

Programmentwurf EasyLog Phone Solutions

Schriftliche Dokumentation

Für die Prüfung zum

Bachelor of Science

Des Studienganges Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Amjad Alnakshbandi

Student: Amjad Alnakshbandi

Kurs: TINF21B4

Dozent: Mirko Dostmann

Abgabedatum: 19.05.2025

Repository: <https://github.com/amjadalnakshbandi/EasyLog>

Inhalt

Use Cases	3
Domain Driven Design	5
Die Ubiquitous Language	5
Analyse und Begründung der verwendeten Muster	6
Value Objects	6
Entities	8
Aggregates	9
Repositories	10
Domain Services	10
Clean Architecture	12
Abstraction Code	12
Domain Code	13
Application Code	14
Adapters Code	14
Plugins	15
Programming Principles	15
Single Responsibility	15
Interface Segregation	16
DRY	17
KISS	17
Information Expert & Creator-Prinzip	19
Refactoring	20
Refactor 1	20
Refactor 2	21
Entwurfsmuster	22
Testen	23
Value Objekten	23
Controller	24
Abbildungsverzeichnis	26

Use Cases

Das Projekt ist eine Java Backend Anwendung, die ein Verwaltung Plattform für Handy Lage darstellt.

Folgende Use Cases wurden definiert:

1. **Benutzerregistrierung:** Als Administrator kann ich neue Benutzerkonten für Administratoren oder Mitarbeiter anlegen, indem ich die erforderlichen Daten wie Vorname, Nachname und Passwort eingebe. Dadurch wird sichergestellt, dass die Nutzer später Zugriff auf die Software erhalten. Nach der Erstellung eines Kontos erhalte ich eine Bestätigung, die mir den erfolgreichen Abschluss des Vorgangs anzeigt. Die neu registrierten Benutzer können sich anschließend mit ihren Zugangsdaten problemlos im System anmelden und die Software nutzen.
2. **Mitarbeiter-Logins:** Mitarbeiter können sich mit ihrer registrierten E-Mail-Adresse und dem zugehörigen Passwort im System anmelden. Nach der Eingabe der Zugangsdaten wird die Authentifizierung durchgeführt. Bei erfolgreicher Anmeldung erhält der Nutzer eine Bestätigung mit dem Status „Erfolgreich“ sowie ein gültiges Zugriffstoken. Dieses Token dient zur sicheren Identifikation bei späteren Anfragen und ermöglicht den Zugriff auf die entsprechenden Systemfunktionen. Sollten die Anmeldedaten nicht korrekt sein oder ein technisches Problem auftreten, wird stattdessen eine Fehlermeldung angezeigt.
3. **Mitarbeiter-Ausloggens:** Mitarbeiter können sich sicher aus dem System abmelden, indem sie eine Anfrage mit ihrer registrierten E-Mail-Adresse und dem aktuellen Zugriffstoken senden. Der Server beendet daraufhin die aktive Sitzung und entwertet das verwendete Token, sodass es nicht mehr für zukünftige Anfragen genutzt werden kann. Bei erfolgreicher Abmeldung erhält der Nutzer eine Bestätigung, dass der Logout-Vorgang abgeschlossen wurde. Falls ein Fehler auftritt – beispielsweise bei einer ungültigen Token- oder E-Mail-Adresse – wird eine entsprechende Fehlermeldung zurückgegeben.
4. **Handybestellung entgegennehmen:** Ein Mitarbeiter kann eine neue Handybestellung im System erfassen. Dafür sendet er eine Anfrage mit den relevanten Bestelldaten, darunter die Handy-ID, Filial-ID, Bezeichnung des Geräts, der Filialname sowie die gewünschte Menge. Zur Autorisierung wird ein

gültiges Zugriffstoken im Header der Anfrage mitgeführt. Nach erfolgreicher Verarbeitung erhält der Mitarbeiter eine Bestätigung, dass die Bestellung angelegt wurde. Die Rückmeldung enthält Details wie eine generierte Bestellnummer, die bestellten Artikel, die zugehörige Filiale, die Menge sowie das Erstellungsdatum der Bestellung.

5. **Benutzerlisten-Abrufs (exklusiv für Super-Admins):** Der Zugriff auf die vollständige Benutzerliste ist eine geschützte Funktion, die ausschließlich dem Super-Admin vorbehalten ist. Dieser authentifiziert sich mit einem speziellen, unveränderbaren Token ("ASE"), der nicht wie reguläre Session-Tokens rotiert oder neu generiert werden kann. Bei erfolgreicher Anfrage erhält der Super-Admin eine Liste aller registrierten Benutzer mit Details wie Vor- und Nachname, E-Mail-Adresse, Rolle (Admin/Mitarbeiter) sowie dem. Der Zugriff auf diese Funktion unterliegt strengen Sicherheitsvorkehrungen. Normale Administratoren besitzen keine Berechtigung, um die Benutzerliste einzusehen – dieses Recht ist ausschließlich dem Super-Admin vorbehalten. Eine manuelle Änderung oder Anpassung des Tokens ist nicht möglich, wodurch die Sicherheit zusätzlich erhöht wird. In der API-Antwort werden zudem aus Sicherheitsgründen niemals aktive Session-Tokens angezeigt. Stattdessen wird dieses Feld stets mit einem Nullwert zurückgegeben, um Missbrauch zu verhindern. Diese Maßnahmen stellen sicher, dass sensible Benutzerdaten optimal geschützt sind und nur autorisierte Super-Admins Zugriff auf diese Informationen erhalten.

Aufgrund zeitlicher Kapazitätsengpässe konnten die folgenden Use-Cases in der aktuellen Version noch nicht umgesetzt werden:

1. Handys ins Lager einbuchen
2. Lagerbestand anzeigen und anpassen
3. Filial- und Lagerstandorte verwalten

Domain Driven Design

Die Ubiquitous Language

Die Ubiquitous Language bildet das zentrale, einheitliche Vokabular für Domänenexperten und Entwicklungsteam. Durch konsequente Verwendung derselben Begriffe in Fachdiskussionen und Quellcode wird Missverständnissen vorgebeugt und eine präzise Kommunikation sichergestellt. Da es sich um ein international ausgerichtetes Projekt handelt, wurde Englisch als Standardsprache festgelegt. Dies gilt sowohl für die Fachbegriffe als auch für die Codebasis.

1. **Order:** Repräsentiert eine Bestellung im System, bestehend aus Artikelreferenz (productId), Filialzuordnung (branchId), Menge (quantity) und Metadaten. Wird durch einen eindeutigen orderId identifiziert.
2. **User:** Oberbegriff für alle Systemnutzer. Unterschieden wird zwischen:
 - **Admin:** Nutzer mit administrativen Rechten, die folgende Aktionen durchführen können:
 - Handys ins Lager einbuchen
 - Handybestellungen entgegennehmen
 - Mitarbeiter und weitere Admins anlegen
 - Lagerbestand anzeigen und anpassen
 - Filial- und Lagerstandorte verwalten
 - **Super-Admin:** Spezieller Admin mit zusätzlichen Rechten:
 - Abruf vollständiger Benutzerlisten
 - **Mitarbeiter:** Standardnutzer mit eingeschränkten Funktionen wie das Handybestellungen entgegennehmen
3. **Constants:** Systemweite unveränderliche Werte wie:
4. **Success Response:** Standardisiertes Antwortformat für erfolgreiche Operationen
5. **Error Response:** Vereinheitlichte Fehlerantwort mit Statuscode und Beschreibung

Diese klare Begriffstrennung und Rollendefinition ermöglicht eine präzise Implementierung der Geschäftslogik und schafft Transparenz über die Systemfunktionalitäten. Die Hierarchie der Benutzerrollen mit ihren spezifischen Rechten ist dabei besonders wichtig für die Sicherheitskonzeption des Systems.

Analyse und Begründung der verwendeten Muster

Value Objects

Im Rahmen dieses Projekts nutze ich ein eigenes Paket namens "valueobject", um Wertobjekte zu verwenden. Allgemein kann festgestellt werden, dass Wertobjekte Konzepte mit Attributen darstellen, und ihre Gleichheit wird durch die Gleichheit ihrer Attributwerte bestimmt. Das ist es, was wir im Projekt sehen werden. In der Abbildung daneben sehen wir das Domain Schicht und was sie von Packages enthält, gefragt kann aber, warum so viel Wertobjekten? Am Beispiel der Password-Klasse (siehe Abbildung unten) lässt sich exemplarisch zeigen, wie Wertobjekte (Value Objects) im Domain-Driven Design umgesetzt werden. Diese Klasse ist im Paket user.valueObject implementiert und demonstriert mehrere zentrale Prinzipien des Value-Object-Konzepts. Die Klasse ist als

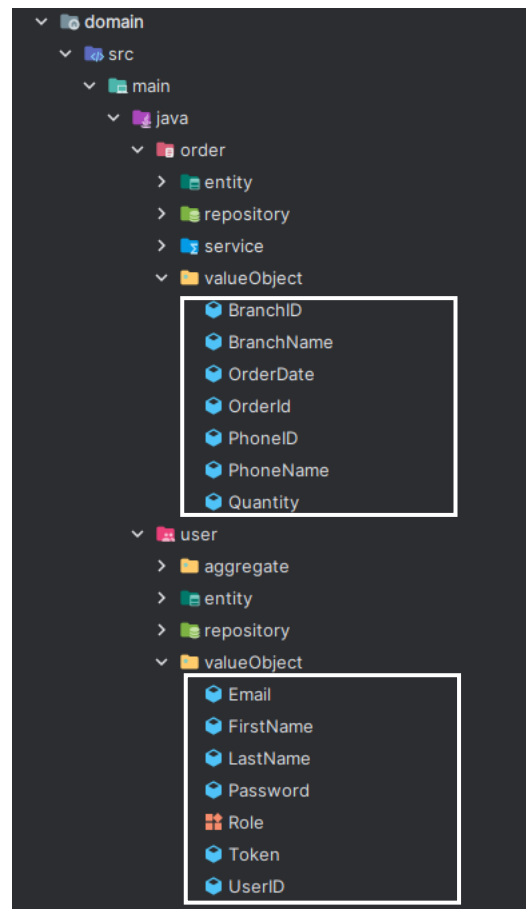


Abbildung 1: Value Objekten

unveränderlich (immutable) gestaltet – das Passwortfeld ist final und wird ausschließlich im Konstruktor initialisiert. Diese Unveränderlichkeit ist ein Kernelement von Wertobjekten, da sie konsistente Zustände garantiert und Nebenwirkungen im Code vermeidet. Ein weiteres zentrales Merkmal ist die wertbasierte Gleichheit: Die überschriebene equals()-Methode stellt sicher, dass zwei Password-Instanzen als gleich gelten, wenn ihre Passwortwerte übereinstimmen – unabhängig von ihrer Objektidentität. Dies entspricht der Semantik von Wertobjekten, bei denen der Vergleich nicht über Referenzen, sondern über den Inhalt erfolgt. Bereits beim Erzeugen eines Password-Objekts erfolgt eine strikte Validierung: Das Passwort darf weder null noch leer sein und muss einem vordefinierten Format entsprechen, das über constants.PASSWORD_RULE definiert ist. Bei ungültigen Eingaben wird eine IOException ausgelöst. Diese Vorgehensweise verhindert die Erstellung invalider Objekte und stellt sicher, dass nur gültige Zustände in der Domäne weiterverwendet werden können.

Die Klasse Password verfolgt eine bewusst fokussierte Verantwortlichkeit: Sie kapselt den Wert, sorgt für Validierung und stellt sicheren Zugriff bereit. Komplexe Geschäftslogik gehört hingegen nicht in ein Wertobjekt – dies würde dem Prinzip der klaren Trennung von Zustandsrepräsentation und Verhalten widersprechen. Darüber hinaus bietet die Klasse eine klare semantische Typisierung. Obwohl sie technisch lediglich einen String kapselt, verleiht sie dem Passwort eine explizite Bedeutung im Domänenkontext. Dadurch entsteht selbst-dokumentierender Code, der Typsicherheit in Methodensignaturen ermöglicht und Sicherheitsaspekte unterstützt, etwa indem verhindert wird, dass unvalidierte oder falsch verwendete Strings als Passwörter eingesetzt werden. Die Kapselung schützt somit vor unvalidierten Werten, unbeabsichtigten Änderungen und semantisch falscher Verwendung. Insgesamt zeigt die Implementierung, wie Value Objects dazu beitragen können, Domänenkonzepte präzise und robust abzubilden. Gleichzeitig fördern sie die Code-Qualität durch Typsicherheit, Validierungsgarantien und eine klare Semantik. Die strikte Trennung zwischen Wertobjekten und Entitäten ist dabei ein zentrales Element einer sauberen, fachlich orientierten Domänenmodellierung.

```
1 package user.valueObject;
2
3 > import ...
7
8 public class Password { 15 usages amjadlnakshbandi
9     private final String password; 5 usages
10
11     public Password(String password) throws IOException { 7 usages amjadlnakshbandi
12         validatePassword(password);
13         this.password = password;
14     }
15
16     private void validatePassword(String password) throws IOException { 1 usage amjadlnakshbandi
17         if (password == null || password.isBlank()) {
18             throw new IOException("Password cannot be null or empty");
19         }
20         if (!password.matches(constants.PASSWORD_RULE)) {
21             throw new IOException("Password does not meet the required format");
22         }
23     }
24
25 > public String getPassword() { return password; }
28
29 @Override amjadlnakshbandi
30 public boolean equals(Object o) {
31     if (o == null || getClass() != o.getClass()) return false;
32     Password other = (Password) o;
33     return Objects.equals(password, other.password);
34 }
35
36 @Override amjadlnakshbandi
37 > public int hashCode() { return Objects.hashCode(password); }
40
41 }
42
```

Abbildung 2: Password Klasse

Entities

In der Domäne ist eine Entität eindeutig identifizierbar. In unserem Projekt gehört auch die Klasse User und Order, die im Paket order.entity implementiert ist, zu den zentralen Entitäten. Sie erfüllt die wesentlichen Merkmale des Entitätskonzepts im Domain-Driven Design.

Die Klasse Order repräsentiert eine bestimmte Bestellung und besitzt eine eindeutige Identität, die durch das Attribut OrderId sichergestellt wird.

Diese ID wird automatisch generiert und bleibt während des gesamten Lebenszyklus der Bestellung unverändert. Damit kann jede Bestellung eindeutig identifiziert und unabhängig von ihren sonstigen Attributwerten unterschieden werden. Darüber hinaus weist die Entität einen potenziell veränderlichen Zustand auf. Auch wenn die aktuelle Implementierung keine expliziten Änderungsmethoden enthält, wird die Bestellung über ein Builder-Muster erstellt, das flexible Initialisierungen zulässt. Attribute wie branchID, phoneID, branchName, phoneName, quantity und orderDate definieren den Zustand einer Bestellung und können in einer erweiterten Version der Anwendung durch entsprechende Methoden verändert oder ergänzt werden, etwa um den Status zu aktualisieren oder Änderungen in der bestellten Menge vorzunehmen. Ein weiteres zentrales Merkmal von Entitäten ist ihr Lebenszyklus. Eine Bestellung wird nicht nur erstellt, sondern durchläuft im realen Geschäftskontext in der Regel mehrere Stadien – zum Beispiel Bearbeitung, Versand, Stornierung oder Abschluss. Auch wenn in der aktuellen Version der Anwendung nur das Erstellen abgebildet ist, lässt die Struktur der Order-Klasse eine Erweiterung um weitere Zustandsübergänge zu. Insgesamt zeigt die Order-Klasse alle relevanten Charakteristika einer Entität im Sinne des Domain-Driven Design. Sie stellt ein eindeutig identifizierbares Objekt innerhalb der Domäne dar, besitzt einen potenziell veränderlichen Zustand und einen fachlich motivierten Lebenszyklus. Die klare Trennung von Identität und Zustandsdaten sowie die Kombination mit passenden Wertobjekten wie OrderId, Quantity oder BranchName sorgt zusätzlich für eine saubere und robuste Modellierung der Domäne.

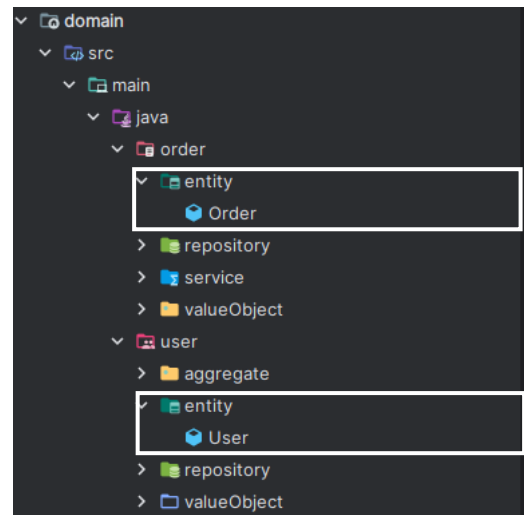


Abbildung 3: Entites

Aggregates

Im Domain-Driven Design stellen Aggregate Gruppen von Entitäten und Value Objects dar, die als logisch zusammengehörige Einheit betrachtet und gemeinsam verwaltet werden. Auch wenn ein Aggregat nur aus einer einzigen Entität besteht, definiert es dennoch eine wichtige strukturelle und fachliche Grenze innerhalb der Domäne. In unserem Projekt wird dieses Konzept unter anderem im Zusammenhang mit der Benutzerverwaltung angewendet. Ein konkretes Beispiel dafür ist die Klasse `Employees`, die im Paket `user.aggregate` implementiert ist und die Entität `User` kapselt.

Die Klasse `Employees` kann als Aggregat betrachtet werden, da sie mehrere charakteristische Eigenschaften eines Aggregats erfüllt. Zunächst fungiert sie als Aggregatwurzel. Das bedeutet, dass sie die zentrale Instanz ist, über die auf die enthaltene Entität – in diesem Fall ein Objekt der Klasse `User` – zugegriffen wird. Das `Employees`-Objekt kontrolliert somit alle Interaktionen mit dem Benutzerobjekt und stellt sicher, dass externe Zugriffe nicht direkt auf die Entität erfolgen, sondern immer über die definierte Wurzel.

Darüber hinaus definiert die Klasse `Employees` eine Konsistenzgrenze. Innerhalb dieser Grenze wird garantiert, dass alle Änderungen am Zustand der enthaltenen Entität kontrolliert und konsistent durchgeführt werden. Das bedeutet konkret, dass jede Änderung am Benutzer – etwa das Setzen eines neuen `User`-Objekts – ausschließlich über das `Employees`-Aggregat erfolgt. Dadurch wird verhindert, dass der Zustand inkonsistent wird oder durch unkoordinierte Zugriffe verändert wird.

Schließlich erfüllt `Employees` auch das Kriterium einer transaktionalen Einheit. In einem erweiterten Anwendungskontext könnten alle Änderungen, die innerhalb dieses Aggregats vorgenommen werden, im Rahmen einer einzigen Transaktion durchgeführt werden – entweder vollständig oder gar nicht. Auch wenn die aktuelle Implementierung noch keine persistente Speicherung oder komplexe Geschäftslogik beinhaltet, bildet `Employees` bereits jetzt eine saubere Grundlage für eine solche Erweiterung.

Insgesamt zeigt die Klasse `Employees`, wie sich das Aggregat-Muster im Domain-Driven Design auch bei einfachen Fallbeispielen sinnvoll einsetzen lässt. Sie stellt die Aggregatwurzel dar, kapselt die enthaltene Entität `User`, definiert eine Konsistenzgrenze und bietet eine Basis für transaktionale Operationen innerhalb der Domänenschicht.

Repositories

Repositories werden benutzt, um festzulegen, wie mit persistierten Entitäten interagiert und wie diese persistiert werden können. Ein Repository ist im Grunde eine Schnittstelle, die verschiedene Operationen ermöglicht, wie zum Beispiel das Erstellen, Lesen, Aktualisieren und Löschen von Entitäten (CRUD-Operationen). Für jede Entität, die persistiert wird, wird ein Repository in Form eines Interfaces in der Domänen-Schicht definiert. Die konkrete Implementierung und Persistierung der Daten erfolgt in der Applikations-Schicht.

Ursprünglich war geplant, einen MySQL-Server zu verwenden, der in Microsoft Azure gehostet wird. Aufgrund finanzieller Einschränkungen wurde dieser Plan jedoch verworfen. Stattdessen wurde entschieden, die Persistierung über lokale CSV-Dateien umzusetzen. Diese Lösung bietet eine einfache, kostengünstige Alternative für die Speicherung von Daten während der Entwicklung und dem Testbetrieb.

Die Nutzung von CSV-Dateien ermöglicht weiterhin eine klare Trennung zwischen den Daten selbst und den Operationen, die auf diesen Daten ausgeführt werden. Das bedeutet, dass Datenbankabhängigkeiten und technische Details der Datenpersistenz vollständig in der Applikations-Schicht gekapselt bleiben, während sich die Domänen-Schicht ausschließlich auf die Geschäftslogik und das Verhalten der Entitäten konzentriert. Diese Trennung trägt wesentlich dazu bei, die Komplexität der Domäne zu reduzieren und eine flexiblere sowie unabhängigere Entwicklung zu fördern.

Domain Services

Domain Services können komplexes Verhalten, Prozesse oder Regeln in der Problem-domäne abbilden, die nicht eindeutig einer bestimmten Entität oder einem bestimmten Wertobjekt zugeordnet werden können. Die Klasse `OrderLimitPolicyService`, die im Paket `Domain-order.service` implementiert ist, stellt ein typisches Beispiel für einen solchen Domain Service im Sinne des Domain-Driven Design dar. Domain Services kommen immer dann zum Einsatz, wenn eine bestimmte fachliche Regel oder ein Geschäftsprozess nicht sinnvoll innerhalb einer einzelnen Entität oder eines Wertobjekts modelliert werden kann, aber dennoch eindeutig zur Domäne gehört. In diesem Fall übernimmt der Service die Aufgabe, eine zentrale Geschäftsregel durchzusetzen: Für jede

Filiale darf die tägliche Gesamtbestellmenge eine bestimmte Grenze – in diesem Fall 100 Einheiten – nicht überschreiten.

Diese Regel ist unabhängig von einer konkreten Bestellung und lässt sich daher nicht direkt in der Order-Entität abbilden. Die Entität kennt nur ihren eigenen Zustand und hat keinen Zugriff auf den gesamten Tageskontext einer Filiale. Der OrderLimitPolicyService hingegen aggregiert die für die Entscheidung notwendige Information, indem er auf die bereits gespeicherten Bestellungen in einer CSV-Datei zugreift, die Mengen summiert und die neue Bestellung in diesem Kontext prüft. Damit übernimmt er eine wichtige fachliche Verantwortung, die für die Integrität des Geschäftsprozesses entscheidend ist. Obwohl zur Umsetzung technische Mittel wie das Einlesen von CSV-Dateien verwendet werden, bleibt der Kern der Methode klar domänengetrieben: Es geht darum, die zulässige Gesamtmenge pro Filiale und Tag zu überprüfen – eine Regel, die direkt aus den Geschäftsanforderungen abgeleitet ist. Genau deshalb ist OrderLimitPolicyService ein Domain Service und kein technischer oder Infrastrukturservice. Charakteristisch für einen Domain Service ist auch, dass er zustandslos arbeitet. Der Service speichert keine eigenen Daten, sondern berechnet bei jedem Aufruf auf Basis externer Informationen (hier: der CSV-Datei), ob eine neue Bestellung zulässig ist. Zudem ist die Verantwortung des Services klar von der Entität Order getrennt. Diese Trennung trägt zu einer sauberen und modularen Struktur bei, in der jede Klasse für eine klar umrissene Aufgabe zuständig ist. Zusammenfassend lässt sich sagen, dass der OrderLimitPolicyService alle Merkmale eines Domain Services erfüllt: Er kapselt eine geschäftsrelevante Regel, die nicht natürlich zu einer bestimmten Entität gehört, ist fachlich motiviert, arbeitet zustandslos und trägt zur Durchsetzung der Geschäftslogik bei, ohne technische Details in die Domänenschicht zu vermischen.

Clean Architecture

Abstraction Code

Die Abstraktionsschicht (abstraction) in diesem Projekt enthält domänenübergreifende Grundbausteine und allgemeine Konzepte, die unabhängig von spezifischen fachlichen Anforderungen in verschiedenen Bereichen der Anwendung genutzt werden können. Ihr Ziel ist es,

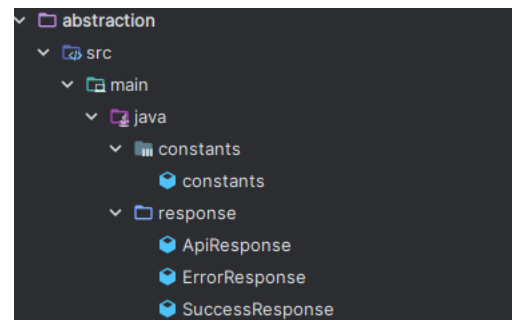


Abbildung 4: Abstraction

allgemeine Schnittstellen, Konstanten und standardisierte Rückgabeformate bereitzustellen, die eine klare Trennung zwischen fachlicher Logik und technischer Infrastruktur ermöglichen. Dies verbesserten die Wartbarkeit, Wiederverwendbarkeit und Testbarkeit des Codes erheblich. Ein zentrales Element dieser Schicht ist die Klasse constants, die zentrale Konstanten wie Dateipfade, Validierungsregeln oder systemweite Parameter definiert. Durch die zentrale Verwaltung dieser Werte an einem Ort wird vermieden, dass sich hartcodierte Strings oder Zahlen im gesamten Code verteilen, was die Anpassung und Pflege wesentlich erleichtert. Ein weiterer wichtiger Bestandteil der Abstraktionsschicht ist das Paket response, das Klassen zur strukturierten Rückgabe von Informationen bereitstellt. Die Klasse ApiResponse bildet dabei die Basis für standardisierte Rückmeldungen in der Anwendung. Sie enthält die beiden Felder status und message und dient als gemeinsamer Rückgabebetyp für Operationen, bei denen eine klare Kommunikation über Erfolg oder Fehler notwendig ist. Die Klasse SuccessResponse<T> erweitert ApiResponse um ein generisches Datenfeld data,

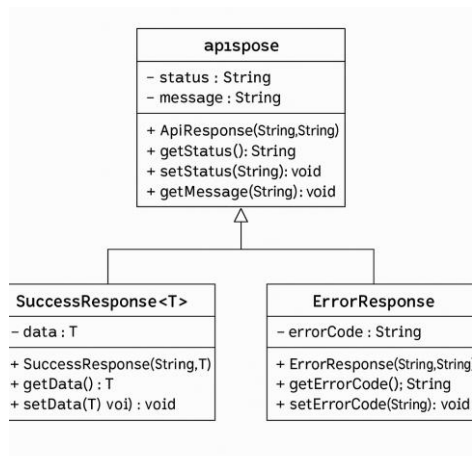


Abbildung 5: UML-Diagramm der ApiResponse-Klasse und ihrer Spezialisierungen

wodurch nicht nur eine Erfolgsmeldung, sondern auch ein konkreter Rückgabewert übermittelt werden kann. Diese Struktur ist besonders nützlich, um Ergebnisse aus Services oder Datenoperationen typisiert zurückzugeben und gleichzeitig eine konsistente Kommunikationsstruktur beizubehalten. Ergänzt wird dieses Konzept durch die Klasse ErrorResponse, die analog aufgebaut ist und eine einheitliche Fehlerdarstellung ermöglicht, etwa durch zusätzliche Informationen zur Fehlerursache

oder einem Fehlercode. Insgesamt trägt die Abstraktionsschicht dazu bei, generische Konzepte wie Konstantenverwaltung und strukturierte Rückgabeformate zentral und unabhängig von der Fachdomäne zu implementieren. Dies fördert eine saubere Trennung der Verantwortlichkeiten innerhalb der Anwendung und sorgt für eine klare, wiederverwendbare und konsistente Architektur.

Domain Code

Die Domänen-Schicht implementiert organisationsweit gültige Geschäftslogik und umfasst Aggregate, also Entitäten und Wertobjekte, sowie Repositories. In der gezeigten Abbildung ist die konkrete Umsetzung dieser Schicht im Projekt dargestellt. Sie ist in zwei zentrale kontextbezogene Bereiche untergliedert: order und user. Jeder dieser

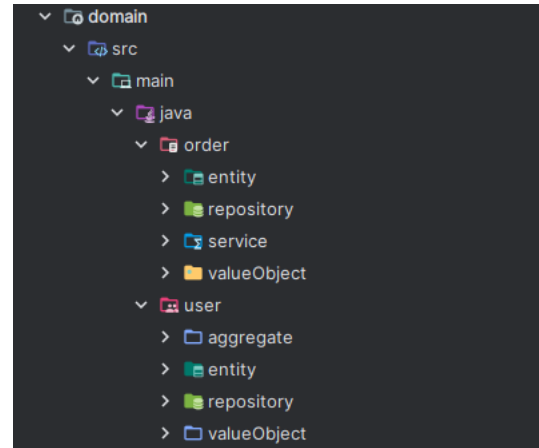


Abbildung 6: Domain Code

Bereiche folgt einer einheitlichen Struktur und enthält jeweils eigene Packages für entity, valueObject und repository. Darüber hinaus existieren in order zusätzliche service-Komponenten und in user ein eigenes aggregate-Package. Diese Aufteilung bringt zahlreiche Vorteile mit sich. Die Trennung der Verantwortlichkeiten zeigt sich in der klaren Gliederung in Entitäten, Wertobjekte und Repositories innerhalb jedes Bereichs. Entitäten wie Order oder User repräsentieren zentrale Objekte mit stabiler Identität, während Wertobjekte wie Quantity, PhoneID oder CustomerName unveränderliche Datenkapselungen darstellen, die bestimmte Eigenschaften eines Objekts beschreiben. Die Repositories dienen dem Zugriff auf persistierte Entitäten und stellen über definierte Schnittstellen sicher, dass die Speicher- und Ladevorgänge vom restlichen Anwendungscode getrennt bleiben. Ein weiterer Vorteil ist die Flexibilität und Erweiterbarkeit. Die Struktur der Packages ist darauf ausgelegt, lose gekoppelte Komponenten über Interfaces bereitzustellen. Diese Schnittstellen können bei Bedarf durch alternative Implementierungen ersetzt oder erweitert werden. Dadurch lassen sich beispielsweise Änderungen an der Persistenzschicht durchführen, ohne dass die Domänenschicht davon betroffen ist. So bleibt die Software offen für Erweiterungen, ohne bestehende Funktionalitäten zu gefährden.

Auch die Testbarkeit wird durch diese Struktur deutlich verbessert. Die konsequente Verwendung von Interfaces in Repositories sowie in anderen Bereichen der Domänenschicht ermöglicht den Einsatz von Mock- oder Fake-Objekten in Unit-Tests. Dadurch lässt sich die Geschäftslogik gezielt isolieren und testen, was die Testabdeckung erhöht und die Qualität des Codes nachhaltig verbessert. Nicht zuletzt sorgen die eingesetzten Interfaces für klare Verträge innerhalb der Anwendung. Entwickler erkennen sofort, welche Methoden von den Komponenten der Domänenschicht bereitgestellt werden und wie sie anzuwenden sind. Dies erleichtert die Zusammenarbeit im Team, reduziert Missverständnisse und sorgt für ein einheitliches Verständnis der Softwarearchitektur. Insgesamt zeigt die dargestellte Struktur, wie die Domänenschicht auf Basis des Domain-Driven Design modular, verständlich und wartungsfreundlich aufgebaut wurde. Sie bildet eine robuste Grundlage für die Abbildung der Geschäftslogik, sorgt für eine klare Trennung der fachlichen Verantwortlichkeiten und schafft gleichzeitig ideale Voraussetzungen für nachhaltige, testbare und flexibel erweiterbare Software.

Application Code

Diese Schicht enthält die Anwendungsfälle:

1. Benutzerregistrierung
2. Mitarbeiter-Logins
3. Mitarbeiter-Ausloggens
4. Handybestellung entgegennehmen
5. Benutzerlisten-Abrufs (exklusiv für Super-Admins)

Die Anwendungsfälle des Projekts sind über Services realisiert. Für jedes Element des Domain Codes, das sich auf der Domain-Schicht wurde ein Service erstellt, der die anwendungsspezifische Geschäftslogik implementiert und die gewünschte Funktionalität ermöglicht

Adapters Code

Diese Schicht vermittelt Aufrufe und Daten an die inneren Schichten und ist für die Formatkonvertierungen zuständig. Im Fall des Projekts wurde aktuell auf diese Schicht verzichtet, da die Domäne und Adapters sehr ähnlich war und ein Mapping für diesen Umfang des Projekts als redundant angesehen wurde.

Plugins

Diese Schicht enthält Frameworks, Datentransportmittel und andere technische Werkzeuge, die zur Integration, Steuerung und Ausführung technischer Prozesse notwendig sind. In diesem Projekt kommt **Spring Boot** als zentrales Backend-Framework zum Einsatz. Es dient der Initialisierung der Anwendung, der Verwaltung von Komponenten sowie der strukturierten Umsetzung technischer Funktionen wie Konfigurationsmanagement, Dependency Injection und dem Routing von Datenflüssen. Durch die Nutzung von Spring Boot wird eine moderne, modulare und erweiterbare Serverarchitektur ermöglicht, die die Trennung von Domäne und Infrastruktur technisch unterstützt. Die Persistierung der Daten erfolgt nicht über eine relationale Datenbank, sondern über das Schreiben und Lesen von CSV-Dateien. Diese Aufgabe wird durch sogenannte RepositoryBridges übernommen. Sie verbinden die Repository-Interfaces der Domänenschicht mit konkreten Implementierungen für die dateibasierte Speicherung. So bleibt die fachliche Logik unabhängig von den technischen Details der Persistenz. Bei Bedarf könnten die Implementierungen leicht gegen alternative Speicherlösungen, etwa eine relationale Datenbank, ausgetauscht werden. Insgesamt ermöglicht die Plugin-Schicht durch den Einsatz von Spring Boot eine saubere Trennung zwischen fachlicher Logik und technischer Umsetzung. Gleichzeitig bleibt die Architektur flexibel, schlank und auf die konkreten Anforderungen des Projekts zugeschnitten – ohne unnötige externe Abhängigkeiten.

Programming Principles

Single Responsibility

Das Single-Responsibility-Prinzip (SRP) besagt, dass eine Klasse genau eine klar abgegrenzte Verantwortung haben sollte. Im vorliegenden Projekt wird dieses Prinzip konsequent auf mehreren Ebenen umgesetzt. Zum einen kommen separate Repository-Interfaces zum Einsatz, die jeweils nur für die Persistenz einer bestimmten Entität zuständig sind. Jedes Interface definiert nur die relevanten Operationen für seine Entität – etwa das Speichern, Laden oder Aktualisieren – und trägt somit ausschließlich die Verantwortung für den Datenzugriff auf ein konkretes Objekt. Diese Trennung erleichtert spätere Änderungen in der Persistenzschicht, da Anpassungen auf eine einzelne Entität beschränkt bleiben, ohne andere Bereiche der Anwendung zu beeinträchtigen. Auch auf Ebene der Anwendungsschicht wird SRP berücksichtigt: Für verschiedene

Anwendungsfälle existieren separate Controller-Klassen. Jede Controller-Klasse ist einem spezifischen Use Case zugeordnet und übernimmt ausschließlich die Steuerung der dazugehörigen Benutzerinteraktion oder Geschäftslogik. Dadurch wird der Code strukturierter, leichter verständlich und einfacher wartbar, da Änderungen am Verhalten einer bestimmten Funktion gezielt an einer Stelle vorgenommen werden können. Ein weiteres Beispiel für die Anwendung des Single-Responsibility-Prinzips ist die konsequente Nutzung von Wertobjekten (Value Objects). Diese kapseln klar definierte Daten wie Quantity oder Password und sind jeweils dafür zuständig, ihren Wert zu repräsentieren und gegebenenfalls zu validieren. Die Validierungslogik ist somit nicht auf andere Klassen verteilt, sondern befindet sich ausschließlich dort, wo sie inhaltlich hingehört. Auch dies führt zu einer klareren Trennung der Verantwortlichkeiten und fördert einen wartbaren, nachvollziehbaren und robusten Code. Insgesamt trägt die konsequente Anwendung des Single-Responsibility-Prinzips dazu bei, die Komplexität der Anwendung zu reduzieren und Änderungen gezielt sowie risikoarm durchführen zu können.

Interface Segregation

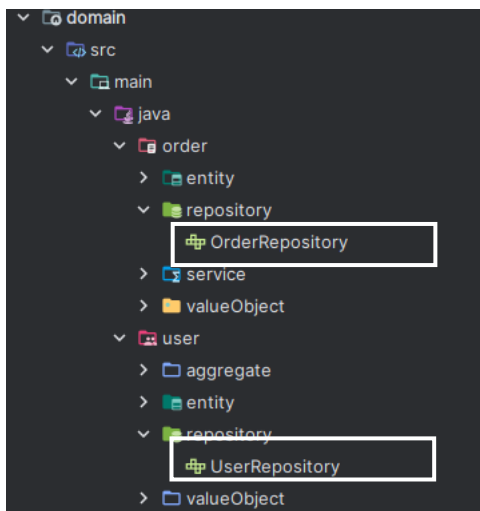


Abbildung 7: Interface Segregation

Das Interface Segregation Principle (ISP) besagt, dass Schnittstellen so gestaltet sein sollten, dass sie jeweils nur auf die spezifischen Bedürfnisse einzelner Clients zugeschnitten sind – anstatt einer umfassenden, generischen Schnittstelle bereitzustellen, die für alle Zwecke dienen soll. Dieses Prinzip wird im vorliegenden Projekt durch die gezielte Verwendung von Repository-Interfaces umgesetzt. Jedes Repository-Interface enthält ausschließlich die Methoden, die für eine

bestimmte Entität relevant sind. Auf diese Weise wird verhindert, dass Klassen gezwungen werden, unnötige oder nicht verwendete Methoden zu implementieren. Stattdessen bleibt jede Schnittstelle fokussiert und übersichtlich. Dies führt zu einer besseren Entkopplung innerhalb des Systems und erhöht die Wartbarkeit und Erweiterbarkeit des Codes. Änderungen können gezielter vorgenommen werden, ohne Auswirkungen auf nicht betroffene Komponenten zu riskieren. Durch die Aufteilung in

schlanke, spezialisierte Interfaces fördert ISP die Flexibilität des Designs und reduziert Abhängigkeiten zwischen den Komponenten. Entitäten oder andere Klassen sind somit nur an die Methoden gebunden, die sie tatsächlich benötigen – was die Wiederverwendbarkeit verbessert und das Risiko von Seiteneffekten bei Änderungen verringert.

DRY

Das DRY-Prinzip (Don't Repeat Yourself) zielt darauf ab, die Wiederholung von Informationen zu vermeiden – insbesondere solcher, die bei Änderungen mehrfach angepasst werden müssten. Ziel ist es, Redundanz im Code zu reduzieren, um Wartungsaufwand, Fehleranfälligkeit und Inkonsistenzen zu minimieren. Im vorliegenden Projekt wird dieses Prinzip unter anderem durch den Einsatz zentraler Abstraktionen und gezielter Strukturierung umgesetzt. Ein konkretes Beispiel hierfür ist die Verwendung der constants-Klasse. Diese Klasse dient als zentrale Stelle zur Verwaltung wiederverwendbarer Konstanten – etwa für Dateipfade, Formatierungsregeln, Konfigurationswerte. Anstatt diese Werte mehrfach im Code zu definieren, werden sie einmalig in constants deklariert und bei Bedarf referenziert. Änderungen, wie etwa ein neuer Dateipfad oder eine geänderte Validierungsregel, müssen somit nur an einer einzigen Stelle vorgenommen werden. Dadurch bleibt der Code konsistent, übersichtlich und weniger fehleranfällig. Neben der constants-Klasse wird das DRY-Prinzip auch durch die Nutzung von Wertobjekten, wiederverwendbaren Service-Klassen oder allgemeinen Hilfsklassen unterstützt. Abstraktionen wie diese sorgen dafür, dass fachliche oder technische Logik nicht mehrfach implementiert wird, sondern zentral gekapselt und wiederverwendbar zur Verfügung steht. So trägt das Projekt zur Einhaltung des DRY-Prinzips bei und verbessert langfristig Wartbarkeit, Klarheit und Erweiterbarkeit des Codes.

KISS

Das KISS-Prinzip („Keep It Simple, Stupid“) besagt, dass Systeme und Methoden so einfach wie möglich gestaltet werden sollen – ohne unnötige Komplexität oder übermäßige Generalisierung. Die Methode getAllEmployeeImplementation(String token) ist ein gelungenes Beispiel für die Anwendung dieses Prinzips im vorliegenden Projekt. Sie erfüllt ihre Aufgabe auf direkte, verständliche und wartbare Weise, ohne sich in komplizierten Strukturen zu verlieren.

Der Aufbau der Methode ist klar und zielgerichtet.

Zunächst wird geprüft, ob der aufrufende Benutzer über ein gültiges Admin-Token verfügt.

Falls nicht, wird frühzeitig eine Ausnahme geworfen – ein typisches Beispiel für **frühes Beenden** ungültiger Fälle, was die Lesbarkeit und den logischen Ablauf des Codes vereinfacht.

Danach wird die CSV-Datei geöffnet und zeilenweise eingelesen. Jede Zeile wird überprüft, und nur Datensätze mit gültigem Rollenwert werden weiterverarbeitet. Fehlerhafte oder unvollständige Einträge sowie unbekannte Rollen werden direkt übersprungen. Dadurch bleibt die Schleife schlank und übersichtlich, ohne tief verschachtelte Bedingungen oder unnötige Zwischenschritte. Die Trennung von Verantwortung innerhalb der Methode ist gut erkennbar: Die Validierung der Rolle erfolgt unabhängig von der Erstellung der Objekte. Gleichzeitig ist die Erzeugung der Employees-Instanzen kompakt und nachvollziehbar gelöst – über einen separaten Methodenaufruf (getEmployees), der die konkreten Werte aus dem CSV-Datensatz übernimmt. Dies erhöht nicht nur die Wiederverwendbarkeit, sondern reduziert auch die Komplexität im Hauptablauf der Methode. Besonders hervorzuheben ist die **Lesbarkeit** des Codes: Jede Anweisung erfüllt einen klaren Zweck und ist leicht nachvollziehbar – auch für Entwicklerinnen oder Entwickler, die den Code zum ersten Mal sehen. Auf überflüssige Abstraktionen oder technische Spielereien wurde bewusst verzichtet, zugunsten eines verständlichen, robusten Kontrollflusses. Insgesamt zeigt die Methode getAllEmployeeImplementation, wie das KISS-Prinzip in der Praxis effektiv umgesetzt werden kann: Die Logik bleibt fokussiert, der Ablauf klar, und die Wartbarkeit hoch – genau das, was gute Softwarearchitektur ausmacht.

```
public List<Employee> getAllEmployeeImplementation(String token) {  
    if (!Constants.SUPER_ADMIN_TOKEN.equals(token)) {  
        throw new SecurityException("Access denied: Only SuperAdmin can view all users.");  
    }  
  
    List<Employee> employeesList = new ArrayList<>();  
  
    try (BufferedReader reader = new BufferedReader(new FileReader(String.valueOf(csv_users_path)))) {  
        reader.readLine(); // Skip header  
        String line;  
        while ((line = reader.readLine()) != null) {  
            String[] data = line.split(",");  
            if (data.length < 6) continue;  
  
            String roleStr = data[5].trim().toUpperCase();  
  
            Role role;  
            try {  
                role = Role.valueOf(roleStr);  
            } catch (IllegalArgumentException e) {  
                continue; // Skip unknown roles  
            }  
  
            if (role != Role.ADMIN && role != Role.MITARBEITER) continue;  
  
            Employee employee = getEmployees(data, role);  
            employeesList.add(employee);  
        }  
    } catch (IOException e) {  
        throw new RuntimeException("Error reading users file", e);  
    }  
  
    return employeesList;  
}
```

Abbildung 8: KISS Beispiel

Jede Zeile wird überprüft, und

nur Datensätze mit gültigem Rollenwert werden weiterverarbeitet. Fehlerhafte oder unvollständige Einträge sowie unbekannte Rollen werden direkt übersprungen. Dadurch bleibt die Schleife schlank und übersichtlich, ohne tief verschachtelte Bedingungen oder unnötige Zwischenschritte. Die Trennung von Verantwortung innerhalb der Methode ist gut erkennbar: Die Validierung der Rolle erfolgt unabhängig von der Erstellung der Objekte. Gleichzeitig ist die Erzeugung der Employees-Instanzen kompakt und nachvollziehbar gelöst – über einen separaten Methodenaufruf (getEmployees), der die konkreten Werte aus dem CSV-Datensatz übernimmt. Dies erhöht nicht nur die Wiederverwendbarkeit, sondern reduziert auch die Komplexität im Hauptablauf der Methode. Besonders hervorzuheben ist die **Lesbarkeit** des Codes: Jede Anweisung erfüllt einen klaren Zweck und ist leicht nachvollziehbar – auch für Entwicklerinnen oder Entwickler, die den Code zum ersten Mal sehen. Auf überflüssige Abstraktionen oder technische Spielereien wurde bewusst verzichtet, zugunsten eines verständlichen, robusten Kontrollflusses. Insgesamt zeigt die Methode getAllEmployeeImplementation, wie das KISS-Prinzip in der Praxis effektiv umgesetzt werden kann: Die Logik bleibt fokussiert, der Ablauf klar, und die Wartbarkeit hoch – genau das, was gute Softwarearchitektur ausmacht.

Information Expert & Creator-Prinzip

Die Klasse Order in der Domänenschicht verwendet das Builder-Entwurfsmuster zur strukturierten Erstellung von Bestellobjekten. Auch wenn sie nicht direkt dem Information-Expert- oder Creator-Prinzip in Reinform entspricht, lassen sich beide Prinzipien gut anhand ihrer Struktur und Funktion nachvollziehen.

Das Information-Expert-Prinzip:

Dieses Prinzip besagt, dass eine Klasse jene Verantwortlichkeiten übernehmen sollte, für die sie über das nötige Wissen verfügt. Die Klasse Order erfüllt dieses Prinzip, indem sie alle relevanten Informationen zu einer Bestellung kapselt: darunter OrderId, BranchID, PhoneID, BranchName, PhoneName, Quantity und OrderDate. Jede dieser Komponenten wird durch ein Wertobjekt repräsentiert und beschreibt einen klar abgegrenzten Aspekt einer Bestellung. Durch die Aggregation dieser Werte kann Order als Informationsexperte fungieren, da sie über alle notwendigen Daten verfügt, um ihre eigene Konsistenz und Identität sicherzustellen. Die Klasse kennt also ihre vollständige Struktur und kann dadurch in fachlicher Hinsicht eigenständig agieren.

Das Creator-Prinzip:

Das Creator-Prinzip besagt, dass eine Klasse die Verantwortung für die Erzeugung von Objekten übernehmen sollte, wenn sie viele Informationen über die zu erstellenden Objekte besitzt oder eng mit ihnen zusammenarbeitet. In der Klasse Order wird dieses Prinzip durch die Verwendung der inneren statischen Builder-Klasse umgesetzt. Diese kapselt die Konstruktionslogik und ermöglicht es, Order-Objekte auf strukturierte und kontrollierte Weise zu erzeugen. Der Builder übernimmt die Zuweisung der einzelnen Wertobjekte und stellt sicher, dass ein Order-Objekt nur dann erstellt wird, wenn alle notwendigen Daten vorhanden und gültig sind. Gleichzeitig fördert diese Trennung der Konstruktionslogik eine klare Verantwortlichkeitsverteilung und entspricht damit auch dem Grundgedanken des Single-Responsibility-Prinzips.

Zusammenfassend lässt sich sagen, dass die Klasse Order zentrale Aspekte sowohl des Information-Expert- als auch des Creator-Prinzips erfüllt. Sie agiert als fachlich zuständige Einheit für Bestellungen und delegiert die Erzeugung ihrer Instanzen an einen

spezialisierten Konstruktor (Builder), was zu einem übersichtlichen, wartbaren und klar verantworteten Code führt.

Refactoring

Refactor 1

„Extract Method für updatedLines und employee zur Verbesserung der Lesbarkeit und Modularität“

Extract Method für updatedLines und employee zur Verbesserung der Lesbarkeit und Modularität Im Rahmen dieses Refactorings wurde eine umfangreiche Methode überarbeitet, indem zwei zentrale Codeabschnitte – die Erzeugung von updatedLines und von employee – in jeweils eigene Methoden ausgelagert wurden. Dies erfolgte durch Anwendung der bewährten Refactoring-Technik Extract Method (auch bekannt als Extract Function). Ziel war es, die Codequalität gezielt zu verbessern, ohne dabei die ursprüngliche Funktionalität zu verändern. Der Hauptgrund für diese Änderung war die Verbesserung der Lesbarkeit. Lange Methoden mit vielen verschachtelten Anweisungen erschweren das Verständnis und die Wartung des Codes. Durch das Herauslösen klar abgegrenzter Teilbereiche in eigene Methoden wird der Hauptcode schlanker und semantisch aussagekräftiger. Darüber hinaus verbessert diese Aufteilung die Modularität, da nun einzelne Teile des Codes wiederverwendet und unabhängig voneinander getestet oder angepasst werden können. Weitere Vorteile dieses Refactorings sind die Reduktion von Duplikationen, die zuvor möglicherweise mehrfach ähnliche Codeblöcke enthielten, sowie eine bessere Testbarkeit und Debugbarkeit. Einzelne Methoden lassen sich leichter isoliert betrachten und analysieren, was insbesondere in Fehlersituationen hilfreich ist. Insgesamt trägt diese Maßnahme dazu bei, den Code wartbarer, verständlicher und zukunftssicherer zu gestalten – ganz im Sinne nachhaltiger Softwareentwicklung.

URL:

<https://github.com/amjadlnakshbandi/EasyLog/commit/a422b81ddccc27a432b9c25f9efce8535e34828c#diff-db6c99b57a345abe0f3326262a26bc0bfcbbef7ba08fdce29df6e51c047b22c4>

Refactor 2

„Einführung von Dependency Injection für OrderLimitPolicyService Hard-Coded Dependency (Tight Coupling)“

Im Rahmen des zweiten Refactorings wurde zur Verbesserung der Flexibilität und Testbarkeit des Codes eine fest verdrahtete Abhängigkeit durch Dependency Injection ersetzt. Konkret wurde die direkte Instanziierung des OrderLimitPolicyService innerhalb der betreffenden Klasse entfernt und stattdessen über den Konstruktor als Parameter übergeben. Diese Maßnahme adressiert das Problem des sogenannten Tight Coupling, also der engen Kopplung zwischen Klassen, bei der eine Klasse selbst für die Erstellung ihrer Abhängigkeiten verantwortlich ist. Eine solche Kopplung erschwert sowohl die Wartung als auch die Wiederverwendbarkeit und behindert insbesondere das Testen von Komponenten in Isolation. Durch das Aufbrechen dieser Kopplung ergeben sich mehrere Vorteile. Zum einen wird die Flexibilität erhöht, da die konkrete Implementierung der Abhängigkeit zur Laufzeit ausgetauscht oder konfiguriert werden kann – beispielsweise durch alternative Implementierungen für verschiedene Anwendungsfälle. Zum anderen verbessert sich die Testbarkeit, da im Testkontext gezielt Mocks oder Stubs eingesetzt werden können, um das Verhalten der abhängigen Klasse kontrolliert zu überprüfen. Darüber hinaus fördert dieser Schritt die Einhaltung der Prinzipien der Clean Architecture. Die Klasse ist nun nicht mehr selbst für die Erstellung ihrer Abhängigkeiten verantwortlich, sondern erhält diese von außen – ein zentrales Merkmal des Dependency Inversion Principle (DIP), welches besagt, dass hohe Module nicht von konkreten Implementierungen abhängig sein sollten. Schließlich trägt diese Änderung auch zur Wartbarkeit und Erweiterbarkeit des Codes bei. Zukünftige Anpassungen an der konkreten Implementierung des OrderLimitPolicyService erfordern keine Änderungen an der aufrufenden Klasse, was die langfristige Pflege und Weiterentwicklung deutlich erleichtert. Die Funktionalität der Anwendung bleibt durch dieses Refactoring vollständig erhalten. Der Code ist nun jedoch modularer, entkoppelter und zukunftssicherer strukturiert – ein wichtiger Schritt in Richtung nachhaltiger Softwarequalität.

URL:

<https://github.com/amjadalnakhshbandi/EasyLog/commit/b21dc7a16eb336cdbab853f0e1ebe3172c58d029#diff-ccd27c5f997e107a6996d301e6e22f02680232a9b91be9153a8bc77fd724aa3a>

Entwurfsmuster

In meinem Projekt zur Verwaltung von Bestellungen habe ich das Builder-Entwurfsmuster verwendet, um Instanzen der Klasse Order zu erzeugen. Der Einsatz dieses Musters ermöglicht es, komplexe Objekte schrittweise, flexibel und übersichtlich zusammenzusetzen – insbesondere dann, wenn viele Parameter beteiligt sind oder nicht alle Attribute zwingend gesetzt werden müssen.

Ein praktisches Beispiel hierfür ist die Erstellung einer Order-Instanz, die wie folgt durchgeführt wird:

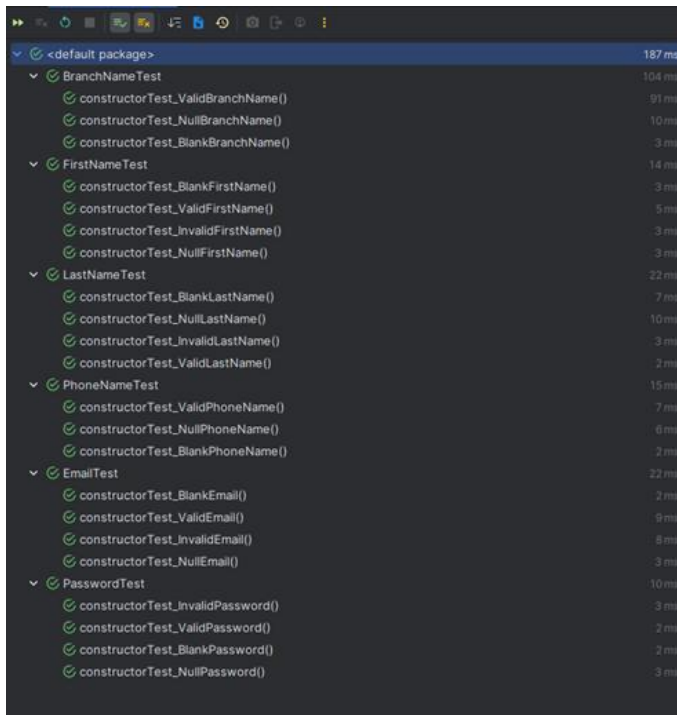
```
Order order = new Order.Builder()
    .phoneID(new PhoneID(dto.getPhoneId()))
    .phoneName(new PhoneName(dto.getPhoneName()))
    .branchID(new BranchID(dto.getBranchId()))
    .branchName(new BranchName(dto.getBranchName()))
    .quantity(new Quantity(dto.getQuantity()))
    .build();
```

Abbildung 9. Erstellung von einer Order

Durch den Einsatz des Builders kann die Initialisierung der Order-Instanz auf eine klare und strukturierte Weise erfolgen. Anstatt alle Parameter in einem langen Konstruktor aufzurufen, werden die einzelnen Attribute explizit über method chaining gesetzt. Das erhöht nicht nur die Lesbarkeit, sondern macht den Code auch wartbarer und erweiterbar. Ein wesentlicher Vorteil des Builder-Musters liegt in seiner Flexibilität: Es ist nicht notwendig, alle möglichen Attribute einer Bestellung bei der Erstellung zu definieren. Stattdessen können nur die tatsächlich relevanten Werte gesetzt werden, während andere intern mit Standardwerten belegt werden können. Dies ist besonders nützlich, um unterschiedliche Anwendungsfälle ohne zusätzliche Konstruktorlogik abzubilden. Darüber hinaus trägt das Muster dazu bei, die Kohärenz und Kapselung im Code zu fördern. Die Erzeugung der Objekte bleibt übersichtlich, und Änderungen an der Order-Struktur – etwa das Hinzufügen neuer Felder – können leicht durch Ergänzung entsprechender Setter-Methoden im Builder vorgenommen werden, ohne dass der bestehende Anwendungscode angepasst werden muss. Insgesamt unterstützt das Builder-Entwurfsmuster in diesem Projekt eine saubere, nachvollziehbare und skalierbare Art der Objekterstellung, die ideal zu komplexen Domänenobjekten wie Order passt.

Testen

Value Objekten



<default package>	187 ms
BranchNameTest	104 ms
constructorTest_ValidBranchName()	91 ms
constructorTest_NullBranchName()	10 ms
constructorTest_BlankBranchName()	3 ms
FirstNameTest	14 ms
constructorTest_BlankFirstName()	3 ms
constructorTest_ValidFirstName()	5 ms
constructorTest_InvalidFirstName()	3 ms
constructorTest_NullFirstName()	3 ms
LastNameTest	22 ms
constructorTest_BlankLastName()	7 ms
constructorTest_NullLastName()	10 ms
constructorTest_InvalidLastName()	3 ms
constructorTest_ValidLastName()	2 ms
PhoneNameTest	15 ms
constructorTest_ValidPhoneName()	7 ms
constructorTest_NullPhoneName()	6 ms
constructorTest_BlankPhoneName()	2 ms
EmailTest	22 ms
constructorTest_BlankEmail()	2 ms
constructorTest_ValidEmail()	5 ms
constructorTest_InvalidEmail()	8 ms
constructorTest_NullEmail()	3 ms
PasswordTest	10 ms
constructorTest_InvalidPassword()	3 ms
constructorTest_ValidPassword()	2 ms
constructorTest_BlankPassword()	3 ms
constructorTest_NullPassword()	3 ms

Abbildung 10: Erfolgreiche Test für Value Objekten

Die Entscheidung, für Value Objects wie `FirstName`, `LastName`, `Email`, `PhoneName`, `BranchName` und `Password` gezielte Tests zu schreiben – wie auf dem Bild zu sehen – ist äußerst sinnvoll und entspricht bewährten Prinzipien der objektorientierten und domänengetriebenen Softwareentwicklung. Value Objects sind nicht bloß Datencontainer, sondern tragen oft fachliche Verantwortung, etwa durch Validierungsregeln, die sicherstellen,

dass nur korrekte und sinnvolle Werte im System verwendet werden. Durch das Testen dieser Objekte kann sichergestellt werden, dass diese Regeln korrekt implementiert und konsequent durchgesetzt werden. Insbesondere Tests wie `constructorTest_NullEmail()` oder `constructorTest_InvalidPassword()` helfen dabei, ungültige Eingaben frühzeitig zu erkennen und zu verhindern, dass fehlerhafte Daten überhaupt ins System gelangen. Das erhöht die Robustheit der Anwendung und reduziert das Risiko von Laufzeitfehlern oder unerwartetem Verhalten. Ein weiterer Vorteil liegt in der Testabdeckung von Grenzfällen. Spezielle Testfälle für leere Strings, null-Werte oder ungültige Formate stellen sicher, dass die Objekte auch in Ausnahmesituationen stabil reagieren. Gleichzeitig dokumentieren diese Tests das erwartete Verhalten der Objekte und dienen als Referenz für andere Entwickler. Sie machen klar nachvollziehbar, welche Eingaben erlaubt sind und welche nicht. Darüber hinaus verbessern solche Tests die Wartbarkeit des Codes erheblich. Sollte sich beispielsweise die Geschäftslogik für Passwörter oder E-Mail-Adressen ändern, geben bestehende Tests sofort Rückmeldung, ob die Änderungen korrekt umgesetzt wurden oder bestehende Funktionalität versehentlich beeinträchtigt wurde. Das schafft Vertrauen beim Refactoring und sorgt für eine nachhaltige Codebasis. Nicht

zuletzt stärken diese Tests auch die domänentreue Modellierung, denn Value Objects sind zentrale Bausteine eines sauberen Domänenmodells. Durch systematisches Testen wird ihre Bedeutung unterstrichen und ihre fachliche Konsistenz gewahrt. Insgesamt trägt die Entscheidung, Value Objects separat zu testen, maßgeblich zur Qualität, Stabilität und Verständlichkeit der Anwendung bei.

Controller

Um die Funktionalität des OrderController isoliert und zuverlässig zu testen, wurde ein gezielter Unit-Test mit Hilfe von Spring Boot's `@WebMvcTest`-Annotation und dem Framework Mockito erstellt. Ziel dieses Tests ist es, die Verarbeitungslogik der HTTP-Anfragen zu prüfen, ohne dabei von der tatsächlichen Implementierung der Persistenzschicht abhängig zu sein.

Test Case	Duration
<default package>	700 ms
OrderControllerTest	700 ms
shouldReturnBadRequestWhenAuthHeaderMissing()	592 ms
shouldCreateOrderSuccessfully()	108 ms

Abbildung 11: Erfolgreiche Testen für Order Controller

Dabei wurde das Interface `OrderRepositoryBridge`, das normalerweise für das Speichern von Bestellungen verantwortlich ist, durch ein Mock-Objekt ersetzt. Diese Technik wird im Rahmen von Dependency Injection über eine spezielle Testkonfiguration (`@TestConfiguration`) realisiert. So kann gezielt kontrolliert werden, wie sich der Controller in bestimmten Situationen verhält, ohne dass reale Daten geschrieben oder externe Abhängigkeiten angesprochen werden. Durch diese Mock-basierte Isolation ergeben sich mehrere Vorteile: Zum einen wird die Testbarkeit stark verbessert, da sich der Test ausschließlich auf das Verhalten des Controllers konzentriert – also auf die HTTP-Kommunikation, Request-Handling und Response-Generierung. Zum anderen erhöht sich die Zuverlässigkeit und Stabilität der Tests, da sie unabhängig von Datenquellen wie Dateien oder Datenbanken ablaufen. Ein Beispiel dafür ist der Test `shouldCreateOrderSuccessfully()`, der überprüft, ob eine korrekt formulierte Bestellung inklusive Authentifizierungs-Header erfolgreich verarbeitet wird und der Controller erwartungsgemäß den HTTP-Status 201 Created mit der entsprechenden Erfolgsmeldung zurückliefert. Die tatsächliche Ausführung der `addOrder()`-Methode wird dabei durch `doNothing().when(orderRepositoryBridge).addOrder(...)` simuliert, um nur die Controller-

Logik zu validieren. Zusätzlich wird mit dem Test `shouldReturnBadRequestWhenAuthHeaderMissing()` ein wichtiger Fehlerfall abgedeckt: Wenn kein Autorisierungs-Header gesetzt ist, gibt der Controller korrekt einen HTTP-Status 400 Bad Request zurück, inklusive einer klaren Fehlernachricht mit dem `errorCode` "Security Error". Die Nutzung von `MockMvc` in Kombination mit `ObjectMapper` erlaubt es außerdem, die JSON-basierte Kommunikation realistisch nachzubilden und die Serverseite vollständig zu simulieren. Dies ist besonders wichtig bei REST-APIs, da hier nicht nur interne Logik, sondern auch die externe Schnittstelle getestet wird. Insgesamt trägt dieser Test wesentlich zur Sicherheit, Wartbarkeit und Qualitätssicherung des Projekts bei. Er validiert, dass der Controller korrekt mit eingehenden Daten umgeht, die erwarteten Statuscodes und Nachrichten zurückliefert und sich auch in Fehlerfällen wie erwartet verhält – und das alles unabhängig von der konkreten Implementierung der darunterliegenden Schichten.

Abbildungsverzeichnis

Abbildung 1: Value Objekten	6
Abbildung 2: Password Klasse	7
Abbildung 3: Entites	8
Abbildung 4: Abstraction	12
Abbildung 5: UML-Diagramm der ApiResponse-Klasse und ihrer Spezialisierungen	12
Abbildung 6: Domain Code	13
Abbildung 7: Interface Segregation	16
Abbildung 8: KISS Beispiel	18
Abbildung 9. Erstellung von einer Order	22
Abbildung 10: Erfolgreiche Test für Value Objekten	23
Abbildung 11: Erfolgreiche Testen für Order Controller	24