

# ThirstyCalc

Paul Bader, Sarah Ficht & Kayra Güler

---

## Einführung (S)

- Übersicht über die Applikation (1 P.)
  - Starten der Applikation (1 P.)
  - Technischer Überblick (2 P.)
- 

## Übersicht

- Getränkeabrechnung
  - Mit eigenem Nutzerprofil und jeweiligem Guthaben lassen sich Getränke abrechnen
  - Zweck ist vereinfachte Abrechnung von gemeinschaftlichen Getränkevorräten durch digitales System
- 

## Starten der Applikation

- `git clone https://github.com/Puggingtons/Getraenkeabrechnung.git`
  - `./gradlew shadowJar`
  - Die gebaute `.jar` ist dann in `build/libs` zu finden.
- 

## Technischer Überblick

- Java (21+)
  - Gradle mit Kotlin DSL (lauffähige Version der Software)
  - SonarCloud (für statische Codeanalyse -> Qualität)
  - Github Actions (Cloc, Tests, Sonar)
- 

## Softwarearchitektur (Paul)

---

## Gewählte Architektur (4P)

- [In der Vorlesung wurden Softwarearchitekturen vorgestellt. Welche Architektur wurde davon umgesetzt? Analyse und Begründung inkl. UML

der wichtigsten Klassen, sowie Einordnung dieser Klassen in die gewählte Architektur]

Gewählt wurde die **Clean Architecture**. Geworden ist es ein **Monolith**.

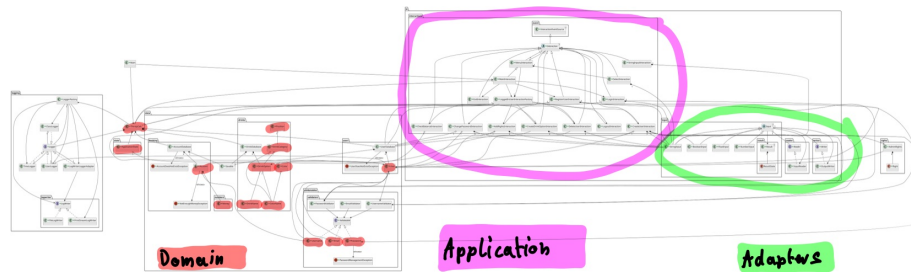


Figure 1: architecture\_clean.jpg

## Wichtigste Klassen –

### Domain Code (1P)

- [kurze Erläuterung in eigenen Worten, was Domain Code ist – 1 Beispiel im Code zeigen, das bisher noch nicht gezeigt wurde]

Domaincode ist die Kern-Business-Logik und ist frei von Abhängigkeiten. Folgender Code ist aus der Klasse `Account.java`:

### Analyse der Dependency Rule (3P)

- [In der Vorlesung wurde im Rahmen der ‘Clean Architecture’ die s.g. Dependency Rule vorgestellt. Je 1 Klasse zeigen, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML (mind. die betreffende Klasse inkl. der Klassen, die von ihr abhängen bzw. von der sie abhängt) und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]

“Abhängigkeiten immer von außen nach innen”

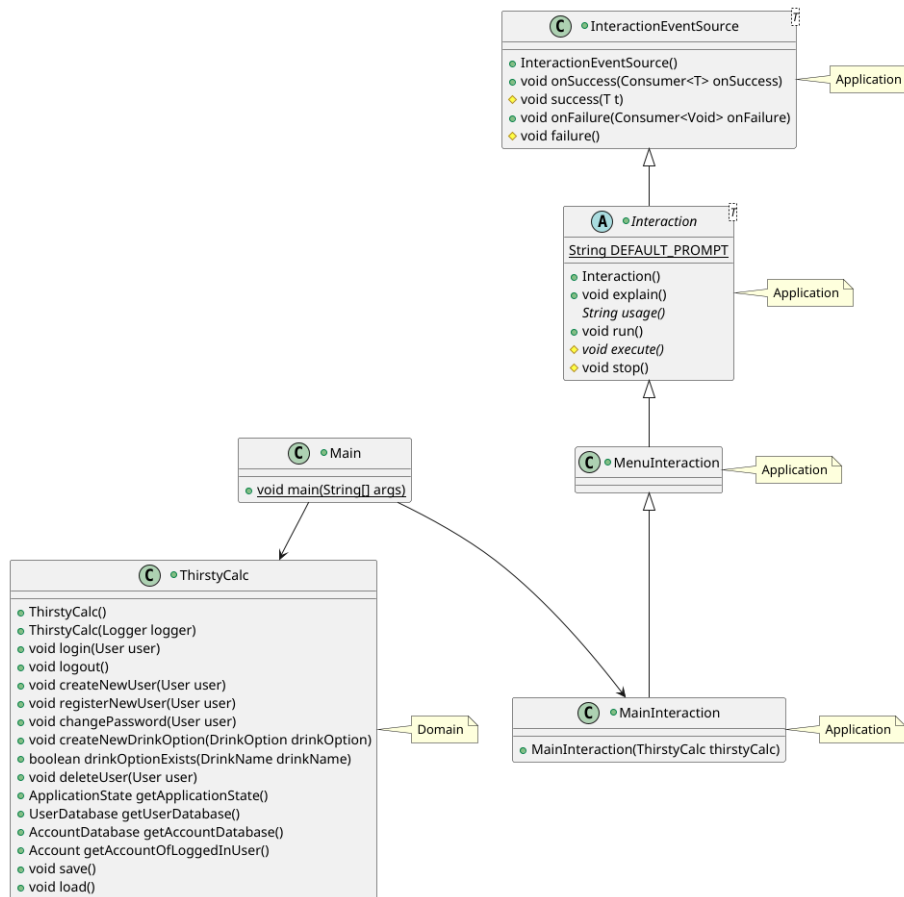


Figure 2: important\_classes.png

```

Account.java

public void deposit(Money amount) {
    balance = balance.add(amount);
}

public Money charge(Money amount) throws NotEnoughMoneyException {
    if (amount.getAmount().compareTo(balance.getAmount()) > 0) {
        throw new NotEnoughMoneyException("Not enough money in account! (available: " + balance + ",
        charged: " + amount + ")");
    }

    balance = balance.subtract(amount);
    return amount;
}

```

Figure 3: domain\_code.png

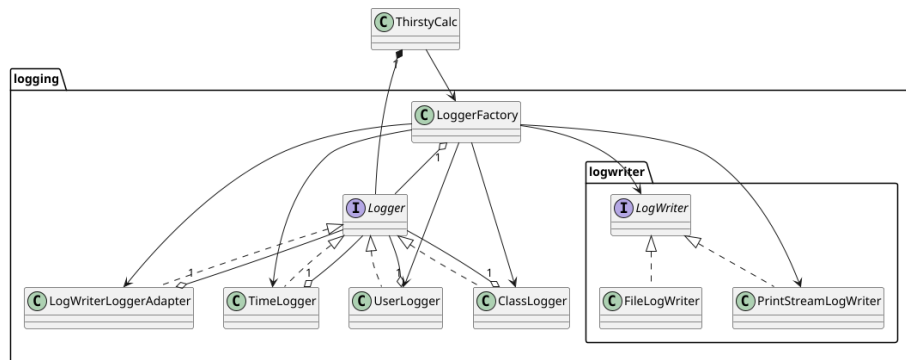


Figure 4: positive\_dependency\_rule.png

**Positiv-Beispiel: Dependency Rule** note: Die Klasse `LoggerFactory` erfüllt hier die Dependency Rule. Die Klasse `ThirstyCalc` hängt von ihr ab. Sie ist nicht von einer äußeren Klasse (hier `ThirstyCalc`) abhängig und hängt nur von inneren Klassen (hier `Logger`, allen Implementierungen von `Logger` sowie `LogWriter` und `PrintStreamLogWriter`) ab.

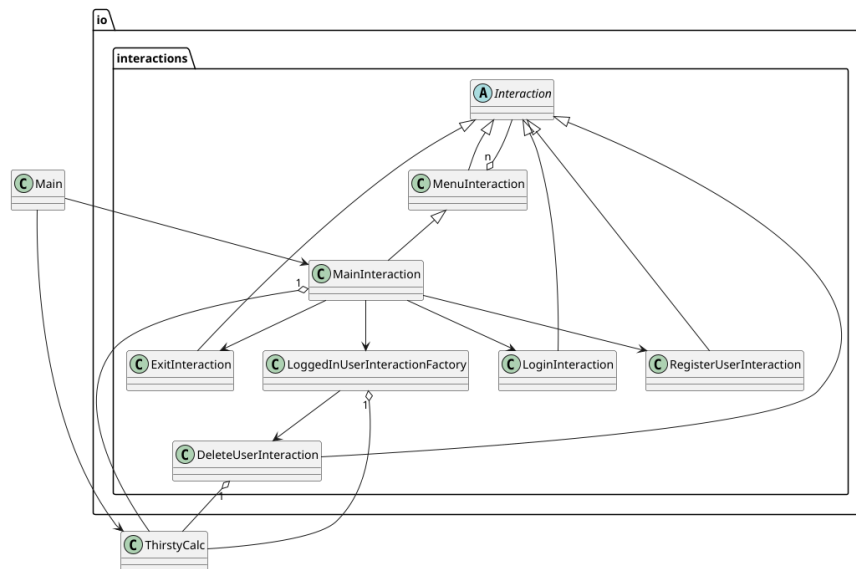


Figure 5: negative\_dependency\_rule.png

**Negativ-Beispiel: Dependency Rule** note: Die Klasse `MainInteraction` erfüllt die Dependency Rule nicht. Die `Main` Klasse ist von `MainInteraction` ab-

hängig. Die Klasse `MainInteraction` ist von `MenuInteraction` durch Vererbung und von `ExitInteraction`, `LoggedInUserFactory`, `LoginInteraction` und `RegisterUserInteraction` durch Nutzung abhängig. Die Klasse `ThirstyCalc`, aus einer oberen Schicht in der Clean Architektur, ist ebenfalls eine Abhängigkeit von `MainInteraction`. Da also eine Klasse aus einer unteren Ebene (`MainInteraction`) von einer aus einer oberen Ebene (`ThirstyCalc`) abhängig ist, ist hier die Dependency Rule gebrochen.

---

## SOLID (Paul)

### Analyse SRP (3P)

- [jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]

**Positiv-Beispiel** Die Klasse `InputReader` hat die einzige Aufgabe, eine Eingabezeile aus einem `InputStream` zu lesen.

**Negativ-Beispiel** note: Die Klasse `AccountDatabase` hat hier mehrere Aufgaben. Zum einen ist sie für die Erstellung, Persistierung und Löschung von `Accounts` zuständig, zum anderen hat sie auch die Funktionalität mit `checkIfAccountBalanceIsZero(User user)`, ob die `Balance` eines `Account` 0 ist.

Eine mögliche Lösung, um das SRP für `AccountDatabase` umzusetzen, ist im folgenden UML-Diagram dargestellt:

note: Hier wurde die Geschäftslogik, die den `User` betrifft in die Klasse `UserAccountService` ausgelagert. Die Klasse `AccountDatabase` ist somit lediglich für Hinzufügen, Entfernen und Persistieren der `Accounts` verantwortlich.

### Analyse OCP (3P)

- [jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML und Analyse mit Begründung, warum das OCP erfüllt/nicht erfüllt wurde – falls erfüllt: warum hier sinnvoll/welches Problem gab es? Falls nicht erfüllt: wie könnte man es lösen (inkl. UML)?]

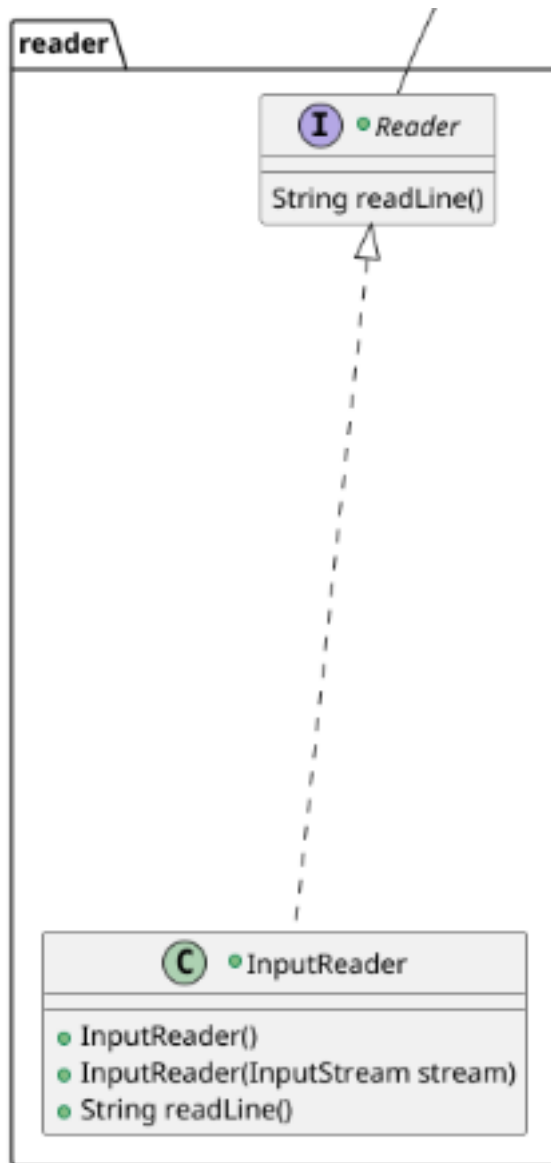


Figure 6: srp\_positive.png

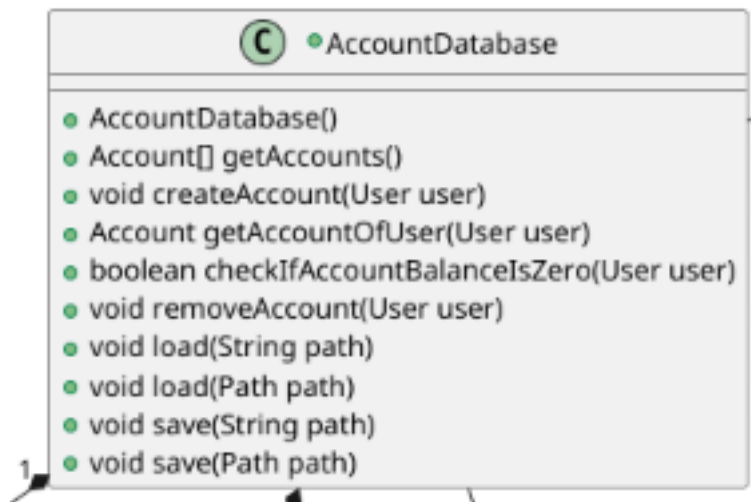


Figure 7: srp\_negative.png

**Positiv-Beispiel** note: Das Interface `Logger` erfüllt hier das OCP. Es können mehrere Implementierungen des `Logger` Interface implementiert werden, ohne, dass deren Aufrufer angepasst werden müssen. Warum wurde das hier umgesetzt? Die Logs sollen unterschiedlich sein, je nachdem, ob ein Benutzer eingeloggt ist oder nicht. Die Klasse `ThirstyCalc`, welche von `Logger` und `LoggerFactory` abhängig ist, ruft bei jedem Log nur die `log(String message)` Methode vom `Logger` Interface auf. Wenn sich ein Benutzer einloggt, wird der `Logger` in `ThirstyCalc` durch eine Instanz des `UserLogger` ersetzt und somit sind die Logs dann auf den `User` bezogen.

**Negativ-Beispiel** note: Die Klasse `AdminRights` erfüllt hier nicht OCP. Wenn man eine neue Benutzerkategorie einführen möchte, müsste man auch das die Aufrufer von `AdminRights` anpassen, um an den benötigten Stellen dann die anderen Rechte zu verteilen.

**Lösung für das Negativ-Beispiel** note: Eine Lösung ist die Abstraktion zu einem `RightsGiver` Interface. Dadurch können neue Rechtegruppen erstellt werden, ohne, dass die Aufrufer von `RightsGiver` angepasst werden müssen.

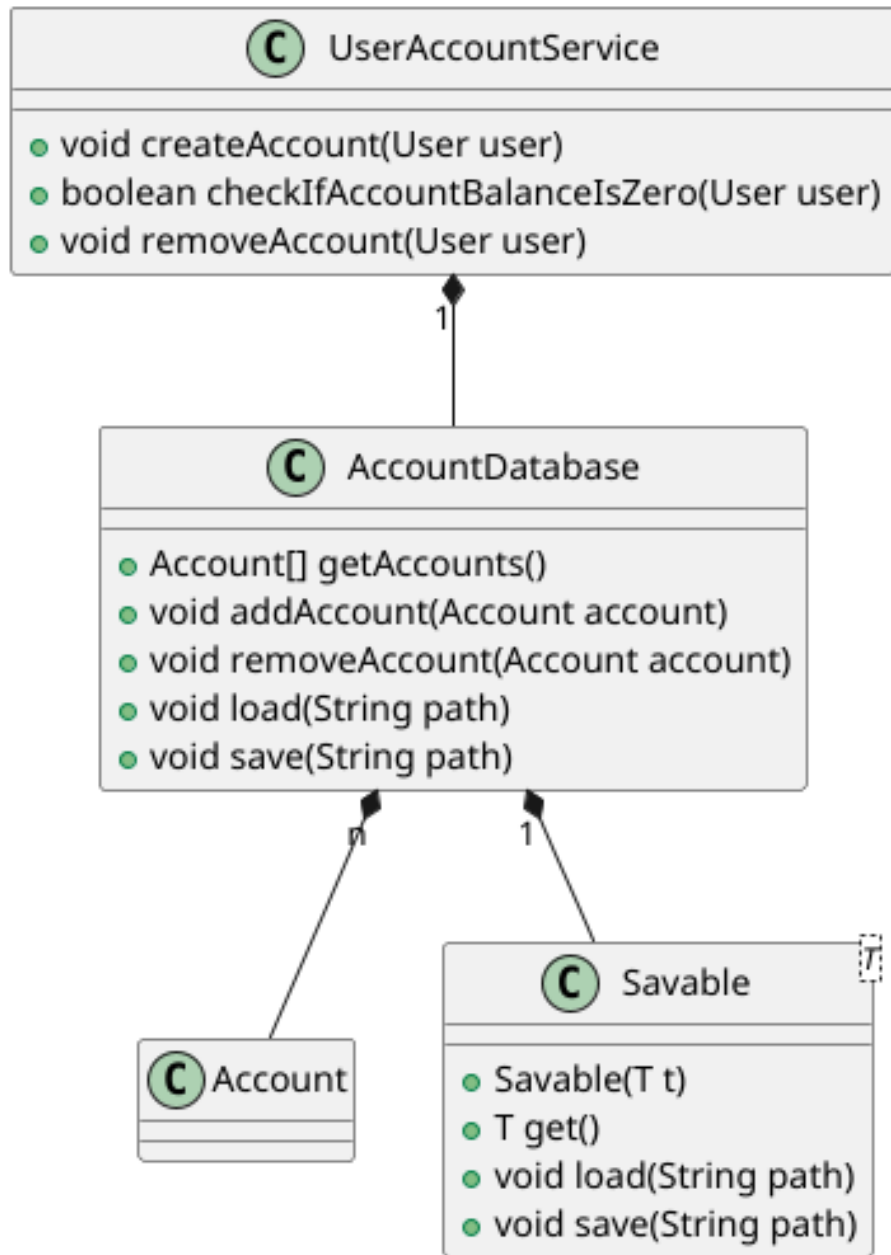


Figure 8: srp\_negative\_soliution.png



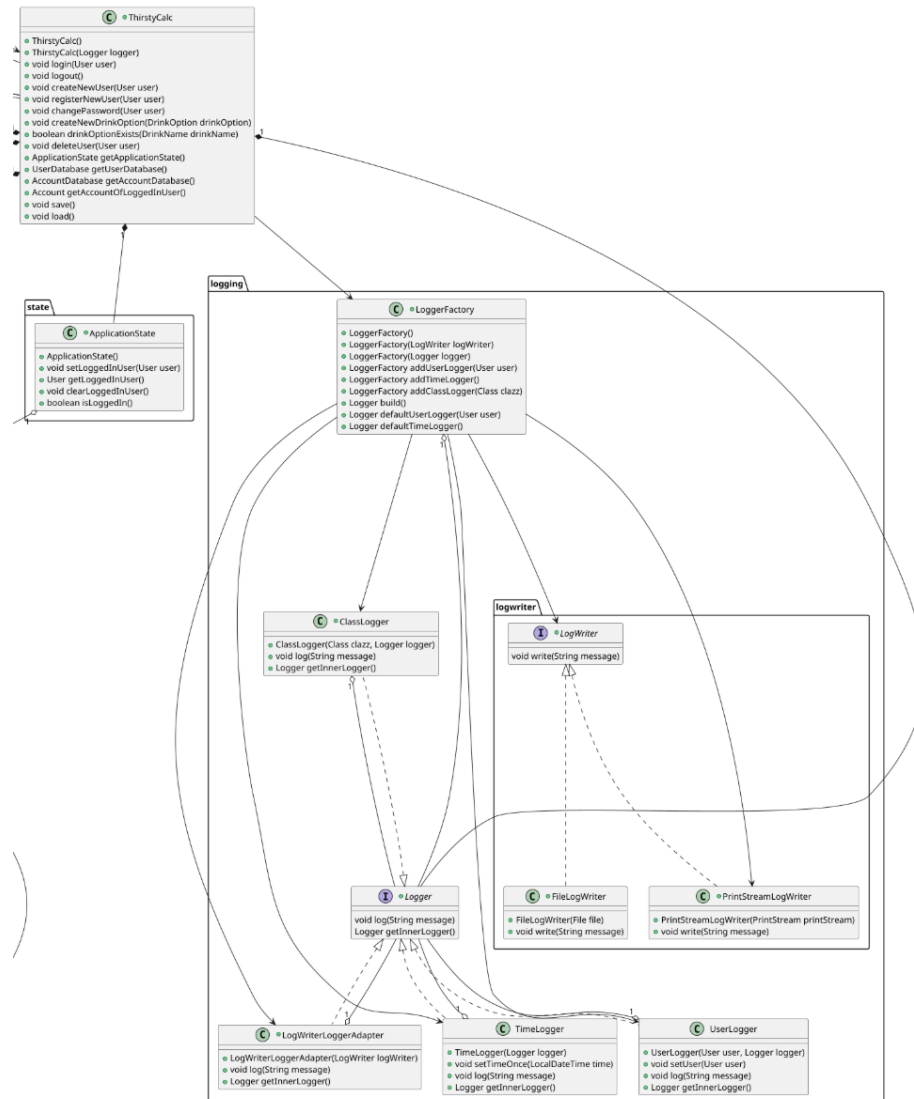


Figure 9: ocp\_positive.png

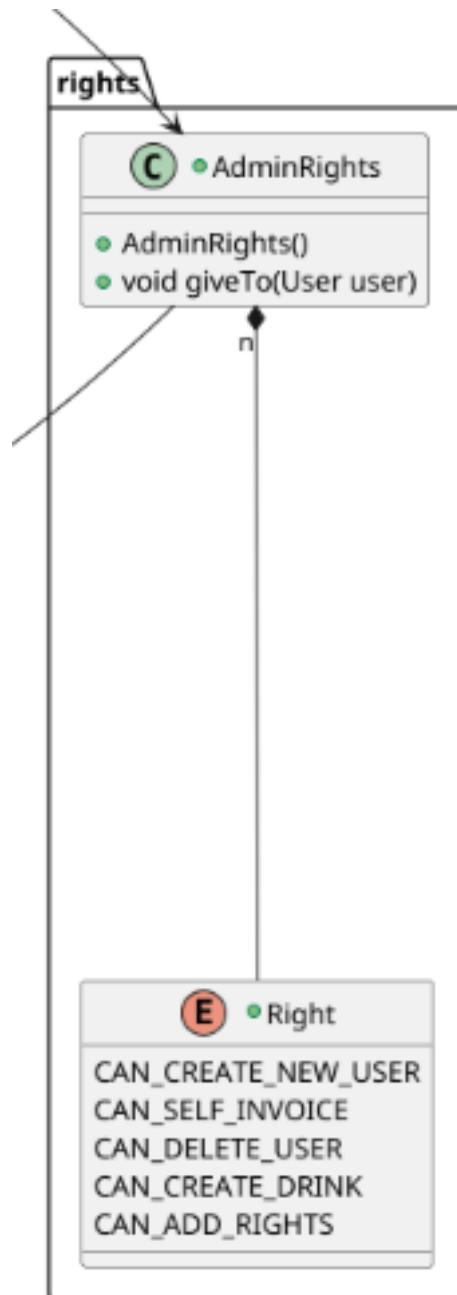


Figure 10: ocp\_negative.png

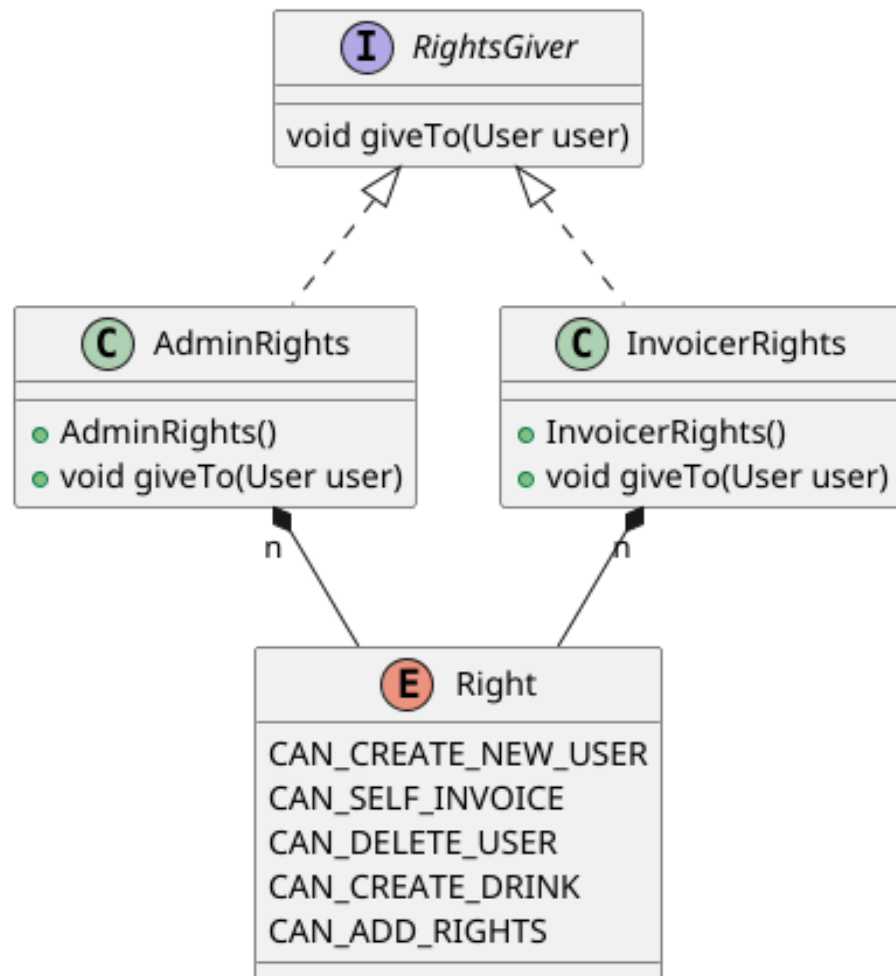


Figure 11: ocp\_negative\_solution.png

## Analyse [LSP/ISP/DIP] (2P)

- [jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP; jeweils UML und Begründung, warum hier das Prinzip erfüllt/nicht erfüllt wird; beim Negativ-Beispiel UML einer möglichen Lösung hinzufügen]
- [Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]

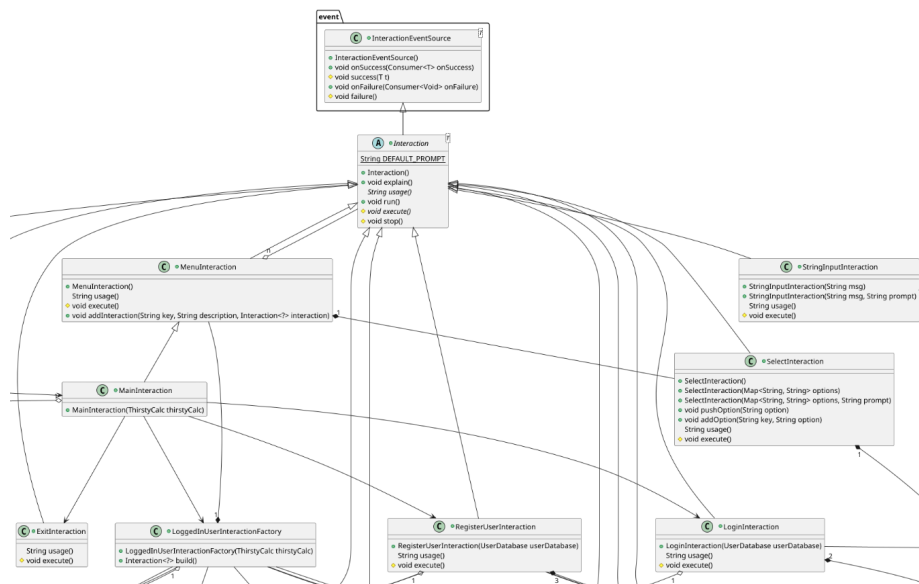


Figure 12: lsp\_positive.png

**Positiv-Beispiel: LSP** note: Die Klasse `Interaction` realisiert das LSP (Liskov Substitution Principle). Jede Implementierung der `Interaction` (z.B. `StringInputInteraction` oder `MenuInteraction`) kann an jeder beliebigen Stelle von `Interaction` benutzt werden, ohne, dass unerwünschte Nebeneffekte auftreten.

**Negativ-Beispiel: LSP** note: Die `LogWriterLoggerAdapter` Klasse kann hier einen unerwünschten Nebeneffekt haben: Beim Aufruf auf `getInnerLogger()` gibt sie sich selbst zurück. Wenn eine Klasse also rekursiv die inneren Logger von den Loggern abrufen, kommt es zu einem Stackoverflow.

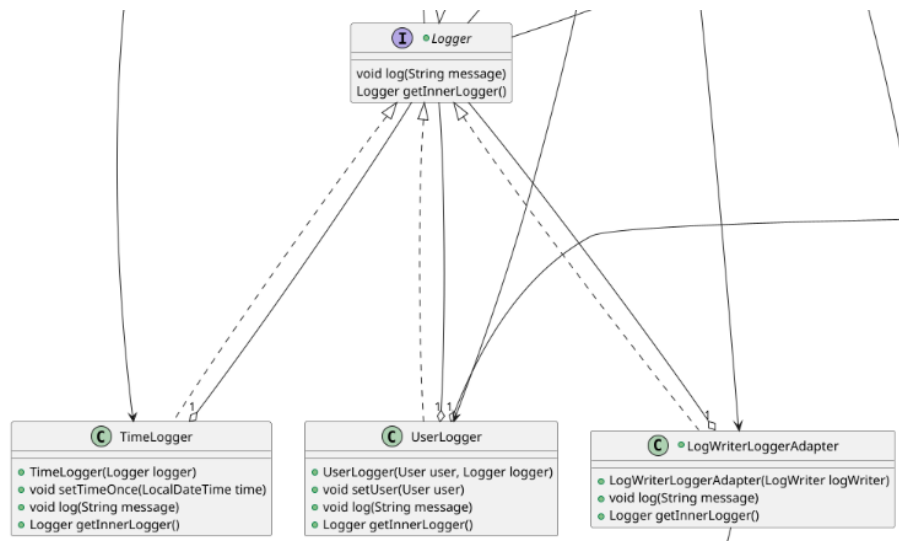


Figure 13: lsp\_negative.png

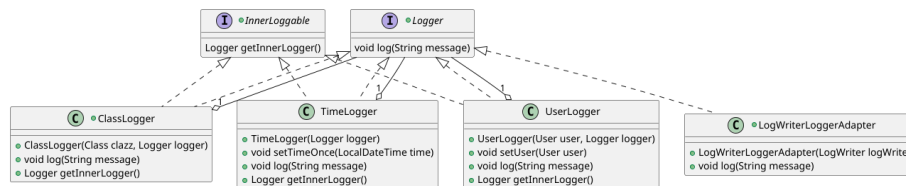


Figure 14: lsp\_negative\_solution.png

**Lösung für das Negativ-Beispiel: LSP** note: Eine Lösung ist das Interface `Logger` in zwei Interfaces zu teilen. Jetzt muss die Klasse `LogWriterLoggerAdapter` nicht mehr die Methode `getInnerLogger()` implementieren und somit ist der unerwünschte Nebeneffekt behoben.

## Weitere Prinzipien (Paul)

### Analyse GRASP: Geringe Kopplung (3P)

- [eine bis jetzt noch nicht behandelte Klasse als positives Beispiel geringer Kopplung; UML mit zusammenspielenden Klassen, Aufgabenbeschreibung]

der Klasse und Begründung, warum hier eine geringe Kopplung vorliegt; es müssen auch die Aufrufer/Nutzer der Klasse berücksichtigt werden]

—

note: Die Klasse `InteractionEventSource` realisiert die geringe Kopplung von GRASP. Durch das hier verwendete Listener Pattern (durch die Methoden `onSuccess(Consumer<T> onSuccess)` und `onFailure(Consumer<Void> onFailure)`)

—

### Analyse GRASP: [Polymorphismus/Pure Fabrication] (3P)

- [eine Klasse als positives Beispiel entweder von Polymorphismus oder von Pure Fabrication; UML Diagramm und Begründung, warum es hier zum Einsatz kommt]

—

**Polymorphismus** note: Die Klasse `Savable` setzt Polymorphismus durch Generics um. Die Klasse kapselt die Funktionalität der De-/Serialisierung und des Ladens und Speicherns von Daten. Da für diese Funktionalität nicht relevant ist, welche Art von Objekt behandelt wird, ist die Klasse generisch gehalten. Somit können mehrere Objekttypen durch die gleiche Klasse verarbeitet werden (hier `List<Account>`, `List<User>`, `List<DrinkOption>`).

—

### DRY (2P)

- [ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher) einfügen; begründen und Auswirkung beschreiben – ggf. UML zum Verständnis ergänzen]

—

git commit: 7050c4c57c00a0a52a48088c0a997e9fa2e227af

—

**Vorher** note: Die unterschiedlichen Spezifizierungen der `Input` Klasse (hier `NumberInput` und `StringInput`) hatten das selbe Verhalten bei der Ausführung eines Prompts. Sie unterschieden sich lediglich beim Erstellen des `Results`.

—

**Nachher** note: Das übereinstimmende Verhalten wurde in die Elternklasse `Input` ausgelagert und die Stelle, die sich bei den Kindklassen unterschieden hat durch eine neue abstrakte Methode `getResult(String input)` ersetzt. Diese

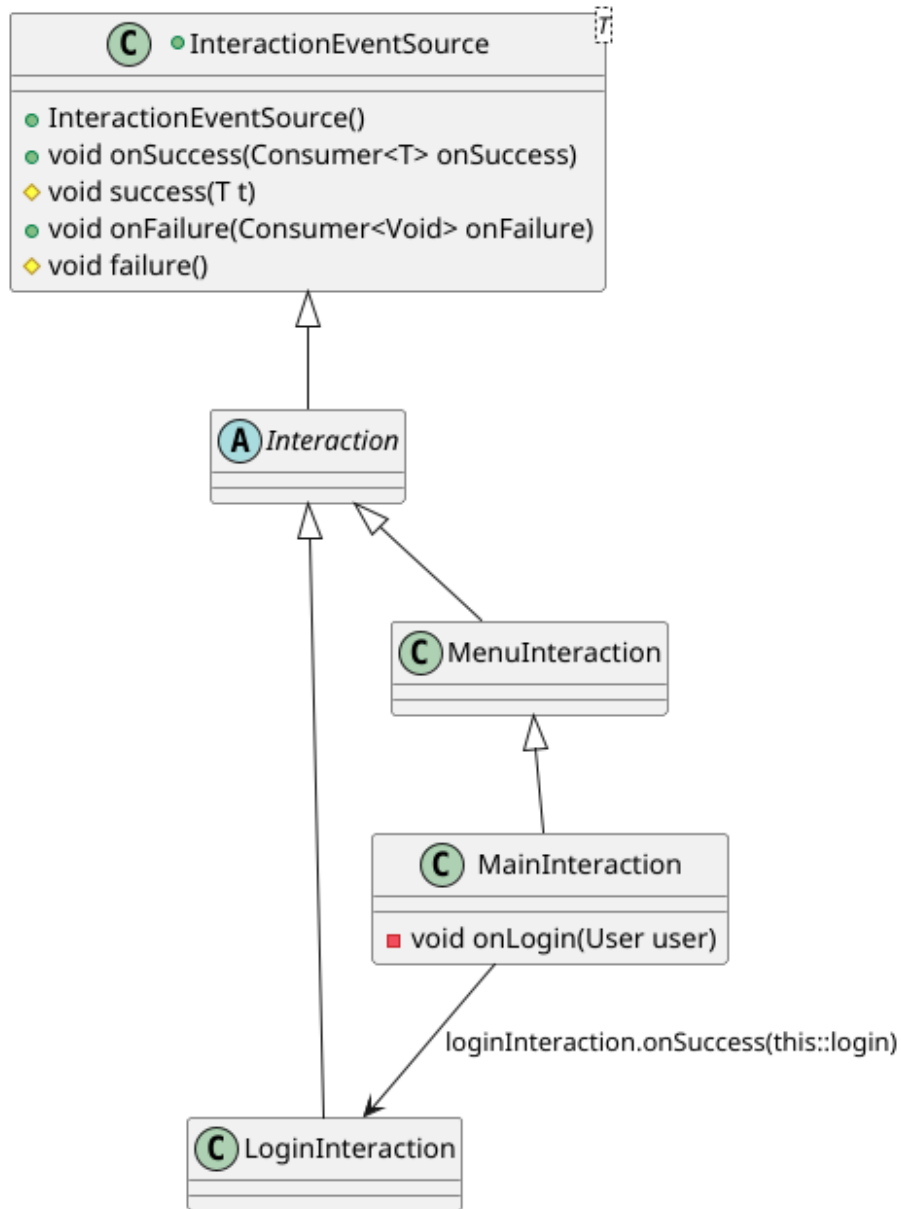


Figure 15: low\_coupling.png

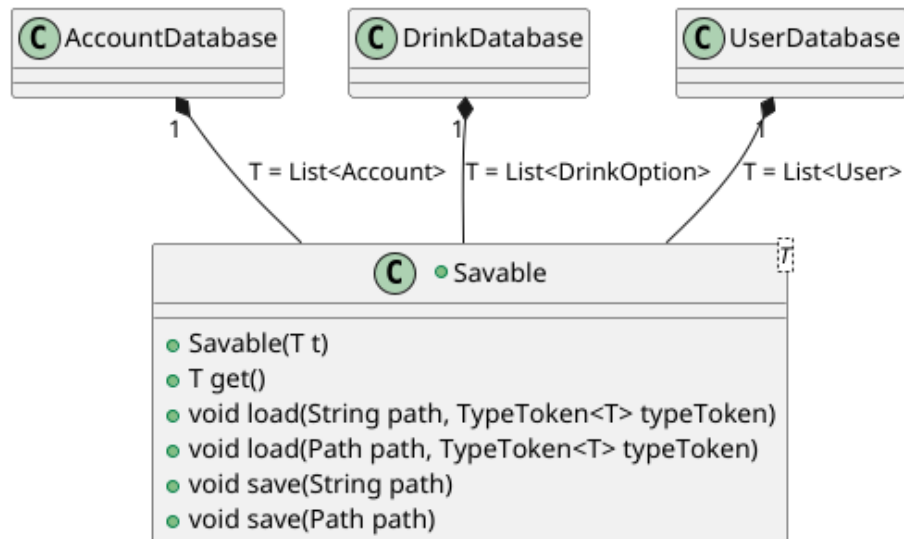


Figure 16: polymorphism.png

implementieren die Kindklassen nun. Somit haben sie weiterhin das selbe Verhalten, der Code wurde aber dedupliziert.

#### Code-Diff

### Unit Tests (S)

- 10 Unit Tests (2 P.)
- ATRIP: Automatic, Thorough und Professional (2 P.)
- Fakes und Mocks (4 P.)

### 10 Unit Tests

- [Zeigen und Beschreiben von 10 Unit-Tests und Beschreibung, was getestet wird] (2 P.)



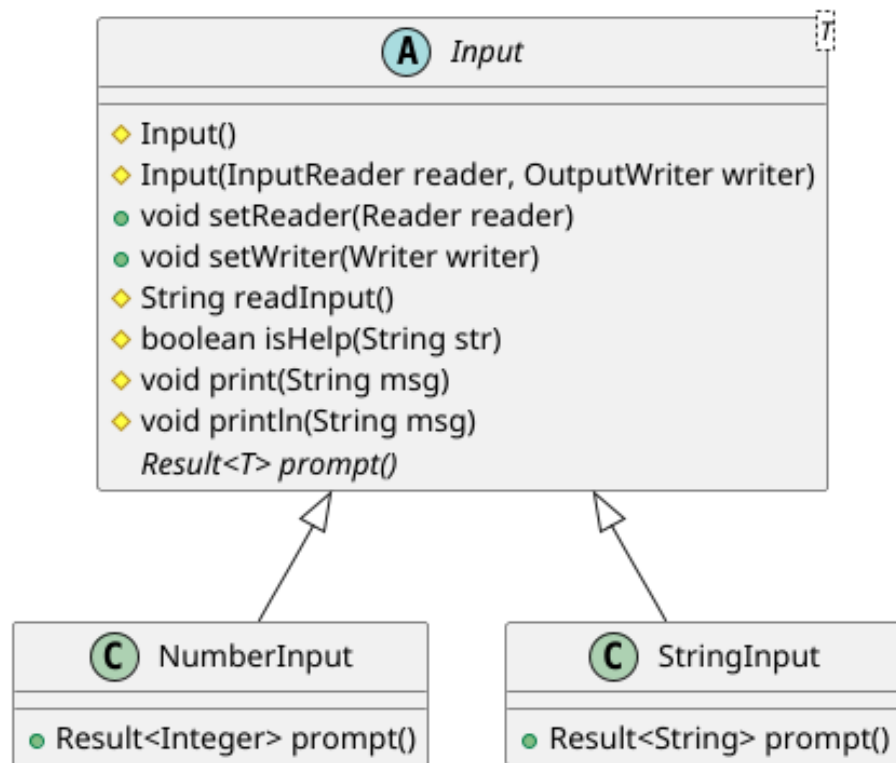


Figure 17: dry\_before.png

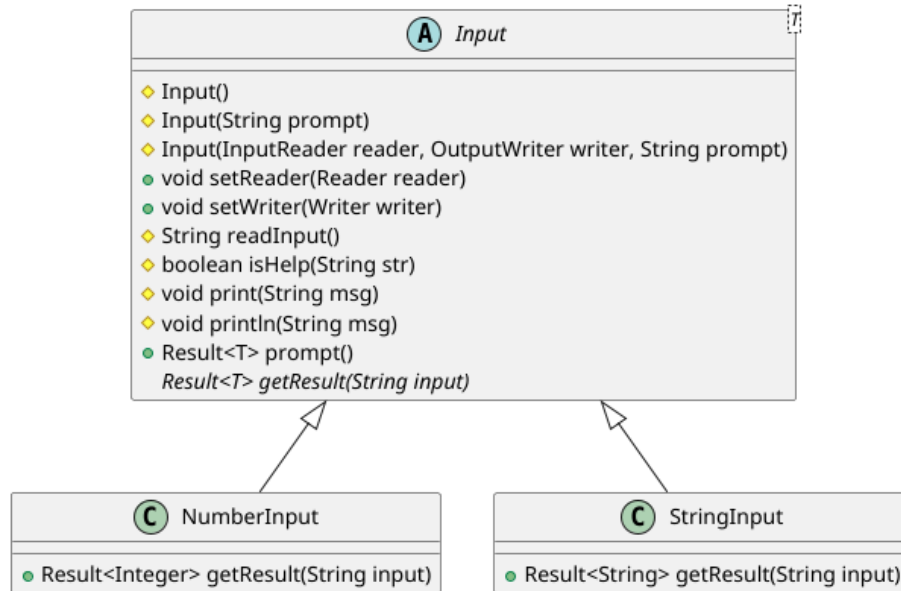


Figure 18: dry\_after.png

```

src/main/java/de/dhbw/karlsruhe/getraenkeabrechnung/io2/input/Input.java
+25 -4

8  abstract class Input<T> {
9
10     private Reader reader;
11     private Writer writer;
12
13     protected Input() {
14         this.reader = new InputReader();
15         this.writer = new OutputWriter();
16     }
17
18     protected Input(InputReader reader, OutputWriter writer) {
19
20         this.reader = reader;
21         this.writer = writer;
22     }
23
24     public void setReader(Reader reader) {
25         @@ -44,5 +49,21 @@
26         writer.writeLine(str);
27     }
28
29     abstract Result<T> prompt();
30
31 }
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48 }
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69 }
50  abstract class Input<T> {
51  + private final String prompt;
52  + private Reader reader;
53  + private Writer writer;
54
55  protected Input() {
56  + this("");
57  }
58
59  protected Input(String prompt) {
60  + this(new InputReader(), new OutputWriter(), prompt);
61  + }
62  +
63  + protected Input(InputReader reader, OutputWriter writer, String prompt)
64  + {
65  +     this.reader = reader;
66  +     this.writer = writer;
67  +     this.prompt = prompt;
68  + }
69
70  public void setReader(Reader reader) {
71
72     writer.writeLine(str);
73 }
74
75 public Result<T> prompt() {
76     print(prompt);
77
78     String in = readInput();
79
80     if (isHelp(in)) {
81         return Result.help();
82     }
83
84     if (in.isBlank()) {
85         return Result.none();
86     }
87
88     return getResult(in);
89 }
90
91 abstract Result<T> getResult(String input);
92 }
  
```

Figure 19: dry\_code\_input.png

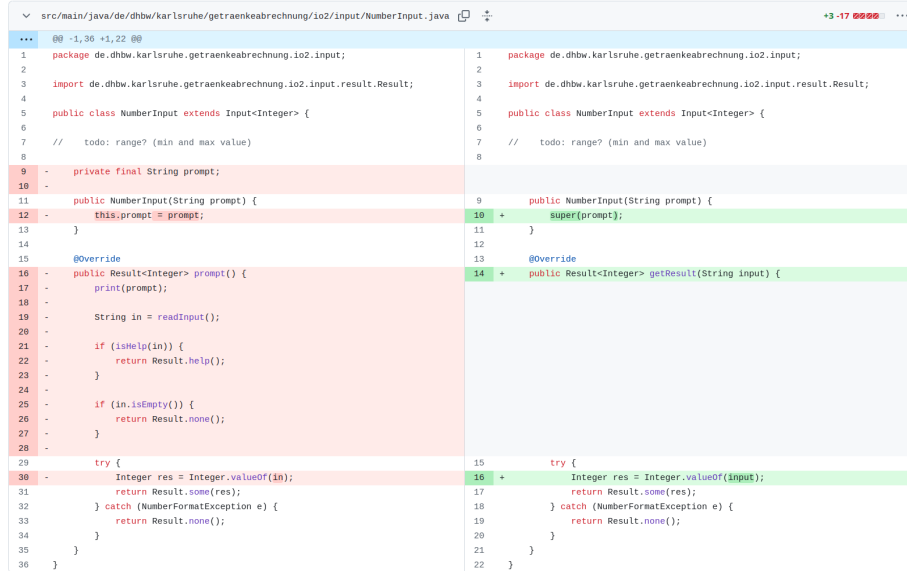


Figure 20: dry\_code\_numberinput.png

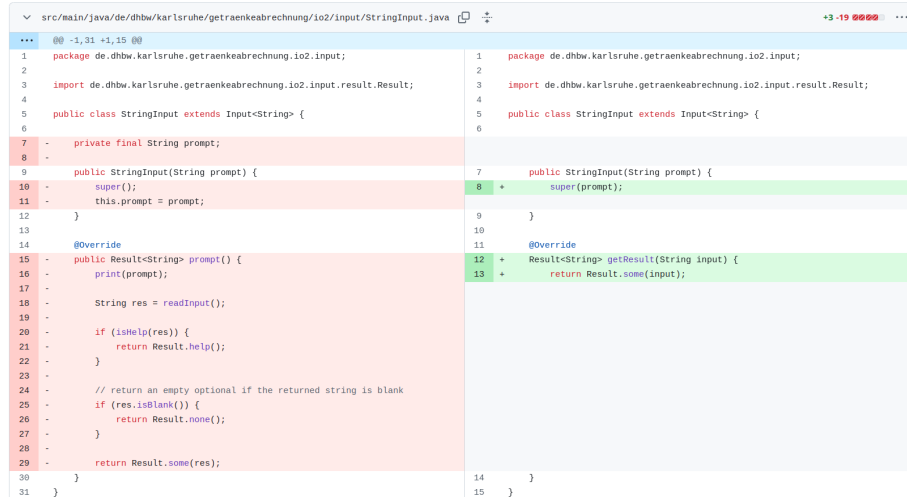


Figure 21: dry\_code\_stringinput.png

```

src > test > java > validators > J UsernameValidatorTest.java > {} validators
1  package validators;
2
3  import static org.junit.jupiter.api.Assertions.*;
4  import org.junit.jupiter.api.Test;
5
6  import de.dhbw.karlsruhe.getraenkeabrechnung.Username;
7  import de.dhbw.karlsruhe.getraenkeabrechnung.validators.UsernameValidator;
8
9  class UsernameValidatorTest {
10
11
12     // Tests for Rule 1: Username consists of alphanumeric characters (a-zA-Z0-9), lowercase, or uppercase.
13     @Test
14     void isValidUsernameBasicShouldReturnTrue() {
15         |   assertTrue(UsernameValidator.isValidUsername(new Username(username:"valid_User123")));
16     }
17
18
19     // Test for Rule 2: Username allowed of the dot (.), underscore (_), and hyphen (-).
20     @Test
21     void isValidUsernameSpecialCharsShouldReturnTrue() {
22         |   assertTrue(UsernameValidator.isValidUsername(new Username(username:"a-valid_u.ser123")));
23     }
24
25
26     // Tests for Rule 3: The dot (.), underscore (_), or hyphen (-) must not be the first or last character.
27     @Test
28     void isValidUsernameWithUnderscoreStartShouldReturnFalse() {
29         |   assertFalse(UsernameValidator.isValidUsername(new Username(username:"_invalidUser123")));
30     }
31
32
33     @Test
34     void isValidUsernameWithDotStartShouldReturnFalse() {
35         |   assertFalse(UsernameValidator.isValidUsername(new Username(username:".invalidUser123")));
36     }
37
38     @Test
39     void isValidUsernameWithHyphenStartShouldReturnFalse() {
40         |   assertFalse(UsernameValidator.isValidUsername(new Username(username:"-invalidUser123")));
41     }
42
43     @Test
44     void isValidUsernameWithUnderscoreEndShouldReturnFalse() {
45         |   assertFalse(UsernameValidator.isValidUsername(new Username(username:"invalidUser123_")));
46     }
47
48     @Test
49     void isValidUsernameWithDotEndShouldReturnFalse() {
50         |   assertFalse(UsernameValidator.isValidUsername(new Username(username:"invalidUser123.")));
51     }
52

```

Figure 22: UnitTestUsernameValidator

## Unit Tests 1/2 Username

## Unit Tests 2/2 Password

```
src > test > java > validators > J PasswordValidatorTest.java > ...
1  package validators;
2
3  import org.junit.jupiter.api.Test;
4
5  import de.dhbw.karlsruhe.getraenkeabrechnung.Password;
6  import de.dhbw.karlsruhe.getraenkeabrechnung.PasswordManagementException;
7  import de.dhbw.karlsruhe.getraenkeabrechnung.validators.PasswordValidator;
8
9  import static org.junit.jupiter.api.Assertions.*;
10
11 class PasswordValidatorTest {
12
13     @Test
14     void isValidPasswordShouldReturnTrue() {
15         assertTrue(PasswordValidator.isValidPassword(new Password(password:"goodPassword=1")));
16     }
17
18     @Test
19     void isValidPasswordWithNoLowerShouldReturnFalse() {
20         assertFalse(PasswordValidator.isValidPassword(new Password(password:"noNumber#")));
21     }
22
23     @Test
24     void isValidPasswordTooShortShouldReturnFalse() {
25         assertFalse(PasswordValidator.isValidPassword(new Password(password:"Short3@")));
26     }
27
28     @Test
29     void isValidPasswordNoSpecialCharShouldReturnFalse() {
30         assertFalse(PasswordValidator.isValidPassword(new Password(password:"no!SpeChar1")));
31     }
32
33     @Test
34     void isValidPasswordNoLowerCharShouldReturnFalse() {
35         assertFalse(PasswordValidator.isValidPassword(new Password(password:"NOLOWER#1")));
36     }
37
38     @Test
39     void isValidPasswordNoUpperCharShouldReturnFalse() {
40         assertFalse(PasswordValidator.isValidPassword(new Password(password:"noupper#2")));
41     }
42
43     @Test
44     void isPasswordHashedCorrectlyShouldReturnTrue() throws PasswordManagementException {
45         Password password = new Password(password:"goodPassword=1");
46         password.hashPassword();
47         assertTrue(password.verifyPassword(password.getPlainPassword(), password.getHashedPassword(), password.getSalt()));
48     }
49
50 }
```

Figure 23: UnitTestPasswordValidator

## Automatic

- [Begründung/Erläuterung, wie ‘Automatic’ realisiert wurde]
  - alles wird in IDE auf einmal ausgeführt zum individuellen testing vor push / pr in main
  - pipeline führt bei jedem merge in main branch die tests aus

## Thorough

- [Code Coverage im Projekt analysieren und begründen]

- SonarQubeCloud bietet Übersicht über Codecoverage spezifisch für lines und branches

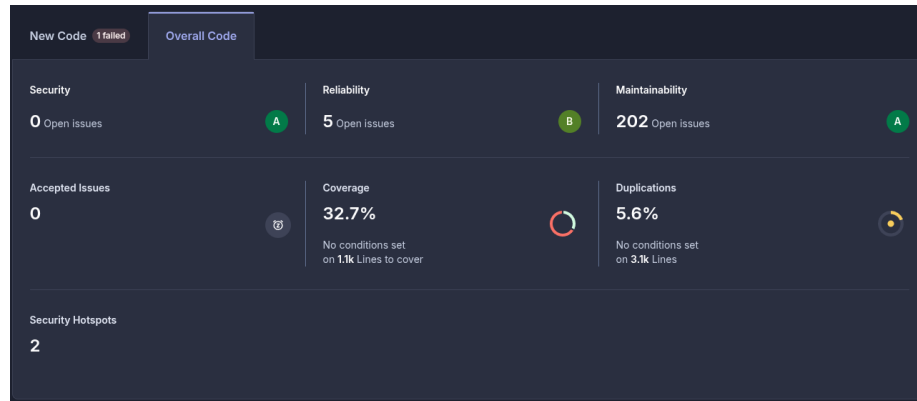


Figure 24: SonarCodeCoverage

## Code Coverage –

### Professional

- [1 positives Beispiel zu ‘Professional’; Code-Beispiel, Analyse und Begründung, was professionell ist]
- Arrange-Act-Assert Pattern: (BooleanInputTest.java) gleicher Aufbau von Tests erhöht Übersichtlichkeit und Verständnis von Code

### Professional: BooleanInputTest

### Fakes und Mocks

- [Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten (die Fake/Mocks sind ohne Dritthersteller-Bibliothek/Framework zu implementieren)]
- Zeigen der Implementierung und Nutzung; zusätzlich jeweils UML Diagramm mit Beziehungen zwischen Mock, zu mockender Klasse und Aufrufer des Mocks]

```

13  public class BooleanInputTest {
14      private InputReaderMock readerMock;
15      private OutputWriterMock writerMock;
16
17      @BeforeEach
18      public void setup() {
19          readerMock = new InputReaderMock();
20          writerMock = new OutputWriterMock();
21      }
22
23      @Test
24      public void itPrintsPrompt() {
25          String prompt = "Prompt";
26
27          BooleanInput input = new BooleanInput(prompt);
28          input.setReader(readerMock);
29          input.setWriter(writerMock);
30
31          input.prompt();
32
33          assertEquals(prompt, writerMock.getOutput());
34      }

```

Figure 25: Diagrammbeschreibung

## Mock: InputReaderMock

- Analyse: Eingabesystem des CLI schwer generisch testbar, daher Mock vorteilhaft.

—

```
src > test > java > io > mocks >  InputReaderMock.java > ...
1  package io.mocks;
2
3  import de.dhbw.karlsruhe.getraenkeabrechnung.io.reader.Reader;
4
5  public class InputReaderMock implements Reader {
6
7      private String input;
8      private boolean hasBeenRead;
9
10     public InputReaderMock() {
11         this.input = "";
12         this.hasBeenRead = false;
13     }
14
15     public void setNextInput(String input) {
16         this.input = input;
17     }
18
19     @Override
20     public String readLine() {
21         this.hasBeenRead = true;
22         String ret = input;
23
24         this.input = null;
25
26         return ret;
27     }
28
29     public boolean hasBeenRead() {
30         return hasBeenRead;
31     }
32 }
```

Figure 26: InputReaderMock.java

Implementierung —

Nutzung —

Nutzung —



- ✓ BooleanInputTest.java src/test/java/io/input
  - import io.mocks.InputReaderMock;
  - private InputReaderMock readerMock;
  - readerMock = new InputReaderMock();
- ✓ FloatInputTest.java src/test/java/io/input
  - import io.mocks.InputReaderMock;
  - private InputReaderMock readerMock;
  - readerMock = new InputReaderMock();
- ✓ NumberInputTest.java src/test/java/io/input
  - import io.mocks.InputReaderMock;
  - private InputReaderMock readerMock;
  - readerMock = new InputReaderMock();
- ✓ StringInputTest.java src/test/java/io/input
  - import io.mocks.InputReaderMock;
  - private InputReaderMock readerMock;
  - readerMock = new InputReaderMock();
- ✓ InputReaderMock.java src/test/java/io/mocks
  - public class InputReaderMock implements Reader {

Figure 27: InputReaderMock.java references

```

13  public class BooleanInputTest {
14      private InputReaderMock readerMock;
15      private OutputWriterMock writerMock;
16
17      @BeforeEach
18      public void setup() {
19          readerMock = new InputReaderMock();
20          writerMock = new OutputWriterMock();
21      }
22
23      @Test
24      public void itPrintsPrompt() {
25          String prompt = "Prompt";
26
27          BooleanInput input = new BooleanInput(prompt);
28          input.setReader(readerMock);
29          input.setWriter(writerMock);
30
31          input.prompt();
32
33          assertEquals(prompt, writerMock.getOutput());
34      }

```

Figure 28: Diagrammbeschreibung

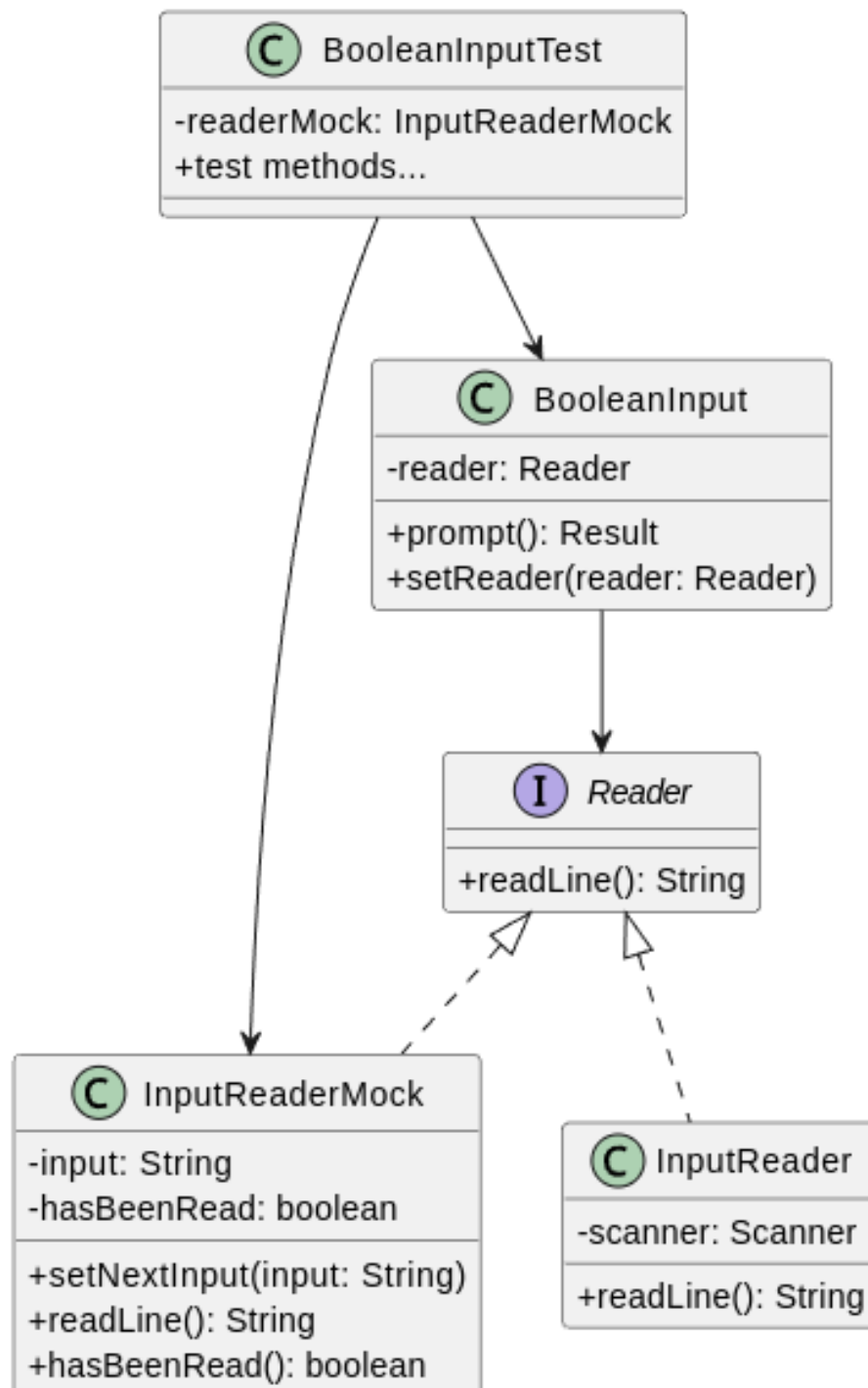


Figure 29: UMLinputReaderMock

## UML für Beziehung –

### Mock: OutputWriterMock

- Analyse: Ausgabesystem des CLI schwer generisch testbar, daher Mock vorteilhaft.

–

```
src > test > java > io > mocks >  OutputWriterMock.java > ...  
1  package io.mocks;  
2  
3  import de.dhbw.karlsruhe.getraenkeabrechnung.io.writer.Writer;  
4  
5  public class OutputWriterMock implements Writer {  
6      private String output;  
7  
8      public OutputWriterMock() {  
9          output = "";  
10     }  
11  
12     @Override  
13     public void writeline(String line) {  
14         output = line;  
15     }  
16  
17     @Override  
18     public void write(String str) {  
19         output = str;  
20     }  
21  
22     public String getOutput() {  
23         return output;  
24     }  
25 }
```

Figure 30: OutputWriterMock.java

## Implementierung –

## Nutzung –

## Nutzung –

## UML für Beziehung

- ✓ BooleanInputTest.java src/test/java/io/input
  - import io.mocks.OutputWriterMock;
  - private OutputWriterMock writerMock;
  - writerMock = new OutputWriterMock();
- ✓ FloatInputTest.java src/test/java/io/input
  - import io.mocks.OutputWriterMock;
  - private OutputWriterMock writerMock;
  - writerMock = new OutputWriterMock();
- ✓ NumberInputTest.java src/test/java/io/input
  - import io.mocks.OutputWriterMock;
  - private OutputWriterMock writerMock;
  - writerMock = new OutputWriterMock();
- ✓ StringInputTest.java src/test/java/io/input
  - import io.mocks.OutputWriterMock;
  - private OutputWriterMock writerMock;
  - writerMock = new OutputWriterMock();
- ✓ OutputWriterMock.java src/test/java/io/mocks
  - public class OutputWriterMock implements Writer {

Figure 31: OutputWriterMock.java references

```

37     @Test
38     public void itReturnsInput() {
39         String in = "input";
40
41         StringInput input = new StringInput(prompt: "");
42
43         input.setReader(readerMock);
44         input.setWriter(writerMock);
45
46         readerMock.setNextInput(in);
47
48         Result<String> res = input.prompt();
49
50         assertTrue(res.hasValue());
51         assertEquals(in, res.getValue());
52     }
--

```

Figure 32: StringInputTest.java

## Domain Driven Design (S)

- Ubiquitous Language (2 P.)
- Repositories (1,5 P.)
- Aggregates (1,5 P.)
- Entities (1,5 P.)
- Value Objects (1.5 P.)

### Ubiquitous Language

- Ubiquitous Language: gemeinsame, strenge Sprache zwischen Entwicklern und Benutzern, basierend auf dem Domänenmodell.
- Ziel ist es, Mehrdeutigkeiten zu minimieren und die Kommunikation und das Verständnis zwischen allen am Softwareentwicklungsprozess Beteiligten zu verbessern.

Bezeichnung	Bedeutung	Begründung
User	Beschreibt interagierenden Anwender	Domänenexperte wird mehrere Menschen haben, die das System bedienen (Privilegien).

Bezeichnung	Bedeutung	Begründung
Balance	Beschreibt Guthaben des Anwenders	Guthaben ist aufgrund Sinn des Programms auch für Domänenexperte relevant.

–

Bezeichnung	Bedeutung	Begründung
Drink	Beschreibt Produkteigenschaften (Name, Kategorie)	Beschreibt aus der Domäne stammende Produkt, für die die Software entwickelt wurde.
Category	Beschreibt Eigenschaften der Kategorie (Name, Preis)	Dient zur Repräsentation des aus der Domäne stammenden Preismodell.

–

### Repositories

- [UML, Beschreibung und Begründung des Einsatzes eines Repositories; falls kein Repository vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde]
- Schnittstelle zwischen der Domäne und der Persistenzschicht (z. B. Datenbank). Es kümmert sich darum, Aggregate (oder manchmal auch Entitäten) zu speichern, zu laden und zu entfernen, ohne dass die Domänenlogik direkt mit der Datenbank oder technischen Details interagieren muss.

–

### DrinkDatabase -> DrinkRepository

–

### Entities

- [UML, Beschreibung und Begründung des Einsatzes einer Entity; falls keine Entity vorhanden: ausführliche Begründung, warum es keine geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde]

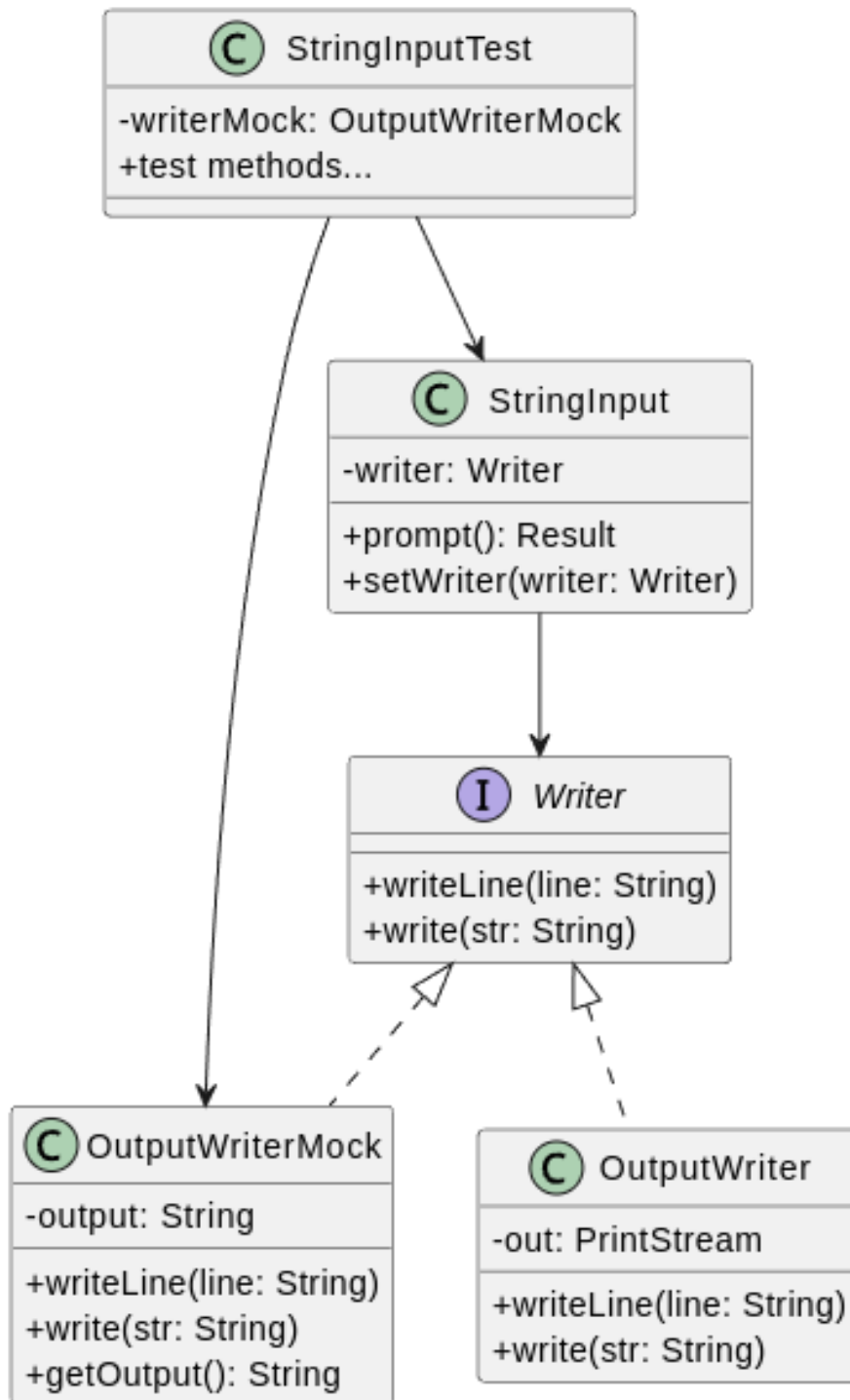


Figure 33: UML `OutputWriterMock`





Figure 34: Diagrammbeschreibung

- Entity: Hat ID & ist veränderbar

### Entity: DrinkName

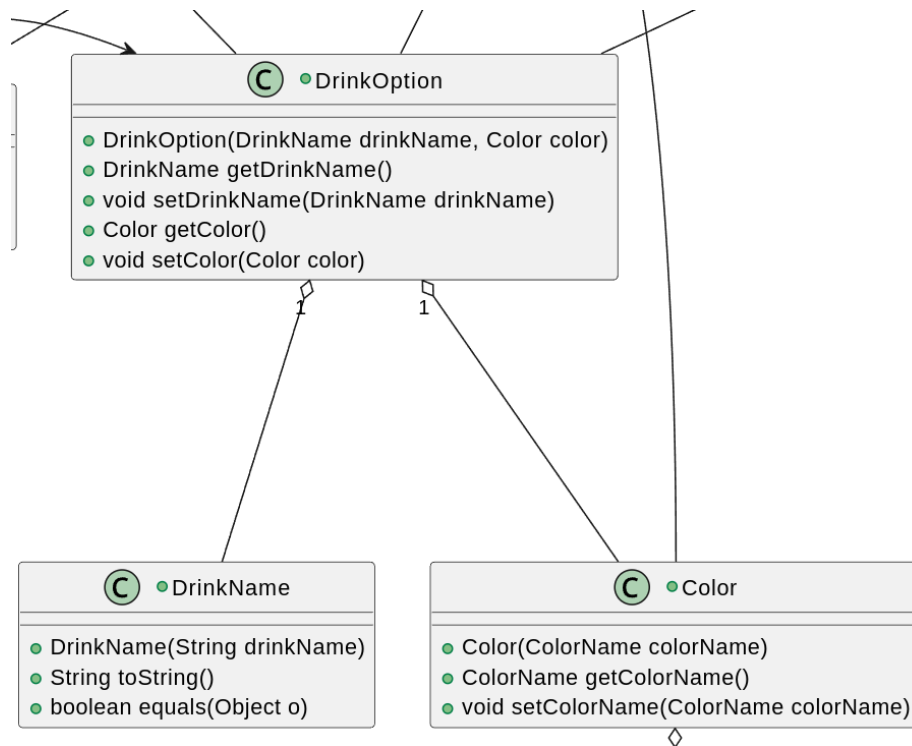


Figure 35: Diagrammbeschreibung

### Value Objects

- [UML, Beschreibung und Begründung des Einsatzes eines Value Objects; falls kein Value Object vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist– NICHT, warum es nicht implementiert wurde]
- Value Object: Hat keine ID & ist nicht veränderbar

### Value Object: CategoryPrice

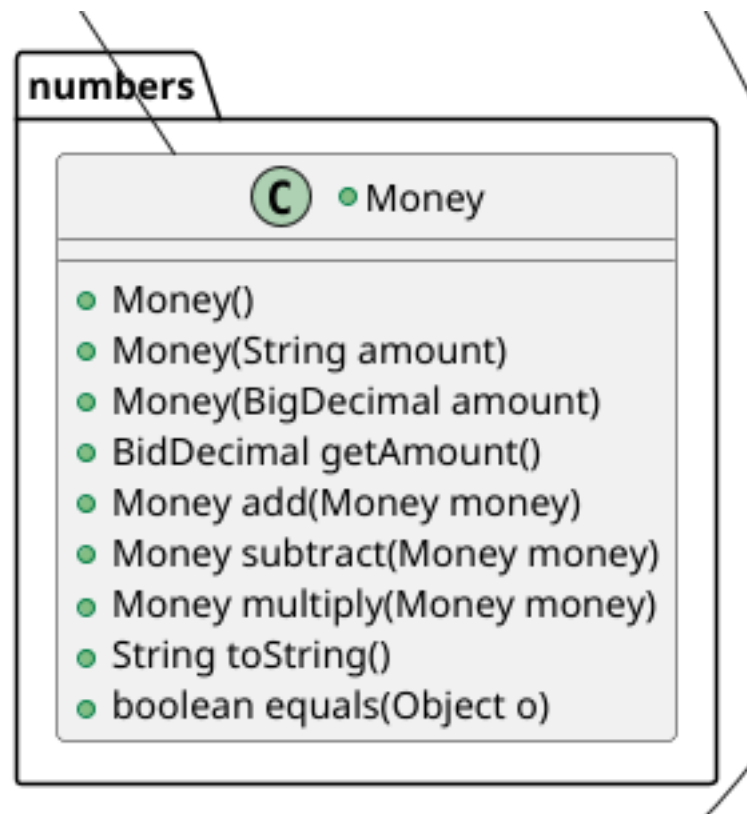


Figure 36: Diagrammbeschreibung

## Aggregates

- [UML, Beschreibung und Begründung des Einsatzes eines Aggregates; falls kein Aggregate vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist– NICHT, warum es nicht implementiert wurde]
- Aggregate: Zusammengesetzt aus Entities sowie Value Objects.

### Aggregate: DrinkOption

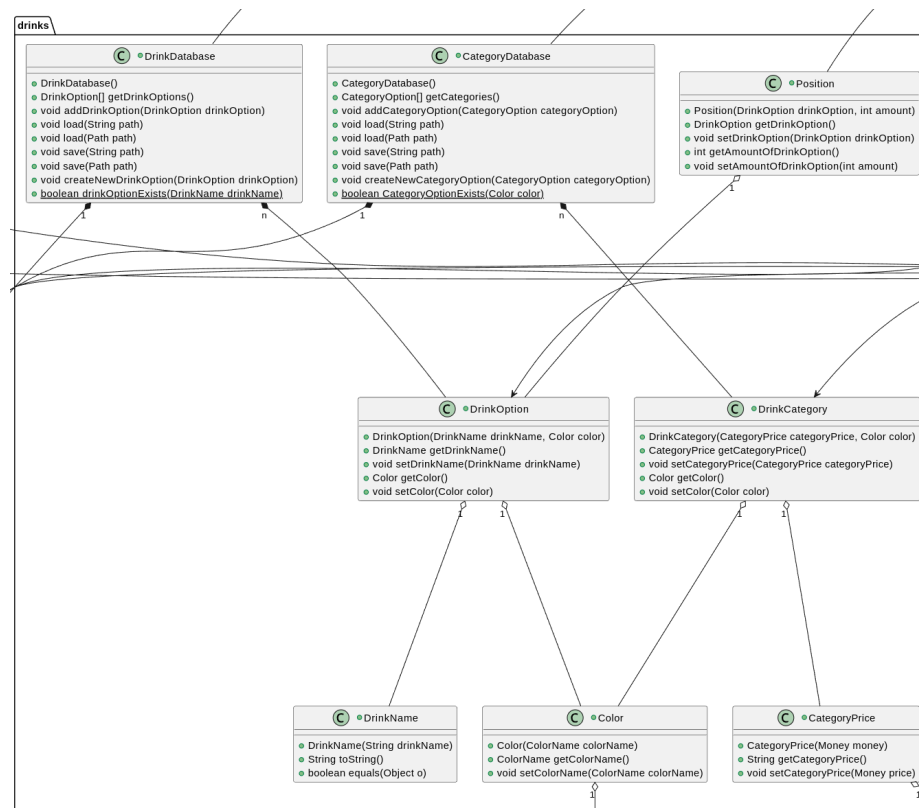


Figure 37: Diagrammbeschreibung

## Refactoring

## Code Smells

- jeweils 1 Code-Beispiel zu 2 unterschiedlichen Code Smells (die benannt werden müssen) aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)

### Code Smell #1: Code Duplication

**getValidInput Methoden** In jeder Interaction Klasse wurde separat eine getValidInput Methode geschrieben.

```
public class CreateDrinkOptionInteraction extends Interaction<DrinkOption> {
    //[...]

    private String getValidInput(StringInput input) {
        while (true) {
            Result<String> result = input.prompt();

            if (result.isHelp()) {
                explain();
                continue;
            }

            if (result.isNone()) {
                System.out.println("Invalid input!");
                continue;
            }

            return result.getValue();
        }
    }
}
```

Figure 38: alt text

### Lösung: Methode in die Basisklasse verschieben

Alle betroffenen Klassen erben von `Interaction<T>`, somit kann man sie in die Basisklasse verschieben, von wo sie dann aufgerufen werden kann.

```
public class AddRightsInteraction extends Interaction<User> {  
    // [...]   
  
    private String getValidInput(StringInput input) {  
        while (true) {  
            Result<String> result = input.prompt();  
  
            if (result.isHelp()) {  
                explain();  
                continue;  
            }  
  
            if (result.isNone()) {  
                System.out.println("Invalid input!");  
                continue;  
            }  
  
            return result.getValue();  
        }  
    }  
}
```

Figure 39: alt

```

public abstract class Interaction<T> {
    //[...]

    protected String getValidInput(StringInput input) {
        while (true) {
            Result<String> result = input.prompt();

            if (result.isHelp()) {
                explain();
                continue;
            }

            if (result.isNone()) {
                System.out.println("Invalid input!");
                continue;
            }

            return result.getValue();
        }
    }
    //[...]
}

```

Figure 40: alt

## Code Smell #2: Method Chains

**Kontostand ist leer** Wenn geprüft werden soll, ob das Konto eines bestimmten Benutzers leer ist, kann sie durch die folgende Method Chain geprüft werden:

### Lösung: Funktion/Methode extrahieren/verschieben

Die einzelnen aufgerufenen Methoden können aufgeteilt und in separaten Funktionen aufgerufen werden. Zusätzlich werden einem Konto separat ein Benutzername zugeordnet, um die Abhängigkeit von der `UserDatabase` zu lösen.

```
if (accountDatabase.getAccountOfUser(userDatabase.getUser(username).getUsername()).isEmpty()) {  
    // [...]  
}
```

Figure 41: alt

```
// In der AccountDatabase Klasse  
  
public boolean checkIfAccountBalanceIsZero(User user) {  
    for (Account a : accounts.get()) {  
        if (a.getUsername().equals(user.getUsername()) && a.isEmpty()) {  
            return true;  
        }  
    }  
    return false;  
}
```

Figure 42: alt



---

## Refactors


- 2 unterschiedliche Refactorings aus der Vorlesung jeweils benennen, anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen – die Refactorings dürfen sich nicht mit den Beispielen der Code Smells überschneiden
- 

### Refactor #1: Replace Error Code With Exception

**User existiert nicht** Die Funktion `getUser(Username username)` soll bei Eingabe eines Benutzernamens aus der Benutzerdatenbank einen Benutzer zurückgeben. Wie sollte die Funktion reagieren, wenn der Benutzer nicht gefunden wird?

---

Ursprüngliche Idee:



```
public User getUser(Username username) {
    for (User u : users.get()) {
        if (u.getUsername().equals(username)) {
            return u;
        }
    }
    return null;
}
```

Figure 43: alt

---

Das zurückgeben eines `null` Values kann unvorgesehene Probleme verursachen, wenn das Ergebnis dieser Methode an eine andere weitergegeben wird. Dies hätte zu einem direkten Absturz des Programms geführt.

---

**Lösung: Eine `UserDoesNotExistException`**

Siehe Commit [a348b06325174bb5c331f1a7031786d727bff9bc](#) Wenn ein Benutzer von einer Datenbank entnommen wird, wird kein null Value als return zurückgegeben, stattdessen eine Custom Exception.

—

```
package de.dhbw.karlsruhe.getraenkeabrechnung.data;

public class UserDoesNotExistException extends RuntimeException {
    public UserDoesNotExistException(String message) {
        super(message);
    }
}
```

Figure 44: alt

—

```
//Implementation in der UserDatabase Klasse

public User getUser(Username username) {
    for (User u : users.get()) {
        if (u.getUsername().equals(username)) {
            return u;
        }
    }

    throw new UserDoesNotExistException("User with the username " + username + " does not exist!");
}
```

Figure 45: alt

—

Falls doch kein Benutzer gefunden wird, kann die Exception gefangen werden und das Programm kann weiterlaufen, ohne vorher abzustürzen.

—

UML davor —

UML danach —

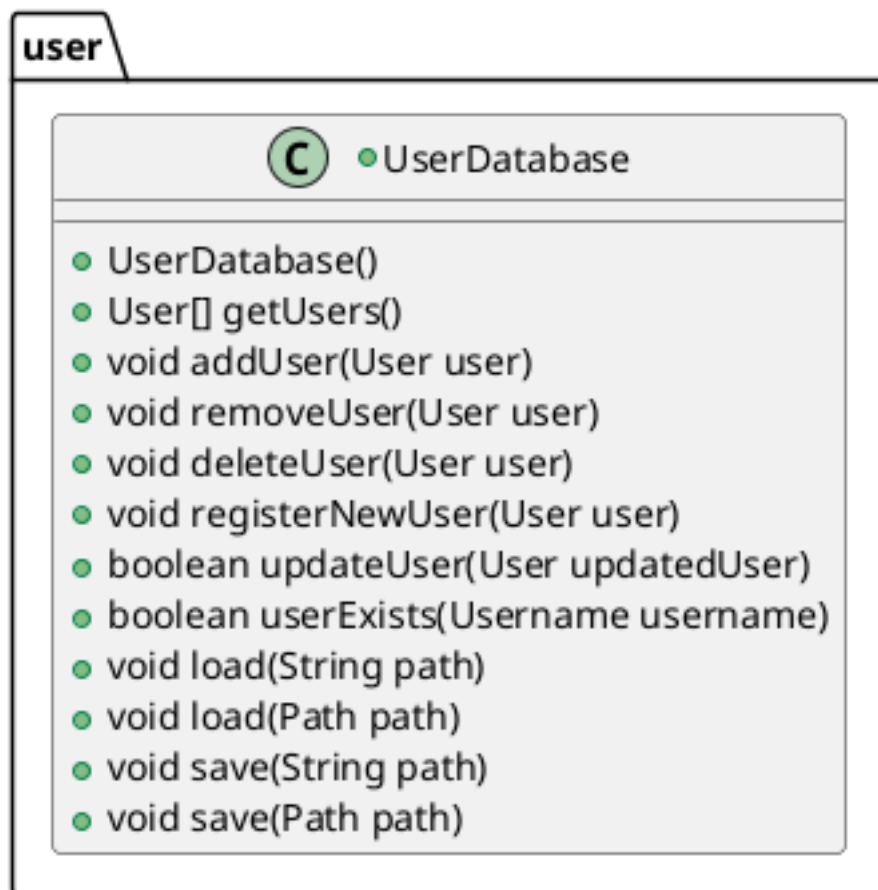


Figure 46: UML davor

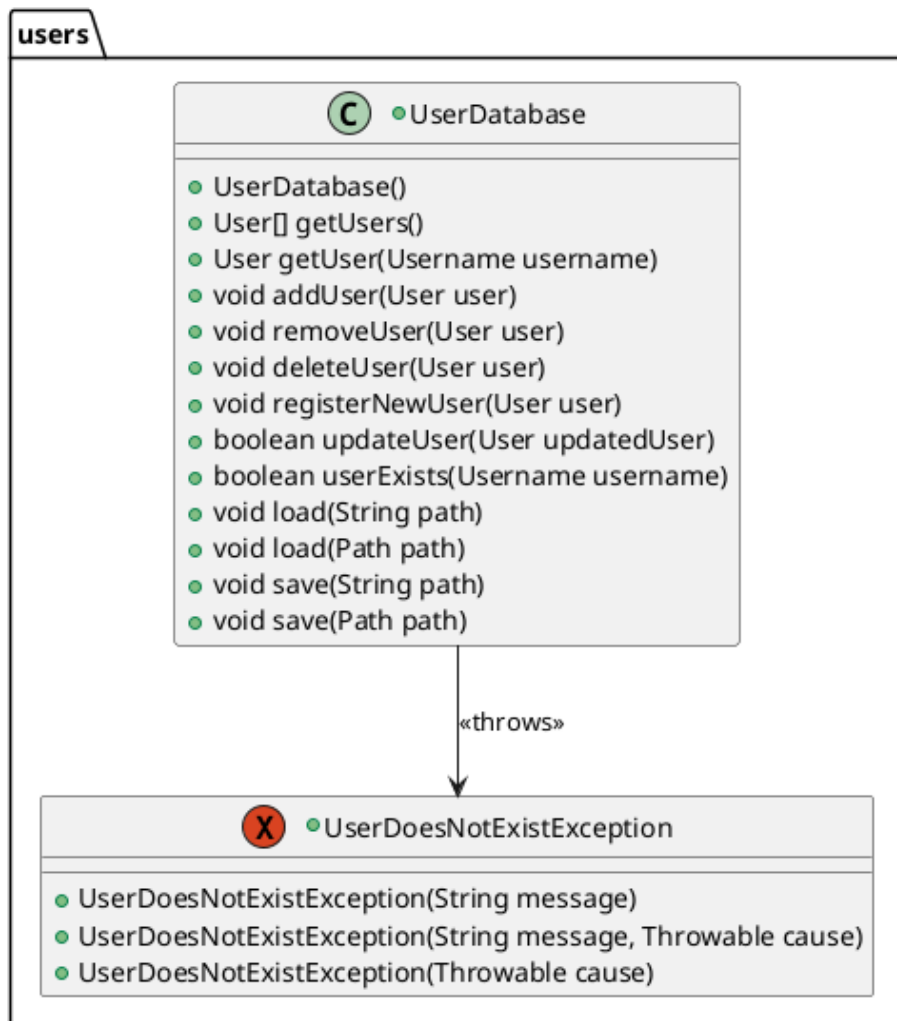


Figure 47: UML danach

## Refactor #2: Extract Method/Class

**Die Validator Klassen** Alle Validator für Benutzernamen, Passwörter etc. dienten zur Validierung von Strings. Sie waren alle eigenständige Klassen. Die Klassen, die diese Methoden nutzen wollen, mussten sie einzeln referenzieren. Dies führte zu unübersichtlichen Abhängigkeiten.

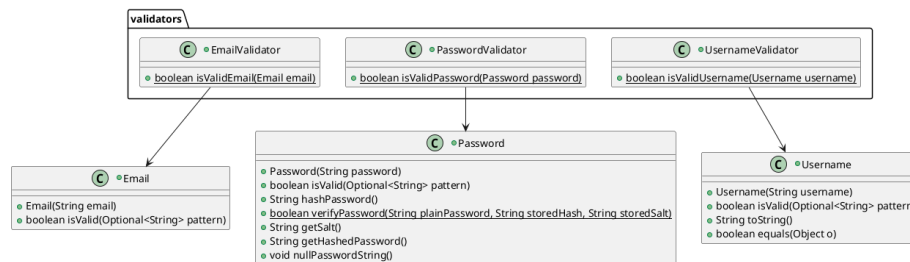


Figure 48: UML davor

## UML davor (Auswahl)

### Lösung: Einführen eines gemeinsamen Interfaces

**Siehe Commit 615ef78145c8b22e15e69c9d2d3cdd3da923a297** Die Validator wurden alle mithilfe eines gemeinsamen Interface `Validatables` zusammengefasst. Klassen, die auf diese Validator beruhen, wurden ebenfalls in den `Validatables` zusammengefasst. Dies führte nicht nur zu einer standardisierten Validierungslogik, sondern auch zu einer verbesserten Codeorganisation mit klaren Methodennamen.

## UML danach:

## Entwurfsmuster

- 2 unterschiedliche Entwurfsmuster aus der Vorlesung (oder nach Absprache auch andere) jeweils benennen, sinnvoll einsetzen, begründen und UML-Diagramm

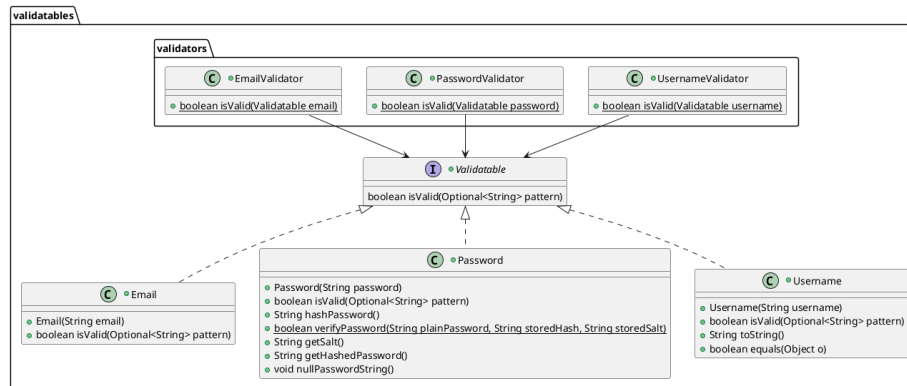


Figure 49: UML danach

## Entwurfsmuster #1: Adapter

Adapter ermöglichen die Zusammenarbeit von Klassen, dessen Schnittstellen eigentlich nicht kompatibel sind.

### LogWriterLoggerAdapter

#### UML (vereinfacht)

- Der Hauptzweck des **LogWriterLoggerAdapter** ist es, einen **LogWriter** als **Logger** verwenden zu können, da beide Klassen separate Aufgaben besitzen.
- Das **Logger** Interface ist für das aufsammeln der Aktivitäten innerhalb der Anwendung da, während der **LogWriter** für das Schreiben der eigentlichen Logeinträge in Dateien zuständig ist.
- Durch den Adapter ist es möglich, dass Code, der mit der **Logger**-Schnittstelle arbeitet, einen **LogWriter** zu übergeben, damit z.b. die gesammelten Logs in einer Datei gespeichert werden können.

## Entwurfsmuster #2: Decorator

Decorator sind eine Art Strukturmuster, die es ermöglichen, Objekten dynamisch zusätzliche Funktionalitäten hinzuzufügen, ohne deren Struktur zu verändern.

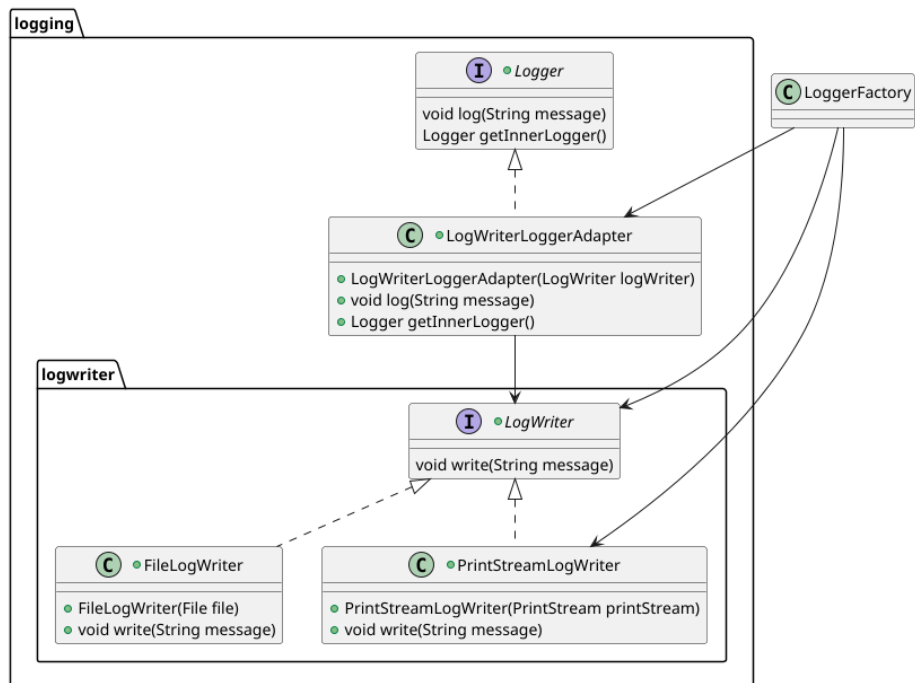


Figure 50: UML

## Das Logger-System

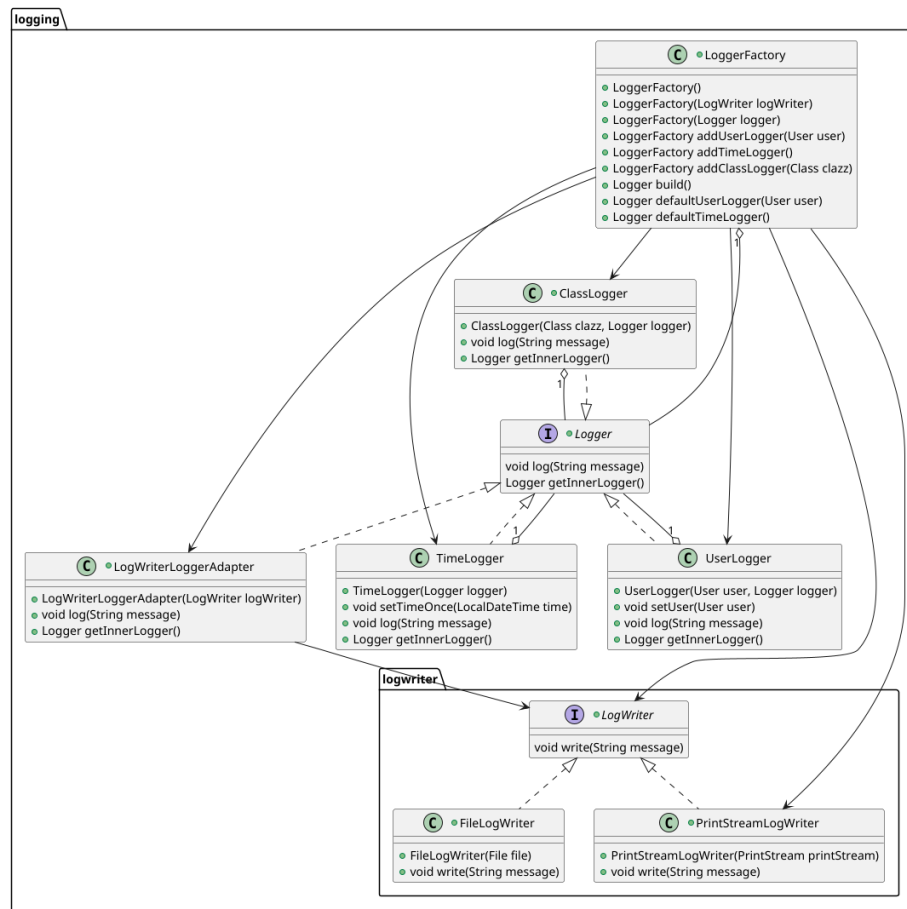


Figure 51: UML

- Das **Logger** Interface definiert die Grundoperationen, die jeder Logger unterstützen muss. Die `getInnerLogger()` Methode ermöglicht den Zugriff auf den eingewickelten Logger.
- Die konkreten Decorator-Klassen hier sind die **TimeLogger**, **UserLogger** und **ClassLogger**. Sie fügen den Logeinträgen jeweils einen Zeitstempel, Benutzerinformationen und Klassennamen hinzu.



- Mithilfe der `LoggerFactory` können die verschiedenen Decorator-Klassen flexibel kombiniert werden.

```
// Beispielhafte Erstellung eines Loggers mit mehreren Decoratoren
Logger complexLogger = factory.addClassLogger(MyClass.class)
                        .addUserLogger(currentUser)
                        .addTimeLogger()
                        .build();
```

Figure 52: alt

---

**Vielen Dank für die Aufmerksamkeit**