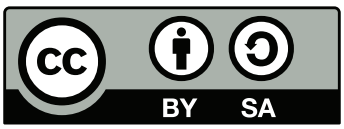
# CleanBoot - Ein Vorschlag für saubere Backendprogrammierung mit Java/Springboot

Dr. Peter Heinrich, ZHAW <peter.heinrich@zhaw.ch>

29. Januar 2026

# Lizenz



# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

Schön, dass Du dieses Skript vor dir liegen hast. Entweder als Vorkurs für den *CAS Secure Software Design and Development* oder einfach so, weil Du Dich für sauber gebaute Software interessierst.

Sauberer Code wird immer wichtiger, da unsere IT-Landschaften immer komplexer werden. Code, den wir verstehen, wird die Grundlage allen Vertrauens in die IT bleiben - auch wenn einem gerade jeder und jede einreden möchte, dass Softwareentwicklung zukünftig über KI stattfinden wird und zwar in einer Form, der man nur noch seine Wünsche eingeben muss und dann innert Minuten die fertige Software bekommt. So allgemein, wie es gerne verkauft wird kann das jedoch nicht funktionieren, denn mindestens muss man seine Anforderungen vollständig formulieren können, oder der Generator trifft willkürliche Annahmen, um die Lücken zu füllen. Dieses Argument ist selbst bei einem perfekt funktionierenden Generator nicht zu entkräften - Halluzinationen der gängigen LLMs machen das alles zuweilen noch schlimmer. In der Informatik gibt es (genau wie in jeder anderen Ingenieursdisziplin) viele richtige und gute Wege ein Problem zu lösen und gesetzte Anforderungen zu erfüllen. Schon darum ist es unmöglich aus einer Problemdefinition **die** Lösung abzuleiten - schon gar nicht im formalen Sinn. Softwareentwicklung wird daher immer ein kreativer Prozess bleiben. Anderenfalls müssen wir starke Einschränkungen des Lösungsraums in Kauf nehmen, wie dies die Low- und No-Code-Plattformen realisieren oder aber wir akzeptieren willkürliche und zufällige Lösungen.

Dieses Skript ist bewusst KI-agnostisch gehalten. Es ist allgemein unwichtig, mit welchen Werkzeugen jemand arbeitet, um ein Ergebnis zu erzielen. Wenn man entsprechende Prompts formuliert, erhält man bestimmt auch Lösungen im Sinne dieses Skriptes. Aber genau das ist der Punkt - wenn die Modelle gut funktionieren ist deren Outputqualität auch immer abhängig von der Qualität des Inputs. Genau darum ist es umso wichtiger dass man den Code wirklich versteht. Sonst kann man gar nichts mehr überprüfen.

Wie dem auch sein mag - wir wollen uns ganz manuell mit dem Erstellen von sauberem Code beschäftigen und dabei ganz besonderen Wert auf das Erzeugen eines gutes Modell von unserer Domäne legen. Mit oder ohne KI - wir müssen die Domäne verstehen, um überhaupt irgendetwas nützliches bereitstellen zu können.

Wir werden mit Hilfe von zwei Projekten Schritt für Schritt eine Business-App bauen, um den Entwicklungsstack besser kennenzulernen. Ganz bewusst adressieren wir in diesem Skript vor allem das Thema Korrektheit und Testbarkeit von Code. Alleine dadurch werden wir ganz viele Security-Themen auch gleich mit erledigen. Oft sind sie eine Folge von schlampiger Programmierung und nicht etwa auf das Fehlen von zusätzlicher Technik und Komplexität zurückzuführen. Input-Validierung werden wir z.B. kaum extra benötigen, da unser Domänenmodell derart strikt sein wird, dass wir überhaupt keine Objekte instanziieren, die nicht valid sein könnten - ganz egal woher die Daten kommen. Oder SQL-Injections - das kann und darf es bei sauberer Entwicklung nicht geben. Klar, es gibt noch x weitere Themen wie Authentication, Authorization, Domain-Model-ACL etc., aber das schauen wir im Kurs an.

Ganz viel Wert werden wir auf das Thema *Separation of Concerns* legen. Wir lassen nicht zu, dass sich Zuständigkeiten in der Codebasis ausbreiten. REST-spezifischer Code gehört ausschliesslich in den Boundary Code, OR-Mapping machen wir nur im Persistenzbereich. Unsere Businesslogik und das Domain-Model sollen und dürfen nichts davon wissen. Anderenfalls gibt es überall kreuz und quer Abhängigkeiten. Das wird eine Hölle zum Testen und verhindert jegliche Portierbarkeit, falls man z.B. die zugrundeliegende Datenbanktechnologie wechseln möchte oder irgendwann mal von Spring Boot wegmigrieren will.

Ebenso werden wir uns soweit irgend möglich von strikten, objektorientierten Entwurfsmustern verabschieden. Es muss nicht alles als vererbte Struktur abgebildet werden. Auch von der Idee, das alles immer und um jeden Preis ein Objekt im klassischen Sinne ist und aus Daten und Verhalten besteht, ist mittlerweile als überholt zu betrachten. Es gibt schon einen Grund, warum *Value Objects* oder *Data Transfer Objects* existieren oder warum man auf Strukturen ausweicht, die *immutable* sind. Aber wir machen das nicht pedantisch sondern dort wo es am meisten Sinn stiftet.

Bevor wir aber zu technisch werden, noch ein paar allgemeine Bemerkungen, vor allem warum wir auch im Jahr 2026 in einem neuen Kurs noch Java einsetzen.

## 1.1 Java ist längst aus der Mode gekommen!

Mode - manche würden das auch Hype nennen - ist offenbar ein wichtiges Entscheidungskriterium für die Auswahl von Technologie. Leider oftmals mit teuren Konsequenzen. Man setzt lieber auf Neues und Unerprobtes das dann viele 'Kinderkrankheiten' hat, viele Probleme, die früher gelöst waren doch nicht adressiert, das ganze über Jahre erarbeitete Praxiswissen obsolet macht und sich letztlich in seinen Konzepten irgendwann selbst wiederholt.
Ein schönes Beispiel sind Konzepte zur Systemintegration. Kennt ihr noch CORBA? Das war die Integrationslösung der 90er-Jahre. Man hatte eine abstrakte Sprache, um ein Interface zu definieren - meist im Sinne von Remote Procedure Calls. Daraus wurden dann sogenannte Stubs für die gewünschte Programmiersprache generiert. Eine solide Idee: Contract first - mit Unterstützung für Interprozesskommunikation über Systemgrenzen hinweg. In den 2000ern kam dann XML auf. Schnell wollte man die Binärübertragungen von CORBA loswerden und erfand zunächst XML-RPC, später dann SOAP. Das war im Grunde das Gleiche in Grün - nur viel komplizierter (okay, fairerweise konnte es auch mehr, etwa durch flexiblere Transportprotokolle). Die zunehmende Komplexität war jedoch oft hinderlich, und HTTP war im Web längst der Standard für Server-/Browser-Kommunikation. Etwa zehn Jahre später war dann REST das Mass der Dinge. REST basiert direkt auf HTTP und hat eine andere Semantik als SOAP - die Operationen werden über HTTP-Methoden wie GET, PUT, POST, DELETE oder OPTIONS abgebildet. Man spricht von Ressourcen statt Prozeduren und verzichtet auf eine formale Beschreibung der Interfaces. Das bringt zwar theoretisch Flexibilität, aber in der Praxis ist man oft doch auf ganz bestimmte API-Versionen angewiesen - schliesslich müssen die Daten genau im erwarteten Format ausgetauscht werden. Später hat man diese Beschreibbarkeit mit Swagger/OpenAPI wieder hinzugefügt.

Google hat dann gRPC ins Leben gerufen. Das unterstützt wieder Binärübertragung, man definiert wieder formal das Interface - und kompiliert daraus erneut Stubs in der jeweiligen Zielsprache. Eigentlich ist man damit 20 Jahre später wieder bei einem sehr ähnlichen Konzept wie CORBA gelandet. Man stelle sich vor, wie viel Geld und Ressourcen diese Odyssee gekostet hat. Und der Zoo an Protokollen bleibt bestehen - denn viele Systeme laufen deutlich länger als ursprünglich angenommen. Rückwärtskompatibilität wird erwartet. Spaß am Rande: Modernste 64-Bit x86-Prozessoren starten noch heute im Real Mode - sie emulieren beim Booten den 8086 aus dem Jahr 1978 im 16-Bit Modus!
Aber zurück zu den Sprachen. Das Alter einer Sprache hat meist nichts damit zu tun, ob die Sprache 'veraltet' ist. C ist das beste Beispiel dafür. Nach über 50 Jahren ist es noch immer weit verbreitet, mindestens im Bereich der Betriebssysteme sowie bei Embedded und IoT-Projekten aber auch grosse Teile der Userland-Werkzeuge in Linux sind in C geschrieben. Pascal hingegen ist

in etwa gleich alt - wird aber kaum noch verwendet (Delphi hält das noch etwas am Leben). Somit gibt es kaum noch eine Community rund um die Sprache, und somit wird auch kaum Infrastruktur (im Sinne von Frameworks) um die Sprache herum gebaut. Relevante Applikationen in Pascal zu schreiben, wird dadurch gleichwohl mühsamer im Vergleich zu anderen Sprachen. Pascal ist somit leider wirklich als veraltet zu betrachten.

Für Java scheint das nicht zu stimmen. Nach aktuellen Statistiken ist es direkt nach Python, auf Platz zwei der meistgenutzten Programmiersprachen [3]. Nicht nur beherrscht Java 'moderne' Konstrukte wie z.B. funktionale Programmierung, sondern kann auch auf eine erfolgreiche Vergangenheit im Enterprise-Umfeld (vor allem Versicherungen und Banken) zurückblicken.

Persönlich finde ich Java lesbarer und schöner als Python aber das ist Geschmackssache und auch eine Frage der Erfahrungen. Die Sprache ist ja meist nicht verantwortlich dafür, ob Code gut oder schlecht ist. Dennoch wird man nicht müde auf Java herumzuhacken weil eine gerade gehypte Sprache angeblich alles besser kann. Und genau hier ist das Problem - keine Sprache kann alles besser, da verschiedene Programmiersprachen parallel existieren um ganz verschiedene Arten von Problemen zu lösen. Seit Turing wissen wir, dass wir jede Sprache in jede andere umwandeln können. Theoretischer Natur kann dieses "besser" also schon mal gar nicht sein. Es ist eben Mode. Dass man im Data-Science-Umfeld besonders auf Python setzt liegt nicht inhärent an der Syntax der Sprache sondern vielmehr an der Verfügbarkeit von Libraries und Frameworks sowie an einer lebendigen Community.

Gesamtheitlich betrachtet bin ich nach wie vor der Meinung, dass Java eine geeignete und noch immer zeitgemässe Sprache für die Entwicklung von Businessapplikationen ist. Man darf aber gerne anderer Meinung sein solange man zeigen kann, dass man in einer alternative Umgebung ebenfalls guten Code erzeugen kann. Aber was ist eigentlich guter Code?

## 1.2 Guter und schlechter Code

Für die Bewertung von Programmcode gibt es unzählige Metriken. Qualität von Software zu quantifizieren versucht man bereits seit den 1980er Jahren mit mässigem Erfolg. Lines-of-Code, Zyklomatische Komplexität, Wartbarkeits-Index u.s.w. Allesamt sind dies aber generische und meist sprach- unabhängige Masse. Aber woher kommen die Vorstellungen, dass C-Code weniger sicher ist als z.B. Code in Python? In JavaScript würde auch niemand ein Betriebssystem schreiben wollen. Nur warum?

Das liegt daran, dass bestimmte Kategorien von Fehlern in manchen Sprachen nicht möglich sind. C ist typisiert, JavaScript hingegen nicht. Fehler durch Zuweisungen von inkompatiblen Typen fallen bei C bereits dem Compiler auf, bei JavaScript erst bei der Ausführung. Python hat im Gegensatz zu C echten Speicherschutz. Ein Buffer-Overflow mit fatalen Folgen (Crash bis hin zu Remote-Code-Execution) ist in einer Python-Umgebung erst mal nicht möglich - zumindest nicht durch die eigenen Programmierfehler.

Der Rest ist Sache des Programmierers und der Frameworks. Es gibt Umgebungen (z.B. PHP) die geradezu animieren, hässlichen Code zu produzieren, der alle Abstraktionsebenen und Verantwortungsbereiche vermischt. Saubere Schichtentrennung und modulares Design wird darin dann schwierig. Und dennoch gibt es viele gute Gegenbeispiele.

Java hat hingegen den Nimbus Code-Qualität durch überbordende Komplexität einzubüssen."Enterprise"-Patterns werden teils verspottet, unzählige Klassen zu produzieren um einfachste Probleme zu lösen. Und dennoch kann man kompakten Code schreiben.

Fazit: Schlechter Code ist primär den Programmierenden geschuldet, sofern man sich der Limitierungen seiner Umgebung bewusst wird. Aber auch der Technologiewahl. Manche Umgebungen (Node.js als negativbeispiel) erzeugen z.B. regelmässig riesige Abhängigkeitsbäume so dass die SBOM gross und schwer zu warten wird. Hier liegt die Gefahr in Sicherheitsmängeln der nahezu unüberschaubaren Anzahl von Drittsoftware, die in das eigene Projekt eingebunden wird (das ist in Python/pip-Umgebungen übrigens kaum besser).

In diesem Script möchte ich mit Java und der Spring-Boot-Umgebung zeigen, wie wir (trotz eventuell bestehender Vorbehalte) sauberen Code erzeugen, der vor allem fachliche Aspekte beschreibt, sinnvolle Abstraktions- und Verantwortungsbereiche trennt und so verständlich wie möglich ist. Dadurch werden wir in der Lage sein, sichere und wartbare Systeme zu erstellen.

## 1.3 Warum noch ein Script?

Bücher zu Java gibt es ja genug, aber mir ist keines bekannt, dass den Inhalt genau auf unsere Bedürfnisse des Security-Kurses angepasst hätte, sodass entweder wichtige Themen fehlen würden oder noch viel 'Ballast' zusätzlich vermittelt wird, der uns in diesem Kontext gar nichts bringt. Mit der Entscheidung ein neues Skript zu erstellen kann ich einen Mittelweg einschlagen. Zudem soll dieses Skript auch nur als zusätzliches Nachschlagewerk dienen. Zu allen Themen gibt es dann auch noch Video-Lektionen und Übungen um das Wissen Schritt für Schritt zu erarbeiten.

Und natürlich, weil ich Freude daran hatte, auch endlich mal ein "Buch" zu schreiben. Freude daran, die Dinge genau so darzustellen, wie ich mir gewünscht hätte, dass man sie mir erklärt hätte. Anhand von Beispielen und immer mit dem Bestreben, den Sinn dahinter zu erklären. Es ist nicht wichtig dass man Dinge so macht, wie sie angeblich gemacht werden - sondern es ist viel wichtiger zu verstehen, warum man sie so (oder eben anders) macht.

## 1.4 Warum eigentlich Springboot?

Tatsächlich haben wir uns lange überlegt, welchen Technologiestack wir für unseren Weiterbildungskurs einsetzen wollen. Letztlich fiel die Wahl auf Spring Boot. In Spring Boot sind 25 Jahre Java-Enterprise-Erfahrungen eingeflossen, immer aus den Fehlern lernend und immer bestrebt mit weniger Boilerplate-Code auszukommen. Am besten rein fachlich arbeiten. Zudem steht mit Spring Secuity ein leistungsfähiges und ebenfalls langzeiterprobtes Security-Framework zur Verfügung, das praktisch alle Belange im Bereich Authorization und Authentication abdecken kann. Alle Teile von Spring Boot werden zentral verwaltet und sind aus einem Guss. Es besteht zwar aus vielen Komponenten, wird aber nicht zur Dependency-Hölle, selbst bei anspruchsvollen Designs. Und - man lernt ohnehin immer nur Konzepte. Sprachen kommen und gehen aber die tiefer liegenden Überlegungen und Problemlösungsstrategien bleiben erhalten und sind auf beliebige, andere Umgebungen anwendbar. Apropos andere Umgebungen: Java ist portierbar und steht auf allen grossen Plattformen zur Verfügung. Mac, Windows, Linux und das ganze auch für ARM64 oder X86. Und es benötigt kaum Infrastruktur. Ein simples Java-JDK genügt. Keine Gigabyte grossen und schwerfälligen Entwicklungsumgebungen. Im Prinzip genügt ein guter Editor.

## 1.5 Die ersten Gehversuche

So, nun geht es aber los mit Docker-DevContainers, und IntelliJ, einer PostgreSQL-DB, einem Applikationsserver ... nein! Nichts davon benötigen wir. Wir starten ganz einfach mit lokalem Development und werden sehen, dass uns das überhaupt nicht im Wege steht. Halten wir uns an die Standards, werden wir schnell bemerken, dass wir auch keine generative KI benötigen, um zum 10x-Developer zu werden. Wir sind es schon, da wir den Aufwand minimieren und bestrebt sind qualitativ hochwertigen Code zu erzeugen. Boilerplate-Code vermeiden wir soweit das irgend geht. Also los!

### 1.5.1 JDK installieren

Alles, was es zum Start benötigt ist das Java-Development-Kit oder JDK. Wir raten aus lizenztechnischen Gründen immer zu einem quelloffenen JDK. Unter Linux entweder OpenJDK oder (und das gilt auch für die anderen Plattformen) das JDK von Adoptium.net. Unter Windows muss man ggf. noch selbst dafür sorgen, dass Java im Systempfad vorhanden ist. Wenn man auf der Command-Line Java aufruft und in etwa das gezeigte Resultat bekommt, hat man alles richtig gemacht:

```
% java -version
openjdk version "21.0.7" 2025-04-15 LTS
OpenJDK Runtime Environment Temurin-21.0.7+6 (build 21.0.7+6-LTS)
OpenJDK 64-Bit Server VM Temurin-21.0.7+6 (build 21.0.7+6-LTS, mixed mode, sharing)
```

Zusätzlich sollte als Umgebungsvariable noch JAVA_HOME gesetzt sein und auf das installierte JDK zeigen. Die exakten Versionen sind dabei nicht so wichtig. So lange wir mind. auf Java 21 arbeiten ist das gut.

### 1.5.2 Git installieren

Falls nicht längst geschehen, müssen wir git installieren. Es gibt keine Ausrede ohne Versionskontrollsystem zu arbeiten. Es muss auch nicht auf einem Server liegen - schon gar nicht zum experimentieren. Git ist dafür ausgelegt, komplett lokal zu laufen. Eigentlich kann es auch gar nichts anderes. Clone/Push/Pull sind spezifische Befehle, den lokalen Stand mit einem Server zu synchronisieren. Commits und Checkouts passieren ohnehin immer erst mal nur lokal.

Probieren wir das doch gleich mal in einem Verzeichnis eurer Wahl aus:

```
% mkdir playground
% cd playground
% git init .
```

## 1.6 Minimales Maven Projektsetup

Man kann sich das Leben einfach machen und https://start.spring.io nutzen, um sich ein Minimalprojekt “zusammenzuklicken” und anschliessend als .zip herunterladen. Oder man nutzt seine Entwicklungsumgebung um ein ’leeres’ Maven-Projekt zu erstellen. Für unser Setup machen wir aber zum Üben einmal alles von Hand um wirklich zu verstehen, was hinter den ganzen Dateien und Verzeichnissen steckt. Generell ist es ohnehin keine gute Praktik ZIP-Files irgendwo herunterzuladen und darin enthaltene Binaries auch noch auszuführen. Auch wir laden im Folgendem fertige Shell-Scripts aus dem Apache GitHub-Repo herunter. Es wird bei solchen Schritten dringend empfohlen die Scripte zumindest grob in Augenschein zu nehmen bevor man diese ausführt. Immerhin ist das Programmcode fremder Personen. Aber der Reihe nach. Zuerst braucht unser Projekt eine geeignete Verzeichnisstruktur.

### 1.6.1 Verzeichnisstruktur anlegen

Maven-Projekte folgen einer einheitlichen Verzeichnis-Struktur, die wir leicht manuell erzeugen können:

```
$ mkdir -p src/main/{java,resources}
$ mkdir -p src/test/{java,resources}
$ mkdir -p src/main/java/ch/zhaw/ssdd/demo
$ mkdir -p src/test/java/ch/zhaw/ssdd/demoTest
$ touch pom.xml
```

Es ist von den Namen her recht klar, was später wo abgelegt wird. Compilierte Artefakte liegen später in ./target, das bei der ersten Ausführung angelegt wird. Die Substruktur "ch/zhaw/ssdd/demo"repräsentiert das Java-Package ch.zhaw.ssdd.demo in welchem wir später unseren Code ablegen wollen. pom.xml ist die zentrale Konfigurationsdatei von Maven, das wir benötigen, um unsere Software automatisch zu übersetzen, packetieren und zu testen. Um das zu nutzen, müssen wir Maven istallieren.

### 1.6.2 Maven-Wrapper installieren

Natürlich könnte man Maven auch systemweit installieren, aber in letzter Zeit hat es sich bewährt, Maven als Teil des Repositories auszuliefern. Nun wäre es unangebracht, Binärdateien (jar-Files) im Repository zu verwalten, die sonst nichts mit der Codebasis zu tun hätten. Daher behilft man sich mit Wrapper-Skripten, welche die Maven-Funktionalität zur Verfügung stellen, indem sie das Maven-Binary zur Laufzeit herunterladen und ausführen. Somit ist keine lokale Maven-Installation mehr notwendig. Mit den folgenden Befehlen laden wir die Skripte für Linux/OSX und Windows herunter und erstellen noch ein notwendiges Konfigurationsfile, damit auch die richtige Maven-Version Verwendung findet. Da wir das Skript direkt von den Maven-Wrapper-Sources nehmen, müssen wir noch selbst die Versionsnummer ersetzen. Dieser Schritt ist notwendig, da wir nicht daran interessiert sind den Maven-Wrapper komplett zu als Binärpaket zu bauen, sondern lediglich die Skripte benötigen.

```
$ curl -o mvnw https://raw.githubusercontent.com/apache/maven-wrapper/refs/tags/mave␣
↪  n-wrapper-3.3.4/maven-wrapper-distribution/src/resources/only-mvnw
$ curl -o mvnw.cmd https://raw.githubusercontent.com/apache/maven-wrapper/refs/tags/␣
↪  maven-wrapper-3.3.4/maven-wrapper-distribution/src/resources/only-mvnw.cmd
$ sed -i "s/@@project.version@@/3.3.4/g" mvnw
$ sed -i "s/@@project.version@@/3.3.4/g" mvnw.cmd
$ chmod u+x ./mvnw
$ mkdir -p .mvn/wrapper
$ cat > .mvn/wrapper/maven-wrapper.properties << EOF
wrapperVersion=3.3.4
distributionType=only-script
distributionUrl=https://repo1.maven.org/maven2/org/apache/maven/apache-maven/3.9.12/␣
↪  apache-maven-3.9.12-bin.zip
EOF
```

Note: Mac-Nutzer nehmen 'gsed' anstelle von 'sed', da die sed-Variante von OSX/BSD das Flag -i nicht kennt und somit keine Ersetzungen in der Datei vornehmen kann.

**§ 1.1 Es werden niemals Skripte heruntergeladen und unüberprüft ausgeführt!**

Security betrifft auch unseren Buildprozess. Sich Malware über heruntergeladene Skripte und Binaries einzufangen sollte wirklich vermeidbar sein.

Eigentlich wären wir an dieser Stelle fertig, aber es ist generell eine gute Praktik bei solchen automatischen Downloads Checksummen zu verifizieren. Sicherheit fängt in der Toolchain an. Zumindest einmal sollte man sich die Mühe machen, den Hash auf der Downloadseite zu finden und manuell zu überprüfen. Noch besser wäre es natürlich auch die Signatur zu prüfen. Das müssten wir aber von Hand machen. Wir benötigen lediglich eine Installation von 'gpg'. Neben dem Binary und dem Hash benötigen wir auch die Signatur-Datei.

```
$ curl -o apache-maven-3.9.12-bin.zip https://repo1.maven.org/maven2/org/apache/mave␣
↪  n/apache-maven/3.9.12/apache-maven-3.9.12-bin.zip
$ curl -o apache-maven-3.9.12-bin.zip.asc https://repo1.maven.org/maven2/org/apache/␣
↪  maven/apache-maven/3.9.12/apache-maven-3.9.12-bin.zip.asc
$ curl -o apache-maven-3.9.12-bin.zip.sha512 https://repo1.maven.org/maven2/org/apac␣
↪  he/maven/apache-maven/3.9.12/apache-maven-3.9.12-bin.zip.sha512
$ gpg --verify apache-maven-3.9.12-bin.zip.asc apache-maven-3.9.12-bin.zip
gpg: Signature made Sat Dec 13 10:19:06 2025 CET
gpg:                using RSA key 84789D24DF77A32433CE1F079EB80E92EB2135B1
gpg:                issuer "sjaranowski@apache.org"
gpg: Can't check signature: No public key
```

GPG bietet keine PKI an, wie wir sie vom Internet her kennen. Insofern kennt unser System den Public Key des Entwicklers noch nicht. Bevor wir den Key importieren, tun wir uns gut daran, diesen zuerst zu verifizieren. Im einfachsten Fall stellt uns Apache ein KEYS file zur Verfügung, dass

den Fingerprint beinhaltet. In unserem Fall wäre dies über https://downloads.apache.org/maven/KEYS zu beziehen. Hat man keine andere Quelle, kann man einfach mal mit einer Suchmaschine danach suchen und Belege sammeln, dass dieser Key auch wirklich zu der angegebenen Mailadresse gehört und diese auch mit einem Entwickleraccount bei apache verknüpft ist. Man sieht auch hier wieder wie schwierig es in der Praxis ist, eine Trustverbindung herzustellen.

```
$ gpg --keyserver keys.ubuntu.com --recv-key 84789D24DF77A32433CE1F079EB80E92EB2135B1
$ gpg --verify apache-maven-3.9.12-bin.zip.asc apache-maven-3.9.12-bin.zip
gpg: Signature made Sat Dec 13 10:19:06 2025 CET
gpg:                using RSA key 84789D24DF77A32433CE1F079EB80E92EB2135B1
gpg:                issuer "sjaranowski@apache.org"
gpg: Good signature from "Slawomir Jaranowski <sjaranowski@apache.org>" [unknown]
gpg:                 aka "Slawomir Jaranowski <s.jaranowski@gmail.com>" [unknown]
gpg: WARNING: The key's User ID is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 8478 9D24 DF77 A324 33CE  1F07 9EB8 0E92 EB21 35B1
```

Die Warnung war zu erwarten, da wir keine weiteren Elemente der Signaturkette eingefügt haben. Für unsere Zwecke genügt die manuelle Verifikation an dieser Stelle. Wir überprüfen noch kurz den Hashwert und erzeugen noch selbst einen SHA256-Hashwert, der uns nicht mit ausgeliefert wird:

```
$ shasum -a 512 apache-maven-3.9.12-bin.zip
0ef0d074761a55e7b982fd7f2cd6ea2da028289c73156ebd488c0513efbff8a64ac5bc0c11c3b8bbced9
↪  30ce15238ecef344b0e065ae54f7cdca8c86cf39e736
↪  apache-maven-3.9.11-bin.zip
$ cat apache-maven-3.9.11-bin.zip.sha512
0ef0d074761a55e7b982fd7f2cd6ea2da028289c73156ebd488c0513efbff8a64ac5bc0c11c3b8bbced9
↪  30ce15238ecef344b0e065ae54f7cdca8c86cf39e736
$ shasum -a 256 apache-maven-3.9.12-bin.zip
305773a68d6ddfd413df58c82b3f8050e89778e777f3a745c8e5b8cbea4018ef
↪  apache-maven-3.9.12-bin.zip
```

Die Zahlen stimmen überein. Den zuletzt berechneten Hash legen wir nun noch in der Datei ".mvn/wrapper/maven-wrapper.propertiesünter dem Schlüssel `distributionSha256Sum` ab. Nun ist alles in Ordnung und auch zukünftig über den Hash geschützt. Wir können die ganzen Dateien, die wir zusätzlich heruntergeladen haben, wieder löschen.

### 1.6.3 Minimales pom.xml

Es ginge sicher noch minimalistischer aber mit diesem Stand beginne ich meist meine SpringBoot-Projekte wie in Listing 1.1 gezeigt.

Wie wir sehen, verwenden wir Spring Boot in der Version 3.5.6. Dies wird über den 'Parent' festgelegt. Alle nachfolgenden Dependencies erhalten keine Versionsnummer mehr, da diese implizit über den 'Parent' gegeben sind. Somit können wir davon ausgehen, dass nur Pakete mit kompatiblen

Versionen geladen werden. Version 3.5.6 ist lediglich jetzt aktuell. Alles im 3.x.x-Bereich sollte funktionieren.

```
<?xml version="1.0"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ch.zhaw.ssdd.demo</groupId>
  <artifactId>SpringDemo</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <name>SpringDemo</name>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.target>21</maven.compiler.target>
    <maven.compiler.source>21</maven.compiler.source>
  </properties>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.5.6</version>
    <relativePath />
  </parent>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
  </dependencies>
</project>
```

Listing 1.1: pom.xml in einer minimalen Ausführung

Nun wäre auch ein guter Zeitpunkt schon mal ein LICENSE-File anzulegen. Mein Code ist ohnehin Open Source, daher finde ich die MIT-License als angebracht.

```
Copyright 2025 Zurich University of Applied Sciences, Peter Heinrich

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
IN THE SOFTWARE.
```

Listing 1.2: LICENSE

Zum Schluss fügen wir noch ein .gitignore dem Projekt hinzu, so dass unnötige, heruntergeladene oder kompilierte Artefakte nicht im Repository landen.

```
.mvn/
target/
*.log
*.jar
```

Listing 1.3: .gitignore

Fertig! Ein guter Stand für unseren ersten commit:

```
$ git add .
$ git commit -m "initial commit"
```

## 1.7 Hello World, Hello Spring Boot

Natürlich folgt jetzt das übliche HelloWorld-Programm. Allerdings direkt als Spring-Boot-Applikation.

```java
package ch.zhaw.ssdd.demo;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringDemo implements CommandLineRunner {
    public static void main(String[] args) {
        SpringApplication.run(SpringDemo.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Hello, World!");
    }
}
```

Listing 1.4: ./src/main/java/ch/zhaw/ssdd/demo/SpringDemo.java

Wenn wir mit einer IDE arbeiten, können wir die Main-Methode direkt starten, sonst geht das aber auch bequem über maven:

```
$ ./mvnw spring-boot:run
```

Wir hätten unser Print-Statement auch in die Main-Methode schreiben können, der Vorteil des Umweges über den CommandLineRunner liegt darin, dass zum Ausführungszeitpunkt Spring Boot vollständig gestartet ist und wir alle seine Vorzüge bereits nutzen können. Schauen wir uns Grundlegende Dinge an, die Spring uns direkt bietet.

## 1.8 Inversion of control pattern

Die Idee dieses Patterns stammt aus der Beobachtung, dass starke Abhängigkeiten zwischen Klassen entstehen, so dass sich einzelne Teile einer Applikation nicht mehr einzeln instantiieren lassen. Somit sind die Komponenten kaum wiederverwendbar - aber schlimmer noch - auch nicht alleinstehend testbar. Das ist vor allem für Unit-Tests problematisch da wir hier ja das “Um-”System genau nicht mit testen möchten.

Um das zu demonstrieren, überlegen wir uns folgendes Trivialbeispiel: Nehmen wir an, wir bauen ein System, das Grafikelemente (Shapes) verarbeitet und diese an einer Stelle drucken möchte. Dafür haben wir uns eine Klasse ShapePrinterService geschrieben:

```
public class ShapePrinterService {
    public void printShape(Shape s) {
        if(s instanceof Triangle) {
            System.out.println("     *     ");
            System.out.println("    * *    ");
            System.out.println("   *   *   ");
            System.out.println("  *     *  ");
            System.out.println(" * * * * * ");
        }
        else if(s instanceof Square) {
            // ...
        }
        // ...
    }
}
```

Listing 1.5: Shape printer in Java-Pseudocode

Nehmen wir nun weiter an, dass wir eine Methode (in einer anderen Klasse) haben, die diesen ShapePrinterService benutzt.

```
public class ListPrinter {
    private ShapePrinterService shapePrinterService;

    public ListPrinter() {
        shapePrinterService = new ShapePrinterService();
    }

    public void printList(List<Shape> shapes) {
        for (Shape s : shapes) {
            shapePrinterService.printShape(s);
        }
    }
}
```

Listing 1.6: Invoking the ShapePrintServer Java-Pseudocode

Wie wir sehen, sind die beiden Klassen ShapePrinterService und ListPrinter fest verbunden. ListPrinter lässt sich ohne die konkrete Implementierung des ShapePrinterService nicht übersetzen und somit auch nicht einzeln testen. Das könnte man, wie in vielen anderen Sprachen auch durch Interfaces lösen.

### 1.8.1 Interfaces und Implementierungen

Eine erste Entkopplung können wir erreichen, in dem wir Beschreibung und Implementierung trennen. Wir würden also ein Interface anlegen, welches den ShapePrintService beschreibt:

```
public interface ShapePrinterService {
     void printShape(Shape s);
}
```

Listing 1.7: Interface of ShapePrinterServer

```
public class ShapePrinterServiceImpl implements ShapePrintService {
    public void printShape(Shape s) {
        if(s instanceof Triangle) {
            System.out.println("     *     ");
        // ...
        }
    }
}
```

Listing 1.8: Implementation of ShapePrinterServer in Java-Pseudocode

Von dieser Trennung haben wir direkt aber noch keine Vorteile, da wir die Impl-Klasse noch immer instantiieren würden. Genau damit müssen wir aufhören und die Kontrolle über die Instantiierung externalisieren. Das ist der Kern von Inversion-of-Control.

### 1.8.2 Constructor based injection

Ein eleganter Weg dies umzusetzen ist es auf Instantiierungen zu verzichten und sich die fertigen Objekte im Konstruktor übergeben zu lassen. Im folgendem Listing ist dies umgesetzt:

```
public class ListPrinter {
    private final ShapePrinterService shapePrinterService;

    public ListPrinter(ShapePrinterService printService) {
        this.shapePrinterService = printService;
    }

    public void printList(List<Shape> shapes) {
        for (Shape s : shapes) {
            shapePrinterService.printShape(s);
        }
    }
}
```

Listing 1.9: Constructor Based Injection

Somit sind wir einen grossen Schritt weiter. Der ListPrinter ist nicht mehr abhängig von einer bestimmten Implementierung ShapePrinter. Wir können uns leicht vorstellen, dass wir zum Testen eine einfachere Version (andere Instantiierung) verwenden können und wollen. SpringBoot automatisiert das für uns. Wenn wir zum Beispiel unsere Klassen mit @Component dekorieren, wird die

ganze Arbeit der Instantiierung der Objekte und ihrer Abhängigkeiten zur Laufzeit übernommen. Starten wir (z.B. für einen Test) Ohne den Spring-Kontext werden diese Dekoratoren schlichtweg ignoriert und wir können unsere Objekte, z.B. selbst verwalten. Aber das sehen wir nachher am Beispiel wenn wir Unit-Tests schreiben.

**§ 1.2 Wir gehen keine Unnötigen Abhängigkeiten durch manuelle Instantiierung ein.**

Der Aufruf von `new` bindet unsere Klasse unweigerlich and das zu instantiirende Objekt. Ist das ungewollt, schwächt das schnell die Testbarkeit unseres Codes, da dieser nicht mehr isoliert lauffähig ist.

### 1.8.3 Umgebungs- und Konfigurationsvariablen

Weiter zur Basis-Infrastruktur gehört der Umgang mit Umgebungs- und Konfigurationsvariablen. Diese stehen über die Environment-Klasse zur Verfügung. Wir können diese, genau wie im Abschnitt zuvor besprochen, einbinden.

```
@SpringBootApplication
public class SpringDemo implements CommandLineRunner {

    private final Environment environment;

    public SpringDemo(Environment environment) {
        this.environment = environment;
    }

    public static void main(String[] args) {
        SpringApplication.run(SpringDemo.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println(environment.getProperty("PATH"));
    }
}
```

Listing 1.10: Properties einlesen

Automatisch stehen uns somit alle Konfigurationsvariablen (z.B. aus der Datei ./src/main/resources/application.properties) zur Verfügung aber ebenso auch alle Umgebungsvariablen. Im Beispiel geben wir die Pfad-Variable vom System aus. In einem realen System könnten hier aber auch API_Keys oder URLs geladen werden, die wir aus Sicherheitsgründen nicht im Repository halten können, sondern die der Deployment-Umgebung erst bei der Ausführung übergeben werden.

Alle weiteren Konzepte schauen wir uns aber an kleineren und grösseren Beispieln an.

# Kapitel 2

# Aller Anfang ist leicht - aber nicht immer trivial

In diesem Kapitel schauen wir uns zuerst an, wie man es aus meiner Sicht eher *nicht* machen sollte. Also ein Antipattern der guten Entwicklungspraxis. Leider findet man solche Code-Schnipsel ohne Warnung auf vielen Tutorial-Seiten. Der Grund liegt wahrscheinlich darin, dass es sehr einfach und verständlich aussieht. Und genau darum wollen wir das auch zuerst so machen. Aber keine Sorge - das Wissen wird nicht vergeblich aufgebaut. Immerhin lernen wir dabei gleich auch wie SpringBoot funktioniert. So belasten wir uns nicht gleichzeitig mit der Komplexität des Frameworks und des neuen Ansatzes. Im Laufe des Kapitels werden wir immer besser und verwandeln unseren anfänglichen Code Schritt für Schritt in Richtung einer sauberen Architektur. Damit es wirklich ganz einfach bleibt, wollen wir ein minimales System bauen, um Bücher zu verwalten. Eine Art Bibliotheksverwaltung aber so simpel wie möglich.

## 2.1 Ganz klassisch: Entity, Controller, Boundary

ECB (*Entity-Controller-Boundary*) ist eines der klassischen Patterns das tief in Frameworks wie *Springboot* verankert ist. Es geht wahrscheinlich auf das Buch *Object-oriented Software Engineering: A Use Case Driven Approach* [6] zurück und beschreibt, wie sich ein Softwaresystem funktional zerlegen lässt. Die Idee dabei ist einfach. Es gibt eine Modellebene (die *Entities*) in der die Datenobjekte modelliert werden, die (Business-)Logik wird in *Controllern* separat gehalten und die Darstellung (oder Präsentation) nach aussen erfolgt über die *Boundary*-Schicht. In ihrer puren Form hat diese Architektur einige Vorteile.

1. Die Logik bleibt von der Darstellung getrennt
2. Die Schichten werden über interfaces voneinander isoliert. Dabei ist sichergestellt, dass die höheren Schichte immer nur auf die unteren Schichten zugreifen aber nie umgekehrt.
3. Zumindest die oberste Schicht ist leicht tausch- und erweiterbar, sollte man zukünftig weitere/andere Schnittstellen benötigen.

Im Laufe der Jahre wurde dieses Pattern immer wieder abgewandelt und leider auch geschwächt. Die allseits bekannten *Model-View-Controller* sind z.B. so ein Negativbeispiel. Dort manipuliert der Controller z.B. direkt die Präsentation und die Daten.

Beginnend mit *Java-EE* und den *Java Beans* hat sich das ECB Pattern auch in den Frameworks manifestiert - in leicht abgewandelt und erweiterter Form.

*Entity* Die *Entity Beans* repräsentieren Objekte aus der Domäne und sind gleichzeitig auch die Datentypen der Persistenz-Schicht. Über *Object-Relational-Mapping* werden diese direkt in die Datenbankschicht serialisiert und deserialisiert.

*Controller* Die Service-Klassen repräsentieren die Business-Logik und steuern die Persistenzschicht. Entweder über ausgelagerte Repositories oder direkt über das Framework mittels *Entity Manager*. Dabei kommt entweder eine abgewandelte, objektorientierte, SQL-Ähnliche Sprache *Java Persistence Query Language* oder natives SQL zum Einsatz.

*Boundary* Die Schnittstelle nach aussen wird meist über Elemente des Frameworks realisiert um z.B. REST-Endpunkte an die eigene Applikation anzubinden. Die Bereitstellung wir dann über einen *Applikationsserver* umgesetzt.

Bevor wir jetzt anfangen an allem herumzunörgeln, bauen wir das einfach mal mit Springboot und schauen uns an, was passiert.

Als *Build-Tool* benutzen wir *Maven*, wie im ersten Kapitel erklärt. Wem die Einführung zu kurz war, der liest am besten noch das Tutorial "Maven in 5 Minutes" [10]. Wer sein Projekt nicht zu Fuss erstellen möchte, kann gerne auch auf `https://start.spring.io` für einen Generator zurückgreifen oder die Funktionen seiner IDE benutzen. Das entsprechende Repository ist unter den Code-Beispielen unter `bookstack_trivial` zu finden. Auf oberster Ebene finden wir neben den Maven-Skripten vor allem auch die *pom.xml* - als das zentrale Konfigurationsfile von Maven. Das wichtigste für uns sind an diesem Punkt die *Dependencies*. Für unser einfaches Beispiel benötigen wir nur ganz wenige Elemente:

- `spring-boot-starter-web`: Die Web-Komponente, damit wir einen integrierten Webserver bekommen und unsere REST-Interfaces exponieren können.
- `spring-boot-starter-data-jpa`: Der OR-Mapper damit wir eine relationale Datenbank als Backend für unsere Persistenzschicht verwenden können.
- `h2`: Eine schlanke SQL In-Memory Datenbank, die man gerne zu Entwicklungszwecken benutzt. Man benötigt somit kein anderes DBMS und kann später durch Konfigurationsänderung auf Postgres, MariaDB, Oracle, oder, oder, oder umsteigen.
- `lombok`: Projekt *lombok* nimmt einen den meisten Boilerplate ab, indem es automatisch Getter/Setter/equals/hashcode oder toString-Methoden generiert. Auch Konstruktoren muss man nicht mehr von Hand schreiben. Funktional notwendig ist das nicht aber enorm angenehm.

Unser gesamtes Projekt besteht aus gerade mal 5 Source-Files.

```
@SpringBootApplication
@AllArgsConstructor
public class BookStack {
        public static void main(String[] args) {
                SpringApplication.run(BookStack.class, args);
        }
}
```

Listing 2.1: Leere Springboot Main-Klasse

Listing 2.1 reicht bereits komplett aus um unseren Spring-Kontext zu starten. Da wir *springboot-starter-web* als Abhängigkeit eingefügt haben bleibt unsere Applikation am laufen, auch wenn die Main-Methode beendet wird. Im Hintergrund wird für uns bereits ein Webserver ( *Tomcat* in diesem Fall) gestartet und unsere Applikation bereits dort automatisch deployed.

### 2.1.1 Persistenz

Beginnen wir im Stack ganz unten mit der Persistenz. Dazu erstellen wir eine einfache Java-Klasse mit ein paar Annotationen, wie Listing 2.9 zeigt:

```
@Entity
@Data
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Long id;

    private String title;
    private String isbn;
}
```

Listing 2.2: Book - Eine Entity-Klasse

`@Entity` ist eine Annotation (technisch eine Art Interface), die SpringBoot zur Laufzeit auswertet und damit Informationen erhält, was es mit der Klasse machen soll. *@Entity* bedeutet für Spring in diesem Fall, dass Instanzen dieses Objekttyps in eine SQL-Tabelle persistiert werden können. Name der Klasse und ihrer Felder werden einfach übernommen. Wenn wir das nicht separate konfigurieren wird zuerst ein Default-Mapping benutzt (*Convention over Configuration*). Wir können alle Namen selbst bestimmen. Wenn wir aber nichts tun, würden wir eine Tabelle `BOOK` mit den Feldern `ID, TITLE und ISBN` erwarten. Mit `@Id` legen wir den Primärschlüssel fest. Dieser ist somit automatisch auch unique aud der DB. Die Annotation `@GeneratedValue` legt fest, dass wir Auto-Increment haben möchten. Fertig! Mehr müssen wir nicht machen.

Damit Spring aber überhaupt weiss, welche Datenbank wir verwenden wollen müssen wir noch eine Konfigurationsdatei im Projektpfad `./src/main/resources/application.properties` anlegen, die bei uns wie folgt aussieht:

```
spring.datasource.url=jdbc:h2:mem:bookstack
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
```

Listing 2.3: Konfiguration über application.properties

Wir weisen spring also an die h2 Datenbank in-memory zu benutzen. Dass wir kein Passwort gesetzt haben ist ok, wie verwenden das momentan nur lokal. Aber Achtung: Die Default-Config würde unsere Anwendung zumindest auf unserem lokalen Netz nach aussen hin exponieren.

Default-Port ist übrigens 8080, aber auch das wäre in dieser Datei konfigurierbar.

Mit `spring.jpa.hibernate.ddl-auto=update` legen wir zudem fest, dass Spring (in diesem Fall spezifisch Hibernate - der verwendete OR-Mapper) uns auch gleich unsere Tabellen anlegt. `spring.h2.console.enabled=true` aktiviert und eine kleine WebApp, mit der wie die Datenbank verwalten können. Probieren wir das doch mal, indem wir unseren Server starten. Das können wir entweder über unsere Entwicklungsumgebung durch Starten der main-Methode machen oder auch über das Terminal mittels:

```
$ ./mvnw spring-boot:run
```

An dieser Stelle auch gleich ein “Gesetz” zum sauberen Aufbau einer Build-Umgebung:

**§ 2.1 Ein Java-Projekt ist auch über die Kommandozeile erstell- und ausführbar**

Saubere Java-Projekte können über ein Text-Terminal erstellt, gestartet und getestet werden. Somit bestehen keine spezifischen Abhängigkeiten zu Entwicklungsumgebungen.

Sollte Spring laufen, können wir uns mit einem Browser auf `http://localhost:8080/h2-console` verbinden und uns einloggen. Dabei müssen wir darauf achten, dass die DB-URL jenem String aus dem Konfigurationsfile entspricht. Nutzername: sa, Passwort: <leer>. Nun können wir unsere Datenbank betrachten (Fig 3.1):

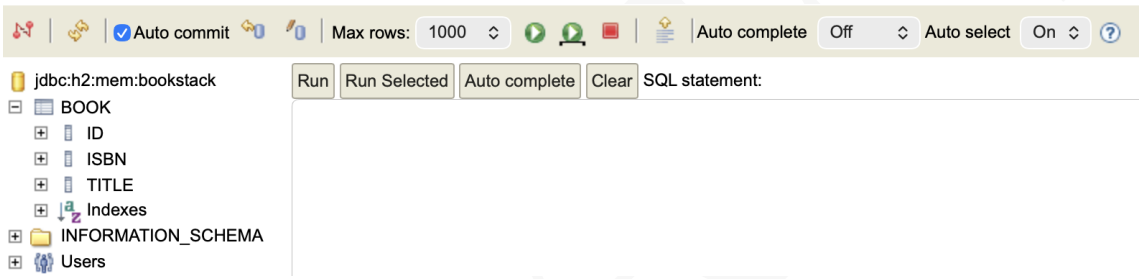



Abbildung 2.1: Zustand der Datenbank nach Programmstart

Damit wir auf unsere Objekte in der Datenbank Zugriff bekommen, benötigen wir noch ein Repository:

```
public interface BookRespository extends JpaRepository<Book, Long> {
    Book findByTitle(String title);
    void deleteByTitle(String title);
}
```

Listing 2.4: Simples Repository für die Book-Entities

Anzumerken ist zu Listing 2.21, dass es sich hierbei nicht um eine Klasse, sondern nur um ein Interface handelt. Wir werden das auch gar nicht selber implementieren. Repository-Code ist derart vorhersehbar, dass Spring uns eine Implementierung zur Laufzeit erzeugt. Da wir `JpaRepository` erweitern, bekommen wir schon die ganzen Standard CRUD-Operaationen wie `findAll()`, `findById()`, `delete()` und einige weitere. Die zusätzlichen Methoden, die wir deklariert haben, folgen einem strikten Namensmuster. Aus `findByTitle` wird Spring selbstständig eine Query a la `SELECT b FROM Book b WHERE b.title = :title`. Das werden auch nach der Übersetzung *Prepared SQL Statements* sein, so dass an diesem Punkt eine SQL-Injection bereits ausgeschlossen wird. Aber das alles bekommen wir nicht zu Gesicht.

### 2.1.2 Business-Logik

Natürlich hat die Business-Logik in unserem Trivialbeispiel nicht viel zu bieten. Insofern passt die gesamte Klasse auf wenige Zeilen:

```
@Service
public class BookService {

    @Autowired
    private BookRespository bookRespository;

    public List<Book> listAllBooks() {
        return bookRespository.findAll();
    }

    public Book createNewBook(String title, String isbn) {
        Book book = new Book();
        book.setTitle(title);
        book.setIsbn(isbn);
        bookRespository.save(book);
        return book;
    }

    public Book bookInfo(String title) {
        return bookRespository.findByTitle(title);
    }

    @Transactional
    public void deleteBook(String title) {
        bookRespository.deleteByTitle(title);
    }

    @Transactional
    public Book updateISBN(String title, String newISBN) {
        Book b = bookRespository.findByTitle(title);
        b.setIsbn(newISBN);
        return b;
    }
}
```

Listing 2.5: BookService - Implementierung der Business-Logik

Auch wenn Listing 2.5 trivial scheint, hier lohnt es sich, ein paar Details anzuschauen. Beginnen wir mit `@Autowired`. Spring bedient sich eines Patterns, das als *Inversion of Control* [5] bekannt ist. Kern der Idee ist es nach Möglichkeit nur noch die Interfaces anzugeben, aber die Instanziierung dem Container zu überlassen. Somit verschwindet die direkte Abhängigkeit und man könnte zu

Testzwecken auch eine andere Implementierung einspeisen, sofern sie das Interface implementiert. Man entkoppelt somit die Schichten. `@Service` ist quasi das Gegenteil davon. Damit deklarieren wir unsere Klasse gegenüber Spring als Komponenten deren Instanz Spring verwalten soll und ggf. in ein anderes Objekt (bei uns später) die Boundary-Schicht injizieren soll. Die letzte Annotation über die wir noch sprechen müssen ist `@Transactional`. Wie wir schon sehen ist die ganze Methode etwas "magic" da es so aussieht, als würden wir gar nicht in die Datenbank schreiben. Und hier kommt der Trick - Entities, die wir aus der Datenbank bezogen haben sind aktiv verwaltet. Sobald wir deren Setter aufrufen, werden die Updates direkt in die Datenbank geschrieben. Voraussetzung ist dazu, dass wir uns noch innerhalb einer laufenden Transaktion befinden. Genau das haben wir mit der Annotation ausgedrückt. Hat alles geklappt, wird am Methodenende ein *Commit* aufgerufen, anderenfalls folgt ein *Rollback*. Ausserhalb des Transaktionskontexts werden keine Änderungen meh übernommen.



### 2.1.3 REST-Interface

Damit sind wir bereits beim *Boundary Layer* angelangt und schauen uns das REST-Interface an. Genau wie bei den anderen Layern lesen wir einfach zuerst den Code:

```java
@RestController
@RequestMapping("/api/book")
public class BookControllerREST {

    @Autowired
    private BookService bookService;

    @GetMapping
    public List<Book> getAllBooks() {
        return bookService.listAllBooks();
    }

    @PutMapping("/{title}")
    public Book updateISBN(@PathVariable String title, @RequestBody String isbn) {
        return bookService.updateISBN(title, isbn);
    }

    @DeleteMapping("/{title}")
    public void deleteBook(@PathVariable String title) {
        bookService.deleteBook(title);
    }

    @PostMapping
    public Book createBook(@RequestBody Book book) {
        return bookService.createNewBook(book.getTitle(), book.getIsbn());
    }
}
```

Listing 2.6: BookControllerREST.java

Auch in diesem Beispiel wird alles über Annotation und Konventionen gesteuert. `@RestController` legt fest, dass diese Klasse REST-Antfragen entgegennehmen kann. mit `@RequestMapping` wird ein golbaler Präfix-Pfad festgelegt, die hinteren Pfaddteile werden durch die Methodenannotationen vervollständigt. Zur Demonstration habe ich in diesem Beispiel alle *HTTP-Verben* - also `GET`, `PUT`, `POST` und `DELETE` verdrahtet. Mit `@PathVariable` sehen wir auch schön, wie wir mit dynamischen Pfaden umgehen können.

### 2.1.4 Erste Inbetriebnahme

Sobald wir unsere Spring-Applikation gestartet haben, können wir das Gesamtkonstrukt “testen”. Testen in Anführungszeichen, da wir hier mehr um ein herumprobieren sprechen als über wirkliche Tests im Sinne von Softwaretests. Das machen wir später. Zwar haben wir kein Frontend aber wir können unser REST-Interface auch einfach mit `curl` ansprechen. Das ist für alle Betriebssysteme verfügbar und gehört schlicht zur Standardausstattung, wenn man irgendetwas mit dem HTTP-Protokoll entwickelt. Wer unbedingt möchte, kann auch komplexere Produkte wie *Postman* einsetzen aber für die ersten Gehversuche ist das schlicht nicht nötig.

Also probieren wir unseren ersten Request:

```
$ curl localhost:8080/api/book
[]
```

Wie erwartet ist die Antwort die leere Liste - unsere Datenbank ist ja leer. Einfügen geht aber Problemlos, wir müssen nur einen *POST-Request* absetzen und ein Buch als *JSON Objekt* im Body mitsenden.

```
$ curl \
    -X POST \
    -H "Content-Type: Application/json" \
    -d '{"title":"Hallo, Welt!", "isbn":"123456"}' \
    localhost:8080/api/book

{"id":1,"title":"Hallo, Welt!","isbn":"123456"}
```

Führen wir nun unseren anfänglichen Request nochmals aus, erhalten wir nun das Buch als einziges Element der Liste zurück:

```
$ curl localhost:8080/api/book
[{"id":1,"title":"HalloWelt","isbn":"123456"}]
```

Wenn wir wollen, können wir mit PUT den Titel ändern, auch wenn das so nicht ganz REST-Konform ist:

```
$ curl \
    -X PUT \
    -H "Content-Type: Application/json" \
    -d 'AB12345678' localhost:8080/api/book/HalloWelt
{"id":3,"title":"HalloWelt","isbn":"AB12345678"}
```

Löschen können wir das Buch auch wieder:

```
$ curl -X DELETE localhost:8080/api/book/HalloWelt
```

## 2.2 Bewertung des ECB-Patterns

Eigentlich ist es ja schon erstaunlich, was man in ca. 80 Zeilen Java-Code alles bekommen kann. Positiv ist sicher die Kürze sowie die Verständlichkeit unseres Source-Codes. Die Tatsache, dass wir

fast alles über Konventionen statt Konfiguration gelöst haben befreit den Ansatz von fast allem Boilerplate. Wir haben uns nirgendwo wiederholt (*DRY-Prinzip*) und auch sonst sind wir fast überall rein fachlich unterwegs gewesen. Selbst der Boundary-Layer ist auf ein absolutes Minimum beschränkt und reicht die Anfragen einfach nur gezielt an die Service-Schicht weiter. Also sind wir jetzt fertig und können so in ein grosses Projekt einsteigen? Na ja ... können vielleicht schon aber gut ist das aus Architektursicht eigentlich nicht wirklich.

Kein Wunder - bereits Albert Einstein soll ja *"Everything should be as simple as possible, but not simpler."* gesagt haben. Und genau das ist es. Es ist zu simpel. Mindestens folgende funktionale Probleme haben wir erzeugt:

1. Keinerlei Input-Validierung. Das heisst ein Buch könnte mit fehlender isbn angelegt werden oder schlimmer noch mit fehlendem Titel.

2. Die ISBN kann ein beliebiger String sein und müsste gar nicht dem ISBN-Format entsprechen.

3. Zwei Bücher könnten mit doppeltem Titel angelegt werden, was dann eine eindeutige Abfrage verhindern würde.

4. Wir leaken Informationen - obwohl es fachlich gar keinen Grund gibt, geben wir unsere Datenbank-IDs an die Aussenwelt.

Zudem haben wir auch Architektur-Probleme:

1. Nur in der Entity-Schicht wären saubere Unit-Test möglich aber in unserem Fall gar nicht relevant, da wir keinerlei prozeduralen Code dort erstellt haben.

2. Bereits die Service-Schicht würde (mehr oder weniger) aufwändige Mocks verlangen um isoliert getestet werden zu können. Da wir das Repository mit *@Autowired* injecten müsste sogar der ganze Spring-Kontext laufen. Das ist dann schon eher ein Integrationstest.

3. Unser Datenmodell ist Abhängig vom der JPA. Wir sind auf immer und ewig an eine relationale Datenbank gebunden. Falls wir das umstellen wollten, müssten wir den ganzen Layer austauschen.

4. In diesem Trivialbeispiel nicht wirklich sichtbar aber oftmals muss man sich nach einer bestehenden Tabellenstruktur richten. Das heisst in unser Domänenmodell werden viele Randbedingungen aus der Datenbankwelt Einzug erhalten. Obwohl das auf dieser Ebene gar nicht notwendig wäre, entwickeln wir so unser Datenmodell entlang der Möglichkeiten relationaler Abbildungen. Die Persistenz-Technologie beginnt unser Datenmodell zu beeinflussen.

Zumindest die Validierungsprobleme würden sich mit Spring-Bordmittel adressieren lassen. Wir könnten direkt im REST-Interface oder auch im Service fordern, dass alle Argumente der Methoden nur valide Datenelemente wären. Dazu würde man den Methodenparameter mit `@Valid` annotieren und die ganze Klasse mit `@Validated`. Ebenfalls mit Hilfe von Annotationen würde man das Entity-Modell anreichern. Für Java ist das in der JSR-380 [7] spezifiziert. Aber selbst das ist unschön, da wir noch immer durch Programmierfehler innerhalb unseres Services invalide Objekte erzeugen könnten. Ok, ja gewisse Kriterien kann man auch nochmals auf der Datenbankschicht sicherstellen (z.B. *uniqueness* der ISBN). Vielleicht bringt uns ein ganz anderer Ansatz weiter ...

## 2.3 Immutable Domain Model

Ohne bereits jetzt zu tief in Paradigmen und Patterns einzutauchen wollen wir schauen, was wir gewinnen können, wenn wir alles Technische vergessen und einfach unsere Domäne modellieren. Ohne Abhängigkeiten und ohne Rücksicht auf irgendetwas. Und wir fordern noch etwas - *immutability*. Immutable Objects können nicht verändert werden. Sie entstehen durch Aufruf des Konstruktors und werden (zumindest in Java) durch den Garbage-Collector beseitigt. Das heisst auch dass alle Validierungen, die wir im Konstruktor vornehmen dadurch zu echten *Invarianten* werden, die immer gelten. Jede Instanz eines Domänenobjekts ist zu jedem Zeitpunkt valid. Probieren wir das für unser Bücherverzeichnis.

```
public record Book(
        String title,
        String isbn
) {

    public static final Pattern TITLE_PATTERN = Pattern.compile(
            "^[\\p{L}0-9 .,:;!?()'\"-]+$");
    private static final Pattern ISBN_PATTERN = Pattern.compile(
            "^(97(8|9))?\\d{9}(\\d|X)$");

    public Book {
        if (title == null)
            throw new IllegalArgumentException(
                "A book must have a non-null title");
        title = title.trim();
        if (!TITLE_PATTERN.matcher(title).matches())
            throw new IllegalArgumentException("Title has invalid characters.");
        if (title.length() < 4 || title.length() > 60)
            throw new IllegalArgumentException(
                "A book's title must be between 4 and 60 characters");

        if (isbn == null)
            throw new IllegalArgumentException("A book must have a non-null ISBN");
        isbn = isbn.replaceAll("[-\\s]", "").toUpperCase();
        if (!ISBN_PATTERN.matcher(isbn).matches())
            throw new IllegalArgumentException("ISBN has invalid format: " + isbn);
    }
}
```

Listing 2.7: Book als immutable Domain Object

Anstelle einer Klasse, nutzen wir im Listing 2.7 direkt den Typ `record`. Damit bringen wir zum Ausdruck dass wir hier keine Klasse im herkömmlichen Sinne haben sondern eher ein Value-Object. Sowohl `title` als auch `isbn` sind final, d.h. nur bei der Instanziierung setzbar. `public Book ...` ist der Schluss des Konstruktors. Hier können wir die Datenfelder auch noch beschreiben und

verändern. Und validieren! Sollte etwas nicht konform sein, werfen wir eine Exception und das Objet wird gar nicht erst erzeugt. Bei so einem einfachen Datentyp könnte man das zwar wirklich so machen aber schöner wäre es die Domäne wirklich sauber auszumodellieren. Ein Buch hat eben keinen String mit Namen Titel - es hat einfach einen Titel. Und eine ISBN. Auch im Normalem Sprachgebrauch sind das eigentlich eigene Typen. Zerlegt wird das noch stringenter. An dieser Stelle zeige ich nur die Titel-Klasse. Die Isbn-Klasse sieht analog dazu aus und kann im Sourcecode nachgeschaut werden.

```
public record Title(
        String text
) {

    public static final Pattern PATTERN = Pattern.compile(
        "^[\\p{L}0-9 .,:;!?()'\"-]+$");

    public Title {
        if (text == null)
            throw new IllegalArgumentException("A book must have a non-null title");
        text = text.trim();
        if (!PATTERN.matcher(text).matches())
            throw new DomainValidationException("Title has invalid characters.");
        if (text.length() < 4 || text.length() > 60)
            throw new DomainValidationException(
                    "A book's title must be between 4 and 60 characters");
    }
}
```

Listing 2.8: Ttiel.java

Mehr Kapselung wie in Listing 2.8 ist praktisch nicht zu bekommen. Die Funktionalität der Klasse wurde auf ihren minimalen Bereich verkleinert. Alles, was man zu einem Buchtitel als Typ sagen kann wird hier ausgedrückt. An diesem Punkt habe ich dem Domain-Model auch noch eine einen Exception-Typ `DomainValidationException` hinzugefügt. Dieser erbt von `RuntimeException` und bringt den Unterschied zwischen einem grundsätzlich nichterlaubten Argument (wie z.B. `null`) und einem in dieser Domäne nicht validierbarem Inhalt zum Ausdruck. Aber das ist nicht unbedingt verpflichtend.

```
public record Book(
        Title title,
        Isbn isbn
) {
    public Book {
        if (title == null)
            throw new IllegalArgumentException("A book must have a non-null title");
        if (isbn == null)
            throw new IllegalArgumentException("A book must have a non-null ISBN");
    }
}
```

Listing 2.9: Book.java

Das Buch selbst (Listing 2.9) wird damit noch einfacher. Es verbleiben nur noch die Null-Checks. Das prüft zwar eine technische Gegebenheit, ist aber dennoch fachlich motiviert. Eine Instanz von `Book` ist nur dann gültig, wenn sie auch einen gültigen Titel und eine gültige ISBN besitzt. Jetzt bekommt man schnell das Gefühl, dass dieser Ansatz sehr viele Klassen erzeugt. Das stimmt aber das schadet nicht. Ganz im Gegenteil - eine Klasse ist für genau eine Sache in der Domäne zuständig. Das bringt uns zu einer weiteren "Regel":

**§ 2.2 Die Qualität einer Codebase hängt nicht von der Mange an Klassen ab**

Wir optimieren unsere Codebase nicht nach einer minimalen Anzahl von Dateien sondern nach Ausdrucksstärke, Lesbarkeit und Verständlichkeit.

Mit unserem Ansatz gewinnen wir nicht nur Ausdrucksstärke in unseren Klassen sondern auch der aufrufende Code wird dadurch transparenter:

```
Book book = new Book(
            new Title("Java ist auch eine Insel"),
            new Isbn("978-3-8362-9544-4")
        );
```

Listing 2.10: Codeschnipsel zur Instanziierung eines `Book`-Objekts

Auch ohne in die Implementierung zu schauen wird an diesem Punkt eindeutig klar, was die parameter des Konstruktors bedeuten. Das würde sogar jeder Nicht-programmierer verstehen.

Aber nun kommt der wirkliche Vorteil: Testbarkeit ohne Abhängigkeiten!

### 2.3.1 Domainmodel-Tests

Was wir an diesem Punkt testen wollen, ist völlig klar - natürlich die Validierungskriterien. Das sind einfache Unit-Tests auf der Ebene der einzelnen Typen.

```
class TitleTest {

    @Test
    void rejects_null_title() {
        assertThrows(
                IllegalArgumentException.class,
                () -> new Title(null)
        );
    }

    @Test
    void rejects_too_short_title() {
        assertThrows(
                DomainValidationException.class,
                () -> new Title("DDD")
        );
    }

    @Test
    void rejects_invalid_characters() {
        assertThrows(
                DomainValidationException.class,
                () -> new Title("Clean Code @")
        );
    }
    @Test
    void accepts_invalid_title_with_trim() {
        var title = new Title(" Java ist auch eine Insel ");
        assertEquals("Java ist auch eine Insel", title.text());
    }
}
```

Listing 2.11: Unit-Test für die Title-Klasse

Unser Code hat 4 mögliche Pfade, die wir allesamt testen wollen. Dabei prüfen wir auch, ob die Exceptions vom richtigen Typ sind und in diesem Fall auch, ob das Trimmen des Titels wie gewünscht funktioniert.

**§ 2.3 Alle Domain-Objects haben Unit-Tests**

Für alle Domain-Objects entwickeln wir Unit-Tests, die 100% Path-Coverage für die Validierung im Konstruktor und etwaiger weiterer Funktionalität aufweisen.

## 2.4 Domain Services

Dann gibt es noch einen sehr wichtigen Punkt: Wir wollen nicht nur, dass unser Domänenmodell *immutable* ist - das haben wir bereits erreicht - sondern dass es ausschliesslich aus *pure functions* besteht. Pure ist in diesem Fall Synonym für *Seiteneffektfrei*. Jegliche Funktionen dürfen also nichts am Zustand des Systems verändern. Bei unseren Domain-Klassen könnten sie das gar nicht (da immutable) aber auch generell sonst nicht. Damit ist auch sämtlicher I/O verboten, was z.B. Datenbankzugriffe aber auch selbst Logging einschlisst. Das können wir natürlich nur garantieren, wenn wir auf diesem Layer geschlossen sind und nur unseren eigenen Code benutzen. Aufrufe in andere Packages sind damit untersagt - von Java Standadfunktionalität mal abgesehen.

**§ 2.4 Alle Funktionen im Domänenlayer sind Seiteneffektfrei**

Der Domainlayer ist in sich geschlossen. Es bestehen keinerlei Abhängigkeiten zu anderen (nicht JDK) *Packages*. Alle Funktionen sind frei von Seiteneffekten und machen - damit einhergehend - keinerlei I/O.

Unsere Domain-Records können neben der Validierung noch weitere Funktionen haben. Allerdings sollte man darauf achten, dass sich diese Funktionen nur auf das Domänenobjekt selbst bezieht. Übergeordnete Funktionen sollten in *Services* ausgelagert werden. Machen wir dazu ein Beispiel, auch wenn es in diesem einfachen Kontext vielleicht etwas gesucht klingt. Nehmen wir an, wir möchten fälschlicherweise doppelt erfasste Bücher identifizieren. Treffen wir dazu die fachliche Entscheidung, dass ein Buch dann als doppelt identifiziert wird, wenn es den gleichen Titel aufweist. Man könnte das wie in Listing 2.12 gezeigt implementieren:

```
public final class DeduplicateBooks {
    public static List<Book> execute(List<Book> inputList) {
        return inputList.stream()
            .collect(Collectors.collectingAndThen(
                    Collectors.toMap(
                        Book::title,
                        Function.identity(),
                        (a, b) -> a,
                        LinkedHashMap::new),
                    m -> List.copyOf(m.values())));
    }
}
```

Listing 2.12: Unit-Test für die Title-Klasse

Das mag auf den ersten Blick etwas unkonventionell erscheinen, ist aber modernes Java. Collectors ind von `java.util`. Was wir hier machen ist unsere Liste in eine Map abzubilden, wobei wir den Titel als Schlüssel und das Book-Objekt direkt als Wert (`Functions.identity()`) für den Eintrag verwenden. Bei doppelten Einträgen gewinnt immer der Erste `(a,b)->a`. Am Schluss wird aus den Werten der (nun duplikatfreien Map) wieder eine Liste erzeugt und zurückgegeben. Mit `List.copyOf(...` erzeugen wir uns übrigens wieder eine immutable List so dass unsere Forderung nach immutable Objects auch hier erhalten bleibt.

Und natürlich schreiben wir auch hierfür einen kompakten Unit-Test (Listing 2.13):

```
public class DeduplicateBooksTest {

    @Test
    void execute_removesDuplicatesByTitle() {

        Book book1 = new Book(new Title("Clean Code"), new Isbn("978-0-13-235088-4"));
        Book book2 = new Book(new Title("Effective Java"), new Isbn("978-0-13-468599-1"));
        // Duplicate Entry to 1 with modified ISBN
        Book book3 = new Book(new Title("Clean Code"), new Isbn("978-0-13-235088-5"));

        List<Book> input = List.of(book1, book2, book3);

        List<Book> result = DeduplicateBooks.execute(input);

        assertEquals(2, result.size());

        assertEquals(book1, result.get(0));
        assertEquals(book2, result.get(1));
    }
}
```

Listing 2.13: Unit-Test für den Deduplicate-Service

Bis jetzt ist unser Projekt bei folgender Struktur angelangt:

```
srcPackageRoot/
    domain/
        exception/
            DomainValidationException.java/
        model/
            Book.java/
            Isbn.java/
            Title.java/
        service/
            DeduplicateBooks.java/
testPackageRoot/
    domain/
        model/
            BookTest.java/
            IsbnTest.java/
            TitleTest.java/
        service/
            DeduplicateBooksTest.java/
```

Damit ist die Entwicklung unseres Domain-Layers abgeschlossen.

## 2.5 Von Waben und Zwiebeln: Hexagonal- und Onion- Architektur

Bevor wir weitermachen müssen wir kurz einen Blick auf die Gesamtarchitektur werfen. Schliesslich wollen wir ja wissen, wo uns die Reise hinführen soll. Was wir am Schluss bauen wollen folgt der Idee einer *Hexagonal Architecture* und darin enthalten auch die Idee der *Onion Architecture*.

### 2.5.1 Hexagonal Architecture

Fangen wir bei der hexagonalen Architektur an, dessen Konzept wirklich von der ersten Sekunde an überzuegt - leider vom Namen abgesehen. Weder hat das Ding sechs Seiten noch sechs ecken noch sonst irgendetwas mit Geometrie zu tun. Dem Internet nach gefiel dem Autor die Form weil es an Waben erinnere. So oder so, das Pattern geht auf Alistair Cockburn anfangs der 2000er Jahre zurück, der die *Hexagonal Architecture* in einem gleichnamigen Buch 2024 nochmals beschrieben hat [2]. Im Folgenden werden diese Ideen verkürzt wiedergegeben.

Der zentrale Punkt des Ansatzes ist die Kapselung eine Stücks zusammenhängender Business-Logik. In etwa so, wie wir das gerade eben programmiert haben. Mit Domain-Modell, Domain-Services, Tests - und wie wir gleich sehen werden noch mit Ports und Adaptern. Die Idee ist auch hier, dass das System innerhalb der Kapsel (also z.B. innerhalb des Hexagons) nicht von spezifischer Technologie abhängig ist sondern diese über abstrakte *Ports* einbindet. Das werden in unserem Fall einfach *Interfaces* sein. Implementiert werden diese dann über sogenannte *Adapter*, die von aussen an unser Hexagon “andocken” und die Funktionalität bereitstellen. Cockburn unterscheidet dabei zwischen *driving ports* und *driven ports* [2]. Die einen “treiben” unser Hexagon an (also alle Anfragen von aussen), die Anderen werden von unserem Hexagon aus “angetrieben” (z.B. die Datenbank- Schnittstelle). *Adapters* sind aussen und stellen die technische Umsetzung unserer fachlichen Anforderungen der *driven ports* dar. Oder es sind Komponenten, die uns mit der Aussenwelt verbinden und uns aufrufen. In Adaptern passiert der gesamte I/O. Dinge wie das REST-Interface, die Datenbankanbindung Zugriff auf das Dateisystem, das passiert alles nur ausserhalb in den Adaptern. Die Implementierung der *driving ports* (also die direkte Umsetzung der Use-Cases) passiert innerhlb unseres Hexagons. Um hier keine Doppeldeutigkeit zu erzeugen nennen wir diese nicht wie von Cockburn vorgesehen *Interactors* sondern schlicht *Services*. Cockburn beschreibt noch sogenannte *Configurers* die dann die konkreten Implementierungen an der richtigen Stelle dem System zuführen. Das müssen wir jedoch nicht implementieren, da uns Springboot genau diese Funktionalität bereits zur Verfügung stellt.

Bevor das jetzt zu verwirrend wird - einigen wir uns darauf, dass die Ports, die unser Hexagon treiben nichts anderes als die Sammlung aller *Usecases* sind, die wir mit unserer Entwicklung bedienen wollen. An der Granularität diese Abbildung herrscht allerdings keine Einigkeit. Während Cockburn alle Usecases eines Actors zusammenfassen würde und dann quasi einen Port pro Actor abieten würde, sehen das die Verfechter von *Clean Architecture* oder *Domain Driven Design* entwas enger und würden für jeden einzelnen Usecase einen eigenen Port definieren. Letztlich ist das wohl Geschmacksache - in unserer Umsetzung in Java wäre das letztlich nur die Frage, ob eine oder mehrere Methoden pro Interface deklariert werden. Sonst ändert sich nichts. Der Vorteil ist auch hier Lesbarkeit. Wenn ein Usecase pro Interface definiert ist, und man die Interfaces halbwegs intelligent benennt, dann sieht man bereits der Deklaration der Klasse an, welche Usecases hier bedient werden. Und wieder ist man der Idee, dass der Code die Dokumentation ist ein Stück näher

gekommen. Wir machen das auch für uns jetzt erst mal so.

Noch eine Randbemerkung zu unserem seiteneffektfreien, immutable und rein funktional operierenden Domänenkern. Das kommt übrigens nicht von der Hexagonal Architecture sondern von einem anderen Pattern: *Functional Core and Imperative Shell*.

### 2.5.2 Functional Core und Imperative Shell

Die Idee geht wohl auf einen Vortrag von Gary Bernhardt [1] zurück und wurde später in einem Vortrag von Scott Wlaschin reiteriert [11]. Beide beschreiben die Idee eines funktionalen Kerns der (wie wir selbst gesehen haben für exzellente Testbarkeit sorgt) und nebenbei auch viele Probleme der Threadsicherheit löst. Um damit ein reales System zu bauen wird erst später (ausserhalb dieses Kerns) I/O erlaubt. Das ist 1:1 kompatibel mit dem Ansatz der hexagonalen Architektur. I/O wird dabei sogar ganz nach aussen - also auf ausserhalb des Hxagons verschoben.

### 2.5.3 Onion Architecture

Der letzte Architekturbaustein ist die Onion-Architektur. Da muss ich gar nicht mehr viel dazu sagen, denn diese haben wir bereits vollständig erreicht. Wit haben bereits eine ganz saubere Schichtentrennung und darüber hinaus auch eine klare Abhängigkeitshierarchie. Das Domänen-Modell ist unabhängig von allem, die Domänen-Services nur von Domänen-Modell. Der Applikationsring nur von der Domäne. Und alles aussen nur von der Domäne und den Ports. Es gibt keine einzige Abhängigkeit die in Richtung Aussen zeigen würde. Und genau das macht den Ansatz so leistungsfähig.

Zusammengefasst bekommt unsere Gesamtarchitektur damit das folgende Bild (Abbildung 2.2). Wie wir sehen, steht unser fachlicher Kern im Inneren des Hexagons. Dort sind die eben besprochenen *immutable* Elemente zusammen mit einer technologiefreien Implementierung der Usecases. Auf dieser Ebene müssen die Funktionen nicht mehr Seiteneffekt- aber zumindest technologiefrei sein. Es wird orchestriert aber alles was nicht zum Domänenkern gehört wird über Interfaces abgetrennt. Das sind die *Ports*. Erst die technologieabhängigen Adapter implementieren letztendlich die gesamte I/O-Funktionalität. Somit können die verschiedenen Architekturansätze, die wir angeschaut haben sinnvoll kombiniert werden.

## 2.6 Die Applikationsschicht

Am einfachsten verständlich werden diese Architekturkonzepte, wenn man sie im Code anwendet. Beginnen wir mit der Definition der Usecases. Auch hier ist der Code gleich wieder die Dokumentation. Endlich haben wir damit auch ein Architekturpattern gefunden in dem sich die Usecases wirklich auch im Code widerspiegeln. Somit wird die Dokumentation nicht zur lästigen Pflichtübung sondern ist integraler Bestandteil des Designs, wie uns Listing 2.14 zeigt:

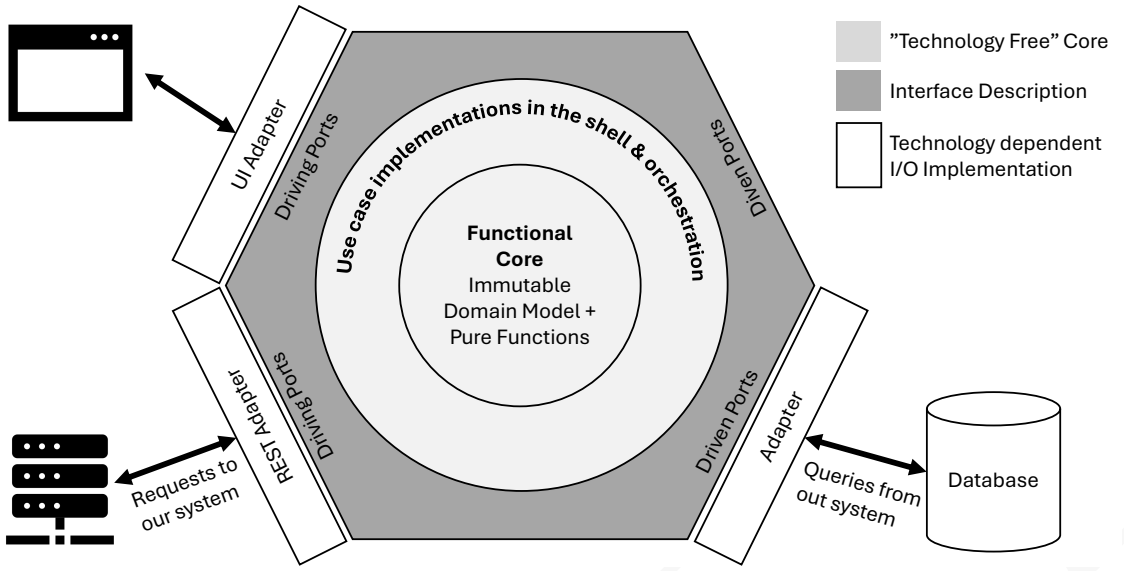



Abbildung 2.2: Kombinierte Darstellung der verschiedenen Architekturansätze

```java
public interface ListAllBooksUseCase {
    List<Book> listAllBooks();
}

public interface LookupBookUseCase {
    Optional<Book> lookupIsbn(Isbn isbn);
}

public interface RegisterBookUseCase {
    void register(Book book);
}
```

Listing 2.14: Die drei UseCases der Applikation

Usecases sind in diesem Fall einfache Interfaces, die selbsterklärende Namen haben sollten. Alleine im Verzeichnisbaum kann man so bereits grob sehen, was die Applikation tut, sprich welche Usecases im Design vorgesehen wurden.

Zu Demonstrationszwecken implementieren wir diese Usecases in zwei Service-Klassen. Man hätte auch alles in eine einzige Klasse packen können aber ich wollte an diesem Punkt die Gestaltungsfreiheit zeigen.

```java
@Service
public class QueryBooksService implements ListAllBooksUseCase, LookupBookUseCase {

    private final BookPersistence findBook;

    public QueryBooksService(BookPersistence findBook) {
        this.findBook = findBook;
    }

    @Override
    public List<Book> listAllBooks() {
        return findBook.findAll();
    }

    @Override
    public Optional<Book> lookupIsbn(Isbn isbn) {
        return findBook.findBy(isbn);
    }

}
```

Listing 2.15: Implementierung der ListAll- und Lookup Usecases

Zugegeben, listing 2.15 sieht leer und redundant aus. Die Anfragen werden einfach durchgereicht. Das liegt aber mehr an der Trivialität unserer Applikation als an der Architektur. Aber ist das nicht grossartig, wie sich die erste Zeile bereits liest? Da steht ja fast in lesbarem Englisch "Die Klasse QueryBooksService implementiert die beiden Usecases ListAllBooks und LookupBook". Mehr müssen wir gar nicht wissen.

Was wir noch nicht besprochen haben ist, wo die BookPersistence auf einmal herkommt. Hier kommen die *driving ports* oder *out ports* ins Spiel. Da wir uns noch innerhalb unseres Hexagons befinden dürfen wir uns noch nicht auf eine spezielle Implementierung festlegen - insofern ist das ebenfalls ein Interface (Listing 2.16).

```java
public interface BookPersistence {
    List<Book> findAll();
    Optional<Book> findBy(Isbn isbn);
    void save(Book book);
}
```

Listing 2.16: BookPersistence outbound Port

Die Zweite Service-Klasse ist übrigens zumindest etwas interessanter, da sie fachliche Logik widerspiegelt (Listing 2.17). An dieser Stelle haben wir festgelegt, dass ein Buch nicht persistiert werden darf, fall ein anderes Buch bereits mit dieser ISBN registriert ist.

```
@Service
public final class RegisterBookService implements RegisterBookUseCase {

    private final BookPersistence bookPersistence;

    public RegisterBookService(
            BookPersistence bookPersistence
    ) {
        this.bookPersistence = bookPersistence;
    }

    @Override
    public void register(Book book) {
        if (bookPersistence.findBy(book.isbn()).isPresent()) {
            throw new IllegalStateException(
                "Book with ISBN " + book.isbn().value() + " already exists"
            );
        }
        bookPersistence.save(book);
    }
}
```

Listing 2.17: Implementierung des RegisterBook Usecase

Man beachte das *@Sercive* über der Klasse. Das wäre eigentlich von der Architektur her verboten, denn es stellt eine technische Abhängigkeit der Klasse zu Spring her. Wenn man das sinnvoll argumentieren kann, dann lohnt sich zuweilen ein gewisser Pragmatismus. So auch in diesem Fall. *@Sercive* ändert nichts am Verhalten der Komponente und könnte bei der Portierung einfach gelöscht werden. Bei den Tests haben wir alleine schon wegen den späteren Integrations- und Systemtests die Abhängigkeiten zu Spring auf Projektebene(!) ohnehin. Insofern ändert sich rein gar nichts und wir ersparen uns dabei das redundante erstellen von den *Configurers*. Spring beherrscht Dependency-Injection bereits in nahezu Perfektion.

### 2.6.1 Applikationsschicht - Tests

Testen ist auch auf der Applikationsschicht notwendig und sinnvoll. Aufgrund unserer sauberen Architektur können wit aber selbst auf dieser Ebene noch vollständig auf UnitTests zurückgreifen. Vor allem haben wir an dieser Stelle beinahe perfekt vorbereitete “Units” - es sind unsere Usecases! Das eignet sich gut, denn das sind ja genau abgegrenzte *units of work*.

Es reicht an dieser Stelle nur einen der Tests zu zeigen, die anderen beiden funktionieren völlig analog zu Listing 2.18.

```
public class RegisterBookUseCaseTest {

    private static BookPersistence bookPersistence;
    private static RegisterBookUseCase usecase;

    @BeforeEach
    void setup() {
        bookPersistence = new InMemoryBookPersistenceAdapter();
        usecase = new RegisterBookService(bookPersistence);
    }

    @Test
    void registers_book_if_not_existing() {
        var book = new Book(
                new Title("Domain-Driven Design"),
                new Isbn("9780321125217")
        );
        usecase.register(book);
        assertTrue(bookPersistence.findBy(book.isbn()).isPresent());
    }

    @Test
    void rejects_book_with_existing_isbn() {
        var book = new Book(
                new Title("Domain-Driven Design"),
                new Isbn("9780321125217")
        );
        usecase.register(book);
        assertThrows(
                IllegalStateException.class,
                () -> usecase.register(book)
        );
    }
}
```

Listing 2.18: RegisterBookUseCase Unit Test

Damit der Test laufen kann benötigen wir ein funktionierendes Repsoitory. Man hätte die echte Datenbank nehmen können - aber dann würde es wieder zu einem Integrationstest. Allerdings ist unsere Persistenzschicht derart einfach, dass man sich einfach eine In-Memory-Implementierung schreiben kann:

```java
public final class InMemoryBookPersistenceAdapter
        implements BookPersistence {

    private final Map<Isbn, Book> store = new HashMap<>();

    @Override
    public Optional<Book> findBy(Isbn isbn) {
        return Optional.ofNullable(store.get(isbn));
    }

    @Override
    public void save(Book book) {
        store.put(book.isbn(), book);
    }

    @Override
    public List<Book> findAll() {
        return store.values().stream().toList();
    }
}
```

Listing 2.19: InMemoryBookPersistenceAdapter

Mehr als in Listing 2.19 braucht es gar nicht, damit wir testen können. Somit sind wir fertig und fügen dem Projekt auf dieser Schicht folgende Files hinzu:

```
srcPackageRoot/
    application/
        port/
            in/
                ListAllBooksUseCase.java/
                LookupBookUseCase.java/
                RegisterBookUseCase.java/
            out/
                BookPersistence.java/
        service/
            QueryBooksService.java/
            RegisterBookService.java/
testPackageRoot/
    adapters/out/memory/
        InMemoryBookPersistenceAdapter.java/
    application/
        ListAllBooksUseCase.java/
        LookupBookUseCase.java/
        RegisterBookUseCase.java/
```

Somit ist unser Applikationskern fertig. Mit 25 Java-Files und 685 Zeilen Code haben wir einen sehr sauberen Applikationskern erstellt, dessen 3 Usecases mit 16 Tests überprüft wurden. Das ist natürlich ein grosser Schritt von den 137 Zeilen des trivialen Codes, der zudem noch die ganze REST- und Persistenzanbindung enthielt. Das ist jedoch genau der Preis, den man in Bezug auf architektonische Excellenz und sauberen Tests bezahlt. Anzahl von Zeilen Code waren eben noch nie ein guter Prädiktor für die Qualität oder der Verständlichkeit einer Software. Aber nun wollen wir den Rest der Applikation noch vervollständigen.

## 2.7 Adapters

Im Wesentlichen benötigen wir zwei Adapter, die an unsere Ports andocken. Einmal den REST-Adapter, der die Schnittstelle nach aussen exponiert und einemal unseren Pesistenzadapter, der die Anbindung an die Datenbank implementiert.

### 2.7.1 Persistence Adapter mit JPA

Der Auftrag ist klar - wir müssen lediglich das Interface BookPersitance aus Listing 2.16 implementieren. Das werden wir ziemlich Analog zu dem Code aus dem trivialen Beispiel umsetzen.

```
@Entity
@Data
@AllArgsConstructor
public class BookEntity {
    @Id
    private String isbn;
    private String title;
}
```

Listing 2.20: BookEntity mit voller JPA und Lombok-Annotation

In Listing 2.20 wird die ISBN direkt als Primärschlüssel verwendet. Ob das performancetechnisch eine gute Idee ist, kommt darauf an, ob es noch andere Relationen gibt, die auf diesen Primärschlüssel verweisen. Anderenfalls sollte man doch auf eine nummerische ID ausweichen.

Um effektiv darauf zugreifen zu können, bneötigen wir noch das passende Repository (Listing 2.21) was in diesem Fall leer ist, da wir mit der Standard-Funktionalität auskommen.

```
public interface BookRepository extends
    JpaRepository<BookEntity, String> {

}
```

Listing 2.21: Leeres BookRepository nur mit JPARepository Standardfunktionen

Somit haben wir bereits alle Teile beisammen um den Adapter zu finalisieren. Es fehlt nur noch seine eigentlich Implementierung, wie in Listing 2.22 gezeigt:

```
@Service
public class JpaBookPersistenceAdapter
        implements BookPersistence {

    private final BookRepository bookRepository;

    public JpaBookPersistenceAdapter(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    @Override
    public Optional<Book> findBy(Isbn isbn) {
        return bookRepository.findById(isbn.value())
                .map(this::toDomain);
    }

    @Override
    public List<Book> findAll() {
        return bookRepository.findAll()
            .stream()
            .map(this::toDomain)
            .toList();
    }

    @Override
    public void save(Book book) {
        bookRepository.save(fromDomain(book));
    }

    private Book toDomain(BookEntity entity) {
        return new Book(
                new Title(entity.getTitle()),
                new Isbn(entity.getIsbn())
        );
    }

    private BookEntity fromDomain(Book book) {
        return new BookEntity(
                book.isbn().value(),
                book.title().text()
        );
    }
}
```

Listing 2.22: Der JPA Persistenz-Adapter

Nichts davon solle wirklich überraschend sein ausser die beiden privaten Funktionen `fromDomain` und `toDomain`. Da unsere Entity-Klasse sehr einfach aufgebaut ist und lediglich Strings benutzt müssen wir zwischen diesen beiden Repräsentationen umwandeln. Hier sei angemerkt, dass die Umwandlung `toDomain` bereits die volle Validierung vornimmt. Wir erinnern uns - ein invalides Domänenobjekt ist nicht intanziierbar. In gegenrichtung können wir uns darauf verlassen, dass wir keine invaliden Objekte an die Datenbank liefern, da bereits das Interface an die Persistenz nur über Domänenobjekte definiert wurde. Somit können wir an dieser Stelle auf weitere Checks verzichten und sind fertig.

Noch ein letzter Hinweis: Persistence ist nur ein Beispiel einer Service-übergreifender I/O Funktionalität. Logging würde man identisch mittels eines Ports und eines Adapters implementieren.



### 2.7.2 REST Adapter mit Springboot-web

Um das Ganze an die Aussenwelt anzubinden müssen wir eine Komponente (einen *Adapter*) schreiben, der unsere *driven ports* entsprechend ansteuert.

```
@RestController
@RequestMapping("/api/books")
public class BookController {

    private final RegisterBookUseCase registerBookUseCase;
    private final ListAllBooksUseCase listAllBooksUseCase;
    private final LookupBookUseCase lookupBookUseCase;

    BookController(
            RegisterBookUseCase registerBook,
            ListAllBooksUseCase listAllBooksUseCase,
            LookupBookUseCase lookupBookUseCase) {
        this.registerBookUseCase = registerBook;
        this.listAllBooksUseCase = listAllBooksUseCase;
        this.lookupBookUseCase = lookupBookUseCase;
    }
    ...
    @GetMapping("/{isbn}")
    ResponseEntity<BookDto> getBook(@PathVariable String isbn) {
        return ResponseEntity.of(
                lookupBookUseCase.lookupIsbn(new Isbn(isbn))
                        .map(this::fromDomain));
    }
    ...
    private BookDto fromDomain(Book book) {
        return new BookDto(
                book.title().text(),
                book.isbn().value());
        }
}
```

Listing 2.23: REST-Adapter implementiert in BookController.java

Auch in Listing 2.23 ist die Verwendung der Usecases erneut sehr schön zu sehen und im Wortlaut zu verstehen. Dieser Controller steuert die Usecases *RegisterBook*, *ListAllBooks* und *LookupBook* an. Perfekt - wir wissen sofort um was es geht. An dieser Aussenschnittstelle endet auch unsere Domäne. Es wäre ein Zufall, wenn unsere Aussenwelt die Formate unsere Domäne so 1:1 verwenden könnte. Daher setzen wir *Data-Transfer-Objects* - oder kurz *DTOs* ein um den Datenaustausch nach aussen zu beschreiben. Da unser Beispiel inhaltlich trivial ist, gibt es hier wirklich ein 1:1-Mapping, ausser bei den Datentypen. Wir bekommen Strings und lifern ein BookDTO (Listing 2.24) als record mit Strings für die Felder. Bei komplexeren Anforderungen ist es auch üblich nach RequestDTOs und ResponseDTOs zu unterscheiden. So wie es halt von den Anforderungen her nötig ist. Das Mapping mittels `fromDomain` und `toDomain` kennen wir bereits aus dem Persistenz-Adapter.

```
public record BookDto (
        String title,
        String isbn
) {
}
```

Listing 2.24: BookDTO.java

Zudem legen wir noch eine Konfigurationsklasse an, damit Spring die Exceptions aus unserem Domain-Model richtig behandelt. In diesem Fall wollen wir nicht `500 Internal Server Error` zurückgeben sondern lieber `400 Bad Request` und den Text der Exception als Content zurückliefern. Ob man so transparent sein möchte ist vor allem eine Securityentscheidung. Wir machen das einfach mal.

```
@ControllerAdvice
public class RestExceptionHandler {

    @ExceptionHandler({
        IllegalArgumentException.class,
        DomainValidationException.class})
    public ResponseEntity<String> handleBadRequestExceptions(RuntimeException ex) {
        return ResponseEntity
                .status(HttpStatus.BAD_REQUEST)
                .body(ex.getMessage());
    }
}
```

Listing 2.25: Konfiguration um Exceptions richtig zu behandeln

### 2.7.3 Adaptertests als Integrationstests

Damit bleibt nur noch unsere Adapter zu testen. Ob man an dieser Stelle Unittest benötigt hängt von der Komplexität der Adapter ab. Für unsere Zwecke genügen an dieser Stelle Integrationstests, da wir eigentlich nur fachlich langweiliges I/O in diesen Adaptern realisieren. Den Applikationskern und die Domäne haben wir ja bereits ausführlich getestet.

```
@SpringBootTest
@Transactional
public class JpaBookPersistenceAdapterTest {

    @Autowired
    JpaBookPersistenceAdapter adapter;

    @Test
    void persists_and_loads_book() {
        Book book = new Book(
                new Title("Effective Java"),
                new Isbn("9780134685991")
        );

        adapter.save(book);
        assertTrue(adapter.findBy(book.isbn()).isPresent());
    }
}
```

Listing 2.26: Integrationtest des JpaBookPersistenceAdapter



Listing 2.26 zeigt wie einfach wir einen Integrationstest bewerkstelligen können. Mit `@SpringBootTest` und `@Transactional` starten wir den Springboot-Context und sorgen dafür dass unsere Testdaten nicht final in die Datenbank geschrieben werden. Es erfolgt ein *Rollback* am Ende der Testmethode. Das wäre bei unserer In-Memory-H2-Datenbank zwar unwichtig aber es ist am saubersten das gleich so zu spezifizieren.

Final testen wir nun noch unseren REST-Adapter (Listing **??**).

```
@SpringBootTest
@AutoConfigureMockMvc
@ActiveProfiles("inmemorytest")
@Transactional
class BookControllerTest {

    @Autowired
    MockMvc mockMvc;

    @Autowired
    BookRepository bookRepository;

    @Test
    void registerAndLoadBook() throws Exception {
        mockMvc.perform(post("/api/books")
                .contentType(MediaType.APPLICATION_JSON)
                .content("""
                            {
                              "title": "Effective Java",
                              "isbn": "9780134685991"
                            }
                        """))
                .andExpect(status().isOk());

        mockMvc.perform(get("/api/books"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.length()").value(1));

        mockMvc.perform(get("/api/books/9780134685991"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.title")
                        .value("Effective Java"));
    }

    @Test
    void testExceptionHandling() throws Exception {
        mockMvc.perform(post("/api/books")
            .contentType(MediaType.APPLICATION_JSON)
            .content("""
                        {
                            "title": "",
                            "isbn": "9780134685991"
                        }
                    """))
            .andExpect(status().is4xxClientError());
    }
}
```

Listing 2.27: Integrationstest des REST Controllers

Damit ist aus Sicht der Entwicklungs- und Testarbeit am Backend alles getan. In der Aussenschicht um unser Hexagon herum haben wir folgende Dateien dem Projekt hinzugefügt:

```
srcPackageRoot/
  adapters/
    in/
      rest/
        BookController.java/
        BookDto.java/
        RestExceptionHandler.java/
    out/
      persistance/
        BookEntity.java/
        BookRepository.java/
        JpaBookPersistenceAdapter.java/
testPackageRoot/
  adapters/
    in/
      rest/
        BookControllerTest.java/
    out/
      persistence/
        JpaBookPersistenceAdapterTest.java/
```

Damit ist das Backend final fertig. Mit 808 Zeilen Code und total 19 Tests haben wir eine saubere, zukunftssichere sehr verständliche Architektur umgesetzt.

## 2.8 Diskussion des Ansatzes

Natürlich muss man das immer relativieren. Der Ansatz ist für sehr kleine Projekte sicher ein "overkill" und selbst bei grösseren Unterfangen nicht immer notwendig. Wenn man sich sehr sicher ist, dass der verwendete Technologiestack sich nicht mehr ändert (also dass man z.B. von Springboot und einer relationalen Datenbank nicht abweicht) kann man auch bewusst eine Abhängigkeit in Kauf nehmen. Ebenso, was den funktionalen Kern angeht. Es ist eine (zugegeben extrem saubere) Möglichkeit eine Domäne in Code abzubilden. Aber wir wissen ja aus unzähligen Projekten davor, dass es noch viele weitere gute Möglichkeiten gibt dies zu tun. Die Lösung, die wir hier angeschaut haben bringt klar mehr Komplexität. Das ist erst mal schlecht - es kommt jedoch darauf an, welche Form von Komplexität man anhäuft (*incidental* vs. *accidental complexity*). Gegen *incidental complexity* kann man nicht viel machen - die passiert einfach, oder liegt einfach vor. Wenn die modellierte Domäne komplex ist, wird das automatisch auch komplexeren Code nach sich ziehen (oder zumindest mehr Code). Wer jetzt denkt, dass das Domänenmodell in unserem naiven Code den Sachverhalt viel weniger Komplex abbilden konnte, der irrt insofern, als dass das Modell es verpasst hat die Komplexität überhaupt abzubilden. Im Endeffekt ist das noch schlimmer - erforderliche Komplexität die in Code und Tests ignoriert wird. Das ist natürlich ein Problem denn so entsteht fehlerhafte Software. Durch fehlende Validierung und ein Modell, dass inkonsistente und falsche Repräsentationen erlaubt entstehen auch schnell echte Sicherheitsprobleme. So oder so, die Domäne muss schon adäquat abgebildet werden. Anders sieht es bei der Zwiebel-, hexagonalen oder

Ports und Adapter- Architektur aus. Je nach Betrachtungsweise ist diese Zerlegung und Abstraktion genau nicht von der Domäne vorgegeben und motiviert. Das ist allein unsere Entscheidung, die wir aus Gründen der Testbarkeit, wiederverwendbarkeit, Flexibilität der Technologie oder einfach nur zur Aufteilung der Bereiche in in grossen Teams treffen. Passiert das unüberlegt, dann baut man *accidential complexity* auf. Unnötig Komplexität anzuhäufen ist so wie sonst im Leben Risiken einzugehen ohne einen Ertrag zu erwarten - Eine dumme Entscheidung!

Um dafür ein besseres Gespür zu erhalten wollen wir eine deutlich komplexere Domäne Modellieren und ein ganzes System mit Frontend sowie technischer Anbindung erstellen um diesen Entscheidungsraum etwas besser zu erforschen.

# Kapitel 3

# Realistischeres Beispiel: PLM für die Elektronikentwicklung

Nun haben wir das Handwerkszeug, um uns einer realistischeren Domäne zu widmen. Für diesen Kurs haben wir uns etwas ausgesucht, das auf den ersten Blick sehr einfach aussieht, aber bei genauerer Betrachtung viele interessante Details zum Vorschein bringt. Vermutlich ist das bei den meisten nicht-trivialen Domänen nicht anders.

Wir wollen ein System bauen, um elektronische Bauteile zu verwalten, katalogisieren und bestellen zu können. Später werden wir das Ganze sogar in Design-Werkzeuge (EDA - *Electronic Design Automation*) integrieren und ein Web-Frontend dazu anbieten. Unsere Daten sollen dabei in einer relationalen Datenbank gespeichert werden. Eine klassische Business-Applikation also.

Die Sachlage ist schnell erklärt. Entwickler elektronischer Baugruppen planen in ihren Designs elektronische Bauteile ein. Wir wollen diese Komponenten nennen. Das könnte z.B. ein Widerstand mit einem bestimmten Wert in einem bestimmten Gehäuse sein. Ganz konkret etwa ein 10k-Ohm-Widerstand in einem 0603-SMD-Package. Auf dieser Ebene ist das erst mal komplett abstrakt - hunderte Hersteller würden einem dieses Bauteil liefern können und man könnte diese über zahlreiche *Distributoren* bestellen. Vielfach ist der konkrete Hersteller auch irrelevant - so lange das Bauteil den Spezifikationen genügt. Viele Betriebe führen daher eigene IPNs (*Internal Part Number*), um Bauteile dieser Art zu gruppieren. Welches spezifische bestellbare Teil dann durch das Procurement beschafft wird ist dem Entwickler egal. Und genau das soll unser System abbilden. Es soll den Zusammenhang zwischen abstrakt/intern verwendeten Komponenten und tatsächlich bestellbaren Bauteilen herstellen und deren Metadaten verwalten. Am Schluss wollen wir dem System eine BOM (*Bill of Material*) übergeben und daraus automatisch Bestellinformationen erzeugen können. Damit das halbwegs übersichtlich bleibt, sollen die Bauteile in Kategorien zusammengefasst werden. Jede Komponente gehört genau einer Kategorie an, und jede Komponente hat ein oder mehrere bestellbare Bauteile. Für alle Bauteile geltende Spezifikationen (vorerst *Attribute* genannt) auf Ebene der Komponente, falls alle bestellbaren Bauteile dies gleichermassen erfüllen. Anderenfalls gibt es bauteilspezifische Attribute auf ebene der konkret bestellbaren Teile. Das klingt eigentlich übersichtlich und einfach, die meisten von euch werden bereits nach diesen paar Zeilen eine recht klare Vorstellung einer passenden Objektstruktur im Kopf haben. Aber lasst uns das wieder sauber beginnen mit der Modellierung unserer Domäne.

## 3.1 Funktionales Domänenmodell - Daten

Genau wie bei unserem Buch beginnen wir uns mit den Elementen unserer Domäne auseinanderzusetzen. Bevor wir uns fragen, was eine “Komponente” im abstrakten Sinne sein soll, können wir beginnen das zu modellieren, was wir am einfachsten beschreiben können. Das wäre in unserer Domäne wohl das tatsächlich bestellbare Bauteil.

Wenn wir wieder fordern wollen, dass unsere Domänenelemente immutable sind, dann müssen wir unsere Felder wieder in einzelne Datentypen zerlegen. In unserem Objekt müssen wir dann wieder nur noch prüfen, dass diese nicht null sind. So sähe unser Record aus:

```
public record OrderableItem (
    ManufacturerPartNumber mpn,
    StockKeepingUnit sku,
    Distributor distributor,
    Quantity minOrderQuantity,
    Multiplier orderMultiple,
    UnitPrice price,
    List<Attribute> attributes
) {
}
```

Listing 3.1: OrderableItem als Record-Typ

Und es müssen auch nicht alles einzelne Werte sein, unsere Typen können ihrerseits durchaus auch komposit sein. Unser `Price`-Typ z.B. Neben dem nummerischen Wert des Preises gehört auch die Angabe einer Währung fest zu diesem Typ. Eine blosse zahl macht hier keinen Sinn, da man elektronische Teile oft in ausländischen Währungen bezahlt. Je nach Anforderungen würde man noch viel mehr Informationen zum Preis einfordern - z.B. ob mit dieser mit oder ohne Mehrwertsteuer angegeben wurde etc. Der Distributor bräuchte dann evtl. noch Informationen zum Steuersatz, Verzollung, Versandkostenpauschalen, etc. Das ist noch beliebig ausbaubar. Aber lassen wir das aktuell noch bewusst simpel. Da wir uns ja nun darauf verlassen können, dass alle FeldTypen immer nur valide Ausprägungen beinhalten, müssen wir nur noch dafür sorgen, dass uns nicht jemand einen Null-Wert setzt. Neben den Null-Checks gibt es noch eine fachliche Überprüfung. Es kann nicht sein, dass das Order-Multiple grösser als die Mindestbestellmenge ist. Das wäre jetzt eine Konsistenzprüfung zwischen den Werten. Dann gibt es am Schluss noch das “Problem” mit den Listen-Typen ( Gleiches gilt auch für Sets und Maps etc.) - so wie deklariert, wäre nur die Referenz auf die Liste immutable, aber noch nicht ihr Inhalt. Das darf natürlich nicht sein und wir erzeugen uns daher auf jeden Fall eine Kopie der bestehenden Liste, die uns übergeben wurde, damit nicht etwa der Aufrufende noch eine Referenz darauf halten kann, um die Liste im Nachhinein (also nach Verlassen unseres Konstruktors) nochzu manipulieren. In diesem Fall tolerieren wir sogar einen Null-Wert und ersetzen diesen sofort durch eine leere Liste. Die `copyOf`-Methode erzeugt uns übrigens eine read-only-Liste. Da nun sowohl die Referenz als auch die Liste als auch die Elemente (weil es alles Domänen-Elemente sind) dieser Liste selbst immutable sind, ist der gesamte Objektbaum immutable.

```
public record OrderableItem(
        ManufacturerPartNumber mpn,
        StockKeepingUnit sku,
        Distributor distributor,
        Quantity minOrderQuantity,
        Multiplier orderMultiple,
        UnitPrice price,
        List<Attribute> attributes) {

    public OrderableItem {
        if (mpn == null)
            throw new IllegalArgumentException(
                    "An OrderableItem must have a valid MPN");
        if (sku == null)
            throw new IllegalArgumentException(
                    "An OrderableItem must have a valid SKU");
        if (distributor == null)
            throw new IllegalArgumentException(
                    "An OrderableItem must have a valid Distributor");
        if (minOrderQuantity == null)
            throw new IllegalArgumentException(
                    "An OrderableItem must have a valid minOrderQuantity");
        if (orderMultiple == null)
            throw new IllegalArgumentException(
                    "An OrderableItem must have a valid orderMultiple");
        if (orderMultiple.value() > minOrderQuantity.value())
            throw new IllegalArgumentException(
                    "minOrder Quantity cannot be smaller than orderMultiple");
        if (price == null)
            throw new IllegalArgumentException(
                    "An OrderableItem must have a valid Price");
        if (attributes == null)
            attributes = List.of();

        attributes = List.copyOf(attributes);
    }
}
```

Listing 3.2: Vollständig validierter OrderItem Record

`Enums` sind übrigends auch eine hervorragende Möglichkeit, das Modell auf gültige Werte zu beschränken. Nehmen wir z.B. unsere `Currency`. Im einfachsten Fall ist das ebenfalls ein simples `enum`:

```
public enum Currency {
    CHF, EUR, USD
}
```

Listing 3.3: Currency als simple Enumeration

Somit können wir nun auch unseren UnitPrice-Record darstellen:

```
public record UnitPrice(
    BigDecimal value,
    Currency currency
) {
    public static final BigDecimal MAX_PRICE = new BigDecimal(1000);

    public UnitPrice {
        if(value == null || value.signum() < 0) {
            throw new IllegalArgumentException(
                "A Price must have a positive value");
        }

        // We always want to have 4 decimal places in pricing information
        value = value.setScale(4, RoundingMode.HALF_UP);

        if(value.compareTo(new BigDecimal("0.0001")) < 0) {
            throw new IllegalArgumentException(
                "A single item price must be >= 0.0001 in any currency");
        }

        if(value.compareTo(MAX_PRICE) > 0) {
            throw new IllegalArgumentException(
                "A single item price must be <= " + MAX_PRICE + " in any currency");
        }
        if(currency == null) {
            throw new IllegalArgumentException(
                "A Price must specify its currency");
        }

    }
}
```

Listing 3.4: Price-Record mit Validation

Listing 3.4 zeigt den Price-Record mit einer Validierung des Wertebereichs und der Sicherstellung der geforderten Genauigkeit der Preisangabe. 4 Nachkommastellen sind nötig, da für kleine Bauteile (z.B. Widerstände) die Stückpreise oft weit unterhalb von einem Rappen liegen. Zu guter Letzt schauen wir uns noch die Validierung der ManufacturerPartNumber an.

```
public record ManufacturerPartNumber(
    String value
) {
    public ManufacturerPartNumber {
        if(value == null)
            throw new IllegalArgumentException(
                "A MPN must not have a null value");
        if(!value.matches("^[0-9a-zA-Z\\-_.:, ]+$"))
            throw new IllegalArgumentException(
                "MPN must match this regexp: \"^[0-9a-zA-Z\\-_.:, ]+$\"");
        value = value.trim();
    }
}
```

Listing 3.5: ManufacturerPartNumber mit Validation

Nun wollen wir noch die Attribute in den Griff bekommen. Diese sind gleich in mehrerer Hinsicht interessant. Erstens müssen sie widerspruchsfrei sein, d.h. jedes Label darf in jeder Liste auf Ebene der `OrderableItem` und auch der `Componnet` nur einmal vorkommen. Alleine daran sehen wir, dass eine `java.util.Map` die bessere Datenstruktur scheint als eine Liste - wenn wir `AttributeLabel` als `key` benutzen und `AttributeValue` als `value` der Map, sind doppelte Einträge innerhalb einer Map nicht möglich. Lediglich zwischen den Maps müssen wir noch für Konsistenz sorgen. Wir können sogar noch präziser sein: Ein bestimmtes Label ist auf Ebene der `Component` gesetzt und gilt damit für alle `OrderableItem` - nun darf es in den Maps der Items nicht mehr vorkommen.

Aber der Reihe nach. Es geht ja darum, physikalische Grössen in Form einer Spezifikation in diesen Attributen abzubilden - z.B. eine maximal zulässige Spannung oder Temperatur, die das Bauteil aushalten muss. Es geht also um Messgrössen. Dabei wollen wir auch gleich hinterlegen, ob diese Grössen negativ werden können. Listing 3.6 zeigt die Umsetzung mit einem `enum`:

```
public enum MeasuredQuantity {

    CURRENT(true),
    VOLTAGE(true),
    TEMPERATURE(true),
    POWER(false),
    RESISTANCE(false),
    CAPACITANCE(false),
    INDUCTANCE(false),
    FREQUENCY(false),
    TIME(false),
    LENGTH(false),
    DIMENSIONLESS(true);

    private final boolean allowNegative;

    Quantity(boolean allowNegative) {
        this.allowNegative = allowNegative;
    }

    public void validate(BigDecimal valueInBaseUnit) {
        if (!allowNegative && valueInBaseUnit.signum() < 0) {
            throw new IllegalArgumentException(
                    name() + " must not be negative: " + valueInBaseUnit
            );
        }
    }
}
```

Listing 3.6: Quantity enum inkl. der Abbildung von Zahlenranges

Hier zeigt sich auch nochmals, wie leistungsfähig `enums` in der Praxis sein können. Neben den Messgrössen benötigen wir noch Einheiten. Auch das ist nochmals ein `enum`. In diesem Fall sogar ein noch ausgefeilteres Vorgehen, da wir auch gleich noch Informationen zur Umrechnung beifügen wollen. 1000mA sind z.B. 1A und 0°C entsprechen 273.15K. Jede Einheit wird dabei einer Messgrösse zugeordnet.

```
public enum Unit {

    VOLT(MeasuredQuantity.VOLTAGE, "V", BigDecimal.ONE, BigDecimal.ZERO),
    MILLIVOLT(MeasuredQuantity.VOLTAGE, "mV", new BigDecimal("0.001"), BigDecimal.ZERO),
    KILOVOLT(MeasuredQuantity.VOLTAGE, "kV", new BigDecimal("1000"), BigDecimal.ZERO),

    ...

    KELVIN(MeasuredQuantity.TEMPERATURE, "K", BigDecimal.ONE, BigDecimal.ZERO),
    CELSIUS(MeasuredQuantity.TEMPERATURE, "°C", BigDecimal.ONE, new BigDecimal("273.15"));

    private final MeasuredQuantity quantity;
    private final String symbol;
    private final BigDecimal scaleToBase;
    private final BigDecimal offsetToBase;

    Unit(MeasuredQuantity quantity, String symbol,
        BigDecimal scaleToBase, BigDecimal offsetToBase) {
            this.quantity = quantity;
            this.symbol = symbol;
            this.scaleToBase = scaleToBase;
            this.offsetToBase = offsetToBase;
    }

    public MeasuredQuantity quantity() {
        return quantity;
    }

    public String symbol() {
        return symbol;
    }

    public BigDecimal toBase(BigDecimal value) {
        return value.multiply(scaleToBase).add(offsetToBase);
    }

    public BigDecimal fromBase(BigDecimal baseValue) {
        return baseValue.subtract(offsetToBase).divide(scaleToBase);
    }
}
```

Listing 3.7: Gekürzt dargestelltes Unit-enum inkl. Umrechnung in Basiseinheiten

So einen rigiden Ansatz kann man natürlich nur machen, wenn die Elemente der Domäne absolut stabil bleiben (oder es hinnehmbar ist mit einem Update bis zum nächsten Release zu

warten). Bei uns ist das bis hierhin gegeben - an der Physik ändert sich in diesem Bereich nicht mehr viel.

Was jetzt noch fehlt sind unsere Labels. Nennen wir sie mal `SpecifiedProperty`. Diesen Properties ordnen wir direkt unsere `Quantities` zu, wie Listing 3.9 zeigt:

```
public enum SpecifiedProperty {

    OPERATING_TEMPERATURE_MIN(MeasuredQuantity.TEMPERATURE),
    OPERATING_TEMPERATURE_MAX(MeasuredQuantity.TEMPERATURE),

    SUPPLY_VOLTAGE_MIN(MeasuredQuantity.VOLTAGE),
    SUPPLY_VOLTAGE_MAX(MeasuredQuantity.VOLTAGE),

    POWER_DISSIPATION_MAX(MeasuredQuantity.POWER),

    CLOCK_FREQUENCY_MAX(MeasuredQuantity.FREQUENCY);

    ...

    private final MeasuredQuantity quantity;

    ComponentAttribute(Quantity quantity) {
        this.quantity = quantity;
    }

    public Quantity quantity() {
        return quantity;
    }
}
```

Listing 3.8: Gekürzt dargestelltes SpecifiedProperties-enum

Hier könnte man wirklich diskutieren, ob man das nicht lieber mit Records abbildet um dann flexibler zu sein. Mit diesem Ansatz wäre hingegen eine parametrische Suche (wie etwa “gib mir alle Bauteil mit jeder eigenschaft in jenem Bereich”) einfacher umzusetzen.

Nun können wir alles zusammenfügen und endlich unsere Attribute verwalten. Da es mehr als nur irgendwelche Attribute sind, wollen wir einen Typ `SpecificationItem` erstellen. Das implementieren wir mit einem record, der die Eigenschaft (samt Einheit), den eigentlichen Wert sowie die Messgrösse speichert. An diesem Punkt können (und müssen) wir wieder vollständig validieren. Dieses Tripplet darf weder null-Elemente beinhalten noch darf die Messgrösse mit der Eigenschaft inkompatibel sein und ausserdem muss der Wertebereich zur Messgrösse passen.

```
public record SpecificationItem(
        SpecifiedProperty property,
        BigDecimal value,
        Unit unit) {
    public SpecificationItem {
        if (property == null)
            throw new IllegalArgumentException(
                    "A SpecificationItem must have a property");
        if (value == null)
            throw new IllegalArgumentException(
                    "A SpecificationItem must have a value");
        if (unit == null)
            throw new IllegalArgumentException(
                    "A SpecificationItem must have a unit");
        if (unit.quantity() != property.quantity()) {
            throw new IllegalArgumentException(
                    "Unit " + unit + " does not match Quantity " + property.quantity());
        }
        BigDecimal baseValue = unit.toBase(value);
        property.quantity().validate(baseValue);
    }

    public BigDecimal valueInBaseUnit() {
        return unit.toBase(value);
    }
}
```

Listing 3.9: Implementierung des SepcificationItem-Record inkl. Validierung

Die Früchte unserer Arbeit erkennen wir spätestens, wenn wir eine Komponenten von Hand instanziieren wollen.

```
Component c = new Component(
    new InternalPartNumber("SMD_R_0603_10k"),
    Category.RESISTORS,
    List.of(
        new OrderableItem(
            new ManufacturerPartNumber("RC0603FR-0710KL"),
            new StockKeepingUnit("2421850"),
            new Distributor("Farnell"),
            new Quantity(10),
            new Multiplier(10),
            new UnitPrice(
                new BigDecimal("0.0058"),
                Currency.CHF),
            Map.of(
                SpecifiedProperty.OPERATING_TEMPERATURE_MAX,
                new SpecificationItem(
                    SpecifiedProperty.OPERATING_TEMPERATURE_MAX,
                    new BigDecimal(155),
                    Unit.CELSIUS),
                SpecifiedProperty.OPERATING_TEMPERATURE_MIN,
                new SpecificationItem(
                    SpecifiedProperty.OPERATING_TEMPERATURE_MIN,
                    new BigDecimal(-55),
                    Unit.CELSIUS)))),
    Map.of());
```

Listing 3.10: Manuelle Instanziierung einer Komponente

Auch ohne, das wir in die Implementierung schauen müssen ist, dieses Stück Code in sich lesbar und verständlich. Und wir können uns blind darauf verlassen, dass wir keine strukturellen Fehler gemacht haben, denn sonst wird das Objekt erst gar nicht erzeugt. Wir wissen, dass alles vorhanden und entlang der spezifizierten Regeln in sich konsistent ist.

## 3.2 Funktionales Domänenmodell - Unit testing

Beim Testen haben wir nun gleich zwei Vorteile. Ersten werden wir weniger Test brauchen, da der Aufgabenbereich des Domänen-Modells viel kleiner geworden ist. Wir können weder grossartig Konsistenzprobleme erzeugen noch kämen wir mit der Welt der Datenbanken oder anderer Schnittstellen in Berührung. Was die Validierung angeht könnten wir für alles und jedes einen Unit-Test schreiben aber viel bringen würde das nicht. Was nutzt es die Funktionalität einer if-then-throw Anweisung zu prüfen? Dass man wirklich alle Elemente des Records auf `null` prüft kann auch mittels Inspektion geprüft werden. Wenn man da etwas übersieht, würde es einem wahrscheinlich auch beim Schreiben des Tests nicht auffallen - insbesondere wenn die gleiche Person den Code und den Test schreibt. Das muss man selbst entscheiden, und ist auch teils Geschmacksache,

TDD-Puristen würden das sicher anders sehen. Damit niemand zu kurz kommt habe ich versucht möglichst vollständige Tests zu erzeugen.

Bei komplexeren Validierungen scheinen Test hingegen wertvoller. Bei Grössenvergleichen vertut man sich schnell mal mit der Reihenfolge im `compareTo` oder wundert sich, dass ein `BigDecimal("10.00")` nicht `equalTo BigDecimal("10")` ist aber `compareTo()` dennoch eine `0` liefert. Oder auch bei den String-Validierungen kann man sich schnell mit den Regexp irren. Das sollten wir daher auf jeden Fall testen. Da wir keinerlei Abhängigkeiten zu irgendwelchen Elementen ausserhalb unseres Domänenmodells haben und nur Standard-Datentypen benutzen ist praktisch nichts an Testinfrastruktur notwendig. Und es geht schnell. Auf meinem Laptop laufen alle 52 Tests für das Domänenmodell insgesamt in etwa einer Sekunde, inklusive Starten der JVM.

Schauen wir uns zumindest einen dieser Tests mal an:

```
class StockKeepingUnitTest {

    @Test
    void nullValueIsRejected() {
        assertThrows(IllegalArgumentException.class,
                () -> new StockKeepingUnit(null));
    }

    @Test
    void invalidCharactersAreRejected() {
        assertThrows(IllegalArgumentException.class,
                () -> new StockKeepingUnit("SKU#001"));
    }

    @Test
    void validValueIsAccepted() {
        StockKeepingUnit sku = new StockKeepingUnit("SKU-001_A");
        assertEquals("SKU-001_A", sku.value());
    }

    @Test
    void valueIsTrimmed() {
        StockKeepingUnit sku = new StockKeepingUnit("  SKU-001  ");
        assertEquals("SKU-001", sku.value());
    }
}
```

Listing 3.11: UnitTest für die StockKeepingUnit-Klasse

Wie wir sehen prüfen wir in Listing 3.12 im Wesentlichen ob unsere Validierungen im Konstruktor wie vorhergesehen funkionieren. Entweder müssen Exceptions geworfen werden oder wir überprüfen ob die Werte entsprechend unseren Wünschen gesetzt wurden.

Das ist auf Stufe des Domainmodells bereits alles - zumindest solange wir nur das Datenmodell anschauen. Nun können wir uns daran machen das erste Stück Funktionalität bereitzustellen.

## 3.3 Funktionales Domänenmodell - Services

So lange wir rein funktional (mit puren Funktionen) unterwegs sind, verbietet sich I/O. Somit gibt es auch keine `load` oder `store` Funktionen.

Aber alles was sich mit unserem Domänenmodell abbilden lässt geht wunderbar. Bisher hat sich under Datenmodell nur mit den Komponenten an und für sich beschäftigt. Wir werden noch etwas mehr benötigen, denn wir wollen mit unserem System ja ausgehend von einem Schaltplan (oder genauer einer Bill-of-Material) letztlich eine Bestellung erzeugen. Typischerweise möchte man bei einigen wenigen Distributoren (idealerweise nur bei einem) bestellen, nur hat nicht immer jeder Distributor alle gewünschten Teile im Angebot. Daher wollen wir uns als erstes Feature eine Funktion anschauen, die eine Liste von Komponenten + Mengenangaben nimmt und uns eine Liste an Bestellungen (für die einzelnen Distributoren) zurückgibt. Da alle Komponenten mindestens ein OrderableItem besitzen muss das immer funktionieren. Zusätzlich wollen wir der Funktion noch einen preferred Supplier mitgeben. Der Einfachheit halber verzichten wir momentan auf sämtliche Optimierungen wie minimaler Preis, minimale Anzahl von Distributoren etc. Wer Spass hat, kann das gerne als Übung umsetzen.

Was wir zunächst brauchen ist eine Struktur um die BOM abzubilden. Das geht mit den folgenden beiden Records:

```
public record BillOfMaterial(
        List<BillOfMaterialEntry> lines) {

    public BillOfMaterial {
        if (lines == null)
            throw new IllegalArgumentException("A BOM must have an entry list");
        lines = List.copyOf(lines);
        if (lines.size() < 1)
            throw new IllegalArgumentException("A BOM must have at least one entry");
    }
}

public record BillOfMaterialEntry(
        InternalPartNumber ipn,
        Quantity quantity) {

    public BillOfMaterialEntry {
        if (ipn == null)
            throw new IllegalArgumentException("A BOM-Entry must have a valid ipn specified");
        if (quantity == null)
            throw new IllegalArgumentException("A BOM-Entry must have a valid quantity specified")
    }
}
```

Listing 3.12: Zwei Records um BOMs abzubilden.

Zusätzlich benötigen wir noch zwei Records um eine Bestellung abzubilden.

```
public record Order(
        Distributor distributor,
        List<OrderEntry> lines
) {
    public Order {
        if (distributor == null)
            throw new IllegalArgumentException(
                "An Order must have a valid distributor specified");
        if (lines == null)
            throw new IllegalArgumentException(
                "An Order must have an entry list");
        lines = List.copyOf(lines);
        if (lines.size() < 1)
            throw new IllegalArgumentException(
                "An Order must have at least one entry");
    }
    public Order withLine(OrderEntry line) {
        if (line == null)
            throw new IllegalArgumentException("OrderEntry must not be null");
        List<OrderEntry> copy = new ArrayList<>(lines);
        copy.add(line);
        return new Order(distributor, copy);
    }
}

public record OrderEntry(
        StockKeepingUnit sku,
        ManufacturerPartNumber mpn,
        Quantity quantity) {
    public OrderEntry {
        if (mpn == null)
            throw new IllegalArgumentException(
                "An OrderEntry must have a valid mpn specified");
        if (sku == null)
            throw new IllegalArgumentException(
                "An OrderEntry must have a valid sku specified");
        if (quantity == null)
            throw new IllegalArgumentException(
                "An OrderEntry must have a valid quantity specified");
    }
}
```

Listing 3.13: Zwei Records um Bestellungen abzubilden.

Im letzten Listing (3.13)sieht man auch gut, wie man ein Listen-Element eines immutable

objects um einen Eintrag erweitert. Man tut das eben nicht sondern erzeugt ein neues, immutable object, das den neuen Eintrag additional beinhaltet. Das alte Objekt bleibt wie es ist.

Wem das ungewohnt erscheint, der erinnere sich daran, wie er/sie seit jeher mit String umgegangen ist. Auch Strings sind in Java schon immer immutable gewesen. Darum gibt String.replace einen neuen String zurück und mutiert den gegeben String eben genau nicht.

Damit ist alles vorbereitet, damit wir unseren ersten Service schreiben können:

```
public class OrderService {

    public static List<Order> generateOrders(
            BillOfMaterial bom,
            Map<InternalPartNumber, Component> componentMap,
            Distributor preferredDistributor) {

        Map<Distributor, Order> orderMap = new HashMap<>();
        for (BillOfMaterialEntry bomEntry : bom.lines()) {
            Component component = componentMap.get(bomEntry.ipn());
            if (component == null)
                throw new IllegalArgumentException(
                    "All BOM-IPNs must be present in the component map");
            /*  Since having at least one orderableItem is an invariant
                we can search the list to find the preferred supplier.
                If that is not present, we fall back to the first one
                specified. */
            OrderableItem orderableItem = component.orderableItems().stream()
                    .filter(item -> item.distributor().equals(preferredDistributor))
                    .findFirst()
                    .orElse(component.orderableItems().get(0));
            Distributor distributor = orderableItem.distributor();
            OrderEntry entry = new OrderEntry(
                    orderableItem.sku(),
                    orderableItem.mpn(),
                    bomEntry.quantity());
            Order existing = orderMap.get(distributor);
            Order updated = (existing == null)
                    ? new Order(distributor, List.of(entry))
                    : existing.withLine(entry);
            orderMap.put(distributor, updated);
        }
        return List.copyOf(orderMap.values());
    }
}
```

Listing 3.14: OrderService mit generateOrders als *pure function*

Alles was wir in Listing 3.14 machen ist schrittweise durch die BOM zu iterieren und Stück für Stück die Bestellungen zu befüllen, die wir uns in einer Map halten. Dadurch können wir elegant auch mit mehreren Distributoren gleichzeitig umgehen. Am Schluss geben wir nur noch die Inhalte der Map zurück und sind fertig.

Wir erwarten vom Aufrufendem Code, dass er uns passend zur BOM eine Map mit den relevanten Komponenten bereitstellt. Das wird später eine Datenbankabfrage benötigen. Dennoch können wir die innere Logik völlig losgelöst davon mit einfachen Unit-Test überprüfen. Genau wie bei den Domänen-Objekten müssen wir auch hier nur für den passenden Input sorgen so dass wir alle Pfade einmal ablaufen, wenn wir mit 100% path-coverage testen wollen. An dieser Stelle erreichen wir beides. Einerseits ist das eine *pure function* - somit ohne I/O und ohne Seiteneffekte und andererseits ausschliesslich von den Datenstrukturen der Domain abhängig.

## 3.4 Applikation und Use-Cases

Nachdem die Domäne an und für sich abgebildet ist können wir beginnen die Usecases abzubilden. Das geht in diesem (trotzdem noch immer) einfachen Fall ganz gut, in grösseren Projekten wird dies wohl eher ein iterativer Prozess. Für dieses System sind 6 Usecases vorgesehen - 3 Stück für das Frontend, sowie 3 weitere für die Anbindung an das EDA System (Listing 3.15):

```
public interface ListAllComponentsUseCase {
    List<Component> invokeListAllComponents();
}

public interface RetrieveComponentInfoUseCase {
    Component invokeRetrieveComponent(InternalPartNumber ipn);
}

public interface AddOrderableItemUseCase {
    Component invokeUcAddItem(InternalPartNumber internalPartNumber,
        OrderableItem orderableItem);
}

public interface EDAListCategoriesUseCase {
    List<Category> invokeEDAListCategories();
}

public interface EDAListCetegoryEntriesUseCase {
    List<Component> invokeEDAListCategories(Category category);
}

public interface EDAAccessPartDetailsUseCase {
    Component invokeEDAPartDetails(InternalPartNumber ipn);
}
```

Listing 3.15: 6 Usecases als Interfaces modelliert (*Driven Ports*)

An dieser Stelle auch nochmals unbedingt der Hinweis, dass wir die Use Cases aus Sicht der Domäne spezifizieren. Hier geht es explizit noch nicht um Datenformate für die Präsentation nach aussen hin. Das gilt sowohl für den Input, als auch für den Output. Man mag versucht sein, die einzige Methode den Use Cases direkt `invoke` zu nennen aber davon ist abzusehen. Sobald ein Service mehrere Dieser Use Cases implementiert, müssten die Methoden strikt eine unterschiedliche Signatur aufweisen, was nicht immer gegeben sein muss. Für die Les- und Debugbarkeit ist das auch nicht förderlich.

Das schöne an dieser Vereinzelung ist, dass ein einziger Blick in das Source-Code-Repository verrät was das System macht, bzw. welche Use Cases es implementiert:

```
srcPackageRoot/
└── application/
    └── port/
        └── in/
            ├── AddOrderableItemUseCase.java/
            ├── EDAAccessPartDetailsUseCase.java/
            ├── EDAListCategoriesUseCase.java/
            ├── EDAListCategoryEntriesUsecase.java/
            ├── ListAllComponentsUseCase.java/
            └── RetrieveComponentInfoUseCase.java/
```

Um diese Use Cases zu implementieren werden wir I/O benötigen. Dazu werden folgende Port-Implementierungen ausserhalb des Hexagons verlangt (Listing 3.16):

```java
public interface LoggingAdapter {
    void debug(String message);
    void info(String message);
    void warn(String message);
    void error(String message);
}

public interface StoragePersistanceAdapter {
    public void persistNewCompnent(Component component);
    public void updateComponent(InternalPartNumber original, Component replacement);
    public Optional<Component> loadComponent(InternalPartNumber ipn);
    public List<Component> loadAllByCategory(Category category);
    public List<Component> loadAll();
    public List<Category> availableCategories();
}
```

Listing 3.16: IO Ports zur Aussenwelt (*Driving Ports*)

Durch dieses Vorgehen lässt wich jegliches I/O aus dem inneren unseres Hexagons Isolieren. Innerhalb des Domänen-Kerns würden wir aber nicht mal dieses Angebot nutzen - dieser soll ja *pure* bleiben. Auf der Schicht aussenrum werden hingegen die Usecases implementiert. An diesem Punkt ist I/O unumgänglich.

Schauen wir uns dazu exemplarisch die Implementierung des AddOrderableItem-Usecase an:

```java
@Service
public class ComponentsUpdateService implements AddOrderableItemUseCase {

    private final RelationalComponentAdapter relationalComponentAdapter;
    private final LoggingAdapter logger;

    public ComponentsUpdateService(
            RelationalComponentAdapter relationalComponentAdapter,
            LoggingAdapter logger) {
        this.relationalComponentAdapter = relationalComponentAdapter;
        this.logger = logger;
    }

    @Override
    public Component invokeUcAddItem(InternalPartNumber internalPartNumber,
            OrderableItem orderableItem) {
        // ********************
        // *** Shell: I/O ***
        // ********************
        Component component = relationalComponentAdapter.loadComponent(
                internalPartNumber).orElseThrow(ComponentNotAvailableException::new);

        // ********************
        // *** Pure Core ***
        // ********************
        component = component.withOrderableItem(orderableItem);

        // ********************
        // *** Shell: I/O ***
        // ********************
        relationalComponentAdapter.updateComponent(internalPartNumber, component);
        logger.info("AddOrderableItemUseCase added: " + orderableItem.mpn()
            + " to Component " + component.ipn() );
        return component;
    }
}
```

Listing 3.17: Hinzufügen einer Bestelloption zu einem Bauteil

An diesem Beispiel (Listing 3.17) sieht man wie im Pattern *functional core & imperative shell* die Trennung zwischen IO und den *pure domain functions* vollzogen wird. Erstens nehmen wir an dieser Stelle selbst keinerlei I/O vor sondern verwenden dazu vollständig die Adapter (oder besser gesagt deren Interfaces). In der Mitte dann der seiteneffektfreie Domain-Code, der aus einer Komponente + neuer Bestelloption eine neue Komponente erzeugt.

Wie schon im Trivialbeispiel sticht auch hier wieder die Lesbarkeit der Signatur der Klasse heraus - *ComponentsUpdateService implements AddOrderableItemUseCase*. Durch den strikten Umweg über die Ports bleiben wir auch auf dieser Ebene noch vollständig technologiefrei und testbar.

## 3.5 Persistence Adapter

Genau wie bei dem Trivialbeispiel benötigen wir einen Persistenzadapter zu unserer Datenbank. Dazu erstellen wir uns Entity-Klassen die die Datenhaltung der relationalen Datenbank als Objekte Repräsentieren. Es genügt die Hauptkomponente `ComponentEntity` anzuschauen:

```
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class ComponentEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NonNull
    @Column(unique = true)
    private String ipn;

    private String symbolRef;

    private String footprintRef;

    @Enumerated(EnumType.STRING)
    private Category category;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @Builder.Default
    private List<OrderableItemEntity> orderableItems = new ArrayList();

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @Builder.Default
    private List<AttributeEntity> attributes = new ArrayList();
}
```

Listing 3.18: Component Entity

Im Vergleich zum Trivialbeispiel sind hier die Relationen zu anderen Entities hervorzuheben. Mit `@OneToMay` bringen wir eine *1:N*-Relation zum Ausdruck. Weiter fordern wir, dass sowohl die `OrderableItemEntity`, als auch die `AttributeEntity` vollständig abhängige Entitäten sind, die

sowohl bei Persist als z.B. auch bei Delete mit angelegt oder entfernt werden. Wir forder sogar, dass es keine nicht-zugeordneten Entitäten von diesen beiden Typen geben darf. Enums können direkt aus dem Domänenmodell hinterlegt werden; wie in der Annotation deklariert, werden sie als Strings auf der Datenbank abgebildet.

Um die Daten in der DB laden und speichern zu können, benötigen wir wieder ein JPA-Repository. In diesem Fall kann es nicht leer bleiben:

```
public interface ComponentRepository extends JpaRepository<ComponentEntity, Long> {
    Optional<ComponentEntity> findByIpn(String ipn);

    boolean existsByIpn(String ipn);

    List<ComponentEntity> findByCategory(Category category);

    void deleteByIpn(String ipn);

    @Query("SELECT DISTINCT c.category FROM ComponentEntity c")
    List<Category> findDistinctCategories();
}
```

Listing 3.19: Component Repository

In Listing 3.19 haben wir die Standard-Funktionen wie *findById, save, deleteByID, usw.* das würde für unsere Zwecke aber nicht genügen. Aus Performance-Gründen haben wir eine numerische ID, die wir aber ausserhalb der Datenbank gar nicht verwenden. Unsere Domänenobjekte kennen diese ID nicht, denn es ist ein rein fachliches Artefakt, das nichts mit der Domäne zu tun hat. Genau an solchen Beispielen sehen wir die konzeptuellen Vorteile eines Domänengetriebenen Ansatzes. So lassen wir überhaupt nicht zu, dass unsere Domänemodelle von technische Randbedingungen kontaminiert werden. Damit wir unsere Komponenten finden und nach Kategorien ordnen können. Das Interface zeigt hier beide Möglichkeiten Queries zu spezifizieren. Einmal über eine strikte Namenskonvention der Interfaces (nachzulesen in [8]), einmal über die Angabe der Query in JPQL (Spezifikation unter [9]).

Somit haben wir alles vorbereitet um unseren Persistenz-Adapter zu erstellen:

```
@Service
@AllArgsConstructor
public class RelationalComponentAdapter implements StoragePersistanceAdapter {

    private ComponentRepository componentRepository;

    @Override
    public void persistNewCompnent(Component component) {

        if (componentRepository.existsByIpn(component.ipn().value())) {
            throw new IllegalArgumentException("Component must not yet exist to be saved");
        }

        ComponentEntity ce = fromDomain(component);
        componentRepository.save(ce);
    }
    ...
    private OrderableItemEntity fromDomain(OrderableItem oi) {
        return OrderableItemEntity.builder()
                .mpn(oi.mpn().value())
                .sku(oi.sku().value())
                .distributor(oi.distributor())
                .minOrderQuantity(oi.minOrderQuantity().value())
                .orderMultiple(oi.orderMultiple().value())
                .price(oi.price().value())
                .currency(oi.price().currency())
                .attributes(oi.attributes()
                        .entrySet()
                        .stream()
                        .map(e -> fromDomain(e.getValue()))
                        .collect(Collectors.toList()))
                .build();
    }
    ...
}
```

Listing 3.20: Persistence Adapter (Klasse nur teilweise dargestellt)

Da wir uns nun ausserhalb unseres Hexagons befinden dürfen wir an dieser Stelle jeden Technologiebezug haben, den wir uns wünschen. Durch die `@Service` Annotation kann Spring die Instanziierung dieser Klasse vornehmen, mittels `@AllArgsConstructor` erhalten wir über Lombok einen Konstruktor, der ein *ComponentRepository* verlangen wird. Spring verdrahtet uns das automatisch. An dieser Stelle müssen wir auch zwischen Domänen und Entity-Objekten konvertieren. In diesem Fall passiert das auf Ebene der Adapterimplementierung. Es hätte aber auch nichts dagegen gesprochen diese Abbildung in der Entity-Klasse vorzunehmen. Das wäre angebracht, wenn

man diese Transformation in mehreren anderen Klassen bräuchte. An diesem Punk gelten einfache DRY-Prinzipien.

## 3.6 Integration in ein EDA-System

Mit dem nächsten Adapter binden wir ein technisches System an unser Backend an. In diesem Fall ein EDA-System *Electronic Design Automation*, also im Wesentlichen ein Elektronik-CAD. Für dieses Beispiel nehmen wir *KiCad*, das aktuell wohl bekannteste Open-Source EDA. KiCad hat selbst eine Bauteilbibliothek, kann aber über mehrere Mechanismen und Plugins externe Bibliotheken einbinden. In grösseren Unternehmen ist es eigentlich die Regel über ein zentralisiertes Bauteilmanagement zu verfügen. Genau das simulieren wir und nutzen die Fähigkeit von KiCad, ein REST-Interface anzusprechen.

Der Spezifikation des REST-Interfaces [4] können wir entnehmen, dass es im Wesentlichen drei Abfragearten gibt: Welche Kategorien es gibt, welche Bauteile sich in einer bestimmten Kategorie befinden und natürlich Details zu einem bestimmten Bauteil abzufragen.

Als Ergebnis wird JSON erwartet. Allerdings mit der Einschränkung, dass alle Elemente reine Strings sind. KiCad parst und konvertiert das eigenständig, wir werden auf Seite der Adapter das gleiche tun.

```
public record KicadComponentDetail(
            String id,
            String name,
            String description,
            String symbolIdStr,
            String exclude_from_bom,
            String exclude_from_board,
            String exclude_from_sim,
            Map<String, KicadFieldData> fields) {

        public static KicadComponentDetail fromDomain(Component c) {
            return new KicadComponentDetail(
                c.ipn().value(), c.ipn().value(), c.ipn().value(),
                c.symbol().value(), "false", "false", "false",
                Map.of("Value", new KicadFieldData(extractValue(c), "true"),
                    "Footprint", new KicadFieldData(c.footprint().value(), "false")));
        }

        private static String extractValue(Component c) {
                Optional<SpecificationItem> nominalItem = c.attributes().entrySet().stream()
                    .filter(e -> e.getValue().property().name().contains("NOMINAL"))
                    .findFirst().map(e -> e.getValue());
                if (nominalItem.isEmpty()) {
                    return c.orderableItems().get(0).mpn().value();
                }
                return nominalItem.get().value() + nominalItem.get().unit().symbol();
        }
}
```

Listing 3.21: KiCad Adapter (KicadComponentDetail - DTO)

Im Beispiel schauen wir uns die Abbildung unserer Components in das entsprechende KiCad-DTO an (Listing 3.21). Da es keinen Rückweg gibt - also wir vom CAD-Programm nie Daten erhalten werden - ist auch nur die Abbildungsrichtung von der Domäne zu KiCad realisiert. Das erste "Problem" ist, dass KiCad einen eindeutigen Identifier benötigt. Den realisieren wir über die *Internal Part Number*, die wir ohnehin für eindeutig erklärt haben. *Name* und *Beschreibung* setzen wir für unsere Tests vorerst ebenfalls auf unsere IPN. Eine gute Beschreibung des Bauteils nebst Links zu den Datenblättern könnte man jederzeit im Domainmodell nachführen. Weitere Werte erwartet KiCad als assioziatives Array - also einer Map. Footprint und Value sind fixe Feldschlüssel, die KiCad versteht. Die Bereitstellung eines "Values" ist eine kleine Herausforderung. Nicht jedes Bauteil hat das. Ein Widerstand z.B. hat natürlich einen klaren Wert - nämlich seinen Widerstandswert. Ein integrierter Schaltkreis nicht unbedingt. Hier habe ich mich entschieden, nach *Properties* zu suchen, die mit `NOMINAL` beginnen. Ein Bauteil mit Werten sollte so ein NOMINAL-Feld bereitstellen. Falls das nicht vorhanden ist, geben wir die MPN des ersten Suppliers an KiCad weiter. Das ist an diesem Punkt sicher noch ausbaufähig. Es zeigt aber schön, wie wir uns beim Modellieren der Domäne (unnötig) eingeschränkt hätten, wenn wir nur von diesem einen CAD-Produkt ausgegangen wären. Andere CAD-Produkte hätten vielleicht ganz andere Anforderungen.

Am Schluss wollen wir noch kurz in den REST-Controller schauen. Überraschungen sollte es nun ja keine mehr geben:

```
@RestController
@RequestMapping("/kicad-api/v1")
@AllArgsConstructor
public class KiCadControllerREST {

    private final EDAAccessPartDetailsUseCase accessPartDetailsUseCase;

    ...

    @GetMapping("/parts/{id}.json")
    public ResponseEntity<KicadComponentDetail> part(@PathVariable String id) {
        return ResponseEntity.ok(
            KicadComponentDetail.fromDomain(
                accessPartDetailsUseCase.invokeEDAPartDetails(
                    new InternalPartNumber(id))
                ));
    }

}
```

Listing 3.22: Ausschnitt des KiCad REST-Controllers

Listing 3.22 zeigt nur einen Ausschnitt aus dem REST-Controller aber die Mechanik ist für alle anderen Endpoints in etwa identisch. Genau wie für menschliche Akteure steuert auch dieser Controller UseCases an, die unser Domänenkern bereitstellt. Da wir bereits alles vorbereitet haben ist die Methode, die den GET-Request abhandelt entsprechend kurz. Genau wie im Trivialbeispiel müssen wir uns an dieser Stelle nicht darum kümmern, ob die `id` wirklich existiert. Sollte das nicht

der Fall sein, wir unser Domänenkern eine `ComponentNotAvailableException` werfen. Dadurch wird unser REST-Controller mit einer 400-Bad-Request antworten, so wie wir das übergreifend konfiguriert haben.

Wenn wir KiCad entsprechend konfigurieren (Hostname, Port und API-Path), dann können wir selbst mit unserer kleinen Implementierung wirklich Buteile laden.

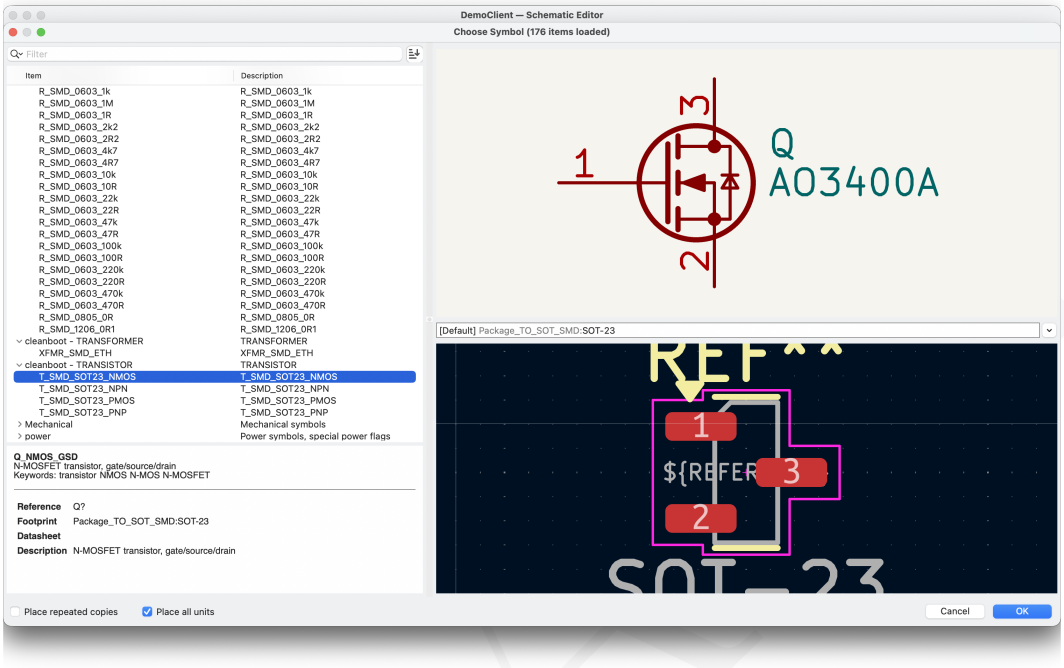



Abbildung 3.1: Zugriff auf die Bauteilbibliothek via KiCad

## 3.7 Ein einfaches Vanilla-JS Frontend

Eigentlich soll Frontend-Entwicklung nicht der zentrale Fokus dieses Skripts sein. Hierzu gibt es unzählige bessere Tutorials und eine nicht abreissende Flut von immer neuen Ansätzen, Frameworks und Rädern die täglich neu erfunden werden. Für jeden, der Informatik seriös studiert hat, ist Javascript wohl nach wie vor eine Zumutung. So schlimm, dass man mit TypeScript versucht hat einen Überbau zu erzeugen, der dann zu Javascript *transpiliert*. TypeScript hat Typen, aber die sind (aufgrund der Rückwärtskompatibilität) wohl mehr als Vorschlag als eine statische und strikte Typisierung zu verstehen. Immerhin sind viele gute Ansätze aus TypeScript zu JavaScript zurück geflossen, so dass wir zumindest ordentliche Modulstrukturen und ES6-Klassen erhalten haben. Hinzu kommen die stetig neuen Frameworks, die meist eines gemeinsam haben: Eine durch *Node* verwaltete Dependency-Hölle. Es ist nicht ungewöhnlich dass diese Scripts bereits für Demoapplikationen hunderte Pakete nachladen von zig verschiedenen Entwicklern. Täglich gibt es überall Bugfixes, teils mit breaking changes - ein Fixieren aller Versionen scheint aufgrund der Bedrohungslage durch Fehler keine gute Idee. Aber vielleicht bin ich zu wenig Frontendentwickler um dafür Verständnis aufzubringen. Ausserdem darf jeder/jede machen, was er/sie will.

Diese Kapitel ist auch weniger meine Vorstellung “wie es sein soll” sondern eher wie man zu einem Frontend kommt ohne auf noch mehr Technologie einzugehen. Vielleicht schreibe ich irgendwann auch mal ein Skript über saubere Frontendentwicklung - falls so etwas überhaupt möglich ist.

Anstelle von grossen Frameworks nehmen wir daher einfach nichts und bauen trotzdem ein Frontend. Ohne Compile-Schritt, ohne Packetierung, ohne Depencency-Hölle. Man kann dieses Kapitel als Exploration lesen, wie sich Webentwicklung ohne Frameworks nur mit WebStandards anfühlt. Am Schluss muss man selbst entscheiden ob das für das eigene Projekt ein gangbarer Weg ist. Wahrscheinlich eher nicht ...

In der aktuellen Form ist das - im Gegensatz zu den überlegungen im Backend - **NICHT** production-ready und soll auch nicht so verstanden werden. Mehr ein Denkansatz zum eigenen Evaluieren eines geeigneten Frontends.

Immerhin sind wir durch die Beschränkung auf Web-Standards in hohem Masse Browserunabhängig und können auch recht sicher sein, dass so ein Frontend auch in 10+ Jahren noch identisch funktioniert.

### 3.7.1 Projekt-Setup

Damit wir ein lauffähiges Frontend erhalten können, benötigen wir zuerst einen Webserver. Spring stellt uns diesen direkt zur Verfügung. Alle Dateien, die unter `_root/src/main/resources/static` abgelegt wird, wird von Spring direkt per http zur Verfügung gestellt. Zumindest solange die Dependency *spring-boot-starter-web* inkludiert ist. Das ist bei uns aufgrund der REST-Schnittstelle aber sowieso gegeben. Keinen externen Webserver zu verwenden hat hierbei noch den Vorteil, dass sowohl Back- als auch Frontend von der gleicher Quelle stammt. Anderenfalls müssten wir uns aktiv um *CORS - Cross Origin Request Sharing* Gedanken machen. Für die lokale Entwicklung eine Hürde mehr. Ganz sicher ist die Antwort darauf nicht die CORS-Protection etwa abzuschalten, so wie man das immer wieder als Empfehlung liest!

Es ginge tatsächlich gänzlich ohne Dependencies aber wer nicht tagelang mit CSS basteln will, der nimmt dann doch einfach Bootstrap, damit die Oberfläche zumindest einigermassen ansehnlich erscheint. Das ist jedoch nur ein einziges .css file. Das können wir problemlos statisch anbieten.

Unseren Web-Folder strukturieren wir wie folgt:

```
src/main/resources/static/
├── 3rd-party/
│   ├── bootstrap.css
│   ├── bootstrap.css.map
│   └── LICENSE
├── components/
├── lib/
│   ├── NPElement.js
│   ├── NPRouter.js
│   └── NPService.js
├── services/
├── app.js
├── index.html
└── LICENSE
```

Unsere `index.html` ist dabei recht spartanisch. Alles was wir hier wollen ist, dass boostrap.css geladen wird und wir unser app.js einbinden.

```
<!DOCTYPE html>
<html>
    <head>
        <link href="./3rd-party/bootstrap.css" rel="stylesheet">

    </head>
    <body>
        <np-app></np-app>
        <script type="module" src="./app.js"></script>
    </body>
</html>
```

Listing 3.23: index.html

Ebenfalls recht übersichtlich ist unser `app.js` (Listing 3.24) aber es verlangt nach ein paar Erklärungen. Zuerst `NPElement` oder generell der *NP*-Präfix. Mir ist nichts besseres eingefallen - “Nano&Pico” soll das heissen. Es sollte ja möglichst kein sein. Egal, die drei Klassen unter `lib/` sollen einfach oft wiederverwendete Funktionen kapseln. `NPElement` erbt selbst von `HTMLElement` was wiederum zum WebStandard gehört. Eine der zentralen Funktionen ist safeHTML dass uns das *Escapen* der String-Literals ermöglicht. Daneben haben wir noch den `NPRouter` der einfache Routing-Funktionalitäten zur Navigation bereitstellt. Wie wir sehen, haben wir nur ganz wenige Routen. Der letzte Eintrag zeigt auch gleich, wie wir eine Seite mit einem übergebenen Parameter laden können. Was wir auch noch erklären müssen ist `_getLocalElementNameByID(id)`. In NP-Element ist so implementiert dass alle `IDs` automatisch mit einer UUID als Prefix versehen werden. Das ist nötig, da wir auf einen Shadow-DOM verzichten und damit sicherstellen müssen, dass wir eindeutig auf unsere gekapselten Elemente zugreifen können. Die genannte Funktion stellt bei der Abfrage dann ebenfalls die korrekte UUID voran.

```
import { NPElement } from './lib/NPElement.js'
import { HomePage } from './components/pages/HomePage.js'
import { ComponentsPage } from './components/pages/ComponentsPage.js'
import { OrderableItemsPage } from './components/pages/OrderableItemsPage.js'


import { NavBar } from './components/navigation/NavBar.js'
import { NPRouter } from './lib/NPRouter.js';


export class App extends NPElement {

    get template() {
        return this.safeHTML`
            <nav-bar></nav-bar>
            <div id="outlet"></div>
        `;
    }

    constructor(parameters) {
        super();
        this.render();

        const router = new NPRouter(this._getLocalElementNameByID('outlet'));
        router.define([
                { path: '/', component: () => new HomePage() },
                { path: '/home', component: () => new HomePage() },
                { path: '/components', component: () => new ComponentsPage() },
                { path: '/components/:id/orderableItems',
                    component: (id) => new OrderableItemsPage(id) },
            ]);

        router.handleInitialLoad();
        router.enableLinkInterception();

        window.router = router;
    }

    render() {
        this.innerHTML = this.template;
    }

}

if (!customElements.get('np-app')) {
    customElements.define('np-app', App);
}
```

Listing 3.24: app.js

`NPRouter.js` und `NPElement.js` sind mit 130 und 53 Zeilen Code inkl. Kommentaren übersichtlich klein. Dazu kommen noch 50 Zeilen in `NPService.js`. Das heisst unser Framework weist weniger als 250 Zeilen Code auf und nimmt uns gleichwohl schon eine Menge Arbeit ab.

### 3.7.2 Anbindung an das Backend

`NPService` ist übrigens nur ein Wrapper rund um die Fetch-API so dass wir bequem auf unser REST-Interface zugreifen können. Listing 3.25 zeigt, wie wir das für unser REST-Interface anwenden können:

```
import {NPService} from '../lib/NPService.js'

export class ComponentService extends NPService {
static #instance = null;

  static getInstance() {
    if (!ComponentService.#instance) {
      ComponentService.#instance = new ComponentService();
    }
    return ComponentService.#instance;
  }

  constructor() {
    super('/api/components');
  }

  getAllComponents() {
    return this.request('');
  }

  getComponent(internalPartNumber) {
    return this.request('/' + internalPartNumber);
  }

  addOrderableItem(internalPartNumber, orderableItem) {
    return this.post('/' + internalPartNumber + "/orderableItems", orderableItem);
  }

}
```

Listing 3.25: ComponentService.js

Somit ist es ein simpler Wrapper damit wir unsere Objekte im Frontend abfragen können ohne, dass sich Darstellungs- und Infrastrukturcode mischen würde.

### 3.7.3 Entwurf der einzelnen Pages

Exemplarisch schauen wir uns die Liste aller Komponenten an.

```
import { NPElement } from '../../lib/NPElement.js';
import { ComponentService } from '../../services/ComponentService.js';

export class ComponentsPage extends NPElement {
  constructor() {
    super();
    this.componentService = ComponentService.getInstance();
    this.render();
    this.loadComponents();
  }

  get template() {
    return this.safeHTML`
      <h1>Available Components</h1>
      <div id="componentList">Loading...</div>
    `;
  }

  render() {
    this.innerHTML = this.template;
  }

  async loadComponents() {
    try {
      const components = await this.componentService.getAllComponents();

      if (!Array.isArray(components)) {
        throw new Error("Books response is not an array");
      }

      this._renderComponents(components);
    } catch (error) {
      this._displayError(`Failed to load components: ${error.message}`);
      console.error('Components loading error:', error);
    }
  }
  ...
}
```

Listing 3.26: ComponentsPage.js - Teil 1

Listing 3.26 zeigt uns einen Teil der ComponentsPage-Komponents. Im Konstruktor besorgen

wir uns eine Referenz auf den Components-Service und rendern erst mal einen Placeholder. Das eigentliche Laden der Komponenten erfolgt Asynchron. Sobald wir alle Komponenten geladen haben, rendern wir erneut und ersetzen unseren Platzhalter. Wie gesagt kein Shadow-DOM - wir ersetzen einfach das innerHTML von uns selbst. Da wir von HTMLElement erben, passt das.

```
import { NPElement } from '../../lib/NPElement.js';
import { ComponentService } from '../../services/ComponentService.js';

export class ComponentsPage extends NPElement {
    ...

    _renderComponents(components) {
        const container = this._getLocalElementByID('componentList');

        const lines = components.map(
            component => this._createComponentEntry(component)).join('');
        container.innerHTML = this.unsafeHTML`
        <table class="table">
            <thead>
            <tr>
                <th scope="col">Part</th>
                <th scope="col">Category</th>
                <th scope="col">Value</th>
                <th scope="col">Footprint</th>
                <th scope="col">Footprint</th>
            </tr>
            </thead>
            <tbody>
            ${lines}
            <tbody>
        </table>`;
  }
  _createComponentEntry(component) {
    return this.safeHTML`
      <tr>
        <th scope="row">
            <a href="/components/${component.ipn}/orderableItems" data-route>
                ${component.ipn}
            </a></th>
        <td>${component.category}</td>
        <td>${component.value}</td>
        <td>${component.symbol}</td>
        <td>${component.footprint}</td>
      </tr>
    `;
  }

  _displayError(message) {
    const container = this._getLocalElementByID('componentList');
    container.innerHTML = this.safeHTML`<p class="text-danger">${message}</p>`;
  }
}
```

Listing 3.27: ComponentsPage.js - Teil 2

Im zweiten Teil der Klasse in Listing 3.27 geht es nur noch um das Rendering der Komponenten. Zu beachten ist die Verwendung von `unsafeHTML` und `safeHTML`. Da `safeHTML` alle Tags in den Daten escapen würde, kann man das nicht verschachtelt aufrufen. Um so wichtiger ist es hier strikt darauf zu achten, dass niemals Daten vom Backend in einem unsafe-Block gerendert werden. Anderenfalls würde man hier Tür und Tor für XSS oder ähnliche Angriffe öffnen. Damit man auch erkunden kann, wie man etwas zum Server schickt, ist in der Klasse OrderableItemsPage als Demo vorgesehen, dass man weitere Bestelloptionen zu einer Komponente hinzufügt. Zum Schluss noch ein Screenshot von unserem Nutzerinterface:

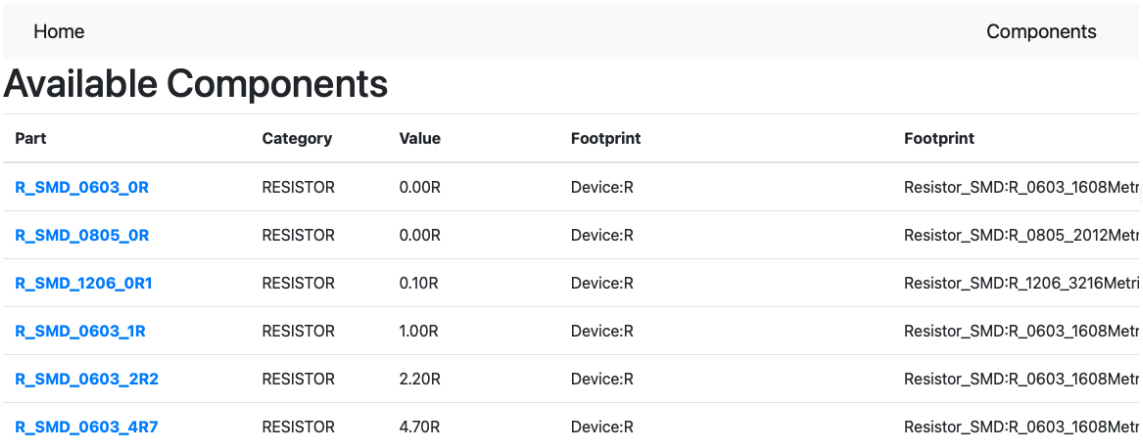



Abbildung 3.2: Frontend - Listing aller Komponenten

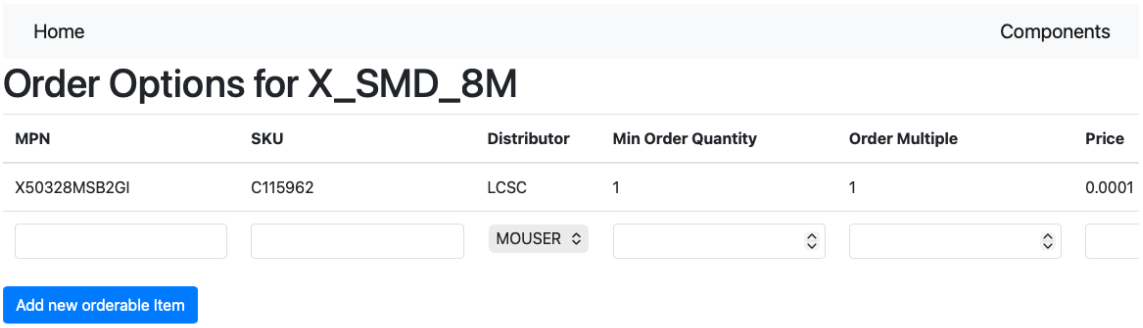



Abbildung 3.3: Frontend - Bearbeiten der Lieferoptionen einer Komponente

Damit ist unser Rundgang durch die Frontend-Welt auch schon beendet. Wer jetzt der Meinung ist, dass das mit Framewoks wie Anguar, Vue oder React alles viel einfacher ginge der mag sicher Recht haben - für unseren Kurs bleiben wir dennoch Frameworkless, da wir nicht noch mehr Technologie in den Stack laden wollten. Ausser XSS prevention passiert bei uns im Frontend nicht viel Es ist für uns eher ein einfacher Weg Abläufe im Backend auszulösen.

# Kapitel 4

# Zum Schluss

Ich hoffe dieses Script hat dazu beigetragen, dass das Java-Wissen etwas aufgefrischt wurde - oder dass die, die bereits seit Jahren mit Java entwickelt haben, dennoch etwas mitnehmen konnten, und wenn es nur aus Architektursicht eine neue Perspektive ermöglicht. Evtl. konnten auf diese Weise auch ein paar *modernere* Sprachfeatures wie die `records`, Stream-API oder das functional-core-Pattern mitgenommen werden.

Gerade im Sicherheitsbereich wurden viele Dinge gar nicht thematisiert. Das war aber ganz bewusst, da wir die Security-Belange ausführlich im Kurs thematisieren. Das Skript sollte an dieser Stelle nicht redundant werden. Über was wir auch noch an keinem Punkt gesprochen haben sind die verschiedenen Deployment-Optionen. Auch das machen wir noch im Kurs.

Wer das Skript bis zum Schluss durchgelesen hat, sollte (zumindest was) Java angeht, bestens auf den Kurs vorbereitet sein.

## 4.1 Contribution

Wer zu dem Sckript etwas beitragen möchte sei herzlich dazu eingeladen. Es wird über GitHub öffentlich als OER zur Verfügung gestellt und kann (und soll!) gerne geforked werden. Gerne dürfen auch Issues aufgebommen oder pull-requests gesendet werden. Für die Vorbereitung unseres Kureses reicht der Inhalt aktuell aus meiner Sicht aber das Ganze darf sich gerne auch zu einem eigenständigen Lehrmaterial weiterentwickeln.

Zum Abschluss wünsche ich allen Teilnehmenden viel Spass mit dem Kurs *CAS Secure Software Design & Development* and der ZHAW.

# Literatur


[1] Gary Bernhardt. *Ruby Conf 12 - Boundaries*. Youtube. 2012. URL: https://www.youtube.com/watch?v=yTkzNHF6rMs.

[2] Alistair Cockburn und Juan Manuel Garrido de Paz. „Hexagonal architecture explained". In: (2024).

[3] *Die beliebtesten Programmiersprachen weltweit laut PYPL-Index im Oktober 2025*. https://de.statista.com/statistik/daten/studie/678732/umfrage/beliebteste-programmiersprachen-weltweit-laut-pypl-index/. Accessed: 2026-1-14.

[4] *HTTP LIBRARIES*. https://dev-docs.kicad.org/en/apis-and-binding/http-libraries/index.html. Accessed: 2026-1-10.

[5] *Inversion of Control Containers and the Dependency Injection pattern*. https://martinfowler.com/articles/injection.html. Accessed: 2025-12-25.

[6] I. Jacobson. *Object-oriented Software Engineering: A Use Case Driven Approach*. Pearson Education, 1993. ISBN: 9788131704080.

[7] *Jakarta Bean Validation 2.0*. Techn. Ber. JSR 380. Java Community Process, 2019.

[8] *JPA Query Methods*. https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html. Accessed: 2026-1-10.

[9] *JPQL Language Reference*. https://docs.oracle.com/html/E13946_04/ejb3_langref.html. Accessed: 2026-1-10.

[10] *Maven in 5 Minutes*. https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html. Accessed: 2025-12-25.

[11] Scott Wlaschin. *Moving IO to the edges of your app: Functional Core, Imperative Shell*. Youtube. 2024. URL: https://www.youtube.com/watch?v=P1vES9AgfC4.