平成22年度 学士論文

プログラムの織り込み関係を可視化するアウトラインビューの提案と実装

東京工業大学 理学部 情報科学科
学籍番号 07-0510-8
大谷 晃司

指導教員
千葉 滋 教授

平成23年2月7日

概要

今日、ソフトウェア開発を行う上で、オブジェクト指向技術は必要不可欠である。オブジェクト指向言語により、ソフトウェアをモジュール化し、関心事を分離する技術は十分に成功したと言える。しかし、より複雑で多様化するアプリケーションに対しても、関心事の分離が達成されているかというと、決してそうとは言えない。分離できずに各ソースコードに散在してしまう関心事がソースコードの質を劣化させてしまっている。そのような例として、ロギング処理や図形エディタの再描画処理などが挙げられる。このようなモジュール間をまたがってしまっている関心事を横断的関心事と呼ぶ。横断的関心事に対して変更を行う際には、複数のモジュール間を横断的に編集しなければならない。このような横断的関心事を別のモジュールにひとまとめにして扱うことの出来る技術として、アスペクト指向がある。アスペクト指向を用いると、横断的関心事をモジュールに分離出来る為、一箇所にまとめて編集を行う事が出来る。アスペクト指向言語の一つとして、GluonJ がある。GluonJ は Java に最小限の拡張を行うことでアスペクト指向言語を実現した言語である。GluonJ では、Java のクラスを使って横断的関心事を別のモジュールに分離させる事が出来る。

しかし、ここで問題となるのは、アスペクト指向を用いて横断的関心事を別のモジュールへと分離させた場合、分離させる前のソースコードからは横断的関心事に関する記述が完全に消えてしまう事である。すなわち、クラスのコードを読んでも、そのクラスに対してどのようなプログラムが織り込まれるかと言う情報を知ることが出来ない。もしその情報を知りたい場合、アスペクト指向を用いて分離させたモジュールのソースコードを実際に読み、該当する箇所を調べる必要がある。これは大変労力のいる作業であり、効率を悪くする原因となる。

そこで、本研究では GluonJ に適応する為のツールとして、プログラムの織り込み関係の情報を表示するアウトラインビューを Eclipse のプラグインとして開発した。プログラムの織り込みを行うモジュール単位であるメソッドを、織り込みが行われるメソッドをトップに、織り込みを行うメソッドの順序に従って階層的に表示を行う。これにより、織り込みが行われるクラスから、どのようなプログラムが織り込まれるかと言う情報を、視覚的に提供する事が出来る。また、開発者は表示されている要素をク

リックする事で、該当するクラスのファイルを開き、該当する行へとジャンプする事が出来る為、織り込みの関係を探索する手間を大幅に削減することが出来る。

また、アスペクト指向での開発を支援するツールとして、AJDT(AspectJ Development Tools) がある。AJDT は AspectJ での開発を支援するツールである。AspectJ では、クラス内に散在している横断的関心事を、アスペクトを定義する事で別のモジュールへと分離している。その為、AJDT ではクラスと織り込みを行うアスペクトとでそれぞれ対応するビューを提供している。一方、本研究ではビューの表示を横断的関心事をモジュールとしてまとめたクラスと、プログラムの織り込みが行われるクラスとで、全く同じ情報を提供している為、AJDT とは異なっている。

謝辞

本研究を進めるにあたり、研究の方針や論文の組み立て方について数々の助言を頂いた指導教員の千葉滋教授に心より感謝致します。また、本研究における仕様や実装方法などの点について指導して頂いた Salikh Zakirov 氏に心から感謝致します。最後に、研究活動を共に行い、多くの助言を頂いた千葉研究室の皆様方に感謝致します。

目 次

図 目 次

表 目 次

第1章　はじめに

今日、ソフトウェア開発を行う上で、オブジェクト指向技術は必要不可欠である。オブジェクト指向言語により、ソフトウェアをモジュール化し、関心事を分離する技術は十分に成功したと言える。しかし、より複雑で多様化するアプリケーションに対しても、関心事の分離が達成されているかというと、決してそうとは言えない。分離できずに各ソースコードに散在してしまう関心事がソースコードの質を劣化させてしまっている。そのような例として、ロギング処理や図形エディタの再描画処理などが挙げられる。このような関心事はプログラム全体に散在して互いにもつれあった状態になってしまっている為、このような関心事が存在しているソースコードはモジュールとしての性質に欠けていると言える。このような関心事を横断的関心事と呼ぶ。本来は同じ関心事に関連するコードは一箇所にまとめて編集を行える事が理想であるが、このような横断的関心事に対して変更を行う際には、複数のモジュール間を横断的に編集しなければならない。このような横断的関心事を別のモジュールにひとまとめにして扱うことの出来る技術として、アスペクト指向がある。アスペクト指向を用いると、横断的関心事をモジュールに分離出来る為、一箇所にまとめて編集を行う事が出来る。

アスペクト指向言語の例として AspectJ が挙げられる。AspectJ はオブジェクト指向言語である Java 言語を拡張する事でアスペクト指向を実現した言語である。AspectJ では、クラス内に散在している横断的関心事を、アスペクトという新たな機構を用いて別のモジュールへと分離している。しかし、AspectJ はクラスではモジュール化する事の出来ない横断的関心事をアスペクトに分離する為、クラスとアスペクトを開発者が上手く使い分ける必要が生じる。一方、同じ Java 言語の拡張で、アスペクト指向を実現した GluonJ では、クラスとアスペクトと言った使い分けを行う必要がない。GluonJ は Java に最小限の拡張を行う事でアスペクト指向を実現している為、横断的関心事もクラスで分離させる事が出来る。しかし、アスペクト指向を用いて横断的関心事を別のモジュールへと分離させた場合、分離させる前のソースコードからは横断的関心事に関するコードが完全に消えてしまう為、分離させた横断的関心事がどこに適用されるかを、元のソースコードの側から知る方法が存在しない。もしその情報を知りたい場

合、アスペクト指向を用いて分離させたモジュールのソースコードを実際に読んで該当する箇所を調べる必要がある。これは大変労力のいる作業であり、効率を悪くする原因となる。このような問題に対し、AspectJ での開発を支援するツールとして AJDT(AspectJ Development Tools) が存在する。しかし、AspectJ はクラスとアスペクトを使い分けてアスペクト指向を実現している為、クラスのみでアスペクト指向を実現している GluonJ には上手く適用する事が出来ない。

そこで、本研究では GluonJ に適応する為の、プログラムの織り込み関係の情報を表示するアウトラインビューを Eclipse のプラグインとして開発した。プログラムの織り込みを行うモジュール単位であるメソッドを、織り込みが行われるメソッドをトップに、織り込みを行うメソッドの順序に従って階層的に表示を行う。横断的関心事をモジュールとしてまとめたクラスと、横断的関心事が分離させられたクラスとで、全く同じ情報を提供する点が AJDT とは異なっている。

本稿の残りは、次のような構成からなっている。第 2 章はアスペクト指向を用いたコードの具体例と、プログラムの織り込み関係を表示する為の支援ツールの必要性を述べる。第 3 章では、本システムの設計、第 4 章では、本システムのこれまでの実装、第 5 章では、既存のツールとの比較、そして第 6 章でまとめを述べる。

第2章　織り込み関係を表示するビューの提案

この章では、モジュール化出来ない横断的関心事の例を基に、アスペクト指向言語による解決法を述べる。その上で、アスペクト指向言語開発を支援するツールの例とその問題点について述べる。

2.1　アスペクト指向言語

オブジェクト指向言語により、ソフトウェアをモジュール化し、関心事を分離する技術は十分に成功したと言える。しかし、より複雑で多様化するアプリケーションに対しても関心事の分離が達成されているかというと決してそうとは言えない。分離出来ずに各ソースコードに散在する関心事が、ソースコードの質を劣化させていると言える。そのような例として、ロギング処理や図形エディタの再描画処理などが挙げられる。関心事がプログラム全体に散在して互いにもつれあった状態になっている為、このような関心事が存在しているソースコードはモジュールとしての性質に欠けていると言える。このような関心事を横断的関心事と呼ぶ。アスペクト指向言語とは、横断的関心事を別のモジュールにひとまとめにして扱うことの出来る技術である。オブジェクト指向言語でもかなり高度なモジュール化を行うことが出来るが、モジュール同士のつなぎ換えを行おうとした時、プログラムの随所を変更する必要が出てくる。これはモジュール同士の結合の度合いが高いためである。アスペクト指向言語とは、モジュールの結合を緩め、再利用性を高めた技術と言える。以下に、オブジェクト指向言語ではモジュール化する事が出来ない横断的関心事の具体例を挙げる。

オブジェクト指向言語である Java を用いて、図形エディタを実装する事を考える。丸や四角といった図形を表すクラスを用意する。図形クラスが持つ機能は、図形全てが持つ機能とその図形独自の機能とで大きく二つに分かれる。このような場合、前者の機能を図形全般を表す親クラス Shape に持たせ、各図形クラスは Shape クラスを継承することで機能を共有する。そして、各図形クラスで後者のような個別の機能を定義する。図

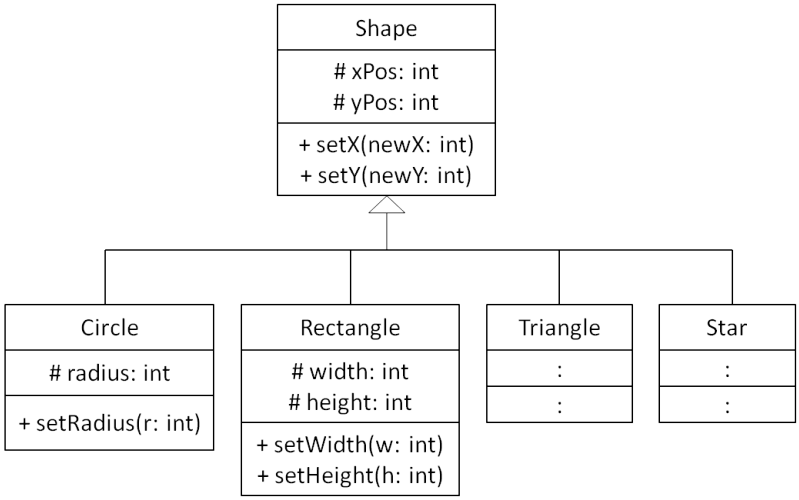

図 2.1: FigureEditor の図形を表すクラスの継承関係

形エディタで用いられる各種図形を表すために作成したクラスの継承関係の例を図 2.1 に示した。

以下では、簡単のため図形を表すクラスは図 2.1 の中の Shape、Circle、Rectangle クラスのみ存在するとして話を進める。図 2.2、図 2.3、図 2.4 に各クラスのソースコードを示した。

```
1  public class Shape {
2     protected int xPos,yPos; // エディタ上の位置
3
4     public void setX(int newX){
5        this.xPos = newX;
6        Screen.repaint();
7     }
8
9     public void setY(int newY){
10       this.yPos = newY;
11       Screen.repaint();
12    }
13       ...
14 }
```

図 2.2: Shape クラスの実装

```
1 public class Circle extends Shape {
2    protected int radius;
3
4    public void setRadius(int r){
5       this.radius = r;
6       Screen.repaint();
7    }
8       ...
9 }
```

図 2.3: Circle クラスの実装

```
1  public class Rectangle extends Shape {
2       protected int width, height;
3
4       public void setWidth(int w){
5            this.width = w;
6            Screen.repaint();
7       }
8
9       public void setHeight(int h){
10           this.height = h;
11           Screen.repaint();
12      }
13
14      public void regularize(int size){
15           setWidth(size);
16           setHeight(size);
17      }
18      ...
19 }
```

図 2.4: Rectangle クラスの実装

ユーザーが図形エディタ上でマウスを用いて各図形の大きさや位置を変更した時、set メソッドが呼ばれてフィールドの値を変更する。しかし、各図形オブジェクトのフィールドの値を変えただけでは、画面に表示されている図形の形は変化しない。フィールドの値を変える度に、画面に対して再描画処理を依頼して画面を更新しなければならない。その為、set メソッドではフィールドの値が変更される度に再描画を依頼する為の repaint メソッドを呼んでいる。

この repaint メソッドは、各図形クラスの様々な部分に散らばってしまっている。このようにモジュール間をまたがっている関心事を横断的関心事

と呼ぶ。横断的関心事が存在することによる問題として、次のような問題が挙げられる。もし repaint メソッドの変更を行った場合、各図形クラスのソースコードから修正すべき部分を探索して変更するという大変労力のいる作業となってしまう。

以上のような問題を解決する為に横断的関心事を一つのモジュールにまとめて扱う為に考案されたのがアスペクト指向言語である。次節では、上で述べられた問題を解決する為の手法を述べる。

2.2　AspectJ

アスペクト指向言語の例として AspectJ[7] を挙げる。AspectJ は Java 言語を拡張する事でアスペクト指向を実現した言語である。AspectJ 用のコンパイラである ajc によって、AspectJ プロジェクトのコンパイル時にはアスペクトとして記述したコードは Java プログラムに変換される。その為、通常の JVM で Java プログラムとして実行することが出来る。

AspectJ ではアスペクトと呼ばれる新しいモジュール単位を作成し、横断的関心事をモジュールにまとめる。まず、AspectJ の機構について説明する。

- **アスペクト**

  横断的関心事を一つにまとめた新しいモジュール単位。ポイントカットとアドバイスから構成される。

- **ポイントカット**

  ジョインポイントの集合で、いつアドバイスを実行するかの指定を行う。条件を定め、プログラム実行中に存在するジョインポイントを限定し、集合を作る。上の問題を解決する為の

- **ジョインポイント**

  プログラム実行中で、アドバイスを織り込む事が可能な時を表す。ポイントカットにより選択される。

- **アドバイス**

  クラスでいうメソッドに相当する。ポイントカットで指定された時に実行される処理。

- **織り込み**

  クラスやアスペクトをジョインポイントで結びつける処理の事。実装によって、アドバイス本体をジョインポイントに埋め込む方法、アドバイス本体を呼び出すコードを埋め込む方法などがある。

AspectJ では上記の機構を用いて横断的関心事をアスペクトというモジュールにまとめる。

2.2.1　AspectJ によるモジュール化

この小節では、実際に AspectJ を用いて前節で述べた問題を解決する方法を紹介する。前節で述べた問題は、repaint メソッドが各図形クラスに散らばってしまい、その修正を行うのが難しいということであった。そこで、AspectJ を用いて再描画の処理を一つのアスペクトにまとめてみる。それぞれの図形クラスは図 2.5、図 2.6、図 2.7 の実装になる。

```
1  public class Shape {
2     protected int xPos,yPos; // エディタ上の位置
3
4     public void setX(int newX){
5        this.xPos = newX;
6        // Screen.repaint();
7     }
8
9     public void setY(int newY){
10       this.yPos = newY;
11       // Screen.repaint();
12    }
13      ...
14 }
```

図 2.5: 再描画処理をアスペクトに分離した後の Shape クラスの実装

```
1  public class Circle extends Shape {
2     protected int radius;
3
4     public void setRadius(int r){
5        this.radius = r;
6        // Screen.repaint();
7     }
8     ...
9  }
```

図 2.6: 再描画処理をアスペクトに分離した後の Circle クラスの実装

各図形クラスに関しては、問題となっていた再描画処理である repaint メソッドを全てコメントアウトしている。つまりこのままの状態では Fig-

```
1  public class Rectangle extends Shape {
2     protected int width, height;
3
4     public void setWidth(int w){
5        this.width = w;
6        // Screen.repaint();
7     }
8
9     public void setHeight(int h){
10       this.height = h;
11       // Screen.repaint();
12    }
13
14    public void regularize(int size){
15       setWidth(size);
16       setHeight(size);
17    }
18    ...
19 }
```

図 2.7: 再描画処理をアスペクトに分離した後の Rectangle クラスの実装

ureEditor に反映がされない。そこで、アスペクトを用いて再描画を行うべき部分に処理を織り込む。図 2.8 がその Repainter アスペクトである。

まず、処理を織り込むべき時を示すポイントカット setMethods を定義している。図形を表すクラスは図形の情報を表すフィールドを持つ。保守性を考えてアクセスレベルを protected にしているため、値を変更するには必ず set メソッドを介す必要がある。つまり、再描画を行うタイミングである図形の情報に変更があった時というのは、set メソッドが呼ばれた時である。そこで、ポイントカット setMethods では set メソッドが呼ばれた時を指定している。

次にアドバイスの定義である。先ほど定義した setMethods ポイントカッ

```
1  aspect Repainter {
2     pointcut setMethods():
3        execution(void set*(..));
4
5     after():setMethods(){
6        Screen.repaint();
7     }
8  }
```

図 2.8: 再描画処理を行う Repainter アスペクトの実装

トを用いて、織り込みを行うタイミングを記述している。再描画処理の依頼は set メソッドによる値を変更する処理が終わった直後であって欲しい為、after アドバイスを用いて、タイミングを指定している。そして、具体的な処理の内容となる、Editor への再描画処理の依頼の為のコードである Screen.repaint() を記述している。

以上で、前節で述べた問題に対処する事が出来る。repaint メソッドに対する変更が行われた場合は、Repainter アスペクトだけを編集すればよくなる為、問題となっていた横断的関心事を一つのモジュールにまとめることが出来た。

2.3 GluonJ

アスペクト指向言語のもう一つの例として GluonJ[2, 3] を挙げる。GluonJ はオブジェクト指向言語である Java に最小限の拡張を加えてアスペクト指向を実現したアスペクト指向言語である。

AspectJ では、オブジェクト指向でのモジュール化はクラスを使い、オブジェクト指向ではモジュール化することが出来ない横断的関心事に対して、アスペクトという機構でモジュール化をしている。

一方、GluonJ ではオブジェクト指向ではモジュール化することが出来ない横断的関心事に対しても、クラスを用いてモジュール化する事が出来る。織り込みを行うクラスに対しては、@Reviser というアノテーションを付ける事で織り込みを行うクラスであることを指定する。このクラスをリバイザ (reviser) と呼ぶ。リバイザはどのクラスに織り込みを行うかを指定する為に織り込みを行うクラスを継承して作成する。

```
1  @Reviser
2  protected class ShapeRepainter extends Shape{
3
4     public void setX(int newX){
5        super.setX(newX);
6        Screen.repaint();
7     }
8
9     public void setY(int newY){
10       super.setY(newY);
11       Screen.repaint();
12    }
13 }
```

図 2.9: 再描画処理を行う ShapeRepainter リバイザの実装

```
1 @Reviser
2 protected class CircleRepainter extends Circle{
3
4    public void setRadius(int r){
5       super.setRadius(r);
6       Screen.repaint();
7    }
8 }
```

図 2.10: 再描画処理を行う CircleRepainter リバイザの実装

```
1  @Reviser
2  protected class RectangleRepainter extends Rectangle{
3
4     public void setWidth(int w){
5        super.setWidth(w);
6        Screen.repaint();
7     }
8
9     public void setHeight(int h){
10       super.setHeight(h);
11       Screen.repaint();
12    }
13 }
```

図 2.11: 再描画処理を行う RectangleRepainter リバイザの実装

図 2.9、図 2.10、図 2.11 に GluonJ のリバイザを用いて、先ほどの repaint メソッドを別のモジュールに分けた場合のコードを示した。ただし、Shape、Circle、Rectangle クラスは図 2.5、図 2.6、図 2.7 と同じの為省略している。

リバイザを作成した後、織り込みを行いたいメソッドをオーバライドしてメソッドを定義する。この例では元の set メソッドである値の変更を行った後、再描画処理を依頼したい。その為、super で織り込む前のメソッドを呼んだ後、再描画処理を依頼する repaint メソッドを呼んでいる。これにより、set メソッドが呼ばれた時に repaint メソッドを呼ぶよう織り込む事が出来る。

ただし、上の例では織り込みを行いたいクラスに対し、一つのリバイザを定義している為、クラスが増えるとその分作成するファイルも増えてしまう。さらに言えば、問題であった repaint メソッドが一つのモジュールにまとめられていない為、モジュール化出来たとは言えない。これに対応する為、GluonJ ではリバイザをグループ化する事を可能にしている。グループ化した場合のソースコードは図 2.12 になる。

```
1  @Reviser
2  public class Repainter{
3
4     @Reviser
5     protected class ShapeRepainter extends Shape{
6
7        public void setX(int newX){
8           super.setX(newX);
9           Screen.repaint();
10       }
11
12       public void setY(int newY){
13          super.setY(newY);
14          Screen.repaint();
15       }
16    }
17
18    @Reviser
19    protected class CircleRepainter extends Circle{
20
21       public void setRadius(int r){
22          super.setRadius(r);
23          Screen.repaint();
24       }
25    }
26
27
28    @Reviser
29    protected class RectangleRepainter extends Rectangle{
30
31       public void setWidth(int w){
32          super.setWidth(w);
33          Screen.repaint();
34       }
35
36       public void setHeight(int h){
37          super.setHeight(h);
38          Screen.repaint();
39       }
40    }
41 }
```

図 2.12: 再描画処理を行う Repainter リバイザの実装

リバイザをインナークラスとして定義する事で、一つのリバイザにグループ化する事が可能である。ただし、グループ化を行う時、インナークラスは static である必要がある。これにより、repaint メソッドという横断的関心事を一つのモジュールにまとめ、問題を解決する事が出来た。

2.3.1　複数の織り込みに対する処理

GluonJ では、同じクラスに対して複数の織り込みが行われる場合を想定して、@Require というアノテーションを定義している。@Require はリバイザ同士の優先順位を決定する為のアノテーションである。例えば、先ほどの例で set メソッドの実行時間を測りたいとする。この時、新たに Timer リバイザを図 2.13 のように定義する。

set メソッドの時間計測は、repaint メソッドも含めて計測を行いたい。その為、織り込みの順番は Repainter リバイザを適用してから Timer リバイザを適用するという順番が重要になってくる。この織り込みの重なりがあった時の順番を決める方法として、2 行目の@Require アノテーションが存在する。@Require は配列の引数を取り、自分より前に適用するリバイザを指定する事が出来る。逆に、二つ以上のリバイザを織り込む際、@Require で互いに適用する順番が定義されていない場合は実行時にエラーが発生する。

2.4　開発支援ツールの必要性

前節では横断的関心事をアスペクト指向言語を用いてモジュール化する方法を述べた。さらに、応用例として織り込みが複数存在する場合の対処法にまで話を進めた。アスペクト指向言語により、横断的関心事を別の機構を用いてモジュール化するが可能になった。しかし、織り込みが行われる側では横断的関心事がどこに織り込まれるかという情報は全く記述されない。また、織り込みを行う側も、他に別の織り込みが行われるかどうかを知ることが出来ない。その為、実行時にコードの中のどの箇所に織り込みが行われるかを知ることが出来ない。織り込みに関する情報を得るには、プロジェクト内の関係のあるソースコードを自分で読んで把握する必要がある。これはプログラマにとって大変労力のいる作業であり、作業効率を低下させる原因となる。その為、このような開発を支援するツールが必要不可欠であると言うのが一般的な見解である。その為、GluonJ においても開発を支援するツールが必要不可欠となる。

```
1  @Reviser
2  @Require({Repainter.class})
3  public class Timer{
4
5     @Reviser
6     protected class ShapeTimer extends Shape{
7
8        public void setX(int newX){
9           double beforeTime = System.nanoTime();
10          super.setX(newX);
11          double afterTime = System.nanoTime();
12          // 実行時間の取得
13          double execTime = afterTime - beforeTime;
14          System.out.println(``setX:'' + execTime);
15       }
16
17       public void setY(int newY){
18          ...
19       }
20    }
21
22    @Reviser
23    protected class CircleTimer extends Circle{
24
25       public void setRadius(int r){
26          ...
27       }
28    }
29
30    @Reviser
31    protected class RectangleRepainter extends Rectangle{
32       ...
33    }
34 }
```

図 2.13: 実行時間を計測する Timer リバイザの実装

2.5　関連研究: AJDT(AspectJ Development Tools)

AJDT(AspectJ Development Tools)[4] は AspectJ 開発を支援する為の Eclipse のプラグインである。Eclipse は統合開発環境の一つでオープンソースのソフトウェアである。AJDT には AspectJ 開発を支援する為のツールが多数用意されている。

Eclipse とは

Eclipse[6] は統合開発環境 (IDE:Integrated Development Environment) の一つである。統合開発環境とは、プロジェクト管理機能やエディタ、デバッガなど、アプリケーション開発に必要な機能を備えたソフトウェアのことである。アプリケーション開発において、統合開発環境は開発にかかる労力を大幅に削減する事が出来る重要な要素である。Eclipse の詳細は第 4 章で述べる。

2.5.1　AJDT の提供するツール

この小節では AJDT が提供する、織り込み関係を表示する為のツールを紹介する。

Outline

アウトラインビューでは現在アクティブになっているファイルの内容を表示する。アウトラインビューはポイントカット名、アドバイスの種類が列挙されるだけの為、どのアドバイスでどのポイントカットが利用されているかなどの情報を知ることが出来ない。

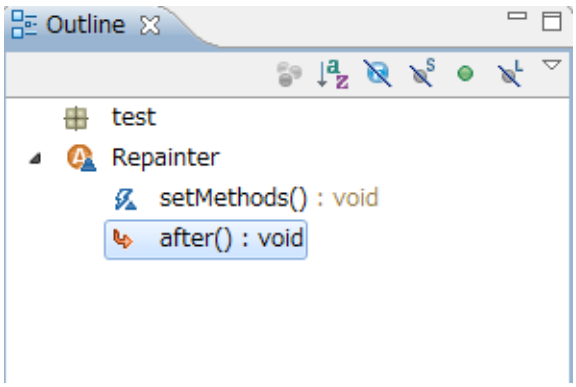

図 2.14: アウトラインビュー

Visualizer

Visualiser ビューは、パッケージエクスプローラでパッケージを選択した時、そのパッケージ内にあるファイルに対してのアスペクトの影響が表示されるようになっている。クラスごとに縦の棒グラフが表示され、内部に横方向に色のついた線が入っている。縦方向はソースコードの相対的な長さを表し、色のついた横線が入っている場所はソースコードでアドバイスが実行される箇所を表している。異なるアスペクトは色の違いで区別され、どのアスペクトのアドバイスも実行されない場合はクラス内の要素は黒色表示となる。クラス内の横線をクリックする事で、ソースコード内の該当する行へとジャンプすることが可能である。

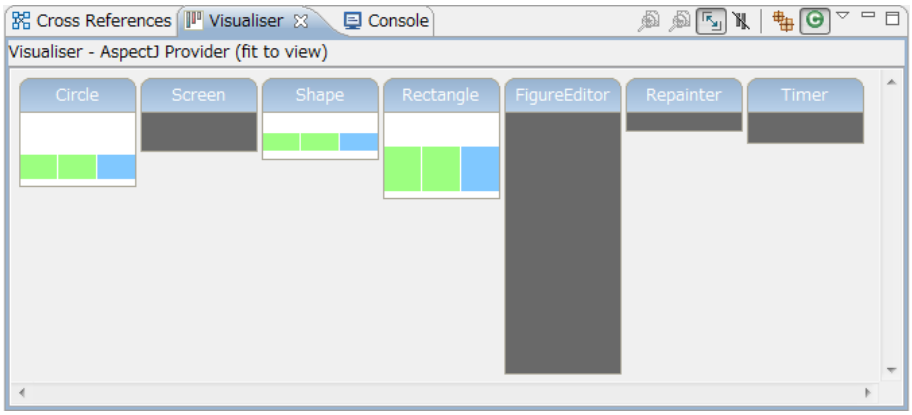

図 2.15: Visualizer ビュー

また、Visualizer Manu ビューを開くことで、Visualizer ビューに反映させるアスペクトを指定する事が出来る。アスペクトのチェックを外すと、Visualizer ビューから指定のアスペクトが表示されなくなる。

このビューにより、アスペクトが全体としてどの箇所にどのアスペクトが織り込まれるかという全体像は把握出来るが、織り込み関係の重なりを表示する方法は存在しない為、織り込み関係の情報が欲しい場合にはエディタにジャンプし、該当する箇所を読まなければならない。

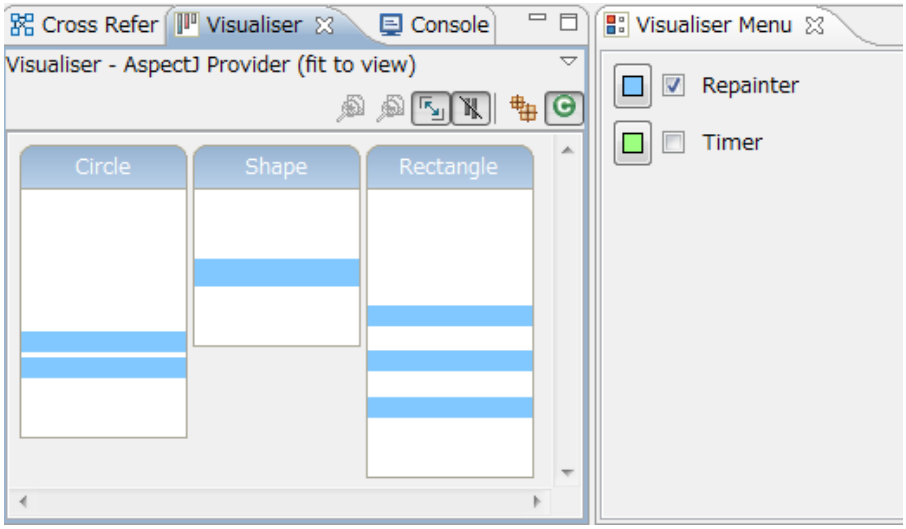

図 2.16: Visualizer Manu ビュー

Cross References

Cross References ビューはアウトラインビューと同様に、現在アクティブになっているファイルの内容を表示する。Cross-References ビューでは、アスペクト側と織り込みが行われるクラス側の双方からお互いの情報を知ることが出来る。例えば、Repainter アスペクトに対応する Cross References ビューは図 2.17 のようになる。

Repainter アスペクト側からは、アスペクトを織り込む先のメソッドを列挙している。また、表示されている要素をクリックすると該当するクラスのエディタを開き、該当する行へとジャンプする事が出来る。例えば図 2.17 の一番上の要素である Shape.setX(int) をクリックすると、Shape クラスの sexX メソッドを定義している箇所にジャンプし、Cross References ビューは図 2.18 の様に変化する。これにより Shape クラスに織り込まれるアスペクトの情報を得る事が出来る。

以上のように、織り込みを行う側と織り込みが行われる側の双方から、お互いの情報を知る事が出来る。しかし、図 2.18 のように、同じ箇所に複数のアスペクトが織り込まれる場合、その優先順位を表示する事が出来ない。

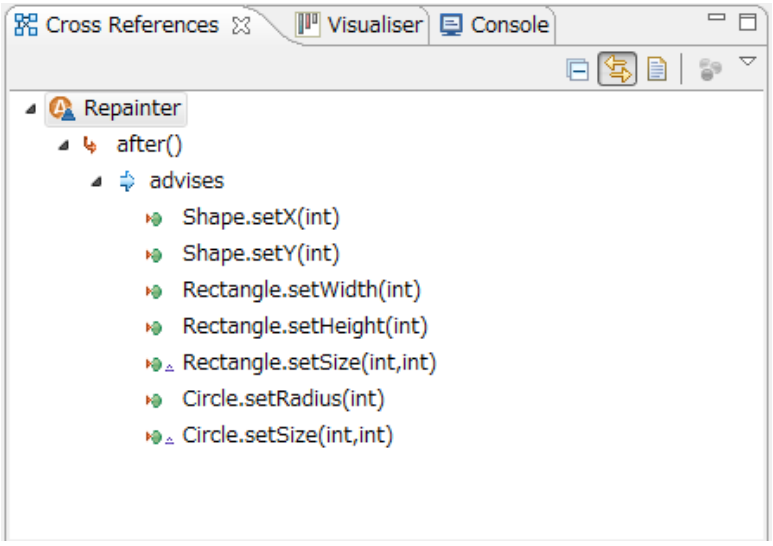

図 2.17: Cross References ビュー

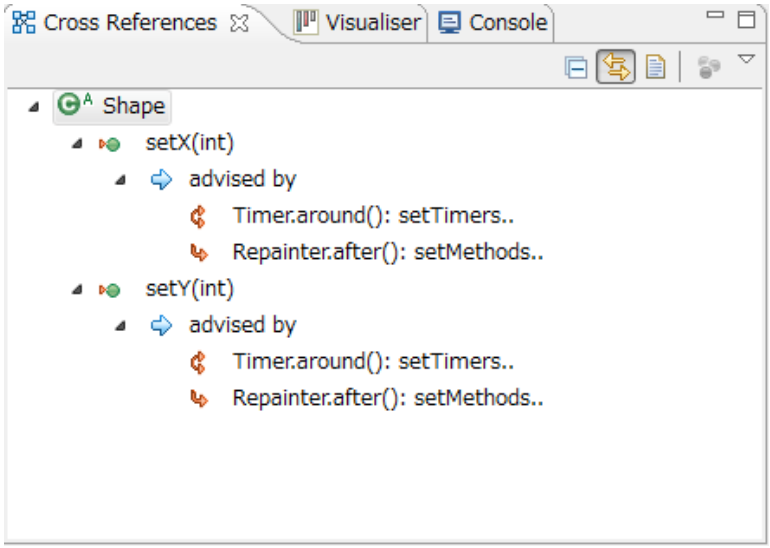

図 2.18: Cross References ビュー

2.5.2　AJDT の提供するツールの問題点

これまで、AJDT が提供するツールについて述べてきた。しかしこれらのツールの手法では GluonJ に対応するアウトラインビューを表示する事が出来ない。本研究では、織り込みの重なりを重視したアウトラインビューを提案する。

2.6　関連研究: Code Bubbles

プログラマは 60〜90%の時間をコードや他のソースを読むのに費やしている。また、従来の IDE ではファイルを基準として様々なビューを提供している。この為、自分の探しているワーキングセットや、関連するコードを探す為に相互作用のあるファイルを開き、スクロールして該当する場所探す。という行為を繰り返し行う必要がある。

CodeBubbles[1] では、この様な問題に対処するためにファイルを基準とせず、関連のあるコードのみを bubble という新しい構造として提供する IDE を提案している。CodeBubbles での開発画面は図 2.19 の様になっている。

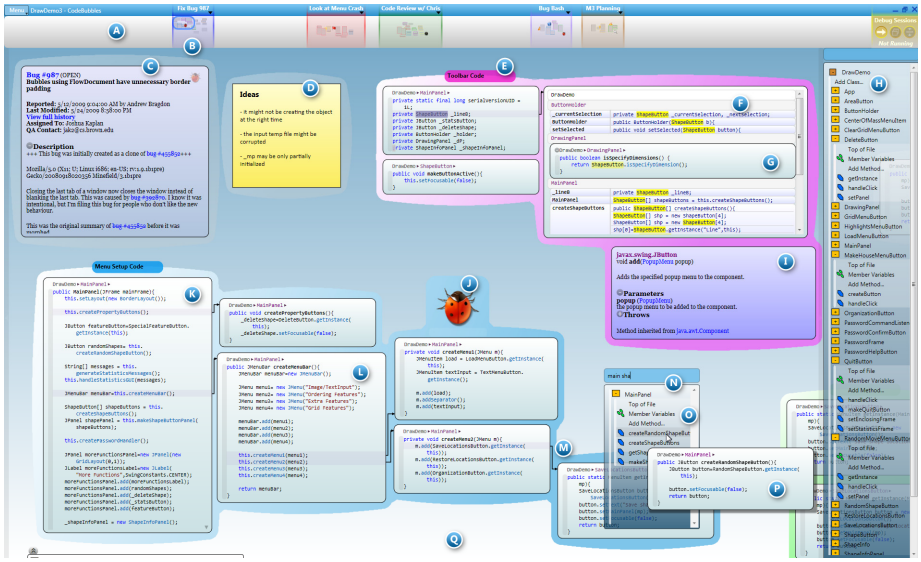

図 2.19: Code Bubbles(論文 Code Bubbles より抜粋)

編集を行いたいメソッドの内容のみを bubble で表示しているのが図 2.19 の K の部分である。また、その中で呼び出しを行なっているメソッドのコードを更に開いているのが図 2.19 の L の部分となっている。必要な情

報のみを bubble として表示し、関連のある bubble は矢印で繋いで一体化させている。これにより、求めているコードを探す手間が無くなり、関係の無いコードを読んで混乱するといった状況にも陥らなくなる。さらに、画面がいっぱいになった場合、図 2.19 の A の部分の bird’s eye ビューを使用する事が出来る。bird’s eye ビューでは、現在行っている作業スペースを保存したまま、別の作業スペースを開くことが出来る。

2.6.1　本研究との関連

Code Bubbles ではファイルを基準とする事無く、関連のあるコードの記述のみを bublles で提供している。ファイル内の情報だけでなく、関連のあるメソッドの情報を提供すると言う点が類似していると考えられる。

第3章　設計

本研究では、アスペクト指向言語である GluonJ に対応したアウトラインビューの設計と実装を行った。この章では本システムの設計について述べる。本システムは Java 開発を支援する為のツールである JDT(Java Development Tools) の拡張を行う事で実現している。既存のアウトラインビューのメニューバーにボタンを追加し、そのボタンを押すことでビューの表示の切り替えを行う。

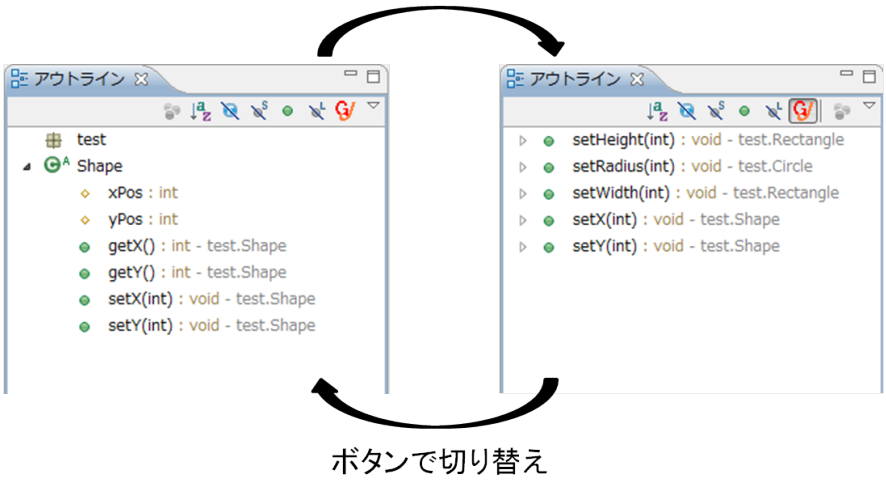

図 3.1: ビューの切り替え

次節では、本システムのアウトラインビューの特徴について詳しく述べる。

3.1　特徴

本システムである GluonJ に対応したアウトラインビューの大きな特徴として、以下の二つが挙げられる。

- 対称性

  前章で述べたように、GluonJ はクラスのみでアスペクト指向を実現している。区別の為に織り込みを行う側をリバイザと呼んでいるが、リバイザもクラスである。第 2.4 節で述べた Cross-References ビューでは、織り込みが行われるクラスと、織り込みを行うアスペクトとで別の階層表示をしている。本システムでは、織り込みを行うリバイザと、織り込みが行われるクラスとで、表示が一切変わる事が無い、全く同じビューを表示させる。

- 織り込みを行うリバイザ同士の重なりを表示

  複数のリバイザが一つのクラスを対象に織り込みを行った場合、織り込みを行うリバイザが、どのような順序で織り込まれるのかという優先順位が重要になる。本システムでは、優先順位を階層状にして表示を行う。

3.1.1　統一されたアウトラインビュー

本システムでは、織り込みを行うリバイザと織り込みが行われるクラスとで、表示方法を変えず、全く同じ表示をする。

第 2.4 節で挙げた FigureEditor を、再び例に挙げる。Circle クラス、Rectangle クラスのそれぞれの set メソッドの中で呼ばれていた repaint メソッドを Repainter リバイザで別のモジュールへとまとめている。これにより、repaint メソッドが各図形クラスに散らばる事無く、一つのリバイザにまとめる事が出来た。しかし一方で、各図形クラスからは repaint メソッドが消えた為、repaint メソッドが呼ばれる事を知る事が出来ない。これを回避するため、リバイザによってプログラムが織り込まれるメソッドを階層にして表示する機能を加えた。

親となるメソッドが織り込み元となる、各図形クラスの set メソッド。子になっているメソッドが、リバイザによって織り込みを行う Repainter の set メソッドとなっている。このアウトラインビューを確認することで、各図形クラスのどのメソッドに織り込みが行われるかを知ることが出来る。一方で、リバイザ側からも全く同じ表示を行う事で、織り込みを行うリバイザと織り込みが行われるクラスとで表示方法が変わる事がなく、統一されたアウトラインビューを実現している。

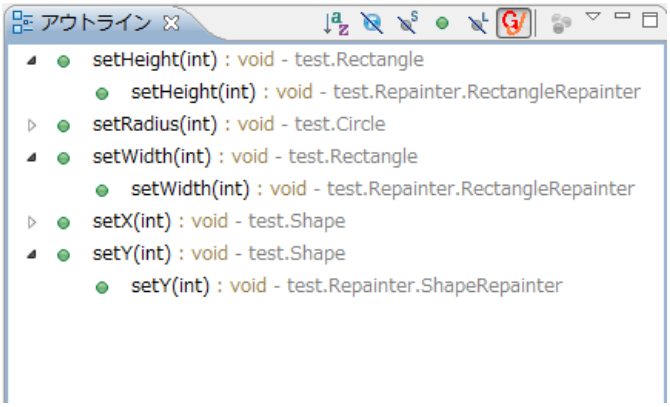

図 3.2: 織り込みを階層表示

3.1.2　優先順位を階層表示

織り込みを行うリバイザが複数存在した場合を考える。リバイザが複数存在した場合、それぞれのリバイザを適用する順序が重要になってくる。この優先順位の情報をアウトラインビューで提供を行う為に、適用する順序にしたがって階層表示を行う機能を加えた。

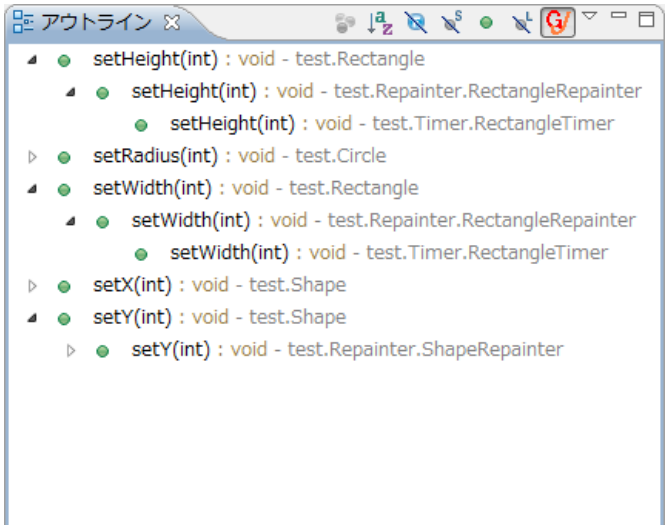

図 3.3: 織り込みの優先順位を階層表示

第 2.3 節で述べた例を挙げると、Timer リバイザによる set メソッドは、Repainter リバイザよりも後に織り込みを行うよう@Require で定義している。その為、set メソッドが上書きされる順番を表示する為に、一番上の

階層が各図形クラスの set メソッド、次の子が Repainter リバイザの set メソッド、次の子が Timer リバイザの set メソッドという階層を作成して表示を行う。これにより、それぞれのリバイザを適用する順序がアウトラインビューから知ることが出来るようになる。

また、仮に Timer リバイザと Repainter リバイザとの優先順位が決められていない場合は、各図形クラスの set メソッドの子にそれぞれのリバイザの set メソッドを子にし、優先順位が定められていないということを明示する。

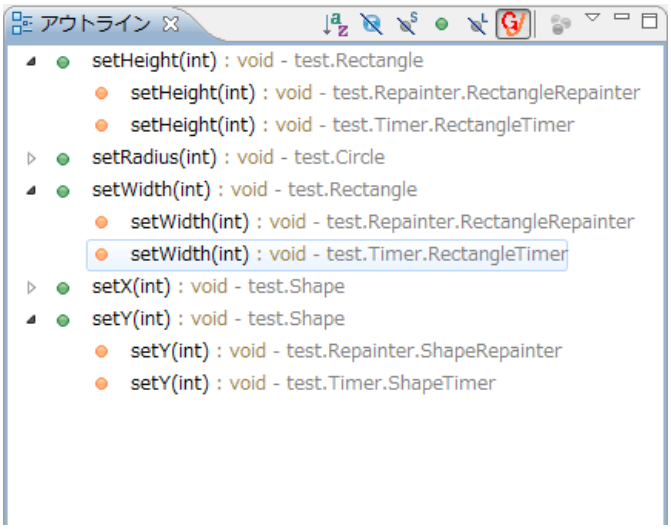

図 3.4: エラーの検出

さらに、それぞれのリバイザのメソッドのアイコンの色を変えることで、色の変わったリバイザの優先順位が定められていないと言う事を強調する。これにより、実行するよりも前に、コーディング中にエラーを知ることが出来る。

3.1.3　表示時のサポート

第 3.1.1 節、第 3.1.2 節で、アウトラインビューの概要について説明をした。この小節ではそのアウトラインビューを提供する前のサポートについて説明する。

リバイザは、プログラムが実行される際に織り込みを行うかどうかが決められる。プログラムの実行時にリバイザが織り込まれるかどうかは、main メソッドを持つクラスを実行する際の VM 引数で指定する必要がある為だ。すなわち、リバイザによって織り込みが行われるかどうかは、実

行する際の VM 引数によって変わって来る。その為、アウトラインビューを提供する前の段階として、main メソッドを持ったプログラムを実行するクラスの VM 引数の設定を行わなければならない。

よって、まず最初に、プログラムを実行する為の main メソッドのクラスをリストアップし、実行するクラスを選択する為のダイアログが開かれる。図 3.5 では Screen クラスのみが挙げられているが、これはプロジェクト内に main メソッドを持つクラスが Screen クラスのみである為である。プロジェクト内に main メソッドを持つクラスが複数存在した時は、リストの中から実行するクラスを選択する。

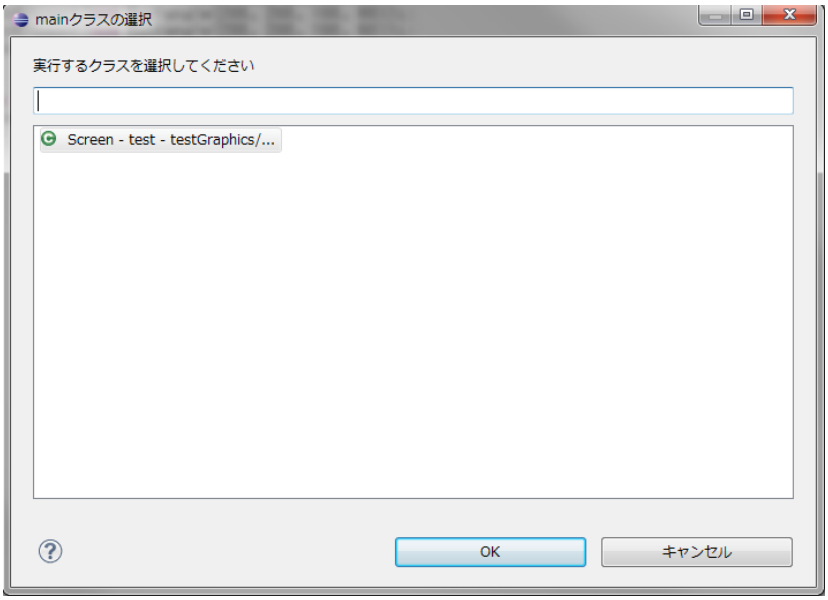

図 3.5: main メソッドを持つクラスを選択

実行するクラスを選択した後は実行する際の VM 引数の設定を行う為のダイアログが開かれる。図 3.6 がそのダイアログである。ここに、実行する際に適用するリバイザを入力する。入力の際はパッケージ名. クラス名というように、パッケージ名から入力を行う。複数のリバイザを選択したい場合はコンマで区切って入力する。今回の例では、Repainter リバイザと Timer リバイザを引数に持たせている。

VM 引数の入力を完了すると、VM 引数で入力したリバイザを基に、適用する順序を決定し、アウトラインビューに表示を行う。

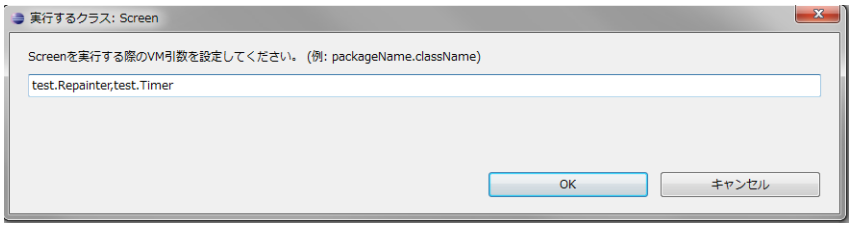

図 3.6: VM 引数の入力

3.2　その他のサポート

これまで、本システムの設計について述べてきた。この節では本システムの細かいサポートについて紹介する。

- 要素をクリックすると該当するクラスへとジャンプ

  本システムでは、さまざまなクラスのメソッドを要素として階層表示を行っている。よって、ある要素の詳しい情報を知りたい時には、そのメソッドを定義しているクラスをエディタで開き、さらにメソッドを定義している行を探す必要がある。本システムはこれに対応している。要素をクリックした際に、そのメソッドを定義しているクラスをエディタで開き、該当する行へとジャンプする事が出来る。

- リバイザの追加がされた際の、ビューの更新

  本システムではプロジェクト内のクラスを全て監視している為、新たなリバイザが定義されると、もしそのリバイザが実行するクラスに関連があった場合は、即座にビューを更新するようにしている。

第4章　実装

本研究の実装はEclipseのJDTを拡張することで行った。本章では、まずEclipseプラグイン開発を行う際に必要な基礎知識を述べ、その後に本システムの実装方法について述べる。

4.1　Eclipseの拡張

EclipseはJavaの統合開発環境として有名であるが、Javaの統合開発環境としての機能は、Java開発支援ツールであるJDTによって提供されている。Eclipseはプラグインによる拡張が可能であり、JDTもプラグインの一つである。JDTはEclipseの提供している拡張ポイントに接続することにより実現されている。[8, 9]

Eclipseには、プラグインの開発用の機能としてPDE(Plugin Development Environment)が標準で含まれている。PDEを用いることにより、プラグイン開発を行う為に必要なファイルの管理や、拡張ポイントの管理などを容易に行う事が出来る。本研究もPDEを用いて実装を行った。拡張ポイントの詳細は後述する。

4.1.1　Eclipseのアーキテクチャ

Eclipseは本来は様々な開発ツールの為の統合プラットフォームを提供する事を目的としている。プラグインのアーキテクチャを図4.1に示した。

図4.1の内、Eclipseのプラグイン・アーキテクチャの核をなすのが最下層に位置しているOSGiランタイムである。このOSGiをプラグインの管理の為に用いている。そして、OSGiの上位に配置されているコンポーネントは全てプラグインとして提供されている。この為、Eclipseはとても高い拡張性を提供し、Java以外の多様な言語への対応を可能にしている。

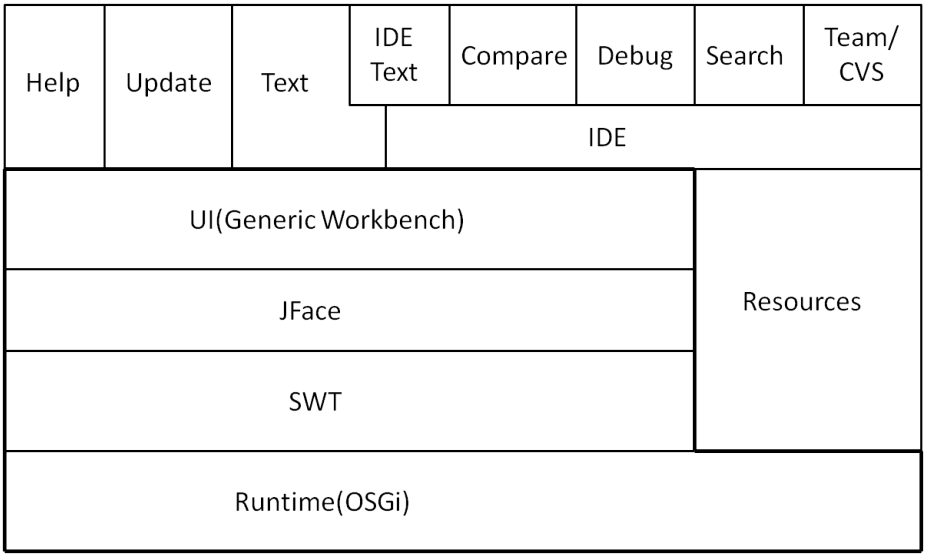

図 4.1: Eclipse アーキテクチャ

4.1.2　ワークベンチ

Eclipse ではユーザーが作業する為のユーザーインターフェース全般の事をワークベンチと呼ぶ。ワークベンチは図 4.2 のような構成になっている。ここでは、ワークベンチの構成とその要素の役割を述べる。

- **ワークベンチウィンドウ**

  Eclipse のウィンドウ。通常は 1 つのワークベンチに対し、1 つのワークベンチウィンドウが開かれるが、1 つのワークベンチが複数のワークベンチウィンドウを開くことも可能である。

- **ワークベンチページ**

  ワークベンチ上で開いているパースペクティブ毎に一つのページが対応する。このページに複数のビューやエディタを含むことが出来る。

- **パースペクティブ**

  ビューやウィンドウのレイアウトなどをセットにして、特定の用途をサポートする為のもの。

- **ビュー**

  開発者にとって有益な情報を提供する役割を果たす。1 つのワークベンチで同じビューを開くことは出来ない。

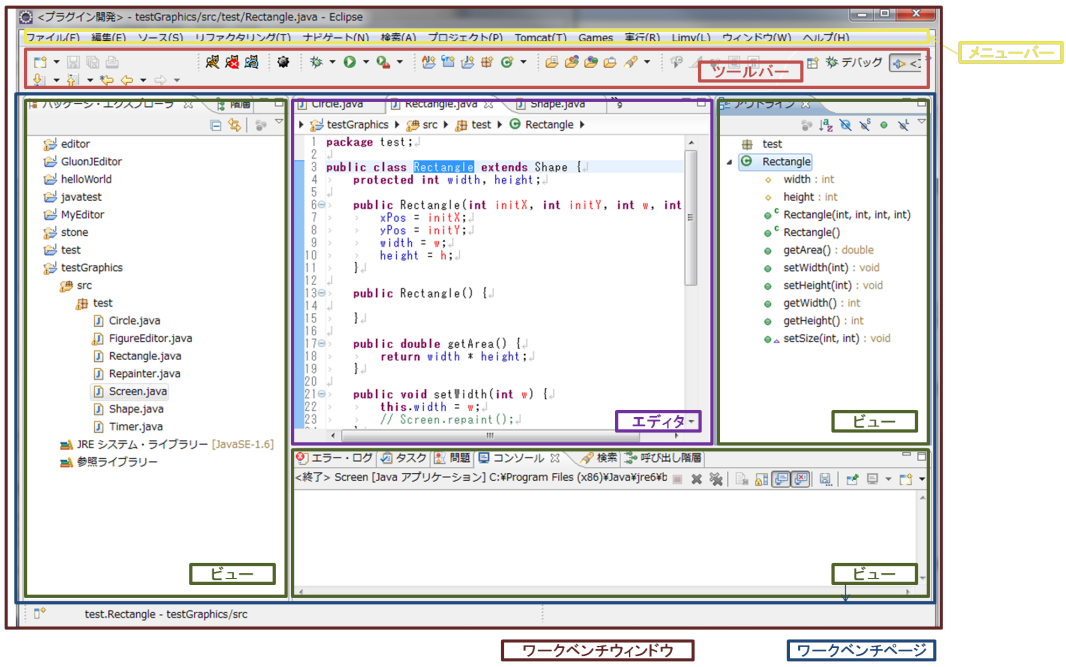

図 4.2: ワークベンチ

- **エディタ**

  ファイルの編集を行う場所。ビューとは違い、1 つのワークベンチで複数のエディタを開くことが出来る。1 つのファイルに対して複数のエディタを開くことも可能である。

4.1.3　PDE(Plugin Development Environment)

PDE は Eclipse に標準で含まれているプラグイン開発用の機能である。新しいプラグイン開発を行う際には、最初にマニフェストファイルと呼ばれるファイルを作成しなければならない。マニフェストファイルは MANIFEST.MF と pulagin.xml という名前のファイルであり、この二つのファイルにプラグインの詳細を記述する。これらの二つのファイルの管理を容易に行う為に用意されているのが PDE である。例として、Eclipse プラグインのテンプレートとなっている「HelloWorld」プラグインの MANIFEST.MF と plugin.xml を図 4.3 と図 4.4 に示す。

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: HelloWorld
Bundle-SymbolicName: helloWorld; singleton:=true
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: helloworld.Activator
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

図 4.3: MANIFEST.MF

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>

   <extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            label="サンプルのアクション・セット"
            visible="true"
            id="helloWorld.actionSet">
         <menu
               label="サンプル・メニュー(&amp;M)"
               id="sampleMenu">
            <separator
                  name="sampleGroup">
            </separator>
         </menu>
         <action
               label="サンプル・アクション(&amp;S)"
               icon="icons/sample.gif"
               class="helloworld.actions.SampleAction"
               tooltip="Hello, Eclipse world"
               menubarPath="sampleMenu/sampleGroup"
               toolbarPath="sampleGroup"
               id="helloworld.actions.SampleAction">
         </action>
      </actionSet>
   </extension>

</plugin>
```

図 4.4: plugin.xml

MANIFEST.MF はプラグインの ID や名称、他のプラグインとの依存関係、実行環境などを記述するファイルである。plugin.xml はプラグインの拡張や拡張ポイントの定義などを記述するファイルである。PDE はこれらのファイルの編集を容易に行う為に、マニフェストエディタというエディタを提供している。マニフェストエディタを図 4.5 に示す。

図 4.5: マニフェストエディタ

マニフェストエディタを用いてマニフェストファイルを開くと、図 4.4 のように 9 つのタブが表示される。ここで編集を行った情報は、MANIFEST.MF や plugin.xml に伝わり、自動的に対応する箇所の書き換えを行ってくれる。以下では、それぞれのタブについて説明する。

- **概要**

  プラグインの ID、バージョン、名称などのプラグイン全体の設定を行う。また、開発を行っているプラグインが組み込まれた状態で Eclipse を起動する事が出来る。これをランタイムワークベンチと呼び、ここでプラグインの動作を確認を行う。さらに、作成したプラグインを配布可能な形式でエクスポートすることも可能である。

- **依存関係**

  プラグインの間の依存関係を設定する。開発中のプラグインを動作するのに必要なプラグインをここで追加する。

- **ランタイム**

  プラグインが別のプラグインに公開するパッケージと、その可視性の設定を行う。公開されたパッケージは別のプラグインでインポートして利用することが可能になる。

- **拡張**

  プラグインが拡張を行う拡張ポイントを設定し、どのような拡張を提供するかを宣言する。プラグインの開発を行う上で重要なページとなっている。

- **拡張ポイント**

  プラグインが提供する拡張ポイントの定義する。他のプラグインに対して提供する拡張ポイントの設定を行う。

- **ビルド**

  プラグインのエクスポートに関する設定を行う。ビルドの際に含めるファイルやフォルダを選択する。ここで編集を行った内容は、build.properties ファイルに書き込まれる。

- **MANIFEST.MF**

  MANIFEST.MF の内容を確認し、直接編集を行う事が出来る。

- **plugin.xml**

  plugin.xml の内容を確認し、直接編集を行う事が出来る。

- **build.properties**

  build.properties の内容を確認し、直接編集を行う事が出来る。

4.2　本システムの実装

本システムの実装は、JDT のアウトラインビューを拡張することで実現した。本節では、まず JDT が提供するアウトラインビューの実装について述べ、次に本システムの実装について述べる。

4.2.1　Java エレメント

Java エレメントとは、メソッドやクラスなどの Java プロジェクトを構成する要素である。表 4.1 は Java エレメントの種類を表にしたものである。各種の Java エレメントは JDT の IJavaElement インターフェースのサブインターフェースとなっている。Java クラスやプロジェクトなどに対する操作を扱うプラグインは、Java エレメントを操作する事で実現する。

4.2.2　JavaOutlinePage

JDT が提供するアウトラインビューを実現しているクラスが JavaOutlinePage クラスである。ビューを構成する要素には、モデル、ビューア、コンテンツプロバイダ、ラベルプロバイダが必要である。JavaOutlinePage の場合では、ビューアが JavaOutlineViewer クラス、コンテンツプロバイダが ChildrenProvider クラス、ラベルプロバイダが DecoratingJavaLabelProvider クラスに当たる。ビューアは表示形式を指定する為のクラスである。アウトラインビューでは階層表示をさせたい為、JavaOutlineViewer は TreeViewer クラスを継承している。モデルは開発者がビューに表示したい内容そのものを指す。アウトラインビューでは、エディタで編集している ICompilationUnit に当たる。コンテンツプロバイダは、モデルからビューのコンテンツを取得する方法を定義する。さらに、コンテンツプロバイダはモデルを監視し、編集を感知するとビューの更新を行うようビューアに指示する役目も担っている。ラベルプロバイダは、ビューアが制御する各要素に、文字列とアイコンを関連付ける役割を持つ。図 4.10 に、ビューを構成する 4 つの要素の関係図を示した。

本システムでは、現在アクティブな状態になっているファイルを含む Java プロジェクトに対して、そのプロジェクト内の全ソースファイル、つまり ICompilationUnit の配列をモデルとして取得している。さらに、コンテンツプロバイダでそのモデルを基に、ビューを表示する為のコンテンツに編集しなおしている。

4.2.3　アクションの追加

本システムでは、既存の Java アウトラインビューにボタンを追加し、ボタンを押すことでビューの切り替えを行うアクションを追加している。アクションを追加する為には、まず Action クラスを継承したクラスを実装する必要がある。ボタンを押した時の動きは、Action クラスの valueChanged メソッドをオーバーライドする事で実装する。

エレメント	説明
IJavaModel	ワークスペースルートを表す Java エレメントである。全ての Java プロジェクトの親となっている。
IJavaProject	ワークスペース内の Java プロジェクトを表す。(IJavaModel の子)
IPackageFragmentRoot	パッケージフラグメントのセットを表し、そのフラグメントを、フォルダー、JAR、Zip ファイルのいずれかである基本リソースにマップする。(IJavaProject の子)
IPackageFragment	パッケージ全体に対応するワークスペースの一部、またはパッケージの一部を表す。(IPackageFragmentRoot の子)
ICompilationUnit	Java ソース (.java) ファイルを表す。(IPackageFragment の子)
IPackageDeclaration	コンパイル単位内のパッケージ宣言を表す。(ICompilationUnit の子)
IImportContainer	コンパイル単位内のパッケージのインポート宣言のコレクションを表す。(ICompilationUnit の子)
IImportDeclaration	1 つのパッケージのインポート宣言を表す。(IImportContainer の子)
IType	コンパイル単位内のソースタイプ、またはクラスファイル内のバイナリー形式を表す。
IField	型内部のフィールドを表す。(IType の子)
IMethod	型内部のメソッド、またはコンストラクターを表す。(IType の子)
IInitializer	型内部の静的イニシャライザ、またはインスタンスのイニシャライザを表す。(IType の子)
IClassFile	コンパイル済みの (バイナリーの) 型を表す。(IPackageFragment の子)

表 4.1: Java エレメントの種類

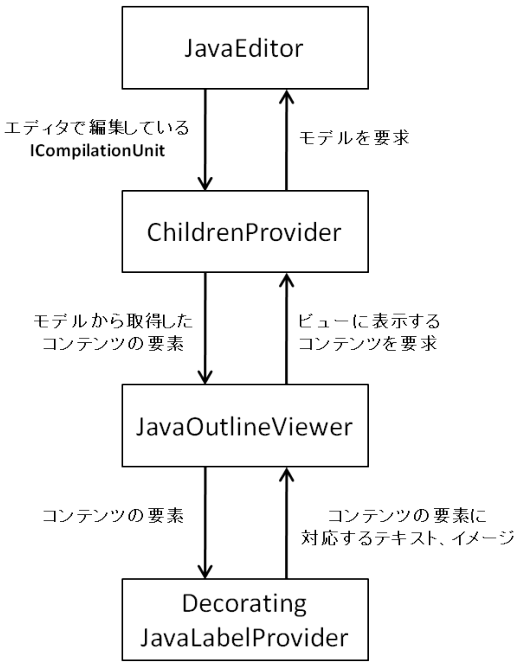

図 4.6: ビューを構成する要素

本システムでは、ボタンを押すと main メソッドを持つクラスをリストアップし、対象となるクラスの選択を求めるダイアログを開く。次に、指定したクラスを実行する際の VM 引数入力を求めるダイアログを開き、VM 引数を元に対象となるリバイザを決定し、ビューに表示する。以上を一連の流れとしたアクションを実装した。リバイザの決定については次節で詳しく述べる。

そして、このアクションを JavaOutlinePage クラスがボタンの管理を行なっている IToolBarManager へと追加する事で、アウトラインビューにボタンの追加を実現した。

4.2.4　リバイザの決定

main メソッドを持つ実行を行うクラスに織り込みを行うリバイザを読む為に、まずモデルとして取得した ICompilationUnit の配列に、@Reviser アノテーションが付いているのかどうかでふるいをかける。@Reviser アノテーションが付いているのかどうかを確認するには、以下のように行う。

ICompilationUnit から IType を取得し、さらに IType からアノテーションの配列を取得する。この配列の名前を確認する事で、特定のアノテー

```
1  public boolean checkSelfReviser(ICompilationUnit cu){
2       // IType の取得
3       IType = cu.findPrimaryType();
4       // アノテーションの配列を取得
5       IAnnotation[] anno = type.getAnnotations();
6
7       for(int i = 0; i < anno.length; i++){
8            if (anno[i].getElementName().equals("Reviser"))
9                 return true;
10      }
11      return false;
12 }
```

図 4.7: @Reviser の有無を確認

ションがついているかどうかを確認することが出来る。上記の方法でプロジェクト内のリバイザを全て取得する。

次に、実行を行うクラスに関連のあるリバイザのみにふるいをかける為、VM 引数からリバイザを取得し、そのリバイザから@Require アノテーションで指定されているリバイザを取得する。さらに取得したリバイザに上で述べた方法でさらなるリバイザが存在するかを調べ、実行時に関連のある全てのリバイザを取得する。

最後に、関連のあるリバイザを適用する順序を決定する。これは、まず関連のあるリバイザのリストに対して@Require で選択されている順番を取得する。取得した順番から、最後に適用するリバイザを決める。最後のリバイザを決める事が出来なければエラーとして処理するよう指示する。最後のリバイザを決める事が出来た場合、次にそのリバイザの前に適用するリバイザを探し、一意に定まれば決定。定まらなければエラーとして処理する。これを関連するリバイザの数だけ続け、最終的にリバイザの順番を直列に並べる。直列に並べる事が出来れば、リバイザの順番が一意に定められていることが分かる。また、エラーとなった場合は、どこで順番が合わなくなったかを記憶しておき、表示する際にアイコンを変えるよう指示する。

4.2.5　階層表示

第 4.2.4 節でリバイザの順序を決定したら、その順序にしたがって階層表示を行う。まず、トップの要素として、リバイザが織り込みを行うメソッドを列挙する。織り込みを行うメソッドを取得するには、オーバーライドしているメソッドを取得する必要がある。

```
1  // IType の取得
2  IType type = method.getDeclaringType();
3  // メソッド名を取得
4  String name = method.getElementName();
5  // メソッドの引数の型を取得
6  String[] parameter = method.getParameterTypes();
7  ITypeHierarchy hierarchy = type.newTypeHierarchy(null);
8  // スーパークラスを取得
9  IType superType = hierarchy.getSuperclass(type);
10 // オーバーライドしているメソッドを取得
11 IMethod superMethod = superType.getMethod(name,parameter);
```

図 4.8: @Reviser の有無を確認

その為、まずリバイザが継承しているクラスを取得し、そのクラスが持っているメソッドの中から一致するメソッドを取得する。実際のコードは図 4.8 のようになっている。ただし、method は IMethod 型だとする。

IMethod にはメソッド名を取得する getElementName メソッドとメソッドの引数の型を取得する getParameterType メソッドが存在する。このメソッドを使ってリバイザの持っているメソッド名と引数の型を取得する。次に、ITypeHierarchy というクラスを使い、リバイザが継承しているクラスを取得する。取得したスーパークラスから、getMethod メソッドを用いて名前と引数の型が一致しているメソッドを取得する事で、リバイザが織り込みを行うメソッドを取得する事が出来る。

以上の方法で織り込みが行われるメソッドを全て取得し、そのメソッドの集合をビューのトップに設定する。その後、第 4.2.4 節で定めたリバイザの順序に従って、織り込みを行うメソッドに対して階層にして表示を行う。

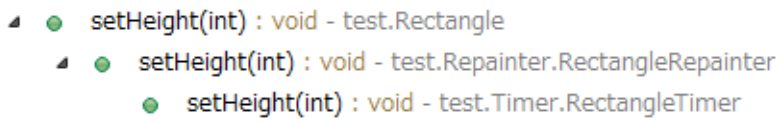

図 4.9: 階層状にして表示

また、第 4.2.4 節で述べた方法でリバイザの順序が決定出来なかった場合、エラーを表示する為に、トップの子として織り込みを行うメソッドを全て同じ階層に表示する。

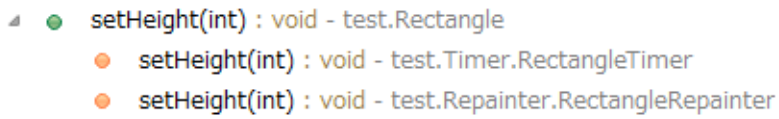

図 4.10: エラー表示

これにより、エラー検出も行う事が出来、コーディング中にエラーを知ることが出来る。

第5章　既存のツールとの比較

この章では本システムに関連のある既存のツールを挙げ、本システムとの違いを述べていく。

5.1　JDT(Java Development Tools)

本システムは Java 開発支援ツールである JDT[5] のアウトラインビューを拡張する事で実装を行ったが、JDT にはアウトラインビューの他にもさまざまなビューが存在する。この節では JDT のツールについて紹介する。

5.1.1　Call Hierarchy

Call Hierarchy ビューは、指定したメソッドを呼んでいる Java メンバを表示する為のビューである。

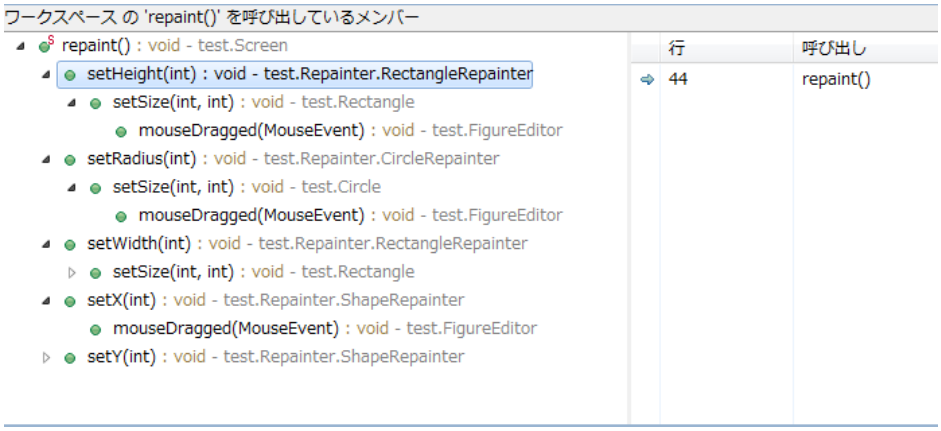

図 5.1: Call Hierarchy ビュー

このビューは指定したメソッドを呼んでいる Java メンバを表示する事が出来、さらにその中のメソッドを呼んでいるものを追加的に表示させる事で、呼び出し関係を階層にして表示する事が出来る。また、表示された。メソッドを選択すると、そのメソッドが実装されている部分までジャ

ンプする事が出来る。Call Hierarchy ビューは呼び出し関係を階層にして可視化させるビューである。織り込み関係を階層にして表示する本システムと、階層表示を行う部分は一致するが、表示させる項目が異なっている為、このビューでは織り込み関係を知る事は出来ない。

5.1.2　Type Hierarchy

Type Haierarchy ビューは二つのビューに分割されているビューである。まず、図 5.2 で示したビューは、指定したクラスのスーパークラス、またはサブクラスを階層にして表示するビューである。このビューを用いる事で、クラスの継承関係を知ることが出来る。階層で表示されている要素をクリックすることで、対応するクラスをエディタで開く事が出来る。

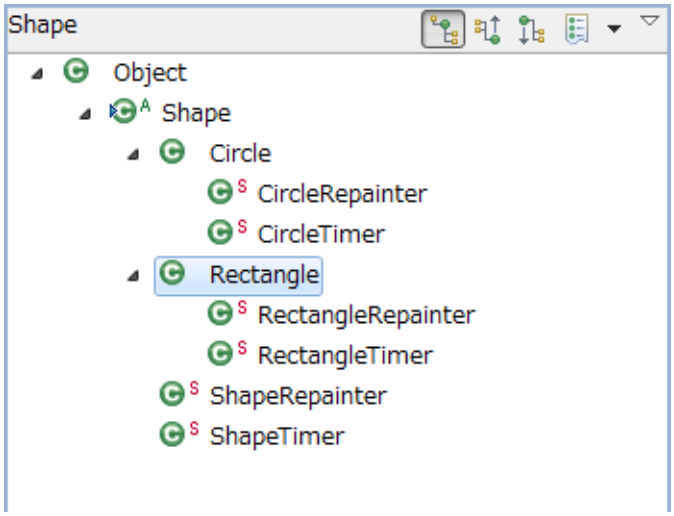

図 5.2: クラスの継承関係を表示するビュー

また、図 5.3 で示したビューは図 5.2 で選択されたクラスのコンストラクタ、フィールド、メソッドを表示するビューである。アウトラインビューと表示する内容は同じだが、アウトラインビューはエディタに対応して切り替わるが、このビューは図 5.2 のビューで選択されたクラスによって切り替わる。また、表示されている要素をクリックすることで、対応するクラスの該当行へとジャンプする事が出来る。

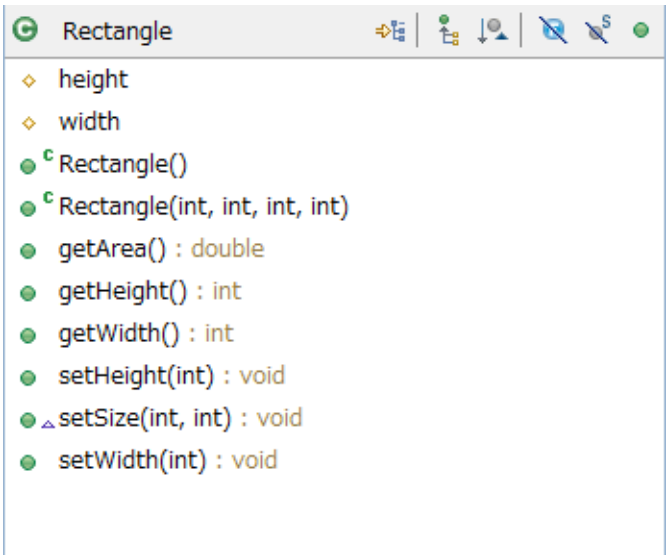

図 5.3: クラスのメンバを表示するビュー

Type Hierarchy ビューはクラスの継承関係を表示出来る為、織り込みたいクラスを継承してリバイザを作成する GluonJ には適しているように見える。しかし、一般的な継承関係のサブクラスと、リバイザとの区別をする方法が無い為、織り込み関係を知ることは出来ない。

第6章　まとめと今後の課題

6.1　まとめ

ソフトウェア開発にオブジェクト指向言語を用いる事で、プログラムをモジュール化し、保守性や拡張性を高める事が出来る。しかしその一方で、オブジェクト指向言語では横断的関心事をモジュール化して分離出来ないという限界もある。オブジェクト指向言語ではモジュール化する事が出来ない横断的関心事をうまくモジュール化する為の言語としてアスペクト指向言語が存在する。

しかし、横断的関心事を新たな構造を用いてモジュール化をすると、それぞれの横断的関心事の情報を知ることが困難になる。アスペクト指向言語によるプログラムの織り込み関係を把握するには、全てのファイルを開き、ソースコードを実際に読まなければならなくなり、大変効率が悪くなる。これを解決する為にアスペクト指向言語の一つである AspectJ の開発支援ツールとして AJDT が存在するが、プログラムの織り込み同士の重なりがあった場合、AJDT のツールでも、織り込み関係の優先順位を表示する方法が存在しない。

この問題を解決する為に、本研究ではアスペクト指向言語の一つである GluonJ に適応したアウトラインビューを提案し、実装を行った。GluonJ はオブジェクト指向言語である Java に最小限の拡張を行う事でアスペクト指向言語を実現している。また、プログラムの織り込みの優先順位をユーザーに指定させ、優先順位を一意に定める為、織り込みの優先順位を可視化させる事は重要になる。本システムでは、プログラムの織り込みの優先順位を階層構造にして表示する事で実現した。これにより、プログラムの織り込み関係をアウトラインビューを見るだけで知ることが出来る。

6.2　今後の課題

今後の課題としては、織り込み関係の表示の仕方の強化が挙げられる。本システムでは GluonJ の機構の内の@Reviser と@Require にしか対応していない。GluonJ には、特定のメソッド内からメソッドが呼ばれた時にプログラムの織り込みを行う@Code や、@Within と言ったアノテーション

も存在する。本研究はプログラムの織り込み関係を可視化させる事を目的としている為、これらにも対応したアウトラインビューを実現する事が、今後の課題である。

参考文献

[1] Bragdon, A., Reiss, S. P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. and Jr., J. J. L.: Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments, *ICSE '10 Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pp. 455–464 (2010).

[2] Chiba, S., Igarashi, A. and Zakirov, S.: Mostly modular composition of crosscutting structures by contextual predicate dispatch, *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pp. 539–554 (2010).

[3] Chiba, S., Nishizawa, M., Ishikawa, R. and Kumahara, N.: GluonJ home page, `http://www.csg.is.titech.ac.jp/projects/gluonj/`.

[4] the Eclipse Foundation: The AspectJ Project, `http://www.eclipse.org/aspectj/`.

[5] the Eclipse Foundation: Eclipse Java development tools, `http://www.eclipse.org/jdt/`.

[6] the Eclipse Foundation: Eclipse.org home, `http://www.eclipse.org/`.

[7] 千葉滋: アスペクト指向入門 Java・オブジェクト指向から AspectJ プログラミングへ, 技術評論社 (2005).

[8] 竹添直樹, 志田隆弘, 奥畑裕樹, 里見知宏, 野沢智也: Eclipse プラグイン開発徹底攻略, 株式会社毎日コミュニケーションズ (2007).

[9] 田中洋一郎: Eclipse プラグイン開発, http://yoichiro.cocolog-nifty.com/eclipse/.