特別研究報告

題目

クラス間関係を利用した
単体テストおよび静的検査の網羅率可視化ツールの試作と評価

指導教員
楠本 真二 教授

報告者
武藤 祐子

平成 22 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 21 年度 特別研究報告

クラス間関係を利用した
単体テストおよび静的検査の網羅率可視化ツールの試作と評価

武藤 祐子

内容梗概

ソフトウェアの品質のひとつの尺度として仕様と実装の一致性を測る方法があり，それを調べる手段として単体テストや静的検査などがある．単体テストとは対象モジュールに対して設計された仕様を満たすかどうかを確認するテストである．実際にはテスト仕様書に基づいてテストデータとなるテストケースを作成し，テストランと呼ばれるプログラムを実行することでテストが行われることが多い．この手法の問題点として，テスト仕様書で想定するケースに漏れがある場合はその箇所のテストが行われないため，テスト結果の品質がテストケースに依存することが挙げられる．一方，プログラムを実行せずにソースコードを検査することを一般に静的検査という．静的検査を行うツールの一つとして ESC/Java2 がある．これは，メソッドが満たすべき性質を仕様として JML(Java Modelling Language) を用いて記述されたソースコードに対して，ソースコードが仕様と矛盾しないか否かを判定するツールである．このツールを用いた静的検査の問題点として，検査の品質が記述した仕様の質に依存するということや，結果がクラス単位であり，かつテキストで出力されるため，クラス間の呼び出し関係が読み取りにくく，結果が分かりづらいということが挙げられる．

本稿で行ったことは以下の通りである．単体テストおよび静的検査の結果とクラスにおけるメソッド呼び出し情報を用いてこれらの情報を総合的に視覚化することにより，より精緻なソフトウェア品質の情報提供を可能にすることを目標に，重み付き有向グラフを表示するツールを Eclipse のプラグインとして試作をした．このツールを JML による仕様が記述された在庫管理プログラムに対して適用し，評価を行った．仕様と実装が一致しているかを確認するためクラスの呼び出し関係と UML(Unified Modeling Language) との比較を行い，前述の問題点を解決するため単体テストの結果と静的検査の結果の比較を行った．結果として，対象ソフトウェアのソースコードにおいて不要な記述を発見でき，テスト品質および対象ソフトウェアの品質を推定することができた．

主な用語

単体テスト，静的検査，可視化，ESC/Java2，JUnit，Eclipse

目 次

図 目 次

1　まえがき

ソフトウェアの品質は国際標準化機構 [2] や日本工業規格 [3] において定義されており，機能性，信頼性，使用性，効率性，保守性，移植性の 6 つの品質特性から構成されている．その内の機能性とはソフトウェアが必要な要件を満たしているか否かという尺度であり，少なくとも仕様に記述されている内容が実装されていなければならない．現在ソフトウェア開発における仕様と実装の一致性を測る手段として単体テストや静的検査などがあり，またソフトウェア製品の様々なメトリクスをタグ化する StagE プロジェクト [4, 5] でもこれらの項目がタグとして仕様制定されている [6].

単体テスト [7] とは対象モジュールに対して設計された仕様を満たすかどうかを確認するテストでありソフトウェアテストの最も初期の段階に位置づけられる．一般にバグが開発の後期の段階に残存するほどバグの修正コストが増大するため， 単体テストで最大限にバグを減らすことが求められる．その方法としてはテスト仕様書に基づいてテストデータとなるテストケースを作成し，テストランと呼ばれるプログラムを実行することでテストが行われる．この手法の問題点として，テスト仕様書で想定するケースに漏れがある場合はそのケースのテストが行われないため， テストの品質がテストケースに依存することが挙げられる．

一方，プログラムを実行せずにソースコードを検査することを一般に静的検査という．静的検査を行うツールの一つとして ESC/Java2[8] がある．これは，メソッドが満たすべき性質を仕様として JML(Java Modelling Language) [9, 10, 11] を用いて記述されたソースコードに対して，ソースコードが仕様と矛盾しないか否かを判定するツールである．このツールを用いた静的検査の問題点として，検査の品質が記述した仕様の質に依存するということや，結果がクラス単位であり，かつテキストで出力されるため，クラス間の呼び出し関係が読み取りにくく，結果が分かりづらいということが挙げられる

本報告では，より精緻なソフトウェア品質の情報提供を可能にすることを目標に， 単体テストおよび静的検査の結果とクラスにおけるメソッド呼び出し情報を用いてこれらの情報を総合的に視覚化した．具体的には，重み付き有向グラフを表示するツールを総合開発環境 Eclipse[12] のプラグインとして試作した.

このツールを JML による仕様が記述された在庫管理プログラム [1] に対して適用し，評価を行った．仕様と実装が一致しているかを確認するためクラスの呼び出し関係と UML(Unified Modeling Language) との比較を行い，前述の問題点を解決するため単体テストの結果と静的検査の結果の比較を行った.

結果として，対象ソフトウェアのソースコードにおいて不要な記述を発見でき，テスト品質および対象ソフトウェアの品質を推定することができた.

以降，2 章で準備として単体テストおよび静的検査に用いられる用語とツールについて述

べ，3 章で提案手法において用いる定義と仮説について述べる．4 章において作成したツールの概要と仕様について述べ，5 章で前章で作成したツールを用いて行った実験の結果とその考察について述べる．最後に 6 章で本研究のまとめと今後の課題について述べる．

2　準備

本章では単体テストおよび静的検査に関する一般的な定義や使用されるツールについて述べる．

2.1　単体テスト

単体テスト [13] とはモジュールを単位として行うテストである．ソフトウェアテストにおいて最も初期の段階に行われる．テストの指標として命令網羅率，分岐網羅率，条件網羅率などがある．

命令網羅率 (Statement Coverage)　テスト対象ソースコードに対する実行されたステートメントの割合であり，全てのステートメントを 1 回以上実行すれば命令網羅率は 100%となる．

分岐網羅率 (Branch Coverage)　テスト対象ソースコードに対する実行された分岐の割合であり，全ての分岐を 1 回以上実行すれば分岐網羅率は 100%となる．

条件網羅率 (Condition Coverage)　テスト対象ソースコードに対する分岐のうち実行された条件の割合であり，全ての条件を 1 回以上実行すれば条件網羅率は 100%となる．

2.1.1　JUnit

JUnit[14] は Java を対象とした単体テストを行うフレームワークである．テストケースと呼ばれるクラスにテスト対象メソッドを実行するコードを記述しておき，それを実行することでテストを行う．

2.1.2　JCoverage

JCoverage[15] はテストの網羅率を算出するツールである．パッケージ単位およびクラス単位で命令網羅率と分岐網羅率を算出したカバレッジ・レポートを提供する．テスト対象ソースコードのバイトコードに対してデバッグプログラムを予め埋め込んでおき，コードが実行されると埋め込まれたデバッグプログラムが作動し情報を収集し，網羅率を求める方式を取っている．従って JUnit などのテストツールに依存しない形となっている．

2.1.3 djUnit

djUnit[16] とは，ビルドツール ant[17] を利用して JUnit と JCoverage を手軽に使えるようにしたツールであり，Eclipse プラグインとして提供されている．JCoverage のカバレッジ・レポートをエクスポートする機能を持つ．なお内部で使用している JCoverage に変更が加えられている部分があり，分岐網羅率に関してはオリジナルの JCoverage とは異なる値となる．

2.2 静的検査

2.2.1 JML

JML とはソースコードに対して満たすべき仕様を表現するための言語であり，Java ソースコード中にアノテーションとして記述する．不変条件，事前条件，代入可否，事後条件などの制約を記述することができる．

不変条件 クラスのインスタンスが存在している間，常に成立しなければならない条件である．invariant 節を用いて記述する．

事前条件 メソッドが実行される前の状態で成立する式であり，主に引数に対する制約である．requires 節を用いて記述する．

代入可否 代入を許可するフィールドを指定する．assignable 節を用いて記述する．

事後条件 メソッドが実行された後の状態で成立する式である．ensures 節を用いて記述する．

次に JML 記述の例を示す．

```
1  class Person {
2    private String name;
3    /*@ public invariant !name.equals(""); @*/
4    /*@ public behavior
5    requires nm != null && !nm.equals("");
6    assignable name;
7    ensures name.equals(nm);
8    @*/
9    public void setName(String nm) {
10     name = nm;
11   }
12 }
```

Person クラスは名前を表す name フィールドおよび name に対する setter である setName メソッドを持つ．名前は常に空文字列であってはならないため，name フィールドに対する不変条件を 3 行目のように記述する．setName メソッドにおいては，引数に与えられた格納

したい名前 nm が Null や空文字列であってはならないため 5 行目のように事前条件を記述し，このメソッドでは name フィールドを更新するため 6 行目のように代入可否を記述し，このメソッドの処理が完了した時点において name フィールドと引数 nm が等しい必要があるため 7 行目のように事後条件を記述する．

2.2.2 ESC/Java2

ESC/Java2[18, 19] は JML で仕様が記述された Java ソースコードに対して，仕様とソースコードが妥当であるかを検査するツールである．ESC とは Extended Static Checker の頭文字であり，静的検査ツールの一種である．検査対象は Java1.4 であり，基本的にはソースコードに JML を記述しておく必要があるが，次に示すパッケージに関しては予め仕様が用意されておりこれらを用いて検査が行われる．

- java.lang の一部 (基本型，例外等)
- java.util の Collection 関係
- java.util.regex (正規表現)
- java.io(入出力関係)
- java.math.BigInteger
- java.awt の一部 (Color や MouseListener 等 6 つ)
- javax.swing の一部 (4 つ)
- junit.framework の一部 (Assert，TestCase)

このツールの入力と出力について述べる．入力は 1 つの Java ソースコードファイルである．出力は各メソッド毎の妥当である (passed) または妥当でない (failed) という結果に加え，通過しなかった場合は反例情報である．

3 提案手法

本章では，本研究において用いる定義と仮説について述べる．

3.1 定義

単体テストおよび静的検査のカバレッジについて述べる．なおいずれもクラスを単位としている．

3.1.1 単体テストのカバレッジ

命令網羅率を用いた．その理由として，命令網羅率は最も単純でありわかりやすいこと，分岐網羅率は djUnit で再計算がおこなわれており JCoverage の結果と異なること，条件網羅率は JCoverage や djUnit が対応していないことが挙げられる．

3.1.2 静的検査のカバレッジ

静的検査のカバレッジの定義は次の通りである．クラス A の持つメソッドのうち妥当であるメソッド数を $M_{passed}(A)$，クラス A の持つメソッド数を $M(A)$ とすると，クラス A の静的検査のカバレッジ $C_s(A)$ を

$$C_s(A) = M_{passed}(A)/M(A) \tag{1}$$

と定義する．

例を挙げて説明する．次の Example クラスはコンストラクタと method メソッドの 2 つのメソッドを持つ．

```
1 public class Example {
2    public Example() {}
3
4    public int method(String str) {
5        return str.length(); //no null check
6    }
7 }
```

これを ESC/Java2 で検査すると次のような出力が得られる．

```
1 ESC/Java version ESCJava-2.0.5
2     [0.023 s 8230232 bytes]
3
4 Example ...
5   Prover started:0.072 s 12311784 bytes
6     [1.018 s 12577160 bytes]
7
8 Example: Example() ...
9     [0.045 s 12278712 bytes] passed
```

```
10
11 Example: method(java.lang.String) ...
12 ----------------------------------------------------------------------
13 Example.java:7: Warning: Possible null dereference (Null)
14       return str.length(); //no null check
15              ^
16 ----------------------------------------------------------------------
17    [0.081 s 12311768 bytes] failed
18  [1.146 s 12312648 bytes total]
19 1 warning
```

9 行目よりコンストラクタは妥当であり，17 行目より method メソッドは妥当でないことが確認できる．従って Example クラスの持つメソッドのうち妥当であるメソッド数 $M_{passed}(Example)$ は 1，Example クラスの持つメソッド数 $M(Example)$ は 2 であり，式 (1) より Example クラスの静的検査のカバレッジ $C_s(Example)$ は 50%となる．

3.1.3 呼び出し関係

クラス A がクラス B の持つメソッドを n 回呼び出す場合，クラス A はクラス B を n 回呼び出すと定義する．呼び出し回数とはソースコード中に登場する回数である．

次に例を示す．

```
1 Class Example2{
2   public void method1() {
3     Class1 c = new Class1(); //call Class1
4     c.method2(); //call Class1
5   }
6 }
```

Example2 クラスは 3 行目で Class1 クラスのコンストラクタを呼び出し，4 行目で Class1 クラスのメソッド method2 を呼び出している．従って Example2 クラスは Class1 クラスを 2 回呼び出していることになる．

3.2 仮説

3.2.1 単体テストのカバレッジが高く，静的検査のカバレッジが低い場合

仕様記述が妥当である場合　テストケースにおいて想定されているパターンが不足していることが考えられ，単体テストの品質が低いといえる．従ってテスト仕様書を見直す必要がある．対象ソフトウェアの品質は高いとは言えない．

仕様記述が妥当でない場合　仕様記述が厳密すぎることが考えられる．対象ソフトウェアの品質は高いことが期待できる．

3.2.2　単体テストのカバレッジが低く，静的検査のカバレッジが高い場合

仕様記述が妥当である場合　テストが途中であり，テストケースの記述が不足している．対象ソフトウェアの品質は高いことが期待できる．

仕様記述が妥当でない場合　仕様記述が充分でなく書き足す必要がある．対象ソフトウェアの品質は高いとは言えない．

3.3　可視化

重み付き有向グラフを表示する．

3.3.1　ノード

ノードはクラスに対応し，単体テストまたは静的検査のカバレッジを円グラフで表現する．

3.3.2　エッジ

エッジは呼び出し関係に対応する．呼び出し回数をエッジの重みとし，エッジの太さで表現する．

4 実装

本章では作成したツールについて述べる．

4.1 概要

Eclipse のプラグインとして実装を行った．

4.1.1 規模

空行とコメント行を除いておよそ 2000 行であり，パッケージ数は 14，クラス数は 33 である．

4.1.2 処理の流れ

処理の流れを図 1 に示す．

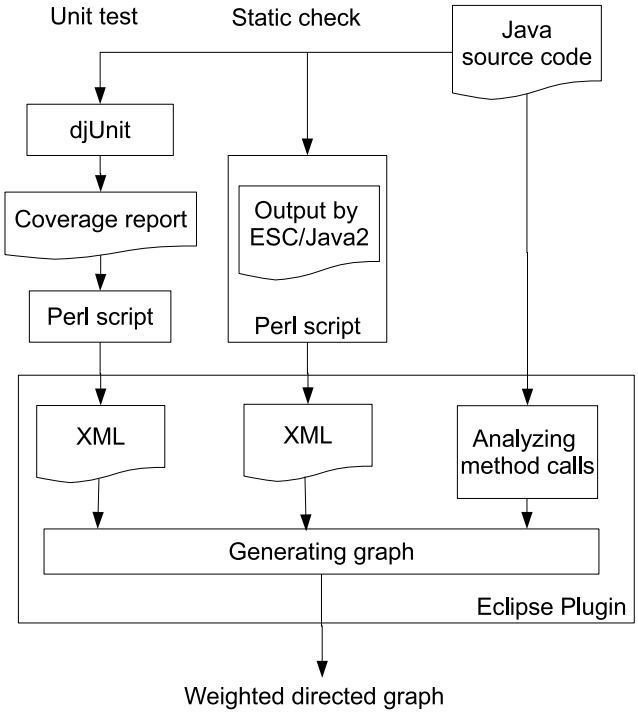

図 1: 処理の流れ

単体テストの処理手順について述べる．ユーザは予め対象ソースコードを djUnit に入力してカバレッジレポートを出力しておく必要がある．Perl スクリプトはカバレッジレポートを

読み込みクラス名やカバレッジ情報などを持つ XML を生成する機能を持っており，Eclipse プラグインが Perl スクリプトを実行することで XML が出力される．

静的検査の処理手順について述べる．Perl スクリプトは ESC/Java2 を実行しその出力からクラス名，カバレッジ情報，メソッド情報などを持つ XML を生成する機能を持っており，Eclipse プラグインが Perl スクリプトを実行することで XML が出力される．

それぞれから出力された XML からカバレッジ等のクラス情報を取得し，対象ソースコードから呼び出し関係を求め，重み付き有向グラフを生成して表示する．

4.2 開発環境

開発言語は Java 1.6 であり，Eclipse Galileo 上で開発を行った．その際 Eclipse 同梱のプラグイン開発環境 PDE(Eclipse Plug-in Development Environment)[20] を利用した．

4.2.1 ライブラリ

Eclipse プラグインの内部においてライブラリとして MASU と JUNG を用いた．

MASU ソースコードを解析しメトリクス計測に必要な情報をユーザに提供するソフトウェアである [21]．クラスがどのメソッドを呼び出しているかを取得することができ，呼び出し関係の算出に使用した．

JUNG Java Universal Network/Graph Framework の略であり，Java を対象としたグラフ構造ライブラリである [22]．

4.3 仕様

4.3.1 入力

入力は対象ソースコード，XML 生成コマンド，XML 設置ディレクトリである．

対象ソースコード 対象言語は Java ソースコードであり，バージョン 1.4 までに対応している．この制約は ESC/Java2 に依るものである．

XML 生成コマンド XML を生成するためのコマンドを入力する．図 1 における Perl スクリプトに対応する．

XML 設置ディレクトリ XML 生成コマンドにより生成される XML を設置するディレクトリであり，ここに設置された XML が Eclipse プラグイン読み込まれる．

4.3.2 使用方法

Eclipse のプロジェクトメニューより，Create Graph を選択することで処理が開始される．同様の処理はプロジェクトエクスプローラにおいて対象としたいプロジェクトの右クリックメニューから Create Graph を選択することや，メインビューのツールバーからでも行うことができる．

4.3.3 XML

XML の形式を次に示す．

```
<?xml version="1.0" encoding="utf-8"?>
<class>
        <packageName>パッケージ名</packageName>
        <simpleName>クラスの単純名</simpleName>
        <methodCount>メソッド数</methodCount>
        <passedCount>通過したメソッド数</passedCount>
        <coverage>カバレッジ</coverage>
        <methods>
                <method id="1">
                        <name>メソッド名</name>
                        <parameter>引数 1,引数 2,...</parameter>
                        <attributes>
                                <attribute>
                                        <title>属性タイトル</title>
                                        <value>属性値</value>
                                </attribute>
                        </attributes>
                </method>
                <method id="2">
                .
                .
                </method>
        </methods>
</class>
```

上記の XML に対する EBNF 記法を用いた構文規則を次に示す．なお XML の構文規則は文献 [23]，Java の構文規則は文献 [24] に示されている．

```
class = '<class>'packageName simpleName (count | coverage) [methods]'</
    class>'
count = methodCount passedCount
methods = '<methods' [ 'id="' Java における正の整数値 '"'] '>' {method} '</
    methods>'
method = '<method>' methodName [parameters] [attributes] '</method>'
attributes = '<attributes>' {attribute} '</attributes>'
attribute = '<attribute>' title value '</attribute>'

packageName = '<packageName>' Java におけるパッケージ名 '</packageName>'
simpleName = '<simpleName>' Java におけるクラス名 '</simpleName>'
methodCount = '<methodCount>' Java における正の整数値 '</methodCount>'
```

```
passedCount = '<passedCount>' Java における正の整数値 '</passedCount>'

methodName = '<name>' Java におけるメソッド名 '</name>'
parameters = '<parameter>' parameter {,parameter} '</parameter>'
parameter = Java におけるクラスの完全限定名

title = '<title>' XML における要素の内容 '</title>'
value = '<value>' XML における要素の内容 '</value>'
```

class 要素が 1 つのクラスに対応しており，パッケージ名，クラスの単純名，カバレッジに加えてメソッド情報を持つことができる．method 要素が 1 つのメソッドに対応しており，メソッド名，引数に加えて複数の属性情報を持つことができる．属性情報は後述するメソッドビューで表示するためのものである．attribute 要素が 1 つの属性情報に対応しており，属性タイトルおよび属性値を持つ．

4.3.4 ビュー

このツールはメインビュー，メソッド情報ビューの 2 つのビューを持つ．図 2 にスクリーンショットを示す．

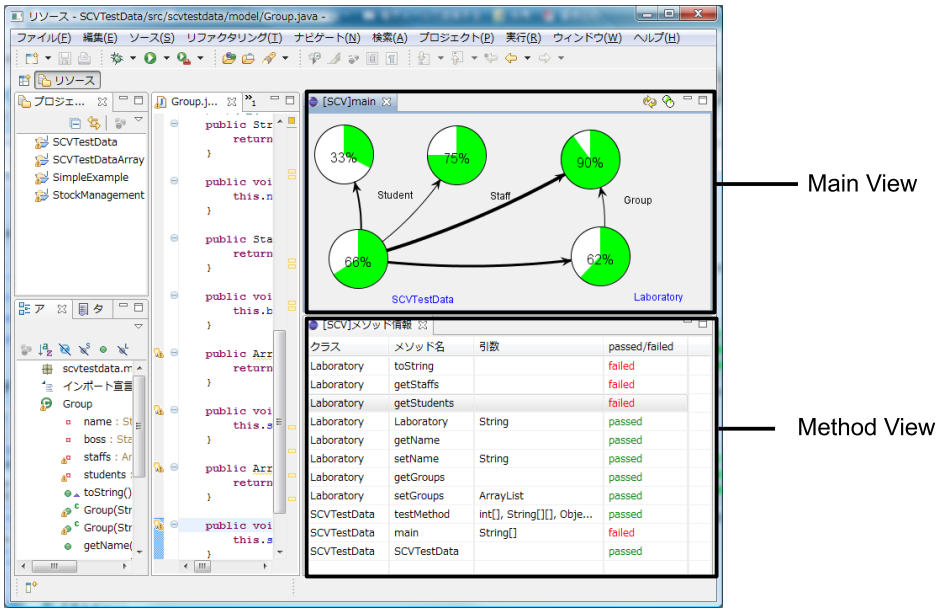

図 2: スクリーンショット

メインビュー　重み付き有向グラフを表示するビューである．スクリーンショットを図 3 に示す．1 つのノードが 1 つのクラスに対応しており，カバレッジを円グラフの形式で表示し，

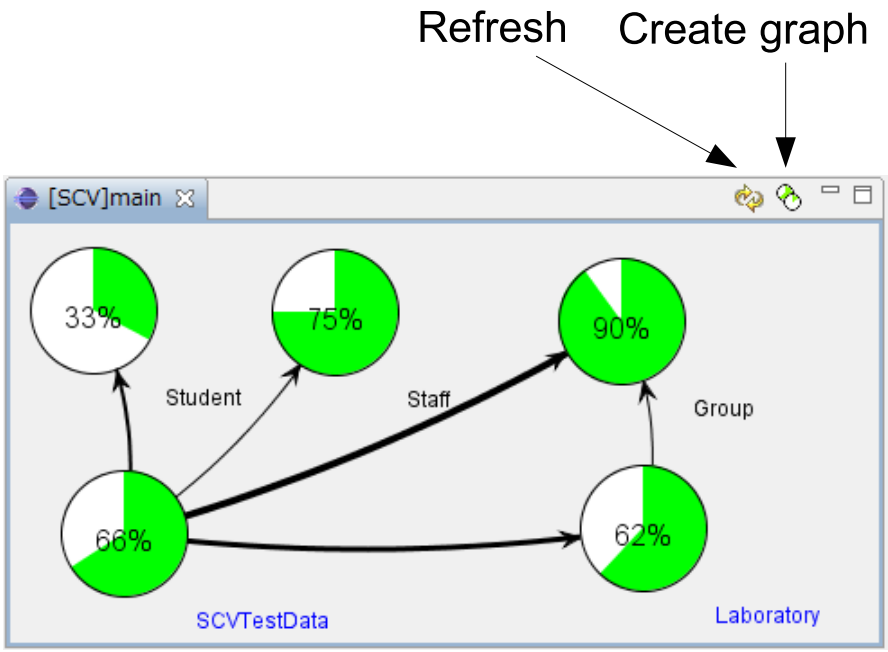

図 3: メインビュー

呼び出し関係をエッジとして表現する．エッジの太さは呼び出し回数に応じて変化させている．ツールバーにリフレッシュボタンとグラフ作成ボタンを持つ．マウスによってノードを複数個選択することができ，選択されているノードはクラス名が青色で表示される．

メソッド情報ビュー　メインビューで選択されているクラスが持つメソッドを表示するビューである．スクリーンショットを図 4 に示す．1 行が 1 つのメソッドに対応しており，行をダブルクリックすることで該当するメソッドのソースコードを表示することができる．1 列目にクラス名，2 列名にメソッド名，3 列目に引数，4 列目以降は属性情報を表示する．例えば図 4 の四角で囲まれている行は Laboratory クラスの String1 つを引数に持つ setName メソッドに対応し，このメソッドは ESC/Java2 による検査の結果，仕様と実装が妥当であったことを示す．

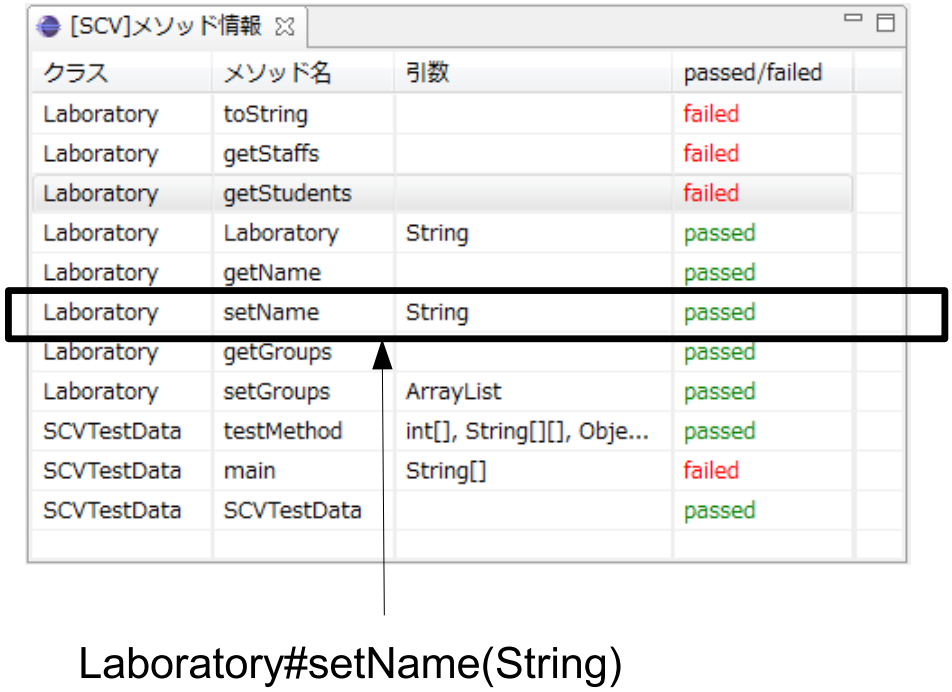

図 4: メソッド情報ビュー

5 評価実験

5.1 目的

仮説に対する実際の例として 1 つの適用例を示すこと，およびツールの有用性について考察するため，評価実験を行った．

5.2 方法

評価対象プログラムをツールに適用し，メインビューにグラフを表示した．

5.2.1 対象

評価対象は在庫管理プログラム [1] であり，次のクラスを持つ．

- ContainerItem
- Customer
- Item

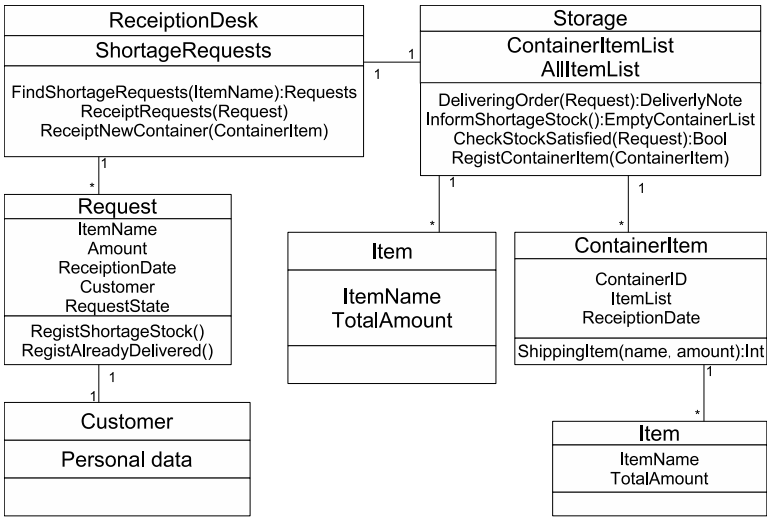

図 5: 在庫管理プログラムのクラス構成 [1]

- ReceiptionDesk
- Request
- Storage
- StockState

このプログラムにはソースコード中に JML が厳密に記述されており，制約条件という意味での仕様と実装の一致率が高く，ソフトウェア品質が高いと言ってよい．

図 5 に在庫管理プログラムの UML 図を示す．

5.2.2 単体テストのテストケース

単体テストを行うため，各クラスに対応するテストケースを独自に記述した．テストケースを付録として本論文の最後に添付する．コンストラクタと setter/getter メソッドを実行することを方針として記述したため，不充分なテストケースとなっている．

Storage クラスは，containerlist フィールドおよび allitemlist フィールドを持つが，これらに対応する getter(setItemList，getContainerItemList) に関するテストケースは記述していない．その理由は該当フィールドを操作するコンストラクタや setItemList，setContainerItem 等の単純な setter がなく，期待値と実際の値との比較ができないためである．

5.3　結果

5.3.1　単体テスト

単体テストに関してメインビューに出力されたグラフを図 6 に示す.

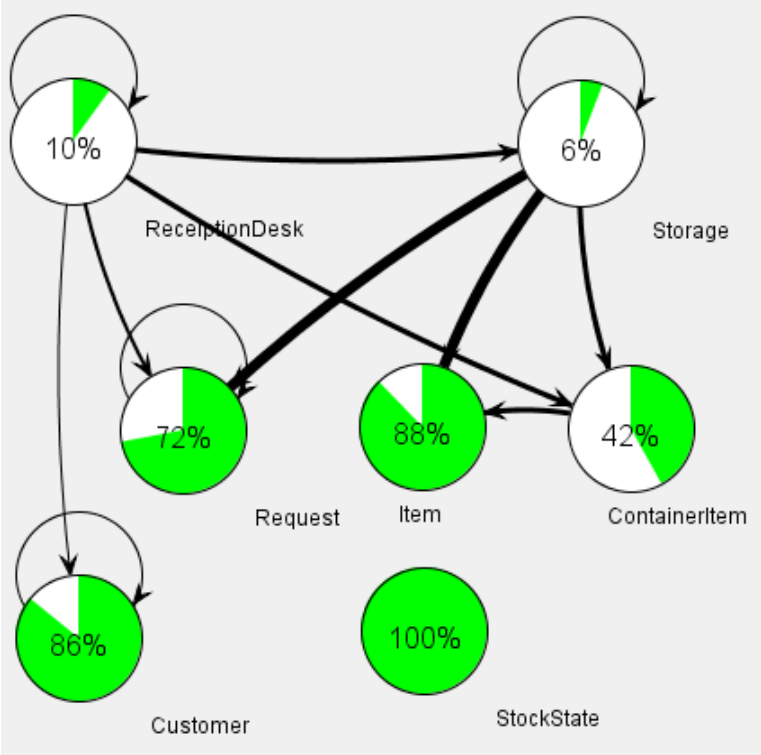

図 6: 単体テストの実行結果

5.3.2　静的検査

静的検査に関してメインビューに出力されたグラフを図 7 に示す.

5.4　考察

5.4.1　単体テスト

図 6 において，呼び出し関係のエッジが多いクラスほどより多くのクラスを呼び出している．従ってエッジの本数から必要なスタブの量を把握することができる．

また葉になるクラスほどカバレッジが高いことが確認できる．テストケースはコンストラクタ，setter/getter メソッドのみを実行するよう記述されているため，葉になるクラスは処

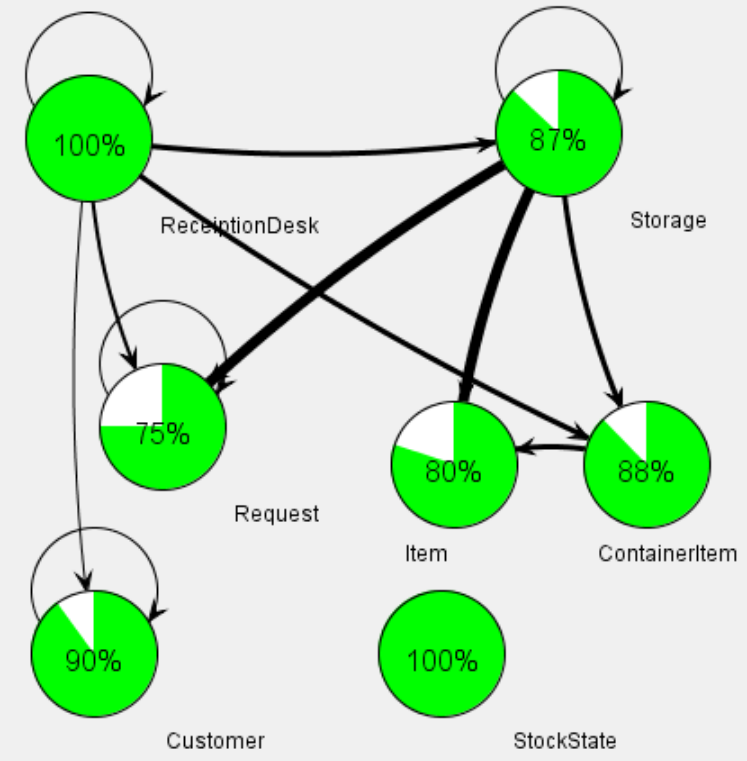

図 7: 静的検査の実行結果

理が単純であることが予測できる．実際に図 5 の UML においても該当するクラスはメソッドが少ないため，処理が単純であると言える．

5.4.2 静的検査

図 7 より全てのクラスが高いカバレッジを持っていることが確認できる．この結果は文献 [1] と比較しても一致する．

呼び出し関係を考慮したカバレッジの算出 ReceiptionDesk クラスを例として取り上げる．ReceiptionDesk クラスはカバレッジが 100%であり問題がないと考えられるが，次のようなクラスを呼び出していることに着目する．

- Storage クラス (87%)
- ContainerItem クラス (88%)
- Request クラス (75%)
- Customer クラス (90%)

ReceiptionDesk クラスに問題が無い場合でも，例えば Request クラスに問題があれば ReceiptionDesk クラスにも影響が出ることが考えられる．従って ReceiptionDesk クラスの品質は呼び出し先クラスに影響を受けるはずであり，呼び出し関係を考慮したカバレッジの算出が今後の課題となる．

5.4.3 単体テストと静的検査の比較

図 6 の単体テストのカバレッジ，図 7 の静的検査のカバレッジを仮説に適用する．なお静的検査の仕様記述に関しては文献 [1] より妥当であると考えてよい．

単体テストのカバレッジと静的検査のカバレッジが共に高いクラス Customer クラス，Request クラス，Item クラス，StockState クラスが該当する．これらのクラスに関しては仕様と記述が一致しており，従ってソフトウェア品質が高いと言える．

単体テストのカバレッジは低く，静的検査のカバレッジは高いクラス ReceiptionDesk クラス，Storage クラス，ContainerItem クラスが該当する．これらのクラスに関しては単体テストが途中であり，テストケースの記述が不足していると考えられる．実際に，用意したテストケースはコンストラクタと setter/getter 以外については記述されておらず，テストケースは不足している．対象ソフトウェアの品質は高いことが期待できる．

5.4.4 呼び出し関係と UML 図の比較

StockState クラス　図 6，図 7 において StockState クラスに関してエッジが存在しないことが確認できる．また図 5 の UML には StockState クラスは存在しない．そのため StockState クラスは未使用であり不要である可能性が考えられる．

次に StockState クラスのソースコードを示す．

```
1 public class StockState{
2   public final static byte SHORTAGE=0, SATISFYED=1, DELIVERED=2, WAIT=3;
3 };
```

StockState は定数のみを持ち，メソッドを持たないクラスであることがわかる．在庫管理プログラムの作成者に問い合わせたところ，StockState クラスは列挙型を表現するために存在するクラスである，との回答を得た．よってエッジが存在しないことから未使用であるかどうかは判別できない．

ソースコード中から StockState クラスを使用している記述を検索した結果，Request クラス，ReceiptionDesk クラス内で使用されていた．例として Request クラスの registShortageStock メソッドの記述を示す．

```
1 public void registShortageStock()
2 {
3         requestState = StockState.SHORTAGE;
4 }
```

このように StockState クラスに関しては，メソッドを呼び出すのではなくフィールドが参照されていることが確認できる．

本研究において呼び出し関係はメソッドが呼び出しのみを対象としているが，フィールドの参照に関しても考慮することが必要となる．

ReceptionDesk クラスから ContainerItem クラスへの呼び出し　図 6，図 7 において ReceptionDesk クラスが ContainerItem クラスを呼び出しているが，これに対応する記述は図 5 の UML には存在しない．

ソースコードにおいて ReceptionDesk クラスにおいて ContainerItem クラスのメソッドを呼び出す記述は次に示す receiptNewContainerItem メソッドのみであった．

```
1 public void receiptNewContainerItem(ContainerItem c) {
2   ContainerItem citem = new ContainerItem(c.getContainerID(),c.
        getContainedItem(),c.getReceiptionDate());
3   storage.registContainerItem(c);
4   deliveringOrder();//未発送リクエストの発送チェック
5 }
```

2 行目の ContainerItem クラスのコンストラクタやその引数での getContainerID メソッド，getContainedItem メソッド，getReceiptionDate メソッドを呼び出す記述が該当する．

しかし 2 行目で宣言されたローカル変数 citem はその後使用されておらず，不要な記述であることがわかる．

6　あとがき

本研究では以下の研究について述べた．品質を求めることを最終目的とし，まず，クラスを単位とする単体テストのカバレッジと静的検査のカバレッジを定義し，それらの関係について仮説を立てた．クラスの呼び出し関係とクラスのカバレッジを可視化するツールを試作し，それを在庫管理プログラムに適用し，仕様と記述の一致性を確認した．結果として，仕様と記述の不一致箇所としてソースコード中の不要記述を発見し，その他の箇所については仕様と記述が一致しておりソフトウェアの品質は高いものの，テスト品質に問題があることを示した．

今後の課題として，メソッドの呼び出しだけでなく，フィールドの参照に関しても呼び出し関係として扱うことについて検討することが挙げられる．また静的検査において，そのクラスのみを考慮してカバレッジの計算をおこなっているが，呼び出しているクラスのカバレッジを用いてそのクラスのカバレッジを算出することで，より実態に近い情報が得られると考えられる．さらに単体テストや静的検査だけでなく，他のメトリクスへ対応することも挙げられる．

謝辞

日頃から御教授下さり, 本研究ならびに本報告書の作成に当たって様々な御指摘と御指導を賜った大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 楠本真二教授に深く感謝の意を表します.

本研究ならびに本報告書の作成に当たり, 細部に至るまで貴重な御指摘と熱心な御指導を賜った大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 岡野浩三准教授に深く感謝の意を表します.

本報告書の作成に当たり適切な御指摘とご助言を頂きました大阪大学 大学院情報科学研究科コンピュータサイエンス専攻 肥後芳樹助教に深く感謝の意を表します.

本研究に多大なるご助言ご指導を頂きました大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 柿元健助教に深く感謝の意を表します.

本研究ならびに本報告書の作成に当たり，大変貴重かつ重要なデータを提供してくださった大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 博士後期課程 尾鷲方志氏に厚くお礼申し上げます.

最後に, お世話になった大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 楠本研究室の皆様に厚くお礼申し上げます.

参考文献

[1] 尾鷲方志, 岡野浩三, 楠本真二: “在庫管理プログラムの設計に対する JML 記述と ESC/Java2 を用いた検証の事例報告”, 電子情報通信学会論文誌. D, Vol. 91, No. 11, pp. 2719–2720, 2008.

[2] ISO: “Software Engineering-Product Quality-Part 1 : Quality Model,” *ISO/IEC : 9126-1:2001*, 2001.

[3] 日本規格協会: “ソフトウェア製品の品質-第 1 部 : 品質モデル”, *JIS X0129-1*, 2003.

[4] K. Inoue: “Software Tag for Traceability and Transparency of Maintenance,” *In Proc. of the 24th IEEE International Conference on Software Maintenance (ICSM 2008)*, pp. 476–477, 2008.

[5] “StagE Project,” *http://www.stage-project.jp/*.

[6] StagE プロジェクト: “ソフトウェアタグ規格 第 1.0 版”,, 2008.

[7] G. J. Myers: “ソフトウェア・テストの技法”, 近代科学社, 1980.

[8] P. Chalin: “Early detection of JML specification errors using ESC/Java2,” *SAVCBS ’06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pp. 25–32, 2006.

[9] P. Chalin, P. James, and G. Karabotsos: “The Architecture of JML4, a Proposed Integrated Verification Environment for JML,” *Department of Computer Science and Software Engineering, Concordia University*, 2007.

[10] C. Yoonsik and P. Ashaveena: “Specifying and checking method call sequences of Java programs,” *Software Quality Journal*, Vol. 15, No. 1, pp. 7–25, 2007.

[11] L. Burdy, M. Huisman, and M. Pavlova: “Preliminary Design of BML: A Behavioral Interface Specification Language for Java bytecode,” *In Fundamental Approaches to Software Engineering (FASE 2007)*, pp. 215–229, 2007.

[12] “Eclipse,” *http://www.eclipse.org/*.

[13] 山田茂: “ソフトウェアマネジメントモデル入門”, 共立出版株式会社, 1999.

[14] “JUnit,” *http://www.junit.org/*.

[15] “JCoverage,” *http://www.jcoverage.com/*.

[16] “djUnit,” *http://works.dgic.co.jp/djwiki/*.

[17] “Apache Ant,” *http://ant.apache.org/*.

[18] C. Flabagan, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata: “Extended static checking for Java,” *Proc. of the ACM SIGPLAN 2002*, pp. 234–245, 2002.

[19] “ESC/Java2,” *http://secure.ucd.ie/products/opensource/ESCJava2/*.

[20] 竹添直樹, 志田隆弘, 奥畑裕樹, 里見知宏, 野沢智也: “Eclipse 3.4 プラグイン開発徹底攻略”, 毎日コミュニケーションズ, 2009.

[21] 三宅達也, 肥後芳樹, 楠本真二, 井上克郎: “多言語対応メトリックス計測プラグイン開発基盤 MASU の開発”, 電子情報通信学会論文誌. D, Vol. 92, No. 9, pp. 1518–1531, 2009.

[22] JO’Madadhain, D. Fisher, S. White, and Y. Boey: “The jung (java universal network/graph) framework,” *http://jung.sourceforge.net/*, 2003.

[23] T. Bray and J. Paoli: “Extensible Markup Language (XML) 1.0 (Fourth Edition),” *http://www.w3.org/TR/2006/REC-xml-20060816/*, 2006.

[24] J. Gosling, B. Joy, G. Steele, and G. Bracha: “Java(TM) Language Specification, The (3rd Edition),” *Addison Wesley*, 2005.

付録

単体テストで用いたテストケースを以下に示す.

ソースコード 1: ContainerItemTest.java

```
package test_for_thesis;

import java.util.Arrays;
import java.util.Date;
import java.util.LinkedList;
import java.util.List;
import StockManagement.ContainerItem;
import StockManagement.Item;
import junit.framework.TestCase;

public class ContainerItemTest extends TestCase {

	private Item[] items;
	private List itemList;
	private Date date;
	private ContainerItem containerItem1;
	private ContainerItem containerItem2;
	private ContainerItem containerItem3;

	protected void setUp() throws Exception {
		super.setUp();
		items = new Item[5];
		items[0] = new Item("品物A",30);
		items[1] = new Item("品物B",60);
		items[2] = new Item("品物C",20);
		items[3] = new Item("品物D",10);
		items[4] = new Item("品物E",50);
		itemList = new LinkedList(Arrays.asList(items));
		date = new Date((long)0);

		//コンストラクタのテスト
		containerItem1 = new ContainerItem(1, itemList, date);
		containerItem2 = new ContainerItem(2,itemList);
		containerItem3 = new ContainerItem(3);
	}

	public void testGetReceiptionDate() {
		assertEquals(date, containerItem1.getReceiptionDate());
	}

	public void testGetContainedItem() {
		assertEquals(itemList, containerItem2.getContainedItem());
	}

	public void testGetContainerID() {
		assertEquals(3, containerItem3.getContainerID());
	}

}
```

ソースコード 2: CustomerTest.java

```
package test_for_thesis;

import StockManagement.Customer;
import junit.framework.TestCase;

public class CustomerTest extends TestCase {

	private Customer customer1;
	private Customer customer2;

	protected void setUp() throws Exception {
		super.setUp();

		//コンストラクタのテスト
		customer1 = new Customer();
		customer2 = new Customer("name2", "address2", "222-2222");
	}

	public void testGetName() {
		assertEquals("name2", customer2.getName());
	}

	public void testGetAddress() {
		assertEquals("address2", customer2.getAddress());
	}

	public void testGetZipCode() {
		assertEquals("222-2222", customer2.getZipCode());
	}

	public void testSetName() {
		customer1.setName("name1");
		assertEquals("name1", customer1.getName());
	}

	public void testSetAddress() {
		customer1.setAddress("address1");
		assertEquals("address1", customer1.getAddress());
	}

	public void testSetZipCode() {
		customer1.setZipCode("111-1111");
		assertEquals("111-1111", customer1.getZipCode());
	}

}
```

ソースコード 3: ItemTest.java

```
package test_for_thesis;

import StockManagement.Item;
import junit.framework.TestCase;

public class ItemTest extends TestCase {

	private Item item1;
	private Item item2;

	protected void setUp() throws Exception {
		super.setUp();

		//コンストラクタのテスト
		item1 = new Item("品物 A",30);
		item2 = new Item("品物 B", 40);
	}

	public void testGetName() {
		assertEquals("品物 A", item1.getName());
	}

	public void testGetAmount() {
		assertEquals(30, item1.getAmount());
	}

	public void testSetAmount() {
		item2.setAmount(50);
		assertEquals(50, item2.getAmount());
	}

}
```

ソースコード 4: ReceiptionDeskItemTest.java

```
package test_for_thesis;

import StockManagement.ReceiptionDesk;
import junit.framework.TestCase;

public class ReceiptionDeskTest extends TestCase {

	private ReceiptionDesk re;

	protected void setUp() throws Exception {
		super.setUp();

		//コンストラクタ
		re = new ReceiptionDesk();
	}

}
```

ソースコード 5: RequestTest.java

```
package test_for_thesis;

import java.util.Date;

import StockManagement.Customer;
import StockManagement.Request;
import StockManagement.StockState;
import junit.framework.TestCase;

public class RequestTest extends TestCase {

	private Customer customer1;
	private Request request1;
	private Request request2;
	private Request request3;

	protected void setUp() throws Exception {
		super.setUp();

		//コンストラクタのテスト
		customer1 = new Customer("Handai␣Taro", "Osaka", "123-4567"
		    );
		request1 = new Request("Item1", 10, customer1);
		request2 = new Request("Item2", 10, customer1, new Date());
		request3 = new Request("Item3", 10, customer1, new Date(),
		    StockState.SATISFYED);
	}

	public void testGetCustomer() {
		assertEquals(customer1, request2.getCustomer());
	}

	public void testGetSetAmount() {
		request1.setAmount(20);
		assertEquals(20, request1.getAmount());
	}

	public void testGetRequestState() {
		assertEquals(StockState.SATISFYED, request3.getRequestState
		    ());
	}

	public void testGetName() {
		assertEquals("Item1", request1.getName());
	}

}
```

ソースコード 6: StockStateTest.java

```
package test_for_thesis;

import StockManagement.StockState;
import junit.framework.TestCase;

public class StockStateTest extends TestCase {

	/* StockState は本来インスタンス化されないクラスだが,
	 * カバレッジレポートに登場させるために記述 */

	private StockState ss;

	public void testStockState() {
		ss = new StockState();
	}

}
```

ソースコード 7: StorageTest.java

```
package test_for_thesis;

import StockManagement.Storage;
import junit.framework.TestCase;

public class StorageTest extends TestCase {

	private Storage storage;

	public void testStorage() {
		storage = new Storage();
	}

	/* 対応する単純なsetter がないため,
	 * getItemList, getContainerItemList は記述できず */
}
```