

# Inhaltsverzeichnis

1. Projektverwaltung mit <b>Maven</b> & Projektstruktur .....	1
2. Test-Driven-Development ( <b>TDD</b> ) .....	4
3. Generierung von Quellcode mit der IDE .....	5
4. Autocompletion .....	7
5. Icons & Symbole in der IDE .....	7
6. Tipps, Tricks, Probleme & Lösungen .....	8
6.1. Die Anzeige von Diagrammen in AsciiDoc Dokumenten schlägt fehl .....	8

## 1. Projektverwaltung mit **Maven** & Projektstruktur

Das Projekt bzw. jedes **Modul** dieser Lehrveranstaltung hat eine ähnliche Struktur, die die Nutzung vereinfacht und die **Wiedererkennbarkeit** verbessert:

Im **Allgemeinen**:

```
■ <Projekt-Wurzel>
  ■ <Modul-Wurzel>
    ■ docs      → Dokumentation zum Modul
    ■ src       → Wurzel für den gesamten Sourcecode
      ■ main    → Wurzel für den "produktiven" Sourcecode
        ■ java ...
      ■ test    → Wurzel für den Test-Sourcecode
        ■ java ...
    ■ target    → Kompilierte Klassen
```

Am **Beispiel** des ersten Moduls **module-classes**:

```
■ D:\Projekte\JavaProgrammierung
  ■ module-classes
    ■ docs
    ■ src
      ■ main
        ■ java      → ab hier "Package" Verzeichnisse
          ■ de
            ■ dhw
              ■ demo  → Klassen zur Demonstration,
                    ggf. mit Unterverzeichnissen
              ■ exercise → Klassen, die im Rahmen von Tests
                    implementiert werden
              ■ solution → Klassen, die Lösungen für die Tests sind
      ■ test
        ■ java
```

```
graph TD; de[de] --- dots[...]; dots --- target[target]
```

Ein sehr weit verbreitetes Framework bzw. Tool zur Erzeugung solcher Projektstrukturen ist das bekannte **Maven** (Alternative **Gradle**).

Die Erstellung einer Software beinhaltet viele Voraussetzungen, aber noch vor Beginn der Implementierung sollte man sich über ein paar Grundlagen Gedanken machen.

Dazu gehört das **Build- & Dependency Management**. Aus diesem Grund wird Maven auch als **Dependency Management Tool** bezeichnet.

→ <https://maven.apache.org>

Es erleichtert an dieser Stelle u.a. ...

- die Verwaltung von *Dependencies*,
- die Verwaltung des *Classpath*,
- den *Compile* des Projektes,
- die *Konfiguration*,
- den *Build* des Projektes und nicht zuletzt
- das *Deployment* des Liefergegenstandes (Deliverable)
- es ermöglicht Erweiterungen durch PlugIns
- ... und viele weitere Aspekte

Das Tool verfolgt das Konzept

Convention over Configuration

das eine geringst mögliche **Grundkonfiguration** ermöglicht, weil das Tool einfach eine Reihe von Annahmen trifft, die für sehr viele Projekte allgemein anerkannt sind und häufig genutzt werden. Erst wenn von diesen *Konventionen* abgewichen werden muss, kann das Projekt entsprechend *konfiguriert* werden.

Die wichtigste Datei für Maven ist das **Project Object Model** in Form der Datei

pom.xml

die sowohl im Gesamtprojekt als auch im Wurzelverzeichnis jeden Moduls zu finden ist (sh. nachfolgend "parent-child-Beziehung"). Diese werden von vielen **IDEs** automatisch erkannt und beim Laden des Projektes für die Projektkonfiguration genutzt.

Bei hierarchischen Projektstrukturen wie in diesem Kurs gibt es mehrere POMs innerhalb der Hierarchie, sogenannte *parent* und *child* POMs, die zusammen gehören (→ vgl. Repository zu dieser


LV).

## Nutzung von Maven

Das Projekt sollte jetzt einmal insgesamt "gebaut" werden:

- entweder über das User-Interface von IntelliJ
- oder über die Kommandozeile

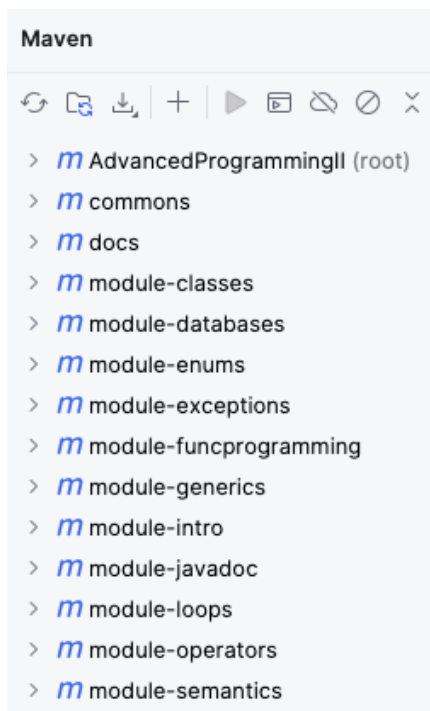
Am besten einfach in IntelliJ:

- Rechts oben am Rand von IntelliJ gibt es ein Symbol  (=maven), das öffnet die Standard Maven-View → **(1)**
- Dann gibt es ein Symbol zur Ausführung von Maven → **(2)**
- darauf hin öffnet sich ein modales Fenster, hier gibt man

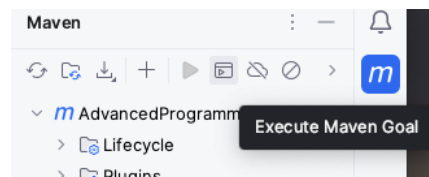
```
mvn clean compile
```

in der Kommandozeile ein oder macht das über die GUI → **(3a)** oder **(3b)**

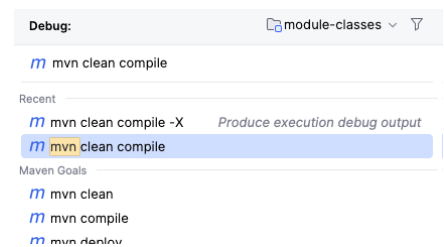
(1)



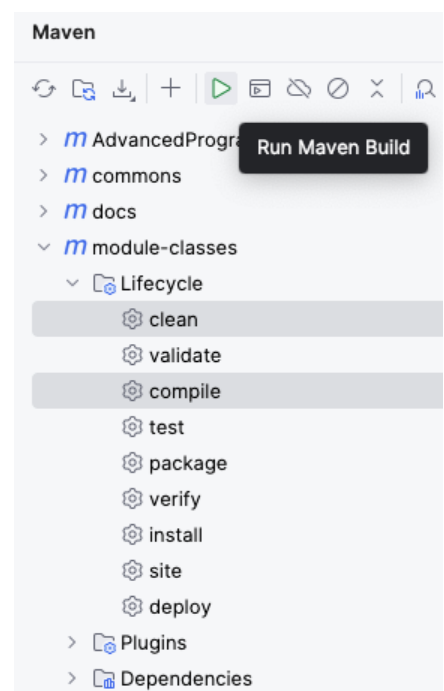
(2)



(3a)



oder (3b)



Das Projekt sollte jetzt insgesamt **"bauen"**, d.h. die Sourcen *kompilieren* und *assembeln* (/target Ordner, spielt hier aber keine Rolle). Im unteren Bereich von IntelliJ öffnet sich automatisch eine Console und das "Build" Ergebnis wird angezeigt ... hoffentlich **SUCCESS** für alle Module ;-) ...

## 2. Test-Driven-Development (TDD)

Test Driven Development (TDD) ist eine gute Praxis, um den Sourcecode von Beginn an - mithilfe von Unit-Tests regelmäßig und bei Änderungen auf Korrektheit zu überprüfen. Die erste Umsetzung erfolgt vielfach durch Testklassen, den **Unit-Tests**.

Unit-Tests haben folgende **Eigenschaften**:

1. Unit-Tests sind **automatisiert**. Ein Unit-Test-Framework führt Tests aus, verifiziert und gibt das Ergebnis zurück, damit es geprüft werden kann
2. Unit-Tests sind **granular**, sie sollen nur einen kleinen Teil des Codes -häufig eine Methode - testen.
3. Unit-Tests **isolieren** das Testziel und sollen möglichst ohne oder nur mit wenig "Vorbereitungen" gestartet werden können
4. Unit-Tests sind **deterministisch**, damit das Testergebnis sauber geprüft werden kann und wiederholt werden kann
5. Unit-Tests sind **unabhängig**. Die Ausführung der Tests dürfen in keiner Weise von anderen Testmethoden abhängen, denn die Reihenfolge der vom Framework ausgeführten Tests ist zufällig bzw. nicht vorhersagbar.

**Umsetzung:**

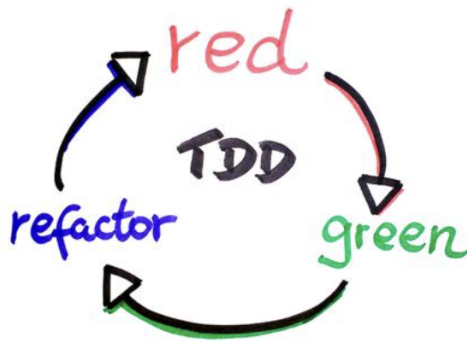
Unit-Tests weisen eine besondere (innere) Struktur auf, d.h. die Art und Weise, wie diese geschrieben werden. Dazu zählen

- **given-when-then** oder
- **arrange-act-assert**.

**Vorgehensweise:**

Auch das Schreiben von Tests folgt vielfach einer Routine bzw. einer "best practice". Neben zahlreichen gleichwertigen Ansätzen ist die so-genannte **"red, green, refactor"** Methode weitestgehend anerkannt.

Quelle: → [Informatik Aktuell](#)



### Red, Green, Refactor:

Der "red, green, refactor" Ansatz hilft Entwicklern, den (zu implementierenden) Code zu entwickeln, indem sie den Fokus in den Phasen auf bestimmte Aspekte lenken:

1. **Red:** *Was soll implementiert werden und wie fühlt sich das Ganze aus Sicht des "Clients" bzw. der Nutzer an?* Nutzer sind hier primär die Entwickler selbst bzw. diejenigen, die die Klassen, Methoden, Algorithmen im Code aufrufen, also nutzen wollen. In dieser Phase *soll* der Test noch scheitern, d.h. "rot" sein, weil hier nicht die Implementierung an sich, sondern die "Außensicht" auf die **API** im Vordergrund steht.
2. **Green:** In dieser Phase geht es darum, den Test "zum Laufen" zu bringen. Es wird also "grün". Im Vordergrund steht die *schnellste und einfachste Implementierung* der Funktionalität. Der Code wird hier also nur technisch funktionsfähig gemacht, es werden aber so wenig wie möglich Überlegungen angestellt, wie der Code gut, schön oder effektiv geschrieben werden muss.
3. **Refactor:** In dieser letzten Phase (dieser Iteration) geht es nun genau darum, was in den beiden vorherigen Phasen absichtlich nicht gemacht werden sollte, nämlich die Verbesserung des Codes hinsichtlich seiner "Qualität" (das Thema Code-Qualität wird gegen Ende des Seminars noch näher betrachtet). Ergebnis dieser (drei) Phasen soll dann ein Unit-Test sein, der lesbar und wartbar ist und auch die Implementierung der Funktionalität einen bestimmten Reifegrad bzw. eine erste hinreichende Qualität erreicht hat, und natürlich letztlich die korrekte Funktionalität liefert.

Dazu eine **Demo**:

```
→ src/test/java/de/dhbw/demo/tdd/TddDemoTest.java
```

## 3. Generierung von Quellcode mit der IDE

In aller Regel ist es sehr sinnvoll, die Entwicklungsumgebung (IDE) weitreichend zu nutzen. Das gilt insbesondere für die **Erzeugung von Quellcode**. Das Ganze ist grundsätzlich eine Sache der Übung, man sollte sich einfach daran gewöhnen, diese Hilfen möglichst viel als Entwickler:In zu nutzen!

Ein paar **Tipps** dazu:

### Generierung von Quellcode (Klassen, Methoden, etc.)

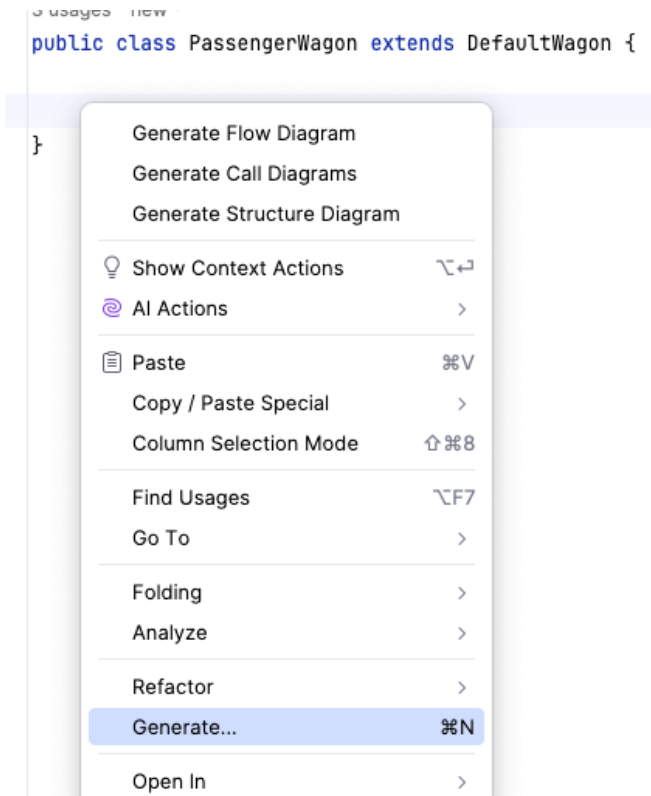
1. mithilfe einer **Tastenkombination**

a. unter MacOS: **Command + N**

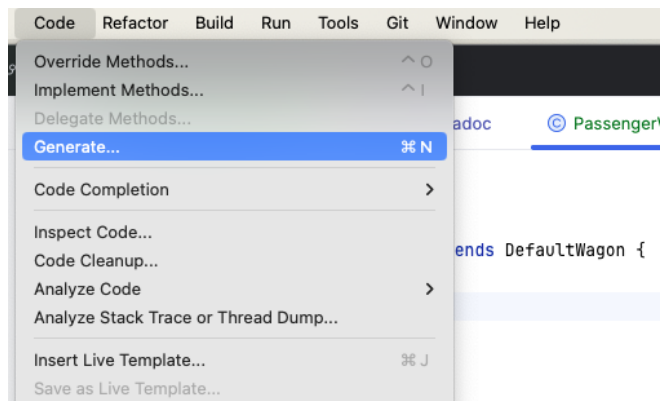
b. unter Windows: **Alt + Einfg** (ggf. auch **Fn + Alt + Einfg**) oder

## 2. mithilfe der **Maus**

An der Cursorposition "Rechts-Klick" > **Generate** ...

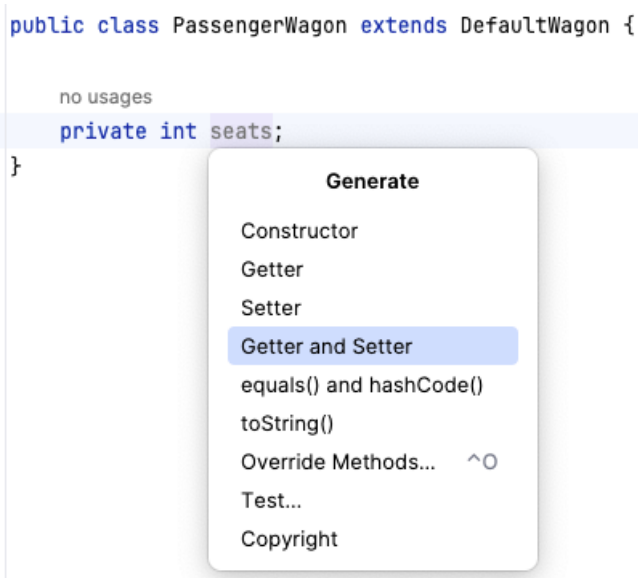


An der Cursorposition > IntelliJ-Menü **Code** > **Generate** ...



Hier wählt man dann einfach die gewünschte **kontext-spezifische** Funktion:

z.B. die Generierung von Methoden "*Getter & Setter*"



oder die Implementierung von Interface-Methoden bzw. die sog. *Overrides* bei Vererbungen



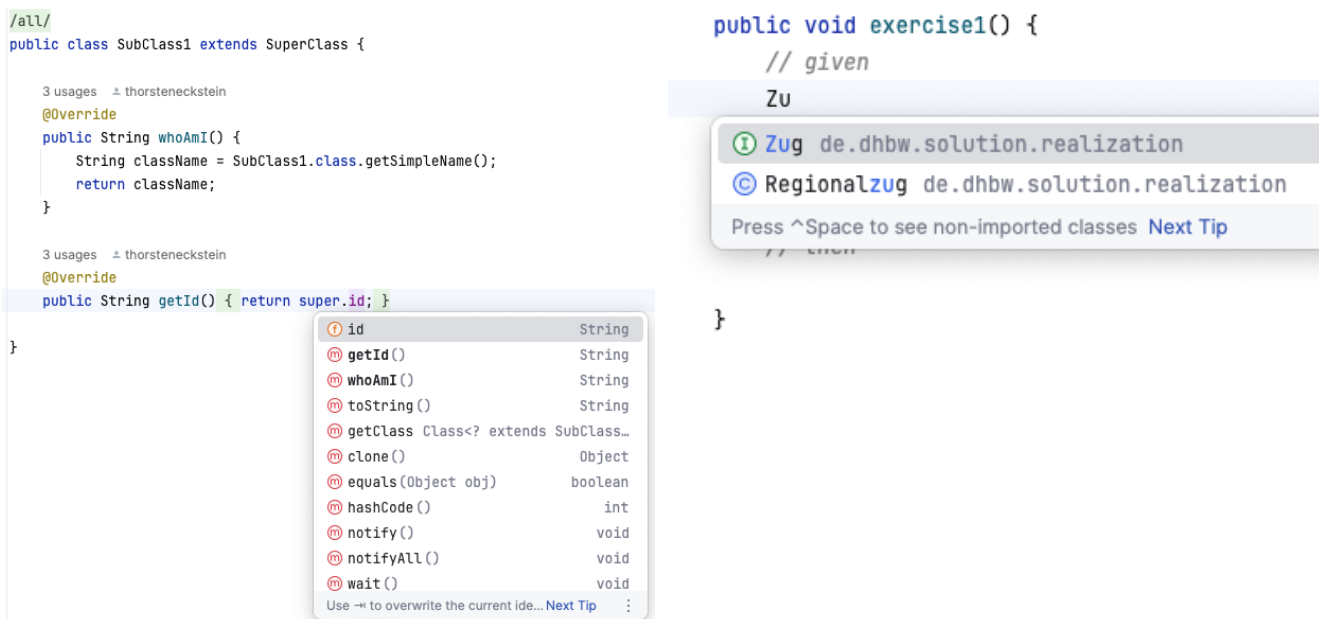
## 4. Autocompletion

Neben der Generierung von Code ist auch die Unterstützung beim Tippen durch die sog. **Autocompletion**, also die automatische Ergänzung/Erkennung des gerade getippten Codes, von besonderer Bedeutung.

Die IDE gibt insgesamt sehr viele, leider zum Teil recht minimalistische, Hinweise und Informationen zum aktuellen "Geschehen" während der Programmierung. Dazu gehören u.a.:

Hier wählt man dann einfach die gewünschte **kontext-spezifische** Funktion:



Anzeige von möglichen Methoden einer Instanz      Auswahlmöglichkeiten für nutzbare Typen, hier Klassen



## 5. Icons & Symbole in der IDE

Die "kleinen Hinweise" der IDE helfen dem geübten Auge mit schnellen Informationen. Dazu ein paar Beispiele, wenn man mit der Maus über das jeweilige Symbol fährt:

Symbol	Symbol (am linken Editor-Rand) & Bedeutung

Symbol	Symbol (am linken Editor-Rand) & Bedeutung
	

Darüber hinaus gibt es viele weitere Infos, deshalb ...



Einfach ausprobieren und die Maus über alles Mögliche "hovern" ;-)

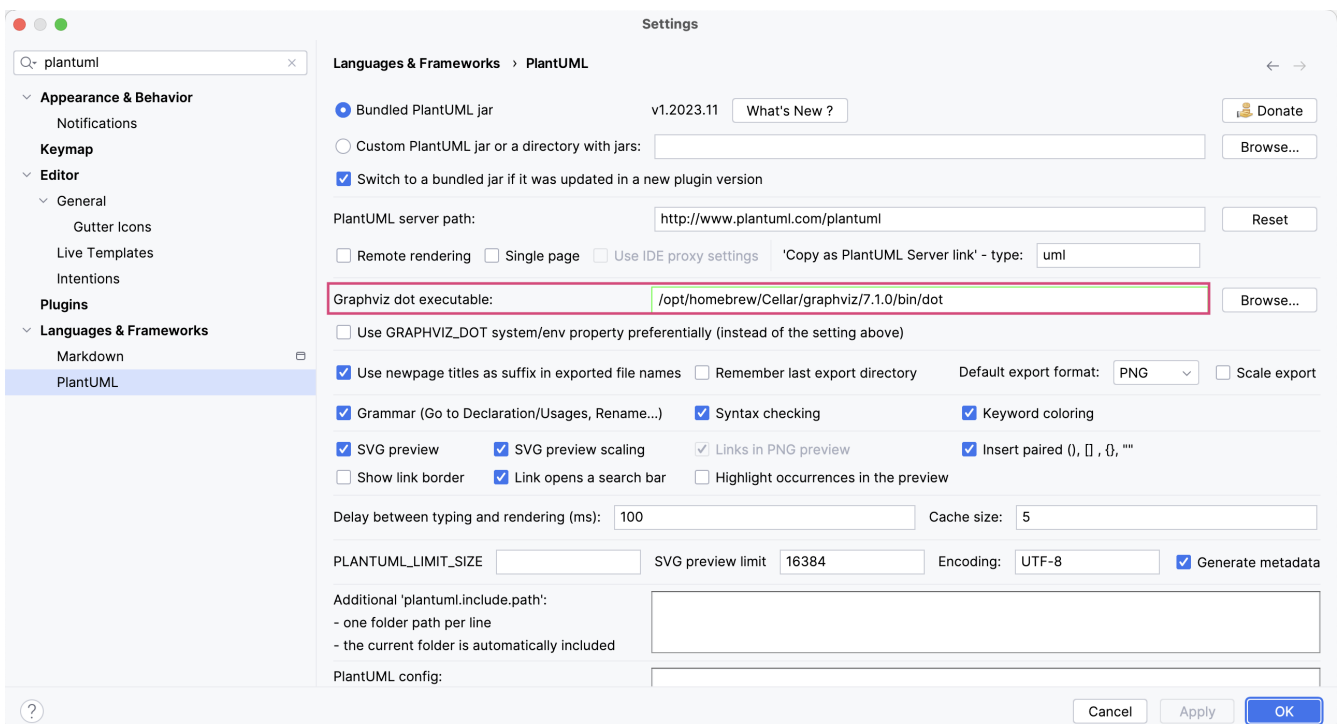
## 6. Tipps, Tricks, Probleme & Lösungen

### 6.1. Die Anzeige von Diagrammen in AsciiDoc Dokumenten schlägt fehl

Fehlermeldung im Diagram signalisiert, dass das Tool **graphviz** fehlt.

Dieses Tool muss ...

1. zuerst installiert werden → <https://graphviz.org/download/> und dann
2. in IntelliJ für das PlugIn AsciiDoc bekannt gemacht werden, durch Angabe des Pfades zur **dot** Ausführungsdatei (z.B. **dot.exe**)



Die Pfade im obigen Bild sind MacOS Pfade, unter **Windows** sehen diese etwas anders aus!