Programmentwurf

- Programmentwurf
  - 1. Einführung
    - 1.1 Starten der Applikation
    - 1.3 Ausführen der Tests
  - 2. Clean Architecture
    - 2.1 Was ist Clean Architecture
    - 2.2 Analyse der Schichten
      - 2.2.1 Domänenschicht
      - 2.2.2 Anwendungsschicht
      - 2.2.3 Adapterschicht
    - 2.3 Analyse der Dependency Rule
      - 2.3.1 Positiv Beispiel : GetraenkeUsecases
      - 2.3.2 Positiv Beispiel : KundenUsecases
  - 3. SOLID
    - 3.1 Open/Closed Principle (OCP)
      - 3.1.1 Positives Beispiel ConsoleAdapter
      - 3.1.2 Negatives Beispiel ConsoleUtils
    - 3.2 Interface Segregation Principle (ISP)
      - 3.2.1 Positives Beispiel KundenInputHandler
      - 3.1.2 Negatives Beispiel CommandRegistrar
    - 3.3 Single Responsibility Principle (SRP)
      - 3.3.1 Positives Beispiel KundenInputHandler
      - 3.3.2 Negatives Beispiel GetraenkeRepositoryImpl
  - 4. Weiter Prinzipien
    - GRASP: Kopplung
      - Positives Beispiel: CustomerRepository
      - Negativ Beispiel ConsoleUtils
    - GRASP: High Cohesion ConsoleUtils
      - Begründung
      - Vorteile hoher Kohäsion
      - Technische Metriken
    - Dont Repeat Yourself (DRY
      - Begründung
      - Refactoring zur Vermeidung von Redundanz

        - Vorteile dieses Refactoring
        - Technische Metriken
    - 5. Design Pattern
        - 5.1 Builder Pattern
        - 5.2 Strategy Pattern
    - 6. Domain Driven Design (DDD)
        - 6.1 Entities
        - 6.2 Valueobjects
        - 6.3 Aggregates
        - 6.4 Repositories
    - 7. Unit Tests
        - 7.1 Zehn Unit Tests - Tabelle
        - 7.2 ATRIP
        - 7.3 Code Coverage
        - 7.4 Fakes und Mocks
            - 7.4.1 Mock-Objekt: Repo
            - 7.4.2 KundenInputHandler
    - 8. Refactoring
        - 8.1 Code Smells
            - 8.1.1 Large Class
            - 8.1.2 Duplicate Code
        - 8.2 Refactorings
        - 7.2.1 Replace Parameter with Builder
        - 7.2.2 Extract Method

1. Einführung

Die Applikation ASE_Getraenke (Adavanced Software Engeniering) ist eine Command Line Interface (CLI), welches zu Unterstützung und Verwaltung der Gertränke des Wohnheims genutzt werden kann. Die Domäin ist, deswegen auch an die Prozesse des Wohnheimes angepasst und funktioniert, deswegen in diesem am reibungslosesten.

Funktionaliäten

1. **Kunden:** Im Kontext der Applikation sind Kunden, Wohnheimbewohner. Diese können über die Konsole erstellt unf verwaltet werden.

2. **Bestandkontrolle:** Die Software verwaltet den aktuellen Bestand der vorhanden Getränke im Wohnheim.
3. **Produktverwaltung:** Es können über die Applikation Produkte verwaltet werden, dazu gehören **TYP** (Kasten oder Flasche), **PFAND** und der **PREIS**.
4. **Bestellungen:** Bestellungen können Kunden zugewiesen oder von Kunden ausgeführt werden um neue Produkte zu Bestellen oder Produkte aus dem Bestand aus dem "Lager" zu nehmen
5. **Ausgabenverteilung:** Die Ausgaben bzw. der Kontostand jedes Kunden wird über seine Bestellungen gespeichert und kann ausgelesehen werden.

Ziel der Applikation
Die Software soll die Getränke Verwaltung des Wohnheims digitalisieren und es für die Benutzer einfacher und nachvollziehbarer machen wie Kosten zustande kommen oder wann neue Getränke bestellt werden müssen

1.1 Starten der Applikation

Prerequirments

- Java Development Kit (JDK) Version 17.0.9
- Apache Maven Version 3.9.9

Anleitung zum Ausführen der Anwendung

1. **Repository klonen**

```
git clone https://github.com/Gamagu/ase_getraenke.git
cd asegetraenke
```

2. Projekt Installieren und bauen

```
mvn clean install
```

Nachdem `mvn clean install` kann um den um den Build-Prozess zu verkürzen `mvn compile` genutzt werden

3. Starten der Anwendung

```
cd getraenkeadapter
mvn exec:java
```

4. Nutzung der Applikation über die Konsole

```
getraenke getstockamountforprodukt
getraenke getallpfandwerte
getraenke setpfandwert
getraenke getpriceforprodukt
getraenke setpriceforprodukt
getraenke addprodukt
getraenke getpfandwert
getraenke getallproducts
getraenke addpfandwert
getraenke acceptlieferung
getraenke setpfandwertprodukt
getraenke getpricehistoryforprodukt
getraenke getproduct
getraenke addbestellung
getraenke addzahlungsvorgang
kunde getallkunden
kunde setname
kunde getkunde
kunde createkunde
kunde getallbestellungen
kunde getkundenbalance
```

Der Nutzer kann nun die präsentierten Funktionen aufrufen und wird durch den Prozess geleitet.

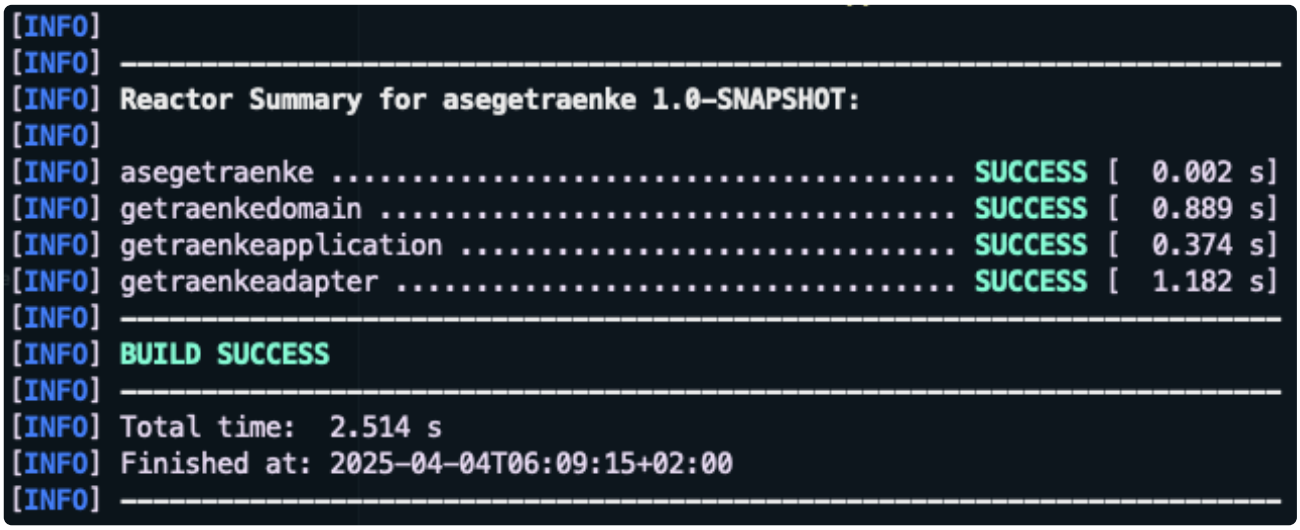
1.3 Ausführen der Tests

Die Tests werden über den Maven-Lifecycle automatischen bei dem Befehl `mvn clean install` mit ausgeführt. Die Anwendung ermöglicht auch das konkrette Ausführen der Tests.

```
cd asegetraenke
```

```
mvn test
```

Die Testergebnisse werden dann in der Konsole angezeigt.



2. Clean Architecture

2.1 Was ist Clean Architecture

Clean Architecture ist ein Softwarearchitekturkonzept mit dem Ziel, Software so zu gestalten, dass sie leicht verständlich, testbar, und flexibel bei Änderungen ist. Clean Architecture zeichnet sich durch eine klare Trennung der Verantwortlichkeiten und Abhängigkeiten aus, wodurch die Kernlogik der Anwendung unabhängig von äußeren Einflüssen bleibt.

Clean Architecture setzt sich aus folgenden Grundprinzipien zusammen:

1. Unabhängigkeit der Geschäftslogik
   Die Geschäftslogik (Use Cases) sollte unabhängig von den äußeren Details sein, wie z.B. Datenbanken, Benutzeroberflächen oder externen Frameworks.
2. Trennung der Verantwortlichkeiten
   Jede Schicht in der Architektur hat eine spezifische und klar definierte Aufgabe, was die Wartbarkeit und die Erweiterbarkeit der Anwendung fördert.
3. Dependency Rule
   Innere Schichten dürfen nichts von äußeren Schichten wissen. Abhängigkeiten sollten immer von außen nach innen zeigen, wobei die Kernlogik der Anwendung (wie die Geschäftslogik) keinen Bezug zu den weniger zentralen Detailbereichen (wie UI oder Datenbank) haben sollte.

2.2 Analyse der Schichten

Das Projekt basiert auf einer klaren Trennung in drei Schichten: die Adapterschicht, die Anwendungsschicht und die Domänenschicht. Diese Schichten folgen den Prinzipien der Clean Architecture und haben jeweils spezifische Aufgaben und Verantwortlichkeiten.

2.2.1 Domänenschicht

Die Domänenschicht bildet die Grundlage der Anwendung, indem sie die Datenbasis und deren Beziehungen definiert. Sie wird ausschließlich für die Datenrepräsentation genutzt und enthält nur minimale Logik. Zu dieser Logik gehören beispielsweise die Generierung von IDs oder die Verknüpfung verschiedener Datentypen. Ein Beispiel hierfür ist die Verknüpfung eines Produkts mit seinem aktuellen Preis. Das Hauptziel dieser Schicht ist es, eine konsistente und zuverlässige Datenbasis sicherzustellen, die unabhängig von äußeren Einflüssen bleibt.

2.2.2 Anwendungsschicht

Die Anwendungsschicht enthält die zentrale Geschäftslogik der Anwendung. Sie wird durch sogenannte Use Cases beschrieben, die die Schnittstelle zwischen der Domänenschicht und der Adapterschicht bilden. Diese Use Cases sind dafür verantwortlich, die Daten aus der Domänenschicht zu verarbeiten und an die Adapterschicht weiterzugeben. Um die Struktur der Anwendung übersichtlich zu halten, sind die Use Cases in zwei Hauptbereiche unterteilt: kundenbezogene und getränkebezogene Funktionalitäten. Diese Unterteilung ermöglicht eine klare Trennung der Verantwortlichkeiten, reduziert die Anzahl der Source-Files und sorgt dafür, dass die Dateien nicht unnötig groß werden.

2.2.3 Adapterschicht

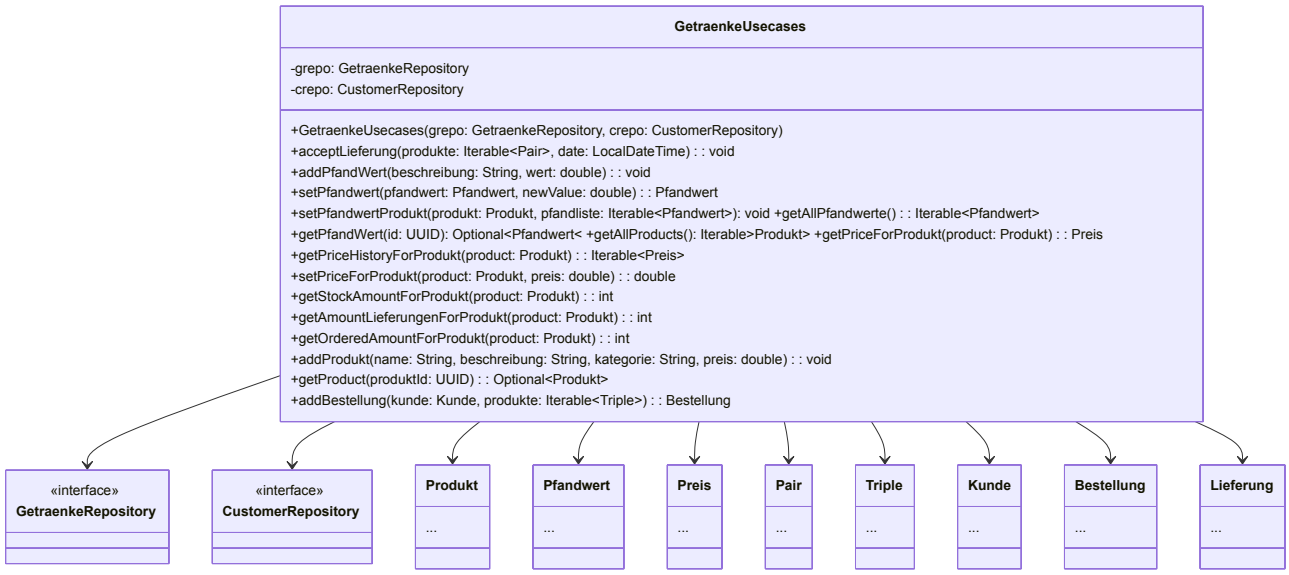
Die Adapterschicht dient als Verbindung zwischen der Geschäftslogik und den äußeren Systemen. Sie ermöglicht den Zugriff auf die Geschäftslogik und die Speicherung der Daten. Die Speicherung erfolgt über ein Repository, das ein Interface der Domänenschicht implementiert. Die Benutzerschnittstelle wird in diesem Projekt über das Terminal bereitgestellt. Dadurch können Benutzer direkt auf die in der Anwendungsschicht implementierten Use Cases zugreifen und die verschiedenen Funktionen der Anwendung nutzen.

2.3 Analyse der Dependency Rule

Das Projekt ist in der Struktur so aufgebaut, dass es nicht möglich ist gegen die Regel der Dependency Rule zu verstoßen. Im Folgenden werden, deswegen keine Negativ Beispiele gezeigt bei denen diese Regel missachtet wird.

```
├── 0-getraenkeadapter
├── 1-getraenkeapplication
├── 2-getraenkedomain
├── README.md
└── pom.xml
```

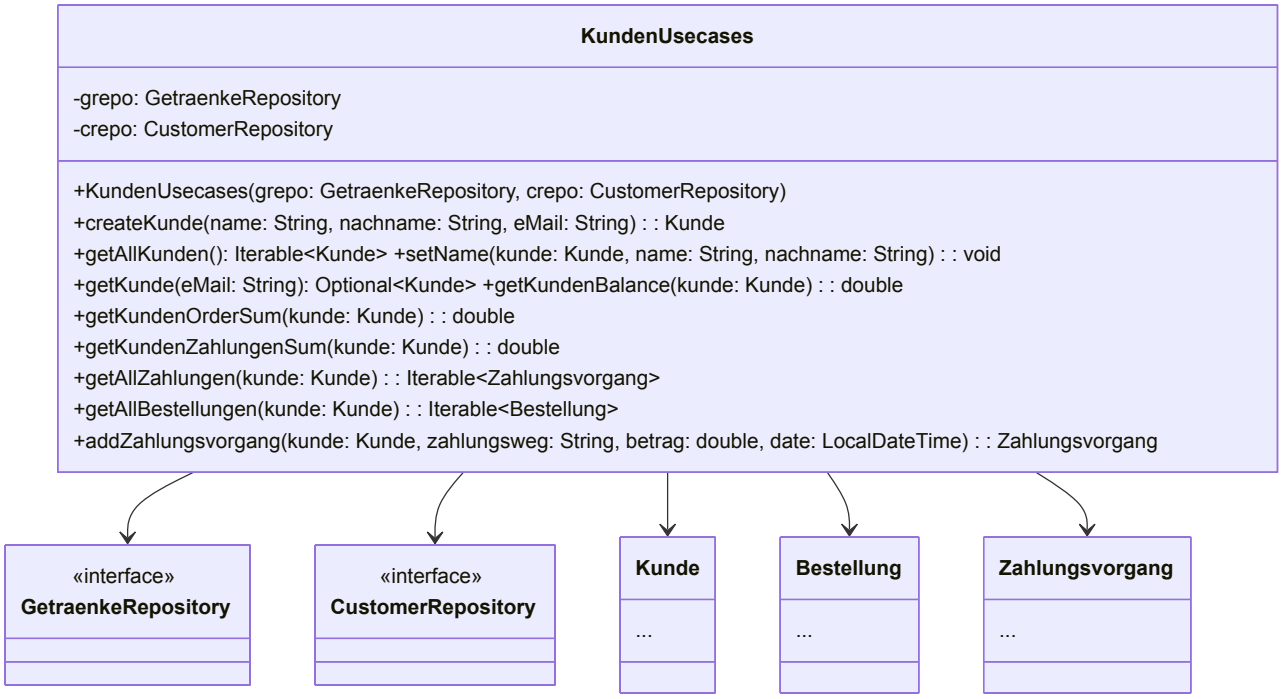
2.3.1 Positiv Beispiel : `GetraenkeUsecases`



Analyse:

- **Abhängigkeiten:** Die Klasse `GetraenkeUsecases` hängt von mehren Entities der Domain Schicht ab und von den zwei Interfaces `GetraenkeRepository` und `CustomerRepository` ab.
- **Einhaltung Dependency Rule:** Die Klasse `GetraenkeUsecases` hat keine Dependencies nach außen. Sie befindet sich auf der Applikations-Schicht und hat nur Abhängigkeiten auf der Domain-Schicht. Somit verlaufen die Abhängigkeiten wie laut der Regel definiert ausschließlich von Innen nach Außen.
  `Commit Stand: ec8012db4f8473d9cf1cad5178f139e92e3f416f`

2.3.2 Positiv Beispiel : `KundenUsecases`

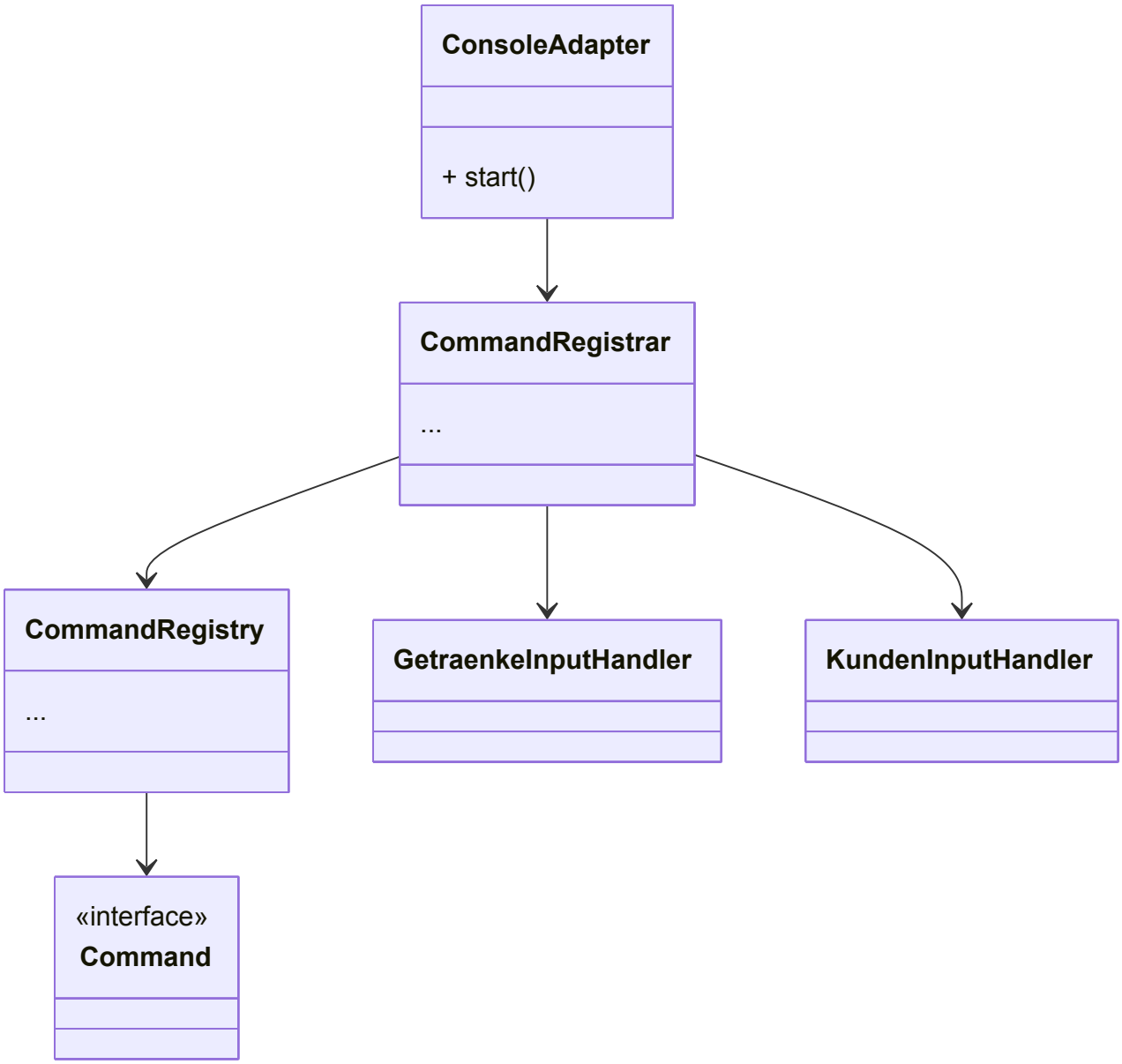


- **Abhängigkeiten:** Genauso wie die Klasse zuvor hat die Klasse `KundenUsecases` Abhängigkeiten zu Entities der Domain-Schicht und zu den Interfaces `GetraenkeRepository` und `CustomerRepository`.
- **Einhaltung der Dependency Rule:** Die Klasse `KundenUsecases` hat keine Dependencies nach außen. Sie befindet sich auf der Applikations-Schicht und hat nur Abhängigkeiten auf der Domain-Schicht. Genauso wie zuvor verlaufen die Abhängigkeiten ausschließlich von Innen nach Außen
  `Commit Stand: ec8012db4f8473d9cf1cad5178f139e92e3f416f`

3. SOLID

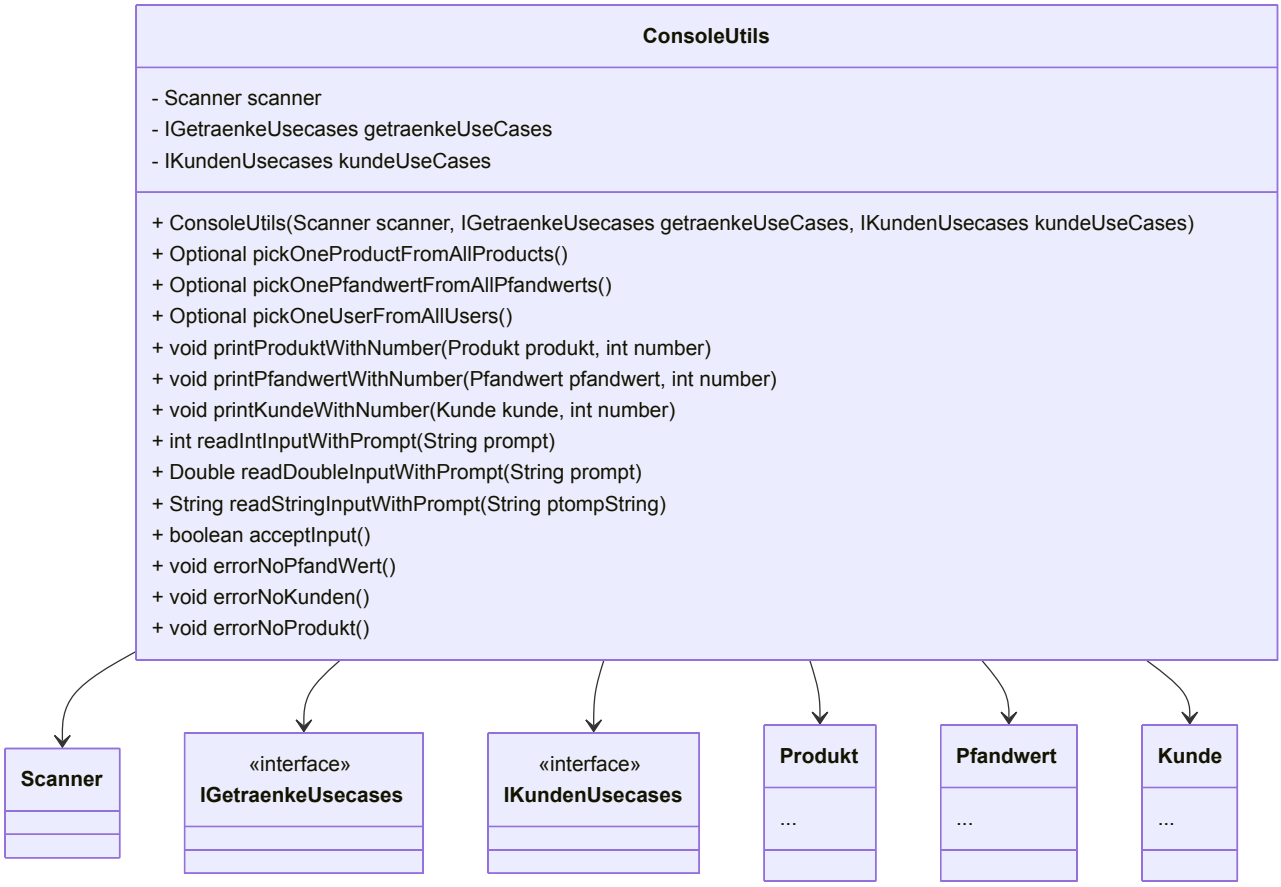
3.1 Open/Closed Principle (OCP)

3.1.1 Positives Beispiel `ConsoleAdapter`



Analyse:
Der `ConsoleAdapter` can einfach um neue Befehle erweitert werden. Die Befehle können hinzugefügt werden indem neue Klassen erstellt werden und die Methoden mit der Annotation `@Command(value = "getstockamountforprodukt", category = "getraenke")` ausgestattet werde. Die Klasse selbst muss sich dann allerdings noch bei dem `CommandRegistrar` selbst übergeben. Dieser Speicher dann den Namen und die Kategorie für die spätere Ausgabe. Das hinzufügen der Methoden kann in der neu eigefügten Klasse erfolgen und benötigt keine Änderungen an der `ConsoleAdapter` Klasse.
`Commit Stand: ec8012db4f8473d9cf1cad5178f139e92e3f416f`

3.1.2 Negatives Beispiel `ConsoleUtils`

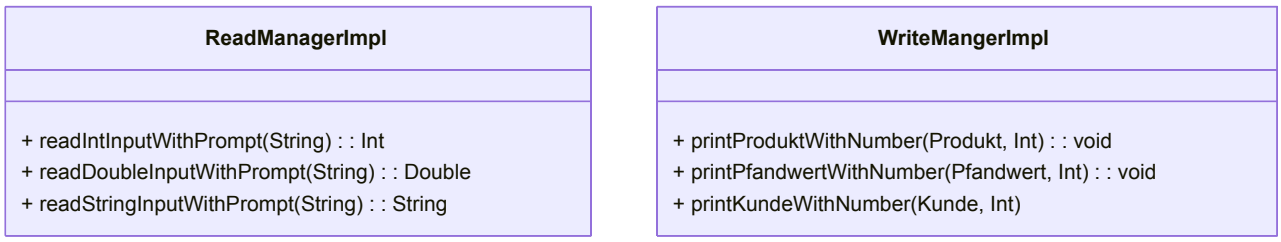


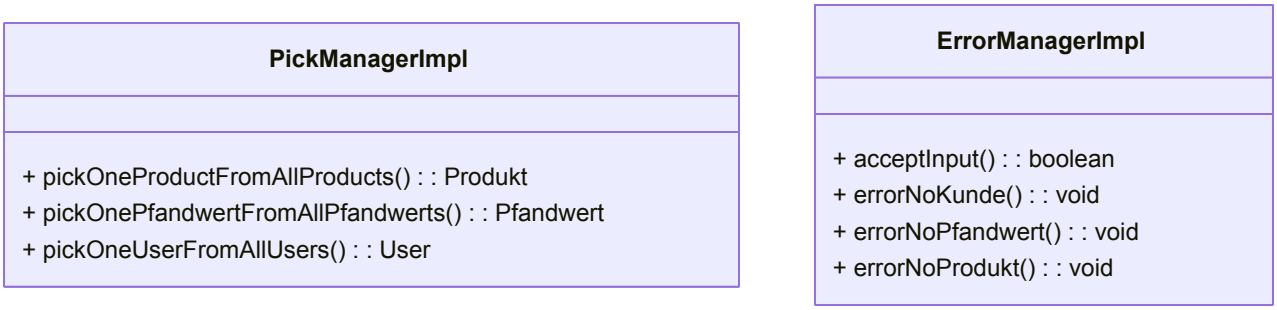
Analyse:

`ConsoleUtils` erlaubt keine Erweiterung ohne in der eigentliche Funktion Änderungen durchzuführen. Sie verstößt außerdem gegen mehrere Prinzipien, da sie darüber hinaus schlecht benannt ist und für mehre Aufgaben übernimmt dazu gehören. das Ein- und Auslesen, Bestätigungsabfragen und Auswahl von verschiedenen Entities. Diese Beziehen sich zwar alle auf die Reine Interaktionen des Benutzer mit der Console kann allerdings deutlich eleganter gelöst werden.

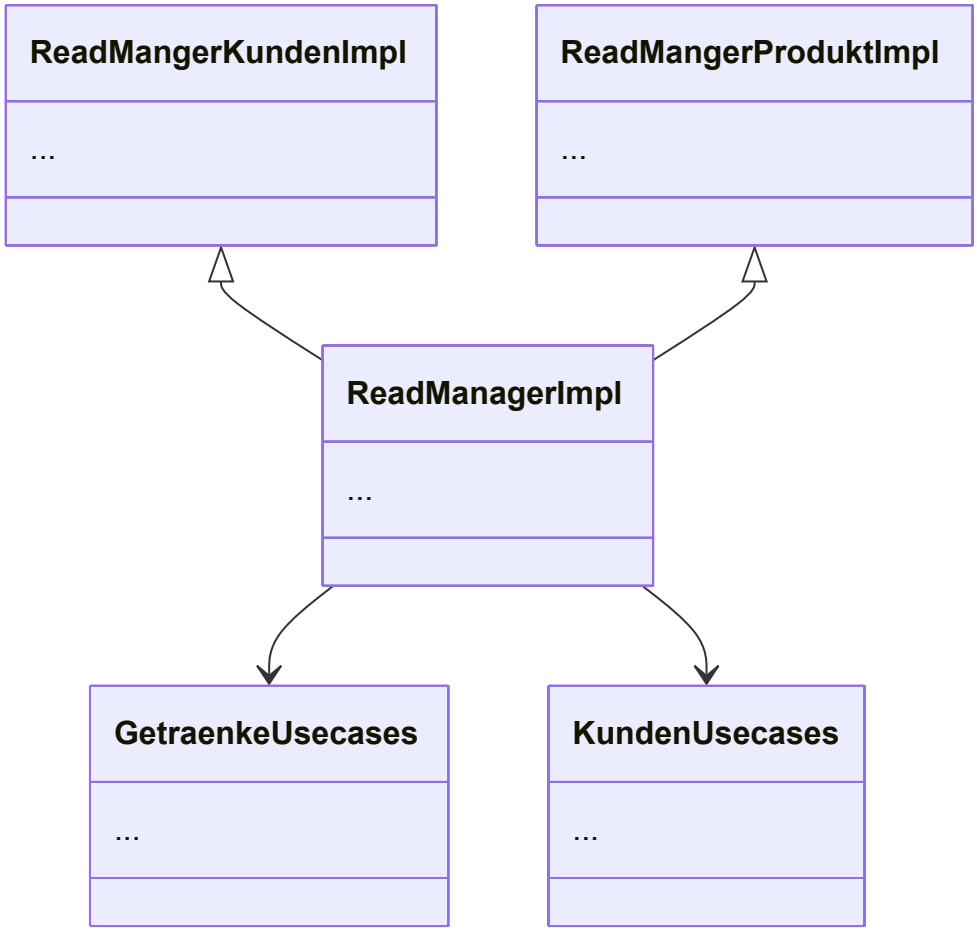
Lösungsvorschlag:

Vorerst sollte erwähnt werden , dass die Funktion erstmal unterteilt werden sollte in Folgende Klassen. Zuerst sollte die Klasse unterteilt werden in:





Wenn wir nun die Erweiterung der Elemente erlauben z.B. anhand von `ReadManagerImpl`. Durch des extenden der `ReadManagerImpl` würde dann ermöglicht werden neue Funktionalitäten hinzuzufügen.



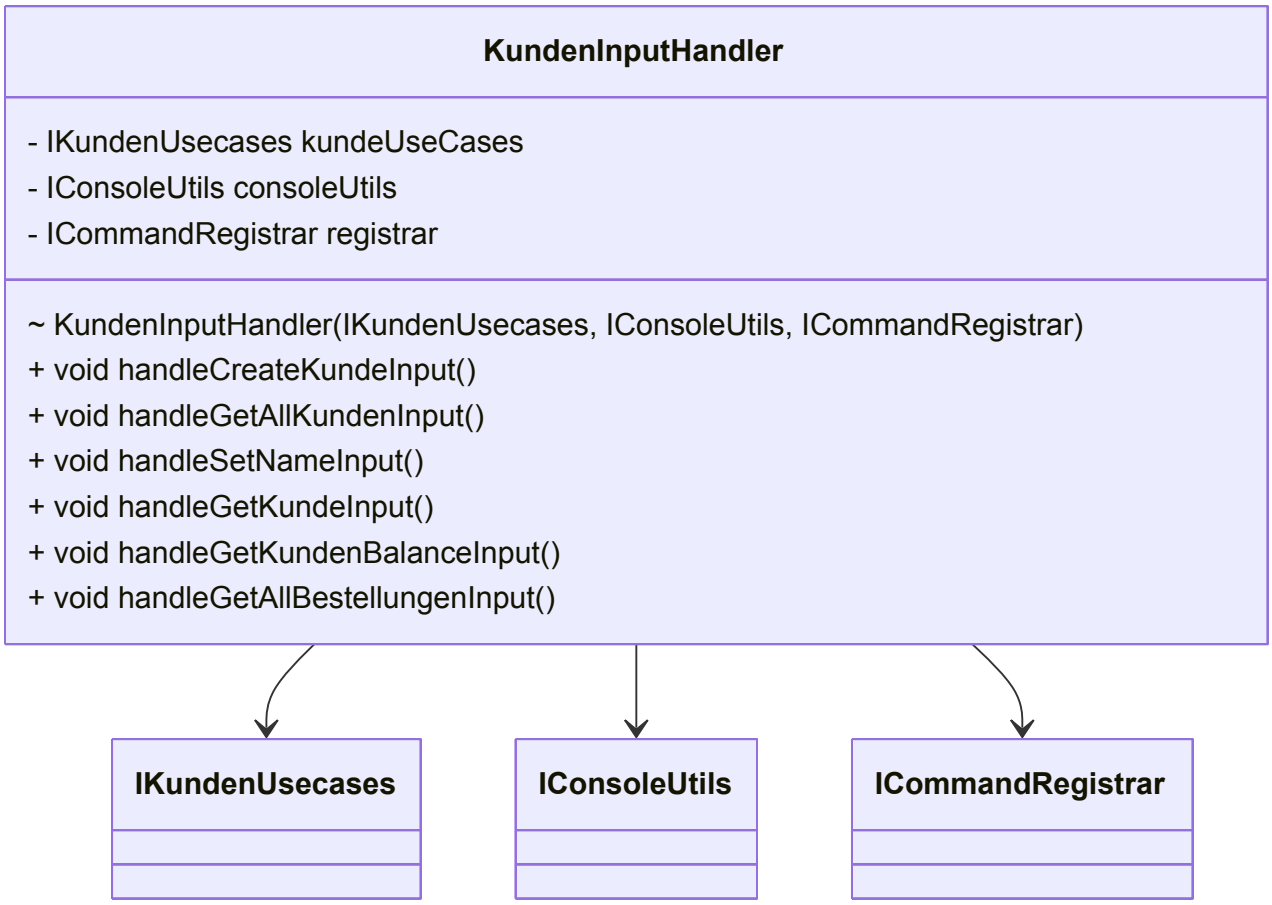
`Commit Stand: ec8012db4f8473d9cf1cad5178f139e92e3f416f`

Verbessert in : `Commit Stand: 5f3b9ef74a6e7ebcc939c045b32dbf242c376569`

3.2 Interface Segregation Principle (ISP)

3.2.1 Positives Beispiel `KundenInputHandler`

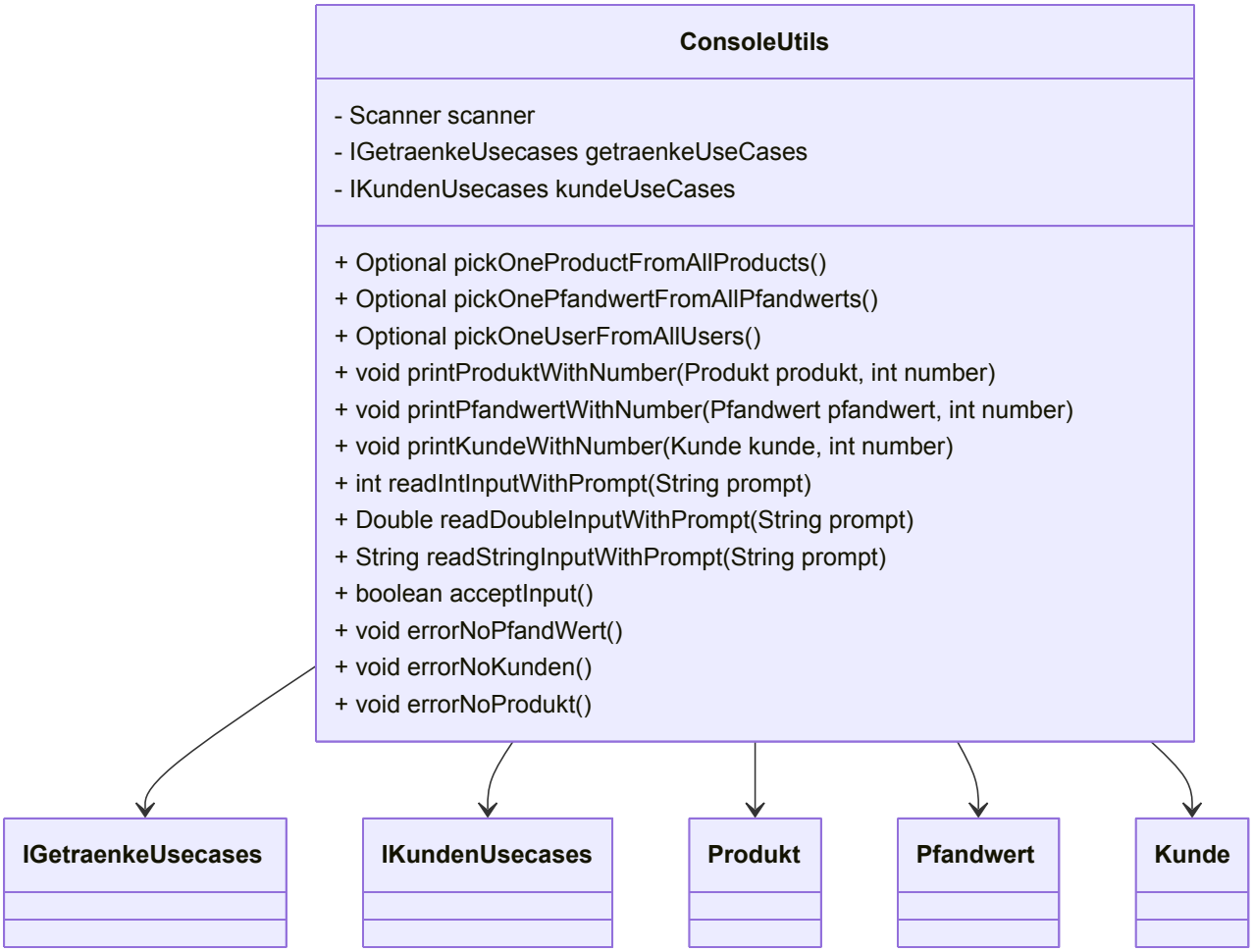
Die Klasse `KundenInputHandler` ist ein gutes Beispiel für das Interface Segregation Principle (ISP). Sie hängt nur von den Interfaces `IKundenUsecases`, `IConsoleUtils` und `ICommandRegistrar` ab und nutzt jeweils nur die für sie relevanten Methoden. Dadurch ist sichergestellt, dass `KundenInputHandler` nicht gezwungen ist, Methoden zu implementieren oder zu kennen, die für ihn nicht notwendig sind. Jedes Interface ist also spezifisch und nicht überladen.



`Commit Stand: ec8012db4f8473d9cf1cad5178f139e92e3f416f`

3.1.2 Negatives Beispiel `CommandRegistrar`

Die Klasse `ConsoleUtils` verstößt gegen das Interface Segregation Principle, da sie Methoden für ganz unterschiedliche Aufgaben bündelt: Einlesen, Ausgabe, Auswahl und Fehlerbehandlung. Dadurch werden andere Klassen gezwungen, ein breites Interface zu kennen und ggf. zu verwenden, auch wenn sie nur Teilfunktionalität brauchen.

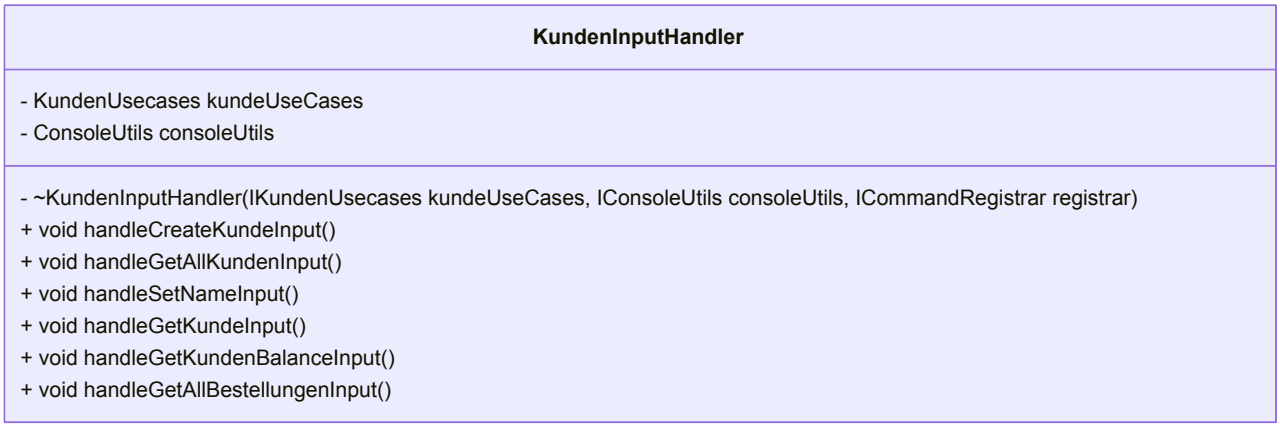


Die Klasse sollte in kleinere, spezialisierte Klassen bzw. Interfaces aufgeteilt werden:

- `ReadManager`: für das Einlesen von Daten
- `WriteManager`: für die formatierte Ausgabe
- `PickManager`: für Auswahlfunktionen
- `ErrorManager`: für Fehlermeldungen
  So könnte z. B. `KundenInputHandler` nur `ReadManager` und `PickManager` verwenden, ohne mit unnötigen Methoden konfrontiert zu werden.

3.3 Single Responsibility Principle (SRP)

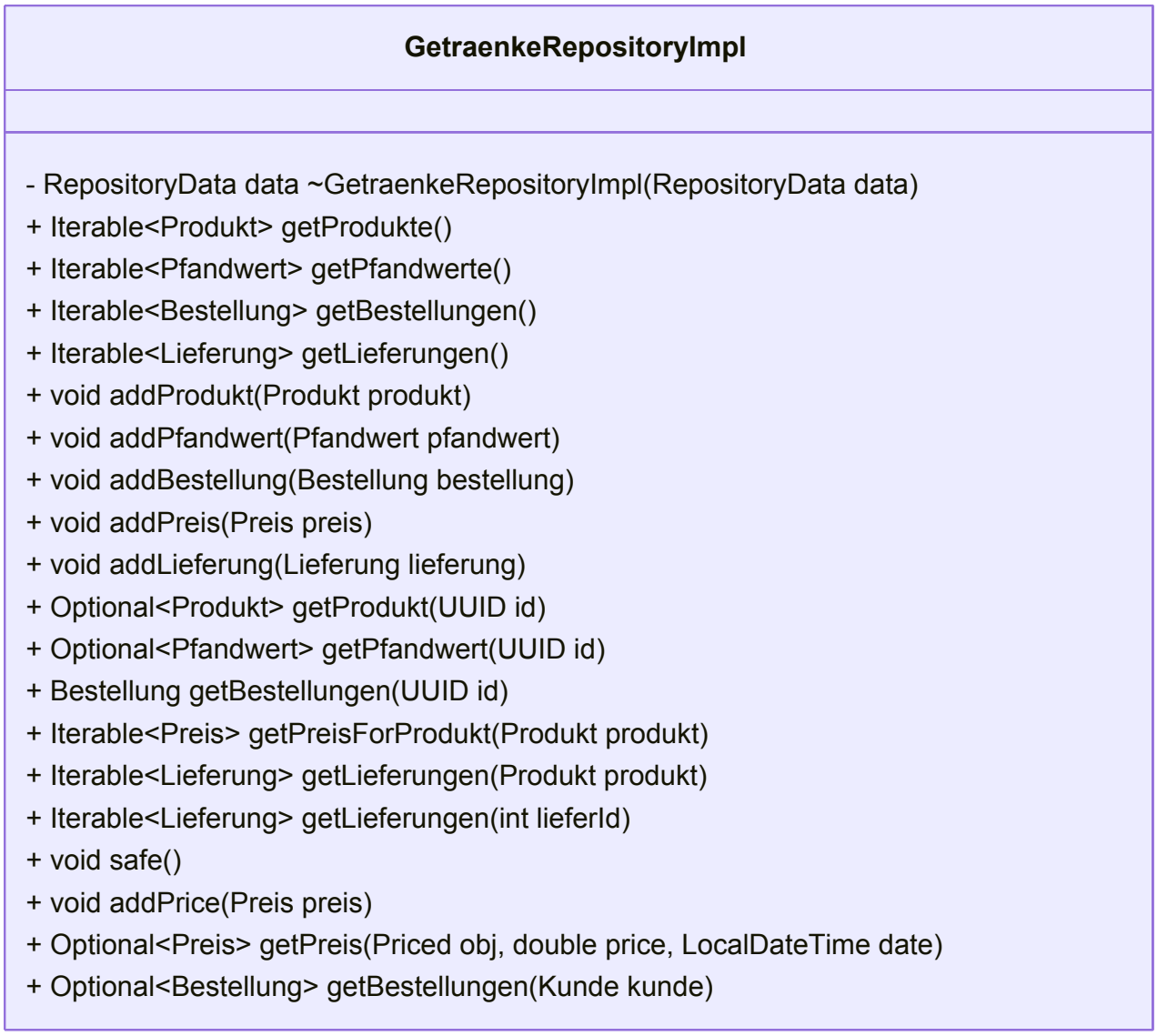
3.3.1 Positives Beispiel `KundenInputHandler`



Aufgabenbereich:
Diese Klasse erfüllt das SRP Prinzip, da sie ausschließlich die kundenbezogenen Daten der Ein- und der Ausgabe verwalten. Aufgaben wie das tatsächliche Ausgeben, das Einlesen oder andere Teile der Entities zu verwalten sind nicht Teil der Klasse. Somit ist das ihre einzige Aufgabe und befolgt das Prinzip der SRP.

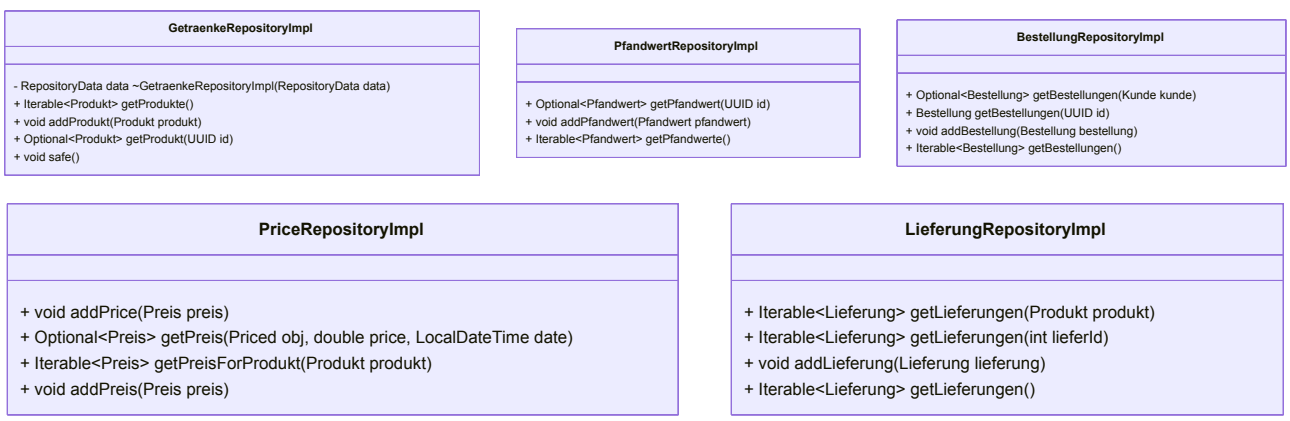
`Commit Stand: ec8012db4f8473d9cf1cad5178f139e92e3f416f`

3.3.2 Negatives Beispiel `GetraenkeRepositoryImpl`



Analyse:

Diese Klasse `GetraenkeRepositoryImpl` kümmert sich um unterschiedliche Entities und verletzt deswegen . Außerdem könnte man diese dann nachdem Auflösen in die Einzelnen Entities, dann nach Auslesen und Einlesen unterteilen.

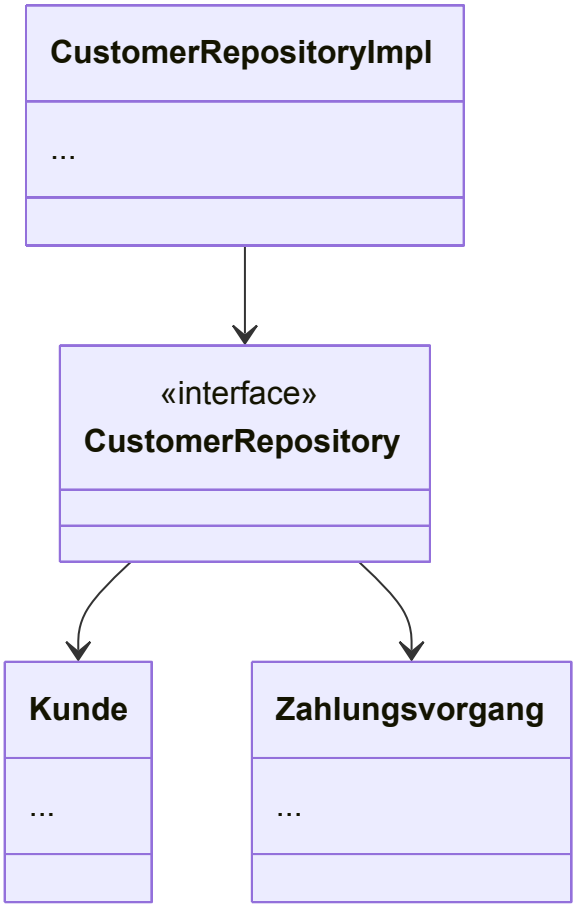


Lösungsvorschlag:
Indem wir die Klasse `GetraenkeRepositoryImpl` in kleiner Klassen aufteilen hier `GetraenkeRepositoryImpl`, `PfandwertRepositoryImpl`, `BestellungRepositoryImpl`, `PriceRepositoryImpl` und `LieferungRepositoryImpl` aufteilen müssten die Klassen nur noch die Implementation machen die sie auch benötigen. Somit wäre SRP nicht weiter durch die Klasse verletzt
`Commit Stand: ec8012db4f8473d9cf1cad5178f139e92e3f416f`

4. Weiter Prinzipien

GRASP: Kopplung

Positives Beispiel: `CustomerRepository`



Analyse:

Diese Verbindung hat wenig Kopplung, da die Implementierung `CustomerRepositoryImpl` nicht direkt von einer Konkretten Implementierung abhängt sondern ein Interface

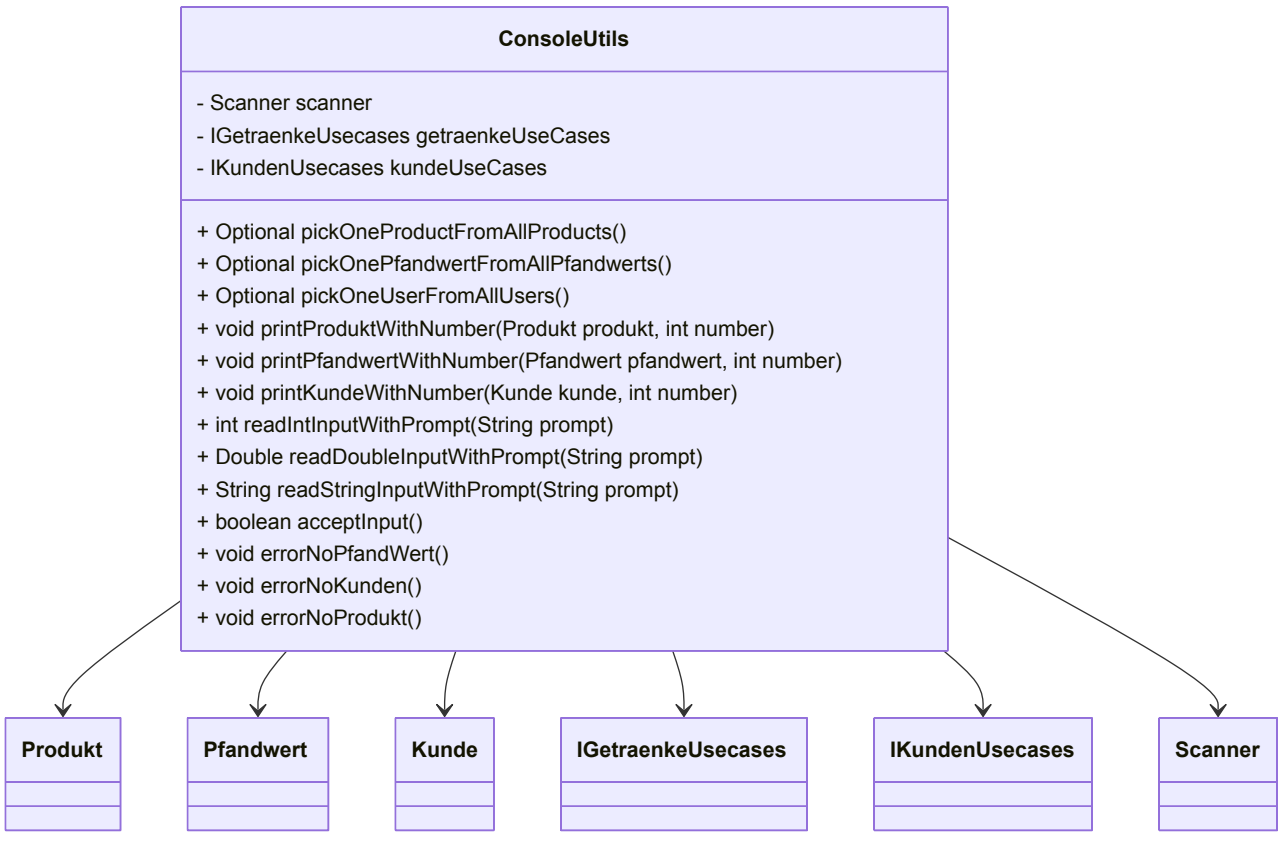
implementiert `CustomerRepository`. Die ermöglicht mehre Möglichkeiten für die Implementierung des `CustomerRepository`.

`Commit Stand: ec8012db4f8473d9cf1cad5178f139e92e3f416f`

Negativ Beispiel `ConsoleUtils`

ConsoleUtils ist ein zentrales Beispiel für zu hohe Kopplung:
Sie kennt und verwendet viele verschiedene Klassen und Interfaces, darunter:

- `Produkt`, `Pfandwert`, `Kunde` (Domain-Entities)
- `IGetraenkeUsecases`, `IKundenUsecases` (Use Cases)
- `Scanner` (technische Komponente)
  Die Klasse ist daher mit zu vielen Komponenten des Systems stark gekoppelt. Das erschwert Testbarkeit, Wiederverwendbarkeit und Wartbarkeit.



Problematisch ist insbesondere:

- Hohe Anzahl an direkten Abhängigkeiten.
- Enge Kopplung an konkrete Implementierungen.

- Direkte Verantwortung für viele verschiedene Aufgabenbereiche.
- Änderungen in Domain-Klassen (z. B. `Produkt`) können sich direkt auf `ConsoleUtils` auswirken.

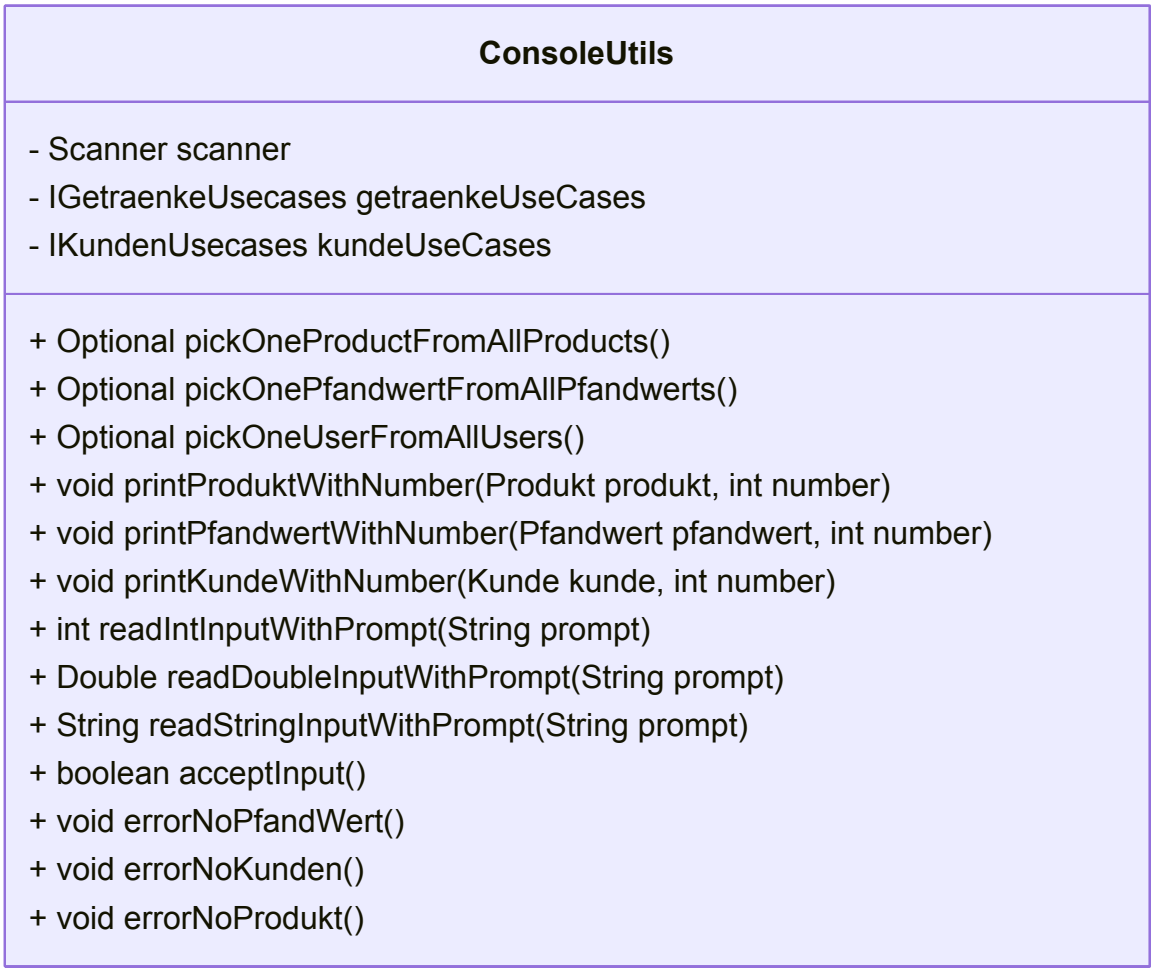
Lösungsvorschlag
Die Kopplung kann durch die Aufspaltung in **spezialisierte Komponenten** deutlich reduziert werden:

- `PickManager` (für Auswahl)
- `ReadManager` (für Eingaben)
- `WriteManager` (für Ausgaben)
- `ErrorManager` (für Fehlerhandling)
  Jede dieser Komponenten hängt dann nur noch von einem spezifischen Teil ab – und kann unabhängig weiterentwickelt oder getestet werden.

`Commit Stand: ec8012db4f8473d9cf1cad5178f139e92e3f416f`

Verbessert in `Commit Stand: 5f3b9ef74a6e7ebcc939c045b32dbf242c376569`

GRASP: High Cohesion `ConsoleUtils`



`Commit Stand: ec8012db4f8473d9cf1cad5178f139e92e3f416f`

Begründung

Die ursprüngliche Klasse `ConsoleUtils` wies eine zu breite Verantwortlichkeit auf: Sie kombinierte Eingabe- und Ausgabelogik, Fehlerbehandlung sowie die Auswahl von Entitäten. Dies führte zu niedriger Kohäsion und erschwerte die Wartbarkeit und Wiederverwendbarkeit.
Im Refactoring wurde `ConsoleUtils` in vier klar abgegrenzte Klassen aufgeteilt:

- `ConsoleReader`: Verantwortlich für das Einlesen und Validieren von Benutzereingaben.
- `ConsolePrinter`: Zuständig für die formatierte Ausgabe von Domänenobjekten.
- `ConsoleError`: Kapselt die Darstellung von Fehlermeldungen.
- `EntityPicker`: Bietet generische Auswahlmechanismen aus Listen, unabhängig vom konkreten Typ.
  Jede dieser Klassen konzentriert sich auf eine eng umrissene Aufgabe und verwendet nur die dafür relevanten Attribute und Methoden.

Vorteile hoher Kohäsion

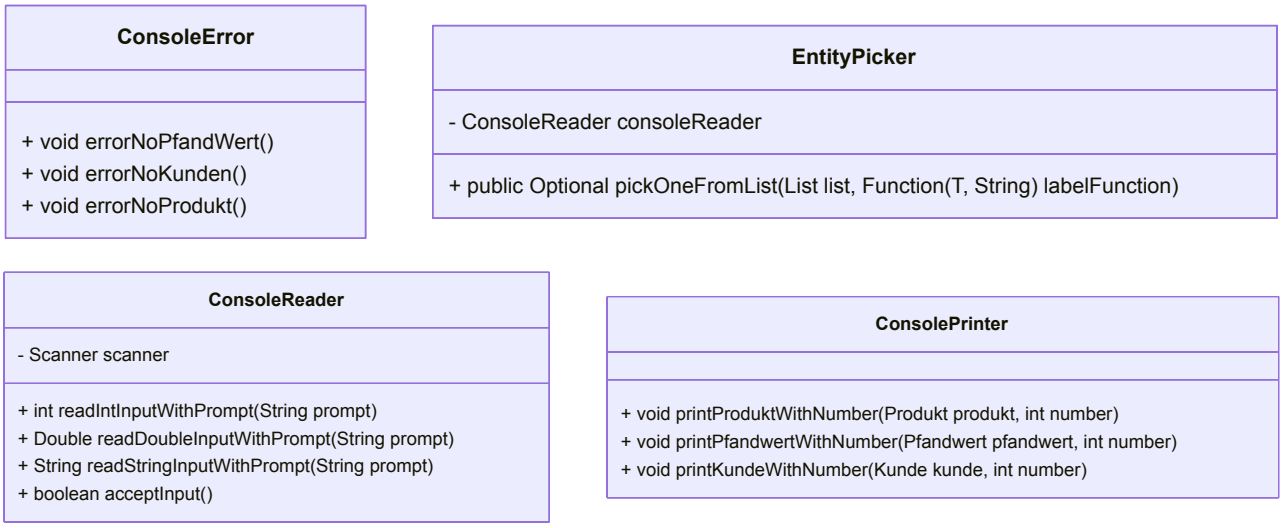
Durch die klare Trennung der Verantwortlichkeiten entstehen Klassen mit hoher Kohäsion: Jede Klasse enthält nur Methoden, die logisch und semantisch zusammengehören. Dies führt zu:

- **Besserer Lesbarkeit**: Die Intention der Klassen ist sofort erkennbar.
- **Einfacherer Wiederverwendung**: Komponenten wie `ConsoleReader` oder `EntityPicker` können auch in anderen Kontexten (z. B. Tests, anderer UI) verwendet werden.
- **Erhöhter Testbarkeit**: Die modularen Klassen lassen sich gezielt und unabhängig voneinander testen.
- **Reduzierter Wartungsaufwand**: Änderungen in der Eingabe- oder Ausgabelogik betreffen nur einzelne, klar abgegrenzte Klassen.

Technische Metriken

Nach dem Refactoring hat jede der vier Klassen:

- Eine überschaubare Anzahl an Methoden
- Eine einheitliche Schnittstelle
- Eine enge Nutzung ihrer Attribute (z. B. `scanner` nur in `ConsoleReader`)
  Dies sind Indikatoren für eine **hohe Kohäsion** im Sinne von GRASP. Die neue Struktur verbessert die Modularität der Konsoleninteraktion signifikant.



`Commit Stand: 5f3b9ef74a6e7ebcc939c045b32dbf242c376569`

Dont Repeat Yourself (DRY

Begründung

Die ursprüngliche Implementierung der Klasse `ConsoleUtils` verstieß gegen das **DRY-Prinzip (Don't Repeat Yourself)**, da mehrfach identische oder stark ähnliche Logik zur Auswahl und Ausgabe von Entitäten implementiert wurde. Die folgenden Methoden zeigen dieses Muster deutlich:

```
public Optional<Produkt> pickOneProductFromAllProducts() { ... }
public Optional<Pfandwert> pickOnePfandwertFromAllPfandwerts() { ... }
public Optional<Kunde> pickOneUserFromAllUsers() { ... }

public void printProduktWithNumber(Produkt produkt, int number) { ... }
public void printPfandwertWithNumber(Pfandwert pfandwert, int number) {
... }
public void printKundeWithNumber(Kunde kunde, int number) { ... }`
```

`Commit Stand: ec8012db4f8473d9cf1cad5178f139e92e3f416f`

Diese Methoden unterscheiden sich fast ausschließlich im Typ der verwendeten Entität, folgen jedoch demselben strukturellen Ablauf (z. B. Listen-Auswahl oder nummerierte Ausgabe).

Refactoring zur Vermeidung von Redundanz

Zur Auflösung der Wiederholungen wurde die generische Hilfsklasse `EntityPicker<T>` eingeführt. Sie kapselt die wiederkehrende Logik zur Auswahl eines Elements aus einer Liste und ist auf verschiedene Typen anwendbar:

```
public Optional<T> pickOneFromList(List<T> list, Function<T, String>
labelFunction)`
```

`Commit Stand: 5f3b9ef74a6e7ebcc939c045b32dbf242c376569`

Die Methode verwendet einen `Function<T, String>`, um eine flexible Beschriftung für beliebige Objekte zu ermöglichen. Dadurch kann dieselbe Methode sowohl für Produkte, Pfandwerte als auch Kunden verwendet werden.

Vorteile dieses Refactoring

- **Weniger Redundanz**: Wiederverwendbare generische Logik ersetzt mehrfach implementierten Code.
- **Erhöhte Erweiterbarkeit**: Neue Entitätstypen können mit minimalem Mehraufwand eingebunden werden.
- **Geringere Kopplung**: Die Logik zur Benutzerauswahl ist entkoppelt von spezifischen Domänenklassen.
- **Bessere Testbarkeit**: Die Auswahl-Logik ist isoliert und kann unabhängig getestet werden.

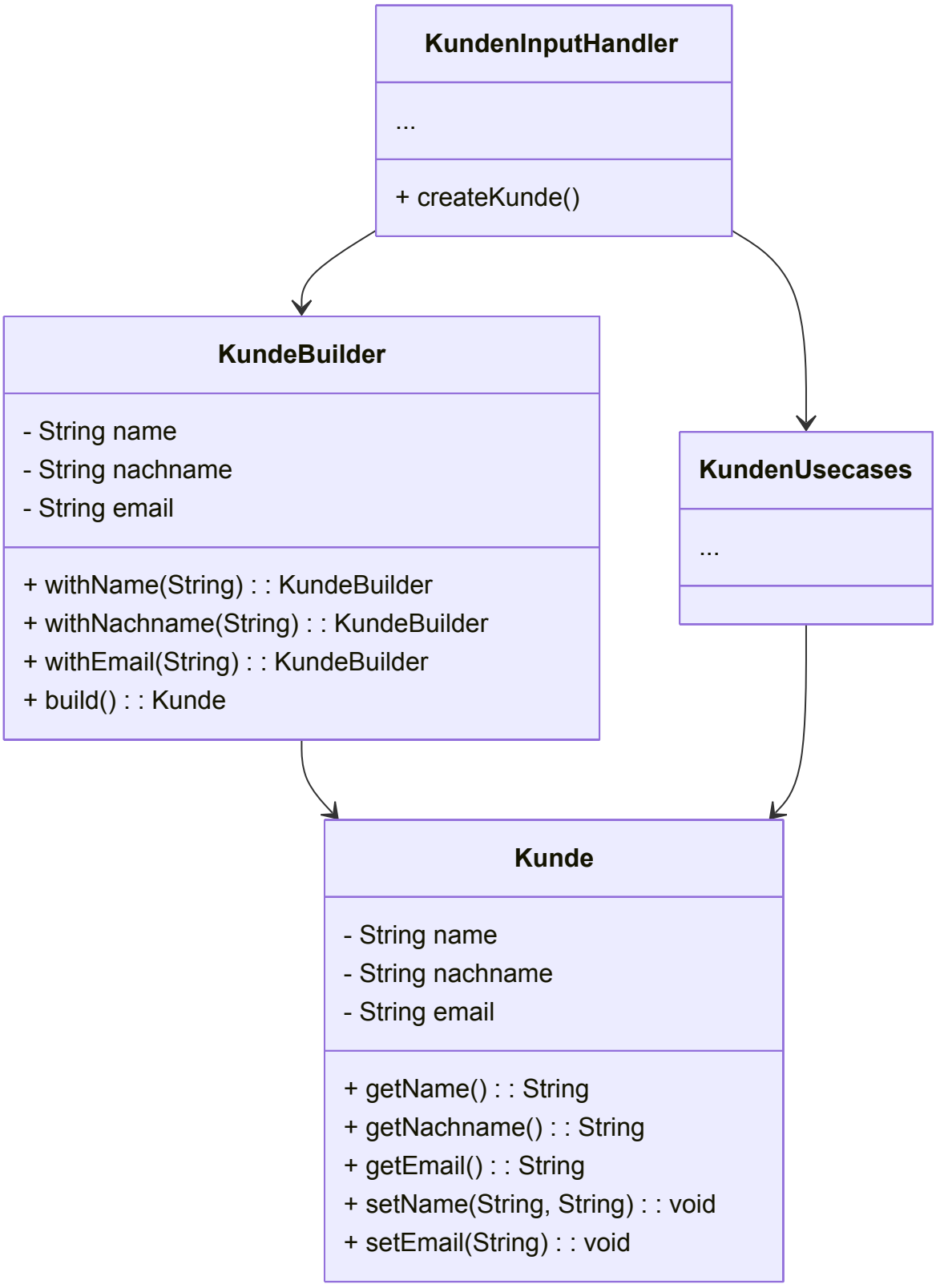
Technische Metriken

Die Anzahl der Methoden in `ConsoleUtils` wurde signifikant reduziert. Die generische `EntityPicker`-Klasse wird mehrfach verwendet und erfüllt somit das DRY-Prinzip auf effektive Weise und ersetzt die drei vorherigen Funktionen.

5. Design Pattern

5.1 Builder Pattern

Das Builder Pattern wird verwendet, um die Erstellung von `Kunde`-Objekten zu strukturieren. Gerade bei mehreren Parametern wie Name, Nachname und E-Mail wird dadurch die **Lesbarkeit erhöht** und die Fehleranfälligkeit durch falsche Parameterreihenfolgen reduziert.

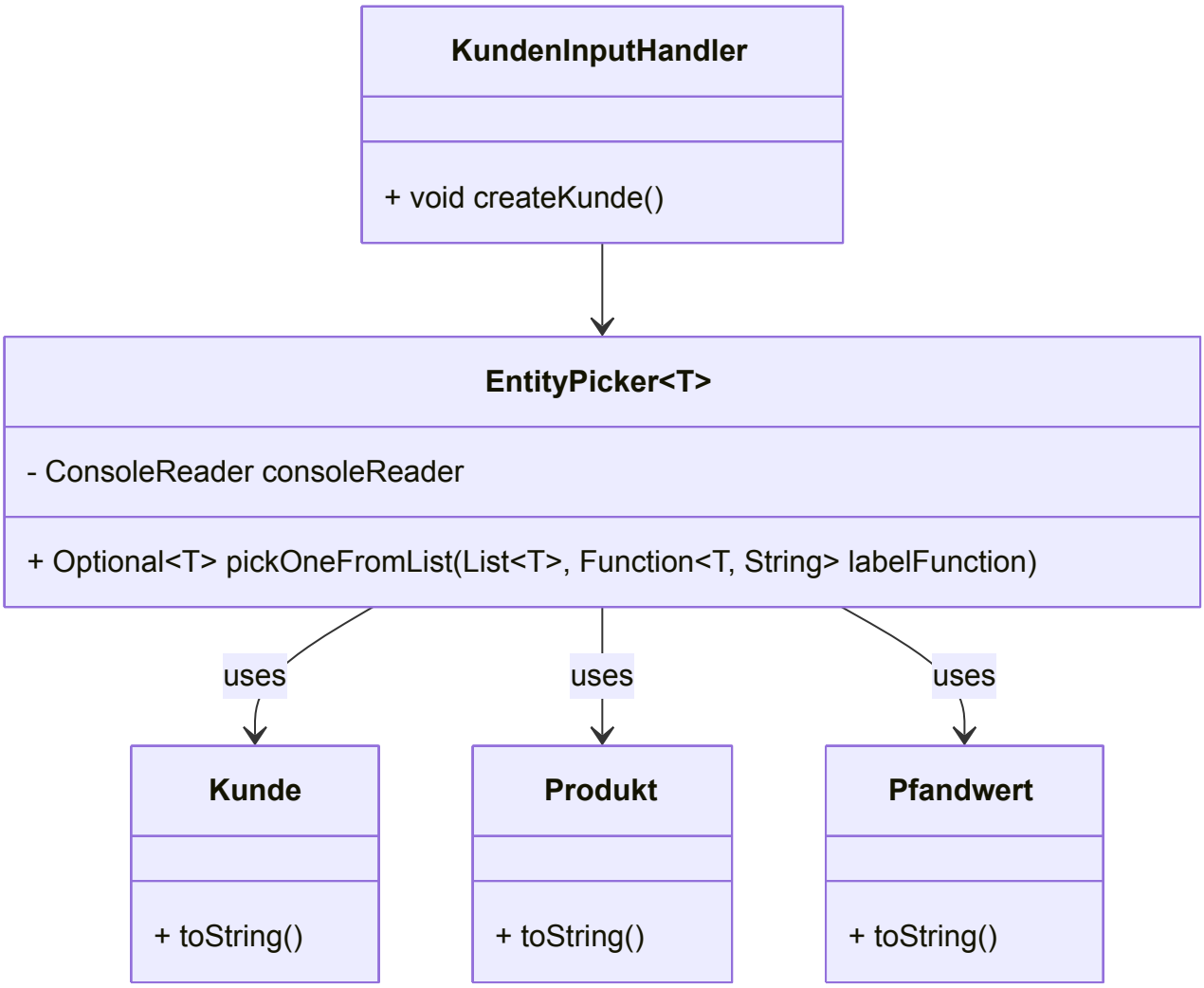


5.2 Strategy Pattern

Das Strategy Pattern wird im Projekt verwendet, um verschiedene Interaktionen mit Benutzereingaben flexibel zu gestalten. Beispielsweise nutzt `EntityPicker<T>` das

Strategy-Prinzip, indem es die Anzeige und Auswahl von Objekten generisch hält und das Labeling über eine **konfigurierbare Strategie** (`Function<T, String>`) erlaubt.

So kann z. B. ein Produkt anders dargestellt werden als ein Kunde, **ohne dass der Auswahlmechanismus verändert werden muss**.



6. Domain Driven Design (DDD)

Ab dem Start der Entwicklung der Projektes wurde die Domäne ins Zentrum der Entwicklung gestellt und das möglichst der reale Ablauf abgebildet wird.

6.1 Entities

Unserer Grund etwas als ein Entity abzubilden ist, wenn die Daten erhalten werden müssen und dabei kein Änderungsverlauf der Daten gefordert ist, bzw. wenn Änderungen eigentlich nicht vorgesehen sind. Am Beispiel eines Nutzers ist es nicht nötig die

Änderungen seines Vornames zu tracken. Als beispiel für wenn ein Verlauf gefordert ist, ist ein Preis für ein Produkt. Im Vollgendem werden unsere eizelnen Entities beschriebn. Die Beschreibung umfasst das Gegenstück, welches aus der echten Welt abgebildet wird und welche Daten gespeichert werden,

- Kunden
  - Ein Kunde beschreibt eine natürliche Person, welche in dem Getränkesystem einkaufen kann.
  - Gepseichert werden Name, Vorname und die Email.
- Produkte
  - Produkte beschreiben alles was verkauft werden kann. Im normalfall sind das Getränke.
  - Dabei werden Name, Beschreibung, Kategorie gespeichert.
  - Zusätzlich dazu werden Verweise zu dem zugehörigem Pfand und dem zugehörigem derzeitigem Preis gespeichert.
- Bestellungen
  - Bestellungen beschreiben eben eine Bestellung welche von einem Kunden aufgegeben wird.
  - Diese ist identifizierbar durch eien Rechnungsnummer und einen Zeitstempel.
  - Es werden darin verweise auf den Kunden und die Bestellten Produkte verwiesen.
  - Die Bestellten Produkte werden mit der Anzahl dieser und dem abgerechnetem Preis gespeichert. (Siehe Bestellprodukt)
- Zahlungsvorgang
  - Ein Zahlungsvorgang beschreibt das Ausgleichen der Schulden, welche durch Rechnungen erzeugt werden.
  - Dabei wird für jede Zahlung der Zahlungsweg, Betrag und der Zeitpunkt gespeichert.
  - Zusätzlich wird ein verweis auf den Kunden gespeichert.
- Lieferungen
  - Lieferungen ermöglichen es alle Produkte welche vom Getränktesystem gekauft werden zu erfassen und dadurch den Lagerbestandzu errechnen.
  - Dafür wird für jedes Produkt die Menge gespeichert, welche gekauft wird pro Lieferung.

6.2 Valueobjects

Valueobjects haben bei uns oft den Zweck um einen Verlauf darzustellen. Diese sind möglichst kein um keine Daten redundant bei vielen Änderungen zu speichern.

- Preis
  - Ein Preis ist immer mit einem Objekt verbunden welches einen Preis haben kann. Dies ist in unserem Fall ein Produkt.
  - Darin wird eine Referenz auf das Produkt gespeichert. Zusätzlich wird der Zeitpunkt zu welchem der Preis gesetzt wird gespeichert und wie hoch der Preis ist.
- Pfandwert
  - Ein Pfandwert beschreibt beispielsweise eine Falsche oder einen Kasten, welcher mit Pfand abgerechnet wird.
  - Dafür wird eine Beschreibung eine Zeitpunkt der Erstellung und die Höhe des Pfandes gespeichert.
  - Obwohl der Pfand einen Preis hat, wird nicht ein Preis-Valueobject benutzt, da der Pfandwert sich normalerweise nicht ändert.
- Bestellungsprodukt
  - Ein Bestellungssprodukt beschreibt eine Bestellposition, d.h. ein Produkt und die zugehörige Menge.
  - Dies ist als Valueobject implementiert, da man für eine Position nachvollziehen kann wie diese geändert wurde und eventuell Bedienfehler oder fehlende Wahre gut und nachvollziehbar verbessert werden kann.

6.3 Aggregates

Aggregate werden als Zusammenfassung von Entities und Valueobjects genutzt, um an zusammenhängende Daten einfach heranzukommen.

- CustomerDashboard
  - Das CustomerDashboard fasst alle Daten zusammen welche sich auf einen Kunden beziehen und welche den Kunden Interessieren können.
  - Diese Informationen bestehen aus dem Kunden Entity, alle Bestellungen des Kunden und alle Zahlungsvorgänge.
  - Zusätzlich wird der gesammt gezahlte Betrag und der gesammte gekaufte Betrag vorberechnet. Daraus kann dann auch die noch geschuldete Summe berechnet werden.
- ProductInformation
  - Die ProduktInformation beschreibt alle Informationen welche ein Produkt bestreffen.
  - Gepspeichert werden: Eine Referenz auf das Produkt, die derzeitigen Pfandwerte des Produktes und der verlauf des Preises.

- Vorberechnet werden die Anzahl welche bestellt wurden und wie viele noch auf Vorrat sind.

6.4 Repositories

Grundsätzlich lässt sich unsere Datenbasis in zwei verschiedene Sub-Domänen unterscheiden. Daten bezüglich der Kunden und der des Getränkesystems. Nach dieser Stuktur wurden zwei Repositories erstellt. Das Kundenrepository umfasst folgende Daten:

- Kundeninformionen, d.h. die Kunden-Entities
- Zahlungsvorgänge, d.h. die Zahlungsvorgänge-Entities.

Dazu sind Methoden vorgesehen diese Daten hinzuzufügen und auszulesen.

Das Getränkerepository umfasst die restlichen Daten:

- Produkte
- Pfandwerte
- Preise
- Lieferungen
- Bestellungen
- Bestellpositionen Dieses Repository umfasst zusäzlich Methoden um diese Daten hinzuzufügen, auszulesen und vorallem schon nach filtervorgaben Auszulesen. Beispielsweise, dass nur ein User nach seiner Email gesucht werden kann.

7. Unit Tests

7.1 Zehn Unit Tests - Tabelle

Unit Test Name	Beschreibung
testBuildValidKunde	Testet, ob der `KundeBuilder` ein gültiges `Kunde`-Objekt erstellt.
testBuildThrowsExceptionWhenFieldsAreNull	Überprüft, ob der `KundeBuilder` eine Exception wirft, wenn erforderliche Felder null sind.
testBuildValidProdukt	Testet, ob der `ProduktBuilder` ein gültiges `Produkt`-Objekt erstellt.

Unit Test Name	Beschreibung
testBuildThrowsExceptionWhenProduktFieldsAreNull	Überprüft, ob der `ProduktBuilder` eine Exception wirft, wenn erforderliche Felder null sind.
testBuildWithPreisAndPfandwerte	Testet, ob der `ProduktBuilder` ein `Produkt` mit gültigen `Preis`- und `Pfandwert`-Objekten erstellt.
testGetAllKundenAndAdd	Testet, ob der `KundenUsecases` alle Kunden korrekt zurückgibt und neue Kunden hinzufügen kann.
testGetKundenBalance	Überprüft, ob die `getKundenBalance`-Methode die korrekte Balance für einen Kunden zurückgibt.
testAddZahlungsvorgang	Testet, ob der `KundenUsecases` einen neuen `Zahlungsvorgang` korrekt hinzufügen kann.
testPreisCreation	Überprüft, ob ein `Preis`-Objekt korrekt erstellt wird und die Attribute richtig gesetzt sind.
testPreisEquality	Testet, ob zwei `Preis`-Objekte mit identischen Attributen als gleich betrachtet werden.

7.2 ATRIP

Die ATRIP-Prinzipien (Automatisch, Tiefgehend, Reproduzierbar, Unabhängig, Professionell) sind entscheidend, um qualitativ hochwertige Tests sicherzustellen. Hier ist eine detaillierte Anwendung dieser Prinzipien mit Beispielen:

1. **Automatic**: Tests sollten ohne manuelle Eingriffe ausführbar sein. Zum Beispiel verwendet der Test `testHandleCreateKundeInput` in `KundenInputHandler` gemockte Abhängigkeiten wie `ConsoleReader` und `KundenUsecases`, sodass der Test automatisch und ohne Benutzereingabe ablaufen kann. Ein schlechtes Beispiel wäre ein Test, der die Ausführung pausiert, um auf eine manuelle Eingabe zu warten, etwa durch einen Aufruf wie `System.in.read()`, was die Automatisierung bricht.
2. **Thorough**: Tests sollten alle relevanten Szenarien abdecken, einschließlich Randfällen. Zum Beispiel stellt `testBuildThrowsExceptionWhenFieldsAreNull` sicher, dass der `KundeBuilder` eine Ausnahme wirft, wenn erforderliche Felder fehlen – ein kritisches Fehlerszenario. Ein schlechtes Beispiel wäre ein Test, der nur den "Happy Path" überprüft, z. B. ob ein `Kunde` erfolgreich erstellt wird, ohne ungültige Eingaben oder Randfälle zu testen.
3. **Repeatable**: Tests sollten unabhängig von der Umgebung konsistente Ergebnisse liefern. Zum Beispiel mockt `testGetKundenBalance` die `KundenUsecases`, um einen

festen Kontostand zurückzugeben, wodurch sichergestellt wird, dass der Test bei jedem Durchlauf gleich funktioniert. Ein schlechtes Beispiel wäre ein Test, der von externen Systemen wie einer Live-Datenbank oder einem Netzwerk abhängt, bei denen die Ergebnisse durch äußere Faktoren variieren können.

4. **Independent**: Tests sollten nicht von der Ausführungsreihenfolge oder gemeinsam genutztem Zustand abhängen. Zum Beispiel isoliert `testAddZahlungsvorgang` die Logik des Hinzufügens einer Zahlung durch das Mocken aller externen Abhängigkeiten, sodass er unabhängig von anderen Tests ausgeführt werden kann. Ein schlechtes Beispiel wäre ein Test, der sich auf einen globalen Zustand verlässt, der von einem vorherigen Test verändert wurde – das führt zu instabilen Ergebnissen, wenn sich die Reihenfolge ändert.
5. **Professional**: Tests sollten gut strukturiert, lesbar und wartbar sein. Zum Beispiel verbessert die Verwendung beschreibender Methodennamen wie `testBuildValidKunde` und klarer Assertions die Lesbarkeit und Professionalität. Ein schlechtes Beispiel wäre ein Test mit vagen Namen wie `test1` oder ohne aussagekräftige Assertions, was das Verständnis und Debugging erschwert.

Durch die Einhaltung dieser Prinzipien werden Tests zuverlässig, wartbar und effektiv bei der Validierung des Verhaltens einer Anwendung.

7.3 Code Coverage

Um die Qualität und Korrektheit der Anwendung sicherzustellen, wurden für die zentralen Use Cases Unit Tests geschrieben. Ein zentrales Ziel war es, eine möglichst hohe Testabdeckung (Code Coverage) in den **Anwendungsfällen (Usecases)** zu erreichen, da diese die Kernlogik des Systems abbilden. Die Code Coverage wurde mit Hilfe des Maven-Plug-ins `jacoco-maven-plugin` gemessen.

Die Coverage-Metrik gibt an, wie viel Prozent des Quellcodes durch Tests tatsächlich ausgeführt werden. Dabei gilt: Eine hohe Abdeckung allein garantiert keine Fehlerfreiheit, ist jedoch ein wichtiger Indikator für die Testtiefe und -qualität. Wichtig ist auch zu erwähnen das Tests nie beweisen können, das Software Fehlerfrei ist nur, dass ein Fehler vorliegt.

Die Tests konzentrieren sich hauptsächlich auf die Anwendungsschicht (Layer `getraenkeapplication`). Diese enthält die Klassen:

- `GetraenkeUsecases`
- `KundenUsecases`

Die Coverage-Berichte zeigen, dass diese Klassen zu einem großen Teil durch automatisierte Tests abgedeckt sind. Besonders häufig getestete Methoden sind:

- `createKunde(...)`, `getKundenBalance(...)`
- `addProdukt(...)`, `getStockAmountForProdukt(...)`, `setPriceForProdukt(...)`

1-getraenke-adapters

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
de.nyg.adapters.asegetraenke.repository		0 %		0 %	46	46	64	64	42	42	3	3
de.nyg.adapters.asegetraenke.console		78 %		61 %	26	66	91	338	2	26	0	3
de.nyg.adapters.asegetraenke		0 %		n/a	2	2	19	19	2	2	1	1
de.nyg.adapters.asegetraenke.console.utils		82 %		83 %	6	21	14	57	4	15	1	4
de.nyg.adapters.asegetraenke.console.consolefunctionmapping		93 %		70 %	4	16	4	29	1	11	0	2
Total	729 of 1.935	62 %	44 of 110	60 %	84	151	192	507	51	96	5	13

2-getraenke-application

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
de.nyg.application.asegetraenke.builder		0 %		0 %	25	25	36	36	14	14	2	2
de.nyg.application.asegetraenke.aggregate		0 %		0 %	6	6	24	24	3	3	2	2
de.nyg.application.asegetraenke		88 %		83 %	11	43	8	69	10	40	1	3
Total	328 of 672	51 %	29 of 34	14 %	42	74	68	129	27	57	5	7

3-getraenke-domain

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
de.nyg.domain.asegetraenke.repository		62 %		33 %	27	45	27	80	22	39	0	2
de.nyg.domain.asegetraenke.entities		52 %		70 %	21	39	29	77	19	34	1	5
de.nyg.domain.asegetraenke.valueobjects		72 %		58 %	7	18	7	34	2	12	0	3
de.nyg.domain.asegetraenke.util		30 %		n/a	2	4	2	6	2	4	2	3
de.nyg.domain.asegetraenke		0 %		n/a	2	2	3	3	2	2	1	1
de.nyg.domain.asegetraenke.usecases		0 %		n/a	1	1	1	1	1	1	1	1
Total	378 of 926	59 %	16 of 34	52 %	60	109	69	201	48	92	5	15

7.4 Fakes und Mocks

7.4.1 Mock-Objekt: `Repo`

Niklas

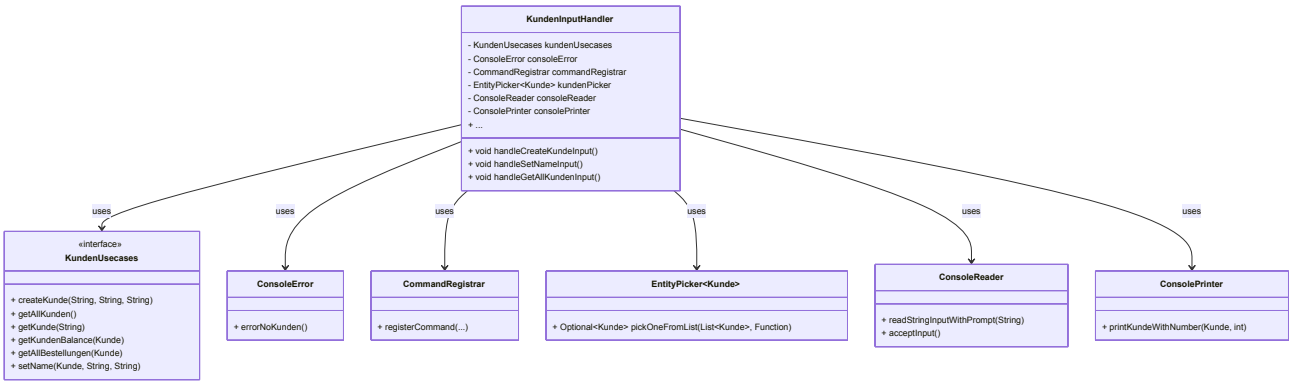
7.4.2 `KundenInputHandler`

In den Tests des `KundenInputHandler` werden alle externen Abhängigkeiten durch **Mock-Objekte** ersetzt. Dies ermöglicht eine **isolierte Testbarkeit** der Benutzereingabe-Logik, ohne auf die reale Implementierung der Geschäftslogik oder auf tatsächliche Benutzereingaben angewiesen zu sein. Durch das mocken der der Abhänigkeiten ist der Test vollkommen isoliert von Benutzereingaben oder anderen Methoden.

Zu den gemockten Abhängigkeiten zählen:

- `KundenUsecases` – zentrale Geschäftsanwendungsfälle
- `ConsoleReader` – zum Einlesen von Benutzereingaben
- `ConsolePrinter` – für formatierte Ausgaben
- `ConsoleError` – für Fehlerausgaben

- `EntityPicker<Kunde>` – zur Auswahl von Kunden
- `CommandRegistrar` – zur Registrierung von Befehlen




```
@Test
public void testHandleCreateKundeInput() {
    when(mockedKundeusecases.createKunde(anyString(),
anyString(),anyString())).thenReturn(null);
    when(mockedConsoleReader.readStringInputWithPrompt("Name:
")).thenReturn("Max");
    when(mockedConsoleReader.readStringInputWithPrompt("Nachname:
")).thenReturn("Mustermann");
    when(mockedConsoleReader.readStringInputWithPrompt("E-Mail:
")).thenReturn("max@example.com");
    when(mockedConsoleReader.acceptInput()).thenReturn(true);

    kundenInputHandler.handleCreateKundeInput();

    verify(mockedKundeusecases).createKunde("Max", "Mustermann",
"max@example.com");
}
```

Beschreibung
Das Mock-Objekt `KundenUsecases` simuliert die Kundenlogik, um den `KundenInputHandler` **unabhängig von der tatsächlichen Business-Logik** testen zu können. Auch Eingaben über `ConsoleReader` und die Auswahl über `EntityPicker` werden gemockt, um automatisierte Tests ohne Benutzereingabe zu ermöglichen. So kann gezielt überprüft werden, ob der Handler auf Eingaben korrekt reagiert und die erwarteten Usecase-Methoden wie `createKunde()` oder `setName()` aufruft – **ohne echte Implementierungen auszuführen**.

Ziel
Getestet wird ausschließlich die **Verarbeitung der Eingaben und Steuerung des Ablaufs**

innerhalb des `KundenInputHandler`. Die tatsächliche Geschäftslogik (wie z. B. Datenbankzugriffe) wird **nicht** aufgerufen.

8. Refactoring

8.1 Code Smells

8.1.1 Large Class

Code Beispiel ist die Klasse `ConsoleUtils` in der Adapterschicht. Auf dem Stand `Commit Stand: ec8012db4f8473d9cf1cad5178f139e92e3f416f`. Diese ist eindeutig zu lange und ist für mehrere Aufgaben zuständig.

```
package de.nyg.adapters.asegetraenke.console.Utils;


import java.util.ArrayList;
...


public class ConsoleUtils {
    private final String NOPRODUKTMESSAGE = "There are no Product/s
found";
    private final String NOPFANDWERTMESSAGE = "There are no Pfandwert/s
found";
    private final String NOKUNDEMESSAGE = "There are no Customer/s
found";

    private final Scanner scanner;
    private final GetraenkeUsecases getraenkeUseCases;
    private final KundenUsecases kundeUseCases;

    public ConsoleUtils(Scanner scanner, GetraenkeUsecases
getraenkeUseCases, KundenUsecases kundeUseCases) {
        this.scanner = scanner;
        this.getraenkeUseCases = getraenkeUseCases;
        this.kundeUseCases = kundeUseCases;
    }

    public Optional<Produkt> pickOneProductFromAllProducts() {
        Iterable<Produkt> productVec =
getraenkeUseCases.getAllProducts();
        List<Produkt> productList =
```

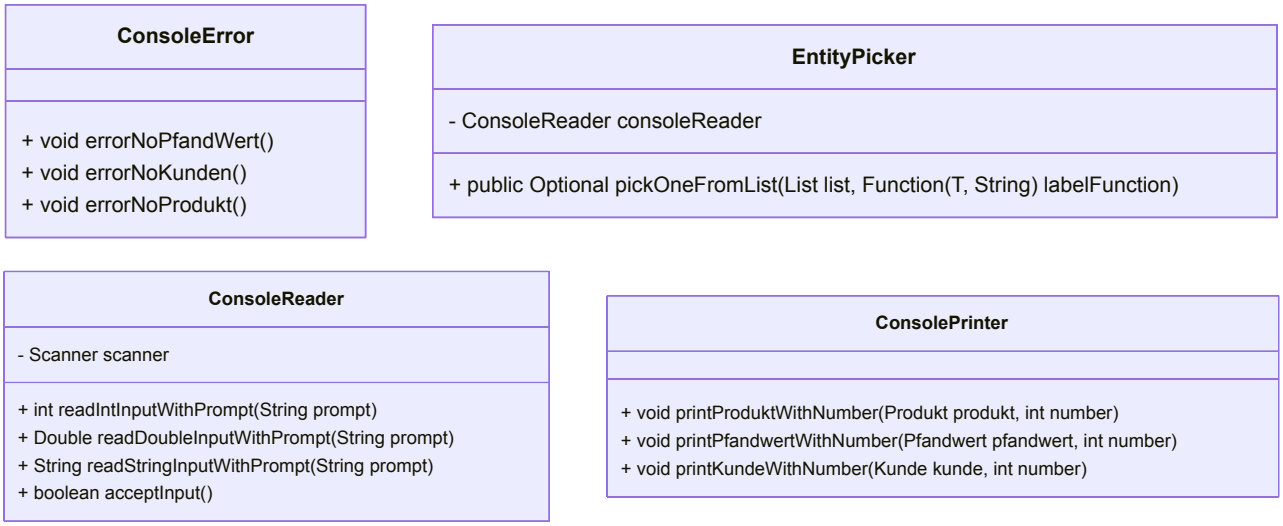
```
StreamSupport.stream(productVec.spliterator(), false)
                                    .collect(Collectors.toList());
        if(productList.isEmpty()){
            return Optional.empty();
        }

        int count = 1;
        for(Produkt product : productList) {
            printProduktWithNumber(product, count);
        }
        int indexProdukt = 0;
        while (true) {
            indexProdukt = readIntInputWithPrompt("Which Customername do
you want to change? Enter the Number: ");
            if(indexProdukt < productList.size()+1 && indexProdukt > 0){
                break;
            }
            System.out.println("Something went wrong the "+ indexProdukt
+  " is not in the list");
        }
        Produkt produkt = productList.get(indexProdukt-1);
        System.out.println("Chosen Produkt: "+ produkt.toString());
        return Optional.of(productList.get(indexProdukt-1));
    }

    public Optional<Pfandwert> pickOnePfandwertFromAllPfandwerts() {
        Iterable<Pfandwert> pfandwertVec =
getraenkeUseCases.getAllPfandwerte();
        List<Pfandwert> pfandwertList =
StreamSupport.stream(pfandwertVec.spliterator(), false)
                                    .collect(Collectors.toList());
        if(pfandwertList.isEmpty()){
            return Optional.empty();
        }

        int count = 1;
        for(Pfandwert pfandwert : pfandwertList) {
            printPfandwertWithNumber(pfandwert, count);
        }
        int indexProdukt = 0;
        while (true) {
            indexProdukt = readIntInputWithPrompt("Which Customername do
you want to change? Enter the Number: ");
            if(indexProdukt < pfandwertList.size()+1 && indexProdukt >
0){
                break;
```

```
            }
            System.out.println("Something went wrong the "+ indexProdukt
+  " is not in the list");
        }
        Pfandwert pfandwert = pfandwertList.get(indexProdukt-1);
        System.out.println("Chosen Produkt: "+ pfandwert.toString());
        return Optional.of(pfandwertList.get(indexProdukt-1));
    }

    public Optional<Kunde> pickOneUserFromAllUsers(){
        Iterable<Kunde> kundenOptVec =
this.kundeUseCases.getAllKunden();
        List<Kunde> kundenList = new ArrayList<Kunde>();
        kundenList = StreamSupport.stream(kundenOptVec.spliterator(),
false).collect(Collectors.toList());
        int count = 1;
        for(Kunde kunde : kundenList) {
            printKundeWithNumber(kunde, count);
        }
        int indexCustomer = 0;
        while (true) {
            indexCustomer = readIntInputWithPrompt("Which Customername
do you want to change? Enter the Number: ");
            if(indexCustomer < kundenList.size()+1 && indexCustomer > 0)
{
                break;
            }
            System.out.println("Something went wrong the "+
indexCustomer +  " is not in the list");
        }
        return Optional.of(kundenList.get(indexCustomer-1));
    }

    public void printProduktWithNumber(Produkt produkt, int number){
        System.out.println(number + ". "+ produkt.toString());
    }

    public void printPfandwertWithNumber(Pfandwert pfandwert, int
number){
        System.out.println(number + ". "+ pfandwert.toString());
    }

    public void printKundeWithNumber(Kunde kunde, int number){
        System.out.println(number + ". "+ kunde.toString());
    }
```

```
    public int readIntInputWithPrompt(String prompt){
        System.out.print(prompt);
        while(true){
            String input = this.scanner.nextLine();
            try{
                int inputCastInt = Integer.parseInt(input);
                return inputCastInt;
            }catch(Exception e){
                System.out.println("The input: "+ input+ " can not be
translated into a number");
            }
        }
    }

    public Double readDoubleInputWithPrompt(String prompt){
        System.out.print(prompt);
        while(true){
            String input = this.scanner.nextLine();
            try{
                double inputCastInt = Double.parseDouble(input);
                return inputCastInt;
            }catch(Exception e){
                System.out.println("The input: "+ input+ " can not be
translated into a number");
            }
        }
    }

    public String readStringInputWithPrompt(String ptompString){
        System.out.print(ptompString);
        return this.scanner.nextLine();
    }

    public boolean acceptInput(){
    ...
    }

    public void errorNoPfandWert() {
        System.out.println(NOPFANDWERTMESSAGE);
    }

    public void errorNoKunden() {
        System.out.println(NOKUNDEMESSAGE);
    }

    public void errorNoProdukt() {
```

```
            System.out.println(NOPRODUKTMESSAGE);
        }
    }
```

Um die Länge und Komplexität dieser Klasse zu reduzieren wurde sie in 4 Klassen aufgeteilt.



Dies macht die einzelnen Klassen einfacher zu warten und spezifiziert die eigentliche Aufgabe, als alles in einem Überbegriff `Utils` zu vereinen.

8.1.2 Duplicate Code

Am Beispiel des `EntityPicker` dieser vereint 3 Methoden die jeweils eine sehr ähnliche Aufgabe ausführen.

```
    public Optional<Produkt> pickOneProductFromAllProducts() {
        Iterable<Produkt> productVec =
getraenkeUseCases.getAllProducts();
        List<Produkt> productList =
StreamSupport.stream(productVec.spliterator(), false)
                                    .collect(Collectors.toList());
        if(productList.isEmpty()){
            return Optional.empty();
        }

        int count = 1;
        for(Produkt product : productList) {
            printProduktWithNumber(product, count);
```

```
        }
        int indexProdukt = 0;
        while (true) {
            indexProdukt = readIntInputWithPrompt("Which Customername do
you want to change? Enter the Number: ");
            if(indexProdukt < productList.size()+1 && indexProdukt > 0){
                break;
            }
            System.out.println("Something went wrong the "+ indexProdukt
+  " is not in the list");
        }
        Produkt produkt = productList.get(indexProdukt-1);
        System.out.println("Chosen Produkt: "+ produkt.toString());
        return Optional.of(productList.get(indexProdukt-1));
    }

    public Optional<Pfandwert> pickOnePfandwertFromAllPfandwerts() {
        Iterable<Pfandwert> pfandwertVec =
getraenkeUseCases.getAllPfandwerte();
        List<Pfandwert> pfandwertList =
StreamSupport.stream(pfandwertVec.spliterator(), false)
                                    .collect(Collectors.toList());
        if(pfandwertList.isEmpty()){
            return Optional.empty();
        }

        int count = 1;
        for(Pfandwert pfandwert : pfandwertList) {
            printPfandwertWithNumber(pfandwert, count);
        }
        int indexProdukt = 0;
        while (true) {
            indexProdukt = readIntInputWithPrompt("Which Customername do
you want to change? Enter the Number: ");
            if(indexProdukt < pfandwertList.size()+1 && indexProdukt >
0){
                break;
            }
            System.out.println("Something went wrong the "+ indexProdukt
+  " is not in the list");
        }
        Pfandwert pfandwert = pfandwertList.get(indexProdukt-1);
        System.out.println("Chosen Produkt: "+ pfandwert.toString());
        return Optional.of(pfandwertList.get(indexProdukt-1));
    }
```

```
    public Optional<Kunde> pickOneUserFromAllUsers(){
        Iterable<Kunde> kundenOptVec =
this.kundeUseCases.getAllKunden();
        List<Kunde> kundenList = new ArrayList<Kunde>();
        kundenList = StreamSupport.stream(kundenOptVec.spliterator(),
false).collect(Collectors.toList());
        int count = 1;
        for(Kunde kunde : kundenList) {
            printKundeWithNumber(kunde, count);
        }
        int indexCustomer = 0;
        while (true) {
            indexCustomer = readIntInputWithPrompt("Which Customername
do you want to change? Enter the Number: ");
            if(indexCustomer < kundenList.size()+1 && indexCustomer > 0)
{
                break;
            }
            System.out.println("Something went wrong the "+
indexCustomer +  " is not in the list");
        }
        return Optional.of(kundenList.get(indexCustomer-1));
    }
```

Diese Methoden können generisch formuliert werden um die Wiederholungen selber Logik zu vermeiden und das anpassen im späteren Verlauf einheitlich zu ändern.

```
public Optional<T> pickOneFromList(List<T> list, Function<T, String>
labelFunction) {
    if (list.isEmpty()) {
        System.out.println("Keine Einträge verfügbar.");
        return Optional.empty();
    }

    for (int i = 0; i < list.size(); i++) {
        System.out.println((i + 1) + ": " +
labelFunction.apply(list.get(i)));
    }

    int choice = consoleReader.readIntInputWithPrompt("Bitte Nummer
eingeben: ");

    if (choice >= 1 && choice <= list.size()) {
        return Optional.of(list.get(choice - 1));
```

```
    }

    return Optional.empty();
}
```

8.2 Refactorings

7.2.1 Replace Parameter with Builder

Begründung:
In der ursprünglichen Implementierung wurde bei der Erstellung eines Kundenobjekts direkt mit mehreren Parametern gearbeitet. Durch das Refactoring wurde der Builder Pattern eingeführt, um die Objekterstellung klarer zu strukturieren und zu kapseln. Das verbessert sowohl die Lesbarkeit als auch die Erweiterbarkeit, z. B. bei zusätzlichen Attributen.

Code vorher:

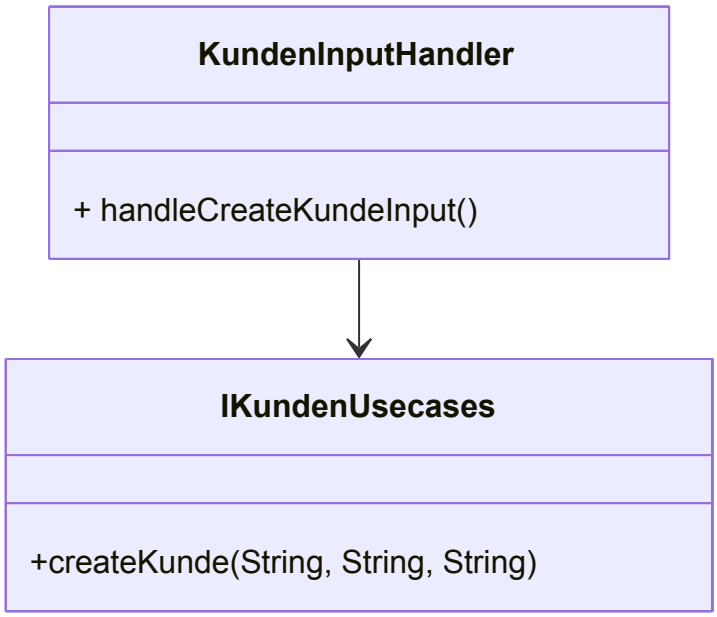
```
kundeUseCases.createKunde(name, nachname, email);
```

Code nachher:

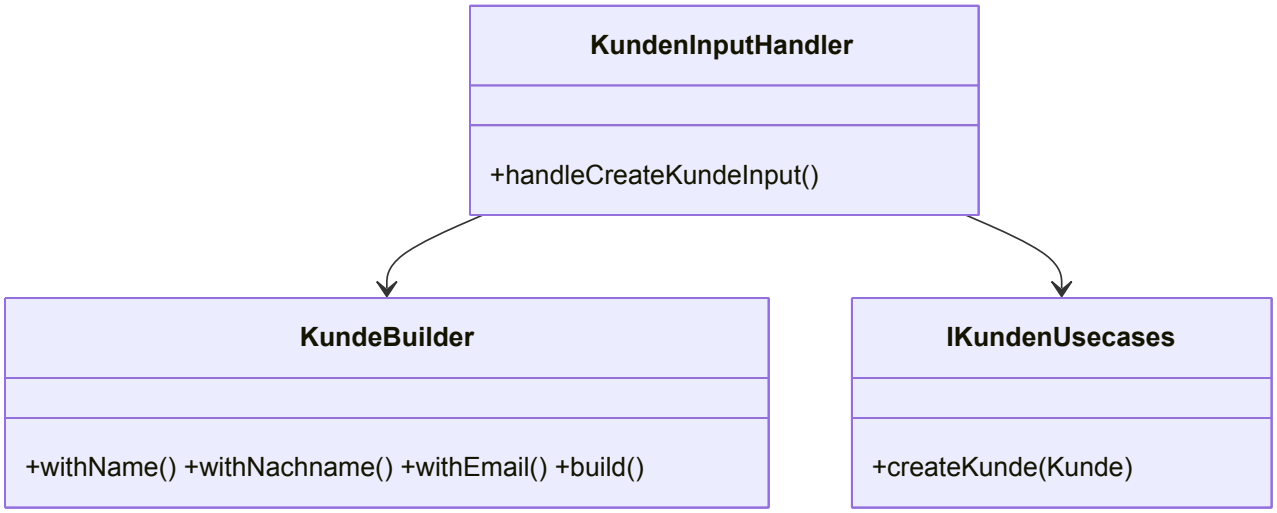
```
Kunde kunde = new KundeBuilder()
    .withName(name)
    .withNachname(nachname)
    .withEmail(email)
    .build();

kundeUseCases.createKunde(kunde);`
```

Commit: 5b8c6a97f985ae77d85279b8d977d2ae98857c00

UML vorher:



UML nachher:



7.2.2 Extract Method

Begründung:
In der Methode wie z.B. `handleAddProduktInput()` war die gesamte Benutzerinteraktion und Objekterstellung inline implementiert. Durch das Refactoring wurde ein Teil der Logik – z. B. das Einlesen der Produktinformationen – in eine eigene Methode ausgelagert. Das verbessert die Übersichtlichkeit und ermöglicht einfachere Wiederverwendung und Tests.

Code vorher:

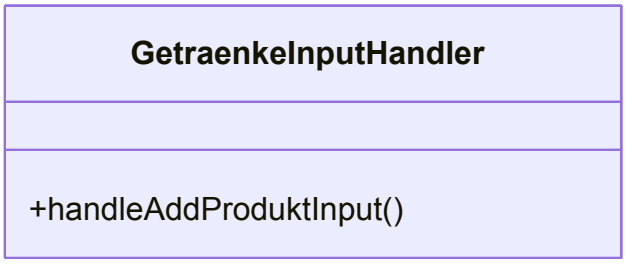
```
public void handleAddProduktInput() {
    String name = ...;
    String beschreibung = ...;
    ...
    Produkt produkt = new Produkt(name, beschreibung, kategorie);
... }
```

Code nachher:

```
public void handleAddProduktInput() {
    Produkt produkt = readProduktInput();
    ...
    }  private Produkt readProduktInput() {     String name = ...;
String beschreibung = ...;     ...     return new Produkt(name,
beschreibung, kategorie); }`
```

Commit:
`a58c90b` – Extraktion der Produkt-Erstellung

UML vorher:



UML nachher: