Prüfung
Programmieren 1
Wintersemester 2020/21

Institut für Anwendungssicherheit
Prof. Martin Johns

Rahmenbedingungen

Ablauf

- Prüfungsform: Erstellung und Dokumentation von Rechnerprogrammen
- Veröffentlichung der Aufgabenstellung: 10.02.2021
- Bearbeitungszeitraum: 10.02.2021 bis 03.03.2021
- Späteste Abgabemöglichkeit: 03.03.2021 23:59 Uhr

Überblick

Jeder der Punkte wird im Folgenden noch in einem separaten Abschnitt detaillierter beschrieben.

- **Einzelarbeit:** Die Bearbeitung erfolgt in Einzelarbeit! Keine Fragen an Kommilitonen, Freunde, Familie, niemanden. Keine Chatgruppen, Internetforen, Q&A Seiten, etc. Bei Verstoß riskieren Sie eine Exmatrikulation.
- **Fragen:** Inhaltliche Fragen zur Aufgabenstellung nur innerhalb der ersten Woche per Mail.
- **Abgabe:** Abgabe nur über unseren Gitea-Server, alles andere wird ignoriert.
- **Struktur:** Gradleprojekt, in dem Sie nur im *student* Ordner implementieren und nichts im Ordner *ias* ändern dürfen.
- **Bewertung:** Sie müssen mindestens 50 der insgesamt 100 Punkte erreichen um die Prüfung mit einer 4,0 zu bestehen.
- **Diverses:** Checkstyle einhalten, nur die erlaubten Pakete verwenden.

Einzelarbeit / Rechtliches

Mit dem Öffnen der Klausuraufgaben bestätigen Sie, dass Sie sich geistig und körperlich in der Lage befinden, die Prüfung abzulegen (d.h. prüffähig sind). Durch Einreichen ihres Rechnerprogrammes auf unserem Abgabeserver bestätigen Sie, dass Sie die oben bezeichnete Prüfung selbstständig und ohne unzulässige fremde Hilfe sowie ohne Heranziehung nicht zugelassener Hilfsmittel bearbeitet haben. Sie bestätigen, dass Ihnen bewusst ist, dass der Verstoß gegen prüfungsrechtliche Regelungen über die Täuschung bei der Erbringung von Prüfungsleistungen nach § 11 Abs. 4 APO als Täuschungsversuch gewertet wird und damit zum Nichtbestehen der Prüfung führt. In besonders schweren Fällen z.B. bei Plagiaten oder bei organisiertem Zusammenwirken mehrerer Personen kann der Prüfungsausschuss zusätzlich das endgültige Nichtbestehen der Prüfung und damit das Scheitern im Studiengang feststellen.

Fragen

Um Fragen und Missverständnisse bezüglich der Aufgabenstellung zu klären, nehmen wir innerhalb der ersten Woche Fragen an.

Ihre Fragen zur Aufgabenstellung müssen bis zum 17.02 um 10 Uhr an `m.musch@tu-braunschweig.de` mit dem Betreff *Prog1 Prüfung Frage* gesendet werden. Fragen die nach diesem Zeitpunkt eintreffen oder mit dem falschen Betreff werden ignoriert. Es werden nur inhaltliche Fragen und Missverständnisse geklärt, keine Lösungsansätze oder technischer Support gegeben.

Sie erhalten keine direkte Antwort auf Ihre Mail. Alle eingehenden Fragen werden zuerst gesammelt und dann spätestens am 18.02 für alle öffentlich schriftlich oder als Videoaufzeichung beantwortet. Die Details hierzu werden wir per StudIP-Ankündigung bekannt geben, Sie sollten daher während der Bearbeitungszeit regelmäßig prüfen, ob es neue Ankündigungen im Kurs gibt.

Sollten im Zuge der Beantwortung der Fragen auch eine Überarbeitung dieser Aufgabenstellung nötig sein, werden wir alle Änderungen im Text farblich markieren. Es werden natürlich keine grundlegenden Änderungen erwartet, lediglich Klarstellungen und Konkretisierungen.

Es finden keine Vorlesung, großen Übungen oder kleinen Übungen während des Bearbeitungszeitraums statt. Es gibt keine weiteren Fragemöglichkeiten außer der oben genannten. Kontaktieren Sie **nicht** Ihren Tutor oder ehemaligen Partner aus den Studienleistungen.

Abgabe

Der Code zur Hausarbeit wird Ihnen in einem Git-Repository unter `https://programming.ias.cs.tu-bs.de/` zur Verfügung gestellt. Die Anmeldung erfolgt analog zum Übungsbetrieb mit Ihrer y-Nummer und dem zugehörigen SSO Passwort. Erstellen Sie sich hiervon eine lokale Kopie mittels *git clone* zur Bearbeitung der Hausarbeit. Denke Sie daran, dass nur Dateien bewertet werden können, die Sie anschließend per *git push* hochladen.

Sollten Sie technische Probleme bei der Anmeldung oder mit der Erreichbarkeit des Servers haben, wenden Sie sich per E-Mail an `m.musch@tu-braunschweig.de` mit dem Betreff *Prog1 Prüfung Support*. Fragen zur regulären Nutzung von Git und Gitea werden wir **nicht** beantworten.

Abgaben werden analog zu der Studienleistung **ausschließlich** in Ihrem persönlichen Git Repository akzeptiert. Für die Prüfung erhalten Sie von uns ein neues Repository mit dem Namen **Exam/yNummer**, auf das nur Sie selbst Zugriff haben. Gewertet wird, was am Ende des Bearbeitungszeitraums in dem **master** branch dieses Repositories bei uns auf dem Server vorliegt. Abgaben per E-Mail werden nicht gewertet. Andere branches können Sie zum Testen und Entwickeln nutzen, werden für die Bewertung allerdings ignoriert. Bearbeiten Sie die Prüfung auf keinen Fall im Repository der Hausaufgaben, auf das auch noch Ihr Gruppenpartner Zugriff hat.

Wir empfehlen Ihnen regelmäßig einen kompilierenden Zwischenstand hochzuladen. Wir können nicht garantieren, dass der Server dauerhaft erreichbar ist, falls hunderte Teilnehmer gleichzeitig spät Nachts am letzten Tag versuchen abzugeben. Außerdem können Sie so problemlos von einem anderen Gerät aus weiterarbeiten, falls ihr Computer während der Bearbeitungszeit ausfällt.

Weiterhin empfehlen wir ganz am Schluss Ihr Repository noch einmal in einen anderen Ordner frisch vom Server zu clonen und erneut die Tests auszuführen. So können Sie sich sicher sein, dass Sie auch wirklich alle relevanten Dateien abgegeben haben.

Achtung: Sollten Sie im Bearbeitungszeitraum nicht mindestens einen Commit pushen, dann gilt die Prüfung als **nicht angetreten**! Ihnen entstehen dann diverse Nachteile bezüglich Freiversuch und mündlicher Ergänzungsprüfung. Stellen Sie also sicher, dass Sie zumindest Ihre persönlichen Daten in die *student.txt* eintragen, damit Sie die Prüfung auch angetreten haben.

Struktur

Der von uns zur Verfügung gestellte Code ist als *Gradle*-Projekt strukturiert. Dies erlaubt es Ihnen direkt die mitgelieferten Tests auszuführen. Gradle war Teil der großen Übung und der Studienleistung im WS 2020/21, schauen Sie sich dazu ggf. die Aufzeichnungen der großen Übung an falls Ihnen das Programm noch nicht vertraut ist. In der mitgelieferten *README.md* finden Sie diverse hilfreiche Befehle, die erklären wie Tests, der Checkstyle oder die Main-Methode gestartet werden können.

Implementieren Sie Ihre Lösung ausschließlich im Paket **student** (*src/main/java/student*). Das Paket **ias** (*src/main/java/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. Die vorgegebene Struktur, die Ordnernamen und die Einstellungen des Gradle-Projektes (*gradle.build*) dürfen ebenfalls **nicht** verändert werden.

Sie erhalten von uns eine Reihe von Tests (*src/test/java/ias*) die Sie bei der Bearbeitung der Aufgabe unterstützen. Die vorgegeben Tests dürfen nicht verändert werden, Sie können aber weitere, eigene Tests im Order *src/test/java/student* hinzufügen. Mit der Datei *MyTests.java* haben wir Ihnen dort ein Beispiel hinterlegt. Manche der Tests verwenden mitgelieferte *.game* Beispieldateien als Eingabe. Die von uns mitgelieferten Beispieldateien im Order *src/test/resources/* dürfen nicht verändert werden, Sie dürfen dort aber neue Dateien für ihre eigenen Testfälle erstellen.

In Ihrem Projekt finden Sie eine Datei *student.txt*, die Sie ausfüllen müssen. Ohne diese Informationen können wir Ihre Note nicht an Ihr Prüfungsamt melden.

Bewertung

Insgesamt können 100 Punkte bei der Bearbeitung der Prüfung erreicht werden. Diese teilen sich auf in 90 Punkte für den Inhalt und 10 Punkte für die Form.

Die 90 Punkte für den Inhalt entsprechen 90 Unittests, wobei Sie für jeden erfolgreichen Test genau einen Punkt erhalten (es gibt keine halben Punkte). Das heißt, neben den 39 mitgelieferten *öffentlichen* Tests haben wir noch 51 weitere *private* Tests, deren Inhalt Ihnen nicht bekannt ist. Diese Tests überprüfen ob Ihr Programm der Spezifikation dieser Aufgabenbeschreibung entspricht.

Die 10 Punkte auf die Form erhalten Sie, wenn Sie alle Checkstyle Regeln einhalten. Die Punkte gibt es nur für einen komplett fehlerfreien Checkstyle; es gibt keine Teilpunkte!

Zum Bestehen müssen Sie mindestens 50 Punkte erreichen. Das heißt, dass eine erfolgreiche Bearbeitung der öffentlichen Tests und ein fehlerfreier Checkstyle noch **nicht** zum Bestehen der Prüfung ausreicht. Testen Sie deshalb Ihren Code ausgiebig auf die in der Aufgabenstellung beschriebene Funktionalität. Wir empfehlend dringend die Implementierung von zusätzlichen eigenen Tests. Versuchen Sie hierbei auch Rand- und Spezialfälle zu testen.

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, wird mit 0 Punkten bewertet! Nach der Abgabefrist führen wir automatisiert alle Tests für Ihr Programm aus. Sollte Ihr Programm nicht kompilierbar/ausführbar sein, werden wir **keine** manuellen Änderungen vornehmen um Ihr Programm nachträglich noch lauffähig zu machen. Dabei ist insbesondere wichtig, dass der Befehl *./gradlew test* Ihr Programm erfolgreich kompilieren und starten kann, da sonst alle Tests fehlschlagen werden.

Diverses

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*. Der Befehl *./gradlew build* führt auch den Checkstyle aus.

Neben dem von uns vorgegeben *ias* Paket dürfen Sie für Ihre Implementierung alle Klassen aus den Paketen *java.lang*, *java.io*, *java.nio*, *java.util* und *org.junit* verwenden. Andere Pakete dürfen **nicht** genutzt werden.

Wir verwenden einen Linux-Server zum Ausführen der Tests. Da es insbesondere wegen unterschiedlichem Verhalten von Dateisystemen zu Problemen kommen kann, wenn Sie Ihre Lösung lediglich unter Windows testen, empfehlen wir dringend Ihr Programm auch unter Linux zu testen. Dafür bietet sich z.B. eine Virtuelle Maschine oder das Windows Subsystem for Linux (WSL) an.

Zusammenfassung

Neben offensichtlichen Gründen, wie zu wenig erreichten Punkten, fallen Sie u.A. auch durch, wenn:

- Sie die Prüfung nicht komplett eigenständig bearbeiten
- Sie Dateien im Paket *ias* ändern
- Sie die vorgegebene Ordnerstruktur ändern
- Sie nicht im *master* Branch des Prüfungsrepositories abgeben
- Sie Klassen aus Paketen importieren, die nicht oben explizit erlaubt wurden
- Ihre Lösung bei uns nicht kompiliert (z.B. weil Sie nicht alle Dateien gepusht haben)
- Ihre Lösung auf unserem Linux Server abstürzt (z.B. wegen hardkodierten Pfaden zu Dateien)
- der Befehl *./gradlew test* die Tests nicht ausführen kann

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 (Abschreibender und Herausgebender) gemeldet!

Aufgabenstellung

Im Folgenden werden die unterschiedlichen Bestandteile des Frameworks und die Spielmechanik vorgestellt. Alle hier vorgestellten Dateien finden Sie im *ias* Paket Ihres Git-Repositories. Kopieren Sie also **nicht** den Code aus der PDF, die Codeausschnitte dienen nur der Veranschaulichung.

Lesen Sie die folgenden Beschreibungen genau durch und schauen Sie sich parallel die beigelegten öffentlichen Tests zu den Methoden an. Die textuellen Beschreibungen in deutscher Sprache sind meistens nicht so präzise wie der Code eines Java-Programms. Viele Missverständnisse und Unklarheiten sollten sich daher durch aufmerksames Lesen der beigelegten Tests aufklären. Um Missverständnisse zu vermeiden, haben wir außerdem bewusst auf eine JavaDoc der Interfaces verzichtet, so dass es nicht zwei (potentiell widersprüchliche oder leicht unterschiedliche) Beschreibungen der selben Methode gibt.

Zu jeder Methode finden Sie in dieser PDF ein oder mehrere *minimale*, beispielhafte Aufrufe. Diese bauen aufeinander auf und können daher nicht isoliert betrachtet werden. Die Beispiele dienen allerdings lediglich der Veranschaulichung! Wenn Ihr Programm lediglich mit den hier beschriebenen Beispielen funktioniert reicht dies **noch nicht** zum Bestehen der Prüfung.

In diesem Projekt sollen Sie Kartenspiele abbilden und die entsprechenden Spielregeln auf eine Auswahl von Karten anwenden. Ihr Programm soll dabei nicht auf ein bestimmtes Spiel spezialisiert sein, sondern viele verschiedene Spiele darstellen können. Aus allen existierenden Karten eines Spiels lassen sich Kartendecks erstellen, die eine Auswahl der Karten beinhalten, mit denen dann gespielt wird. Alle relevanten Klassen und Interfaces werden im Folgenden näher beschrieben.

1 GameException

Alle Tests, die Ihr Programm auf Verhalten bei fehlerhaften Eingaben testen, erwarten diese *GameException*. Werfen Sie nie explizit andere Exceptions außer dieser *GameException*.

Der Inhalt der Nachricht wird nicht bewertet, wir empfehlen allerdings sinnvolle Fehlermeldungen damit Sie sich selbst das Debugging erleichtern. Sie müssen mit dieser Klasse nichts machen, außer sie an den richtigen Stellen als Exception zu werfen.

```java
public final class GameException extends Exception {

    public static final String ERROR = "Error! ";

    public GameException(String message) {
            super(ERROR + message);
    }
}
```

Abbildung 1: GameException.java

2 Game

Dieses Interface enthält die grundlegende Funktionalität zur Erstellung eines Kartenspiels, wie z.B. Karten und Regeln definieren.

Sie müssen alle Methoden des Interfaces implementieren und wir empfehlen hiermit zu starten, da das *Deck*-Interface an vielen Stellen ein funktionierendes *Game* voraussetzt. Beachten Sie auch, dass Ihre Implementierung des *Game*-Interfaces zusätzlich auch alle Methoden der *Factory* enthalten muss (siehe nächster Abschnitt).

```java
public interface Game {
    public void defineCard(String name) throws GameException;

    public void defineProperty(String name, String type) throws GameException;
    public void setProperty(String cardName, String propertyName, String value) throws
    ↪  GameException;
    public void setProperty(String cardName, String propertyName, int value) throws GameException;

    public void defineRule(String propertyName, String operation) throws GameException;
    public void defineRule(String propertyName, String winningName, String losingName) throws
    ↪  GameException;

    public String[] get(String type, String name) throws GameException;

    public void saveToFile(String path) throws GameException;

    public Deck createDeck();
}
```

Abbildung 2: Game.java

2.1 defineCard

Definiert eine neue Karte, die danach im Spiel verwendet werden kann. Die Methode sorgt also lediglich dafür, dass der Name der Karte danach von anderen Methoden verwendet werden kann. Auch wenn ein Deck später beliebig viele Karten mit dem gleichen Namen haben kann, so kann eine Karte nur einmal *definiert* werden.

Beispiel: *defineCard(“Emrakul”)*

2.2 defineProperty

Definiert eine neue Eigenschaft mit dem angegebenen Namen. Ohne solche Eigenschaften wäre das Spiel nicht spielbar und es könnten keine Regeln angelegt werden. Die Eigenschaften unterscheiden sich in zwei Typen: Solche, die eine reine Zahl darstellen, z.B. die Angriffskraft einer Karte, und solche, die einen beliebigen Text haben können, z.B. die Rasse und Klasse einer Karte. Der Parameter *type* muss daher immer *“string”* oder *“integer”* sein. Der Name der Eigenschaft darf nur einmal vergeben werden.

Beispiel: *defineProperty(“power”, “integer”)*
Beispiel: *defineProperty(“type”, “string”)*

2.3 setProperty

Fügt eine vorher angelegte Eigenschaft einer vorher angelegten Karte hinzu und setzt dabei den Wert. Ist der übergebene *value* ein String, muss der *propertyName* also auf eine Property verweisen, die mit Typ “string” angelegt wurde. Das selbe gilt äquivalent für *values* die ein Integer sind.

Beispiel: *setProperty("Emrakul", "power", 15)*
Beispiel: *setProperty("Emrakul", "type", "Eldrazi")*

Karten müssen nicht alle vorher definierten Eigenschaften haben. Mehrmals *setProperty* mit der selben Karten- und Eigenschaftsnamen Kombination aufzurufen ist nicht erlaubt.

2.4 defineRule

Definiert eine Regel für eine vorher angelegte Eigenschaft. Hier wird also festgelegt, welche Werte einer Eigenschaft im direkten Vergleich gewinnen. Für Eigenschaften vom Typ Integer gewinnt logischerweise entweder der höhere oder der niedrigere Wert. Der Parameter *operation* muss daher immer *">"* oder *"<"* sein.

Beispiel: *defineRule("power", ">")*

Für Eigenschaften vom Typ String ist das hingegen nicht so pauschal zu beantworten. Stattdessen müssen in den Parametern *winningName* und *losingName* zwei Werte referenziert werden, wobei der Erste dann den Zweiten schlägt.

Beispiel: *defineRule("type", "Eldrazi", "Human")*

Es muss nicht für alle vorher angelegten Eigenschaften eine Regel geben. Bei Integer-Regeln sind keine Widersprüche erlaubt, die selbe Eigenschaft darf nicht mit größer und kleiner definiert werden. Bei String-Regeln sind auch gegensätzliche Regeln erlaubt, etwa Eldrazi schlägt Human und Human schlägt Eldrazi. Doppelte, identische Regeln sind nicht erlaubt. Eine Karte kann sich außerdem nie selbst schlagen. Es können auch nur für einen Teil der Karten String-Regeln angelegt werden.

2.5 get

Diese Methode wird hauptsächlich für die Tests benötigt, um die vorher definierten Karten, Eigenschaften und Regeln abzufragen. Hintergrund ist, dass wir so eine partielle Bearbeitung der Aufgaben bewerten können, ohne dass alle anderen Methoden funktionieren müssen (wie das Speichern von Dateien, das bald dazukommt). Wenn Sie diese *get* Methode nicht implementieren, können wir also nicht überprüfen, ob die bisher oben beschriebenen Methoden auch korrekt umgesetzt wurden! Sie können diese *get* Methode auch in Ihrem internen Programmablauf verwenden, müssen das aber nicht.

Der Parameter *type* ist immer entweder *"game"*, *"card"*, *"property"*, oder *"rule"*. Der Parameter *name* ist ein beliebiger String.

Existiert ein Objekt mit dem entsprechenden Namen und Typ, so wird genau dieses eine Ergebnis in einem Array erwartet.

Beispiel: *get("card", "Emrakul")* ergibt ["Emrakul"]
Beispiel: *get("property", "power")* ergibt ["power"]

Wurde keine Karte/Eigenschaft/Regel mit dem entsprechenden Namen und Typ definiert, so wird ein leeres Array erwartet.

Beispiel: *get("card", "Ulamog")* ergibt []
Beispiel: *get("property", "Emrakul")* ergibt []

Als dritte Möglichkeit kann statt eines konkreten Namens der String "*" als Platzhalter übergeben werden. Dann sollen alle definierten Namen des angegebenen Types zurückgegeben werden. Die erwartete Stringdarstellung von Regeln wird durch das Beispiel veranschaulicht. Eine partielle Su-

che mit Platzhalter wie etwa “Emra*” ist **nicht** verlangt.

Beispiel: *get(“card”, “*”)* ergibt [“Emrakul”]
Beispiel: *get(“rule”, “*”)* ergibt [“power>”, “type:Eldrazi>Human”]

Wird der Name des Spiels mit *“game”* abgefragt, soll der *name* Parameter ignoriert werden. Der Spielname wird im nächsten Kapitel erklärt.

2.6 saveToFile

Diese Methode dient dazu, das angelegte Spiel zu *serialisieren*. Das heißt, alle definierten Karten, Eigenschaften und Regeln sollen in einer Text-Datei gespeichert werden. Der Parameter *path* enthält den Pfad und damit auch den Namen der Datei, die erstellt/überschrieben werden soll.

Wird die Methode aufgerufen nachdem alle bisherigen Beispiel-Befehle ausgeführt wurden, soll die zu erstellende Datei wie folgt aussehen. Die erste Zeile mit *Game: Magic* wird im nächsten Kapitel erklärt, alle anderen Zeilen sind lediglich eine textuelle Darstellung der bisher definierten Objekte. Die Reihenfolge der verschiedenen “Arten” von Zeilen soll die Bedingungen der äquivalenten Methoden einhalten. So müssen zuerst die *Card* und *Property* Zeilen geschrieben werden, bevor der Karte mit *CardProperty* ein Wert zugewiesen wird. Das heißt, alle Daten die in einer Zeile referenziert werden, müssen vorher definiert worden sein.

Die Formatierung innerhalb einer Zeile muss exakt so übernommen werden, insbesondere muss als Trennzeichen | verwendet werden, immer genau ein Leerzeichen (keine Tabulatoren) verwendet werden und es darf auch keine weiteren Zeichen am Ende der Zeile geben.

Beispiel: *saveToFile(/tmp/basic.game”)*

```
Game: Magic
Card: Emrakul
Property: power | integer
Property: type | string
CardProperty: Emrakul | power | 15
CardProperty: Emrakul | type | Eldrazi
GameRuleInteger: power | >
GameRuleString: type | Eldrazi | Human
```

Abbildung 3: /tmp/basic.game

2.7 createDeck

Erstellt eine neue Instanz eines Decks. Diese Methode dient lediglich dazu, eine Referenz zu erhalten und muss keine weitere Funktionalität bieten. Es können mehrere verschiedene Decks für das selbe *Game* erstellt werden. Das *Deck*-Interface wird im Anschluss an die *Factory*-Klasse erklärt.

Beispiel: *createDeck()*

3 Factory

Die Factory erstellt eine neue Instanz der Game-Klasse. Da Sie diese Klasse (wie auch alle anderen Dateien im *ias* Paket) nicht ändern dürfen, ist der Klassenname *MyGame* für Ihre Implementierung des *Game*-Interfaces fest vorgegeben. Sie müssen in Ihrer *MyGame*-Klasse alle Methoden implementieren, die diese Factory aufruft.

```
public final class Factory {
    public static Game createGame(String name) throws GameException {
        return new MyGame(name);
    }

    public static Game loadGame(String path) throws GameException {
        return MyGame.loadGame(path);
    }
}
```

Abbildung 4: Factory.java

3.1 createGame

Erstellt eine neue Instanz eines Spiel mit dem angegebenen Namen. Dabei handelt es sich um ein komplett neues Spiel, ohne bestehende Karten, Eigenschaften, Regeln, etc. Diese Methode dient hauptsächlich dazu, eine Referenz auf Ihre Implementierung des *Game*-Interfaces zu erhalten. Der Name wird erst relevant falls das Spiel gespeichert wird, dann soll die erste Zeile mit der *Game:* Definition diesen enthalten.

Beispiel: *createGame("Magic")*

3.2 loadGame

Anstatt ein komplett neues Spiel anzulegen wird hiermit ein vorher gespeichertes Spiel geladen. Der Parameter *path* enthält den Pfad zur Datei. Beim Laden eines Spieles gelten alle bisher erklärten Regeln: So dürfen keine Karten/Eigenschaften referenziert werden die (noch) nicht existieren, keine Karten dürfen doppelt definiert werden, etc. Außerdem darf es nur genau eine *Game:* Definition geben direkt in der ersten Zeile.

Durch Aufruf dieser Methode wird ein komplett initialisiertes Spiel zurückgegeben, das alle in der geladenen Datei enthalten Objekte enthält. Tritt an irgendeiner Stelle während des Ladens ein Fehler auf, so wird der gesamte Vorgang abgebrochen.

Beispiel: *loadGame("/tmp/basic.game")*

4 Deck

Bisher können lediglich Karten, Eigenschaften und Regeln definiert und diese gespeichert und geladen werden. Um diese dann auch anwenden zu können, müssen Sie zuletzt noch das Interface *Deck* implementieren. Es können mehrere verschiedene Decks für das selbe *Game* existieren.

Die Beschreibung dieser Klasse ist absichtlich etwas knapper gehalten. Wenn Sie mit Ihrer Implementierung bis an diesen Punkt gekommen sind, dann erwarten wir, dass Sie sich genauer mit den beigefügten Tests beschäftigen und damit Rückschlüsse auf das gewünschte Verhalten dieses Interfaces ziehen.

```
public interface Deck {
    public void addCard(String cardName) throws GameException;
    public String[] getAllCards();
    public String[] getMatchingCards(String propertyName, int value) throws GameException;
    public String[] getMatchingCards(String propertyName, String value) throws GameException;
    public String[] selectBeatingCards(String opponentCard) throws GameException;
}
```

Abbildung 5: Deck.java

4.1 addCard

Fügt eine vorher im Spiel definierte Karte dem Deck hinzu. Ein Deck kann beliebig viele unterschiedliche Karten enthalten und die selbe Karte kann in einem Deck beliebig oft vorkommen.

Beispiel: *deck1.addCard("Emrakul"); deck1.addCard("Emrakul")*
Beispiel: *deck2.addCard("Emrakul")*

4.2 getAllCards

Gibt alle Karten im Deck zurück, ohne Möglichkeiten die Suche einzuschränken. Leere Decks erzeugen ein leeres Array.

Beispiel: *deck1.getAllCards()* gibt ["Emrakul", "Emrakul"] zurück
Beispiel: *deck2.getAllCards()* gibt ["Emrakul"] zurück

4.3 getMatchingCards

Gibt alle Karten im Deck zurück, die den Suchparametern entsprechen. *propertyName* ist eine vorher im Spiel definierte Eigenschaft, der *value* Parameter entweder ein String oder Integer. Ähnlich wie bei *Game.defineRule*, muss der übergebene *value* Parameter zum vorher angelegten Typ der Property passen (String value zu string Property, Integer value zu integer Property).

Beispiel: *deck1.getMatchingCards("power", 15)* gibt ["Emrakul", "Emrakul"] zurück
Beispiel: *deck2.getMatchingCards("type", "Human")* gibt [] zurück

4.4 selectBeatingCards

Gegeben ein Deck und eine beliebige Karte *opponentCard*, findet alle Karten im Deck welche die Karte des Gegners schlagen. Eine Karte schlägt eine andere Karte, wenn sie von den die Karte betreffenden Regeln in mehr Fällen als der Gegner gewinnt. Sind drei der vorher definierten Regeln anwendbar und der Gegner gewinnt bei einer davon, so muss die Karte im Deck also die gegnerische Karte in mindestens zwei davon schlagen. Gewinnt beim Beispiel mit drei Regeln der Gegner für keine dieser Regeln, so reicht es auch wenn die eigene Karte bei einer Regel gewinnt. Sollte es

zu einem Unentschieden kommen, ist es keine beatingCard. Eine Karte kann sich nie selbst schlagen.

Für diese Methode haben wir als Einzige kein Beispiel hier im Dokument, da diese Beispiele sehr unübersichtlich werden. Schauen Sie sich hierfür stattdessen die veröffentlichten Tests inklusive der vordefinierten Spiele an.

5 Allgemeines / Ergänzende Hinweise

Es ist normal, dass das Projekt am Anfang nicht kompiliert. Sie müssen zuerst die fehlenden Klassen und Methoden zumindest als Platzhalter hinzufügen, bevor Sie die Tests starten können.

Das Spiel benötigt eigentlich keine Main-Methode, für die Bewertung testen wir die gesamte Funktionalität nur über Unit-Tests. Damit Sie trotzdem das Spiel einmal interaktiv ausprobieren können, haben wir eine *Main*-Methode geschrieben, welche die gesamte Funktionalität über die Kommandozeile zur Verfügung stellt. Diese Main-Methode überprüft außerdem beim Start ob Sie eine der wichtigen Dateien verändert haben, die Sie nicht ändern dürfen und beendet die Ausführung ggf. vorzeitig. Den passenden Befehl um die Main-Methode zu starten finden Sie in der *README.md* Datei.

Die insgesamt 90 Unittests verteilen sich auf die vorgegebenen Klassen und Interfaces wie folgt:

- Factory: 22 Tests (10 öffentliche, 12 private)
- Game: 42 Tests (20 öffentliche, 22 private)
- Deck: 21 Tests (7 öffentliche, 14 private)
- End2End: 5 Tests (2 öffentliche, 3 private)

End2End-Tests sind Tests welche die vielen verschiedenen Funktionalitäten in einem Test kombinieren. Diese testen also nicht eine einzelne Klasse separat, sondern die Kombination von mehreren Klassen und simulieren damit ein größeres Szenario.

Für **alle** String-Argumente **aller** Methoden gilt, dass *null* oder ein *leerer String* keine validen Eingaben sind. Hierfür sind Sie verantwortlich, dass Ihr Programm bei solchen Eingaben eine Exception wirft. Alle Strings sind *case-sensitive*, *Emrakul* und *emrakul* sind also zwei verschiedene Karten.

Für alle Strings sind die Zeichen **A-Z**, **a-z**, **0-9**, sowie Leerzeichen erlaubt. Insbesondere nicht erlaubt sind *Zeilenumbrüche* und das Trennzeichen |. Darauf müssen Sie **nicht** testen, auch die privaten Tests verwenden in Strings nur valide Zeichen.

Alle Methoden, die einen Karte/Eigenschaft/Regel referenzieren sollen fehlschlagen wenn dieses Objekt noch nicht existiert – außer in der Beschreibung der Methode wurde explizit ein anderes Verhalten gefordert. Das heißt die Reihenfolge der Methodenaufrufe beim definieren des Spiels ist wichtig, z.B. können Sie nicht *setProperty* mit einem Kartennamen aufrufen für eine Karte die erst in der nächsten Zeile angelegt wird.

Wenn als Rückgabewert ein Array verlangt ist, dann spielt die Reihenfolge der Elemente in diesem Array **keine** Rolle. Wichtig ist aber, dass Ihr Array die **richtige Länge** hat. Wenn wir ein String-Array mit zwei Elementen erwarten und Sie ein String-Array mit den gesuchten zwei Elementen, aber weiteren Werten wie z.B. leeren Strings oder *null* zurückgeben, ist dies nicht korrekt!

Wir behalten keine alten Referenzen auf ein *Deck* wenn sich das *Game* ändert. Das heißt wenn das aktuelle Spiel durch Laden eines anderen Spiels überschrieben wird, greift kein Test mehr auf ein (mittlerweile ungültiges) Deck zu.

Viel Erfolg!