Delegation

Aufgabe 6:

Betrachten Sie noch einmal die Klasse `Rational` aus der Vorlesung. Implementieren Sie eine Methode `doubleValue()`, die den repräsentierten Wert als `double` liefert. Zahlen werden in den einzelnen Ländern unterschiedlich dargestellt. So werden in England die Tausender durch Kommata getrennt, in Deutschland durch Punkte:

`1,234,567`	England
`1.234.567`	Deutschland

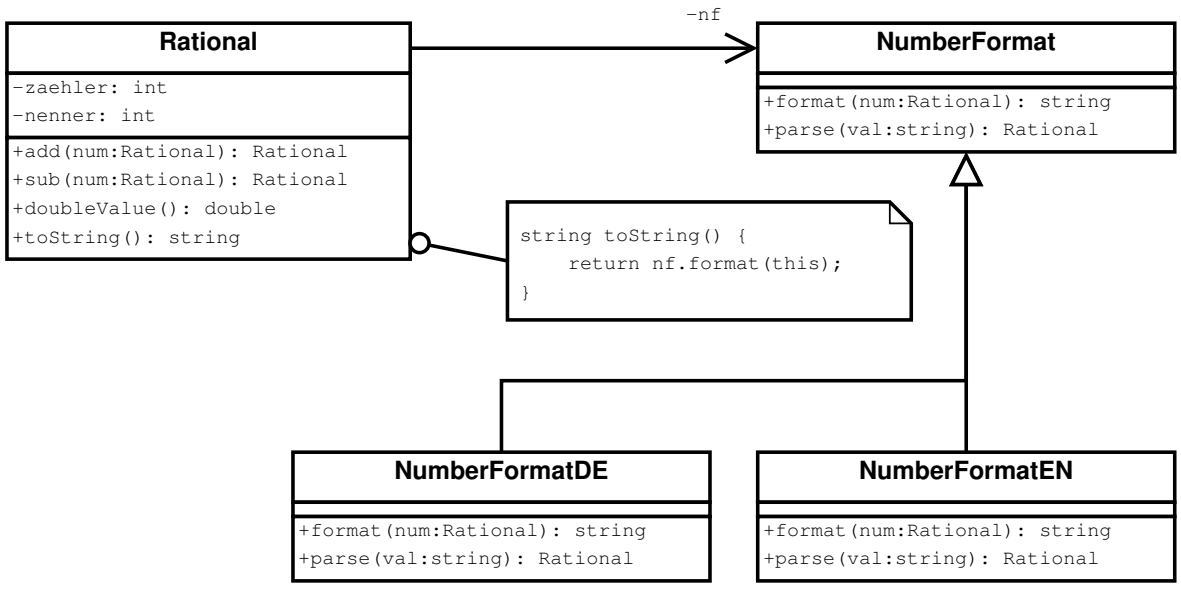
Andererseits werden in Deutschland die Nachkommastellen durch ein Komma vom ganzzahligen Teil getrennt, in England durch einen Punkt:

`742.13`	England
`742,13`	Deutschland

Damit wir `Rational`-Werte in verschiedenen Formaten ausgeben bzw. einlesen können, erstellen wir eine Klasse `NumberFormat`.

- Die Methode `format` der Klasse `NumberFormat` erstellt zum angegebenen `Rational`-Wert eine `string`-Darstellung, die dann ausgegeben werden kann.
- Dahingegen wandelt die Methode `parse` den gegebenen `string` in einen `Rational`-Wert. Falls der `string` keine korrekte `Rational`-Repräsentation darstellt, soll eine Ausnahme geworfen werden.

Leiten Sie von der Basisklasse `NumberFormat` zwei Klassen `NumberFormatDE` und `NumberFormatEN` ab, die eine Ausgabe im deutschen bzw. englischen Format ausgeben. Welche Methoden sollten abstrakt, welche als `virtual` oder `static` definiert sein? Ergänzen Sie das Klassendiagramm.



Aufgabe 7:

Unter *Wrapping* versteht man das Verpacken von Daten und Funktionalität eines im prozeduralen Programmierstils geschriebenen Systems in eine objektorientierte Schnittstelle. So kann bspw. eine prozedurale Bibliothek an ein objektorientiertes Softwaresystem angepasst werden. Wir stellen nach außen eine objektorientierte Schnittstelle bereit und verbergen alle internen Details, also die Verwendung der alten Prozeduren, Funktionen und Daten, innerhalb des Wrappers.

> In der Vorlesung wurde eine Implementierung eines String-Tokenizers besprochen, die verschiedene Methoden der Klasse `string` nutzt. Implementieren Sie jetzt eine Klasse `Tokenizer`, die die Funktionalität der C-Funktion `strtok` kapselt und als Klasse zur Verfügung stellt.

Aufgabe 8:

Eine Bank verwaltet Konten, wobei jedes Konto einer Person zugeordnet ist. Leiten Sie von der Klasse `Konto` zwei Klassen `Girokonto` und `Sparkonto` ab. Folgende Funktionen haben alle Kontenarten gemeinsam:

```
void hebeAb(int betrag, Datum d);
void zahleEin(int betrag, Datum d);
int kontostand();
string kontoauszug();
```

Bei einem Sparkonto darf nur dann Geld abgehoben werden, wenn der Kontostand dadurch nicht negativ wird. Bei einem Girokonto darf der Kontostand zwar negativ werden, aber nur bis zum angegebenen `dispo`-Wert, dem Kreditlimit.

Das Guthaben auf einem Sparkonto wird verzinst, beim Girokonto gibt es dagegen nur einen Sollzins, also einen Zinssatz für Überziehungen. Außerdem stellt das Girokonto eine `ueberweise`-Methode bereit:

```
void ueberweise(string kontonr, int betrag, Datum d);
```

Implementieren Sie obige Klassen. Überlegen Sie sich, welche Attribute die Klassen haben und welche davon in der Basisklasse deklariert werden können. Welche der Methoden müssen als `virtual` deklariert und in einer Unterklasse überschrieben werden? Welche Methoden sind abstrakt? Müssen weitere Klassen oder Strukturen definiert werden? Wie werden Kontobewegungen vermerkt? Welche Methoden stellt die Bank bereit? Wie könnte eine Benutzeroberfläche aussehen?

Aufgabe 9: Just for fun!

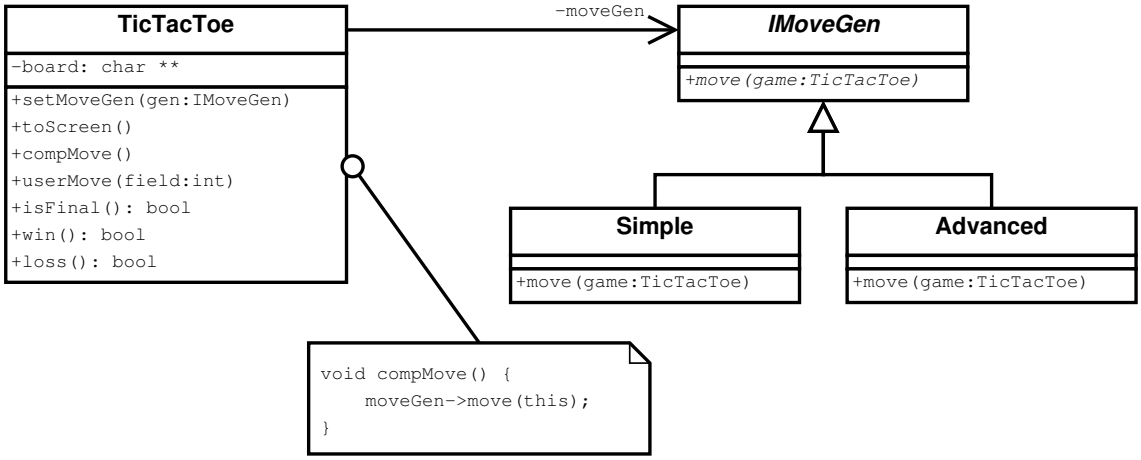
Erstellen Sie ein Programm, gegen das wir „Tic-Tac-Toe“ spielen können. Das Spielbrett kann mittels einfacher ASCII-Grafik dargestellt und Interaktion kann mittels der `curses`-Bibliothek implementiert werden. Ein möglicher Spielverlauf kann bspw. so aussehen:

```
|1|2|3|          |1|2|3|          |1|2|3|          |x|2|3|
|4|5|6|          |x|5|6|          |x|5|6|          |x|5|6|
|7|8|9|          |7|8|9|          |7|8|o|          |7|8|o|

Gamer: 4         Computer: 9      Gamer: 1         Computer: 7
```

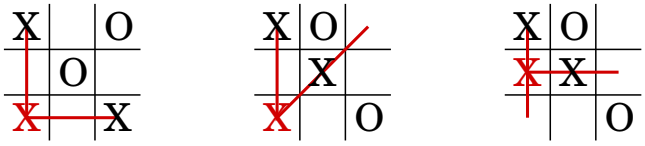
Als erste primitive Spielstrategie können Sie einen Zufallsspieler implementieren, der unter allen erlaubten Zügen einen zufällig auswählt. Diesen Gegner sollten Sie noch leicht schlagen können.

Anschließend implementieren wir eine Klasse, die bessere Züge generiert. Es bietet sich an, eine Oberklasse `IMoveGen` zu definieren, die die Schnittstelle eines allgemeinen Zuggenerators definiert. Davon können zwei Klassen `Simple` und `Advanced` abgeleitet werden.



Bei komplexeren Spielen wie Dame, Schach oder Go sind pro Spielphase verschiedene Zuggeneratoren denkbar. Ein Zuggenerator für die Eröffnung, einer für das Mittelspiel und einer für das Endspiel. Bei der verbesserten Version unseres Zuggenerators erzeugen wir einen Zug nach folgender Liste, die unter `http://en.wikipedia.org/wiki/Tic-tac-toe` beschrieben ist:

- Wenn es bereits eine eigene Reihe mit zwei besetzten Feldern gibt, besetze das dritte Feld der Reihe, um einen eigenen Sieg zu erzielen.
- Wenn der Gegner eine Reihe mit zwei besetzten Feldern hat, besetze das dritte Feld der Reihe, um den gegnerischen Sieg zu verhindern.
- Besetze ein Feld, sodass sich zwei Gewinnmöglichkeiten ergeben, ein sogenannter Fork. Der Gegner kann im nächsten Zug nur eine der Gewinnmöglichkeiten verhindern.



- Besetze ein Feld, sodass ein gegnerischer Fork, also zwei gegnerische Gewinnreihen, verhindert wird.
- Besetze das mittlere Feld.
- Wenn der Gegner ein Eckfeld besetzt, besetze das gegenüberliegende Eckfeld.
- Besetze ein Eckfeld.
- Besetze das mittlere Feld einer Seite.

Die Liste wird von oben nach unten durchlaufen, die erste Möglichkeit wird ausgeführt. Gegen dieses Programm sollten Sie schon deutlich mehr Mühe haben, es zu schlagen.

Aufgabe 10: Just for fun!

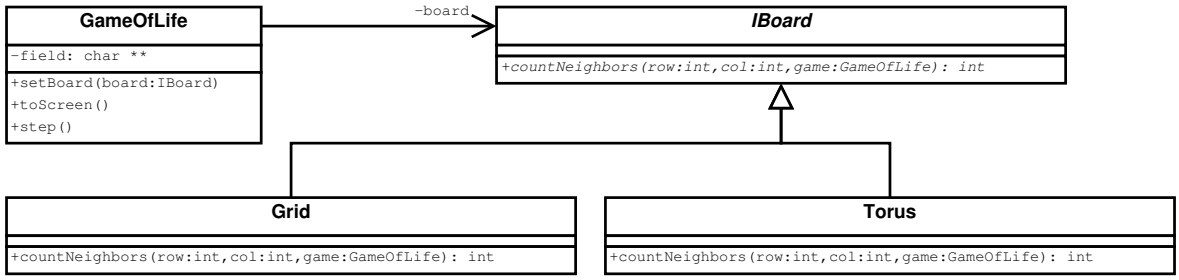
Implementieren Sie Conway's Game of Life mittels der `curses`-Bibliothek. Conway's Game of Life ist ein vom Mathematiker John Horton Conway 1970 entworfenes System, basierend auf einem unendlich großen, zweidimensionalen Gitter. Im Prinzip stellt es eine Computersimulation dar, bei der sich Lebewesen (Bakterien) auf den Feldern des Gitters verteilen. Je nach Nachbarschaft bleiben sie am Leben, sterben oder bringen neues Leben hervor.

Zunächst wird eine Anfangsgeneration von lebenden Zellen auf dem Spielfeld platziert. Die jeweils nächste Generation ergibt sich durch die Befolgung einfacher Regeln, wobei die Folgegeneration immer für alle Zellen gleichzeitig berechnet wird und die aktuelle Generation ersetzt.

- Überlebensregel: Lebende Zellen überleben, wenn sie zwei oder drei lebende Nachbarn haben.
- Sterberegel: Lebende Zellen mit weniger als zwei oder mehr als drei lebenden Nachbarn sterben an Einsamkeit bzw. Überbevölkerung.
- Geburtsregel: Tote Zellen mit genau drei lebenden Nachbarn werden neu geboren.

Die Anfangspopulation soll entweder zufällig erstellt oder aus einer Datei eingelesen werden.

Da ein reales Spielfeld immer einen Rand hat, muss das Verhalten dort festgelegt werden. Im einfachsten Fall belegen wir den Rand durch tote Zellen. Wir wollen aber unterschiedliche Strategien der Randbehandlung implementieren und austauschbar machen. Wenn wir ein Torus-förmiges Spielfeld verwenden, kommt alles, was das Spielfeld nach unten verlässt, oben wieder heraus und umgekehrt. Das Gleiche gilt für die Seiten: Alles, was das Spielfeld nach links verlässt, tritt rechts wieder ein und umgekehrt.



Stillleben sind stabile Populationen, die sich nicht ändern. Solche Populationen enthalten nur Lebewesen mit 2 oder 3 Nachbarn.

	X	
X		X
	X	

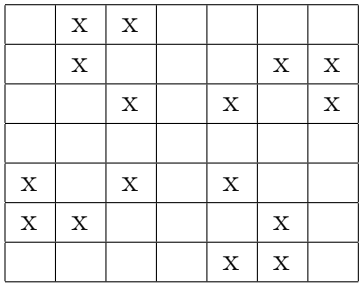
X	X	
X	X	

X	X	
X		X
	X	X

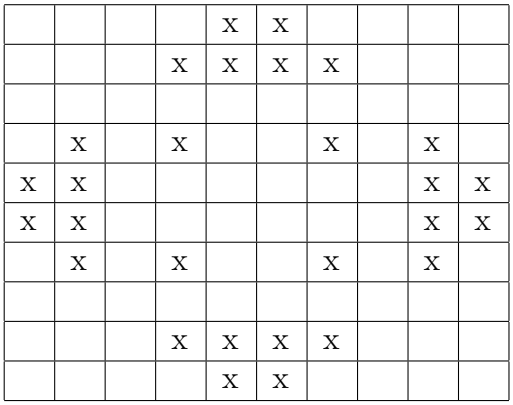
X	X		
X			X
		X	X

Oszillatoren sind Populationen, die sich nach gewissen Perioden wiederholen. Bei den P2-Oszillatoren wiederholen sich die Populationen nach zwei Perioden, bei P3-Oszillatoren nach drei Perioden.

P2-Oszillator:



P3-Oszillator



Es gibt Populationen, die sich über das Feld bewegen. Dazu gehören die Gleiter und die Raumschiffe.