Javaプログラム単体テスト自動実行ツール開発に向けた一考察

宮崎大学工学部情報システム工学科
松岡慎吾, 片山徹郎

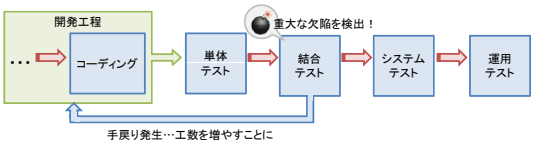
背景

- IT技術の進歩とともに情報システムの重要性が増大
  - システムの大規模化・複雑化・短納期化
  - 高い品質, 生産性の向上が求められる
- ソフトウェアテストの重要性
  - 開発に遅れが生じると, テスト工程が縮小される傾向がある
  - テストを効率的に行うことが, プログラムの信頼性, 生産性の向上につながる
  - 効率的なテストを行う為の一つの手段として, **テストの自動化**が挙げられる

→ テスト自動化を目的としたテスト支援ツールが様々提案されてきている

単体テストの重要性

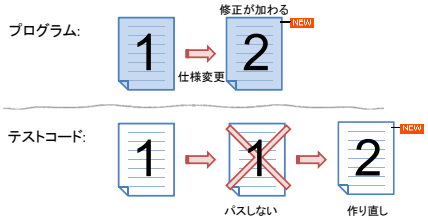
- 上流工程での品質確保として, 単体テストは非常に重要な役割を持つ
  - 後工程で見つかった欠陥ほど修正コストが増大する



- 単体テストは頻繁に繰り返すことが理想的
  - テスト工程の中でも自動化の優先度が特に高い

従来の単体テスト自動実行ツール –(1)

- テストコードはプログラムの変更に伴って常に修正しなければならない



- 結局手間がかかる → 期待ほどの効果が得られない

従来の単体テスト自動実行ツール –(2)

- テストコードをユーザが記述する必要性
  - ユーザはテストコードを記述する時間を費やす
  - ユーザがテストコードを記述できる能力も必要

⇩

テスト対象コードのみを与えることでテストを実施

⇩

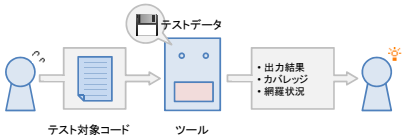
ユーザがテストコードを記述する必要性が無くなる

⇩

テスト実施が容易になり, 空いた工数で他のテストを実施

今回考察するツールの特徴

- テストデータを**コードから**自動的に生成する.
  - テスト対象コードをツールに入力するだけ
- 生成したテストデータを用いて, ステートメント**カバレッジ(C0)を計測**する.
  - カバレッジ100%達成で自動的にテストを終了する
- ステートメントの**網羅状況を表示**する.
  - ステートメントの実行状況をリアルタイムで観察できる



手順

① ツールにテスト対象コードを入力する

② テスト対象コードの解析，およびテストデータの生成

解析方法，テストデータの生成方法は7-10ページ

③ テスト用コードを作成

テスト用コードは，生成したテストデータを入力する為に必要となる．

④ テスト用コードにテストデータを入力し，テストを実施

テスト対象コードのカバレッジ取得の為，テスト用コードと確認しながら実行する

テスト対象コードの解析

- 指定のプログラムコードに静的解析を行う
- 具体的には，**正規表現**を用いて変数名，条件分岐文および条件式，クラス名等を検索または取得する

変数の型を取得　変数名を取得

```
class Test{
  public static void main(String args[]){
    int a;
    System.out.println("Please input number:");
    a=readNumber();
  if(a>0)System.out.println("Greater than 0.");
}
```

分岐条件を取得　標準入力部分を検索　分岐通過時の出力を取得

正規表現

- 汎用的なパターン記法を用いてテキストの記述や解析のできる表現法

メタ文字	説明	正規表現例	一致する文字列
¥¥.	「.」に一致。通常メタ文字として用いる文字を，意味を持たない通常の文字として表記する。	Hello¥¥.	Hello.
¥¥w	単語構成文字(小文字a~z, 大文字A~Z, 数字0~9)に一致。	Hell¥¥w	Hella, HellA, Hell0, …
.	任意の一文字を意味する。	Hell.	Hella, Hell!, Hell@, …
|	「または」の意味を持ち，隔てられた表現のうち，いずれかと一致。	grey|gray	grey, gray
( )	| の対象領域を制限するのに用いる。	gr(e|a)y	grey, gray
*	直前にある文字に対して，最低0回以上の繰り返しを要求。	Hello*	Hell, Hello, Helloo, …
+	直前にある文字に対して，最低1回以上の繰り返しを要求。	Hello+	Hello, Helloo, Hellooo, …

– 正規表現例(if文): 『.*if¥¥(¥¥w+(<|>|!=|<=|>=|==)¥¥w+¥¥).*』

テストデータの自動生成

- **境界値分析**の概念に基づいたテストデータ生成方法
  – コードから境界値を取得し，分岐命令の重複等があれば，前後の条件式に応じてテストデータを生成する

<例>
入力A(1~999までの整数)
入力B(1~999までの整数)
出力C=A×B

	有効同値クラス	無効同値クラス	境界値
入力A	1～999	0以下，1000以上	0, 1, 999, 1000
入力B	1～999	0以下，1000以上	0, 1, 999, 1000

→ テストデータ{0, 1, 999, 1000}

具体的なテストデータ生成の例

※ 変数a，変数bに標準入力によって値が代入される

例文	生成されるテストデータ	備考
if(a > 0 && a < 100){…}	a=0, 1, 99, 100	

if(a > 0){
if(b > 0){
…}}
	{a,b}={1, 1}, {1, 0}, {0, 1}, {0, 0}	分岐が重なる場合は，{正，正}，{正，誤}，{誤，正}，{誤，誤} のように，組み合わせて生成する．

switch(a){ case 1: case 3: case 5: …}	a={0, 1, 3, 5}	
for(i=0;i<a;i++){…}	a={0, 1, 32167}	特に条件がない場合は，**0, 1, 大数**をテストデータに設定する．

if(a >0){
if(b>0){
if(a>b){
…}}}
	{a,b}={1, 1}, {1, 0}, {0, 1}, {0, 0}, **{2, 1}, {1, 2}**	変数同士の比較を行う条件式では，**[有効値の最小値+1]**と，**[有効値の最小値]**を用いてテストデータを生成する．

switch(a+1){
case 0:
case 1:
…
case 9:
}
	a=-1, 0, 1, 2, 3, 4, 5, 6, 7, 8	条件式に数式が挿入されている場合は，逆算(例文の場合は，a=case-1)してテストデータを生成する．

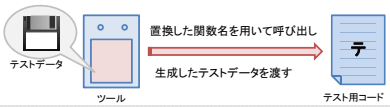
テスト用コードの作成

- 生成したテストデータを，テスト対象コードに入力することが目的
- テスト用コードは，テスト対象コードに対してパターンマッチを行い，以下の書き換えを施す

書き換え前	書き換え後	説明
class [クラス名]{	class MySample{ int [変数名];	ツール内で呼び出すMySampleというクラス名に置換．

public static void main	public MySample(int[] x){
	this.[変数名]=x[0];
	}
	public void MyMain
		ツール内で呼び出すMyMainという関数名に置換．また，main文直前に，テストデータを受け取るコンストラクタを挿入．

int [変数名];	/* int [変数名] */	コンストラクタでグローバル変数として宣言する．
[変数名] = [標準入力命令];	/* [変数名] = [標準入力命令]; */	テストデータを代入する為不要．



テスト用コード: 例

- テスト対象コード

```
class Test{
  public static void main(String args[]){
    int a;
    System.out.println("Please input number:");
    a=readNumber();
  if(a>0)System.out.println("Greater than 0.");
  }
  …
```

整数の入力を行う関数

- テスト用コード

```
class MySample{                                   (1)
  int a;                                          (2)
  public MySample(int[] x){                       (3)
    this.a=x[0];                                  (4)
  }
  public void MyMain{                             (5)
    /* int a; */                                  (6)
    System.out.println("Please input number:");
    /* a=readNumber(); */                         (7)
  if(a>0)System.out.println("Greater than 0.");
  }
}
…
```

(1) クラス名のTestをMySampleに変更.

(2) main関数内で削除した変数aを, グローバル変数として再宣言.

(3) コンストラクタMySample: ツールから呼び出される.

(4) グローバル変数で宣言されている為, 変数宣言をコメントアウト.

(5) ツールから受け取ったテストデータをグローバル変数aに代入.

(6) main関数をMyMainに変更.

(7) テストデータ代入の為,標準入力を求める命令文をコメントアウト.

ステートメントの網羅状況を表示

- プログラムの実行状況に伴い, ステートメントの網羅状況をリアルタイムに表示
  - 網羅状況はステートメントに色を付けることで表示する
  - カバレッジ100%達成で自動的に停止する
- プログラムの流れを追いながら, 実行されないステートメントを観察することができる

```
if(a>0){
  System.out.println("Greater than 0.");
  if(a==0){
    System.out.println("Just 0.");
  }
}
if(a<0){
  System.out.println("Less than 0.");
}
```

条件を満たさない為, 実行されない
→ ステートメントカバレッジ 66%

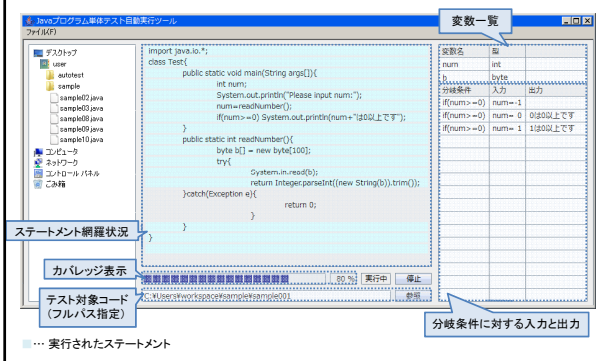
カバレッジ取得において

- テストデータ代入の為にテスト用コードを作成したが, テスト対象コードとはステートメントが異なる.
- テスト対象コードのカバレッジを取得するため, テスト用コードとテスト対象コード間で置換した命令を常時確認しながらテストを行う.
  - class MySample{とclass Test{
  - 新たに宣言するグローバル変数と削除するローカル変数
  - テストデータ挿入命令と標準入力命令
  - public static void main(String args[]){とpublic void MyMain{

```
class Test{
  public static void main(String args[]){
    int a;
    System.out.println("Please input number:");
    a=readNumber();
  if(a>0)System.out.println("Greater than 0.");
}
```

```
class MySample{
  int a;
  public MySample(int[] x){
    this.a=x[0];
  }
  public void MyMain{
    /* int a; */
    System.out.println("Please input number:");
    /* a=readNumber(); */
  if(a>0)System.out.println("Greater than 0.");
  }
}
```


ツールの外観(仮)



… 実行されたステートメント

適用例 - テストデータの生成

```
class Test{
  public static void main(String args[]){
    int year;
    System.out.println("Please input year:");
    year=readNumber();
    if(year<0) year=-year;
    switch(year%12){
      case 0:  System.out.println(year+"は申年です");break;
      case 1:  System.out.println(year+"は酉年です");break;
      case 2:  System.out.println(year+"は戌年です");break;
      case 3:  System.out.println(year+"は亥年です");break;
      case 4:  System.out.println(year+"は子年です");break;
      case 5:  System.out.println(year+"は丑年です");break;
      case 6:  System.out.println(year+"は寅年です");break;
      case 7:  System.out.println(year+"は卯年です");break;
      case 8:  System.out.println(year+"は辰年です");break;
      case 9:  System.out.println(year+"は巳年です");break;
      case 10:  System.out.println(year+"は午年です");break;
      case 11:  System.out.println(year+"は未年です");break;
    }
  }
  public static int readNumber(){
    byte b[] = new byte[100];
    System.in.read(b);
    return Integer.parseInt((new String(b)).trim());
  }
}
```

- 西暦を入力し, 干支を確かめるプログラム
  - 変数の数は1つ
  - 分岐はあるものの, 重複はない

無効同値クラスの **−1** と,有効同値クラスの **0** を生成　—(1)

条件式(year%12)を逆算すると,
テストデータ=12−case
よって, **1,2,3,4,5,6,7,8,9,10,11,12**
が生成される　—(2)

- テストコードに渡す最終的なテストデータは, (1), (2) を合わせた,
  −1,0,1,2,3,4,5,6,7,8,9,10,11,12
  となる

適用例 - テスト用コードの作成

```
class MySample{
  int year;
  public MySample(int[] x){
    this.year=x[0];
  }
  public void MyMain{
    /* int year; */
    System.out.println("Please input year:");
    /* year=readNumber(); */
  if(year<0) year=-year;
    switch(year%12){
      case 0:  System.out.println(year+"は申年です");break;
      case 1:  System.out.println(year+"は酉年です");break;
      case 2:  System.out.println(year+"は戌年です");break;
      case 3:  System.out.println(year+"は亥年です");break;
      case 4:  System.out.println(year+"は子年です");break;
      case 5:  System.out.println(year+"は丑年です");break;
      case 6:  System.out.println(year+"は寅年です");break;
      case 7:  System.out.println(year+"は卯年です");break;
      case 8:  System.out.println(year+"は辰年です");break;
      case 9:  System.out.println(year+"は巳年です");break;
      case 10:  System.out.println(year+"は午年です");break;
      case 11:  System.out.println(year+"は未年です");break;
    }
  }
  public static int readNumber(){
    byte b[] = new byte[100];
    System.in.read(b);
    return Integer.parseInt((new String(b)).trim());
  }
}
```

テストデータ
−1,0,1,2,3,4,5,6,7,8,9,10,11,12
が, 順次入力される.

入力	出力
year=−1	
year=0	0は申年です
year=1	1は酉年です
year=2	2は戌年です
year=3	3は亥年です
year=4	4は子年です
year=5	5は丑年です
year=6	6は寅年です
year=7	7は卯年です
year=8	8は辰年です
year=9	9は巳年です
year=10	10は午年です
year=11	11は未年です
Year=12	12は申年です

まとめ

- テストデータをコードから自動的に生成
  - テストコード生成の手間が省け，テストの実施が容易になる
  - 単体テスト工程の早期完了，または，省略できた時間でソフトウェアの機能やより複雑なテストケースに焦点を当てることができる
- 生成したテストデータを用いてステートメントカバレッジ(C0)を計測
  - カバレッジという定量的なデータを示すことができる
- ステートメントの網羅状況を表示
  - プログラムの流れを追いながら，ステートメントの網羅状況を把握することができる
  - なぜ通過しないステートメントがあるのか，原因解明のヒントに？

今後の課題

- 現段階では整数型の変数のみを扱っている
  - 浮動小数点，文字型変数等のテストデータ生成方法を考察する必要がある
- 例外的な入力に対応できない
  - 整数型変数に対する文字型データの入力等
- テストデータ，実行結果等の記録方法の検討
  - 可読性を考慮した，データ記録方法を検討する必要がある
- さらに効果的なテストデータ生成方法の検討