データ構造とプログラミング：テスト

情報工学科 山本哲男

ArrayListのaddメソッド

```
public void add(int p, E x) {
    if (p < 0 || p > size)
        return;
    if (size == elements.length)
        if (!growList())
            return;
    for (int i = size; i > p; i--) {
        elements[i] = elements[i - 1];
    }
    elements[p] = x;
    size++;
    return;
}
```


ArrayListのaddメソッド（つづき）

```
public void add(E x) {
    return add(size, x);
}
```

再利用できるメソッドはする！

ArrayListのgrowListメソッド

```
private boolean growList() {
  try {
    // 倍にするかどうかは要検討事項
    int newsize = elements.length * 2;
    Object newelements[] = new Object[newsize];
    for (int i = 0; i < elements.length; i++) {
      newelements[i] = elements[i];
    }
    elements = newelements;
    return true;
  } catch (OutOfMemoryError e) {
    return false;
  }
}
```

メモリエラーを補足するかどうかは難しいところ

ArrayListのremoveメソッド

```
public boolean remove(E p) {
    if (p < 0 || p >= size)
        return;
    E old = elements[i];
    for (int i = p; i < size; i++) {
        elements[i] = elements[i + 1];
    }
    size--;
    // GCを促すために
    elements[size] = null;
    return old;
}
```


LinkedListのaddメソッド

- 基本操作は以下のコード

挿入したい場所の直前のノードを取得

```
Node<E> prevnode = getPrevNode(p);
Node<E> newnode = new Node<E>(x);
newnode.setNext(prevnode.getNext());
prevnode.setNext(newnode);
```

nextの付け替えの順番を間違えないように

単体テスト

モジュール内部に存在するエラーを検出

- ブラックボックステスト(block-box test)
  - テストデータを与えて，実行結果を観察することでエラーを検出
  - プログラムの外部仕様(機能)に着目
  - プログラムの詳細(内部構造や内部論理)を無視

  同値分割法，限界値分析法，原因結果グラフ法
- ホワイトボックステスト(white-box test)
  - テストデータを与えて，実行のようすを追跡することでエラーを検出
  - プログラムの内部仕様(構造や論理)に着目
  - 制御の流れに基づくテスト網羅

  テスト網羅技法
- コードレビュー(code review)
  - コードウォークスルー(walk-through)： 非形式的，正当性に関するコメント
  - インスペクション(inspection)： 形式的，リストとコードとの照合

同値分割法

- プログラムの入力領域を同値クラスに分類することでテストケースを作成

(1) 同値クラスの識別

機能仕様の入力条件を満足する範囲(有効同値クラス)と満足しないクラス(無効同値クラス)に分割

(2) クラスに基づくテストケースの作成

(1) 同値クラスを検査するテストケースを作成

e.g., amku5ge

(2) 1つの無効同値クラスと残りの同値クラスを検査するテストケースを作成

e.g., xy9, jdsi5enjcd, abcdef, 123456

例)

入力条件	有効同値クラス	無効同値クラス
文字数	4 ~ 8	3以下，9以上
文字の種類	英字と数字の組合せ	英字のみ，数字のみ

限界値分析法

- 入出力条件の境界値を詳しくテストするテストケースを作成
  - (1) 入出力条件の識別
    機能仕様の入出力条件に着目し、境界を判別する
  - (2) 境界に基づくテストケースの作成
    以下の例の場合
    0,1,2,63,64,65

例)

条件	1～64の数字
境界	1と64

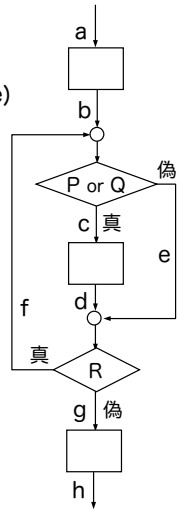
テスト網羅技法(1)

命令網羅，節点網羅(statement coverage, C0 coverage)
- ✓ プログラム中のすべての文を1回以上実行
  例) P or Qが真，Rが偽 (パス: abcdgh)

網羅(coverage) = 実行した文 / 全文

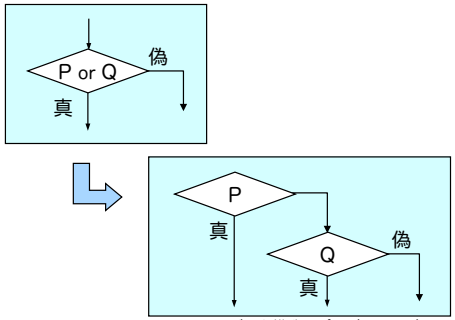
枝網羅，分岐網羅(edge coverage, C1 coverage)
- ✓ プログラム中のすべての枝を1回以上実行
  例) P or Qが真，Rが真(パス: abcdf)
  P or Qが真，Rが偽(パス: abcdgh)
  P or Qが偽，Rが真(パス: abef)
  P or Qが偽，Rが偽(パス: abegh)

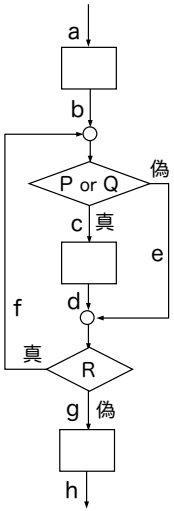
網羅(coverage) = 通過した枝 / 全枝



テスト網羅技法(2)

条件網羅(condition coverage)
- ✓ プログラム中のすべての判定条件を1回以上実行
  e.g., PとQを区別
  Pが真，Qが偽 or 偽
  Pが偽，Qが真 or 偽

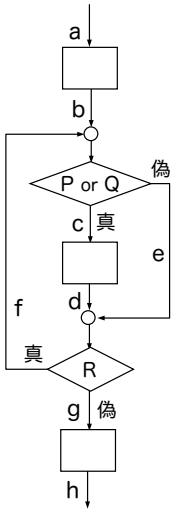




テスト網羅技法(3)

パス網羅(path coverage)
- ✓ 判定条件間の依存性(条件の組合せ)を考慮
- ✓ プログラム中のすべてのパスを1回以上実行
  e.g., abcdgh + abcdfcdgh + abegh
  + abefegh + abcdfegh + …

網羅(coverage) = 実行したパス / 全パス



Javaにおける単体テスト

- 各クラスの各メソッドについてテスト
- テストするクラスに対応するテストクラスを作成
  - 例）LinkedList -> LinkedListTest クラス
- 各テストケースのコードをテストクラスのメソッドとして記述

クラス名やメソッド名の先頭や最後に
test(Test)という名前を付けることが多い

テストクラス

```
public class TestLinkedList {
        private LinkedList<String> list;

        public static void main(String[] args) {
                (new TestLinkedList()).start();
        }

        /**
         * 実際にテストする項目を列挙
         */
        private void start() {
                testAdd1();
                testAdd2();
                testAdd3();
                testAdd4();
                testAddP1();
                testAddP2();
                testAddP3();
                   ...
        }
```



```
        /**
         * 各テストに共通の初期化
         */
        private void setUp() {
                list = new LinkedList<String>();
        }

        /**
         * add(int x)のテスト
         * 空から一回だけ
         */
        public void testAdd1() {
                setUp();
                list.add("a");
                String methodName =
                     new Throwable().getStackTrace()[0].getMethodName();
                if (list.get(0).equals("1") && list.size() == 1)
                        System.out.println(methodName + " OK");
                else
                        System.out.println(methodName + " NG");
        }
```

addメソッド呼び出し

呼び出し後の状態をチェック

Listのテストケース

- クラスが状態を持つので複雑
- リストの場合だと，最低限
  - 先頭
  - 最後尾
  - 真ん中

でのチェックは必要

- 各状態を準備し，各メソッドのテストをする
- あるメソッドのテストをする場合，他のメソッドは動作することを仮定

シナリオベースのテストケース

- 各メソッドの仕様をテストするのではなく，一連のシナリオが正しく動作するかテスト
  - 空のリストに要素を追加
  - リストにある程度要素があり，削除
  - …
- 各シナリオにパラメータが必要になる
  - ある程度の数
  - 追加，削除する位置
  - …
- 汎用的なテストメソッドを作成し，パラメータを変えて呼び出してテスト

単体テストツール

- テストクラスは定型な処理を大量に記述
  - 同じような事を全部記述する行為が無駄
- テスティングフレームワークの活用
  - テストケースの内容等を記述するだけでテストが実行可能
  - JUnit
    - Java向け単体テストフレームワーク

JUnitを用いたテストクラス

- テストクラス名はTestで終わる
- junit.framework.TestCaseを継承
- 各テストケースの
  - 初期化メソッド名はvoid setUp(void)
  - 後始末メソッド名はvoid tearDown(void)
- テストケースのメソッド名はtestで始める

※バージョン4以降では異なる記述方法も存在


```
package ms.gundam.dsap.exercise02;
import static org.junit.Assert.*;
import junit.framework.TestCase;

import org.junit.Test;

public class LinkedListTest extends TestCase {
        LinkedList<String> list;

        /**
         * 各テストに共通の初期化
         */
        public void setUp() {
                list = new LinkedList<String>();
        }

        /**
         * add(int x)のテスト
         * 空から一回だけ
         */
        public void testAdd1() {
                list.add("a");
                assertEquals("a", list.get(0));
                assertEquals(1, list.size());
        }
}
```

mainメソッドは必要ない

テストケースを列挙する必要がない

setUp()を明示的に呼ぶ必要はない

メッセージ表示等を自分で記述する必要はない assert???? というメソッドを利用することで自動的にチェック

assert系メソッド

- assert??? メソッドを呼び出し
  - 一つ目に引数に期待する値
  - 二つ目の引数に実際の値

を記述し呼び出す

- assertEquals が基本

assert系メソッド一覧

メソッド	説明
assertArrayEquals(arrays expected, arrays actual)	配列同士を比較，同じ場合は成功
assertEquals(Object expected, Object actual)	オブジェクト同士を比較，同じ場合は成功
assertFalse(boolean condition)	条件がfalseである事を検証，falseの場合は成功
assertNotNull(Object obj)	オブジェクトがnullで無いことを検証，nullでない場合は成功
assertNotSame(Object expected, Object actual)	参照先が異なるか比較，異なる場合は成功
assertNull(Object obj)	オブジェクトがnullである事を検証，nullの場合は成功
assertSame(Object expected, Object actual)	参照先が同じか比較，同じ場合は成功
assertTrue(boolean condition)	条件がtrueである事を検証，trueの場合は成功
fail()	常に失敗させる 『ここに来たらダメ(テスト失敗)』な箇所に配置

例外のテスト

```
public void testException() {
  try {
     // 例外を発生させるテストを実行
     ...
     fail("例外が発生するはずなのでここに到達しない");
  } catch(発生すべき例外 e) {
  }
}
```

failを記述しないと例外が発生しなかった場合にテストが成功する

例外発生時に追加でチェックしたい項目があれば記述（空でも可）

コンパイル・実行方法

- フレームワークなのでmainはJUnit側に
  - junit.textui.TestRunner にmainが存在
- JUnitのjarファイルがカレントディレクトリにありpackageがms.gundam.e04の場合は

```
% javac -cp junit-4.11.jar ms/gundam/e04/LinkedListTest.java
```

```
% java -cp junit-4.11.jar:. junit.textui.TestRunner ms.gundam.e04.LinkedListTest
```

JUnitの実行結果

テスト成功時

```
02.LinkedListTest
..
Time: 0.002

OK (2 tests)
```

テスト失敗時

失敗時はここを確認

```
Time: 0.001
There was 1 failure:
1) testAdd1(ms.gundam.e04.LinkedListTest)junit.framework.AssertionFailedError: expected:<0> but
was:<1>
        at ms.gundam.e04.LinkedListTest.testAdd1(LinkedListTest.java:25)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 0
```


オートボクシング

- プリミティブ型に対応するオブジェクト型がある場合，相互変換を自動的にする仕組み
  - javacが賢く変換してコンパイルしてくれる

```
Integer i = 2;

Integer intObj = new Integer(10);
int i2 = intObj;

List<Integer> intList = new LinkedList<Integer>();
intList.add(10);
```