung: Software muss gewartet werden, denn ...
  - Früher oder später tauchen Fehler auf, die behoben werden müssen, oder
  - Der Kunde möchte die Software um neue Funktionen erweitern, oder
  - Bestimmte Teile des Codes sollen aus Performancegründen überarbeitet werden, oder
  - ...
- Software muss gut zu warten sein
  - Umfassende Dokumentation
  - Kommentare an kritischen Stellen
  - Keine „schmutzigen Tricks“ (der Kollege muss es auch verstehen)
  - Einhaltung von **Konventionen**



# Konventionen in Java

- Konventionen sind nicht zwingend vorgeschrieben (der Compiler akzeptiert auch schlecht formatierten den Code), gehören aber zum guten Ton
  - Wenn ein Java-Entwickler den Bezeichner „MyNumber“ liest, vermutet er eine Klasse und keine Integer-Variable!
- Bezeichner: in Java wird CamelCase verwendet
  - Variablen, Methoden und Felder beginnen mit Kleinbuchstaben
  - Klassen beginnen mit einem Großbuchstaben
  - Pakete beginnen hinter einem Punkt mit Kleinbuchstaben: z.B. java.lang
- <http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html>