Javaプログラムを対象とした単体テスト自動実行および可視化ツール“Jvis”の開発

松岡　慎吾 a)・喜多　義弘 b)・片山　徹郎 c)

Development of Automatic Unit Testing and Visualization Tool “Jvis” for Java Programs

Shingo MATSUOKA, Yoshihiro KITA, Tetsuro KATAYAMA

Abstract

Testing process in software development is vital to ensure the quality of software. It's difficult to verify the quality of software sufficiently by increased scale and complexity of software. This paper aims to improve the efficiency of testing in software development by visualizing of testing progress. As an approach to achieve the goal, An automatic unit testing and visualization tool “Jvis” (tool for Java programs to visualize testing) has been implemented. Targeted programming language of Jvis is Java language. Jvis conducts automated testing based on C0(statement coverage) and C1(branch coverage) for the test target program. And Jvis visualizes current progress of testing in real-time. Hence, Jvis can reduce the time spent on understanding and sharing current progress of testing. As a result, Jvis will lead to improve the efficiency of testing in the unit testing process.

Keywords: Software testing, Visualization of software testing, Automated software testing, Unit testing, Java programs

1. はじめに

今日、情報システムは社会に広く浸透し、人々の生活にとって欠かせない存在となっている。一方で、ソフトウェアの不具合を原因とした社会的および経済的な損害は計り知れない。米国では、ソフトウェアの不具合が年間 7 兆円の損害を与えているという報告もある [1]。このような背景から、ソフトウェアの品質保証がより重要視されている。ソフトウェアの品質を保証する上で、ソフトウェア開発におけるテスト工程は欠かすことのできない工程である。しかし、テスト工程にかかる工数は開発全体の 50%以上と言われ、非常にコストがかかる工程である [2]。さらに、ソフトウェアの大規模化、複雑化、短納期化によって、ソフトウェアの品質を十分に検証することが難しくなっている。そこで、テスト工程におけるコストの削減やテスト効率の向上を目的として、「テスト自動化」が提案されている [3]。

ソフトウェア開発におけるテスト工程は、いくつかの工程によって構成される。その中でも、単体テストは最も初期に実施され、検証すべき項目が最も多い工程である。単体テスト工程で多くの欠陥を検出することができれば、以降のテスト工程において手戻りを防ぐことができる。単体テストを完全に手動で実施することは大量のテスト項目を検証する上で効率的ではない。また、プログラムの修正を行う度に、正しく修正が行われたかどうか、または修正によって新たな欠陥が入り込んでいないかどうかを、単体テストの実施によって確認することが理想的である。このため、単体テストは他のテスト工程と比べても繰り返し実施されやすい工程であると言える。このように、単体テストの自動化はテスト工程の中でも特に効果的である [4]。

近年、ソフトウェア開発における上流工程の可視化手法(例えば、UML: Unified Modeling Language、DFD: Data Flow Diagram、ERM: Entity-Relationship Model)が提案されている [5]。これらの可視化手法は、自然言語で記述される設計書と比べて、ソフトウェアの構造の理解や開発者間の情報共有を容易にする。自然言語による文章の曖昧性を排除し、ソフトウェアの欠陥を埋め込む一番の原因だと言われている情報共有のミスを防ぐ効果もある。一方で、ソフトウェア開発における下流工程の可視化については、十分に研究が進んでいない領域である。一般に、自動化されたテストでは多量の実行結果が出力される。大量に羅列された実行結果や、自然言語で記述されたログから、テストの進捗状況を瞬時に理解することは容易ではない。また、このことは、テスト技術者間の円滑な情報共有を難しくする。テスト実施状況の理解と共有に時間を費やすことは、テス

a) 情報システム工学専攻大学院生
b) 情報システム工学専攻研究生
c) 情報システム工学科准教授

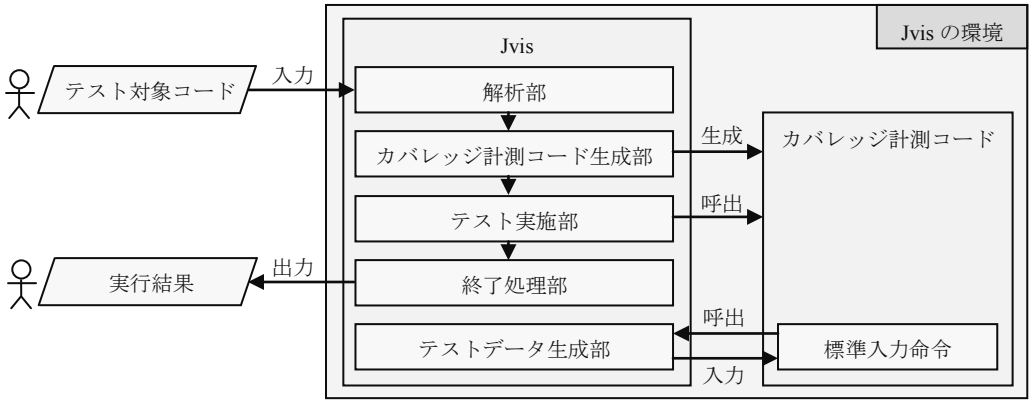

図 1. Jvis の全体構成

トの効率を低下させる 1 つの要因であると考える。

そこで、本研究では、単体テスト工程の可視化によるテスト効率の向上を目的として、Javaプログラムを対象とした単体テスト自動実行および可視化ツール“Jvis” (Tool for **J**ava programs to **vis**ualize automated testing)を開発する。Jvisは、テスト対象プログラムに対する自動テストを実施し、テスト実施状況の可視化をリアルタイムに行う。

Jvisの概要を、以下に示す。まず、テスト対象コードを元にカバレッジ計測コードを生成する。“カバレッジ計測コード”とは、テスト対象コードを元に、カバレッジの計測に必要な命令文の挿入や、命令文の書き換えを行ったコードである。次に、ステートメントカバレッジ(C0)およびブランチカバレッジ(C1)に基づいた自動テストを実施する。Jvisはテスト手法としてランダムテストを採用しており、テスト対象コードに対して、短時間で多量のテストデータを入力することができる。また、テストの実施中、テスト対象コードのハイライトによる可視化や、命令文毎の実行回数の表示によって、現在のテスト実施状況をユーザに提示する。これによって、デッドコードおよび未分岐箇所の発見を支援する。最後に、テスト実施結果をCSV(Comma-Separated Values)形式[6]でファイルに出力する。

以下、本稿の構成は次のとおりである。第2章では、Jvisの機能と実装方針について述べる。第3章では、Jvisの適用例を示す。第4章では、Jvisについての考察を行う。第5章では、本研究のまとめと今後の課題について述べる。

2. Jvis の機能と実装方針

この章では、Jvis の機能および実装方針について説明する。

2.1 機能

Jvis は、大きく 3 つの機能を持っている。

- テストデータの自動生成およびテストの自動実施
  Jvis はテストデータをランダムに生成し、テスト対象コードに入力することによってテストを自動実施する。
- C0 および C1 の計測
  テストを実施することによって、テスト対象コードの C0 と C1 を計測する。計測した C0 と C1 のうち、C1 は自動テストの終了条件として用いる。
- テスト実施状況の見える化
  テストの実施によって網羅した命令文をハイライトし、テストの実施状況を視覚化する。また、命令文毎の実行回数をユーザに提示する。

Jvis の全体構成を、図 1 に示す。Jvis は、解析部、カバレッジ計測コード生成部、テスト実施部、テストデータ生成部、終了処理部の 5 つで構成する。

まず、ユーザはテスト対象の Java プログラムをファイル選択ダイアログから選択する。解析部では、ユーザの選択したテスト対象コードを読み込む。また、読み込んだテスト対象コードに対して、C1 に基づいたテストの可視化に必要な命令文を追記したコード(C1 可視化コード)を、Jvis 上に表示する。

解析部の処理が終わると、カバレッジ計測コード生成部に移行する。カバレッジ計測コード生成部では、C1 可視化コードに対して正規表現 [7]を用いたパターンマッチを行い、カバレッジの計測に必要な命令文の挿入や、命令文の書き換えを行う。また、標準入力命令をテストデータ生成部の呼び出し命令に書き換え、ユーザによる入力を自動化する。

カバレッジ計測コードの生成が終わると、テスト実施部に移行し、カバレッジ計測コードを実行する。テスト実施部が呼び出すテストデータ生成部は、標準入力命令によるユーザの入力に代わってランダムデータを生成し、カバレッジ計測コードに対して入力を行う。テスト実施部がカバレッジ計測コードを 1 回実行する度に、命令文および分岐の網羅状況を取得し、C0 および C1 を算出する。取得した各網羅状況は、Jvis 上に表示する C1 可視化コードをハイライトすることによって視覚化する。カバレッジ計測コードを実行し、C1 可視化コードをハイライトするという一連の処理は、C1 の値が 100%に達するまで繰り返し行う。

テスト実施部が終了すると、終了処理部に移行し、入力に使用したテストデータと、テストデータの入力によって取得した実行結果を、CSV 形式でファイルに出力する。

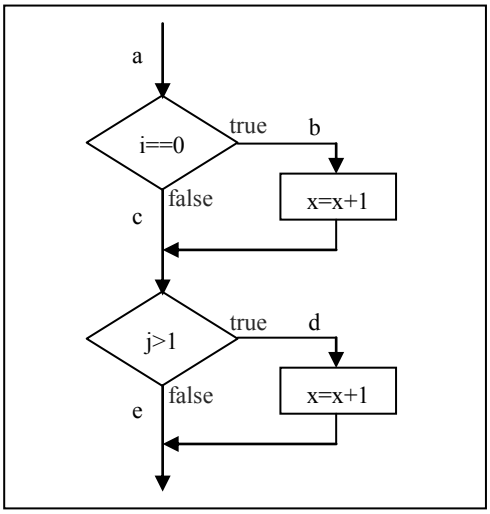

図 2. フローチャート

C0 に基づいたテストでは、テスト対象コードの C0 を 100%満たし、すべての命令文を網羅していても、一度も実行されないパスが存在する可能性がある。図 2 のフローチャートを用いて説明すると、例えば、i=0、j=2 を入力すればすべての命令文を網羅できるが、パス c、パス e を通るテストが抜け落ちてしまう。仮に、2 番目の分岐条件を間違えて j>0 と実装していた場合の欠陥は、C0 に基づいたテストでは発見することができない。一方、C1 に基づいたテストでは、各条件分岐が true、false の結果を少なくとも 1 回は持つようにテストを実施するため、フローチャートに示すすべてのパスを網羅できる。このように、C1 に基づいたテストは、C0 に基づいたテストだけでは検出できない欠陥を網羅できる、より強力なテスト手法である。Jvis では、元のテスト対象コードだけでは検出できない欠陥を、C1 可視化コードとしてユーザに提示することによって、分岐条件のミスによる不具合の検出を支援する。

Jvis の外観を、図 3 に示す。以下に、各パーツの説明を行う。

① ソースコード表示ラベル
C1 可視化コードを表示し、テスト実施によって網羅した命令文を随時ハイライトする。また、命令文毎の実行回数を表示する。なお、網羅した命令文は若草色で表示する。

② メッセージボックス
ユーザへのメッセージを表示する。テストの実施中、現在の C0 および C1 の値を表示する。カバレッジ計測コードが実行時にハングアップした場合は、ハングアップを起こした行番号および命令文を表示する。

③ 参照ボタン
ボタンを押すと、ファイル選択ダイアログを表示する。ファイル選択ダイアログからテスト対象コードを選択すると、テスト対象コードを読み込み、カバレッジ計測コードの生成を行う。

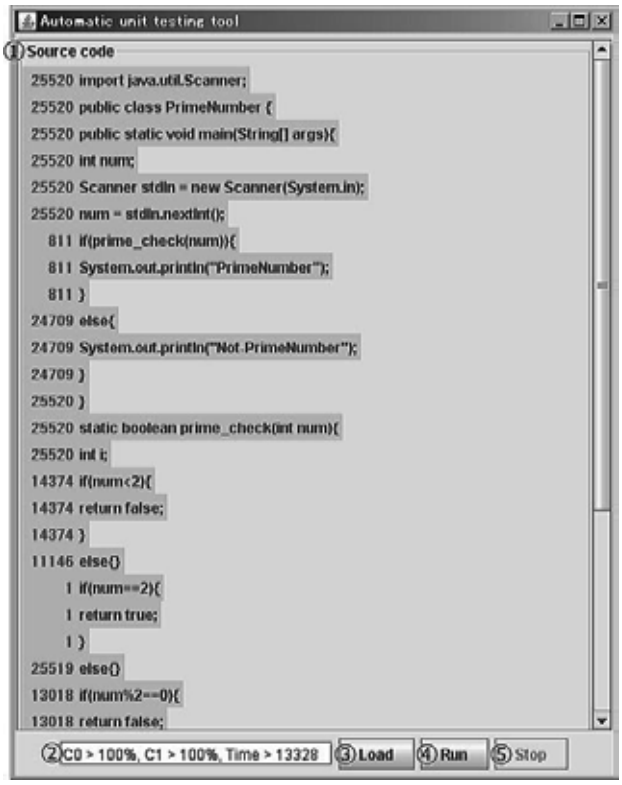

図 3. Jvis の外観

④ 実行ボタン
ボタンを押すと、生成したカバレッジ計測コードに対して自動テストを実施する。

⑤ 停止ボタン
ボタンを押すと、自動テストの実施を終了する。停止ボタンを押すか、実施中のカバレッジ計測コードが C1 の値を 100%満たすまでテストを実施し続ける。

2.2 実装方針

この節では、Jvis を構成する各部の実装方針について説明する。

2.2.1 解析部

解析部では、ユーザが選択したテスト対象コードを読み込む。この際、テスト対象コードを 1 行ずつ読み込み、String 型の配列に格納する。なお、C1 に基づいたテストの実施状況を可視化するため、else 文を持たない if 文と、default 文を持たない switch case 文に対しては、それぞれ else 文、default 文を追記した C1 可視化コードとして格納する。ここで追記する else 文、default 文は、それぞれ else{}、default: とし、テスト対象コードの処理内容に変更が生じることはない。また、C1 可視化コードに正規表現を用いたパターンマッチを行うことによって、

表 1. カバレッジ計測コード生成部の主な処理一覧

処理内容	パターン	パターンマッチ後の処理
パッケージの挿入	なし ※先頭行に挿入	→ package autotest;
クラス名の書き換え	.*class¥¥s+"+[クラス名]+"¥¥s*¥¥{.*	class [クラス名]{ → class MySample{
main 関数の書き換え	.*public¥¥s+static¥¥s+void¥¥s+main¥¥s*¥¥(.*¥¥).*	Public static void main(){ → public void MyMain(){
標準出力の記憶	.*System.out.(print|println).*	System.out.println([出力]); → System.out.println([出力]);Sample.output[実行回数][出力数]=[出力];
標準入力命令の書き換え	.*System.in.read.* など	System.in.read([変数名]); → Sample.read([変数名]);
命令文に対するカウンタの挿入	なし ※挿入によるエラーが発生しないすべての命令文が対象。例外として、解析部で追記した命令文は対象外。エラーが発生する文法箇所はパターンマッチで検索する。	[命令文]; → [命令文];Sample.statement[行数]++;
分岐命令に対するカウンタの挿入	.*if¥¥s*¥¥(.*¥¥).* など	if(条件式){ → if(条件式){ Sample.branch[行数]++;

- C1 の値を算出する際の分母として必要な分岐数のカウント
- カバレッジ計測コードの生成時に必要なクラス名の検索および抽出

を行う。最後に、C1 可視化コードを、ソースコード表示ラベルに表示する。このとき、配列に格納した C1 可視化コード 1 行毎に、HTML(HyperText Markup Language)[8]の<TABLE>タグで囲む処理を行い、表示を行う。各命令文の実行を検出すると、<TABLE>タグの背景色を若草色に変更し、命令文単位で網羅状況の視覚化を行う。

2.2.2 カバレッジ計測コード生成部

カバレッジ計測コード生成部では、C1 可視化コードを元に、カバレッジの計測に必要な命令文の挿入や、命令文の書き換えを行ったカバレッジ計測コードを生成する。命令文の挿入や書き換えを行うため、C1 可視化コードに対して 1 行ずつパターンマッチを行う。カバレッジ計測コードの生成における主なパターンと、命令文の挿入および書き換えの処理を、表 1 に示す。

命令文を網羅したことを検出するため、int 型の配列 statement を宣言し、すべての命令文に対してカウンタ statement[行番号]++;を挿入する。配列 statement は、各要素の初期値を 0 とし、命令文を実行すると、対応した配列要素をインクリメントする。なお、解析部で else 文を持たない if 文と default 文を持たない switch case 文に対して追記した else 文、default 文については、カウンタ statement[行番号]++;の挿入を行わない。

また、分岐を網羅したことを検出するため、int 型の配列 branch を宣言し、すべての分岐命令に対してカウンタ branch[行番号]++;を挿入する。命令文に対するカウンタ同様に、配列 branch は、各要素初期値を 0 とし、分岐を行うと、対応した配列要素をインクリメントする。カウンタ branch[行番号]++;は、解析部で追記した else 文、default 文についても挿入を行う。

以上の命令文および分岐に対するカウンタの設置によって、C1 可視化コードを元に生成したカバレッジ計測コードでも、ユーザが入力したテスト対象コードの C0 および C1 を正しく求めることができる。

また、カバレッジ計測コード生成部では、ユーザへの入力を求める標準入力命令を、メソッド名を元に、対応するテストデータ生成部の呼び出し命令に書き換える。

2.2.3 テスト実施部

テスト実施部では、カバレッジ計測コードを実行する。また、カバレッジ計測コードの実行によって取得した命令文および分岐の網羅状況を元に、ソースコード表示ラベルに表示する C1 可視化コードを 1 行毎にハイライトする。網羅した命令文は、若草色でハイライトする。また、カバレッジ計測コードを実行する度に、ソースコード表示ラベルに表示する各命令文の実行回数を、カウンタの値を元に更新する。

なお、取得した命令文および分岐の網羅状況を元に、C0 および C1 を算出し、C1 の値が 100%に達する、もしくは停止ボタンを押すまで、テスト実施部はカバレッジ計測コードの実行を繰り返す。

2.2.4 テストデータ生成部

テストデータ生成部では、ユーザへの入力を求める標準入力命令に代わって、ランダムなテストデータを生成する。

表 2. 対応メソッドと戻り値

メソッド名	テストデータ生成部が return する値
nextInt()	Min: -32678, Max: 32767 の整数
nextDouble()	Min: -32678.0, Max: 32767.0 の浮動小数点数
nextBoolean()	true または fault
nextLine()	a-z, A-Z で構成する 5-10 文字の文字列
System.in.read()	Min: -32678, Max: 32767 の整数の byte 値

カバレッジ計測コード生成部によってテストデータ生成部の呼び出し命令に書き換えることができるメソッドには制限があるが、int 型、double 型、Boolean 型、String 型の生成および入力に対応している。現バージョンにおいて、ランダムデータの生成に対応しているメソッドと、戻り値として return する値の一覧を、表 2 に示す。なお、入力ストリームから入力された数値や文字、文字列を byte 型で受け付ける System.in.read メソッドは、その後のキャストで任意の型に変換できる汎用的なメソッドだが、現バージョンで Jvis が生成する型は、int 型に限る。

2.2.5 終了処理部

カバレッジ計測コードの C1 の値が 100%に達した場合、または停止ボタンを押した場合、テスト実施部から終了処理部に移行する。終了処理部では、実行結果をファイルに出力する。ここで、ファイルには CSV 形式を用いる。実行結果として、

- 入力に使用したテストデータ
- 標準出力命令による出力

をファイルに書き出す。上記2点の実行結果を記録するため、カバレッジ計測コード生成の際に、これらを出力する命令文を挿入している。なお、それぞれの実行結果は、1 回のカバレッジ計測コードの実行につき、0個～複数個出力する。1回の実行で使用したテストデータと、標準入力による出力は、1 レコードとして、同一行にまとめてファイルに出力する。

3. 適用例

本研究で開発を行った Jvis が正しく動作することを検証するため、2 つの Java プログラム「素数判定プログラム」と「誤った FizzBuzz プログラム」を Jvis に適用した。

図 4 に、「素数判定プログラム」を示す。このプログラムをテスト対象コードとして Jvis に与え、カバレッジ計測コード生成部が生成したカバレッジ計測コードを、図 5 に示す。以下に、図 4 のプログラムを元に、図 5 のカバレッジ計測コードを生成する処理について説明する。なお、図 4、図 5 の左にある番号は行番号である。

- カバレッジ計測コードの呼び出しに必要なパッケージの挿入を行う(図 5 の 0 行目)。
- クラス名を、テスト実施部であらかじめ指定しているクラス名「MySample」に書き換える(2 行目)。
- main 関数を、テスト実施部あらかじめ指定しているメソッド名「MyMain」に書き換える(3 行目)。
- 命令文の網羅状況を取得するため、挿入によってエラーが発生しないすべての命令文の直後、または直前にカウンタ statement[行番号]++;を挿入する(3, 4, 5, 6, 7, 8, 10, 11, 12, 14, 15, 16, 17, 19, 20, 21, 23, 24, 25, 27, 28, 29, 30, 32, 33, 34 行目)。
- 分岐の網羅状況を取得するため、分岐命令の直後にカウンタ branch[行番号]++;を挿入する(7, 10, 16, 19, 20, 23, 24, 27, 29, 32 行目)。
- 標準出力命令 System.out.println()の直後に、出力の格納を行う代入文を挿入する(8, 10 行目)。
- 標準入力命令である Scanner クラスの nextInt()を、テストデータ生成部の呼び出し命令 Sample.nextInt()に書き換える(6 行目)。

```
 1  import java.util.Scanner;
 2  public class PrimeNumber {
 3      public static void main(String[] args){
 4          int num;
 5          Scanner stdIn = new Scanner(System.in);
 6          num = stdIn.nextInt();
 7          if(prime_check(num)){
 8              System.out.println("PrimeNumber");
 9          }
10          else{
11              System.out.println("Not-PrimeNumber");
12          }
13      }
14      static boolean prime_check(int num){
15          int i;
16          if(num<2){
17              return false;
18          }
19          if(num==2){
20              return true;
21          }
22          if(num%2==0){
23              return false;
24          }
25          for(i=3;i*i<=num;i+=2) {
26              if(num%i==0){
27                  return false;
28              }
29          }
30          return true;
31      }
32  }
```

図 4. 素数判定プログラム

図 2 は、素数判定プログラムのテストを終了した後の Jvis の外観である。今回適用した素数判定プログラムは、C1 の値を 100%満たすことができたため、テストの実施を自動的に停止した。図 2 のソースコード表示ラベルに表示するすべての命令文、および解析部で追記を行った else 文を、若草色にハイライトしていることから、すべての命令文および分岐を実行できたことが分かる。

図 6 に、Microsoft Office Excel で開いた素数判定プログラムの実行結果の一部を示す。A 列は、テストデータ生成部で生成し、カバレッジ計測コードに対して入力を行ったテストデータであり、B 列は、テストデータに対するプログラムの出力結果である。

```
 0  package autotest;
 1  import java.util.Scanner;
 2  public class MySample{
 3      public void MyMain(){Sample.statement[3]=1;
 4              int num;Sample.statement[4]=1;
 5              Scanner stdIn = new Scanner(System.in);Sample.statement[5]=1;
 6              num = Sample.nextInt();Sample.statement[6]=1;
 7              if(prime_check(num)){Sample.statement[7]=1;Sample.branch[7]=1;
 8              System.out.println("PrimeNumber");Sample.statement[8]=1;Sample.output[Sample.run_count][Sample.count++]="PrimeNumber";
 9              }
10              else{Sample.statement[10]=1;Sample.branch[10]=1;
11              System.out.println("Not-PrimeNumber");Sample.statement[11]=1;Sample.output[Sample.run_count][Sample.count++]="Not-PrimeNumber";
12              }Sample.statement[12]=1;
13      }
14      static boolean prime_check(int num){Sample.statement[14]=1;
15              int i;Sample.statement[15]=1;
16              if(num<2){Sample.statement[16]=1;Sample.branch[16]=1;
17                      Sample.statement[17]=1;return false;
18              }
19              else{Sample.branch[19]=1;}
20              if(num==2){Sample.statement[20]=1;Sample.branch[20]=1;
21                      Sample.statement[21]=1;return true;
22              }
23              else{Sample.branch[23]=1;}
24              if(num%2==0){Sample.statement[24]=1;Sample.branch[24]=1;
25                      Sample.statement[25]=1;return false;
26              }
27              else{Sample.branch[27]=1;}
28              for(i=3;i*i<=num;i+=2){Sample.statement[28]=1;
29                      if(num%i==0){Sample.statement[29]=1;Sample.branch[29]=1;
30                              Sample.statement[30]=1;return false;
31                      }
32                      else{Sample.branch[32]=1;}
33              }Sample.statement[33]=1;
34              Sample.statement[34]=1;return true;
35      }
36  }
```

図 5. 素数判定プログラムのカバレッジ計測コード

次に、誤った FizzBuzz プログラムを適用する。誤った FizzBuzz プログラムの一部を、図 7 に示す。if 文を用いた分岐の構成を間違えており、3 番目の分岐(“FizzBuzz”の表示)が実行されることはない。すなわち、この分岐内のコードはデッドコードとなる。図 8 は、誤った FizzBuzz プログラムのカバレッジ計測コードを、約 5 分間実行し続けた Jvis の外観である。デッドコードはハイライトせず、C0 の値は 89%、C1 の値は 83%で頭打ちになっている。誤った FizzBuzz プログラムの行数 28 行に対して網羅した命令文が 25 行(25÷28≒0.893)であり、プログラム中に 6 個ある分岐に対して網羅した分岐が 5 個(5÷6≒0.833)であるため、テスト対象である誤った FizzBuzz プログラムの C0 および C1 の値を正しく取得している。C1 の値が 100%に達することによるテストの自動停止が不可能なため、停止ボタンを押すことによってテストの実施を停止した。実行結果の確認を行ったところ、デッドコードである“FizzBuzz”の出力は確認できなかった。

以上 2 つのプログラムを適用した結果、実装方針どおりにカバレッジ計測コードの生成ができており、Jvis が正しく動作することを確認できた。また、ソースコード表示ラベルに表示する C1 可視化コードをハイライトすることによって、デッドコードおよび未分岐箇所を、ユーザに正しく提示できることを確認できた。

4. 考察

本研究では、可視化によるテスト効率の向上を目的として、単体テスト自動実行および可視化ツール“Jvis”を開発した。Jvis は、テスト対象コードに対してカバレッジに基づいた自動テストを実施する。また、命令文および分岐の

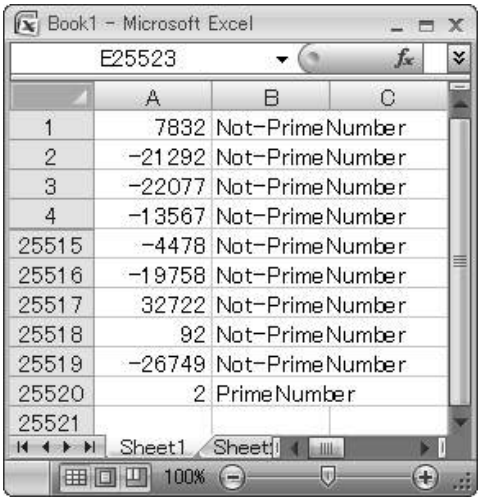

図 6. 素数判定プログラムの実行結果の一部

網羅状況を取得するカウンタを挿入することによって、テスト実施状況をリアルタイムに視覚化し、ユーザに提示する。この可視化によって、C0 および C1 カバレッジを基準としたテスト実施状況の直感的な理解を支援する。C1 はテストの終了基準として用い、テスト対象コードが C1 の値を 100%満たすと自動的にテストを終了する。テスト実施後、取得した実行結果を CSV 形式でファイルに出力する。

Jvis は、テスト手法としてランダムテストを採用している。手動によるランダムテストは、同値分割や境界値分析といったドメインベースのテストに比べて効率が悪い。しかし、自動テストとしてランダムテストを用いた場合、短時間で相当数のテストデータを入力することができる。また、ランダムデータの入力によるプログラムの網羅状況を

```
if(i%3==0){
        System.out.println("Fizz");
}
else if(i%5==0){
        System.out.println("Buzz");
}
else if(i%3==0 && i%5==0){
        System.out.println("FizzBuzz");
}
else{
        System.out.println(String.valueOf(i));
}
```

図 7. 誤った FizzBuzz プログラムの一部

視覚化することによって、自動テストによる多量のテストデータが、どこをどれだけテストしたのかを容易に理解することができる。命令文毎の実行回数に応じて重点的にテストを行うなど、検証すべき箇所を絞った効率的なテスト実施の可能性を考えることができる。また、テスト実施状況をリアルタイムに視覚化しているため、一般のテスト結果報告書では確認できないテスト対象コードの網羅状況を、動的に見ることができる。その振る舞いから、探索型テスト[9]実施に役立つ情報を得られる可能性がある。

単体テスト自動実行ツールとして、GAIO TECHNOLOGY 社の“カバレッジマスターwinAMS”[11]がある。カバレッジマスターwinAMS は、テスト対象コードに対する入力値と、入力値に対応する期待結果を記述した CSV ファイルを入力として与えることによって、テストを自動実施するツールである。テスト実施によって、入出力結果のレポートや、カバレッジのレポートが出力される。また、C1 を視覚化するために計装した C1 可視化コードや、各カバレッジを計測するために生成したカバレッジ計測コードのようなプローブを使用せず、テスト対象コードをそのまま実行することができる。

Jvis とカバレッジマスターwinAMS を比較した際の、Jvis の短所を以下に示す。

- 入力値に対する実行結果の照合はユーザが行う必要がある: 想定外のテストデータを入力した場合や、長時間連続で実行した場合に発生するハングアップの検出は容易だが、多量のテストデータに対する実行結果の照合を行うには人手が必要である。
- 時間のタイミングによって動作が変わるようなプログラムの場合、期待する動作を行うことができない場合がある: Jvis が動的に実行するコードは、テスト対象コードに手を加えたカバレッジ計測コードである。ロジックに変更が生じないように計装を行っているが、新たに挿入したカウンタや、出力結果の格納を行う命令文の影響によって、実行時間が異なる可能性がある。
- プログラムの修正に伴うデグレードの確認および回帰テストに手間がかかる: テスト実施の度に入力値が変わるため、同じデータで再度テストを実施することが困難である。

しかし、Jvis は、テスト対象コードの構造の理解や、テストコードの記述を行うことなく、多量のテストデータによって短時間でカバレッジベースの自動テストを行うことができる。また、プログラムの構造や開発者の思考に関係なくテストデータを生成するため、テスト対象コードの信頼性が高まることも考えられる。また、膨大なテストデータの入力を行うものの、命令文毎に網羅状況を示し、実行回数の表示も行っているため、どこをどれだけテストしているのかを一目で理解することができる。以上から、Jvis を用いたテストを実施することによって、自動テストによるテスト実施状況の理解が容易になり、テスト効率が向上することが考えられる。

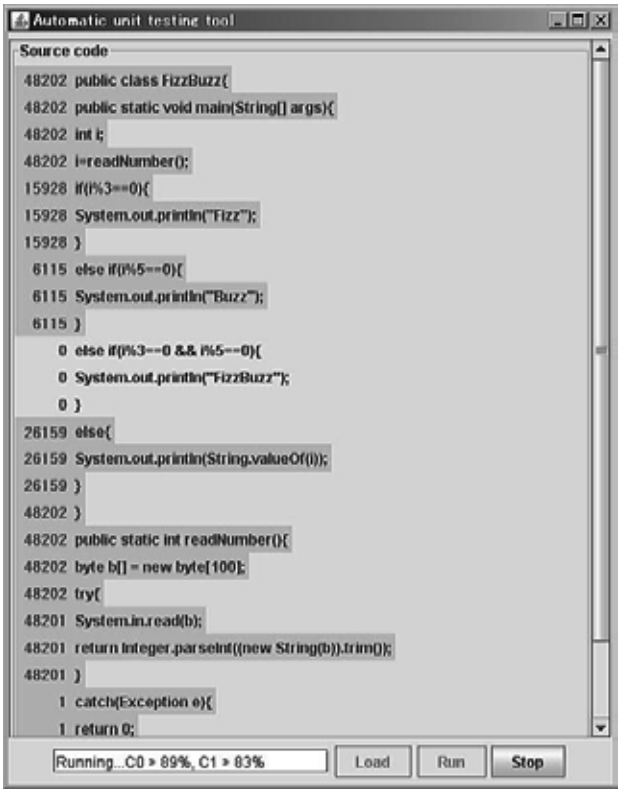

図 8. 誤った FizzBuzz プログラムを実行中の Jvis の外観

また、デグレードの確認や回帰テストのように何度も繰り返すテストを効率的に実施することができないという短所については、対応策を考えている。実行結果の保存形式としてCSV ファイルを用いていることから、2 回目以降のテスト実施において、1 回目のテスト結果を保存した CSV ファイルの再利用が可能である。このため、テストを繰り返し実施する場合に、前回の実行結果を次回の期待結果として用いることによって、プログラム修正後の実行結果と比較できるような拡張を考えている。

5. おわりに

本研究では、ソフトウェア開発における単体テスト工程の可視化によるテスト効率の向上を目的とし、Java プログラムのための単体テスト自動実行および可視化ツール“Jvis”を開発した。

Jvis は、カバレッジ計測コードを生成し、カバレッジ計測コードにテストデータを入力することによってテストを自動実施する。また、カバレッジ計測コードに命令文お

よび分岐の網羅状況を取得するカウンタを埋め込むことによって、ユーザに対してテストの実施状況をリアルタイムに提示する。テスト終了後、取得した実行結果は CSV 形式でファイルに出力する。

Jvis を使用した自動テストを実施することによって、ユーザは、テスト対象コード中のどこをどれだけテストしたのかを容易に理解することができる。また、リアルタイムなテスト実施状況の可視化は、ユーザに対して、より詳細なテストの実施に役立つ情報を提供できる可能性がある。

以上から、本研究で開発した Jvis を使用することによって、可視化によるテスト実施状況の理解を促進し、単体テスト工程のテスト効率が向上することが見込まれる。また、リアルタイムなテスト実施状況の可視化によって、ソフトウェアの品質向上に繋がることが考えられる。

以下に、今後の課題を挙げる。

- 見える化の強化

現バージョンでは、C0 および C1 カバレッジのみを視覚化している。今後は、デシジョンテーブルや同値クラス、境界値分析など、テスト実施におけるプログラムの動きや構造の見える化に取り組む。実行回数の少ない命令文に対して、どういったテストを実施すれば詳細なテストを実施できるのか、といった品質向上に役立つ情報の提示にも繋がるのではないかと考えている。

- 可視性の向上

テスト対象コードのみを用いたテスト実施状況の可視化では、可視性の向上に限界がある。例えば、C1 に基づいたテストの可視化については、現状の命令文を追記しハイライトする方法ではなく、フローチャートを用いた分岐の見える化を行った方が可視性が向上することが考えられる。

- 入力値と期待値の比較

現バージョンでは、入力値に対する実行結果の照合をユーザが行う必要がある。テストコードを記述することなく、C0 および C1 ベースの自動テストを実施できるメリットはあるが、入力値に対する出力が正しい結果なのかを膨大な実行結果から検証することは難しい。上流工程での成果物である仕様書を新たな入力として与えるなど、期待値を抽出できる仕組みを考える必要がある。または、入力値と実行結果を視覚化することによって、入力値と期待値の比較検証の負担を軽減できる仕組みを考える。

参考文献

1) G. Tassey: The Economic Impacts of Inadequate Infrastructure for Software Testing, National Institute of Standards and Technology, 2002.
2) E. Kit: Software Testing in the Realworld; Improving the Process, ACM Press/Addison-Wesley, 1995.
3) J. Edvardsson: A Survey on Automatic Test Data Generation, Proc. 2nd Conference on Computer Science and Engineering, pp. 21-28, 1999.
4) A. Bacchelli, P. Ciancarini, D. Rossi: On the Effectiveness of Manual and Automatic Unit Test Generation, Proc. 3rd International Conference on Software Engineering Advances, pp. 252-257, 2008.
5) Y. T. Lee: Information Modeling; From Design to Implementation, Proc. 2nd World Manufacturing Congress, pp. 27-30, 1999.
6) J. Repici: HOW-TO; The Comma Separated Value (CSV) File Format, http://www.creativyst.com/Doc/Articles/CSV/CSV01.htm, accessed Feb. 22. 2012.
7) J. E. F. Friedl: Mastering Regular Expressions (3rd edition), O'Reilly Media, 2006.
8) B. Kennedy, C. Musciano: HTML; The Definitive Guide (3rd edition), O'Reilly Media, 1998.
9) J. Itkonen, K. Rautiainen: Exploratory Testing; A Multiple Case Study, 2005 International Symposium on Empirical Software Engineering, pp. 84-93, 2005.
10) International Software Testing Qualifications Board: Standard Glossary of Terms Used in Software Testing (version 2.1), 2010.
11) GAIO TECHNOLOGY Co., LTD.: カバレッジマスター winAMS, http://www.gaio.co.jp/product/dev_tools/pdt07_winams.html, accessed Feb. 22. 2012.