

Text-Adventure

(eine fakultative Ferienaufgabe)

Universität zu Köln
Informationsverarbeitung
Wintersemester 2016/2017
Softwaretechnologie: Java I
Dozent: Mihail Atanassov

Inhaltsverzeichnis

TextAdventure	2
Implementation	2
Dateiformate	2
places.config	2
world.config	3
initial_game.config	3
Ein- und Ausgabe.....	3
Runtime.....	4
Interaktion mit dem Spieler	4
Mini-Spiele	4
Vorschlag zur Vorgehensweise.....	5
Rückmeldung	5
Design	5
Dokumentation.....	6
API Docs.....	6

TextAdventure

Gegenstand der Hausarbeit ist die Implementation eines Text-Adventures. Diese inzwischen kaum noch verbreitete Form des Computerspiels zeichnet sich dadurch aus, dass auf graphische Ausgaben vollständig verzichtet wird und Spieler und Spiel ausschließlich durch Ein- und Ausgaben auf der Konsole interagieren - wesentlicher Hintergrund dieses minimalistischen Ansatzes war die eingeschränkte Leistungsfähigkeit von PCs in den 70er und 80er Jahren. Eines der bekanntesten Spiele ist das 1984 erschienene „Per Anhalter durch die Galaxis“, das inzwischen online und kostenlos unter <http://www.douglasadams.com/creations/infocomjava.html> gespielt werden kann.

Implementation

Deine Aufgabe ist es, die fehlende Funktionalität der Programmvorlage zu implementieren. Dazu gehört im Wesentlichen: Laden der notwendigen Dateien, Verwalten von Spiel-Zuständen und -Logik sowie Laden und Speichern von Spielständen.

Dateiformate

Ein Text- Adventure wird hier in Form von 3 Dateien (UTF-8 kodiert) beschrieben, die in einem gemeinsamen Verzeichnis liegen müssen. In der Datei **places.config** sind die Details zu jedem Ort definiert, während in der Datei **world.config** angegeben ist, wie die Räume angeordnet sind. Die Datei **initial_game.config** definiert schließlich, in welchem Raum der Spieler mit dem Spiel beginnt. Im Folgenden eine kurze Beschreibung des jeweiligen Dateiformats:

`places.config`

Die Liste aller Orte in **places.config** ist durch leere Zeilen voneinander getrennt. Jeder Raum wird durch verschiedene Attribute beschrieben, von denen die meisten optional sind. Ein Attribut besteht aus einem eindeutigen Bezeichner (z.B. *name* oder *action*), gefolgt vom Gleichheitszeichen und einem weiteren String, der als Wert des Attributs interpretiert werden muss. Folgendes Beispiel zeigt alle Attribute eines Ortes:

```
name=Zülpicher Straße
add_conditions=student_id
in_conditions=volljährig,durstig
remove_conditions=hungrig
action=Du stehst vor dem Stiefel.
game=schnickschnackschnuck
```

Lediglich **name** ist hier obligatorisch – der angegebene String dient zudem dazu, den Ort innerhalb der **world.config** Datei zu referenzieren, was bei Änderungen beachtet werden muss!!! **action** definiert die Ausgabe, die bei erstmaligem Betreten des Raumes ausgegeben werden soll. **in_conditions** ist eine durch Kommata getrennte Liste von Bedingungen, die erfüllt werden müssen, damit der Raum betreten werden darf (s.u.) - falls eine oder mehrere der Bedingungen nicht erfüllt sind, darf der Spieler den Raum nicht betreten. **add_conditions** ist eine durch Kommata getrennte Liste von Bedingungen, die dem Spieler hinzugefügt werden, wenn er den Raum erfolgreich betreten hat. **remove_conditions** ist ebenfalls eine durch Kommata getrennte Liste – die angegebenen Bedingungen werden von der Liste des Spielers entfernt, sobald der Raum betreten wurde. **game** definiert schließlich, welches Mini-Spiel am jeweiligen Ort gespielt werden muss, um den Raum betreten zu können.

`world.config`

Die Datei **world.config** definiert die Anordnung der Räume in der Welt und kann als 2-dimensionales Array bzw. als Matrix interpretiert werden. Mehrere Räume werden in jeder Zeile durch das Zeichen „|“ (Pipe) voneinander getrennt. Nicht vorhandene Räume sind durch „----“ gekennzeichnet.

`initial_game.config`

Beim Speichern und Laden eines Spiels müssen die aktuelle Position, die Liste der erfüllten Bedingungen sowie die Liste der bereits besuchten Orte gespeichert bzw. geladen werden. Die Datei **initial_game.config** definiert den initialen Spielstand, der zu Beginn geladen werden muss. In den ersten zwei Zeilen ist angegeben, an welcher Position sich der Spieler befindet, die folgenden zwei Zeilen erhalten erneut durch Kommata separierte Listen, die sich auf die Bedingungen bzw. Orte beziehen.

```
x=1
y=2
conditions=
entered=
```

Ein- und Ausgabe

Die notwendigen Methoden sind im Interface `spinfo.textadventure.io.IO` definiert, jedoch ergibt sich daraus nicht, in welcher Reihenfolge sie aufgerufen werden müssen – dies wird daher im Folgenden kurz erklärt.

Wird eine Instanz einer Implementation des Interfaces erzeugt, so muss zunächst die Methode `setGameDirectory()` aufgerufen werden: Damit wird festgelegt, welches Spiel gespielt werden soll, und wo die relevanten Dateien zu finden sind. Das Interface ist so angelegt, dass anschließend die Methode `loadPlaces()` aufgerufen werden soll, um die Liste der Orte bzw. Räume zu laden, und diese als Parameter der Methode

`loadWorld()` zu übergeben – so können Orte in der Welt positioniert werden. Nach diesem Schritt muss die Methode `loadPlayer()` aufgerufen werden, wobei hier durch den boolean-Parameter unterschieden wird, ob es sich um ein initiales Spiel oder um einen gespeicherten Spielstand handelt: Je nach Parameter muss entweder die Datei `initial_game.config` oder die Datei `save_game.config` geladen werden.

Runtime

Die Klasse `Runtime` ist bereits vollständig implementiert, so dass hier nichts verändert werden muss. `Runtime` ist für die Interaktion des Spielers mit der Welt verantwortlich, bzw. interpretiert die Eingaben auf der Konsole. Folgende Befehle sind möglich:

- **map** - Gibt die Spielwelt auf der Konsole aus, indem die Methode `World.printMap()` aufgerufen wird.
- **exit** - Beendet das Spiel (durch Aufruf von `System.exit()`).
- **save** - Speichert das Spiel im aktuellen Spiel-Verzeichnis, indem die Methode `IO.saveGame()` aufgerufen wird.
- **load** - Lädt das Spiel im aktuellen Spiel-Verzeichnis indem die Methode `IO.loadGame()` aufgerufen wird.
- **go north/south/east/west** - Bewegt den Spieler in den entsprechenden Raum, relativ zur aktuellen Position (falls der Raum vorhanden und das Betreten erlaubt), indem die Methode `Player.move()` aufgerufen wird.

Interaktion mit dem Spieler

Damit der Spieler sich im Spiel zurechtfindet, musst Du zwei Dinge umsetzen: (a) Zum einen musst Du dafür sorgen, dass die Methode `World.printMap()` eine formatierte Abbildung der Welt ausgibt – wie bei der Vorlage muss auch hier die Position des Spielers durch ein *-Symbol dargestellt werden, zudem muss die benötigte Spaltenbreite berechnet und bei der Ausgabe berücksichtigt werden. (b) Zum anderen musst Du an geeigneter Stelle dafür sorgen, dass der Spieler eine Rückmeldung über die ausgeführte Aktion bekommt: Beim ersten Betreten eines neuen Raums muss, falls vorhanden, die **action**-Eigenschaft des jeweiligen Raums ausgegeben werden. Sollte es hingegen nicht möglich sein, einen Raum zu betreten, so muss eine entsprechende Information ausgegeben werden.

Mini-Spiele

Die Vorlage enthält ein „Spiel-im-Spiel“, ist aber um beliebige weitere Spiele erweiterbar. Dazu muss zunächst das Interface `Game` aus dem Package `spinfo.textadventure.games` implementiert werden. Sie sollten das Spiel Schere-Stein-Papier (oder auch Schnick-Schnack-Schnuck) versuchen zu implementieren, optional können jedoch auch andere Spiele (bspw. Tic-Tac-Toe) implementiert werden.

Ein Mini-Spiel muss in der Methode `enter()` der Implementation von `spinfo.textadventure.data.Place` gestartet werden, wenn...

- ein Spiel mit dem Raum verknüpft wurde, indem es in der **places.config** Datei angegeben wurde und
- der Spieler sämtliche sonstigen Bedingungen, die für das Betreten des Raumes erforderlich sind, erfüllt und
- der Spieler den Raum zum ersten Mal betritt

Vorschlag zur Vorgehensweise

Ein Großteil des Programmieraufwands bezieht sich auf das Laden von Welt und Räumen sowie das Interpretieren der Relationen zwischen diesen. Es bietet sich daher an, mit diesem Punkt zu beginnen, und bspw. zunächst die Interfaces `World` und `Place` zu implementieren, um anschließend die entsprechenden Methoden des `IO`-Interfaces hinzuzufügen. Danach kann damit begonnen werden, die Funktionalität, die von der Klasse `Runtime` gefordert wird, umzusetzen.

Zur Verwaltung diverser Zustände empfiehlt sich (bei jetzigem Kenntnisstand) der Einsatz von Listen, konkret der Klasse `java.util.ArrayList`. Diese bietet nahezu alle Funktionen, die für die Implementation benötigt werden.

Die Implementation des Min-Spiels ist unabhängig vom restlichen Spiel und kann getrennt erfolgen – hier bietet es sich an, die Konfigurations-Datei zunächst zu modifizieren und auf das Mini-Spiel zu verzichten.

Rückmeldung

Um zu prüfen, ob die Implementation korrekt arbeitet, bietet es sich an, die Ausgabe mit der Ausgabe der Referenz-Implementation zu vergleichen. Um dies zu erleichtern, enthält das Projekt zudem JUnit-Tests, die das Testen der Funktionalität automatisieren. So wird bspw. automatisch überprüft, ob nur Räume betreten werden können, die (a) existieren und (b) deren Eintritts-Bedingungen erfüllt sind. Zum Ausführen der Tests muss die Klasse `AdventureTestSuite` aktiviert werden und der Punkt (Run as > JUnit Test) im Kontext-Menü ausgewählt werden. Die Tests sind nicht vollständig – so wird bspw. nicht überprüft, ob ein Mini-Spiel korrekt implementiert wurde. Schlägt jedoch ein Test fehl, so deutet dies darauf hin, dass ein Fehler im Programm vorliegt.

Design

Vermeide lange Methoden oder wiederkehrende, identische Codeabschnitte in Methoden! Wird eine Methode zu lang, so ist sie nicht mehr gut lesbar, und Fehler

lassen sich nur schwer finden. Prüfe in diesem Fall, ob die Methode in mehrere kleinere Methoden „zerschlagen“ werden kann. Wenn Du feststellst, dass ein Codeabschnitt mehr als einmal verwendet wird, solltest Du ihn lieber in einer eigenen Methode implementieren, und diese Methode mehrmals aufrufen.

Dokumentation

API Docs

Du solltest die von Dir angelegten Klassen mit JavaDoc dokumentieren. Das heißt:

- Jede Methode und jede Variable, welche nicht als `private` markiert ist, sollte dokumentiert werden.
- `private` Methoden sollten dokumentiert werden, wenn sich die Funktionalität nicht ohne weiteres erschließt, oder wenn bestimmte Anforderungen an die Parameter gestellt werden (`x` darf nicht `null` sein, `String` darf keine Leerzeichen enthalten o.ä.).
- Je mehr Methoden und Variablen `private` deklariert wurden, und je besser die Namen der Methoden/Variablen gewählt sind, desto weniger Aufwand ist bei einer internen Dokumentation erforderlich: `String s = ...;` sollte in jedem Fall dokumentiert werden, `String currentLine = ...;` nicht unbedingt.