平成24年度 修士論文

アスペクト指向言語のための視点に応じた編集を可能にするツール

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 11M37063

大谷 晃司

指導教員

渡辺 治 教授, 千葉 滋 教授

平成25年1月25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。

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。このような問題を解決する為に、アスペクト指向言語での開発を行う際は、アスペクト指向言語での開発を支援するツールが必要不可欠である。

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、AJDT では、複数のアスペクトが同じ場所に織り込みを行う場合に、それぞれのアスペクトは、別のアスペクトの存在を知ることが出来ないという問題ある。これは、アスペクト同士の衝突の危険をツールで提供出来ていない為、プログラムが開発者の意図しない動

きをする原因となってしまう。

また、アスペクトを用いて横断的関心事を別モジュールにまとめた事により、元のソースコードからは、横断的関心事に関するコードが消えてしまっている。織り込みを行う前の状態を知る為には、元のソースコードと、横断的関心事をまとめたソースコードとの複数ファイルを確認しなければならなくなる。この為、横断的関心事を別モジュールに分割する前は一つのファイルを確認するだけで知ることが出来た情報が、複数ファイルを確認しなければならなくなってしまう。その為、横断的関心事を一つのファイルに集めて表示を行うツールも必要であると考えられる。しかし、AJDT ではそのようなツールは存在しない。

そこで、本研究ではアスペクト指向言語の一つである GluonJ の為の開発支援ツールとして、編集を行なっているファイルの織り込みの情報を提供する拡張アウトラインビューと、織り込みが行われるクラスを編集している際に、織り込みが行われる関心事の情報を一つのファイルにまとめて表示を行う拡張エディタの二つのツールを提案し、実装を行った。拡張アウトラインビューでは、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の下に、織り込みの情報を階層的に表示する事で、織り込みの情報を提供している。拡張エディタでは、織り込みが行われるメソッドに、織り込みを行うメソッドの情報を複数ファイルから集め、その情報をまとめて表示を行う事で、複数ファイルの情報を一つのファイルで表示する事を実現している。

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、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。

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。このような問題を解決する為に、アスペクト指向言語での開発を行う際は、アスペクト指向言語での開発を支援するツールが必要不可欠である。

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、AJDT では、複数のアスペクトが同じ場所に織り込みを行う場合に、それぞれのアスペクトは、別のアスペクトの存在を知ることが出来ないという問題ある。これは、アスペクト同士の衝突の危険をツールで提供出来ていない為、プログラムが開発者の意図しない動

きをする原因となってしまう。

また、アスペクトを用いて横断的関心事を別モジュールにまとめた事により、元のソースコードからは、横断的関心事に関するコードが消えてしまっている。織り込みを行う前の状態を知る為には、元のソースコードと、横断的関心事をまとめたソースコードとの複数ファイルを確認しなければならなくなる。この為、横断的関心事を別モジュールに分割する前は一つのファイルを確認するだけで知ることが出来た情報が、複数ファイルを確認しなければならなくなってしまう。その為、横断的関心事を一つのファイルに集めて表示を行うツールも必要であると考えられる。しかし、AJDT ではそのようなツールは存在しない。

そこで、本研究ではアスペクト指向言語の一つである GluonJ の為の開発支援ツールとして、編集を行なっているファイルの織り込みの情報を提供する拡張アウトラインビューと、織り込みが行われるクラスを編集している際に、織り込みが行われる関心事の情報を一つのファイルにまとめて表示を行う拡張エディタの二つのツールを提案し、実装を行った。拡張アウトラインビューでは、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の下に、織り込みの情報を階層的に表示する事で、織り込みの情報を提供している。拡張エディタでは、織り込みが行われるメソッドに、織り込みを行うメソッドの情報を複数ファイルから集め、その情報をまとめて表示を行う事で、複数ファイルの情報を一つのファイルで表示する事を実現している。

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の情報を表示する為の支援ツールの必要性と、関連研究について述べる。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、本システムの評価、そして第 6 章でまとめを述べる。

第2章　関連研究

この章では、オブジェクト指向言語ではモジュール化することの出来ない横断的関心事の例を使って、アスペクト指向言語による解決法を述べる。次に、アスペクト指向言語での開発を支援するツールとして必要不可欠な要素を挙げ、既存のツール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、より複雑な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、そうとは言えない。一つのモジュールに分離する事が出来ず、複数のファイルに散在してしまっている関心事が、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。関心事がプログラム全体に散在し、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、これらの横断的関心事を別のモジュールへと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、アスペクト指向言語を紹介していく。

例として、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。最初に、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けられる。このような場合、前者の機能を図形全般を表す親クラスである 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

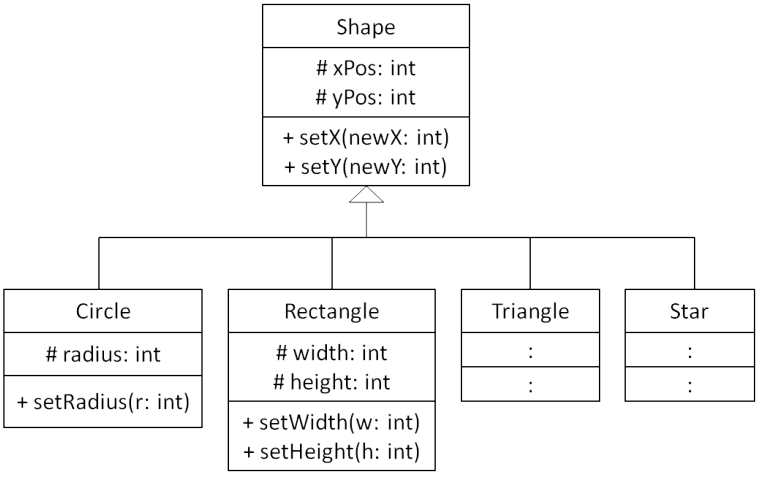


図 2.1: FigureEditor の図形を表すクラスの継承関係

別の機能を定義する。図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を図 2.1 に示した。

```
1  public class Shape {
2     // エディタ上の位置
3     protected int xPos,yPos;
4
5     public void setX(int newX){
6        this.xPos = newX;
7        Screen.repaint();
8     }
9
10    public void setY(int newY){
11       this.yPos = newY;
12       Screen.repaint();
13    }
14       ...
15 }
```

図 2.2: Shape クラスの実装

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 は各クラスのソースコードとなっている。

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

```
1  public class Circle extends Shape {
2     // 半径
3     protected int radius;
4
5     public void setRadius(int r){
6        this.radius = r;
7        Screen.repaint();
8     }
9       ...
10 }
```

図 2.3: Circle クラスの実装

```
1  public class Rectangle extends Shape {
2     // 縦と横の長さ
3     protected int width, height;
4
5     public void setWidth(int w){
6        this.width = w;
7        Screen.repaint();
8     }
9
10    public void setHeight(int h){
11       this.height = h;
12       Screen.repaint();
13    }
14
15    public void regularize(int size){
16       setWidth(size);
17       setHeight(size);
18    }
19      ...
20 }
```

図 2.4: Rectangle クラスの実装

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。そのため、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。

ここで、以下のような状況を考える。

問題事例

> 開発途中で、repaint メソッドの代わりに update という別のメソッドを用いることになった為、repaint メソッドを用いている部分に修正を施したい。

repaint メソッドは、各図形クラスの様々な部分に散らばっている。システムが小規模で、開発者が repaint メソッドを呼んでいる箇所を全て把握できているのであれば、修正することはさほど難しくはない。しかし、大規模なシステムでこのような修正を行おうとした場合、修正を行う部分を探索しなければならなくなり、作業が煩雑になってしまう。探索が足りず、修正すべき場所を修正出来ない。または本来修正するべきではない場所を修正してしまうなどと言った問題も発生しやすい。この原因は、repaint メソッドを呼んでいるコードが複数のファイル間をまたがってしまい、理想的なモジュール化を行えていないためである。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。

オブジェクト指向言語では、このような横断的関心事を一つのモジュールにまとめる事は極めて難しい。複数のモジュールに関心事がまたがってしまうと、大規模なシステムになればなるほど修正が難しくなってしまう。

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へと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、アスペクト指向言語を用いて、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 を用いた解決法

アスペクト指向言語の例として AspectJ[10] 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、コンパイル時に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。まず最初に、AspectJ の機構について紹介する。

- **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、複数のモジュール間をまたがってしまっている横断的関心事をアスペクトという一つのモジュールにまとめることが出来る。

前節で述べた問題は、repaint メソッドが各図形クラスという複数のモジュールに散らばってしまっている為、その修正を行うことが難しいということであった。そこで、前節で述べた問題を解決するために、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。

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 FigureEditor 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ことで、repaint メソッドを呼び出している。図 2.8 がその Repainter アスペクトである。

```
1  public class Shape {
2     // エディタ上の位置
3     protected int xPos,yPos;
4
5     public void setX(int newX){
6        this.xPos = newX;
7        // Screen.repaint();
8     }
9
10    public void setY(int newY){
11       this.yPos = newY;
12       // Screen.repaint();
13    }
14      ...
15 }
```

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

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。処理を織り込むべき時とは、再描画を行うタイミング、すなわち図形の情報に変更があった時である。図形を表すクラスは図形の位置情報、大きさ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 メソッドを介している。すなわち、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

```
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

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 メソッドを update メソッドへと変更したい場合、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。

```
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

2.3　GluonJ を用いた解決法

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。

2.3.1　Reviser

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行なっていた。

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 という注釈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

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。

リバイザを作成して織り込みを実現する場合、織り込みを行いたいメソッドをオーバライドする事で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 メソッドを用いて、織り込む前のメソッドを呼んだ後、再描画処理を

```
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

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に織り込みを実現することが出来る。

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、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。

リバイザをインナークラスとして定義する事で、複数のリバイザを一つのファイル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。

```
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

2.3.2 Require 節

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。

```
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

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。

2.3.3　Within メソッド

GluonJ では、AspectJ のように織り込みを行う時を指定する必要が無い為、オーバーライドしたメソッドが呼び出される時は全て織り込みが行われる。そこで、特定のクラス、またはメソッドから呼び出された時にのみ、リバイザの織り込みを行いたい場合には Within という機構を用いる。例えば、上の例で、FigureEditer クラスの mouseDragged(MouseEvent) メソッドから Shape クラスの set メソッドが呼ばれた時のみ、織り込みを行い、他の場所から呼び出された場合には織り込みは行わないようにしたい時には、図 2.14 のように記述する。

```
1  @Reviser
2  protected class ShapeRepainter extends Shape{
3     @Within(FigureEditor.class)
4     @Code("mouseDragged(MouseEvent)")
5     public void setX(int newX){
6        super.setX(newX);
7        Screen.repaint();
8     }
9     ...
10 }
```

図 2.14: Within を用いた ShapeRepainter リバイザ

Within メソッドを定義するには、@Within と@Code の注釈を用いる。@Within にはクラス、@Code にはメソッドを指定する事で、指定されたクラスとメソッド内から呼び出された時にのみリバイザの織り込みを行う事が出来る。上の例だと、FigureEditor クラスの mouseDragged(MouseEvent) メソッド内から、Shape クラスの setX(int) メソッドが呼ばれた時にのみ、ShapeRepainter リバイザで定義されたソースコードの織り込みが行われる。GluonJ ではこれらの機構を用いて、横断的関心事を一つのモジュールへと分割することを実現している。

2.4　開発支援ツールの必要性

第 2.3 節、第 2.4 節では、アスペクト指向言語を用いて横断的関心事を一つのモジュールへと分割する方法について述べた。アスペクト指向言語により、ソースコード上に存在する横断的関心事の分離は達成された。しかし、分離した事により、ソースコード上から、クラスとアスペクトやリバイザとの関係性を読み取る事が出来ない。この性質を obliviousness と言う。しかし、ソースコード上からクラスとアスペクトやリバイザとの関係性が読み取れないからといって、クラスとアスペクトやリバイザとの依存関係が無くなった訳ではない。横断的関心事をよく理解せずに、プログラムを変更しようとすると、開発者の予期せぬ問題が発生してしまう可能性がある。

代表的な例として、fragile pointcut problem が挙げられる。fragile pointcut problem とは、クラスのソースコードを変更した時に、アスペクトやリバイザが織り込まれるべき場所に織り込まれなかったり、意図しない場所に織り込まれたりしてしまう問題である。例えば、図 2.2 で定義されている setX メソッドを changeX というメソッド名に変更した場合の事を考える。この時、AspectJ では図 2.8 の Repainter アスペクトで定義したジョインポイントの中に、changeX メソッドは選択されなくなってしまう為、アドバイスの織り込みが行われなくなってしまう。また、GluonJ では、図 2.12 の Repainter リバイザ内で定義した setX メソッドがオーバーライドでは無くなってしまう為、こちらでも織り込みが行われなくなってしまう。

このように、クラスとアスペクトやリバイザは、互いの情報をソースコード中に含んでいない場合でも、影響しあっている。したがって、開発者が横断的構造を理解せずにクラスやアスペクトやリバイザに変更を加えると、織り込みが行われるべき場所で織り込みが行われなかったり、意図しない織り込みが発生してしまったりする可能性がある。この問題を避ける為には、開発者がプログラムが持つ横断的構造を常に理解する為に、常にクラスとアスペクトやリバイザのソースコードを照らし合わせながら編集を続けなければならなくなり、開発者の集中力の妨げとなってしまう。その為、アスペクト指向言語での開発する際、織り込みの情報を視覚化する為のツールが必要である。よって、本研究ではアスペクト指向言語での開発を行うに当たって、以下の機能が必要であると考えた。

1. 編集中のファイルの情報 (フィールドやメソッドなど) を表示する機能
2. 織り込みの情報をクラスとアスペクトやリバイザの双方から知る機能
3. 同じ場所に織り込みを行うアスペクトやリバイザ同士の衝突の危険を表示する機能

4. アスペクトやリバイザなど、別のファイルに分割されたコードを一つにまとめて表示する機能

次節では、既存の開発ツールと、機能面での問題点を紹介する。

2.5　既存の開発支援ツール

前節では、アスペクト指向言語での開発支援ツールの必要性について述べた。本節では、既存の開発支援ツールを紹介する。

2.5.1　Java Development Tools(JDT)

JDT[6] は Java 開発を支援する為の Eclipse で標準で組み込まれているツールである。Eclipse[7] とは統合開発環境 (IDE: Integrated Development Environment) の一つでオープンソースのソフトウェアである。統合開発環境とは、プロジェクト管理機能や、エディタ、アウトライン、デバッガなど、アプリケーション開発にあたって必要な機能を備えたソフトウェアである。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ことが出来る重要な要素である。Eclipse の詳細は第 4 章で述べる。

アウトラインビュー

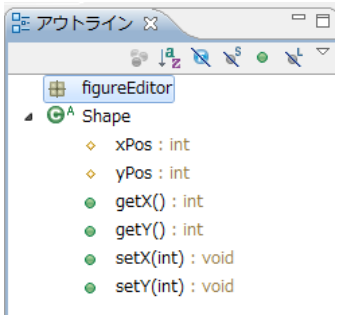


図 2.15: Shape クラスの情報を表示する Java アウトラインビュー

アウトラインビューでは、アクティブになっているエディタで開いているファイルの内容を表示する。図 2.15 は図 2.2 で示した Shape クラスのファイルの情報を表示するアウトラインビューとなっている。Java アウトラインビューは、現在開いているファイルのクラス、フィールド、メソッドの情報を階層上にして表示を行う。このアウトラインビューを使用することで、現在編集しているファイルの情報を知ることが出来るが、どのア

スペクトやリバイザから織り込みが行われるのか、またどのアスペクトやリバイザが織り込みを行うのかを知ることが出来ないという問題がある。

Call Hierarchy

Call Hierarchy ビューは、指定したメソッドを呼んでいるメソッドを表示する、呼び出し関係を表示するビューである。図 2.16 が実際に Call Hierarchy 使った場合の例である。

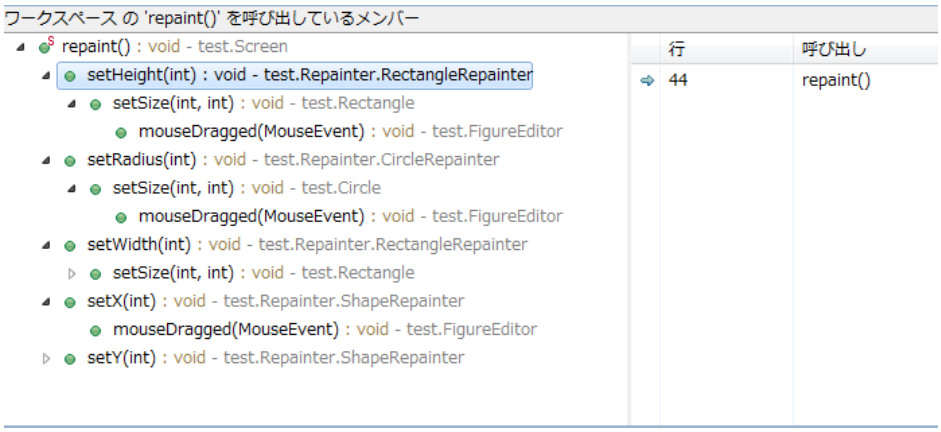


図 2.16: Call Hierarchy ビュー

このビューは指定したメソッドを呼んでいるメソッド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クリックすると、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。このビューを使用する事で、メソッドの呼び出し関係を知る事が出来るが、織り込みの情報を表示する事は出来ない。

Type Hierarchy

Type Haierarchy ビューは二つのビューによって構成されるビューである。まず、図 2.17 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。この図では、Shape クラスを継承しているサブクラスを階層にして表示を行なっている。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。

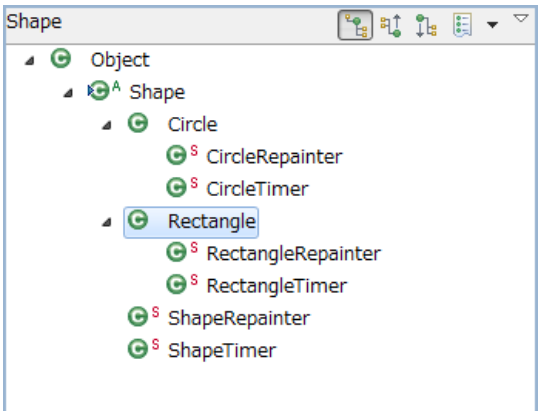


図 2.17: クラスの継承関係を表示する Type Hierarchy ビュー

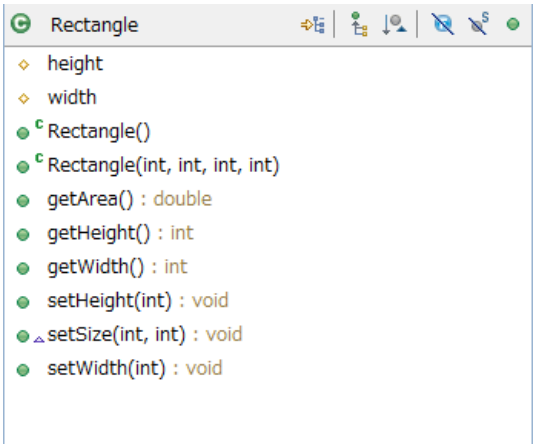


図 2.18: クラスのメンバを表示する Type Hierarchy ビュー

また、図 2.18 で示したビューは図 2.17 で選択されたクラスのフィールドと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ビューであるのに対し、このビューは図 2.17 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。

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、結局織り込みの情報を知る為には、実際にファイルをエディタで開いて確認する必要があるという問題がある。

2.5.2　AspectJ Development Tools(AJDT)

AJDT[5] は AspectJ 開発を支援する為の Eclipse のプラグインである。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。この小節では、AJDT が提供する、織り込みの情報を表示する為のツールについて紹介する。

エディタ

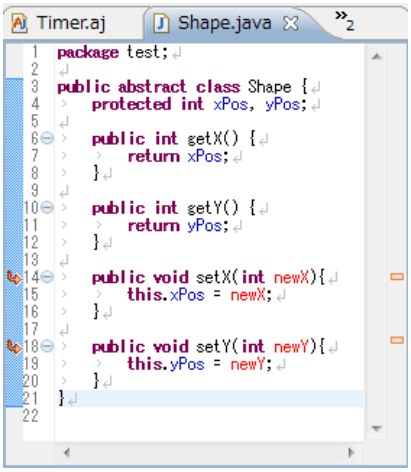


図 2.19: Shape クラスを編集する時の AspectJ エディタ

エディタでは、ソースコード内のどこに織り込みが行われるかを表示する事が出来る。図 2.19 が Shape クラスを開いている時のエディタ、図 2.20 が Repainter アスペクトを開いている時のエディタである。このように、クラスとアスペクトからの両方から織り込みが行われる、織り込みを行うという情報を知る事が出来る。

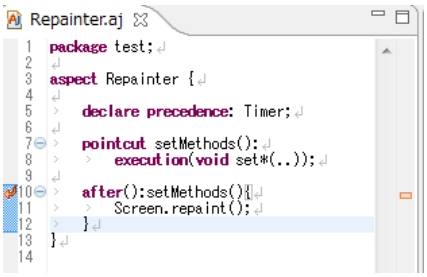


図 2.20: Repainter アスペクトを編集する時の AspectJ エディタ

アウトラインビュー

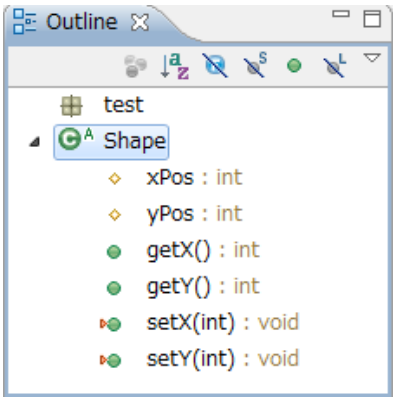


図 2.21: Shape クラスの情報を表示する AspectJ アウトラインビュー

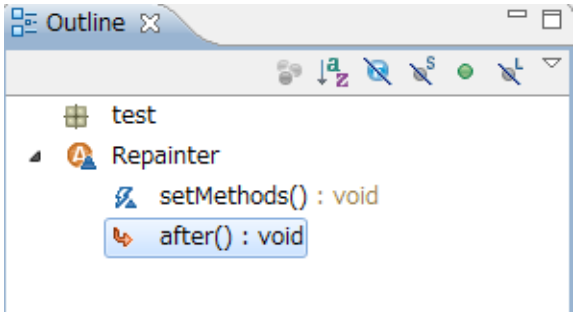


図 2.22: Repainter アスペクトの情報を表示する AspectJ アウトラインビュー

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。AJDT が提供するアウトラインビューでは、クラス側とアスペクト側で表示が異なる。クラス側のアウトラインビューは図 2.21 である。これは Shape クラスの情報を表示しているアウトラインビューである。Shape クラスの setX メソッド、setY メソッドには、メソッド名の前にあるマークに赤い矢印が付けられている。このように、織り込みが行われるメソッドに対して、メソッド名の前にあるマークに赤い矢印をつける事で、織り込みが行われるという情報を提供している。これにより、このメソッドに織り込みが行われるという事を知る事が出来るが、どのアドバイスが織り込まれるかという情報を知る事は出来ない。

また、アスペクト側のアウトラインビューは図 2.22 のようになっている。これは、Repainter アスペクトの情報を表示しているアウトラインビューである。こちらはアスペクト内で定義している内容、ポイントカット名、

アドバイスの種類を列挙している。しかし、どこにアドバイスを織り込むかという情報は提供していない為、その情報を知る事は出来ない。

Cross-References ビュー

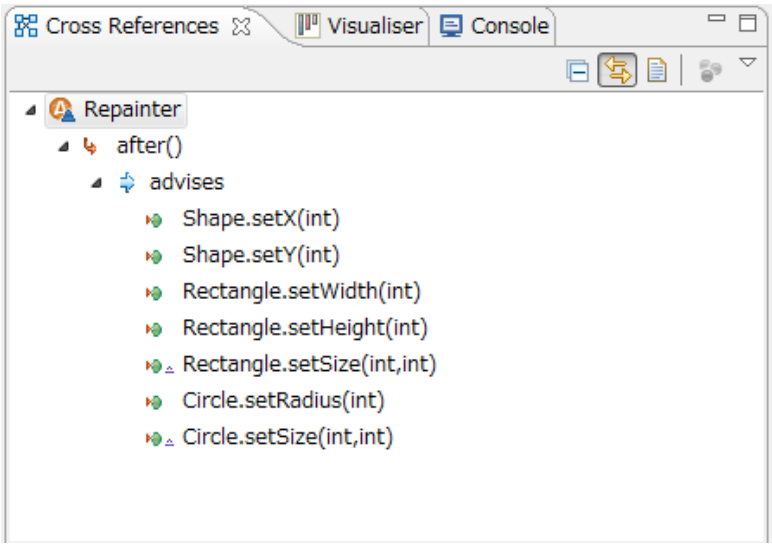


図 2.23: Reapinter アスペクトの情報を表示する Cross References ビュー

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、織り込みの情報を知る事が出来る。

アスペクトに対応する Cross References ビューは図 2.23 のようになる。これは、Repainter アスペクトの情報を表示している Cross References ビューである。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 2.23 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 2.24 の様に、Shape クラスの情報を表示する Cross References ビューへと変化する。

逆に、図 2.24 からは、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、織り込み関係を知る事が出来る。

しかし、Cross References ビューではアスペクトを編集している際、同じ箇所に別のアスペクトからの織り込みが行われる場合、そのアスペクトの存在を知る事が出来ないという問題がある。例えば、図 2.24 では、

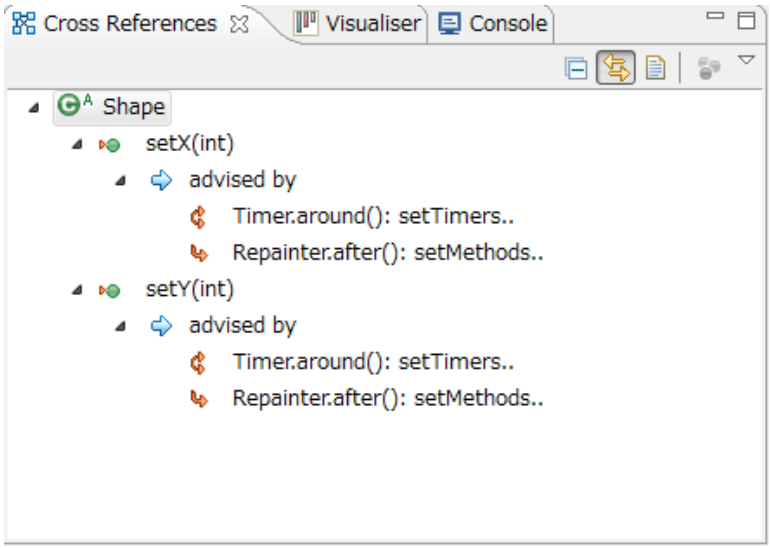


図 2.24: Shape クラスの情報を表示する Cross References ビュー

Shape クラスの setX メソッドに織り込みが行われるのは Repainter アスペクトと Timer アスペクトの二つであるという情報が示されている。つまり、Repainter アスペクトと Timer アスペクトは双方が同じ箇所に織り込みを行なっている為、二つのアスペクトには依存関係が存在している。しかし、Repainter アスペクトの情報を表示している図 2.23 の Cross References ビューには、Timer アスペクトの情報が表示されていない。これは、同じ場所に織り込みを行うアスペクトの情報を表示出来ていない為、開発者は Timer アスペクトの情報を知る事が出来ない。これは、アスペクト同士の衝突の情報を表示出来ていない為、開発者が意図していた動きとは別の挙動になってしまう危険がある。

Visualizer ビュー

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ビューである。

クラスごとに縦の棒グラフが表示され、織り込みが行われる箇所に、横方向で色のついた線が入れられる。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織り込みが行わ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。

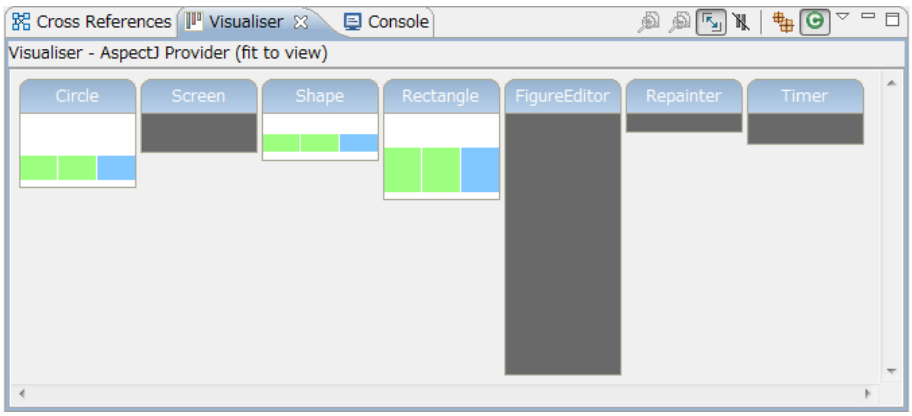


図 2.25: Visualizer ビュー

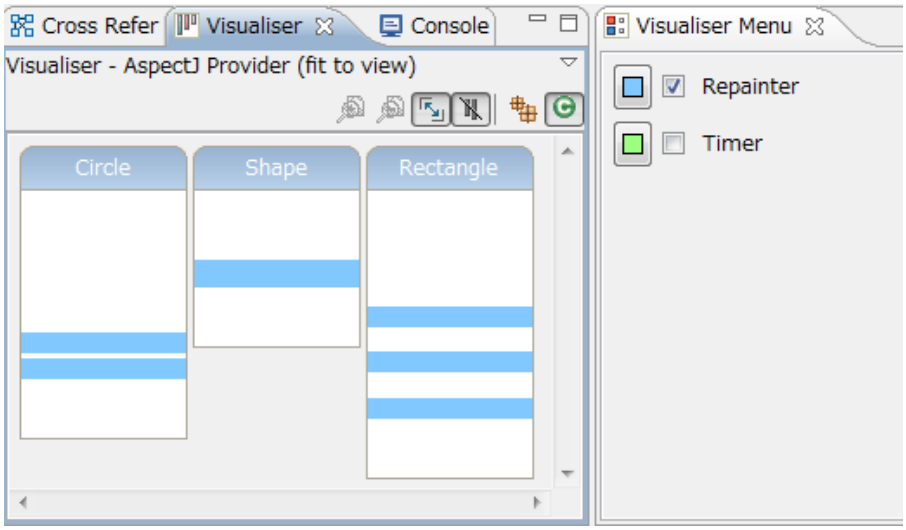


図 2.26: Visualizer Manu ビュー

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。

2.5.3 AspectMaps

AspectMaps[4] はアスペクト指向言語である AspectJ の開発支援ツールである。AspectMaps は次の三つの点に重点を置いて、織り込み関係の可視化を図っている。

- アスペクトがシステム内のどこに適用するように指定されているか、ジョインポイントを基準に可視化する。
- アスペクトが、それぞれのジョインポイントでどのような相互作用を及ぼすかを可視化する。
- 拡張性を考慮して、複数レベルのズーム機能を備えている。

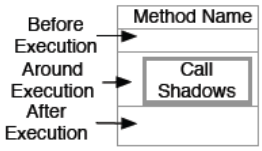


図 2.27: execution ポイントカットの情報を表示する AspectMaps

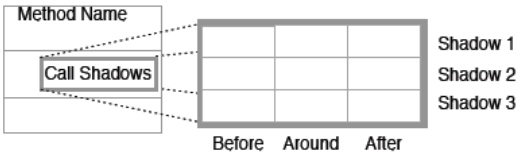


図 2.28: Call Shadows

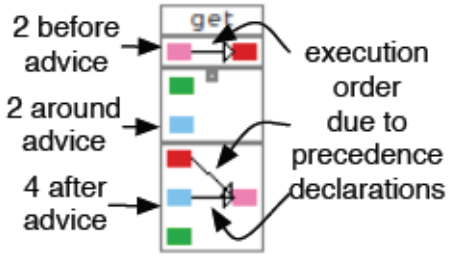


図 2.29: AspectMaps ビュー

図 2.27 は実際の AspectMaps の見方である。メソッド名の下が三段に分かれ、上から順番に before execution、around execution、after exceution のジョインポイントに対応している．また、Call Shadows は図 2.28 のようになっており、こちらは call のジョインポイントに対応している。

実際使用した例が図 2.29 である。アスペクトごとに色が付けられ、それぞれアドバイスがどこで定義されるかを表示する。AspectJ には、declare precedence で織り込みの順番を決める事が出来る。これにより、織り込みの順番が決められたアスペクト同士には、図 2.29 のように、矢印で織り込まれる順番を提供している。

このツールを使う事により、プロジェクト全体の織り込みの情報を知る事が出来る。しかし、現在編集を行なっているクラスやアスペクトの情報であるフィールドやメソッド、ジョインポイントやアドバイスの情報は表示する事が出来ないという問題がある。

2.5.4　Kide

Kide[8, 9] は Java での開発を支援するツールである。通常、エディタはファイル単位で開くが、Kide では、指定した関心事を複数ファイルから集めた仮想ファイルを作成し、そのファイルを開く事を可能にしている。これにより、複数ファイルをまたがった編集を、一つのエディタのみで行うことが出来る。図 2.30 は実際に再描画処理を持ったメソッドだけを複数のファイルから集め、一つの Kide エディタにまとめたものである。このように、Kide エディタでは複数ファイルから、必要な関心事だけを集めて、一つのエディタで表示する事が可能となっている。

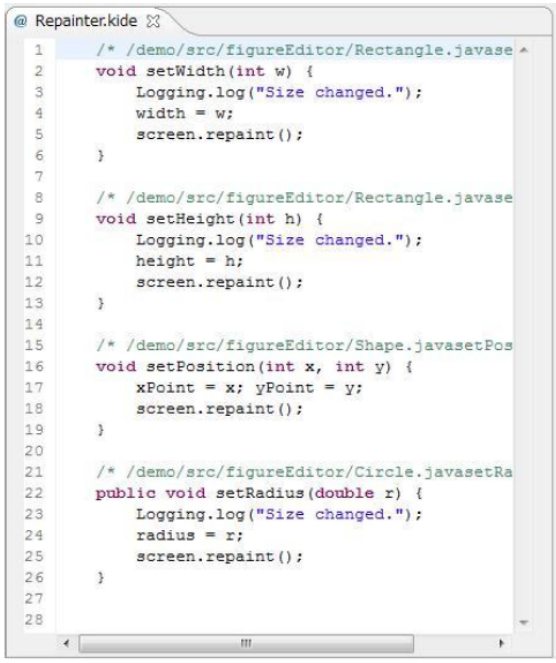


図 2.30: 再描画処理を含むメソッドを集めた Kide エディタ

関心事を集める際には、開きたいメソッドに対して Kide エディタで開く為の関心事として登録する必要がある。関心事を登録する為にはエディタで開いているメソッドの上で右クリックをし、図 2.31 のポップアップメニューを開き、defineConcern を選ぶ事で登録出来る。defineConcern を選択すると、図 2.32 のダイアログが開かれ、どの関心事のグループとして登録するかを入力する。

入力を終えると、図 2.33 のように、入力した名前のグループがあればそのグループとしてメソッドが登録され、無ければ新しいグループとしてメソッドが登録される。登録されたグループをクリックする事で、登録したメソッドを集めた Kide エディタを開くことが出来る。

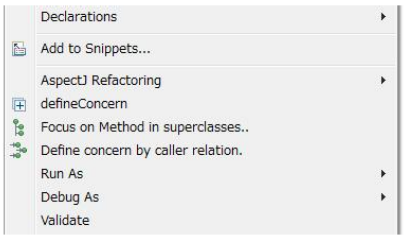


図 2.31: 関心事を登録する為のポップアップメニュー

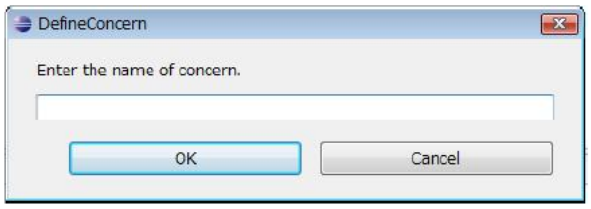


図 2.32: 関心事を登録する為のダイアログ

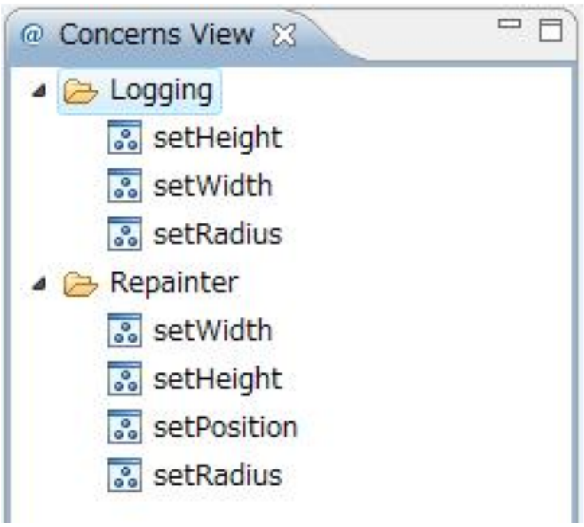


図 2.33: Concerns ビュー

しかし、Kide は Java 用の開発支援ツールである為、アスペクトやリバイザによる織り込みの情報を提供する事は出来ない。

2.5.5　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 2.34 の様になっている。

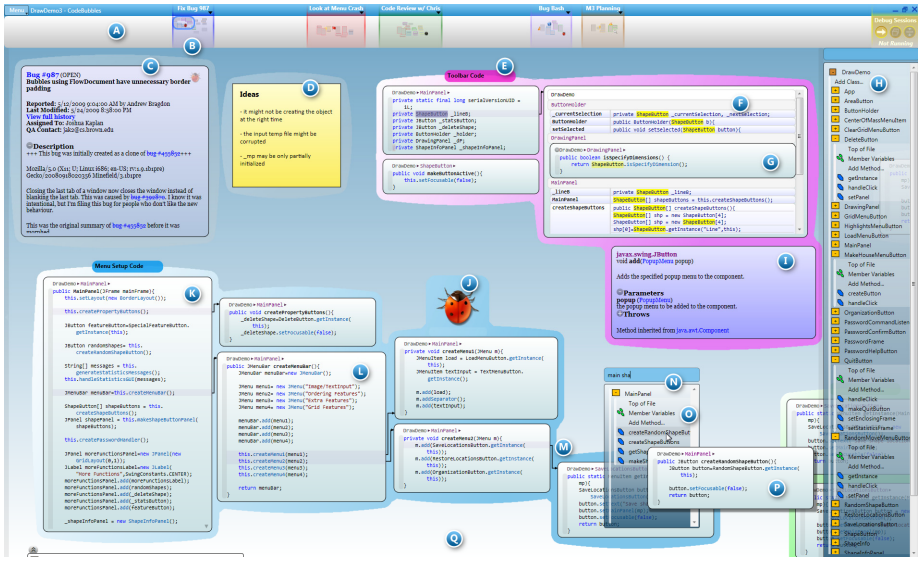


図 2.34: Code Bubbles

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 2.34 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 2.34 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。

ファイル単位ではなく、メソッド単位で開き、編集する事を可能にしている為、関連のある関心事のみを表示し、編集する事が可能となっている。しかし、Code Bubbles は Java 用の開発支援ツールである為、、アスペクトやリバイザによる織り込みの情報を提供する事は出来ない。

2.6　既存の開発支援ツールの問題点

前節では、既存の開発支援ツールについて紹介した。ここで、第 2.4 節で述べた、アスペクト指向言語の為の開発支援ツールに必要な 4 つの機能と、前節で紹介したツールの達成率を、表 2.1 に表した。表 2.1 の 1〜4 の項目は、以下の内容である。

1. 編集中のファイルの情報 (フィールドやメソッドなど) を表示する機能

2. 織り込みの情報をクラスとアスペクトやリバイザの双方から知る機能
3. 同じ場所に織り込みを行うアスペクトやリバイザ同士の衝突の危険を表示する機能
4. アスペクトやリバイザなど、別のファイルに分割されたコードを一つにまとめて表示する機能

ツール	1	2	3	4
JDT	○	×	×	×
AJDT	○	○	△	×
AspectMaps	×	○	○	×
Kide	○	×	×	○
Code Bubbles	○	×	×	○

表 2.1: 既存ツールが提供する機能一覧

JDT はアウトラインビューで現在編集しているファイルの情報を知る事が出来る為、1 は達成出来る。しかし、JDT は Java 用の開発支援ツールである為、他の機能は達成する事が出来ない。

AJDT はアウトラインビュー、Cross References ビューを使用する事で、1 と 2 の機能は提供する事が出来る。しかし、Cross References ビューではクラスを編集している際、アスペクト同士の衝突を表示する事が出来るが、アスペクトを編集している時には、同じ箇所に別のアスペクトからの織り込みが行われる場合、そのアスペクトの存在を知る事が出来ない為、3 は△とした。

AspectMaps はプロジェクト全体の織り込みの関係を知ることが出来る為、2 と 3 の機能は達成出来るが、現在編集しているファイルの情報を表示する事が出来ない為、1 は達成する事が出来ない。

Kide は、JDT を拡張したツールである為、1 は達成する事が出来る。そして、複数ファイルをまたがった関心事を一つのエディタでまとめて表示する事が出来る為、4 も達成出来る。しかし、アスペクト指向言語の為の開発支援ツールでは無い為、2 と 3 は達成出来ない。

最後に、Code Bubbles は bubble を用いる事で、現在編集しているコードの情報と、関連のある関心事のみを集めて表示する事が可能である為、1 と 4 が達成出来る。しかし、こちらもアスペクト指向言語のための開発支援ツールでは無い為、2 と 3 は達成出来ない。

以上から、既存の開発支援ツールでは、第 2.4 節で示した、アスペクト指向言語開発において必要な機能の全てを備えたツールは存在しない。そ

こで、本研究では上で述べた 4 つの機能を全て提供する事の出来るツールを提案する。

第3章　設計

本研究では、前章で述べたアスペクト指向言語の開発支援ツールに必要な4つの機能を備えたツールを開発する為に、アスペクト指向言語であるGluonJの為のアウトラインビューとエディタ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の拡張を行う事で実現している。

3.1　拡張アウトラインビューによるサポート

実際の拡張アウトラインビューが図3.1である。図3.1は織り込まれる方であるクラス側のアウトラインビューとなっている。まず、クラス名の葉に、フィールド、メソッドなど、既存のアウトラインビューと同様にクラスの要素を列挙している。これにより、1は達成される。そして、織り込みが行われるメソッドに対して、葉として織り込みを行うリバイザであるShapeRepainterリバイザの名前を表示している。これにより、クラス側から織り込みが行われるリバイザの情報を知る事が出来る為、2が達成される。

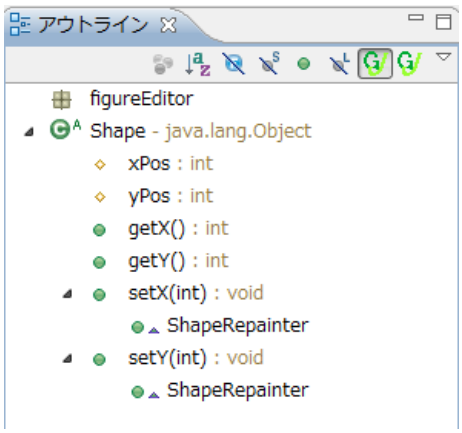


図3.1: リバイザの表示を行う拡張アウトラインビュー(クラス側)

また、同じメソッドに複数のリバイザが織り込みを行おうとしていた場

合は、図 3.2 のように、織り込みが行われる順番でリバイザを列挙するようにしている。図 3.2 の場合では、ShapeRepainter リバイザ、ShapeTimer リバイザの順番で setX メソッド、setY メソッドに織り込みが行われるということを表している。これにより、クラス側から、同じ場所に織り込みが行われる複数のリバイザの存在を順番を含めて知ることが出来る為、3 も達成される。

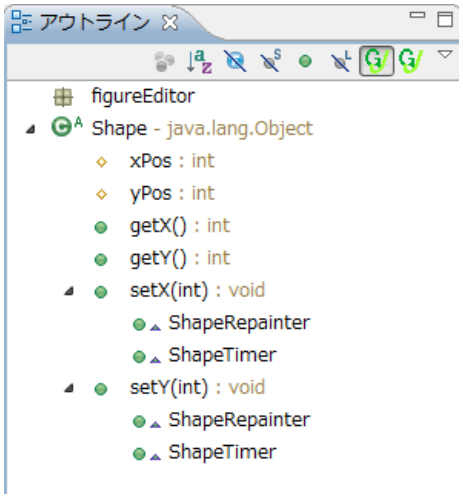


図 3.2: リバイザの順番を表示する拡張アウトラインビュー (クラス側)

一方、織り込みを行うリバイザ側でのアウトラインビューは、図 3.3 のようになっている。リバイザ側でも同様に、まず要素をすべて列挙する事で、1 を達成する。続いて、メソッドにはさらに葉として、そのメソッドに織り込みを行うリバイザを集め、織り込みを行う順番で、リバイザの名前を列挙する。また、リバイザ側では、織り込みを行う対象のクラスを自分のクラス名の隣に表示している。図 3.3 の場合、ShapeRepainter の行にある figureEditor.Shape の事である。これにより、どこのメソッドに織り込みを行うかという情報を知ることが出来、リバイザ側からも、同じ場所に織り込みを行う別のリバイザの存在を知ることが出来る為、2 と 3 が達成される。

以上から、本システムの拡張アウトラインビューは、クラス側の視点とリバイザ側からの視点のどちらの視点から見ても、1 から 3 までの機能を全て備えている為、拡張アウトラインビューを使用することで、ファイルの情報だけでなく、織り込みの情報を全て知る事が出来る。

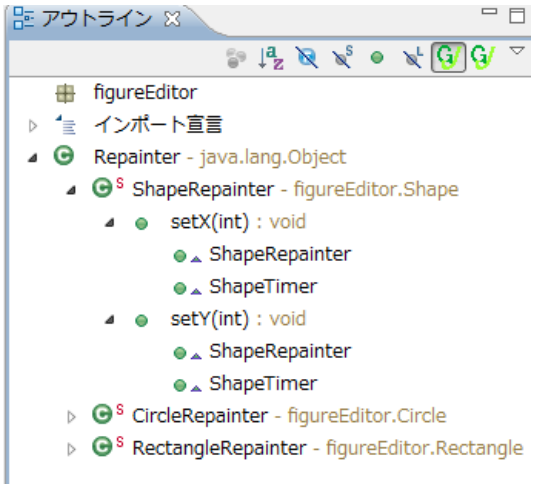
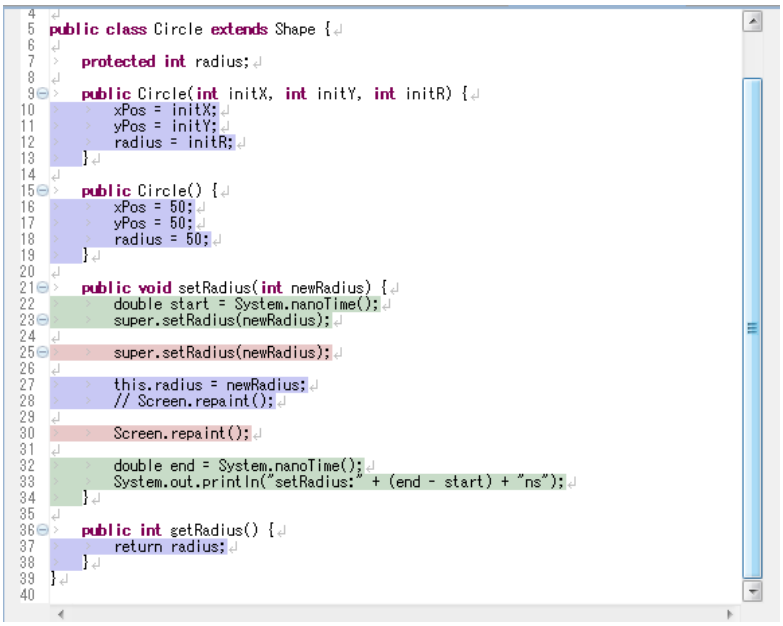


図 3.3: リバイザの情報を表示する拡張アウトラインビュー (リバイザ側)

3.2　拡張エディタによるサポート

エディタに対するサポートについては、クラス側に対してのみ行なっている。リバイザ側からの視点では、リバイザという一つのモジュールに横断的関心事をまとめられている為、関心事が複数ファイルをまたがっていない為である。一方、クラス側の視点では、本来のメソッド内にあった横断的関心事をリバイザという別モジュールへと分割してしまった為、メソッドの関心事はクラスとリバイザの両方のファイルにまたがってしまっている事になる。その為、クラス側に対しては、織り込みを行うリバイザの内容を集めた仮想ファイルを作成し、それを一つのエディタで表示を行うような拡張エディタを作成した。それが、図 3.4 である。

図 3.4 は Circle クラスの情報を集めた拡張エディタとなっている。setRadius メソッドには、CircleRepainter リバイザと、CircleTimer リバイザがこの順番で織り込みが行われる為、一番最後に織り込みが行われる CircleTimer リバイザを setRadius メソッド内の一番外側のコードとして表示を行なっている。そして、次に CircleTimer リバイザ内で super.setRadius メソッドが呼ばれた場合は、その前に織り込みが行われる CircleRepainter リバイザの情報が呼ばれるので、CircleTimer リバイザ内の super.setRadius メソッドを呼んでいる次の行に、CircleRepainter リバイザのコードを表示している。最後に、CircleRepainter リバイザ内で super.setRadius メソッドが呼ばれた場合は、織り込み先となっている Circle クラスの setRadius メソッドが呼ばれる為、これも同様に、CircleRepainter リバイザの setRadius メソッドの次の行に表示を行なっている。また、どのコードがどこのクラ



```
4
5  public class Circle extends Shape {
6
7      protected int radius;
8
9      public Circle(int initX, int initY, int initR) {
10         xPos = initX;
11         yPos = initY;
12         radius = initR;
13     }
14
15     public Circle() {
16         xPos = 50;
17         yPos = 50;
18         radius = 50;
19     }
20
21     public void setRadius(int newRadius) {
22         double start = System.nanoTime();
23         super.setRadius(newRadius);
24
25         super.setRadius(newRadius);
26
27         this.radius = newRadius;
28         // Screen.repaint();
29
30         Screen.repaint();
31
32         double end = System.nanoTime();
33         System.out.println("setRadius:" + (end - start) + "ns");
34     }
35
36     public int getRadius() {
37         return radius;
38     }
39 }
40
```

図 3.4: リバイザの情報を一つのファイルに集めた拡張エディタ

ス、リバイザのコードであるかを識別する為に、背景色を変えることによって区別している。図 3.4 の例では、緑色が CircleTimer リバイザ、赤色が CircleRapinter リバイザ、青色が Circle クラスのコードとなっている。

このように、織り込みが行われるクラスのメソッドに、織り込みを行うリバイザの情報を集めて表示を行う事で、アスペクト指向言語の開発支援ツールとして必要な機能 4 を達成している。

以上から、拡張アウトラインビュー、拡張エディタの二つのツールを使用する事で、アスペクト指向言語の開発支援ツールとして必要な 4 つの機能全てを提供する事が出来る。

3.3　その他のサポート

第 3.1 節、第 3.2 節では拡張アウトラインビュー、拡張エディタについての概要を説明した。この節では、拡張アウトラインビューと拡張エディタが提供している、その他のサポートについて説明する。

Within メソッドに対するサポート

GluonJ では、特定のメソッド内から呼び出された時にのみ織り込みを行う Within メソッドが存在する。拡張アウトラインビューと拡張エディ

タでは、この Within に対するサポートを行なっている。

　拡張アウトラインビューでは図 3.5 のように、Within メソッドを定義したリバイザの葉に Within の情報を階層表示するようにしている。図 3.5 の例では、FigureEditor クラス内の、mouseDragged メソッド内から setX メソッドが呼ばれた時にのみ ShapeRepainter リバイザの織り込みを行うという事を表している。

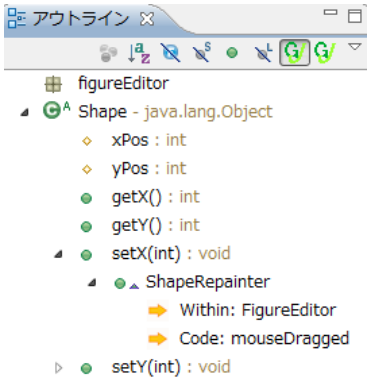


図 3.5: Within メソッドの表示を行う拡張アウトラインビュー

　また、拡張エディタでは図 3.6 のように、Within メソッドで定義されたメソッドに対しては同名メソッドを用意し、Within メソッドに該当するクラス内から呼び出された時のメソッドの挙動と、それ以外の場所での挙動との二つの表示を行なっている。図 3.6 の例では、Select within to FigureEditor.class Select code to mouseDragged と書かれている setRadius メソッドには、CircleRepainter クラスの setRadius メソッドの情報の表示しているが、その上の setRadius メソッドでは、CircleRepainter クラスの setRadius メソッドの情報は表示していない。このように、Within メソッドで定義された場所で呼び出された場合のメソッドの挙動と、それ以外の場所で呼び出された場合のメソッドの挙動との両方の表示を行なっている。

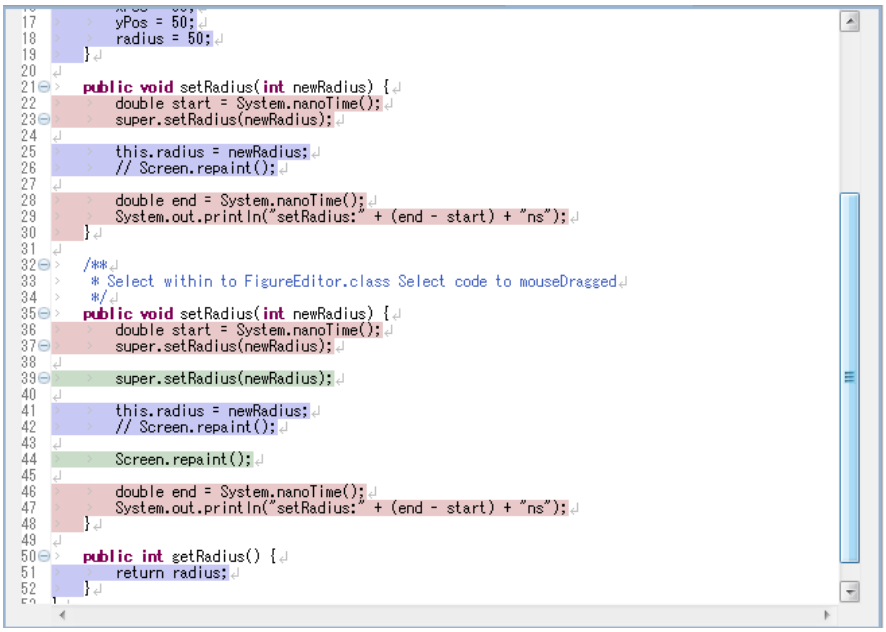


図 3.6: Within メソッドの表示を行う拡張エディタ

織り込みの順番が定まらない場合のエラー表示

GluonJ では、複数のリバイザを織り込む際、Require 節を用いて、リバイザの織り込む順番を定義する必要がある。Require 節によって定義されていなかったり、リバイザの順番が一意に定められない場合には、実行時にエラーとなってしまう。このような問題を避ける為に、拡張アウトラインビューではエラー表示を行うようにしている。

それが図3.7となっている。例えばShapeRepainterリバイザとShapeTimerの織り込みを行う順番がRequire等の情報から一意に定まらない場合には、実行時にエラーが起きてしまう。その為、その情報をメソッドの前に付いているアイコンの色を変える事でリバイザの順番が定められていないという情報を提供している。

また拡張エディタに対しては、リバイザの順番が一意に定められない場合には、表示を行わないようにしている。

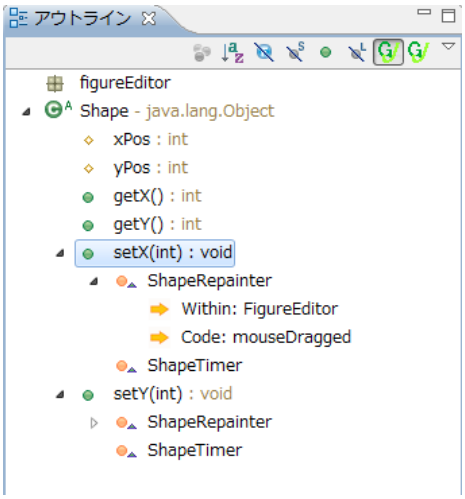


図 3.7: エラー表示を行う拡張アウトラインビュー

要素をクリックすると該当するクラスへとジャンプ

これは拡張アウトラインビューで提供しているサポートである。拡張アウトラインビューでは、織り込みの情報の表示を行なっているが、それぞれの要素の詳しい情報を知りたい時には、その項目をクリックする事で、リバイザのファイルを開き、該当メソッドまでジャンプしたり、織り込み先のクラスのファイルを開いたりする事が可能となっている。

拡張エディタで編集を行った場合の同期

これは拡張エディタ開いた仮想ファイルを編集した場合のサポートである。拡張エディタでは、複数のファイルの情報を集めた仮想ファイルを表示しているエディタである。よって、この拡張エディタで編集した際は、対応するクラスやリバイザのファイルに編集が伝わるようになっている。これにより、拡張エディタで開いた一つのファイルだけで、複数のファイルの編集を行う事が可能である。

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ていく。

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[11, 12]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。

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となる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。

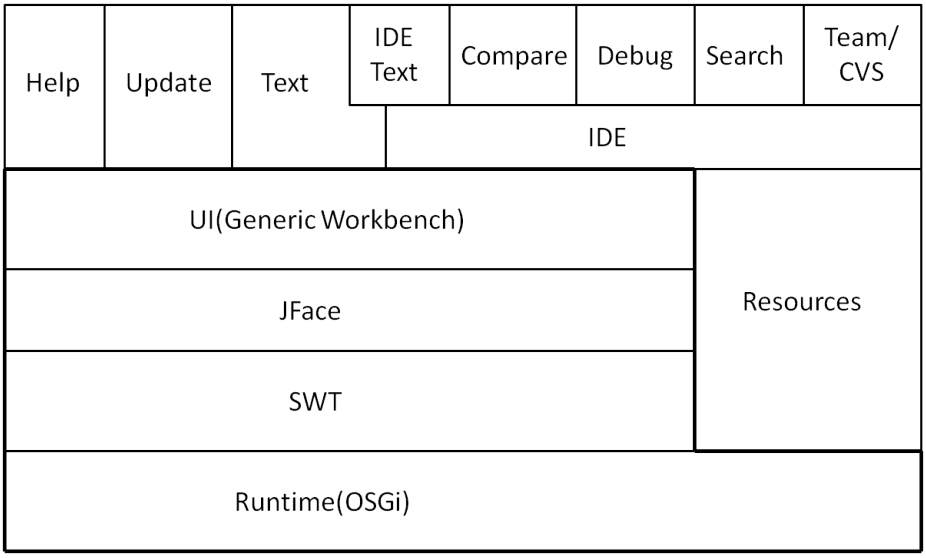


図 4.1: Eclipse アーキテクチャ

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。

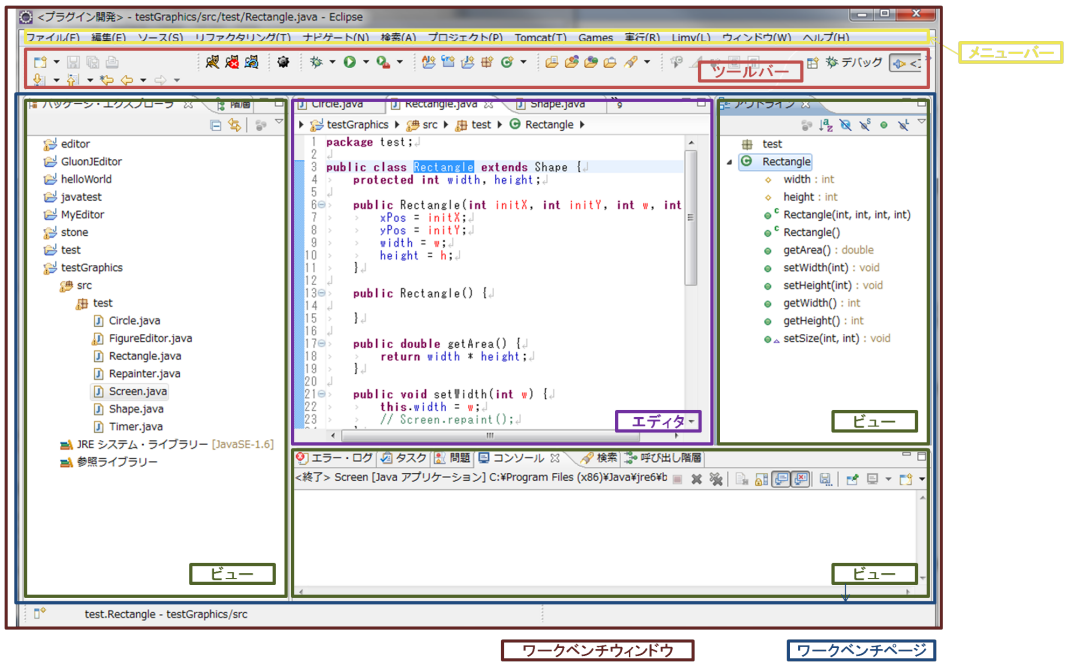


図 4.2: ワークベンチ

- **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　拡張アウトラインビューの実装

本システムの拡張アウトラインビューの実装は、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る為、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 4.6 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

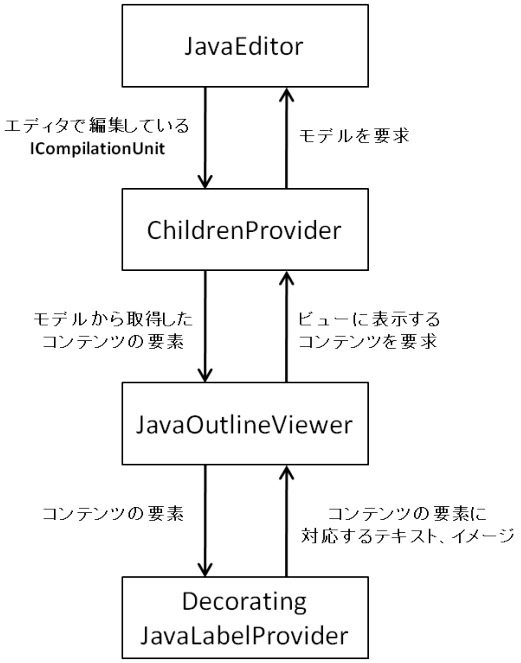


図 4.6: ビューを構成する要素

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、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。@Require が付いているかどうかは図 4.7 と同じように調べる。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。

```
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

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にしている。

4.2.5　階層表示

第 4.2.4 節で述べた方法でリバイザの順序を決定した後、実際にアウトラインビューで表示を行う。まず、通常の Java のアウトラインビューと同じように、現在編集を行なっているクラスのフィールドやメソッドなど、ファイルの情報を階層表示する。次に、織り込みが行う、織り込みが行われるメソッドに対しては、織り込みを行うリバイザの順番で列挙する事で、図 3.2 や図 3.3 のような表示を実現している。

また、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。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

```
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: オーバーライドしているメソッドの取得方法

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。

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、順番が定められないリバイザに対して、図 3.7 のように、メソッドの前に付いているアイコンの色を変えて、表示を行う。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。

4.3　拡張エディタの実装

続いて、本システムの拡張エディタの実装について述べる。拡張エディタの実装も、拡張アウトラインビューと同様に、JDT のエディタを拡張することで実現した。本節では、まず JDT が提供するエディタ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。

4.3.1　JavaEditor

JDT が提供するエディタを実現しているクラスが JavaEditor クラスである。JavaEditor を構成するクラス図を図 4.9 で示した。それぞれのクラスの役割は以下のようになっている。

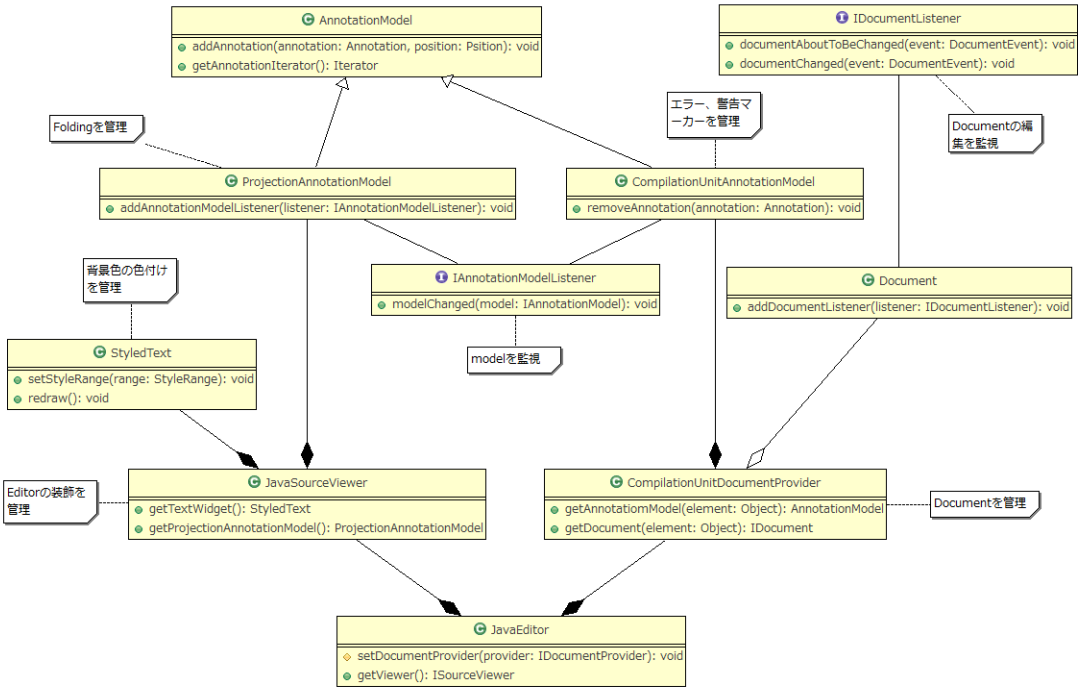


図 4.9: JavaEditor の実装

- JavaEditor

  エディタを提供する為のクラス。これが無いとエディタを実装出来ない。

- JavaSourceViewer

  エディタ上の装飾を一括して管理するクラス。

- StyledText

  文字や背景色など、色付けを行うクラス。StyledRange クラスを追加する事で一部分の背景色を変える事が可能。

- ProjectionAnnotationModel

  Folding を管理するクラス。addAnnotation メソッドを用いて Folding を追加する事が可能。

- CompilationUnitDocumentProvider

  Document を管理しているクラス。

- Document

  Java ソースコードの内容を保持しているクラス。IDocumentListener の実装クラスを追加する事で Document の編集を監視する事が可能。

- CompilationUnitAnnotationModel

  CompilationUnitDocumentProvider の内部クラス。エラーマーカーと警告マーカーの情報を担当。addAnnotation を用いてマーカーを追加する事が可能。

以上が、JDT のエディタを実装しているクラス群となっている。今回、これらのクラスを継承する事で拡張エディタの実装を行なっている。

まず最初に新たな Document のインスタンスを作成し、そのインスタンスに第 4.2.4 節で決定したリバイザの情報から、複数ファイルの情報を一つにまとめる。

次に、JavaEditor を継承した GluonJEditor クラスを用意し、さらに JavaSourceViewer と CompilationUnitDocumentProvider を継承したクラス、GluonJSourceViewer と GluonJDocumentProvider クラスを作成し、GluonJEditor のフィールドに持たせる。

GluonJSourceViewer クラスでは、第 4.2.4 節で決定したリバイザの情報を元に、背景色とフォールディングを行う。また、GluonJDocumentProvider クラスでは、Document を監視する IDocumentListener インターフェースを実装したリスナーを持たせ、エディタでの編集を監視する。編集を感知したら、編集が行われた場所が、どのリバイザのソースコードであるかを判断し、その編集を、元のリバイザを格納しているファイルにも適応させる。これにより、複数のリバイザを一つのファイルにまとめて表示を行い、そのファイルを編集した際に、もとのファイルの方にも編集を通知させることを実現している。

第5章　評価

本システムを用いる事で、GluonJ での開発を行うに当たって、織り込みの情報を知る事が容易になると考えられる。この章では、開発者が織り込みの情報を知るに当たって、閲覧する項目の数が以下に減っているかの評価を行い、それに対する考察を述べる。

5.1　拡張アウトラインビューの評価

本評価では、GluonJ を用いて開発された小さなコンパイラのソフトウェアを題材に用い、拡張アウトラインビューが織り込みの情報を知る際に、いかに余分な項目を排除出来ているのかを計測する。Eclipse が提供している JDT と、拡張アウトラインビューとで、次の 4 つのシチュエーションに対して、見なければならない項目の数を計測する。(1.) リバイザを編集中に織り込み先のクラスのファイルを開きたい時。(2.) リバイザを編集中に同一のクラスに織り込みを行う別のリバイザのファイルを開きたい時。(3.) リバイザを編集中に Within で指定したクラスのファイルを開きたい時。(4.) クラスを編集中にそのクラスに織り込みを行うリバイザのファイルを開きたい時。

JDT で調べる場合は、パッケージエクスプローラーを用いて開くとする。また、計測する際は、現在開きたいファイルが存在するパッケージのみを開き、残りのパッケージは閉じている状態で、表示されているパッケージの数と、ファイルの数を計測する。拡張アウトラインビューの場合は、開きたいファイルへジャンプ出来る項目まで階層を開いた状態で、拡張アウトラインビューで表示されている項目の数を計測する。ただし、インポート宣言の部分は閉じている状態とする。また、それぞれの状態で、ファイル開いている際に、既存の JDT のアウトラインビューで表示されている項目の数も計測した。

その結果は表 5.1 である。(1) から (4) までの全てのシチュエーションにおいて、Java パッケージエクスプローラーで開く時よりも、拡張アウトラインビューを用いて開くほうが、見る項目が少なくて済んでいる。これは、今回評価に用いたプロジェクトが、1 ファイルのコード量よりも、プロジェクト全体のファイル数の方が多い為だと思われる。よって、プロ

	Java パッケージエクスプローラー	Java アウトラインビュー	本システム
(1.)	28.33	9.17	9.17
(2.)	16	11.5	13.5
(3.)	15	12.75	14.75
(4.)	25.4	7.2	10.6

表 5.1: 拡張アウトラインビューの評価結果

ジェクトが巨大であればあるほど、かつ 1 ファイルのコード量が少なければ少ないほど、拡張アウトラインビューの有用性は高くなると考えられる。ソフトウェアを開発する際、しっかりとしたモジュール化を行なっていれば、プロジェクト自体が大きくなったとしても、1 ファイルの担当する処理は少なくて住む為、この拡張アウトラインビューの効果が期待が出来る。

また、既存の JDT が提供するアウトラインビューと拡張アウトラインビューを比較すると、平均 1.85 個の項目が増えている。これは、元のファイルの情報を表示した上で、織り込みの情報を表示するにあたって、関係のない項目を表示することなく、ほぼ的確に求めている情報が表示出来ていると考えられる。

5.2　拡張エディタの評価

本評価では、拡張アウトラインビューと同様、GluonJ を用いて開発された小さなコンパイラのソフトウェアを題材に用いた。Eclipse が提供している JDT のエディタと、拡張エディタとで、クラスを編集する際に関連のあるメソッドの情報知るために開いたファイルのコード行数を計測する。JDT のエディタでは、クラスのファイルと、織り込みを行うリバイザのファイルの行数の合計、拡張エディタでは、関心事を一つのファイルに集めて表示を行ったファイルの行数を計測する。これを、実際に織り込みが行われるクラスに対して、計測を行った。

その結果は図 5.1 である。拡張エディタでは、既存エディタと比べて、閲覧しなければならないコード行数を平均 80 %のコード量をカットする事が出来ている事が分かった。これは、リバイザが複数ファイルをまたがった横断的関心事を一つのファイルにまとめている為、リバイザにはある一つのクラスに関連するコード量以外のソースコードが多く存在している為であると考えられる。

また、一部のクラスでは、20 %程度しかコード量を削減出来ないクラスが存在した。これは、関連するリバイザが、そのクラスにのみ織り込み

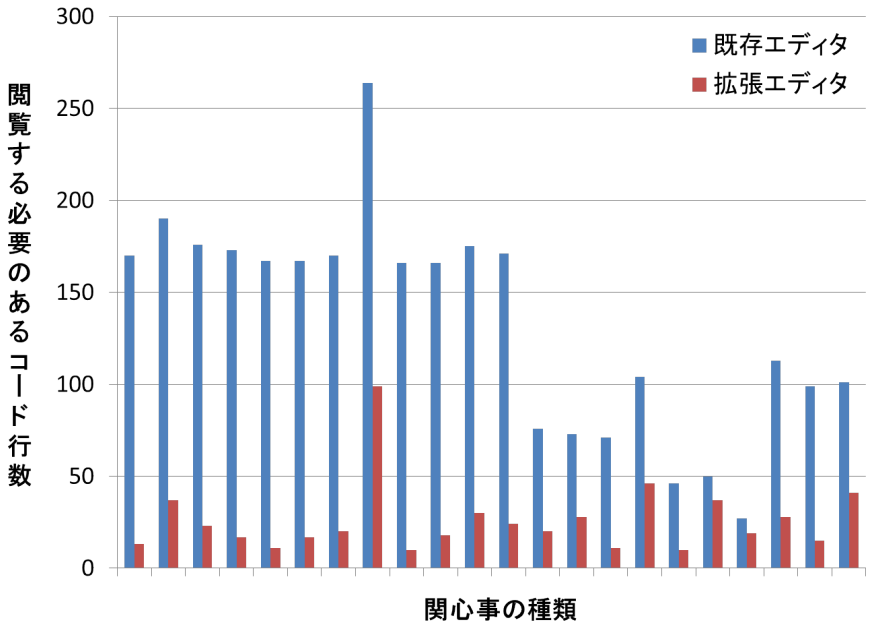


図 5.1: 拡張エディタの評価

を行うリバイザとなっており、削減できたコードがインポート宣言の行数のみとなっている為である。以上からリバイザでまとめた横断的関心事が様々なクラスに散らばっている程、拡張エディタの有用性が高いと考えられる。

第6章　まとめと今後の課題

6.1　まとめ

ソフトウェアを開発する際、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可能である。しかし、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存在する。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ってしまう。アスペクト指向言語によるプログラムの織り込みの情報を把握するには、全てのファイルを開き、織り込みに関連のあるソースコードを実際に読まなければならなくなり、大変効率が悪くなってしまう。この織り込みの情報をより簡単に知るために、アスペクト指向言語での開発をする際には、開発支援ツールが不可欠となっている。

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 に適応した拡張アウトラインビューと拡張エディタを提案し、実装を行った。GluonJ では、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が重要である。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の情報ををアウトラインビューを見るだけで知ることが出来る。さらに、別モジュールへと分割した横断的関心事を集め、一つのファイルで開くことが出来る拡張エディタの実装も行った。これにより、織り込みが行われる元を編集中に、どのような織り込みが行われるかを一つのエディタを開くだけで知ることが出来る。

最後に、本システムの評価として、既存の JDT で開発を行う際に織り込みの情報を把握する為に見なければならない項目の数の比較を行った。JDT に比べて、織り込みの情報を知るに当たり、より少ない項目を見るだけで織り込みの情報を把握することが可能となっており、本システムの有用性を示した。

6.2　今後の課題

今後の課題としては、実際に人に使用してもらい、さらなる評価をする事が挙げられる。リバイザから織り込み先となるクラスのファイルを開くのにかかる時間、クラスから、織り込みが行われるリバイザのファイルを開くのにかかる時間などを、JDT を使用した場合と、本システムを使用した場合とを比較する事を考えている。本システムでは、現在アクティブになっているファイルと関連のあるリバイザやクラスの情報を表示している為、本システムを利用する事で織り込みの情報を知る時間が削減出来るのではないかと考えられる。

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] Fabry, J., Kellens, A. and Ducasse, S.: AspectMaps: A Scalable Visualization of Join Point Shadows, *Proceedings of 19th IEEE International Conference on Program Comprehension*, pp. 121–130 (2011).

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

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

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

[8] 金澤圭, 堀江倫大, 千葉滋: Kide: 流動的なモジュラリゼーションのための IDE サポート, 日本ソフトウェア科学会第 27 回大会 (2010).

[9] 金澤圭, 堀江倫大, 千葉滋: Kide: 開発環境によるオブジェクト指向言語でのアスペクト指向開発の支援, 第 13 回プログラミングおよびプログラミング言語ワークショップ (2011).

[10] 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).

[11] 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).

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