Universität der Bundeswehr
Fakultät für Informatik – Institut 2
Prof. Dr.-Ing. Mark Minas

FT 2025
Klausurvorbereitungsaufgabe

Objektorientierte Programmierung

Mau-Mau

Hinweis: Lesen Sie sich vor der Bearbeitung diese Angabe einmal komplett durch. Lesen Sie sich auch in jeder Aufgabe die Teilaufgaben gewissenhaft und vollständig durch, bevor Sie mit deren Bearbeitung beginnen!

1 Einleitung

In dieser Klausurvorbereitungsaufgabe geht es um das Kartenspiel Mau-Mau, dessen Spielregeln beispielsweise bei Wikipedia nachgelesen werden können. Hier soll das Spiel mit einem Skatblatt gespielt werden, also französischen Karten mit den Werten 7, 8, 9, 10, Bube, Dame, König und Ass. Um auch mit einer großen Zahl von Spielerinnen spielen zu können, verwenden wir gegebenenfalls mehrere Skatblätter, so dass dann alle Karten mehrfach vorkommen. Als Sonderkarten verwenden wir wie üblich 7 („Zwei-Ziehen“), 8 („Aussetzen“) und Bube („Wünschen/Allesleger“). Es soll verboten sein, einen Buben auf einen zuvor abgelegten Buben zu legen.

Zu Spielanfang wird festgelegt, mit wie vielen Skatblättern gespielt wird und wie viele Karten jede Spielerin anfangs bekommt. Dann werden alle Karten gemischt, jede Spielerin bekommt die festgelegt Anzahl an Karten. Eine der verbleibenden Karten wird offen aufgedeckt und bildet den Anfang des Ablagestapels. Da diese Karte von keiner Spielerin abgelegt wurde, hat sie keine Sonderfunktion, d.h. 7, 8 und Bube werden als ganz „normale“ Karten gewertet. Die restlichen Karten bilden den Zugstapel. Dann sind die Spielerinnen reihum am Zug. Wenn eine Spielerin keine Karte ablegen kann oder will, muss sie eine Karte vom Zugstapel nehmen. Wenn eine Spielerin ihre vorletzte Karte ablegt, muss sie „Mau“ sagen, bei ihrer letzten Karte „Mau-Mau“. Vergisst sie das, muss sie eine Karte vom Zugstapel aufnehmen. Ist der Zugstapel komplett aufgebraucht, werden die Karten des Ablagestapels – mit Ausnahme der obersten Karte – neu gemischt und anschließend als Zugstapel verwendet.

Es gewinnt die Spielerin, die als erste alle ihre Karten abgelegt hat. Die verbleibenden Spielerinnen spielen weiter, bis die „zweite Siegerin“ feststeht, und so weiter. Das Spiel endet, wenn

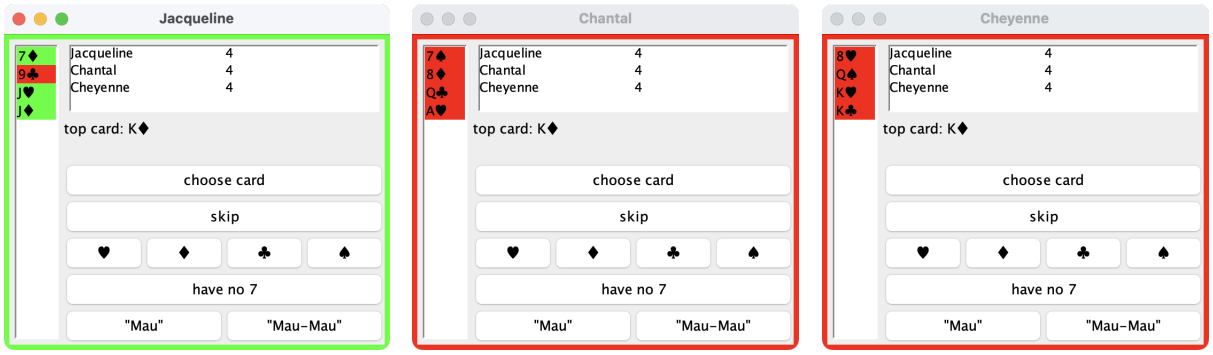

Abbildung 1: Screenshots eines Spiels mit den drei Spielerinnen Jacqueline, Chantal und Cheyenne.

die vorletzte Spielerin alle ihre Karten abgelegt hat und nur noch eine Spielerin Karten auf der Hand hat. Das ist dann die „letzte Siegerin“.

2 GUI

Ziel der folgenden Aufgaben ist es, die Spielmechanik von Mau-Mau zu realisieren. Um sie zu testen und auszuprobieren, stellen wir Ihnen einige JUnit-Testfälle sowie eine rudimentäre graphische Benutzerschnittstelle (GUI) zur Verfügung. Abb. 1 zeigt Screenshots dieser GUI, wenn man die Spielmechanik vollständig realisiert und ein Spiel mit drei Spielerinnen (Jacqueline, Chantal und Cheyenne) begonnen hat. Beachte, dass es bei dieser KVA nicht um das Spielerlebnis geht. Dazu wäre es notwendig, das Spiel auf unterschiedlichen Rechnern spielen zu können, die über das Netzwerk kommunizieren, wodurch der Rahmen dieser KVA aber gesprengt würde.[1] Stattdessen spielt man alle Spielerinnen eines Spiels gemeinsam auf einem Rechner. Die drei Fenster erscheinen daher alle gleichzeitig auf dem eigenen Bildschirm, und man muss reihum die Rollen von Jacqueline, Chantal und Cheyenne übernehmen. Das ist aus Sicht des Spiels zwar langweilig und dem Spielerlebnis sicher nicht zuträglich, hier geht es aber – wie gesagt – lediglich um die Realisierung der Spielmechanik.

Im Folgenden gehen wir auf die Bedienung der GUI ein. Daraus folgen dann letztlich die Anforderungen an die Spielmechanik: Gestartet wird ein Spiel mit der vorgegebenen Klasse `cards.maumau.MauMauGame`. Als Kommandozeilenparameter ist anzugeben, wie viele Karten jede Spielerin zu Anfang erhalten und wie viele Skatblätter verwendet werden sollen. Außerdem sind die Namen der teilnehmenden Spielerinnen anzugeben. In diesem Fall erhält jede Spielerin vier Karten, es wird mit einem Skatblatt gespielt, und es spielen Jacqueline, Chantal und Cheyenne. Die Kommandozeilenparameter sind daher:

```
4 1 Jacqueline Chantal Cheyenne
```

Das Programm zeigt dann für jede Mitspielerin ein eigenes Fenster. Die Reihenfolge der Spielerinnen ist in jedem Fenster in einer Tabelle rechts oben zu sehen, zusammen mit der Information, wie viele Karten jede Mitspielerin noch auf der Hand hat. An der Reihe ist jeweils die erstgenannte Spielerin, d.h. der Tabelleninhalt ändert sich mit fortlaufendem Spiel ständig. Zusätzlich

[1] Tatsächlich wird es im Programmierprojekt um die Realisierung von Spielen mit Netzwerkunterstützung gehen.

gibt der grüne Fensterrahmen an, wer an der Reihe ist; alle anderen Fenster haben einen roten Rahmen.

Unterhalb der Tabelle mit den Namen der Spielerinnen wird die oberste Karte das Ablagestapels angezeigt, hier der Karo-König. Jeweils links ist das Blatt der jeweiligen Spielerin zu sehen. Grün hinterlegt sind die Karten, die gespielt werden können; das sind hier die Karo-Sieben und die beiden Buben, die auf jede Karte außer einen gespielten Buben passen. Jacqueline spielt eine dieser Karten, indem sie sie mit der Maus auswählt und anschließend den Knopf *choose card* betätigt. Ist bei Betätigung dieses Knopfes eine unzulässige Karte ausgewählt (hier die Kreuz-Neun), soll nichts passieren. Wenn man keine Karte legen kann (dann sind sie alle rot hinterlegt) oder will, betätigt man den Knopf *skip*, wodurch die Spielerin automatisch eine zusätzliche Karte vom Zugstapel erhält. Die vier Knöpfe mit den Farben ♥, ♦, ♣ und ♠ benötigt man zur Wahl der entsprechenden Farbe: Spielt man einen Buben, muss man unmittelbar danach den der gewünschten Farbe entsprechenden Knopf betätigen.

Die drei restlichen Knöpfe haben die folgende Bedeutung: *have no 7* betätigt eine Spielerin, wenn eine Sieben auf den Ablagestapel gelegt wurde und sie selbst keine Sieben spielen kann oder will. Dann erhält sie doppelt so viele Karten vom Zugstapel, wie zuvor Siebener in Folge abgelegt worden waren. Deren Zahl wird im Fenster unmittelbar oberhalb des Knopfes *choose card* angezeigt. In Abb. 1 ist hier nichts angezeigt, weil keine Sieben abgelegt wurden.

Die Taste *"Mau"* bzw. *"Mau-Mau"* muss eine Spielerin betätigen, wenn sie ihre vorletzte bzw. letzte Karte abgelegt hat. Zur Strafe erhält sie eine Karte vom Zugstapel, wenn sie das vergisst und nicht die entsprechende Taste betätigt, bevor die nächste Spielerin ihren Spielzug beendet hat.

Eine Spielerin, die alle ihre Karten abgelegt hat, erscheint nicht mehr in der Tabelle der aktuellen Spielerinnen rechts oben in jedem Fenster, ihr Fenster bleibt aber sichtbar; das Spiel läuft dann mit den verbleibenden Spielerinnen weiter. Wenn nur noch eine Spielerin verbleibt, wird in der Tabelle die Rangfolge der Siegerinnen dargestellt; dann ist keine Spielerin mehr an der Reihe, d.h. alle Fenster haben einen roten Rahmen.

Den Programmcode zur GUI, den Spielkarten und der Klasse `cards.maumau.MauMauGame` finden Sie in einer Zip-Datei in Ilias (siehe folgende Aufgabe). Es fehlt also noch die Realisierung des gesamten Verhaltens, das man unter Nutzung der GUI auslöst und das den obigen Erläuterungen zugrunde liegt. Im Folgenden ist dieses Verhalten mit Zustandsautomaten spezifiziert, die Sie zur Realisierung des geforderten Verhaltens implementieren sollen. Zuvor gehen wir aber auf die zugrunde liegende Struktur ein. Um sich in den bestehenden Programmcode einzuarbeiten, bearbeiten Sie die folgende Aufgabe:

Aufgabe 1 (Vorbereitung)

Bereiten Sie im Folgenden Ihre Programmierumgebung für die Lösung der nachstehenden Aufgaben vor.

a) Sofern noch nicht vorhanden, laden Sie sich eine geeignete Entwicklungsumgebung für Java herunter (am besten *IntelliJ*) und installieren Sie diese auf Ihrem Rechner. Stellen Sie sicher, dass Sie in Ihrer Entwicklungsumgebung die aktuelle Java-Version verwenden.

b) Erstellen Sie ein neues Java-Projekt mit geeignetem Namen (z.B. „KVA“) in Ihrer Entwicklungsumgebung.

c) Laden Sie von Ilias die Zip-Datei *KVA.zip* herunter und entpacken sie diese in Ihrem Projekt. Dadurch befinden sich in Ihrem Projekt bereits die in Abb. 2 abgebildeten Dateien. Ihre Entwicklungsumgebung sollte den enthaltenen Quellcode problemlos und ohne Fehlermeldungen übersetzen können. Ausführen lässt sich `MauMauGame` aber noch nicht; weil wesentlicher Code noch fehlt, bricht es mit einer Exception ab.

d) Ergänzen Sie in Ihrem Projekt die JUnit-Unterstützung, wie es in Aufgabe 1b des zweiten Übungsblatts beschrieben ist, und führen Sie alle Testfälle aus. Sie schlagen aber alle fehl, weil die getestete Funktionalität noch nicht realisiert ist.

e) Machen Sie sich nun mit dem bestehenden Programmcode vertraut, indem Sie beispielsweise die Methoden identifizieren, die Sie im Folgenden realisieren sollen. Sie sind alle mit einem TODO-Kommentar gekennzeichnet.

Abb. 3 zeigt ein Klassendiagramm des Mau-Mau-Programms, das dem *Model-View-Controller*-Muster (siehe Vorlesungskapitel 8) folgt, in dem aber die meisten Attribute und Methoden aus Gründen der Übersichtlichkeit fehlen. Die im Rechteck dargestellten Klassen etc. sind Teil des Modells des MVC-Musters, die verbleibenden Klassen gehören zu View und Controller. Alle dargestellten Klassen etc. sind in der vorgegebenen Zip-Datei enthalten oder gehören zur Java-API (`JFrame` und `Comparable`). Im Folgenden besprechen wir schrittweise ihre Aufgaben und Funktionen. Zwischen die Erläuterungen sind jeweils Aufgaben eingestreut, in denen Sie noch fehlende Methoden und ggf. weitere Klassen realisieren sollen. Schritt für Schritt sollen so jeweils mehr Testfälle erfolgreich durchlaufen, bis Ihr Code am Schluss alle Testfälle erfüllt und Sie das Spiel ausführen können.

3 `Card`, `Rank` und `Suit`

Spielkarten werden mit dem Record `Card` und den Aufzählungstypen `Rank` und `Suit` realisiert; alle drei sind in der vorgegebenen Zip-Datei enthalten. `Suit` repräsentiert die Farben ♥, ♦, ♣ und ♠ mit den englischen Bezeichnungen *Hearts*, *Diamonds*, *Clubs* und *Spades*, der Aufzählungstyp `Rank` die Zahlenkartenwerte $2, 3, \ldots, 9, 10$, außerdem Bube, Dame, König und Ass, abgekürzt J, Q, K, A für die englischen Bezeichnungen *Jack*, *Queen*, *King* und *Ace*. Der Record `Card` bildet einfach Paare von `Rank` und `Suit`.

Zur Erläuterung von Records: Seit Java 16 werden in Java auch sog. *Datenklassen* unterstützt.[2] Ihr Zweck ist es, Daten in unveränderlichen Objekten zu halten und dem Programmierer Schreibarbeit zu sparen. Ein sog. *kanonischer Konstruktor* steht ebenso automatisch zur Verfügung wie Getter-Methoden für die Attribute sowie kanonische `equals`-, `hashCode`- und `toString`-Methoden. Die Getter-Methode von `Card` sind demnach `rank` und `suit`.

`Card` implementiert das Interface `Comparable<Card>` und damit eine lineare Ordnung auf ihnen. Wie man in Abb. 1 sieht, werden die Karten der Spielerinnen sortiert dargestellt. Hierfür ist der Methodenaufruf `sorted()` in der `updateCardList`-Methode von `PlayerFrame` zuständig. Wie man der API-Dokumentation von `sorted()` entnehmen kann, muss `Card` dazu das `Comparable`-Interface und daher die `compareTo`-Methode implementieren.

[2] https://openjdk.java.net/jeps/395

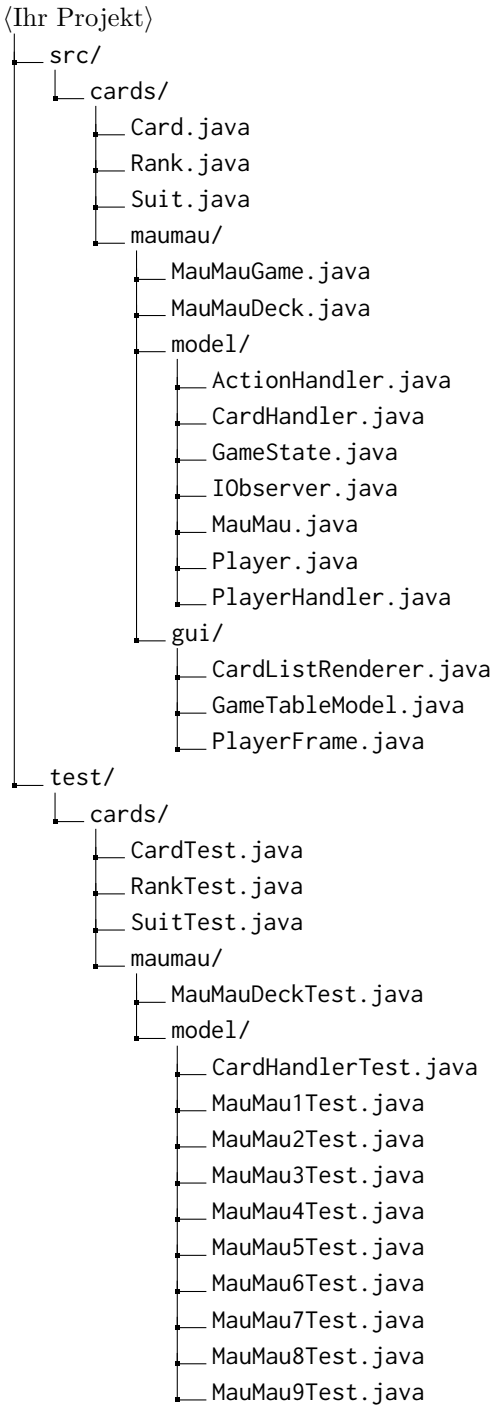

Abbildung 2: Verzeichnisstruktur nach Entpacken der Zip-Datei *KVA.zip*

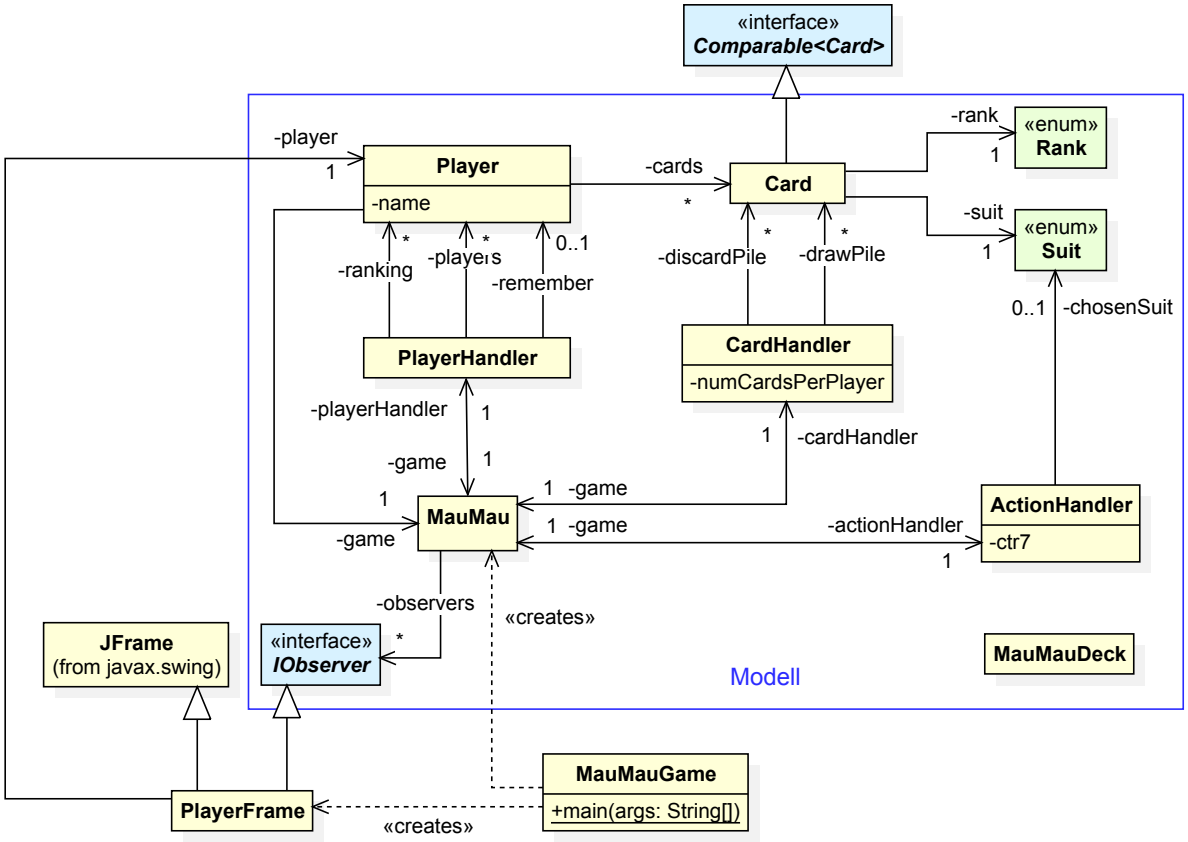

Abbildung 3: Grobes Klassendiagramm von Mau-Mau.

Aufgabe 2 (Spielkarten)

a) Implementieren Sie in den Aufzählungstypen `Rank` und `Suit` die `toString`-Methode, so dass sie für `Rank`-Instanzen nicht `"TWO"`, `"THREE"` usf., sondern $2, 3, 4, \ldots, J, Q, K, A$, und für `Suit`-Instanzen ♥, ♦, ♣ bzw. ♠ zurückgeben.

Beachten Sie hier die beiden Testklassen `RankTest` und `SuitTest`.

b) Implementieren Sie die `compareTo`-Methode im Record `Card`, so dass sie eine lexikographische Ordnung zuerst auf dem `Rank`- und anschließend auf dem `Suit`-Wert realisieren. `Ranks` seien hier gemäß

$$2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A$$

und `Suits` gemäß

$$♥ < ♦ < ♣ < ♠$$

sortiert, so dass sich auf Spielkarten die folgen de Ordnung ergibt:

$$2♥ < 2♦ < 2♣ < 2♠ < 3♥ < 3♦ < 3♣ < 3♠ < \cdots < A♥ < A♦ < A♣ < A♠$$

Lesen Sie dazu die API-Dokumentation zur `compareTo`-Methode im Interface `Comparable`. Beachten Sie ferner, dass alle Aufzählungstypen, und damit auch `Rank` und `Suit`, automatisch das `Comparable`-Interface implementieren!

Beachten Sie hier die Testklasse `CardTest`.

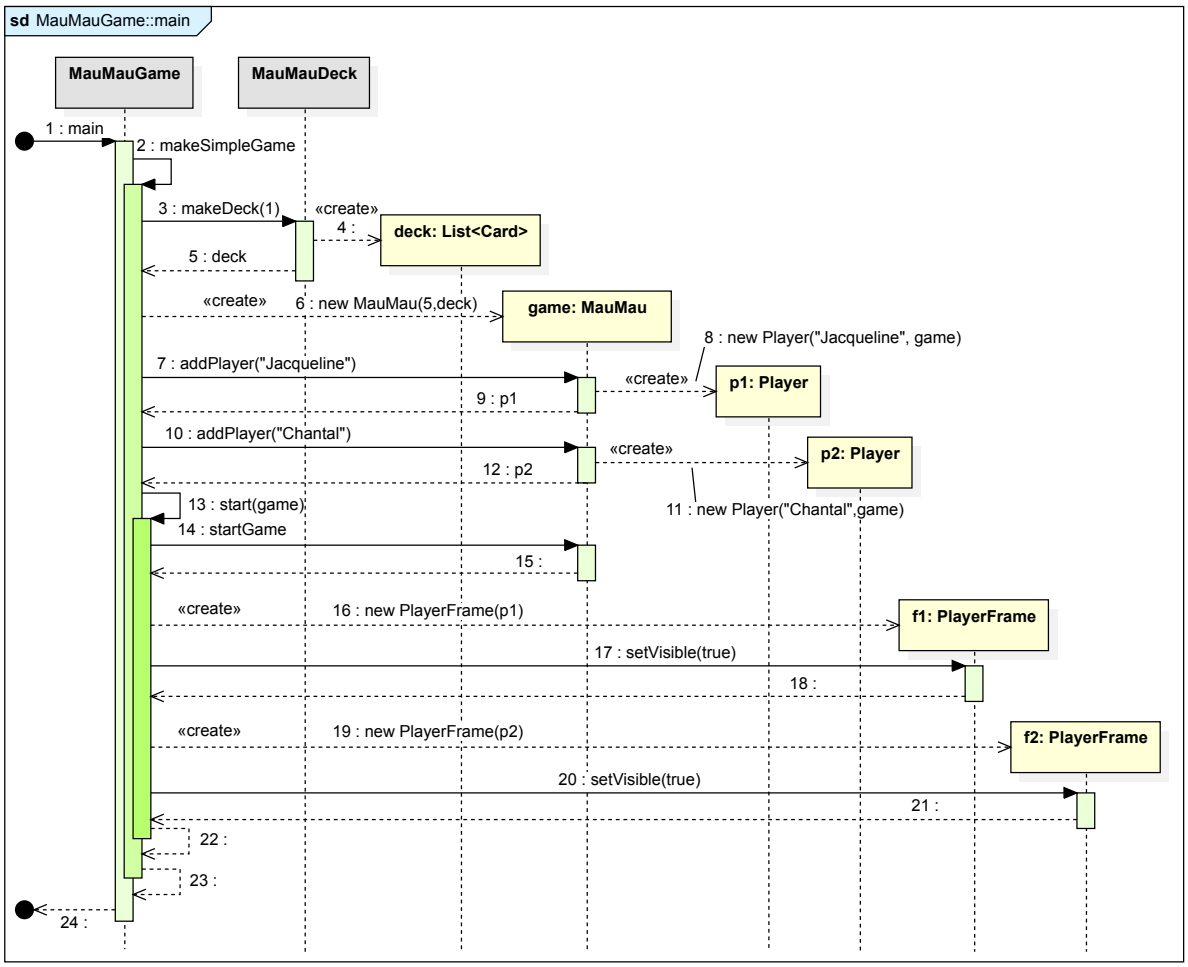

Abbildung 4: Sequenzdiagramm des Starts eines Mau-Mau-Spiels mit zwei Spielerinnen.

c) Implementieren Sie nun die Methode `makeDeck` der Klasse `MauMauDeck`. Der übergebene Parameter gibt an, wie viele Skatblätter verwendet werden. Die Methode soll eine Liste aller Karten der entsprechenden Zahl an Skatblättern gemischt zurückgeben, die man für Mau-Mau braucht. Beachten Sie, dass bei Mau-Mau die Karten mit Rank $2, 3, 4, 5$ und 6 nicht verwendet werden. Zum Mischen der Karten können Sie die Methode `shuffle()` der Klasse `java.util.Collections` verwenden.

   Beachten Sie hier die Testklasse `MauMauDeckTest`.

4 `MauMauGame`

Wenn man `MauMauGame` ohne Kommandozeilenparameter startet, dann wird mittels seiner (statischen) Methode `makeSimpleGame` ein einfaches Spiel mit einem Skatblatt und den zwei Spielerinnen Jacqueline und Chantal, die beide anfangs je fünf Karten auf die Hand bekommen, gestartet. Abb. 4 zeigt den Ablauf als Sequenzdiagramm. Dazu ruft `makeSimpleGame` die statische Methode `makeDeck(1)` vom `MauMauDeck` auf (Nachricht 3), die eine gemischte Liste von Spielkarten eines Skatblatts zurückgibt; in Aufgabe 2 haben Sie diese Methode realisiert.

Mit Nachricht 6 wird eine Instanz der Klasse `MauMau` mit dieser Liste als Kartenstapel und der Anzahl anfänglich den Spielerinnen auszuteilenden Karten (hier fünf) erzeugt. Mit den folgenden Methodenaufrufen werden `Player`-Instanzen für die beiden Mitspielerinnen erzeugt. Dabei wird jeweils auch die `addPlayer`-Methoden im `ActionHandler` aufgerufen. Sie sind im Sequenzdiagramm weggelassen.

Anschließend wird mit Nachricht 14 das Spiel gestartet. Die `startGame`-Methode von `MauMau` wird den Spielerinnen jeweils fünf Karten geben, das ist im Sequenzdiagramm aber ebenfalls nicht dargestellt, weil Sie das im Zuge der Realisierung der Klasse `ActionHandler` umsetzen sollen (Aufgabe 5). Zuletzt wird für jede Mitspielerin ein Fenster als Instanz von `PlayerFrame` erzeugt und mittels `setVisible` auf dem Bildschirm dargestellt.

Die `main`-Methode von `MauMauGame` endet mit Nachricht 24. Das laufende Programm terminiert aber nicht, da die Fenster der beiden Spielerinnen aktiv sind und Nutzerinteraktionen ermöglichen. Das Spiel funktioniert nun in ähnlicher Weise, wie Programme mit graphischer Benutzerschnittstelle im Vorlesungskapitel 8 besprochen worden sind: Das Betätigen eines Knopfes in einem der Fenster führt dazu, dass vom zugehörigen `Player`-Objekt die zugehörige Methode ausgeführt wird. Drückt man beispielsweise den Knopf *"Mau"* in Jacquelines Fenster, dann wird die Methode `mau()` des Objekts `p1` ausgeführt (Suchen Sie den entsprechenden Programmcode in der Klasse `PlayerFrame`). Andererseits werden die `PlayerFrame`-Objekte, die das `IObserver`-Interface implementieren, unter Nutzung des Observer-Musters (siehe Vorlesungskapitel 8) über jede Änderung des Modells benachrichtigt. Inspizieren Sie dazu die `update`-Methode der Klasse `PlayerFrame`. Diese Methode wird (über das `IObserver`-Interface) immer dann vom `MauMau`-Objekt aufgerufen, wenn die Darstellung im Fenster an das geänderte Modell angepasst werden muss.

5 `IObserver`

Listing 1 zeigt das Interface `IObserver`. Die `update`-Methode muss bei jeder Änderung im Modell aufgerufen werden, die `message`-Methode führt in ihrer Implementierung in `PlayerFrame` dagegen dazu, dass die übergebene Zeichenkette bei jeder Spielerin als Dialog angezeigt wird und von ihr bestätigt werden muss. Beispielsweise kann man diese Methode nutzen, um alle darüber zu informieren, dass eine Spielerin Mau-Mau zu sagen vergessen hat.

6 `Player` and `MauMau`

Die Klassen `Player` und `MauMau` sind vollständig vorgegeben. Ihre Methoden werden von der GUI in `PlayerFrame` aufgerufen, wenn man beispielsweise einen Knopf betätigt. Der jeweilige Zweck der Methoden und die gegebenenfalls zugehörigen Parameter sowie Rückgabewerte sollten sich aus den JavaDoc-Kommentaren erschließen.

Beide Klassen stellen die Facade des Modells (im Sine des MVC-Musters) dar, um einzelne Spielerinnen und das Spiel als ganzes zu repräsentieren. `PlayerFrame` verwaltet die Karten, die die Spielerin auf der Hand hält, weitere Zustandsattribute und konkrete Funktionalität stellen beide Klassen aber nicht zur Verfügung. Vielmehr dienen sie als Facade für das ganze Modell und delegieren die meisten Methodenaufrufe an Instanzen der Klassen `PlayerHandler`, `CardHandler` sowie `ActionHandler`, auf die die folgenden Abschnitte eingehen. Es wäre durchaus möglich, auf diese drei Klassen zu verzichten und ihre Funktionen komplett in `MauMau` zu realisieren,

Listing 1: Vorgegebenes Interface `IObserver`

```java
/**
 * Interface for observing changes in the Mau-Mau game.
 */
public interface IObserver {
    /**
     * Method called to notify the observer of a general update in the game.
     */
    void update();

    /**
     * Method called to send a message to the observer.
     *
     * @param msg The message to be sent.
     */
    void message(String msg);
}
```

das würde aber diese Klasse überfrachten und dem Entwurfsprinzip widersprechen, dass Klassen nur einem einzigen Zweck dienen sollen,[3] wodurch der objektorientierte Entwurf schwerer verständlich und schlechter zu warten wäre.

7 `CardHandler`

`CardHandler` verwaltet die Spielkarten, die von keiner Spielerin auf der Hand gehalten werden und daher entweder auf dem Ablage- oder dem Zugstapel liegen. Die beiden Stapel sind als `LinkedList`-Attribute realisiert. Die Klasse stellt ferner Methoden bereit, um Karten auf dem Ablagestapel abzulegen (`discardCard`, die von der Methode `playCard` von `Player` genutzt wird), und eine Karte vom Zugstapel aufzunehmen (`drawCard`).

Die letztgenannte Methode kümmert sich auch um die Situation, wenn der Zugstapel leergezogen wurde. In diesem Fall werden mit Ausnahme der obersten alle Karten des Ablagestapels als Zugstapel wiederverwendet (Methode `reuseDiscardedCards`). Natürlich werden die Karten zuvor gemischt. Außerdem kümmert sich die Methode auch um den Fall, dass die Zahl der Karten nicht mehr ausreicht, weil der Zugstapel leer ist und der Ablagestapel nur noch aus der obersten Karte besteht. In diesem Fall muss das Spiel abgebrochen werden, realisiert durch den Methodenaufruf

```
game.getActionHandler().cancelGame();
```

in `drawCard`. Die Methode `cancelGame` wird in `ActionHandler` realisiert. Sie ist außerdem ein Beispiel für das *single responsibility principle*: `CardHandler` ist für die Verwaltung der Spielkarten verantwortlich, aber eben nicht für den Spielablauf, wofür wiederum `ActionHandler` zuständig ist.

[3] *Single responsibility principle*: „A class should have one and only one reason to change, meaning that a class should have only one job.“

Aufgabe 3 (`CardHandler`)

Ergänzen Sie die Implementierung der Methode `dealCards`, die in der Klasse `CardHandler` noch fehlt. Ihre Aufgabe besteht darin, allen Mitspielerinnen reihum nacheinander jeweils eine Karte vom Zugstapel zu geben, bis jede Spielerin die benötigte Zahl an Karten hat. Zum Schluss muss noch eine Karte vom Zugstapel genommen und als oberste Karte auf den Ablagestapel gelegt werden.

Hinweise:

- Überlegen Sie sich zuerst, wie Sie in der Methode `dealCards` Zugriff auf die Liste aller Mitspielerinnen erhalten. Ein Blick auf das Klassendiagramm in Abb. 3 und auf die von den anderen Klassen zur Verfügung gestellten Methoden hilft dabei.
- Finden Sie heraus, wie viele Karten jede Mitspielerin ausgeteilt bekommen muss und wo Sie diese Information finden.
- Beachten Sie, dass eine Spielerin mittels der Methode `drawCards` in `Player` entsprechend viele Karten vom Zugstapel nimmt.
- Beachten Sie ferner die Testklasse `CardHandlerTest`.

8 `PlayerHandler`

Die Klasse `PlayerHandler` ist für die Verwaltung der Mitspielerinnen zuständig. Dazu nutzt sie zwei Listen: Die Liste im Attribut `ranking` enthält diejenigen Spielerinnen, die bereits alle ihre Karten ablegen konnten und damit ihr Spiel beendet haben. Die Reihenfolge in dieser Liste entspricht der, in der diese Spielerinnen ihr Spiel beenden konnten. Die Liste im Attribut `players` enthält hingegen die Mitspielerinnen, die aktuell noch aktiv mitspielen und noch nicht alle ihre Karten ablegen konnten. Solange diese Liste nicht leer ist, ist das gesamte Spiel noch nicht beendet (es sei denn, es wurde mangels genügend vieler Karten bereits abgebrochen; siehe Abschnitt 7.) Andernfalls gibt `ranking` wieder, wer das Spiel gewonnen hat, wer die Zweitplatzierte ist usf.

Die Reihenfolge der Liste in `players` entspricht der, in der sie an die Reihe kommen. Die erstgenannte ist die, die aktuell am Zug ist. Sie wird von der Liste entfernt, wenn sie ihren Zug abgeschlossen hat, und wieder hinten angestellt. Damit ist die nächste Spielerin an der Reihe. Hat die vorherige aber eine Acht gelegt, muss sich auch diese wieder hintan anstellen. Außerdem soll `PlayerHandler` sich darum kümmern, dass eine Spielerin, die ihre vorletzte oder letzte Karte gelegt hat und dementsprechend „Mau“ bzw. „Mau-Mau“ sagen muss, dem nachgekommen ist, bevor die nächste Spielerin ihren Zug abgeschlossen hat. Wenn sie das vergessen hat, muss sie eine Karte vom Zugstapel nehmen (bzw. die Karte wird ihr automatisch zugeteilt). Das Zustandsdiagramm in Abb. 5 beschreibt das Verhalten jeder `PlayerHandler`-Instanz.

Im Folgenden sind die verwendeten Ereignisse (d.h. Methoden in `PlayerHandler`), die zusätzlichen Bedingungen (jeweils in eckigen Klammern) und die Aktionen (jeweils nach dem Schrägstrich) erklärt. Beachten Sie, dass Transitionen ohne Ereignis sofort schalten, wenn ihre Bedingung in eckigen Klammern erfüllt ist. Transitionen ohne Ereignis und ohne zusätzliche Bedingung schalten sofort.

Gegenüber den Zustandsdiagrammen, die Sie in der Vorlesung kennengelernt haben, werden hier auch Fallunterscheidungen verwendet, dargestellt durch Rauten. Wenn eine Transition in

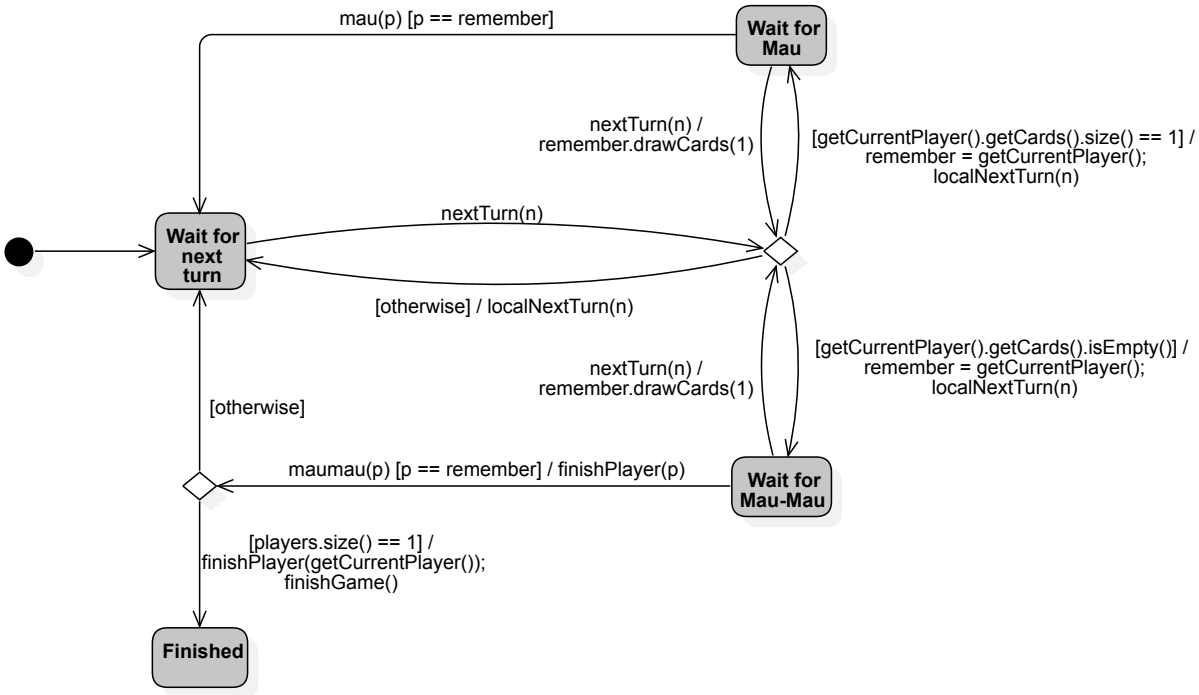

Abbildung 5: Zustandsdiagramm des `PlayerHandlers`.

eine solche einläuft, wird sie mit genau einer auslaufenden Transition fortgesetzt. Dazu sind alle auslaufenden Transitionen mit Bedingungen in eckigen Klammern versehen. Es muss sichergestellt sein, dass immer genau eine dieser Bedingungen erfüllt ist; die entsprechende Transition wird dann verwendet. Eine mit der Bedingung *[otherwise]* versehene Transition wird verwendet, wenn keine der Bedingungen der anderen Transitionen erfüllt ist. Die auslaufenden Transitionen dürfen Aktionen auslösen (nach dem Schrägstrich angegeben), aber nicht mit Ereignissen versehen sein (vgl. dazu UML-Spezifikationen 2.5.1 - Kap. 14.2.4 StateMachine).

Die nicht-privaten Methoden werden von anderen Objekten aufgerufen, beispielsweise `mau` von `Player`; ihre Bedeutung erschließt sich aus den JavaDoc-Kommentaren. Die privaten Methoden werden von den Transitionen aufgerufen und haben die folgende Bedeutung:

localNextTurn(int n) wiederholt n-mal die folgende Aktion: Die in `players` zuerst aufgeführte Spielerin wird aus dieser Liste entfernt und wieder hinten eingefügt. Auf diese Weise hat der Aufruf von `nextTurn(`n`)` zur Folge, dass mit dem daraus resultierenden Aufruf von `localNextTurn(`n`)` die nächste Spielerin (wenn $n = 1$) bzw. die übernächste Spielerin (wenn $n = 2$) an der Reihe ist.

finishPlayer(Player p) wird aufgerufen, wenn Spielerin p ihr Spiel erfolgreich beendet hat. Demzufolge soll diese Methode die Spielerin p aus der `players`-Liste entfernen und stattdessen hinten in die `ranking`-Liste einfügen.

Aufgabe 4 (`PlayerHandler`)

Ergänzen Sie die Implementierung der noch nicht realisierten Methoden in `PlayerHandler` an Hand des Zustandsdiagramms in Abb. 5. Hier empfiehlt sich die Anwendung des *State Patterns*, wofür Sie natürlich auch zusätzliche Klassen realisieren müssen.

9 `ActionHandler`

Die Klasse `ActionHandler` verwaltet den gesamten Spielablauf und ist in Abb. 6 als Zustandsdiagramm angegeben. Das Spiel läuft im zusammengesetzten Zustand GamePlay, in den Zuständen Game finished und Game canceled endet das Spiel regulär, wenn alle bis auf eine Spielerin ihre Karten ablegen konnten bzw. im Fall eines Abbruchs, weil die Zahl der zur Verfügung stehenden Karten nicht ausreicht (siehe Abschnitt 7).

Im Zustand GamePlay beginnt das Spiel im Zustand Initialized, in dem die Mitspielerinnen durch Aufruf von `addPlayer` hinzugefügt werden. Nur in diesem Zustand sind `addPlayer`-Aufrufe erlaubt und führen dazu, dass die entsprechende Methode im `PlayHandler` aufgerufen wird. Das ist erneut ein Beispiel des *Single responsibility principles*: `ActionHandler` steuert, wann mit `addPlayer` neue Spielerinnen zum Spiel hinzugefügt werden können, die tatsächliche Verwaltung der Spielerinnen übernimmt aber `PlayerHandler`.

Das Spiel beginnt dann mit Aufruf der Methode `startGame`. Ab diesem Zeitpunkt kann keine Mitspielerin über `addPlayer` hinzugefügt werden. Dem Zustandsdiagramm kann außerdem entnommen werden, dass `startGame` einen Aufruf von `dealCards` zur Folge hat, mit dem den Mitspielerinnen ihre Karten ausgeteilt werden. Die Methode `dealCards` ist in `ActionHandler` nicht zu finden, es gibt sie aber in der Klasse `CardHandler`, d.h. Sie müssen diese Methode in geeigneter Weise aufrufen. Im Folgenden ist eine kurze Erläuterung der in Abb. 6 verwendeten Methoden zu finden:

dealCards() ruft die gleichnamige Methode in `CardHandler` auf.

drawCards(n), playCard(c) rufen die gleichnamigen Methoden des `Player`-Objekts der Mitspielerin auf, die aktuell am Zug ist. Welche das ist, verwaltet `PlayerHandler`.

nextTurn(n) ruft die gleichnamige Methode von `PlayerHandler` auf.

canPlay(c) wird mit einer Karte als Parameter aufgerufen und soll `true` oder `false` zurückgeben, je nachdem, ob die Karte c im aktuellen Zustand gespielt werden darf oder nicht. Beispielsweise darf `canPlay(`c`)` im Zustand Suit chosen nur dann `true` zurückgeben, wenn die Farbe der Karte c mit der zuvor gewählten Farbe (abzufragen über die Methode `getChosenSuit`) übereinstimmt, aber kein Bube ist (weil Buben nicht auf Buben gelegt werden dürfen.) Diese Methode müssen Sie in der folgenden Aufgabe realisieren.

Beachten Sie, dass die Ereignisse in Form der Methodenaufrufe, die in den Transitionen vor dem Schrägstrich angegeben sind, jeweils Methodenaufrufe für die aktuelle Spielerin (die, die am Zug ist) sind, dass aber die Aufrufe von `nextTurn(`n`)` die aktuelle Spielerin ändern. Hierfür ist der `PlayerHandler` verantwortlich (*Single responsibility principle*!).

Beachten Sie ferner, dass die in Abb. 6 in eckigen Klammern angegebenen Bedingungen teilweise einer vereinfachten Syntax folgen. Beispielsweise entspricht die Bedingung `[c.rank()==8]` in Java tatsächlichen der Bedingung `c.rank() == Rank.EIGHT`.

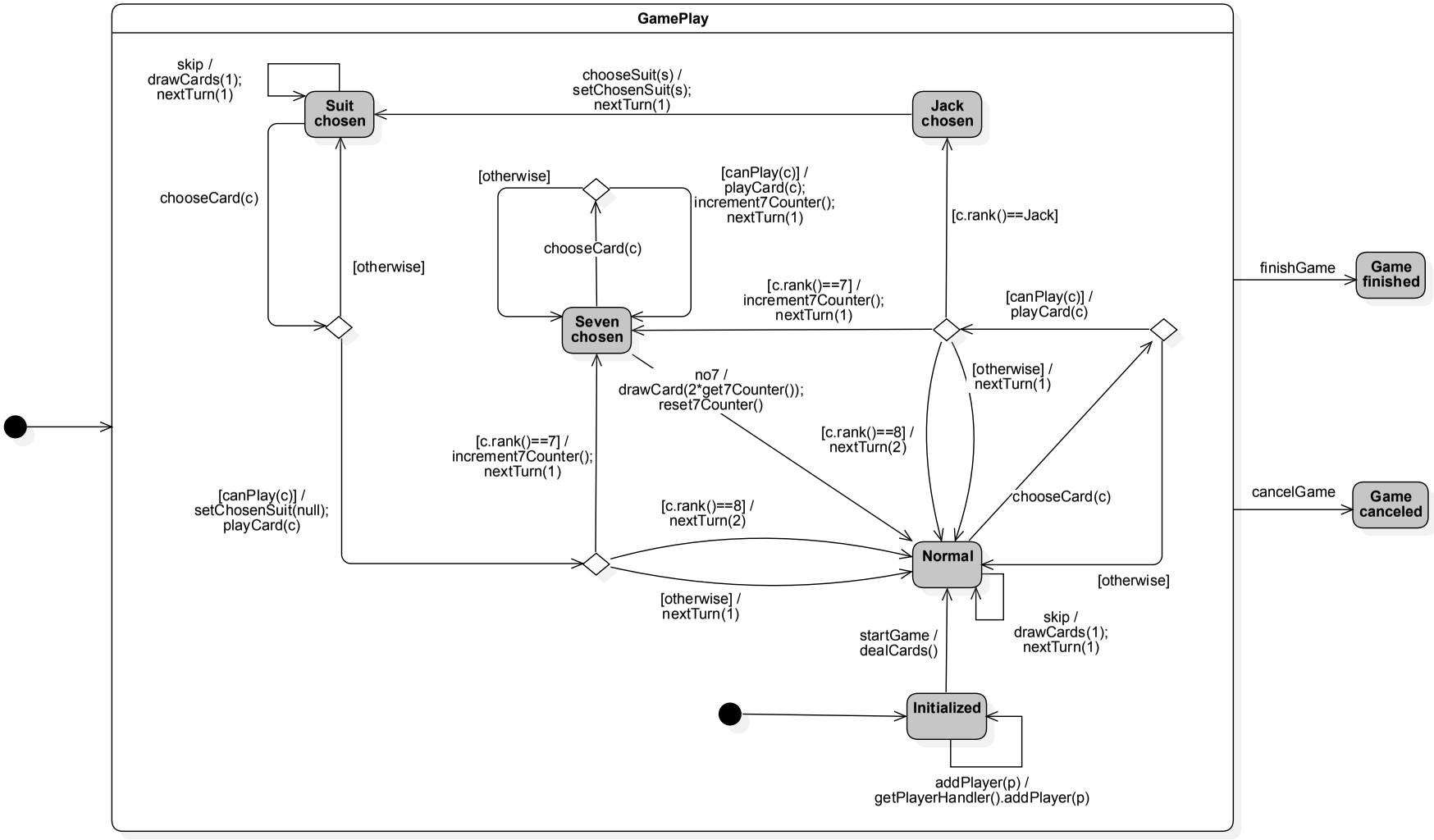

Abbildung 6: Zustandsdiagramm des `ActionHandlers`.

Listing 2: Vorgegebener Aufzählungstyp `GameState`

```java
/**
 * Represents the state of the Mau-Mau game.
 */
public enum GameState {
    /**
     * The game has been initialized, but has not yet started.
     */
    GAME_INITIALIZED,
    /**
     * The game is over. The final ranking of players can be
     * obtained using {@link cards.maumau.IMauMau#getRanking()}.
     */
    GAME_OVER,
    /**
     * The game has been canceled due to insufficient cards.
     */
    GAME_CANCELED,
    /**
     * The game is currently in progress with players taking turns.
     */
    PLAY,
    /**
     * The game is in progress and the current player has played
     * a Jack, and is required to choose a suit.
     */
    CHOOSE_SUIT
}
```

Tabelle 1: Zuordnung von `GameState`-Werten zu den einzelnen Zuständen in Abb. 6.

`GameState`-Wert	korrespondierender Zustand in Abb. 6
`GAME_INITIALIZED`	Initialized
`GAME_OVER`	Game finished
`GAME_CANCELED`	Game canceled
`PLAY`	Normal, Suit chosen oder Seven chosen
`CHOOSE_SUIT`	Jack chosen

Zuletzt muss noch auf die Methode `getGameState` in der Klasse `ActionHandler` eingegangen werden. Sie kommt im Zustandsdiagramm nicht vor, wird aber von der gleichnamigen Methode in der Klasse `MauMau` aufgerufen. Ihre Aufgabe ist es, zu jedem Zeitpunkt des Spiels Auskunft über den Zustand des Spiels, in Form des Aufzählungstyps `GameState` (siehe Listing 2), zu geben. Tabelle 1 fasst zusammen, welchen Wert die `getGameState`-Methode zurückgeben soll, wenn der Automat in Abb. 6 im jeweiligen Zustand ist. Dabei vernachlässigen wir den zusammengesetzten Zustand GamePlay.

Aufgabe 5 (`ActionHandler`)

Ergänzen Sie die Implementierung der noch nicht realisierten Methoden in `ActionHandler` an Hand des Zustandsdiagramms in Abb. 6 und den Erläuterungen zuvor. Auch hier empfiehlt sich wieder die Anwendung des *State Patterns*.

Wenn Sie diese Aufgabe abgeschlossen haben, müssen alle Testfälle erfolgreich durchlaufen.