

THIRSTY CALC

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>
- ./gradlew shadowJar
- Die gebaute .jar ist dann in build/libs zu finden.

TECHNISCHER ÜBERBLICK

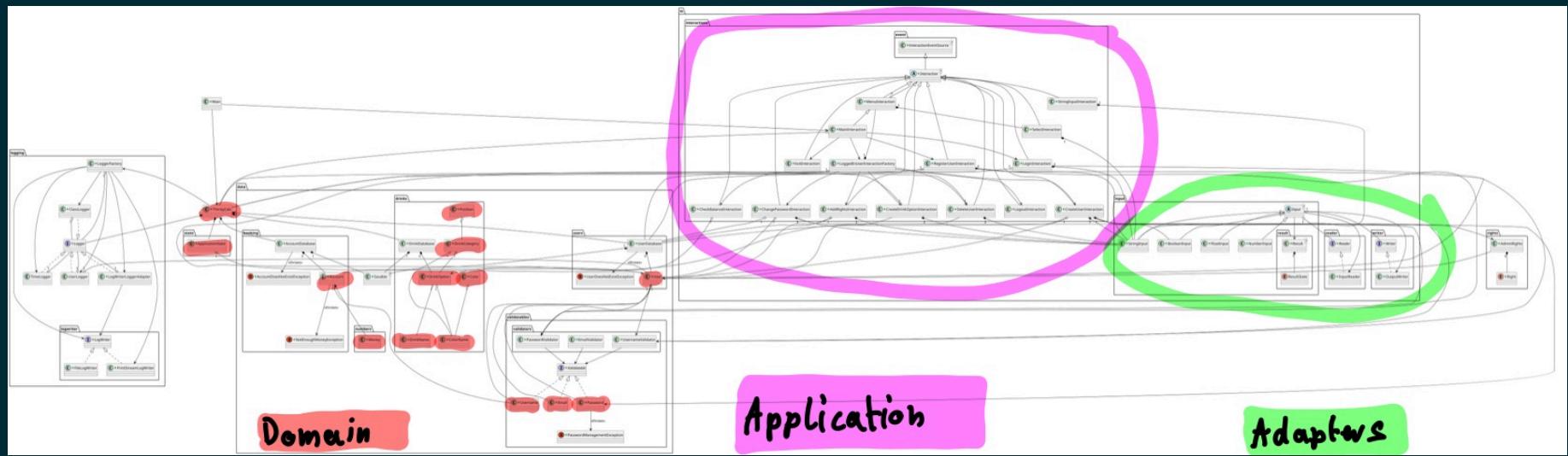
- Java (21+)
- Gradle mit Kotlin DSL (lauffähige Version der Software)
- SonarCloud (für statische Codeanalyse -> Qualität)
- Github Actions (Cloc, Tests, Sonar)

SOFTWAREARCHITEKTUR (PAUL)

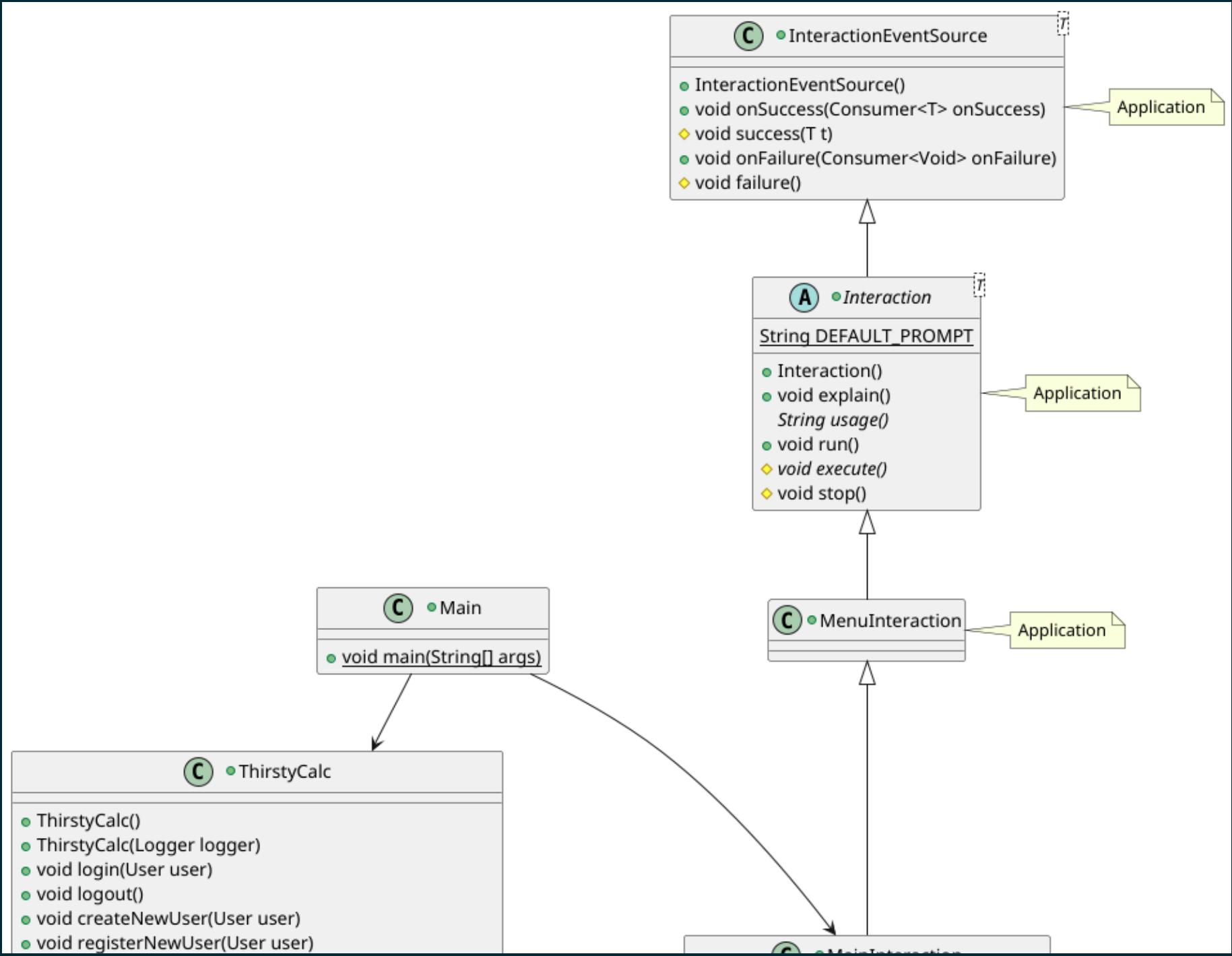
GEWÄHLTE ARCHITEKTUR (4P)

- [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.



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:

```
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 + ")");
    }

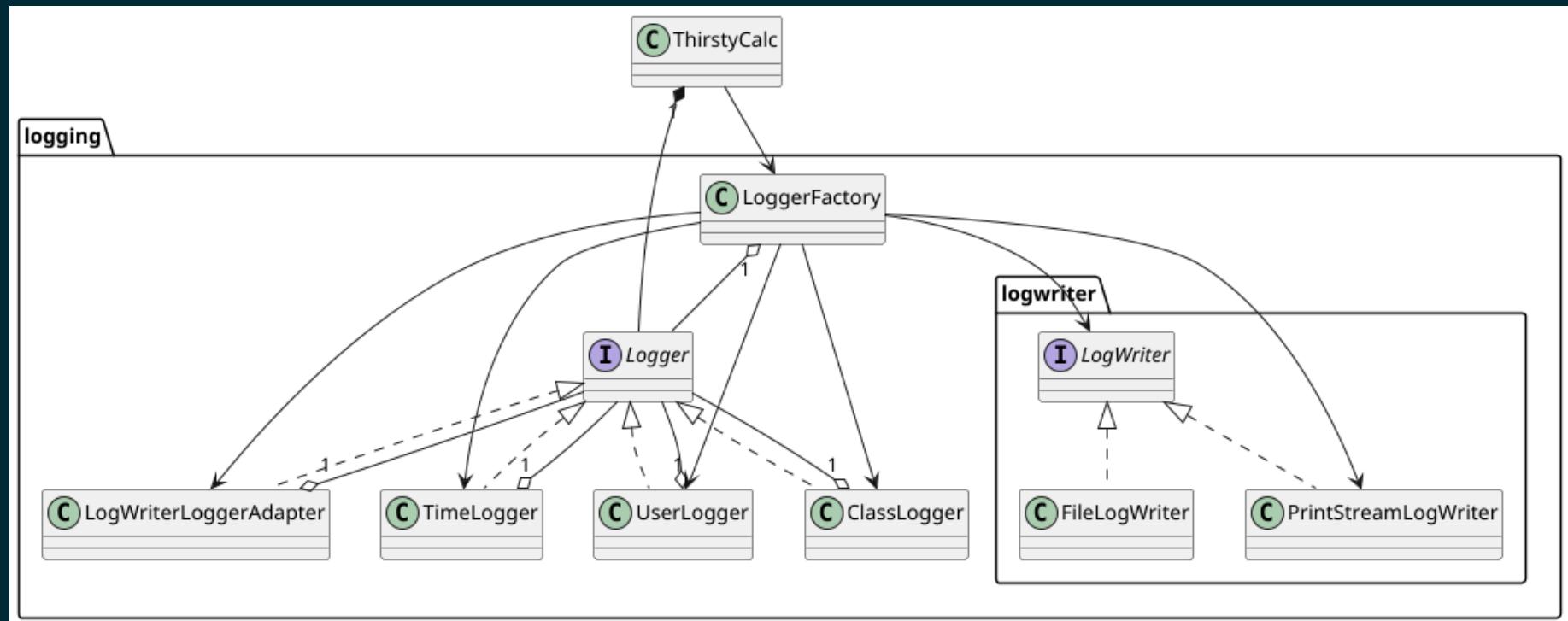
    balance = balance.subtract(amount);
    return amount;
}
```

ANALYSE DER DEPENDENCY RULE (3P)

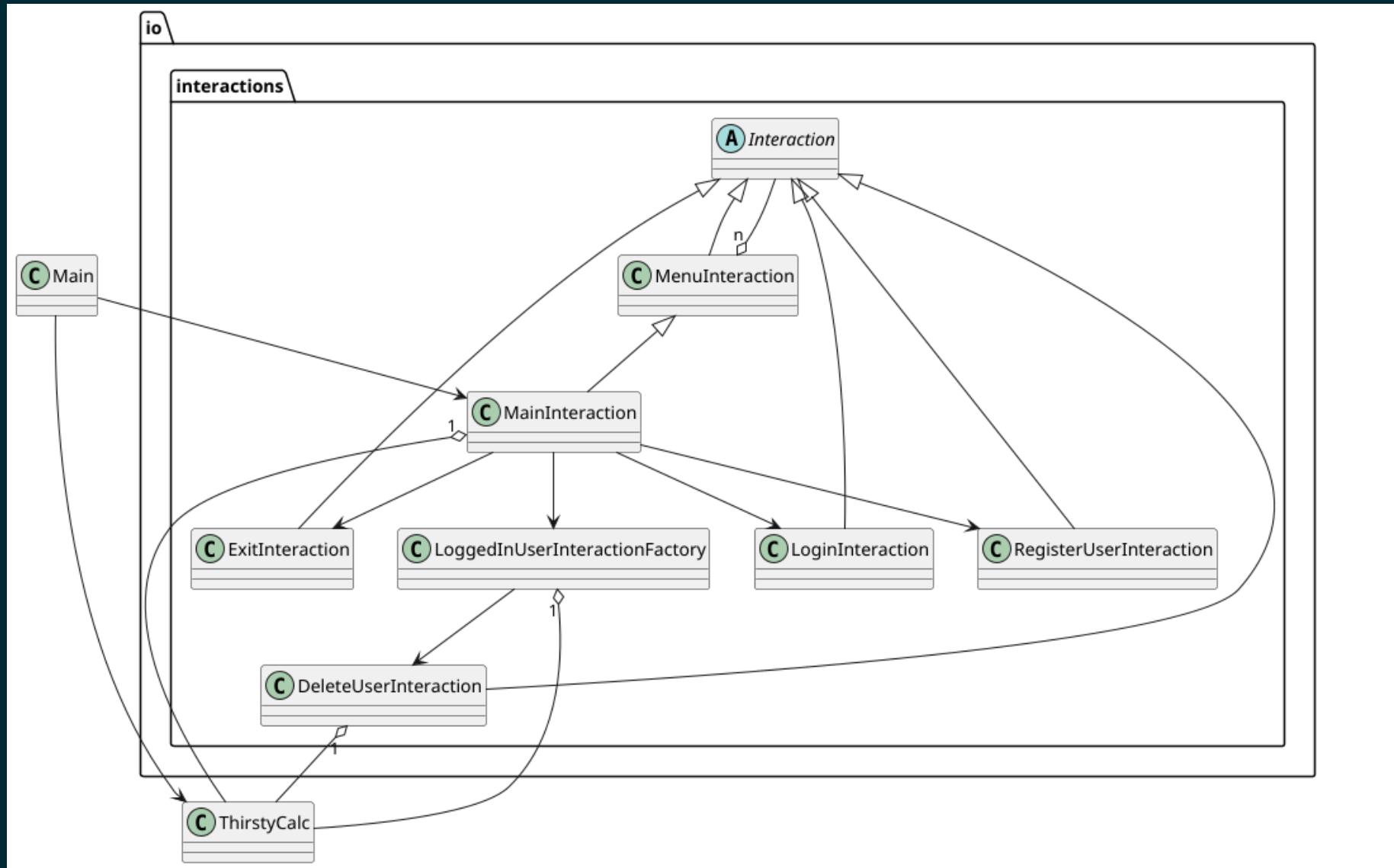
- [In der Vorlesung wurde im Rahmen der ‘Clean Architecture’ die s.g. Dependency Rule vorgestellt. Je 1 Klasse zeigen, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML (mind. die betreffende Klasse inkl. der Klassen, die von ihr abhängen bzw. von der sie abhängt) 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”

POSITIV-BEISPIEL: DEPENDENCY RULE



NEGATIV-BEISPIEL: DEPENDENCY RULE



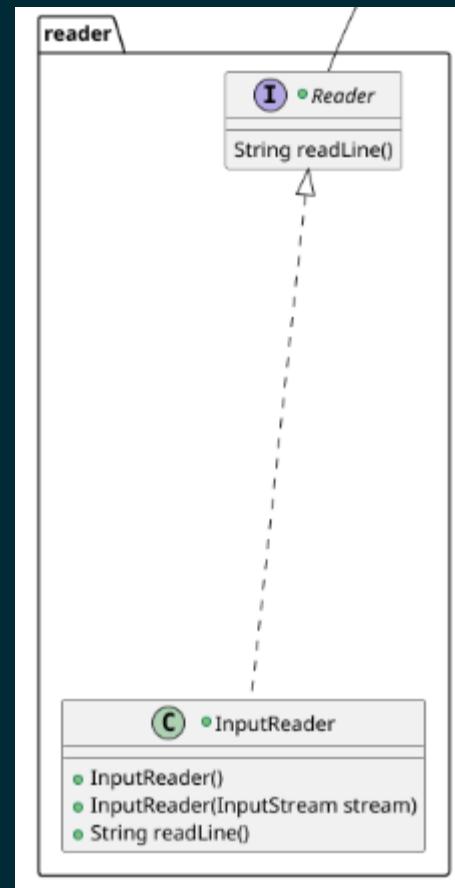
SOLID (PAUL)

ANALYSE SRP (3P)

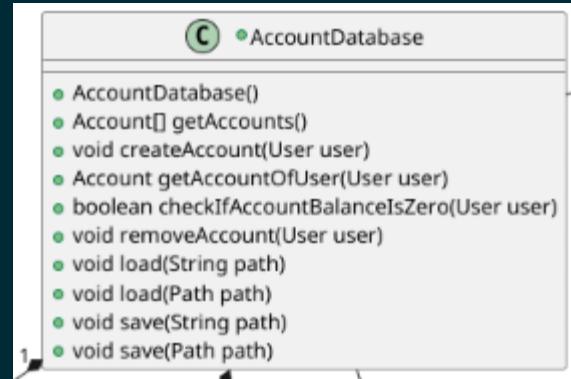
- [jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]

POSITIV-BEISPIEL

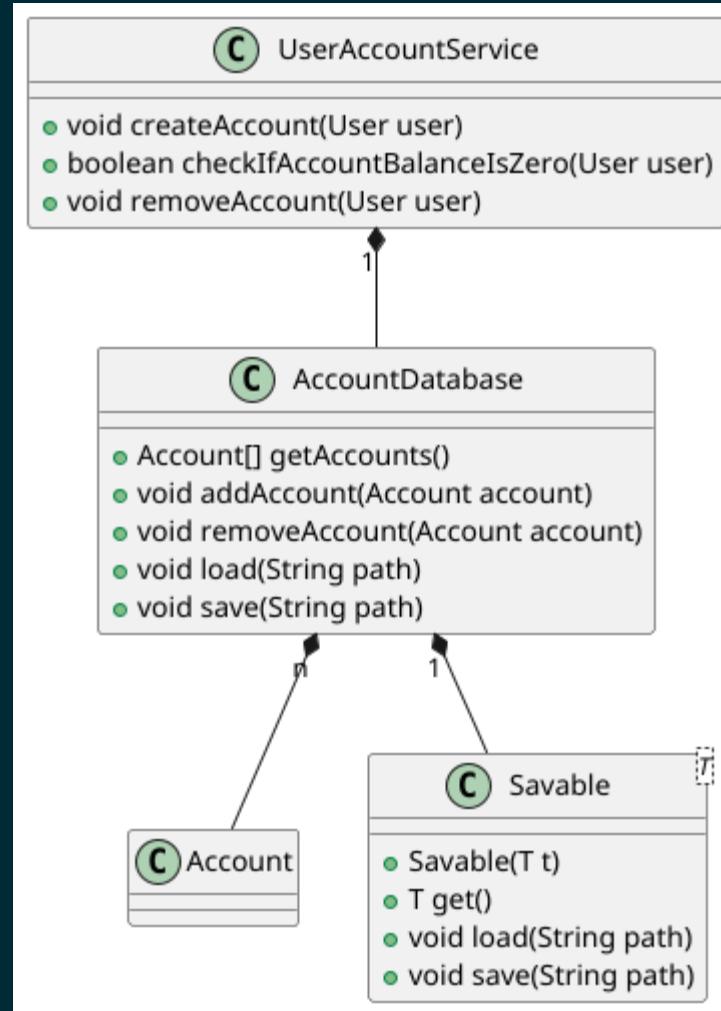
Die Klasse `InputReader` hat die einzige Aufgabe, eine Eingabezeile aus einem `InputStream` zu lesen.



NEGATIV-BEISPIEL



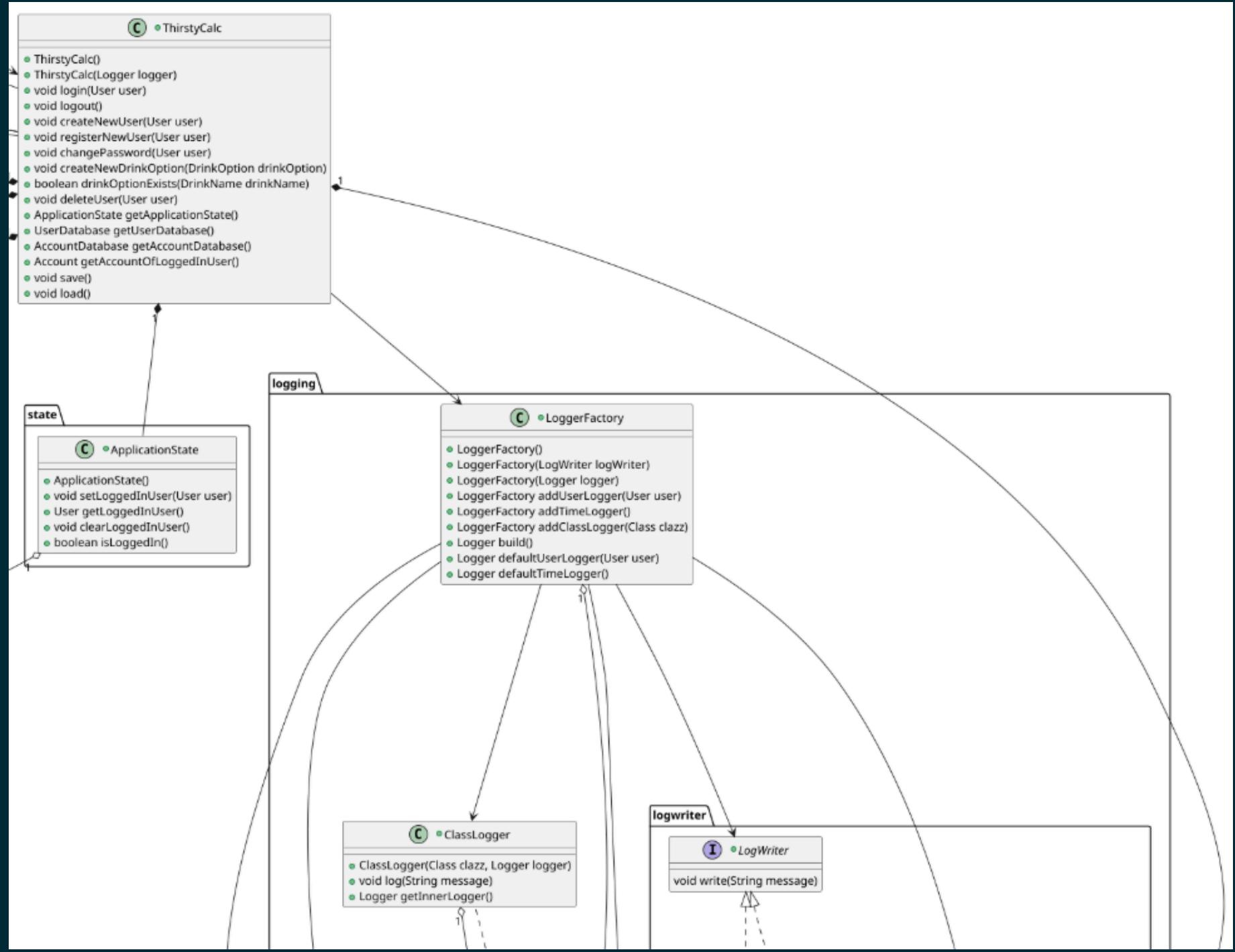
Eine mögliche Lösung, um das SRP für AccountDatabase umzusetzen, ist im folgenden UML-Diagramm dargestellt:



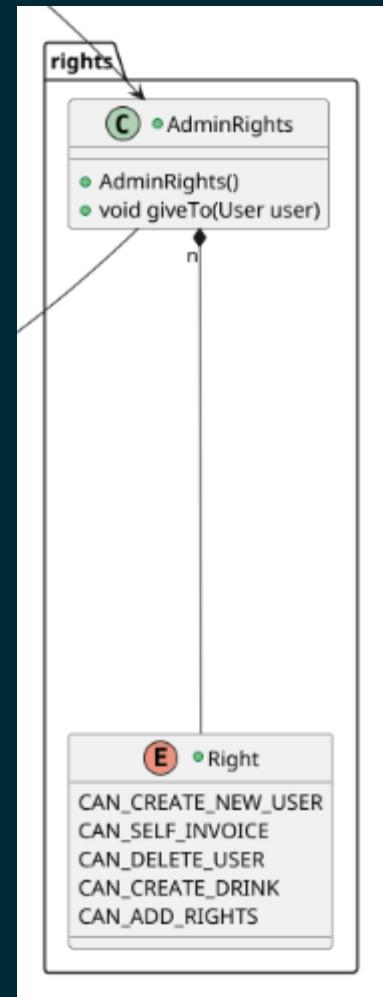
ANALYSE OCP (3P)

- [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)?]

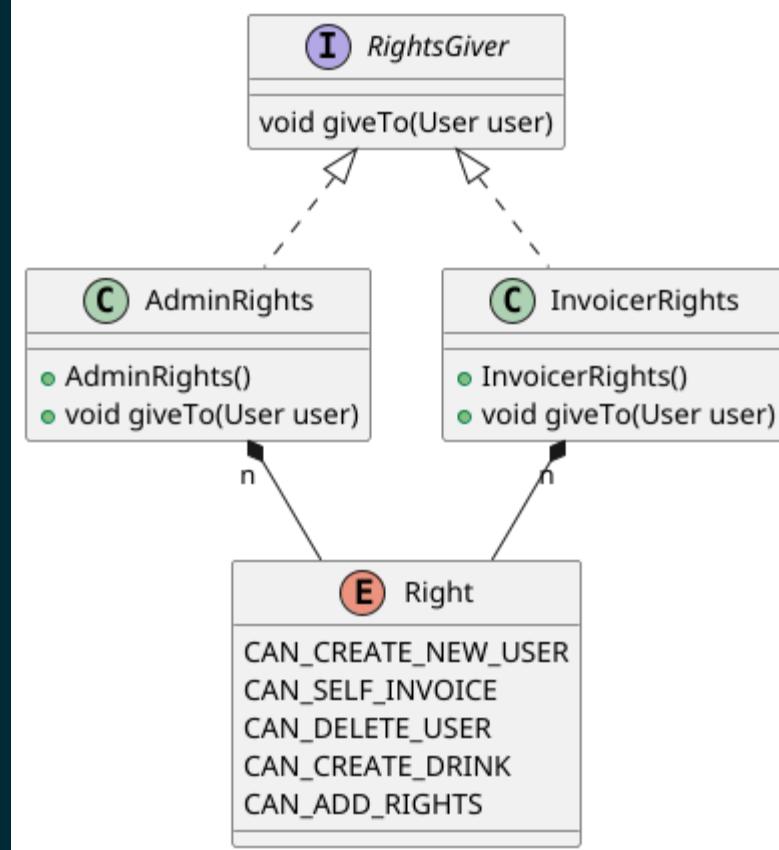
POSITIV-BEISPIEL



NEGATIV-BEISPIEL



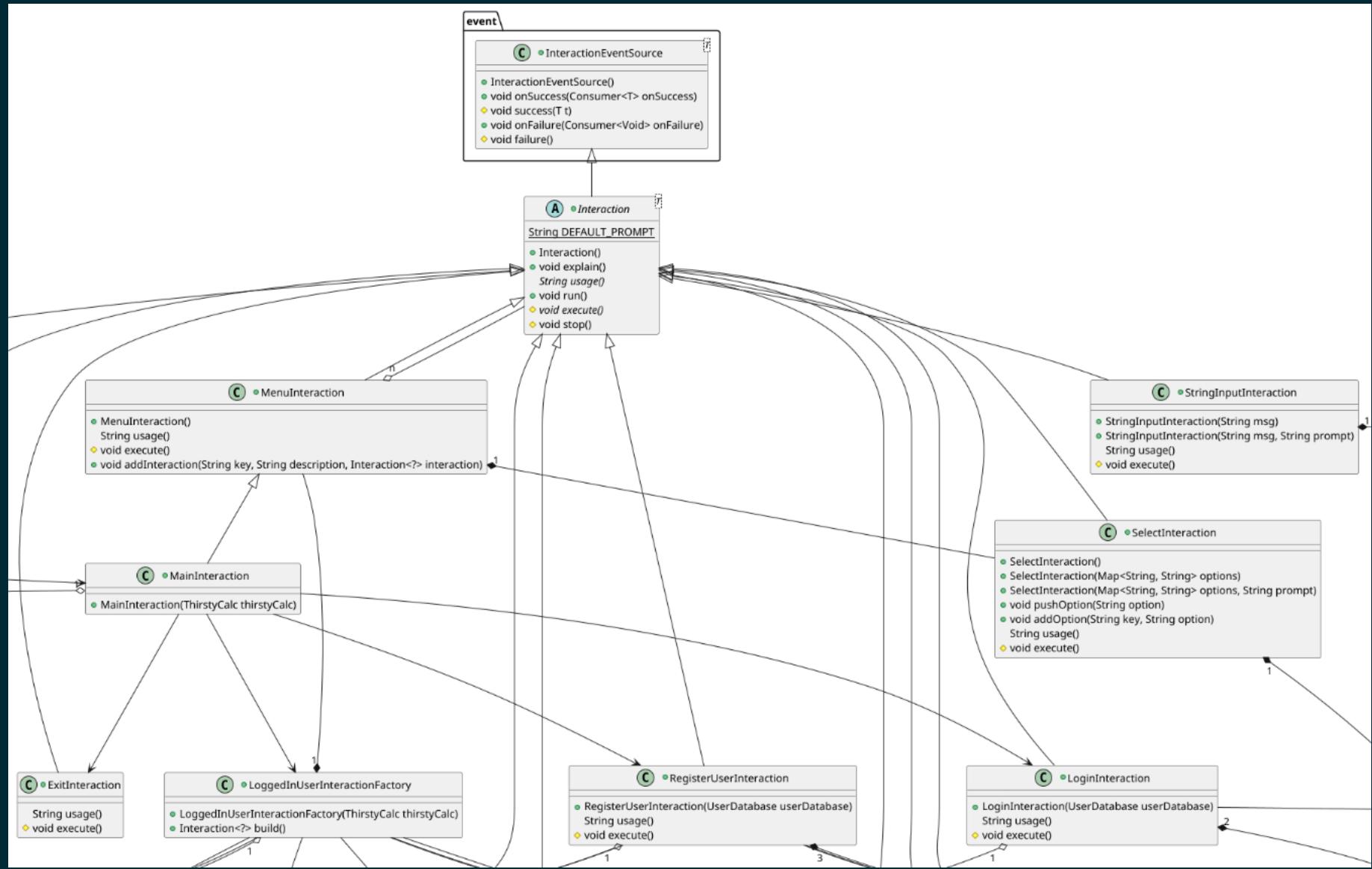
LÖSUNG FÜR DAS NEGATIV-BEISPIEL



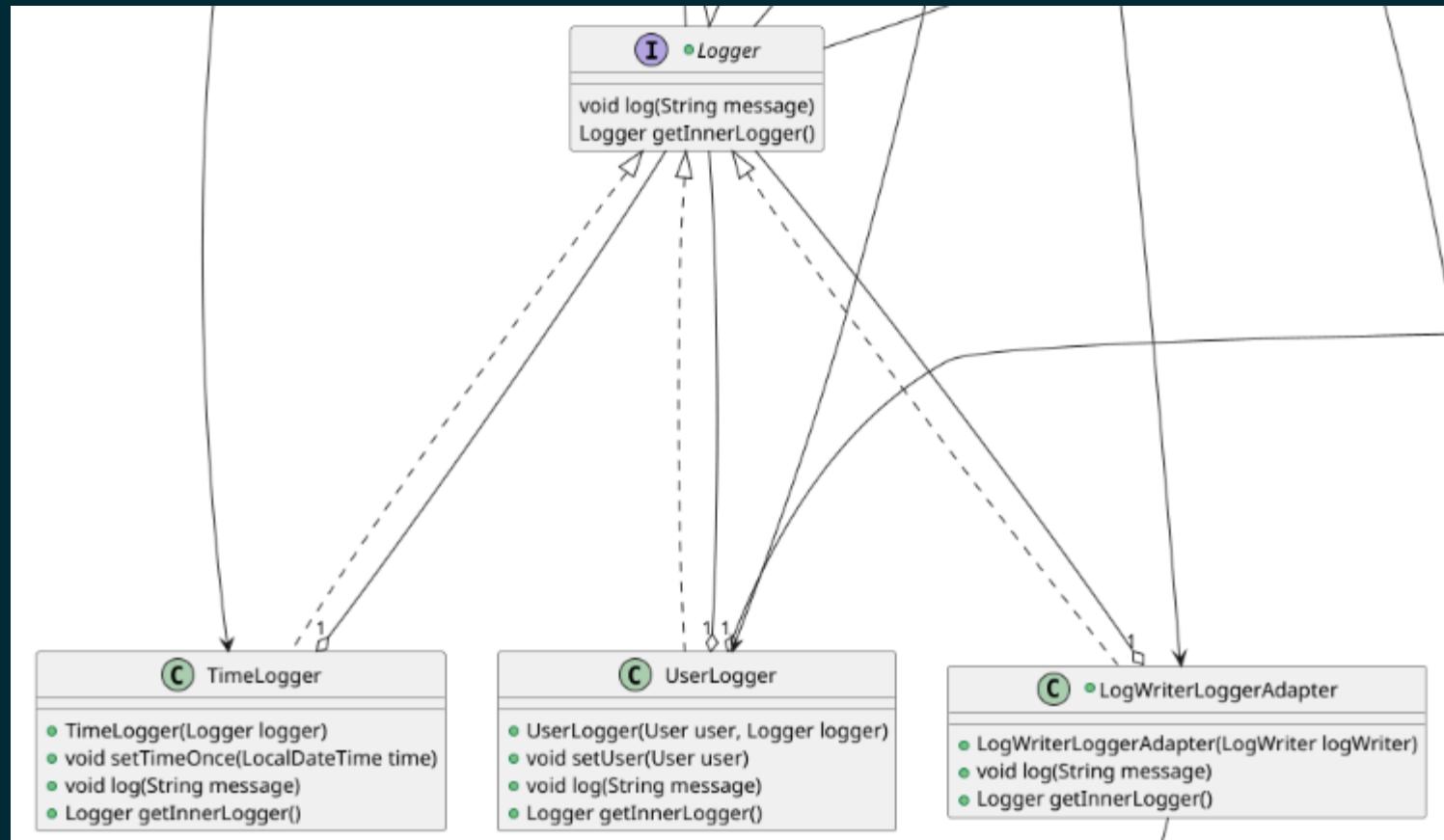
ANALYSE [LSP/ISP/DIP] (2P)

- [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]

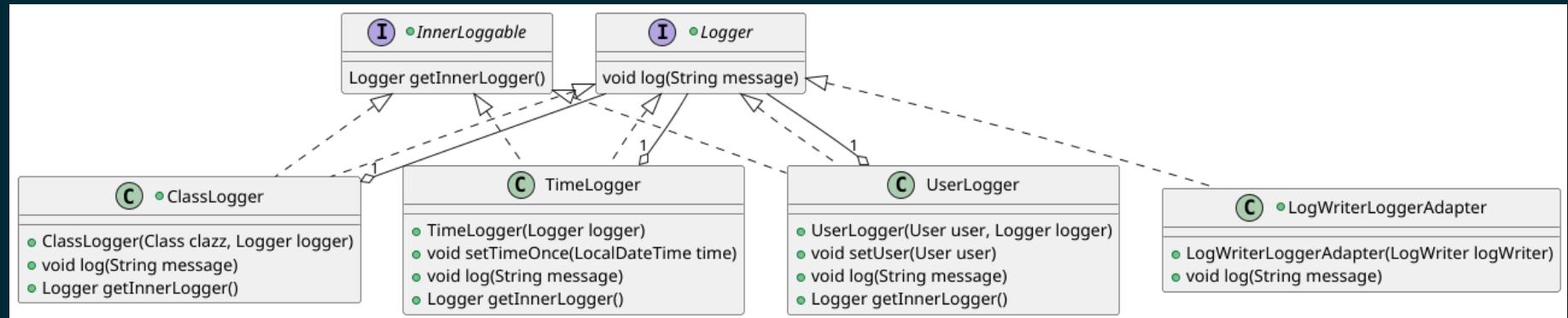
POSITIV-BEISPIEL: LSP



NEGATIV-BEISPIEL: LSP



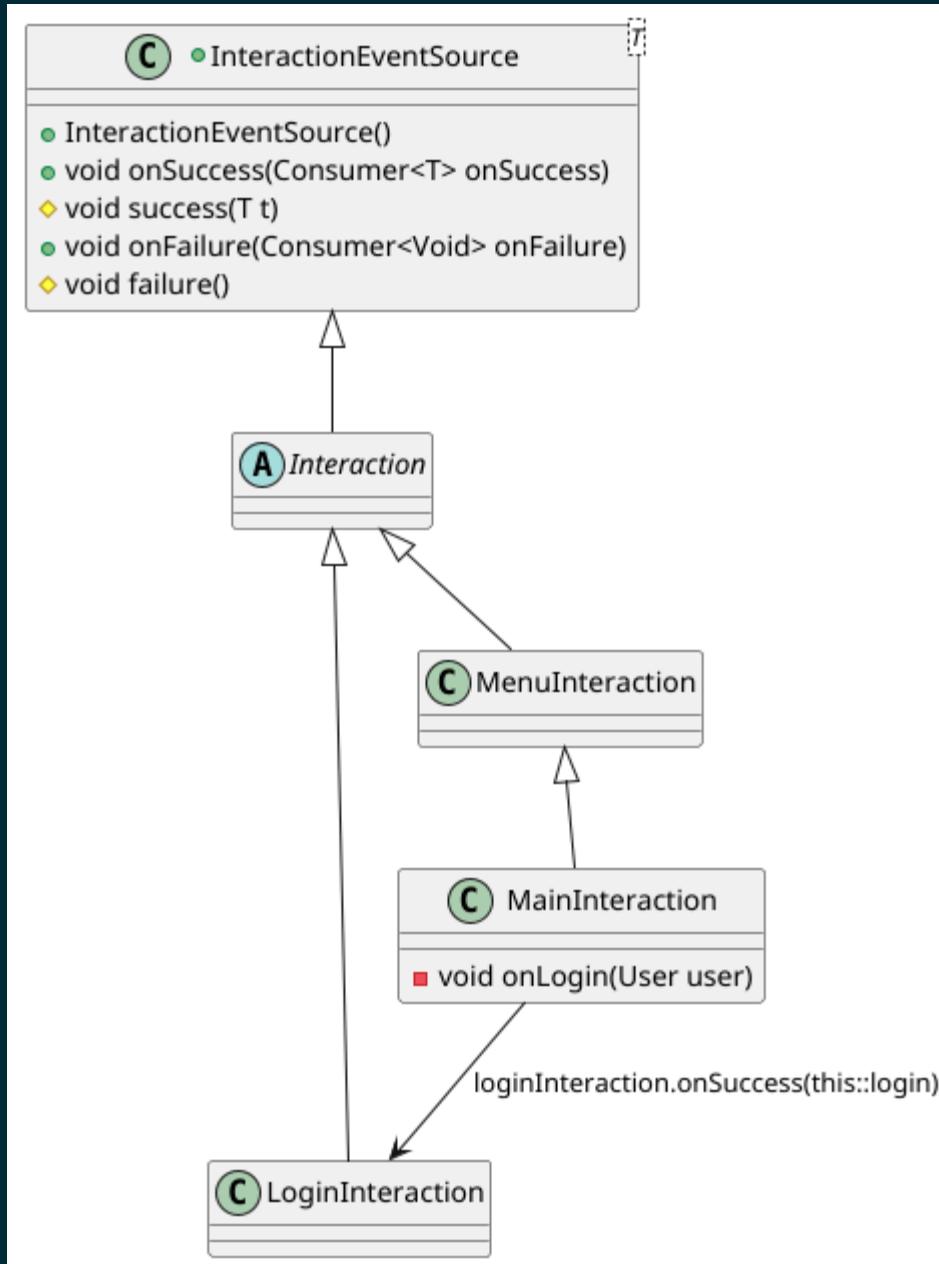
LÖSUNG FÜR DAS NEGATIV-BEISPIEL: LSP



WEITERE PRINZIPIEN (PAUL)

ANALYSE GRASP: GERINGE KOPPLUNG (3P)

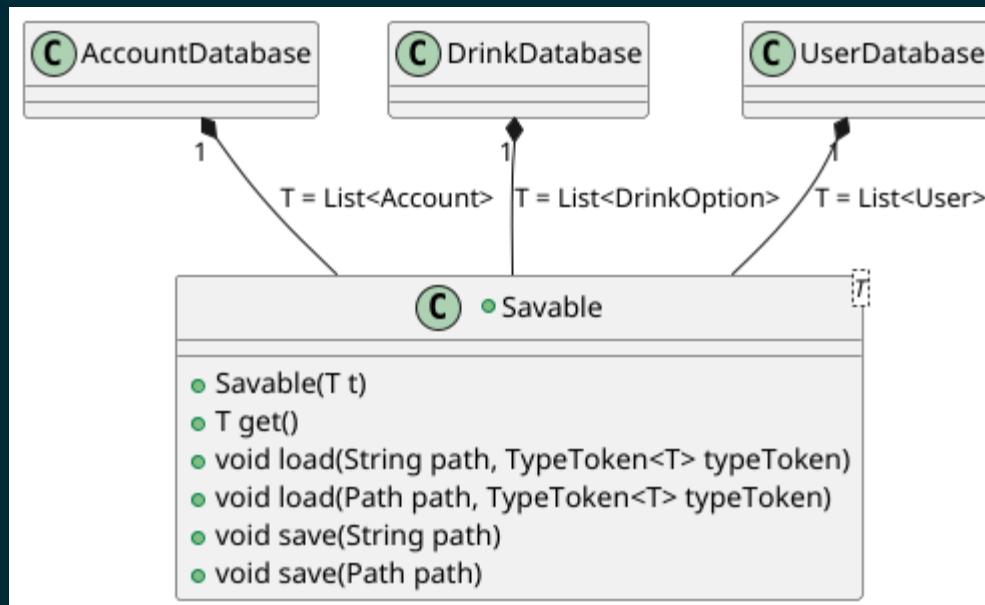
- [eine bis jetzt noch nicht behandelte Klasse als positives Beispiel geringer Kopplung; UML mit zusammen spielenden Klassen, Aufgabenbeschreibung der Klasse und Begründung, warum hier eine geringe Kopplung vorliegt; es müssen auch die Aufrufer/Nutzer der Klasse berücksichtigt werden]



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



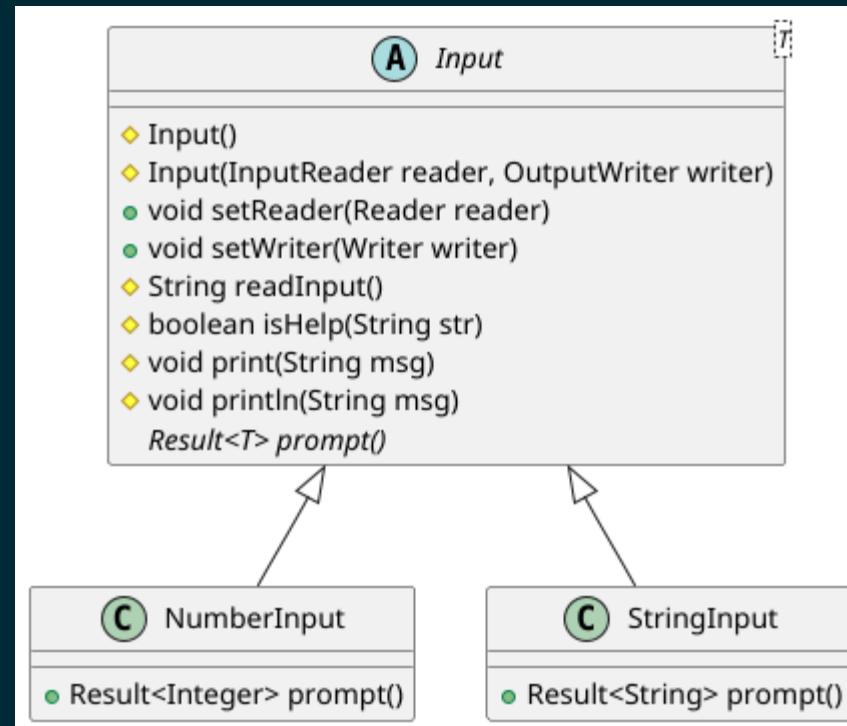
DRY (2P)

- [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]

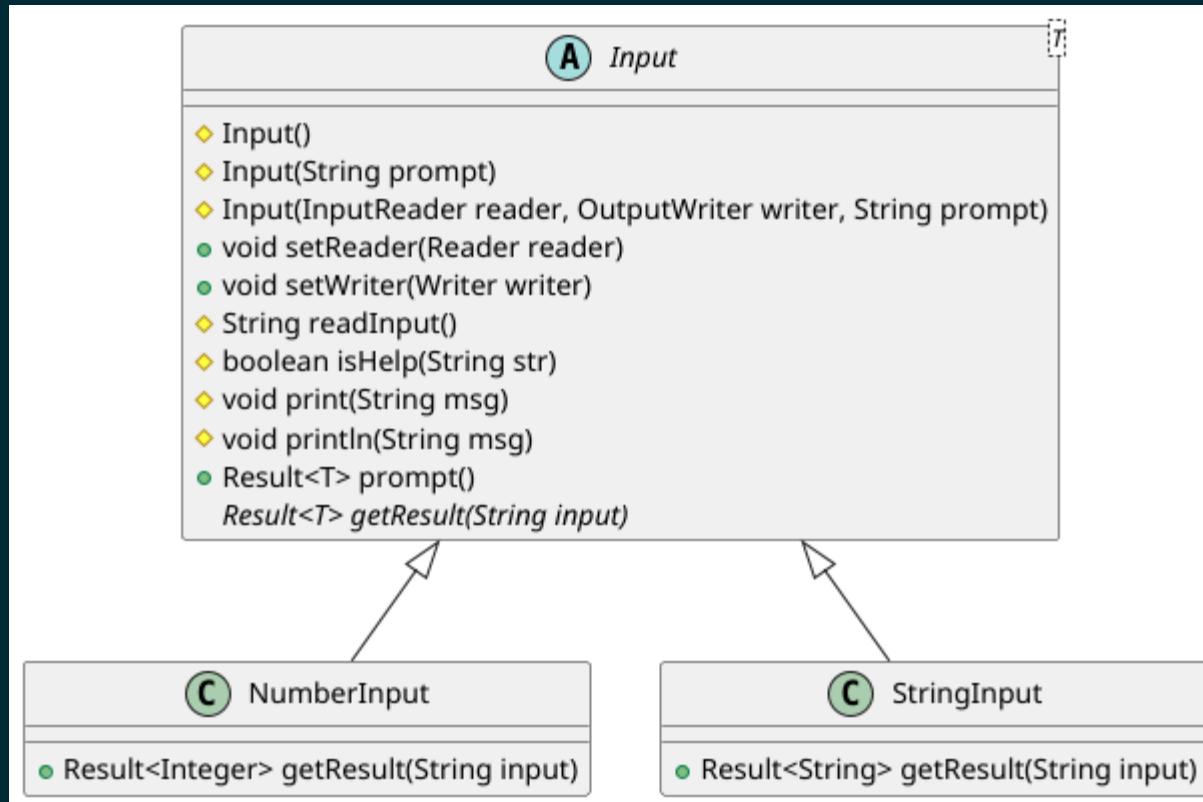
git commit:

7050c4c57c00a0a52a48088c0a997e9fa2e227af

VORHER



NACHHER



CODE-DIFF

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

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

```
... @@ -1,36 +1,22 @@
1 package de.dhbw.karlsruhe.getraenkeabrechnung.io2.input;
2
3 import de.dhbw.karlsruhe.getraenkeabrechnung.io2.input.result.Result;
4
5 public class NumberInput extends Input<Integer> {
6
7     // todo: range? (min and max value)
8
9     -     private final String prompt;
10    -
11
12    -     public NumberInput(String prompt) {
13        -         this.prompt = prompt;
14    }
15
16    @Override
17    -     public Result<Integer> prompt() {
18        -         print(prompt);
19
20        -         String in = readInput();
21
22        -         if (isHelp(in)) {
23            -             return Result.help();
24        }
25
26        -         if (in.isEmpty()) {
27            -             return Result.none();
28        }
29
30        -         try {
31            -             Integer res = Integer.valueOf(in);
32            -             return Result.some(res);
33        } catch (NumberFormatException e) {
34            -             return Result.none();
35        }
36    }
```

```
1     package de.dhbw.karlsruhe.getraenkeabrechnung.io2.input;
2
3     import de.dhbw.karlsruhe.getraenkeabrechnung.io2.input.result.Result;
4
5     public class NumberInput extends Input<Integer> {
6
7         // todo: range? (min and max value)
8
9         public NumberInput(String prompt) {
10            +     super(prompt);
11        }
12
13        @Override
14        +     public Result<Integer> getResult(String input) {
15
16            try {
17                +             Integer res = Integer.valueOf(input);
18                +             return Result.some(res);
19            } catch (NumberFormatException e) {
20                +                 return Result.none();
21            }
22        }
```

src/main/java/de/dhbw/karlsruhe/getraenkeabrechnung/io2/input/StringInput.java

+3 -19 00000 ...

...

@@ -1,31 +1,15 @@

```
1 package de.dhbw.karlsruhe.getraenkeabrechnung.io2.input;
2
3 import de.dhbw.karlsruhe.getraenkeabrechnung.io2.input.result.Result;
4
5 public class StringInput extends Input<String> {
6
7 -     private final String prompt;
8 -
9     public StringInput(String prompt) {
10 -         super();
11 -         this.prompt = prompt;
12     }
13
14     @Override
15 -     public Result<String> prompt() {
16 -         print(prompt);
17 -
18 -         String res = readInput();
19 -
20 -         if (isHelp(res)) {
21 -             return Result.help();
22 -         }
23 -
24 -         // return an empty optional if the returned string is blank
25 -         if (res.isBlank()) {
26 -             return Result.none();
27 -         }
28 -
29 -         return Result.some(res);
30     }
31 }
```

```
1 package de.dhbw.karlsruhe.getraenkeabrechnung.io2.input;
2
3 import de.dhbw.karlsruhe.getraenkeabrechnung.io2.input.result.Result;
4
5 public class StringInput extends Input<String> {
6
7     public StringInput(String prompt) {
8         super(prompt);
9     }
10
11     @Override
12     +     Result<String> getResult(String input) {
13         +     return Result.some(input);
14     }
15 }
```

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.)

UNIT TESTS 1/2 USERNAME

```
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     @Test
33     void isValidUsernameWithDotStartShouldReturnFalse() {
34         assertFalse(UsernameValidator.isValidUsername(new Username(username: ".invalidUser123")));
35     }
36
37     @Test
38     void isValidUsernameWithHyphenStartShouldReturnFalse() {
39         assertFalse(UsernameValidator.isValidUsername(new Username(username: "-invalidUser123")));
40     }
41
42     @Test
43     void isValidUsernameWithUnderscoreEndShouldReturnFalse() {
44         assertFalse(UsernameValidator.isValidUsername(new Username(username: "invalidUser123_")));
45     }
```

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     }
47 }
```

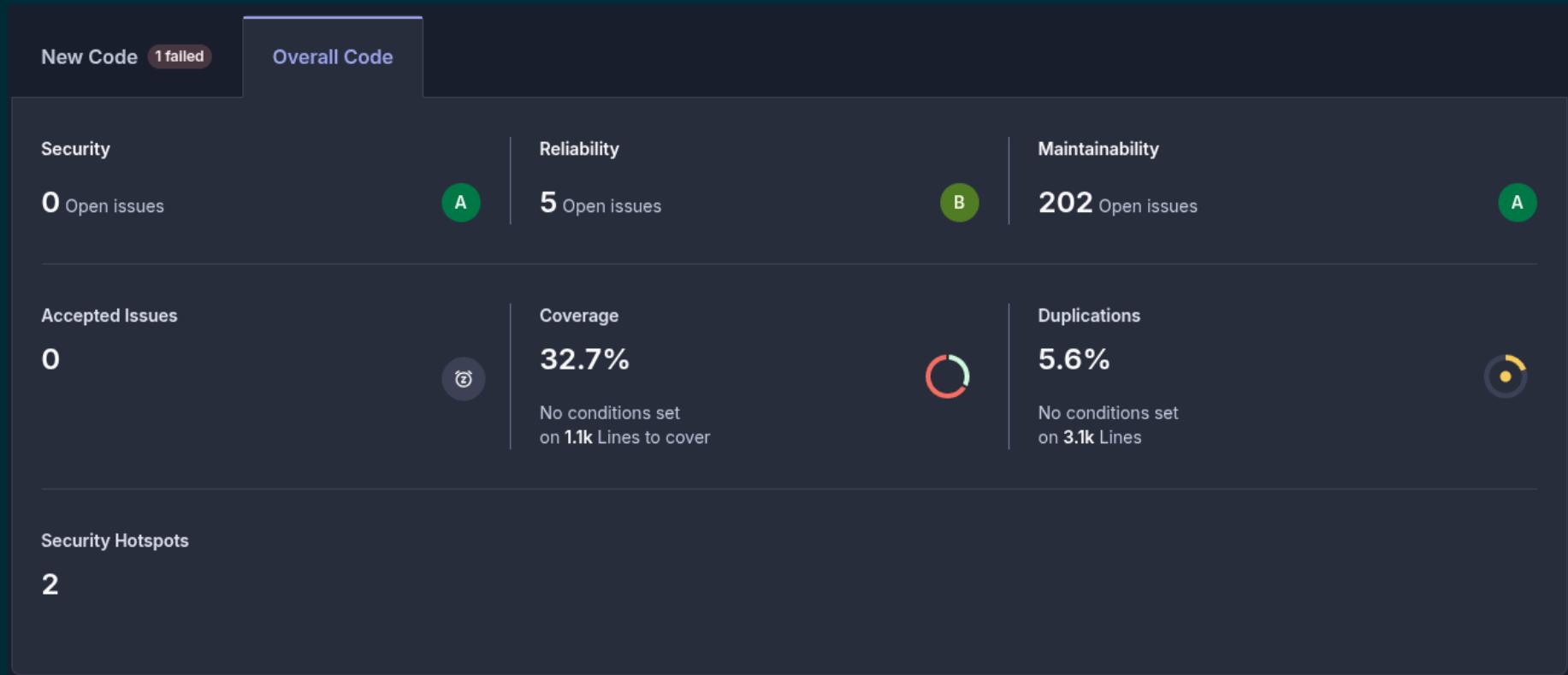
AUTOMATIC

- [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

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

```
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      }
}
```

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]

MOCK: INPUTREADERMOCK

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

IMPLEMENTIERUNG

src > test > java > io > mocks > **J** 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    }
}
```

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();

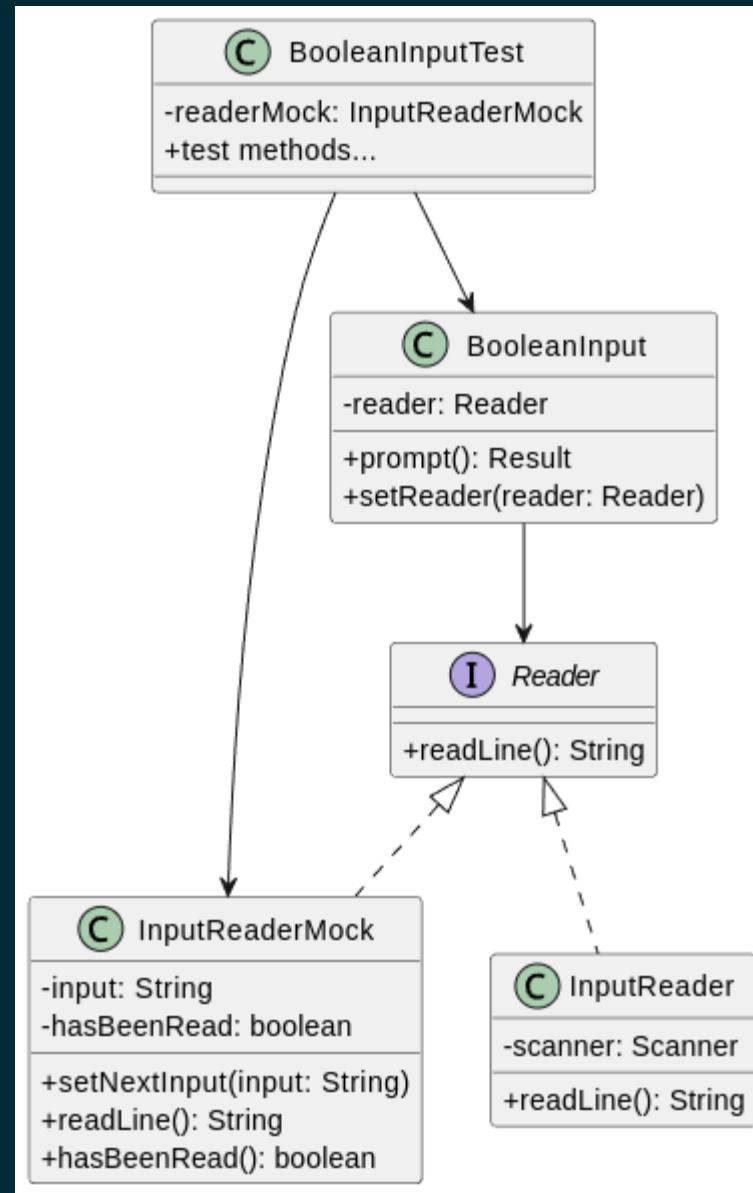
✓ StringInputTest.java src/test/java/io/input
    import io.mocks.InputReaderMock;
    private InputReaderMock readerMock;
    readerMock = new InputReaderMock();

✓ InputReaderMock.java src/test/java/io/mocks
    public class InputReaderMock implements Reader {
```

NUTZUNG

```
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     }
```

UML FÜR BEZIEHUNG



MOCK: OUTPUTWRITERMOCK

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

IMPLEMENTIERUNG

src > test > java > io > mocks > **J** 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;
```

NUTZUNG

```
✓ 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();

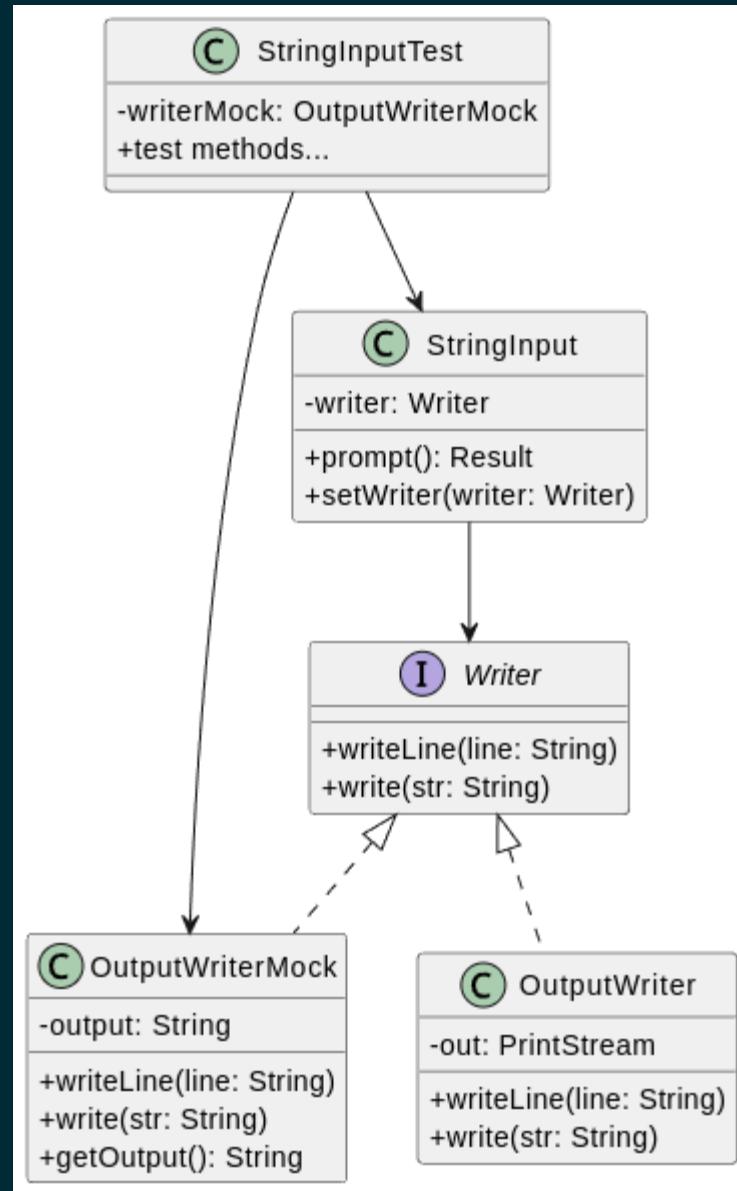
✓ StringInputTest.java src/test/java/io/input
    import io.mocks.OutputWriterMock;
    private OutputWriterMock writerMock;
    writerMock = new OutputWriterMock();

✓ OutputWriterMock.java src/test/java/io/mocks
    public class OutputWriterMock implements Writer {
```

NUTZUNG

```
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     }
```

UML FÜR BEZIEHUNG



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).
Balance	Beschreibt Guthaben des Anwenders	Guthaben ist aufgrund Sinn des Programms auch für

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

Bezeichnung Bedeutung

Begründung

Kategorie (Name,
Preis)

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]
- Entity: Hat ID & ist veränderbar

ENTITY: DRINKNAME

C

•DrinkOption

- DrinkOption(DrinkName drinkName, Color color)
- DrinkName getDrinkName()
- void setDrinkName(DrinkName drinkName)
- Color getColor()
- void setColor(Color color)

1

1

C

•DrinkName

- DrinkName(String drinkName)
- String toString()
- boolean equals(Object o)

C

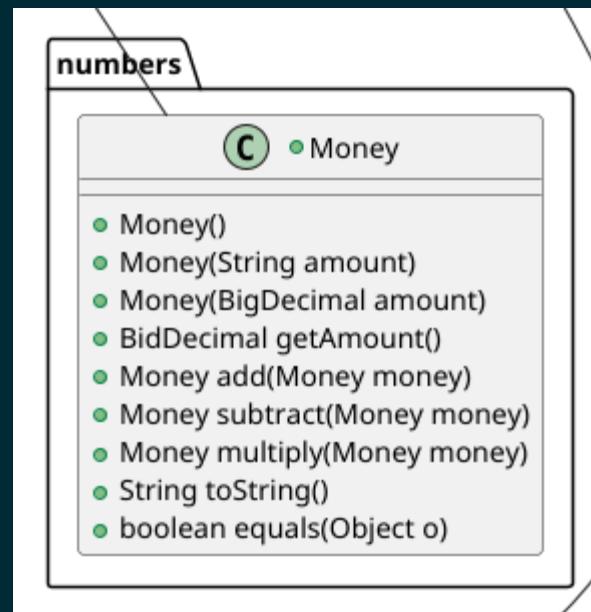
•Color

- Color(ColorName colorName)
- ColorName getColorName()
- void setColorName(ColorName colorName)

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

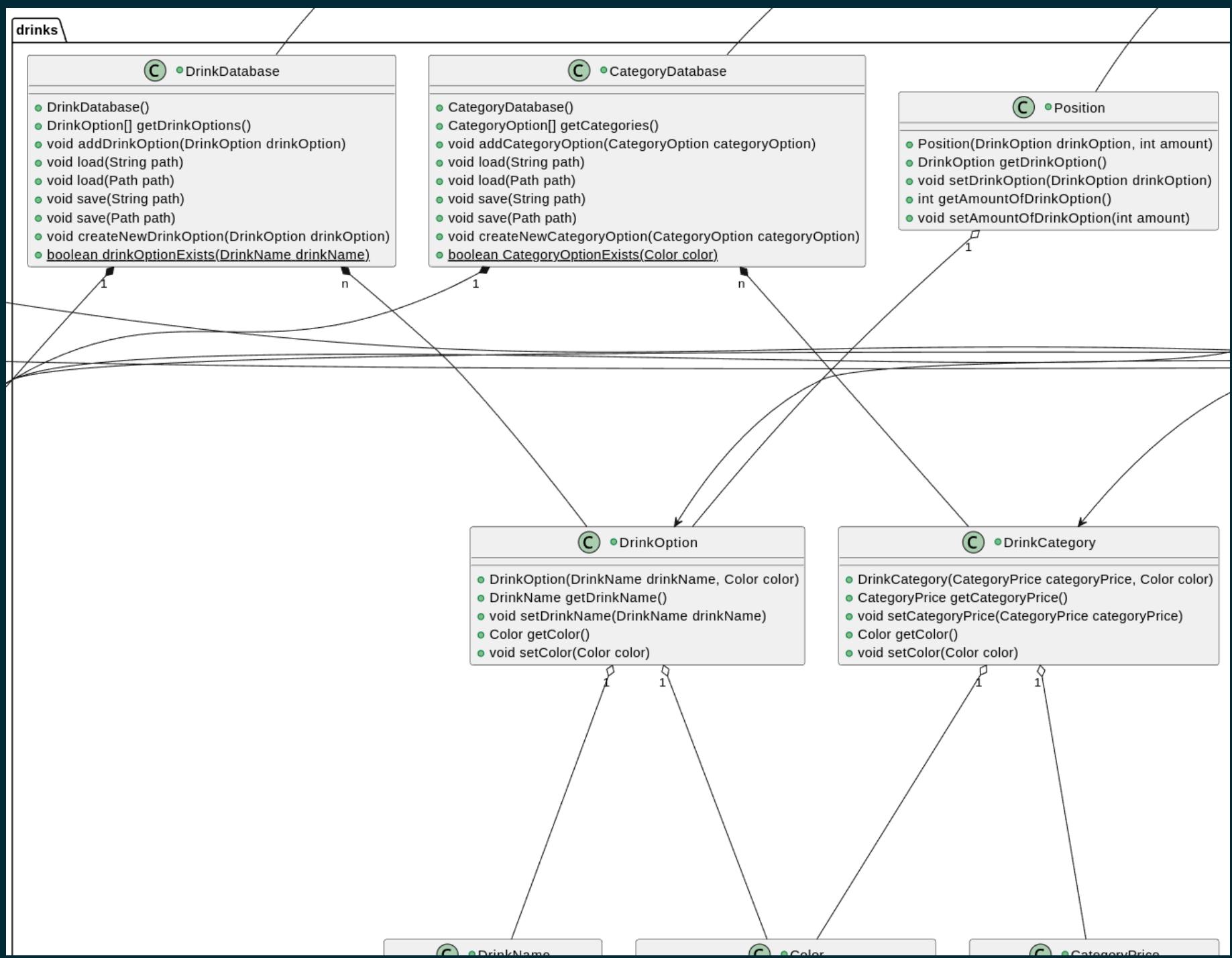


AGGREGATES

- [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

drinks



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();  
        }  
    }  
}
```



```
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();  
        }  
    }  
}
```

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 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();  
        }  
    // [...]  
}
```

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:

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

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.

```
// 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;
}
```

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

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;

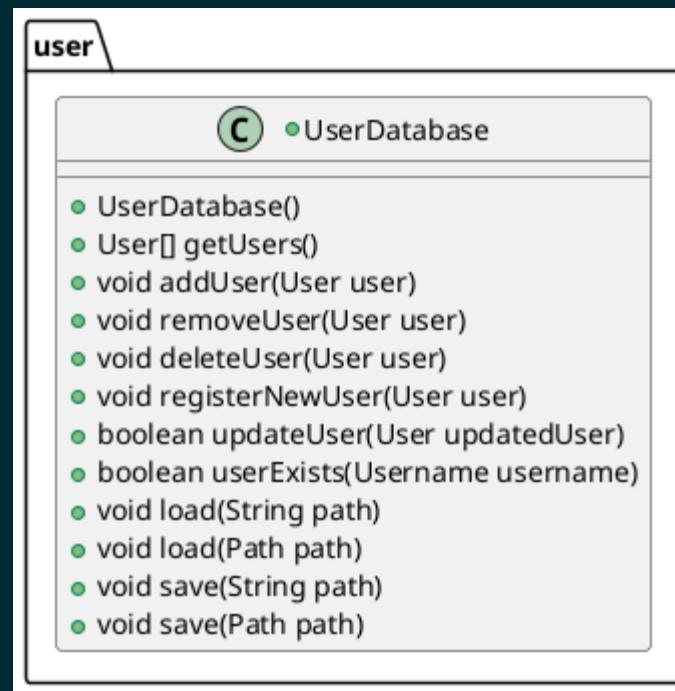
public class UserDoesNotExistException extends RuntimeException {
    public UserDoesNotExistException(String message) {
        super(message);
    }
}
```

```
//Implementation in der UserDatabase Klasse

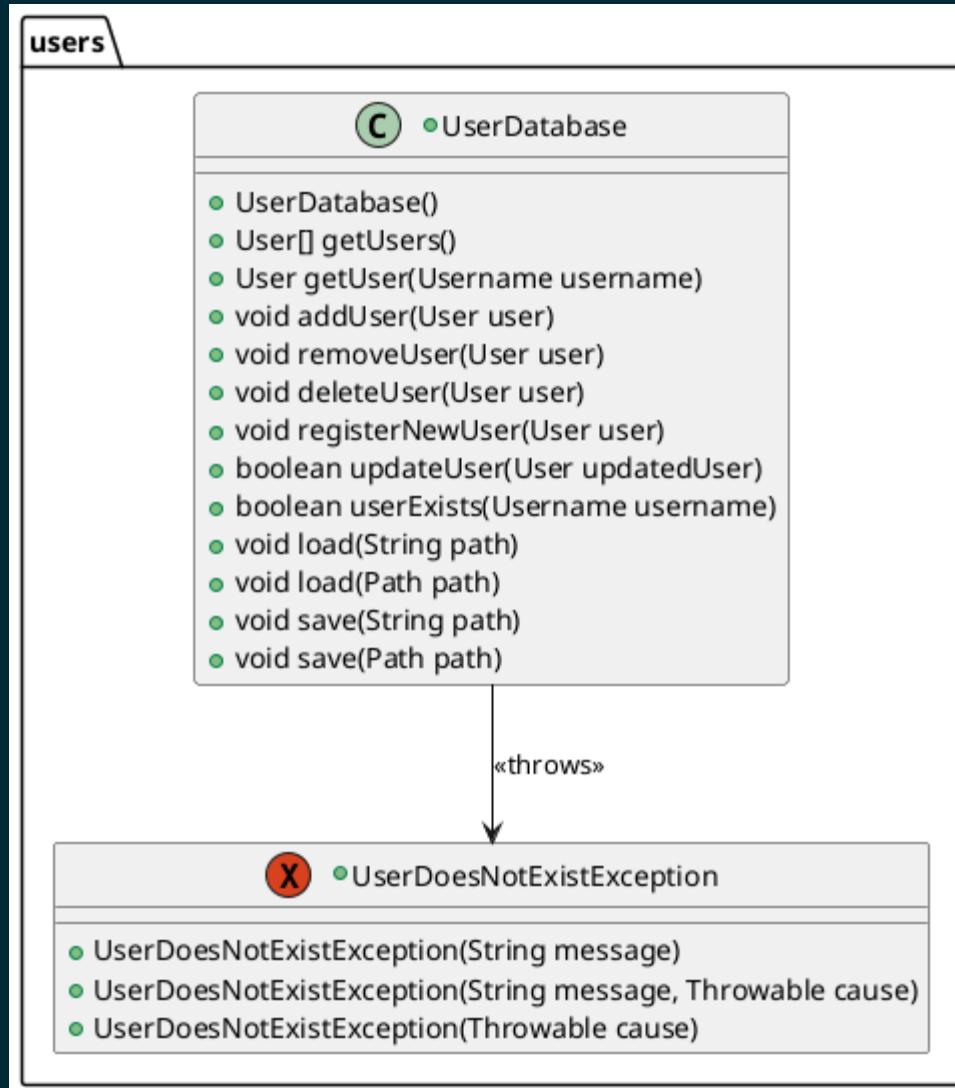
public User getUserByUsername(Username username) {
    for (User u : users.get()) {
        if (u.getUsername().equals(username)) {
            return u;
        }
    }
    throw new UserDoesNotExistException("User with the username " + username + " does not exist!");
}
```

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

UML DAVOR



UML DANACH

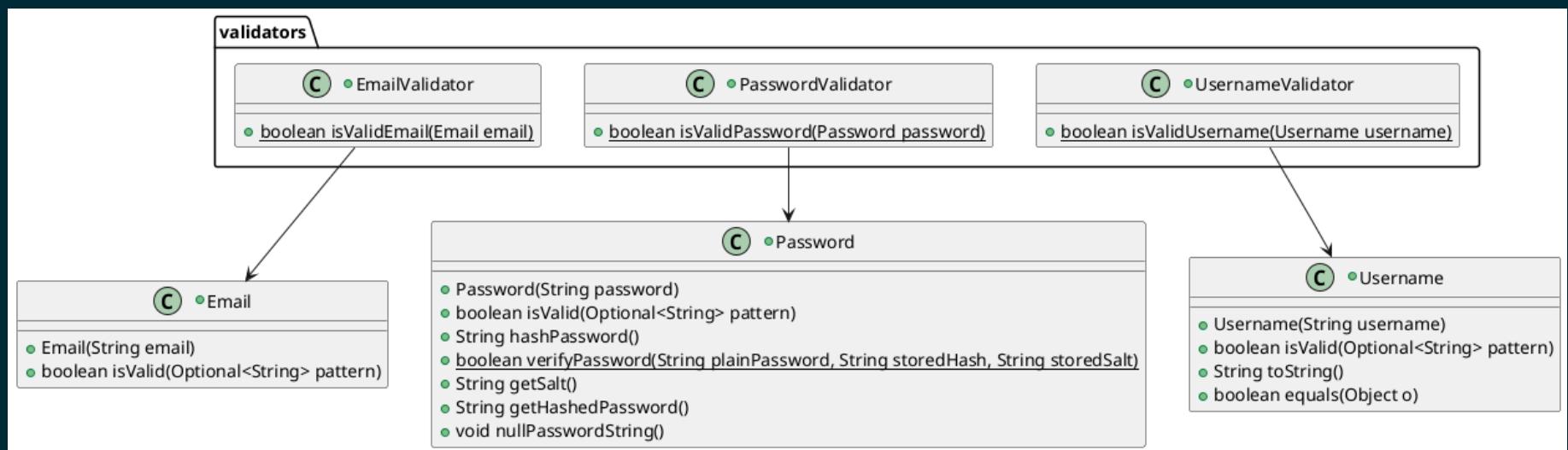


REFACTOR #2: EXTRACT METHOD/CLASS

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.

UML DAVOR (AUSWAHL)

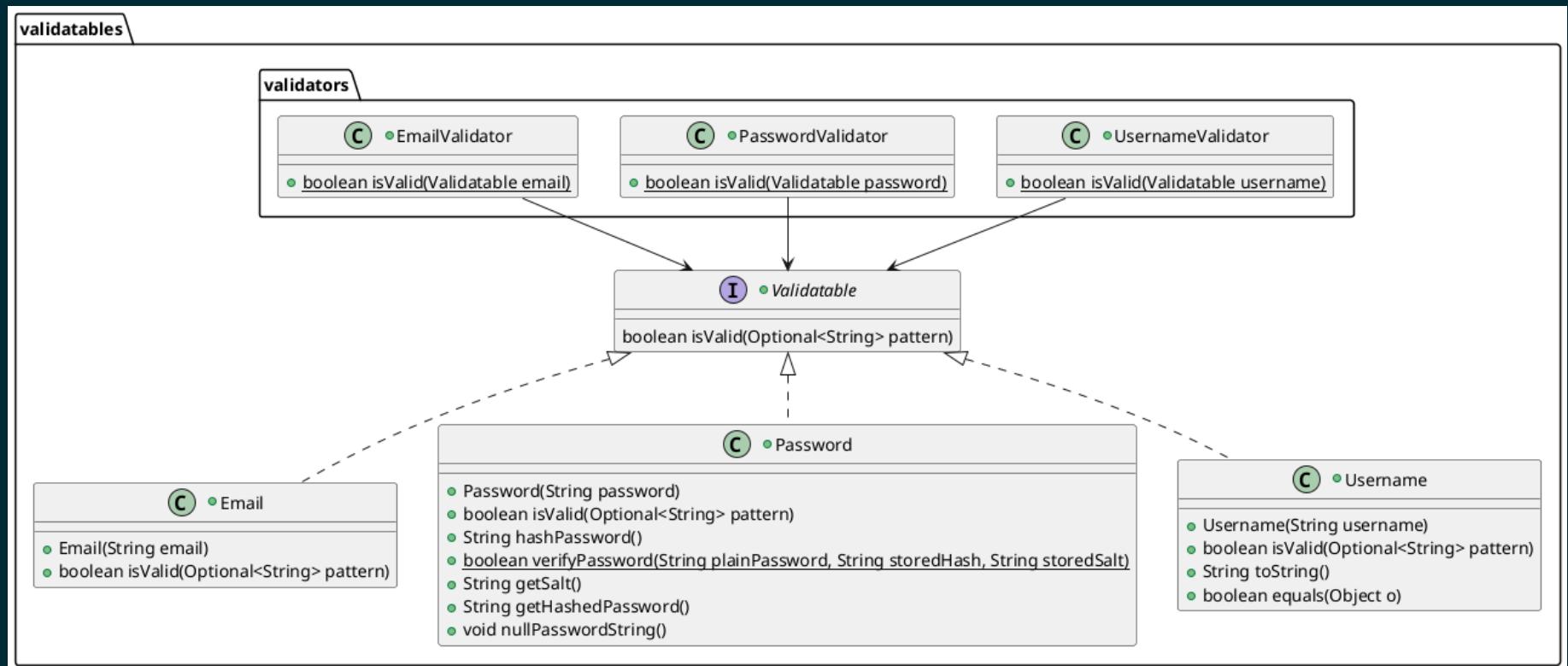


LÖSUNG: EINFÜHREN EINES GEMEINSAMEN INTERFACES

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

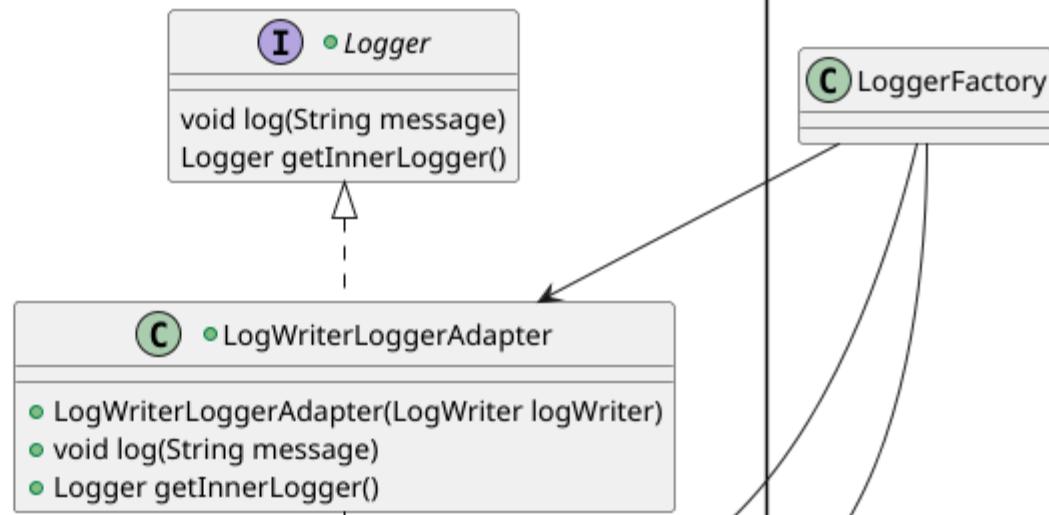
ENTWURFSMUSTER #1: ADAPTER

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

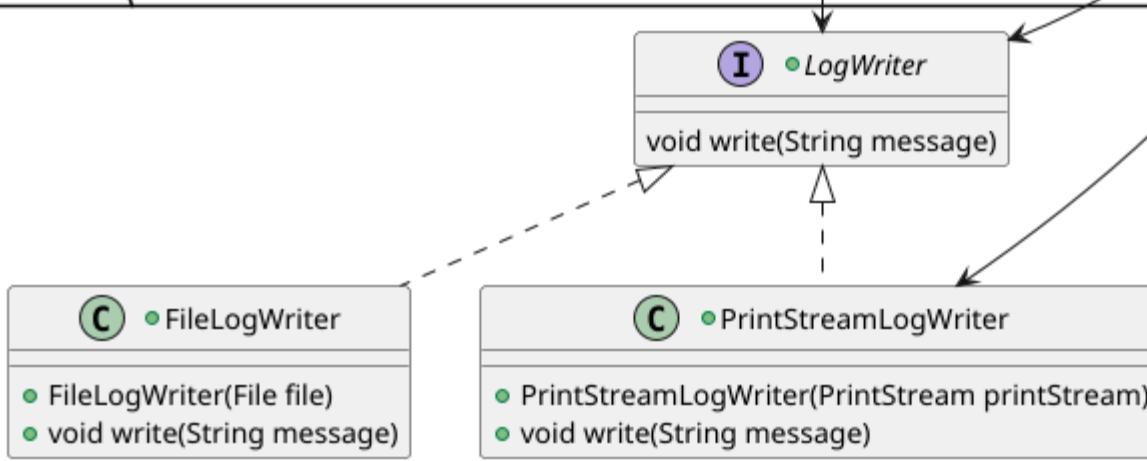
LOGWRITERLOGGERADAPTER

UML (VEREINFACHT)

logging



logwriter



- 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.

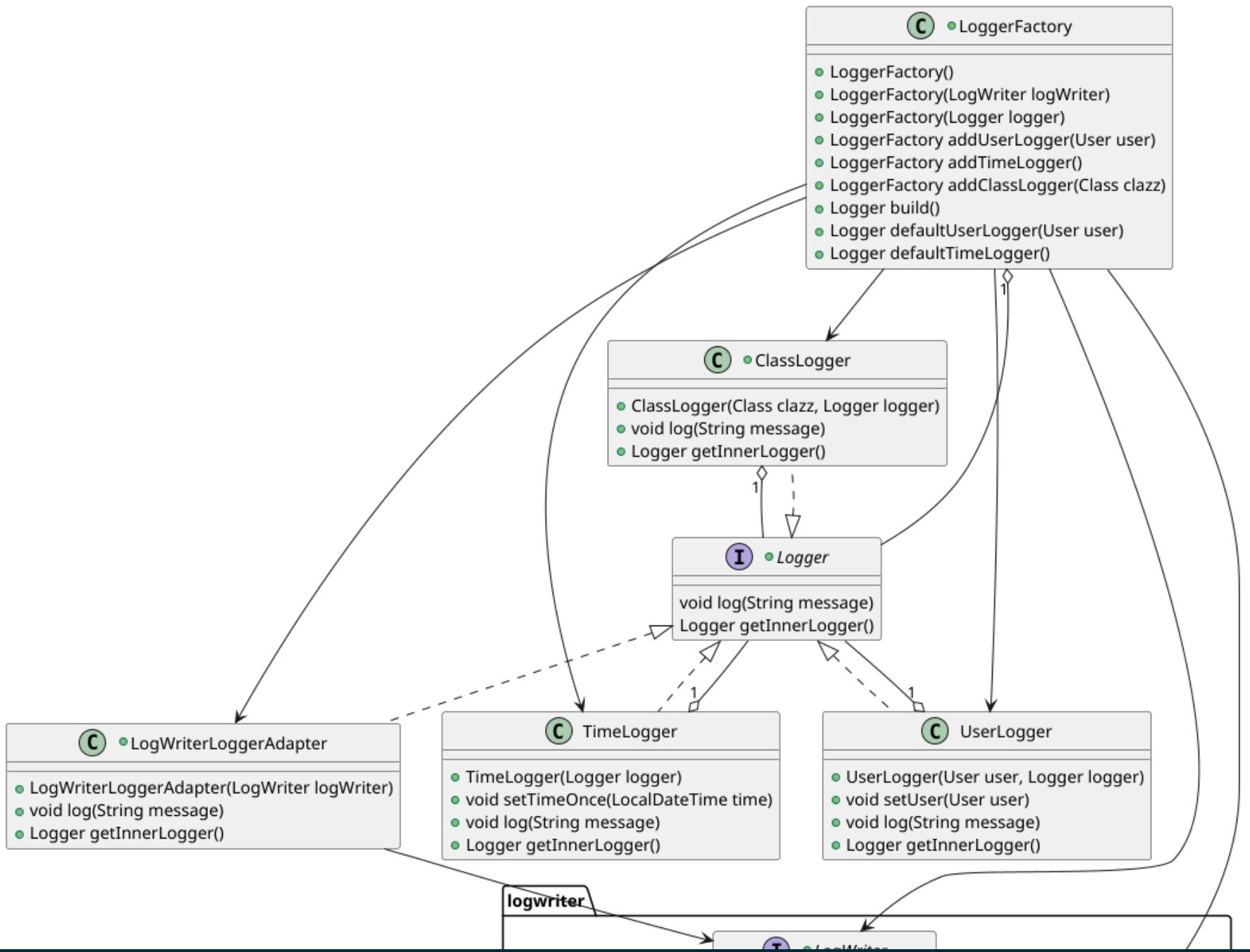
- Durch den Adapter ist es möglich, dass Code, der mit der Logger-Schnittstelle arbeitet, einen LogWriter zu übergeben, damit z.b. die gesammelten Logs in einer Datei gespeichert werden können.

ENTWURFSMUSTER #2: DECORATOR

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

DAS LOGGER-SYSTEM

logging



- 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();
```

VIELEN DANK FÜR DIE
AUFMERKSAMKEIT

