ソフトウェアテストシンポジウム　2012　九州

Javaプログラムのための 単体テスト並列実行ツールの試作

宮崎大学　工学部　情報システム工学科

西川拳太，松岡慎吾，片山徹郎

背景

- 近年、ITが社会の隅々まで浸透
  - システムやソフトウェアのもたらす経済的・社会的影響の増大
- ソフトウェアの重要性
  - 高い品質が求められる

⇩ 一方で・・・

- 情報システムの傾向
  - **大規模化**
  - **複雑化**
  - **短納期化**

ソフトウェアの品質確保が問題となっている

品質確保の手段

- **ソフトウェアの品質を確保するには？**

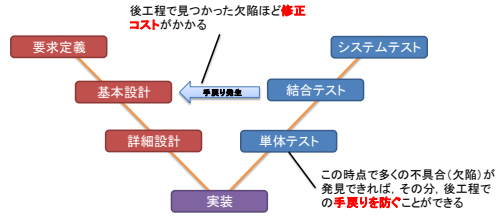
ソフトウェアの不具合を減らすこと

⇩ そのためには

- テストは欠かすことの出来ない工程
  - その中でも，単体テストは特に重要

単体テストはなぜ重要か？-(V字モデル)

- 単体テスト
  - テスト工程において、最初の段階で行われるテスト



単体テスト

- 単体テスト
  - デバックや仕様変更の際には，全ての単体テストからテストし直すことが理想的である

⇒繰り返しテストを行うことが必要とされる

  - テストの対象数が最も多く，実行や管理に手間がかかる

ソフトウェアの品質を確保するためにも，テストに多くの時間を割きたい

情報システムの傾向

- **大規模化**
- **複雑化**
- **短納期化**

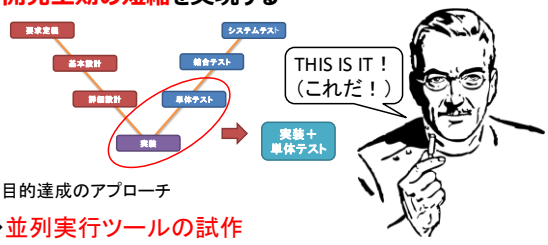
◆ テストに多く時間を割きたい

⇩ しかし

◆ 短納期化の傾向から，テスト時間は圧縮されがち

目的

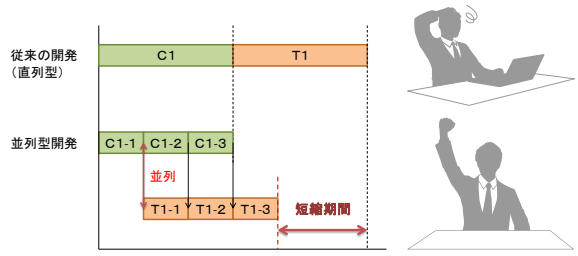
- **コーディング（実装）とテストの並列化を行うことで，開発工期の短縮を実現する**



- 目的達成のアプローチ

⇒並列実行ツールの試作

コーディングとテストの並列化



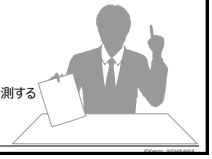
出来たものから随時テストを実行することで，開発工期を短縮できる

単体テスト並列実行ツールの特徴

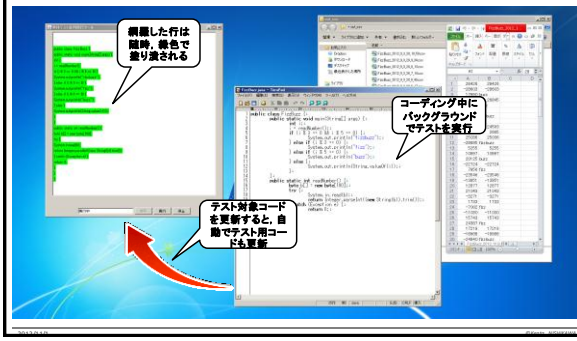
1. 対象言語：Java

2. テスト用コードを自動で生成，実行，更新
   - テスト用コードは，カバレッジの取得やテストデータを入力するために必要となる
   - テスト対象コードの更新を検知した後，テスト用コードも自動で更新する
   - 更新とはテスト対象コードを上書き保存することであり，コードの中身を変更せずに上書き保存した場合も更新したとみなす

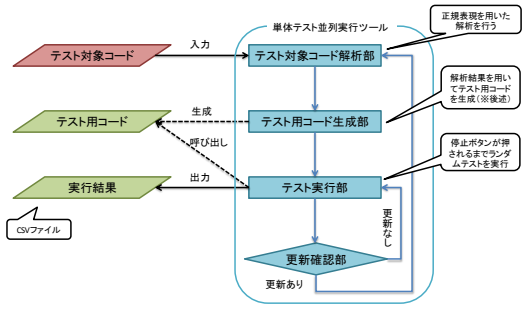
3. ランダムテストとカバレッジの計測
   - -32768～32767の範囲でランダムにテストデータを生成
   - テスト用コードを用いて，ステートメントカバレッジ（C0）を計測する



ツール外観



ツールの全体構成



テスト用コードの生成規則と生成例-（更新前）

- 解析結果を用いてテスト用コードを生成

```
public class FizzBuzz {
  public static void main(String[] args) {
    int i;
  }
}
```

テスト対象コード（更新前）

生成

解析

```
package TestPack;
public class TestCode {
  public static void TestMain() {JTool.c0[2]=1;
    int i;JTool.c0[3]=1;
  }
}
```

テスト用コード

置換前	置換後	説明
なし	package TestPack;	パッケージの挿入
class [クラス名]{	class TestCode{	クラス名の書き換え
public static void main()	public void TestMain()	main関数の書き換え
System.in.read([変数名]);	JTool.read([変数名]);	標準入力命令の書き換え
[命令文];	[命令文];JTool.c0[行数]=1;	カバレッジ計測文の埋め込み

テスト用コードの生成規則と生成例-（更新後）

- テスト対象コードが更新されるたびに，テスト用コードを生成する

```
public class FizzBuzz {
  public static void main(String[]  args) {
    int i;
    i = readNumber();
  }
  public static int readNumber() {
    byte b[] = new byte[100];
    try {
      System.in.read(b);
      return Integer.parseInt((new String(b)).trim());
    }catch (Exception e) {
      return 0;
    }
  }
}
```

テスト対象コード（更新後）

生成 →

```
package TestPack;
public class TestCode {
  public static void TestMain() {JTool.c0[2]=1;
    int i;JTool.c0[3]=1;
    i = readNumber();JTool.c0[4]=1;
  }
  public static int readNumber()  {JTool.c0[6]=1;
    byte  b[] = new byte[100];  JTool.c0[7]=1;
    try  {JTool.c0[8]=1;
      JTool.read(b); JTool.c0[9]=1;
      JTool.c0[10]=1;return  Integer.parseInt((new
String(b)).trim());
    }catch (Exception  e) {JTool.c0[11]=1;
      JTool.c0[12]=1;return  0;
    }
  }
}
```

テスト用コード

実行結果の出力

- テスト対象コードの更新を確認したらCSVファイルを作成
  - [テスト対象ファイル名]_年_月_日_時_分_秒.csv
  - 例）FizzBuzz_2012_11_1_15_40_23.csv
- 作成されたCSVファイルにテストの実行結果を出力
  - テスト実行時に生成された乱数
  - 標準出力

適用例

- 試作したツールが正しく動作することを検証するため，以下に示すJavaプログラムに適用した

➢FizzBuzzプログラム
- 入力した数字が3で割り切れる場合は“fizz”，5で割り切れる場合は“buzz”，両者で割り切れるなら“fizzbuzz”，どれにも当てはまらない場合は数字をそのまま出力するプログラム

```
public class FizzBuzz {
        public static void main(String[] args) {
                int i;
                i = readNumber();
                if (i % 3 == 0 && i % 5 == 0) {
                        System.out.println("fizzbuzz");
                } else if (i % 3 == 0) {
                        System.out.println("fizz");
                } else if (i % 5 == 0) {
                        System.out.println("buzz");
                } else {
                        System.out.println(String.valueOf(i));
                }
        }
        public static int readNumber() {
                byte b[] = new byte[100];
                try {
                        System.in.read(b);
                        return Integer.parseInt((new String(b)).trim());
                } catch (Exception e) {
                        return 0;
                }
        }
}
```


検証内容と方法

➢ 検証内容
1. テスト用コードを自動で生成，実行，更新していること
2. ステートメントカバレッジ（C0）が取得できること
3. テストの実行結果を参照できること

➢ 検証方法
- 右図に示すプログラムを最終的に作成するプログラムとして，ツールを適用しながらコーディングを開始する

```
public class FizzBuzz {
        public static void main(String[] args) {
                int i;
                i = readNumber();
                if (i % 3 == 0 && i % 5 == 0) {
                        System.out.println("fizzbuzz");
                } else if (i % 3 == 0) {
                        System.out.println("fizz");
                } else if (i % 5 == 0) {
                        System.out.println("buzz");
                } else {
                        System.out.println(String.valueOf(i));
                }
        }
        public static int readNumber() {
                byte b[] = new byte[100];
                try {
                        System.in.read(b);
                        return Integer.parseInt((new String(b)).trim());
                } catch (Exception e) {
                        return 0;
                }
        }
}
```


ファイルの更新-（1更新目）

- クラスとメインメソッドを実装したテスト対象コードに対し，ツールがテスト用コードを自動で生成
- コーディング中に，バックグラウンドでランダムテストを実行

テスト対象コード

```
public class FizzBuzz {
        public static void main(String[] args) {
        }
}
```

テスト用コード

```
package TestPack;
public class TestCode {
        public void TestMain() {Jtool.c0[2]=1;
        }
}
```


ファイルの更新-（2更新目）

- プログラマがコーディング中に，新たにメソッドを実装し更新（上書き保存）すると，テスト用コードも自動で更新

テスト対象コード

```
public class FizzBuzz {
        public static void main(String[] args) {
                int i;
                i = readNumber();
                if (i % 3 == 0 && i % 5 == 0) {
                        System.out.println("fizzbuzz");
                } else if (i % 3 == 0) {
                        System.out.println("fizz");
                } else if (i % 5 == 0) {
                        System.out.println("buzz");
                } else {
                        System.out.println(String.valueOf(i));
                }
        }
        public static int readNumber() {
                byte b[] = new byte[100];
                try {
                        System.in.read(b);
                        return Integer.parseInt((new String(b)).trim());
                } catch (Exception e) {
                        return 0;
                }
        }
}
```

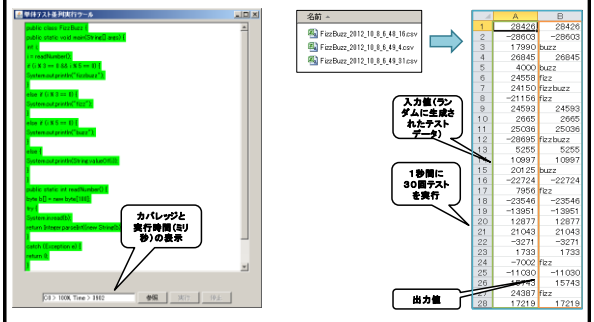
更新 / 更新

→

テスト用コード

```
package TestPack;
public class TesaCode {
        public void TestMain() {Jtool.c0[2]=1;
                int i;Jtool.c0[3]=1
                i = readNumber();Jtool.c0[4]=1
                if (i % 3 == 0 && i % 5 == 0) {Jtool.c0[5]=1;
                        System.out.println("fizzbuzz");Jtool.c0[6]=1;
                } else if (i % 3 == 0) {
                        System.out.println("fizz"); Jtool.c0[8]=1;
                } else if (i % 5 == 0) {
                        System.out.println("buzz"); Jtool.c0[10]=1;
                } else {
                        System.out.println(String.valueOf(i)); Jtool.c0[12]=1;
                }
        }
        public static int readNumber() {Jtool.c0[15]=1;
                byte b[] = new byte[100]; Jtool.c0[16]=1;
                try {Jtool.c0[17]=1;
                        System.in.read(b); Jtool.c0[18]=1;
Jtool.c0[19]=1;          return Integer.parseInt((new String(b)).trim());
                } catch (Exception e) {
Jtool.c0[21]=1;          return 0;
                }
        }
}
```

カバレッジの取得状況と実行結果



検証結果

- 適用例から，以下の検証内容をすべて満たした
  1. テスト用コードを自動で生成，実行，更新していること
  2. ステートメントカバレッジ（C0）が取得できること
  3. テストの実行結果を参照できること

□ **すなわち，試作したツールが正しく動作することを確認できた**

考察

- 開発工期の短縮を目的として，**単体テスト並列実行ツールを試作**した
  - □本研究で試作したツールを使用することで，Javaプログラムを対象としたコーディングとテストの並列化を可能にする
  - □コーディングとテストを並列化することで，ソフトウェア開発における開発工期の短縮，開発量の増加，さらには品質の確保につながると考えられる

考察-（問題点）

- 適用例から，以下に示すいくつかの問題点を発見した
  - □テストデータに重複が出てくる場合がある
  - □int型以外の型を引数としているメソッドに対応できない
  - □プログラムによっては，カバレッジを100%満たすまでに時間を要する可能性がある

まとめ

目的

開発工期の短縮

アプローチ

単体テスト並列実行ツールの試作

□コーディングとテストを並列化することで・・・

- テスト時間が圧縮されず，開発量の増加が見込まれる
- ソフトの品質確保につながり，開発工期の短縮が期待できる

今後の課題

- 実用性の検証
  - 具体的に，どの程度の割合で工期の短縮が実現できるのか
- ツールの制限
  - int型で表せる数値の範囲のみをテストデータとしている
  - 実行結果と期待値の照合ができない
- ブランチカバレッジ（C1）への対応
  - C0だけではテスト強度として弱い