BerufsAkademie Rhein-Main
University of Cooperative Education - staatlich anerkannt -

Projekt „Getränkehandel“

Studiengruppe:	AI_WS20_III
Autoren (Matrikelnummer - Firma):	Jan Bauer (208765 - Werkstoffbit) Simon Hoim (208615 – DB Systel GmbH) Sebastian Kirner (200017 – Commerzbank AG) Felix Köhler (208619 – Volksbang eG) Jonas Kroker (200001 – DB Systel GmbH) Cedric Schmitt (200005 – Bundesamt für Wirtschaft und Ausfuhrkontrolle)
Datum:	19.03.2021
Fach:	Programmieren, Design und Implementierung von Algorithmen
Dozent	Herr Jürgen Rolf
Institution	Berufsakademie Rhein Main

BerufsAkademie Rhein-Main
University of Cooperative Education - staatlich anerkannt -

Abstract

Das folgende Projekt wurde in Gruppenarbeit realisiert. Es wurden Untergruppierungen gebildet. Diese bestehen aus folgenden Personen und Themen:

Team	Kommilitonen
GUI	Jan Bauer, Jonas Kroker
Klassen und Vererbung	Sebastian Kirner, Cedric Schmitt
Logik	Simon Hoim, Felix Köhler

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Problembeschreibung

Die Aufgabe ist, ein objektorientiertes Programm zur Verwaltung eines standortübergreifenden Getränkehandels zu erstellen, welches eine Klassenhierarchie abbilden soll. Die folgende Tabelle zeigt die Sollvorgaben:

Getränkeart	Eigenschaft	Flaschen pro Kasten	Zentrallager (in Flaschen)	Standort 1 (in Flaschen)	Standort 2 (in Flaschen)
Mineralwasser still	Glas- oder Plastikflasche	6	200	100	50
Mineralwasser mit Kohlensäure	Glas- oder Plastikflasche	12	400	200	100
Apfelsaft	Fruchtgehalt	6	200	100	50
Orangensaft	Fruchtgehalt	6	400	200	200
Limonade	Fruchtgehalt	12	300	150	100
Bier	Alkoholgehalt	24	200	150	150

Tabelle 1: Sollbestand

Beim Programmstart sind alle Lagerbestände nach Sollvorgabe befüllt.
Bei jeder der folgenden Funktionen wird der Lagerbestand aktualisiert.
Durch eine Umbuchfunktion kann zwischen allen Standorten jede Getränkesorte in Kästen verschoben werden. Wenn ein Kasten nicht mehr vollständig befüllt ist, kann dieser nicht mehr umgebucht werden.
Mit Hilfe einer weiteren Funktion soll der Lagerbestand eines Standortes mit Kästen aus dem Zentrallager aufgefüllt werden.
Die Getränkesorten können sowohl Flaschen- als auch Kastenweise von jedem Standort verkauft werden.
Durch eine automatische Nachbestellung wird für jeden Standort der Soll-Lagerbestand geprüft und fehlende Kästen nachbestellt.
In einer Detailansicht sollen die Eigenschaften der Getränkesorten angezeigt werden.
Wenn Flaschen einzeln verkauft werden und ein Kasten leer wird, dann wird der leere Kasten aus dem Lagerbestand entfernt.

2. Lösungsalternativen

2.1 Anpassen der Aufgabenstellung

Das Anpassen der Aufgabenstellung bezieht sich auf die vorgegebenen Werte, welche die Lagerkapazitäten der jeweiligen Standorte festlegen.
Die vorgegebenen Startwerte verursachen einen Konflikt bei dem Befüllen der Standorte. Das Zentrallager, aus welchem die einzelnen Standorte befüllt werden, hat bei der Getränkeart „Bier" eine kleinere Lagerkapazität als die kombinierte Lagerkapazität der Standorte. Somit kann diese Getränkeart nie bei allen Standorten komplett aufgefüllt sein. Aus diesem Grund könnte man, um dieses Problem zu vermeiden, die Lagerkapazität dieser Getränkesorte bei dem Zentrallager anheben.

2.2 Klassenstruktur

Es gibt mehrere Wege der Klassenstruktur, welche zum Ziel führen. Die Hauptbestandteile des Programms sind die Getränkearten.
Eine Möglichkeit wäre, alle Getränkearten mit einer gemeinsamen Klasse zu realisieren. Diese Klasse würde beim Initialisieren die für die Getränkeart jeweils spezifischen Werte, wie Art und Flaschen pro Kiste, übernehmen. Das für die jeweilige Getränkeart spezielle Attribut Alkoholgehalt, Fruchtgehalt oder Flaschenart könnte ebenfalls in dieser Getränkeart-Klasse untergebracht werden. Dafür könnte man zwei Variablen einbauen. Eine dieser Variablen beschreibt das Attribut, welches dieses repräsentiert, z.B.: Fruchtgehalt, die andere Variable den dazugehörigen Wert. In diesem Beispiel wäre das eine Zahl, welche aus Kompatibilitätsgründen als Zeichenkette gespeichert werden müsste.
Das Getränkeart spezifische Attribut könnte man ebenfalls mit einer eigenen Klasse umsetzen. Dies würde die Klassenstruktur übersichtlicher gestalten. Jedoch haben beide Varianten das gleiche Problem. Dadurch, dass je nach Getränkeart in der Variable, welche den Wert des Attributes speichert, eine Zeichenkette oder eine Zahl sein kann, muss dies bei dem Programmieren sehr genau beachtet werden. Falls diese Eigenschaft nicht beachtet wird, kann es sehr schnell zu Problemen kommen.
Eine Ebene weiter oben könnte man eine Kasten-Klasse verwenden. In dieser Klasse würden dann Flaschen mit der jeweiligen Getränkeart abgespeichert werden. Diese Klasse vereinfacht die Handhabung mit dem Verschieben des Bestandes von einem Standort zu einem anderen. Bei diesem Vorgang gibt es die Vorgabe, dass nur volle Kästen bewegt werden dürfen. Über eine Methode könnte bei dieser Klasse der aktuelle Stand abgefragt werden, aus welchem resultiert, ob dieser Kasten bewegt werden darf.
Noch eine Ebene weiter könnte mittels einer Lager-Klasse der aktuelle Stand der jeweiligen Standorte umgesetzt werden. Diese Klasse würde von den jeweiligen Standorten initialisiert werden und repräsentiert deren Lagerbestand. Über „getter" und „setter" Methoden könnte die Standort-Klasse auf die Werte des eigenen Lagers zugreifen.

2.3 Datenverwaltung

Für das Verwalten der Daten gibt es erneut mehrere Möglichkeiten:
Mittels Serialisierung und Deserialisierung können die aktuellen Werte gespeichert und wieder geladen werden. Bei dem Beenden des Programms werden die aktuellen Werte, wie z.B.: Flaschen im Lager etc., in einer Datei gespeichert. Durch die Verwendung des XML oder JSON Formates wäre zudem eine einfache Einbindung möglich. Diese Daten können zudem auf anderen Systemen oder für Backups verwendet werden. Bei dem Start des Programmes werden die Daten aus der Datei, falls vorhanden, geladen und der Ablauf kann fortgeführt werden.
Durch die Aufteilung des Systems in Server und Clients ist eine strikte Trennung der Datenintegrität möglich. Die Standorte inklusive des Zentrallagers verwenden die Clients. Diese verbinden sich mit dem Server, einem Computer, welcher nur für die Organisation und das Speichern der Daten zuständig ist. Über definierte Befehle können die Clients mit dem Server interagieren und die auf dem Server abgelegten Daten abrufen oder modifizieren.
Die Daten auf dem Server werden mittels einer Datenbank gespeichert und organisiert. Diese Datenbank speichert z.B.: aktuelle Lagerstände, Getränkearten, Standorte etc. Mittels Tabellen werden diese Daten getrennt voneinander abgespeichert. Über Kreuztabellen können die Werte der Tabellen dem passenden Standort zugewiesen werden.

2.4 Visualisierung

Für die Visualisierung der Daten gibt es erneut mehrere Möglichkeiten:
Die Darstellung mit einer Konsole hat den Vorteil, dass diese Visualisierung sehr einfach umsetzbar ist. Zudem hat jedes Betriebssystem eine Konsole und ist somit universell einsetzbar. Das Eingeben von Befehlen, um mit dem System zu interagieren, kann jedoch schnell mühsam werden.
Eine Alternative zu der Konsole ist eine Browser-basierte Darstellung. Das Erstellen einer solchen Darstellung ist zwar aufwendiger, jedoch deutlich einfacher zu bedienen. Die Webseite kann die lokalen Daten darstellen oder auch in das Server-Client-System integriert werden. Dafür wird die Webseite auf dem Server gehostet und von den Standorten über den Browser aufgerufen.

3. Begründete Auswahl einer Alternative

Die Auswahl zwischen verschiedenen Alternativen kann man grob in zwei Unterkategorien aufteilen. Zum Einen gibt es Zusätze zur Aufgabenstellung, die zwar nicht unbedingt notwendig, aber sinnvoll sind und zum Anderen gibt es die Wahl zwischen Alternativen, die zwingend getroffen werden mussten, da sonst Grundbausteine im Programm fehlen würden.

3.1 Kleine Zusätze

In den Klassen „WaterSparkling" und „WaterNonSparkling" (Mineralwasser mit und ohne Sprudel) wird das Attribut „bottleType" (Flaschenart) nicht parallel zu den anderen Tochterklassen von „DrinkType", die mit einem Integer gefüllt werden, mit einem String oder einer eigenen Klasse gefüllt, sondern mit einem Enum. Dieser hat im Gegensatz zum String den Vorteil, dass er Tippfehler beim Initialisieren einer der Wasserflaschen eventuelle Tippfehler vorbeugt. Außerdem ist er übersichtlicher und speicheroptimierter als eine eigene Klasse.
Die Ausgangsbestände des Zentrallagers wurden angepasst, damit keine einzelnen Flaschen übrigbleiben, wenn man die Standorte mit dem Zentrallager auffüllt. Die Ausgangsbestände in Flaschen sind nun durch die Anzahl der Kästen teilbar.

3.2 GUI

Obwohl es von der Aufgabenstellung nicht explizit gefordert ist, enthält das Programm ein „Graphical User Interface" (kurz „GUI"), mit welcher der Benutzer die Flaschen transferieren und die Bestände der Standorte einsehen kann. Im Gegensatz zu beispielsweise einer Terminalausgabe gibt es hier farbig hinterlegtes Feedback und durch Klicken statt Tippen eine einfachere und intuitivere Bedienung. Dadurch entsteht eine deutlich höhere Nutzerfreundlichkeit. Außerdem wird das Programm so auf allen Systemumgebungen, die Java installiert haben, gleich angezeigt, was den Umstieg zwischen Systemen vereinfacht.

3.3 Automatisches Nachfüllen des Zentrallegers

Die Funktion „Lager Nachfüllen" für einen Standort verhält sich so, wie auf dem Workflowdiagramm (Abb. 1) beschrieben.

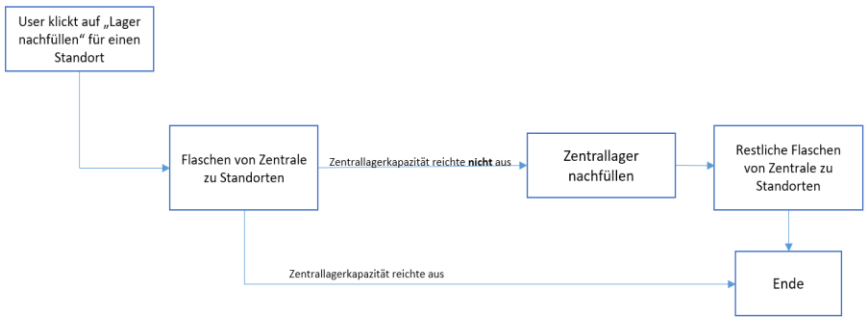

Abbildung 1: Workflowdiagramm - Nachfüllen von Standorten

So, wie es im Moment geregelt ist, ist das Zentrallager nicht voll aufgefüllt, wenn der Prozess zu Ende ist. Dadurch kann der Benutzer sehen, dass nicht alle Standorte aufgefüllt werden, sondern das Programm nur dann das Zentrallager auffüllt, wenn es nötig ist.

3.4 Getränkeklassenstruktur

Es gibt eine Mutterklasse „DrinkType", von welcher die verschiedenen Getränkearten über Tochterklassen erben (Abb. 2). „DrinkType" enthält Funktionen, wie „getter" und „setter", sowie Attribute, die für alle Tochterklassen gelten. Durch die gewählte Form ist die Struktur ordentlich und die Dateien kompakt. Außerdem kann man bei dieser Form einfach neue Tochterklassen hinzufügen.

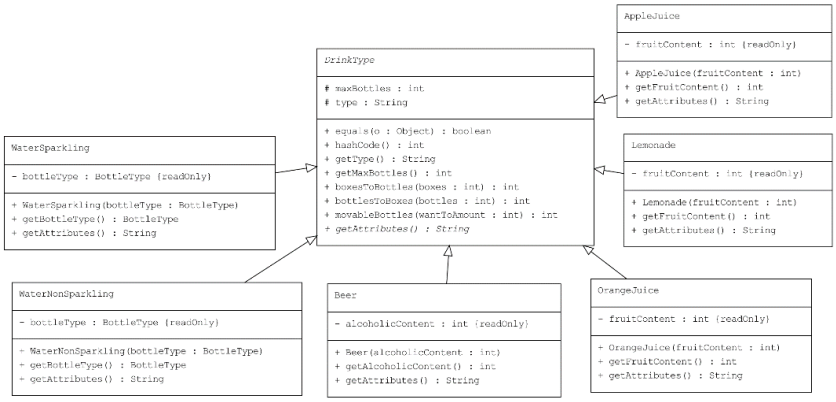

Abbildung 2: UML Diagramm - Getränkeklassenhierarchie

3.5 Lagerbestand

Der Bestand und die Kapazität der Lager werden als Map in der Klasse „Location" gespeichert. Dadurch kann man die Daten einfach in „Location" abrufen, initialisieren und durchsuchen. Alternativ könnte man der Klasse Location zwei Tochterklassen hinzufügen, die diese Werte speichern. Dies wäre jedoch weniger effizient in der Ressourcennutzung und die beiden Tochterklassen müssten untereinander aufeinander zugreifen. Außerdem müssten in „Location" immer noch Helferfunktionen sein, was den Sinn der beiden Unterklassen aufhebt.

4. Programmvorstellung

Beim Programmstart sind drei Standorte vorhanden: Standort 1, Standort 2 und ein Zentrallager. Diese sind beliebig erweiterbar. Die vordefinierten Getränkesorten sind Mineralwasser still, Mineralwasser mit Kohlensäure, Apfelsaft, Orangensaft, Limonade und Bier.
Der Bestand und die Getränkesorte kann verändert oder auch neue Getränkesorten im Nachhinein hinzufügt werden. Durch die integrierte Detailansicht lassen sich zu jeder Getränkesorte Zusatzinformationen, wie den Frucht- oder Alkoholgehalt und ob es sich um eine Glas- oder Plastikflasche handelt, darstellen.
Folgende Funktionen sind standardmäßig integriert:

- Verkaufen: Von jedem Standort kann jede Getränkesorte in Flaschen verkauft werden.
- Umbuchen: Zur Steigerung der Flexibilität kann zwischen den Standorten jede Getränkesorte umgebucht werden.
- Auffüllen: Diese Funktion füllt alle Standorte (außer dem Zentrallager) nach Sollvorgabe vom Zentrallager auf. Wenn beim Auffüllen nicht genug

Flaschen im Zentrallager vorhanden sind, wird eine Fehlermeldung generiert. Das Zentrallager muss dann zuerst aufgefüllt werden.

- Farbkennzeichnung: Der Bestand wird farblich visualisiert. Wenn der Ist-Lagerbestand bei Null liegt, zeigt die Zeile rot, bei weniger als 20% gelb und bei mehr als 20% grün.
- Hilfestellung: Durch eine integrierte Bedienungsanleitung spart man Schulungs- und Zeitkosten, da schnell Abhilfe geleistet wird.
- Benutzerfreundlich: Bei der Eingabe von Negativzahlen in die Eingabefelder wird eine Fehlermeldung angezeigt. Bei eingegebenen Buchstaben passiert hingegen nichts.

Jeder Reiter ist mit einem Icon versehen. Durch die visuelle Gestaltung findet man sich schneller zurecht.

Für die Versionsverwaltung wurde Git verwendet. Das Projekt ist öffentlich einsehbar unter folgendem Link: https://github.com/J-S-Bach/Getraenkehandel-Aufgabe-8

5. Ausarbeitung GUI

5.1 GUI Komponenten Aufbau

Die GUI ist die graphische Schnittstelle zwischen Benutzer und Programmcode. Die Struktur der GUI sieht folgendermaßen aus:

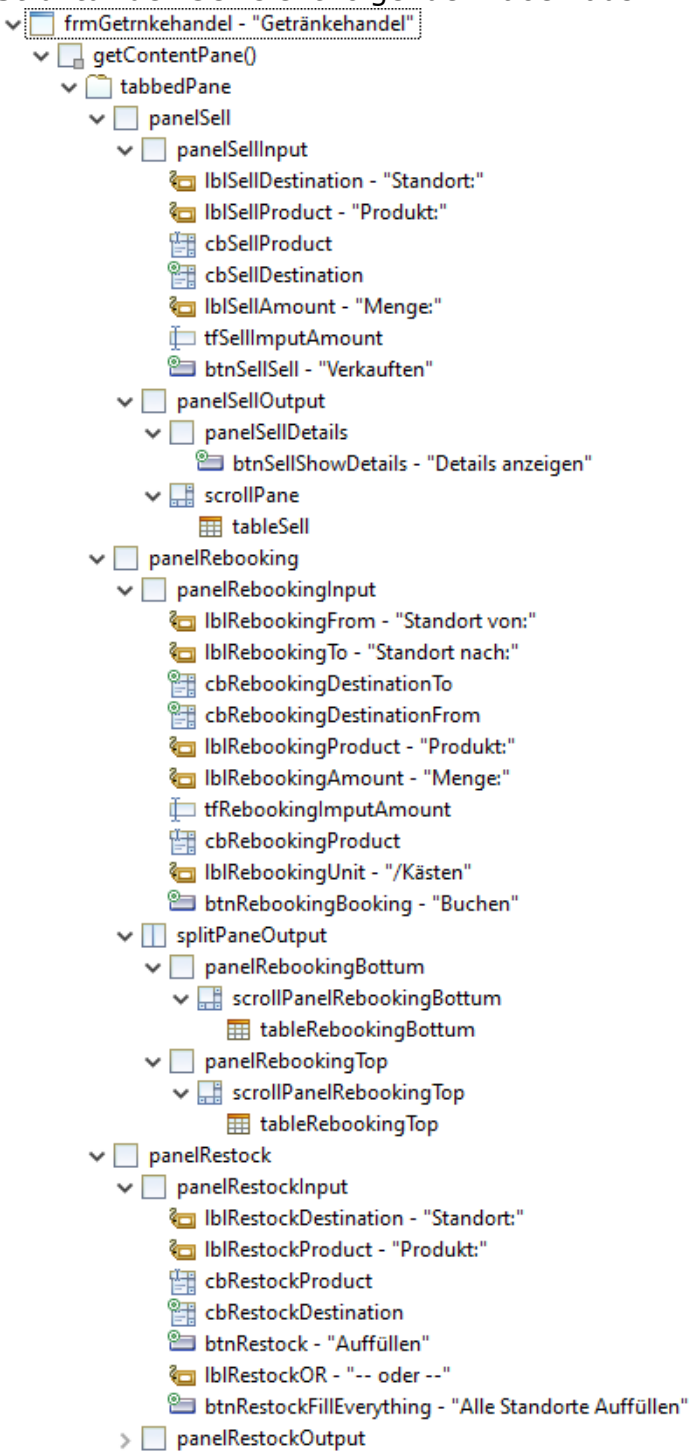

Abbildung 3: Struktur der Komponenten der GUI

5.2 ColorRenderer

Die Klasse ColorRenderer erbt von der Klasse JLabel und implementiert TableCellRenderer. Der TableCellRenderer bekommt aus der Klasse GUIHelperMethods über eine Schleife immer die nächste Zeile aus der Tabelle. Die Methode TableCellRenderer speichert sich die Werte aus der Zeile und vergleicht den Bestand mit festen Werten. Je nach dem welcher Fall eintrifft, werden die einzelnen Zeilen gefärbt.

5.3 GUIHelperMethods

Die Klasse GUIHelperMethods bietet Hilfsmethoden zur Unterstützung in der GUI.

5.3.1 createMergedObject

Die Methode bekommt zwei Zweidimensionale Arrays vom Typ Objekt übergeben und vereint beide Arrays zu einem Zweidimensionalen Array von Typ Objekt.

5.3.2 generateImageIcon

Die Methode übernimmt einen Pfad zum Bild als String und gibt das gewünschte Bild als ImageIcon zurück.

5.3.3 getAllLocationNames

Die Methode gibt einen Array von Typ String zurück, der mit den Namen aller Standorte gefüllt ist.

5.3.4 getAllProductsFromLocation

Die Methode gibt alle Produkte eines Standorts in einem Array zurück.

5.3.5 getSortimentAndAttributesOfLocation

Die Methode übernimmt als Parameter einen Index, der zur Bestimmung des Standortes gebraucht wird. In der Methode wird ein Zweidimensionaler Array von Typ Objekt mit allen Produkten und Eigenschaften der Produkte gefüllt.

5.3.6 getSortimentOfLocation

Die Methode übernimmt als Parameter einen Index, der zur Bestimmung des Standortes gebraucht wird. In der Methode wird ein Zweidimensionaler Array von Typ Objekt mit allen Produkten des Standortes gefüllt.

5.3.7 renderColor

Die Methode übernimmt als Parameter den Überschriftenarray der Tabelle sowie auch die Tabelle. Die Methode überliefert der Klasse ColorRenderer die einzelnen Zeilen der Tabelle.

5.3.8 stingToDrinkType

Die Methode übernimmt als Parameter einen Index, der zur Bestimmung des Standortes gebraucht wird sowie den Produktnamen als Objekt. Die Klasse

vergleicht die DrinkTypenamen mit dem angegeben Parameter und gibt den passenden DrinkType zurück.

5.4 Images

In dem Ordner Images liegen alle Bilder/Logos, die in dem Projekt verwendet wurden.

5.5 StyledTabbedPanelUI

StyledTabbedPanelUI erbt von der Klasse BasicTabbedPanelUI und gestaltet die Tabs im TabbedPanel neu. Über die beiden Methoden paint und paintBackgroud bekommt der Tab eine 3D Artige Form.

6. Programmbeschreibung

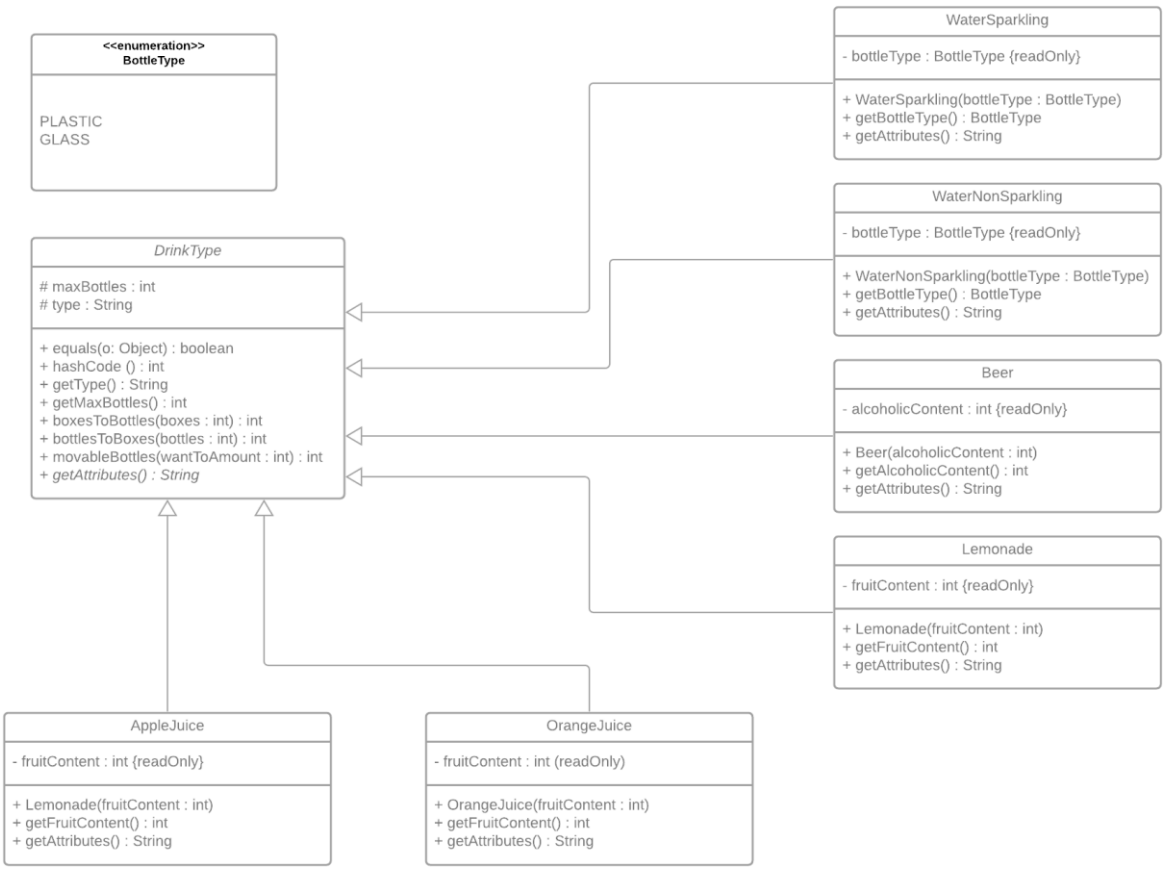

Abbildung 4: UML Diagramm - Vererbung der Getränkesorten

6.1 DrinkType

Die Klasse „DrinkType" ist die Mutterklasse der verschiedenen Getränke. Sie stellt verschiedene Attribute und Funktionen bereit, die die Tochterklassen erben, also alle gemein haben. Im oberen Teil des UML Klassendiagramms sind die Attribute zu sehen, im unteren Teil die Funktionen. Mit der Eigenschaft „maxBottles" wird angegeben, wie viele Flaschen maximal in einen Kasten passen und mit „type", welche Art von Getränk es ist.
Die Funktion „equals(Object o)" überprüft, ob ein neu erstelltes Objekt („o") schon einmal instanziiert wurde und gibt in diesem Fall „true" zurück.

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    DrinkType drinkType = (DrinkType) o;
    return type.equals(drinkType.type);
}
```

Abbildung 5: Programmausschnitt - Funktion „equals"

„hashCode()" erstellt für ein instanziiertes Objekt einen Hashcode. „boxesToBottles(bottles: int)" und „bottlesToBoxes(boxes: int)" geben für einen gegebenen Wert die entsprechende Anzahl Flaschen bzw. Kästen zurück. Sie beziehen sich hierbei auf die Eigenschaft „maxBottles". Ein besonderer Getter ist „getAttributes()", da hier die Attribute der Tochterklassen zurückgegeben wird.

6.2 AppleJuice, Lemonade, OrangeJuice

Diese Klassen besitzen alle ein zusätzliches Attribut namens „fruitContent". Dieser ist vom Typ Integer und gleicht dem Fruchtgehalt in Prozent. Dieser wurde durch den Modifikator „final" auf Readonly gesetzt. Sobald ein „final" Attribut initialisiert wurde, kann dieser nicht mehr durch zum Beispiel einem Setter geändert werden. In diesen Klassen wurde die „getAttributes()" Methode implementiert, um den Fruchtgehalt als Klartext ausgeben zu können.

```
@Override
public String getAttributes() {
    return "Fruchtgehalt: " + getFruitContent() + "%";
}
```

Abbildung 6: Programmausschnitt - Funktion "getAttributes"

6.3 Beer

Bei „Beer" wurde das zusätzliche Attribut „alcoholicContent" implementiert. Dieses Attribut ist wie der „fruitContent" ebenfalls ein Integer und gleicht der Alkoholgehalt in Prozent.
In dieser Klasse wurde die „getAttributes()" Methode implementiert, um den Alkoholgehalt als Klartext ausgeben zu können.

6.4 WaterSparkling, WaterNonSparkling

Diese Tochterklassen haben ihr Attribut „bottleType" mit dem Enum „bottleType" gefüllt, weil hier zwei individuelle, aber festgelegte Werte gebraucht werden.
In diesen Klassen wurde die „getAttributes()" Methode implementiert, um den Flaschentyp als Klartext ausgeben zu können.

6.5 BottleType

„BottleType" ist ein Enum (Aufzählungstyp) der alle Arten von Flaschen (Glas oder Plastik) beinhaltet. Hier wurde ein Enum und nicht nur ein Text verwendet, damit sichergestellt werden kann, dass eine Flasche nur genau diese Werte hat. So wird sichergestellt, dass Rechtschreibfehler keine Auswirkung haben, da diese direkt vom IDE angezeigt werden.

```java
public enum BottleType {
    GLASS("Glas"), PLASTIC("Plastik");

    private final String name;
    BottleType(String s) {
        name = s;
    }
    public String toString() {
        return this.name;
    }
}
```

Abbildung 7: Programmausschnitt - Enum "BottleType"

Da Enums intern Integer-Werte haben (Beginnend bei 0 und dann immer +1) haben wir den Konstruktor mit einem String-Parameter überladen, um jedem „BottleType" auch einen eigenen Namen geben zu können. Dieser Name wird intern als final String „name" gespeichert. Zum Ausgeben wurde die „toString()"-Methode überschrieben. Diese gibt nun nur noch unseren eigenen Namen zurück.

6.6 Location Management Logik

6.6.1 Location

Die Klasse „Location" dient der Repräsentation eines Standortes. Jeder Standort hat einen eigenen Namen, eine festgelegte Kapazität und einen aktuellen Lagerstand zu jeder Getränkesorte.
Die Kapazität und der aktuelle Lagerstand wird mittels einer Map umgesetzt. Diese Map hat als Schlüssel die jeweilige Sorte des Getränks und der dazu korrespondierende Wert stellt die Kapazität oder den aktuellen Bestand dar.

```java
public class Location {
    protected String name = "";
    protected Map<DrinkType, Integer> capacity = new HashMap<>();
    protected Map<DrinkType, Integer> drinks = new HashMap<>();

    ...
}
```

Abbildung 8: Programmausschnitt - Funktion "Location"

Die Methoden der Klasse liefern Auskünfte zu dem aktuellen Lagerstand, zu der maximalen Kapazität oder auch der Differenz zwischen diesen. Zudem gibt es Methoden, welche Getränke hinzufügen, entfernen oder auch von einem Standort zu einem anderen bewegen.
Diese Methoden besitzen Logik, welche verhindert, dass zu viele Flaschen entfernt, hinzugefügt oder bewegt werden. Währenddessen werden auch die Vorgaben beachtet, dass zum Beispiel die Methode zum Bewegen der Kästen nur volle bewegen darf.

```java
public boolean addDrink(DrinkType drinkType, int amount) throws Exception {
    if (amount < 0)
        throw new Exception("Negative value not valid!");
    if (this.getDrinkCapacity(drinkType) < this.getDrinkAmount(drinkType) + amount)
        return false; // not enough capacity
    this.drinks.put(drinkType, this.getDrinkAmount(drinkType) + amount);
    return true;
}
```

Abbildung 9: Programmausschnitt - Funktion "addDrink"

Des Weiteren gibt es zwei Methoden zum Auffüllen der Location. Die erste Methode füllt den aktuellen Standort von einem anderen und zieht bei diesem die entnommenen Kästen ab. Die zweite Methode führt die vorherige Methode für jede Getränkesorte aus und erleichtert somit die Handhabung des Auffüllens.
Zuletzt gibt es eine „toString" Methode, welche das Objekt leserlich in der Konsole darstellt.

6.6.2 CentralStorage

Die Klasse „CentralStorage" erbt von der Klasse Location, da das zentrale Lager ein Standort mit einer zusätzlichen Methode ist, welche bei den normalen Standorten nicht benötigt wird. Bei dem Initialisieren des Objektes werden direkt die Kapazitäten zu den Getränkesorten festgelegt und im Anschluss wird die Methode „fill" aufgerufen, welche alle Getränkesorten des Zentrallagers füllt.

```java
public void fill() { // on startup after capacity is set
    for (DrinkType dt : this.getDrinkTypes()) {
        try {
            this.addDrink(dt, this.getDrinkCapacity(dt));
        } catch (Exception e) {}
    }
}
```

Abbildung 10: Programmausschnitt - Funktion "fill"

6.6.3 LocationManager

Die Klasse „LocationManager" verwaltet alle Standorte und das Zentrallager. Bei dem Initialisieren des Objektes wird die Methode „setupLocations" aufgerufen, welche das Zentrallager und alle Standorte erzeugt. Die Standorte bekommen die Kapazitäten zu den jeweiligen Getränkesorten zugewiesen und zum Schluss werden alle Standorte in einem Array abgelegt.
Die Methode „fillLocations" iteriert über alle Standorte und versucht bei diesen die jeweiligen Getränkesorten von dem Zentrallager aufzufüllen. Dies würde jedoch bei der Anwendung fehlschlagen, da die Standorte kombiniert mehr Kapazität für einzelne Getränkesorten haben als das Zentrallager.
Für dieses Problem gibt es die Methode „autoFill". Diese Methode iteriert ebenfalls über alle Standorte und versucht jede Getränkesorte aufzufüllen. Falls das Zentrallager jedoch weniger Kästen von dieser Getränkesorte gelagert haben soll,

wird diese nachbestellt und in dem Zentrallager komplett aufgefüllt, bevor diese dann in den jeweiligen Standort bewegt wird.

```
public void autoFill() {
    for (Location l : locations) {
        for (DrinkType dt : l.getDrinkTypes()) {
            if (l.getMissing(dt) > central.getDrinkAmount(dt)) {
                try {
                    central.addDrink(dt, central.getMissing(dt));
                } catch (Exception e) {}
            }
            try {
                l.fillFromLocation(dt, central);
            } catch (Exception e) {
            }
        }
    }
}
```

Abbildung 11: Programmausschnitt - Funktion "autoFill"

7. Fazit und Ausblick

Zurückblickend auf das Projekt „Saftladen-Manager", ist zu sagen, dass durch anfängliche Planung und durch Aufteilung in einzelne Gruppen mit verschiedenen Aufgabengebieten das Projekt schnell und nahezu fehlerfrei voranschritt. Durch wöchentliche virtuelle Treffen (per Discord) hielten wir uns gegenseitig auf dem jeweiligen Stand der einzelnen Gruppen. Da wir zwischenzeitlich stetig Tests vollführten, wurden wir schnell auf Fehler oder Komplikationen aufmerksam und konnten diese zeitnah beseitigen. Somit wurden zukünftige Komplikationen minimiert.
Bezüglich eines Wachstums des Programms durch Erweiterungen der Standorte, Zentrallager oder auch Produktpalette etc., kann durch Anstieg der Datenmenge eine Datenbank in Betracht gezogen werden. (siehe 2.3, „Datenverwaltung")
Ein weiteres Feature wäre, dass man automatisiert vor Ort Getränke nachbestellen könnte. Hierzu wäre beispielsweise ein Scanner nötig, welcher per REST API („Representational State Transfer Application Programming Interface") in das System eingebunden werden könnte. Durch einen Scanner wäre das Nachbestellen einfacher und schneller. So könnte man im Arbeitsalltag beispielsweise eine Flasche scannen und die Anzahl der Kästen bzw. Flaschen sehen. Falls eine „kritische" Anzahl erreicht ist, könnte man dann dieses Produkt automatisch nachbestellen lassen oder die Anzahl verändern bzw. das Produkt aus der Produktpalette rauswerfen. Die Benutzeroberfläche könnte dann beispielsweise so aussehen:

Abbildung 12: Mögliche Erweiterung der Benutzeroberfläche

Bezüglich der einzelnen Standorte könnte man auch verschiedene Benutzer einrichten, welche auflisten, wer wo und wann etwas nachbestellt hat. So hat man immer einen Überblick über die Verkäufe. Das System kann dann auf Verkaufszahlen aufmerksam machen und dementsprechend zeigen, in welche Getränkesorte man am besten Werbung investiert und welches Produkt entfernt werden sollte. Außerdem kann das Programm, falls eine sehr hohe Verkaufszahl vorliegt, eine Erhöhung des Lagerbestandes vorschlagen.

8. Quellen

Logodesign: Lasse Soenksen