Vol.2014-SE-186 No.19
2014/11/14

オープンソースソフトウェアを用いたシステムの開発支援ツール

青山 裕介[1,a)]　黒岩 丈瑠[1,2]　久代 紀之[1]

概要：オープンソースソフトウェアを用いたソフトウェア開発では，導入するオープンソースソフトウェアと独自開発部分のソフトウェアとの間に問題が発生しないかを調査するためにオープンソースソフトウェアのソースコードを読み解かなければならないことがある．しかし，開発者にとってそのオープンソースソフトウェアがどのように実装されているのかは未知であることがあり，そのような未知のソースコードの処理の構造を理解することは容易ではない．本研究では，ソースコード中のメソッドの構造や，クラス間の参照関係を可視化することで，ソースコードの処理の構造の理解や影響関係の発見を支援するツールの開発を行った．

キーワード：プログラム構造可視化，システム開発支援ツール

Support Tool for Software Design on Open Source Software

Yusuke Aoyama[1,a)]　Takeru Kuroiwa[1,2]　Noriyuki Kushiro[1]

Abstract: In the case of software development on open source software, developers should read source code of the open source software and understand structure of the open source software in order to check whether there isn't any defect between the open source software and own developed software. However, it is not easy work for developers because the developers don't know how the open source software is implemented. This paper describes a tool for support of understanding structure of source code and relationship of dependency among open source software and own developed software by visualizing structure of a method in source code and relationship among classes which reference other classes or are referenced by other classes.

Keywords: visualization of program structure

1. はじめに

オープンソースソフトウェアを用いた開発では，導入するオープンソースソフトウェアと独自開発部分のソフトウェアとの間に予期せぬ参照・被参照の影響関係が発生し，不具合が発生する可能性がある．開発者はこの問題が発生しないことを確認するために，オープンソースソフトウェアが独自開発部分と関わる処理について，オープンソースソフトウェアのソースコード中での処理の構造を読み解かなければならない事があるが，開発者が開発に関わっていないソースコードが行う処理の構造について理解することは容易な作業ではない．

本研究ではこのような開発者が開発に関わっていないソースコードの処理に対し，ソースコード中に記述されている処理フローと，各処理を定義しているモジュール同士の影響関係を可視化することで，導入するオープンソースソフトウェアのモジュールの起動順序や分岐などの処理の構造の理解と，導入するオープンソースソフトウェアによって影響を受けるモジュールの発見を支援するツールの

[1] 九州工業大学大学院情報工学研究院情報創成工学科
Department of Creative Informatics, Kyushu Institute of Technology
[2] 三菱電機（株）住環境研究開発センター
Mitsubishi Electric Co.Ltd Living Environment Systems Laboratory
[a)] aoyamak@minnie.ai.kyutech.ac.jp

開発を行った．

1.1 影響関係の抽出について

本研究において，モジュールとはソースコード上でのまとまりのある機能のことを表し，モジュールは，パッケージやクラスの粒度で実現される．ツールではモジュールとしてのクラスを想定した実装を行った．

ソフトウェアは連携した複数のモジュールによって構成される．モジュール間の連携はソースコード上では変数・メソッドの参照によって実現される．あるモジュールが他のモジュールの変数・メソッドを参照した時，このモジュールの振る舞いは参照したモジュールの変数・メソッドの内容に影響を受ける．逆に，あるモジュールが外部から参照されている変数・メソッドは，他のモジュールから参照された際に，そのモジュールの振る舞いに影響を与えると考えられる．したがって，本研究ではこの参照・被参照の関係を各モジュール間の影響関係と捉え，ソースコード中で出現する変数・メソッドのうち，他のモジュールから影響を受けていると考えられるものとしてモジュール外部で定義されている変数・メソッドに，他のモジュールに影響を与えうると考えられるものとしてモジュール外部で参照されているモジュール内部の変数・メソッドに対応した．本研究ではそれら抽出した影響関係をプログラムの処理フローを可視化した図上で表示する．

2. 作成したツールについて

2.1 ツール概要

本ツールは Eclipse Foundation より提供されている IDE（Integrated Development Environment）である Eclipse[1] 上の plugin として動作する．

Eclipse 上のソースファイル一覧を表示する機能を持つ Package Explorer や，ソースファイルのアウトラインを表示する機能を持つ Outline 上から可視化を行いたいメソッドを選択し，選択したメソッド上で右クリックを行うことでコンテキストメニューに選択メソッドを可視化するための項目 “visualize this method” が表示される（**図 1**）．この項目を選択することによって，該当メソッドが本ツールにより可視化される．

表示された図形のうち 2.4.5 項で述べるような矩形図形はモジュールを表し，これは 2.3 節で述べるように，同一クラスに対する連続する複数回のメンバ呼び出しをまとめたものである．図上でこのモジュールに相当する図形をクリックすると，**図 23** のように選択したモジュール（図中緑色）を中心として，影響関係にあるモジュールが影響を受けているモジュール（図中赤色），影響を与えているモジュール（図中青色）相互に影響を与えているモジュール（図中橙色）でハイライトされ，また影響関係にモジュール間に直線が引かれる．これにより，導入するオープンソースソフトウェアと影響関係にあるモジュールの発見を支援することができる．

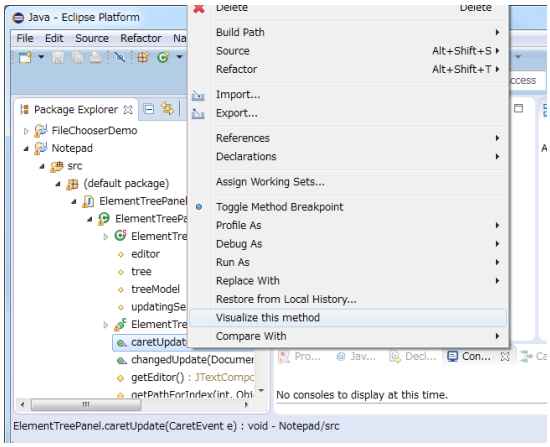

図 1 “visualize this method” 表示例
Fig. 1 Example of diplaying “visualize this method”

さらに，2.4 節で述べるように，条件分岐などを含むプログラムの処理フローをそれぞれ図を用いて表すことで，導入するオープンソースソフトウェアの各モジュールの振る舞いの理解や，起動順序の整合性の確認を支援することができる．

2.2 対応言語

本ツールを適用しようとしている製品の開発言語は Java 言語であったため，対応言語は Java SE 7 Edition の Java 言語 [2] とした．

2.3 可視化内容

本研究ではの処理フローと，依存関係の可視化を行った．プログラムの処理フローについては，どのクラスがどのような順序で呼び出されているのかということを可視化するために，それら順序が記述されているメソッド中の構造を可視化した．可視化の対象とする構造は，if 文, while 文, do 文, for 文, 拡張 for 文, switch 文, try 文の制御文，変数呼び出しやメソッド呼び出しなどクラスのメンバ呼び出しである．2.4 節で述べるような図形をそれぞれの構造に割り当て，表示されるようにした．また，影響関係については，影響関係を持つもの同士を直線で接続し，図上で関係があることを示すことによって実現した．

プログラムのクラス間の関係を可視化する際に，連続して同一クラス・及びそのメンバが出現し，図全体が複雑化すること避けるために，プログラム中に表れる連続した同一クラスに対するメンバ呼び出しについては，それら連続する呼出しをまとめてモジュールという構造で表現した．

2.4 処理フローの図式表現

本研究では，プログラム中に出現する次の項目についての可視化を行った．

```
public class IfDemo {
    public void demo(boolean flag){
        if(flag)
            System.out.println("if");
        else
            System.out.println("else");
    }
}
```

図 2　if 文を可視化した例（ソースコード）
Fig. 2　Example of visualization for “if statement” (source code)

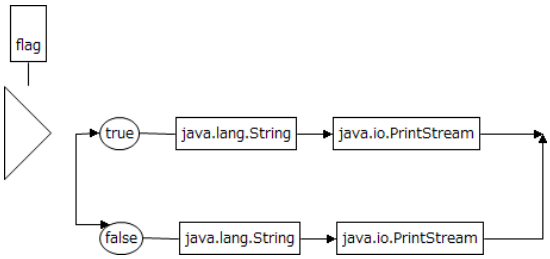

図 3　if 文を可視化した例
Fig. 3　Example of visualization for “if statement”

```
public class SwitchDemo {
    public void demo(int num){
        switch(num){
        case 0:
            System.out.println("case");
            break;
        case 1:
            System.out.println("case");
            break;
        default:
            System.out.println("default");
        }
    }
}
```

図 4　switch 文を可視化した例（ソースコード）
Fig. 4　Example of visualization for “switch statement” (source code)

2.4.1　分岐

プログラム言語 Java 中に出現する条件分岐の構文には if 文と switch 文がある．それぞれについて**図 3**, **図 5** のような図を用いて可視化を行った．

2.4.2　繰返し

繰返しの構文には for 文，拡張 for 文，while 文，do 文がある．それぞれについて**図 7**, **図 9**, **図 11**, **図 13** の図を用いて可視化を行った．

2.4.3　例外

例外処理の構文としては try 文を用いることができる．この構文について，**図 15** のような図を用いて可視化を行った．

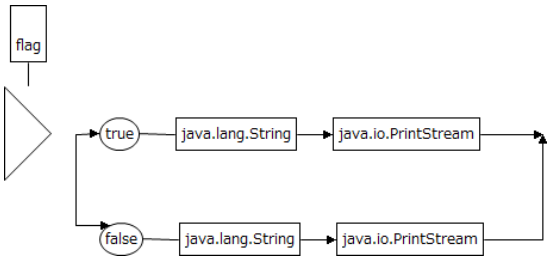

図 5　switch 文を可視化した例
Fig. 5　Example of visualization for “switch statement”

```
package visualize;

public class ForDemo {
    public void demo(int num){
        for(int i = 0; i < num; i++)
            System.out.println("demo");
    }
}
```

図 6　for 文を可視化した例（ソースコード）
Fig. 6　Example of visualization for “for statement” (source code)

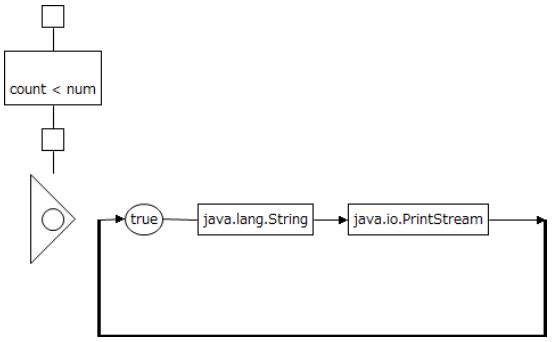

図 7　for 文を可視化した例
Fig. 7　Example of visualization for “for statement”

```
public class EnhancedForDemo {
    public void demo(int[] num){
        for(int n : num)
            System.out.println("demo" + n);
    }
}
```

図 8　拡張 for 文を可視化した例（ソースコード）
Fig. 8　Example of visualization for “enhanced for statement” (source code)

2.4.4　block 文

複数の文をまとめたり，変数スコープを定義したりすることの出来る構文として block 文がある．これは分岐や繰返しの構文の body 部分に複数の文を含めたい場合に用いることができる．しかし，分岐，繰返しの body 部分に文が複数あることは項 2.4.1, 2.4.2, 2.4.3 で述べた図によって示せるので，これら構文中以外で出現するときのみ block

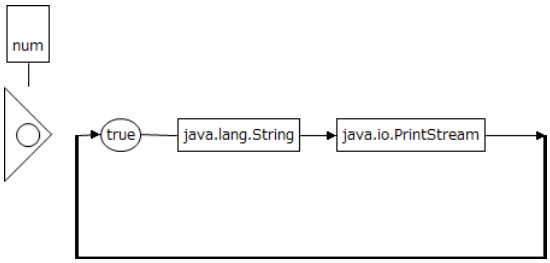

図 9　拡張 for 文を可視化した例
Fig. 9　Example of visualization for “enhanced for statement”

```
public class WhileDemo {
    public void demo(int num){
        while(num++ > 0)
            System.out.println("demo");
    }
}
```

図 10　while 文を可視化した例（ソースコード）
Fig. 10　Example of visualization for “while statement” (source code)

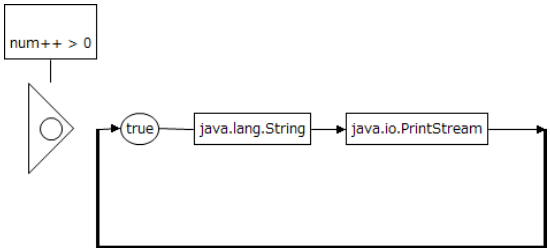

図 11　while 文を可視化した例
Fig. 11　Example of visualization for “while statement”

```
public class DoDemo {
    public void demo(int num){
        int count = 0;
        do {
            System.out.println("demo");
        } while(count++ < num);
    }
}
```

図 12　do 文を可視化した例（ソースコード）
Fig. 12　Example of visualization for “do statement” (source code)

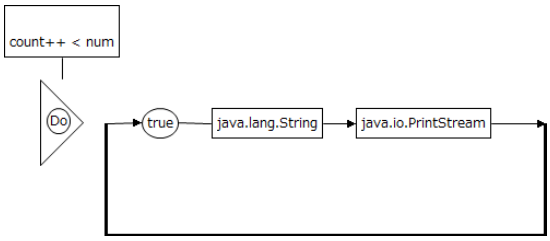

図 13　do 文を可視化した例
Fig. 13　Example of visualization for “do statement”

```
package visualize;

public class TryDemo {
    public void demo(int left, int right){
        try{
            System.out.print(left / right);
        } catch(ArithmeticException e){
            e.printStackTrace();
        } finally{
            System.out.println();
        }
    }
}
```

図 14　try 文を可視化した例（ソースコード）
Fig. 14　Example of visualization for “try statement” (source code)

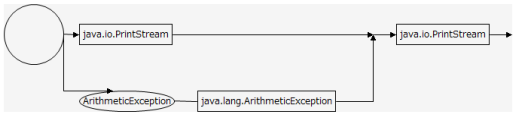

図 15　try 文を可視化した例
Fig. 15　Example of visualization for “try statement”

```
public class BlockDemo {
    public void demo(){
        {
            System.out.println("demo");
        }
    }
}
```

図 16　block 文を可視化した例（ソースコード）
Fig. 16　Example of visualization for “block statement” (source code)

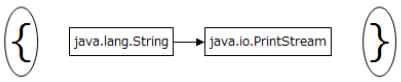

図 17　block 文を可視化した例
Fig. 17　Example of visualization for “block statement”

文として表現することとした（図 17）.

2.4.5　モジュール

プログラミング言語 Java で定義されている構文以外に，モジュールという構造を定義し，図を用いて表した．本研究においてモジュールとは，連続するメンバ呼び出しをまとめ，クラス名のラベルを付けたものとする．これは，2.3 節で述べたように，同一モジュールに対する連続する複数回のメンバ呼び出しによって，図が複雑化することを避けるための表記上の工夫であり，クラスと同等のものを表現している．これを図上に配置することによりどのよう

```
public class ModuleDemo {
    public void demo(){
        System.out.println("demo");
    }
}
```

図 18　モジュールを可視化した例（ソースコード）
Fig. 18　Example of visualization for “module” (source code)

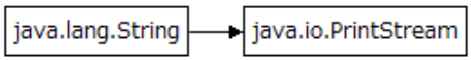

図 19　モジュールを可視化した例
Fig. 19　Example of visualization for “module”

```
public class ABC {
        BCD bcd;
        public ABC(){
        }

        public int methodABC(){
                BCD bcd;
                bcd = new BCD();
                bcd.methodBCD();
                bcd.bb = 3;
                return bcd.bb;
        };
}
```

図 20　影響関係を可視化した例（ソースコード）
Fig. 20　Example of visualization for “dependency relation” (source code)

な順序でモジュール呼び出しが行われているのか，どのようなモジュールと影響関係があるのかを示すことが出来る（図 **19**）．

2.4.6　影響関係

モジュール間の影響関係はモジュールの図を相互に結ぶ直線で示されてる．これは影響関係を調査したいモジュールの図を選択することで，例えば図 **20**, 図 **21**, 図 **22** からなるプロジェクトの場合，図 22 の methodEFG() は図 23 のように表現される．

2.5　可視化方法

2.1 節で説明した Package Explorer や Outline 上で可視化したいメソッドを選択すると，本ツールの入力として該当メソッドを表す IMethod が与えられる．以下ではこの入力以降について説明を行う．

(1) 影響関係の抽出
与えられた該当メソッドから Eclipse JDT[3] を用いて AST（Abstract Syntax Tree）[4] を生成し，生成された AST を元に，1.1 項で述べたような影響関係の抽出を行う．

(2) 可視化
与えられたソースコードの構文構造を， 2.4 項で説明した図形に置換し，処理フローに基づいて 2 次元配置を行う．得られた影響関係情報は，影響関係をもつ図形間を結ぶリンクとして実装される．なお，可視化機能の実現には図形編集を行うエディタ作成するためのフレームワークである GEF（Graphical Editing Framework）[5] を用いた．

```
public class BCD {
        public int bb;

        public int methodBCD(){
                EFG efg = new EFG();
                efg.e = 3;
                return new EFG().hashCode();
        }
}
```

図 21　影響関係を可視化した例（ソースコード）
Fig. 21　Example of visualization for “dependency relation” (source code)

```
public class EFG {
        int e;

        public void methodEFG(){
                ABC a = new ABC();
                a.bcd = new BCD();
                a.methodABC();
                a.bcd.methodBCD();
        }
}
```

図 22　影響関係を可視化した例（ソースコード）
Fig. 22　Example of visualization for “dependency relation” (source code)

図 23　影響関係を可視化した例（図 22 の methodEFG()）
Fig. 23　Example of visualization for “dependency relation” (methodEFG() in 図 22)

3.　実験

本ツールによるプログラムの処理フローの理解や影響関係を持つクラスを発見の支援によって，プログラム作成にかかる作業時間が減少することを確認するために実験を行った．

表 1　課題割り当て
Table 1　Assignments

被験者	Notepad	FileChooserDemo
A	ツールあり	ツールなし
B	ツールなし	ツールあり
C	ツールなし	ツールあり

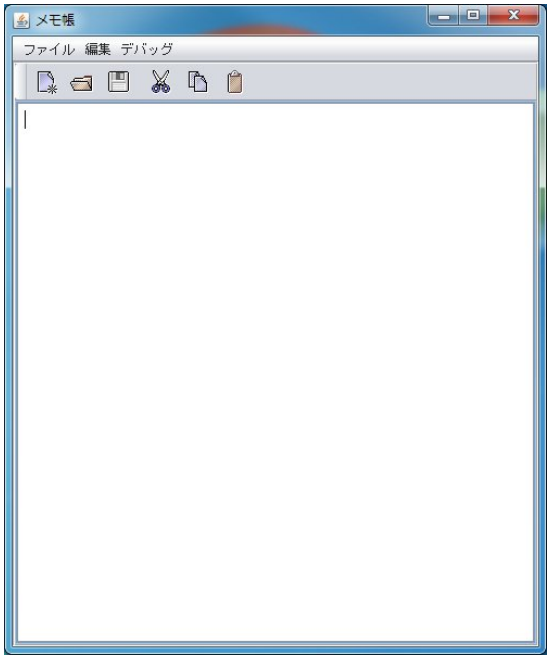

図 24　Notepad 実行例
Fig. 24　Example of execution of Notepad

3.1　実験方法

被験者として，共同開発企業から Java 言語による開発業務を行っている開発者 2 名と，九州工業大学情報工学部 4 年生 1 名をツールを使用する群，ツールをしない群に分け，それぞれの群に既存プログラムに対する機能追加の課題を与えることを，群を交換してそれぞれ 1 度行った（**表 1**）．

機能追加に用いるプログラムには，Oracle より提供されている，Java SE Development Kit 7u65 Demos and Samples[6] 中に含まれる Notepad と FileChooserDemo を用いた．

課題内容は以下である．

- Notepad
  Java SE Development Kit 7u65 Demos and Samples [6] 中では実行すると**図 24** のような小さなメモ帳が開く．メモ帳左上のメニューに有る “ファイル” の中にメモ帳を終了させる項目があり，それを選択することで Notepad を終了することができる．
  課題では Notepad.java から Notepad の終了処理を削除したもので置き換えたものを与えた．課題の作業内容として，Notepad の終了用の処理を追加させた．
- FileChooserDemo
  実行するとファイル，ディレクトリの選択を行うウィンドウが開き，ファイル・ディレクトリを選択すると，その名前を表示することができる．
  課題の作業内容として，表示される名前に続いて選択したものがファイルである場合はそのファイル中の行数を表示するように処理を追加させた．

プログラムの規模は，**表 2** である．

表 2　試験用プログラムの規模
Table 2　A scale of program for test

規模指標	Notepad	FileChooserDemo
ファイル数	2	3
クラス数	15	8
メソッド数	68	34
行数	933	744

表 3　作業時間
Table 3　Time of tasks

被験者	Notepad	FileChooserDemo
A	23 分 17 秒	33 分 18 秒
B	59 分 58 秒	49 分 44 秒
C	48 分 24 秒	66 分 19 秒

3.2　実験結果

課題の作業時間は**表 3** に示す．

課題解答中の作業を観察したところ，ツールあり・なしによらず，プログラムを実行した時に現れる GUI 部品のラベルやメニューの文字列から課題の仕様を満足するための処理の追加箇所を探し，処理を追加するという手順で行われていた．ツールは，処理の追加箇所を探す際に利用されており，ツールの利用される場面については，被験者 C の場合は出現するメソッド全てについてツールを使っており，被験者 A, B の場合は，課題のプログラム中において，描画されるメニューの生成・配置時のアクションの有無・アイコンの有無・配置場所の選定などを含んだメソッド（Notepad の createMenuItem）や，ファイルパスの選択が複数かの判定・ファイルパスがディレクトリかファイルかの判定選択が正しく行われているかの判定などを行うメソッド（FileChooserDemo の getResultString）のようにメソッド中の分岐が入れ子になっている場合や，複数出現している場合に用いられていた．逆に，ほとんど分岐が存在しないものに関してはツールは利用されず，ソースコードを直接読んでいることが観察された．

4.　考察

4.1　実験結果について

表 3 をみると，被験者 A, B どちらもツールありのほうがツールなしの時に比べて作業時間が短くなっているが，被験者 C ついてはツール使用時のほうが作業時間が長くなっている．これについて，被験者 A, B は共同研究企業の技

術者から選出されているため，ツールの利用目的や意義についての考えを共有しているために，試験中にいつツールを利用すればよいのか，どのような問題にツールを使えばよいのかについて理解していたのに対し，被験者 C については本研究のツールの使い方に関する指導のみではどのような問題に適用できるかについて理解することができず，ツールによる支援が十分に得られなかったためであると考えられる．これは試験終了後に，被験者 C に対して，ツールの利用目的を理解していたかどうかを尋ねたところ，

> どの部分でツールを利用すればよいのかわからなかった．

という回答が得られたことや，どのような部分でツールを用いたか，また，ツールを使うとどのようなことがわかったかという質問について，

> メソッドが出現すると全てに用いた．

という回答や，

> ツールを使うとメソッド中でどの部分に処理を追加すればよいのかの見当をつけることができたが，それがソースコード中のどの部分に対応しているのかがわからず，結局ソースコードも全部見る必要があった．

という回答していたことから確認できる．今後の実験では，試験を行う前にツールを用いたプレテストを行い，ツールの利用方法だけでなく利用箇所に関する指導も行って，理解度に差が生じないようにしていく．

また，3.2 項より，被験者 A, B については，ツールはメソッド中の構造が複雑である場合に利用されたことや，実験後にツール使用時の利点を尋ねるアンケートで，被験者 A については，

> 長いメソッドの動きを大まかに把握できるのが良い

被験者 B については，

> 注目したメソッドの流れ，使われるクラスを追えるのは良い

という回答が得られたことから，被験者 A, B の作業時間が短縮されているのは，複雑な構造を持つメソッドのソースコードを読み解く前に予めその構造の概要を知ることで理解が早まったためであると考えられる．

しかし，逆に分岐がほとんど存在しない場合にはツールが利用されていなかったこと，試験後にツールとソースコードとの対応が取れなかったという回答が得られたことを踏まえると，本ツールは単純な構造のメソッドに対しては，ツールによる構造理解の支援後にソースコード中の対応箇所を見つけることと，ソースコードのみから構造を理解することとを比較した際にツール支援の利点が見出されなかったと考えられる．

5. 課題

実験後，被験者の方々にツールの使用感についてのアンケートを行ったところ，以下の様にツールの問題点を指摘された．

(1) 図上で変数名・メソッド名を表示してほしい
(2) 図を複数表示したい
(3) 図からソースコードに飛べる機能がほしい

1 について，本研究ではモジュール間の影響関係の発見を支援するために，2.3 節で述べたように，クラスをモジュールという構造に置き換えて可視化を行っていた．ツール開発時はモジュール間の起動順序を示すことを目標としていたため，図は，2.4.5 項で述べたようにクラス名のみでの表示を行っており，ソースコード中で実際に呼び出されている変数名やメソッド名がわからない．そのため，予め図上で概要を理解しても，再度ソースコードを追う必要が出るために，不便を感じたとのことであった．これに関しては，モジュールの図中にソースコード上での変数名や，メソッド名に関する情報を追加し，ソースコードとの対応を取れるようにすることで改善した．

2 について，ツールは同一ファイルに対してひとつしか開けなかったため，同一クラスのメソッドはひとつしか開けない，という問題があった．これに対し，複数の他のメソッドとの比較を行いたいという要求があった．これは，ツールを開くために用いていたメソッドが，すでに同一ファイルを入力としてツールが起動されているときは複数起動しないという処理を行っていたからであったため，この判定が行われないものと置き換えることで解決できる．

3 については，可視化した図上のモジュールを選択した時に，そのモジュールに関するソースコードを開く機能があれば，該当するクラスをソースファイル群から探す必要が省略でき，操作性が向上するのではないか，という提案であった．本研究で開発したツールは Eclipse 上の他の機能との連携が行われていなかった．これに関しては，外部のエディタを開く処理を追加するなどを行い改善していく．

6. 関連研究

プログラムの理解を支援する関連研究として，[7], [8], [9] がある．[7] は，メソッドが入力データとして使用している変数の情報を木構造で表示するものであり，提示された情報から関係するクラスやメソッドを調査することも出来る．[8] は，ソースコード中のデータフローをメソッド・クラスにまたがってをグラフ化しており，このグラフからメソッド間，及びそれを含むクラス間の関係も知ることが出来る．[9] は，プログラムを実行し，実行中のデータを記録・解析することで，個々のメソッドがどのような入力を必要とし，どのような出力を行うのか，といった事

前・事後の条件を出力する．これらの条件からメソッドの理解のほかテストケースの生成などに利用することができる．ただし，プログラムの実行により動的に不変条件を推論するため，推論する不変条件は実行するテストケースに依存し，誤りが生じることがある．この誤りは，[10], [11] などの工夫によって削減することはできるが，完全ではない．また，推論された不変条件は JML や ESC/Java などのフォーマットで出力することができるが，グラフィカルな可視化機能は持たない．

ソースコードを理解する際には，定義されたクラス・メソッドがどのように振る舞うのか，についても調査する必要が有るため，これらクラス・メソッドが呼び出されているメソッドの中でどのような順序で出現しているのかということを調べる必要がある．しかし，上記研究ではクラス・メソッドがどのような順序関係で用いられるのかという側面では情報を提供していない．したがって， 本研究ではクラス同士がどのように用いられているのかについて示すために，メソッド中で出現するクラスの順序関係に基づいて可視化を行い，プログラムの理解を支援するというアプローチを取った．

参考文献

[1] Eclipse Foundation: Eclipse, https://www.eclipse.org/ide/.
[2] Gosling, J., Joy, B., Steele, G., Bracha, G. and Buckley, A.: *The Java Language Specification, Java SE 7 Edition*, Addison-Wesley Professional, California, USA, 1st edition (2013).
[3] Eclipse Foundation: JDT – Java development tools, http://projects.eclipse.org/projects/eclipse.jdt.
[4] Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers: Principles, Techniques, and Tools (2nd Edition)* (原田賢一 (訳), コンパイラ [第 2 版] –原理・技法・ツール–, サイエンス社, 2009)，chapter 2, pp. 75–76, Addison-Wesley Longman Publishing Co., Inc. (2006).
[5] Eclipse Foundation: GEF – Graphical Editing Framework, https://projects.eclipse.org/projects/tools.gef.
[6] Oracle: Java SE Development Kit 7u65 Demos and Samples, http://www.oracle.com/technetwork/jp/java/javase/downloads/jdk7-downloads-1880260.html.
[7] 鹿島悠， 石尾隆，井上克郎：エイリアス解析を用いたメソッドの入力データの利用法可視化ツール，ソフトウェアエンジニアリングシンポジウム 2012 論文集，Vol. 2012, pp. 1–8 (2012).
[8] 悦田翔悟， 石尾隆，井上克郎：変数間データフローグラフを用いたソースコード間の移動支援，情報処理学会研究報告. ソフトウェア工学研究会報告， Vol. 2011, No. 12, pp. 1–8 (2011).
[9] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S. and Xiao, C.: The Daikon system for dynamic detection of likely invariants, *Science of Computer Programming*, Vol. 69, No. 1–3, pp. 35–45 (2007).
[10] 宮本敬三， 堀直哉，岡野浩三，楠本真二：Daikon 生成表明改善のためのテストケース自動生成手法とその評価実験，コンピュータソフトウェア，Vol. Vo.28, No. No.4, pp. 306–317 (2011).
[11] 小林和貴, 佐々木幸広, 岡野浩三, 楠本真二：PDG と SMT ソルバを利用した表明自動導出手法の提案と評価，電子情報通信学会論文誌 D，Vol. J96-D, No. 11, pp. 2657–2668 (2013).