

Programmmentwurf

StoreMe

Kurs: Tinf19B5

Name: Schirmeier, Nico

Matrikelnummer: 1825563

Name: Steinbrunn, Lukas

Matrikelnummer: 5713633

Abgabedatum: 15.08.2022

Allgemeine Anmerkungen:

- *es darf nicht auf andere Kapitel als alleiniger Leistungsnachweis verwiesen werden (z.B. in der Form "XY wurde schon in Kapitel 2 behandelt, daher hier keine Ausführung")*
- *alles muss in UTF-8 codiert sein (Text und Code)*
- *sollten mündliche Aussagen den schriftlichen Aufgaben widersprechen, gelten die schriftlichen Aufgaben (ggf. an Anpassung der schriftlichen Aufgaben erinnern!)*
- *alles muss ins Repository (Code, Ausarbeitung und alles was damit zusammenhängt)*
- *die Beispiele sollten wenn möglich vom aktuellen Stand genommen werden*
 - *finden sich dort keine entsprechenden Beispiele, dürfen auch ältere Commits unter Verweis auf den Commit verwendet werden*
 - *Ausnahme: beim Kapitel "Refactoring" darf von vorne herein aus allen Ständen frei gewählt werden (mit Verweis auf den entsprechenden Commit)*
- *falls verlangte Negativ-Beispiele nicht vorhanden sind, müssen entsprechend mehr Positiv-Beispiele gebracht werden*
 - *Achtung: werden im Code entsprechende Negativ-Beispiele gefunden, gibt es keine Punkte für die zusätzlichen Positiv-Beispiele*
 - *Beispiele*
 - *"Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt." (2P)*
 - *Antwort: Es gibt keine Klasse, die SRP verletzt, daher hier 2 Klassen, die SRP einhalten: [Klasse 1], [Klasse 2]*
 - *Bewertung: falls im Code tatsächlich keine Klasse das SRP verletzt: 2P ODER falls im Code mind. eine Klasse SRP verletzt: 1P*
- *verlangte Positiv-Beispiele müssen gebracht werden*
- *Code-Beispiel = Code in das Dokument kopieren*
- *Gesamt-Punktzahl: 60P*
 - *zum Bestehen (mit 4,0) werden 30P benötigt*

[DISCLAIMER]

Viele der Commits wurden über CodeWithMe im Pair-Programming durchgeführt. Deshalb gibt es eine Diskrepanz zwischen der Anzahl an Commits beider Teampartner!

Kapitel 1: Einführung (4P)

Übersicht über die Applikation (1P)

Die Applikation dient der Planung und Inventur von Haushalten. Verbrauchsgüter und andere Gegenstände können nach Belieben in Form von Items hinzugefügt und verwaltet werden. Die Items werden in sogenannten Containern organisiert. Beim Erstellen eines Items kann ein sogenanntes Item Template erzeugt werden. Dies erleichtert das erneute Hinzufügen eines bereits existierenden Items und ermöglicht das Erstellen von Rezepten. Rezepte können eine beliebige Anzahl von Item Templates enthalten. Die Templates in Rezepten enthalten zusätzlich eine Mengen-Information. Recommended Recipes zeigt alle kochbaren Rezepte an. Darüber hinaus verfügt die Applikation über ein Tagging System, mit dem Items in Einkaufslisten und andere beliebige Tags gefiltert werden können.

Items verfügen außerdem über ein Ablauf- und Verbrauchsdatum. Wird dieses überschritten, werden die entsprechenden Items in der Übersicht angezeigt. Die Elemente (außer der Templates) können jederzeit aufgerufen und geändert/gelöscht werden. Alle Aktionen werden bei Bestätigung direkt in den entsprechenden Json-Dateien gespeichert.

Wie startet man die Applikation? (1P)

Die Applikation kann über die jar Releases in GitHub [heruntergeladen](https://github.com/NicoSchirmeier/StoreMe/releases/tag/JarReleases) werden.

(<https://github.com/NicoSchirmeier/StoreMe/releases/tag/JarReleases>)

Die StoreMe.jar kann über die Konsole ausgeführt werden.

Das Projekt benötigt Java 18 oder neuer.

Für Beispieldaten kann der data Ordner aus der entsprechenden Zip-Datei entpackt werden.

Dieser muss im selben Ordner wie die StroMe.jar Datei liegen.

Alternativ kann das Projekt über die IDE ausgeführt werden. Dazu müssen die vier Jackson Libraries/Module in den Project Settings hinzugefügt werden. Diese befinden sich im Projektordner unter plugins. Alles andere wird typischerweise durch IntelliJ automatisch importiert oder als Import angeboten. Danach sollte die Main Methode ausgeführt werden können.

Technischer Überblick (2P)

- OpenJDK 18
 - Dient als Basissprache und eignete sich besonders aufgrund von Vorerfahrung im Projektteam.
- Jackson für JSON Handling
 - Wurde als alternative zu Gson gewählt, da Jackson ein Modul zur (De-) Serialisierung der Java Klasse LocalDate beinhaltet (Gson besitzt diese funktion nativ nicht). Es wurden keine weiteren Alternativen betrachtet.
- Jupiter für JUnit Testing
 - Wurde ebenfalls auf Grund von Vorerfahrung eingesetzt. Jupiter besitzt außerdem eine native Unterstützung durch IntelliJ.

Kapitel 2: Clean Architecture (8P)

Was ist Clean Architecture? (1P)

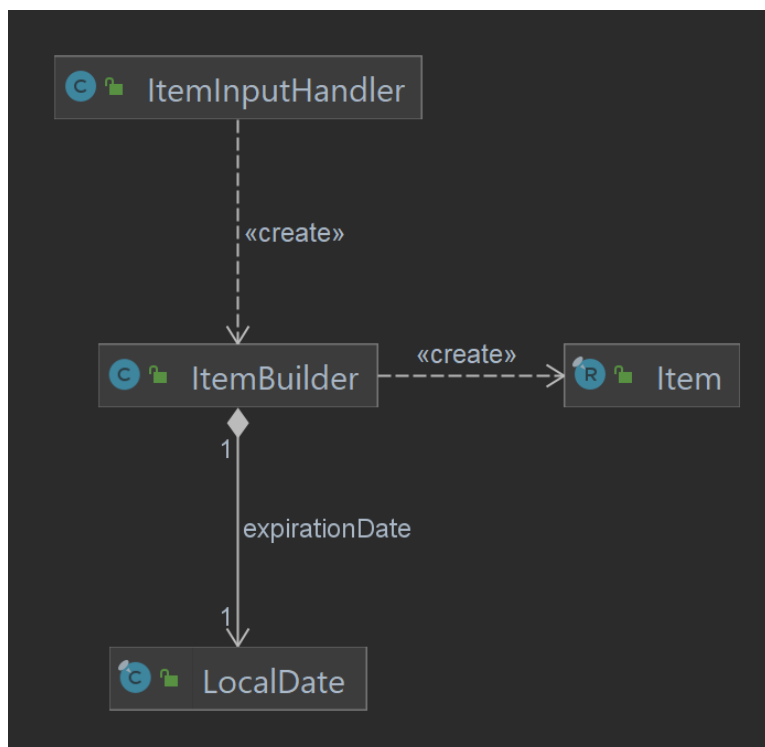
Clean Architecture beschreibt eine Praxis zur Gestaltung / Positionierung von Systemteilen. Sie sind in zwiebelringartigen Schichten aufgeteilt. Die inneren Schichten bilden dabei den Programm-Kern, welcher selten geändert wird. Nach außen hin kommen immer eindeutigere Programmteile, welche bei einer Änderung der Systemumgebung sehr wahrscheinlich angepasst werden müssen. Abhängigkeiten finden immer nur von außen nach innen statt, sodass eine Änderung der eindeutigen Systemteile keine Auswirkung auf den Programmkern hat.

Analyse der Dependency Rule (2P)

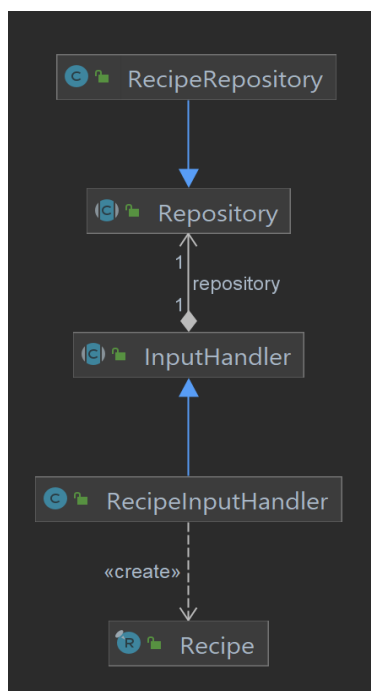
Positiv-Beispiel: Dependency Rule

Das Item gehört zur DomainCode-Schicht. Es stellt eine Entity dar und hält die Informationen der lagerbaren Items. Da es über ein Erbauer-Entwurfsmuster generiert wird, liegt die ItemBuilder-Klasse auf derselben Ebene und steht in Verbindung mit anderen Schichten:

1. Schicht Application Code: Der ItemInputHandler
2. Schicht Abstraction Code: Java-eigene Library LocalDate



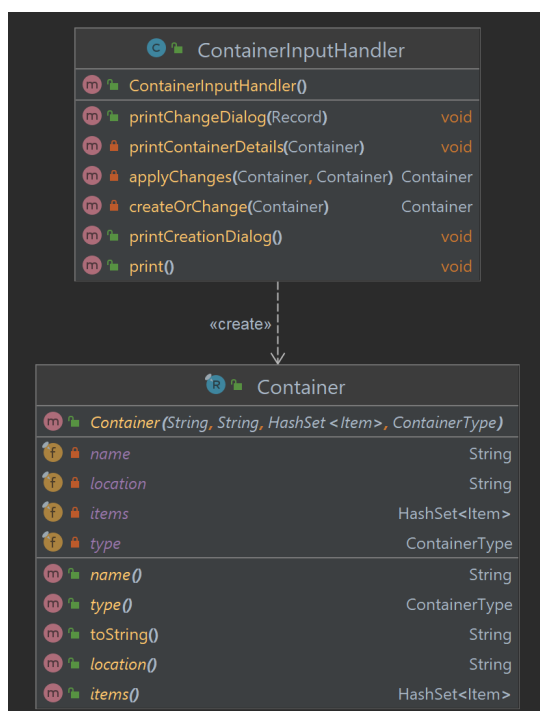
Negativ-Beispiel: Dependency Rule



Der RecipeInputHandler verarbeitet die verschiedenen Aktionen der Benutzereingaben. Dazu werden die CRUD-Operationen aus den Repository-Klassen benötigt. Dadurch entsteht aber eine Abhängigkeit in diesen Klassen, welche sich in einer weiter äußeren Schicht, der Adapter-Schicht, befindet. Somit ist die Regel, dass nur äußere Schichten von inneren Schichten abhängig sein dürfen, verletzt. Die Abhängigkeit dürfte nur anders herum bestehen. Durch Anwendung des 'Dependency Inversion Principles' könnte die Abhängigkeit aufgelöst werden.

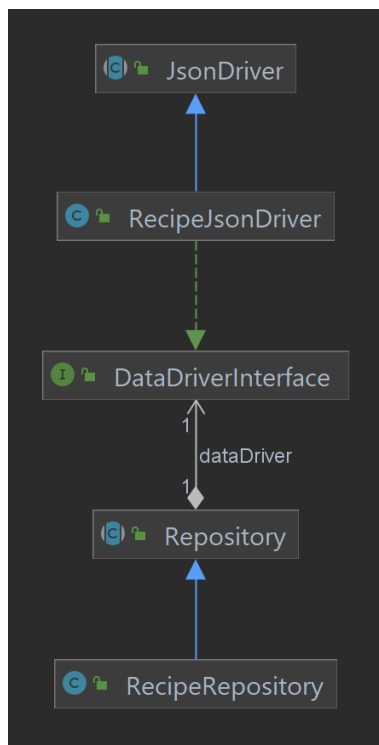
Analyse der Schichten (5P)

Schicht: Application Code



Die Klasse ContainerInputHandler aus der ApplicationCode-Schicht verarbeitet das Erstellen, Verändern und Auslesen von Containern vorangegangener Benutzeraktionen. Dazu nutzt sie die Klasse Container aus der DomainCode-Schicht. Somit steuert die Handler-Klasse die Daten von und zu den Entities der DomainCode-Klassen.

Schicht: Adapters

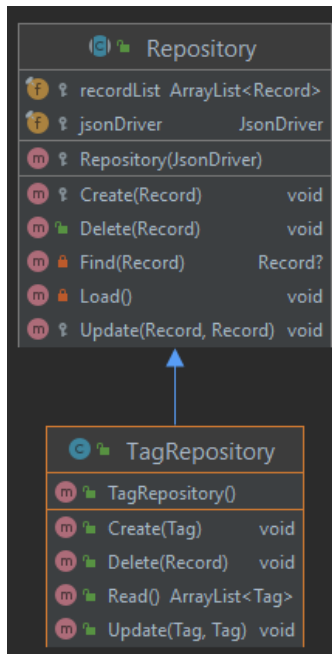


Mit dem **DataDriverInterface** wird das 'Dependency Inversion Principle' angewandt, um die Dependency Rule der Clean Architecture Schichten einzuhalten. Die **Repository** Klassen dienen als Gateway und liefern die CRUD-Operationen, weswegen sie in der Adapter-Schicht liegen. Sie vermitteln zwischen den UseCases aus der **ApplicationCode**-Schicht und der **JsonDriver**-Klassen der **Plugin**-Schicht für das Schreiben in die **JSON**-Files. Mithilfe des **DataDriverInterfaces** sind sie nicht direkt von den **JsonDriver**-Klassen abhängig.

Kapitel 3: SOLID (8P)

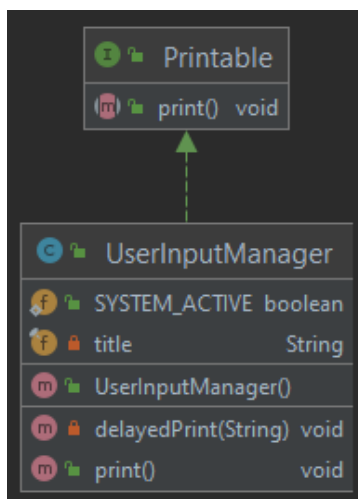
Analyse SRP (3P)

Positiv-Beispiel



Das TagRepository dient ausschließlich dazu, die CRUD Operationen für Tags zu implementieren. Das Tag-Repository hat nur eine spezielle Aufgabe, weshalb es SRP erfüllt.

Negativ-Beispiel

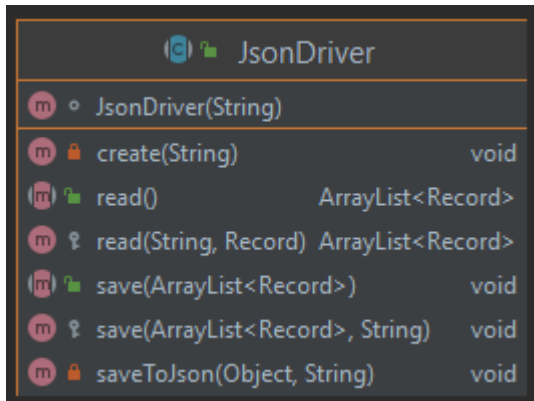


Der UserInputManager dient als Einstiegspunkt für die Texteingaben des Benutzers. Er implementiert neben den Auswahlmöglichkeiten für die einzelnen Submenüs und Informationen auch eine eigene Funktion zur Ausgabe von Text im Hauptmenü (`delayedPrint()`).

Die `delayedPrint` Methode sollte in eine der dedizierten `ConsoleUtil` Klassen ausgelagert werden.

Analyse OCP (3P)

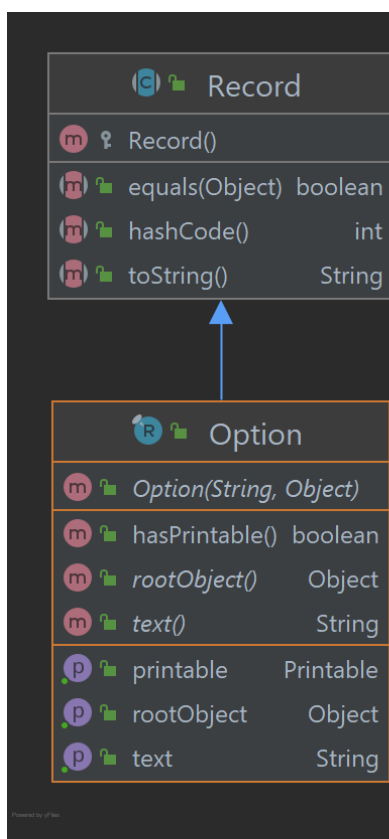
Positiv-Beispiel



Die `JsonDriver`-Klasse abstrahiert das Speichern verschiedener Datenobjekte. Durch die Implementierung von Subklassen kann ein neuer Speicherort implementiert und ein neuer Datentyp direkt referenziert werden. So wie z. B. beim `ContainerRepository`.

Das hierdurch gelöste Problem war es, eine geeignete Struktur zu finden, um die entsprechenden Daten getrennt in eigene Dateien zu schreiben.

Negativ-Beispiel



Die `Option` Klasse speichert die nötigen Informationen zum Ausführen von Aktionen innerhalb der Handler Klassen. Das `Root Object` referenziert entweder ein `Printable` oder eine `Action` Instanz. Dies führt dazu, dass für jede Aktion, die als `Option` für den Nutzer angezeigt wird, eine zusätzliche `If-Abfrage` durchgeführt werden muss, um auf die entsprechende Methode oder das entsprechende `Printable` zu verweisen.

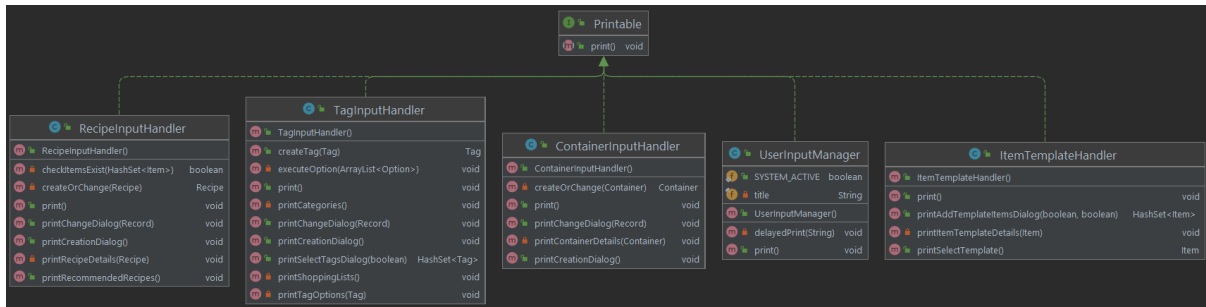
```
Option option =
ConsoleSelectionUtils.displayOptions(options);

if(option.getRootObject() instanceof Container) {
    printContainerDetails((Container)
option.getRootObject());
} else if (option.getRootObject() instanceof
Action action) {
    if(action.equals(Action.CREATE)) {
        printCreationDialog();
    }
}
```

Zur Lösung dieses Problems könnte anstatt eines `RootObjects` ein `Command Pattern` eingeführt werden, indem jede mögliche `Action` inklusive der `print` Methode der `Printables` eine eigene Implementierung eines `Commands` erhält. So könnten alle möglichen Aktionen durch bsp. `command.execute()` abgedeckt werden.

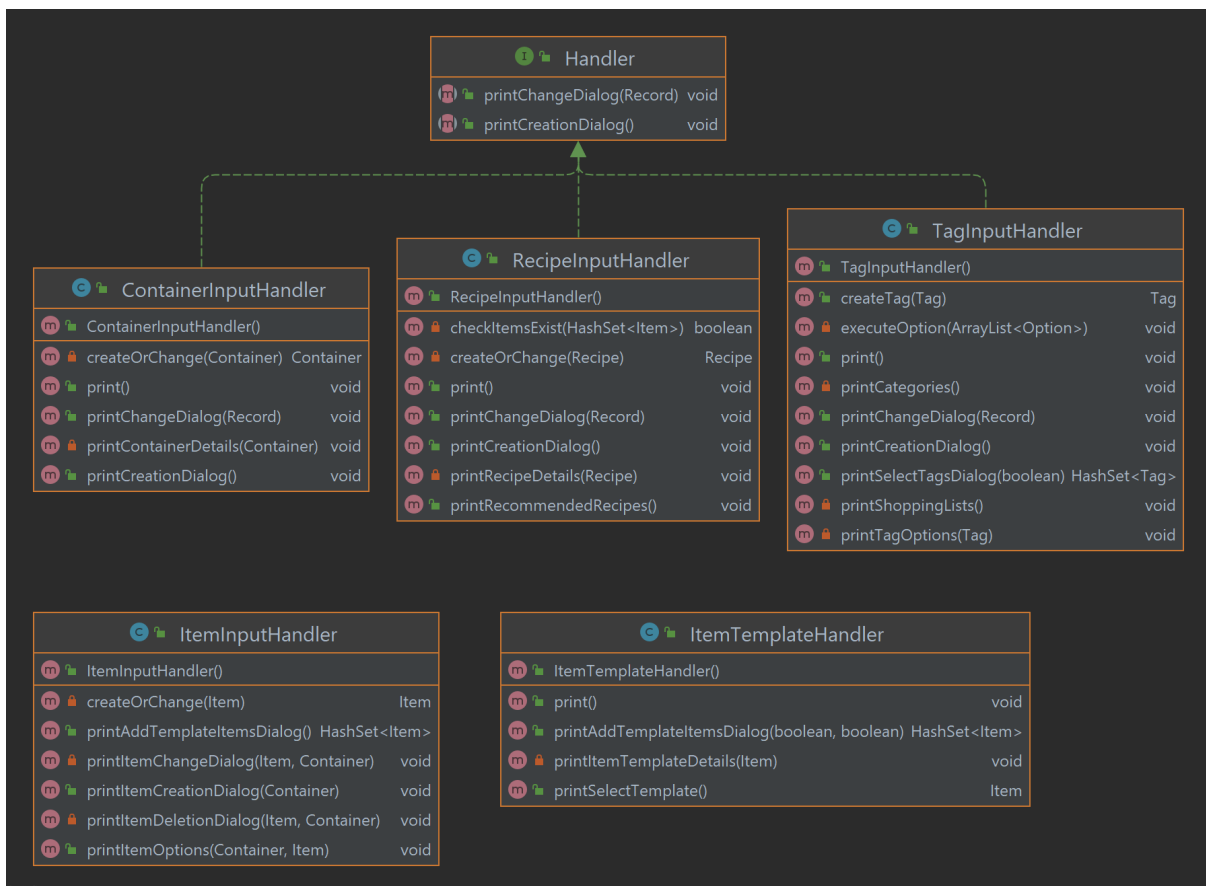
Analyse [ISP] (2P)

Positiv-Beispiel



Das Printable Interface abstrahiert die print Funktion. Jede der implementierenden Klassen bekommt durch das Interface so genau eine zusätzliche Funktion.

Negativ-Beispiel



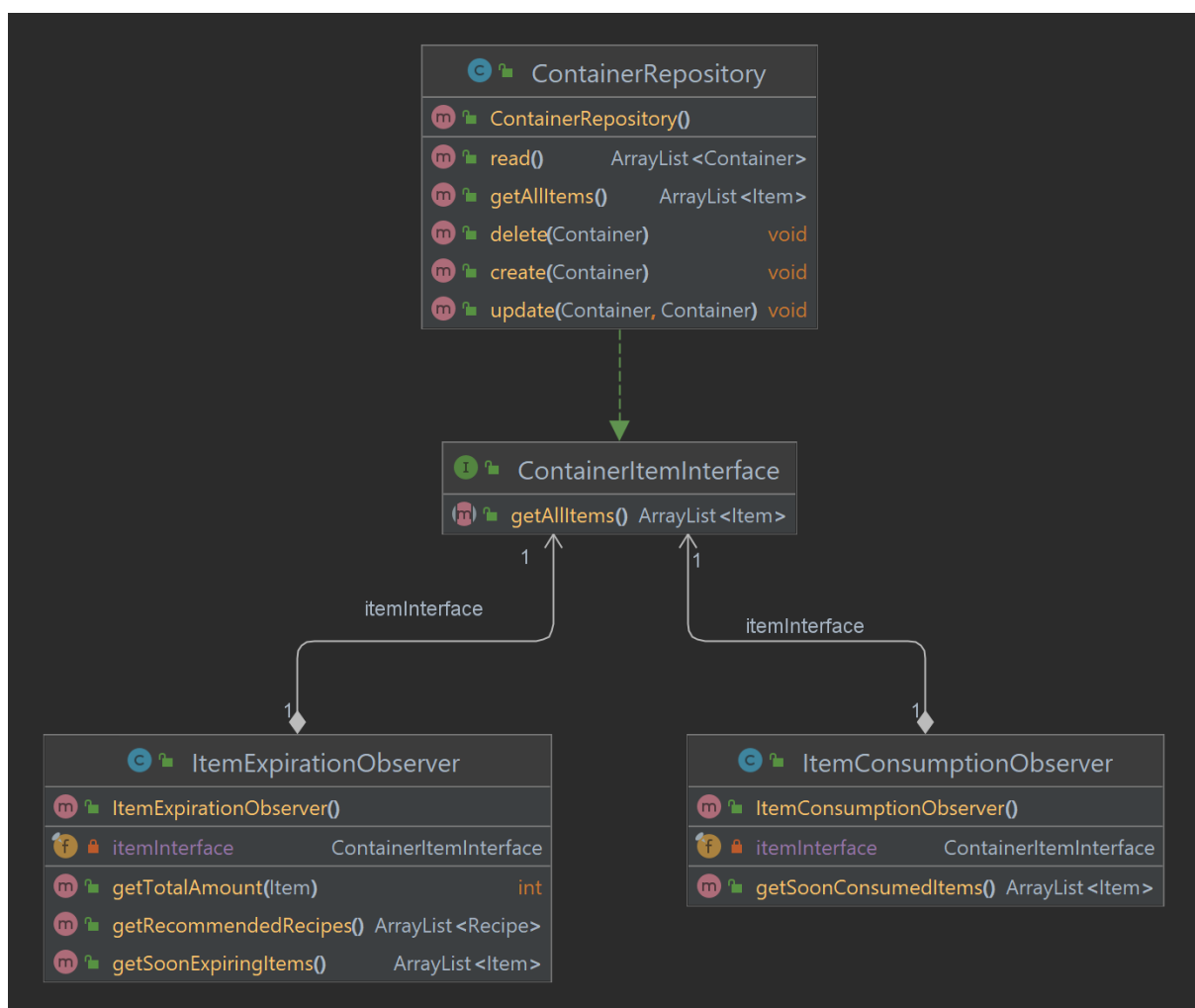
Sowohl ItemInputHandler, als auch ItemTemplateHandler können das Handler Interface nicht implementieren, da sie andere Voraussetzungen für die entsprechenden Methoden besitzen. Hier sollten zwei Interfaces bsp. Creatable und Changeable eingeführt werden.

Kapitel 4: Weitere Prinzipien (8P)

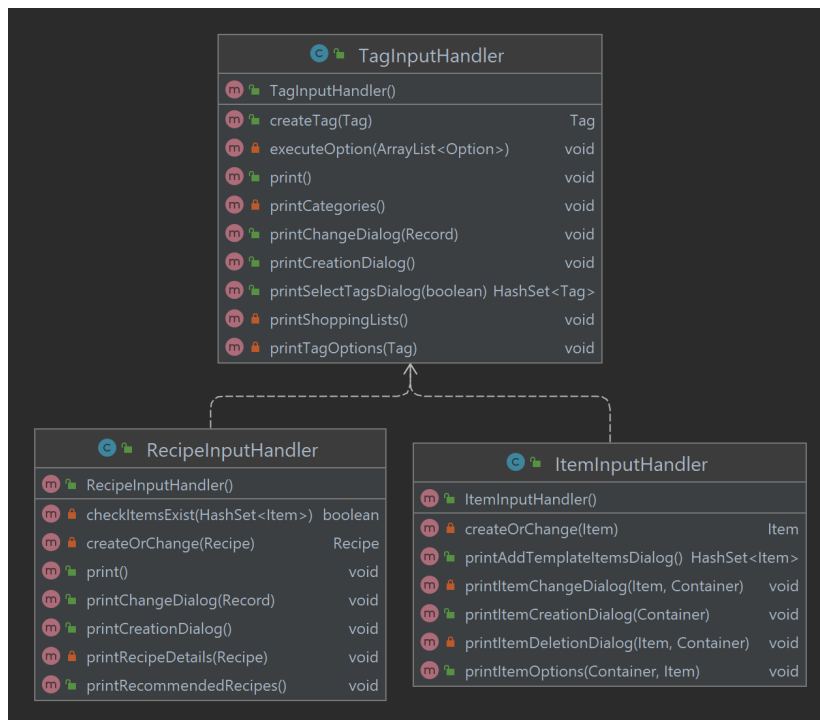
Analyse GRASP: Geringe Kopplung (4P)

Positiv-Beispiel

Die beiden Klassen haben die Aufgabe, Items zu laden, die bald ablaufen (ItemExpirationObserver) bzw. die bald nach Plan aufgebraucht sind (ItemConsumptionObserver). Für den benötigten Funktionsumfang der Observer an das ContainerRepository, dem Abrufen aller Items, wird ein Interface bereitgestellt. Dieses wird durch das ContainerRepository implementiert. Eine Abhängigkeit der Observer zu dem ContainerRepository wird somit aus dem Weg gegangen.

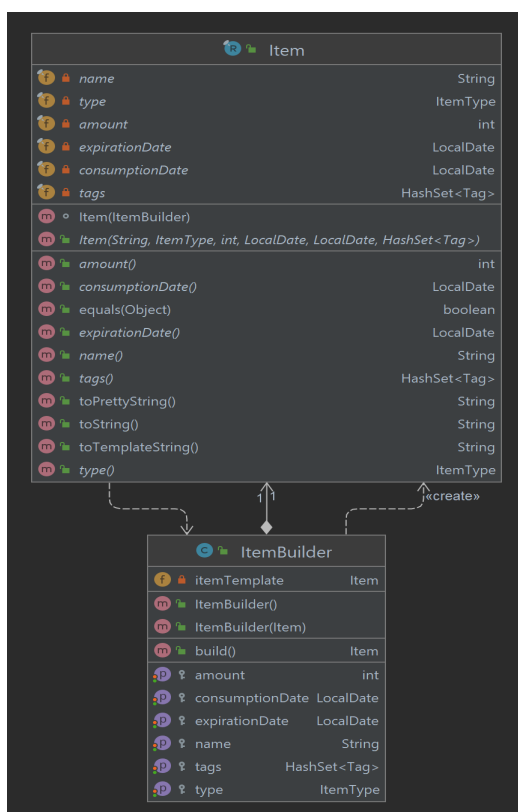


Negativ-Beispiel



Die Klassen RecipeInputHandler und ItemInputHandler nutzen nur die Funktion printSelectTagsDialog. Haben aber Vollzugriff auf die Klasse TagInputHandler. Hier könnte ein zusätzliches Interface eingeführt werden, durch das nur relevante Funktionen aufrufbar wären. So könnte zum Beispiel der ItemHandler nicht auf die print Methode des TagInputHandlers zugreifen.

Analyse GRASP: Hohe Kohäsion (2P)



Die ItemBuilder Klasse ist stark abhängig von der Item Klasse. Ihre Funktionalität ist ebenfalls abhängig von allen Attributen, da sie benötigt werden, um ein Item aus den Informationen zu erzeugen.

DRY (2P)

Commit: d392ca65b30038074b44911e34d0dc9bee07b3ff

Vorher:

Methode printItemCreationDialog()

```
public void printItemCreationDialog(Container container) {
    ArrayList<Option> options = new ArrayList<>();

    System.out.println("Enter Name:");
    String name = ConsoleUtils.readString();
    System.out.println("Select Type:");
    ItemType itemType = (ItemType)
ConsoleUtils.printTypeSelection(ItemType.values());
    System.out.println("Enter Amount:");
    int amount = ConsoleUtils.getAmount(1, 0);
    System.out.println("Enter Expiration Date: (" +
ConsoleUtils.dateFormat + ")");
    LocalDate expirationDate = ConsoleUtils.getDate();
    System.out.println("Enter Consumption Date: (" +
ConsoleUtils.dateFormat + ")");
    LocalDate consumptionDate = ConsoleUtils.getDate();
    System.out.println("Select Tags:");
    HashSet<Tag> tags = tagInputHandler.printSelectTagsDialog();

    Item item = new Item(name, itemType, amount, expirationDate,
consumptionDate, tags);
    System.out.println(item);
    boolean confirmed = ConsoleUtils.printConfirmationDialog("Create
Item");

    if (confirmed) {
        container.items().add(item);
        repository.Update(container, container);
    }
}
```

Methode printItemChangeDialog()

```
private void printItemChangeDialog(Item item, Container container) {
    ArrayList<Option> options = new ArrayList<>();

    System.out.println("Enter Name: (Write \"!\" to skip)");
    String name = ConsoleUtils.readString();
    if (name.equals("!")) name = item.name();
    System.out.println("Select Type:");
    ItemType itemType = (ItemType)
ConsoleUtils.printTypeSelection(ItemType.values());
    System.out.println("Enter Amount:");
    int amount = ConsoleUtils.getAmount(1, 0);

    System.out.println("Enter Expiration Date: (" +
ConsoleUtils.dateFormat + ")");
    LocalDate expirationDate = ConsoleUtils.getDate();
    System.out.println("Enter Consumption Date: (" +
ConsoleUtils.dateFormat + ")");
    LocalDate consumptionDate = ConsoleUtils.getDate();

    System.out.println("Select Tags:");
    HashSet<Tag> tags = tagInputHandler.printSelectTagsDialog();

    Item changedItem = new Item(name, itemType, amount, expirationDate,
consumptionDate, tags);

    System.out.println(item);
    System.out.println("->");
    System.out.println(changedItem);
    boolean confirmed = ConsoleUtils.printConfirmationDialog("Change
Item");

    if (confirmed) {
        container.items().remove(item);
        container.items().add(changedItem);
        repository.Update(container, container);
    }
}
```

Nacher:

Aufgrund von sehr ähnlicher Funktionalität wurden die beiden Methoden zusammengefasst. Hierdurch konnte durch Extrahierung der Methode die Lesbarkeit und Verständlichkeit des Codes gesteigert werden, was ebenfalls die Fehleranfälligkeit reduziert (Änderungen müssen nur an einer Stelle durchgeführt werden).

```
private Item createOrChange(Item toChange) {
    if(toChange == null) {
        System.out.println("- Create Item -");
    } else {
        System.out.println("- Change Item -");
        System.out.println(toChange + " (Enter \"!\" to skip)");
    }

    System.out.println("Enter Name:");
    String name = ConsoleReadingUtils.readString();
    System.out.println("Select Type:");
    ItemType itemType = (ItemType)
    ConsoleSelectionUtils.printTypeSelection(ItemType.values());
    System.out.println("Enter Amount:");
    int amount = ConsoleReadingUtils.getAmount(1, 0);
    System.out.println("Enter Expiration Date: (" +
    ConsoleUtilConfiguration.DATE_FORMAT + ")");
    LocalDate expirationDate = ConsoleReadingUtils.getDate();
    System.out.println("Enter Consumption Date: (" +
    ConsoleUtilConfiguration.DATE_FORMAT + ")");
    LocalDate consumptionDate = ConsoleReadingUtils.getDate();
    System.out.println("Select Tags:");
    HashSet<Tag> tags =
    DataManager.TAG_INPUT_HANDLER.printSelectTagsDialog();

    if(toChange != null) {
        if(name.equals("!")) name = toChange.name();
    }

    return new Item(name, itemType, amount, expirationDate,
    consumptionDate, tags);
}
```

Kapitel 5: Unit Tests (8P)

10 Unit Tests (2P)

Unit Test	Beschreibung
Item#equals(Item item)	Testen des Vergleichs zwischen zwei Items mit aus dem ItemBuilder erzeugten Items
ItemExpirationObserver #getDaysBetween()	Testet ob die Differenz zweier Daten korrekt berechnet wird
JsonDriver#save() #read()	Testet das Speichern und Auslesen eines TestRecord's durch Jackson
Repository#create()	Testet das Erstellen eines TestRecord's über ein TestRepository
Repository#update()	Testet das Verändern eines TestRecord's über ein TestRepository
Repository#delete()	Testet das Löschen eines TestRecord's über ein TestRepository
ConsoleReadingUtils #readString()	Testet das korrekte Auslesen eines Strings über eine künstliche Konsoleneingabe
ConsoleReadingUtils #readAmount()	Testet das korrekte Auslesen eines Integers über eine künstliche Konsoleneingabe
Option #getRootObject()	Testet das Auslesen eines root Objektes aus der Option klasse
Option #hasPrintable()	Testet die korrekte Bestimmung, ob ein Option objekt ein root object des Typs Printable besitzt.

ATRIP: Automatic (1P)

Die Tests können einfach über die IDE durchgeführt werden. Dort können einzelne oder mehrere gewählte Tests per Knopfdruck durchgeführt werden (Alle Tests über den Ordner Tests ausführbar). Sie erfordern keine Eingabe von außen (Dafür, wurden z. B. künstliche Konsoleneingaben realisiert). Die IDE bestimmt selbstständig, ob ein Test fehlschlägt oder besteht.

ATRIP: Thorough (1P)

Die als kritisch eingestuften Stellen waren:

- Grundlegende Benutzereingaben (Text und Integer)
- Speichern und Lesen der Daten (Auch wegen Json Bibliothek)
- Vergleichsoperation zwischen Item und Item Template
- Bestimmung von abgelaufenen und mögl. Verbrauchten Items durch Abgleich von Daten

Diese wurden durch die Tests abgedeckt.

Es wurden bei JsonDriver und Repository ausgewählte Sonderzeichen getestet. getDaysBetween wurde durch ein Standard-Szenario getestet, andere Eingaben können zu unerwartetem Verhalten führen. Dies liegt jedoch im Ermessen des Benutzers. Bei der equals Funktion des Items wurden bestimmte Attribute über den ItemBuilder verändert. Es wurden zwei Vergleiche getestet, die auf den Vergleichswerten der equals Funktion basieren. Für die Tests der Konsolen-Eingaben wurden korrekte Eingaben getestet, da bei inkorrekt eingabe kein Schaden für den Endnutzer entsteht (Kein Datenverlust, höchstens temporäre Eingaben).

ATRIP: Professional (1P)

Für die Tests wurden speziell angelegte Typen eingeführt, um eine klare Trennung der Test- und Anwendungsdaten zu ermöglichen.

Fakes und Mocks (3P)

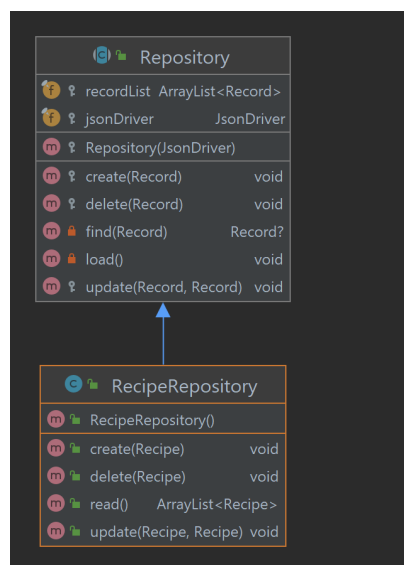
Es wurde ein TestRecord zum Testen der JsonDriver und Repository Klassen angelegt. Dieser wurde entsprechend in den Tests als Mock verwendet. Ebenfalls wurde ein TestJsonDriver und ein TestRepository implementiert, welche ebenfalls nur für die Tests instanziiert werden.

Kapitel 6: Domain Driven Design (8P)

Ubiquitous Language (2P)

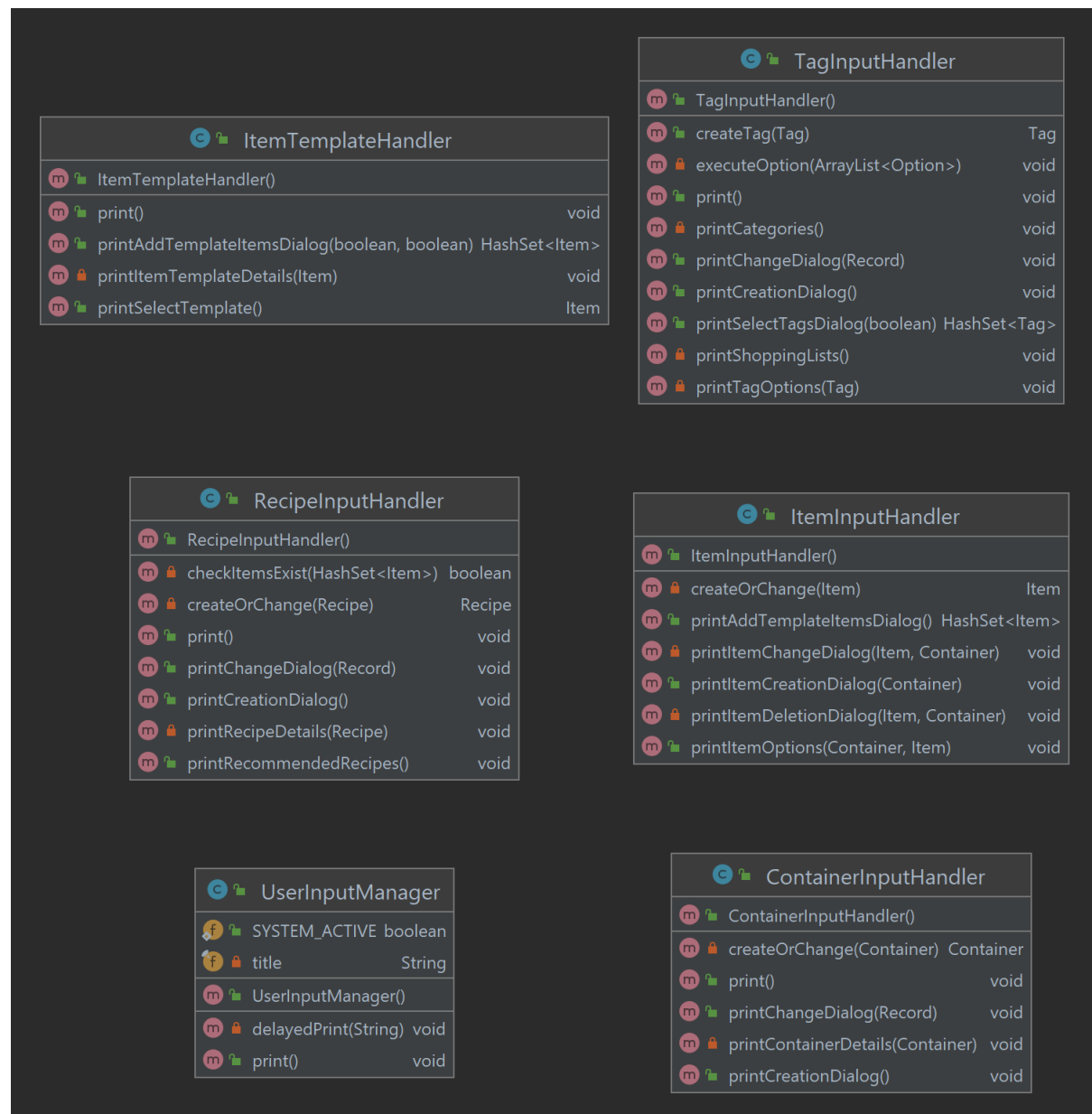
Bezeichnung	Bedeutung	Begründung
Item	Jeglicher Gegenstand, der im Rahmen der Applikation abgelegt und gespeichert werden kann der kein Container ist	Item lässt sich direkt als Gegenstand/Artikel übersetzen und wird sowohl in Code als auch in der Anwendung als solches bezeichnet.
Container	Elemente, welche als Ablageort von Items definiert werden können	Container dienen als Speicher/Lagerplatz. So können diese auch in der Anwendung aufgefasst werden.
Recipe	Ein Rezept als Anleitung zur Durchführung beliebiger Aktionen	Rezepte sind im normalen Sprachgebrauch vorhanden. Der Begriff sollte von der Zielgruppe ohne Weiteres verstanden werden.
Item Template	Ein Template ist eine Vorlage, aus der etwas erstellt werden kann.	Templates sind auch in anderen Anwendungen zu finden. Sie stellen ein bekanntes Muster zur Anwendung dar. (Office Programme, etc.)

Repositories (1,5P)



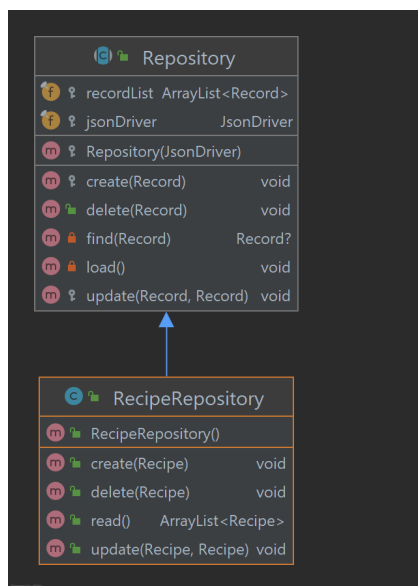
Die Repositories sind in der Anwendung die Subklassen der Repository Klasse. Jedes Repository ist für einen anderen Datentyp der Anwendung zuständig. Die Repositories dienen als Adapter zwischen der Datenbank (JsonDriver Klassen) und der Anwendungsschicht (InputHandler Klassen). Unter anderem kann hierdurch die Datenbank ausgetauscht werden, ohne die Anwendungsschicht anpassen zu müssen.

Aggregates (1,5P)



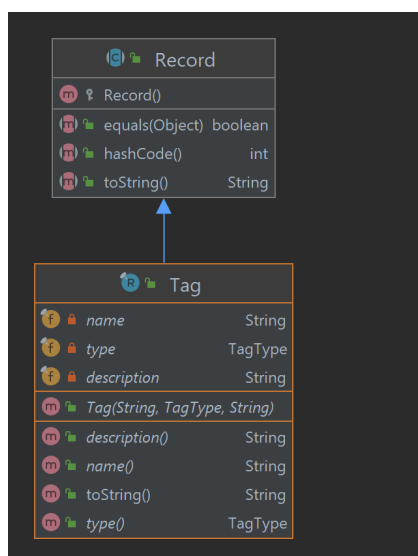
Die StoreMe Anwendung ist in verschiedene Bereiche unterteilt, welche jeweils auf logisch abgegrenzte Teilbereiche der Anwendung zugreifen. Die InputHandler implementieren den Zugriff des Benutzers auf die verschiedenen Bereiche der Anwendung. Der ContainerHandler dient zur Interaktion mit den verschiedenen Containern und den darin befindlichen Items. Der ItemHandler dient zur Interaktion mit einzelnen Items, wenn eines ausgewählt wurde. Über den RecipeHandler kann mit verschiedenen Rezepten interagiert werden. Über den ItemTemplateHandler können Templates eingesehen und gelöscht werden und über den TagHandler können alle Tags verwaltet werden. Durch diese Unterteilung kann eine logische Abtrennung sowohl im Code als auch für den Benutzer realisiert werden. So kann die Applikation mit den verschiedenen Handler gesamt all Aggregate zusammengefasst werden.

Entities (1,5P)



Die Repositorys sind einzigartig in der Domäne. Jedes der Repositorys enthält eine Liste an Value Objects, die sich während der Laufzeit ändern kann. Jedes der Repositorys ist während der Laufzeit einzigartig und somit auch eindeutig identifizierbar.

Value Objects (1,5P)



Alle Datenobjekte, welche innerhalb der Anwendung zum Einsatz kommen sind Records und damit auch immutable. Die Objekte der Datenklasse haben keine eindeutige Identität und sind durch Ihre Eigenschaften voneinander abgegrenzt. So z. B. Objekte der Item Klasse. Die VOs wurden eingesetzt, da die Objekte nicht identifizierbar sein müssen, um die gewünschte Funktionalität zu realisieren.

Kapitel 7: Refactoring (8P)

Code Smells (2P)

Code Smells: Duplicated Code (ItemTemplateHandler)

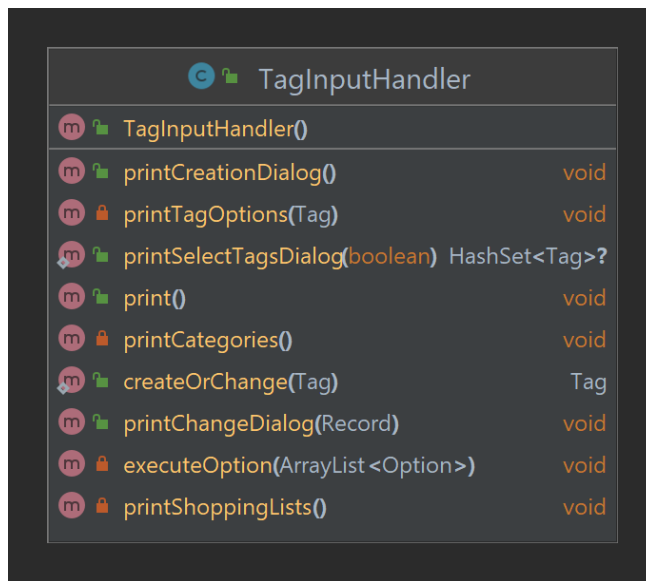
```
public void print() {
    System.out.println("- Item Templates -");
    ArrayList<Option> additionalOptions = new ArrayList<>();
    for (Item template : DataManager.ITEM_TEMPLATE_REPOSITORY.read()) {
        additionalOptions.add(new Option(template.toTemplateString(),
        template));
    }
    Option option = ConsoleSelectionUtils.displayActions(additionalOptions,
    Action.BACK);
    if (option.getRootObject() instanceof Item template) {
        printItemTemplateDetails(template);
    }
}
```

```
public Item printSelectTemplate() {
    System.out.println("- Item Templates -");
    ArrayList<Option> additionalOptions = new ArrayList<>();
    for (Item template : DataManager.ITEM_TEMPLATE_REPOSITORY.read()) {
        additionalOptions.add(new Option(template.toTemplateString(), template));
    }
    Option option = ConsoleSelectionUtils.displayActions(additionalOptions,
    Action.BACK);
    if (option.getRootObject() instanceof Item template) {
        return template;
    }
    return null;
}
```

Die beiden Ausgaben der Option's sind Duplikate. Diese wurden durch Refactoring aufgelöst. Hierzu wurde eine neue Methode eingeführt:

```
private Option printItemTemplateOptions() {
    System.out.println("- Item Templates -");
    ArrayList<Option> additionalOptions = new ArrayList<>();
    for (Item template : DataManager.ITEM_TEMPLATE_REPOSITORY.read()) {
        additionalOptions.add(new Option(template.toTemplateString(), template));
    }
    return ConsoleSelectionUtils.displayActions(additionalOptions, Action.BACK);
}
```

Code Smells: Large Class

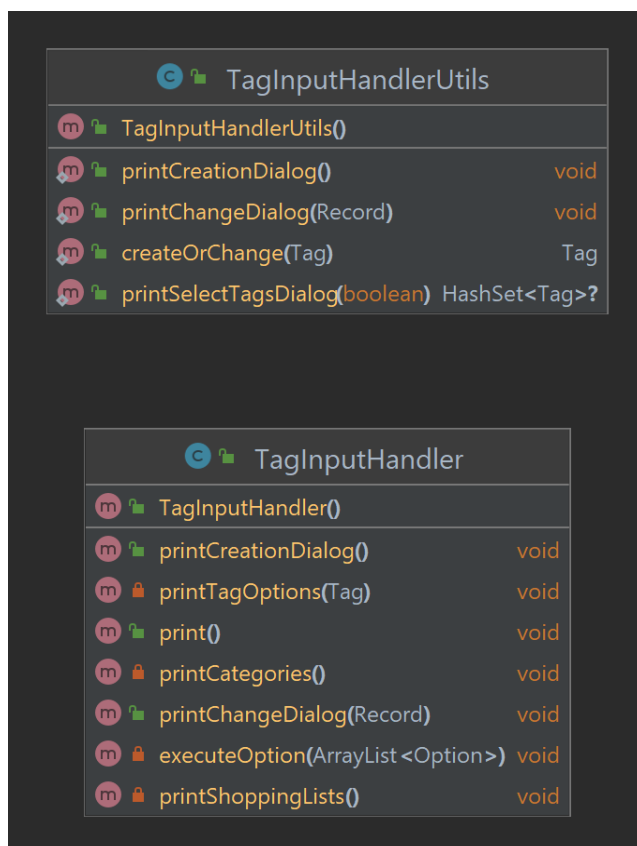


Zuvor:

> 100 Zeilen Code

Zu viele Methoden

Lösung: Auslagern



Danach:

Durch auslagern der `printCreationDialog`, `printChangeDialog`, `createOrChange`, `printSelectTagsDialog` Methoden konnte die Klasse auf unter 100 Zeilen code gebracht werden. Die `printCreationDialog` und `printChangeDialog` Methoden in `TagInputHandler` verweisen auf die statischen Methoden in der `Utils` Klasse.

Hier wurde statt des Codebeispiels eine Darstellung durch UML gewählt, da hier Methoden aus Klassen ausgelagert wurden, wobei der eigentliche Inhalt der Methoden nicht verändert wurde.

2 Refactorings (6P)

Extract Method: (ItemHandler)

Da die createOrChange Methode der ItemHandler Klasse sehr unübersichtlich war, wurden die einzelnen Schritte der Methode in einzelne Methoden extrahiert. So konnten die Übersichtlichkeit und Lesbarkeit deutlich gesteigert werden.

Zuvor:

```
private Item createOrChange(Item baseItem) {
    boolean isItemChange;

    ItemBuilder builder;
    if (baseItem == null) {
        isItemChange = false;
        builder = new ItemBuilder();

        System.out.println("- Create Item -");
    } else {
        isItemChange = true;
        builder = new ItemBuilder(baseItem);

        System.out.println("- Change Item -");
        System.out.println(baseItem + " (Enter \"!\" to skip)");
    }

    System.out.println("Enter Name:");
    builder.setName(ConsoleReadingUtils.readString(isItemChange));

    System.out.println("Select Type:");
    Object itemType = ConsoleSelectionUtils.printTypeSelection(ItemType.values(),
isItemChange);
    if (itemType == null) {
        builder.setType(null);
    } else {
        builder.setType((ItemType) itemType);
    }

    System.out.println("Enter Amount:");
    builder.setAmount(ConsoleReadingUtils.getAmount(1, 0, isItemChange));

    System.out.println("Enter Expiration Date: (dd/MM/YYYY)");
    builder.setExpirationDate(ConsoleReadingUtils.getDate(isItemChange));

    System.out.println("Enter Consumption Date: (dd/MM/YYYY)");
    builder.setConsumptionDate(ConsoleReadingUtils.getDate(isItemChange));

    System.out.println("Select Tags:");
    builder.setTags(TagInputHandlerUtils.printSelectTagsDialog(isItemChange));
}
```

```

    return builder.build();
}

```

Danach: (Commit: 89fe05148e32e42184c730909a7fa016f41c431f)

```

private Item createOrChange(Item baseItem) {
    boolean isItemChange = baseItem != null;
    ItemBuilder builder = createItemBuilder(isItemChange, baseItem);

    printCreateOrChangeWelcomeText(isItemChange, baseItem);

    getName(isItemChange, builder);
    getType(isItemChange, builder);
    getAmount(isItemChange, builder);
    getExpirationDate(isItemChange, builder);
    getConsumptionDate(isItemChange, builder);
    getTags(isItemChange, builder);

    return builder.build();
}

```

ItemInputHandler		
m	ItemInputHandler()	
m	printItemDeletionDialog(Item, Container)	void
m	getTags(boolean, ItemBuilder)	void
m	printItemCreationDialog(Container)	void
m	createItemBuilder(boolean, Item)	ItemBuilder
m	getType(boolean, ItemBuilder)	void
m	getName(boolean, ItemBuilder)	void
m	printItemOptions(Container, Item)	void
m	createOrChange(Item)	Item
m	getAmount(boolean, ItemBuilder)	void
m	printItemChangeDialog(Item, Container)	void
m	printCreateOrChangeWelcomeText(boolean, Item)	void
m	getExpirationDate(boolean, ItemBuilder)	void
m	getConsumptionDate(boolean, ItemBuilder)	void

(Es gab in einem späteren Commits weitere Änderungen an der createOrChangeItem Methode: a906a9f20ac066291ba059252590baba1e5e8143, 369bce0026fda458625f6fb8a899a4f0665ff30f)

Rename Method: (ConsoleReadingUtils)

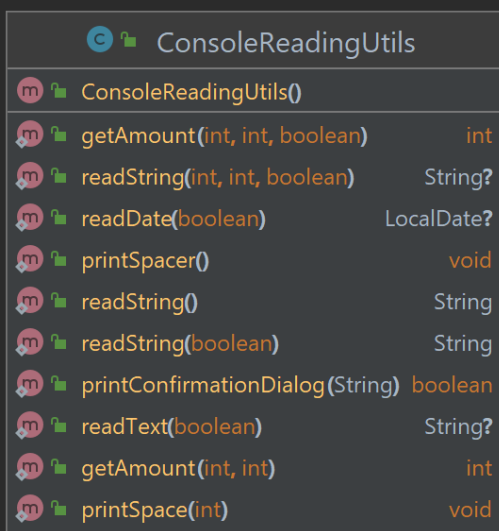
Da die getDate Methode nicht der einheitlichen Benennung der “Read” Methoden entsprach, wurde diese entsprechend zu readDate umbenannt. Dies führt zu einer besseren Verständlichkeit, da die Methode das Datum einer Benutzereingabe ausliest, was durch “get” nicht beschrieben war.

Zuvor:

```
public static LocalDate getDate(boolean canBeSkipped) {  
    ...  
}
```

Danach: (Commit: 470392b8a36fa8feefedbaaff393bc7e3d0b55da)

```
public static LocalDate readDate(boolean canBeSkipped) {  
    ...  
}
```

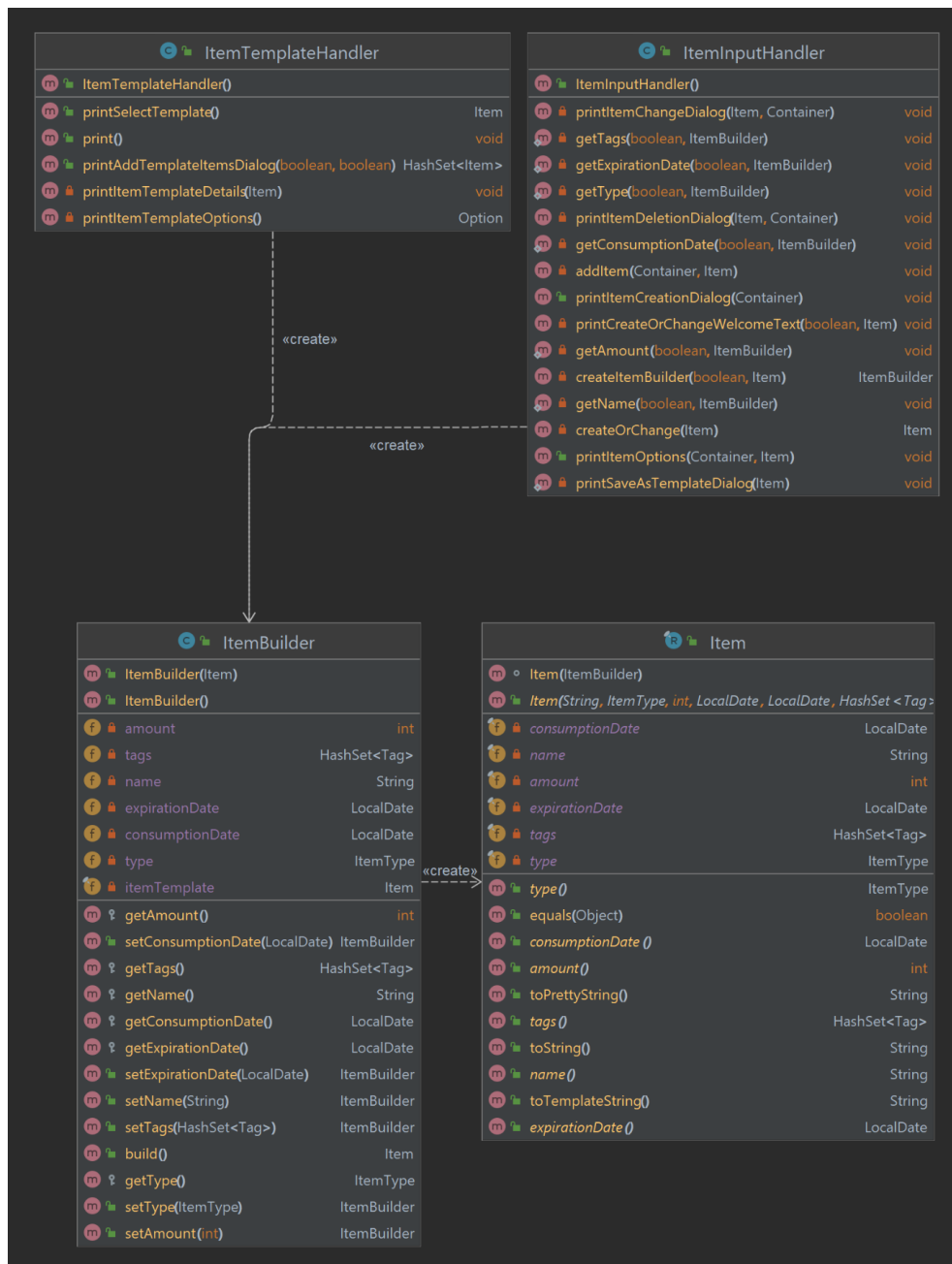


ConsoleReadingUtils	
ConsoleReadingUtils()	
getAmount(int, int, boolean)	int
readString(int, int, boolean)	String?
readDate(boolean)	LocalDate?
printSpacer()	void
readString()	String
readString(boolean)	String
printConfirmationDialog(String)	boolean
readText(boolean)	String?
getAmount(int, int)	int
printSpace(int)	void

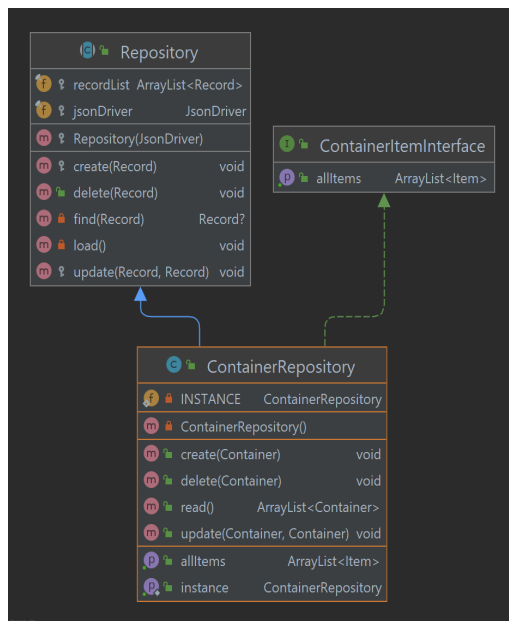
Kapitel 8: Entwurfsmuster (8P)

Entwurfsmuster: Builder Pattern (4P)

Der ItemBuilder ermöglicht es, Items in verschiedenen Varianten zu erstellen. Nicht alle Attribute des Items sind immer notwendig. Zusätzlich wird der ItemBuilder genutzt, um ein Item aus einem ItemTemplate-Record zu erstellen. Das Übernehmen der bereits existierenden Attribute eines ItemTemplates wird durch den ItemBuilder realisiert. Dazu werden die 'statischen' Werte wie der Name, Type und Tag aus dem Template genommen und 'dynamische' Werte wie ConsumptionDate und ExpirationDate durch eine Benutzereingabe ergänzt.



Entwurfsmuster: Singleton (4P)



existiert.

Das `ContainerRepository` besitzt einen privaten Konstruktor, sodass die einzige Möglichkeit, eine Instanz zu erzeugen, die statische `getInstance` Methode ist. Diese Methode prüft, ob bereits eine Instanz existiert und gibt entsprechende die erzeugte oder existierende Instanz zurück. So wird sichergestellt, dass es immer nur eine Instanz des `ContainerRepository` zur Laufzeit geben kann. Das `ContainerRepository` wurde als Singleton implementiert, da es nur eine Instanz geben soll, von der aus auf Container zugegriffen werden kann. Mehrere Instanzen hätten hier keinen Nutzen, da im `Repository` immer alle Container Instanzen hinterlegt sind und diese nicht mehrfach geladen werden müssen. So wird sichergestellt, dass von jedem aus den `Json-Dateien` geladenen Container immer nur eine Instanz zur Laufzeit