Assignment 5

Abgabe bis zum 23. Mai 2022 (Dienstag) um 23:59 Uhr

Geben Sie Ihr Assignment bis zum 23. Mai 2022 (Dienstag) um 23:59 Uhr auf folgender Webseite ab:
https://assignments.hci.uni-hannover.de
Ihre Abgabe muss aus einer einzelnen zip-Datei bestehen, die den Quellcode und ggf. ein PDF-Dokument bei Freitextaufgaben enthält. Beachten Sie, dass Dateien mit Umlauten im Dateinamen nicht hochgeladen werden können. Entfernen Sie die daher vor der Abgabe die Umlaute aus dem Dateinamen. Überprüfen Sie nach Ihrer Abgabe, ob der Upload erfolgreich war. Bei technischen Problemen, wenden Sie sich an Kevin Schumann oder Dennis Stanke. Es wird eine Plagiatsüberprüfung durchgeführt. Gefundene Plagiate führen zum Ausschluss von der Veranstaltung. In dieser Veranstaltung wird ausschließlich die Java/JavaFX Version 17 verwendet. Code und Kompilate anderer Versionen sind nicht zulässig. Ihre Abgaben sollen die vorgegebenen Dateinamen verwenden. Wiederholter Verstoß gegen diese Regel kann zu Punktabzügen führen.

Aufgabe 1: Schatzsuche

Innerhalb dieser Aufgabe werden Sie einen endlichen deterministischen Automaten implementieren. Endliche Automaten bestehen aus einer endlichen Menge an Zuständen (States). Abhängig vom aktuellen Zustand geht man bei einer Eingabe (später Aktion) entweder in einen anderen Zustand über oder bleibt im selben Zustand. Deterministisch bedeutet, dass für jede Eingabe in dem aktuellen Zustand ein einzigartiger Folgezustand erreicht wird. Endlichen Automaten haben einen Startzustand und ggf. sogar mehrere Zielzustände. Ziel ist es den in Figure 1 gezeigten Automaten zu realisieren. States werden durch Inseln repräsentiert. Pfeile stellen Transitionen dar. Es sind in jedem State die Eingaben A und B möglich. Nicht eingezeichnete Pfeile sind Transitionen, die auf den selben Zustand verweisen.

Legen Sie selbständig eine geeignete Packagestruktur an. Schreiben Sie für alle Klassen, Methoden und Member-Variablen, die Sie erstellen oder ändern, JavaDoc Kommentare. Halten Sie sich an die bisherige JavaDoc Konvention.

a) Implementieren Sie den Enum-Typen *Action*, welche die Werte "A" und "B" annehmen kann. Erstellen Sie ebenfalls die Methode `public char str()`, welche eine eindeutige Repräsentation des Wertes zurück gibt.

b) Erstellen Sie das Interface *State*, welches die Methoden `public State transition(Action action)`, `public String str()` und `public String info()` darstellt. Die Methode *transition*, soll die Transition abhängig vom aktuellen Zustand und der übergebenen Aktion realisieren. Die Methode gibt den Folgezustand zurück. *Info()* gibt die Bezeichnung des aktuellen Zustands sowie die Übergangsmöglichkeiten (bspw. Aktion a -> Shipwreck Bay; Aktion b -> Musket Hill) als String zurück. *Str()* returned lediglich die Bezeichnung des jeweiligen Zustands.

c) Erstellen Sie für jeden in Figure 1 gezeigten Zustand eine Klasse mit dem jeweiligen Zustandsnamen (Inselnamen), welches das Interface aus b) implementiert.

d) Implementieren Sie die Klasse *StateMachine*, welche den kompletten Automaten darstellt. StateMachine speichert die aktuellen Zustand als Attribut und ob der Zielzustand erreicht wurde. Der Startzustand lautet *Pirates' Island*, der Zielzustand *Treasure Island*. Ergänzen Sie eine main Methode innerhalb derselben Klasse

Vorlesung: apl. Prof. Matthias Becker xmb@hci.uni-hannover.de
Übung: Kevin Schumann k.schumann@stud.uni-hannover.de
Organisation: M.Sc. Dennis Stanke dennis.stanke@hci.uni-hannover.de

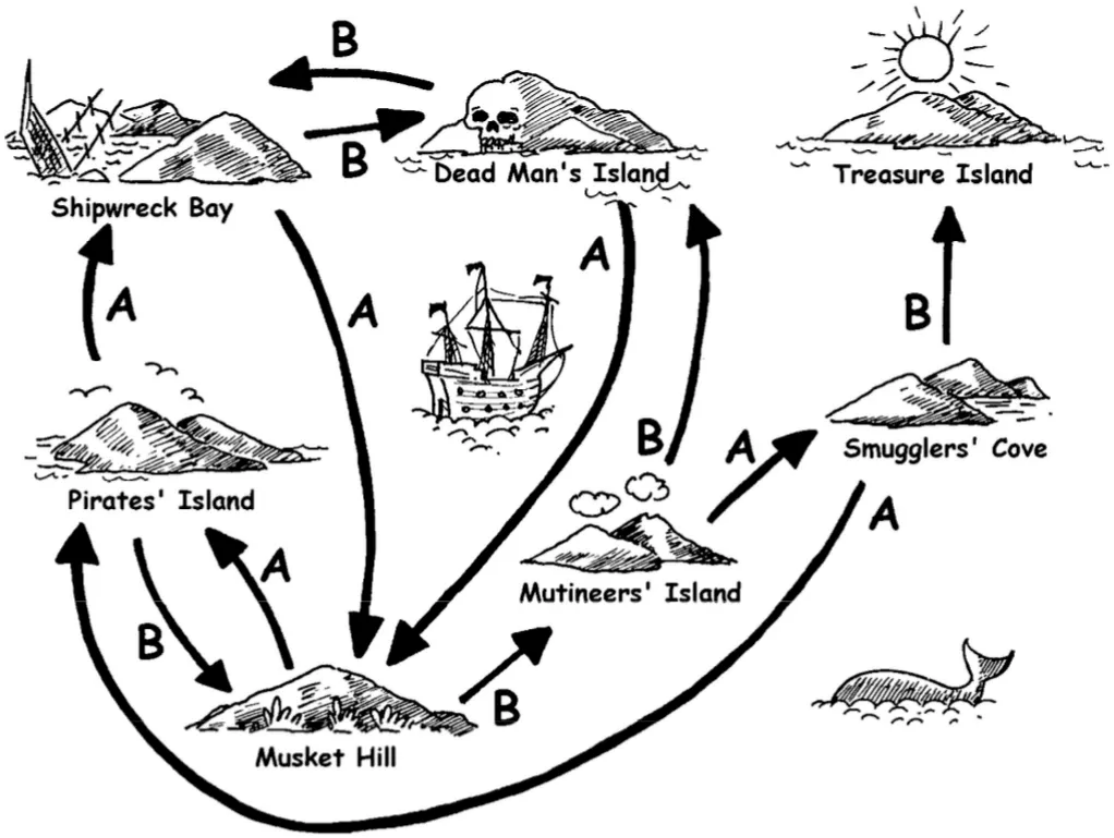

Figure 1: Endlicher Automat: States werden durch Insel repräsentiert. Pfeile stellen Transitionen dar. Es sind in jedem State die Eingabe A und B möglich. Nicht eingezeichnete Pfeile sind Transitionen, die auf den selben Zustand verweisen. Quelle

um ein Kommandozeileninterface, welches den Nutzer solange nach der Wahl zwischen den Aktionen ”A” und ”B” fragt, bis der Zielzustand erreicht wurde. Nach jedem Zustandsübergang soll die Info des aktuellen Zustands ausgegeben werden. Wurde der Zielzustand erreicht soll der komplette bewältigte Weg von dem Startzustand bis zum Endzustand ausgegeben werden.

Aufgabe 2: Jump ’n Run 2

Sie sollen in dieser Aufgabe mit bereits existierendem Code arbeiten. Diesen Code mitsamt generiertem JavaDoc finden Sie in der Task2.zip im StudIP.

Schreiben Sie für alle Klassen, Methoden und Member-Variablen, die Sie erstellen oder ändern, JavaDoc Kommentare. Halten Sie sich an die bisherige JavaDoc Konvention. Arbeiten Sie in der vorgegebenen Packagestruktur aus der Vorlage.

a) Das Interface `GameInterface` wird von Klassen implementiert, die ein auf der Kommandozeile spielbares Spiel darstellen. Lesen Sie sich das generierte Javadoc des Interfaces durch. Erstellen Sie eine Klasse `Main` mit einer `main` Methode und erstellen Sie ein Kommandozeileninterface, mit welchem Sie ein Spiel, das das `GameInterface` Interface implementiert, spielen können. Nutzen Sie dabei ausschließlich die Methoden in `GameInterface` um mit dem Spiel zu interagieren. Um zu testen, ob Ihr Kommandozeileninterface funktioniert, können Sie die Snake Klasse nutzen.

b) Im Folgenden werden Sie eine Abwandlung des von Ihnen in Assigment 3 angefertigten Jump ’n Run Spiel implementieren. Diesmal soll sich der Spieler in alle Himmelsrichtungen bewegen können. Außerdem betrachten man das Spiel nicht mehr von der Seite, sondern von oben, sodass man das ganze Spielbrett sehen kann. Das Spiel soll das `GameInterface` Interface implementieren. Sie werden innerhalb dieser Aufgabe selbständig Klasse, Enum-Typen etc. erstellen, um das Spiel zu nach folgenden Regeln zu realisieren:

- Das Spielfeld besteht aus 20x20 Spielobjekten.
  - Ein Spielobjekt kann entweder begehbarer Boden, eine Wand, Lava, ein Hindernis in der Luft oder das Ziel sein.
  - Das Spielfeld ist komplett von Wänden umgeben, welche sich innerhalb des Spielfelds befinden.
  - Wenn das Spielfeld initialisiert wird, sollen zusätzlich 40 Wand, Lava und Lufthindernisse zufällig auf dem Spielfeld verteilt werden.
  - Das Spiel hat ein einziges Ziel, was von Lava und Lufthindernissen umrandet ist.
- Die Spielfigur startet auf einem zufälligen Boden-Tile.
- Die Spielfigur hat eine Blickrichtung (Norden, Osten, Süden, Westen).
- Die Spielfigur kann zwischen stehen und hocken umschalten, indem der ”B” Knopf ”gedrückt” wird.
- Die Spielfigur kann in seine aktuelle Blickrichtung ”dashen”, indem der ”A” Knopf ”gedrückt” wird. Dashen bedeutet sich direkt um 2 Felder in seine Blickrichtung zu bewegen. ”Dashen” ist nur im Stehen möglich.
- Hockt die Spielfigur, kann sie sich unter Lufthindernissen bewegen.
- Kollidiert die Spielfigur mit einer Wand, steht auf Lava, oder läuft gegen eine Lufthindernis, hat man verloren.

- Kollidiert die Spielfigur während des ”dashen” mit einem Lufthindernis, hat man verloren.
- Steht die Spielfigur auf dem Ziel, hat man gewonnen.

Ein beispielhaftes Spielbrett ist in Figure 2 zu sehen.

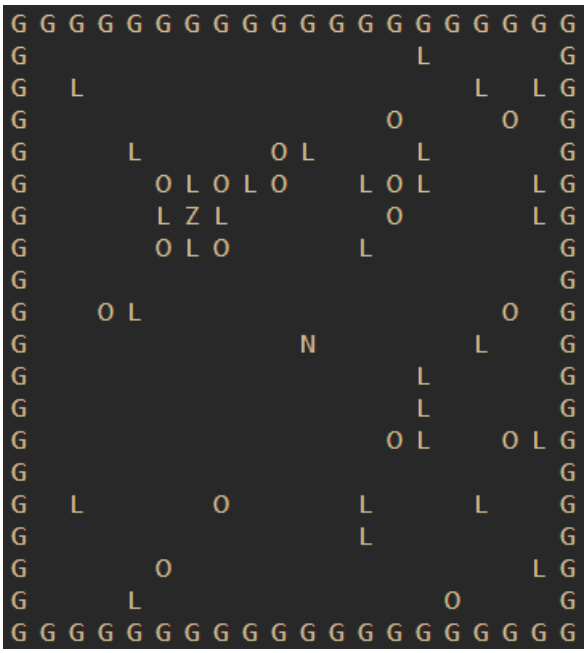

Figure 2: Beispielhaftes Spielbrett: G steht für eine Wand, L für Lava, O für eine Lufthindernis und Z für das Ziel. Der Spieler hat den Buchstaben N, da er nach Norden blickt.

Ändern Sie Ihre `main` Methode, sodass der Nutzer am Anfang gefragt wird, welches Spiel gespielt werden soll, Ihr eigenes oder das Jstris Spiel aus der Vorlage.

Lösungshinweise:

- Spielobjekte lassen sich durch ein Enum-Typen repräsentieren, wie in Assignment 3.
- Die Blickrichtung lässt sich ebenfalls durch einen Enum-Typen darstellen. Es kann hilfreich sein, die Koordinatenänderung in einer 2D-Welt bei einer Bewegung in die jeweilige Blickrichtung innerhalb des Enum-Wertes zu speichern.
- Das Erstellen einer Player-Klasse, die die aktuellen Koordinaten und die Blickrichtung speichert, ist sinnvoll. Das Gleiche gilt für Methoden, die zwischen Stehen und Hocken umschalten, in die aktuelle Blickrichtung gehen oder ”dashen”.
- Das Spielfeld kann, wie in Assignment 3, durch ein 2D-Spielobjekt-Array dargestellt werden.

c) Beschreiben Sie einen kurzen Text, wie sie die Rolle von Interfaces bei der Modularität von Code einschätzen. Gehen Sie dabei auch auf Probleme und Herausforderungen ein, die Sie bei der Verwendung von Interfaces sehen. Sofern Sie keine Probleme oder Herausforderungen sehen, schreiben Sie dies explizit in Ihren Text. Schreiben Sie diesen Text in einen Kommentar unter die `main` Methode.

Aufgabe 3: Debugging

In diesen Aufgaben werden Sie einen fehlerhaften Java Codeabschnitt bekommen. Diese Fehler können **syntaktisch** oder **semantisch** sein. Es kann sich dabei um Compiler- oder Laufzeitfehler handeln. Sie dürfen den Code kompilieren und ausführen um die Fehler zu finden. Arbeiten Sie in der Template-Datei `Debug5.zip`. Diese finden Sie im Stud.IP.

a) Der Codeabschnitt enthält 5 Fehler darunter auch fehlende Methoden. Kompilieren und führen Sie den Code aus. Suchen Sie anhand der Fehlermeldungen des Compilers oder der Laufzeitfehlermeldungen Fehler im Code und korrigieren Sie diese. Kommentieren Sie am Ende der jeweiligen Zeile, was Sie korrigiert haben.

Erstellen Sie während der Fehlerkorrektur einen Blockkommentar unter dem Code, in dem Sie die Fehler dokumentieren. Schreiben Sie die Zeile(n) des Fehlers auf und beschreiben Sie den Fehler. Kopieren sie die Fehlermeldung, sofern es eine gab, anhand welcher Sie den Fehler erkannt haben. Die Dokumentation soll wie in folgendem Beispiel aussehen:

```
1  public class Debug { //K: class falsch geschrieben (ckass)
2
3  ...
4
5  /*
6  ...
7  Zeile 1: class Keyword falsch geschrieben (ckass):
8  Fehlermeldung:
9  **************
10 Debug.java:1: error: class, interface, or enum expected
11 public ckass Debug {
12        ^
13 **************
14 Der Compiler erwartet eines der drei oben angegebenen Keywords, hat aber nur das falsch
       ↪ geschriebene ckass bekommen.
15 ...
16 */
```

Im Code vorgegebene Kommentare beschreiben immer das **korrekte** Verhalten des Programms.