Vol.2016-SE-191 No.29
2016/3/15

デバッグ環境にオブジェクト図を提示するEclipseプラグインの開発

久保田 吉彦[1,a)]　山崎 翔[1,b)]　紫合 治[1,c)]

概要：プログラム開発に統合開発環境 (以下，IDE) を用いた場合，デバッグ時には IDE の提供するデバッガを用いる．ユーザは Graphical User Interface(以下，GUI) 上でブレークポイントを設定し，デバッグ実行を行ない，ブレークポイントで実行を中断させプログラムの確認をする．ユーザは変数上にマウスカーソルを移動させることによる変数の値の確認や，変数の値を階層的に表現した GUI を展開しながら変数の値を確認する．これら GUI を操作しながら値を確認する作業は，プログラム実行中に作成されたオブジェクトの参照関係やオジェクトのフィールド変数の値など全体を俯瞰できるような視点をユーザに与えていない．本稿ではデバッグ実行時に IDE にオブジェクト図を表示する View を追加し，ブレークポイントの設定・解除とステップ実行を可能としたまま，ブレークポイント到達時にオブジェクト図を提示するプラグインの開発について述べる．

キーワード：デバッグ，オブジェクト図，Eclipse プラグイン

Development of Eclipse Plug-in to present an Object Diagram to Debug Environment

Kubota Yoshihiko[1,a)]　Yamazaki Sho[1,b)]　Shigo Osamu[1,c)]

Abstract: In Eclipse Debug Environment, it is difficult that a user understand relationships between objects. The debug environment has Variables View that is hierarchical structure. The user must expand it manually for confirming variable's value. By using object diagram, the user can easily understand relationships between objects. In this paper, we propose to use object diagram in the debug environment. When a program execution is stopped at a breakpoint, our plug-in presents its object diagram.

Keywords: Debug Environment, Object Diagram, Eclipse Plug-in

1. はじめに

プログラマは，プログラムを実行させ想定外のプログラムの動作や変数の異常値を観測し，バグが混在している箇所を推定する．ブレークポイントの設定とステップ実行で変数の値やオブジェクトの参照の変化を IDE の提供する GUI により確認する．

Eclipse の提供するデバッグ環境はブレークポイントで実行を停止させた後，変数の値やオブジェクトの参照をたどるといった詳細の検証用途に向いているが，実行を停止させた時に存在するオブジェクトとオブジェクト間の関係を俯瞰する視点を与えていない．

本稿では，Eclipse のデバッガ機能を保持したまま，ブレークポイントで実行が停止された時点のオブジェクトの状態とオブジェクト間の関係をオブジェクト図として表示し，デバッガにより変数の値を変更した際にもオブジェクト図に反映するプラグインの実装について述べる．

[1] 東京電機大学
Tokyo Denki University, Inzai, Chiba 270–1382, Japan
[a)] y.kubota@mail.dendai.ac.jp
[b)] 15jkm22@ms.dendai.ac.jp
[c)] shigo@mail.dendai.ac.jp

2. 関連研究

プログラムの動作を視覚化する先行研究について述べる．

オブジェクト図を IDE に統合する研究として，JIVE[1][2] は Eclipse のデバッガに機能を追加し，Java プログラムの実行状況を拡張したオブジェクト図やシーケンス図をプログラムの進行と共に図を変化させ，ユーザに提示することができる (図 1)．また，変数の値やオブジェクトの関係等を保存しておき，プログラムを中断させ過去の変数の値，オブジェクト図，シーケンス図を確認することが出来る機能を有している．

しかし，多数のオブジェクトが出現するプログラムには適用が困難となる．プログラムの実行開始から Java Platform Debugger Architecture のイベントを取得し，逐次情報を抽出し蓄積するため，オーバーヘッドが性能に大きく影響する．多数のオブジェクトを表示する場合，オブジェクト図やシーケンス図の更新が遅くなり適用が困難となる [5]．

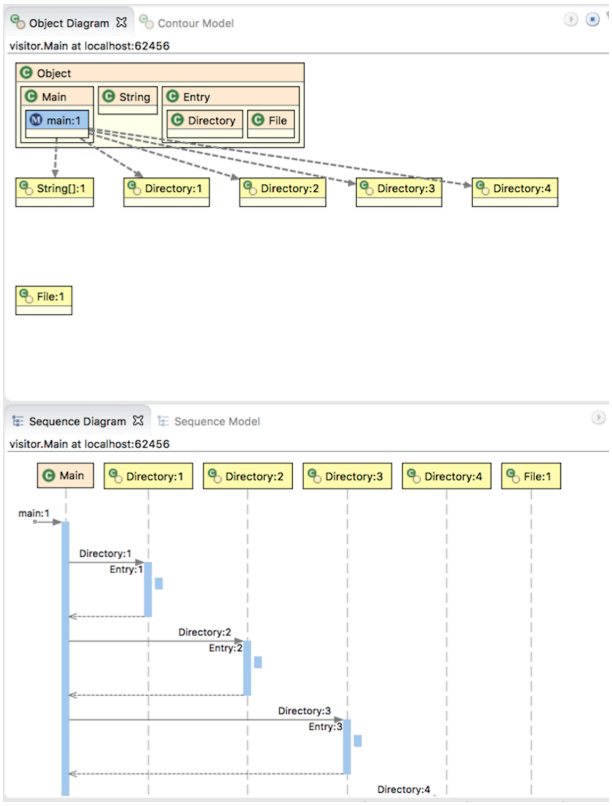

図 1 JIVE のオブジェクト図とシーケンス図

山崎らは，動作中の Java プログラムを拡張したオブジェクト図として表現し，プログラムの進行と共にアニメーションとしてオブジェクト図を変化させるシステムを開発した [3]．しかし，IDE との統合を行なっていないためブレークポイントの設定やステップ実行の指示が困難である．

3. 提案手法

本システムでは Eclipse のデバッガの機能の拡張としてオブジェクト図を表示させ，デバッガの GUI だけでは把握の難しいオブジェクト間の関係を俯瞰できるオブジェクト図を追加する．JIVE との違いとして，本システムではプログラム実行開始時からの実行情報は蓄積せず，次の 2 つのイベント発生時に情報を収集しオブジェクト図の表示・更新を行なう．

- ブレークポイント到達時
- ステップ実行時

プログラム実行時にはシステム関連のクラスのオブジェクトも大量に生成されているため，対象となるプログラムのパッケージ名によるフィルタリングも行なう．デバッグ実行開始からブレークポイントに到達するまでは通常のデバッグ実行と同様の振舞いをし，ブレークポイントに到達すると，対象となるクラスのオブジェクトを全て収集する．その後，オブジェクトのフィールドの値の取得とオブジェクトを参照しているオブジェクトを再帰的に収集する．

3.1 準備

本システムでは，Eclipse のデバッガの設定画面で通常のデバッガとオブジェクト図付きデバッガのどちらを使用するか選択出来るようにしている (図 2)．ユーザが選択することでオブジェクト図付きデバッガが起動する．

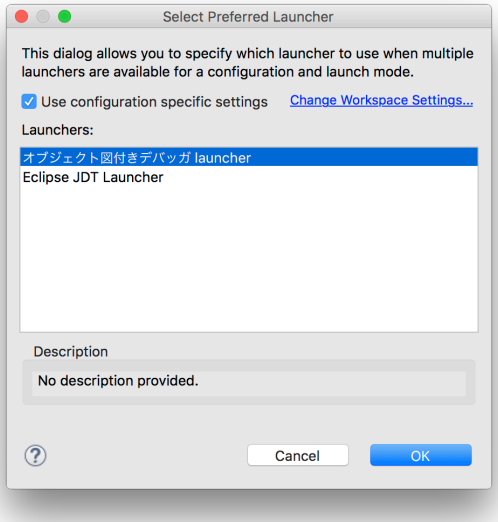

図 2 デバッガの切り替え画面

3.2 オブジェクト図の表記

本システムでのオブジェクトの表記について述べる．UML のオブジェクト図を元に変更を行なっている (図 3)．変更は次の通りである．

- オブジェクト図の上段は ’id 番号 : クラス名’ の形式とする．オブジェクトの id 番号は Eclipse のデバッガが示すオブジェクトの id 番号と同値となるように ObjectReference.uniqueId() メソッドから返される値を表示する
- 下段のフィールド欄は’ フィールド名:型 = 値’ の形式とする．フィールドが複数ある場合は改行して表示する．型が基本型の場合はその値を表示し，オブジェクトへの参照の場合はオブジェクトの id 番号を山括弧で囲って表示する
- 参照先のオブジェクトがある場合，そのオブジェクトへ参照を示す矢印が表示される

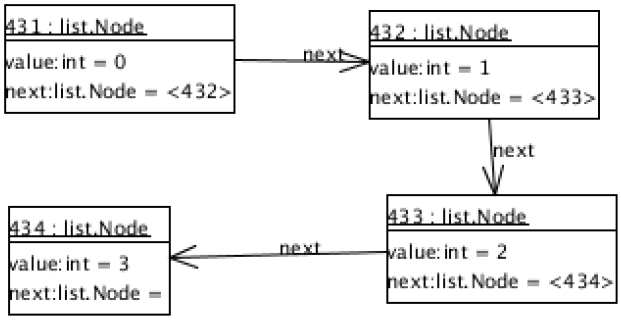

図 3 本システムのオブジェクト表現の一例

4. システムの実現

4.1 オブジェクトの収集

Eclipse での Java プログラム開発は Java Development Tools(以下，JDT) が用いられており，JDIDebugTarget を拡張することでブレークポイント到達時やステップ実行の際に発生するイベントを取得できる．オブジェクト図を作成する手順は次の通りである．

(1) 設定したブレークポイントにデバッグ実行が到達した際に発生するイベントを補足する．
(2) デバッグ対象の Java 仮想機械から，ユーザが設定したフィルタを介し必要なクラス群を取得する．
(3) 取得した各クラスのオブジェクトを全て取得する．取得には JDI の ReferenceType 型のオブジェクトへ instances(0) メソッドを使用する．取得後，id 番号をキーとして HashMap に保存する．
(4) 取得したオブジェクトを参照しているオブジェクトを再帰的に取得する．JDI の ObjectReference 型オブジェクトへ referringObjects(0) メソッドを再帰的に使用する．参照元が取得出来た場合，各参照元オブジェクトを id 番号をキーとして HashMap に保存する．
(5) 取得したオブジェクトの id 番号と参照元オブジェクトの id 番号からオブジェクト図を描画する．

4.2 オブジェクト図の具体例

本システムを使用して得られるオブジェクト図をサンプルプログラムを使用して提示する．サンプルプログラムは以下の 3 つのファイルからなる．

- Main.java(図 4)
- Class1.java(図 5)
- Class2.java(図 6)

Class1 はコンストラクタで Class2 のオブジェクトを引数として与えると，フィールド value に代入する．Class2 はコンストラクタで int 型の整数と String 型の文字列を与えると，フィールド value に整数が，name に文字列が代入される．ここで，Main(図 4) の 6 行目にブレークポイントを設定し，デバッグ実行がブレークポイントに到達した時点で描画されるオブジェクト図が図 7 右部である．

```
1  package sample;
2
3  public class Main {
4    public static void main(String[] args) {
5      Class1 c1 = new Class1(new Class2(1, "c1"));
6      Class1 c2 = new Class1(new Class2(2, "c2"));
7    }
8  }
```

図 4 Main.java

```
1  package sample;
2
3  public class Class1 {
4    private Class2 value;
5    public Class1(Class2 value){
6      this.value = value;
7    }
8    public Class2 getValue(){
9      return this.value;
10   }
11 }
```

図 5 Class1.java

```
1  package sample;
2
3  public class Class2 {
4    int value;
5    String name;
6    public Class2(int value, String name){
7      this.value = value;
8      this.name = name;
9    }
10   public int getValue(){
11     return this.value;
12   }
13   public String getName(){
14     return this.name;
15   }
16 }
```

図 6 Class2.java

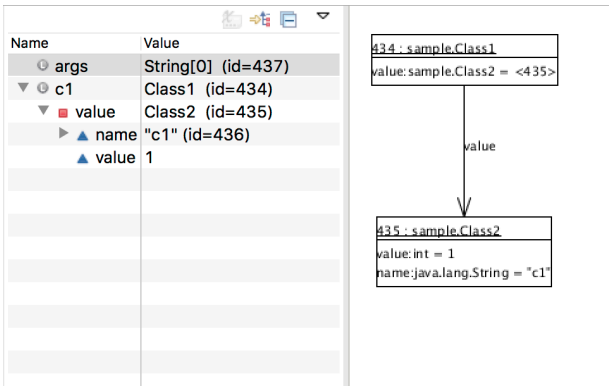

図 7 システムの動作例

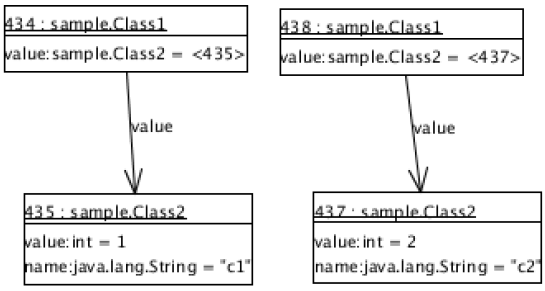

図 8 ステップ実行後の本システムのオブジェクト図

ブレークポイントに到達すると，その時点の変数の値を参照することができる．図 7 左部は Eclipse が提供する変数の値を確認することができる Variables View である．局所変数 c1 が参照しているオブジェクト (id=434) を展開すると変数 value は Class2 のオブジェクト (id=435) を参照していることがわかる．変数 value を展開すると Class2 のオブジェクトのフィールドが持つ変数とその値が表示される．変数の値を確認するために GUI を操作することによって順次展開されて変数の値が表示される．そのためプログラマがオブジェクトの参照関係を即座に把握することが困難である．

図 7 右部は今回開発したオブジェクト図を表示するプラグインである．Variables View のように GUI を操作することなくオブジェクトの関係を把握することができる．ステップ実行することによりオブジェクト図も変化していく．ステップ実行を行なった後は図 8 となる．

JIVE では Variables View で変数の値を変更してもオブジェクト図に反映されない．本システムでは変数の値を変更した場合，オブジェクト図に反映される．

例として，Eclipse の Variables View から id 番号が 435 のオブジェクトのフィールド value の値を 999 に変更した場合，オブジェクト図の表示も更新される (図 9)．

複数のオブジェクトを描画する図となる場合，常にフィールドの表示を行なうと全体の把握が困難となる．ユーザの操作によってフィールドの表示/非表示を切り替えることを可能とし，オブジェクトの参照関係のみを表示することもできる (図 14)．

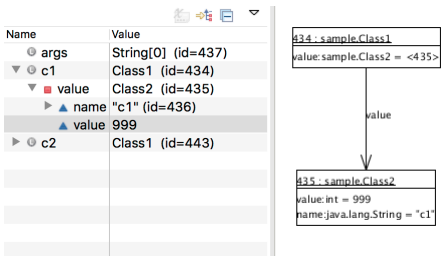

図 9 変数の値を変更

4.3 ArrayList への対応

本システムでは，オブジェクトの参照関係の収集の方式により，JIVE が対応していない ArrayList といった集合オブジェクトから参照されているオブジェクトへも参照の矢印が描画される．

例として，複数のオブジェクトが生成される Visitor パターン [4] で JIVE とのオブジェクト図の比較を行なう．このプログラムはディレクトリとファイルの構造を表現したものとなっている．

ブレークポイントを設置し実行した結果が図 10 であ

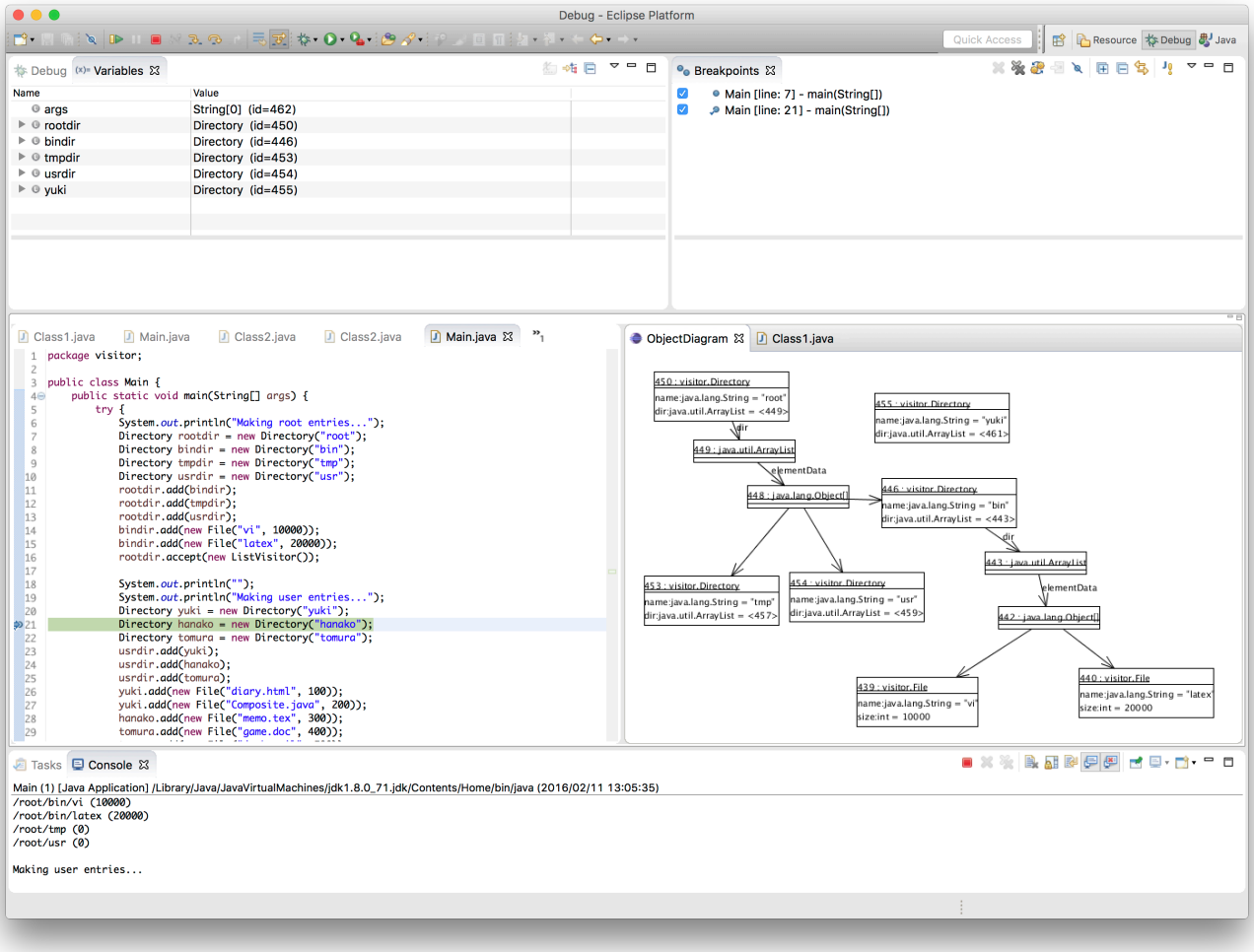

図 10 本システムでの集合オブジェクトの表示例

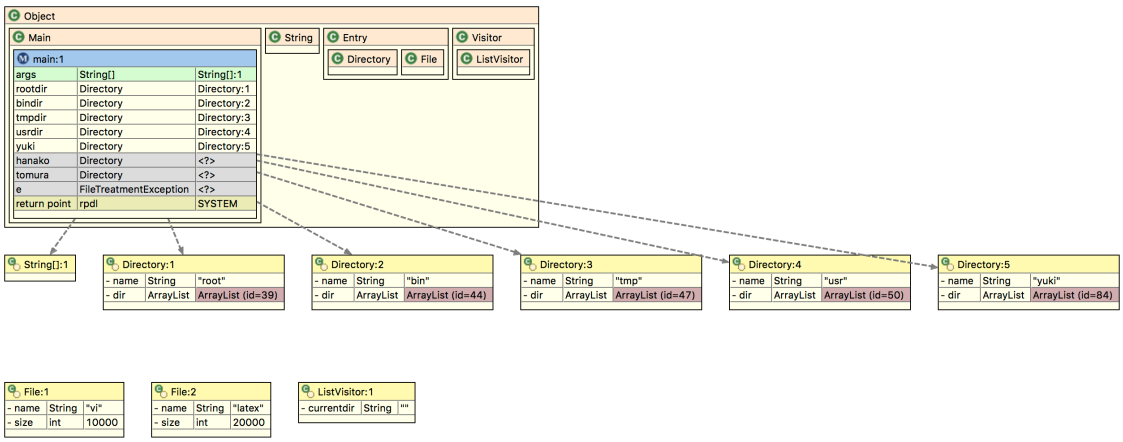

図 11 JIVE で同一プログラムを動かした時のオブジェクト図

る．本システムで実行した場合，Directory オブジェクトが参照している ArrayList と，ArrayList が参照している Directory オブジェクトや File オブジェクトを表示することができる．

図 11 は JIVE で同一のブレークポイントまで実行させた時の図である．JIVE はオブジェクト図が表示する情報量

の選択が出来る．図 11 は詳細に表示される’Stacked with Tables’ を選択して表示される図である．

Directory オブジェクトが 5 つ，File オブジェクトが 2 つ作成されていることはわかるが，File オブジェクトが Directory オブジェクトから参照されていることが図からはわからない．

5. 評価

本システムでは，デバッグ実行開始からの実行情報の蓄積ではなく，ブレークポイントでプログラムが中断した時点でのオブジェクトのフィールド値とオブジェクトへの参照を収集している．JDI のイベント発生毎に情報を収集するオーバーヘッドを避け，即座にプログラマにオブジェクト図を提示するためである．本システムと JIVE を用い 2 つの比較実験を行なった．

(1) オブジェクトの個数によるブレークポイント到達時間の比較
(2) 単純な GUI プログラムを対象としたブレークポイント到達時間の比較

使用したコンピュータは Mac Book Pro 15’ ,Intel Core i7 プロセッサ 2.8Ghz，メモリ 16G，JDK のバージョンは 1.8.0_73 である．

5.1 オブジェクトの個数によるブレークポイントまでの到達時間の比較

デバッグ実行をした場合のブレークポイントまでの到達時間と，本システムと JIVE を使用した場合のブレークポイントまでの到達時間をベンチマークプログラム (図 12) を作成し，生成するオブジェクトの個数 (N の値) を変化させ計測した．

図 12 の 14 行目にブレークポイントを設定し，オブジェクト図が描画されるまでの時間を計測した．実行時間は各 5 回実行し平均したものである．(表 1).

結果として，ブレークポイントに到達するまでの時間は通常のデバッグ実行に対し本システムも JIVE も生成するオブジェクトの個数により線形に増加する．

本システムを使用した場合，ブレークポイントまでの到達時間は，デバッグ実行のみの場合と比較すると約 1.1〜1.5 倍となった．JIVE の場合，デバッグ実行のみの場合と比較すると約 10000〜15000 倍の時間が必要となる．

本システムではブレークポイントに到達するまでの時間は JIVE より短かいが，オブジェクト図を表示するまで JIVE より多くの時間が必要となる．

5.2 単純な GUI プログラムを対象としたブレークポイントまでの到達時間の比較

GUI を持つプログラムは単純なプログラムであっても内部では複数のオブジェクトで構成されている．図 13 のような JFrame,JPanel,JButton 各 1 つのオブジェクトを生成し表示されるだけのプログラム (図 15) を作成し，17 行にブレークポイントを設定し，ブレークポイント到達までの時間を計測した．

```
1  package benchmark;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class Main {
7
8    public static void main(String[] args) {
9      List<Main> list = new ArrayList<>();
10         Long start = System.nanoTime();
11         for(int i=0; i<N;i++)
12       list.add(new Main());
13           System.out.println(System.nanoTime()−start);
14           System.out.println("end");
15   }
16 }
```

図 12 ベンチマークプログラム

表 1 実行結果

N	デバッグ実行 (ms)	JIVE(ms)	本システム (ms)
100	0.07	703.07	0.10
200	0.11	1498.73	0.17 + 描画に約 1 秒
500	0.25	3962.14	0.30 + 描画に約 3 秒
1000	0.47	6837.75	0.52 + 描画に約 10 秒

5 回実行しブレークポイント到達までの時間を平均した．デバッグ実行では約 2.7ms となり，本システムでは約 3.1ms 程度でブレークポイントまで到達しオブジェクト図 (図 14) を表示した．JIVE ではデバッグ実行開始からのオブジェクト図の変化やシーケンス図の変化は観測できるものの 1 分を経過してもブレークポイントには到達しなかった．

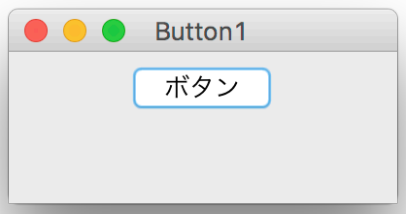

図 13 実験に使用した GUI プログラム

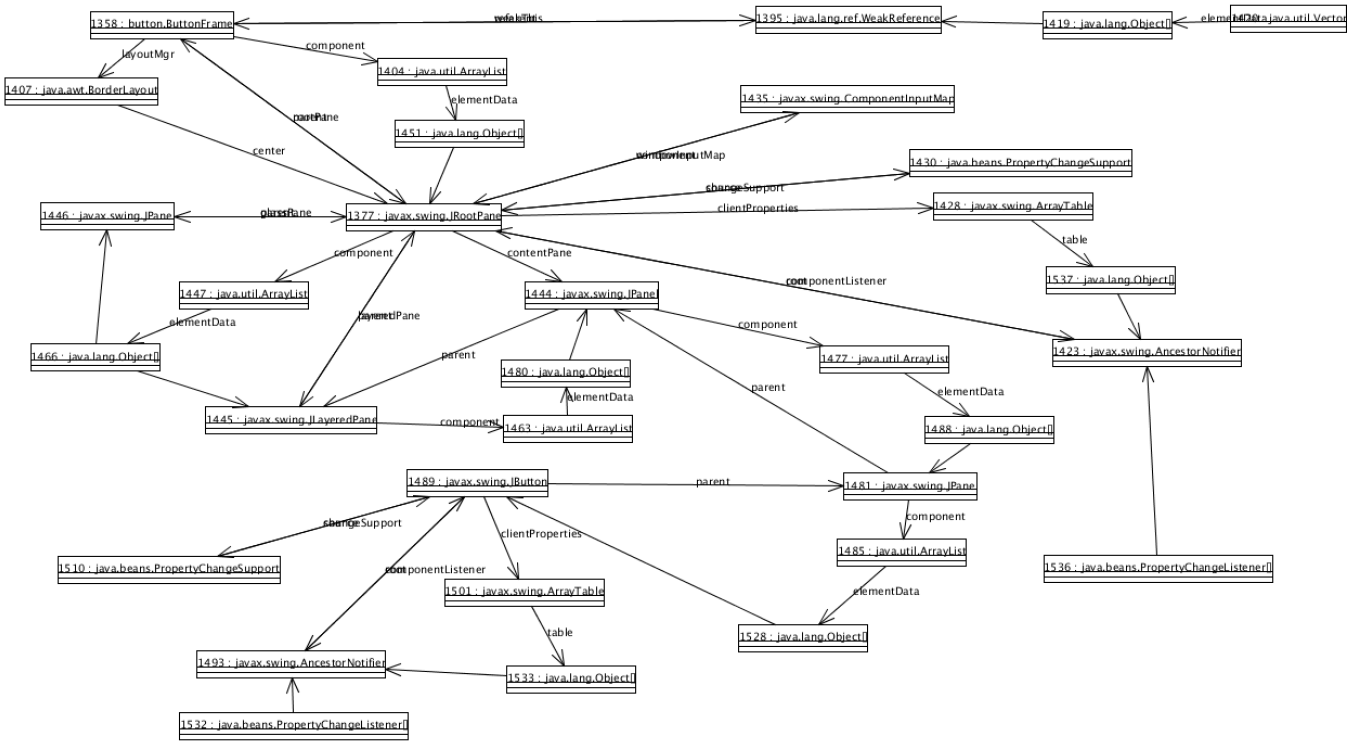

図 14 GUI プログラムを実行して得られたオブジェクト図

```
1  package button;
2
3  import javax.swing.JButton;
4  import javax.swing.JFrame;
5  import javax.swing.JPanel;
6
7  public class ButtonFrame extends JFrame{
8    public ButtonFrame(){
9      this.setTitle("Button1");
10     this.setSize(200, 100);
11     this.setDefaultCloseOperation(
12       JFrame.EXIT_ON_CLOSE);
13         JPanel panel = new JPanel();
14         JButton button = new JButton("ボタン");
15         panel.add(button);
16         this.getContentPane().add(panel);
17         this.setVisible(true);
18   }
19
20   public static void main(String[] args){
21         new ButtonFrame();
22   }
23 }
```

図 15 GUI を表示するプログラム

6. まとめ

Eclipse のデバッグ環境にオブジェクト図を表示するプラグインを開発した．JIVE が対応していない ArrayList 等の集合オブジェクトが参照するオブジェクトも図示することができる．

本システムではブレークポイント到達時にその時点のオブジェクトの情報を取得しオブジェクト図を提示する．プログラム実行開始からの情報を蓄積しないため，JIVE のように過去の変数の値やオブジェクトの参照関係を見ることはできないが，プログラムの実行時間にほとんど影響を及ぼさずにブレークポイントまで到達し，オブジェクト図を提示することができる．

しかし，JIVE よりオブジェクト図が表示されるまでの時間を要する問題がある．また，JIVE はオブジェクト図の自動レイアウト機能を有しているが本システムでは実現できていない．

評価として実行時間だけを比較しており，ユーザが本システムを利用した場合のデバッグ作業やプログラム把握がどれだけ効率化されたかを評価していない，これも今後の課題である．

本システムは Java 仮想機械に依存しており，JDI

の VirtualMachine.canGetInstanceInfo() が true を返すことを前提として作成されている．その為，クラスのオブジェクト一覧やオブジェクトの参照元といった情報が，ReferenceType.instances(long) や ObjectReference.referringObjects(long) を使用して，常に取得できるとは限らない．

現在，Java 仮想機械に依存せずクラスのオブジェクトを取得する方法としてクラスのバイトコード変換を行ない，コンストラクタで作成されたオブジェクトを保存しておく方法を検討している．その際，ガベージコレクタに影響を与えないよう弱参照を扱うコレクションを使用する予定である．

参考文献

[1] Czyz, Jeffrey K., and Bharat Jayaraman. ”Declarative and visual debugging in eclipse.” Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange. ACM, 2007.
[2] JIVE: Java Interactive Visualization Environment http://www.cse.buffalo.edu/jive/
[3] 山崎翔, 久保田吉彦, and 紫合治. ”オブジェクト図のアニメーション.” ソフトウェアエンジニアリングシンポジウム 2015 論文集 2015 (2015): 129-136.
[4] 結城浩. 増補改訂版 Java 言語で学ぶデザインパターン入門. SB クリエイティブ, 2004.
[5] Alsallakh, Bilal, et al. ”Visual tracing for the eclipse java debugger.” Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on. IEEE, 2012.