Hausarbeit
Programmieren 1
Sommersemester 2020

Institut für Anwendungssicherheit
Prof. Martin Johns

3. August 2020

Rahmenbedingungen

Zeitplan

- Veröffentlichung der Aufgabenstellung: 03.08.2020
- Bearbeitungszeitraum: 03.08.2020 bis 15.09.2020
- Abgabezeitraum: 16.08.2020 bis 15.09.2020 23:59 Uhr
- Große Übung: 13.08.2020 um 15:00 Uhr
- Es finden keine kleinen Übungen während des Bearbeitungszeitraums statt

Es finden keine kleinen Übungen während des Bearbeitungszeitraums statt! Fragen zur Aufgabenstellung müssen bis zum 11.08.2020 an `m.karl@tu-bs.de` gesendet werden. Die Fragen werden im Zuge der großen Übung am 13.08.2020 um 15:00 Uhr in BBB (`https://webconf.tu-bs.de/man-7kk-pwd`) geklärt. **Außerhalb dieser großen Übung werden keine inhaltlichen Fragen zur Aufgabenstellung beantwortet!**

Hinweise

Achten Sie bei Ihrer Abgabe besonders auf das Einhalten der...

- maximalen Zeilenlänge von 120 Zeichen,
- Kompilierbarkeit Ihrer Lösung,
- Checkstyle-Regeln.

Bei den Checkstyle-Regeln handelt es sich um die von den Übungsblättern bereits bekannten Regeln. Diese werden Ihnen zusammen mit dem Projekt in Ihrem Git-Repository zur Verfügung gestellt. Sie finden die Regeln im Ordner src/main/resources.

Neben den spezifisch für die Aufgabe gestellten Anforderungen, gelten für die Hausarbeit die folgenden Regeln:

- Klassen, Methoden und Attribute erhalten sinnvoll gewählte Sichtbarkeiten, Namen und Attribute.
- Im normalen Programmablauf soll Ihre Abgabe keine zusätzlichen Ausgaben, beispielsweise zum Debugging, erzeugen.

Dieses Dokument soll Ihnen den generellen Aufbau der Aufgabe vermitteln und die einzelnen Komponenten vorstellen. Eine detailliertere Beschreibung der jeweiligen Methoden, deren Parameter und Rückgabewerte finden Sie in den JavaDoc Kommentaren der jeweiligen Interfaces und Klassen.

Sie dürfen für Ihre Implementierung alle Klassen aus den Paketen *java.lang* und *java.util* verwenden. Andere Pakete dürfen nicht genutzt werden.

Denken Sie daran, dass die Hausarbeit in Einzelarbeit bearbeitet werden muss. Die Abgaben werden einem Plagiatscheck unterzogen, bei dem auch Quellen aus dem Internet herangezogen werden. Ein Plagiat ist ein Täuschungsversuch und wird entsprechend für alle Beteiligten gemeldet!

Sollten Sie im Drittversuch sein und die Hausarbeit antreten, jedoch nicht bestehen, haben Sie KEIN Anrecht auf eine mündliche Ergänzungsprüfung (APO §13 Absatz 5)! Wir können nicht garantieren, dass dieser Passus für alle gilt. Lesen Sie hierzu in der für Ihren Studiengang relevanten Prüfungsordnung nach.

Abgabe

Der Code zur Hausarbeit wird Ihnen in einem Git-Repository unter `https://dev0.ias.cs.tu-bs.de/` zur Verfügung gestellt. Die Anmeldung erfolgt analog zum Übungsbetrieb mit Ihrer y-Nummer. Erstellen Sie sich hiervon eine lokale Kopie mittels *git clone* zur Bearbeitung der Hausarbeit.

Sollten Sie technischen Probleme mit Ihrem Repository haben, wenden Sie sich bitte per E-Mail an `manuel.karl@tu-bs.de`. Fragen zur regulären Nutzung von Git werden wir **nicht** beantworten.

Abgaben werden analog zu der Studienleistung **ausschließlich** in Ihrem persönlichen Git Repository akzeptiert. Gewertet wird, was am Ende des Bearbeitungszeitraums in dem **master** branch auf dem Server vorliegt. Abgaben per E-Mail werden nicht nicht gewertet.

Andere branches können Sie zum Testen und Entwickeln nutzen, werden für die Bewertung allerdings ignoriert.

Neben den von uns bereitgestellten JUnit-Tests existieren auch viele nicht veröffentlichte Tests, die in die Bewertung Ihrer Abgabe mit einfließen. Es genügt nicht, nur die bereitgestellten Tests zu erfüllen. Testen Sie Ihre Lösung zusätzlich selbstständig, um unnötige Fehler und damit fehlende Punkte in der Bewertung zu vermeiden. Hierzu können Sie auch eigene JUnit Tests im Ordner src/test/java/student anlegen. Versuchen Sie hierbei auch Rand- und Spezialfälle zu testen.

Struktur

Der von uns zur Verfügung gestellte Code ist als *Gradle*-Projekt strukturiert. Dies erlaubt es Ihnen direkt die mitgelieferten Tests auszuführen und den Code zu kompilieren. Bei Fragen zum Aufbau dieses Projektes lesen Sie sich das mitgelieferte README durch. Des Weiteren finden Sie in StudIP im Kurs **Programmieren 1 Sommersemester 2020** eine Video Aufzeichnung zum Thema *Gradle* sowie ein Testprojekt dazu.

Implementieren Sie Ihre Lösung ausschließlich im Paket *student*. Das Paket *ias* ist reserviert für Dateien, die von uns bereitgestellt werden. Hierzu zählen verschiedene Klassen, die Sie nutzen müssen sowie Interfaces, die Sie in Form eigener Klassen implementieren müssen. Die Dateien im Paket *ias* dürfen **nicht** modifiziert werden. Dies wird durch die *.gitignore* Datei sichergestellt. Diese darf nicht geändert werden.

In Ihrem Projekt finden Sie eine Datei *student.txt*, diese füllen Sie bitte aus. Ohne diese Informationen können wir Ihr Ergebnis nicht an Ihr Prüfungsamt melden.

Bewertung

Insgesamt können 100 Punkte bei der Bearbeitung der Aufgabe erreicht werden. Diese teilen sich auf in 10 Punkte für die Formatierung des Quellcodes und 90 Punkte für den Inhalt. Bitte bedenken Sie, dass eine erfolgreiche Bearbeitung der öffentlichen Tests und das Erfüllen der Checkstyle-Regeln nicht zum Bestehen der Abgabe ausreichen. Testen Sie deshalb Ihren Code ausgiebig auf die in der Aufgabenstellung und in den JavaDoc-Kommentaren beschriebene Funktionalität.

Beachten Sie des Weiteren, dass nur kompilierende Abgaben als Lösung gelten. Ein Programm, das Teillösungen beinhaltet aber nicht kompiliert oder aus anderen Gründen nicht ausführbar ist, führt zum nicht Bestehen der Hausarbeit!

Einleitung

In dieser Hausarbeit sollen Sie ein Framework zum Erstellen und Spielen von Text-Adventure-Games programmieren. Hierbei bewegt der Spieler eine Spielfigur auf einem Spielfeld und kann unterschiedliche Aktionen auf den Feldern ausführen. Ein Spiel besteht aus einem Ausgangszustand, der beschreibt, wie eine Menge von Objekten über dem Spielfeld verteilt sind. Mutationsregeln geben an, wie Objekte vom Spieler in andere Objekte überführt werden können.

Ein Beispiel

Objekte

- See
- Eimer
- EimerWasser
- brennenderBaum
- verkohlterBaum

Mutationsregeln

- (Eimer, See) → (EimerWasser, See)
- (brennenderBaum, EimerWasser) → (verkohlterBaum, Eimer)

Ausgangszustand

- Feld (0, 0) hat brennenderBaum
- Feld (1, 0) hat Eimer und See

Zielzustand

- Feld (0, 0) hat verkohlterBaum

Mögliche Folge von Spielzüge zum Erreichen des Zielzustands

- gehe zu Feld (1, 0)
- nehme Eimer
- kombiniere Eimer und See
- gehe zu Feld (0, 0)
- kombiniere EimerWasser mit brennenderBaum

Aufgabenstellung

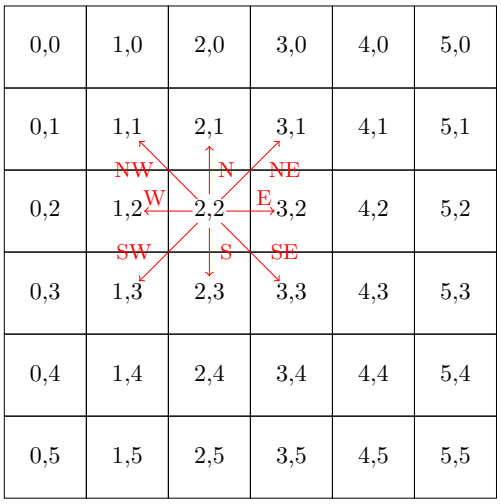
Im Folgenden werden die unterschiedlichen Bestandteile des Frameworks und die Spielmechanik vorgestellt.

1 Spielfeld

Das Spielfeld ist zweidimensional und besteht ähnlich wie ein Schachbrett aus viereckigen Feldern. Durch die Struktur des Spielfelds kann die Spielfigur von jedem Feld, das nicht ein Randfeld ist, auf acht direkt benachbarte Felder laufen: NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST und NORTHWEST.

Das nordwestlichste Feld hat die Nummer (0, 0). Das Feld (1, 0) liegt östlich von dem Feld (0, 0). Das Feld (0, 1) liegt südlich von dem Feld (0, 0). Das Feld (1, 1) liegt südöstlich von dem Feld (0, 0). Alle weiteren Felder liegen entsprechend, wie auch in Abbildung 1 zu sehen ist.

Abbildung 1: Beispiel eines 6x6-Spielbrett und der möglichen Zugrichtungen



In Abbildung 1 sind exemplarisch die Nachbarschaftsbeziehungen für das Feld (2, 2) angegeben. Machen Sie sich mit den unterschiedlichen Nachbarschaftsbe-

ziehungen vertraut und erarbeiten Sie sich hieraus möglichst einfache Regeln für die Umrechnung der Nachbarschaftsbeziehungen. Setzen Sie dies daraufhin in Ihrer Implementierung um, indem Sie die Umrechnung in einer eigenen Klasse kapseln.

2 Spielmechanik

Erzeugt wird ein Spielfeld F mit Spielfeldern F_{ij}. Jedes Spielfeld F_{ij} hat eine Menge von auf ihm befindlichen Objekten. Die Menge der Objekte ist nicht begrenzt. Der Spieler befindet sich zu jedem Zeitpunkt auf genau einem Spielfeld und führt eine Menge von Objekten in seinem Inventar mit sich. Auch das Inventar hat keine maximale Größe. Es gibt zwei Grundarten von Objekten, das sind *Scenery-Objekte* und *Items*. Während *Scenery-Objekte* nur auf Spielfeldern auftauchen können, dürfen *Items* ins Inventar aufgenommen werden.

Das Framework soll es erlauben, verschiedene sgn. Objekt-Typen zu definieren. Allen Objekten, die in der Spielwelt auftauchen, soll ein solcher Typ zugeordnet werden können. Es kann mehrere Objekte des gleichen Typs geben. Wie der Spieler mit diesen Objekten in Aktion treten kann, wird über verschiedene Arten von Mutationsregeln definiert. Die Menge der Mutationsregeln ist nicht limitiert. Sei T die Menge aller Typen, dann können wir die Mutationsregeln mithilfe der folgenden Relationen beschreiben:

- Komposition: $T\ x\ T \rightarrow T$
- Dekomposition: $T \rightarrow T\ x\ T$
- Transformation: $T\ x\ T \rightarrow T\ x\ T$

Bei der Komposition werden zwei Objekte zu einem neuen Objekt kombiniert. Beispiele für eine Kompositionsregeln:

- (Wasser, Kohlensäure) → Sprudel
- (Kohlensäure, Wasser) → Sprudel
- (brennenderBaum, Wasser) → verkohlterBaum

Die Dekomposition funktioniert genau umgekehrt und zerlegt ein Objekt in zwei neue Objekte. Beispiele für Dekompositionsregeln:

- Baum → (Wurzel, Holzlatten)
- vollerKoffer → (Kleider, leererKoffer)

Die Transformation überführt zwei Objekte in zwei neue Objekte. Beispiele für Transformationsregeln:

- (Eimer, See) → (EimerWasser, See)
- (See, Eimer) → (EimerWasser, See)
- (Wasser, Wasserkocher) → (kochendesWasser, Wasserkocher)

Der Spieler kann die unterschiedlichen Regeln nur auf Objekte in seiner Reichweite anwenden. Dazu zählen nur Objekte auf dem Feld, auf dem er sich gerade befindet, sowie Objekte, die sich in seinem Inventar befinden.

3 Schnittstellen

Wir stellen Ihnen verschiedene Interfaces im Paket *ias* zur Verfügung, die Sie implementieren müssen. Dabei dürfen Sie Ihre eigene Implementierung ausschließlich im Paket *student* erstellen. Das Paket *student* wird über das Git abgeben werden. Die Klassen und Interfaces des Pakets *ias* dürfen Sie **nicht** modifizieren. Hierzu wird eine entsprechende *.gitignore* Datei zur Verfügung gestellt.

3.1 Interface TextAdventure

Das Interface *TextAdventure*, dessen Methoden Sie implementieren müssen, ermöglicht es ein Text-Adventure-Game anzulegen. Im Paket *student* finden Sie eine Klasse *Main*, die das Interface *TextAdventure* nutzt, um verschiedene Spiele zu initialisieren. Diese können Sie nach erfolgreicher Implementierung der Interfaces durchspielen. Die Spielszenarien sollen Ihnen beim ausgiebigen Testen Ihres Codes helfen. Für die Interaktion mit dem Spiel stehen Ihnen alle Methoden des Interface *Player* zur Verfügung. Schauen Sie sich hierzu auch die Klasse *GameStarter* an. In dieser ist die Interaktion mit dem Spieler implementiert. Unsere automatisierten Tests benutzen ebenfalls das Interface *TextAdventure* um Test-Spielszenarios zu erstellen.

Das Interface *TextAdventure* bietet verschiedene Methoden, um ein neues Spiel zu erzeugen. Überlegen Sie sich für jede Methode, in welchen Fällen ihre Ausführung fehlschlagen kann. Werfen Sie in diesen Fällen eine *TextAdventureException*. Diese wird Ihnen ebenfalls im Paket *ias* zur Verfügung gestellt. Sobald der gewünschte Ausgangszustand hergestellt ist, kann mit *startGame* ein Spiel gestartet werden.

Erzeugen von Objekt-Typen

- *addItemType(String **id**, String **description**)*
- *addSceneryType(String **id**, String **description**)*

Die Methoden *addItemType* und *addSceneryType* erstellen neue Objekt-Typen. Sie erhalten beide als Argumente je einen Typ-Bezeichner **id** und eine Typ-Beschreibung **description**. Für den Typ-Bezeichner gilt, dass dieser ausschließlich aus Buchstaben bestehen und nicht leer sein darf. Außerdem sind die Typ-Bezeichner, über die Menge aller Item- und Scenery-Typen hinweg, eindeutig. Beachten Sie des Weiteren die Regeln in Kapitel 2.

Erzeugen von Objekten

- *placeItem(String **type**, int **x**, int **y**)*

Die Methode *placeItem* platziert ein neues Objekt vom Typ **type**, wobei **type** identisch mit dem Typ-Bezeichner eines zuvor erstellten Objekt-Typs sein muss. Das neu erstellte Objekt wird direkt an der angegebenen Position (x, y) auf dem Spielfeld platziert. Es können beliebig viele Instanzen eines Objekttyps auf dem Spielfeld platziert werden und während des Spiels vom Spieler in das Inventar

aufgenommen werden. Bedenken Sie, dass nur zuvor erstellte Objekt-Typen platziert werden können.

Erzeugen von Mutationsregeln

- *addComposition(String* $\mathbf{in_1}$*, String* $\mathbf{in_2}$*, String* **out***, String* **description***)*
- *addDecomposition(String* **in***, String* $\mathbf{out_1}$*, String* $\mathbf{out_2}$*, String* **description***)*
- *addTransformation(String* $\mathbf{in_1}$*, String* $\mathbf{in_2}$*, String* $\mathbf{out_1}$*, String* $\mathbf{out_2}$*, String* **description***)*

Mit Hilfe der obigen Methoden werden die verschiedenen Arten von Mutationsregeln erstellt. Jede Mutationsregel erhält mit dem letzten Parameter *description* eine Tätigkeitsbeschreibung. Die Parameter *in*, in_1, in_2 und *out*, out_1 und out_2 entsprechen jeweils dem Typ-Bezeichner.

Die Methode *addComposition* legt eine neue Kompositionsregel der Form $(in_1, in_2) \rightarrow out$ an. Die Methode *addDecomposition* legt eine neue Dekompositionsregel der Form $in \rightarrow (out_1, out_2)$ an. Die Methode *addTransformation* legt eine neue Transformationsregel der Form $(in_1, in_2) \rightarrow (out_1, out_2)$.

Es darf für jedes Objekt-Paar höchstens eine gültige Komposition oder Transformation geben. Außerdem sind Transformationen und Kompositionen symmetrisch. Das bedeutet, dass für jede gültige Regel $(in_1, in_2) \rightarrow *$ auch die Regel $(in_2, in_1) \rightarrow *$ gültig ist. Des Weiteren darf es für jedes Objekt nur eine Dekompositionsregel geben, damit der Aufruf einer Dekomposition eindeutig ist.

Spiel starten

- *Player startGame(int* $\boldsymbol{x}$*, int* $\boldsymbol{y}$*)*

Die Methode *startGame* erstellt ein neues Spielfigur-Objekt. Die Spielfigur wird auf dem Feld (**x**, **y**) platziert. Wir benutzen die Methoden des Interfaces Player, um die Spielfigur zu steuern.

Name des Spielszenarios

- *String getName()*

Jedes Text-Adventure-Game benötigt einen eindeutigen und nicht leeren Namen. Die Methode *getName()* fungiert als Getter-Methode, um den Spielnamen auszulesen.

3.2 Interface Player

Mit den Methoden des Interface *Player* wird, sobald das eigentliche Spiel begonnen hat, die Spielfigur gesteuert. Das Interface ist so gestaltet, dass eine Kommandozeilenschnittstelle eine Reihe von Methoden mit *String*-Ein- und Ausgaben emuliert. Hiermit soll die Testbarkeit mit JUnit-Tests gewährleistet werden und Ihnen alle nötige Freiheit für Ihre Implementierung gegeben werden. Nutzen Sie daher dieses Interface nicht als Vorbild für Ihre eigenen Interfaces, sondern

als einfacheren Ersatz für die Kommandzeilenschnittstelle.

Die Parameter der Methoden der Klasse *Player* sind direkt aus der Spieleingabe übernommene *Strings*. Die Abbildung dieser Strings auf die von ihnen repräsentierten Objekte müssen Sie selbst vornehmen. Auch die Rückgabewerte sind für die direkte Interaktion mit dem Spieler gedacht, daher werfen die Methoden von *Player* auch keine *Exceptions*, sondern geben Fehlermeldungen oder Erfolgsmeldungen als *String* zurück. Verfassen Sie eigene Fehlermeldungen für alle Fälle, in denen die Eingabe des Benutzers nicht wie gewünscht umgesetzt werden kann. Achten Sie hierbei darauf, dass die Meldungen immer aussagekräftig sind und - zu Zwecken der Testbarkeit - mit dem Präfix "Sorry, "beginnen. Auf dieses Präfix wird getestet und Fehlermeldungen, die das Präfix nicht enthalten, werden als fehlende Fehlermeldungen gewertet.

Bewegungen

- *String go(String **direction**)*

Der Befehl *go* bewegt die Spielfigur in die mit *direction* bezeichnete Richtung. Die Richtungen werden in englisch übergeben. Dabei nimmt direction einen der folgenden Werte an: "N", "NE", "E", "SE", "S", "SW", "W", "NW". Der Rückgabewert lautet im Erfolgsfall "You go "+ x. Dabei gilt:
$x \in \{"north", "northeast", "east", "southeast", "south", "southwest", "west", "northwest"\}$.

Status

- *String[] look()*
- *String[] inventory()*

Der Befehl *look* gibt ein Array von Objekten zurück, die sich auf dem aktuellen Spielfeld befinden. Der Befehl *inventory* gibt ein Array von Objekten, die sich im Inventar des Spielers befinden, zurück. Dabei steht jeder Arrayeintrag für ein Objekt und ist wie folgt aufgebaut: Typ-Bezeichner + "-"+ Typ-Beschreibung.

Aufnehmen und Ablegen

- *String take(String **item**)*
- *String drop(String **item**)*

Die Parameter dieser Befehle sind Typ-Bezeichner. Der Befehl *take* entfernt ein Objekt angegebenen Typs vom Spielfeld und fügt es dem Inventar hinzu. Bei erfolgreicher Ausführung lautet der Rückgabewert "You pick up the "+ item. Der Befehl drop entfernt ein Objekt angegebenen Typs aus dem Inventar und fügt es dem Spielfeld hinzu. Bei erfolgreicher Ausführung lautet der Rückgabewert "You drop the"+ item. Im Fehlerfall soll eine sinnvolle Fehlermeldung, beginnend mit dem Präfix "Sorry, "ausgegeben werden.

Mutieren

- *String convert(String* $\mathbf{item_1}$*, String* $\mathbf{item_2}$*)*
- *String decompose(String* **item***)*

Die Parameter dieser Befehle sind Typ-Bezeichner. Der Befehl *convert* entfernt je ein Objekt vom Typ $item_1$ und vom Typ $item_2$ vom Inventar oder Spielfeld. Bedenken Sie hierbei, dass $item_1$ und $item_2$ vom selben Typ sein können. Der Befehl *decompose* entfernt ein Objekt vom Typ *item* vom Inventar oder Spielfeld. Dabei wird jeweils zuerst im Inventar gesucht und danach auf dem Spielfeld. Dann wird die passende Transformations-, Kompositions- oder Dekompositionsregel benutzt, und es werden neue Objekte vom Ergebnistyp der Mutation erstellt. Diese müssen so verteilt werden, dass Items automatisch im Inventar und Scenery-Objekte automatisch auf dem Spielfeld landen. Im Erfolgsfall wird die Beschreibung der angewendeten Mutation zurückgegeben. Der Befehl *convert* soll entweder eine Kompositionsregel oder Transformationsregel auslösen, falls eine passende Regel vorhanden ist.

3.3 Interface Terminal

Mit den Methoden des Interface *Terminal* wird der Spieler zu einer Eingabe aufgefordert, die anschließend eingelesen wird. In der Klasse *GameStarter* wird das Interface *Terminal* zur Interaktion mit dem Spieler genutzt.

- *void prompt(String* ***input****)*
- *String[] readInput()*

Die Methode *prompt* gibt den übergebenen Text auf dem Terminal aus. Somit kann der Spieler zu einer Eingabe aufgefordert werden. Die Methode *readInput* liest den Nutzerinput vom Terminal ein. Die Eingabe soll bei Leerzeichen getrennt und die Bestandteile als *String* Array zurückgegeben werden.

3.4 Klasse Factory

Zuletzt müssen Sie die Methode *getGame(**width**, **height**)* und die Methode *getTerminal()* der Klasse *Factory* implementieren. Die Methode *getGame(**width**, **height**)* erzeugt ein neues, leeres Spielfeld und gibt dieses als Rückgabewert zurück. Die Parameter width und height geben hierbei die Größe des zu erstellenden Spielfeldes an. Die Methode *getTerminal()* erzeugt ein neues Terminal und gibt dieses als Rückgabewert zurück. Dieses wird genutzt, um User-Input über das Terminal einzulesen und Aufforderungen an den User auf dem Terminal auszugeben.

Die Klasse *Factory* dient als eindeutiger, minimaler Einstiegspunkt der automatisierten Tests in Ihre Abgabe. Obwohl ein Rahmen für diese Klasse von uns vorgegeben ist, liegt diese Klasse im Paket *student*. Diese soll als Teil des Pakets *student* von Ihnen mit abgegeben werden.