# Programmentwurf IPRINTER Customer Service Manager

# SCHRIFTLICHE DOKUMENTATION

für die Prüfung zum

Bachelor of Science

des Studienganges Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

Von

**Mohamad Naser Alnakshbandi**

| | |
|---|---|
| Student: | Mohamad Naser Alnakshbandi |
| Kurs: | TINF20B4 |
| Dozent: | Mirko Dostmann |
| Abgabedatum: | 19.03.2024 |

Repository: https://github.com/Mohmad-Naser-alnakeshbandi/ASE2

# Inhaltsverzeichnis

# Use Cases

Das Projekt ist eine Java Desktop Anwendung, die ein Customer Service Manager darstellt. Folgende Use Cases wurden definiert:

- **Neues Beschwerdeticket erstellen**: Als Benutzer kann ich ein neues Beschwerdeticket erstellen. Das Ticket muss einem Kunden zugeordnet sein. Existiert der Kundenname nicht, wird die Erstellung des Tickets verweigert. Hat der Kunde kein Servicevertrag für die jeweiligen Drucker, kann er für es keine Beschwerde haben. Bei der Verweigerung der Erstellung einer Beschwerde, soll auch eine Erklärung für der Kunden entstehen.
- **Report über einen Kunden erstellen**: Als Benutzer kann ich einen Report über einen Kunden erstellen, der angibt, wie viele Beschwerden er hat in einen bestimmten Zeitraum.
- **Report über einen Drucker erstellen**: Als Benutzer kann ich einen Report über einen Drucker erstellen, der angibt, wie oft das Unternehmen Beschwerden über diesen Drucker erhalten hat und bei welchen Kunden es waren.
- **Wöchentlicher Report erstellen**: Als Benutzer kann ich einen wöchentlichen Report erstellen, der die Entwicklung der erhaltenen Beschwerden des Unternehmens und die auch betroffenen Kunden zeigt. Der Tag mit dem maximalen und die minimale Anzahl von Beschwerden, werden durch eine Farbe kennen gezeichnet.
- **Report Exportieren:** Als Benutzer kann ich den Report über einen Kunden, Report über einen Drucker und das Wöchentlicher Report, den ich erstell habe, es auch Exportieren und es als CSV-Datei speichern.
- **Report über alle Beschwerden erstellen:** Als Benutzer kann ich einen Report für all vorhanden Beschwerden in dem System tabellarisch sehen.

Alle Use Cases wurden erfolgreich im Lauf des Projekts umgesetzt.

# Domain Driven Design

## Die Ubiquitous Language

Die Ubiquitous Language ist das gemeinsame Vokabular von Domänenexperten und Entwicklern. Durch die Verwendung der gleichen Begriffe in der Domäne und im Sourcecode soll sichergestellt werden, dass eine einheitliche Sprache verwendet wird.

## Analyse der Ubiquitous Language

Da das Projekt von Anfang an international ausgerichtet ist, wird Englisch als Sprache gewählt. Im folgenden Abschnitt werden die Bedeutungen der verwendeten Begriffe definiert.

- Complaints: Sammlung aller Beschwerden Tickets.
- Complaint: Eine Beschwerde Ticket.
- Customer Report: Sammlung aller Beschwerden Tickets für einen bestimmten Kunden in einen bestimmten Zeitraum.
- Printer Report: Sammlung aller Beschwerden Tickets für einen bestimmten Drucker.
- Weekly Report: Sammlung aller Beschwerden Tickets für einen bestimmten Woche ein Jahr.
- Used Colors: Sammlung alle Farben, die in der Applikation verwendet sind.
- Constants: Sammlung alle Konstanten, die in der Applikation verwendet sind.
- Success: Bezeichnung von einem erfolgreichen Prozess.
- Common: Sammlung von gemeinsamen Geschäftslogik.

## 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?

Das können wir einfach an „ComplaintDescription“ Klasse sehen:

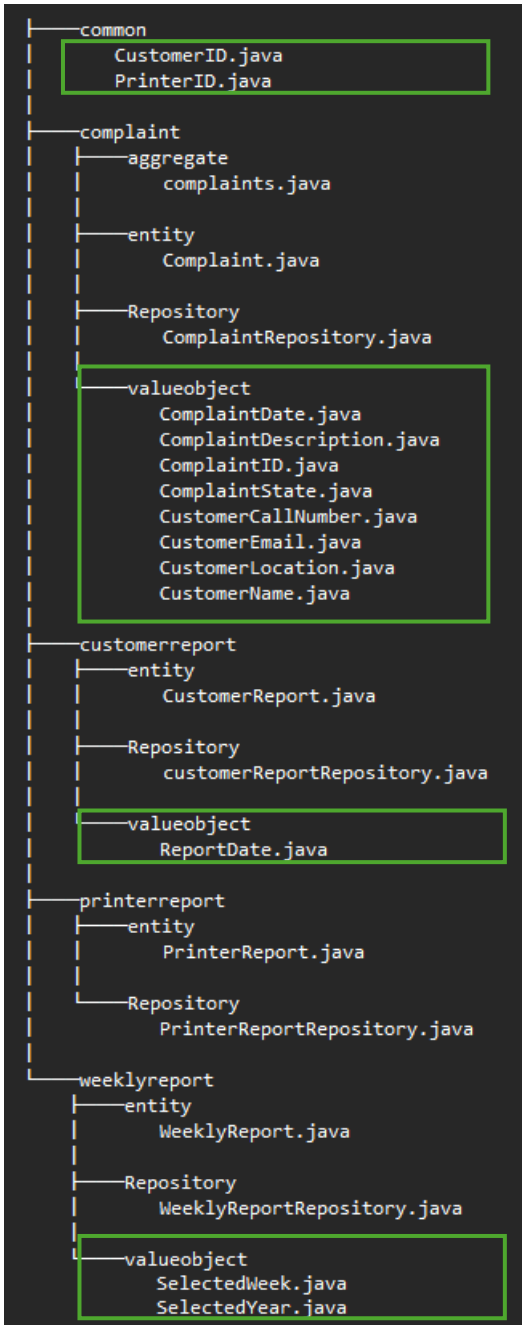

*Abbildung 1: Wertobjekte*

1. **Unveränderlicher Zustand**: Die Klasse hat nur private endgültige Felder (title und body), die über den Konstruktor gesetzt werden und danach nicht mehr geändert werden können. Diese Unveränderlichkeit stellt sicher, dass der Zustand einer ComplaintDescription-Instanz nach der Erstellung nicht mehr geändert werden kann.
2. Gleichheit basierend auf Wert: Die Methode equals() ist implementiert, um die title- und body-Attribute zweier ComplaintDescription-Instanzen zu vergleichen und sicherzustellen, dass die Gleichheit auf ihren Werten und nicht auf ihrer Identität beruht.
3. Irrelevanz des Verhaltens: Die Klasse enthält kein komplexes Verhalten jenseits von Konstruktion, Validierung und Repräsentation. Sie kapselt hauptsächlich den Zustand einer Beschreibung einer Beschwerde (Titel und Text) und bietet Methoden zum Zugriff und zur Validierung dieser Daten.
4. Kapselung von zusammenhängenden Attributen: Die Klasse ComplaintDescription kapselt zusammenhängende Attribute (title und body), um sicherzustellen, dass sie immer als eine einzige Einheit behandelt werden. Dies hilft, die Konsistenz zu gewährleisten und zu vermeiden, dass diese Attribute über verschiedene Teile des Code-Basissystems verteilt werden.

5. Wertsemantik: Instanzen von ComplaintDescription werden basierend auf ihrem Titel und ihrem Text verglichen, was darauf hinweist, dass die Wertsemantik signifikant ist. Zwei Instanzen mit demselben Titel und Text sollten als gleich betrachtet werden, unabhängig von ihrer Referenz.

Ähnliche Überlegungen treffen wir auch bei allen anderen Wertobjekten, jedoch sollten zwei Wertobjekte genauer betrachtet werden: die CustomerID und der CustomerName. Warum werden sie nicht zusammengeführt? Das liegt daran, dass beim Use Case "**Report über einen Kunden erstellen**" der Bericht lediglich über die CustomerID und den Zeitraum erstellt werden muss, während der CustomerName damit nichts zu tun hat.

## Entities

In der Domäne ist eine Entität eindeutig identifizierbar. Wir haben in unserem Projekt 4 Entitäten, die in der Abbildung danebenzusehen sind. Ein Beispiel dafür, warum diese als Entitäten betrachtet werden können, findet sich im Complaint.

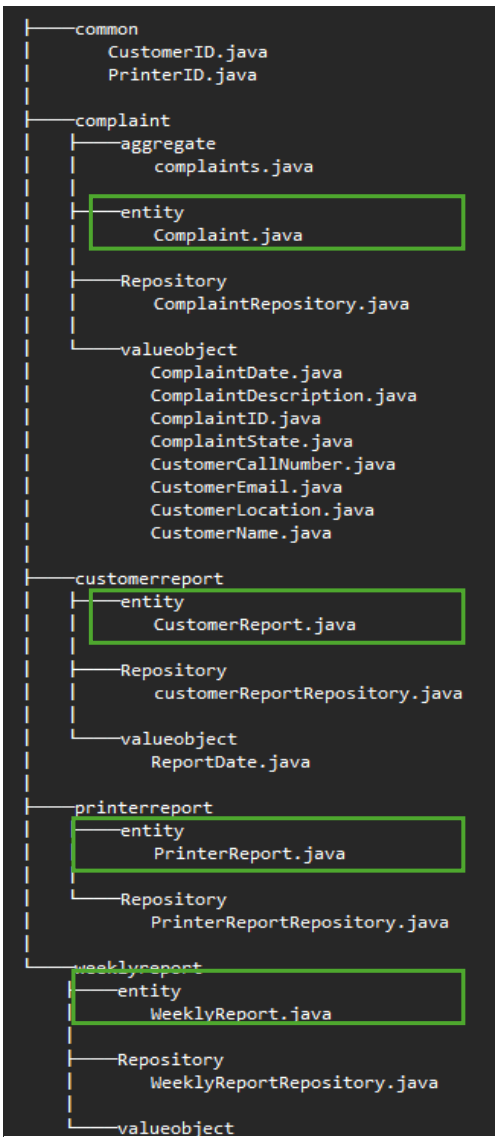

*Abbildung 2: Entities*

Die Klasse Complaint kann als Entität betrachtet werden. Hier sind die Gründe dafür:

1. Identität: Eine Entität repräsentiert ein eindeutiges Objekt innerhalb der Domäne, identifiziert durch eine eindeutige Identität. In diesem Fall repräsentiert eine Complaint-Entität eine bestimmte Beschwerde, die durch ihre Beschwerde-ID (ComplaintID) eindeutig identifiziert werden kann.
2. Veränderlicher Zustand: Entitäten haben in der Regel einen veränderlichen Zustand, was bedeutet, dass ihre Attribute sich im Laufe der Zeit ändern können. In diesem

Fall können sich die Attribute einer Beschwerde, wie der Beschreibung, des Kundenstandorts oder des Beschwerdedatums, ändern, wenn sich der Status der Beschwerde ändert oder neue Informationen verfügbar werden.

3. Lebenszyklus: Entitäten haben einen Lebenszyklus, der mehrere Zustände und Interaktionen innerhalb der Domäne umfasst. Eine Complaint-Entität kann erstellt, aktualisiert, geschlossen oder erneut geöffnet werden, was den typischen Lebenszyklus einer Beschwerde darstellt. Implementiert wird aber jetzt nur das Erstellen von einer Beschwerde.

Insgesamt erfüllt die Klasse Complaint die Charakteristika einer Entität im Domain-Driven Design, indem sie ein eindeutiges Objekt innerhalb der Domäne mit Identität, veränderlichem Zustand, Lebenszyklus und möglicherweise zugehörigem Verhalten darstellt.

## Aggregates

Aggregates sind Gruppen von Entitäten und Value Objects, die zu gemeinsam verwalteten Einheiten zusammengefasst werden. Jede Entität gehört zu einem Aggregat, auch wenn das Aggregat nur aus dieser einen Entität besteht. Aggregate werden im Fall des Projekts beispielerweise für „Complaints“ verwendet, bei den eine Entität „Complaint“ verwaltet wird.  Warum das aber als Aggreagte betrachten werden soll, ist:

1. Aggregatwurzel: Ein Aggregat hat eine Aggregatwurzel, die die gesamte Aggregatstruktur kontrolliert und den Zugriff auf die darin enthaltenen Entitäten ermöglicht. In diesem Fall ist complaints die Aggregatwurzel, da sie die einzige Komponente des Aggregats ist und den Zugriff auf das enthaltene Complaint-Objekt ermöglicht.
2. Konsistenzgrenze: Aggregatwurzeln definieren eine Konsistenzgrenze, innerhalb derer die Konsistenz der enthaltenen Entitäten gewährleistet ist. Änderungen an den enthaltenen Entitäten erfolgen über die Aggregatwurzel, um die Konsistenz sicherzustellen. In diesem Fall erfolgen alle Operationen an der Beschwerde über das complaints-Objekt, was die Konsistenz der Beschwerde innerhalb des Aggregats sicherstellt.

3. Transaktionale Einheit: Aggregatwurzeln werden oft als transaktionale Einheit behandelt, was bedeutet, dass Operationen auf dem Aggregat entweder vollständig durchgeführt werden oder fehlschlagen. In diesem Fall würde jede Operation auf dem complaints-Objekt atomar sein, um die Integrität des Aggregats zu erhalten.

Aufgrund dieser Merkmale entspricht die Klasse complaints der Definition eines Aggregats im Domain-Driven Design, indem sie eine Aggregatwurzel darstellt, die die Konsistenzgrenze für die enthaltene Complaint-Entität definiert und als transaktionale Einheit behandelt werden kann.

## 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 auf dem Domain-Schicht erstellt. Die konkrete Implementierung und Persistierung der Daten erfolgt in der Applikation-Schicht mithilfe von CSV-Dateien. Dies ermöglicht eine klare Trennung zwischen den Daten selbst und den Operationen, die auf diesen Daten ausgeführt werden. Dies bedeutet, dass die Datenbankabhängigkeiten und die technischen Details der Datenpersistenz in der Applikation-Schicht behandelt werden, während die Domänen-Schicht sich auf die Geschäftslogik und das Verhalten der Entitäten konzentriert. Diese Trennung hilft dabei, die Komplexität der Domäne zu reduzieren und ermöglicht eine flexiblere und unabhängigere Entwicklung.

## Domain Services

Domain Services können komplexes Verhalten, Prozesse oder Regeln in der Problemdomäne abbilden, die nicht eindeutig einer bestimmten Entität oder einem bestimmten Wertobjekt zugeordnet werden können. Die Servises befinden sich im Fall des Projekts auf der Application-Schicht und realisieren die Use Cases des Projekts. Repositories werden von Domain Services genutzt, um Daten zu speichern oder mit bereits gespeicherten Daten zu interagieren.

# Clean Architecture

## Abstraction Code

Abstraktion Schicht enthält domänenübergreifende Grundbausteine und allgemeine Konzepte und Algorithmen. Der Zweck der Abstraktionsschicht besteht darin, Schnittstellen und abstrakte Klassen bereitzustellen, die Verträge und Abstraktionen für die Interaktion mit externen Abhängigkeiten oder Frameworks definieren. Im Projekt wurde diese Schicht durch Klassen:

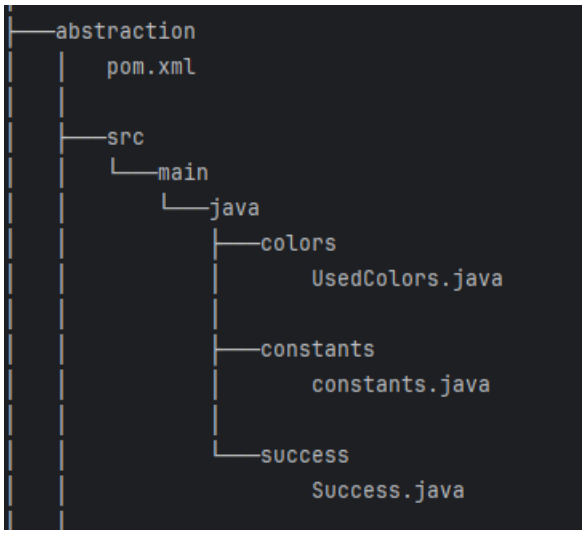


*Abbildung 3: Abstraktion Code*

UsedColors: Diese Klasse definiert eine Reihe von Farbkonstanten, die in der Benutzeroberfläche verwendet werden können. Durch die Verwendung von konkreten Farbwerten als statische Konstanten anstatt direkter Verweise auf Farbobjekte ermöglicht diese Klasse eine zentrale Verwaltung und Konsistenz der Farben in der Anwendung. Dies fördert auch die Wartbarkeit und erleichtert die Anpassung des Erscheinungsbildes der Anwendung, da Farben an einer einzigen Stelle festgelegt werden.

constants: Diese Klasse definiert verschiedene Konstanten, die in der Anwendung verwendet werden, wie z.B. den Namen des UI-Themas, Titel für wöchentliche Berichte, Dateipfade für Datenquellen usw. Durch die Zentralisierung solcher Konstanten in einer separaten Klasse wird die Wartbarkeit und Lesbarkeit des Codes verbessert, da Änderungen an diesen Werten an einer einzigen Stelle vorgenommen werden können.

Success: Diese Klasse stellt eine Abstraktion für die Anzeige von Erfolgsmeldungen in einem Dialogfenster dar. Indem sie die JOptionPane-Klasse von Swing kapselt und eine Methode bereitstellt, um eine Erfolgsmeldung anzuzeigen, abstrahiert sie die Details der Benutzeroberfläche und erleichtert die Wiederverwendbarkeit und Testbarkeit des Erfolgsmanagements in der Anwendung.

Insgesamt dienen diese Klassen als Bausteine der Abstraktionsschicht, indem sie Schnittstellen und abstrakte Konzepte bereitstellen, die Verträge und Abstraktionen für verschiedene Aspekte der Anwendung definieren. Sie ermöglichen eine klarere Trennung zwischen domänenübergreifenden Grundbausteinen und der eigentlichen Implementierung der Anwendung, was die Wartbarkeit, Erweiterbarkeit und Testbarkeit des Codes verbesserten.

## Domain Code

Die Domänen-Schicht implementiert organisationsweit gültige Geschäftslogik und umfasst Aggregate (Entitäten und Wertobjekte) sowie Repositories. Innerhalb dieser Schicht sind die Packages Common, Complaint, Customer Report, Printer Report und Weekly Report enthalten, jeweils unterteilt in Entität, Wertobjekt und Repository in Form von Interfaces. Dieser Treenung bringt uns viele Vorteile wie:

*Abbildung 4: Domain Code*

1. Trennung der Verantwortlichkeiten: Die Aufteilung in Entität, Wertobjekt und Repository spiegelt die Trennung der Verantwortlichkeiten innerhalb der Domänen-Schicht wider. Jedes dieser Konzepte hat eine spezifische Rolle in der Geschäftslogik: Entitäten repräsentieren die Kernkonzepte der Domäne, Wertobjekte kapseln gemeinsame Datenstrukturen und Repositories bieten Zugriff auf persistente Daten.
2. Flexibilität und Erweiterbarkeit: Indem die Struktur der Packages darauf abzielt, Interfaces für Entitäten, Wertobjekte und

Repositories bereitzustellen, wird die Flexibilität und Erweiterbarkeit des Codes verbessert. Interfaces ermöglichen es, verschiedene Implementierungen auszutauschen oder hinzuzufügen, was die Anpassung an zukünftige Anforderungen erleichtert.

3. Testbarkeit: Die Verwendung von Interfaces für Entitäten, Wertobjekte und Repositories fördert die Testbarkeit des Codes. Durch die Definition von Interfaces können Mock-Implementierungen für Tests erstellt werden, was die Isolierung von Unit-Tests erleichtert und die Testabdeckung erhöht.
4. Klare Verträge: Die Verwendung von Interfaces definiert klare Verträge für die Interaktion mit Entitäten, Wertobjekten und Repositories. Dies erleichtert die Zusammenarbeit im Team, da Entwickler genau wissen, welche Methoden und Funktionen von den verschiedenen Teilen der Domänen-Schicht erwartet werden.

## Application Code

Diese Schicht enthält die Anwendungsfälle:

1. Neues Beschwerdeticket erstellen.
2. Report über einen Kunden erstellen.
3. Report über einen Drucker erstellen.
4. Wöchentlicher Report erstellen.
5. Report Exportieren.
6. Report über alle Beschwerden erstellen

Die Anwändungsfälle 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

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 angesen wurde.

### Plugins

Diese Schicht enthält Frameworks, Datentransportmittel und andere Werkzeuge. Im Projekt kommen keine spezifischen Frameworks im Einsatz. Stattdessen erfolgt die Datenpersistenz durch das Schreiben in CSV-Dateien, wobei diese Funktionalität durch den RepositoryBridges implementiert ist. RepositoryBridges rufen Implementierungen von Repository-Interfaces, die Teil der Domänen-Schicht sind, und ermöglichen den Zugriff auf die persistierten Daten. Zusätzlich zur Datenpersistenz ist auf der Plugin-Schicht auch eine GUI vorhanden, die mit Java Swing umgesetzt wurde. Sie ermöglicht die Interaktion mit dem Benutzer und stellt die Ergebnisse der Datenverarbeitung dar. Es werden keine externen Frameworks verwendet. Die Implementierung erfolgt spezifisch für das Projekt. Dadurch erzielt man die Flexibilität, Funktionalitäten genau an die gegebenen Anforderungen anzupassen und auf die Verwendung zusätzlicher Frameworks zu verzichten.

## Programming Principles

### Single Responsibility

Im vorliegenden Projekt wird das Single-Responsibility-Prinzip auf verschiedene Weisen umgesetzt. Zum einen werden Repository-Interfaces verwendet, um die verschiedenen Funktionen für die Persistenz jeder Entität zu repräsentieren. Dadurch wird die Verantwortlichkeit klar abgegrenzt, da jedes Repository-Interface spezifische Operationen für eine bestimmte Entität definiert. Diese klare Trennung ermöglicht es, Änderungen an der Persistenzschicht vorzunehmen, ohne die anderen Teile der Anwendung zu beeinflussen. Darüber hinaus erfüllt das Projekt das Prinzip der Single Responsibility durch die Verwendung von mehreren GUI-Klassen, wobei jede Klasse für einen spezifischen Use Case zuständig ist. Jede GUI-Klasse hat eine klare Verantwortlichkeit und ist nur für die Darstellung und Interaktion mit einem bestimmten Teil der Benutzeroberfläche zuständig. Dies ermöglicht eine bessere Wartbarkeit und Lesbarkeit des Codes, da Änderungen an der Benutzeroberfläche nur in der entsprechenden GUI-Klasse vorgenommen werden müssen. Ein weiteres Beispiel für die Anwendung des Single Responsibility-Prinzips ist die Verwendung von Wertobjekten

(Value Objects). Wertobjekte haben die Verantwortlichkeit, bestimmte Werte zu repräsentieren und können spezifische Geschäftslogik enthalten, die nur für sie relevant ist. Durch die Verwendung von Wertobjekten wird die Verantwortlichkeit für die Handhabung und Validierung bestimmter Daten klar abgegrenzt, was zu einem klareren und wartbareren Code führt.

## Interface Segregation

Das Interface Segregation Principle (ISP) besagt, dass es besser ist, viele clientspezifische Schnittstellen zu haben, anstatt eine Allzweckschnittstelle. Im vorliegenden Projekt wird dieses Prinzip durch die Verwendung von Repositories umgesetzt. Jedes Repository-Interface definiert nur die relevanten Methoden für eine bestimmte Entität. Dadurch wird sichergestellt, dass jede Entität nur die Methoden implementiert, die sie tatsächlich benötigt.

ISP zielt darauf ab, ein System entkoppelt zu halten, was es einfacher macht, es zu refaktorieren, zu ändern und neu bereitzustellen. Anstatt alle Methoden in einer einzigen Schnittstelle zu definieren, werden durch ISP nur die erforderlichen Methoden in separaten Schnittstellen bereitgestellt. Dies fördern eine bessere Entkopplung und Flexibilität im Code, da Entitäten nur diejenigen Methoden implementieren müssen, die für ihre spezifischen Anforderungen relevant sind.

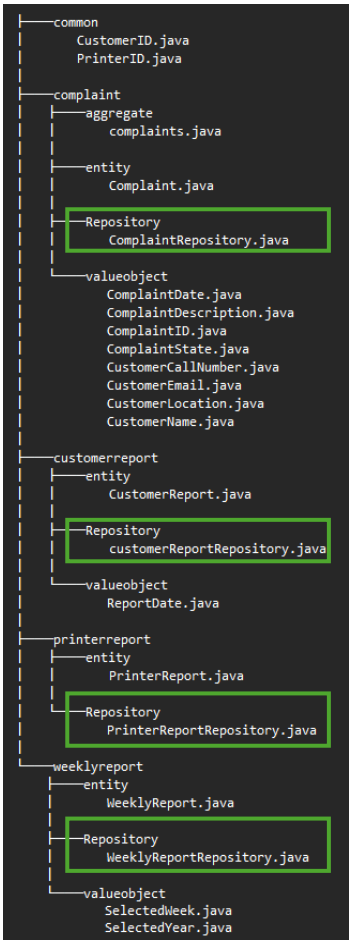


*Abbildung 5: Interface Segregation*

## DRY

Das DRY-Prinzip (Don't Repeat Yourself) zielt darauf ab, die Wiederholung von Informationen zu reduzieren, die wahrscheinlich geändert werden müssen. Dies wird durch die Verwendung von Abstraktionen erreicht, die weniger anfällig für Änderungen sind, oder durch die Nutzung von Datennormalisierung, um Redundanz von vornherein zu vermeiden.

Im vorliegenden Projekt wurde ein separates Paket namens "Common" erstellt, um Value Objects zu speichern, die in mehreren Bereichen der Anwendung verwendet werden. Dies entspricht dem DRY-Prinzip, da es die Wiederholung von Wertobjekten in verschiedenen Teilen der Anwendung vermeidet. Durch die zentrale Speicherung von gemeinsam genutzten Wertobjekten im "Common"-Paket wird Redundanz vermieden und die Wartbarkeit des Codes verbessert. Indem das "Common"-Paket als zentrale Quelle für wiederverwendbare Wertobjekte dient, wird vermieden, dass dieselben Wertobjekte in verschiedenen Teilen der Anwendung dupliziert werden müssen. Dies führt zu einem saubereren und wartbareren Code, da Änderungen an den gemeinsam genutzten Wertobjekten nur an einer einzigen Stelle vorgenommen werden müssen, was die Pflege und Aktualisierung erleichterten.

## POLA

POLA (Principle of Least Astonishment) zielt darauf ab, Schnittstellen und Verhaltensweisen so zu gestalten, dass sie für Benutzer oder Entwickler intuitiv sind und Überraschungen oder unerwartetes Verhalten minimieren. Im vorliegenden Codebeispiel, das eine Schnittstelle für die „customerreport.repository“ definiert, wird das Prinzip der geringsten Überraschung (POLA) in gewissem Maße eingehalten. Hier ist, wie der Code mit POLA übereinstimmt:

```java
package customerreport.repository;
import customerreport.entity.CustomerReport;
import java.io.IOException;
import java.text.ParseException;
public interface CustomerReportRepository {
    void createCustomerReport(CustomerReport customerReport) throws IOException, ParseException;
    void saveCustomerReport(CustomerReport customerReport, String filePath) throws IOException;
}
```

*Abbildung 6: Implementierung von POLA  Programming Principles*

1. Intuitive Namensgebung: Der Paketname customerreport.repository und der Schnittstellenname CustomerReportRepository sind intuitiv und folgen gängigen Java-Namenskonventionen. Dies erleichtert es anderen Entwicklern, den Zweck des Codes zu verstehen, ohne umfangreiche Dokumentation zu benötigen.
2. Klare Methodensignaturen: Die Methodensignaturen createCustomerReport und saveCustomerReport sind klar und beschreibend. Sie geben an, welche Aktionen

die Methoden ausführen und welche Parameter sie benötigen, was die Lesbarkeit verbessert, und die Ambiguität reduziert.

3. Konsistente Fehlerbehandlung: Beide Methoden geben die Ausnahmen an, die sie auslösen können (IOException und ParseException), was konsistent ist und Entwicklern hilft, potenzielle Fehler-Szenarien im Voraus zu verstehen.
4. Minimale kognitive Belastung: Die Schnittstelle ist prägnant und konzentriert sich auf die Kernfunktionalität des Erstellens und Speicherns von Kundenberichten. Sie vermeidet unnötige Komplexität oder zusätzliche Methoden, was die kognitive Belastung für Entwickler, die mit diesem Code interagieren, verringert.

## KISS

KISS (Keep It Simple, Stupid) ist ein Prinzip, das besagt, dass Systeme und Lösungen so einfach wie möglich gehalten werden sollten, ohne dabei die Funktionalität zu beeinträchtigen. Im vorliegenden Codebeispiel von den Weekly Report Service, das eine Methode zur Konvertierung eines Wochentags in einen entsprechenden Zeichenfolgenwert zeigt, wird das KISS-Prinzip eingehalten. Hier ist, wie der Code dem KISS-Prinzip folgt:

```
private String getDayOfWeekString(int dayOfWeek) {
return switch (dayOfWeek) {
case Calendar.SUNDAY -> "Sunday";
case Calendar.MONDAY -> "Monday";
case Calendar.TUESDAY -> "Tuesday";
case Calendar.WEDNESDAY -> "Wednesday";
case Calendar.THURSDAY -> "Thursday";
case Calendar.FRIDAY -> "Friday";
case Calendar.SATURDAY -> "Saturday";
default -> "";
};
```

*Abbildung 7: Implementierung von KISS Programming Principles*

1. Einfachheit der Lösung: Die Methode getDayOfWeekString verwendet eine einfache Switch-Anweisung, um den Wochentag basierend auf dem übergebenen numerischen Wert zu konvertieren. Diese Lösung ist einfach und leicht zu verstehen, was dem KISS-Prinzip entspricht.
2. Lesbarkeit des Codes: Die Verwendung einer Switch-Anweisung mit klaren Fällen für jeden Wochentag macht den Code leicht lesbar und verständlich. Entwickler können schnell erkennen, wie die Methode funktioniert und was sie tut, ohne komplexe Logik oder Abhängigkeiten.
3. Direkter Ansatz: Die Methode verwendet einen direkten Ansatz, um den Wochentag in einen entsprechenden Zeichenfolgenwert umzuwandeln, ohne unnötige Verzweigungen oder komplexe Bedingungen. Dies trägt dazu bei, die

Wartbarkeit des Codes zu verbessern und potenzielle Fehlerquellen zu minimieren.

## Information Expert & the Creator-Prinzip

Die Klasse Complaint implementiert den Builder-Entwurfsmuster, was nicht direkt den Prinzipien des Informationsexperten oder des Schöpfers aus dem objektorientierten Design entspricht. Dennoch zeigt die Klasse indirekt einige Aspekte dieser Prinzipien:

**Die Informationsexpertenprinzip:**

Das Informationsexpertenprinzip besagt, dass eine Klasse die Verantwortung für das Verwalten von Informationen tragen sollte und über die notwendigen Informationen verfügen sollte, um ihre Aufgaben zu erfüllen. In diesem Fall enthält die Klasse Complaint verschiedene Informationen zu einer Kundenbeschwerde, wie z.B. Kundenname, Beschreibung, Telefonnummer, E-Mail usw. Jedes Feld innerhalb der Klasse Complaint repräsentiert ein spezifisches Stück Information im Zusammenhang mit einer Beschwerde. Zum Beispiel CustomerName, ComplaintDescription, CustomerCallNumber usw. Diese Felder liefern gemeinsam die notwendigen Informationen über eine Beschwerde und halten somit das Prinzip des Informationsexperten ein.

**Das Schöpfungsprinzip:**

Das Schöpfungsprinzip besagt, dass eine Klasse dafür verantwortlich sein sollte, Instanzen anderer Klassen oder Objekte zu erstellen. In der Klasse Complaint dient die innere Klasse Builder als Erzeuger für Instanzen der Klasse Complaint. Die Builder-Klasse umschließt die Konstruktionslogik und ermöglicht es Clients, Instanzen von Complaint mit einer fließenden Schnittstelle zu erstellen. Die Builder-Klasse erleichtert die Erstellung von Complaint-Objekten, indem sie Methoden bereitstellt, um die verschiedenen Attribute einer Beschwerde festzulegen. Durch die Trennung der Konstruktionslogik von der Klasse Complaint selbst wird das Schöpfungsprinzip eingehalten, da es die Trennung von Anliegen fördert und eine einzige Verantwortung beibehält.

Zusammenfassend lässt sich sagen, dass während die Klasse Complaint hauptsächlich das Builder-Muster demonstriert, sie indirekt dem Informationsexpertenprinzip entspricht, indem sie notwendige Informationen enthält, und dem Schöpfungsprinzip, indem sie eine separate Builder-Klasse zur Objekterstellung verwendet. Diese Prinzipien tragen zur allgemeinen Designqualität und Wartbarkeit des Codes bei.

# Refactoring

## ComplaintService, Verbesserung der Modularität und Lesbarkeit

Die Klasse ComplaintService ist für zwei unterschiedliche Aufgaben zuständig: das Lesen von Dateien und die Validierung von Verträgen. Um die Code-Struktur zu verbessern und dem Prinzip der Einzelverantwortlichkeit (SRP) zu entsprechen, ist es ratsam, diese Aufgaben in verschiedene Klassen oder Methoden aufzuteilen. Insbesondere sollte die Vertragsvalidierung in eine separate Klasse ausgelagert werden. Durch die Extraktion der Vertragsvalidierungslogik in eine eigene Klasse wird eine bessere Trennung der Verantwortlichkeiten erreicht. Dies verbesserten die Lesbarkeit, Wartbarkeit und Testbarkeit des Codes. Darüber hinaus fördert es eine modularere und skalierbarere Architektur, in der jede Klasse oder Methode einen klaren und fokussierten Zweck hat. Die neue Klasse, die für die Vertragsvalidierung verantwortlich ist, kann alle erforderlichen Validierungslogiken umfassen, was die Verwaltung und Aktualisierung unabhängig von anderen Teilen des Codebestands erleichtert. Dieser Ansatz entspricht bewährten Methoden des Software-Designs und trägt zu einem robusteren und organsierteren System bei.

GitHub:

https://github.com/Mohmad-Naser-alnakeshbandi/ASE2/commit/2f2b7a8539367fae3da45db96d6303a4217d231d

### Effizientes Datumsparsing verbessern

Früher wurde in der Methode getCustomerComplaintImplementation für jeden Beschwerdeeintrag ein neues SimpleDateFormat-Objekt erstellt. Diese Vorgehensweise ist ineffizient und kann durch die Erstellung einer einzigen statischen Instanz von SimpleDateFormat verbessert werden

GitHub:

https://github.com/Mohmad-Naser-alnakeshbandi/ASE2/commit/21cd1ecaaf5501c293b9fe7808323ac67a07d7f2

### Entwirren von Verantwortlichkeiten in der GUIComplaint-Klasse

Die GUIComplaint-Klasse übernimmt zu viele Aufgaben. Sie kümmert sich um die Einrichtung der grafischen Benutzeroberfläche, Benutzerinteraktionen und interagiert sogar direkt mit Repositories. Dies verstößt gegen das Single Responsibility Principle. Es ist besser, die Verantwortlichkeiten auf separate Klassen zu delegieren

GitHub:

https://github.com/Mohmad-Naser-alnakeshbandi/ASE2/commit/0c3e4a8deec326c355de68bb69626a81a9fb0099

## Entwurfsmuster

In meinem Projekt zur Erstellung von Beschwerden habe ich das Builder-Entwurfsmuster verwendet, um eine Complaint-Instanz zu erstellen. Durch den Einsatz dieses Musters kann ich die Konstruktion komplexer Objekte schrittweise und flexibel durchführen, indem ich verschiedene Attribute je nach Bedarf festlege.

```
Complaint complaint = new Complaint.Builder()
        .name(new CustomerName(getFirstNameTextField().getText(), getLastNameTextField().getText()))
        .description(new ComplaintDescription(getTitleTextArea().getText(), getDescriptionTextArea().getText()))
        .callNumber(new CustomerCallNumber(getCallNumberTextField().getText()))
        .email(new CustomerEmail(getEmailInputTextField().getText()))
        .customerID(new CustomerID(getCustomerIDInputTextField().getText()))
        .location(new CustomerLocation( getCountryTextField().getText(),
                                        getStateTextField().getText(),
                                        getCityTextField().getText(),
                                        getStreetTextField().getText(),
                                        getLocationNumberTextField().getText()))
        .printerID(new PrinterID(getPrinterIDTextField().getText()))
        .complaintID(new ComplaintID())
        .complaintDate(new ComplaintDate())
        .complaintState(ComplaintState.RECEIVE)
        .build();
```

*Abbildung 8: Erstellung von einer Beschwerde mit dem Builder-Entwurfsmuster*

Der Builder ermöglicht es mir, die Initialisierung der Complaint-Instanz auf eine klare und übersichtliche Weise durchzuführen. Anstatt alle Parameter auf einmal festzulegen, wie es bei anderen Ansätzen der Fall sein könnte, kann ich jedes Attribut separat und lesbar setzen. Beispielsweise setze ich den Namen des Kunden, die Beschreibung der Beschwerde, die Kontaktnummer, die E-Mail-Adresse usw. nacheinander mit entsprechenden Methodenaufrufen. Ein großer Vorteil des Builder-Entwurfsmusters besteht darin, dass ich nicht alle Attribute beim Erstellen der Instanz angeben muss. Ich kann nur die relevanten Attribute setzen und die anderen werden mit Standardwerten initialisiert. Dies bietet Flexibilität, da es mir ermöglicht, verschiedene Szenarien zu handhaben, ohne komplizierte Logik implementieren zu müssen. Darüber hinaus macht die Verwendung des Builder-Musters den Code wartbarer und erweiterbar. Wenn sich die Anforderungen ändern und neue Attribute hinzugefügt werden müssen, kann ich einfach neue Methoden im Builder hinzufügen, ohne den bestehenden Code zu ändern. Dadurch bleibt der Code sauber und leicht zu pflegen.