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

8. Übung

Abgabe bis 16.12.2019, 10:00 Uhr

Einzelaufgabe 8.1: Kontrollflussorientierte Überdeckungstestverfahren — 15 EP

Beim sogenannten *Zweigüberdeckungstest* muss die Testfallmenge für eine Methode $\mathcal{P}$ so gewählt werden, dass jede Verzweigung des Kontrollflusses von $\mathcal{P}$ (siehe z.B. Abb. 1: Sie kennen bereits eine äquivalente Darstellung als Programmablaufplan) mindestens einmal durchlaufen wird, also jede Bedingung/Verzweigung jeweils mindestens einmal zu *wahr* und (ggf. in einem anderen Testfall) mindestens einmal zu *falsch* ausgewertet wird.
Um zu erfassen, welche Teile des Codes während der Ausführung der Testfälle überdeckt wurden, instrumentieren einige Werkzeuge den Code ähnlich wie in der vorgegebenen Methode `SchiffVersenken.neuesSpielfeld`. Die hier eingesetzten Aufrufe an `Log.log(...)` protokollieren die Ausführung derjenigen Anweisungsblöcke, die in Abb. 1 durch ebenso nummerierte Kreise dargestellt werden.

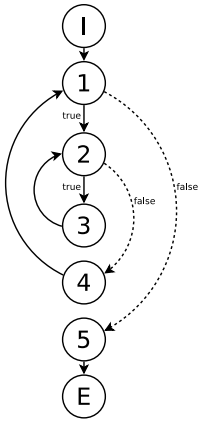


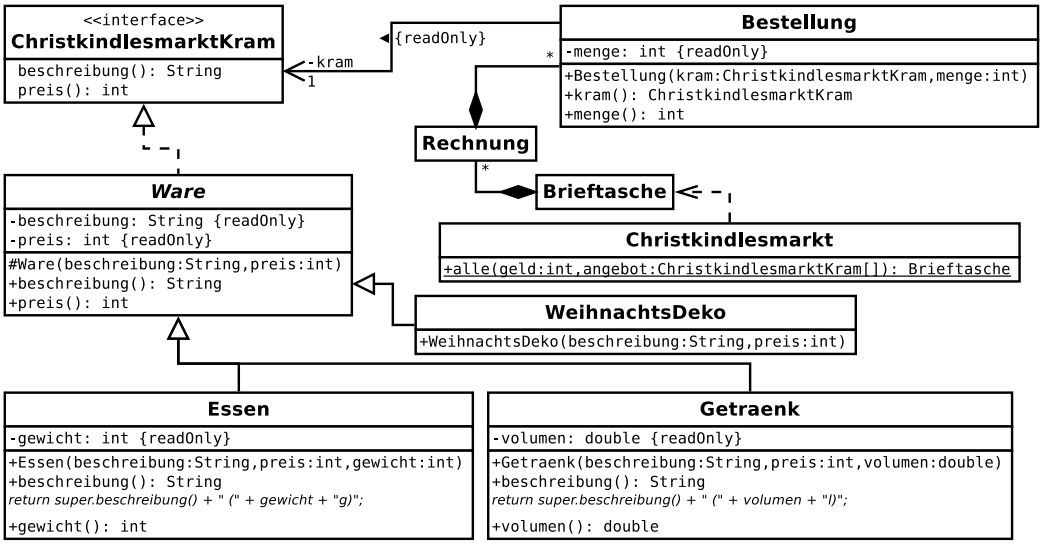
Abbildung 1: `neuesSpielfeld`

Implementieren Sie eine vereinfachte, an JUnit angelehnte Testfallklasse `SchiffVersenkenZweigueberdeckungsTest`. Die Testfälle müssen dabei durch nicht-statische Methoden *ohne* Argumente repräsentiert werden, die zusätzlich mit der *vorgegebenen* Annotation `Test` markiert werden (diese Annotation ist stark vereinfacht und unterstützt keine Parameter wie `timeout` oder Ausnahmen!). Nach Ausführung aller Testfälle sollten Sie eine ***vollständige*** Zweigüberdeckung ($C_1 = 100\%$) des ***gesamten*** „SUT“ erreicht haben. Prüfen Sie zusätzlich bei ***jedem*** Aufruf ***aller*** nicht-`void` Methoden ***sofort*** deren Rückgabewert mit der vorgegebenen Methode `Assert.assertEquals`.
Tipp: Ein guter Testfall ist minimalistisch! Sie sollten dementsprechend in jeder Testmethode *höchstens eine* Instanz des „SUT“ `SchiffVersenken` erstellen, danach bei Bedarf *schnell* in den gewünschten Zustand bringen und schließlich die zu testende Methode aufrufen.
Wichtige Hinweise: Nutzen Sie *keine* Klassen, Methoden oder Annotationen aus dem offiziellen JUnit-Paket, sonst wird Ihre Abgabe mit 0 Punkten bewertet, weil sie nicht alleine übersetzbar ist. Verwenden Sie stattdessen die vom AuD-Team vorgegebenen Annotationen/Klassen. Nach der Bearbeitung dieser Aufgabe sollten Sie diese aus dem Klassenpfad zukünftiger Aufgaben ausschließen, sonst werden Sie evtl. keine vorgegebenen öffentlichen Testfälle mehr ausführen können!

Einzelaufgabe 8.2: Gierig – Christkindlesmarkt — 18 EP

Stellen Sie sich vor, es ist Christkindlesmarkt in Nürnberg, und ~~keiner geht hin~~ Sie gehen hin, um Ihr ganzes Erspartes auf den Kopf zu hauen. Als fleißiger AuD-Teilnehmer betrachten Sie natürlich auch diese Aufgabe als spannendes algorithmisches Problem. Noch vor dem ersten Glühwein stellen Sie sich ein Array mit allem `ChristkindlesmarktKram` zusammen, den Sie auf dem Markt erstehen können. Die Festwirte garantieren Ihnen, dass Sie von jedem Nahrungsmittel und jeder `WeihnachtsDeko` beliebig viel haben dürfen, solange Sie dafür noch bezahlen können.

a) Implementieren Sie die Klassen (sofern sie nicht schon vorgegeben sind) aus dem folgenden UML-Diagramm ***EXAKT***: Beachten Sie dabei Sichtbarkeiten und zusätzliche Modifikatoren (z.B. `{readOnly}` $\widehat{=}$ `final`) sowie dass das Diagramm auch <<Schnittstellen>> und *abstrakte Klassen* enthält. Die Methoden sind entweder Getter des gleichnamigen Attributs oder als Kommentar vorgegeben. ***NUR*** in der Klasse `Christkindlesmarkt` dürfen Sie bei Bedarf zusätzliche `private` (!) Hilfsmethoden ergänzen.



b) Ergänzen Sie nun die Methode `Christkindlesmarkt.alle` so, dass diese *alle denkbaren* Einkaufslisten zusammenstellt, die Sie mit Ihrem `geld` (in Cent wie alle Preise) aus dem verfügbaren `angebot` *gerade noch* kaufen können. Dabei ist es entscheidend, dass Sie so viel von Ihrem Geld wie möglich ausgeben, d.h. jede Einkaufsliste könnte nicht mal mehr mit der billigsten `Ware` ergänzt werden, ohne dabei „ins Minus“ zu rutschen – z.B.:

```
======================================================================
   11x Gluehwein (0.2l)                          4,50 :   49,50
                                                        ----------
                                          TOTAL (EUR) :   49,50
                                           REST (EUR) :    0,50
======================================================================
                              ... oder ...
======================================================================
    2x Drei im Weggla (200g)                     3,50 :    7,00
    3x Zwetschgenmaennle (0.1l)                  6,20 :   18,60
    2x Elisen-Lebkuchen (250g)                   6,95 :   13,90
    1x Gluehwein (0.2l)                          4,50 :    4,50
    1x Nostalgische Schneekugel                  5,99 :    5,99
                                                        ----------
                                          TOTAL (EUR) :   49,99
                                           REST (EUR) :    0,01
======================================================================
```

WICHTIG: Außer *String* und *IllegalArgumentException* dürfen Sie ***keine*** Klassen oder Methoden aus der Java-API verwenden! Prüfen Sie stets *alle* Parameter auf Plausibilität und werfen Sie ggf. eine *IllegalArgumentException* – aber nur dann, wenn sie *auch wirklich* ungültig sind! Auf dem Markt werden auch mal kostenlose Geschenke verteilt – davon dürfen Sie *höchstens eines* von jeder Art mitnehmen! Dass Sie schon pleite zum Markt gehen oder der Markt geschlossen hat und daher das Angebot leer ist, sollte hingegen nicht ungewöhnlich sein.

Gruppenaufgabe 8.3: Catch Up? Throw Up? — 10 GP

Aufgabenstellung und Abgabe als E-Test in StudOn.

Gruppenaufgabe 8.4: Mach das Licht aus! — 17 GP

In dieser Aufgabe betrachten wir eine Abwandlung des klassischen Spiels *Lights Out*: Zum einen ist das Spielfeld beliebig rechteckig (`cols` × `rows` statt genau 5×5) und zum anderen gibt es „blinde“ Felder, die sich nicht bedienen lassen und auch niemals leuchten (`mask`).

a) Bevor wir damit loslegen, erstellen Sie zunächst eine Klasse `BitOps` mit den folgenden vier Klassenmethoden. Alle diese Methoden müssen die gleiche Parameterliste haben, nämlich `(long bitSet, int bitIndex)`. Sie führen bestimmte Bitoperationen auf das Bit an der Position `bitIndex` in der Zahl `bitSet` aus (wie üblich hat das niederwertigste Bit stets die Position 0) und geben das Ergebnis entsprechend zurück:

- `long set`: setzt das Bit an der Position `bitIndex` in der Zahl `bitSet` auf 1.
- `boolean isSet`: gibt genau dann `true` zurück, wenn das entsprechende Bit 1 ist.
- `long clear`: setzt das Bit an der Position `bitIndex` in der Zahl `bitSet` auf 0.
- `long flip`: schaltet das entsprechende Bit um („kippt“ das Bit: $0 \leftrightarrows 1$).

b) Erstellen Sie nun eine Klasse `LightsOut`. Ihr Konstruktor bekommt die Anzahl der Spalten (Breite `cols`) und die Anzahl der Zeilen (Höhe `rows`) des Spielfelds sowie zwei `long`s `state` und `mask`, deren Bits das Spielfeld zeilenweise abdecken, als „würde man ein Buch lesen“ – das niederwertigste Bit steht dabei für die linke/obere Ecke des Spielfelds. In `state` wird verwaltet, ob ein Teilfeld leuchtet (Bit an der entsprechenden Position in `state` ist 1) oder nicht. Mittels `mask` werden einzelne Teilfelder „deaktiviert“ (Bit an der entsprechenden Position in `mask` ist 1).
Ihr Konstruktor muss zusätzlich den übergebenen Zustand `state` so bereinigen, dass *keine* Teilfelder mehr „leuchten“, die durch `mask` zu blinden Teilfeldern geworden sind oder die außerhalb des Spielfelds `cols` × `rows` liegen.

c) Ergänzen Sie eine Methode `long getState()`, die den aktuellen Zustand des Spiels (*nach* der Bereinigung durch den Konstruktor) zurückgibt.

d) Implementieren Sie eine Methode `toggle(col, row)` die einem Spielzug in *„Lights Out“* entspricht. Ist das Teilfeld an Position `(col, row)` durch `mask` deaktiviert, passiert *nichts*. Andernfalls werden das Teilfeld selbst sowie die vier Nachbarn oberhalb, unterhalb, links und rechts von `(col, row)` umgeschaltet ($an \leftrightarrows aus$) – aber natürlich nur, wenn sie jeweils nicht durch `mask` deaktiviert sind. Behandeln Sie ungültige Parameter wie beim Konstruktor.

e) Ergänzen Sie nun eine Methode `ZahlenFolgenMerker solve()`, die beim aktuellen Zustand beginnend die **kürzeste Lösung** für das gegenwärtige Rätsel sucht. Der zurückgegebene `ZahlenFolgenMerker` enthält dabei die Positionen der Teilfelder in der Reihenfolge, in der ein menschlicher Spieler sie `toggle`n müsste. Falls es mehrere kürzeste Lösungen gibt, dann genügt eine beliebige davon. Gibt es keine Lösung, dann soll `solve` `null` zurückgeben. Ist das Spiel bereits gelöst, dann ist ein leerer `ZahlenFolgenMerker` zurückzugeben.
Wichtig: Der Zustand (`state`) des aktuellen Spiels darf hierbei **nicht verändert** werden!

Hinweis: Klassisches Backtracking (wie in der Vorlesung) durchsucht den Lösungsraum i.d.R. zuerst jeweils bis zur maximalen „Tiefe“. Um damit aber die *kürzeste* Lösung zu finden, müsste immer der komplette (meist *immense*!) Suchraum abgetastet werden. Hier bietet es sich daher an, die Suchtiefe schrittweise immer weiter zu vergrößern (natürlich ohne die bisherigen Schaltfolgen immer wieder neu zu bestimmen), bis erstmals eine Lösung gefunden wurde. Gehen Sie sparsam mit Rechenzeit und Speicher (insbesondere auf dem Programmstapel) um! Beachten Sie dazu unbedingt auch die öffentlichen Testfälle und deren Laufzeitgrenzen! Besuchen Sie ggf. rechtzeitig mind. eine Tafelübung!

WICHTIG: Prüfen Sie alle Parameter *öffentlicher* Methoden *stets* auf Plausibilität und behandeln Sie fehlerhafte Eingaben, indem Sie eine `IllegalArgumentException` werfen.
ACHTUNG: Es sind **sämtliche** Aufrufe in die Java-API **untersagt**, außer die vorgeschriebene Verwendung von `IllegalArgumentException` und `String` (falls Sie eine Nachricht mit der Ausnahme mitgeben möchten)! Testen Sie Ihre Lösung gründlich und spielen Sie es mit der bereitgestellten GUI. *Hinweis:* Nicht jeder automatisch generierte Startzustand hat eine Lösung!

33 EP + 27 GP = 60 Punkte