Selbstlernaufgabe OMP-SST-02 (Lösungshinweise)

Objektorientierte Modellierung und Programmierung (inf031) Sommersemester 2020
Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik

Dr. C. Schönberg

Entwerfen und implementieren Sie ein Computerspiel *Hanse* (eng. *Hansa*). Dabei handelt es sich um eine Aufbau- und Handelssimulation:

- Es gibt verschiedene Typen von *Gebäuden* (*buildings*): *Kontor* (*office*), *Hafen* (*harbour*), *Lagerhaus* (*warehouse*), *Schiffswerft* (*shipyard*) und *Markt* (*market*).
- Es gibt verschiedene Typen von *Schiffen* (*ships*): *Kleines Handelsschiff* (*small trading ship*), *Mittleres Handelsschiff* (*medium trading ship*) und *Großes Handelsschiff* (*large trading ship*).
- Schiffe haben abhängig von ihrem Typ eine *Geschwindigkeit* (*speed*) und eine *Ladekapazität* (*storage capacity*) für Güter.
- Es gibt verschiedene Typen von *Gütern* (*goods*): *Stein* (*stone*), *Holz* (*wood*), *Metall* (*metal*), *Stoff* (*cloth*) und *Getreide* (*grain*).
- Es gibt mehrere *Orte* (*locations*), auch *Hansestädte* (*Hanseatic cities*) genannt: *Bremen*, *Danzig*, *Hamburg*, *Kiel*, *Krakau*, *Riga*, *Rostock*, *Stettin* und *Stockholm*.
- Jeder Ort ist (per Schiff) von jedem anderen Ort erreichbar, nur die Entfernungen zwischen den Orten sind unterschiedlich.
- Jedes Gebäude befindet sich an einem festen Ort, also in einer Hansestadt.
- Schiffe befinden sich entweder an einem festen Ort (einer Hansestadt), oder sind unterwegs zwischen zwei Orten. In diesem Fall wird eine *Ankunftszeit* (*time of arrival*) berechnet, die abhängig von der Entfernung der Orte und der Geschwindigkeit des Schiffs ist.
- Gebäude und Schiffe kosten *Geld* (*money*) und Güter zum Bauen. Gebäude können gegen Geld und Güter zu höheren *Stufen* (*levels*) ausgebaut werden.
- Gebäude haben verschiedene Funktionen:
  - Im Kontor findet die Verwaltung statt. Andere Gebäude können *höchsten* so hoch ausgebaut werden, wie der Kontor momentan ausgebaut ist.
  - Im Hafen *ankern* (*dock*) Schiffe. Für jede Stufe findet ein Schiff dort Platz. Schiffe die nicht ankern können sind in einer *Warteschlange* (*queue*), bis ein Platz frei wird. Schiffe können vom Hafen aus zu einem anderen Ort geschickt werden.
  - Im Lagerhaus werden Güter gelagert. Die Ladekapazität ist abhängig von der Stufe des Gebäudes. Schiffe können vom Lagerhaus aus mit Gütern be- und entladen werden.
  - In der Schiffswerft können neue Schiffe *gebaut* (*build*) werden. Höhere Gebäudestufen ermöglichen neue Schiffstypen.
  - Auf dem Markt können Güter aus dem Lagerhaus für Geld *gekauft* (*buy*) und *verkauft* (*sell*) werden. Höhere Gebäudestufen führen zu einem leicht verbesserten *Kurs* (*rate*).
- Es gibt mehrere *Spieler* (*player*) im Spiel:

- Jeder Spieler beginnt mit einem Kontor (Stufe 1), einem Hafen (Stufe 1), einem Lagerhaus (Stufe 1) und einem Markt (Stufe 1) an *jedem* Ort.
- Jeder Spieler beginnt mit einer Menge Geld und Güter, sowie einem Kleinen Handelsschiff an *einem* Ort.
- Gebäude und Schiffe *gehören* (*own*) genau zu einem Spieler, d.h. kein Spieler kann die Gebäude anderer Spieler nutzen.

- Güter werden in einer abstrakten *Maßeinheit* (*unit*) gezählt, die bezüglich des Volumens vergleichbar ist. Das bedeutet, dass eine „Einheit" Metall genau so viel Platz benötigt wie eine „Einheit" Getreide. Das Gewicht wird nicht betrachtet.

Aufgaben:

a) OOP, Vererbung, Interfaces, UML (Vorlesungen 1, 2, 3 und 5)
Modellieren Sie die notwendigen Klassen als UML Klassendiagramm. Achten Sie auf die sinnvolle Verwendung von Abstrakten Klassen, Interfaces und Enumerations.
Überlegen Sie sich, an welchen Stellen eine Klassenhierarchie sinnvoll ist und wo nicht. Gemeinsame Attribute oder Methoden sind beispielsweise oft ein Indikator für eine mögliche Hierarchie.

b) Polymorphie, Kapselung (Vorlesungen 3, 4, 5 und 8)
Überarbeiten Sie Ihre Modellierung (sofern nötig), so dass die Lagerung von Gütern für Schiffe und Lagerhäuser gleich funktioniert. Außerdem soll es möglich sein, alle Handelsschiffe im Hafen gleich zu behandeln.
Ergänzen Sie sinnvolle und notwendige Attribute und Methoden.
Gemeinsame Funktionalität kann grundsätzlich mindestens über zwei Mechanismen umgesetzt werden: Vererbung und Delegation.

c) Java (Vorlesungen 2, 3, 4 und 5)
Legen Sie Ihre modellierten Klassen in Java an.
Implementieren Sie eine Methode zum Ausbau von Gebäuden auf die nächste Stufe. Diese Methode funktioniert für alle Gebäudetypen gleich, bis auf das Kontor: Dort ist keine Überprüfung der Stufe des Kontors am gleichen Ort nötig. Wie können Sie das am geschicktesten umsetzen?
Denken Sie dabei an die Möglichkeit, Methoden in Oberklassen zu schreiben und sie in Unterklassen zu *über*schreiben.

d) Generics, JDK (Vorlesungen 9 und 10)
Verwenden Sie Klassen des JDK wie `Collection`, `List`, `Set` oder `Map`, wo dies sinnvoll ist.
Verwenden Sie Generics, um für beliebige Güter den Lagerbestand zu speichern.
Verwenden Sie eine Datenstruktur, die es Ihnen ermöglicht, mittels Generizität ein Gut auf seinen Lagerbestand abzubilden.

e) Exceptions, I/O (Vorlesungen 7 und 13)
Implementieren Sie Methoden zum Laden und Speichern eines Spielstands.
Verwenden Sie Exceptions und Exception-Handling für ungültige Operationen, beispielsweise das Verkaufen von leeren oder negativen Mengen, das Ausbauen von Gebäuden höher als der Kontor, sowie für I/O Fehler.
Es gibt verschiedene Möglichkeiten, eine Java-Klassenstruktur zu speichern, die unterschiedliche Eigenschaften haben. Z.B. führt die Verwendung von `DataInputStream` bzw. `DataOutputStream` üblicherweise zu laufzeit- und speichereffizienten Lösungen, ist aber aufwändig zu implementieren. Die Verwendung von Serializable ist deutlich weniger effizient, aber für den Programmierer auch weniger aufwändig.

f) Streams (Vorlesung 12)
Verwenden Sie Streams, um Anfragen an den Spielstand zu stellen:

- Finde alle Spieler, die mindestens ein Vermögen von X Geld besitzen.
- Finde alle Spieler mit untätigen Schiffen (d.h. Schiffe, die im Hafen liegen und unbeladen sind).

- Liste alle Gebäude (außer dem Kontor) von Spieler X auf, die nicht so weit ausgebaut sind wie möglich (also deren Level unter dem des lokalen Kontors ist).

Spätestens hier ist es sinnvoll, eine Klasse *Spiel* (*Game*) anzulegen, die eine Liste aller Spieler enthält.

g) Threads (Vorlesungen 16 und 17)
Verwenden Sie Threads und einen Mechanismus zur Synchronisation, um Schiffe, die unterwegs sind, zur Ankunftszeit automatisch zum Hafen hinzuzufügen. Es ist möglich, dass mehrere Schiffe zum gleichen Zeitpunkt ankommen.
Es gibt verschiedene Möglichkeiten, kritische Abschnitte zu schützen, beispielsweise `synchronized`, Locks oder Semaphoren. Wenn Sie die Ankunftszeit als *Reisezeit* (*travel time*) vom Typ `long` modellieren, dann können Sie das Warten auf die Ankunft eines bestimmten Schiffes sehr leicht implementieren.

h) Lösungsstrategien (Vorlesungen 19 und 20)
Nehmen Sie an, dass es an allen Orten feste Ein- und Verkaufspreise gibt. Finden Sie unter dieser Annahme diejenige Handelsroute, die den größten Gewinn erzeugt.
Wir betrachten hier zur Vereinfachung nur Handelsrouten von einer Länge L (Anzahl der Haltepunkte), mit beliebigem Start- und Endpunkt, befahren von einem Schiff mit Ladekapazität K, das an jedem Haltepunkt exakt N Einheiten von einem beliebigen Gut verkauft und N Einheiten von einem beliebigen anderen Gut kauft. L, K und N sind beliebig, aber fest.

- Implementieren Sie ein Verfahren, das *Backtracking* verwendet, um alle Lösungen aufzulisten.
- Optimieren Sie anschließend Ihr Verfahren, indem Sie schlechte Lösungen möglichst früh ausschließen (*Branch and Bound*).
  Wie kann man frühzeitig erkennen, dass eine Lösung nicht mehr besser wird als eine bereits vorher gefundene Lösung?
- Implementieren Sie ein Verfahren, das die *Lokale Suche* verwendet, um eine möglichst gute Lösung zu finden.
  Was sind sinnvolle Initiallösungen, Nachbarschaften, Qualitätskritierien und Abbruchbedingungen?

i) SML (Vorlesung 22)
Folgende Modellierung eines Lagerhauses mit einer Reihe von Gütern und ihrer jeweiligen Anzahl, sowie eines Marktes mit einer Reihe von Gütern und ihrem jeweiligen Preis sei gegeben:

```
1 val warehouse = [(”grain”, 1), (”wood”, 2), (”stone”, 7)];
2 val market = [(”grain”, 4),(”wood”, 3),(”stone”, 5)];
```

- Schreiben Sie eine Funktion `getPrice` mit nachfolgender Signatur, die für ein Gut und einen Markt den Preis des Gutes bestimmt:

```
1 val getPrice = fn : ’’a * (’’a * int) list -> int
```

  Möglicherweise gibt Ihre Implementierung eine Warnung aus, diese können Sie ignorieren:

```
1 stdIn:18.35 Warning: calling polyEqual
```

- Schreiben Sie eine Funktion `profit`, welche die maximale Geldsumme berechnet, die durch Verkauf aller Güter im Lager erzielt werden kann. Sie dürfen dabei die Funktion `getPrice` verwenden.

```
1 val profit = fn : (’’a * int) list * (’’a * int) list -> int
```

Achten Sie darauf, dass Lagerhaus und Markt nicht die gleichen Güter enthalten müssen, und auch nicht in der gleichen Reihenfolge.

j) Prolog (Vorlesung 23)

- Modellieren Sie die Orte und die Verbindungen zwischen den Orten in Prolog. Anders als oben angegeben müssen hier nicht alle Orte miteinander verbunden sein.
- Schreiben Sie eine Relation `path`, welche für zwei Orte angibt, ob es eine (direkte oder indirekte) Verbindung dazwischen gibt.

Sie dürfen beliebige Hilfs-Relationen definieren.

k) Drools (Vorlesung 24)
Definieren Sie eine Drools-Regel zur Verwaltung der Warteschlange an einem Hafen.
Verwenden Sie für den Hafen selbst eine eigene `Collection`-Implementierung, die eine Methode `isFull` bereitstellt, und für die Warteschlange eine `LinkedList`.

Wenn Sie die UML-Teilaufgaben in UMLet modellieren, dann können Sie den UML `Validator` verwenden, um Ihre Lösung zu prüfen und um Feedback und Hilfestellungen zu bekommen. Der UML `Validator` benötigt eine korrekt installierte und aktuelle Java Laufzeitumgebung (korrekt gesetzter Pfad und Classpath, mindestens Java 13).
Speichern Sie dazu Ihre (Teil-)Lösung in einer UMLet-Datei ab. Laden Sie aus Stud.IP die zu dieser Aufgabe passende Validierungs-`.class`-Datei herunter (z.B. `OMP_SST_02_a.class` für Aufgabe `OMP-SST-02-a`). Laden Sie aus Stud.IP die Datei `UMLValidator.zip` herunter und entpacken Sie diese. Führen Sie nun entweder die `UMLValidator.jar` direkt aus, oder rufen Sie die `run.bat` (unter Windows) bzw. die `run.sh` (unter Linux/OS X) auf. Dort wählen Sie nun Ihre UMLet-Datei und die entsprechende `.class`-Datei und klicken auf "Validate".
Beachten Sie bitte, dass sich der UML `Validator` noch in einer *frühen Testphase* befindet.