

Inhaltsverzeichnis

1. Kurstag 1 - Setup & Basics 1/2
 - 1.1. Inhalte
 - 1.2. Frameworks, Tools & Setup
 - 1.2.1. Einrichtung eines lokalen *Workspaces*
 - 1.2.2. Java
 - 1.2.3. IDE IntelliJ
 - 1.2.4. SCM mit Git und Github
 - 1.2.5. Projekt in IntelliJ laden
 - 1.2.6. Projektverwaltung mit Maven
 - 1.2.7. Test Driven Development (TDD)
 - 1.2.8. H2 Database
 - 1.3. Fachlicher Schwerpunkt des Kurses (Fachlichkeit)
 - 1.3.1. Kontext
 - 1.3.2. Klassenmodell
 - 1.4. Das erste Klassenmodell
 - 1.5. Access Modifier
 - 1.6. Objektvertrag
 - 1.7. Übungen

1. Kurstag 1 - Setup & Basics 1/2

1.1. Inhalte

Setup:

- ☐ Tool-Stack: IDE, Tools, Frameworks
- ☐ Test-Driven-Development
- ☐ Build- & Dependency Management
- ☐ Unit-Tests mit JUnit

Theorie & Objektmodell:

- ☐ Klassen & Objekte
- ☐ Objektvertrag (hashCode, equals)
- ☐ Vererbung & Interfaces
- ☐ Abstrakte Klassen
- ☐ Access Modifier (Sichtbarkeiten, Scopes)

Mögliche Zusatzinhalte:

- ☐ Referenzsemantik (→ siehe Kurstag 10)

1.2. Frameworks, Tools & Setup

1.2.1. Einrichtung eines lokalen *Workspaces*



Zuerst wird ein lokaler *Workspace* eingerichtet, d.h. ein Wurzelverzeichnis, in das das SourceCode-Repository bzw. die Kursinhalte "geklont/gespeichert" werden kann.

(Natürlich kann man hier immer den persönlichen Präferenzen für dieses Wurzelverzeichnis folgen, für die Einrichtung des Kurses ist es aber besser, den Empfehlungen zu folgen ...)

Man kann alles Folgende auf unterschiedliche Arten erledigen, vor allem mit dem Unterschied

- Über die GUIs bzw. Menüs der Tools, z.B. bei `IntelliJ` oder
- Über die Kommandozeilen Tools, z.B. `CMD`, `PowerShell` oder die `bash`

(Ich persönlich richte meine lokale Arbeitsumgebung (den Workspace) wann immer möglich, etwa für ein Projekt, mit exakt der gleichen Struktur wie die des (Online/Remote-) Repositories ein, dazu später mehr)

Wir verfolgen hier für bestimmte Schritte des Setups die Kommandozeilen Tools, insbesondere für die initiale Nutzung von `git`. Zunächst aber der Workspace:

- ▶ Einrichtung Workspace unter **Windows** (11) am Beispiel
- ▶ Einrichtung Workspace unter **MacOS** am Beispiel

1.2.2. Java

Empfehlung: Im Kurs soll folgende Java Version verwendet werden:

```
Java 17 (LTS=Long Term Support)
```

Ggf. muss diese Version noch installiert werden, wobei die Versionsnummern ("Minor") *hinter* der Hauptversionsnummer 17 ("Major") nicht so wichtig sind.

- In den neuen Workspace, momentan noch leeren Ordner (s.o.), wechseln

```
cd Projekte
```

- Prüfen, ob `java` vorhanden ist

```
java -version
```

Dann sollte eine Java Version angezeigt werden.

1.2.3. IDE IntelliJ

IDE. Die empfohlene IDE ist `IntelliJ`. Die Seminarinhalte sollten sowohl in der *Community* als auch in der *Enterprise* Edition funktionieren.

Der Code für den Kurs wurde mit folgender Version erstellt, die installiert sein muss (falls noch nicht vorhanden, bitte installieren):

```
IntelliJ IDEA 2022.3.2 (Community Edition)
```



Alternativ kann auch **MS Visual Studio Code** genutzt werden und diese Umgebung sollte auch bei der Einrichtung keine besonderen Probleme verursachen. Aber ... auf eigene Gefahr ;-), das Seminar wurde (nur) für IntelliJ vorbereitet!

IntelliJ bietet ein neben dem Standard-Layout auch ein experimentelles, das aktiviert werden kann. Der Standard ist aber vollkommen ok.

IntelliJ PlugIns

Bundled PlugIns

Mit IntelliJ werden diverse PlugIns automatisch installiert, dazu gehören die folgenden, die für das Seminar erforderlich sind und anfangs mal geprüft werden sollen:

PlugIn	Kommentar
Git & GitHub	<i>Source Code Management, lokaler Git-Client sollte mind. die Version 2.39.1 haben.</i>
Maven home path: <input type="text" value="Bundled (Maven 3)"/> (Version: 3.8.1)	<i>Build- & Dependency Management</i>

Non-bundled PlugIns

Für den Code und Tests und die Dokumentation sind folgende PlugIns erforderlich, die nicht sowieso mit IntelliJ installiert werden:

PlugIn	Kommentar
 AsciiDoc  0.38.9 AsciiDoctor IntelliJ Plugin Project	<i>Dokumentations-Markup, Version 0.38.9 (oder höher)</i>
 Diagrams.net Integration  0.1.14 Docs As Code	<i>integrierte Erstellung von Diagrammen, Version 0.1.14 (oder höher)</i>
 PlantUML Integration  5.22.0 Eugene Steinberg, Vojtech Krasa	<i>Erstellung von UML-Diagrammen, Version 5.22.0 (oder höher)</i>
 Database Navigator  3.3.5889.0 Dan Cioca	<i>Zugriff auf H2 Datenbank, Version 3.3.5889.0</i>

PS: Ich persönlich aktualisiere sehr zeitnah die PlugIns.

1.2.4. SCM mit Git und Github

- Bitte zuerst prüfen, ob `git` installiert ist. Das geht am besten im Kommandozeilen Tool durch

```
git --version
```

- was zum Beispiel folgende Ausgabe ergibt: `git version 2.39.1`
- Zunächst muss das Repository (<https://github.com/ThorstenEckstein/dhbw-advanced-programming>) "geklont", d.h. heruntergeladen werden, in Form einer lokalen Kopie:

```
git clone https://github.com/ThorstenEckstein/dhbw-advanced-programming.git -b hands-on
```

- Zur Prüfung, auf welchem Branch man sich aktuell befindet, wechselt man in das neue, lokale Verzeichnis des Repositories und gibt ein:

```
git branch
```

- Hier sollte jetzt etwas Ähnliches erscheinen:

```
main
develop
* hands-on
```



... es soll kein Code in das `remote` Repository **gepushed** werden, da es sonst zu Konflikten bei möglicherweise erforderlichen Updates (durch *pullen*) kommen kann!

► Hilfe zu bzw. Anzeige von Umgebungsvariablen

1.2.5. Projekt in IntelliJ laden

- IntelliJ starten
- Öffnen des Projektes mit `Open` und das Kurzverzeichnis (Repository) mit auswählen. Das Projekt sollte sich jetzt grundsätzlich "selber" einrichten, z.B. werden erforderliche Bibliotheken im Hintergrund automatisch heruntergeladen, siehe `Maven`, dazu gleich mehr).
- Ggf. stellt IntelliJ dem Nutzer noch Fragen zu bestimmten Aspekten, etwa:
 - a. Sprache: Soll das `German-Language` Pack downgeloaded werden? ⇒ Ja, download
 - b. Rückfrage zu `Kroki-Settings` (gehört zum AsciiDoc PlugIn)? ⇒ Ja, download
 - c. Rückfrage zu `asciidoctorj-diagrams` (ebenfalls im AsciiDoc Kontext) ⇒ Ja, downlaod
 - d. evtl. weitere ..., die müssen wir uns dann einfach zusammen anschauen.

Das Projekt sollte jetzt in IntelliJ geladen und verfügbar sein.

1.2.6. Projektverwaltung mit Maven

Ein sehr weit verbreitetes Framework bzw. Tool ist `Maven`.

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**. → Maven (<https://maven.apache.org/>) 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)
- und viele weitere Aspekte ...

Das Tool verfolgt den Ansatz

Convention over Configuration

was eine geringst mögliche Basiskonfiguration 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 den 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 im Projekt im Wurzelverzeichnis zu finden ist, von vielen IDE automatisch erkannt und beim Laden des Projektes für die Projektkonfiguration genutzt wird.

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

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

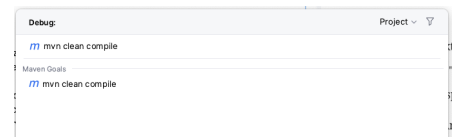
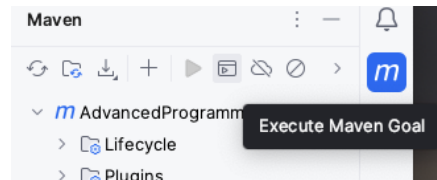
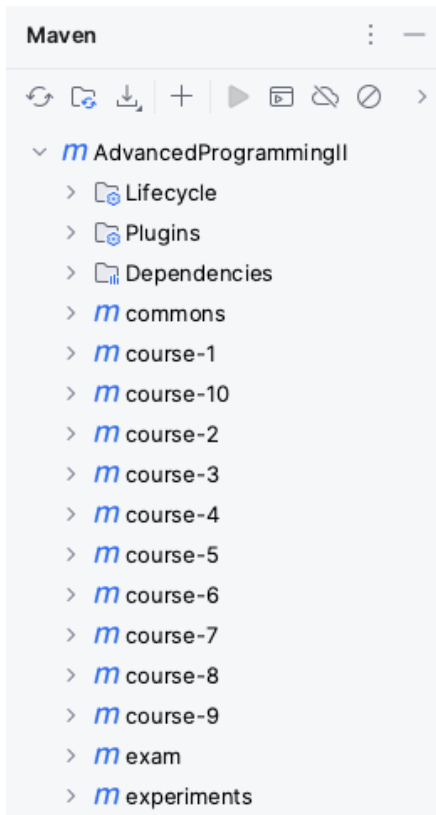
```
mvn clean compile
```

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

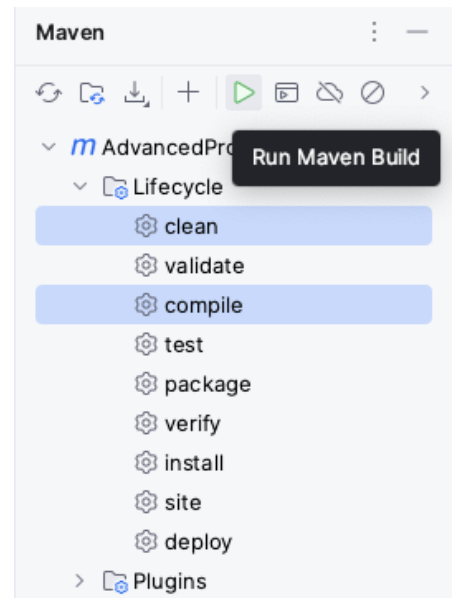
(1)

(2)

(3a)



oder (3b)



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

1.2.7. 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

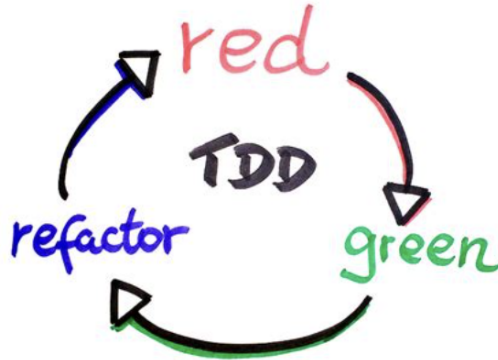
- given-when-then oder

- `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 (<https://www.informatik-aktuell.de/entwicklung/methoden/tdd-erfahrungen-bei-der-einfuehrung.html>)



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 ein **Beispiel**:

```
test/de/dhbw/course1/tdd/TddByExampleTest.java
```

1.2.8. H2 Database



In der IntelliJ *Ultimate* Edition kann das bereits enthaltene PlugIn **Database Tools** genutzt werden, leider aber nicht in der *Community* Edition. In diesem Fall wird das PlugIn **Database Navigator** genutzt. Dessen Nutzung wird an Kurstag 7 näher erläutert!

Wenn das H2 Database PlugIn korrekt installiert ist, kann man eine Verbindung zu einer H2 Datenbank (Instanz) erstellen.

Das wird im Detail näher in Kurs 7 beschrieben:

→ Kurstag 7: H2 Database Setup

1.3. Fachlicher Schwerpunkt des Kurses (Fachlichkeit)

1.3.1. Kontext

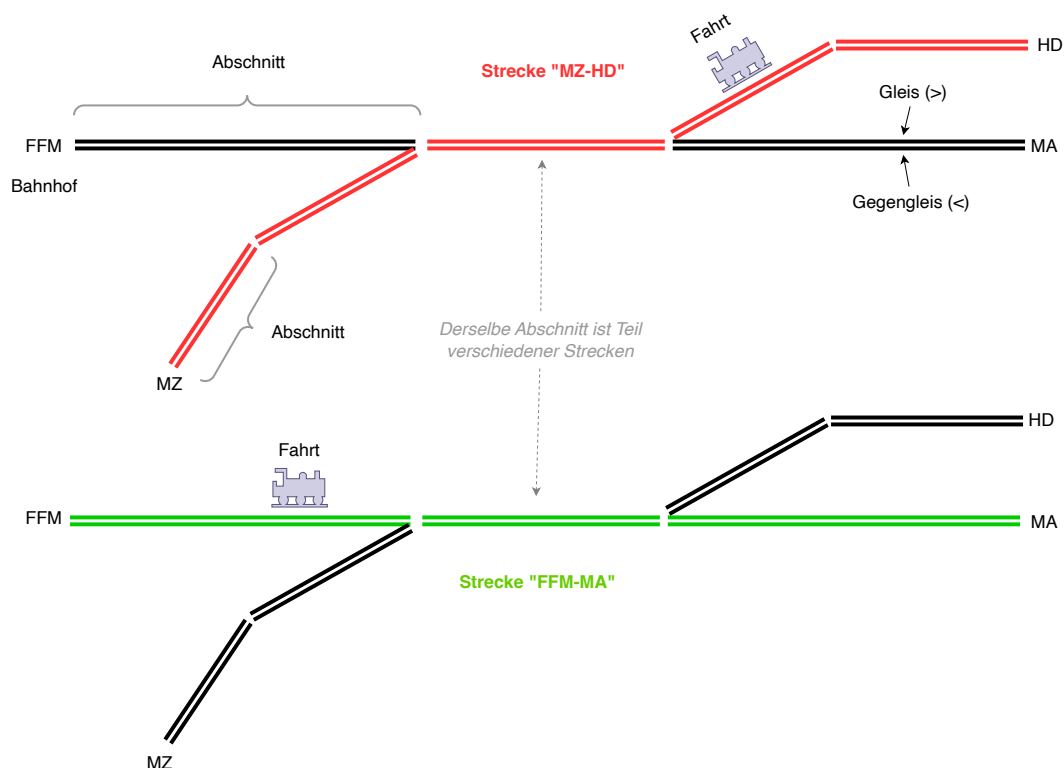
Vorbereitung: Darstellung des fachlichen Kontextes

- + Überwachungssystem in der Betriebsleitung
- + Zug, Zugtypen, Triebfahrzeug (Tbf), Wagons
- + Streckennetz
- + Fahrplan
- + Störung, Weichenausfall, Abfahrt, Ankunft (z.B. verspätete Ankunft im Bahnhof)
- + Dispositionsmaßnahme (z.B. Umleitung)

Der Zugbetrieb kann grundsätzlich unterteilt werden in:

1. **infrastrukturelle** Sicht (*Stammdaten*)
2. **planerische** Sicht auf den Betrieb (*Bewegungsdaten, SOLL*)
3. **operative** Sicht auf den Betrieb (*Bewegungsdaten, IST*)

Ein paar Begriffe in der Übersicht:



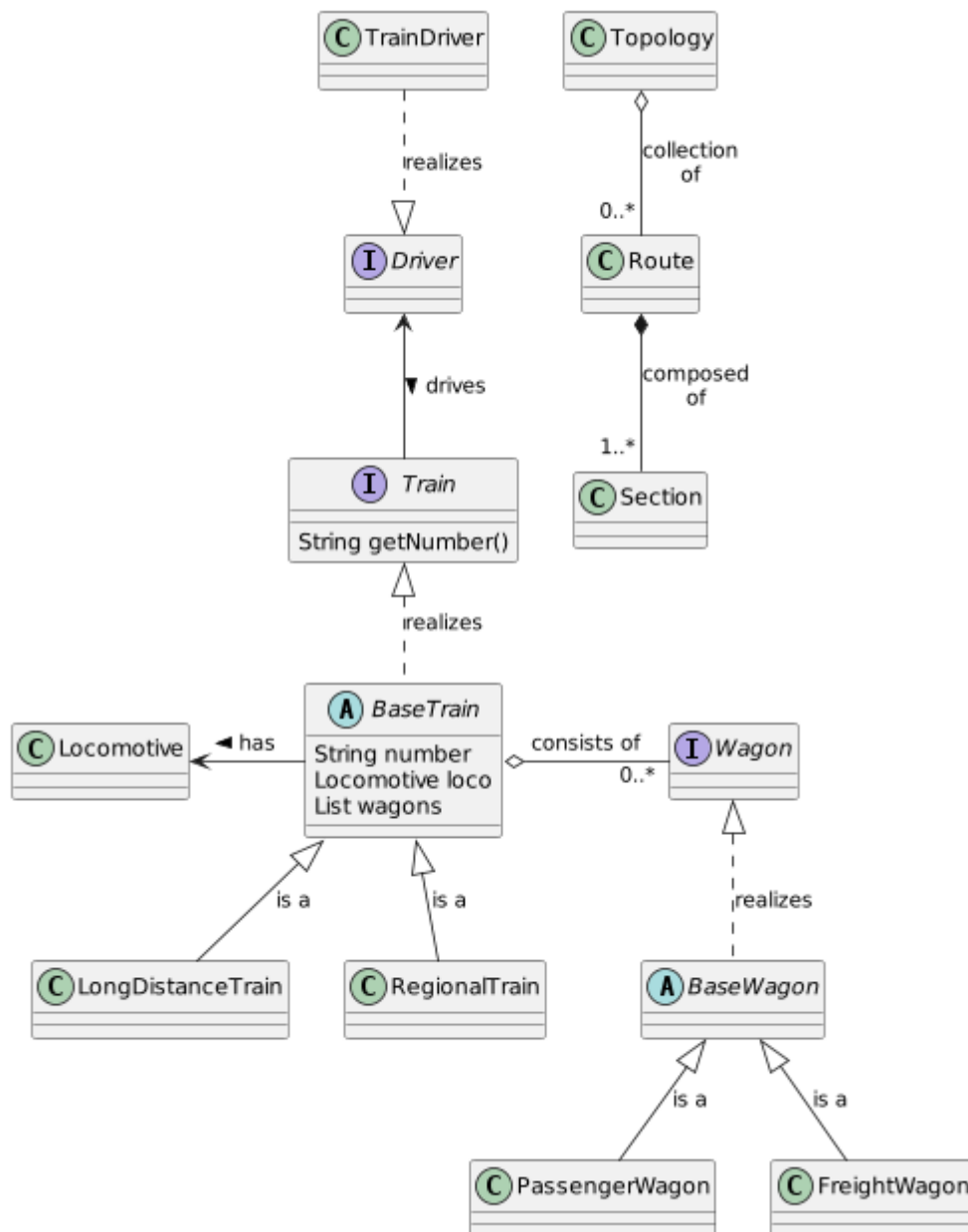
1.3.2. Klassenmodell

Die Fachlichkeit (s.o.) kann in einem (Fach-) **Klassenmodell** abgebildet werden, z.B. mit folgenden Objekten:

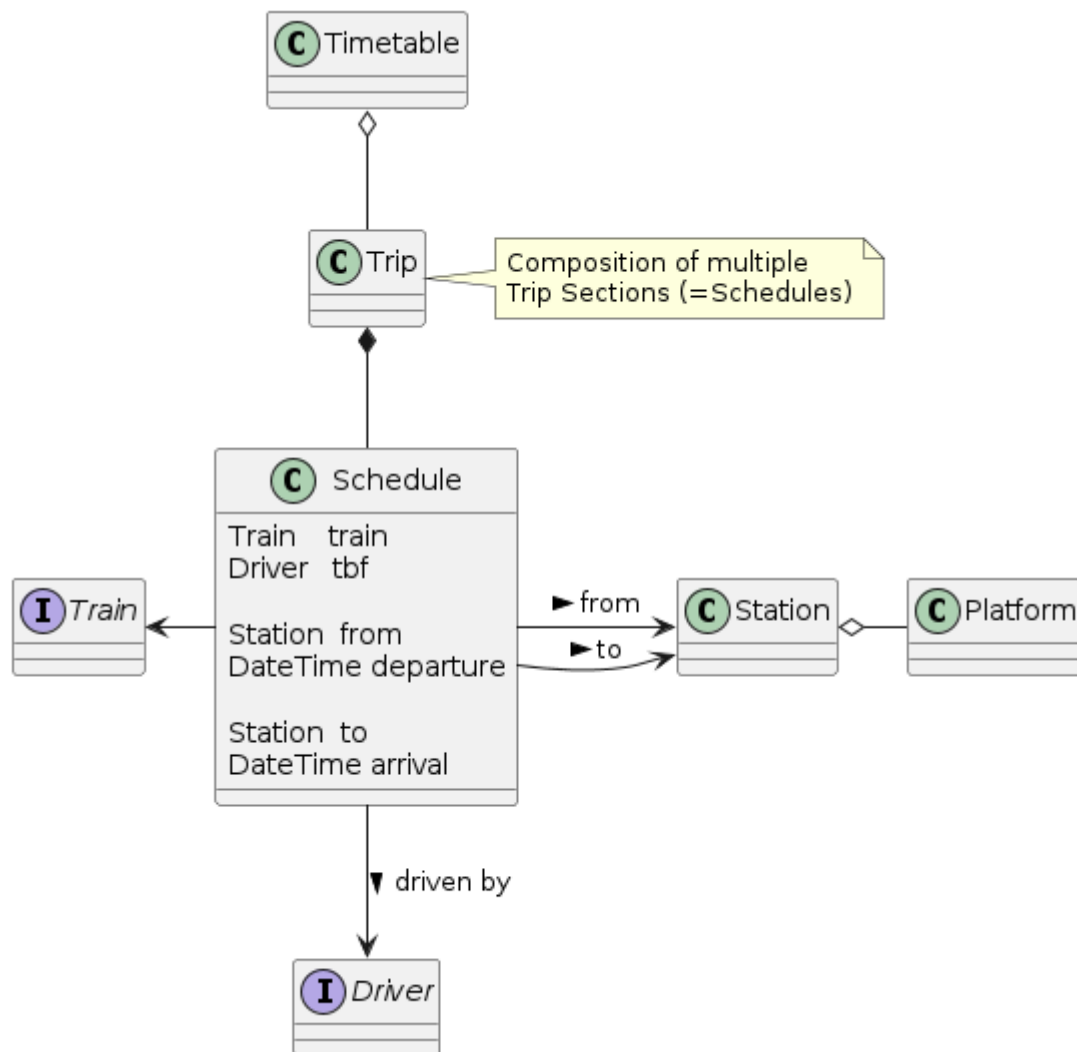
- + Zug (Zugarten), Wagons (Wagonarten)
 - + Strecke, Abschnitt, Gleis
 - + Ereignisse: Abfahrt, Ankunft
 - + Fahrt
- (+ Dispositionsmaßnahme, Umleitung)

Überblick über das grundsätzliche **Fachklassenmodell** in zwei "Geschmacksrichtungen":

Infrastrukturelle Sicht (statische Sicht):



Planerische Sicht (dynamische Sicht):



Für mehr Einblicke in das Modell und die Objekterzeugung, siehe auch

```
test/de/dhbw/course1/model/ModelTest.java
```

1.4. Das erste Klassenmodell

Die wichtigsten Elemente von Java:

- **interface**
- **(abstract) class**
- **field**
- **method**

JAVA

Erzeuge ein erstes, kleines **Klassenmodell**:

- Ein Interface **Train** mit mit zugehörigen Methoden für ein Attribut namens **train number**
- Eine Klasse **EuroExpress**, die das Interface **Train** implementiert, also auch das erforderliche Attribut beinhaltet
- Ebenso eine weitere Klasse **Intercity**, die das Interface **Train** implementiert
- Einen Test, der je eine Instanz der zwei Implementierungen erzeugt und deren Zugnummer ausgibt

Frage : Was fällt bei der Implementierung jetzt besonders ins Auge?
 Antwort : ?

1.5. Access Modifier

Die Standard "Access Modifier" sind:

- `default`
- `public`
- `protected`
- `private`

JAVA

```
@Test
public void canCheckVisibility() {
    // given
    VisibilityExampleClass someClass = new VisibilityExampleClass();

    // What is the reason for fields A, C and D being not accessible?

    //someClass.fieldA
    someClass.fieldB = "some value for field B";
    //someClass.fieldC
    //someClass.fieldD

    // when
    String fieldBValue = someClass.fieldB;

    // then
    assertNotNull(fieldBValue);
}
```

1.6. Objektvertrag

In Java erben alle Objekte von der Klasse `Object`, denn ... alle Klassen in Java *sind* Objekte:

→ <https://docs.oracle.com/.../java/lang/Object.html>

(<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html>)

Die Klasse stellt den Grundvertrag (*Object Contract*) für alle Java Objekte dar, die folgenden zwei sollen dabei näher betrachtet werden, da sie eine besondere Bedeutung haben:

- `equals()`
- `hashCode()` (im *Debug Modus* erkennt man den *HashCode* nicht direkt, nur so etwas wie `EuroExpress@1234`)

The `Object` class defines both the `equals()` and `hashCode()` methods, which means that these two methods are implicitly defined in every Java class, even if they are not implemented explicitly.

The default implementation of `equals()` in the `Object` class says that **equality** is the same as object **identity**.

Gleich oder Identisch?

```

1  @Test
2  public void cannotCommitEqualInstances() {
3      // given
4      Passenger passenger1 = new Passenger("Max Mustermann");
5      Passenger passenger2 = new Passenger("Max Mustermann");
6
7      // when
8      boolean areEqual = passenger1.equals(passenger2);
9
10     // then
11     assertFalse(areEqual);
12     logger.log(String.format(
13         "%s != %s",
14         passenger1.hashCode(),
15         passenger2.hashCode()));
16 }

```

Gleich oder Identisch?

```

1  @Test
2  public void canCommitEqualInstances() {
3      // given
4      Train train1 = new Train("RB-10");
5      Train train2 = new Train("RB-10");
6
7      // when
8      boolean areEqual = train1.equals(train2);
9
10     // then
11     assertTrue(areEqual);
12     logger.log(String.format(
13         "%s == %s",
14         train1.hashCode(),
15         train2.hashCode()));
16 }

```

1.7. Übungen

Die **Übungen** sollen in Form von **Unit-Tests** in folgendem *Package* implementiert werden:

```
src/test/java/de/dhbw/course1/exercises/ExerciseTests.java
```

Die **Testobjekte**, also die *echten* Klassen, Interfaces oder anderer Sourcecode sollen getrennt, nämlich hier umgesetzt werden:

```
src/main/java/de/dhbw/course1/exercises
```

Übungsaufgabe 1

siehe Kapitel → "das erste Klassenmodell"

Übungsaufgabe 2a

Erzeuge ein zweites Klassenmodell aus dem folgenden Anwendungsfall:

→ *"Ein Bahnhof besteht aus einer Wartehalle und Bahnsteigen mit Gleisen. Bahnhöfe werden nach Fern- und Regionalbahnhöfen differenziert."*

Übungsaufgabe 2b

Erweitere dein Modell aus Übungsaufgabe 2, z.B. die Klasse `Bahnhof`, um die Methoden `equals()` und `hashCode()` und schreibe einen Test zur Prüfung ob 2 Bahnhöfe gleich sind.

Übungsaufgabe 3

Versuche "Reference Semantics" grafisch zu erläutern!

Testfragen

Im Modul `/exam` finden sich weitere kleine Übungen für die Inhalte des Kurses 1, und zwar in der dortigen Testklasse:

```
<your-repo>/exam/test/de/dhbw/exam/course1/ExamTest.java
```