FAU, Informatik 2, AUD-Team
aud@i2.cs.fau.de

Bonus. Übung

Abgabe bis 13.01.2017, 10:00 Uhr

Frohes Fest und Guten Rutsch!

Das AuD-Team wünscht Ihnen ein Frohes Fest und einen Guten Rutsch in ein erfolgreiches und gesundes neues AuD-Jahr!

Einzelaufgabe Bonus.1: AuD2048 — 20 EP

Machen Sie sich theoretisch und praktisch mit dem Spiel 2048 vertraut – beachten Sie dabei aber bitte unbedingt, dass das Spiel süchtig machen kann ...
Laden Sie nun die Vorgaben `AuD2048.java` und `AuD2048Logic.java` von der AuD-Homepage: Erstere stellt Ihnen die GUI für verschiedene Varianten des Spiels 2048 zur Verfügung und letztere ist eine generische Schnittstelle für die Umsetzung unterschiedlicher Spiellogiken. Die Kommentare an den (abstrakten, also noch zu implementierenden) Methoden erklären jeweils ihren Zweck.

a) Setzen Sie in der Klasse `AuD2048LogicNormalGame` das „ursprüngliche Spiel“ um. Neue „Zufallszahlen“ sind entweder 2 oder 4, wobei die Wahrscheinlichkeit für eine 2 mit 75% im Mittel 3× höher ist, als für eine 4.

b) Implementieren Sie in der Klasse `AuD2048LogicEleven` die „Variante 11“, bei der benachbarte gleichwertige Felder zur nächst-höheren Zahl zusammengelegt werden, bis mindestens ein Feld den Wert 11 erreicht hat. Neue „Zufallszahlen“ sind entweder 1 oder 2, wobei die Wahrscheinlichkeit für eine 1 mit 75% im Mittel 3× höher ist, als für eine 2.

c) Erstellen Sie in der Klasse `AuD2048LogicFibonacci` eine „Fibonacci-Variante“, bei der benachbarte Felder nur dann zusammengelegt (und dabei addiert) werden, wenn es aufeinanderfolgende Fibonacci-Zahlen sind, bis mindestens ein Feld den Wert 2584 erreicht hat. Neue „Zufallszahlen“ sind entweder 1 oder 2, wobei die Wahrscheinlichkeit für eine 1 mit 75% im Mittel 3× so hoch ist, wie für eine 2.

Tipp: Die obigen Spielvarianten haben (abgesehen von der Schnittstelle) noch viel mehr gemeinsam: Nutzen Sie Vererbung, um Code-Duplikation zu vermeiden, indem Sie gemeinsame Hilfsmethoden in der „Zwischenklasse“ `AuD2048LogicCommon` ablegen.
Sie können Ihre jeweiligen Implementierungen ~~spielen~~ testen, indem Sie `AuD2048` mit dem Namen der Spiellogik-Klasse und/oder anderen Spielfeldgrößen (!) aufrufen.

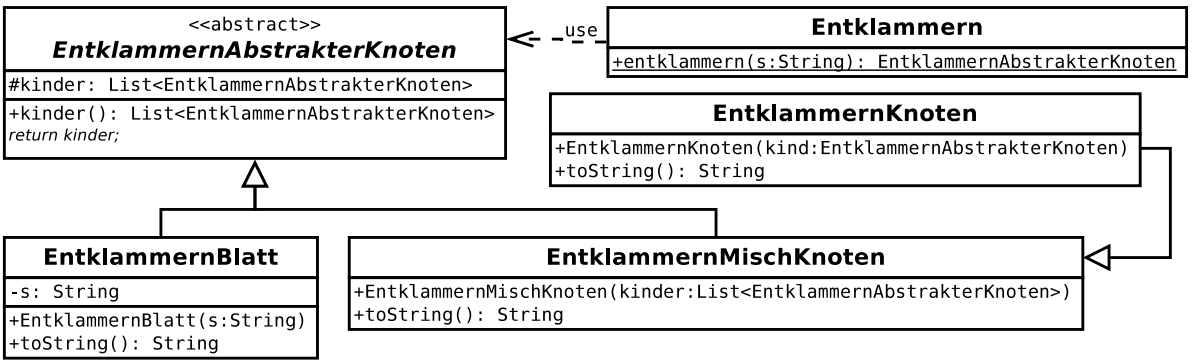
Einzelaufgabe Bonus.2: ((Je)de) (M(eng)e) (K(l)a((mm)er)n) — 20 EP

a) Implementieren Sie eine Klasse `Klammern` mit einer Methode `klammern` die eine Zeichenkette `s` erhält und eine Liste aller möglichen Klammerungen der Zeichen von `s` zurückgibt:

`"ab"` ⟼ `[ab, (a)b, ((a)(b)), a(b), (ab), (a)(b), ((a)b), (a(b))]`

Ist `s` leer oder `null`, so muss auch das Ergebnis eine leere Liste sein. Die Reihenfolge der Elemente in der Ergebnisliste ist irrelevant. *Wichtig* aber ist, dass *kein* Element in der Ergebnisliste mehrmals vorkommt und dass *kein* Klammerpaar leer bleibt oder ein anderes Klammerpaar direkt umschließt, also z.B. *nicht* „~~()~~“, *nicht* „~~((a))b~~“ und auch *nicht* „~~(((a)(b)))~~“.

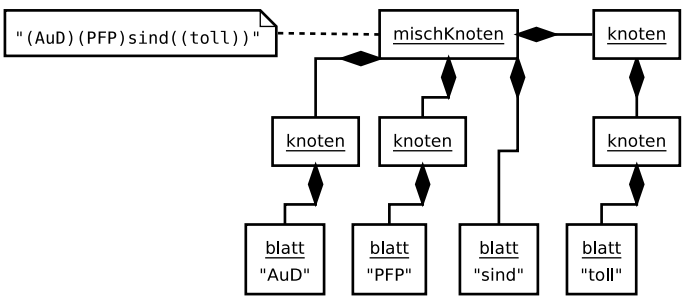
b) Im Folgenden sollen Sie nun das „Gegenteil“ der vorangehenden Teilaufgabe programmieren: Sie sollen Klammerstrukturen parsen und in eine Baumstruktur überführen. Implementieren Sie die allesamt öffentlichen Klassen *exakt* nach dem folgenden UML-Diagramm:



c) Implementieren Sie in der Methode `Entklammern.entklammern` einen einfachen Parser, der die Klammerstrukturen in der übergebenen Zeichenkette zerlegt und diese Zerlegung (hierarchisch gemäß ihrer Klammerung) als Baum zurückgibt. Dabei gilt:

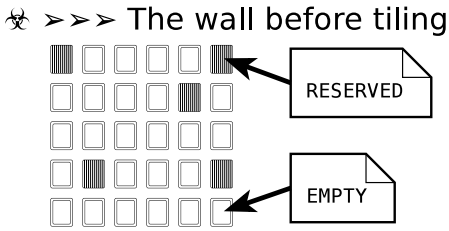
- Ist die Eingabe `null` oder treten die Klammern nicht mathematisch korrekt und paarweise geschachtelt auf, muss die Methode eine `IllegalArgumentException` werfen. Im Gegensatz zu **a)** ist hierbei Mehrfachklammern wie z.B. „`(A)u((D))`“ erlaubt.
- Die Zerlegung erfolgt *ausschließlich* an den Klammern – alles davor, dazwischen oder danach wird zusammenhängend als Zeichenkette betrachtet und durch einen Knoten vom Typ `EntklammernBlatt` dargestellt.
- Ein `EntklammernBlatt` mit einer leeren Zeichenkette darf nur auftreten, um entweder die gesamte Eingabe darzustellen, sofern sie eine leere Zeichenkette ist, oder um den „Inhalt“ eines leeren Klammerpaares „()“ zu repräsentieren – *zwischen* zwei aufeinander folgenden Klammerpaaren „)(“ gibt es *keinen* Leerstring.
- Jedes Klammerpaar der Form „$(\mathcal{X})$“ wird durch einen `EntklammernKnoten` dargestellt, der genau einen Kindknoten mit dem Baum zu „$\mathcal{X}$“ hat.
- Aneinandergereihte Klammerpaare oder gemischte Teile der gleichen Hierarchiestufe werden zu einem Knoten vom Typ `EntklammernMischKnoten` zusammengelegt, dessen Kinder jeweils die Teilausdrücke darstellen – diese Kinder können nur vom Typ `EntklammernBlatt` oder `EntklammernKnoten` sein.
- Die Reihenfolge der Kinder muss die Reihenfolge in der Eingabe wiedergeben – d.h. `toString` traversiert die Knoten *post-order* und erzeugt wieder die Eingabe.

Beispiel:



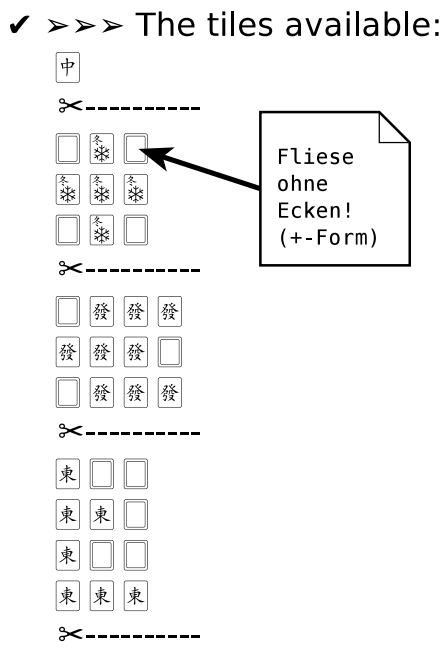
Einzelaufgabe Bonus.3: Wall·E - Der letzte fliest die Wand an... **20 EP**

Falls Sie mehr Zeit auf dem Christkindlesmarkt als mit Ihren AuD-Übungen verbracht haben, dann könnte Fliesenleger(in) für Sie eine Alternative zu Taxifahrer(in) sein... Als Vorgeschmack bekommen Sie nun eine beliebig große Wand vorgesetzt (z.B. wie in Abb. 1a), in der bereits einige Aussparungen für Steckdosen und Wasserhähne markiert sind (`AtomicTile.RESERVED`), die Sie *nicht zufliesen* dürfen – die restlichen „Kacheln" sind leere Platzhalter (`AtomicTile.EMPTY`).



(a) Leere Wand vor dem Fliesen

(b) Wand nach dem Fliesen



(c) Fliesenmodelle

Abbildung 1: Fliesenlegen

Ihnen stehen unterschiedliche und von jeder Sorte beliebig viele Designer-Fliesen zur Verfügung, wie exemplarisch in Abb. 1c gezeigt. Die dort leeren „Teilkacheln" sind nicht Bestandteil der Fliese selbst (sie überdecken also z.B. keine Aussparungen in der Wand oder andere Fliesen).
Sie sollen nun die leere Wand *optimal* verfliesen: Jede *leere Teilkachel muss bedeckt* sein, jede reservierte *Aussparung muss frei* bleiben, die Fliesen dürfen *nicht über den Rand* der Wand hinausragen (es gibt *keinen Verschnitt*), sie dürfen einander (auch teilweise) *nicht überdecken* und Sie sollen so sparsam wie möglich sein, d.h. auch wenn hier alle Fliesen gleich viel kosten, so sollen Sie insgesamt *möglichst wenige* davon verbrauchen. Abb. 1b zeigt das „fertige" Wandbeispiel mit 7 Fliesen.
Laden Sie zunächst die vorgegebenen Klassenrümpfe `WallE.zip` von der AuD-Homepage und machen Sie sich mit dem darin enthaltenen „legacy code" vertraut. Implementieren Sie die Methode `stickTiles` in der Klasse `TiledWall` so, dass *genau* die verwendeten Fliesen im Feld `stickyTiles` stehen. Falls es keine Lösung gibt, dann soll das Feld `null` bleiben; gibt es mehrere optimale Lösungen, dann genügt eine davon.
Zum Testen sollten Sie ein Betriebssystem mit guter Unicode-Unterstützung nutzen.
Geben Sie die von Ihnen geänderte Datei `TiledWall.java` über EST ab.

60 EP + 0 GP = 60 Punkte