Vol.2013-SE-182 No.5
2013/10/24

デルタ抽出プラグインの開発とそのプログラム理解に対する有効性評価

松岡 智大[1,a)]　新田 直也[1,b)]

概要：デバッグや再利用を適切に行う上で対象となるプログラムの理解は必要不可欠である．一般に, プログラムの動的振る舞いは広範囲に渡るソースコードによって実現されている場合が多く, その理解のために, ソースコードに加えてデバッガなどで得られるプログラム実行時の情報も参照されることが多い．しかしながら, 実用規模プログラムでは実行時に生成される情報が非常に膨大になり, その中から必要な情報を取り出すには多くの時間と労力を要する．そこで本研究では, オブジェクト指向プログラムの実行履歴から, プログラム理解に有用な情報を効率良く取り出すための抽象化手法 (デルタ抽出) を提案している．本稿では, プログラム理解の支援を目的としてデルタ抽出を行う Eclipse プラグインの実装を行った．また, プラグインの有効性を評価するため技術者を対象とした実証実験を行ったので, その結果について紹介する．

1. はじめに

デバッグや再利用を適切に行う上で対象となるプログラムの理解は必要不可欠である．一般にプログラムを理解する方法としては静的なアプローチと動的なアプローチがある．静的なアプローチでは, ソースコードを直接読んだりツールを用いてソースコードの解析を行ったりすることによって, プログラムの理解が進められる．一方, 動的なアプローチでは, コンパイルされたプログラムを実行し, 実行時に生成される情報を詳細に調べることによって, プログラムの理解が進められる．実際の解析作業においては, ほとんどの場合両方のアプローチが組み合わされて用いられる．とりわけオブジェクト指向プログラムにおいては, 動的束縛などの処理機構によって呼び出されるメソッドが実行時に決定されるため, 動的なアプローチは必要不可欠である [6].

しかしながら, 一般に実行時に生成される情報は非常に膨大であり, その中から必要な情報を取り出すためには何度もプログラムの実行を余儀なくされる場合が少なくない．たとえば, デバッガを用いるとプログラムの実行を任意の時点で中断し, その時点での呼び出しスタックからアクセス可能なすべての変数の値を確認したり, ステップ実行によって制御の移動を 1 行ずつ追跡していくことも可能であるが, プログラムの過去の実行履歴は取得することができない．文献 [1] では, 不具合が発生した時点での呼び出しスタック中にその原因を特定できる情報が含まれていないケースが全体の約 5 割を占めることが報告されており, そのような場合, デバッガ上で不具合の発生を確認してもすでにその原因を特定するために有用な情報は失われているため, 不具合の原因を特定するために少なくとも 1 回以上のプログラムの再実行を余儀なくされる．

このような状況に対応するため, 本研究では Java プログラムを対象に, プログラムの実行のある時点で, あるオブジェクトがある変数によって参照されるようになった経緯を過去に遡って辿ることができる動的解析手法としてデルタ抽出を提案する．同様の手法としては, オブジェクトフロー [2], [3], [4] を挙げることができる．オブジェクトフローを用いると, 着目しているオブジェクトがどのような参照を経由して渡されてきたかという来歴を追跡することができる．ただし, オブジェクトフローで追跡することができるのはそのオブジェクトの流れに直接関与したコードのみであり, そのオブジェクトの流れを間接的に決定付けているコードについては追跡することができない．一方デルタ抽出を用いると, オブジェクトの流れを間接的に決定付けているコードも含めオブジェクトの流れに寄与しているコードを網羅的に辿ることができる．これにより, デバッグだけでなく以下のようなアクティビティも支援することが可能になると考えられる．

- アプリケーションフレームワークのサンプルアプリケーションを解析して, フレームワークの利用例を抽出する．具体的には, アプリケーション固有の機能を実

[1] 甲南大学大学院 自然科学研究科
Graduate School of Natural Science, Konan University
[a)] m1324011@center.konan-u.ac.jp
[b)] n-nitta@konan-u.ac.jp

装したオブジェクトがフレームワーク側の所定の位置で参照されるようにするために必要なアプリケーション側のコードを, サンプルアプリケーションの中から網羅的に抽出する (フレームワークの利用例の抽出).

- 既存のプログラムにおける様々な管理機構を理解する. 例えば, 特定のイベントに固有のオブジェクトがイベントに依存しないオブジェクトによってどのようにして参照されるようになったかを追跡することによって, そのプログラムのイベントの管理機構を理解することができる. 同様に, 特定のプラグイン固有のオブジェクトが, プラグインに依存しないオブジェクトに参照されるようになった経緯を追跡することによって, プラグインの管理機構を理解することができる (アーキテクチャ理解).

本稿では, プログラム理解の支援を目的としてデルタ抽出を行う Eclipse プラグインの実装を行った. さらにプラグインの有効性を評価するため, 技術者を対象としてプラグインを使用した場合と使用しなかった場合のプログラム理解の速度を比較する実証実験を行ったので, その結果について紹介する.

2. デルタ抽出

本研究では, プログラム理解を支援する動的解析手法としてデルタ抽出を提案している [7], [8]. 動的解析手法では主に実行中のプログラムから得られる情報が解析対象となるが, 一般にプログラムが生成する情報は全体で膨大な量になる傾向にある. そのため, それらの情報の中から有益な情報を抽出することによって, プログラム理解を支援する研究が行われている [2], [9], [10], [11]. オブジェクト指向プログラムの実行時に生成される情報の中で特に重要なものとして, オブジェクト間の参照に関する情報が挙げられる. デルタとは指定したオブジェクト間の参照がどのような経緯で生成されたかを表す実行時の構造である. デルタ抽出の概要について以下に説明する.

一般にオブジェクト指向プログラムの実行においては, ある場所で生成されたオブジェクトが別の場所に渡されることによってプログラム中の離れた場所で依存関係が発生する. たとえば図 1 に示したプログラムの Main.main() から始まる実行において, 21 行で生成されたクラス C のインスタンス (o とおく) は, 23, 17, 28 行を経由して 29 行に渡される. このとき o の流れには, 23, 17, 28 行に出現する同じ o への参照 (エイリアス) だけでなく, たとえばクラス B のインスタンスを参照している 11 行の b なども間接的に関与する. なぜならば, この b がクラス B の別のインスタンスを参照していた場合, 17 行の getC() メソッドによって取得される C のインスタンスも o とは別のものに変わり得るからである. そこで本研究では, プログラムの実行中の特定のオブジェクトの流れに関与した参照からなる構造をデルタとして定義し, このデルタを実行履歴の中から抽出することによってプログラム理解を支援することを考える. デルタの基本的な考え方は以下の通りである. 図 1 に示したプログラムを例に考える. このプログラムを実行すると, 図 2 のオブジェクト図で示したようなオブジェクトとその間のフィールドを介した参照 (オブジェクト間参照と呼ぶ) が生成される. 以下では説明の簡単のため, 各ク

```
 1: class Main {
 2:     A a = new A();
 3:     void main() {
 4:         a.m();
 5:     }
 6: }
 7: class A {
 8:     B b = new B();
 9:     D d = new D();
10:     void m() {
11:         d.passB(b);
12:     }
13: }
14: class D {
15:     E e = new E();
16:     void passB(B b) {
17:         e.setC(b.getC());
18:     }
19: }
20: class B {
21:     C c = new C();
22:     C getC() {
23:         return c;
24:     }
25: }
26: class E {
27:     C c = null;
28:     void setC(C c) {
29:         this.c = c;
30:     }
31: }
```

図 1 サンプルプログラム

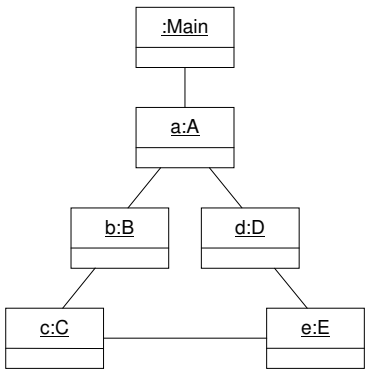

図 2 サンプルプログラムのオブジェクト図

ラスのインスタンスを図 2 中に示したオブジェクト名で呼ぶ．ここでは，オブジェクト c がオブジェクト e に渡された経緯，すなわち e から c へのフィールド c によるオブジェクト間参照が生成された経緯について考える．まず，c が e に渡されるためには，e と c の両方を “知っている” オブジェクトが少なくとも 1 つは必要である．このプログラムでは，オブジェクト a がそれに相当する．したがって e から c へのオブジェクト間参照 r は，a が c を e に “紹介” することによって生成されたとみなすことができる．このとき，a, c, e の間のオブジェクト間参照を結んで得られる構造を r の**デルタ**と呼び，a が c を e に紹介する際に行われたメソッド m() の実行を r の**コーディネータ**と呼ぶ．また，r から r のデルタを求めることを**デルタ抽出**と呼ぶ．直観的にある実行例において生成されたオブジェクト間参照 r のデルタは，その実行例において r が生成されるために必要な事前条件を表す．ここで，デルタがオブジェクト間参照を対象に抽出される一方で，抽出されたデルタもオブジェクト間参照によって構成される点に注意されたい．このことからデルタ抽出を再帰的に実行できることがわかる．デルタ抽出を繰り返すことによって，広い範囲のプログラムを効率的に探索できることが期待される．

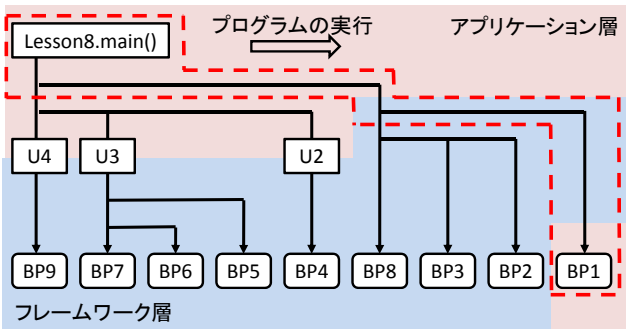

図 3 Lesson8.java の実行における呼び出し木

3. デルタ抽出を用いたプログラム理解

オブジェクト指向プログラムの理解において，プログラムの実行時に，あるオブジェクトがある変数によって参照されるようになった経緯を知りたくなる場合が多々ある．オブジェクトフロー [2] は，プログラムの実行時に生成された特定のオブジェクトへの参照の生成および複製を追跡することによって，そのオブジェクトがプログラム中のどこで生成されてどのように渡されてきたかを調べることができる動的解析手法である．しかしながら，オブジェクトフローは基本的に単独のオブジェクトのみに着目して来歴を調べる手法であるため，追跡したいオブジェクトがユーザの理解に応じて変化していくような解析には適さない．これに対しデルタ抽出では，指定したオブジェクト間参照の参照先オブジェクトだけでなく，その参照の生成に関わった他のオブジェクトに関する情報も同時に抽出することができる．そのためデルタ抽出を用いることによって，より広範囲の探索が必要とされるプログラム理解の支援が可能になると期待される．

実際のプログラム理解にデルタ抽出を適用した事例を用いて，デルタ抽出の上記性質が具体的にどのように機能するかについて説明する．この事例では，jMonkeyEngine [5] という 3D ゲームエンジンを用いて，あるゲームアプリケーションを実装できるか否かについて調査を行った．jMonkeyEngine は Java で書かれた 315KSLOC のオープンソースプログラムである．ゲームアプリケーションを実装する上では，物理的な性質を持つ物体が物理世界の中でどのようにして管理されているか，物体と物体の衝突イベントはどのようにして検出されるか，物体間で衝突が発生したときのイベント応答処理をどのようにして記述するか，jMonkeyEngine の中でイベントはどのようにして管理されているかといった情報を知る必要があった．そこでデバッガを用いて，jMonkeyEngine の物理演算サブシステムに含まれていたサンプルプログラム Lesson8.java を詳細に解析することで，上記情報についての調査を行った．調査は本研究室の大学院生 1 名によって，数週間の時間をかけて行われた．最終的に理解するまでに，合計で 60 箇所にブレークポイント，7 箇所にウォッチポイントを配置することになった．解析は基本的にプログラムの実行とは逆向きに進められた．そのため，ブレークポイントを配置する場所を変えながらプログラムを何度も実行する必要があった．この事例において実際に有効であったと判断されるブレークポイントは，**図 3** に示す BP1∼BP9 の 9 箇所である．図 3 は，Lesson8.java を実行して BP1 に至るまでの呼び出し木の一部を抜き出したものである．BP1 は解析に当たって最初に配置を行ったブレークポイントで，Lesson8.java 内に記述された衝突イベントに対する応答処理内に配置したものである．同等の解析をデルタ抽出を用いて行った結果を**図 4** に示す．この図ではオブジェクトを円で示し，a∼o までの識別子で各オブジェクトを区別している．また，オブジェクト間参照を矢印，コーディネータが実行されたオブジェクトを赤丸，静的メソッドの呼び出しを破線矢印でそれぞれ示している．Lesson8.java の実行が BP1 で停止したときの呼び出しスタック中のオブジェクト群から開始して，色付けしたオブジェクト間参照に対してデルタ抽出 (Δ_1∼Δ_8) を繰り返すことによって，デバッガによる手動の解析と同等の解析結果を得ることができた．

同様の解析をオブジェクトフローを用いて行うことを考える．BP1 の呼び出しスタックから開始して例えばオブジェクト d に対して追跡を行うとする．オブジェクトフローでは d に対する参照の由来を追跡することになるので，r_1 から開始して，Δ_1 中の a から d への参照を追跡することになる．ここで，Δ_1 中の r_3 や r_4 については，d への参

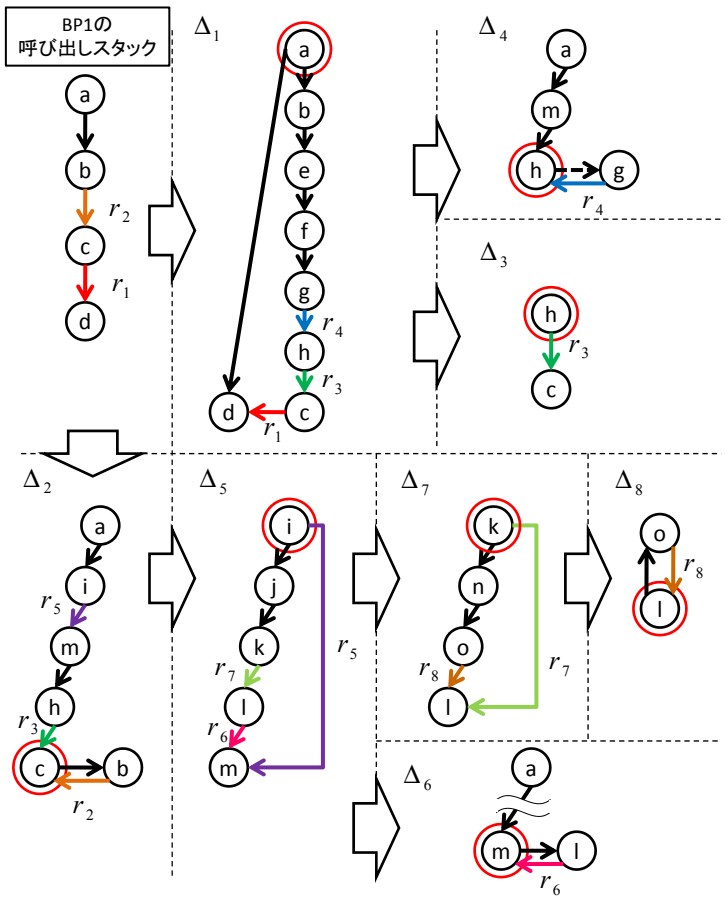

図 4　jMonkeyEngine のデルタ抽出を用いた解析例

照ではないため追跡の対象とならないことに注意が必要である．同様に c への参照を追跡する場合でも，r_2, r_3 と追跡できるが，r_5 以降は追跡の対象とならない．したがってこの事例においては，オブジェクトフローによる追跡のみでデバッガによる手動の解析と同等の解析結果を得られることはないことがわかる．このことから，デルタ抽出による解析はより広い範囲の探索が必要とされるプログラム理解に適しているといえる．

4. デルタ抽出 Eclipse プラグインの開発

4.1 プラグインの概要

本研究では，対象となるプログラムの実行履歴からデルタ抽出を行い，得られたデルタを基にプログラム理解の支援をすることを目指している．そのためにツールとして実装する機能として，

- デルタ抽出機能，
- デルタの可視化機能，
- 関連するソースコードの表示機能，
- 再帰的デルタ抽出機能，
- タブ表示機能，

の 5 つを考えた．これらの機能を統合開発環境である Eclipse プラットフォーム上で実現するために，Eclipse プラグインとしてツールの実装を行った．以下，それぞれの機能の詳細について説明する．また，プラグインを実行した画面を図 5 に示す．

4.2 デルタ抽出機能

デルタの抽出については文献 [8] のツールをそのまま利用した．初めに解析対象プログラムの実行履歴を記録したトレースファイルを選択し，抽出対象となるオブジェクト間参照の参照元と参照先のクラス名を入力することで，間接的にオブジェクト間参照を指定してデルタ抽出を行う．ここで，一般に実行履歴中には参照元クラス名と参照先クラス名が指定したものと一致するようなオブジェクト間参照は複数存在する．そのような場合は実行履歴中で一番最後に生成されたものを対象とする．なお，トレースファイルは文献 [8] のツールを解析対象プログラムに組み込み，そのプログラムを実行することによって予め作成しておくものとする．

4.3 デルタの可視化機能

文献 [8] のツールでは，デルタ抽出の結果は，コンソールにテキスト表示されるのみであったため，それらの情報からプログラムの動作を直感的に理解するのは難しかった．

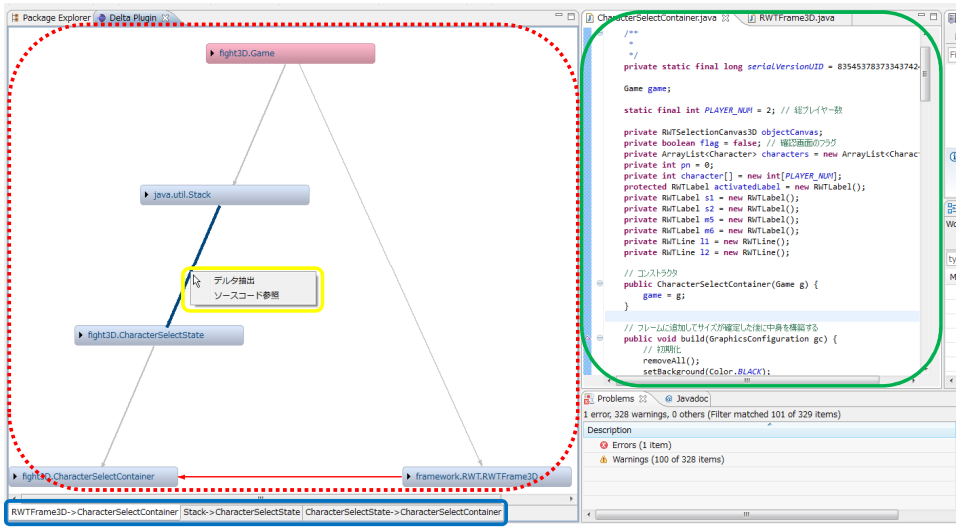

図 5 プラグインの実行画面

表 1 アンケート項目

問番号	アンケート内容
Q1	実験内容は実際の業務における困難性と合致していましたか？
Q2	マシンの環境（スペック等）はプログラムの読解作業を行う上で適切でしたか？
Q3	問題文はプログラムの理解を問うものとして適切でしたか？
Q4	デルタという概念は理解し易かったですか？
Q5	デルタ抽出についてより深く知りたいと思いましたか？
Q6	ツールの使い方は分かりましたか？
Q7	ツールを用いなかった場合のプログラム読解は難しかったですか？
Q8	ツールを用いた実験でデルタ抽出を使いましたか？
Q9	いつどこでデルタ抽出を使えばよいか判断できましたか？
Q10	デルタ抽出はプログラム読解において役に立ちそうですか？
Q11	もう一度このような実験を行う機会があれば参加してみたいですか？

そのためデルタ抽出によって抽出したデルタの構造を可視化する機能が必要と考えた．可視化においては Eclipse の視覚化ツールキットである Zest[*1] を用い，オブジェクトを頂点，オブジェクト間参照を有向辺とする三角形のグラフでデルタの構造を表現する．頂点は階層化されており，オブジェクトを表す頂点の内部にメソッド実行を表す頂点が含まれる．また，メソッド実行間の呼び出し関係は破線の有向辺で表現する．このような可視化により，プログラムの動作をより直感的に理解することができると考えられる．可視化したデルタは，図 5 の赤色の破線の枠で示した領域に表示される．

4.4 関連するソースコードの表示機能

一般にソースコードにはプログラムの動作の理解に必要な多様な情報が含まれており，可視化したデルタの構造だけでプログラムを理解することは困難であると考えられる．そこで，デルタの可視化と連携する形でソースコードを表示する機能を実装する．具体的には抽出したデルタの中の頂点であるオブジェクトを選択し，図 5 の黄色の枠で示したようなコンテキストメニューの”ソースコード参照”を選択することでそのクラスのソースコードを表示する．ソースコードは Eclipse プラットフォーム内の java エディタとして表示される (図 5 の緑色の枠で示した領域参照)．可視化されたデルタとソースコードの両方を参照することで，プログラム理解の支援に繋がると考えられる．

4.5 再帰的デルタ抽出機能

2 節で述べたように，デルタはオブジェクト間参照に対して抽出される．一方，抽出されたデルタはオブジェクト間参照によって構成される．したがってデルタの抽出は再帰的に実行することが可能である．具体的には，可視化されたグラフの中で，有向辺にあたるものがオブジェクト間参照であるので，個々の有向辺に対してもデルタ抽出が行えるように実装する．デルタ抽出機能と違い，この機能ではオブジェクト間参照を直接指定して，図 5 の黄色の枠で

*1 http://www.eclipse.org/gef/zest/

示したようなコンテキストメニューから”デルタ抽出”を選択することで, デルタ抽出を行うことが可能である. これにより, 再帰的なデルタ抽出を容易に行うことができる.

4.6 タブ表示機能

図 4 の解析過程からわかるように, 抽出した 1 つのデルタから複数のデルタを抽出する必要がある場合がある. その場合 1 度抽出したデルタを何度も参照する必要があるため, 抽出したすべてのデルタをタブとして残す機能を実装した. これにより図 4 の解析過程をすべて辿ることができる. タブを図 5 の青色の枠で示した領域に示す.

5. 実証実験

5.1 実験概要

デルタ抽出プラグインのプログラム理解に対する有効性を評価するために被験者を使った実験を行う. 具体的には, 本プラグインを用いた場合と用いない場合の被験者のプログラム理解の速度の比較を行う. 被験者を 2 つのグループに分け, 解析対象もそれに対し 2 つ用意する. 実験は午前の部, 午後の部に分け, 午前と午後でグループ毎に解析対象を入れ替えてプログラム理解を問う設問に解答してもらう. また, 午前の部と午後の部の合間にデルタについての簡単な講義を行う. 午前と午後の各部において, 2 つのグループに対し用意した 2 つの解析対象のうちそれぞれ別のものを配り, 午前は両グループともプラグインを用いずに解答してもらう. その後課題を入れ替えて, 午後は両グループともプラグインを用いて解答してもらう. 各設問に解答できた際にはその時刻を記録してもらい, すべての設問に解答できた場合, 正解による正誤判定を行う. 全問正解した時点で実験終了とする. 誤答があった場合は. 実験時間内であれば再度解答可能とする.

5.2 解析対象

対象アプリケーションは ArgoUML[*2]と, jEdit[*3]を用い, それぞれに対し設問 4 問を与えた. ArgoUML は, java で書かれたオープンソースの UML モデリングツールであり, プログラムのサイズは 186KLOC である. jEdit は, java で書かれたオープンソースのテキストエディタである. プログラムのサイズは, 117KLOC である. 設問内容は上記のプログラム理解を問うものとした. 設問と, それに対する解答, また対象プログラムに関する簡単な資料は事前に用意した. 用意した課題を図 6, 図 7 に示す. これらを午前の部と午後の部で入れ替えて実験を行った.

5.3 実験の実施

java による開発の実務経験が 1 年以上ある技術者 8 名を被験者とした. 8 名を 2 グループに分ける際, グループ毎にスキルが均等になるように実験前に課題テストを被験者に解いてもらった. 問題に正解し, テストを終えた順位でグループ分けを行った. 実験時間はそれぞれ, 午前の部と午後の部が 2 時間 30 分, 講義が 1 時間で実施した. マシンスペックは, OS:Windows 7 Professional, CPU:Intel(R) Xeon(R) 3.30GHz, メモリ:16.0GB, JVM build 1.7.0_15 である. また Eclipse は, Eclipse 4.2 juno の日本語化プラグイン環境を使用した. プラグインを用いた実験では対象プログラムの動的解析を行う必要がある. そのための解析対象のトレースファイルはあらかじめ用意しておいた. また, 実験開始時におけるヒントとしてブレークポイントの配置位置の指定と, ブレークポイントに至るまでの実行シナリオの説明を行った. その後実際にプログラムを実行し, ブレークポイントを入れた時点で停止することを確認後, 実験を開始するものとした. 実験終了後, 被験者すべてにアンケート調査を行った. アンケートの項目を表 1 に示す. これらの項目に対して, 5 段階のリッカート尺度で回答してもらった.

ArgoUML/GEF の主要クラス理解問題用紙

以下の [①] ～ [④] の空欄に当てはまるクラスまたはインタフェースを見つけてください。

[①] は、グラフィック文書を操作するための GEF の中核オブジェクトである。ユーザからの入力に対して実際に動作する複数のオブジェクトを保持しているが、自身はそれらのオブジェクトを連携させる以外、特別な動作は行わない。

[②] は、グラフィック文書に対するユーザからの入力を解釈し、適切な処理を行う。具体的にどのような処理を行うかは、具象クラスの種類によって異なる。

[③] は、(Fig クラスによって表現された) 図形の配置先オブジェクトで、透明のシートのような役割を持つ。

[④] は、グラフィック文書中のどのオブジェクトが次に実行される処理の対象となっているかを示す。

以上

図 6 ArgoUML の設問

jEdit の主要クラス理解問題用紙

以下の [①] ～ [④] の空欄に当てはまるクラスを見つけてください。

[①] は jEdit の全体のウィンドウに相当するクラスである。

[②] は jEdit の (TextArea クラスで表現された) テキスト編集エリア上で編集しているテキストの内容を表すクラスである。

[③] はテキスト編集エリア上で現在選択されているすべての選択領域を管理しているクラスである。

[④] は jEdit においてマウスのイベントを処理するクラスである。

図 7 jEdit の設問

5.4 実験結果

実験結果を表 2 に示す. また, アンケート集計結果を表 3 に示す. ここでは, プログラム理解の速度は制限時間内に

*2 http://argouml.tigris.org/
*3 http://www.jedit.org/

解答できた正解数の合計で評価するものとする.

表 2 実験結果

グループ		ArgoUML	jEdit
A	ツール使用	○	×
	正解数	4	5
B	ツール使用	×	○
	正解数	10	11

表 3 アンケート集計結果

内容	平均	標準偏差
Q1	3	1.195
Q2	3.5	1
Q3	3.375	0.992
Q4	3.75	0.433
Q5	4.25	0.661
Q6	3.125	1.053
Q7	3.75	0.829
Q8	2.625	1.218
Q9	1.75	0.661
Q10	3.5	0.5
Q11	3.625	0.992

表 2 から問題に対する正解数の合計は, ツールを用いた場合と用いなかった場合で変わらなかったことがわかる. またアンケートの集計結果から, 被験者達がデルタの概念は理解できていたが, 午後の部であまりツールを用いていないことがわかった.

6. 考察

結果としては, ツールを用いた場合と用いなかった場合で正解数の合計は変わらなかった. しかし, 実験後に行ったアンケートの結果から, 被験者達がツールをあまり用いなかったことと, その一方でデルタの概念は理解できていたことの 2 点がわかった. 前者は表 2 の Q8 の平均値が 2.625 と低かったことからわかる. 後者については表 2 の Q4 の平均値が 3.75 と高く, 標準偏差が 0.433 と低いことから明らかである. また, 被験者のスキルによって理解度に差が生まれるのかどうかを, 課題テスト結果の上位, 下位 4 名ずつに分けそれぞれの間で Q4 の値を比較する t 検定によって検証した. t 検定の値は 0.18 となり有意差が認められなかった. このことからスキルに関係なく, デルタの概念が被験者達に理解されていたと言える. しかし, 理解していたにもかかわらず被験者達はツールを利用していなかった.

この原因について, Q9 の結果を元に考察する. Q9 の平均値が 1.75 と低いことから被験者達がツールをどのように用いればよいかの判断がついていないことがわかる. この結果に対しても上記と同様の t 検定を行った結果, 値が 0.35 となり有意差が認められなかった. 以上のことからデルタの概念は理解できていたが, 実際にプログラムの解析に用いる際に, どのようにツールを使えば効率よく理解につながるか, また, 使わない場合と比べてどのような効果があるのかといった判断ができなかったことが推測される. また, そのような状況は被験者のスキルに依存しないこともわかった.

これらの原因として, ツールの実習不足が挙げられる. ツール自体の使い方の簡単な説明は講義の時間に行ったが, やはりそれだけで使いこなすのは難しいと思われる. 今回の実験は一日ですべての工程を行ったため, ツールの実習時間が取れなかった. 使い方がよくわからずにツールを用いようと試行錯誤したことで, ツールを用いない場合に比べて, 逆に時間がかかってしまった側面もあったかもしれない.

今後の課題としては, 被験者達のツールを用いた実習時間を十分に取り, 実験全体に 2 日以上の時間を費やして再度実験を行うことが挙げられる.

7. おわりに

プログラム理解の支援を目的として, 本研究室で提案している動的解析手法であるデルタ抽出を行うツールを, Eclipse プラグインとして開発した. また本ツールの有効性を確認するため, ツールを使った場合と使わない場合とでプログラムの理解に要する時間に差が生じるか否かについての評価実験を行った. しかしながら, 理解に要する時間に明確な差は認められなかった. 同時に行ったアンケート調査の結果から, ツールの実習時間が不足していたことが, 主な要因として考えられる.

今後の課題として, ツールの実習時間を十分に確保して再度実験を行うことが挙げられる. また, 本ツールが具体的にどのような種類のプログラム理解に適しているのかについての調査も続けていく必要がある.

謝辞

評価実験の実施に当たって多大なご協力を頂きました竹下春香さんを始め, 東京コンピュータサービス株式会社の皆様に深く感謝致します.

参考文献

[1] Liblit B., Naik M., Zheng A., Aiken A. and Jordan M.: Scalable Statistical Bug Isolation, *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 15–26 (2005).

[2] Lienhard, A., Ducasse, S., Gîrba, T. and Nierstrasz, O.: Capturing How Objects Flow at Runtime, *Proc. International Workshop on Program Comprehension through Dynamic Analysis*, pp. 39–43 (2006).

[3] Lienhard, A., Greevy, O. and Nierstrasz, O.: Tracking Objects to Detect Feature Dependencies, *Proc.*

15th International Conference on Program Comprehension, pp. 59–68 (2007).

[4] Lienhard, A., Gîrba, T. and Nierstrasz, O.: Practical Object-Oriented Back-in-Time Debugging, *Proc. 22nd European Conference on Object-Oriented Programming*, pp. 592–615 (2008).

[5] Mark Powell ら, jMonkeyEngine, 販売者無し (2003).

[6] Wilde, N. and Huitt, R.: Maintenance Support for Object Oriented Programs,

[7] 新田直也, 手塚裕輔: デルタ抽出: 実行履歴の大域的構造を効率良く把握するための抽象化手法, 情報処理学会研究報告, 2011-SE-173(6) (2011).

[8] 山根敬史, 新田直也: デルタ抽出のための効率の良いアルゴリズムの開発と実装, 電子情報通信学会信学技報, SS2011-31 (2011).

[9] Sridharan, M., Fink, S. J. and Bodík, R.:Thin slicing, *Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 112–122 (2007).

[10] Hamou-Lhadj, A. and Lethbridge, T.: Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system, *Proc. International Conference on Program Comprehension (ICPC)*, pp. 181–190 (2006).

[11] Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., Wijk, J. J. van and Deursen, A. van Understanding execution traces using massive sequence and circular bundle views, *Proc. International Conference on Program Comprehension (ICPC)*, pp. 49–58, (2007).