テスト
(授業の試験じゃないよ)

2011年6月17日

海谷 治彦

目次

- まえふり
- ホワイトボックステスト
- ブラックボックステスト
  - テストケース作成

- Eclipse上でのテストツール JUnit
  - HPの方を参照

テストとは何か？

- 「本来どうあるべきか」(正解)と「実際はどうなのか」を比較すること.
  - コレは入試や期末試験も同じ.

- ソフトウェアの場合
  - 本来どうあるべきか？
    - 「仕様書」で規定されている(はず).
  - 実際，どうなのか？
    - コードを追ってみる.
    - 動かしてみる.

何故テストしないといけないか？

- あるべき動作と実際の動作が一致してることを確認し，ソフトウェアが正しく作られていることを確認しないといけないから.
- 一致しない場合，その原因を探さないといけないから.
- バグ
  - 上記の差異が発生した原因.
  - 仕様に反した振る舞いを引き起こすコード.
- あるべき動作や結果とは？
  - 仕様(書)に指定してあるソフトウェアの期待される振る舞いや結果，例えば，
    - ある入力に対して，ある出力がでる機能を持つ.
    - その機能をある効率で行える.
    - その機能は使いやすい・・・等

実例

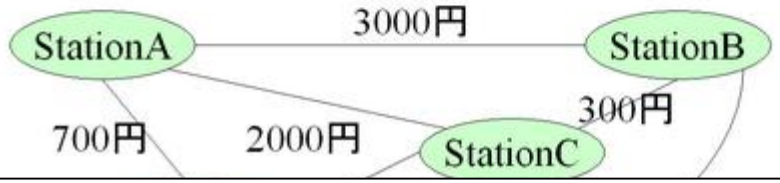
2011 ソフトウェア工学 演習1

問題

オイスターカード(London)やSuica(関東)等の，先払い式の電子切符の役割を模倣する
る．

- 入金することで先払い運賃をカードに追加する，所謂，Top-upする．
- 乗車する駅を記録する．
- 下車する駅において料金を支払う．支払えない場合，改札を出れない・・・

駅もクラスとして実現し，本演習では以下のStationA, B, C, Dの4つの駅が，以下のよ
由して目的駅に到着したとしても，運賃は途中駅を経由せず移動した場合の金額(最



「追加」してない
じゃん！(バグ)

```
private int amount;
public void topUp(int a){
  amount = a;
}
```

エラー，欠陥，障害

- エラー error
  - ソフトウェア開発中の人間の誤った判断.
  - 例: プログラムミス，仕様の理解誤り.
- 欠陥 fault (もしくは defect)
  - ソフトウェアが意図した振る舞いをしない原因となったソフトウェアの特性(コードの箇所).
  - 所謂，バグと考えてよい.
- 障害 failure
  - ソフトウェアが実行中に意図した振る舞いから逸脱したこと.

バグ取りのためのテストでは無い

建前ですが・・・

- テストはあるべき動作と実際の動作が一致することの確認を通して，ソフトウェアが正しく作成されていることを確認すること.

現実的には，

- 一致しないことを見つけて，その原因(バグ)を探すためにテストが行われることが多い.

テストで最も必要な能力

- 日本語(や英語)の読解力.
  - 仕様書は通常，日本語や英語等の自然言語で書かれています.

- ソフトウェア技術者は数学よりも国語(英語)のほうが重要です.
  - 言葉の意味わかってますか？
  - 意味わかる言葉を書けますか？

  多分，どの技術分野でもそうだと思う.

何をテストするか？テストレベル

- 受け入れテスト
  - ソフトウェアはどこかの誰か(ユーザー)に使われることを想定しているので，そのユーザーに使ってもらい，あるべき動作をしているか確認.
- システムテスト
  - 基本，受け入れテストと同じだが，開発者側が模擬的な環境で行う.
- 統合テスト
  - 単体(後述)を組み合わせてテスト. 例えばmainから始まるプログラムを全統合して.exe等を作り動作をテストする.
- 単体テスト
  - 個々の単体部品を部品毎にテストする.
  - 単体部品: Cなら関数. Java, C#ならクラス.

テストケース

- テストを行うための基本単位
- 最低，以下から構成される.
  - 入力
  - 期待される出力結果
- その他，以下の情報があると良い.
  - 実行順序
    - 対話的な処理の場合，順序によって結果が変わるため.
  - 実行環境
    - データベースやファイル等，定常的なデータ，OSや実行環境(Javaの場合，VM: virtual machine)も結果に影響を与えるため.
    - ソフトウェアの場合，天気や放射線量は関係無い・・・はず.

例: 本授業の演習1

- 統合テストのためのテストケースといえる.

- 入力:
  ハードコーディングされている.
- 期待される出力:
  コメント文に書いてある.
- 実行順序
  ハードコーディングされている.

- mainメソッドにおいて，以下のテストプログラムが動作するようにクラスを設計せよ.

```
Card card=new Card();

Station a=new Station("StationA");
Station b=new Station("StationB");
Station c=new Station("StationC");
Station d=new Station("StationD");
/*
 駅間の運賃情報を持たせるような処理を自分でこの辺に書いてください.
 */

   card.topUp(3000); // 3000円 topUp

   card.getOn(a); // StationAで乗車
   if(card.getOff(c)) // StationCで下車 2000円かかる
     System.out.println(card.getBalance()); // 1000円のはず
   else
     System.out.println("shortage"); // まぁ，残高不足の場合もあるよね

   card.getOn(c); // StationCで乗車
   if(card.getOff(d)) // StationDで下車 500円かかる
     System.out.println(card.getBalance()); // 500円のはず
   else
     System.out.println("shortage"); // まぁ，残高不足の場合もあるよね

   card.getOn(d); // StationDで乗車
   if(card.getOff(a)) // StationAで下車 700円かかる
     System.out.println(card.getBalance());
   else
     System.out.println("shortage"); // 500円しかもってなかったので下車できん
```

テストの典型的な分類

- ホワイトボックステスト
  - プログラムの流れを直接追跡することで，期待する結果をもたらす命令列か否かをチェックする.
  - 手間がかかるので全部やるのは一般に無理.

- ブラックボックステスト
  - 実行結果が期待する結果(仕様書)と一致するか否かをチェック.

ホワイトボックステスト

二つの代表手法

- 制御フローテスト
  - if, case, whileなど，分岐が起きる部分に注目し，可能な限り全ての実行順序インスタンス(フロー)を見直すテスト.
  - ありうるフローをどれだけテストできるかが重要
    カバレッジ
  - ループがあった時点で，全てやるのはほとんど無理となる.
- データフローテスト
  - 変数を定義(宣言では無く初期化の代入等)してから，利用しているかのチェック.
  - null pointer exception (ぬるぽ)を探す常套手段.

制御フロー

- 基本，if, case, while, for 等でプログラムの進行が変わる部分に従い，プログラム命令の流れを列挙すること.
- 左例の場合，
  - 文1, 文3, 文4
  - 文2, 文3, 文4
  - 文1, 文3, 文5
  - 文2, 文3, 文5
  - 文1, 文3
  - 文2, 文3

  が考えられるフロー群.

```
if(条件1){
   文1;
}else{
   文2;
}
文3;
switch(式2){
case 値3: 文4; break;
case 値4: 文5; break;
}
```

フローを網羅してどうする？

- それぞれのフローについて，
  - そもそも，そんなフローがあっていいのかをチェック．要人間の判断．
    - 参照にnull代入した後に，その参照アクセスしちゃぁダメでしょ．とか．
  - その順序で(命令)文を実行すると，期待する結果が得られるかをチェック．
    - 基本，人間がコンピュータのかわりの計算手順を確認するので，人手が必要．
- 安心感
  - フローの網羅性が高く，それぞれテストされていれば，実際の実行においてマズいこと(障害)が起きないことを保証できる．

フローの網羅性のレベル

- ステートメント・カバレッジ (statement coverage)
  - 全命令文(statement)は最低一回実行している網羅性のレベル.
  - しょぼいが最低ラインとされる.
- ブランチ・カバレッジ (branch coverage)
  - ifやcase等の条件分岐において，全分岐の組み合わせを網羅する.
  - 結構イイカンジだが，条件自体の複雑さを考慮していないので，ちょっくら危険.
  - 単純なif文がN段続くとフローの数は 2^N となる．組み合わせ的に爆発している.
- さらに詳細な網羅法もある.

100%カバレッジの例

```
if(条件1){
   文1;
}else{
   文2;
}
文3;
switch(式2){
case 値3: 文4; break;
case 値4: 文5; break;
}
```

ステートメント・カバレッジ
文1, 文3, 文4
文2, 文3, 文5

ブランチ・カバレッジ
文1, 文3, 文4
文2, 文3, 文4
文1, 文3, 文5
文2, 文3, 文5
文1, 文3
文2, 文3

データフローテスト

- 変数が定義(代入)されてから利用されているかを制御の流れから追うテスト.
- 値がnullの参照に対してメソッドを送ってしまうようなバグを探すのに使われる.
- Javaの場合，基本変数，参照ともに宣言時点での値が決まっているので，ある意味危ない.
- Cの場合，宣言時点では値は不定なので，データフローテストをちゃんとやらないと，かなり危ない.
- 変数のスコープの問題もからんでくるが，その辺はコンパイラやEclipseでは自動でチェックしてくれる.
  - そんな変数の宣言は無いぞ･･･とか.

変数の3つの状態

- d: 定義.
  - 代入文や引数等で値が設定されている状態.
  - 例 var = ほげほげ.
- u: 使用.
  - 代入文の右辺の式や条件式，メソッドの引数等で使われている.
  - 例 ほげほげ = var + ...
- k: 消滅.
  - ブロックやメソッドから抜けて変数自体がなくなっている.
  - インスタンスが消滅すれば属性も消滅している.

マズい遷移

- k-u
  - 無い変数を使う. null pointer exception が典型.

以下は少しマズい

- d-k
  - 値を設定したのに使ってない. 何故？
- d-d
  - 値代入後に, また代入. 途中値は使わないの？

例

```
SomeRef sr;
if(条件1){
  sr=new SomeRef(); // 文1
}
文3;
switch(sr.getValue()){
case 値3: 文4; break;
case 値4: 文5; break;
}
```

文1は条件1によって実行されない場合がある.
よって, srはswitch時点でnullの可能性がある.
しかし, switchの判定でムゲにsrのメソッドを呼んでいる.
ヌルポの危険あり.

ブラックボックステスト

ブラックボックステストの概要

- 名前の通りテスト対象(クラスやプログラム全体)を黒箱(中身が見えない)ものとして入力を入れて，出てきた結果を吟味する，すなわち，
- 実行結果が期待する結果(仕様書)と一致するか否かをチェック.
- テスト対象は一般に多数(無限？)のバリエーションの入力を受け付けるので，テストケースの網羅性が重要となる.

網羅性のための技法

- 同値クラステスト
- 境界値テスト
- ディシジョンテーブルテスト

同値クラステスト

- 仕様書から見て，同じ扱いをされる入力群から1個程度，テストをする.
- この同じ扱いをされる入力群を同値クラスと呼ぶ.
- 加えて，プログラムが仕様書から読み取れる同値クラスを素直に反映して実現されているという前提が必要.
  - ヒネたプログラマのコードだとマズい・・・

同値クラスの典型例

- 例えば，税率の計算は収入額によって多分異なる.

(以下は日本の実情とは当然異なります)

1. ～300万円/年 x%
2. ～800万円/年 y%
3. ～1000万円/年 z%
4. それ以上，a%

- 上記の4つの場合分けを同値クラスとして，例えば，テストデータとして，

100万円，500万円，900万円，2000万円

くらいを用意する.

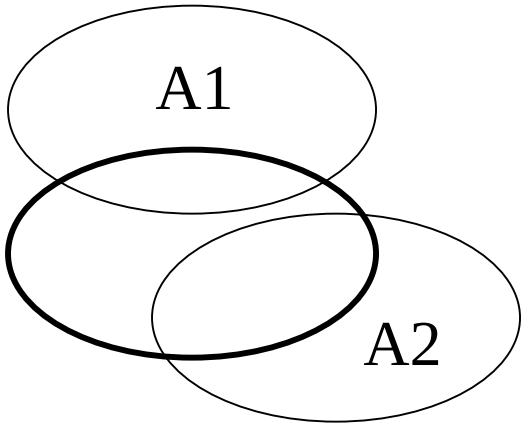
無効な値も同値クラスの一つとすべきか？

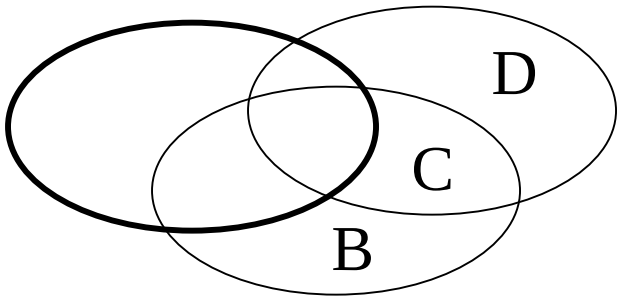
- 前述の例の場合，「－200万円/年」の収入に対する税率の計算とか．
- 学術的な見解からは，無効値を同値クラスとすべきか否かは，ソフトウェアの設計ポリシーによる．
  - あるメソッドや関数，プログラムが，どのような値を受け付けるかも仕様で規定されるべきなので，「想定外」の値での振舞いをプログラムは保証する必要は無い．
- 現実的な対応としては，無効値も同値クラスとしたほうが良い．
  - 想定外と仕様書に書いても，それを実施するヤツは必ず居るため．人は約束を守れない．

演習2での同値クラス

ただしお勧め機能のみ

- 顧客が1名しか居ない場合
  - お勧めは存在しないはず.
- 顧客が2名以上の場合
  - 購入商品の重複が無い場合,
  - 二名に有る場合,
  - 三名に有る場合, Cを二重カウントしない.





集合として考えれば,
全て同値クラスだけど.
∪SetOfC – 自身 だから.

境界値テスト

- 異なる同値クラス間の境界の値がアブないという経験に基づくテスト.
- 前述の税率問題
  - ～300万円/年 x%
  - ～800万円/年 y%
  - ～1000万円/年 z%
  - それ以上, a%

  の場合,
- 以下が境界値

  -1, 0, 1, 299, 300, 301, 799, 800, 801, 999, 1000, 1001
- 無効値も同値クラスの一種としている.

ディシジョンテーブルテスト

- Decision Table
- 複数の条件(入力の種類)に従って，アクション(結果)が決まるようなソフトウェアの振る舞いを表としてまとめた仕様書.
- そのままテストケースとして利用できるので便利.
- 同値クラスをN次元に拡張したと考えて良いでしょう.

ディシジョンテーブルの標準様式

ルール		1	2	…	p
条件	1				
	2				
	…				
	m				
アクション	1				
	2				
	…				
	n				

例

10個の同値クラスがあるので，10個のテストケースをやれば良い．

ルール		1	2	3	4	5	6	7	8	9	10
条件	収入	～0		～300		～800		～1000		それ以上	
	家族	有	無	有	無	有	無	有	無	有	無
アクション	税率(%)	0	0	3	5	10	15	20	25	30	35

※ 日本の実情とは異なります，多分．

演習1 topUpについて

ルール		1	2	3	4
条件	残金	無 0		有 N	
	追加金	正 M	負	正 M	負
アクション	結果	M	0	N+M	N

前述のバグは問題中のテストケースがルール1のものしか試していないことに起因する.
せめて，ルール3もやってくれないと・・・

それ以前に日本語読んで理解しないと・・・

再録

実例

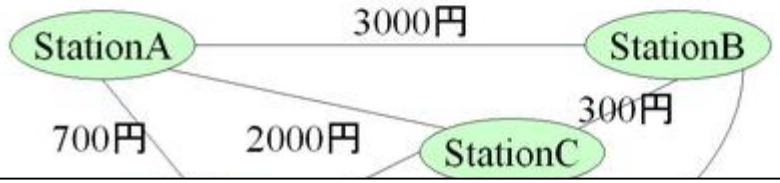
2011 ソフトウェア工学 演習1

問題

オイスターカード(London)やSuica(関東)等の，先払い式の電子切符の役割を模倣する
る．

- 入金することで先払い運賃をカードに追加する，所謂，Top-upする．
- 乗車する駅を記録する．
- 下車する駅において料金を支払う．支払えない場合，改札を出れない・・・

駅もクラスとして実現し，本演習では以下のStationA, B, C, Dの4つの駅が，以下のよ
由して目的駅に到着したとしても，運賃は途中駅を経由せず移動した場合の金額(最



「追加」してない
じゃん！(バグ)

```
private int amount;
public void topUp(int a){
  amount = a;
}
```

おわりに

- 産業界(プロ)のソフトウェア開発ではテストの関心は異常に高いです.
- ソフトウェア工学 = テスト

  といってもいいくらい.
- 製品の品質をテストで担保している実情を考えれば，当然のことと思います.

以上