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

鹿島 悠[1,a)] 石尾 隆[1,b)] 井上 克郎[1,c)]

概要：プログラム理解において，開発者はメソッドの実行中に行われるデータの入出力の調査に多くの時間を費やしている．Java プログラムでは，メソッドを実行するために必要な入力データは，引数やフィールド，クラス変数を介して与えられる．これらの変数や，それを介して参照されるフィールドのうち，実際にメソッドの実行中に使用されるものを把握するには，注目するメソッドから呼び出されるすべてのメソッドを調査する必要がある．そこで本研究では，指定されたメソッドとそこから呼び出されるメソッドを自動的に列挙し，メソッド実行中に使用されうる引数やクラス変数とそれらのフィールドを抽出し可視化する手法を提案する．引数として渡されたオブジェクトは別の変数に代入されて使用される場合もあるが，本研究ではエイリアス解析を行い，開発者が注目するメソッドでの変数名を用いた可視化を行う．提案手法を実装したツールを用いて対照実験を行った結果，ツールを用いた被験者の方が，プログラム理解の課題の解答のための作業時間が減少したことを確認した．

A tool for visualizing the usage of the input data for a method using alias analysis

Yu Kashima[1,a)] Takashi Ishio[1,b)] Katsuro Inoue[1,c)]

Abstract: In program comprehension, developers often spend a lot of time for the investigation of input/output during an execution of a method. In Java program, input for a method includes arguments, fields and class variables. To identify fields and class variables used by a method, developers must invetigate all methods which may be called from the method. In this paper, we propose a tool for visualizing input of a method including fields and class variables by investigating the method and methods which may be called from it. Using alias analysis, this tool shows field access using their access paths from variables of the specified method. We performed a controlled experiment in which participants perform program comprehension tasks. The result shows the time spent for an investigation with our tool is less than the time without the tool.

1. はじめに

開発者はプログラムの保守作業に多くの時間を費やしており，その中でも，多くの時間をプログラム理解に費やしていると言われている [1], [2], [3]．プログラム理解の際に行う作業の 1 つとして，メソッドの実行中に行われるデータの入出力を調査するという作業がある．LaToza ら [4] はこの作業に開発者がしばしば時間をかけていると指摘している．

現在広く用いられているオブジェクト指向プログラミング言語である Java において，あるメソッドを実行するために必要な入力データは，メソッドの引数やクラス変数を介して与えられる．これらの変数には多数のフィールドを持つオブジェクトが代入されている場合もあり，注目するメソッドの実行中に実際に使用されるデータ項目を特定するには，そのメソッドから呼び出されるすべてのメソッドを調査しなければならない．また，クラス変数は任意のメ

[1] 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
[a)] y-kasima@ist.osaka-u.ac.jp
[b)] ishio@ist.osaka-u.ac.jp
[c)] inoue@ist.osaka-u.ac.jp

© 1959 Information Processing Society of Japan

ソッドからアクセス可能であるため，同様の調査が必要である．

そこで本研究では，開発者に指定されたメソッドのデータフローを自動的に探索し，メソッドの入力データとして利用される引数やクラス変数とそのフィールドを抽出し，木構造で可視化する手法を提案する．さらに，それらのデータを直接利用する命令のあるソースコード上の位置と命令の種類も併せて可視化し，開発者に提示してプログラム理解支援を行う．

提案手法では，指定されたメソッドから呼び出し関係を通じて到達可能なすべてのメソッドを対象としてエイリアス解析を実行し，メソッドの実行中にアクセスされるすべてのフィールドが，指定されたメソッドで利用可能な変数を基点とした場合に，どのデータに該当するのかを特定する．たとえば，開発者が注目しているメソッドで変数 x として参照されているオブジェクトが，呼び出し先メソッドで変数 y として参照され，そのフィールド f が使用されたとき，本手法では注目しているメソッドでの変数名を使用し，x.f が使用されると表示する．

本研究では，プログラム解析部と解析結果表示部を別々に実装し，解析結果表示部は Eclipse プラグインとして実装した．このプラグインは，起動時に解析結果を読み込み，Java エディタ上でクリックされたメソッドに対して解析結果を表示する．

ツールによるプログラム理解作業への影響を評価するために，8 名の被験者に 2 つのプログラムを理解する作業を行なってもらう対照実験を実施した．この実験では被験者に指定された機能が実装されているソースコードの場所を発見するという課題を解いてもらい，解答時間や正答率を測定した．実験の結果，ツール有りの被験者の方が，ツール無しの被験者よりも，課題を解答するまでの作業時間が減少する傾向が見られた．また，ツールは，メソッド内で使用される変数の確認や，メソッドの処理と関連する変数の検索などに使われていた．ただし，正答率についてはツールによる改善は得られなかった．

本論文の主要な貢献は以下の 3 つである．

- メソッドの入力データを列挙する方法を提案した．既存手法が入力データのソースコード上の位置を可視化する手法であるのに対し，本手法は入力データの一覧を可視化している．
- 木構造を用いたデータ構造の可視化方法を提案した．このとき，フィールド名については注目しているメソッドからみた参照関係で示している．たとえば，注目しているメソッドで変数 x として参照されているオブジェクトが，他のメソッドの変数 y に代入されてフィールド f を使用されたとき，x.f が使用されたと表示する．
- Eclipse プラグインとして提案手法を実装し，対照実験を行い効果を確認した．

以降，2 章では研究の動機を提示し，3 章では提案手法を述べ，4 章では提案手法の実装を述べ，5 章では評価実験について述べる．6 章では関連研究について述べ，最後に 7 章ではまとめと今後の課題を述べる．

2. 研究の動機

LaToza ら [4] は，プログラム理解の作業で，メソッドの実行中に行われるデータの入出力を開発者はしばしば時間をかけて調査しなければならないことを指摘している．このうち，メソッドにデータが入力される場合は基本的に引数を介してデータが与えられる．そのため，入力されるデータを把握するには引数を見ればよく，一見簡単な作業に思われる．

しかし，引数には，単純な数値データだけではなく，多数のフィールドを持つオブジェクトが与えられる場合がある．このときメソッドの実行中にオブジェクトが持つすべてのフィールドを利用するわけではない．例として，**図 1** に教育プログラム ITSpiral[5] で提供されている和歌山大学教務システムの一部であるソースコード片を示す．このメソッド validateForm は，ウェブブラウザから入力されたデータ form を検証するメソッドである．しかし，このメソッドでは form の持つ 16 個のフィールドのうち，form.Id，form.usedId，そして form.userKubun の 3 つしか利用していない（それぞれ，4 行目の form.getId()，7 行目の form.getUsedID()，8 行目の form.getUserKubun() の呼び出し先で読み込みを行っている）．利用されるデータを正確に把握するには，引数やクラス変数のデータフローを探索する必要があるが，これはコストの大きい作業である．

また，メソッドには引数だけではなく，クラス変数を介してデータが渡される場合もある．クラス変数に対してはソースコード中の任意の位置からアクセスできるため，注目しているメソッドではクラス変数に直接アクセスしていなくても，そのメソッドから呼び出されるメソッドが重要なクラス変数にアクセスしている場合がある．たとえば，図 1 の validateForm では，7 行目の findByUserID というメソッドが，いくつかのメソッド呼び出しを介して HibernateUtil.SESSION というデータベースのセッションを管理するクラス変数にアクセスしている．あるメソッドの実行中にアクセスされるクラス変数を把握するには，注目するメソッドから呼び出されるメソッドすべてを読解しなければならず，大きなコストがかかる．

以上の問題に対し，プログラムスライシング [6], [7], [8], [9], [10], [11], [12] やデータフローの可視化 [13] により，ソースコードの読み手がメソッドの入力を調査するコストを削減する方法が提案されている．しかし，これらの手法は読むべきソースコードを特定する手法であり，注目するメソッドの入力データは読み手が手作業で特定しなければ

© 1959 Information Processing Society of Japan

```
 1: protected boolean validateForm(
 2:   final HttpServletRequest request,
 3:   final UserForm form) throws ServiceException {
 4:   if (form.getId() == 0) {
 5:     User user = ServiceFacotry
 6:       .getService(IUserService.class)
 7:       .findByUserID(form.getUserId(),
 8:         UserKubun.valueOf(form.getUserKubun()));
 9:     if (user != null) {
10:       addError(request, "errors.ucm02.exist.user");
11:       return false;
12:     }
13:   }
14:   return true;
15: }
```

図 1 メソッド validateForm（和歌山大学教務システム中のソースコード）

ならないのが現状である．

3. 提案手法

本研究では，Java プログラムに対して自動的に探索を行い，指定されたメソッドの入力として使われた変数の利用情報を抽出し，一覧表示することで，プログラム理解支援を行う．変数の利用情報とは，利用された変数，変数を利用するメソッド，変数を利用した命令がある位置のソースコード上の行番号，命令の読み書きの種別，の 4 要素の組であり，以降，(v:変数, m:メソッド, l:行番号, RW:読み書きの種別) と記述する．

本論文において変数とは，メソッドのローカル変数とクラス変数，そしてレシーバーオブジェクトを表す this と，それらから辿ることのできるフィールドを指す．たとえば，ローカル変数 v と，v に格納されたオブジェクトのフィールド f1，f1 に格納されたオブジェクトのフィールド f2 が存在するとき，v, v.f1, v.f1.f2 が変数となる．

本研究において，メソッドの入力変数とは，メソッドの実行前に初期化済みの変数のうち，メソッドの実行中にアクセスする変数を指す．メソッド m の入力変数は具体的には以下の 3 つが含まれる．

- m の引数．m がクラスメソッド以外の場合，m のレシーバーオブジェクトも含む．
- m の実行中にアクセスする可能性のあるすべてのクラス変数．
- m の実行中にアクセスされるフィールドのうち，m の実行前に初期化されているもの．

本手法は，指定されたメソッド m に対して，m の入力変数の利用情報の集合 I(m) を抽出し，開発者に可視化する．本手法は，(1) メソッド m に対する I(m) の取得，(2) I(m) の可視化，という 2 つのステップに分かれており，以降それぞれについて詳解する．

3.1 メソッドの入力変数の利用情報の抽出

指定されたメソッド m に対する変数の利用情報の集合 I(m) の取得アルゴリズムを**図 2** に示す．このアルゴリズムは，メソッドの入力に対するアクセス情報を抽出し，I(m) に追加する．図 2 の $\leftarrow$ は変数に対する代入を表す．$\Leftarrow$ は左辺の変数が集合を要素に持つとき，集合への要素の追加を表す．

図 2 の関数 reachables() と isAlias()，そして isContinueSearch() は，アルゴリズムのパラメータとなる関数である．reachables() は引数にメソッドを取り，与えられたメソッドから推移的なものも含め呼び出されうるメソッドの集合を返す．isAlias() は引数に変数 2 つを取り，与えられた変数 2 つが同じポインタを指しうるエイリアス関係であれば真を返す．isContinueSearch() は探索を継続するかを決める関数であり，フィールドの階層構造を基に判断する．

図 2 のアルゴリズムは以下の 4 つのステップで構成されている．

Step1（1-5 行目） まず，I(m) を空集合に初期化し，m の引数の利用情報を I(m) に追加する．このとき，引数の利用情報のうちメソッド，行番号，RW は空とする (アルゴリズムの記述上は記号 “*” で空を示している)．また，m と m からの推移的な呼び出し関係で到達できるメソッドの集合 R(m) を求めておく．

Step2（6-8 行目） R(m) からクラス変数を利用している命令を探し，クラス変数の利用情報を I(m) に追加する．

Step3（9-11 行目） スタック worklist に，Step2 までで I(m) に追加した引数とクラス変数を追加する．

Step4（12-22 行目） worklist から変数を 1 つ取り出す．R(m) に含まれるフィールドを利用する命令のうち，レシーバーオブジェクトと worklist から取り出した変数がエイリアスであるもの (m の実行よりも前に作られたオブジェクトを使用している可能性のあるもの) を列挙する．そして，I(m) にフィールドの利用情報を追加する．最後に，探索を続けるならば v.f を worklist に追加する．

3.2 メソッドの入力変数の利用情報の可視化

本ステップでは，3.1 項で得られたメソッドの入力変数の利用情報を整理し，変数とフィールドの階層構造に合わせた木構造で表示する．この模式図を**図 3(a)** に示す．根として m のメソッド名を表示し，根の子要素として引数，そして I(m) に含まれるクラス変数が出力される．また，メソッドがインスタンスメソッドである（レシーバーオブジェクトが引数に含まれている）とき，this を子要素として加える．

引数については，I(m) に引数を親とするフィールドが含まれていればそれを表示し，そのフィールドに対して行わ

Input: m
Output: $I(m)$

1: $I(m) \leftarrow \emptyset$
2: **for** $p \in parameters(m)$ **do**
3: $\quad I(m) \twoheadleftarrow (p, *, *, *)$
4: **end for**
5: $R(m) \leftarrow \{m\} \cup reachables(m)$
6: **for all** $(v, m', l, RW) \in searchClassVariableUsage(R(m))$ **do**
7: $\quad I(m) \twoheadleftarrow (v, m', l, RW)$
8: **end for**
9: **for all** $\{v | (v, m', l, RW) \in I(m)\}$ **do**
10: $\quad worklist \twoheadleftarrow v$
11: **end for**
12: **while** **not** $isEmpty(worklist)$ **do**
13: $\quad v \leftarrow pop(worklist)$
14: $\quad$ **for all** $(x.f, m', l, RW) \in fieldUsage(R(m))$ **do**
15: $\quad\quad$ **if** $isAlias(v, x)$ **then**
16: $\quad\quad\quad I(m) \twoheadleftarrow (v.f, m', l, RW)$
17: $\quad\quad\quad$ **if** $isContinueSearch(v.f)$ **then**
18: $\quad\quad\quad\quad worklist \twoheadleftarrow (v.f)$
19: $\quad\quad\quad$ **end if**
20: $\quad\quad$ **end if**
21: $\quad$ **end for**
22: **end while**
23: **return** $I(m)$

図 2 I(m) の取得アルゴリズム

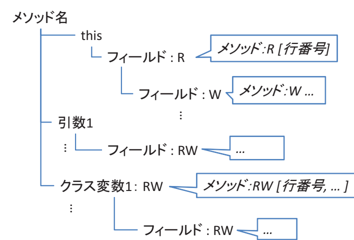
れた読み書きについても表示する（図中では読み込みを R，書き込みを W と表現している）．また，フィールドに付随する情報として，フィールドを読み書きする命令があるメソッドの名前と，命令のある行番号の集合を表示する．さらに，フィールドにオブジェクトが格納される場合は，そのオブジェクトのフィールドへのアクセスが I(m) に含まれていれば，フィールドの子要素として，同様にフィールドの情報を追加していく．

クラス変数については，クラス変数自身に対する読み書き，クラス変数に対する読み書きの命令のあるメソッドの情報を表示する．また，クラス変数の持つフィールドが I(m) に含まれていれば，同様に表示を行う．

4. 実装

実装では，メソッドから I(m) を求める解析部と，I(m) の表示部を，別々のプログラムとして実装した．I(m) の表示部は Eclipse プラグインであり，起動時に解析結果をすべて読み込み，Eclipse の Java ソースコードエディタ上でクリックされたメソッド m に対し，I(m) をツリービュー形式でを可視化する．根には，クリックしたメソッドの戻り値とシグネチャを表示している．さらに，変数を利用する命令のあるメソッドのメソッド名をダブルクリックすれば，そのメソッドがあるクラスにジャンプできる．**図 3(b)** は，図 1 の validateForm の I(m) を可視化した結果の一部である．図中には，引数 form の子要素に id，userId，userKubun の 3 つだけが表示されており，メソッドの実行中にこれら 3 つのフィールドしかアクセスしないことが一目でわかる．また，HibernateUtil.SESSION も表示されており，データベースにアクセスすることが一目で判断できる．

提案手法の中で実装のパラメータとした 3 つの関数については以下の通りに定めた．まず，reachables() で呼び出されるメソッドを取得するときの動的束縛の解決には，呼び出す可能性のあるメソッドを漏れ無く取得できるクラス階層解析 [14] を用いた．isAlias() で用いるエイリアス解析には，Java を対象として実行でき，高速に動作する Yan らの手法 [15] を用いた．最後に，isContinueSearch() で行う探索の打ち切りについては，引数およびクラス変数から 2 段階までのフィールド参照でアクセスできる変数までを解析し，それより先のフィールドは探索しないという実装にした．



(a) I(m) の表示の模式図

```
boolean validateForm(HttpServletRequest request, UserForm form)
  this(UserUpdateAction)
  request(javax/servlet/http/HttpServletRequest)
  form(UserForm)
    id(int) <R>
      UserKeyForm::int getId() <R>[29]
    userId(java/lang/String) <R>
      UserForm::String getUserId() <R>[67]
    userKubun(java/lang/String) <R>
      UserKeyForm::String getUserKubun() <R>[53]
  HibernateUtil.SESSION(java/lang/ThreadLocal<org/hibernate/Session>) <R>
    HibernateUtil::static Session currentSession() <R>[70, 74]
  HibernateUtil.SESSION_FACTORY(org/hibernate/SessionFactory) <R>
    HibernateUtil::static Session currentSession() <R>[72]
  User.UserKubun.GAKUSEI(User.UserKubun) <R>
    UserHibernateDAO::User findByUserId(String userId, User$UserKubun userK
```

(b) I(m) 可視化ビュー

図 3 I(m) の模式図と実装

5. 評価実験

本ツールによりメソッドの理解が促進されることで，プログラム理解の作業の速度と正確さが向上するか確認するため評価実験を行った．本実験では，2 つのアプリケーションを対象にプログラム理解の課題を被験者に解答してもらい，ツールの有無による課題の正答率と解答時間の有意な差があるか調査する．

© 1959 Information Processing Society of Japan

5.1 実験方法

被験者は，大阪大学基礎工学部情報科学科の4年生3人と，大阪大学大学院情報科学研究科博士前期課程1年の学生5人である．全被験者は，研究活動を通じて，JavaとJavaの統合開発環境であるEclipseに習熟している．被験者のプログラミング経験は4年から7年であり，Javaプログラミングの経験は1年から6年である．

実験では，被験者は2つのアプリケーションに対してプログラム理解の作業を行なった．その際，一方のアプリケーションはツール有りで作業し，もう一方のアプリケーションではツール無しで作業した．そして，課題の正答率と解答時間を測定した．

実験で被験者にプログラム理解を行ってもらうアプリケーションには，Java言語で記述され，オープンソースソフトウェアとして配布されている，UML図作成ツールArgoUML 0.34[16]とガントチャート作成ツールGanttProject 2.0.9[17]を選択した．ArgoUMLは.javaファイルが数2289で，約190KLOCである．GanttProjectは.javaファイルが478ファイルで，約44KLOCである．

各アプリケーションの課題は以下の通りである．

ArgoUML もし，ArgoUMLのスタートアップ時に空のクラス図ではなく，空のシーケンス図を出すよう改造するとしたら，どのクラスを変更する必要があるか，クラス名を答えよ．

GanttProject GanttProjectは先行タスクの機能をサポートしている．先行タスクのタスク期間が変化したとき，先行タスクに依存しているタスクのタスク期間を更新するのは，どのコードか，メソッド名と行番号を答えよ．

作業の実施手順は以下の通りである．

(1) 被験者のEclipseに関する知識を平均化するため，Eclipseの代表的な5つの機能についてのレクチャーを行う．本実験でEclipseの代表的な5つの機能としたのは，File Search，Java Search，Open Call Hierarchy，Open Type Hierarchy，Open Declarationである．
(2) 本ツールについてのレクチャーを行う．
(3) 1つ目のアプリケーションのユースケースを被験者に見せる．
(4) 1つ目のアプリケーションに対する実験を行う．
(5) 2つ目のアプリケーションのユースケースを被験者に見せる．
(6) 2つ目のアプリケーションに対する実験を行う．

被験者に示したArgoUMLとGanttProjectのユースケースは以下の通りである．

ArgoUML ArgoUMLを起動し，空のクラス図が現れることを確認し，クラス図中に空のクラスを2つ配置する．

GanttProject GanttProjectを起動し，新規タスクを2つ作り，新規タスクの1つをもう一方のタスクの先行タスクに設定する．そして，先行タスクの終了日を元の終了日より後ろにずらすと，先行タスクに依存しているタスクの開始日も自動的に後ろにずれることを確認する．

1アプリケーションの課題の制限時間は45分である．制限時間に達した時点で作業は終了とするが，被験者は課題が解答できるならば解答する．このとき，課題の解答にかかった時間は制限時間の45分とする．

課題に対する学習効果などを考慮し，被験者に対するアプリケーションの割り当ては**表1**のように行った．表中の（有）と（無）はツールの有無を示している．また，実験は被験者1人ずつ個別に行う．

表 1 課題の割り当て

	1回目	2回目
被験者 A,B	ArgoUML（有）	GanttProject(無)
被験者 C,D	GanttProject(無)	ArgoUML（有）
被験者 E,F	ArgoUML（無）	GanttProject(有)
被験者 G,H	GanttProject(有)	ArgoUML（無）

表 2 課題の正答率

	ArgoUML	GanttProject
ツール有り	0.5（4人中2名）	0.25（4人中1名）
ツール無し	0.25（4人中1名）	0.25（4人中1名）

なお，プログラム全体を一切の手がかり無しで探索するには作業時間が足りないと考え，課題に関連のあるクラスとして，以下に示すクラスを提示した．

ArgoUML org.argouml.uml.diagram.static_structure.ClassDiagramGraphModel

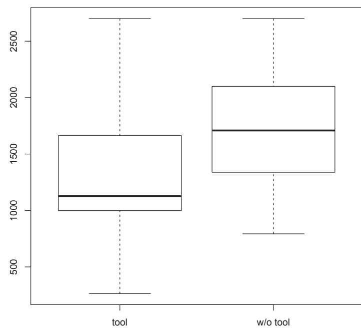
GanttProject net.sourceforge.ganttproject.GanttTask

作業環境には，27インチのディスプレイ（解像度1920 × 1200）1枚とEclipse 3.7を使用した．そして解答の際には，解答用紙にボールペンで記述してもらった．他に使用可能なのは，Windows標準のメモ帳（notepad.exe）とメモ用紙だけである．

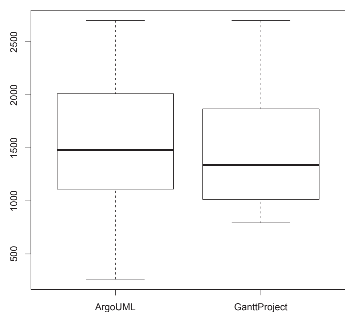
5.2 実験結果

各課題の解答時間の分布について，ツールの有無によらず課題のアプリケーションでグループ化した解答時間の分布を**図 4(b)**に，ツールの有無のみでグループ化した解答時間の分布を**図 4(a)**に示す．ただし，ArgoUMLの課題をツール無しで解いた被験者1名が制限時間に到達した際に解答できなかったため，その値は欠損値とした．

図4(a)より，ツール有りの被験者の方が，ツール無しの被験者よりも，探索を終えたと思うまでの時間が早かったことが分かる．また，図4(b)が示す両アプリケーションの被験者の解答時間より，2つの課題の解答時間に大きな差は無かったことが分かる．よって，本ツールには，作業



(a) ツール有無でグループ化



(b) アプリケーションでグループ化

図 4 課題の解答時間

者の探索時間を減らす効果があったと考えられる．

作業時間の差が有意であるかどうか検証するため，ツール有りの被験者の作業時間とツール無しの被験者の作業時間をウィルコクソンの符号順位和検定により，片側検定を行った．帰無仮説は「ツール有りの被験者の作業時間の母代表値は，ツール無しの被験者の作業時間の母代表値と差はない」，対立仮説は「ツール有りの被験者の作業時間の母代表値は，ツール無しの被験者の作業時間の母代表値よりも小さい」である．検定の結果，p 値=0.07752 が得られ，有意水準 5%では帰無仮説が採択され，有意水準 10%では対立仮説が採択された．

データフローグラフを可視化する悦田ら [13] の手法と比較した場合，彼らの対照実験では有意水準 5%で有意差を確認しており，比較基準の違いはあるが，悦田らの手法の方が効果が明確に表れている．その理由の 1 つとして，課題の性質が考えられる．本実験の課題では，利用されたデータ項目を列挙するだけではなく，それらのデータ項目の使われ方も確認する必要があった．悦田らのツールは，データ項目の列挙は手作業であるが，データ項目の使われ方の調査には本ツールよりも有効に働いたと考えられる．

また，正答率について，課題とツールの有無でグループ化した結果を**表 2** に示す．表 2 の通り，ツールの有無に関わらず正答率は低く，多くの被験者は正答できなかった．そのため，本ツールによりプログラム理解支援で正しい解を得るための支援はできなかったと考えられる．この理由としては，課題が難しすぎた，プログラムに関する予備知識が無かったため変数名からその変数の役割を推測するのが困難であったということが考えられる．

課題の実行中，本ツールは，変数の使われ方の確認，処理と関連がある名前の変数の検索，アクセスされるフィールドの検索，処理と関連があるメソッドの閲覧に使われていた．また，一部の被験者は，フィールドにアクセスしているメソッドがあるクラスへジャンプする機能を好んでいた．逆にジャンプ機能で，直接アクセス箇所に飛べないことを不満としていた被験者もいた．

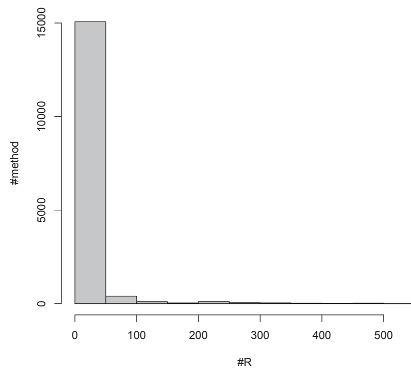
本ツールでメソッドの入力一覧を出した際，ツール上に表示される要素の数が極端に多いことが何度かあったため，これについて追加の調査を行った．**図 5(a)** と**図 5(b)** は，それぞれ ArgoUML と GanttProject に対する表示結果のうち，1 メソッド毎の，読込のみ（ツール上では R）と表示されたフィールドまたはクラス変数の数についてのヒストグラムである．ヒストグラムの横軸は R と表示された要素の数で，縦軸はメソッド数である．図より，殆どのメソッドでは僅かなフィールドとクラス変数しか表示されていないが，一部のメソッドでは，50 個や 100 個以上の要素が表示されてしまうことが分かる．この傾向は，書込のみ（ツール上では W）と表示された要素や読み書き両方（ツール上では RW）と表示された要素でも変わらなかった．表示される要素数がこのように多い場合，被験者がツール出力を閲覧し，有用な情報を得るのは困難だったと考えられる．実験でも，被験者がツールの出力の読解を諦める様子が観測された．

実験結果と著者らの経験により，本ツールは次のような場合に有効であると考えられる．

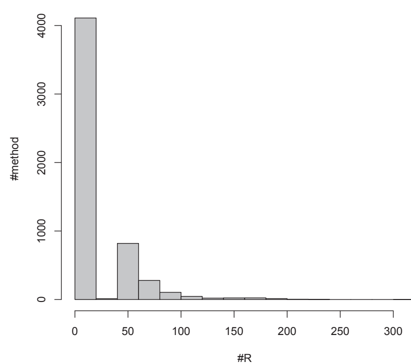
- 重要なグローバル変数が事前に分かっている場合．図 1 の例では，図 3(b) より，HibernateUtil.SESSION を利用していることがわかり，データベースにアクセスしていると判断できる．
- 利用される変数を把握することが重要な場合．図 1 の例では，図 3(b) より，引数 form の持つフィールドのうち，メソッドの実行中に利用されるフィールドが 3 つだけだと分かれば，それ以外のフィールドは検証していないと判断できる．

一方で，変数の使われ方が重要な場合や，多数の変数が利用される場合には本ツールはあまり有効でないと考えられる．変数の使われ方が重要な場合の対策としては，本ツールのジャンプ機能を拡充すること，メソッドのコールグラフと合わせて表示することで，データフローの探索の支援も行うことが考えられる．多数の変数が利用される場合には，変数のグループ化して表示するということが考えられる．たとえば，データベースに関するグローバル変数

© 1959 Information Processing Society of Japan



(a) ArgoUML



(b) GanttProject

図 5 読込のみと表示されたフィールドとクラス変数の数の分布

の一群をまとめて,「データベース」という項目で表示するということを考えている.

5.3 妥当性への脅威

課題の一般性 読解の対象としたのは,実際に配布されているオープンソースソフトウェアであり,問題は機能の実装箇所を探す,いわゆるフィーチャーロケーションまたはコンセプトロケーション [18], [19], [20] である.これは現実のプログラム理解でも行われる作業であり,一般的な作業であると考えている.

被験者に対する説明の差 本実験は,被験者ごとに個別に実験を行なっており,Eclipse や本ツールの説明に対して差が生まれる可能性がある.しかし,説明の際には,同じ原稿を読み上げ,同じテキストを用いて説明を行ったので,この要素は実験結果に影響を与えていないと考えられる.

被験者の技量 参加者は学生であり,課題・研究などで Java プログラミングと Eclipse の使用方法に習熟しているが,企業の熟練労働者と比べると未熟である.そのため,企業の熟練労働者を被験者にした場合は異なる実験結果となりうる.

実験対象のプログラムに対する知識 被験者はみな実験対象のプログラムである ArgoUML と GanttProject に対する保守を行ったことが無く,深い知識は持っていなかった.実際には読解の対象となるソフトウェアに対する知識を持った状態でプログラム理解を行う場合もあり,その場合本実験の結果が完全に当てはまるとは言えない.

6. 関連研究

6.1 プログラム理解に関する関連研究

本研究のプログラム理解の側面で関連する研究として,プログラムスライシングが挙げられる.プログラムスライシングは,ある変数や文を対象に,その変数に依存する文を抽出する手法である [6], [7], [8], [9], [10], [11], [12].プログラムスライシングを用いることで,たとえば,ある引数やフィールドに関係する文だけを抽出することができ,変数が読み込まれる箇所,書きこまれる箇所の特定に有用であると考えられる.

また,データフローを可視化し,プログラム理解を支援手法する手法について,悦田ら [13] は,データフローグラフの可視化によるコードナビゲーションツールを提案した.このツールはデータフローをグラフとしてみることにより,複数のコード片を対象に横断的にデータフローを調査する際有効に働く.これはメソッドの引数が持つフィールドのうち,使用されるフィールドの調査を行う際や,メソッドに入力されるデータの使用法を調査する際に役立つと考えられる.

両研究に対して本研究では,最終的な出力がデータの一覧となっている.そのため,プログラムスライシングの出力である文の集合よりも出力のサイズは小さい.また,悦田らの手法によるデータフローを可視化した場合と異なり,メソッドの入力を把握する際にデータフローをツールの使用者が自ら探索する必要は無い.

6.2 解析手法に関する関連研究

動的束縛の解決には,実装で用いたクラス階層解析 [14] の他に,Variable Type Analysis[21],Rapid Type Analysis[22] 等が提案されている.より正確な動的束縛の解決方法を用いれば,実際にはアクセスし得ないクラス変数やフィールドを,I(m) から取り除くことができると考えられる.

しかし,クラス階層解析は,リフレクションを考慮しない限りは,動的束縛により呼び出し得るメソッド全てを網羅することができるという利点がある.これにより,アクセスする可能性のある変数が I(m) から漏れることは無くなるため,実装では本手法を用いた.ただし,I(m) に実際にはアクセスし得ない変数が含まれる可能性は存在する.

本研究で解析の際に用いたエイリアス解析,またエイリ

© 1959 Information Processing Society of Japan

アス解析の一種であるポインタ解析には，様々な手法が提案されている [15], [23], [24], [25]．こちらもより正確な手法を用いれば，正確にエイリアスを特定することができ，実際にはメソッドの入力でないデータを I(m) の解析結果から除去できる．しかし，正確な手法を用いれば，時間的・空間的コストは増大し実用的なアプリケーションのサイズでは，エイリアス解析を行うのが難しくなる．そのため，実装では高速な解析が可能な Yan らの手法 [15] を用いた．もし，より正確な解析を用いるのであれば，探索するメソッドのコールツリーの深さを限定するなど，探索空間を狭める工夫が必要になると考えられる．

7. まとめと今後の課題

本研究では Java プログラムと指定されたメソッドを対象に自動的に探索を行い，メソッドの実行中に使用される引数やクラス変数とそのフィールドを抽出し，木構造で可視化するツールを作成した．

今後の課題としては，多すぎる出力の要約が挙げられる．また，アクセス箇所を単純に表示するのではなく，コールグラフの表示機能と一部統合させて，メソッドの呼び出し関係とメソッドの入力変数の利用情報を同時に閲覧できるようにすることも考えられる．

謝辞 本研究は一部，日本学術振興会科学研究費補助金基盤研究（A）（課題番号:21240002）と日本学術振興会科学研究費補助金若手研究 (A)（課題番号:23680001）の助成を受けた．

参考文献

[1] Corbi, T. A.: Program understanding: challenge for the 1990's, *IBM Systems. Journal.*, Vol. 28, pp. 294–306 (1989).
[2] Fjeldstad, R. K. and Hamlen, W. T.: Application program maintenance study: report to our respondents, *Proceedings of GUIDE 48* (1979).
[3] LaToza, T. D., Venolia, G. and DeLine, R.: Maintaining mental models: a study of developer work habits, *Proceedings of the 28th International Conference on Software Engineering*, pp. 492–501 (2006).
[4] LaToza, T. D. and Myers, B. A.: Developers ask reachability questions, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, Vol. 1, pp. 185–194 (2010).
[5] ITSpiral: `http://it-spiral.ist.osaka-u.ac.jp/`.
[6] Chen, Z. and Xu, B.: Slicing Object-Oriented Java Programs, *ACM SIGPLAN Notices*, Vol. 36, No. 4, pp. 33–40 (2001).
[7] Harman, M. and Danicic, S.: Amorphous Program Slicing, *In proceedings of the 5th International Workshop on Program Comprehension*, pp. 70–79 (1997).
[8] Horwitz, S., Reps, T. and Binkley, D.: Interprocedural slicing using dependence graphs, *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 35–46 (1988).
[9] Liang, D. and Harrold, M. J.: Slicing Objects Using System Dependence Graphs, *Proceedings of the International Conference on Software Maintenance*, pp. 358–367 (1998).
[10] Reps, T., Horwitz, S., Sagiv, M. and Rosay, G.: Speeding up slicing, *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 11–20 (1994).
[11] Sridharan, M., Fink, S. J. and Bodik, R.: Thin slicing, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pp. 112–122 (2007).
[12] Weiser, M.: Program slicing, *Proceedings of the 5th International Conference on Software Engineering*, pp. 439–449 (1981).
[13] 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, Vol.2011-SE-171, No.12，pp. 1–8 (2011).
[14] Dean, J., Grove, D. and Chambers, C.: Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis, *Proceedings of the 9th European Conference on Object-Oriented Programming*, London, UK, UK, Springer-Verlag, pp. 77–101 (1995).
[15] Yan, D., Xu, G. and Rountev, A.: Demand-driven context-sensitive alias analysis for Java, *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 155–165 (2011).
[16] ArgoUML: `http://argouml.tigris.org/`.
[17] GanttProject: `http://www.ganttproject.biz/`.
[18] Eisenbarth, T., Koschke, R. and Simon, D.: Locating features in source code, *Software Engineering, IEEE Transactions on*, Vol. 29, No. 3, pp. 210 – 224 (2003).
[19] Koschke, R. and Quante, J.: On dynamic feature location, *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 86–95 (2005).
[20] Rajlich, V. and Wilde, N.: The role of concepts in program comprehension, *Proceedings of 10th International Workshop on Program Comprehension*, pp. 271 – 278 (2002).
[21] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E. and Godin, C.: Practical virtual method call resolution for Java, *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 264–280 (2000).
[22] Bacon, D. F. and Sweeney, P. F.: Fast static analysis of C++ virtual function calls, *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pp. 324–341 (1996).
[23] Milanova, A.: Light context-sensitive points-to analysis for Java, *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 25–30 (2007).
[24] Shapiro, M. and Horwitz, S.: Fast and accurate flow-insensitive points-to analysis, *Proceedings of the 24th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 1–14 (1997).
[25] Sridharan, M. and Bodík, R.: Refinement-based context-sensitive points-to analysis for Java, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pp. 387–400 (2006).

© 1959 Information Processing Society of Japan