修士学位論文

題目

メソッドの入出力の可視化による
プログラム理解支援ツール

指導教員
井上 克郎 教授

報告者
鹿島 悠

平成 24 年 2 月 7 日

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 ソフトウェア工学講座

平成 23 年度 修士学位論文

メソッドの入出力の可視化によるプログラム理解支援ツール

鹿島 悠

内容梗概

ソフトウェア開発において，開発者は，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．また，その中でも時間をかけて行われているのが，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作業であると言われている．

Java プログラムにおいて，あるメソッドが処理を実行するために必要な入力データは，メソッドの引数と，メソッドを実行するオブジェクトのフィールド，プログラムの任意の位置から参照できるクラス変数という 3 つの方法によって与えられる．これらの引数やフィールド，クラス変数には，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格納されている場合もある．メソッドの入力を特定するには，引数やフィールドに格納されたオブジェクトのうち，それらのどのフィールドにアクセスするかを調査する必要があるが，これにはメソッド実行中に生じるデータフローを追跡する必要があり，調査には大きな手間がかかる．また，クラス変数は，どのメソッドからでもアクセスできるという特徴から，開発者が注目しているメソッドから呼び出されるすべてのメソッドでの利用状況を調査する必要があり，これにも大きな手間がかかる．

そこで本研究では，Java プログラムにおいて，開発者が指定したメソッドに対して自動的にデータフローの探索を行い，メソッドの入力データとしてアクセスされる可能性のある引数やクラス変数のフィールドを抽出する手法を提案する．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．この解析結果を，アクセス命令が存在するソースコード上の位置やアクセスの種類とともに可視化し，開発者に提示するツールを作成した．

ツールの有無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ってもらう対照実験を実施した．その結果，ツール有りの被験者の方がツール無しの被験者よりも，課題を解答するために行った作業時間が減少する傾向が見られた．

主な用語

プログラム理解
プログラム解析
エイリアス解析
メソッドの入出力

目 次

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いると言われ，保守作業の中でも，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 [4, 5, 9]．そのプログラム理解の際に行う作業の一つとして，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 [8] はこの作業に開発者がしばしば時間をかけているという指摘をしている．

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，あるメソッドが処理を実行するために必要な入力データは，メソッドの引数，メソッドを実行できるオブジェクトのフィールド，プログラムの任意の位置から参照できるクラス変数という 3 つの方法で与えられる．

このうち，クラス変数は任意のメソッドからアクセス可能なため，注目するメソッドの実行中にアクセスする可能性のあるクラス変数を把握するには，注目するメソッドの実行中に呼び出される可能性のあるメソッド全てに注意を払わなければならず，大きな手間が必要である．

さらに，引数やフィールド，クラス変数には，単純な数値データだけでなく，多数のフィールドを持つオブジェクトが格納されている場合もあり，メソッドの実行中に全てのフィールドが使用されるわけではない．使用されるフィールドを把握するには，変数のデータフローを追跡する必要があり，大きな手間がかかる．また，フィールドに格納されているのがオブジェクトである場合もあり，このときはそのフィールドのデータフローも追跡しなければ，使用されるデータを把握することはできないため，さらに手間が必要になる．

そこで本研究では，指定されたメソッドを対象に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アクセスされる可能性のある引数やクラス変数のフィールドを抽出する手法を提案する．さらに，アクセス命令が存在するソースコード上のデータがアクセスされるソースコード上の位置とアクセスの種類も併せて可視化し，開発者に提示することで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全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全てのフィールドが，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．例えば，呼び出し元メソッドで変数 x だったものが，呼び出し先メソッドでは変数 y として参照され，そこで y のフィールド f が使用されたとき，本手法では，y.f を使用されると表示するのではなく，呼び出し元メソッドの変数名を使用し，x.f が使用されると表示する．このエイリアス解析には，高速な解析が可能であり，さらに必要な変数間の関係だけを調査することのできる，Yan らの提案した Demand-Driven エイリアス解析 [20] を

利用する．

この提案手法のうち，開発者に解析結果を表示する部分は，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．

ツールの有無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プログラム理解の課題として，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．また，被験者にアンケートを行い，ツールが役に立った場面やツールに対する不満点などを質問した．

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．しかし，正答率についてはツールの有無に関わらず全体的に低いという結果になった．

以降，2 章では Motivating Example を提示し，3 章では Motivating Example と関連する既存研究について述べる．4 章では提案手法を述べ，5 章では提案手法の実装を述べ，6 章では評価実験について述べる．最後に，7 章では，実装に関する議論を述べ，8 章ではまとめと今後の課題を述べる．

2 Motivating Example

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いると言われ，保守作業の中でも，プログラム理解にかかるコストが大きな割合を占めると言われている [4, 5, 9].

プログラム理解で行う作業の中でも，LaToza ら [8] は，メソッドの実行で行われるデータの入出力について，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が指摘されている．具体的には，あるメソッドを実行したときに読み取られるフィールドの調査や，どのフィールドにデータを書き込むかの調査等を時間をかけて行なっていることが指摘されている．

このうち，メソッドにデータを与える方法として，引数として与える方法の他にも，クラス変数を介して必要なデータを与える方法もある．しかし，クラス変数はどのメソッドからでもアクセスが可能なため，注目しているメソッドとは別のメソッドで重要なクラス変数にアクセスしている場合がありえる．和歌山大学教務システムのプログラム中のメソッド GakuseiShowAction.getForm を例として説明する．このメソッドのソースコードを，ソースコード 1 に掲載する．このプログラムには，データベースのセッションを管理する変数である，HibernateUtil.SESSION というクラス変数が存在している．GakuseiShowAction.getForm では，直接はこのクラス変数にアクセスしていない．しかし，11 行目で呼び出している load メソッドは，幾つかのメソッド呼び出しを経て，HibernateUtil.currentSession というクラスメソッドを呼び出し，このクラスメソッドが HibernateUtil.SESSION にアクセスしている．このようなクラス変数への自明でないアクセスを見つけ出すには，メソッドの呼び出し関係を追跡し，呼び出されるメソッド全てを読解する必要があり，大きなコストがかかる．

また，メソッドは，与えられた引数が持つ全てのデータにアクセスするわけではない．先ほどと同じく，和歌山大学教務システムのプログラム中のメソッド UserUpdateAction.validateForm を例に説明する．GakuseiUpdateAction.execute 等で，このメソッドが呼ばれ，引数として与えられた form の検証を行っている．ただし，form は form.Id，form.userId，form.userKubun，form.name，form.password といった多数のフィールドを保持しているが，このメソッドは引数で渡している form のフィールドの一部にしかアクセスしていない．具体的には，4 行目のメソッド getId の呼び出しにより form.Id に，7 行目のメソッド getUserId により form.userId に，8 行目の getUserKubun により form.userKubun にアクセスしているが，それ以外のフィールドにはアクセスしていない．このことから，form が持っている form.name 等，他のフィールドの検証は，このメソッドでは行っていないと類推できる．逆に，アクセスしているフィールドを把握することで，form が持つ他のフィールドの検証は行っていないことが分かり，メソッドの挙動を理解する上で参考になると考えられる．しかし，アクセスされる可能性のあるフィールドを把握するには，データフローやメソッドの呼び出し関係を追跡

ソースコード 1: GakuseiShowAction.getForm

```
1  private GakuseiForm getForm(final GakuseiKeyForm keyForm)
2          throws ServiceException {
3    GakuseiForm form = null;
4
5    // id == 0 の場合、新規作成とみなす
6    if (keyForm.getId() == 0) {
7      form = new GakuseiForm();
8    } else {
9      Gakusei user = (Gakusei) ServiceFacotry
10                   .getService(IUserService.class)
11                   .load(keyForm.getId(), UserKubun.GAKUSEI);
12     form = new GakuseiForm(user);
13   }
14   form.setGakubuId(keyForm.getGakubuId());
15   form.setGakkaId(keyForm.getGakkaId());
16   form.setCourseId(keyForm.getCourseId());
17   return form;
18 }
```

ソースコード 2: UserUpdateAction.validateForm

```
1  protected boolean validateForm(final HttpServletRequest request,
2          final UserForm form) throws ServiceException {
3
4      if (form.getId() == 0) {
5          User user = ServiceFacotry
6                  .getService(IUserService.class)
7                  .findByUserID(form.getUserId(),
8                    UserKubun.valueOf(form.getUserKubun()));
9
10         if (user != null) {
11             addError(request, "errors.ucm02.exist.user");
12             return false;
13         }
14     }
15     return true;
16 }
```

する必要があり，これも大きなコストがかかる．

このように，メソッドの読解を行う際には，メソッドに入力されるデータのうち，実際に参照されたり，代入されたりするものが何か把握することが重要であると考えられる．また，参照や代入が行われるソースコード上の位置も，詳細な理解を行う上で重要な情報になると考えられる．

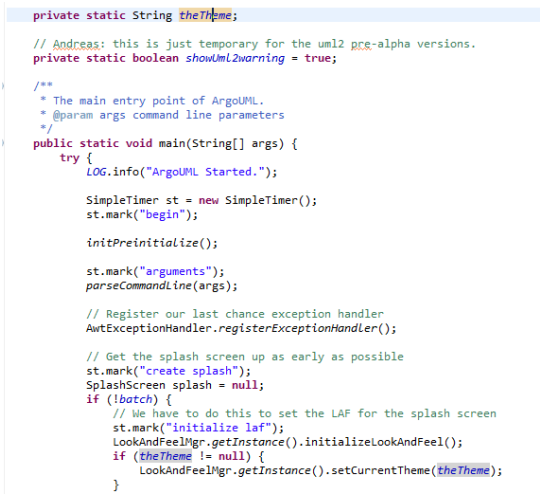

図 1: 識別子の強調表示機能

3 関連研究

本章では，既存のプログラム理解支援機能を紹介し，Motivation Example との関連を述べる．

3.1 統合開発環境 Eclipse のプログラム理解支援機能

本節では，Java の統合開発環境として広く用いられている Eclipse[1] を例として，統合開発環境に備わっているプログラム理解支援機能を紹介し，Motivating Example との関連を述べる．

3.1.1 識別子の強調表示機能

Eclipse の Java エディタ上で，注目する識別子にカーソルを合わせることで，ファイル内の同じ識別子が強調表示される．図 1 では，theTheme というフィールドを強調表示させている．この機能により，開発者は現在注目しているファイル内で識別子が参照されている箇所を容易に見つけることができ，データフローを追跡する際に有用である．

3.1.2 定義への移動機能，定義の閲覧機能

Java プログラムのソースコードの読解を行う際，呼び出されているメソッドや参照されている変数の定義を調査しなければならない場合がある．その際 Eclipse では，Java エディタ上でマウス操作やショートカットキーを用いて，呼び出されているメソッドや参照されて

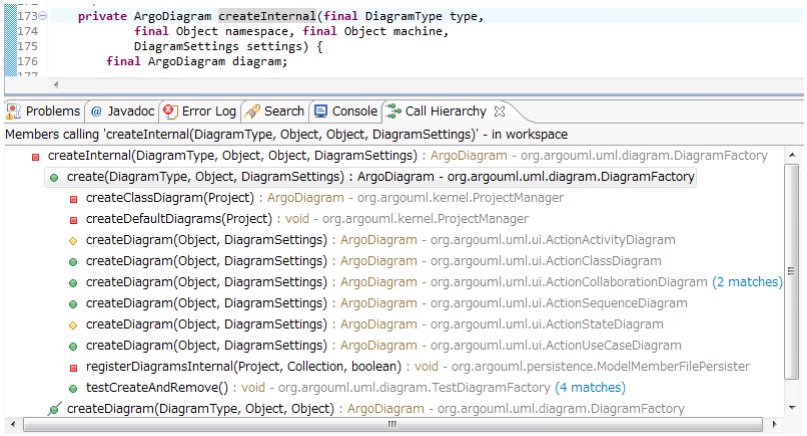

図 2: 呼び出し関係検索機能

いる変数の定義が記述されているコード片に容易に移動できる．

また，Declaration ビューでは，Java エディタ上で注目する識別子にカーソルを合わせることで，その識別子の定義を Declaration ビュー内に表示することができる．例えば，メソッド呼び出しにカーソルを合わせれば，呼び出されているメソッドの定義を表示できる．Declaration ビューにより，メソッド呼び出し先のコード片と呼び出し元のコード片を見比べたり，呼び出し先のコード片を詳細に読解する必要があるのか容易に知ることができる．

この機能は，メソッドの呼び出し関係を追跡する際に有用な機能である．

3.1.3 呼び出し関係検索機能

Eclipse では，プログラム内で定義されているフィールドやメソッド，コンストラクタについて，それらが呼び出されている箇所を Java エディタ上からマウス操作やショートカットキーを用いて検索することができる．また，メソッドやコンストラクタについては，それらが呼び出すメソッドを検索することもできる．検索結果は Call Hierarchy ビュー上にツリービュー形式で表示され，推移的な呼び出し関係をビュー上のツリーの子要素の展開させるだけで辿ることができる．図 2 では，メソッド createInternal の呼び出し元を検索した結果を Call Hierarchy ビューに表示している．

この機能により，注目しているメソッドが呼ばれる理由や状況を調査するのが容易となり，メソッドの呼び出し関係の調査に大変有用な機能である．

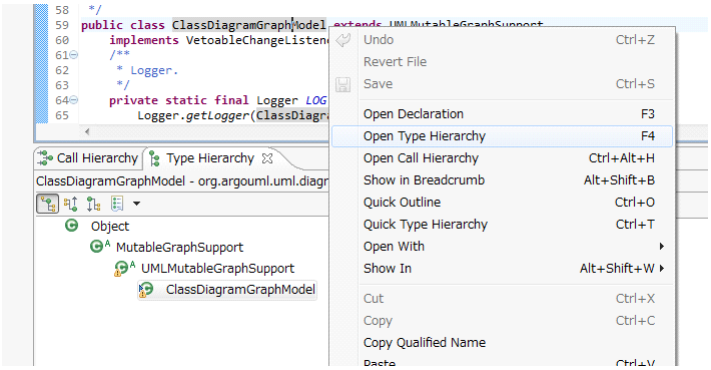

図 3: クラス階層表示機能

3.1.4 クラス階層表示機能

Java プログラムでは，継承を用いて，クラスやインターフェースに親子関係を持たせることができる．このため，開発者はプログラム理解の際，動的束縛を考慮しなければならず，注目するクラスと親子関係をもつクラスや，注目するメソッドをオーバーライドするメソッドに注意を払わなければならない．

Eclipse では，マウス操作やショートカットキーを用いて，選択したクラスの継承関係を表示することができる．また，選択したメソッドをオーバーライドするメソッドを表示することもできる．クラス階層を表示する際には，ツリービューの形式で表示され，多層的な継承関係を容易に辿ることができる．図 3 では，クラス ClassDiagramGraphModel を対象にクラス階層を表示させており，Object クラスまでのスーパークラスを階層的に表示させている．

この機能により，注目しているクラスの継承関係を把握しながらプログラム理解を進められる．したがって，メソッドの呼び出し関係の理解に有用といえる．

3.1.5 検索機能

Eclipse では，ファイル内に存在する文字列を検索する機能に加え，Java ソースコード中で定義されている識別子を検索する機能がある．この際識別子の種類や，識別子が現れる理由を条件にして検索することができる．図 4 に注目するプロジェクトの.java ファイルに対して「diagram」という文字列で検索した結果を表示している．検索結果はディレクトリ階層に応じて表示される．注目する文字列，識別子に対してファイルを横断的に調査したい場合に有用な機能である．

この機能は，メソッドの呼び出し関係の調査の際には，注目するメソッドの名前で検索することで，メソッドの呼び出し箇所を横断的に調査できる．また，データフローの調査の際

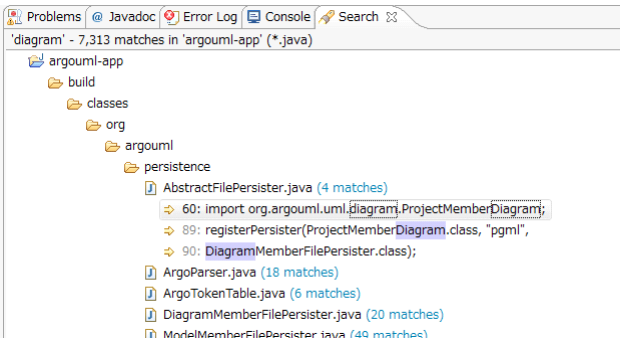

図 4: 検索機能

には，注目する変数の名前で検索することで，その変数を参照している箇所を網羅できる．

3.2 プログラムスライシング

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 [3, 6, 7, 10, 13, 16, 19]．プログラムスライシングを用いることで，例えば，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，注目しているメソッドに存在する文全てを入力としてプログラムスライシングを行えば，現在注目しているメソッドと関係のある文だけを残すことができ，注目しているメソッドが他の部分から受ける影響や他の部分に与える影響の特定に有用である．

Motivating Example で提示した，クラス変数への自明でないアクセスが存在する問題は，注目するメソッドに関係のある文を抽出し，残った文だけを探索することで，効率的にクラス変数へのアクセスを見つけだせる．また，使用されるフィールドを把握したいときには，注目する変数を対象にプログラムスライシングを行い，残った文に存在するフィールドアクセスを探索すれば，効率的な把握が可能となる．

しかし，プログラムスライシングの問題として，出力される文が多すぎるという問題がある．一般的にプログラムスライシングを行った場合，残る文の割合は，ソースコード全体に比して 3 割程度であり， 大規模なソフトウェアに対しては，プログラムスライシングを行なっても有用でない場合がある．また，正確なプログラムスライシングを行う場合，時間的・空間的なコストが大きなものとなり，実用上問題となる場合もある．

そして，プログラムスライシングでは，ある変数または文に依存する文全てを残すが，これはプログラム理解の際には問題となることを LaToza らは指摘している [8]．具体的には，プログラムスライシングは条件分岐等で実際には実行されない文を残してしまい，それが作

業者を混乱させてしまうことを指摘している．

3.3 データフローグラフによるコードナビゲーション

我々の研究グループの悦田らは，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 [21]．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．よって，使用されるフィールドの把握を行う際に有効に働くと考えられる．

4　提案手法

2 章で述べた Motivating Example に対し，既存研究はソースコードの探索を支援することにより，効率的にメソッドの入力を把握することができる．しかし，依然として探索の手間そのものが無くなったわけではない．

そこで本研究では，注目しているメソッドに対して自動的に探索を行い，メソッドに入力されるデータの一覧，データに対するアクセスの位置とアクセスの種類を抽出し，一覧表示することにより，プログラム理解支援を行う．

提案手法は，Java プログラムと指定されたメソッド m を対象に，そのメソッドの入力である変数名の集合と各変数に対して読み書きが行われたか，またその読み書きの命令が記述されたソースコード上での位置を出力する．

以降の節では，まず用語の定義を行い，メソッドの入力の定義，メソッドの入力の取得方法とその表示方法について詳解する．

4.1　用語の定義

変数 本論文において，変数とは，メソッドのローカル変数またはクラス変数とそれらから辿ることのできるフィールドを指す．例えば，ローカル変数 v とフィールド f1，f1 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．

変数へのアクセス 変数へのアクセスとは変数に対する代入または参照命令を指し，以下の 4 つ組で構成される．

- アクセス対象の変数
- 命令があるメソッド
- 命令があるソースコード上の行番号
- 読取命令であるか書込命令であるかの 2 値 (以下 RW と略記)

本論文においては，書込命令とは代入命令をさし，参照命令等それ以外の命令は全て読取命令であるとする．

4.2　メソッドの入力

本研究では，メソッドの入力とは，メソッドの実行前に初期化済みの変数のうち，メソッド実行中にアクセスした変数を指す．よって，以下の 2 つとなる．

(1) 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)

(2) m 実行中にアクセスする可能性のあるクラス変数全て

4.3 メソッドの入力取得手法と表示手法

本手法は，4.2 節で定義したメソッドの入力に対して行われるアクセスの組 I(m) を抽出し，それを表示する．

本手法は，大きく 2 つのステップに分かれている．

1. メソッド m に対する I(m) の取得
2. I(m) の表示

以降それぞれについて詳解する．

4.3.1 メソッドの入力に対するアクセスの取得

指定されたメソッド m に対する変数へのアクセスの集合 I(m) の取得アルゴリズムを Algorithm 1 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を抽出し，I(m) に追加する．アクセスとは，4.1 節で述べた通り，(v:変数, m:メソッド, l:行番号, α:RW) の 4 つ組の情報である．

このアルゴリズムで使用している関数の定義を表 1 に示す．アルゴリズム中の $\leftarrow$ は変数に対する代入を表す．$\twoheadleftarrow$ 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．

表中の関数 reachables(m) と isAlias(v1,v2)，isContinueSearch(v) は，アルゴリズムのパラメータとなる関数である．

- reachables(m) は，あるメソッドが呼び出しうるメソッドを取得する関数であるが，このときメソッド呼び出しの動的束縛を解決する必要がある．動的束縛の解決方法には，クラス階層解析の他，Rapid Type Analysis[2], Variable Type Analysis[18] 等の方法が提案されており，実装の際にどの手法を用いるか選択する．また，ライブラリのメソッド呼び出しに到達した際に探索を打ち切るかどうかもこの関数のにより指定される．

- isAlias(v1,v2) は，ある変数 v1 と v2 が同じポインタを指しうるエイリアス関係にあるかどうかを判定する関数である．これはエイリアス解析と呼ばれ，現在様々な方法が提案されている [11, 14, 15, 20]．これも rechables(m) と同様に，実装の際にどの手法を用いるのか選択する．

- isContinueSearch(v) は，ある変数の解析が終わったときに，それ以上探索を続けるかどうかを決める関数である．本手法は，フィールドの階層構造が複雑な場合やフィールドに再帰構造が含まれている場合に，解析が終了しない場合がある．また，あまりにも細かな情報を提示しても，作業者が表示された情報を全て見るのは非常に手間がかかると考えられるため，ある変数の解析時点で，その変数のフィールドにアクセス

しているかどうかに関わらず，探索を打ち切ることにした．これも実装でどのような場合に探索を打ち切るか選択する．

アルゴリズムを 4 つのステップに分けて解説する．

Step1 まず，I(m) を空集合に初期化し，m と m が呼びうるメソッドの集合を得る．m の引数を I(m) に追加する．このとき，引数へのアクセスは空とする．(アルゴリズム上では*を追加しているが，これは空を示す)

Step2 m と m が呼びうるメソッドの中に出現する，クラス変数へのアクセスを探して I(m) に追加する．

Step3 スタック worklist を空集合に初期化し，Step2 までで I(m) に追加した引数とクラス変数を worklist に追加する．

Step4 worklist から変数を一つ取り出す．m と m が呼びうるメソッドに含まれるフィールドアクセスのうち，取り出した変数と，フィールドアクセスのレシーバーオブジェクトがエイリアスであるものを列挙する．そして，I(m) にフィールドとフィールドアクセスを追加する．そして，探索を続けるならば v.f を worklist に追加する．

4.4 メソッドの入力に対するアクセスの表示

4.3.1 項で得られたメソッドの入力に対するアクセスを整理して，変数とフィールドの階層構造に合わせツリー状に表示する．この模式図を図 5 に示す．ルートノードとして m のメソッド名等を表示し，ルートノード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．なお，引数にレシーバーオブジェクトが含まれていれば，this という名前で表示する．

また，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，それを表示する．また，そのフィールドに対して読み書きのアクセスが行われたか表示する．(図中では RW と表現している) そして，フィールドの子要素として，フィールドアクセスがあるメソッドの名前と，そのメソッド内に読み書きが出現するか，フィールドアクセスのある行番号の集合が表示される．さらに，I(m) に先ほど表示したフィールドのフィールドが含まれていれば，フィールドの子要素として，同様にフィールドの情報を表示する．

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アクセス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，上述と同様に表示していく．

表 1: Algorithm 1 で使用する関数の定義

関数名	定義
params(m:メソッド)	m の引数の集合を返す.
reachables(m:メソッド)	m が呼びうるメソッドの集合を返す．この集合には m が推移的に呼びうるメソッドも含まれる.
searchClassVarAccess(M:メソッドの集合)	M に含まれるメソッドに存在するクラス変数へのアクセスの集合を返す.
isEmpty(c:集合)	c が空であれば真，そうでなければ偽を返す.
isAlias(v1:変数, v2:変数)	v1 と v2 が同じポインタを指しうるエイリアスの関係にあれば，真を返す．そうでなければ偽を返す.
variables(I:I(m))	I に含まれる変数の集合を返す.
pop(s:スタック)	s の要素を一つ返し，その要素を s から削除する.
fieldAccess(M:メソッドの集合)	M に出現するフィールドアクセスの集合を返す.
isContinueSearch(v:変数)	v のフィールドの深さを基に，探索を続けるかどうか返す．この関数はアルゴリズムのパラメータである.

Algorithm 1 I(m) の取得アルゴリズム

Input: m

Output: $I(m)$

1: $I(m) \leftarrow \emptyset$
2: $reachableMethods \leftarrow \{m\} \cup reachables(m)$
3: **for** $p \in params(m)$ **do**
4: $\quad I(m) \twoheadleftarrow (p, *, *, *)$
5: **end for**
6: **for all** $(v, m', l, \alpha) \in searchClassVarAccess(reachableMethods)$ **do**
7: $\quad I(m) \twoheadleftarrow (v, m', l, \alpha)$
8: **end for**
9: **for all** $v \in variables(I(m))$ **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** $(x.f, m', l, \alpha) \in fieldAccess(reachableMethods)$ **do**
15: $\quad\quad$ **if** $isAlias(v, x)$ **then**
16: $\quad\quad\quad I(m) \twoheadleftarrow (v.f, m', l, \alpha)$
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** $I(m)$

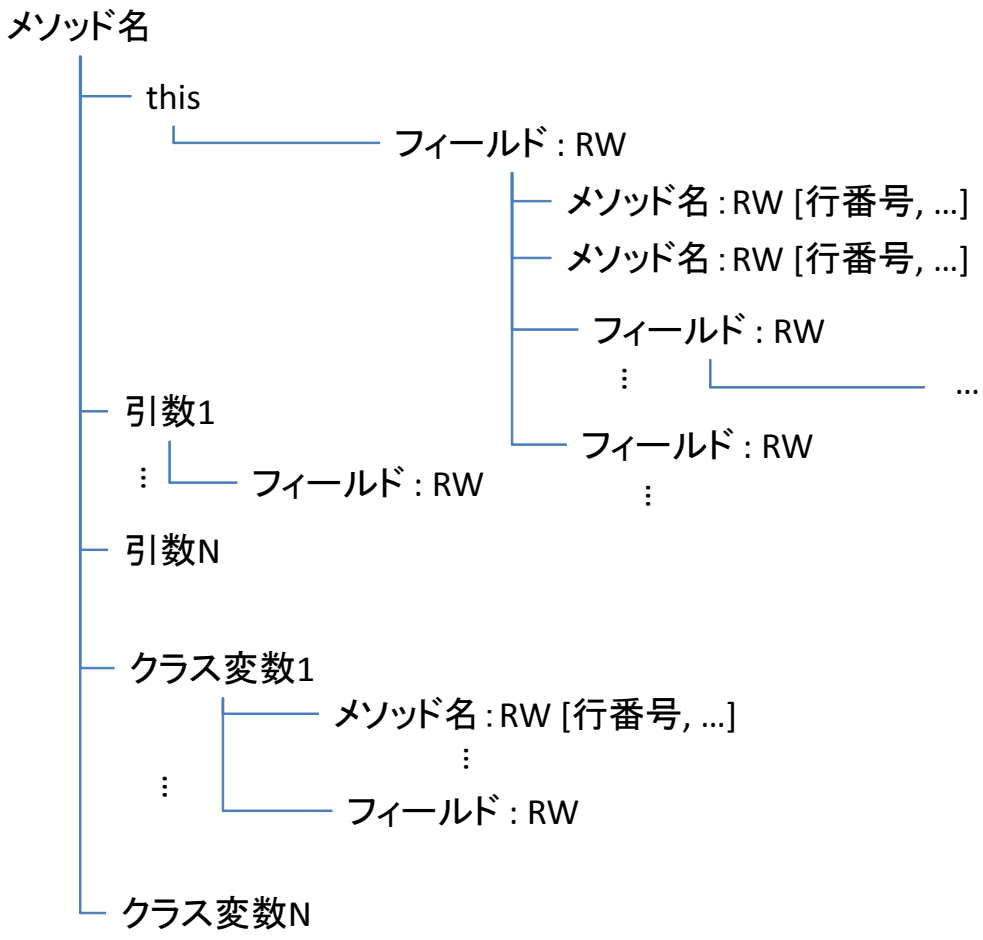

図 5: I(m) の表示の模式図

I(m) を表示する具体的なアルゴリズムを Algorithm 2 に示す．アルゴリズムの入力は，I(m) とツリーのルートノード rootNode である．

このアルゴリズムで表 1 以外の関数で新たに定義した関数を表 2 に示す．アルゴリズム中の矢印については，Algorithm 1 と同じく，← が代入を，↞ が集合への要素の追加を表す．

表示の方法は，対象メソッドのレシーバーオブジェクト（非クラスメソッドの場合），引数，I(m) に属するクラス変数を，ルートノードとして，それぞれに対しツリーを作成し表示する．ルートノードには，変数名，変数の型を表示する．クラス変数のノードについては，さらに，読み書きの情報と，アクセスが行われた命令があるメソッドの名前と，ソースコード上の位置も併せて表示する．

ルートノードをレシーバーオブジェクトとするフィールドが I(m) に含まれている場合，ルートノードの子ノードにフィールドを表すノードを追加する．ノードには，フィールド名，フィールドの型，読み書きの情報を表示する．さらにフィールドアクセスが行われた命令があるメソッドの名前とソースコード上での位置も併せて表示する．

さらに，追加した子ノードをレシーバーオブジェクトとするフィールドがあれば，先ほどと同様に子ノードにノードを追加する．

このアルゴリズムは主部と 2 つの手続きに分かれている．以下それぞれについて説明する．

主部 I(m) に含まれるレシーバーオブジェクト，引数，クラス変数を取り出す．レシーバーオブジェクトまたは引数の場合は，printVariable を呼び出し，ルートノードの子要素として変数を表示させる．クラス変数の場合は，printVariableAndAccess を呼び出し，ルートノードの子要素として変数とアクセスを表示させる．

手続き：printVariable 引数の node の子要素 c を取得し，そこに引数 v を表示する．つまり，引数のノードの子要素に，変数 v の情報を表示する．そして，v のフィールドが I(m) に含まれていれば，v のフィールドと c を引数として，printVariableAndAccess を呼ぶ（11-13 行目）．

手続き：printVariableAndAccess 引数の node の子要素 c を取得する．そして，変数 v に対するアクセス全てを取得する（17 行目）．c の内容として，変数 v の情報，変数 v に対するアクセスに読み込み命令があるか，変数 v に対するアクセスに書き込み命令があるか表示する．さらに変数 v に対するアクセスを，アクセスが出現するメソッドごとにグループ化する (22-24 行目)．そして，c の子要素に，メソッドの情報，メソッド内に読み込み命令があるか，メソッド内に書き込み命令があるか，命令がある行番号の集合を表示する（25-26 行目）．最後に，変数 v のフィールドが I(m) に含まれていれば，v のフィールドと c を引数として，printVariableAndAccess を呼ぶ（28-30 行目）．

表 2: Algorithm 2 で使用する関数の定義

関数名	定義
rootVariables(I:I(m))	I に含まれる，m の引数とクラス変数の集合が返される.
isClassVariable(v:変数)	v がクラス変数なら真を返し，そうでなければ偽を返す.
childNode(n:ノード)	ツリーでの n の子ノードを作成し，返す.
printNode(n:ノード, l:ラベル)	ラベルは出力する情報の組である．l を n の内容として画面等に出力する.
fields(I:I(m), v:変数)	I(m) の持つ組に含まれている v のフィールドの集合を返す.
containsRead(A:アクセスの集合)	A に読取命令が含まれていれば真を返す．そうでなければ偽を返す.
containsWrite(A:アクセスの集合)	A に書込命令が含まれていれば真を返す．そうでなければ偽を返す.
accessPoints(I:I(m), v:変数)	I の持つ組のうち，v が含まれている組に出現するアクセスの集合を返す.
methodsInAccessPoints(A:アクセスの集合)	A に含まれるアクセス命令が出現するメソッドの集合を返す.
lineNumbers(A:アクセスの集合)	A に含まれるアクセス命令が出現するソースコード上での行番号の集合を返す.

以上の手続きを呼び出し，I(m) に含まれる変数とフィールド，それらに対するアクセスの情報を表示する．

Algorithm 2 I(m) の表示アルゴリズム

Input: $I(m), rootNode$

```
1: for all v ∈ rootVariables(I(m)) do
2:     if not isClassVariable(v) then
3:         printVariable(v, rootNode)
4:     else
5:         printVariableAndAccess(v, rootNode)
6:     end if
7: end for
8: procedure printVariable(v, node)
9:     c ← childNode(node)
10:    printNode (c, v)
11:    for all f ∈ fields(I(m), v) do
12:        printVariableAndAccess(f, c)
13:    end for
14: end procedure
15: procedure printVariableAndAccess(v, node)
16:    c ← childNode(node)
17:    all ← accessPoints(I(m), v)
18:    printNode(c, (v, containsRead(all), containsWrite(all)))
19:    d ← childNode(c)
20:    for all m ∈ methodsInAccessPoints(all) do
21:        acInM ← ∅
22:        for all (v, m′, l, α) ∈ all ∩ m′ = m do
23:            acInM ↞ (v, m′, l, α)
24:        end for
25:        printNode(d,
26:        (m, containsRead(acInM), containsWrite(acInM), lineNumbers(acInM)))
27:    end for
28:    for all f ∈ fields(I(m), v) do
29:        printVariableAndAccess(f, c)
30:    end for
31: end procedure
```

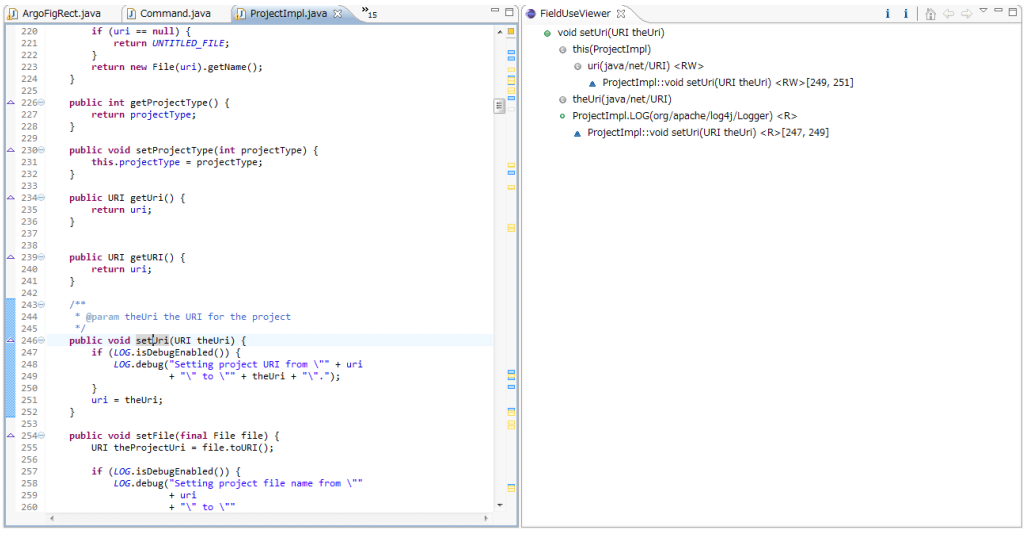

図 6: I(m) の表示部分実装

5 実装

実装では，メソッドから I(m) を求める解析部分と，I(m) を表示する部分は，別々のプログラムとして実装し，I(m) を表示する部分は Eclipse プラグインとして実装した．この Eclipse プラグインは起動時に解析結果を全て読み込み，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に対し，I(m) を表示する．図 6 では，メソッド setUri をクリックした時の出力を表示している．図右側の FieldUseViewer という名前の付いたビューが I(m) の表示ツールである．

I(m) 表示ツールは，Eclipse のツリービュー形式で実装されており，メソッドをクリックした際全要素を展開した状態で表示する．ルートノードには，クリックしたメソッドの戻り値とシグネチャを表示する．また，変数やフィールドの情報として，その変数やフィールドの型名も表示している．さらにアクセス命令のあるメソッド名をダブルクリックした際には，そのメソッドがあるクラスにジャンプするようにした．

実装では，動的束縛の解決に，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を用いる．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 [20] を用いる．動的束縛の解決，エイリアス解析には別の手法を選択することもできるが，これについての議論は 7 章で行う．また，I(m) の解析の際の探索の打ち切りについては，ある変数のフィールドのフィールドまでを解析し，そのフィールドから先は探索を打ち切るという実装にした．

また，I(m) の解析は，Java バイトコードを対象に解析を行った．Java バイトコード上で

は，フィールドに対するアクセス命令は，GETFIELD 命令または PUTFIELD 命令の 2 種類であり，それぞれフィールドに対する読取命令と書込命令となっている．そのため，フィールドアクセスが読取か書込かの判定については，実装では GETFIELD か PUTFIELD かの判定となっている．同様にクラス変数に対するアクセス命令も，Java バイトコード上では，GETSTATIC 命令と PUTSTATIC 命令の 2 つであり，それぞれ読取命令と書込命令である．これも同様に，アクセスが読取か書込かの判定は，GETSTATIC 命令か PUTSTATIC 命令かの判定となっている．

表 3: 被験者のプログラム経験等

	プログラミング経験年数	Java プログラミング経験年数	最長行数	保守経験
被験者 A	5	2	3000	Y
被験者 B	5	2	3500	N
被験者 C	5	2	1000	Y
被験者 D	5	2	1000	N
被験者 E	6	6	4000	N
被験者 F	7	1	1500	N
被験者 G	7	5	1500	Y
被験者 H	4	2	3000	Y

6 評価実験

本ツールが実際のプログラム理解の活動に有効であるか評価実験を行った．本実験では，2 つのアプリケーションを対象にした課題を出し，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．また，被験者にはアンケートを行った．アンケートの質問内容は，作業を効率的にできたかどうか，本ツールが作業に役立ったと思うか，統合開発環境の各機能について役立ったと思うか，などである．なお，本実験は [12] を参考にしており，課題内容や実験方法はほぼ [12] のものと同じである．

以降では，実験の詳細と実験結果を述べる．

6.1 被験者

被験者は，大阪大学基礎工学部情報科学科 4 年の 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 5 人である．全被験者は，Java と Java の統合開発環境である Eclipse に習熟している．

表 3 にアンケートにより自己申告してもらった，被験者のプログラミング経験年数，Java プログラミングの経験年数，過去に作成したプログラムの最長行数，他人の書いたプログラムの保守経験の有無について記述している．

6.2 実験方法

実験では被験者に 2 つの異なる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てもらう．その際，一方のアプリケーションはツール有りで作業してもらい，もう一方のア

プリケーションではツール無しで作業をしてもらう．アプリケーション一つにつき課題を 3 つ用意し，課題の正答率と解答時間を測定する．また，アンケートを行い，被験者の解答プロセスの効率性や解答に対する確信度，本ツールや Eclipse の機能が課題の解答にどのくらい貢献したのか，などを質問する．

まず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であるが，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，ArgoUML（0.34）[1]と GanttProject（2.0.9）[2]を選択した．ArgoUML は UML 図作成ツールであり，GanttProject はガントチャート作成ツールである．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．なお，LOC の測定には cloc-1.55[3]を使用した．

各アプリケーションには課題が 3 つ用意されており，課題は以下の通りである．

ArgoUML

課題 1 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．

課題 2 ユーザが新たな要素を図に追加する操作（例えばクラスをクラス図に追加する操作）を実装しているメソッドがどれか答えよ．

課題 3 どの要素が選択されたのか，記録し保存する役割を持つクラスはどれか答えよ．

GanttProject

課題 1 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．

課題 2 どのクラスが GanttDiagram 内の task（box）を描画する役割を持つのか答えよ．

課題 3 どのクラスが，タスク間の依存性の情報を持っているのか？（クラス名をパッケージ名付きで答えよ）

実験全体の手順は以下の通りとなる．

[1] http://argouml.tigris.org/
[2] http://www.ganttproject.biz/
[3] http://cloc.sourceforge.net

1. 実験全体の説明，Eclipse の代表的な 5 つの機能についてのレクチャー，本ツール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. 一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．
3. 一つ目のアプリケーションに対する実験を行う．課題全体の制限時間は 45 分とする．
4. 一つ目のアプリケーションに対する実験についてのアンケートを行う．
5. 二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. 二つ目のアプリケーションに対する実験を行う．課題全体の制限時間は 45 分とする．
7. 二つ目のアプリケーションに対する実験についてのアンケートを行う．

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，新規タスクの一つをもう一方のタスクの先行タスクとする．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．

被験者は一つの課題を解答する際に必要なだけ時間を使うことができる．ただし，一度解答し終えた課題に戻ることはできない．また，アプリケーションごとには 45 分の制限時間を割り当てており，制限時間に達したところで作業は終えてもらう．制限時間に達したときに行なっていた課題が解答できるようなら，被験者には課題の解答を行ってもらった．このとき，課題の解答にかかった時間は，制限時間に達した時の時間とする．

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 4 のように行った．また，実験は被験者一人ずつ個別に行う．

各アプリケーションに対する実験が終わった後に行うアンケートの内容は以下の通りである．

- タスクを効率的にこなすことができたと思いますか？5 段階評価で回答してください (5: 効率的だった 4: まあまあ効率的だった 3: わからない 2: あまり効率は良くなかった 1: 効率的では無かった)

表 4: 課題の割り当て

	1 回目	2 回目
被験者 A,B	ArgoUML（ツール有り）	GanttProject(ツール無し）
被験者 C,D	GanttProject(ツール無し）	ArgoUML（ツール有り）
被験者 E,F	ArgoUML（ツール無し）	GanttProject(ツール有り）
被験者 G,H	GanttProject(ツール有り）	ArgoUML（ツール無し）

- 正確な答えが得られたと思いますか？ 5 段階評価で回答してください (5: 正確だと思う 4: まあまあ正確だと思う 3: わからない 2: あまり正確ではないと思う 1: 正確でないと思う)
- タスク作業において以下のツールが役立ったか，5 段階評価で回答してください (5: 役にたった 4: まあまあ役に立った 3: わからない 2: あまり役に立たなかった 1: 役に立たなかった)
  - File Search
  - Java Search
  - Open Call Hierarchy
  - Open Type Hierarchy
  - Open Declaration
  - 本ツール（ツールを使用していない場合は，飛ばしてください）
- 作業中に本ツールが役に立ったと思える点があれば回答してください．(ツールを使用していない場合は，飛ばしてください)
- 本ツールに対する感想・不満点等を回答してください． (ツールを使用していない場合は，飛ばしてください)
- 作業全体に関する感想等あれば記述してください．

なお，被験者には各アプリケーションで，課題と関連のあるクラスが一つずつ提示されている．これらのクラスは，作業を開始時に Eclipse の Java エディタで開いている状態とした．具体的には以下の通りである．

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

GanttProject net.sourceforge.ganttproject.GanttTask

表 5: 解答時間（ArgoUML）

	ツール有無	課題 1	課題 2	課題 3
被験者 A	有	18 分 29 秒	14 分 03 秒	時間切れ
被験者 B	有	24 分 40 秒	12 分 27 秒	時間切れ
被験者 C	有	6 分 23 秒	8 分 15 秒	12 分 25 秒
被験者 D	有	18 分 34 秒	時間切れ	
被験者 E	無	時間切れ		
被験者 F	無	38 分 32 秒	時間切れ	
被験者 G	無	時間切れ		
被験者 H	無	28 分 29 秒	5 分 23 秒	9 分 11 秒

6.3 作業環境

被験者は作業環境として，27 インチのディスプレイ（解像度 1920 x 1200）1 枚を使い，Eclipse3.7 を使用する．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う．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．ただし，実際にメモ帳を使用した被験者はおらず，メモ用紙を使用した被験者も 1 名のみであった．

Eclipse にはプログラムのデバッガが付属しているが，本実験ではデバッガの使用は禁止とした．また，プログラムを Eclipse 上で実行することそのものも禁止した．これは，静的なプログラム理解に対するツールの有効性を検証したかったからである．

6.4 実験結果

6.4.1 正答率と解答時間

各課題の解答時間について，ArgoUML については表 5，GanttProject については表 6 に示す．表中の「時間切れ」とは，その課題の作業中に，各アプリケーションごとに定められた制限時間に到達し，作業を打ち切ったことを指している．

また，各課題の解答の正誤を，ArgoUML については表 7，GanttProject については表 8 に示す．

表 5，表 6 で示している通り，殆どの被験者は全ての課題を解答することができなかった．よって，正答率と解答時間の比較は，ArgoUML と GanttProject の課題 1 のみに対して行う．表 9 にツールの有無でグループ化した課題 1 の正答率を示す．また，図 7 にツールの有無でグループ化した課題 1 の解答時間の箱ひげ図によるプロットを示す．さらに，図 8 に正

表 6: 解答時間（GanttProject）

	ツール有無	課題 1	課題 2	課題 3
被験者 A	無	24 分	6 分 40 秒	3 分 26 秒
被験者 B	無	20 分 37 秒	18 分 38 秒	時間切れ
被験者 C	無	13 分 13 秒	20 分 05 秒	9 分 20 秒
被験者 D	無	31 分 28 秒	時間切れ	
被験者 E	有	時間切れ		
被験者 F	有	14 分 48 秒	時間切れ	
被験者 G	有	30 分 47 秒	時間切れ	
被験者 H	有	19 分 01 秒	時間切れ	

表 7: 解答の正誤（ArgoUML）

	ツール有無	課題 1	課題 2	課題 3
被験者 A	有	○	×	×
被験者 B	有	○	○	×
被験者 C	有	×	×	×
被験者 D	有	×	×	×
被験者 E	無	×	×	×
被験者 F	無	×	×	×
被験者 G	無	○	×	×
被験者 H	無	×	×	○

表 8: 解答の正誤（GanttProject）

	ツール有無	課題 1	課題 2	課題 3
被験者 A	無	○	○	○
被験者 B	無	×	×	×
被験者 C	無	×	×	×
被験者 D	無	×	×	×
被験者 E	有	×	×	×
被験者 F	有	×	×	×
被験者 G	有	○	○	×
被験者 H	有	×	×	×

表 9: 課題 1 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）

答した被験者のみの解答時間の箱ひげ図によるプロットを示す．また，図 9 にツールの有無によらず，対象アプリケーションでグループ化した解答時間の箱ひげ図によるプロットを示す．ただし，制限時間に到達した際に解答できなかった被験者の解答時間は欠損値とした．

6.4.2 アンケート結果

本項ではアンケート結果について述べる．

まず，アンケート内容のうち，5 段階評価をしてもらった項目について示す．このうち，「効率的に課題をこなせたと思うか」と「正確な解答を得られたと思うか」のアンケート結果を表 10 に示した．また，Eclipse の代表的な機能 5 つと本ツールについてのアンケート結果については表 11 に示した．両方の表中の 2 列目には，アプリケーション名を示しており，ツールを使用していればアプリケーション名の後ろに「（有）」と記した．また，表 11 では，Eclipse の代表的な機能の名前に省略名称を使用している．省略名称はそれぞれ File Search(FS)，Java Search(JS)，Open Call Hierarchy(CH)，Open Type Hierarchy(TH)，Open Declaration(OD) である．

次に，自由記述のアンケート項目について述べる．

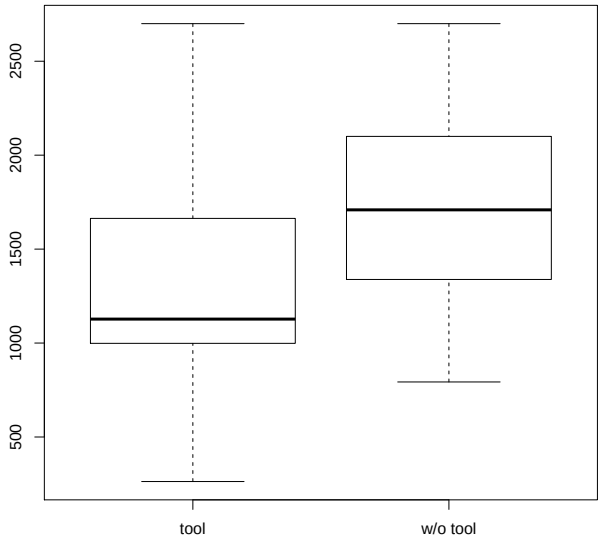

図 7: 課題 1 の解答時間 (秒)

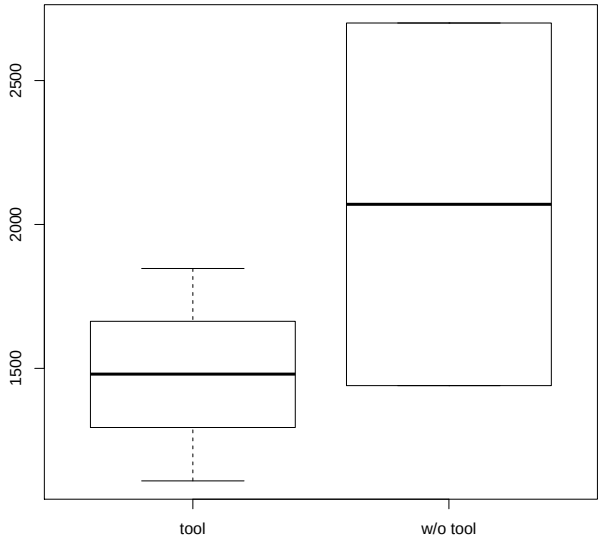

図 8: 課題 1 の正答者の解答時間 (秒)

表 10: 課題の解答についての効率性・正確性についてのアンケート結果

		効率的に課題をこなせたと思うか	正確な解答を得られたか
被験者 A	ArgoUML（有）	2	2
	GanttProject	4	4
被験者 B	ArgoUML（有）	2	3
	GanttProject	3	2
被験者 C	ArgoUML（有）	3	3
	GanttProject	2	3
被験者 D	ArgoUML（有）	3	4
	GanttProject	1	2
被験者 E	ArgoUML	2	1
	GanttProject（有）	3	4
被験者 F	ArgoUML	2	2
	GanttProject（有）	2	2
被験者 G	ArgoUML	2	3
	GanttProject（有）	4	4
被験者 H	ArgoUML	3	2
	GanttProject（有）	2	1

表 11: 各ツールの有効性に関するアンケート結果

		FS	JS	CH	TH	OD	本ツール
被験者 A	ArgoUML（有）	1	2	5	1	5	3
	GanttProject	1	4	5	1	4	
被験者 B	ArgoUML（有）	2	1	4	2	2	3
	GanttProject	2	2	4	2	2	
被験者 C	ArgoUML（有）	1	1	2	5	5	4
	GanttProject	2	2	4	5	5	
被験者 D	ArgoUML（有）	4	1	4	3	5	4
	GanttProject	3	1	5	3	5	
被験者 E	ArgoUML	2	4	4	4	5	
	GanttProject（有）	2	4	4	3	5	4
被験者 F	ArgoUML	2	4	4	3	3	
	GanttProject（有）	2	3	3	3	1	3
被験者 G	ArgoUML	2	2	5	2	5	
	GanttProject（有）	3	4	5	2	5	3
被験者 H	ArgoUML	2	1	5	3	5	
	GanttProject（有）	1	1	3	4	4	2

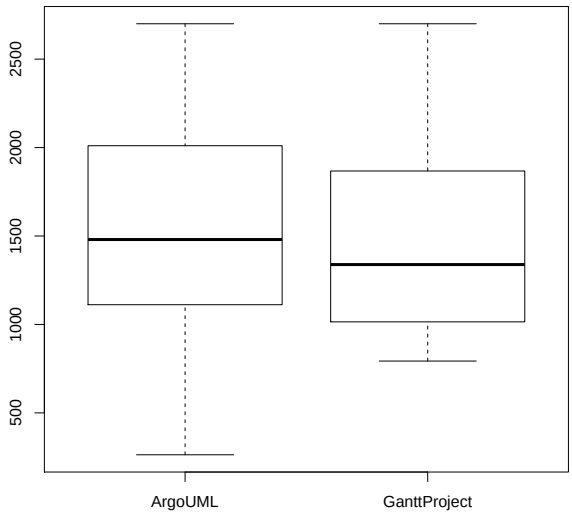

図 9: 各アプリケーションの解答時間 (秒)

まず,「作業中に本ツールが役にたったと思える点があれば回答してください」の項目のアンケート結果は以下の通りであった.

- 変数の使われ方の確認
- メソッド選択で自動的に表示できるので，スムーズに閲覧することができた
- 関係ありそうなメソッドをクリックしたときに，そのメソッドと関連のあるクラスファイルに簡単に飛べるのが良かった
- あるメソッドの中身は効率的に大体イメージできる
- RW によりどこで書き込みが行われているか把握できる
- 入力変数を多く見ることで，処理と関連がありそうな名前の変数を探すことができる
- どんなフィールド・メソッドが使用されるのか一覧表示されるのは，何かを探すときにいちいち別の画面に切り替わらなくて，思考を停止しない

「本ツールに対する感想・不満点等を回答してください」の項目のアンケート結果は以下の通りであった.

- キャッシュを使って欲しい

- 表示数を減らして欲しい（ツリーを閉じる等）
- 正直なところ，今回の課題でどう活用すればよかったのか分からないまま終わってしまった
- 使いはじめであったので，どのようなときに活用できるかいまいちつかめなかった．継続して利用してみたいと思った
- 表示される結果が多すぎるときには使おうという気にならない
- たまに多すぎる情報が出てよくわかりません．その時 Eclipse が重くなる
- ダブルクリックしたとき，該当する行番号に飛ぶといいかも
- 長いメソッドになると，列挙されるものが多すぎて何処を見ればよいか分からない場合がある
- ただ一覧でずらっと文字だけが表示されると，少し見にくい．一覧が見やすくなる工夫がされていると使いやすいと思う

「作業全体に関する感想等あれば記述して下さい」の項目のアンケート結果は以下の通りであった．

- 課題が難しかった
- インターフェースから先をどう辿るかが難しかった
- コードを行ったり来たりして時間がかかってしまった
- 英語わからなくて辛かった
- 補助なしだとなかなか大変だと思った．特に英語
- プロジェクト全体のイメージ把握できない
- このツールの使い方をもっと練習したい
- 探す方針がなかなか定まらずに時間がかかった．クラスの構造や変数名，メソッド名などの意味がある程度予測できないと難しい
- Eclipse もツールも，もう少し使いこなすことができれば作業効率を上げられたと思う
- 自分には Java の知識が無いことを実感した

6.5 考察

実験結果の図 7 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 9 が示す両アプリケーションの被験者の解答時間より，二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，作業者の探索時間を減らす効果があったと考えられる．

しかし，表 9 の通り，ツールの有無に関わらず正答率は非常に低く，殆どの被験者は正解できなかった．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たとは言えない．ただし，本ツールを使用した被験者の正答率がツール無しの被験者の正答率よりも低いわけではない．よって，ツール有りの被験者の探索時間が減少は正答率に影響を与えたわけではない．

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．有意水準は 0.05，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，$p > 0.05$ より，帰無仮説が採択された．よって，ツールの有無による作業時間の差は有意と言えるほどのものでは無かった．

課題の際の本ツールの使用例には以下が挙げられる．

- 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

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．そのため，ジャンプ機能については，アクセス箇所に直接飛べるように改良した方が望ましいと考えられる．

本ツールが有効に働かなかった原因として考えられるのが，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もあったことである．これは，アンケートの本ツールに対する感想・不満点でも触れられており，表示数が多すぎる場合に被験者はツール出力を見るのを諦めたか全く見ようとはしなかった．また，大量の要素を表示しようとして，被験者の作業を妨害してしまった場合もあった．

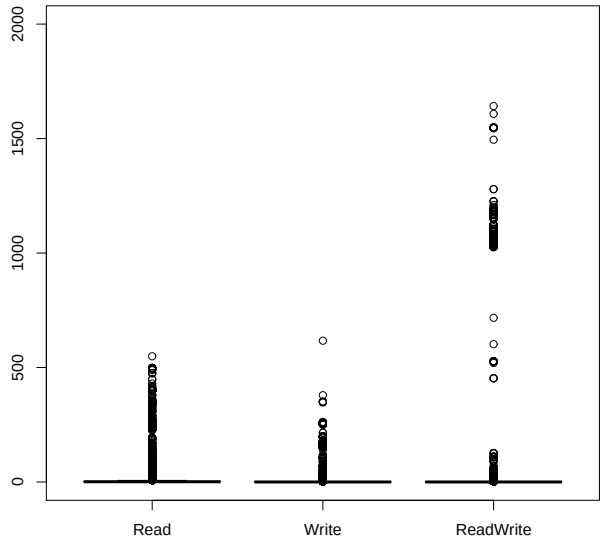

図 10: ArgoUML に表示された要素数の分布

本ツールで表示される要素数については，追加の調査を行った．図 10 と図 11 は，それぞれ ArgoUML と GanttProject に対する表示結果のうち，一メソッドに表示された，R，W，RW とされたフィールドまたはクラス変数の数を箱ひげ図によりプロットした図である．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，100 個や 1000 個以上の要素が表示されてしまうことが分かる．要素数がこれ程多い場合，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．

また，メソッドに出力される要素数とメソッド数のヒストグラムを ArgoUML の R，W，RW，GanttProject の R，W，RW についてそれぞれ図 12,13,14,15,16,17 に示す．全てのヒストグラムは，最小値 0 で 10 刻みで階級を区分している．全てのヒストグラムで共通して見られる傾向として，殆どのメソッドで表示される要素数は 10 より少ないということが挙げられ，本ツールで表示される要素数には大きなばらつきがあることが分かった．そのため，本ツールは，表示される要素の数が少ない場合はそのまま表示し，多い場合には出力の要約を行ってから表示しなければならないと考えられる．

6.6 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の問題で

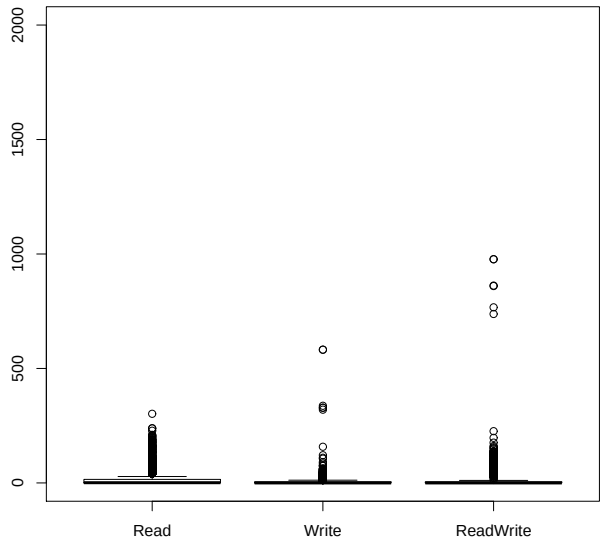

図 11: GanttProject に表示された要素数の分布

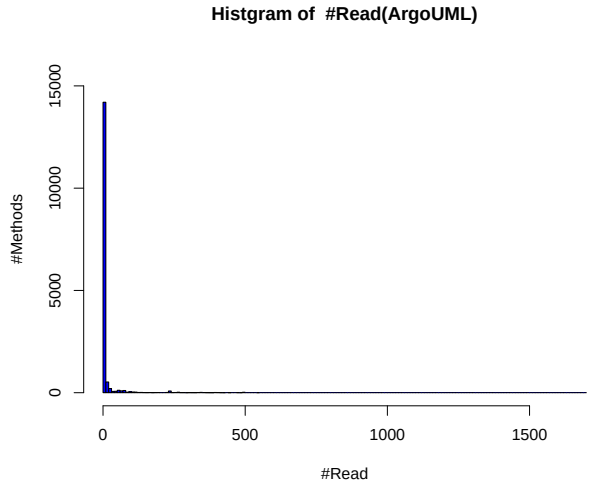

図 12: Read と表示された要素数とメソッド数のヒストグラム (ArgoUML)

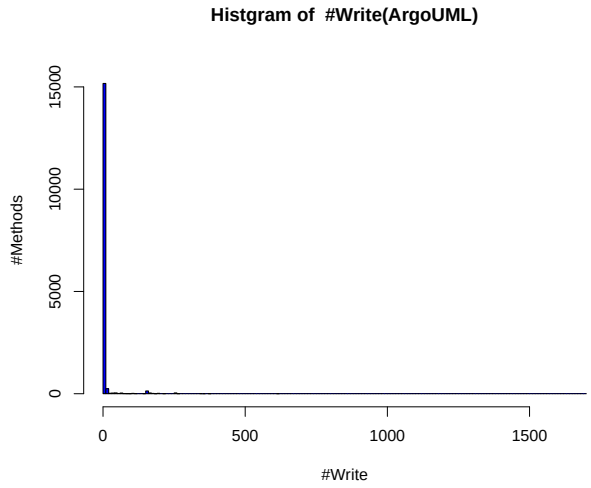

図 13: Write と表示された要素数とメソッド数のヒストグラム (ArgoUML)

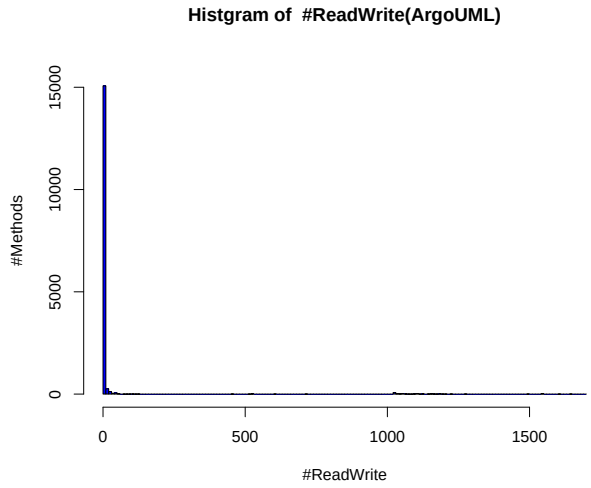

図 14: ReadWrite と表示された要素数とメソッド数のヒストグラム (ArgoUML)

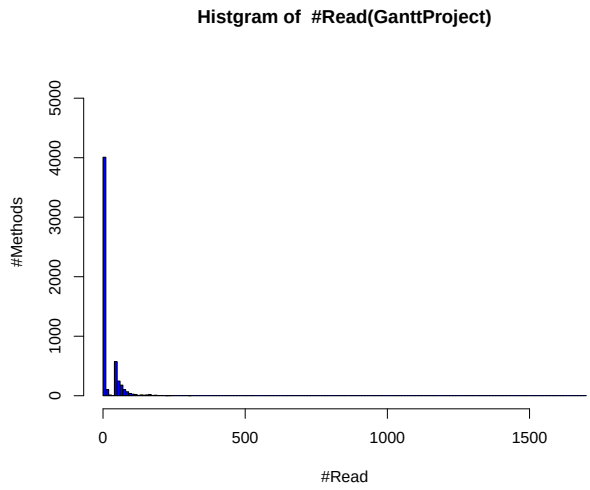

図 15: Read と表示された要素数とメソッド数のヒストグラム (GanttProject)

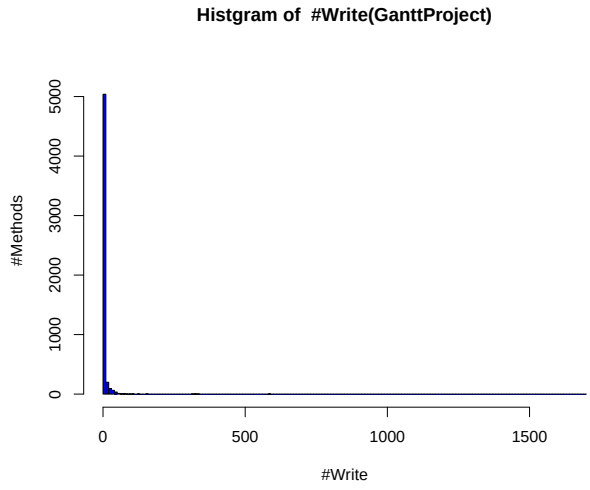

図 16: Write と表示された要素数とメソッド数のヒストグラム (GanttProject)

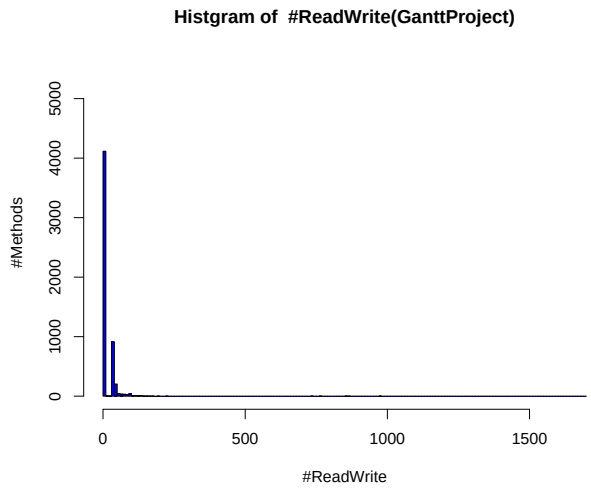

図 17: ReadWrite と表示された要素数とメソッド数のヒストグラム (GanttProject)

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してある程度知識を持った状態でプログラム理解を行う場合もあり，そのような場合に本実験の結果が完全に当てはまるとは言えない．

7　議論

本章では，提案手法でアルゴリズムのパラメータとした，動的束縛の解決方法とエイリアス解析について，実装で用いた方法とそれ以外の方法を紹介し，議論を行う．

7.1　動的束縛の解決方法に関する議論

動的束縛の解決には，実装で用いたクラス階層解析の他に，変数型解析（Variable Type Analysis）[18]，高速型解析（Rapid Type Analysis）[2] 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) の解析結果から取り除くことができる．

しかし，実装で用いたクラス階層解析による動的束縛の解決は，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．よって，I(m) に出現する変数やフィールドには，実際にアクセスし得るものよりも大きな集合となるものの，小さくなることはなく，解析された範囲ではメソッドの入力を見落とすことは無いと言える．

7.2　エイリアス解析に関する議論

エイリアス解析，またエイリアス解析とほぼ同様の解析であるポインタ解析には，様々な手法が提案されている [11, 14, 15, 17, 20]．より正確な手法を用いれば，正確にレシーバーオブジェクトを特定することができ，実際にはメソッドの入力からアクセスしたわけではないフィールドを 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 [20] 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．

8　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，メソッドに入力され使用される可能性のある引数のフィールドやクラス変数を抽出し，それらに対するアクセスが出現するソースコード上の位置とアクセスの種類を可視化するツールを作成した．評価実験では，被験者にプログラム理解の課題を解いてもらい，ツールの有無による解答時間や正答率の差を測定した．その結果，本ツールが作業者が解答を得るまでの作業時間を減らすのに効果があったことを確認した．

今後の課題としては，出力の表示方法の改良，ジャンプ機能の拡充，多すぎる出力の要約が挙げられる．また，メソッドの戻り値についても本手法と同様に読み書きされるフィールドやその命令の場所を表示するということが考えられる．他には，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へのアクセスを同時に閲覧できるようにすることも考えられる．

謝辞

本研究の全過程を通じて，常に適切な御指導および御助言を賜り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井上克郎教授に心より深く感謝いたします.

本研究の全過程を通じて，適切な御指導および御助言を賜り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松下誠准教授に心より深く感謝いたします.

本論文を作成するにあたり，常に適切な御指導および御助言を賜り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石尾隆助教に心より深く感謝いたします.

本論文を作成するにあたり，適切な御指導および御助言を賜り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眞鍋雄貴特任助教に心より深く感謝いたします.

評価実験の実施にあたり，被験者として快く協力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井上研究室学生の皆様に心より深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂い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井上研究室の皆様に深く感謝いたします.

参考文献

[1] Eclipse. `http://www.eclipse.org/`.

[2] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '96, pp. 324–341, 1996.

[3] Zhenqiang Chen and Baowen Xu. Slicing object-oriented java programs. *SIGPLAN Notices*, Vol. 36, pp. 33–40, April 2001.

[4] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Systems. Journal.*, Vol. 28, pp. 294–306, June 1989.

[5] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: report to our respondents. In *Proceedings of GUIDE 48*, 1979.

[6] Mark Harman and Sebastian Danicic. Amorphous program slicing. *In proceedings of the 5th International Workshop on Program Comprehension*, pp. 70–79, April 1997.

[7] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pp. 35–46, June 1988.

[8] Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, Vol. 1 of *ICSE '10*, pp. 185–194, May 2010.

[9] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pp. 492–501, May 2006.

[10] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pp. 358–367, November 1998.

[11] Ana Milanova. Light context-sensitive points-to analysis for java. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pp. 25–30, June 2007.

[12] Jochen Quante. Do dynamic object process graphs support program understanding? - a controlled experiment. *In Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC'08*, pp. 73–82, June 2008.

[13] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, FSE '94, pp. 11–20, December 1994.

[14] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pp. 1–14, January 1997.

[15] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the ACM SIGPLAN 2006 conference on Programming language design and implementation*, PLDI '06, pp. 387–400, June 2006.

[16] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the ACM SIGPLAN 2007 conference on Programming language design and implementation*, PLDI '07, pp. 112–122, June 2007.

[17] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pp. 59–76, October 2005.

[18] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pp. 264–280, October 2000.

[19] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pp. 439–449, March 1981.

[20] Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pp. 155–165, July 2011.

[21] 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.

付録

付録では実験で行った各課題に対する被験者の解答と，こちらが正解とした解答を記載する．

表 12: ArgoUML 課題 1 の被験者の解答

被験者	解答
A	org.argouml.kerne.ProjectManager
B	org.argouml.kernel.ProjectManager
C	org.argouml.diagram.static_structure.ui.UMLClassDiagram
D	org.argouml.diagram.DiagramFactory
E	解答なし
F	org.argouml.uml.diagram.static_structure.ui.UMLClassDiagram
G	org.argouml.kenel.ProjectManger
H	org.argouml.uml.diagram.static_structure.ui.UMLClassDiagram

表 13: ArgoUML 課題 2 の被験者の解答

被験者	解答
A	org.argouml.diagram.static_structure.ui.UMLClassDiagram
B	org.argouml.diagram.ui.ModeAddToDiagram.mouseReleased
C	org.argouml.uml.diagram.static_structure.ClassDiagramGraphModel.addNode(Object)
D	解答なし
E	解答なし
F	解答なし
G	解答なし
H	org.tigris.gef.graph.MutableGraphSupport.fireNodeAdded

表 14: ArgoUML 課題 3 の被験者の解答

被験者	解答
A	org.argouml.uml.diagram.UMLMutableGraphSupport
B	解答なし
C	org.argouml.model.Model
D	解答なし
E	解答なし
F	解答なし
G	解答なし
H	org.argouml.ui.targetmanager.TargetManager $ HistoryManager

表 15: GanttProject 課題 1 の被験者の解答

被験者	解答
A	RecalculateTaskScheduleAlgorithm.fulfilConstraints (TaskDependency):188-222
B	net.sourceforge.ganttproject.task.algorithm. AdjustTaskBoundsAlgorithm.adjustNestedTasks:69
C	net.sourceforge.ganttproject.task.MutatorImpl.commit:行番号無し
D	net.sourceforge.ganttproject.task.algorithm. AdjustTaskBoundsAlgorithm.adjustNestedTasks:42
E	net.sourceforge.ganttproject.task.algorithm. AdjustTaskBoundsAlgorithm.adjustNestedTask(Task supertask):42-75
F	net.sourceforge.ganttproject.task.TaskImpl.shift():980-994
G	RecalculateTaskScheduleAlgorithm.fulfilConstraints:222
H	TaskImpl.recalculateActivities

表 16: GanttProject 課題 2 の被験者の解答

被験者	解答
A	GanttGraphiArea
B	net.sourceforge.ganttproject.io.GanttXFIGSaver
C	net.sourceforge.ganttproject.task.FacedFactoryImpl （の importData(Task, Task, Map) と解答）
D	net.sourceforge.ganttproject.gui.GanttTaskPropertiesBean
E	解答なし
F	解答なし
G	GanttGraphicArea (の paintComponent: 289 行目と解答）
H	解答なし

表 17: GanttProject 課題 3 の被験者の解答

被験者	解答
A	net.sourceforge.ganttproject.task.dependency.TaskDependencyCollectionImpl
B	解答なし
C	net.sourceforge.ganttproject.task.dependency.TaskDependencyImpl
D	解答なし
E	解答なし
F	解答なし
G	解答なし
H	解答なし

表 18: 各課題の正解

課題	解答
ArgoUML 課題 1	org.argouml.kernel.ProjectManager
ArgoUML 課題 2	org.argouml.uml.diagram.ui. ModeAddToDiagram.mouseReleased(MouseEvent me)
ArogUML 課題 3	org.argouml.ui.targetmanager. TargetManager.HistoryManager
GanttProject 課題 1	net.sourceforge.ganttproject.task. algorithm.RecalculateTaskScheduleAlgorithm. fulfilConstraints (TaskDependency dependency) : 150 行目から 223 行目 （このメソッドを指定していれば正解とした）
GanttProject 課題 2	net.sourceforge.ganttproject. GanttGraphicArea
GanttProject 課題 3	net.sourceforge.ganttproject.task.dependency. TaskDependencyCollectionImpl