平成21年度 学士論文

関心事ごとに視点を切り替えて プログラムを編集できる 統合開発環境の提案と実装

東京工業大学 理学部 情報科学科
学籍番号 06-0646-4
金澤 圭

指導教員
千葉 滋 教授

平成22年2月3日

概要

今日、ソフトウェア開発にオブジェクト指向技術は必要不可欠である。オブジェクトごとに関心事を分けることができ、プログラムの保守性や拡張性を高められる。しかし、全てのシステムをうまくモジュール化できるわけではない。例えば、ログ出力処理や図形エディタの再描画処理などのオブジェクト間をまたがる関心事（横断的関心事）をうまく単一のモジュールに分離できない。そのため、プログラムを変更する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。本来ならば、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のが理想である。オブジェクト指向よりも強力なモジュール化機能を備えた技術に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アスペクトと呼ばれるモジュールに分離できるため、一箇所にまとめて編集を行える。しかし、一度決定したモジュール構造を変更しにくく、着目する関心事に合わせてその都度視点を切り替えることが難しい。一方、統合開発環境（IDE）が提供する機能を用いるとオブジェクト指向のプログラムでも横断的関心事をひとつの IDE 上にまとめて編集できる。しかし、開発者は複数のエディタや各種のビューアを IDE 上で組み合わせて使用しなければならず効率が悪い。

そこで、本研究では関心事ごとに視点を切り替えてプログラムを編集できる統合開発環境 KIDE を提案し、実装を行った。本提案を用いるとアスペクト指向と同様に横断的関心事を一箇所で編集できる上に、そのモジュール構造を柔軟に切り替えることができる。KIDE は、ユーザーからの要求に応じ適宜横断的関心事を抽出し、それを KIDE エディタ上に表示する。そのため、KIDE エディタ上では通常とは異なる視点からプログラムを閲覧でき、抽出元のファイルとの同期をとることで、モジュール間をまたがる編集を KIDE エディタ上だけで行える。また、関心事を抽出する方法を数種類用意した。例えば、呼び出し側のメソッドやオーバーライドメソッドを、クラスによるモジュール化に関係なくひとまとめにして抽出できる。さらに、抽出候補となったメソッドの中から、実際に KIDE エディタ上に抽出するものを限定できる。KIDE エディタ上で追加的にメソッドを抽出したり、関心のなくなったメソッドを除去することもできるため、視点を柔軟に切り替えられる。

我々は、本システムを Eclipse プラグインとして実装した。Eclipse が提供する拡張ポイントを用いて、関心事を抽出するためのアクションを既存の Java ソースコードエディタに追加した。KIDE エディタは、既存の Java ソースコードエディタを拡張することで作成した。ドキュメントリスナを用いて KIDE エディタ上での編集を監視し、対応する抽出元ファイルにその編集を随時伝えることで同期を行う。抽出時のサポートはウィザードを用いて、抽出後のサポートは KIDE エディタに別箇アクションを加えることで行った。

最後に、本システムを用いて Eclipse の実装を閲覧するシナリオを想定し、KIDE が実際に有用か検証を行った。KIDE エディタを用いる方が、既存の Eclipse だけで探索するよりも、画面の切り替えを少なくコードを辿れることが確認できた。

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有用な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、論文のスタイルファイルを作成していて頂いた九州工業大学の光来健一准教授に心より感謝致します。

堀江倫大氏には、システムの設計と実装方法、発表や論文の組立て方、環境構築など多岐にわたって指導して頂きました。心から感謝致します。

別役浩平氏には、システムの実装や環境構築の面で様々な助言を頂きました。心から感謝致します。

最後に、様々な場面で情報科学に関する知識や研究を進める上での助言を頂いた研究室の先輩の皆様、そして共に研究に励んだ学部生の皆に心から感謝致します。

目 次

第 1 章　はじめに　9

第 2 章　従来手法とその問題点　11
2.1 IDE のビューアのサポートを用いる方法 14
2.1.1 Call Hierarchy View 14
2.1.2 Type Hierarchy View 14
2.1.3 問題点 14
2.2 AOP(Aspect-Oriented Programming) を用いる方法 15
2.2.1 アスペクト指向とは 15
2.2.2 AOP を用いた解決策 16
2.2.3 問題点 18
2.3 関連研究：Fluid AOP Join Point Models 18
2.3.1 概要 18
2.3.2 本研究との関連 20
2.4 関連研究：Mylar 21
2.4.1 概要 21
2.4.2 評価 23
2.4.3 本研究との関連 24

第 3 章　関心事ごとに視点を切り替えてプログラムを編集できる IDE の提案　25
3.1 本システムの概要 25
3.2 本システムの機能 25
3.2.1 抽出の仕方 26
3.2.2 抽出時のサポート 29
3.2.3 その他のサポート 30

第 4 章　実装　32
4.1 Eclipse プラグイン開発の基礎知識 32
4.1.1 Eclipse のアーキテクチャ 32
4.1.2 ワークベンチ 34
4.1.3 PDE(Plugin Development Environment) 37

4.2 本システムの実装 39
4.2.1 Editor の作成 40
4.2.2 コンテキストメニューに新しいアクションを追加 . 41
4.2.3 抽出を行う部分の実装 43
4.2.4 同期処理を行う部分の実装 45
4.2.5 追加機能の実装 52
4.2.6 抽出する断片を限定する Wizard 53

第 5 章 評価 **59**

第 6 章 まとめと今後の課題 **62**
6.1 まとめ 62
6.2 今後の課題 62
6.2.1 機能の強化 62
6.2.2 プログラマを実際に使った評価 63

図 目 次

2.1 DrawEditor の図形を表すクラスの継承関係 11
2.2 Shape クラスの実装 12
2.3 Circle クラスの実装 12
2.4 Rectangle クラスの実装 12
2.5 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 16
2.6 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 17
2.7 再描画処理を行う Repainter アスペクトの実装 17
2.8 Before/After/ITD JPM(論文 Fluid AOP より抜粋) 19
2.9 Gather JPM(論文 Fluid AOP より抜粋) 20
2.10 Overlay JPM(論文 Fluid AOP より抜粋) 20
2.11 企業規模のシステムをビューアでサポートした際の IDE ウィンドウ (論文 Mylar より抜粋) 21
2.12 AspectJ を用いて企業規模のシステムを実装した際の IDE ウィンドウ (論文 Mylar より抜粋) 22
2.13 Mylar を用いて企業規模のシステムを実装した際の IDE ウィンドウ (論文 Mylar より抜粋) 23
2.14 Mylar を用いた実験結果 (論文 Mylar より抜粋) 24

3.1 repaint メソッドを用いているコード断片を抽出した例 . . 26
3.2 図 3.1 で setWidth メソッドを呼んでいるコード断片を追加抽出した例 27
3.3 ColorRectangle クラスの実装 28
3.4 regularize メソッドを実装している断片を抽出した例 . . . 28
3.5 抽出断片を限定する Wizard 29
3.6 断片の除去を行う Popup メニュー 31
3.7 除去する断片を選択するダイアログ 31

4.1 Eclipse のアーキテクチャ 32
4.2 Java モデルの継承関係 33
4.3 Java モデルを用いたコード例 34
4.4 eclipse のワークベンチ 34

4.5 Eclipse の主なパースペクティブ 35
4.6 Eclipse の主なビューア 36
4.7 ワークベンチへのアクセス方法 36
4.8 編集対象のパスを取得する実装 37
4.9 MANIFEST.MF 37
4.10 plugin.xml 38
4.11 PDE のマニフェスト・エディタ 39
4.12 マニフェスト・エディタのタブの種類 39
4.13 作成したエディタの実装 41
4.14 コード断片を文字列として取得する実装 43
4.15 コード断片をファイルとして出力する実装 44
4.16 全ての親クラスを取得する実装 44
4.17 オーバーライドしているメソッドを取得する実装 45
4.18 IMethod から IDocument を取得する実装 46
4.19 IFile から IDocument を取得する実装 46
4.20 同期処理を担当する MyDocumentListener クラスの実装 . 47
4.21 同期処理の流れ 48
4.22 ドキュメントの修正に関する情報を取得する実装 49
4.23 ConcernMethods の調整を行う実装 50
4.24 メソッドのソースを取得する実装 51
4.25 replace メソッドを用いて同期を行う実装 51
4.26 RemoveDialog 52
4.27 RemoveMethodAction 53
4.28 ウィザードの基本構成 54
4.29 MyWizard クラスの実装 54
4.30 MyWizardPage クラスの雛形 55
4.31 MyWizard クラスの createControl メソッドの実装 55
4.32 MyTreeModel の実装 56
4.33 createModel メソッドの実装 57
4.34 ツリービューアにリスナを追加する実装 58

5.1 実験結果 60

表 目 次

4.1 editor タグの属性 40
4.2 viewerContribution タグの属性 41
4.3 action タグの属性 42
4.4 Java ソースコードエディタに追加したアクション 42
4.5 コード断片編集用エディタに追加したアクション 42

第1章　はじめに

今日、開発現場ではオブジェクト指向 (Object Oriented Programming, OOP) が頻繁に用いられている。その理由として、OOP を用いるとシステムの実装をモジュール化しやすく、保守性・拡張性を高められることが挙げられる。開発現場では、クライアントの要求に対し柔軟に対応できるようにこの様な性能が求められる。一般に言われている OOP のメリットは以下の通りである。

- カプセル化が可能
  オブジェクト内の構造を外部から隠蔽することで、個々のオブジェクトの独立性を高めることができる。その結果、オブジェクト内部の仕様変更が外部に影響しにくくなる。

- コードの再利用性が高い
  ソフトウェアが大規模で複雑になったため、短期間で質の高いシステムを作成するには、既存のコードを再利用する必要が高まった。OOP ではカプセル化を行った結果、各オブジェクトの詳細を理解する必要がなくなったため、コードを部分的に再利用しやすくなった。

- メンテナンスが容易
  同じシステムを長い期間使用する際、時間とともに部分的な修正を施す必要が出てくる。その際にシステムのプログラムをモジュール化しておくことでその局所的な修正を行いやすくなる。OOP のモジュール化のしやすさが、メンテナンスの容易性も保証している。

しかし、OOP ではオブジェクト間をまたがる関心事を単一のモジュールに分離できない。そのため、ユーザーはモジュール間をまたがる煩雑な作業を強いられてしまう。アスペクト指向 [6] を用いると横断的関心事をモジュールに分離することができる。しかし、この方法を用いるには新たなプログラミング言語を習得する必要があり、ユーザーの負担が大きい。

そこで、本研究では関心事ごとに視点を切り替えてプログラムを編集できる IDE を提案する。統合開発環境がユーザーの関心事に合わせて新たなモジュールを一時的に生成し、横断的関心事を処理しやすくする。

本稿の残りは、次のような構成からなっている。第 2 章では、既存の技術で横断的関心事を処理する際の問題点を詳しく述べるとともに、関連

研究でのアプローチを考察する。第 3 章では、本システムの提案を、第 4 章では、本システムの実装を述べる。第 5 章では、本システムを用いた場合と用いない場合の、作業効率の比較を行う。そして、第 6 章では、まとめと今後の課題を述べる。

第2章　従来手法とその問題点

本章では、まず、OOP でモジュール化することが難しい横断的関心事の具体例を挙げ、既存の手法では何故うまく対処できないかを述べる。さらに、関連研究を紹介し、この問題にどのようにアプローチしているかを示す。

オブジェクト指向言語の Java を用いて、図形エディタを実装することを考える。まず、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の大きく 2 つに大別される。この様な場合、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、後者のような個別機能を別箇定義する。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の例を図 2.1 に示す。

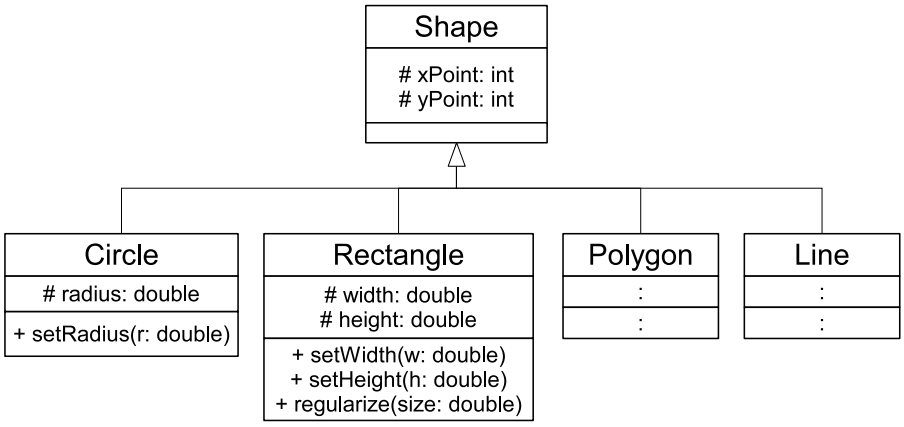

図 2.1: DrawEditor の図形を表すクラスの継承関係

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、Circlr、Rectangle クラスのみ存在するとして話を進める。まず、図 2.2、図 2.3、図 2.4 に各クラスのソースコードを記載する。

ユーザーが図形エディタ上でマウスを用いて各図形の長さを変更した際に、set メソッドが呼ばれフィールドの値を変更する。しかし、各図形オ

```
1  public class Shape {
2    // エディタ上の位置
3    protected double xPoint, yPoint;
4    ...
5  }
```

図 2.2: Shape クラスの実装

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

図 2.4: Rectangle クラスの実装

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。その都度画面に対して、再描画処理を依頼しなくては画面が更新されない。そのため、set メソッドではフィールドの値が更新される度に再描画を通知するための repaint メソッドを呼んでいる。

ここで以下のよう状況を考える。

問題事例 1

開発途中で、repaint メソッドの代わりに update というメソッドを用いることになったため、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部分を把握できるならば修正することは容易だ。しかし、大規模なシステムでこの様な修正を行う場合、修正を行う部分を探索しなくてはならず、作業が煩雑になる。この原因は、言うまでもなく repaint メソッドを呼んでいるコードがモジュール間をまたがっていることである。このようなモジュール間をまたがる関心事を横断的関心事という。

関心事とは

プログラムを作成する際、あるいは設計段階で、自然に抽出されるひとまとまりの処理のこと。このような処理は、1 つのモジュールとしてまとめられるのが自然であり望ましいとされる。上で挙げた Java の場合、クラスやパッケージが 1 つのモジュールである。

横断的関心事とは

処理内容が他の関心事に強く依存して、他の関心事抜きに説明できない関心事のこと。このような関心事は、一部のコードが他の関心事を実装するモジュールの中に散らばってしまう。そのため、横断的関心事を取り除いたり、交換したりする際は、その他のモジュールのプログラムも修正しなくてはならなくなる。

OOP では横断的関心事を 1 つのモジュールにまとめるのは難しい。複数のモジュールに関心事がまたがってしまうと、モジュールをつなぎかえる作業が必要になり、大規模なシステムになればなるほどこの作業による負担は大きくなる。

横断的関心事を分離できないことから生じる問題をもう一つ述べる。

問題事例 2

既存のクラスを継承し、新しいクラスを作成する場合を考える。あるメソッドをオーバーライドするときに、そのメソッドの親クラスにおける実装を把握する必要があったとする。

この場合も、システムの規模が大きいとユーザーが逐一親クラスを辿っていかなくてはならない。

ここから、従来技法でこれらの問題に対処する方法とその問題点について考察を行う。

2.1　IDE のビューアのサポートを用いる方法

近年ソフトウェア開発の場では、ソースコードエディタ・コンパイラ・デバッガなどのソフトウェア開発のサポートを行うツールを一つの環境から利用できるようにした統合開発環境（IDE）を用いることが多い。この IDE の機能を用いると横断的関心事への編集が行いやすくなる。Java IDE である Eclipse[1] を例に挙げ、横断的関心事に対処する方法を示す。

Eclipse は開発者をサポートするための様々なビューアを提供している。その中のいくつかを用いて前記した問題に対処する例を示すとともに、その問題点を挙げる。

2.1.1　Call Hierarchy View

指定したメソッドを呼んでいる Java メンバーを表示するためのビューアである。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ことで、呼び出し関係の階層を作ることもできる。表示されたメソッドを選択すると、そのメソッドの実装部までジャンプすることができる。Call Hierarchy ビューアを用いれば先ほど挙げた**問題事例 1** に対処できそうである。

2.1.2　Type Hierarchy View

一方、クラスの継承関係が関心事である**問題事例 2** に対処するには、Type Hierarchy View を用いるのがよい。このビューアは指定したクラスの親クラスを階層にして表示するビューアだ。Call Hierarchy ビューアのようなメソッドの宣言部までジャンプする機能はないが、小規模なシステムであれば**問題事例 2** に対処できる。

2.1.3　問題点

IDE のビューアを用いれば、横断的関心事に十分対応できるように思える。しかし、大規模なシステムに対応することを考慮すると、問題がある。

ビューアを用いた場合、まず java ファイルを開き、該当する箇所を探し、修正を行う。特に Type Hierarchy View を用いる場合は該当する箇所は開発者が探さなくてはならない。大規模なシステムでは、開発者が複数存在し、自分がコーディングした部分以外に修正をしなくてはならない場合があり、この作業はとても効率が悪い。当然、修正すべき点を見落とす、修正すべきでない部分に変更を加えてしまう、といったことも起こ

りうる。このような開発者に負担をかけるサポートでは十分とはいえない。また、該当するメソッドまでジャンプする機能のある Call Hierarchy View にも問題がないわけではない。最初に述べたように、java ファイルを開く必要があるのだが、今修正を修正を行いたい点はその中のごく一部である。しかし、Java の最小単位は java ファイルであるので、その都度開かなくてはならない。これでは、IDE 上に java ファイルがあふれかえってしまう。もう少し細かい粒度で編集対象を抜き出してくることができないだろうか。

もうひとつ問題がある。Mylar[5] によると、Eclipse を用いた開発では開発者は多くの時間をソースコードエディタ上での編集に費やしている。しかし、このビューアを用いる方法ではその時間が相対的に減少する。システム開発を行う上で、ソースコードを書く時間が減ることはよろしくないことであるのは明らかである。開発者をサポートするのであれば、そのサポートがソースコードエディタ上で終結するのが望ましい。

2.2　AOP(Aspect-Oriented Programming) を用いる方法

強力なモジュール化機能によってプログラムの保守性、拡張性を高めたオブジェクト指向だが、その普及に伴い、限界があることが知られてきた。上記のような横断的関心事がその一例である。そこで、その限界を補い、さらに強力なモジュール化を目指し研究されたのがアスペクト指向 (AOP) である。

2.2.1　アスペクト指向とは

AOP とは OOP ではモジュール化することが困難な横断的関心事をアスペクトとして分離し、モジュール化するためのプログラミング技法である。まず、AOP の概念について説明する。[10, 7]

アスペクト
横断的関心事をひと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。

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ことが可能な時を表す。ポイントカットにより選択される。[8]

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こと。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。

上記のように、AOP とはポイントカットとアドバイス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方法である。

2.2.2　AOP を用いた解決策

この小節では、実際に AOP を用いて**問題事例 1** を解決する方法を紹介する。ここではアスペクト指向言語としてもっとも有名で、広く使用されている AspectJ を用いる。

問題事例 1 では、repaint メソッドが至る所に散らばってしまい、その修正を行うのが大規模システムでは難しい、ということであった。そこで、AspectJ を用い、再描画の処理をひとつのアスペクトにまとめてみる。それぞれの図形クラスとアスペクトは以下のような実装になる。

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

図 2.5: 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

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

図 2.6: 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

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

図 2.7: 再描画処理を行う Repainter アスペクトの実装

まず 2 つのクラスに関しては、問題に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、DrawEditor に反映が行われない。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 2.7 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、その値を変更するには、set メソッドを必ず介す必要がある。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、ワイルドカードを用い、set メソッドが呼ばれた時を指定している。

次に、アドバイスを定義している。まず、先ほど定義した setMathod

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。after アドバイスを用いることで、set メソッドによる値変更という一連の処理が終わった直後、というタイミングの指定ができる。そして、具体的な処理の内容として Editor への再描画依頼のためのコード Screen.repaint() を記述している。

以上で、**問題事例 1** には対処できる。もし、repaint メソッドの代わりに update メソッドを用いたい時には、Repainter アスペクト内のアドバイス 1 箇所を変更すれば修正ができる。問題になっていた横断的関心事をひとつのモジュールにまとめることができた、ということだ。

2.2.3　問題点

AOP を用いて repaint メソッドを呼び出す実装をアスペクトにまとめることができたため、当初の問題であったモジュール間をまたがった編集を行う必要はなくなった。しかし、AOP を用いるには、新たなプログラミング言語知識が必要になる。さらに、アスペクトとして一度決定したモジュール構造を変更しにくく、着目する関心事に合わせてその都度視点を切り換えることが難しい。

2.3　関連研究：Fluid AOP Join Point Models

2.3.1　概要

AOP を用いることで、開発者は横断的関心事の実装をモジュール化することができる。この AOP の利点を言語によるアプローチで生み出したのが AspectJ などのアスペクト指向言語であり、新しいモジュール化のサポートを行うメカニズムを提供する。しかし、アスペクト指向言語には 2 つの制限がある。1 つ目は、AspectJ を用いるとプログラムのモジュールは横断的にできるが、これらモジュールは変わらずそのままであり、簡単に動かせないということ。2 つ目は新しい言語知識が必要になってしまうことである。

Fluid-AOP[4] は、開発環境によって横断的関心事のモジュール化を支援する研究である。このシステムは、相互に干渉しあう横断的な構造をもつシステムを、開発者の要求に応じて一時的に異なる方向から見ることができるようなメカニズムを提供する。また、Fluid AOP では、以下のような 3 つのジョインポイントモデル (利用可能なジョインポイントを規定したもの) を設計している。

Before/After/ITD JPM

before アドバイス、after アドバイス、そしてインタータイプ宣言 (intertype declarations, ITD) をサポートする。ITD は、クラスに新たなメソッドやフィールドを追加する仕組みのことである。before、after アドバイスをジョインポイントに挿入することで、その部分に編集を施すことが可能になる。ITD についても同様で、クラスに導入した部分と ITD の間で連結した編集が可能になる。使用例を図 2.8 に示す。

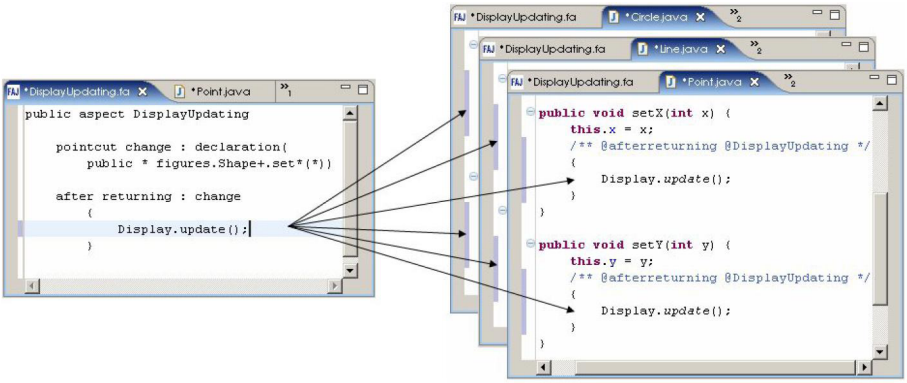

図 2.8: Before/After/ITD JPM(論文 Fluid AOP より抜粋)

Gather JPM

いくつかグループに当てはまる構造を編集可能なビューアにまとめるサポートを行う。このビューアは、元々その構造があった部分のジョインポイントとリンクしてあるため、編集を伝えることができる。使用例を図 2.9 に示す。ロギングに関するを実装を持つ構造集めているが、一番下の setY メソッドのみそれを記述し損ねていることが分かる。このビューアで修正を行い、元の構造にも反映させることができる。

Overlay JPM

重複する構造を持つジョインポイントをビューアにまとめることができる。全てのビューアを表示するのではなく、抽象化された 1 つのビューアが提供される。ジョインポイントを比較する方法は多くあるが、ここでは単純な文字列による比較を行っている。使用例を図 2.10 に示す。内部にロギングコードのある set メソッドを表している。

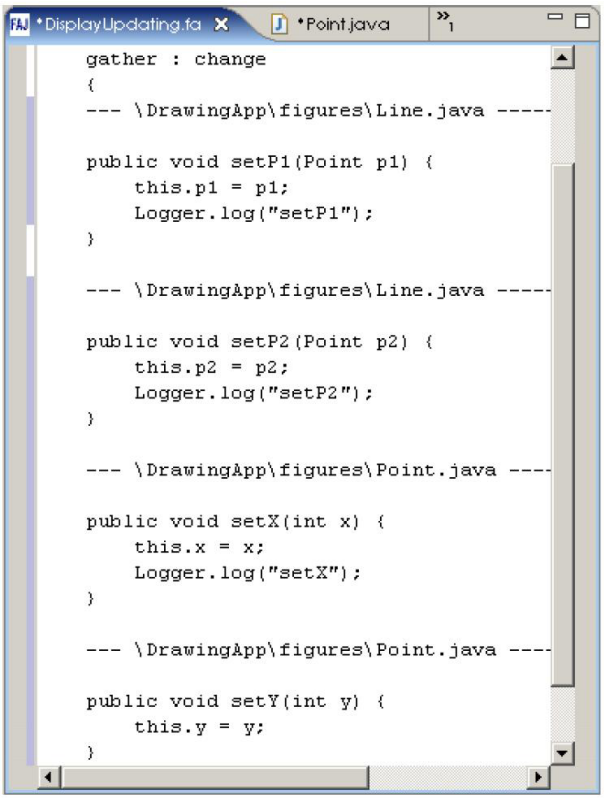

図 2.9: Gather JPM(論文 Fluid AOP より抜粋)

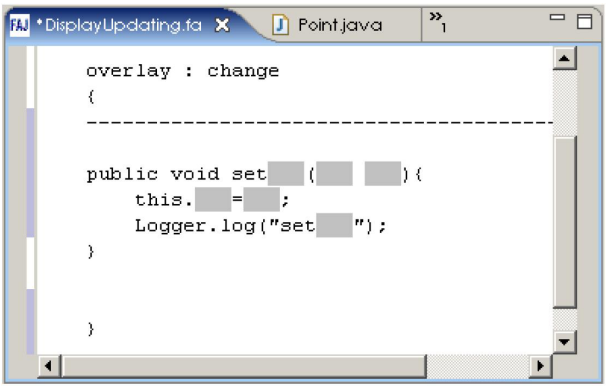

図 2.10: Overlay JPM(論文 Fluid AOP より抜粋)

2.3.2　本研究との関連

Fluid AOP は AOP の利点を言語によってではなく、IDE によって生み出すことを目的としたの研究であり、基本的な考え方は本システムと同

様である。このシステムを用いると、前記した問題にも柔軟に対応できそうだが、現段階でこのシステムを用いるにはユーザーにアスペクト指向言語の知識が必要である。本研究では、OOP の知識のみで IDE のサポートにより AOP の利点を生み出すことを目的としている。

2.4　関連研究：Mylar

2.4.1　概要

大規模なソフトウェアシステムに変更を加えたい時、それが十分にモジュール化されていたとしても、編集に費やす時間よりも多くの時間を編集すべき部分の探索に費やしてしまう。特に、オブジェクト指向言語やアスペクト指向言語ではコードが散らばる傾向があるため、このような事態に陥りやすい。

この様な時にコード探索に用いられるのが IDE のビューアである。しかし、システムが大きければ大きいほど、ビューアの実用性は減少してしまう。例えば、企業向けの 1897 クラスから構成されるシステムにビューアを用いたとする。その時の IDE は図 2.11 のようになる。

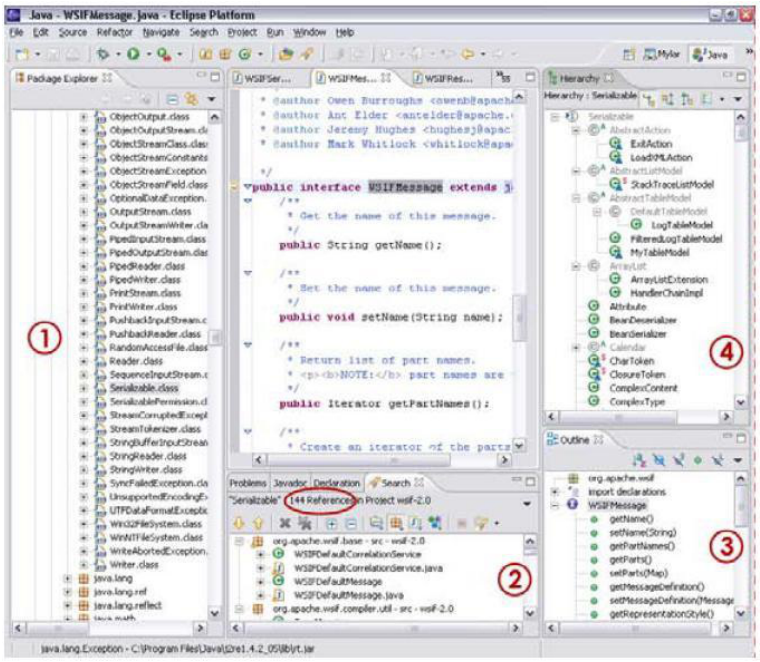

図 2.11: 企業規模のシステムをビューアでサポートした際の IDE ウィンドウ (論文 Mylar より抜粋)

例えば、「Serializable」という単語で Java Search を用いたとすると、144 ものアイテムが検索に掛かり、ビューアが図 2.11② のように溢れかえってしまう。これでは、あまり実用的とは言えない。ほかのビューアに関しても大規模システムになると役に立たない。このようにビューアが溢れかえってしまう原因の一つが、ユーザーの関心事が横断的関心事であることである。コードが分散するために、必要以上に多くのファイルが検索に掛かってしまっている。そこで、AspectJ を用いて解決を試みた結果が図 2.12 である。

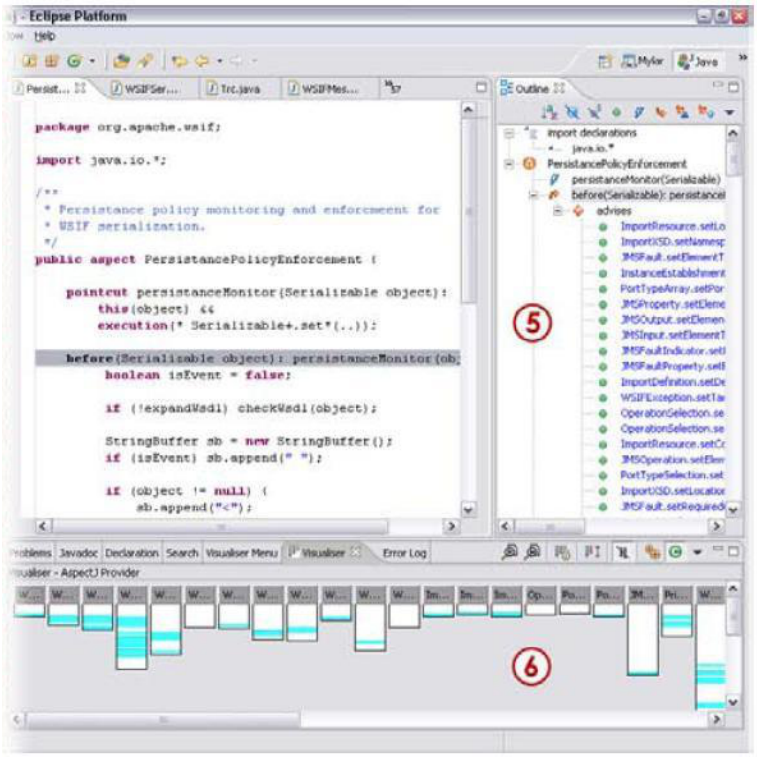

図 2.12: AspectJ を用いて企業規模のシステムを実装した際の IDE ウィンドウ (論文 Mylar より抜粋)

コードのモジュール性は向上したが Java を用いた際と同じ問題が生じてしまう。結局、Java を用いても AspectJ を用いても IDE のビューアのサポートではユーザーのタスクに関連した構造だけを示すことは難しいことが分かった。そこで開発されたのが Mylar[5] である。Mylar は開発者の Eclipse 上での作業を監視し、関心度の高いものを優先的に Pachage Explorer や Outline に表示させるシステムである。Mylar を用いて、先ほどの大規模システムを IDE 上で編集したものが図 2.13 である。

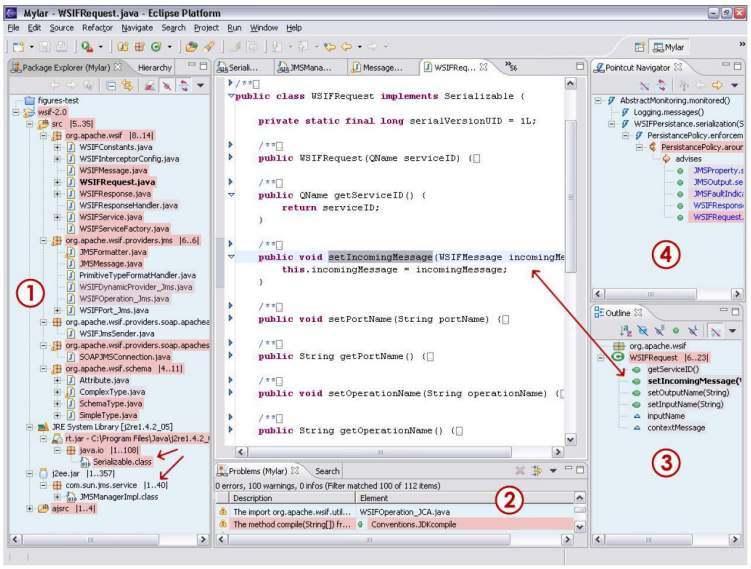

図 2.13: Mylar を用いて企業規模のシステムを実装した際の IDE ウィンドウ (論文 Mylar より抜粋)

各ビューア内の情報を限定したことで縦方向のスクロールバーがなくなっている。ビューア内でファイルやメンバを探す負担を軽減させ、ユーザーはエディタ上での作業に集中することができる。

ユーザーの関心の高さを測るために、Mylar では、degree-of-interest(DOI) という開発者の関心度を表すモデルを用いている。DOI は Java や AspectJ の各要素に関心値という値を与える。ある要素が IDE 上で選択されたり、編集されたりした時に、対応するその要素の関心値を上げる。逆に、選択や編集の対象にならなかった要素の関心値は時間とともに減少させる。そして、一定の時間ごとに関心値の高い要素を各ビューアに表示させる。

2.4.2　評価

Usage statistics

この研究では、アプローチが正しいことを示すために、実際に IBM に所属するプログラマに Mylar を用いてもらう実験を行っている。57 時間分の eclipse の使用ログをとり、エディタと各ビューアを選んだ回数をカウントしている。その結果を図 2.14 に示す。

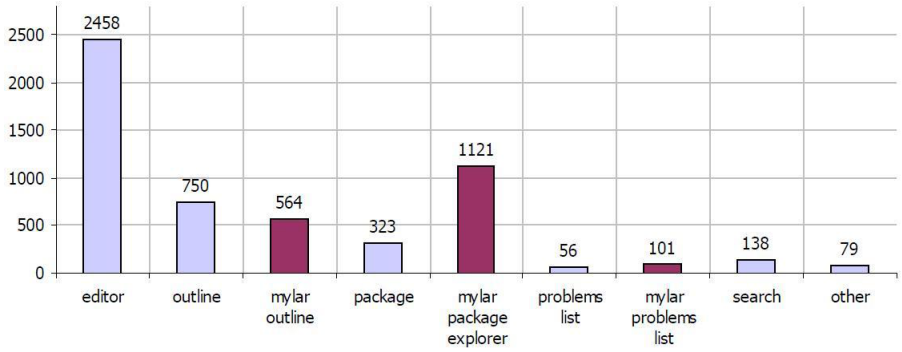

図 2.14: Mylar を用いた実験結果 (論文 Mylar より抜粋)

青いグラフが既存の eclipse のエディタとビューア、赤いグラフが Mylar が提供するビューアである。package explorer と problems list に関しては、既存のものよりも Mylar が提供するユーザーの感心に対応するビューアの方が好まれていることが分かる。outline に関しては逆の結果になっているが、これはプログラマのうちの 1 人が Mylar の outline の使用方法を知らずに使っていたためだそうだ。

Edit ratio

この研究はユーザーにエディタでの編集に集中してもらうサポートをするのが目的であると前記した。Mylar を用いることでこの目的が達成されたかを調べている。開発者が開発時間の中でエディタ上でのキータイプに割いた時間を edit ratio と定義すると、その比率が Mylar 導入前に比べて 15 パーセント大きくなっている。それだけコード探索に費やす時間を減らせたということだ。

2.4.3　本研究との関連

Mylar ではエディタに対する特別なサポートは行っていない。しかし、本研究の目的であるユーザーの関心によってシステムの構造を異なる視点から閲覧させるという目的と、ユーザーの関心に合わせてビューアの表示を柔軟変えるという点が類似していると考えられる。

第3章　関心事ごとに視点を切り替えてプログラムを編集できるIDEの提案

前章で、既存の手法では流動的に変わる横断的関心事に対して対応することが難しいことと対応するには新たな言語の知識が必要なことを述べた。そこで本研究では、開発者がオブジェクト指向の知識のみで目的を達成するために、関心事ごとに視点を切り替えてプログラムを編集できる crosscuttingIDE(KIDE) の提案と実装を行った。実装の詳細に関しては４章に記載する。本章ではその提案・システムの概要と設計について記載する。

3.1　本システムの概要

Eclipse などの IDE では、Java の最小モジュール単位である Java ファイルごとにソースコードエディタが開かれる。しかし、エディタの表示は、言語が提供するモジュール構造に合わせる必要はない。言語のモジュール構造に合わせた管理は IDE の内部で行うが、その構造とは別の視点でコードを閲覧、編集できるような機構があってもよい、ということだ。しかし、このような機構は既存の Eclipse には存在しない。そのため、本研究の目的である横断的関心事への対応が難しくなってしまっている。そこで、本研究では Java ファイル単位という従来の視点以外の見方ができる新しいソースコードエディタ KIDE エディタを既存の IDE に組み込む拡張を行った。新たな言語の知識なしに、アスペクト指向の利点を IDE 上で生み出すことを目的としている。

3.2　本システムの機能

本システムを用いることで、ユーザーの望む関心事に対応したコード断片を KIDE エディタ上に抽出できる。KIDE エディタは、既存の Java ソースコードエディタと同期をとるので、開発者の編集対象の推移に応じ

て既存のソースコードエディタと KIDE ソースコードエディタを使い分けることが可能である。

本節では KIDE の具体的なの機能について述べる。

3.2.1　抽出の仕方

KIDE エディタにコード断片を抽出する方法を 2 つ用意した。

抽出方法：呼び出し関係にあるメソッド

既存の Eclipse の Call Hierarchy View に相当する抽出方法である。既に完成しているシステムのソースコードを閲覧、編集する際の利用を想定している。

前章の**問題事例 1** に対してこの抽出を用いてみる。この事例では既存システムの repaint メソッドを update メソッドに修正したい、ということであった。この時の関心事は repaint メソッドを使用しているコードであるので、対応するコード断片を 1 つの KIDE エディタに抽出し、編集することにする。この場合に生成される KIDE エディタを図 3.1 に示す。

```
1   /* ../../Circle */
2   public void setRadius(double r){
3     this.radius = r;
4     Screen.repaint();
5   }
6
7   /* ../../Rectangle */
8   public void setWidth(double w){
9     this.width = w;
10    Screen.repaint();
11  }
12
13  /* ../../Rectangle */
14  public void setHeight(double h){
15    this.height = h;
16    Screen.repaint();
17  }
```

図 3.1: repaint メソッドを用いているコード断片を抽出した例

既存システムのように、Java ファイル単位ではなくメソッド単位という細かい粒度でコードを扱うことができるため、関心事に対応するメソッドをひとつの KIDE エディタ上で閲覧・編集することが可能である。この事例であれば、この KIDE エディタ上だけで全ての repaint メソッドを update メソッドに変更することができる。また、repaint メソッドか

ら update メソッドに変更するものと、そうでないものがあった場合も、local 変数などのメソッド内部の実装を見ながら編集を行える。もし、Java ファイル全体を閲覧しなくてはならない場合は、コメントの抽出クラスを記述している部分からジャンプすることができる。

また、KIDE エディタのコンテキストメニューに AddMethods というアクションを追加している。このアクションを用いることで、Call Hierarchy View と同様、KIDE エディタ上でさらに呼び出し関係のあるメソッドを抽出することが可能である。例えば、図 3.1 で setWidth メソッドを呼んでいるものを抽出したいと考えた場合にこのアクションを用いると KIDE エディタは以下のように変更される。

```
1   /* ../../Circle */
2   public void setRadius(double r){
3     this.radius = r;
4     Screen.repaint();
5   }
6
7   /* ../../Rectangle */
8   public void setWidth(double w){
9     this.width = w;
10    Screen.repaint();
11  }
12
13  /* ../../Rectangle.regularize -> ../../Rectangle.setWidth */
14  void regularize(double size){
15    setWidth(size);
16    setHeight(size);
17  }
18
19  /* ../../Rectangle */
20  public void setHeight(double h){
21    this.height = h;
22    Screen.repaint();
23  }
```

図 3.2: 図 3.1 で setWidth メソッドを呼んでいるコード断片を追加抽出した例

図 3.2 の 13 行目から 19 行目が加わった regularize メソッドである。addMethods で追加されたものには、どのメソッドがどのメソッドを呼んでいるという情報がコメントとして付加される。このコメントからも元の Java ファイルにジャンプすることができる。また、さらに regularize メソッドを呼んでいるメソッドを抽出、ということも可能である。

抽出方法：オーバーライドしたメソッド

指定したメソッドの親クラスにおける実装コードを全て抽出する。この抽出方法は、既存のクラスを継承し、新しいクラスを作成する際に利用することを想定している。

ここでは**問題事例 2** の状況を想定して KIDE エディタを用いてみる。例えば、図 2.4 の Rectangle クラスを継承した ColorRectangle クラスを作成することになったとする。regularize メソッドでフィールド coler への操作も加えるため、既存の regularize メソッドをオーバーライドしなくてはならない場合を考える。ColorRectangle クラスの大枠は図 3.3 のようになるはずである。

```
1 public class ColorRectangle extends Rectangle{
2   Color color;
3
4   @Override
5   void regularize(double size){
6
7   }
8  ...
9 }
```

図 3.3: ColorRectangle クラスの実装

ColorRectangle クラスの regularize メソッドを具体的に実装する際に、親クラス Rectangle の regularize メソッドの実装を見ながら作業したい場合に、この抽出方法を用いる。ここで生成される KIDE エディタを図 3.4 に示す。

```
1   /* ../../Rectangle */
2   void regularize(double size){
3     setWidth(size);
4     setHeight(size);
5   }
6
7   /* ../../ColorRectangle */
8   void regularize(double size){
9
10  }
```

図 3.4: regularize メソッドを実装している断片を抽出した例

Rectangle クラスを別のソースコードエディタで開かずとも、編集を行うことができる。ここでは、継承関係が小さいため 2 クラス間のメソッド

のみ抽出されたが、さらに上のクラスで regularize メソッドの実装を行っているクラスがあった場合は、その断片も抽出することができる。

3.2.2　抽出時のサポート

抽出断片の限定

3.2.1 で抽出の方法を示したが、そのコンテキストメニューから抽出を行う際に図 3.5 のような Wizard が現れる。この Wizard の左側のツリービューアを用いることで、指定したパッケージやクラスに属する断片のみを限定して抽出することができる。

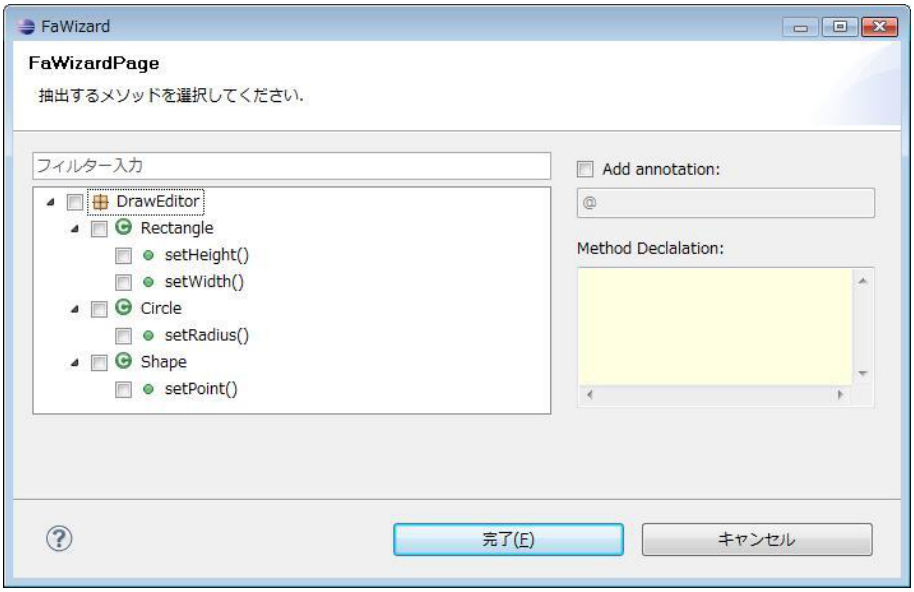

図 3.5: 抽出断片を限定する Wizard

例えば、パッケージにチェックを入れれば、そのそのパッケージ内にあるコード断片を全て KIDE エディタに抽出することができる。この限定を用いることで、より柔軟にユーザーの関心事に対応することが可能となる。

また、ツリービューアの上に正規表現フィルターを用意した。抽出候補が莫大になった際に、例えば「set*」メソッドだけを抽出したいというような AspectJ のワイルドカードの役割を果たす。

アノテーションの挿入

抽出したメソッドを抽出する際にアノテーションを挿入できるようにした。図3.5の右上のAdd annotationにチェックを入れ、テキストボックスで挿入するアノテーションを指定できる。アノテーションはKIDEエディタ上でも編集が可能であるので、不必要になれば抽出後に取り除くことも可能である。

3.2.3　その他のサポート

KIDEエディタにコード断片を抽出した後の主なサポート機能を以下に記載する。

断片の除去

関心のなくなったコード断片をKIDEエディタ上から除去する機能を追加した。この機能は、KIDEエディタのコンテキストメニューから実行することができる。KIDEエディタのコンテキストメニューを図3.6に示す。

図3.6の「Remove Concern Methods」から図3.7のような除去する断片を選択するためのダイアログを開くことができる。

抽出元との同期

KIDEエディタに抽出されたメソッドに対し修正を加えると、そのメソッドの抽出元にもその修正が反映される。このような同期処理機能を搭載したので、ユーザーの関心事に合わせたコード断片を抽出し閲覧するだけではなく、その抽出先での編集も行うことができる。

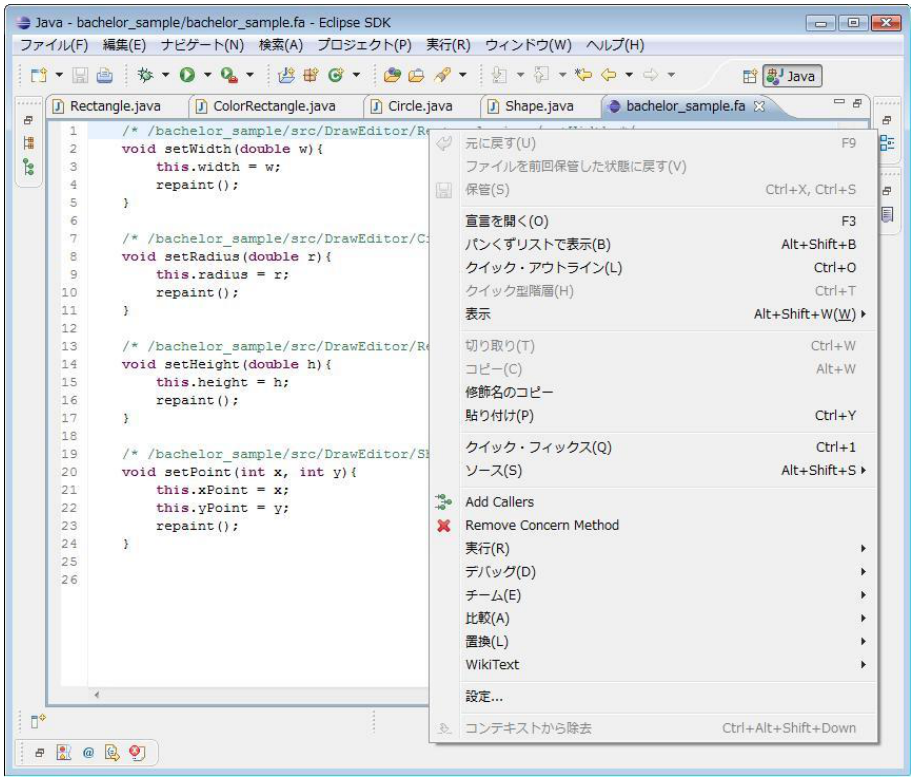

図 3.6: 断片の除去を行う Popup メニュー

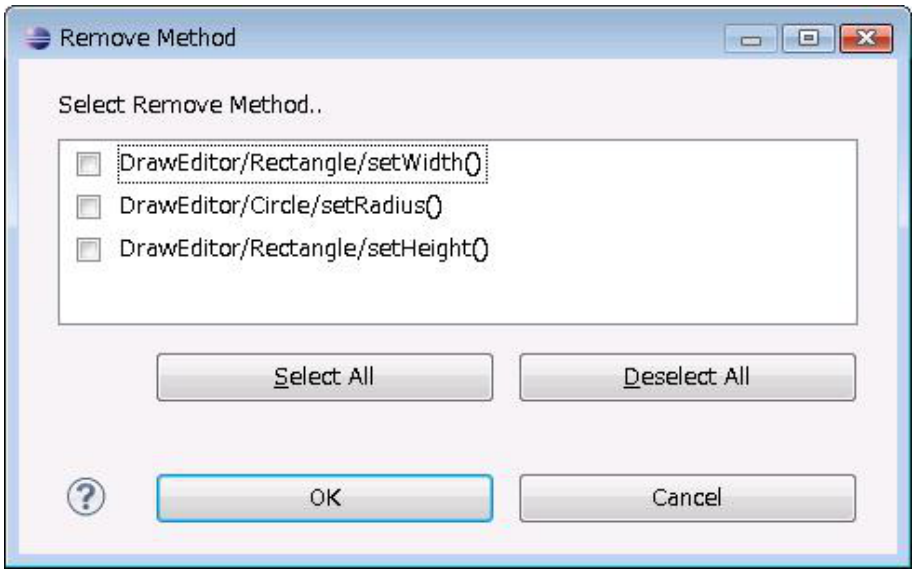

図 3.7: 除去する断片を選択するダイアログ

第4章　実装

我々は、本システムを Eclipse プラグインとして実装をした。この章では、まず Eclipse プラグイン開発の基礎知識ついて述べ、その後に本システムの実装について詳しく述べる。

4.1　Eclipse プラグイン開発の基礎知識

Eclipse プラグイン開発を行う上で知っておくべき事項である Eclipse のアーキテクチャ、ワークベンチ、そして PDE(Plugin Development Environment) について述べる。[11]

4.1.1　Eclipse のアーキテクチャ

Eclipse は Java の IDE として広く用いられているが、本来は様々な開発ツールのための統合プラットフォームを提供することを目的としている。そのため、図 4.1 のような自由に機能を追加できるプラグイン・アーキテクチャを採用している。

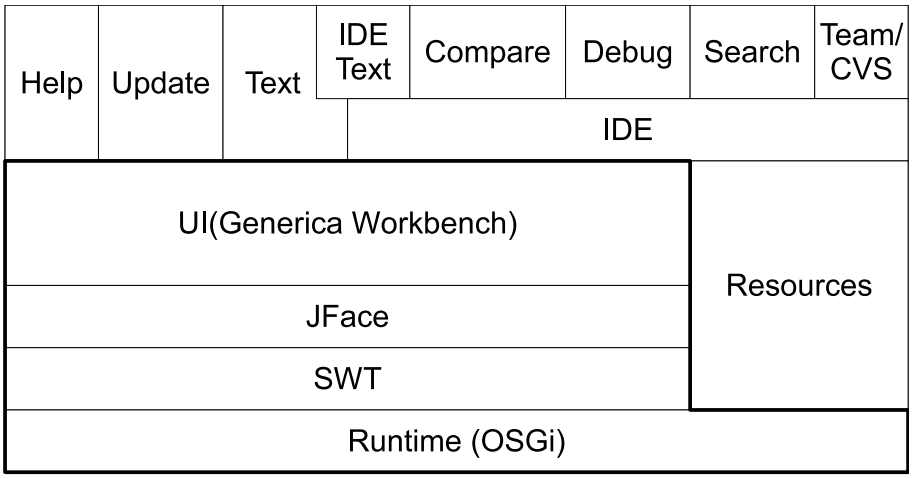

図 4.1: Eclipse のアーキテクチャ

この中で、Eclipse のプラグイン・アーキテクチャの基幹をなすのが最下層に位置している OSGi ランタイムであり、この OSGi をプラグイン管理のために用いている。そして、OSGi の上位に配置さてるコンポーネント群は全てプラグインとして提供される。このために拡張性がとても高く、Java 以外の多様な言語への対応が可能となっている。

また、Java 開発環境 (JDT) 自体もプラグインとして実装されている。JDT は、Java 要素と Eclipse 特有の制作物からなるモデルを中核にしている。Java モデルはクラス、フィールド、メソッドのような Java ファイルの構文上の内容を表現するものも IType、IField、IMethod というインターフェースとして含んでいる。これらの Java モデルを表すクラス・インターフェースは全て IJavaElement インターフェースを実装している必要がある。図 4.2 が Java モデルの一覧である。

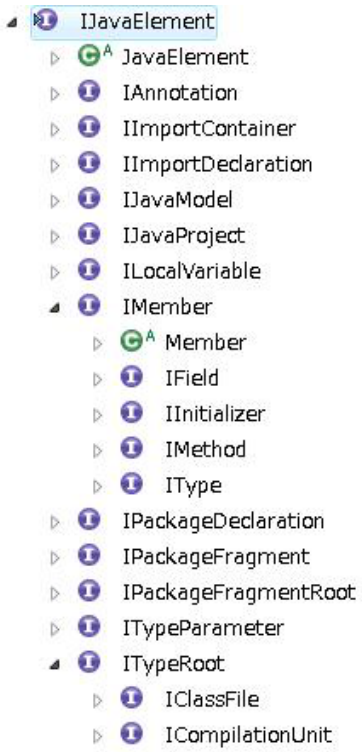

図 4.2: Java モデルの継承関係

Java モデルを扱う簡単なコード例を記載する。あるメソッドを表すインスタンスを得た時に、そのメソッドがどのクラスに属するものかを調べるメソッドを作成したいとする。そのメソッドは以下のように作成するこ

とができる。

```
1 public IType getType(IMethod myMethod){
2     IMethod myMethod = (IMethod) mem;
3     while(myJE.getElementType() != IJavaElement.TYPE){
4       myJE = myJE.getParent();
5     }
6     IType myType = (IType) myJE;
7     return myType;
8 }
```

図 4.3: Java モデルを用いたコード例

4.1.2　ワークベンチ

Eclipse では開発者が作業するためのユーザーインターフェース全般を指してワークベンチ (図 4.4) と呼ぶ。ワークベンチは様々なワークスペースリソースの作成、管理、ナビゲーション、および制御を行う中心となる場所である。

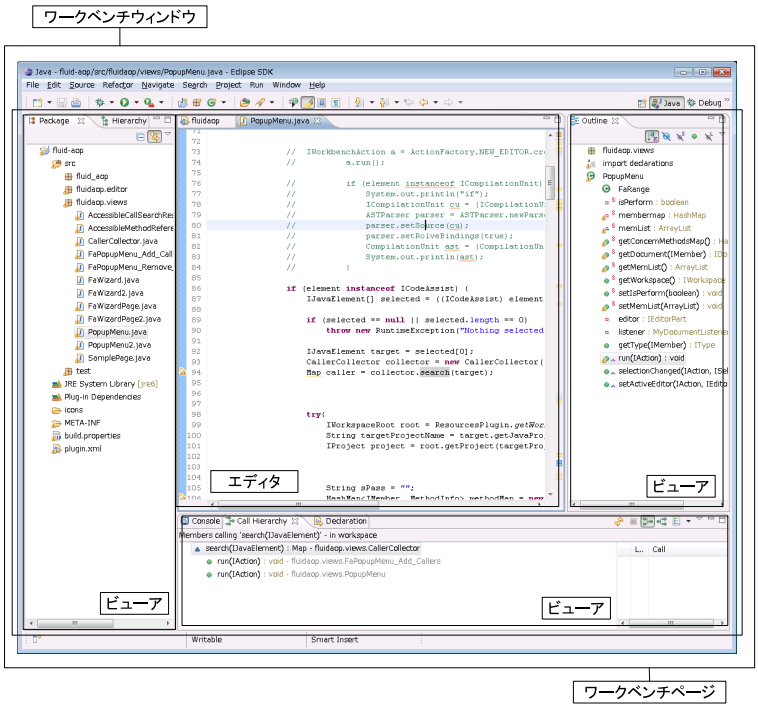

図 4.4: eclipse のワークベンチ

ここでは、そのワークベンチの構成と各構成要素の役割について述べる。

ワークベンチウィンドウ

Eclipse のウィンドウ。通常はワークベンチ 1 つに対し、1 つのウィンドウが開かれるが、複数の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ごとに 1 つのページが対応する。複数のビューアとエディタを含む。

パースペクティブ

特定のタスクをサポートするための、ビューアの初期セットと、ワークベンチのレイアウトの定義を行ったもの。例えば、通常の開発用の「Java」パースペクティブ、デバッグ用の「Debug」パースペクティブなどが存在する。主なパースペクティブは以下の通りである。

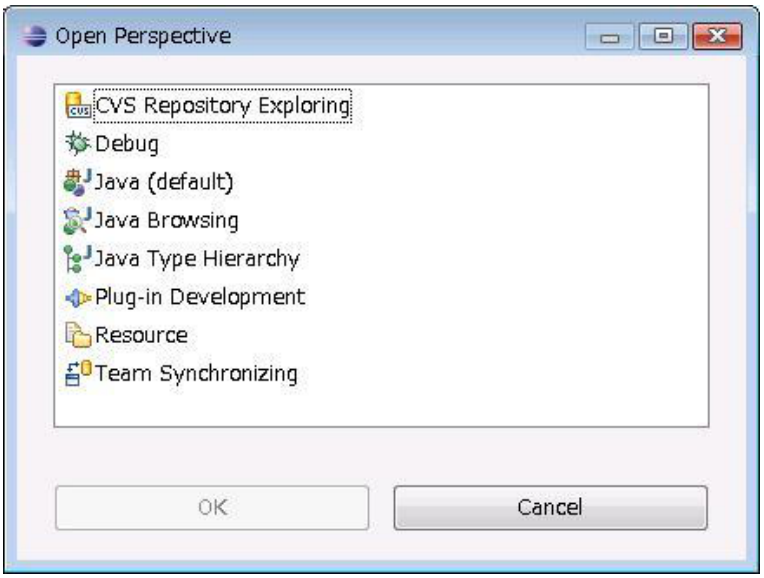

図 4.5: Eclipse の主なパースペクティブ

エディタ

ファイルの編集を行う箇所。同時に複数のエディタを開くことが可能である。また、「NewEditor」メニューを用いることで 1 つのファイルに対して複数のエディタを開くこともできる。

ビューア

エディタ上での開発を支援する役割を果たす。主なビューアは以下の通りである。

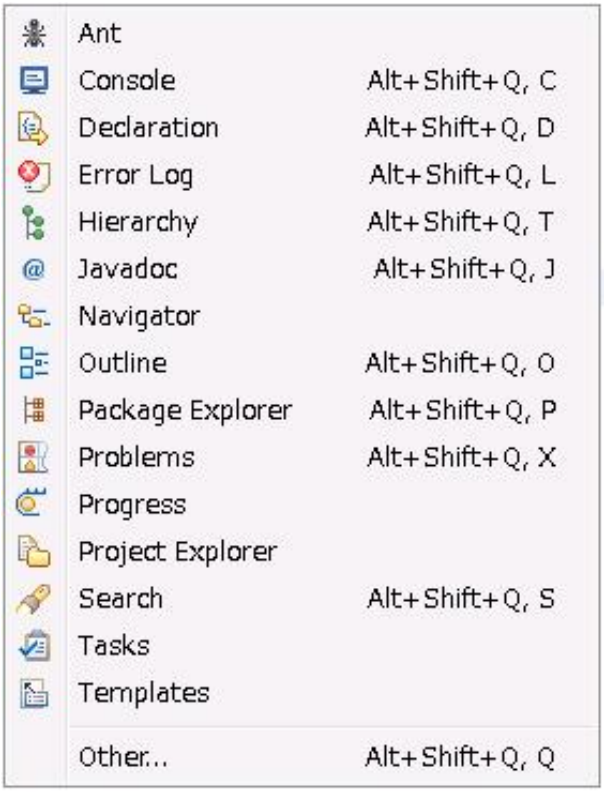

図 4.6: Eclipse の主なビューア

プラグイン開発では、コードからワークベンチにアクセスすることがある。例えば、本研究では既存の Java ソースコードエディタを監視する必要があるため、開発者が編集を行っているエディタを常に把握しなくてはならない。プラグインのコードからワークベンチにアクセスするには図 4.7 のように org.eclipse.ui.PlatformUI を用いる。

```
1 // ワークベンチを取得
2 IWorkbench workbench = PlatformUI.getWorkbench();
3 // ワークベンチウインドウを取得
4 IWorkbenchWindow window = workbench.getActiveWorkbenchWindow();
5 // ワークベンチページを取得
6 IWorkbenchPage page = window.getActivePage();
```

図 4.7: ワークベンチへのアクセス方法

ワークベンチを取得できれば、そのワークベンチ上で行われている作業に関するモデルを取得できる。例えば、編集対象になっている Java ファイルのパスを取得するには図 4.8 のようにすればよい。

```
1 // 現在編集を行っているエディタを取得
2 IEditorPart editorPart = page.getActiveEditor();
3 // 編集を行っている Java ファイル名のパスを取得
4 String filePath = editorPart.getEditorInput().getToolTipText();
```

図 4.8: 編集対象のパスを取得する実装

プラグイン開発では、このようにワークベンチにアクセスすることが頻繁にある。他の事例については、本システムの実装の中で述べる。

4.1.3　PDE(Plugin Development Environment)

PDE は Eclipse に標準で付属されているプラグインの開発環境である。プラグインの実装部分は Java で記述するが、PDE ではこれに加えプラグイン開発を支援するための機能を提供している。その中で最も重要なものがマニフェスト・エディタである。

新たにプラグインを作成するには、プラグインの各種設定を記述するための MANIFEST.MF と plugin.xml というマニフェスト・ファイルを用意しなくてはならない。Eclipse Templates plugin のうちのひとつである「Hello World」プラグインの MANIFEST.MF と plugin.xml を図 4.9 と図 4.10 に示す。

```
1  Manifest-Version: 1.0
2  Bundle-ManifestVersion: 2
3  Bundle-Name: Sample
4  Bundle-SymbolicName: sample; singleton:=true
5  Bundle-Version: 1.0.0.qualifier
6  Bundle-Activator: sample.Activator
7  Require-Bundle: org.eclipse.ui,
8   org.eclipse.core.runtime
9  Bundle-ActivationPolicy: lazy
10 Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

図 4.9: MANIFEST.MF

MANIFEST.MF はプラグインの ID や他のプログラムとの依存関係を記述するためのファイルである。plugin.xml はプラグイン・ランタイム・クラスの情報、拡張した機能、他のプラグインに公開する拡張ポイントの

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
   <extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            label="Sample Action Set"
            visible="true"
            id="sample.actionSet">
         <menu
               label="Sample &amp;amp;Menu"
               id="sampleMenu">
            <separator
                  name="sampleGroup">
            </separator>
         </menu>
         <action
               label="&amp;amp;Sample Action"
               icon="icons/sample.gif"
               class="sample.actions.SampleAction"
               tooltip="Hello, Eclipse world"
               menubarPath="sampleMenu/sampleGroup"
               toolbarPath="sampleGroup"
               id="sample.actions.SampleAction">
         </action>
      </actionSet>
   </extension>
</plugin>
```

図 4.10: plugin.xml

情報を、プラグインの管理を行う Eclipse プラットフォームのランタイムに教える役割を担う。マニフェスト・エディタはこれらのファイルをグラフィカルに編集するためのエディタである。eclipse マニフェスト・エディタを図 4.11 に示す。

マニフェスト・エディタでは画面下のタブでページを切り替えながら設定を行う。タブの種類は図 4.12 の通りである。

マニフェスト･エディタで編集した内容は、MANIFEST.MF、plugin.xml、bulid.properties　に自動的に反映される。例えば、Extensions タブで拡張に関する情報を編集すれば、その情報は即座に plugin.xml に伝わる。マニフェスト・エディタによって、煩雑なファイルの編集を行いやすくしている。

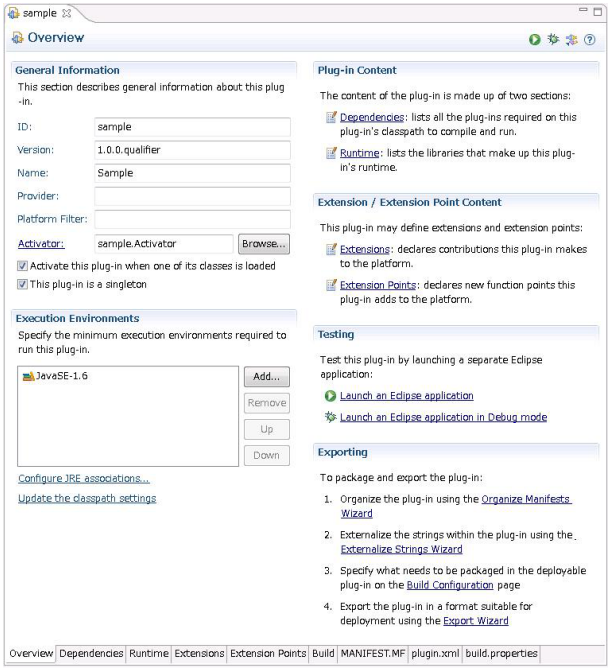

図 4.11: PDE のマニフェスト・エディタ

ページ	説明
Overview	プラグインのID、バージョン、名称といったプラグイン全体の情報を設定する。ランタイム・ワークベンチの起動、配布用アーカイブの作成を行うこともできる。
Dependencies	プラグイン間の依存関係を設定する。動作に必要なプラグインは「Required Plug-ins」に追加しておく必要がある。
Runtime	エクスポートするパッケージやクラスパスの設定を行う。
Extensions	プラグインがどの拡張ポイントに対し、どのような拡張を提供するかを宣言する。
Extension Points	他のプラグインに対して提供する拡張ポイントの定義を行う。
Build	ビルド時にアーカイブに含めるファイルやディレクトリなどを設定する。
MANIFEST.MF	MANIFEST.MF のソースを表示する。
plugin.xml	plugin.xml のソースを表示する。
build.properties	build.properties のソースを表示する。

図 4.12: マニフェスト・エディタのタブの種類

4.2　本システムの実装

本システムの実装について述べる。本システムの実装を、

- KIDE エディタの作成
- 既存の Java ソースコードエディタと KIDE エディタに、新しいコンテキストメニューを追加する
- 既存の Java ソースコードエディタのコンテキストメニューのアクションを具体的に実装する (コード断片の抽出)
- 既存の Java ソースコードエディタと KIDE エディタとの間で同期を行えるようにする
- KIDE エディタのコンテキストメニューのアクションを具体的に実装する
- その他の機能

と大別し、記載する。

4.2.1　Editor の作成

まず、抽出したコード断片を表示・編集するための KIDE エディタを作成した。拡張ポイントとして、org.eclipse.ui.editors を指定する。この拡張ポイントの配下に記述される editor タグの主な属性を表 4.1 に示す。

属性	説明
id	エディタの id を指定
name	エディタの名前を指定
extensions	エディタに関連付けるファイルの拡張子を指定
class	エディタの実装クラスを指定する

表 4.1: editor タグの属性

後述する抽出部分でファイルを作成し、抽出したコード断片をそのファイルに書き込んでいるが、その際に extensions で指定した拡張子のファイルを作成することで、コード断片が記述されたファイルをこのエディタで開くことができる。エディタの主な実装部分は class 属性で指定したクラスで記述する。今回はエディタ自体に特別な機能を加えないが、Java の予約語などの色を既存の Java ソースコードエディタと同様に強調したい。そこで、既存の Java ソースコードエディタを表す CompilationUnitEditor クラスを継承するした。本システムで作成したエディタのコードを図 4.13 に示す。

```
1 public class MyEditor extends CompilationUnitEditor {
2   public MyEditor()
3     super();
4     setEditorContextMenuId("#MyEditorContext");
5   }
6 }
```

図 4.13: 作成したエディタの実装

コンストラクタで setEditorContextMenuId メソッドを呼んでいる部分以外は、CompilationUnitEditor クラスと同様である。setEditorContextMenuID メソッドは各エディタのコンテキストメニューにアクションを追加する際に用いる識別子を変更するためのメソッドである。本システムでは、Java ソースコードエディタと KIDE エディタに異なるアクションを追加するために、この識別子のみを既存の Java ソースコードエディタと異なるものにした。4.2.2 でアクション追加に関する詳細を述べる。

4.2.2　コンテキストメニューに新しいアクションを追加

既存の Java ソースコードエディタと KIDE エディタのコンテキストメニューに新しいアクションを追加する方法を示す。拡張ポイントとして、org.eclipse.ui.popupMenus を指定する。その配下には、viewerContribution タグを指定する。viewerContribution は、特定の ContextMenuID を持つエディタに対してアクションの集合を追加する際に用いる。viewerContribution タグの主な属性は以下の通りである。

属性	説明
id	追加する Contriburion の id を指定
targetID	アクションを加えるエディタの ContextMenuID を指定

表 4.2: viewerContribution タグの属性

この viewerContribution タグはアクションを追加したいエディタの ContextMenuID ごとに作成することになるので、Java ソースコードエディタ用に targetID を ♯CompilationUnitEditorContext にしたタグとコード断片編集用のエディタ用に targetID を ♯MyEditorContext したタグを用意した。また、今回は用いなかったが、viewerContriburion タグの代わりに objectContribution タグを用いることも可能である。この objectContribution タグを用いると、オブジェクトに対してアクションを追加する

ことができる。例えば、拡張子が xml のファイルに対してアクションを追加することもできる。

viewerContribution タグの指定後、さらにその配下に action タグを用いて、コンテキストメニューに追加するアクションの情報を設定する。viewerContribution タグの配下には複数の action タグを指定できる。action タグの主な属性が表 4.3 である。

属性	説明
id	追加するアクションの id を指定
label	追加するアクションの表示名を指定
class	追加するアクションの実装クラス名を指定
menubarPath	メニューの挿入位置を指定
icon	追加するアクションのアイコンを指定

表 4.3: action タグの属性

menuberPath に何も記述しない場合、コンテキストメニューに表示されない。デフォルトの追加位置に追加したい場合は、“additions” と記述する必要がある。

本システムでは既存の Java ソースコードエディタに 2 つ、KIDE エディタに 2 つ、合計 4 つのアクションをコンテキストメニューに追加した。表 4.4、表 4.5 がその一覧である。

アクション名	用途
Focus on this callers	呼び出し関係にあるメソッドを抽出する
Focus on methods in superclasses	オーバーライドしたメソッドを抽出する

表 4.4: Java ソースコードエディタに追加したアクション

アクション名	用途
Remove Concern Method	関心のなくなったメソッドを除去する
Add Callers	呼び出し関係にあるメソッドを追加する

表 4.5: コード断片編集用エディタに追加したアクション

各アクションごとに、action タブの class で実装するクラスを指定する。4.2.3 で、表 4.4 のアクション実装について述べる。これらは Java ソースコードエディタ上からコード断片を抽出する機能を担う。また、表 4.5 の

アクションに関してはエディタ間の同期部分に関係が深いので、同期処理の実相について述べた後の 4.2.5 に記載した。

4.2.3　抽出を行う部分の実装

Java ソースコードエディタに追加したコードの抽出を行うアクションの実装について述べる。

”Focus on this caller” の実装

呼び出し関係にあるメソッドを抽出するアクションの実装について述べる。

まず、作業しているワークスペースから指定されたメソッドを呼んでいるコード断片を探索する必要がある。この探索に関しては、既存の Call Hierarchy View の実装を参考にすることで解決した。この実装を用いると、指定したメソッドを用いているコード断片を Map として取得することができる。これを用いてコード断片をファイルとして出力する。

最初に、コード断片を文字列としてまとめる。そのソースコードを図 4.14 に示す。

```
1  // 断片を取得
2  Map caller = ....search(target);
3
4  // ファイルに出力するための String
5  String sPass = ‘‘’’;
6
7  for (Iterator it = caller.keySet().iterator(); it.hasNext();) {
8    IMember callers =
9      ((MethodCall) caller.get(it.next())).getMember();
10   String comment = "\t/* " + callers.getPath() + "/"
11                            + callers.getElementName() + " */\r\n";
12   String source = callers.getSource();
13   sPass = sPass + comment + ‘‘\t’’ + source + ‘‘\r\n\r\n’’;
14 }
```

図 4.14: コード断片を文字列として取得する実装

次に、ファイルを新しく生成し、図 4.14 で取得した文字列を書き込む。ファイルの生成と書き込みに関するコードを図 4.15 に示す。

コードを抽出したファイルは指定されたメソッドが所属するプロジェクトの配下に作成する。図 4.15 の前半部でその情報を取得している。また、図 4.15 の 11 行目でファイルを生成しているが、ここでファイルの名前と

```
1  // ファイルを生成する位置の取得
2  IWorkspaceRoot root
3    = ResourcesPlugin.getWorkspace().getRoot();
4  String targetProjectName
5    = target.getJavaProject().getElementName();
6  IProject project
7    = root.getProject(targetProjectName);
8
9  // ファイルの生成
10 IContainer container = project;
11 IFile file =
12   container.getFile(new Path(targetProjectName + ".fa"));
13 file.create(new ByteArrayInputStream(
14   sPass.getBytes()), true, null);
```

図 4.15: コード断片をファイルとして出力する実装

拡張子を設定する。この拡張子を図 4.1 の extensions で指定したものに設定することで、ファイルを KIDE エディタで開くことができる。

”Focus on methods in superclasses” の実装

次にオーバーライドしたメソッドを抽出するアクションの実装について述べる。ファイルを作成する部分は前記したアクションと同様であるので、差分である抽出部分を文字列で取得する実装についてのみ記載する。

まず、指定されたメソッドが宣言されているクラスの親クラスを全て取得する。そのコードを図 4.16 に示す。

```
1  IJavaElement target = ...;
2  IMethod tMethod = (IMethod) target;
3
4  // tMethod が宣言されているクラスを取得
5  IType tType = ...;
6
7  // tType の親クラスを取得
8  ITypeHierarchy hierarchy =
9    targetType.newTypeHierarchy(targetType.getJavaProject(), null);
10 IType[] superTypes = hierarchy.getAllSupertypes(tType);
```

図 4.16: 全ての親クラスを取得する実装

前半で指定したメソッドが宣言されているクラス tType を取得した。その後、ITypeHierarchy インターフェースを用いることで親クラスの配列 superTypes を取得できる。次に、この IType の配列から指定されたメソッドと同じ名前かつ同じシグネチャのものを取得する。そのコードを

図 4.17 に示す。

```
String tMethodName = tMethod.getElementName();
String tMethodSignature = tMethod.getSignature();

ArrayList mList = new ArrayList();

for(IType type: superTypes){
  // そのクラスで宣言されている要素を取得する
  IJavaElement[] superTypeChildren = type.getChildren();

  for(IJavaElement je: superTypeChildren){
    // 宣言されたメソッドから条件に当てはまるもののみを取得
    if(je instanceof IMethod){
      IMethod sMethod = (IMethod) je;
      String sMethodName = sMethod.getElementName();
      String sMethodSignature = sMethod.getSignature();
      if(superTypeMethodName.equals(tMethodName) &&
        superTypeMethodSignature.equals(tMethodSignature)){
          mList.add(superTypeMethod);
      }
    }
  }
}
```

図 4.17: オーバーライドしているメソッドを取得する実装

最初の 2 行で、指定したメソッドの名前とシグネチャを取得している。その後、全ての親クラスの中から条件に該当する IMethod を探し mList に格納する。この mList を文字列に直し、KIDE エディタで表示するためのファイルにその文字列を書き込めば抽出が完了する。

4.2.4　同期処理を行う部分の実装

抽出したファイルと抽出元である Java ファイルとの間で両方向の同期を行えるようにした。ここでは、その実装について述べる。[9, 12]

本システムの同期処理は DocumentListener クラスを継承した MyDocumentListener クラスを抽出元と抽出先にリスナとして埋め込むことで行う。DocumentListener を埋め込むにはには、IDocument インターフェースによって表されるドキュメントを取得する必要がある。抽出元では IMethod から IDocument を、抽出先では IFile から IDocument を取得した。以下にその方法を示す。

まず、図 4.18 に IMethod から IDocument を取得する方法を示す。

IMethod から ICompilationUnit インターフェースによって表される Java ファイルを取得し、そこから IDocument を取得している。

次に、IFile から IDocument を取得する方法を図 4.19 に示す。

```
1  // IMethod から ICompilationUnit を取得
2  IMethod method = ...;
3  IJavaElement je = method;
4  while(je.getElementType() != IJavaElement.COMPILATION_UNIT){
5    je = je.getParent();
6  }
7  ICompilationUnit cu = (ICompilationUnit)je;
8
9  // ICompilationUnit から IDocument を取得
10 try {
11   IEditorPart ep = JavaUI.openInEditor(cu);
12   if(ep instanceof CompilationUnitEditor){
13     CompilationUnitEditor cuEditor = (CompilationUnitEditor)ep;
14     IDocumentProvider docProv = cuEditor.getDocumentProvider();
15     IDocument doc = docProv.getDocument(ep.getEditorInput());
16   }
17 } catch (...) {...}
```

図 4.18: IMethod から IDocument を取得する実装

```
1  IFile file = ...;
2  file.create(...);
3
4  // IFile から IEditorPart を取得
5  IEditorPart ep = IDE.openEditor(workbenchPage, file);
6
7  // IEditorPart から IDocument を取得
8  if(ep instanceof CompilationUnitEditor){
9    CompilationUnitEditor cuEditor = (CompilationUnitEditor) ep;
10   IDocumentProvider dp = cuEditor.getDocumentProvider();
11   IDocument doc = dp.getDocument(faEditorPart.getEditorInput());
12 }
```

図 4.19: IFile から IDocument を取得する実装

IEditorPart から IDocumentProvider を取得し、そこから IDocument を取得することができる。

監視を行いたい IDocument を取得することができたので、次にリスナを追加する。取得した IDocument に対して IDocumentListener を追加するには、IDocument の addDocumentListener メソッドを用いる。4.2.3 で示したコード片を抽出する際に抽出元の IDocument を取得しリスナを加える。抽出先には、ファイルを生成する際にリスナを加える。

次に、これらのドキュメントに加えるリスナである MyDocumentListerner クラスの実装について述べる。MyDocumentListener クラスの骨子を図 4.20 に示す。

このクラスは Singleton パターン [2, 3] を用いて、システム内でインス

```
1  public class MyDocumentListener implements IDocumentListener {
2
3    // Singleton
4    private static MyDocumentListener listener;
5    private MyDocumentListener() {}
6    public static MyDocumentListener getMyDocumentListener() {
7      if (listener == null) {
8        listener = new MyDocumentListener();
9      }
10     return listener;
11   }
12
13   private static IDocument myDocument;
14
15   private static ConcernMethods concernMethods
16                                   = new ConcernMethods();
17
18   boolean syn = true;
19   public void documentChanged(DocumentEvent event){
20     if(syn){
21       ...
22     }
23   }
24
25   ...
26 }
```

図 4.20: 同期処理を担当する MyDocumentListener クラスの実装

タンスがひとつしか生成されないようにし、全てのドキュメントに同じリスナを加えることにしている。

フィールド concernMethods には抽出したメソッドの情報を格納している。情報はメソッドごとに以下のものがある。

- そのメソッド自身を表す IMethod
- 抽出元のドキュメントを表す IDocument
- 抽出元の Java ファイルのどこにメソッドが定義されているかを表す offset と length
- 抽出先のファイルのどこにメソッドが定義されているかを表す offset と length

同期処理は図 4.20 の documentChanged メソッドにおいて行われる。リスナを追加したドキュメントに変更が加わるとこのメソッドが呼び出される。引数である event から、イベントの情報を取得できるので、event と前記した concernMethods を用いて同期処理を制御する。documentChanged メソッド内の流れを図 4.21 に記載する。

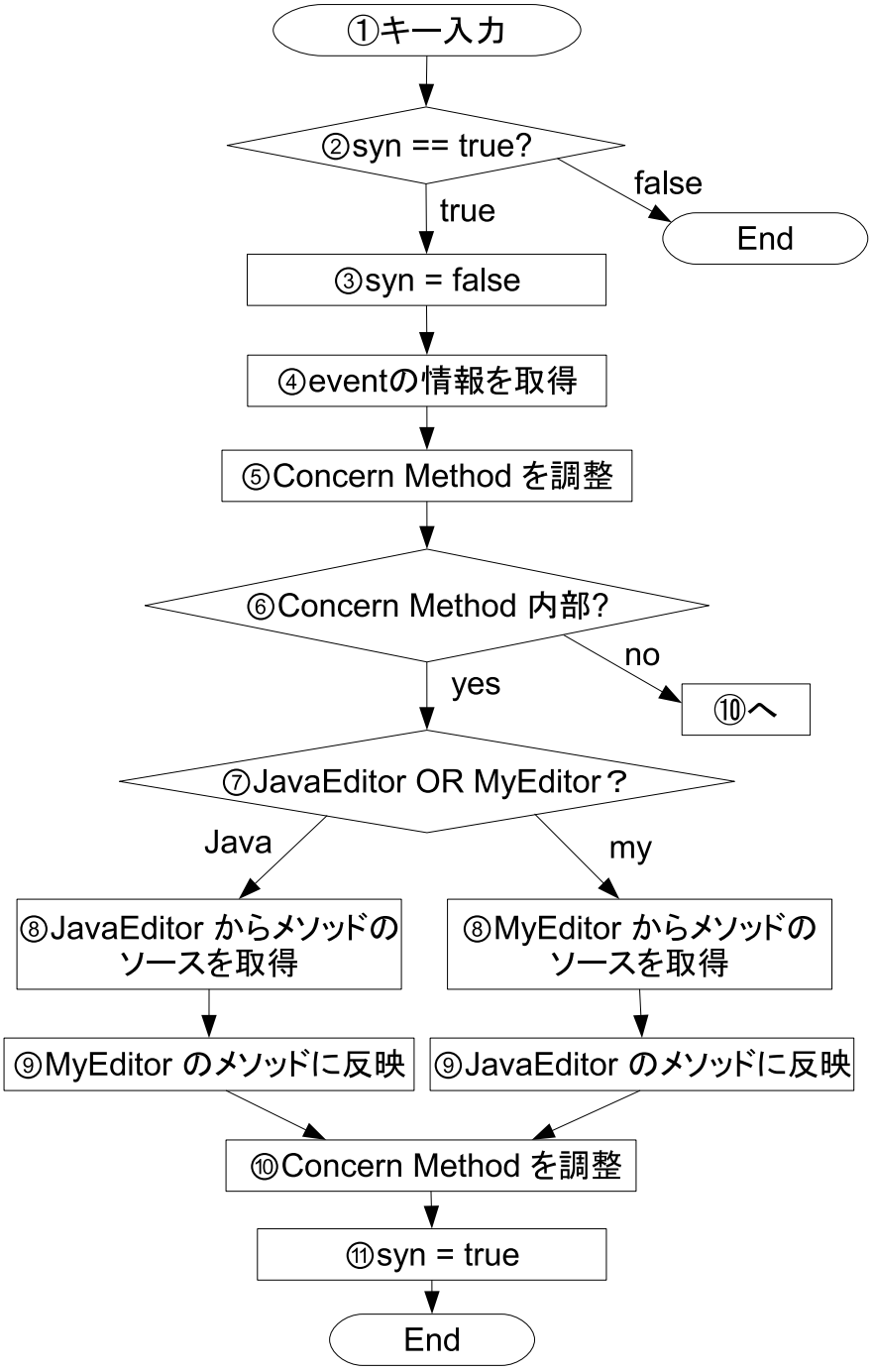

図 4.21: 同期処理の流れ

図 4.21 の詳細を以下で述べる。

① キー入力を感知
リスナが追加されたドキュメントに変更があると documentChanged メソッドが実行される。

② 排他制御
図 4.20 の 19 行目で宣言されているフィールド syn は排他制御を行うためのものである。もしこの値が false であれば、他のドキュメントが同期処理をしている最中であることを意味するので、ここで処理を終える。

③ 同期処理を開始
排他制御のために syn を false に変更する。

④ イベントの情報を取得
documentChanged メソッドの引数である event からドキュメントの変更に関する情報を取得する。図 4.22 にそのコードを示す。

```
1 // イベントが起きたドキュメントを取得
2 IDocument eventDoc = event.getDocument();
3
4 // イベントが起きた位置を取得
5 int eventOffset = event.getOffset();
```

図 4.22: ドキュメントの修正に関する情報を取得する実装

⑤ 抽出したメソッドの情報を調整
抽出したメソッドの情報を ConcernMethods として保持しているが、ドキュメントへの修正次第ではその情報が変わることがある。例えば、以下の foo メソッドの場合を考える。

```
 /* (1) */
foo(...){
...
 /* (2) */
...
}
 /* (3) */
```

キー入力が (1) で起こった場合、foo メソッドの offset がずれる。(2) で起こった場合は foo メソッドの length がずれる。(3) の場合は offset も length もそのままである。

そこで、ドキュメントに変更がある度に、その場所に応じて保持している情報を更新する。そのコードを図 4.23 に示す。

```
1  for(/* ConcernMethods で保持している全てのメソッドに対して */...){
2    /* 抽出したメソッドの中の一つ */
3    IMethod method = ...;
4
5    /* メソッドの KIDE エディタ内での位置 */
6    int mOffset = ...;
7    int mLength = ...;
8
9    if(eventOffset < mOffset){
10     /* このメソッドの上部での入力であったことになるので
11        入力の分だけ mOffset を変化させる */
12   }
13
14   if(mOffset < eventOffset && eventOffset < (mOffset + mLength)){
15     /* このメソッドの内部で入力があったことになるので
16        入力の分だけ mLength を変化させる */
17   }
18 }
```

図 4.23: ConcernMethods の調整を行う実装

全てのメソッドに大して、キー入力の場所との位置関係を調べて修正を行っている。

⑥ 同期処理を行う必要があるか
本システムでは、抽出したメソッドに関連するコードに修正があった時のみ同期を行う。つまり、入力の場所がどのメソッドの内部にも当てはまらない場合は同期は行わない。その場合は syn を true に変更し、ここで処理を終える。

⑦ Java ソースコードエディタでの修正か、KIDE エディタでの修正か
⑥ で同期を行うことになった場合、その修正が Java ソースコードエディタ上で行われたか、KIDE エディタ上で行われたかを調べなくてはならない。そうでないと、抽出先のコードを抽出元に伝えるべきか抽出元のコードを抽出先に伝えるべきかの判断ができない。本システムでは、イベントから取得した IDocument と ConcernMethods が保持している IDocument の比較を行うことで判断する。

⑧ 修正されたメソッドのソースを取得
⑦ の結果に合わせて修正があった IDocument からメソッドのソースを取得する。ソースの取得の方法を図 4.24 に示す。ここでは、KIDE エディタで修正を行った場合の例を挙げる。IDocument の get メソッドを用いて文字列を取得している。get メソッドには取得した

```
1  // ConcernMethods からメソッドの位置と IDocument を取得
2  int offset = ....getKIDERange().getOffset();
3  int length = ....getKIDERange().getFaRange();
4  IDocument doc = KIDEDocument;
5
6  try {
7    String newText = doc.get(offset, length);
8  } catch (BadLocationException e) {
9    e.printStackTrace();
10 }
```

図 4.24: メソッドのソースを取得する実装

い文字列のドキュメント内での offset と length を渡す必要があるので、これらを ConcernMethods から取得する。

⑨ 同期処理

⑧ で得たソースを対応する箇所に伝える。その方法を図 4.24 の続きとして図 4.25 に示す。

```
1  int javaOffset = ....getJavaRange().getOffset();
2  int javaLength = ....getJavaRange().getLength();
3  IDocument javaDoc = ....getJavaDocument();
4
5  try {
6    javaDoc.replace(javaOffset, javaLength, newText);
7  } catch (BadLocationException e) {
8    e.printStackTrace();
9  }
```

図 4.25: replace メソッドを用いて同期を行う実装

IDocument の replace メソッドを用いる。get メソッド同様、ソースを挿入する位置と挿入する文字列を引数に渡している。

⑩ 抽出したメソッドの情報を修正

⑨ でソースが挿入されたことによって、再び保持しているメソッドの位置情報を変更する必要が出てくるので ⑤ と同様の処理を行う。

⑪ 同期処理終了

同期処理が終わったため、syn を true に変更する。

以上が同期処理の流れである。排他制御を行ったのは ⑨ で replace した際に、ドキュメントリスナが変更を感知し、再び documentChanged メソッドが呼ばれてしまう、という無限ループを防ぐためである。

4.2.5　追加機能の実装

KIDE エディタに追加したコンテキストメニューの実装について述べる。

”Remove Concern Method” の実装

KIDE エディタから関心のなくなったメソッドを取り除くアクションの実装を記載する。前記した通り、抽出したメソッドの情報は MyDocumentListener の ConcernMethods フィールドに格納されている。そのため、抽出したファイルから対応する箇所を除去するだけでは不十分であり、ConcernMethods からも除去する必要がある。

まず、開発者が取り除くべきメソッドを選択するためのダイアログを用意した。既存の ListSelectionDialog を継承した RemoveDialog クラスでこの実装を行っている。このクラスの実装は図 4.26 の通りである。

```
1 public class RemoveDialog extends ListSelectionDialog {
2   public RemoveDialog(...) {}
3
4   protected void configureShell(Shell newShell){
5     super.configureShell(newShell);
6     newShell.setText("Remove Method");
7   }
8 }
```

図 4.26: RemoveDialog

configureShell メソッド内で、ダイアログに載せるテキストを設定している。“Remove Concern Method” アクションを実行したら、このダイアログが開くようにする。

このダイアログを用いてアクションを実装した。そのコードを図 4.27 に示す。

19 行目で ConcernMethods から情報を取り除いている。これで、同期処理の対象から外れる。21 行目で KIDE エディタからこのメソッドの記述部を消去している。

”Add Callers”の実装

KIDE エディタ上でさらに呼び出し関係にあるメソッドを追加抽出するアクションの実装に関しては、4.2.3 の呼び出し関係にあるメソッドの抽出とほぼ同じである。ただし、一点だけ異なる点がある。このアクションは KIDE エディタ上で抽出したメソッドからひとつメソッドをマークし、

```
1  public void run(IAction action) {
2    // dialog に表示させるもののリストを渡す
3    ArrayList methodList = MyDocumentListener.getConcernMethods();
4    RemoveDialog dialog = new RemoveDialog(..., methodList, ...);
5
6    // dialog のどのボタンが押されたかを取得する
7    int ret = dialog.open();
8    if(ret == IDialogConstants.OK_ID){
9      // Check されたメソッドのみ取得する
10     Object[] result = dialog.getResult();
11
12     // ArrayList に変換
13     ArrayList removeMethos = ...;
14     for(/* result の数だけ */){
15       removeMethods = (/* cast */) result[...];
16     }
17
18     for(/* removeMethods の数だけ */){
19       MyDocumentListener
20            .getConcernMethods().remove(/* 取り除く Method */);
21       MyDocumentListener.getMyDocument().replace(..., ‘‘’’);
22     }
23   }
24 }
```

図 4.27: RemoveMethodAction

探索を行う。Java ソースコードエディタ上ではメソッドをマークすれば IMethod として認識できた。しかし、KIDE エディタ上では、そのマークした対象を文字列としてしか認識できない。そこで、マークした文字列と ConcernMethods との比較を行い、対応する IMethod を取得している。

4.2.6　抽出する断片を限定する Wizard

最後に、3.2.2 で述べた抽出時のサポートの実装について記載する。実装は、図 3.5 の Wizard として行った。まず最初に、Wizard の基本構成を図 4.28 に示す。

ウィザードはウィザードダイアログによってダイアログとして表示され、Wizard クラスと WizardPage クラスで構成される。本システムでは、この 2 つのクラスについて実装を行った。まず、大枠となる Wizard クラスの実装を図 4.29 に示す。

addPages メソッドで、このウィザードに貼り付けるウィザードページを設定する。ここでは後述する MyWizardPage クラスのインスタンスを引数に渡し、設定している。ウィザードが提供する機能は、主に WizardPage クラスで実装することになる。

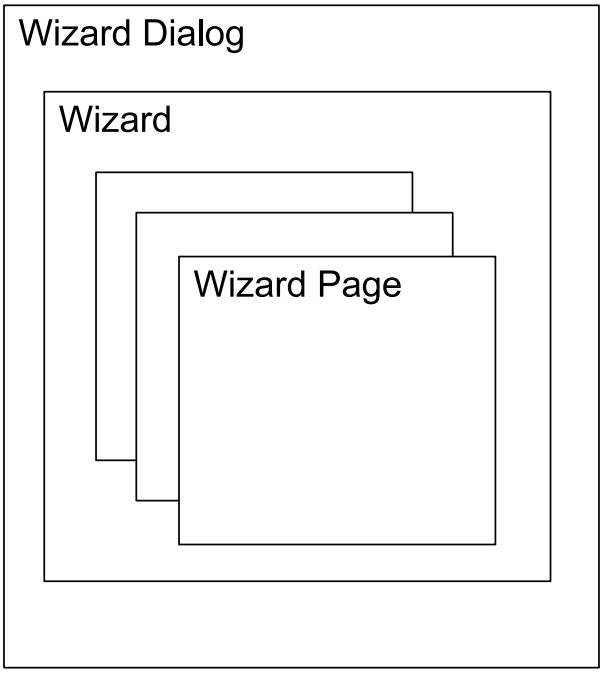

図 4.28: ウィザードの基本構成

```
1  public class MyWizard extends Wizard {
2    public MyWizard(){
3      setWindowTitle("MyWizard");
4    }
5
6    public void addPages(){
7      addPage(new MyWizardPage());
8    }
9
10   public boolean performFinish() {...}
11   public boolean performCancel() {...}
12 }
```

図 4.29: MyWizard クラスの実装

ここから WizardPage クラスの実装について述べる。まず、図 4.30 にこのクラスの雛形を示す。

コンストラクタでこのページのタイトルや説明などの基本情報を設定する。そして、createControl メソッド内に Text や Button などを配置することでユーザーインターフェースを構築する。以降、この createControl メソッドの実装についての詳細を述べる。

まず、ウィザードページに貼り付けられるものは、Control クラスを継承

```
1  public class MyWizardPage extends WizardPage {
2    ...
3    protected MyWizardPage(){
4      super("MyWizardPage");
5      setTitle("MyWizardPage");
6      setDescription("抽出するメソッドを選択してください.");
7    }
8
9    public void createControl(Composite parent) {
10     ...
11   }
12   ...
13 }
```

図 4.30: MyWizardPage クラスの雛形

しているものでなくてはならない。今回は、ツリービューアとそのビューアにフィルターをかけるテキストボックスの組を提供する FilteredTree クラスを用いた。createControl メソッドを図 4.31 に示す。

```
1  public void createControl(Composite parent) {
2    ...
3    // ウィザードに filteredTree を貼り付ける
4    PatternFilter pf = new PatternFilter();
5    FilteredTree ft =
6        new FilteredTree(parent, SWT.CHECK |..., pf, true);
7
8    // filteredTree のビューアにモデルを渡す
9    TreeViewer tv = ft.getViewer();
10   tv.setLabelProvider(new MyTreeLabelProvider());
11   tv.setContentProvider(new MyTreeContentProvider());
12   tv.setInput(createModel());
13   ...
14 }
```

図 4.31: MyWizard クラスの createControl メソッドの実装

図 4.31 の詳細について記載する。

3〜4 行目

ウィジェットに貼り付ける FilteredTree の基本設定を行っている。4 行目のコンストラクタの第一引数にはウィジェットの貼り付け先を指定している。第二引数には作成するウィジェットの設定行っている。SWT.CHECK と記述することで、TreeViewer にチェックボックスをつけることができる。

6〜9 行目

FilteredTree のビューアで表示させるモデルを与えている。setLa-

belProvider メソッドでビューアのラベルプロバイダを設定する。ここで引数に与えることができるのは、ILabelProvider インタフェースを実装しているクラスで、ビューアのレイアウトをここで設定する。各ノードに画像を貼り付ける役割もラベルプロバイダが担う。setContentProvider メソッドで、ビューアのコンテンツプロバイダを設定する。引数には、IContentProvider を実装しているクラスを与える。コンテンツプロバイダは、ビューアクラスからの通知を元にモデルを提供する。最後に、setInput メソッドを用いてビューアにモデルを渡している。CreateModel メソッドについては後述する。

以上でウィザードに FilteredTree を貼り付けることができた。最後に、TreeViewer のモデルを作成する方法を示す。まずは、木のノードを表すクラスを作成した。そのコードを図 4.32 を示す。

```
1  class MyTreeModel {
2    MyTreeModel parent;
3    public ArrayList children = new ArrayList();
4    String name;
5    IJavaElement data;
6    boolean isChecked = false;
7
8    public MemberTreeModel(MemberTreeModel parent, String str) {
9      this.parent = parent;
10     this.name = str;
11   }
12     ...
13 }
```

図 4.32: MyTreeModel の実装

parent は親ノード、children は子ノードを表す。data と name がこのノードが表す IJavaElement とその名前を示す。boolean 型の isChecked は、ビューア上でノードがチェックされているかどうかを表す。

次に、パッケージを渡すと配下にあるクラスのリストを取得できる HashMap PackageToType と、クラスを渡すと配下にあるメソッドの取得できる HashMap TypeToMethod を作成する。これらは、関心事に対応するコード断片を抽出した際に、メソッドからクラスとパッケージを取得することで作成することができる。メソッドからクラスやパッケージを取得する方法は前記したので詳細についてはここでは省略する。最後に、これらを用いてモデルを構成する createModel メソッドを定義する。そのコードを図 4.33 に示す。

まず、木の根に対応するモデル root を生成する。root の子ノードをパッケージを表すノードにし、さらにその子ノードをクラスを表すノード

```
1  private MemberTreeModel createModel() {
2    MemberTreeModel root = new MemberTreeModel(null,"root");
3    MemberTreeModel tmp;
4
5    for(IPackageFragment pf : PackageToTypes.keySet()){
6      MyTreeModel packModel
7          = new MyTreeModel(root,pf.getElementName());
8      packModel.setData(pf);
9      root.child.add(packModel);
10     PackageToModel.put(pf, packModel);
11
12     ArrayList list1 = (ArrayList) PackageToTypes.get(pf);
13     for(int i = 0; i < list1.size(); i++){
14       IType type = (IType) list1.get(i);
15       MyTreeModel typeModel
16           = new MyTreeModel(packModel,type.getElementName());
17       typeModel.setData(type);
18       packModel.child.add(typeModel);
19       TypeToModel.put(type, typeModel);
20
21       ArrayList list2 = TypeToMethods.get(type);
22       for(int j = 0; j < list2.size(); j++){
23         IMember mem = (IMember) list2.get(j);
24         MemberTreeModel methodModel
25             = new MemberTreeModel(typeModel, mem.getElementName());
26         methodModel.setData(mem);
27         typeModel.child.add(methodModel);
28         MethodToModel.put(mem, methodModel);
29       }
30     }
31   }
32     return root;
33 }
```

図 4.33: createModel メソッドの実装

にする。最後にクラスを表すノードの子ノードをそのクラスに属するメソッドノードにする。これでモデル root は木を構成することができた。

ここまでで、抽出するメソッドを選択するツリービューアを作成することができた。最後に、ツリービューア上でユーザが抽出するメソッドを選択する際の動作に関する実装について述べる。まず、ユーザのビューアのノードにチェックを入れる、チェックを外す、という動作を感知する必要がある。そのために、ツリービューアの持つツリーに対して、図 4.34 のように addListener メソッドを用いてリスナを加える。

加えたいリスナを addListener メソッドの引数に渡す。ここでは、無名クラスとして、リスナの定義を行った。ツリービューアに対して、チェックを入れる、外すというイベントが発生すると、handleEvent メソッドが呼び出される。図 4.34 では、event からイベントの起きたノードを TreeItem 型で取得し、チェックが入れられているかを getChecked メソッドを用い

```
1 ft.getViewer().getTree().addListener(SWT.Selection, new Listener(){
2   public void handleEvent(Event event) {
3     if (event.detail == SWT.CHECK) {
4       TreeItem item = (TreeItem) event.item;
5       boolean checked = item.getChecked();
6       checkItems(item, checked);
7     }
8   }
9 });
```

図 4.34: ツリービューアにリスナを追加する実装

て調べている。これをビューアの各ノードの isChecked フィールドに反映させるメソッド checkItems メソッドに渡している。

以上でウィザードにユーザーがメソッドを選択するための filteredTree を構築することができた。ウィザードの Finish ボタンを押されたときにチェックが入れられいるメソッドのみを KIDE エディタで編集するファイルに書き込むことで、抽出物の限定を行っている。

第5章　評価

KIDE の有用性を検証するために、Eclipse のソースコードを閲覧するテストを行う。org.eclipse.jdt.core.JavaCore クラスは JDT コアプラグインの機能を利用するためのエントリポイントとなるクラスで、ワークスペースのリソースから Java モデルを取得するための static メソッドを提供する。例を以下に挙げる。

```
public static IJavaElement create(IFile) {...}
```

ワークスペースリソースの IFile から、Java モデルである IJavaElement を取得するためのメソッドである。このメソッドを用いる場合、引数に与える IFile を取得しなくてはならない。その取得方法を得るのに有効なアプローチが、Eclipse を実装しているコードから create メソッドを用いている部分を探し出す方法である。しかし、Eclipse のコードは膨大で、開発者が自力で該当するコードを探すことはほぼ不可能である。そこで Eclipse のビューアを用いる。メソッドを使用している部分を探す時に役立つのが Call Hierarchy View であるが、前記した通り、ビューアを用いるとエディタとの間を行き来することが増えてしまう。そこで、KIDE で作成したエディタを用いることで、この切り替え回数を減らすことができるか調べてみた。具体的な調査の方法は以下の通りである。

- JavaCore からいくつかメソッドを選ぶ。
- Eclipse のコードの中でそのメソッドが実際に使われている部分を探す。メソッドの引数に入れる値の取得方法が分かれば、そこで終了。もし、他のメソッドを見てみないと分からない場合は、取得方法が分かるまでコードをたどる作業を行う。
- 探索が終わった時点までのエディタや各ビューア間の切り替え回数を数える。
- Eclipse のみで探索を行った場合と、KIDE のエディタも用いて探索を行った場合の比較を行う。

結果を図 5.1 に示す。

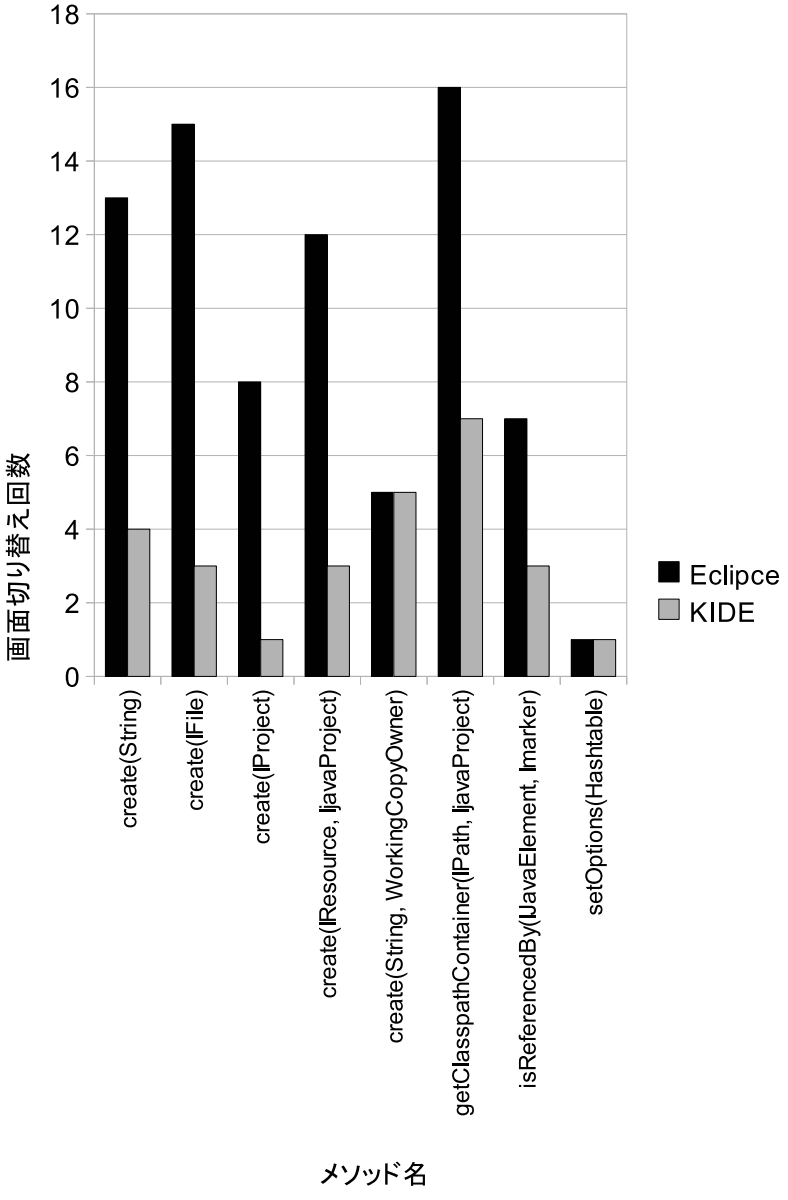
18
16
14
12
10
8
6
4
2
0
画面切り替え回数
create(String)
create(IFile)
create(IProject)
create(IResource, IJavaProject)
create(String, WorkingCopyOwner)
getClasspathContainer(IPath, IJavaProject)
isReferencedBy(IJavaElement, Imarker)
setOptions(Hashtable)
Eclipce
KIDE
メソッド名

図 5.1: 実験結果

同じ作業を行う場合、KIDE の方がエディタとビューア間の切り替え回数を少なく、探索を行えることが分かった。KIDE の方がユーザーに周辺情報を多く提供できることと、エディタ上で再度呼び出し関係にあるメソッドを抽出できることが探索を行いやすくした理由だと考えられる。

第6章　まとめと今後の課題

6.1　まとめ

ソフトウェア開発に OOP を用いることで、プログラムをモジュール化しやすく、保守性・拡張性を高めることができる。一方で、OOP では横断的関心事を 1 つのモジュールとして分離できないという限界も知られている。AOP を用いることで横断的関心事をうまくモジュール化できるが、そのモジュール構造は一度決めてしまうと変更しにくく、着目する関心事に合わせてその都度視点を切り替えることが難しい。IDE の提供する機能を用いる方法は、複数のエディタや各種ビューアを IDE 上で組み合わせて用いなくてはならず、効率が悪い。

このような問題を処理するために、本研究では関心事ごとに視点を切り替えてプログラムを編集できる統合開発環境 KIDE の提案を行い、eclipse プラグインとして実装した。KIDE は適宜横断的関心事を KIDE エディタ上に抽出し、通常とは異なる視点からプログラムを表示する。KIDE エディタは、抽出元のファイルと同期を取るため、モジュール間をまたがる編集をこのエディタ上だけで行うことが可能になった。呼び出し側のメソッドやオーバーライドメソッドなど、関心事の抽出方法は幾つか用意した。

6.2　今後の課題

6.2.1　機能の強化

より柔軟にユーザーの関心事に対応するために、抽出方法や周辺のサポートをより充実させる必要がある。現在の本システムは、AOP による利点の一部しかサポートできていない。言語の知識なしに AOP の利点をユーザーに提供することが本研究の目的であるので、その差を埋めていく必要がある。闇雲に機能を追加するのではなく、ユーザーの捉えたい関心事を調査し、その需要に答えられるような機能を提供していく。

6.2.2　プログラマを実際に使った評価

本研究のようなユーザーインターフェースを提供する研究は、評価の仕方が難しいと言われている。そこで、このようなアプローチが有効であるかをユーザーの視点で測るために、2.4 で述べた関連研究 Mylar のように実際にプログラマに本システムを用いて評価してもらいたいと考えている。そのためには、システムを整え、プラグインとして配布する準備をしなくてはならない。

参考文献

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

[2] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. M.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional (1994).

[3] Hannemann, J. and Kiczales, G.: Design pattern implementation in Java and aspectJ, *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, ACM, pp. 161–173 (2002).

[4] Hon, T. and Kiczales, G.: Fluid AOP join point models, *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, New York, NY, USA, ACM, pp. 712–713 (2006).

[5] Kersten, M. and Murphy, G. C.: Mylar: a degree-of-interest model for IDEs, *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 159–168 (2005).

[6] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and riswold, W. G. G.: An Overview of AspectJ, *ECOOP '01 - Object-Oriented Programming: 15th European Conference, LNCS 2072*, Springer, pp. 327–353 (2001).

[7] Masuhara, H. and Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms, *ECOOP '03 – Object-Oriented Programming*, Springer-Verlag, pp. 219–233 (2003).

[8] Masuhara, H., Kiczales, G. and Dutchyn, C.: Compilation semantics of aspect-oriented programs, *FOAL 2002 Proceedings of Foundations of Aspect-Oriented languages Workshop at Aspect-oriented software development (AOSD 2002)*, pp. 17–26 (2002).

[9] Shavor, S., D7Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J. and McCarthy, P.: Java 開発者のための Eclipse エキスパートガイド, コンピュータ・エージ社 (2004).

[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, 技術評論者 (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, 株式会社毎日コニュニケーションズ (2007).

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