FernUniversität in Hagen

Begleitmaterial zum Kurs 1580/1582/1584

Aufgabenbeschreibung
für das
Java-Programmierpraktikum
im Wintersemester 2013/2014

Lehrgebiet Datenbanksysteme für neue Anwendungen

1 Das Programmierpraktikum

1.1 Ziele des Programmierpraktikums

Das Programmierpraktikum dient der Übung im praktischen Umgang mit einer gängigen Programmiersprache. Zur Zeit findet dazu die Programmiersprache *Java* Anwendung, die aufgrund ihrer Konzepte und insbesondere auch aufgrund ihrer Eignung für Internet-Anwendungen weithin Akzeptanz gefunden hat. Gute Java-Kenntnisse sind deshalb Voraussetzung für eine erfolgreiche Teilnahme an diesem Praktikum. Sie sollen lernen, die Lösung zu einem überschaubaren Problem zu finden und mit Hilfe der Programmiersprache Java zu implementieren.

Wie bereits in den letzten vom Lehrgebiet „Datenbanksysteme für neue Anwendungen“ durchgeführten Programmierpraktika können Sie je nach Interesse und Fähigkeiten zwischen verschiedenen Themen wählen. Diese sind im einzelnen:

- Kulami
- Nonogramme
- Vektorisierung von Bildern
- Morsezeichen

Die detaillierten Aufgabenbeschreibungen finden Sie in weiteren Dokumenten. Unsere Absicht ist es, mit diesen Aufgaben ein so breites Spektrum von Themen abzudecken, dass jeder etwas darunter finden kann, das seinen persönlichen Neigungen entspricht. Wir haben an manchen Stellen bewusst auf eine detaillierte Spezifikation des von Ihnen zu erstellenden Programms verzichtet, insbesondere in Bezug auf die graphische Benutzeroberfläche und die Interaktion von Programm und Benutzer oder Benutzerin, um Ihnen die Möglichkeit zu geben, selbst kreativ zu werden und eigene Konzepte zu realisieren. Wir hoffen, dass Ihnen dadurch die Arbeit an Ihrer Aufgabe noch mehr Spaß macht.

Die Aufgaben weisen einen unterschiedlichen Schwierigkeitsgrad auf, der auch von Ihrer individuellen Vorbildung abhängig ist. Wählen Sie Ihre Aufgabe sorgfältig aus; berücksichtigen Sie dabei auch den Aspekt der Implementierbarkeit im Zeitrahmen des Programmierpraktikums.

Beachten Sie, dass die gewählte Aufgabe von Ihnen allein zu lösen ist. Gruppenarbeiten sind nicht zulässig!

„Allein lösen“ bedeutet, dass Sie Ihre Lösung selbständig und individuell erarbeiten und ausführen sollen. Bei der Lösung von kleineren Problemen erwarten wir dagegen, dass

Sie sich gegenseitig unterstützen, etwa durch Teilnahme an den Diskussionen innerhalb der Newsgroups (siehe Abschnitt 1.3).

Hinweis: Falls Ihnen nur die schwarz-weiße Fassung des Begleitmaterials vorliegt, finden Sie eine farbige Fassung als PDF-Datei auf der Homepage des Lehrgebiets (http://www.fernuni-hagen.de/mathinf/studium/lehre/praktika/programmierpraktikum/2013_2014_ws.shtml).

1.2 Ablauf und Anforderungen des Programmierpraktikums

Das Praktikum wird mit 10 Creditpoints gewertet. Dies entspricht einem erwarteten Aufwand von etwa 300 Arbeitsstunden. Nach dem Praktikum werden Sie um Ihre Beteiligung an der Veranstaltungsevaluation gebeten. Dabei wird auch nach Ihrem tatsächlichen Arbeitsaufwand für das Praktikum gefragt. Es wäre daher gut, wenn Sie im Hinblick darauf Ihren Aufwand täglich oder wöchentlich notieren könnten.

Wir haben für das Programmierpraktikum folgenden verbindlichen Zeitplan erarbeitet:

Veröffentlichung der Aufgabenstellungen:	01.10.2013
Anmeldeschluss zum freiwilligen Präsenztag:	06.10.2013
Freiwilliger Präsenztag:	12.10.2013
Abgabe des Entwurfs:	17.11.2013
Abgabe der Lösung samt Dokumentation:	05.01.2014
Präsenzphase mit Präsentation der Lösung:	13.-15.02.2014

Bitte beachten Sie die angegebenen Fristen. **Bei Versäumnis einer Abgabefrist oder des vereinbarten Termins zur Präsentation Ihrer Lösung werden Sie das Praktikum nicht bestehen.**

1.2.1 Freiwilliger Präsenztag

Zum *freiwilligen* Präsenztag am 12.10.2013 stellen wir Ihnen die einzelnen Aufgaben in kurzen Präsentationen vor. Sie haben die Möglichkeit, Fragen direkt an einen Betreuer oder eine Betreuerin zu stellen.

Falls Sie sich schon zuvor bereits für ein Thema entscheiden können, müssen Sie selbstverständlich nicht an diesem freiwilligen Präsenztag teilnehmen.

1.2.2 Abgabe des Entwurfs

Um Ihre Bemühungen bereits frühzeitig in gelenkte Bahnen zu bringen, verlangen wir von Ihnen, dass Sie sich rechtzeitig Gedanken über die von Ihnen gewählte Aufgabe machen und das Programmierprojekt planen. Dies dokumentieren Sie in einem schriftlichen Entwurf mit einem Umfang von etwa 3-5 DIN-A4-Seiten. Wie Sie zu einem hoffentlich geeigneten objektorientierten Entwurf gelangen, beschreibt Abschnitt 2.2.

Zu diesem Entwurf gehören ein „Klassendiagramm“ sowie eine kurze Beschreibung der Funktionalität aller Klassen („Schnittstellenbeschreibung“). Außerdem sollen Sie die Beziehungen zwischen den Klassen erläutern.

Dieser Entwurf dient vor allem drei Zwecken: (1) Wir sehen, dass Sie sich mit dem Problem hinreichend auseinandergesetzt haben. (2) Sie machen sich klar, wie Sie die gewählte Aufgabe lösen wollen. (3) Im weiteren Verlauf des Praktikums haben Sie die Gelegenheit, die Umsetzbarkeit Ihres Entwurfs zu überprüfen, ihn ggf. zu korrigieren und die Notwendigkeit Ihrer Anpassungen im Abschlussbericht zu erläutern.

Wir denken, dass dieses Vorgehen den Lernprozess fördert, indem Sie dazu angehalten werden, Ihre Entwurfsentscheidungen zu reflektieren.

Senden Sie uns bitte Ihren Entwurf **in elektronischer Form als PDF-Dokument** rechtzeitig zu, spätestens am **17.11.2013** per E-Mail an propra@fernuni-hagen.de. **Nachdem Sie den Entwurf erstellt und abgesendet haben, sollen Sie keinesfalls auf eine Rückmeldung seitens der Kursbetreuer warten.** Diese erfolgt ausschließlich, falls bei der Durchsicht Ihres Entwurfs der Eindruck entsteht, dass dieser nicht zu einer brauchbaren Lösung der Aufgabe führen kann.

1.2.3 Abgabe der Lösung samt Dokumentation

Senden Sie Ihre Lösung spätestens am **05.01.2014** an propra@fernuni-hagen.de. Für das Format gilt das gleiche wie beim Entwurf. Zur Lösung gehören der Quellcode in komprimierter Form (als zip- oder rar-Datei), Javadoc-Dateien und ein jar-Archiv mit Ihrem Programm sowie Ihre Dokumentation.

Zu einer vollständigen Abgabe gehören also:

1. Die ausreichend kommentierten und javadoc-fähigen Quelldateien; beachten Sie insbesondere die Programmierrichtlinien (Abschnitt 3).
2. Die vollständige Dokumentation **in elektronischer Form als PDF-Datei**; beachten Sie hierzu die Ausführungen zur Dokumentation in Abschnitt 4.

3. Falls sich Ihre Programmarchitektur gegenüber dem von Ihnen eingesandten Entwurf verändert hat, beschreiben Sie **innerhalb** Ihrer Dokumentation zusätzlich kurz die Änderungen und die Gründe, aus denen Sie sie vorgenommen haben (ca. 1-3 DIN-A4-Seiten).

Nach Eingang der Abgaben werden wir diese sichten und bei erkennbaren schwerwiegenden Mängeln Kontakt mit Ihnen aufnehmen. In solchen Fällen geben wir Ihnen einmalig Gelegenheit, Ihre Abgabe noch einmal zu überarbeiten.

1.2.4 Präsenzphase mit Präsentation der Lösung

Im Zeitraum **13.-15.02.2014** wird eine zweite – diesmal verpflichtende – Präsenzphase durchgeführt. An **einem** dieser Tage müssen Sie in Hagen erscheinen, um mit einem der Betreuer oder Betreuerinnen Ihre Lösung durchzusprechen. Wir werden Ihre Lösung zuvor eingehend inspizieren und erwarten von Ihnen, dass Sie uns Auskunft zu Ihrem Programm, den angewendeten Verfahren und zum Entwicklungsprozess beantworten können. Wir werden Sie auf möglicherweise bestehende Probleme hinweisen. Letztendlich werden wir Ihnen mitteilen, ob Sie das Praktikum bestanden haben oder (noch) nicht. Falls wir nämlich heilbare Mängel an Ihrer Abgabe feststellen, werden wir Ihnen einmalig Gelegenheit geben, diese innerhalb einer gesetzten Frist zu beheben und dies bei einem weiteren individuell vereinbarten Präsenztermin in Hagen nachzuweisen.

Näheres zur Organisation dieser Präsenzphase gibt die Kursleitung rechtzeitig über die Praktikumswebseite bzw. die entsprechende Newsgroup bekannt (siehe Abschnitt 1.3).

1.3 Weitere Hinweise

Die offizielle Webseite zum Praktikum sollte Ihnen bereits bekannt sein (http://www.fernuni-hagen.de/mathinf/studium/lehre/praktika/programmierpraktikum/2013_2014_ws.shtml). Dort wird die Kursbetreuung offizielle Ankündigungen, Hinweise und Nachrichten veröffentlichen. Daher sollten Sie sich hier mindestens wöchentlich auf den neuesten Informationsstand bringen.

Weiterhin haben wir auf dem Newsserver news://feunews.fernuni-hagen.de/ der Fern-Universität zwei Newsgroups zum Programmierpraktikum eingerichtet:

`feu.informatik.kurs.1580+82+84.betreuung.ws`: Diese Newsgroup dient der Koordination von organisatorischen Aspekten des Praktikums, etwa der Veröffentlichung von offiziellen Mitteilungen der Kursbetreuung oder dem Austausch von Details zu den Präsenztagen.

`feu.informatik.kurs.1580+82+84.diskussion.ws`: Diese Newsgroup dient vor allem als Plattform für Ihren Austausch mit anderen Praktikumsteilnehmern und Praktikumsteilnehmerinnen. Hier sollen Sie Fragen wie etwa zur Java-Programmierung, zu Algorithmen und Datenstrukturen, zu den Aufgaben oder anderen Problemen *untereinander* klären. Die Kursbetreuung liest die Beiträge mit und wird Beiträge verfassen, wenn dies wirklich notwendig erscheint, etwa um auf grundlegend fehlerhafte Beiträge aufmerksam zu machen oder missverständliche Aufgabenstellungen klarzustellen.

Ferner können Sie jederzeit auch persönlich Kontakt zur Kursleitung aufnehmen. Unsere Adressen und Telefonnummern finden Sie auf der Praktikumswebseite.

2 Entwurf

Um bei der Programmerstellung nicht in Sackgassen zu geraten, sollte man sich *vorher* überlegen, welche Objekte (Klassen) und welche Beziehungen zwischen diesen Klassen benötigt werden. Um die erstellten Klassen auch in anderen Kontexten verwenden zu können, ist eine strikte Trennung von der graphischen Benutzerschnittstelle unabdingbar. Da Sie zum jetzigen Zeitpunkt Ihres Studiums (im Normalfall) noch keine Entwurfsmethoden kennengelernt haben, verlangen wir natürlich nicht, dass Sie einen Entwurf erstellen, der alle Methoden des Software-Engineering ausreizt. Für ein kleines Projekt, das im Rahmen dieses Praktikums erstellt werden soll, wäre dies auch maßlos übertrieben. Trotz der Größe (bzw. der Kürze) des zu erstellenden Programms ist es vorteilhaft, sich über den grundlegenden Entwurf im Klaren zu sein, bevor man mit der eigentlichen Programmierung beginnt. Um Ihnen zu zeigen, welchen Detaillierungsgrad Ihr Entwurf haben soll und um Ihnen einige Hinweise zu geben, wie man von einer gegebenen Aufgabenstellung einen Entwurf ableiten kann, werden wir dies hier an einem Beispiel vorführen. Dazu verwenden wir eine Aufgabe, die in einem der früheren Durchläufe des Praktikums verwendet wurde.

2.1 6 nimmt!

2.1.1 Aufgabenbeschreibung

Das Ziel bei dieser Aufgabe ist es, das Spiel „6 nimmt!“ zu implementieren. Es handelt sich hierbei um ein Kartenspiel für 2-10 Spieler, bei dem es darum geht, am Ende des Spiels möglichst wenig Minuspunkte zu haben. Die Applikation erlaubt es, sowohl gegen eine beliebige Anzahl von Computergegnern als auch gegen eine beliebige Anzahl von menschlichen Gegnern (am selben Computer) oder eine Kombination (bis insgesamt maximal 10) daraus zu spielen. Die momentane Spielkonfiguration soll graphisch ansprechend dargestellt werden. In verschiedenen Schwierigkeitsgraden für den Computergegner verwendet dieser unterschiedliche Strategien.

2.1.2 Die Regeln

Im Spiel gibt es insgesamt 104 Karten, die von 1 bis 104 durchnumeriert sind. Jede Karte hat den Wert 1 (weiße Karten) mit folgenden Ausnahmen: Karten mit Zahlen, die durch 5 teilbar sind, haben den Wert 2 (blaue Karten), Karten mit Zahlen, die durch 10 teilbar

sind, haben den Wert 3 (orange Karten), Karten mit Zahlen, die durch 11 teilbar sind, haben den Wert 5 (rote Karten) und die Karte mit der Zahl 55 hat den Wert 7 (lila Karte).

Gespielt werden mehrere Spiele, solange bis ein Spieler am Ende eines Spiels mehr als 66 Punkte hat. Es gewinnt dann der Spieler, der die wenigsten Punkte hat.

Zu Beginn jedes Spiels werden die Karten gemischt. Jeder Spieler bekommt 10 Karten verdeckt ausgeteilt, die sich nur der jeweilige Spieler ansehen darf. Wie dies mit mehreren menschlichen Spielern an einem Rechner zu lösen ist, wird später erklärt.

Reihen bilden

Vom restlichen Kartenstapel werden die obersten 4 Karten offen in der Tischmitte ausgelegt (siehe Abbildung 2.1). Jede Karte bildet den Anfang einer Reihe, die einschließlich dieser ersten Karte auf maximal 5 Karten anwachsen darf. Der Kartenstapel, der jetzt noch übrig ist, wird erst wieder im nächsten Spiel benötigt.

Spielablauf

1. Karte ausspielen

   Alle Spieler legen verdeckt 1 Karte von ihren Handkarten vor sich auf den Tisch. Erst dann, wenn der letzte sich entschieden hat, werden die Karten aufgedeckt.

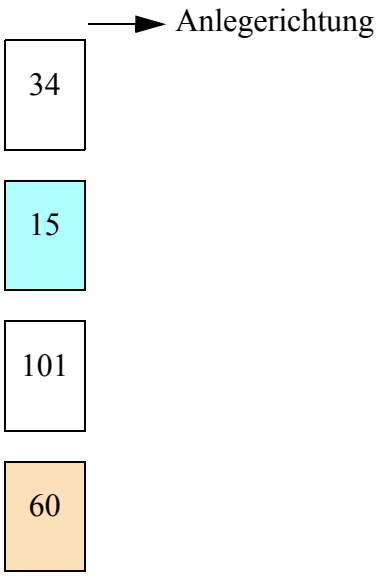


Abbildung 2.1: Startkonfiguration

Wer die niedrigste Karte ausgespielt hat, legt sie als erste Karte an eine der vier Reihen, dann kommt die zweitniedrigste Karte in eine Reihe usw., bis auch die höchste Karte dieser Runde in einer Reihe untergebracht wurde. Die Karten werden in einer Reihe immer nebeneinander gelegt. Danach wiederholt sich dieser Vorgang so oft, bis alle 10 Karten ausgespielt wurden.

Wie werden die Karten zugeordnet?

Jede ausgespielte Karte passt immer nur in eine Reihe. Es gelten folgende Regeln:

- „Aufsteigende Zahlenfolge“
  Die Karten einer Reihe müssen immer eine aufsteigende Zahlenfolge haben.
- „Niedrigste Differenz“
  Eine Karte muss immer in die Reihe gelegt werden, deren letzte Karte die niedrigste Differenz zu der neuen Karte aufweist.

2. Karten kassieren

   Solange man eine Karte in einer Reihe unterbringt, ist alles bestens. Wenn aber in eine Reihe, in die man anlegen muss, keine weitere Karte mehr passt oder wenn man eine Karte in keine Reihe anlegen kann, muss der Spieler, der die entsprechende Karte gespielt hat, Karten kassieren.

3. „Volle Reihe“

   Eine Reihe ist mit 5 Karten voll. Wenn nach Regel 2 eine sechste Karte in diese Reihe gelegt werden muss, dann muss der Spieler, der diese Karte ausgespielt hat, alle 5 Karten dieser Reihe an sich nehmen. Seine sechste Karte stellt den Anfang der neuen Reihe dar.

4. „Niedrigste Karte“

   Wer eine Karte ausspielt, deren Zahl so niedrig ist, dass sie in keine Reihe passt, muss alle Karten einer beliebigen Reihe nehmen und seine „niedrige“ Karte stellt dann die erste Karte dieser Reihe dar.

Karten, die kassiert werden mussten, werden mit ihren Werten als Minuspunkte für den entsprechenden Spieler angeschrieben. Diese Karten sind aus dem Spiel und werden nicht auf die Hand genommen.

Spielende

Das Spiel endet, wenn alle Karten ausgespielt wurden. Die Ergebnisse (Minuspunkte) werden für jeden Spieler vermerkt und es beginnt ein neues Spiel. Das Spiel endet, wenn ein Spieler insgesamt mehr als 66 Minuspunkte gesammelt hat.

2.1.3 Zur Umsetzung

Das Spiel ist wie in den Regeln beschrieben umzusetzen. Für die graphische Darstellung der Karten können sowohl Scans von den Originalkarten (diese werden vom Lehrgebiet nicht zur Verfügung gestellt) als auch alternative Darstellungen gewählt werden. Neben der Anzeige der Spielfläche mit den vier Kartenreihen soll für den menschlichen Spieler, der gerade an der Reihe ist, eine Anzeige der Handkarten existieren. Die Auswahl der zu spielenden Karte ist sowohl über die Maus, als auch über die Tastatur (z.B. Zahlen 0-9) zu steuern, damit bei mehreren menschlichen Spielern ebenfalls eine gewisse Geheimhaltung möglich ist. D.h., die Handkarten eines Spielers können zwar von den anderen menschlichen Spielern eingesehen werden, allerdings können die anderen Spieler nicht erkennen, welche Karte vom aktuellen Spieler durch Tastendruck ausgewählt wird.

Wenn alle Karten ausgespielt sind, beginnt die Anlegephase. Per „Weiter“-Knopf soll diese Phase für den Betrachter in einzelne Schritte aufgeteilt werden können, d.h., mit einem Druck auf diesen Knopf wird die nächste Karte angelegt. Wenn der Fall „Niedrigste Karte“ eintritt, muss der entsprechende Spieler aufgefordert werden, eine Reihe auszuwählen. Durch entsprechende Textmeldungen sollen zudem die Züge aller Spieler nochmal erläutert werden, z.B. „Spieler Jörg legt an Reihe 2 an“, „Computer-Spieler Apfel nimmt Reihe 1“. Der „Weiter-Knopf“ soll deaktiviert werden können, um ein schnelleres Spiel zu gewährleisten.

Um taktisches Spiel zu erlauben, muss zu jedem Zeitpunkt des Spiels eine Tabelle mit den Punktetänden vor Beginn des aktuellen Spiels einzusehen sein. Am Ende jedes Spiels gibt es eine zusätzliche Anzeige des aktuellen Standes.

Zu Beginn eines Spiels kann bestimmt werden, wieviele Spieler (2-10) insgesamt an dem Spiel teilnehmen und wieviele davon vom Computer gesteuert werden (auch ein Spiel Computer-Computer soll möglich sein). Jeder Spieler kann seinen Namen eingeben. Außerdem kann entschieden werden, ob die Standard-Spielgrenze von 66 Punkten geändert werden soll (auf einen beliebigen Wert) oder ob evtl. stattdessen eine beliebige Anzahl an Runden gespielt wird. Für jeden Computer-Gegner ist eine Spielstärke auszu-

wählen. Zusätzlich soll es möglich sein, auch die Handkarten des Computers einzusehen, wenn er an der Reihe ist. Dies ermöglicht es, die Strategie des Computers zu überprüfen.

Für die Computer-Strategie gibt es verschiedene Überlegungen, die noch erweitert werden können und sollen:

Einfache Spielstärke: In der Kartenauswahlphase wird eine zufällige Karte gewählt. Im Fall „Niedrigste Karte“ wird immer die Reihe mit der niedrigsten Gesamtpunktzahl ausgewählt.

Mittlere Spielstärke: Bei der Kartenauswahl wählt der Computer eine Karte, von der er glaubt, dass er in der Folge keine Kartenreihe nehmen muss. Dies kann z.B. eine Karte mit niedriger Differenz zu einer Kartenreihe mit weniger als fünf Karten sein oder z.B. eine Karte mit besonders hoher Differenz zur höchsten Reihe, in der Hoffnung, dass ein anderer Spieler diese Reihe vorher nehmen muss. Es sollte mehrere solcher Regeln geben, die dann gewichtet werden und unter denen die beste ausgewählt wird. Im Fall „Niedrigste Karte“ soll der Computer bewusst gegen denjenigen von den verbleibenden Spielern spielen, der momentan die niedrigste Punktzahl hat.

Hohe Spielstärke: Der Computer merkt sich alle gespielten Karten und bezieht dieses Wissen in seine Berechnungen ein. Außerdem zählt er die aktuellen Punktzahlen auch während des Spiels mit. Bei der Kartenauswahl spielt der Computer auch bewusst Karten, um den Fall „Niedrigste Karte“ zu erzwingen, wenn er sich damit wenig schadet, aber dem führenden Spieler mehr schaden kann.

In der Dokumentation zu Ihrer Implementierung sind die von Ihnen gewählten Computerstrategien ausführlich zu erläutern.

2.1.4 Optionale Erweiterung

Es gibt noch eine Profivariante des Spiels für 2-6 Spieler, bei der zwei zusätzliche Regeln hinzukommen:

1. Alle Karten im Spiel sind bekannt
   Man spielt nur mit so vielen Karten, wie Spieler teilnehmen. Die Regel hierfür ist: Anzahl der Spieler mal 10 plus 4 Karten. Beispiel: 3 Spieler, 34 Karten von 1-34. Alle darüberliegenden Kartennummern werden aussortiert.
2. Jeder Spieler wählt seine 10 Karten selbst aus.
   Die Karten werden offen auf dem Tisch ausgelegt. Reihum, beginnend mit einem zufällig ausgewählten Spieler, nehmen sich die Spieler immer eine Karte, bis sie 10 Karten auf der Hand haben. Jetzt müssen noch 4 Karten auf dem Tisch liegen. Diese 4 Karten stellen die 4 Reihen dar.

Der weitere Ablauf des Spiels entspricht dem Grundspiel.

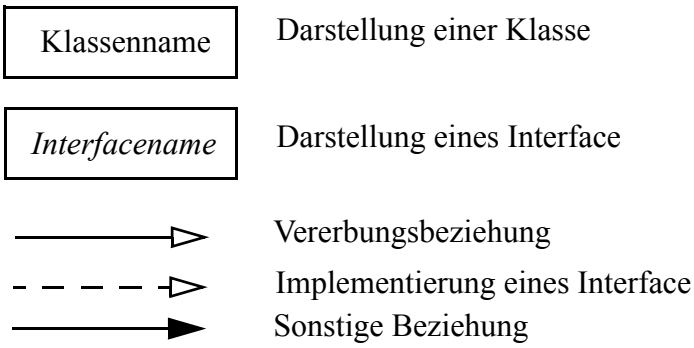
Je nach eingestelltem Schwierigkeitsgrad für die Computergegner soll sich hier das Verhalten der Gegner natürlich auch unterscheiden.

2.2 Erkennen von Klassen und Beziehungen

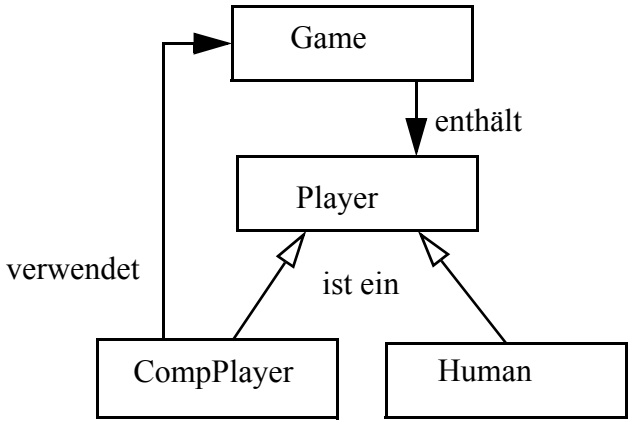
Im Entwurf geht es darum, mögliche Klassen und deren Beziehungen in der Aufgabenstellung zu erkennen. Als Faustregel gilt, dass (wichtige) Substantive in der Aufgabe Kandidaten für Klassen sind. Weniger wichtige Substantive stellen häufig Attribute von Klassen dar. Verben symbolisieren oft Beziehungen zwischen den verschiedenen Klassen. Wir versuchen nun aus der Aufgabenstellung „6 nimmt!“ Klassen und deren Beziehungen herzuleiten.

Schon im ersten Satz der Aufgabestellung entdecken wir den ersten Kandidaten für eine Klasse *Spiel* (*Game*). Dies könnte eine Klasse werden, die die Koordination der anderen Klassen übernimmt. Auch der zweite Satz der Aufgabenstellung liefert uns wichtige Hinweise. Offenbar gibt es im Spiel *Spieler* (*Player*), die jeweils *Minuspunkte* (*points*) besitzen können. Die Minuspunkte lassen sich durch einfache Integerzahlen darstellen, so dass dafür keine eigene Klasse notwendig ist. Wir ordnen sie als Attribut der Klasse *Player* zu. Es gibt zwei Arten von Gegnern, nämlich Computergegner und menschliche Gegner. Dabei handelt es sich jeweils um Spieler. Wir verfeinern daher unsere *Player* Klasse, indem wir zwei neue Klassen (*CompPlayer*) und (*Human*) ableiten. Ein Computergegner verfolgt eine bestimmte Strategie, die wir als Attribut speichern.

Für unsere Diagramme verwenden wir die folgenden Symbole:

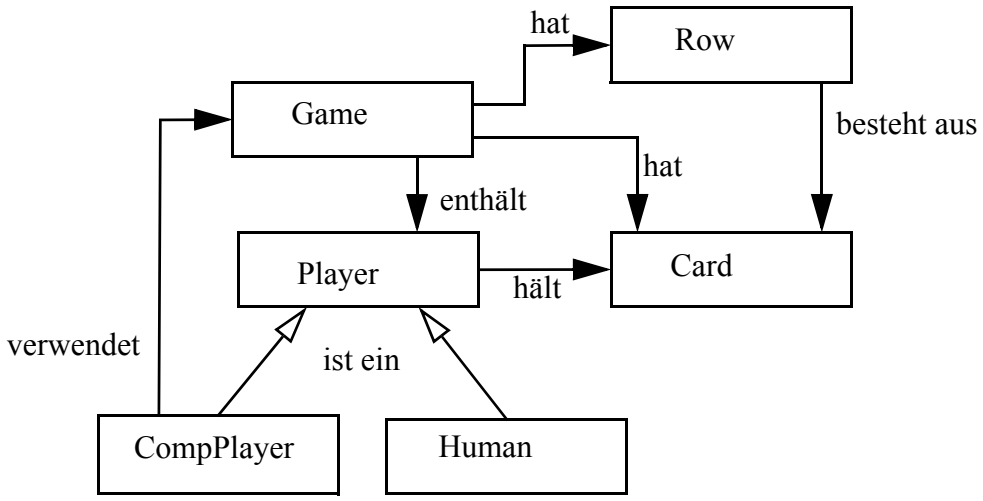


Um eine strikte Trennung zwischen Programmlogik und Oberfläche zu gewährleisten, lassen wir zunächst die graphische Oberfläche aus unseren Entwurf weg. Vorläufig haben wir so allein aus der Aufgabenbeschreibung folgende Klassen und Beziehungen abgeleitet:



Nun betrachten wir den zweiten Teil der Aufgabenstellung („Die Regeln“). Es gibt im Spiel eine Menge von Karten, die jeweils eine Nummer und einen Wert besitzen. Dabei fällt auf, dass der Wert einer Karte direkt aus deren Nummer ableitbar ist. Wir modellieren die Nummer daher als Attribut und den Wert als Methode einer Klasse *Card*. Alle Karten werden zu Beginn des Spiels gemischt und anschließend wird eine Teilmenge der Karten auf die Spieler verteilt. Dies modellieren wir als Beziehungen zwischen Spiel und Karte sowie zwischen Spieler und Karte.

Offenbar gibt es im Spiel 4 Kartenreihen, die maximal 5 Karten aufnehmen können. Eine solche Reihe stellen wir durch die Klasse *Row* dar. Wir erweitern unser bisheriges „Klassendiagramm“ wie folgt:

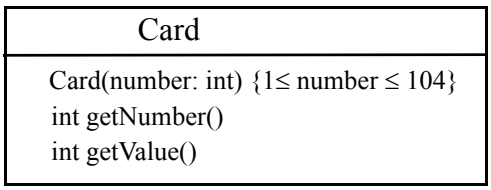


Ein Spieler kann den Spielverlauf durch Auswahl einer seiner Karten gestalten. Danach greifen Regeln, auf die der Spieler zunächst keinen Einfluss hat. Eine Ausnahme bildet die Auswahl der Reihe beim Fall „niedrigste Karte“. Ein Spieler muss Karten kassieren können. Hieraus sind keine neuen Klassen ableitbar, wir merken uns jedoch, dass ein Spieler entsprechende Funktionen (Methoden) anbieten muss.

2.2.1 Attribute und Methoden

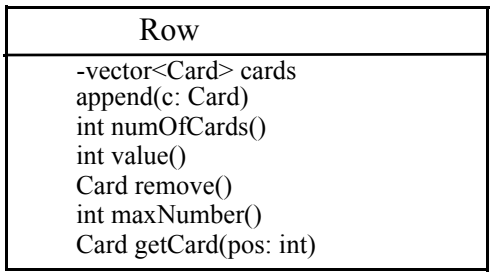
In unseren Überlegungen haben wir bereits einige Attribute und notwendige Methoden zu unseren Klassen vorgemerkt. Jetzt wird es Zeit, sich genauer Gedanken über die benötigten Schnittstellen unserer Klassen zu machen.

Die einfachste Klasse stellt eine Karte dar. Es muss eine Karte mit einer gewünschten Nummer erzeugt werden können. Diese Nummer sowie der Wert (Punktzahl) der Karte bleiben im Spiel unverändert und müssen abgefragt werden können (*getNumber* und *getValue*). Wir erhalten:



Obwohl wir in dieser Darstellung einige `get`-Methoden angeben, werden `get`- und `set`- Methoden in der Klassenbeschreibung meist weggelassen.

Als nächstes schauen wir uns eine Reihe an. Diese besteht aus 0 bis 5 Karten. Es muss eine Karte angelegt werden können (*append*). Weiterhin sollte man die Anzahl der Karten (*numOfCards*), den aufsummierten Wert aller Karten (*value*) sowie die Nummer der obersten Karte (*maxNumber*) abfragen können. Um das „Kassieren“ von Karten implementieren zu können bieten wir eine Methode an, um die oberste Karte zu entfernen (*remove*). Auf eine beliebige Karte lässt sich durch die Methode *getCard* zugreifen. Somit erhalten wir die folgende Klasse:



private Member kennzeichnen wir mit einem vorangestellten -

Betrachten wir nun die Klasse *Player*. Laut Aufgabenstellung soll jeder Spieler einen Namen (*name*) haben. Ihm müssen Karten zugeteilt werden können (10 Stück), die er auf der Hand hat (*handCards*). Die aktuelle Punktzahl (Punktzahl vor dem aktuellen Spiel)

speichern wir in einem Attribut *points*. Kassierte Karten bekommt er mittels *take* aufgedrückt. Wir speichern solche Karten im Attribut *badCards*. Nach Abschluss des Spiels werden die kassierten Karten mittels *back* wieder eingesammelt. Sonstige Aktionen eines Spielers sind die Auswahl einer seiner eigenen Karten (*selectCard*) und die Auswahl einer aufzunehmenden Reihe (*chooseRow*). Diese werden völlig unterschiedlich von den zwei Benutzertypen (Mensch, Computer) realisiert, so dass wir diese als abstrakte Methoden kennzeichnen. Um Spieler über die Züge der anderen Spieler zu informieren, fügen wir entsprechende Funktionen hinzu. Wir erhalten:

Player
-vector<Card> handCards -vector<Card> badCards -int points -String name add(card: Card) take(card: Card) Card back() Card selectCard(rows: Row[]) {abstract} int chooseRow(rows: Row[]) {abstract} void infoCard(p: Player, card: Card) {abstract} void infoRow(p: Player, row: Row) {abstract}

Der menschliche Spieler hat keine weiteren Funktionen. Der Computer hat noch ein Attribut für die gerade verwendete Strategie. Wir verzichten an dieser Stelle auf eine Darstellung der Klassen.

Die koordinierende Klasse *Game* enthält einen Kartenstapel (*cards*) sowie die 4 Reihen (*rows*), verwaltet die Spieler (*player*) und koordiniert das Spiel. Um ein neues Spiel zu beginnen, gibt es eine Methode *init*. Anschließend können die verschiedenen Spieler hinzugefügt werden (*addPlayer*). Die Methode *start* dient dazu, ein neue Runde zu starten. Eine Funktion *next* wird verwendet, um den nächsten Schritt des Spiels auszuführen. Daraus ergibt sich:

Game
-vector<Card> cards -vector<Player> player -array[Row] rows init() addPlayer(player: Player) start() next()

2.2.2 Die graphische Oberfläche

Der bisherige Entwurf reicht bereits aus, um das „6 nimmt!“-Spiel vollständig zu beschreiben. Nun werden noch Klassen für die graphische Oberfläche benötigt. Weiterhin

benötigen wir auch noch entsprechende Verbindungen zu den Klassen des eigentlichen Spiels.

Die verschiedenen graphischen Komponenten ordnen wir innerhalb eines Fensters (Klasse *MainFrame*) an. Zuerst muss es möglich sein, neue Spieler anzulegen. Dies erledigen wir innerhalb eines dafür vorgesehenen Dialogs (*PlayerDialog*). Diese Klasse benötigt natürlich Zugriff auf die Klasse *Player*. Die graphische Darstellung der Karten überlassen wir einer eigenen Klasse *CardPainter*. Diese enthält eine Funktion *paintCard*, die eine gegebene Karte in ein dafür vorgesehenes Rechteck zeichnet. Dafür eine eigene Klasse zu implementieren, erscheint vielleicht etwas übertrieben, erleichtert aber das spätere Ändern der Kartendarstellung. Das Zeichnen des gesamten Spiels überlassen wir einer Klasse *Display*. Da diese die aktuelle Spielsituation darstellen soll, benötigt sie Zugriff auf die Klasse *Game*. Damit alle Informationen auch dargestellt werden können, fügen wir der Klasse *Game* noch Methoden hinzu, die den aktuellen Zustand des Spiels beschreiben (aktueller Spieler, Anzahl der Minuspunkte für jeden Spieler, Angabe der Reiheninhalte). Um die Umsetzung menschlicher Spieler zu ermöglichen, spendieren wir dieser Klasse auch die Funktionen *selectCard* und *chooseRow*. Die Klasse *Human* muss auf diese Funktionen zugreifen können.

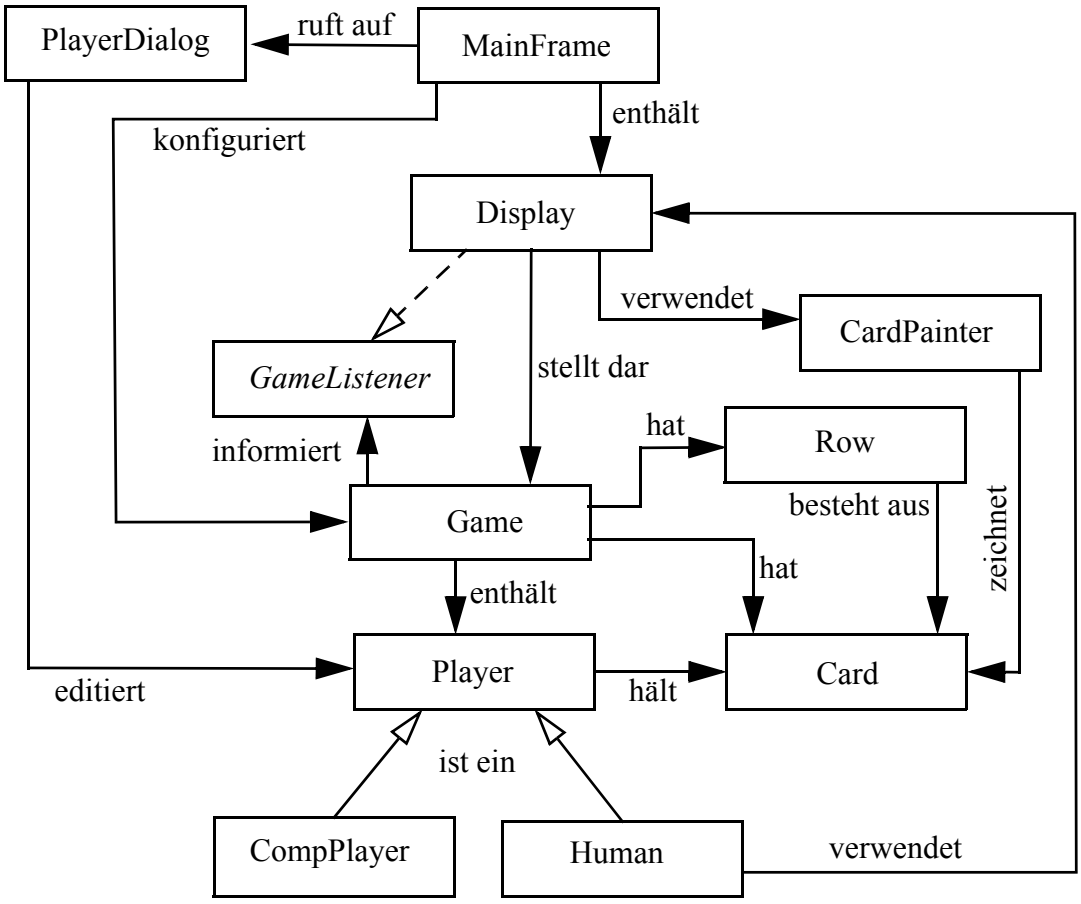
Nun müssen wir uns noch überlegen, wie wir die Oberfläche von Änderungen der Spielsituation informieren können, **ohne** dass unsere Basisklassen Zugriff auf die Oberfläche selbst haben. Dies wird wie folgt modelliert. Ein Interface *GameListener* enthält eine Menge von Funktionen, die über Änderungen der aktuellen Spielsituation informieren. Solche Änderungen können sein:

- eine neue Runde wird eröffnet (*startRound*)
- ein anderer Spieler kommt an die Reihe (*currentPlayer*(*p*: *Player*))
- ein Spieler hat eine Karte (verdeckt) ausgewählt (*selectCard*(*p*: *Player*))
- Karten der Spieler werden aufgedeckt (*selectCard*(*p*: *Player*, *c*: *Card*))
- ein Spieler spielt eine Karte aus (*play*(*p*: *Player*, *c*: *Card*))
- ein Spieler wählt eine Reihe (*playerSelectRow*(*p*: *Player*, *r*: *Row*))
- ein Spieler nimmt eine Reihe auf, da diese voll ist (*fullRow*(*r*: *Row*, *p*: *Player*))
- eine Karte wurde an eine Reihe angelegt (*dock*(*r*: *Row*))
- eine Runde wurde beendet (*roundEnds*())
- ein ganzes Spiel wurde beendet (*gameEnds*())

Der Klasse *Game* verpassen wir ein Attribut vom Typ *GameListener*. Ist dieses gesetzt (nicht *null)*, so wird bei einer Änderung der Spielsituation die passende Funktion aufgerufen. Mittels der Funktion *setGameListener* kann dieses Attribut gesetzt werden. Unsere Anzeigeklasse (*Display*) implementiert nun genau dieses Interface. Nach Aufbau der Oberfläche durch die Klasse *MainFrame* erzeugt diese auch eine Instanz der Klasse

Game und ruft *setGameListener* auf. Somit wird die graphische Oberfläche informiert, obwohl die Basisklassen kein Wissen über die Existenz einer graphischen Oberfläche haben.

Zusammen mit der Oberfläche erhalten wir so den folgenden Entwurf:



Wie ein Entwurf aussehen kann, der aus diesen Überlegungen entstanden ist, sehen Sie in Abschnitt 5.

3 Programmierrichtlinien

Beim Programmentwurf und Implementierung sind folgende Punkte zu beachten:

- **Vollständigkeit**. Die in den Aufgabenstellungen beschriebenen Mindestanforderungen müssen implementiert sein.
- **Korrektheit**. Das Programm muss sich immer den Vorgaben entsprechend verhalten. Ein Programmabsturz muss ausgeschlossen sein, so unsinnig die Benutzereingaben auch sein mögen.
- **Verständlichkeit**. Der Programmquelltext muss so einfach zu verstehen sein, dass es später von einem anderen Programmierer ohne großen Aufwand gepflegt werden kann. Ein wichtiges Konzept, um dies zu erreichen, ist die
- **Modularisierung**. Unterteilen Sie ein großes Problem solange in mehrere kleinere, bis jedes Teilproblem klein genug ist, um auf einfache Weise gelöst zu werden.
- **Effizienz**. Ihr Programm sollte effizient bezüglich Ausführungszeit und Speicherplatzbedarf sein. Dies darf jedoch keinesfalls zulasten der anderen Forderungen gehen.
- **Portierbarkeit**. Das Programm muss auf anderen PCs kompilierbar und lauffähig sein. Verwenden Sie deshalb bitte nur JAVA™ SDK, Standard Edition, ab Version 1.5. Verweisen Sie nur auf solche Dateien, die von Ihnen selbst erzeugt wurden, und benutzen Sie dabei niemals absolute Pfadnamen.

Insbesondere die Forderung nach Verständlichkeit hat nicht nur Auswirkungen auf den Entwurf, sondern auch auf den Programmcode:

- Namen von Klassen, Methoden und Variablen müssen immer eine sinnvolle Bedeutung haben. Beachten Sie die „*Code Conventions for the Java Programming Language*“ (http://www.oracle.com/technetwork/java/codeconv-138413.html) möglichst vollständig. Diese haben sich vielfach bewährt und werden weithin beachtet. Dazu gehören insbesondere:
  - Variablen- und Methodennamen beginnen stets mit einem Kleinbuchstaben.
  - Klassennamen beginnen immer mit einem Großbuchstaben.
  - Jede Programmzeile enthält höchstens eine Anweisung.
- Achten Sie auf die Länge der Methoden. Ab einer gewissen Länge (z.B. 40 Zeilen) sollten Sie die Methode gemäß der Forderung nach Modularisierung unterteilen.

- Der Kopf jeder Methode wird um einen Javadoc-Kommentar (zu Javadoc siehe http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html) ergänzt, der
  - die Bedeutung jedes Parameters erklärt, sofern sie nicht eindeutig aus dem Namen des Parameters hervorgeht;
  - die Aufgabe und Funktionsweise der Methode gut verständlich erläutert;
  - die Voraussetzungen an den Systemzustand beschreibt, unter denen diese Methode aufgerufen werden darf. Dies betrifft im wesentlichen den zulässigen Wertebereich von Eingabeparametern oder globalen Variablen und Annahmen über den vorherigen Aufruf anderer Methoden. So setzen beispielsweise Dateioperationen voraus, dass die betroffene Datei zuvor geöffnet worden ist.
- Jede Klasse ist mit einer Beschreibung im Javadoc-Stil zu versehen, aus der hervorgeht, welche Bedeutung diese Klasse hat und welche Methoden implementiert sind.
- Zusätzlich wird jede erklärungsbedürftige Anweisung um einen leicht verständlichen Kommentar ergänzt.
- Schachtelungstiefen von Anweisungsblöcken sind durch entsprechend tiefe Einrückungen zu visualisieren.
- Logisch zusammengehörende Anweisungsteile müssen als solche erkennbar sein. Dies kann durch Kommentare oder sinnvolles Einfügen von Leerzeilen erreicht werden.

Beschränken Sie sich auf die mit dem JAVA™ SDK, Standard Edition, ab Version 1.5 mitgelieferten Pakete. Alle über diese Pakete hinausgehenden Funktionalitäten sind eigenständig zu implementieren. Es ist allerdings dringend anzuraten, in den Paketen zunächst nach Methoden mit bestimmter Funktionalität zu suchen, bevor sie neu implementiert werden.

4 Dokumentationsrichtlinien

Eine vollständige Dokumentation besteht aus folgenden Teilen:

1. Deckblatt (Abbildung 4.1 zeigt ein Beispiellayout)

Abbildung 4.1: Ein Beispiel für ein Deckblatt

2. maximal zweiseitige Anleitung zu Installation und Aufruf Ihres Programmes; beschreiben Sie auch Systemanforderungen (z.B. die verwendete Java-Version)
3. kurze Bedienungsanleitung zur Benutzerschnittstelle
4. Überblick über die in Ihrem Programm zur Lösung der Aufgabe verwendeten Konzepte im Umfang von etwa 1-3 DIN-A4-Seiten
5. Beschreibung der tatsächlich implementierten Programmarchitektur mit Klassen- und Methodenbeschreibungen, entsprechend dem Beispielentwurf (Abschnitt 5)
6. Falls sich Ihre Programmarchitektur gegenüber dem von Ihnen eingesandten Entwurf verändert hat, beschreiben Sie zusätzlich kurz die Änderungen und die Gründe, aus denen Sie sie vorgenommen haben (ca. 1-3 DIN-A4-Seiten).

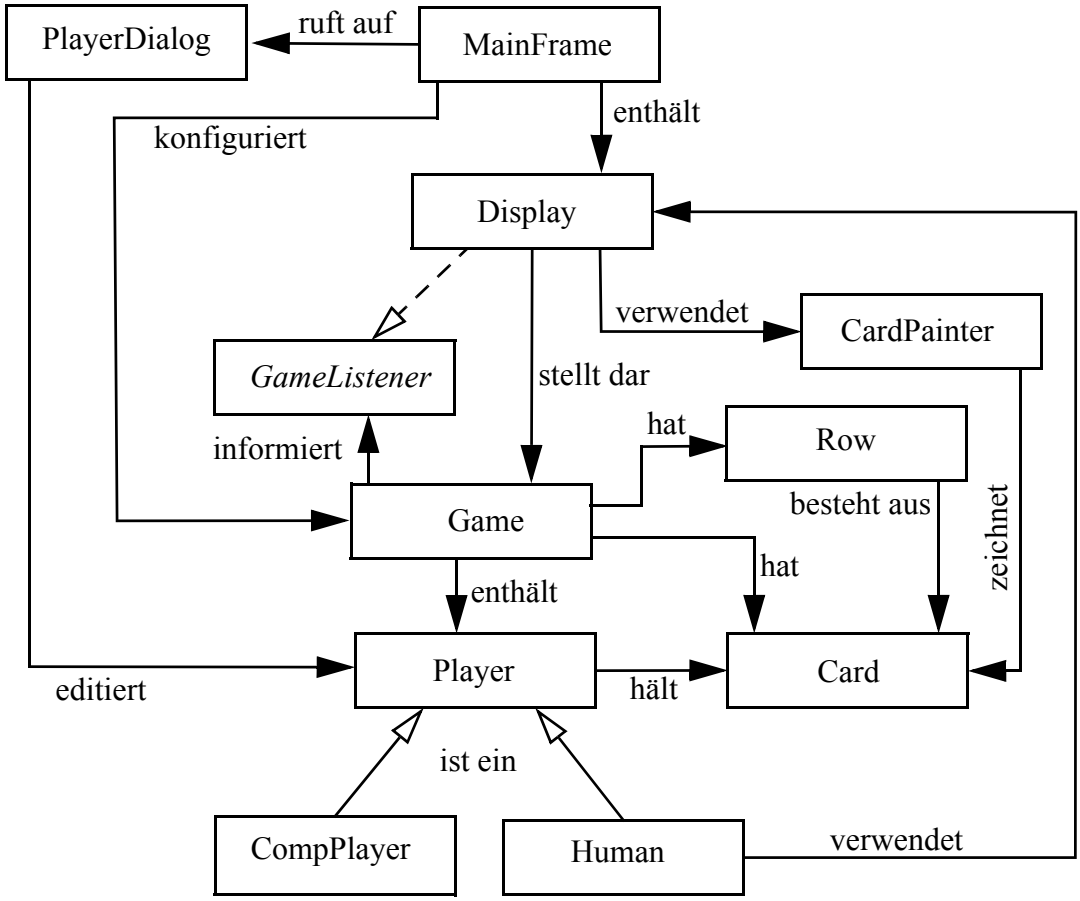
Das gesamte Dokument ist, beginnend mit der ersten Textseite, mit fortlaufenden Seitenzahlen zu versehen.

Zusätzlich sollen Sie eine Dokumentation Ihrer Implementierung mit *javadoc* (im SDK enthalten) erstellen.

5 Entwurf zum „6 nimmt!“-Spiel

5.1 Klassendiagramm

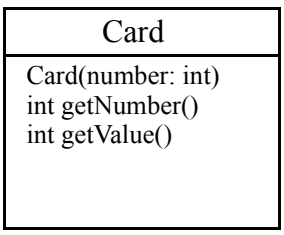
Für die Realisierung des „6 nimmt!“-Spiels ist folgende Programmarchitektur vorgesehen:



5.2 Funktionalität der Klassen

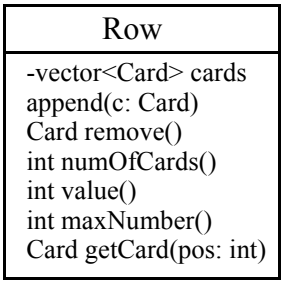
Card

Die Klasse *Card* dient zur Beschreibung einer Karte des „6 nimmt!“-Spiels. Sie besitzt eine Nummer sowie einen Wert. Der Wert einer Karte ist direkt aus ihrer Nummer ableitbar. Die



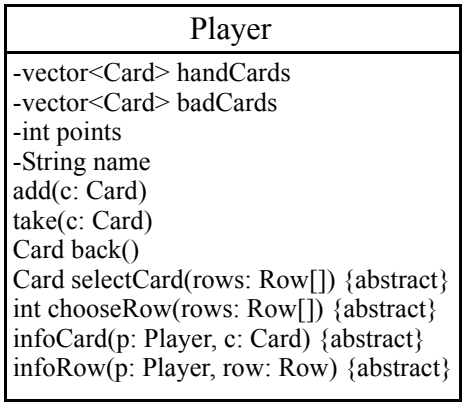
Nummer einer Karte wird im Verlauf des Spiels nicht mehr geändert. Nummer und Wert einer Karte lassen sich abfragen.

Row



Eine Reihe wird durch die Klasse *Row* repräsentiert. Diese besteht aus 0-5 Karten. Sie bietet Funktionen zum Anlegen einer Karte (*append*) sowie zum Löschen der obersten Karte (*remove*). Es lassen sich die Anzahl der Karten in der Reihe (*numOfCards*), der Wert der obersten Karte (*maxValue*), die Karte an einer gegebenen Position (*getCard*) sowie der aufsummierte Wert aller enthaltenen Karten (*value*) abfragen. Zur Darstellung der Folge von Karten wird ein privates Attribut *cards* verwendet.

Player



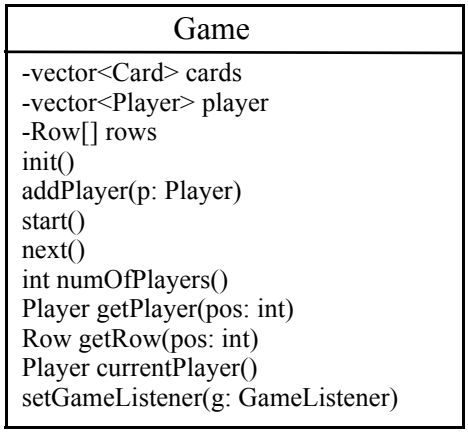
Mittels Instanzen der Klasse *Player* werden die Spieler des Spiels dargestellt. Jeder Spieler hat einen Namen (*name*) sowie eine Anzahl von Minuspunkten (*points*). Karten, die momentan auf der Hand sind, werden in einem Attribut (*handCards*) dargestellt. Die während einer Runde einkassierten Karten werden ebenfalls in einem Attribut (*badCards*) festgehalten. Um das Austeilen von Karten zu ermöglichen, gibt es die Funktion *add*, die den Handkarten eine neue Karte hinzufügt. Eine „kassierte“ Karte erhält ein Spieler mittels *take*. Solche Karten werden am Ende der Runde durch die Funktion *back* wieder dem Spiel übergeben, wobei eine entsprechende Anpassung der Minuspunkte erfolgt. Ein Spieler kann durch Auswahl einer seiner Karten (*selectCard*) oder durch Auswahl einer Reihe (*chooseRow*) ins Spielgeschehen eingreifen. Weiterhin gibt es Funktionen, die einen Spieler über Aktionen von anderen Spielern informieren. Die Implementierungen der letztgenannten Funktionen hängen stark davon ab, ob es sich um einen menschlichen oder einen Computerspieler handelt und werden daher als *abstrakt* gekennzeichnet.

CompPlayer und Human

Die beiden Klassen *CompPlayer* und *Human* sind Unterklassen der Klasse *Player* und implementieren die dort definierten abstrakten Funktionen auf unterschiedliche Weise.

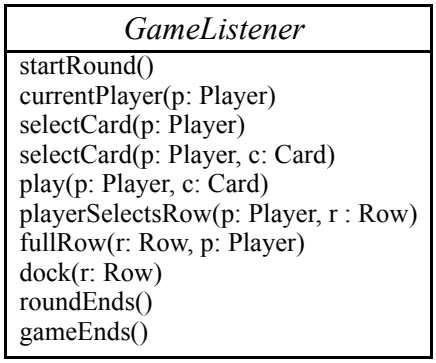
Die Klasse *CompPlayer* enthält noch ein zusätzliches Attribut, um die gerade verfolgte Strategie zu speichern.

Game



Die Klasse *Game* organisiert das gesamte Spiel. Der Kartenstapel, von dem die Karten verteilt werden, ist in einem Attribut *cards* festgehalten. Die am Spiel teilnehmenden Spieler werden im Attribut *player* gespeichert. Mittels *addPlayer* kann ein neuer Spieler hinzugefügt werden. Die Funktion *init* dient dazu, die Ausgangskonfiguration eines Spiels (ohne Spieler) herzustellen. Mittels *start* wird eine neue Runde gestartet. Diese Methode sollte aufgerufen werden, wenn alle Karten ausgespielt wurden, jedoch noch nicht die erforderliche Maximalpunktzahl erreicht wurde, um das gesamte Spiel zu beenden. Durch *next* wird ein vollständiger Schritt des Spiels angestoßen. Dabei spielen alle Spieler ihre Karten aus, die dann automatisch an die Kartenstapel angelegt werden, falls dies möglich ist. Falls nicht, wird die übervolle oder ausgewählte Reihe dem „Opfer“ zugedacht. Weitere Funktionen dienen dazu, sich über den aktuellen Zustand des Spiels zu informieren. Falls man über Änderungen der Spielsituation informiert werden möchte, registriert man einen *GameListener* bei der Klasse *Game*.

GameListener



Dieses Interface wird verwendet, um Interessenten über Änderungen im aktuellen Spiel zu informieren. Es enthält eine Reihe von Funktionen, die aufgerufen werden, wenn die passende Veränderung der Spielsituation eingetreten ist. Den Beginn und das Ende einer Runde signalisiert der Aufruf der Funktion *startRound* bzw. *roundEnds*. Wechselt der Spieler, der momentan an der Reihe ist, wird *currentPlayer* aufgerufen. Die verdeckte Auswahl einer Karte wird mittels *selectCard* angezeigt. Wird dieser Funktion zusätzlich noch die selektierte Karte übergeben, so entspricht dies dem Aufdecken dieser Karte durch den Spieler. Die Funktion *play* wird verwendet, um das Ausspielen einer Karte durch einen Spieler kenntlich zu machen. Muss ein Spieler eine volle Reihe aufnehmen, wird *fullRow* aufgerufen. Das Aufnehmen einer selbst

gewählten Reihe ist durch *playerSelectsRow* mitzubekommen. Die Funktion *dock* entspricht dem Anlegen einer Karte an eine Reihe. Das Ende des Spiels erfährt ein *GameListener* durch den Aufruf seiner Funktion *gameEnds*.

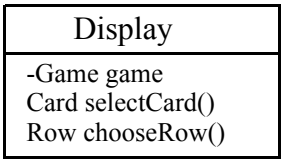
CardPainter

Diese Klasse enthält (vorerst) nur eine einzige Funktion *paintCard*. Neben der Karte erhält diese Funktion noch einen Graphikkontext, auf dem die Karte gemalt werden soll, sowie die Position und Größe eines Rechtecks, in das die Karte zu zeichnen ist.

Playerdialog

Diese Klasse stellt ein Fenster dar, in dem nach Auswahl der Anzahl der Mitspieler jedem Spieler ein Name zugewiesen werden kann. Außerdem kann hier angegeben werden, ob es sich um einen menschlichen oder um einen Computergegner handelt. Nach Bestätigung der Einstellungen über einen entsprechenden Button werden die Spieler gemäß diesen Einstellungen erzeugt. Nach außen hin bietet diese Klasse nur eine Funktion *createPlayers*, die einen Vektor von Spielern zurückgibt.

Display



Diese Klasse (abgeleitet von JPanel) übernimmt die gesamte Darstellung des Spiels. Um über Änderungen der Spielsituationen informiert zu werden, implementiert sie das *GameListener* Interface. Die manuelle Auswahl von Karten oder das Aufnehmen einer Reihe wird hier ebenfalls implementiert.

MainFrame

Hier wird das Hauptfenster des Spiels angezeigt. Es organisiert die verschiedenen Elemente des Spiels. Die restlichen Klassen des „6 nimmt!“-Spiels verwenden keine Funktionen dieser Klasse.

5.3 Beziehungen zwischen den Klassen

Die Vererbungsbeziehungen im Diagramm wurden bereits in den Klassenbeschreibungen erläutert. Da die Klasse *Game* für den Ablauf des Spiels verantwortlich ist, ist dies auch die Klasse, die einen evtl. registrierten *GameListener* informieren kann. Dabei wer-

den alle durch den *GameListener* angebotenen Funktionen verwendet. Da ein Spiel einen Kartenstapel hat und dafür verantwortlich ist, Karten zwischen diesem Stapel, Spielern und Reihen auszutauschen, ist eine Beziehung zu den Klassen *Card*, *Player* und *Row* notwendig. Eine Reihe ist eine geordnete Menge von 0-5 Karten. Um ihren Gesamtwert und ihre maximale Nummer berechnen zu können, muss sich eine Reihe auf Funktionen der Klasse *Card* abstützen.

Um eine Karte korrekt darstellen zu können, muss die *CardPainter*-Klasse Wert und Nummer einer Karte erfahren können. Die Klasse *Display* nutzt die Funktionalität von *CardPainter*, um die sichtbaren Karten des Spiels zu zeichnen. Bei wem sich die sichtbaren Karten des Spiels aufhalten, „weiß“ nur die Klasse *Game*. Die *Display*-Klasse ist daher auf diese Klasse angewiesen, um das Spiel darzustellen.

Der *PlayerDialog* erzeugt und manipuliert die Spieler und benötigt daher Zugriff auf diese Klasse. Der Dialog selbst wird von der Klasse *MainFrame* aufgerufen. Die vom *PlayerDialog* erzeugten Spieler werden durch eine Instanz der Klasse *MainFrame* an das Spiel weitergereicht (*addPlayer*). Weiterhin wird auch die Spielsteuerung hier bewerkstelligt (Aufruf von *init*, *start* und *next*). Innerhalb des Hauptfensters ist auch eine Instanz der Klasse *Display* angeordnet, die die graphische Ausgabe übernimmt. Da ein menschlicher Spieler nur über die Oberfläche seine Karten und Reihen auswählen kann, ist die Klasse *Human* auf die Funktionen *selectCard* und *chooseRow* der Klasse *Display* angewiesen.

Pakete

Wir teilen unser Projekt in zwei Pakete auf. Ein Paket *game* übernimmt die Spielelogik und soll keine Klassen des anderen Pakets verwenden. Das zweite Paket *gui* ist für die graphische Darstellung sowie die Kommunikation mit menschlichen Spielern verantwortlich. Zum Paket *game* zählen die folgenden Klassen: *Card*, *Row*, *CompPlayer*, *Player*, *Game*, *GameListener*. Alle anderen Klassen werden im Paket *gui* untergebracht.