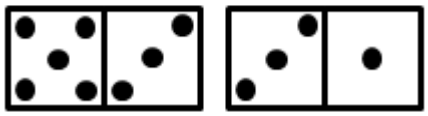
Szenario

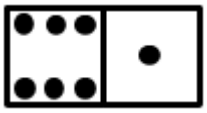
Die Informatik-AG möchte eine computergestützte Version des Spiel "Domino" entwickeln.



zwei passende Dominosteine

Dominosteine haben immer jeweils zwei Zahlen von 1 bis 6.

Domino ist ein Legespiel, d.h. man versucht seine eigenen Steine passend an Steine, die schon auf dem Tisch liegen, anzulegen. "Passend angelegt" bedeutet dabei, dass die benachbarten Zahlen gleich sind, wie im Bild rechts die beiden Dreien.



Dieser Stein lässt sich oben rechts anlegen - man muss ihn nur umdrehen!

Anlegen darf man immer nur an den beiden Enden der Kette; so entsteht auf dem Tisch eine lange Kette. Dabei darf man den Stein, den man anlegt auch umdrehen; vgl. der Stein "6-1" rechts.

Teile des computergestützten Domino-Spiels sollen in dieser Aufgabe modelliert und implementiert werden.

a)

Zeichnen Sie ein Klassendiagramm für die Klasse Dominostein gemäß der Klassendokumentation im Anhang. (4 Punkte)

Implementieren Sie die Klasse Dominostein. (7 Punkte)

Auf dem Tisch liegt beim Domino die Kette der abgelegten Dominosteine. Das soll in der computergestützten Spielvariante durch die Klasse `Tisch` abgebildet werden.

Die Klasse `Tisch` braucht also eine geeignete Datenstruktur, um die Kette der abgelegten Dominosteine zu speichern.

b) Erläutern Sie jeweils in einem Satz, warum Stacks, Queues und Arrays ungeeignet sind, um die Kette der abgelegten Dominosteine zu speichern. (6 Punkte)

Die Informatik-AG hat sich dafür entschieden, in der Klasse `Tisch` die Dominosteine in einem Attribut `kette` vom Typ `List` zu speichern.

c) Implementieren Sie für die Klasse `Tisch` die Methoden

```
public boolean anlegenVorne(Dominostein pStein)
```
und
```
public boolean ketteIstGueltig()
```
gemäß den Anforderungen der Dokumentation der Klasse `Tisch` (s. Anhang). (16 Punkte)

Abschließend soll das komplette Dominospiel in den Klassen `Dominospiel, Spieler` und Tisch modelliert werden.

Zu einem Dominospiel gehören bis zu vier Spieler. Zu Beginn bekommt jeder 5 Dominosteine; der Rest wird als Vorrat verdeckt auf den Tisch gelegt. Die Spieler sind der Reihe nach am Zug. Wer am Zug ist, wählt aus seinen Dominosteinen einen aus und entscheidet, ob der ausgewählte Stein vorne oder hinten an die Kette auf dem Tisch angelegt werden soll. Wenn der Stein nicht passt, dann bekommt der Spieler automatisch einen weiteren Dominostein vom Vorrat (solange der nicht leer ist). Wer zuerst alle seine Steine anlegen konnte, hat gewonnen.

d)

Zeichnen Sie gemäß diesen Anforderungen ein Implementationsdiagramm mit den Klassen `Dominospiel, Tisch, Spieler` und `Dominostein` und mit ggf. notwendigen Klassen zur Datenspeicherung. Attribute und Methoden brauchen in diesem Implementationsdiagramm nicht angegeben werden. (12 Punkte)

Erläutern Sie, wie man bei Ihrer Modellierung feststellen kann, welcher Spieler gerade am Zug ist und wie man bei Ihrer Modellierung realisieren kann, dass die Spieler reihum an den Zug kommen. (5 Punkte)

Die Klassen `Dominospiel`, `Spieler` und `Tisch` sollen jetzt geeignete graphische Oberflächen haben, um den jeweiligen Zustand des Spieles anzuzeigen. Außerdem gibt es in der Klasse `Spieler` die Methoden

- `public Dominostein steinAuswaehlen()` : ermöglicht die Auswahl eines Dominosteines aus den eigenen Steinen. Dieser Stein wird im Attribut `gewaehlterStein` gespeichert.
- `public boolean vorneWaehlen()` : ermöglicht dem Spieler zu entscheiden, ob der zuvor ausgewählte Stein vorne oder hinten an die Kette auf dem Tisch angelegt werden soll. Wenn sich der Spieler für vorne entscheidet, dann gibt die Methode `true` zurück, sonst `false`.

Darüber hinaus hat die Klasse `Dominospiel` eine Methode `spielAblauf()`, die den Ablauf einer ganzen Spielrunde realisiert.

e)

Zeichnen Sie Klassendiagramme für die Klassen `Dominospiel, Spieler` und `Tisch` mit allen notwendigen Attributen und Methoden, die für die Realisierung des Spiels notwendig sind. Dabei soll in der Klasse `Dominospiel` die Zahl der Spieler im Konstruktor übergeben werden. Für die Modellierung soll das Geheimnisprinzip gewahrt werden.

Anhang

Dokumentation der Klasse `Dominostein`

In Objekten der Klasse `Dominostein` werden die beiden Zahlen des Dominosteins geeignet gespeichert und für den Abruf bereit gestellt.

Konstruktor	**`Dominostein(int pZahl1, int pZahl2)`** Erzeugt ein Dominostein-Objekt mit den im Parameter angegeben Zahlen.
Anfrage	**`public int gibZahl1()`** gibt die erste Zahl des Dominosteines zurück.
Anfrage	**`public int gibZahl2()`** gibt die zweite Zahl des Dominosteines zurück.
Auftrag	**`public void umdrehen()`** vertauscht die erste und die zweite Zahl. Damit kann man den Dominostein auch "umgedreht" anlegen.

Dokumentation der Klasse `Tisch`

Objekte der Klasse `Tisch` bilden den Tisch beim Dominospielen ab. Insbesondere verwalten Objekte der Klasse `Tisch` in einem Attribut `kette` vom Typ `List` die Dominosteine, die von den Spielern auf den Tisch gelegt wurden.

Konstruktor	**`Tisch()`** Erzeugt ein Tisch-Objekt; die `kette` ist leer.
Auftrag	**`public boolean anlegenVorne(Dominostein pStein)`** versucht den `Dominostein pStein` vorne an die `kette` anzulegen. Dabei wird `pStein` nötigenfalls umgedreht. Wenn `pStein` vorne an die `kette` passt, wird er zum ersten Glied der `kette` und es wird `true` zurückgegeben. Sonst wird lediglich `false`.zurückgegeben.
Auftrag	**`public boolean anlegenHinten(Dominostein pStein)`** versucht den `Dominostein pStein hinten` an die `kette` anzulegen. Dabei wird `pStein` nötigenfalls umgedreht. Wenn `pStein` hinten an die `kette` passt, wird er zum letzten Glied der `kette` und es wird `true` zurückgegeben. Sonst wird lediglich `false`.zurückgegeben.
Anfrage	**`public boolean ketteIstGueltig()`** ueberprüft, ob in `kette` die jeweils benachbarten Dominosteine die gleiche Zahl zeigen, und gibt dementsprechend `true` bzw. `false` zurück. Wenn `kette` leer ist oder nur einen Stein enthält, dann wird `true` zurückgegeben.