平成23年度 修士論文

Kide: 開発環境による
対話的なモジュール分割と
文書作成の支援

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 10-3705-4

金澤 圭

指導教員

千葉 滋 教授

平成24年1月31日

概要

本論文ではプログラムの対話的なモジュール分割を支援する開発環境 Kide を提案する。ソフトウェア開発者はプログラムを関心事ごとにモジュール分割し拡張性や保守性を高める。従来のモジュール分割はプログラミング言語の言語機構だけを使って行われるが、この方法では一度決めた分割を変更しにくく、開発者の捉える関心事の推移や設計時に捉えきれなかった関心事に対応しにくい。この場合、モジュール間をまたがる煩雑な作業が必要になり、バグを生む遠因となりうる。Kide は開発者の要求に応じて、元の言語機構によるモジュール分割とは異なる分割でプログラムを表示する。関心事は事前に開発者が Kide 上で定義し、その関心事に着目すると Kide は登録した要素のみを仮想ファイル上に表示する。本来は複数のファイルにまたがって実装されたプログラム断片をひとつのエディタ上で編集することができる。また、着目する関心事に関連する要素のみを表示するので、編集箇所を探す必要がなくなり、編集作業自体に集中することができる。関心事はメソッド、あるいはメソッドの一部の集まりとして定義でき、開発者が手動で定義する方法とメソッドの呼び出し関係から自動的に定義する方法がある。また、定義した関心事は開発者間で受け渡しすることもできる。さらに、Kide はプログラムを含むドキュメントの作成と利用を支援する。従来の方法でドキュメントにプログラムを掲載すると、元のプログラムに修正があった場合、ドキュメント側にも同様の修正が必要になる。Kide は関心事として定義したプログラム断片をドキュメントにマージして表示でき、一方での修正がもう一方に伝わるため上記のような複数箇所の修正を行う必要がない。これら Kide の機能は Eclipse プラグインとして実装し、開発環境の機能として提供しているため新しい知識なしに使用できる。本論文では、Kide が有用なケースとして Javassist の開発での利用シナリオを記載する。また、開発環境を用いたモジュール分割支援を行う先行研究と Kide との比較を行う。さらに、モジュール分割に 従来手法であるプログラミング言語の言語機構のみを用いた場合と Kide を用いた場合のプログラム閲覧性の評価を Javasisst と AjHotDraw の実装を題材に行う。

This paper proposes a new Integrated Development Environment(IDE) support named Kide, which allows interactive modularization and documentation. Current developers use only programming language mechanisms for modularizing programs. Once a program is modularized, changing its module structure is difficult. This fact makes it difficult that developers browse and edit the program from when their concern is changed. Kide can display a different view of source code from original one according to the developers' concern. A concern is defined by giving a set of methods or parts of methods. Kide also support documentation including source code fragment. Since Kide provides these support through IDE's function, developer can easily learn them. This paper also shows a usage scenario of Kide, which we found though the development of open source software Javassist. Furthermore, this paper compares browsability among Kide, Java, and AspectJ source code. Finally, this paper shows related work to Kide, in the domain of program decomposition by IDE support.

謝辞

指導教員の千葉滋教授には、本研究を進めるにあたり、研究方針、論文執筆、システム仕様決定などに関する数々の有用な助言を頂きました。海外学会を含め、多くの学会に参加させて頂き、その都度論文やスライドを修正して頂きました。心より感謝致します。

IBM 東京基礎研究所の堀江倫大氏には、システムの設計と実装方法、論文執筆から発表まで多岐にわたって指導して頂きました。卒業後も定期的に論文を読んで頂き、多くのアドバイスを頂きました。心より感謝いたします。

研究室の先輩方、同期、後輩の皆さまには、学会の度に発表練習に付き合って頂き、様々な助言を頂きました。また、実装や環境構築の面で様々な助言を頂きました。本当にありがとうございました。

最後に、研究室生活を進める中で多岐にわたって支援してくれた家族に心から感謝致します。ありがとうございました！

目 次

図 目 次

表 目 次

第1章　はじめに

プログラムを記述する上で拡張性や保守性を高めることはとても重要である。仕様を満たすプログラムであっても実装が複雑になってしまい、機能の追加に柔軟に対応できないものであれば、それは良いプログラムとは言えない。プログラムの修正作業は修正箇所を特定し、その箇所に修正を行う、という大きく二つの作業に分類できるが、入り組んだプログラムでは修正箇所が様々な場所に散らばってしまい、そのコストが膨らんでしまう。また、修正箇所が一箇所でないため、修正箇所の特定と修正作業を何度も繰り返し行う必要があり、開発者に負担のかかる煩雑な作業を強いてしまう。修正箇所を探す時間は修正作業全体の約6割から9割を占めると言われており、この時間を減らすためにも拡張性と保守性の高いプログラムを書く必要がある。ソフトウェア開発者はプログラムの保守性・拡張性を高めるためにプログラムを関心事ごとにモジュール分割する。関心事とはプログラム記述時、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である。近年では、より良いモジュール分割手法やそれらの手法を実装したプログラミング言語が提案・研究されている。現在の技術を用いれば、設計段階で捉えられた関心事のほとんどをモジュール分割できると我々は考える。

では、開発フェーズのどの時点にも適するモジュール分割をプログラミング言語の言語機構のみを用いて行うことは可能だろうか。我々は難しいと考えた。なぜならば、開発者の捉える関心事は推移するものであり、特定の関心事に特化して行ったモジュール分割がそれ以外の関心事を捉える際にも有用とは限らない。また、プログラミング言語の言語機構によるモジュール分割では、プログラム断片を静的に各ファイルに記述する必要があり、複数の関心事に関連するプログラム断片はどちらかの関心事を修正する上で適切なファイルに記述され、もう一方の関心事を編集する際には編集しにくいモジュール分割にならざるを得ない。さらに、設計時に全ての関心事を想定することは難しく、捉えきれなかった関心事は分割できないという問題もある。

そこで我々は、開発環境によって動的なモジュール分割を支援するシステム Kide を提案する。Kide は事前に開発者に関心事を登録させ、その関心事に着目したい際は関心事に該当するプログラム要素のみを集めた仮

想ファイルを表示させる。仮想ファイルへの編集は元のプログラムにも伝わるため、本来は複数のファイルをまたがって行わなくてはならなかった修正作業を一つのエディタ上で行うことができる。また、関心事に該当しているプログラム断片のみが表示されるので、修正する箇所を探す時間も短縮できる。さらに、Kide はソースコードを含む文書の作成と利用についても支援を行う。

本論文の構成は以下の通りである。2 章では、プログラム言語を用いるモジュール分割と開発環境によるモジュール分割支援に関する関連研究について述べる。3 章では、言語機構のみを用いてモジュール分割する具体的例とその問題点について述べる。4 章では、開発環境を用いて動的なモジュール分割と文書作成を支援するツール Kide を提案し、その詳細について述べる。5 章では、Eclipse プラグインの実装に関する基礎知識と、Kide の実装に関して記載した。6 章では、実際のソフトウェア開発における Kide の利用法を述べる。7 章では、Kide を用いることで開発者の編集作業行数をどれだけ削減できるかという評価を行う。8 章で本論文をまとめる。

第2章　関連研究

本研究は、開発環境の機能を用いることで対話的なモジュール化を支援する。本章では、アスペクト指向を用いたモジュール化、開発環境によりモジュール化支援を行う関連研究について述べる。

2.1　アスペクト指向を用いたモジュール分割

アスペクト指向プログラミング (Aspect Oriented Programming, AOP) を用いることで、オブジェクト指向プログラミング (Object Oriented Programming, OOP) ではモジュール化することができず、異なるファイルに処理が散らばってしまう関心事をひとつのモジュールにまとめることができる。代表的なアスペクト指向言語としては AspectJ、Hyper/J などが挙げられる。

はじめに、AOP における用語を以下で簡単に説明する。[17]

- アスペクト

  複数のモジュールに散らばった関心事をひとつにまとめるための新しいモジュール単位で、ポイントカットとアドバイスから構成される

- アドバイス

  クラスでいうとメソッドに相当し、ポイントカットで指定されたときに実行される処理

- 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し、ポイントカットにより選択される

- 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

次に、AspectJ を用いて AOP の簡単な使用例を示す。図 2.1 は、デバッグのために、メソッドが呼ばれたらログを出力する処理を各メソッドに記

```
public class Hoge {
    public void hoge() {
        Log.log("メソッドが呼ばれました.")
        System.out.println("ほげ");
    }
}

public class Fuga {
    public void fuga() {
        Log.log("メソッドが呼ばれました");
        System.out.println("ふが");
    }
}
```

図 2.1: OOP での簡単な記述例

述している。もし、この処理全てを取り除きたい場合、記述された全てのメソッドを探して修正を行う必要がある。この例では 2 つのクラスのみを修正すればよいが、大規模システムのプログラムの場合、100 を超えるクラスを行き来する必要があるかもしれない。このような関心事を横断的関心事と呼ぶ。この関心事をアスペクトモジュールに分離した例が図 2.2 である。ログ出力処理は全て Logging アスペクト内にまとめられている。logMethodCall ポイントカットで、Hoge クラスの hoge メソッドと Fuga クラスの fuga メソッドの実行を指定し、その下のアドバイスで logMethodCall ポイントカットに該当するときの前にログ処理を織り込むよう指定している。このようにひとつのアスペクトモジュールにまとめられたことで、修正作業をこのモジュール内で完了することが可能になる。

このようなプログラミング言語によるモジュール化にはひとつ問題があるが、これは次の章で述べることにする。

2.2　開発環境を用いたモジュール分割の支援

プログラミング言語のモジュール化機能を強化するのではなく、それを使用する開発環境によってモジュール化を支援することもできる。本研究もそのアプローチを取っているが、ここでは同様に開発環境によるモジュール化支援を行う関連研究 [1, 8, 5, 9, 10, 7] の一部を取り上げ、それぞれの概要と問題点について述べる。

```
aspect Logging {
  pointcut logMethodCall():
    (execution(void Hoge.hoge())
     || execution(void Fuga.fuga()))

  before: logMethodCall() {
      Log.log("メソッドが呼ばれました.")
  }
}

public class Hoge {
    public void hoge() {
        System.out.println("ほげ");
    }
}

public class Fuga {
    public void fuga() {
        System.out.println("ふが");
    }
}
```

図 2.2: AOP での簡単な記述例

2.2.1　Eclipse

概要

Java 言語を Eclipse 上で用いることを最初に考える。Eclipse の機能を用いると、複数のモジュールに散らばった処理の探索が容易になる。例えば、先ほどの図 2.1 のようなプログラムに Call Hierarchy View (図 2.3) を用いることで、ログ出力処理を行っているメソッドの一覧を取得することができる。ビューア上に表示された各要素をクリックすることでその箇所に飛ぶことができるため、開発環境を用いない場合より容易に修正作業を行うことができる。他にも Type Hierarchy View (図 2.3) を用いるとクラスの継承関係を閲覧することができ、親子関係にあるクラスを修正したい場合に利用できる。これらの機能は、全て GUI 経由で利用できるために、新しい知識を覚えることなく利用することができる。

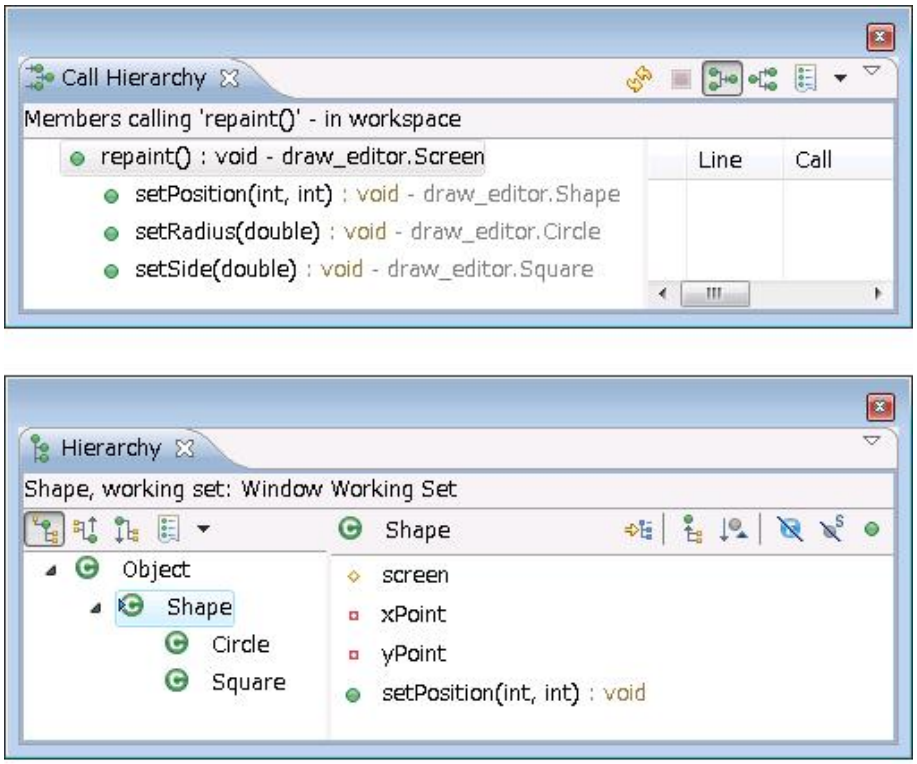

図 2.3: Eclipse の Call Hierarchy View と Type Hierarchy View

問題点

ビューア経由で該当箇所へ飛ぶことができるが、複数のファイルをまたがって修正を行うことに変わりはない。そのため、大規模なプログラムを修正する場合は、ビューアに表示される要素が莫大になり、どこまで修正したかを覚えながら作業を進めなくてはならない。

2.2.2　AspectJ Development Tools(AJDT)

概要

AJDT[1] は，AspectJ のための開発支援ツールであり、織り込まれるアドバイスをエディタ上に表示する。AspectJ などの AOP 言語をを用いて特定の処理をアスペクトモジュールに抜き出すと、別の関心事を閲覧するときに一部処理が他のモジュールに分離されていることでプログラムの閲覧が難しくなるという問題がある。ここで AJDT も用いることでアドバイスが織り込まれる箇所が分かるため、閲覧性は維持することができる。

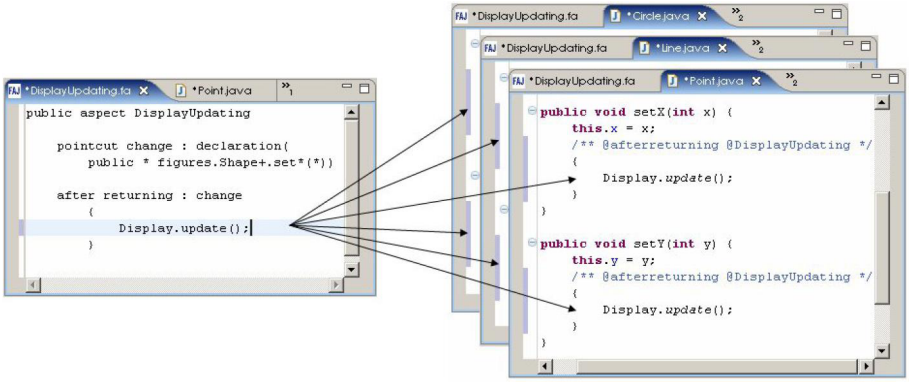

図 2.4: Before/After/ITD JPM

問題点

実際にその関心事に関して修正を行う場合には、そのモジュールだけでなく、織り込まれるアスペクトモジュールにも編集を行わなくてはならないため、複数ファイルをまたがる修正作業が必要になってしまう。

2.2.3　Fluid AOP

概要

Fluid AOP[8] は，AOP の機能を IDE の機能によって実現するツールである。Fluid AOP では 3 種類の方法で横断的関心事に関連するコードを閲覧することができる。一つ目が Before/After/ITD JPM (図 2.4) という方法で，ビューア上でポイントカットを事前に記述しておき，ビューア上での書き込みをそのポイントカットに該当する部分全体に反映させることができる。二つ目が Gather JPM (図 2.5) という方法で，記述したポイントカットに該当したプログラムを一箇所のビューアに表示し，編集することができる。三つ目が Overlay JPM (図 2.6) で、ポイントカットに該当した全てのコードの中から，共通部分のみを表示するビューアを作成することができる。

問題点

Fluid AOP の問題点として、関心事をポイントカットを用いて定義しなくてはならないことが挙げられる。この方法では、関心事に該当するコードを全て抽出できているかどうかの判断が難しい。[15, 6] また、ポ

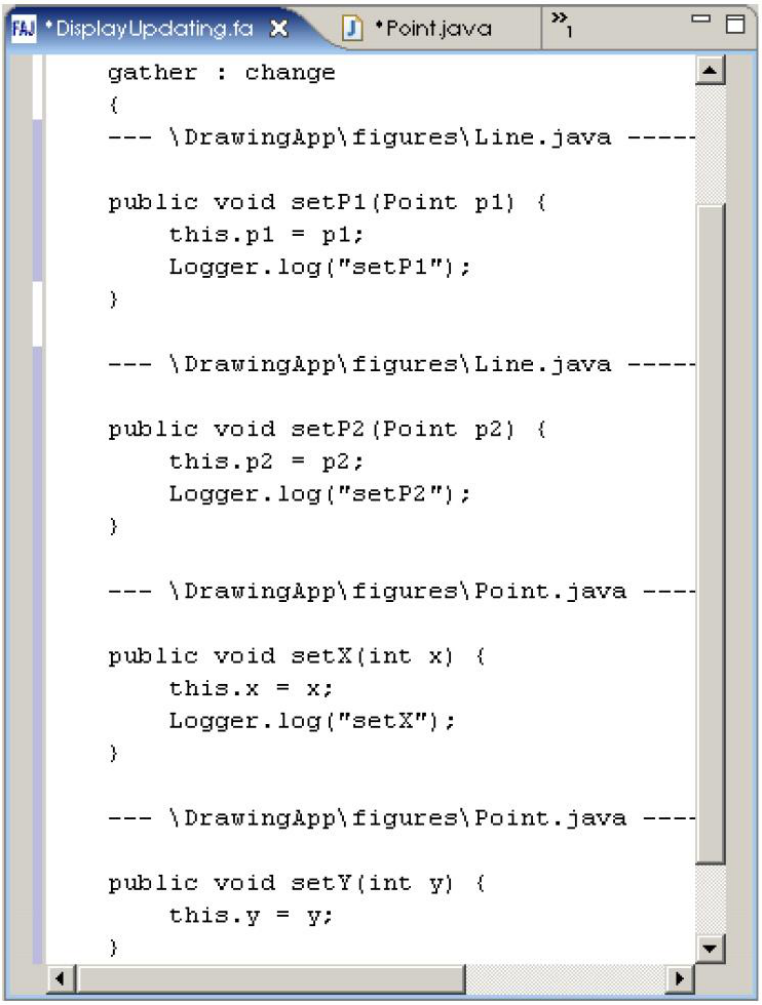

図 2.5: Gather JPM

イントカットに該当したメソッド全てを表示するので、抽出する対象に長いメソッドがあった場合、その大部分が関心事に関連しないこともあり、閲覧性の低下につながることがある。

2.2.4　Code Bubbles

概要

Code Bubbles[5, 3, 4] は，プログラム断片を図 2.7 のように bubble として表示することで細かな粒度での編集を可能にしている．図 2.7（3）のように，開発者の着目するメソッドのみを集めることができる．また，IDE

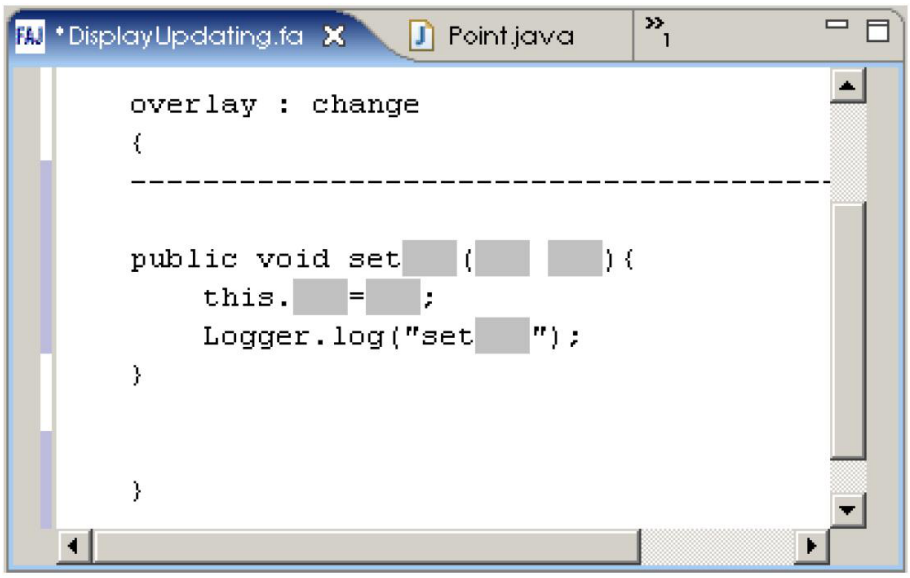

図 2.6: Overlay JPM

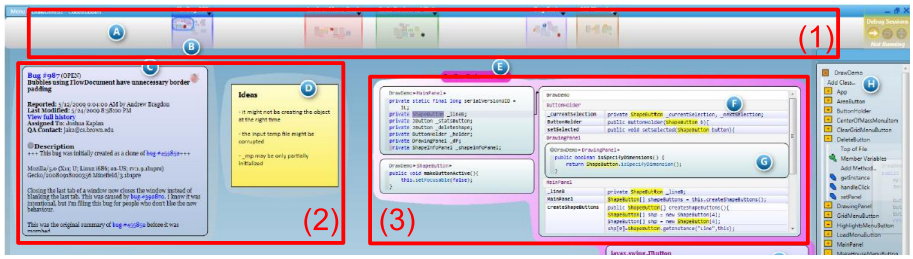

図 2.7: CodeBubbles

上で開いている bubble の集合を workspace （図 2.7（1））に登録することができるので，関心事の推移に合わせて表示するプログラム断片も変えることができる．さらに Code Bubbles では，図 2.7（2）のように，プログラム以外の要素も登録することができる．

問題点

しかし，Code Bubbles には関心事を全て手作業で集める必要があり、構文上の規則性を用いて関心事を定義することができない。そのため、大規模プログラムに Code Bubbles を用いるときに関心事の定義コストが増えてしまうことが予測される。また、Code Bubbles では、プログラムをメソッド単位の粒度でしか集めることができず、Fluid AOP と同様にメ

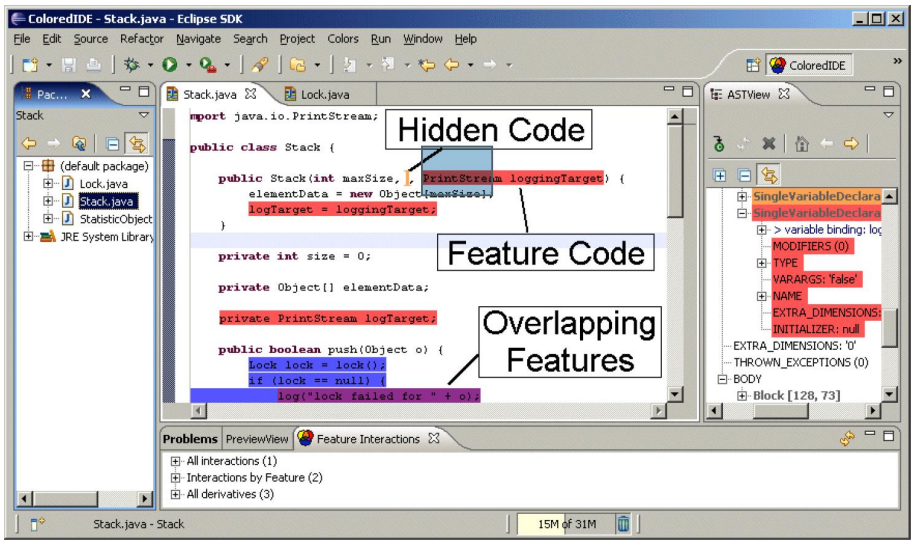

図 2.8: ColoredIDE

ソッドの一部のみが関心事に該当する場合でもメソッド全体を閲覧しなくてはならない。

2.2.5　Colored IDE(CIDE)

概要

CIDE(図 2.8)[9] は，色づけを用いてモジュール分割を行うツールである．OOP や AOP のようなプログラムの分割によるモジュール化手法では，メソッド単位という荒い粒度でしかモジュールが分割できない．C/C++ などの明示的なタグを用いる Annotative なモジュール化手法を用いればこの問題は解消できる反面，コードを難読化する傾向があった．そこでタグの代わりにプログラム断片に色付けを行い，同じ色を付けたプログラム要素は同一の関心事に属することとした．また，同一関心事に属したプログラム断片をアスペクトとして出力することができる．

問題点

CIDE ではプログラムの AST を用いて上記機能を提供しているため、プログラムのみを対象としたモジュール化には適しているが、文章を含むようなモジュールを作成することはできない。

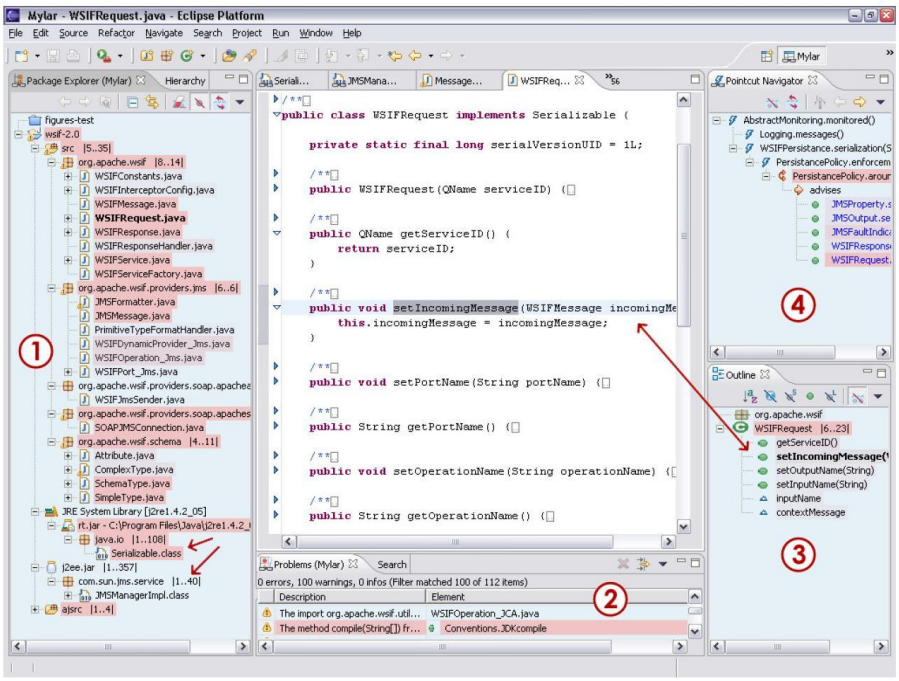

Figure 2: Mylar views (figure numbers correspond to list items above)

図 2.9: Mylar を用い、大規模システムを実装したさいの IDE ウィンドウ

2.2.6　Mylar

概要

Mylar[10] は，開発者が行っているタスクごとに Java や AspectJ で書かれたプログラムをモジュール化することができる。タスクとはコーディングやテストなどの一連の作業を指し，開発者が事前に Mylar に登録する必要がある。Mylar は各プログラム要素に DOI(degree-of-interest) という数値基準を与え，開発者の作業を監視し，修正などが頻繁に行われるプログラム断片は現在のタスクに関わりが深いと判断し DOI を上昇させる。逆に長い時間修正対象にならなかった断片は現在の作業に関係が薄いとみなし DOI を下げる。そして、Mylar は開発者が行っているタスクに関連の深い DOI の高いプログラム要素のみを Mylar Outline や Mylar Packege Explorer などに表示する (図 2.9)。そのため、開発者は行っているタスクに関連する要素のみが表示された環境で開発を行うことができる．実際に開発者が開発時間の中で編集作業にあてた時間の割合が 15 パーセント向上する結果が得られている。

問題点

Mylar は関心事に該当するプログラム断片をビューア上で表示する機能のみを提供している。そのため、修正箇所はビューア上で閲覧できるが、実際に修正作業を行うには各ファイルを開かなくてはならない。そのため、複数のファイル間を行き来する編集は強いられてしまう。また、手動で関心事を定義することはできないので、関心事に該当するプログラム要素が全て集められている保証はない。

第3章　言語機構によるプログラムのモジュール分割とその課題

近年，様々なプログラムのモジュール分割手法やそれを実現するプログラミング言語が提案されている．例えば前章で挙げたアスペクト指向はオブジェクト指向ではモジュール化できない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。他に Hyper/J[12] という AOP 言語を用いると静的にクラスを細かいモジュールに分割し、それを組み直すことができるため、その細かいモジュールにそれぞれの関心事を分離することが可能である。しかし，プログラミング言語のモジュール化機能が高度になったとしても、プログラミング言語の言語機構だけを用いて開発者にとって常に最適なモジュール分割を行うことは難しいと我々は難しいと考える．なぜならば、プログラムはコーディング時に、一次元のテキストとして静的に配置され、それぞれのファイルに分割される。それに対して、開発者の注目する関心事は開発状況によって推移する。例えば、ログ処理を修正するために最適なモジュール化をした場合、ログ処理に対する修正は行いやすい。しかし、開発が進むと当然それ以外の修正を行わなくてはならず、事前に行ったモジュール化がその修正に適しているかは分からない。さらに、開発段階では捉えられなかった関心事が保守段階になって見つかった場合も、その関心事への修正が行いにくいモジュール化が行われているかもしれない。[16, 13] 本章では、この問題をより具体的なプログラムを用いて述べる。

3.1　言語機構によるプログラムのモジュール分割例

図 3.1，図 3.2 は，AspectJ [11] を用いて同一機能の図形エディタプログラムを異なる切り口からモジュール分割した例である．

図 3.1 は，図形の種類に注目してモジュール分割したプログラムである．長方形に関する機能は Rectangle クラス，円に関する機能は Circle クラス，各図形に共通する機能は継承機能を用いて Shape クラスにまとめている．それぞれの図形の位置、大きさが変わった場合には screen.repaint() を呼び、スクリーンへ再描画の通知を行っている。また、面積が変わった場合には Logging.log(”Size changed.”); を呼び、ログ出力を行ってて

```
[Shape.java]
  public class Shape {
      private int xPoint, yPoint;
      protected Screen screen;

      public void setPosition(int x, int y) {
          xPoint = x; yPoint = y;
          screen.repaint();
      }
  }

[Circle.java]
  public class Circle extends Shape {
      private double radius;

      public void setRadius(double r) {
          Logging.log("Size changed.");
          radius = r;
          screen.repaint();
      }
  }

[Rectangle.java]
  public class Rectangle extends Shape {
      private double width, height;

      public void setWidth(double w) {
          Logging.log("Size changed.");
          width = w;
          screen.repaint();
      }

      public void setHeight(double h) {
          Logging.log("Size changed.");
          height = h;
          screen.repaint();
      }
  }
```

図 3.1: 図形の種類に注目したプログラム分割の例

いる。実装言語として AspectJ を用いていると述べたが、この分割では AspectJ 特有の機能は特に用いずに分割ができており、Java で実装することも可能である。各図形ごとにクラスを作り実装がまとめられているため、特定の図形に関する修正はその図形を表すモジュール内だけで完了することが可能である。

図 3.2 は，ログ出力処理や再描画処理といった処理の種類に着目してモジュール分割したプログラムである。このプログラムでは AspectJ 特有の機能を用いて、Logging, Repainter のそれぞれのアスペクトモジュール

```
[Shape.java]
  public class Shape {
          :
      public void setPosition(int x, int y) {
          xPoint = x; yPoint = y;
      }
  }

/*
 * [Circle.java], [Rectangle.java] の set* メソッドから
 * Logging.log("Size changed.");, screen.repaint(); を取り除く
 */

[Logging.aj]
  aspect Logging {
      pointcut logSizeChanged():
          (execution(void Circle.setRadius(double))
           || execution(void Circle.setWidth(double))
           || execution(void Circle.setHeight(double)));

      /** Output log */
      before(): logSizeChanged() {
          Logging.log("Size changed.");
      }
  }

[Repainter.aj]
  aspect Repainter {
      pointcut repaintedMethods():
          execution(void set*(..));

      /** Updates screen */
      after(): repaintedMethods() {
          screen.repaint();
      }
  }
```

図 3.2: 処理に注目したのモジュール分割の例

にログ出力と再描画処理に関するコードをまとめている。そのため、例えば，システム内全ての再描画処理に適用させる必要のある修正を行いたい場合は、 Repainter アスペクトモジュール内で完了することができる。

3.2　言語機構によるプログラムのモジュール分割の問題点

このように開発者はプログラミング言語の言語機構を用いて，着目する関心事に合わせた適切なモジュール分割を行うことができる．近年の高

級なプログラミング言語を用いれば、事前に見つけたほぼ全ての関心事も同様にモジュール化し、ひとつのモジュールにまとめることが可能だと考えられる。しかし、開発者が常に最適な視点から編集を行えるようなモジュール化は存在するだろうか。我々は存在しないと考える。なぜならば、開発者は開発時にモジュール分割をひとつ選択しなくてはならず、その分割が適さない関心事を編集する場合は煩雑な作業が強いられてしまうからである。例えば、上記の 2 つのモジュール分割のいずれかを選択した場合を考える。設計時から図形の種類に関する修正が予測されているならば図 3.1 の分割を選択すればそのような修正を容易に行える。しかし，途中でログ出力を全て取り除く必要が出たときにはログ出力処理を行っている全てのモジュールを行き来する煩雑な修正が必要になってしまう．これを避けるには図 3.2 のモジュール分割を選択すればいいが，今度は各図形を移動させる部分でバグを発見した場合に修正が難しい．このモジュール分割では，図形移動に関連する処理が Shape クラスと Repainter アスペクトに分散しており，やはり複数ファイルを横断する修正が課されてしまうからである．図形移動に関する修正を行うという観点から見ると図 3.1 のモジュール分割の方が良い．理想的には 2 つのモジュール分割を動的に切り替え、編集対象に合う視点から作業が行えればよいのだが、冒頭で述べたようにプログラミング言語の言語機構だけを使う手法では難しい．状況によって適切なモジュール分割というものは変わってくるため，言語機構だけでは常に最適なモジュール分割を行うことは難しいといえる．[14] また Hyper/J を用いた場合も、どのようにモジュールを組みかえるかは静的に決定しなくてはならない。異なる視点からの編集を行うには、プログラムの一部を書き換えてモジュールを組み替える必要があるため、行き来する作業を減らすために新たな労力が必要になってしまう。

第4章　Kide による対話的なモジュール分割と文書作成の支援

4.1　Kide を用いた対話的なモジュール分割

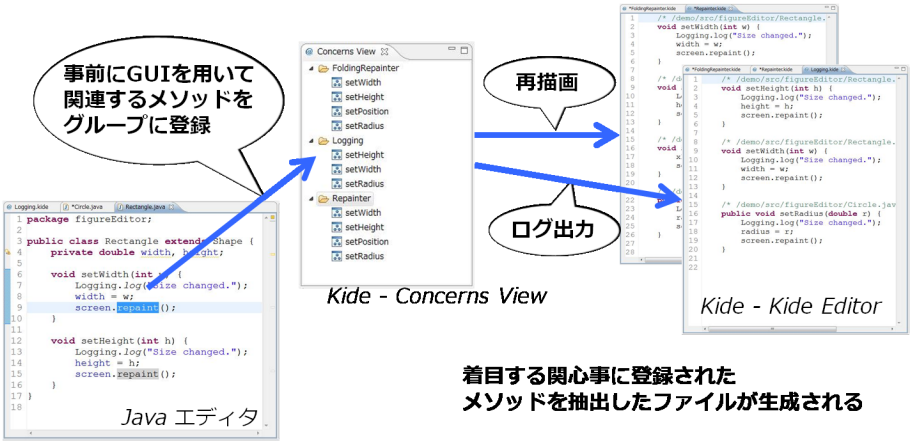

図 4.1: Kide の利用方法の流れ

我々は開発環境を用いて対話的なモジュール分割を支援するツール Kide を提案する。Kide は静的に分割されたモジュールとは異なる分割でプログラムを表示する。Kide の利用の流れを図 4.1 に示した。まず Kide に関心事を登録する。関心事にはメソッドかメソッドの一部を登録することができる。関心事定義についての詳細は 4.1.2 で述べる。登録された関心事は Concerns View 上に表示される。開発者は登録した関心事を編集する必要が出たら Concerns View から Kide エディタを開くことができる。Kide エディタには登録されたプログラム断片のみが表示されるため、編集作業に集中することができる。Kide エディタに関する詳細は 4.1.1 で述べる。

4.1.1　元のモジュール分割とは異なる分割でプログラムを表示する Kide エディタ

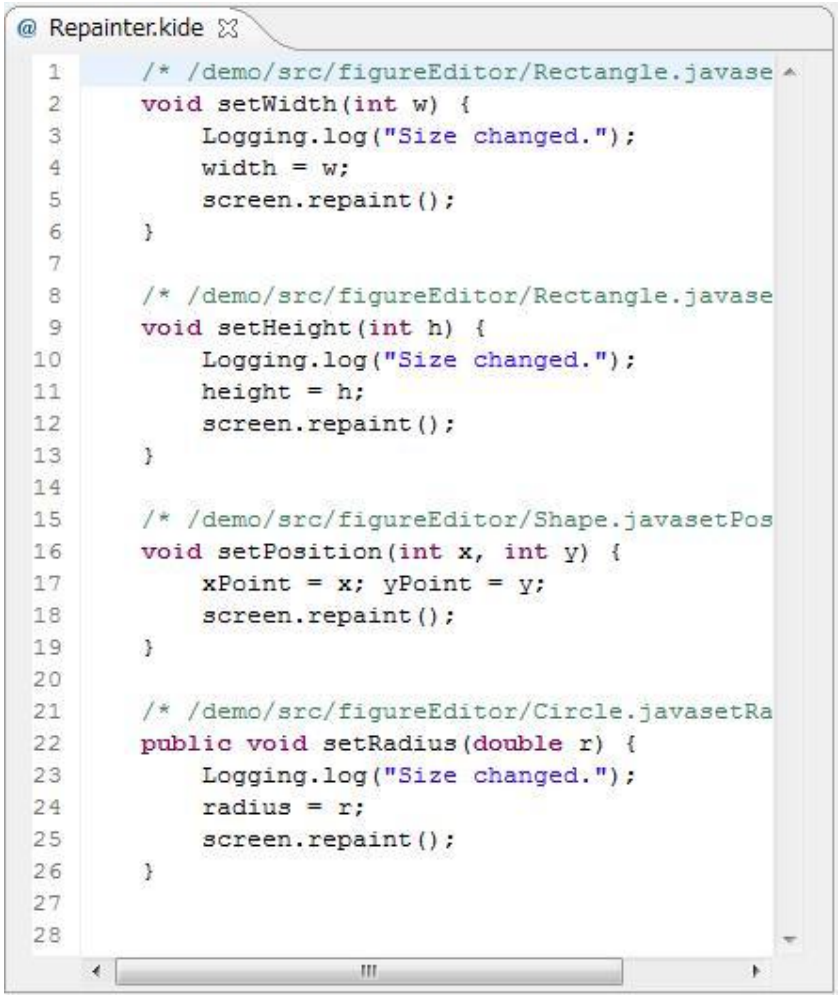

図 4.2: 再描画処理を含むメソッドを集めた Kide エディタ

開発者は Kide を用いることで静的には異なるモジュールに属しているが共通の関心事に該当しているプログラム断片群を仮想ソースファイルに集め，Kide エディタ上に表示できる．Kide エディタに表示する関心事は開発者の関心の推移に合わせて対話的に変更することができるため，常に開発者は注目する関心事を閲覧するのに最適な視点からプログラムを閲覧できる．例えば，図 3.1 のプログラムを再描画処理に着目して閲覧したい際には図 4.2 のような Kide エディタを開くことができる．本来は Shape, Circle, Rectangle の 3 つのファイルに別々に定義された 4 つのメソッド

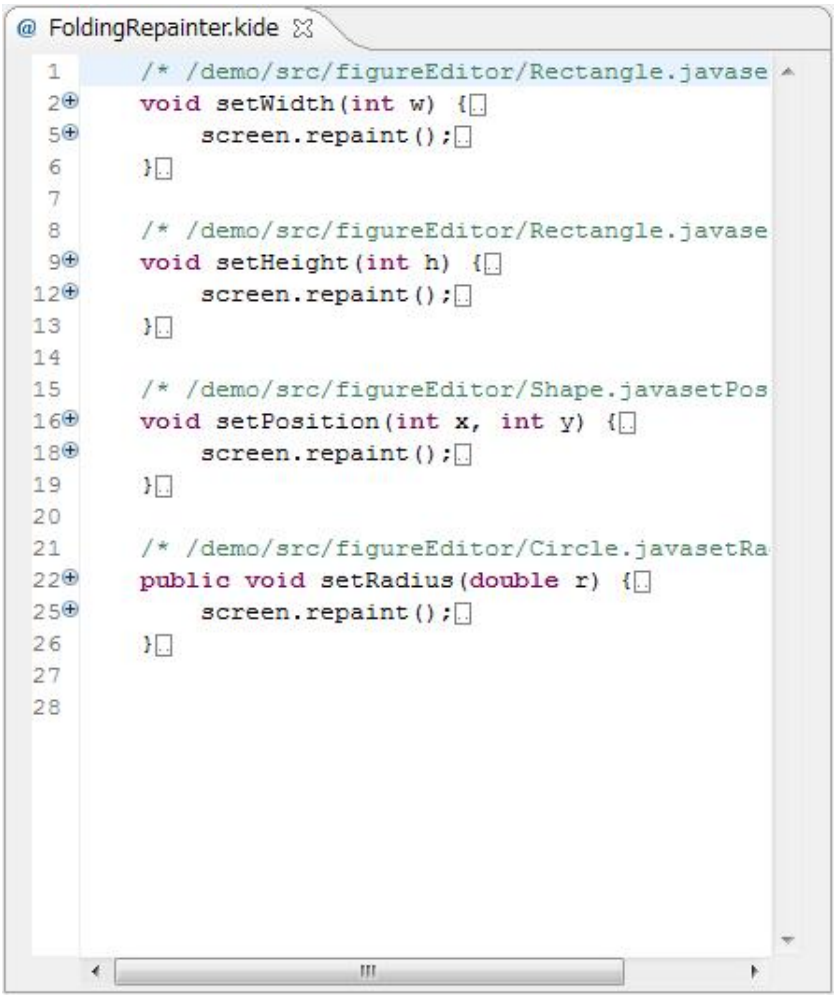

図 4.3: 折りたたみ機能を用いて再描画処理部分のみを表示した KideEditor

がひとつのエディタ上で閲覧できる．プログラム断片は基本的にはメソッド単位であるが，メソッドボディの一部のみを関心事とし，他の部分を折りたたんで表示することもできる．図 3.1 のプログラムのうち，折りたたみ機能を使って再描画処理の部分だけを表示した例が図 4.3 である．例えば、再描画処理を含む 100 行のメソッドがあったとする。そのメソッドのうち再描画処理に関する部分は 2 行だけっだとしても、上記の Kide の表示方法ではこのメソッドを全て表示しなくてはならず、結局編集箇所を探す必要がでてきてしまう。そのようなときに、この機能を用いることで関心事に該当する 2 行のみを登録することで、その部分のみが表示される。再描画処理に関する Kide エディタが開かれるときは、再描画処理の修正を行いたいときのみなので、2 行のみが表示されれば十分である。ま

た、Kide エディタ上での編集は元のソースファイルに伝わるため，開発者はひとつのエディタ上で複数ファイルにまたがる編集を完了することができる．

4.1.2　Kide における関心事の定義方法

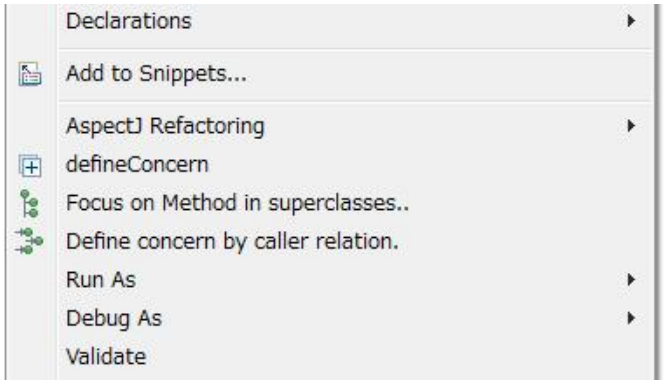

図 4.4: Kide 独自のポップアップメニュー

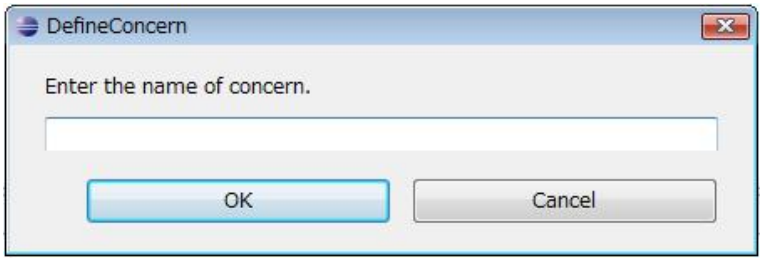

図 4.5: 関心事を入力するダイアログ

Kide で扱う関心事はプログラム構造からは自明ではなく，GUI を用いて定義する必要がある．例えば，図 3.1 において再描画処理に注目する関心事 Repainter を定義することを考える．まず，標準の Java エディタで再描画処理を含む Java ファイルを開く．そして，再描画処理を行っており，関心事として登録したいメソッドの名前をマークし，Kide 独自のポップアップメニュー (図 4.4) である”Define Concern”という項目を選択する．すると、関心事名を入力するダイアログ (図 4.4) が開かれるので，ここに”Repainter”と入力することでそのメソッドが関心事”Repainter”として登録される．また，メソッドの一部分のみを関心事に加えたい場合

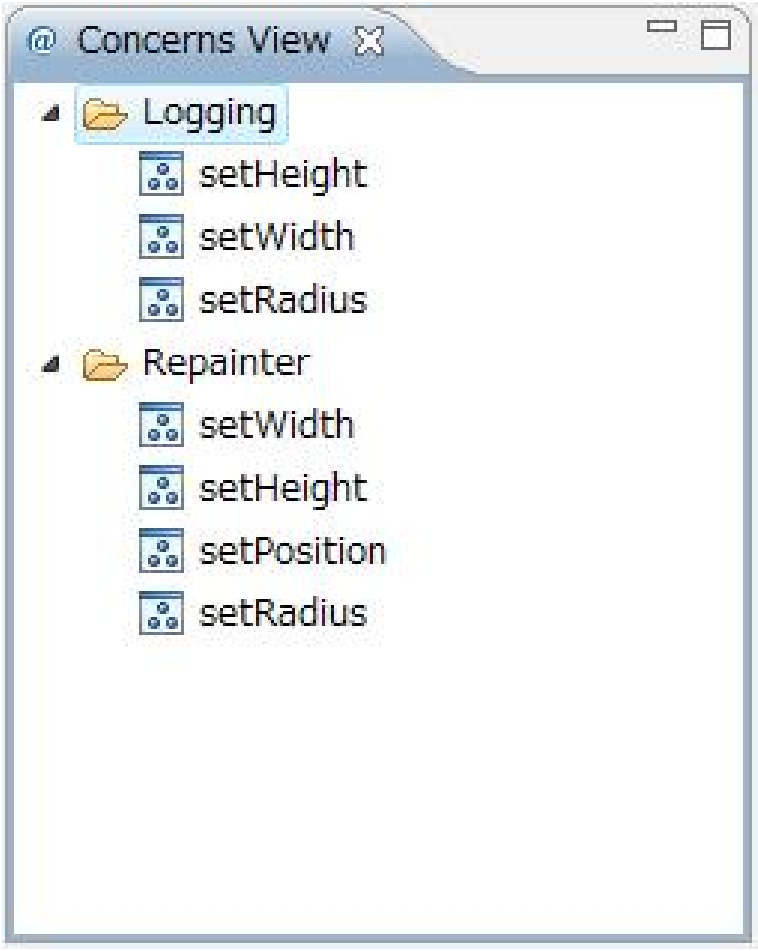

図 4.6: Concerns ビューア

は，その部分をマークし，その後は上記した方法と同様に登録できる．定義された関心事は ConcernsView(図 4.6) で閲覧でき，このビューア上で関心事を選択することでその関心事に登録されたプログラム断片を集めた Kide エディタを起動できる．

プログラム断片の構文上の規則性を用い，複数のプログラム断片を関心事に一括登録することも可能である．現状の Kide では，呼び出し関係と継承関係を用いた一括定登録ができる．呼び出し関係は指定したメソッドを呼び出しているメソッド群を，継承関係は指定したメソッドとそのメソッドをオーバーライドしているメソッド群を登録候補としてウィザード上でユーザに提示する．そのウィザードが図 4.7 である．開発者は関心事の名前を入力し，提示された候補の中から関心事として登録したいメソッドにチェックを入れ，一斉登録することができる．この支援も全て GUI

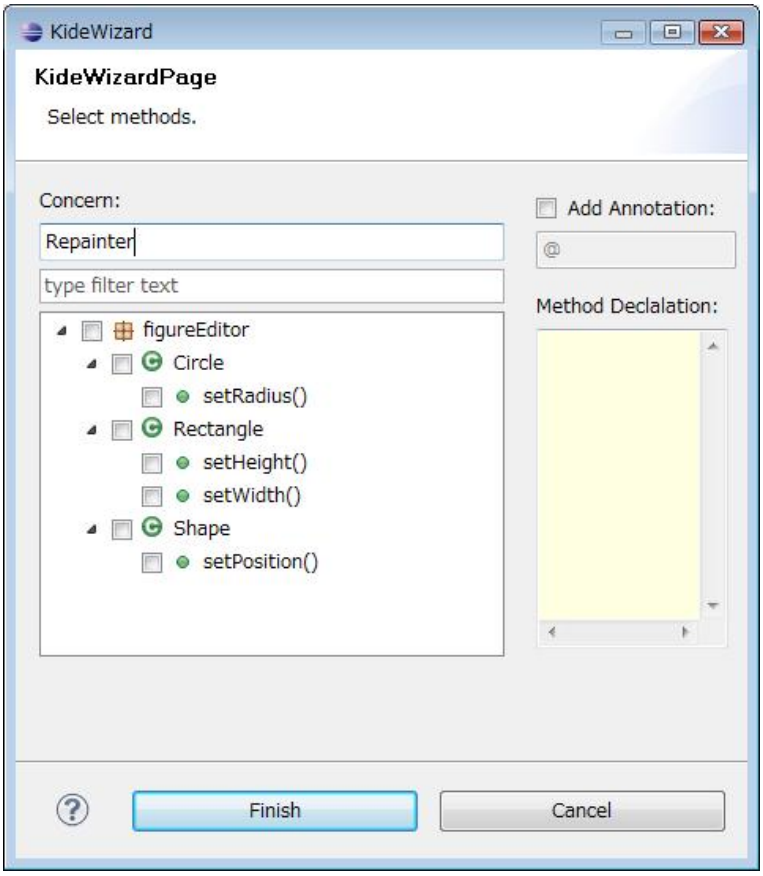

図 4.7: Kide ウィザード

から利用することができる．

また，開発者間でソースコードと共に定義した関心事を受け渡すことができる．関心事は Kide 内データベースに定義されているが，ポップアップメニューから外部ファイルに書き出すことができる．ソースコードを受け取る側はそのファイルをポップアップメニューから読み込むことで関心事を共有できる．

4.2　Kide による対話的な文書作成と利用の支援

Kide では Readme やコードレビュー文書のようなソースコードを含む文書の作成も支援する．このような文書を作成するには，掲載したいプログラム断片群を関心事として定義し，それを開いた Kide エディタ上で断片と断片の間にテキストを挿入していけばよい．図 4.8 が Kide を用いて

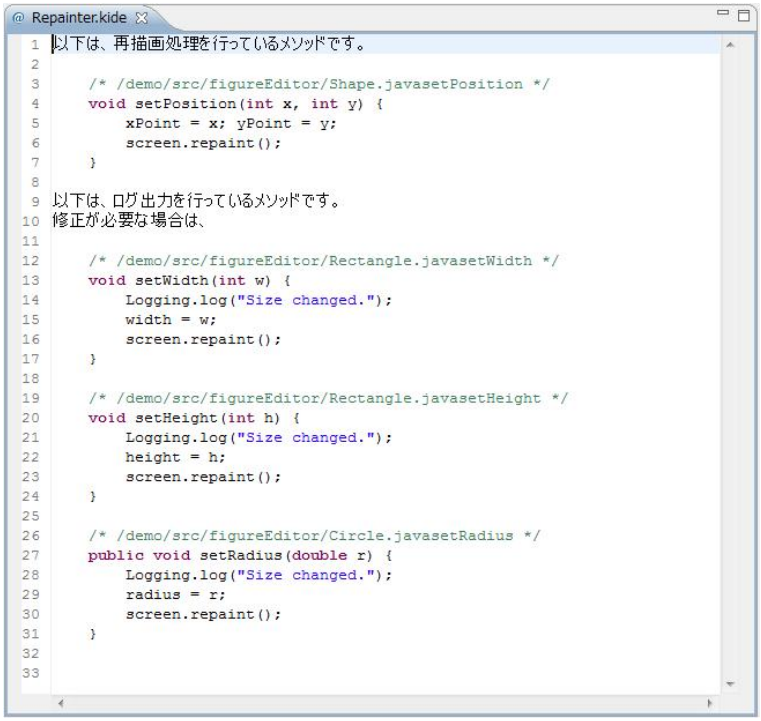

図 4.8: Kide による文書作成

作成した文書の例である。このような文書を作成すると，ソースコード側で行われた修正が，文書上の同一箇所にも適用される．また，Kide エディタ上でプログラム断片に対して行われた修正も元のプログラムに伝わる．従来，このような文書は掲載するプログラム断片をソースコードからコピー&ペーストし作成される．しかし，この方法では元のプログラムが修正されたときに文書側にも同様の修正が必要になってしまう．Kide を用いることでこの煩雑な作業を避けることができる．

第5章　実装

我々は、Kide を Eclipse プラグインとして実装した。本章では、Eclipse プラグインを開発する上での基礎知識と、本システムの実装に関して記載する。

5.1　Eclipse プラグイン開発に関する基礎知識

5.1.1　Eclipse のアーキテクチャ

Eclipse は Java を含む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する開発環境である。そのため、機能を追加しやすいプラグイン・アーキテクチャを採用している。Eclipse のアーキテクチャを図 5.1 に示す。

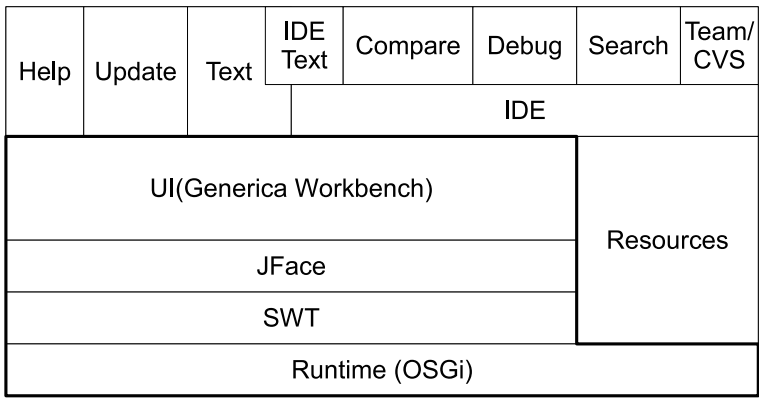

図 5.1: Eclipse のアーキテクチャ

この中で、Ecliopse のプラグイン・アーキテクチャの期間をなすのが最下層に位置している OSGi ランタイムである。OSGi ランタイムをプラグイン管理のために用いる。OSGi の上位に配置されているコンポーネント群は全てプラグインとして提供されており、拡張性に富んだ設計となっている。

また、Java 開発環境自体もプラグインとして実装されている。Java 開発環境は、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 モデルの一覧を図 5.2 に示す。Java モデルはクラス、フィールド、メソッドなどの Java ファイルの構文上の内容を表現するものが、それぞれ IType, IField, IMethod というインターフェースとして実装されている。図 5.2 に示したように、Java モデルを表すクラスと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。

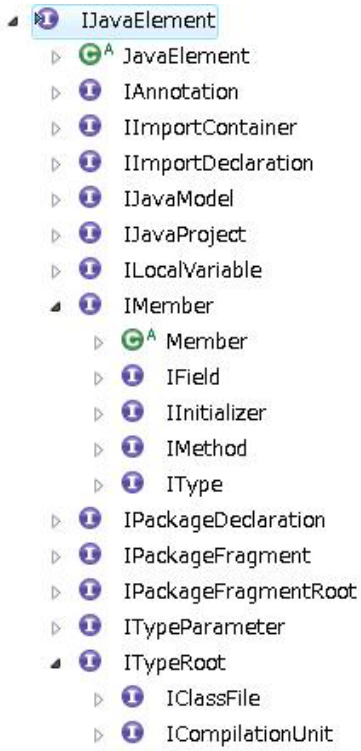

図 5.2: Eclipse の Java モデル

Java モデルを扱う簡単なコード例として、プロジェクトの名前やメソッドシグネチャが分かっている状態でメソッドを表すインスタンスを取得するメソッドを図 5.3 に示す。このように、メソッドをはじめとした全ての Java 要素をインスタンスとして取得することが可能である。

5.1.2　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と呼ぶ。ここでは、ワークベンチの構成と各構成要素

```
IJavaModel javaModel =
  JavaCore.create(ResourcesPlugin.getWorkspace().getRoot());

// プロジェクト projectName 中にあるパッケージの一覧を取得し、
// packegeName に該当するパッケージを取得
IPackageFragment[] packFrags =
  javaModel.getJavaProject(projectName).getPackageFragments();
IPackageFragment pack = null;
for(IPackageFragment p: packFrags){
  if(p.getElementName().equals(packageName)){
    pack = p;
    break;
  }
}

// パッケージ pack 中にある typeName クラスのメソッド群を取得し
// methodName メソッドを取得
IMethod[] methods =
  pack.getCompilationUnit(CompilationUnitName).getType(typeName).getMethods();
for(IMethod m: methods){
  if(m.getElementName().equals(methodName)
    && m.getSignature().equals(methodSignature))
    return m;
}
```

図 5.3: IMethod インスタンスを取得するプログラム例

の役割を説明する。Eclipse のワークベンチを図 5.4 に示す。

ワークベンチウィンドウ
Eclipse の大枠であるウィンドウ。通常はワークベンチひとつに対して、ひとつワークベンチウィンドウが開かれるが、複数のウィン号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一つのページが対応し、ビューアとエディタによって構成される。

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を選択でき、開発フェーズによって使い分けることが可能で、開発者が自ら定義することも可能である。

ビューア
エディタ上での開発を支援する役割を担う。特定の文字列でプロジェ

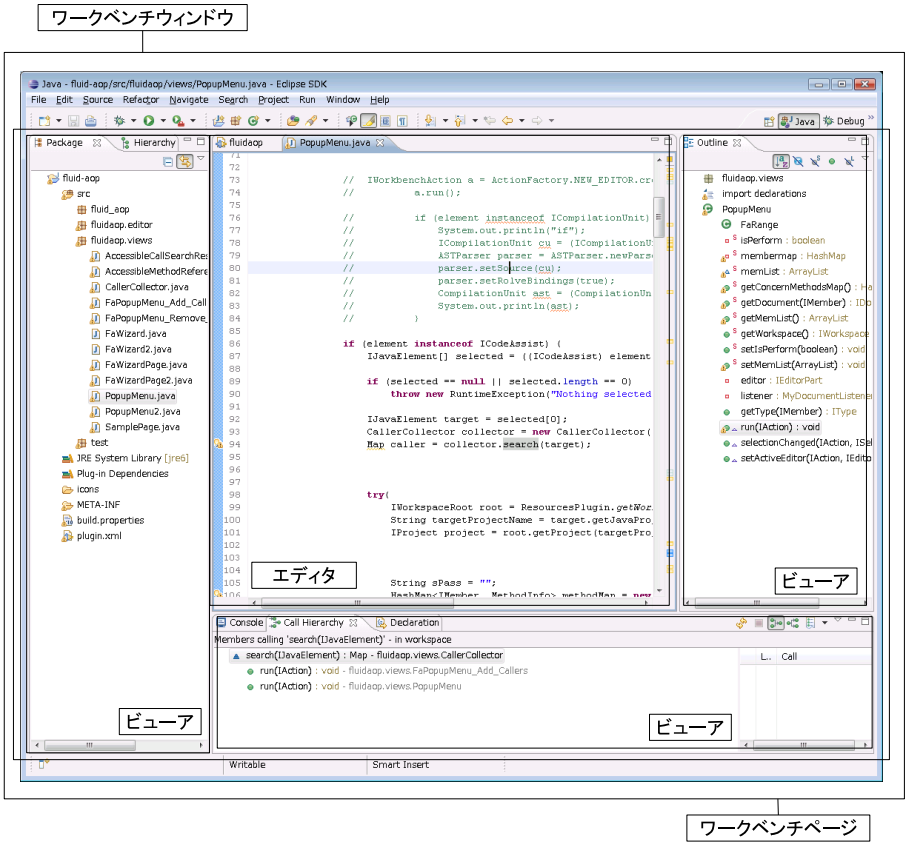

図 5.4: Eclipse のワークベンチ

クト内を走査する Search ビューア、指定したフィールドやメソッドを呼び出しているメソッドの一覧を示す Call Hierarchy ビューアなどがある。主なビューアを図 5.5 に示す。

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できる。また、同一ファイルを複数のエディタで開くことも可能である。

本システム Kide のようなプラグイン開発では、プラグインの実装から Eclipse ワークベンチの各機能にアクセスする必要がある。例えば、Java エディタで開いているファイルへの修正を常に監視する必要があり、その際はアクティブになっている Java エディタを取得する必要がある。そのプログラムを図 5.6 に示す。PlatformUI クラスの getWorkbench メソッドを用いることでワークベンチを取得することができる。ワークベンチは複数起動されている可能性があるが、この場合はアクティブなもののを取

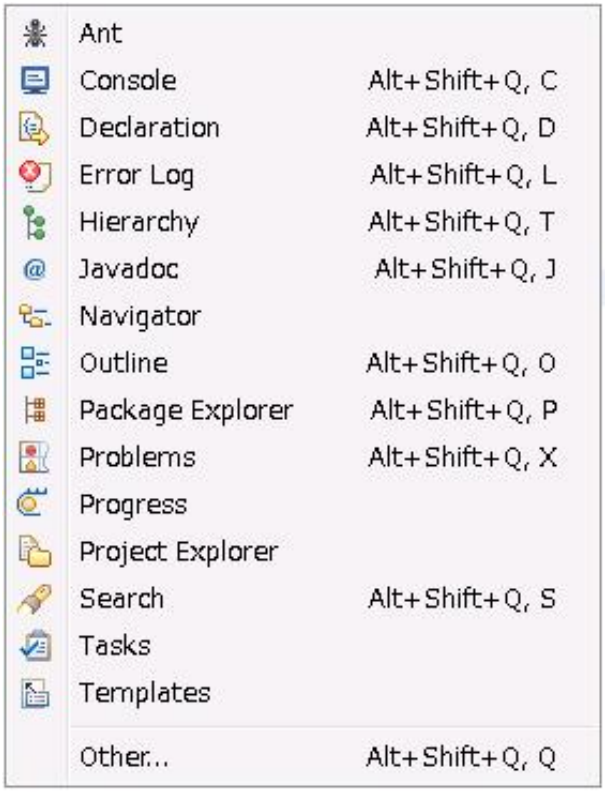

図 5.5: Eclipse の主なビューア一覧

```
// ワークベンチを取得
IWorkbench workbench = PlatformUI.getWorkbench();
// ワークベンチウィンドウを取得
IWorkbenchWindow workbenchWindow = workbench.getActiveWorkbenchWindow();
// ワークベンチページを取得
IWorkbenchPage workbenchPage = workbenchWindow.getActivePage();
// アクティブなエディタの取得
IEditorPart editorPart = workbenchPage.getActiveEditor();
```

図 5.6: アクティブエディタを取得するプログラム

得できればよく、ワークベンチウィンドウ、ワークベンチページ、エディタとそれぞれアクティブなものを取得している。

5.1.3　PDE(Plugin Development Environment)

PDE は Eclipse に標準でつ属されているプラグイン開発環境であり、Java で記述されるプラグインの実装部分とは別に、開発を支援するため

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

図 5.7: MANIFEST.MF

の機能を提供している。その機能の核となるのがマニフェスト・エディタである。新たなプラグインを作成するためには、依存関係や拡張ポイントといった実装する上で必要なプラグインの各種設定をするエディタである。このエディタを用いて、MANIFEST.MF と plugin.xml というファイルを編集することで設定を行う。ツールバーにボタンを追加し、そのボタンをクリックすると Hello, World と書かれたダイアログを表示するプラグインの MANIFEST.MF と plugin.xml を図 5.7 と図 5.8 に示した。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。図 5.9 にマニフェスト・エディタを示す。マニフェストエディタで編集した内容は、MANIFEST.MF、plugin.xml、build.properties の各ファイルに自動的に反映される。マニフェスト・エディタはマルチページエディタであり、画面下のタブでページを切り替えながら設定を行う。各タブの詳細は以下の通りである。

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 に追加する必要がある。

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

```
<?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 5.8: plugin.xml

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。

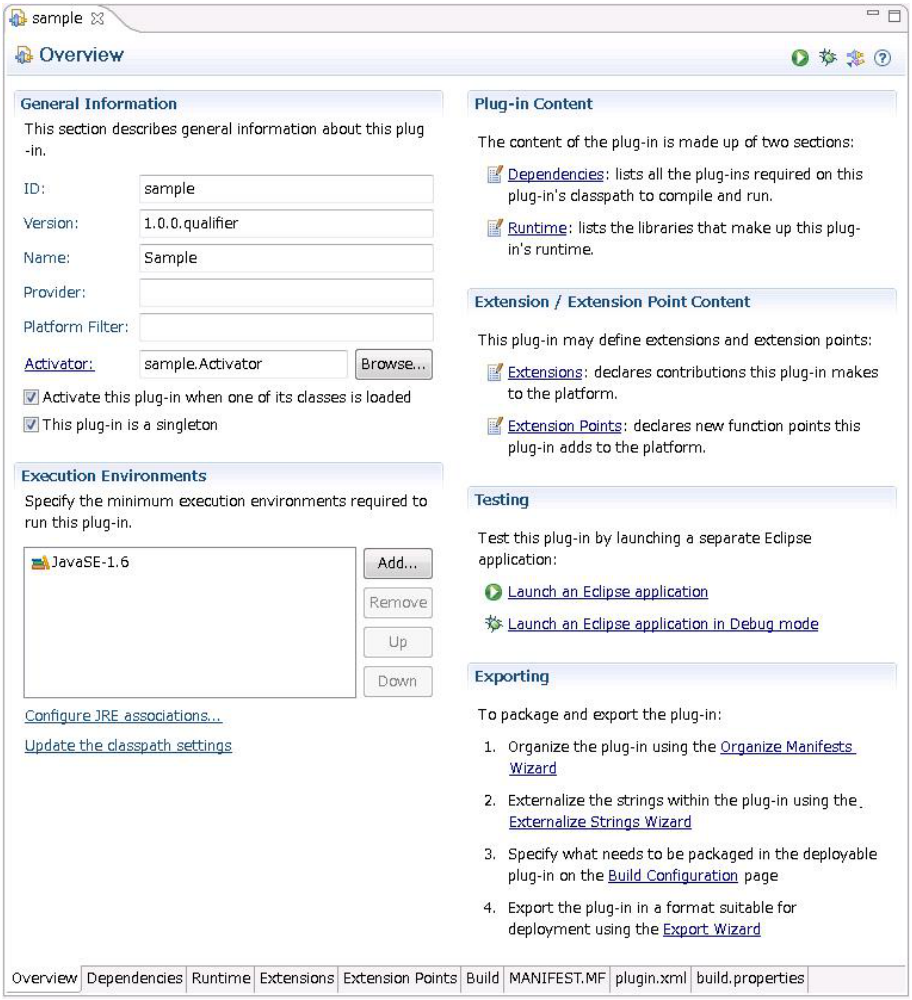

図 5.9: マニフェスト・エディタ

5.2　Kide の実装

ここでは Kide の各機能の実装について簡単に述べる。

5.2.1　Kide データベースに関する実装

Kide では定義された関心事をローカルの MySQL に保存している。MySQL との連携は JDBC という API を用いている。ここでは DB と接続する必要のある機能のプログラムを簡単に示す。

- Kide 初期化時に DB に Kide 用のテーブルを作成する。DriverManager からコネクションを取得し、ステートメントを取得する。

ステートメントの executeUpdate メソッドを用いてテーブルを作成する SQL 文を実行している。

```
private KideDB() {
  try {
    // Load JDBC driver.
    Class.forName(JDBC_DRIVER);

    // Connection to DB.
    System.out.println("Connecting to " + DB_URL);
    connect = DriverManager.getConnection(
        DB_URL, DB_USER_NAME, DB_PASSWORD);
    st = connect.createStatement();

    // make table for Kide if not exists.
    st.executeUpdate("CREATE TABLE IF NOT EXISTS KideConcerns(" +
        "ID INT(10)," +
        "KIND CHAR(1)," +
        "CONCERN VARCHAR(30)," +
        "PRIORITY INT(10)," +
        "PROJECT VARCHAR(100)," +
        "PACKAGE VARCHAR(100)," +
        "FILE VARCHAR(100)," +
        "TYPE VARCHAR(100)," +
        "METHOD VARCHAR(100)," +
        "SIGNATURE VARCHAR(100)," +
        "PATH VARCHAR(100)," +
        "OFFSET INT(10)," +
        "LENGTH INT(10)," +
        "PRIMARY KEY (ID)" +
        ");");
  } catch (ClassNotFoundException e) {
    // ドライバを読み込めなかった場合
  } catch (SQLException e) {
    // データベース接続エラー
  }
}
```

- メソッドが関心事に登録された場合、関心事の名前とメソッドのシグネチャなどを DB に登録する必要がある。登録を行うメソッドは以下の通りである。登録されるメソッドは IMethod インスタンスとして渡される。そこから、JavaProject、Packege、CompilationUnit、Type、メソッドの名前とシグネチャをそれぞれ取得し、DB に文字列の形で登録を行う。

```
/**
   * insert method information to DB.
   * @param m Method which is inserted to DB
   * @param concern Concern of m
   * @return 1: succeed, 2: previously exist, 0: false
   */
  public int insertMethodToConcernDB(IMethod m, String concern) {
    if(!isExist(m, concern)){
      try {
        int id = calcID();
        String projectName = m.getJavaProject().getElementName();
        IType targetType = (IType)m.getParent();
        ICompilationUnit targetComp = targetType.getCompilationUnit();
        IPackageFragment targetPack = (IPackageFragment) targetComp.getParent();
        String packageName = targetPack.getElementName();
        String compName = targetComp.getElementName();
        String typeName = targetType.getElementName();
        String methodName = m.getElementName();
        String methodSignature = m.getSignature();

        int i = st.executeUpdate(/* SQL 文は省略 */)
        return 1;
      } catch (SQLException e) {
        // TODO Auto-generated catch block
        System.out.println("SQLException: " + e);
        e.printStackTrace();
      } catch (JavaModelException e) {
        // TODO Auto-generated catch block
        System.out.println("JavaModelException: " + e);
        e.printStackTrace();
      }
    }else{
      return 2;
    }

    return 0;
  }
```

5.2.2　Kide Editor 起動と Kide Editor への入力を元の Java コードに通知する機能の実装

登録された関心事を Concerns View から Kide エディタで開き、Kide エディタ上での編集と元のプログラムとを同期するモデルを図 5.10 に示した。① で Concerns View 上で要素へのクリックを感知すると、Kide Controller にその関心事を登録する (②)。Kide Controller はその関心事を引数に openKideEditor メソッドを呼ぶ (③)。openKideEditor メソッドの一部を図 5.11 に示した。ここでは簡略化するため、折りたたみ機能や文書が含まれる場合の処理は記載していない。DB から得られた関心事に該当するメンバを取得し、それぞれのプログラム断片を getSource メ

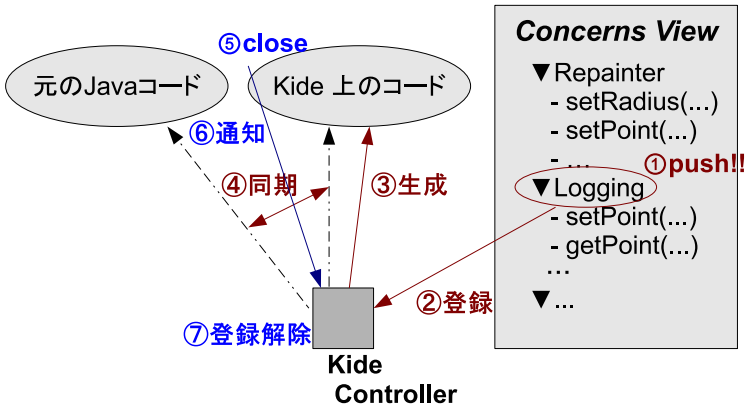

図 5.10: Kide Editor 起動と同期に関するモデル

```
.. openKideEditor(String concern) {
  // 関心事に該当するメンバの取得
  ArrayList<Object> members = kdb.selectConcern(concern);

  /* Kide がメンバを抽出した仮想ファイルのうち、どこからどこまでがどのメ
ソッドに関する記述かを示すリスト */
  ArrayList<MemberRange> mRagneList = new ArrayList<MemverRange>();
  String sorce = "";
  for(IMember mem: members) {
    source = source + "/* " + path + mem.getElementName() + " */\r\n\t";
    source = source + mem.getSource() + "\r\n\r\n\t";
    mRangeList.add(new MemberRange(mem, mem.getSourceRange());
    // 以下、折りたたみ箇所の指定を行うがここでは省略
  }

  // 仮想ファイルを入れるフォルダを取得
  IFolder folder = container.getFolder(new Path("kide"));

  // 仮想ファイルの作成
  String fileName = concern + ".kide";
  IFile file = container.getFile(new Path(KIDE_PATH + fileName));
  file.delete(true, null);
  file.create(new ByteArrayInputStream(source.getBytes()), true, null);

  // Kide エディタの起動
  IEditorPart part = IDE.openEditor(workbenchPage, file);
}
```

図 5.11: openKideEditor メソッドの一部

ソッドで取得している。仮想ファイルを kide フォルダ上に作成し、それを Kide エディタで起動する。次に ④ の同期処理について述べる。同期

```
public void sync(DocumentEvent event) throws Exception {
  if(isOnKideEditor(event)) {
    ..

    ArrayList<MemberRange> memberRanges = targetConcern.getMemberRanges();
    adjustKRange(memberRanges, event);

    // 以下で同期処理を行う
    try {
      MemberRange mRange = getTargetMember(event);
      String newKideText = event.getDocument().get(mRange.getkOffset(), mRange.getkLength());
      Object oMember = mRange.getMember();
      if(oMember instanceof IMember) {
        IMember targetMember = (IMember)oMember;
        int off = mRange.getjOffset();
        int len = mRange.getjLength();

        IDocument doc = Kide.getIDocument(targetMember);
        doc.replace(off, len, newKideText);

        // replace した部分の Range の修正
        adjustJRange(targetMember, memberRanges, event);

      } else if(oMember instanceof KideText) {
        // targetMember が text の場合は省略
      }
    } catch (Exception exception) {

    }
}
```

図 5.12: Kide エディタ上での同期処理

処理を行う sync メソッドを示す。ここでは Kide エディタ上でエディタへの入力が行われた場合を図 5.12 に示す。入力があった方のエディタではプログラム断片の Range がずれるため、それを最初に修正する。その後、replace メソッドを用いて同期を行い、入力がなかった方のプログラム断片の Range の調整を行う。Kide エディタが閉じられた時は、⑤ から ⑦ のように Kide Observer に登録解除の通知を行う。

5.2.3　エディタ上での折りたたみ機能の実現

前述した通り、Kide では関心事を登録する際にメソッドボディの一部のみを登録することが可能である。メソッドボディのうち指定されなかった箇所は折りたたみ機能を用いてデフォルトでは表示させないことにしている。Kide エディタ上でこの機能を実現するには以下のような実装を行った。

- 拡張ポイント　org.eclipse.jdt.ui.foldingStructureProviders を追加
- 折りたたみ機能を統括する provider クラスを以下のように作成

```
public class MyFoldingStructureProvider implements IJavaFoldingStructureProvider {
    private ITextEditor editor;
    private ProjectionViewer projectionViewer;
    private static ArrayList<MyFoldingStructureProvider> instances
        = new ArrayList<MyFoldingStructureProvider>();

    public static ArrayList<MyFoldingStructureProvider> getInstances() {
        return instances;
    }

    public void install(.. editor, .. viewer) {
        if(editor instanceof 自分の作成した Editor クラス) {
            this.editor = editor;
            projectionViewer = viewer;
            instances.add(this);
        }
    }
}
```

- provider クラスを用いて、折りたたむ箇所は以下のようなコードで
  実現することができる。

```
.. instance = MyFoldingStructureProvider.getInstances().get(..);
instance.getProjectionViewer().getProjectionAnnotationModel().addAnnotation(
    new ProjectionAnnotation(),
    new Positoion(folding 開始 offset, folding する length));
```

第6章　Kide を用いたソフトウェア開発例

前章で Kide の機能について説明を行った。本章では、Kide を実際のソフトウェア開発に用いることを考え、簡単なチュートリアルを示す。以下では、Java バイトコードを操作するためのライブラリである Javassist の開発において実際に存在した課題を題材に、開発における個々の課題の報告、検討、修正という一連のサイクルにおいて Kide をどのようにに利用できるかを説明する。Javassist は redhat/JBoss のオープンソースソフトウェアとして 10 年以上にわたって開発され、多数の商用システムの開発にも用いられており、Java 言語を用いた全体で 3 万行弱のソフトウェアで、ソースコード自体は subversion で、各課題は JIRA で管理されている。

6.1　Javassist 開発時における実際に起こった問題の概要

利用者から報告されたメモリ消費量が大きすぎるという不具合 [2] を例に挙げ、それに対する対応をまず初めに示す。

6.1.1　消費メモリ量を軽減するための修正

開発チームはこの報告に対し検討を行った。その結果、この問題は単純なメモリリークによるものではないことを確認したものの、何らかの方法で消費メモリの量を減らす修正を行うべきであるという結論に達した。消費メモリ量を減らすための具体的な方策として、1 つのクラスのバイトコードの修正が終わった後は、他のクラスのバイトコードの修正に必要な情報だけを残し、それ以外の情報を全て破棄する修正が計画された。そのため、情報を保持する幾つかのクラスに、情報の破棄を実行する prune（枝狩り）メソッドを追加することになった。また、prune メソッドによってそぎ落とされた情報への編集を受け付けないために、クラス定義を変更する各メソッドの冒頭に変更対象が編集可能かどうかを調べる

JASSIST 28 に関する修正は以下の通りである。

- メモリ消費を軽減するため、クラスファイルから不必要な部分をそぎ落とす prune メソッドを追加した。
さらにその影響で、コードを操作するメソッドの冒頭に、その操作ができるかどうかのチェックを付加した。
関連する修正は以下の通りである。

```
public void prune(..) {
  /* 省略 */
}

public void addMethod(..) throws .. {
  checkModify(..);
  /* 省略 */
}

public void removeMethod(..) throws .. {
  checkModify(..);
  /* 省略 */
}
```

図 6.1: レビューシートの例

checkModify メソッドを実装した。これら 2 つのメソッドは特性上様々なクラスのメソッドに散らばってしまう。これを防ぐためには、Java で実装されている Javassist を AspectJ を用いて実装し直し、上記メソッドをひとつのアドバイスにまとめるしかない。

6.1.2　修正後のコードレビュー時の利用

大規模な組織で行う理想的な開発プロセスであれば、修正後のプログラムを Subversion にチェックインする前に、この修正についてのコードレビューを行うべきである。コードレビューを行う場合、例えば図**??**のようなコードレビュー文章を用意する、このコードレビューの結果、修正案が承認されれば修正されたプログラムをチェックインし、この修正を終えることになる。しかし、レビュー時にプログラムへの修正が提案された場合にはこのような文書では不都合が生じることがある。特に、大規模なプログラム、大規模な修正ほどその作業は煩雑になっていく。これを回避する方法は特になく、転記作業が必要になった場合のその安全性は、開発者が

いかに入念にそれを行うかにかかってしまう。

6.2　Kide を用いる例

上記の作業について Kide を用いることを考える。

まず、Kide が利用できれば、prune メソッドを追加したメソッド群を pruning という関心事に、checkModify メソッドを追加したメソッド群を checkModifying という関心事に関連付けることで、全てのメソッドを一覧でき、それらを対象とした修正をひとつのエディタ上で全て行うことができる。同じ修正を多くのメソッドに行う負担は変わらないが、編集箇所を探す必要がないので従来よりも安全に作業を行うことができると言える。また、Kide の折りたたみ機能を用いれば、関心事に該当する部分、この例では prune メソッド、checkModify メソッドの呼び出しのみを表示することも可能である。さらに、本課題への対応後もこれらの関心事を保存しておくことで、将来の保守時のプログラム修正の際に役立てることができる。関心事はデータベースで覚えており、コード自体にはタグ付けなどは行っていないため、最終的に利用されなかった関心事を長期間覚えていたとしてもそれ以外の作業を妨げることはない。

次に、コードレビュー文書の利用を考える、Kide を用いてコードレビュー文書を作成することで対話的なコードレビュー会議が行える。コードレビュー中にプログラムへの修正が必要になった時は、文書上の該当するプログラムを修正することで元のプログラムにも修正が行われる。新しいプログラムが動作するかをその場で確認でき、コードレビュー文書と元のプログラムの整合性も取れているため、修正後もそのコードレビュー文書を用いて会議を行える。従来のコピー&ペーストを用いる方法でコードレビュー文書を用意すると、上記した通り、コードレビュー中にプログラムへの修正が必要になったときに文書にそれを書き留めるかプログラムをその場で修正する必要がある。前者の方法では実際に修正後のプログラムを動かして正常に動作するか検討ができない上、会議の後にその修正の転記作業が必要になってしまう。後者の方法も、コードレビュー文書とプログラムの整合性が取れなくなってしまうので、その後の会議で文書を使用しづらくなってしまう。

これらの Kide の機能は全て GUI を用いて用いることができるため、新しい言語を導入する場合よりも開発者への負担が少なくなる。特に、大規模人数で開発に当たる場合に、この点が活きると考えられる。また、大規模システムではポイントカットのような言語による関心事の定義は難しい。なぜならば、開発者が関心事として捉えたい要素が全て定義することが難しくなり、さらに定義できたとしても、それを確認することも難し

い。AJDT のような支援を受けても、目視によって一つ一つ確認していかなければならない。Kide では GUI を用いて関心事の定義ができるので、定義漏れを起こしにくく、さらに Concerns View 上で定義した関心事を簡単に確認することができる。

以上のように、Kide は保守時に特に有用であると考える。バグの発見によって見つかる当初は捉えきれなかった関心事も Kide であれば容易に定義できるからである。逆に全ての関心事を開発初期に捉えられるのであれば、それらを適切にモジュール化する設計を行えるので、Kide の効果は薄くなってしまう。

第7章　評価

Kide を用いることで開発者は着目する関心事に関連するプログラム断片だけを見て編集作業を行うことができ、生産性を向上できると考える。本章では開発者がひとつの関心事への修正を行う際に、閲覧するプログラムをいかに Kide が減らしているか評価を行い、それに対する考察を述べる。

7.1　評価方法

本評価では、オープンソースとして公開されているソフトウェアを題材に用い、Kide が言語機構だけを用いるモジュール分割に比べどれほど閲覧性を向上できるかをひとつの修正作業に必要なプログラム閲覧行数を基準にして計測する。本評価では、リッチな言語機構を持つプログラミング言語の代表として AspectJ を比較対象言語に選び、その AspectJ で実装された Javassist と AjHotDraw を閲覧対象のソフトウェアとした。AjHotDraw は描画に関するソフトウェアで、その実装のモジュール化がしっかり行われていることで有名である。まず、それぞれのソフトウェアに対して機能に注目した分割と処理に注目した分割を行ったプログラムの二種類を用意した。前者の分割は AspectJ 固有の機能を用いずに済んだため、Java 言語による分割と言ってもよい。

次に、そのプログラムに対し、幾つかの関心事に着目した修正作業を行うことを想定し、(1) 従来エディタを用いて処理に注目した分割を行ったプログラムを修正した場合、(2) 従来エディタを用いて機能に注目した分割を行ったプログラムを修正した場合、(3) Kide を用いて機能に注目した分割を行ったブログラムをそれぞれ修正した場合に閲覧しなくてはならないメソッド、ポイントカット、アドバイスの行数の和を計測した。修正に対して適切なモジュール分割ができていた場合、修正箇所がまとまっているため閲覧しなくてならないコード行数が少なくなる。そうでない場合は多くのモジュールを辿る必要があるため多くの行数のコードを閲覧しなくてはならなくなる。

例えば、図 3.1 を機能に注目した分割を行ったプログラム、図 3.2 を処理に注目した分割を行ったプログラムとみなし、再描画処理の修正をした結

果は以下の通りになる。(1) の場合は、再描画処理は図 3.2 の Repainter アスペクト内に全てまとまって実装されている。再描画処理の修正には、この中の再描画処理の織り込み先を示す repaintedMethods ポイントカット（2 行）と再描画処理に関する after アドバイス（3 行）を対象に行えばよいので、この修正を行う上で閲覧する必要があるソースコードの行数は 5 行となる。次に、(2) の場合は、図 3.1 の各セッターメソッドに再描画処理が散らばって実装されている。再描画処理の修正を行うには、screen.repaint(); という行を含む Shape クラスの setPosition メソッド（4 行）、Circle クラスの setRadius メソッド（5 行）、Rectangle クラスの setWidth メソッド（5 行）と setHeight メソッド（5 行）の 4 つのメソッドを修正しなくてならない。以上より、このプログラムの場合は再描画処理を修正するために合計 19 行のコードを閲覧しなくてはならないことが分かる。最後に、(3) の Kide を用いる場合を考える。この場合は図 4.3 のような Kide エディタを用いることができ、折りたたみ機能を用いて再描画処理に関係ない行は表示していないため、4 つのセッターメソッドがそれぞれ 3 行で表示されている。よって、再描画処理に関する修正をするために閲覧する必要があるコードは計 12 行になる。2 つのソフトウェアから 15 の関心事を選び、上記のルールに従い評価を行った。

7.2　評価結果と考察

Javassist に関する評価結果を表 7.1 に、AjHotDraw に関する評価結果を表 7.2 に示した。まず、Javassist の処理に関する関心事に着目した場合を考える。従来エディタを用いて処理に着目した分割を行ったプログラムを閲覧した場合は合計 53 行であるのに対し、機能に着目した分割を行ったプログラムを閲覧した場合は合計 340 行と約 6.4 倍の行数のコードを閲覧する必要があることが分かる。それに対し、Kide を用いて処理に関する関心事を修正する場合は、合計 97 行と 1.8 倍程度のコードを閲覧するだけで済んでいる。この 1.8 倍の差は、処理に着目した分割を行ったプログラムでは着目する処理がアドバイス本体にのみに記述されるのに対し、Kide を用いた場合はその処理が各メソッドに記述され、それらを全て表示することによって生まれている。次に、Javassist の機能に関する関心事に着目することを考える。従来エディタを用いて機能に着目した分割を行ったプログラムを閲覧した場合と Kide を用いた場合は合計 61 行のコードを閲覧する必要がある。それに対し、従来エディタを用いて処理に着目した分割を行ったプログラムを閲覧した場合は 159 行と約 2.6 倍のコードを見なくてはならないことが分かる。これらの結果から、Kide を用いる場合は動的にモジュール分割を切り替えられるので、どの

表 7.1: 評価結果: Javassist

修正対象/分割方法	処理に着目	機能に着目	Kide
CtField クラス内 ModifyCheck	9	21	15
CtMethod クラス内 ModifyCheck	6	28	6
CtClassType クラス内 ModifyCheck	16	106	36
CtBehavior クラス内 ModifyCheck	15	155	30
classNameChangedCache の追加	8	30	11
(処理に関する修正合計)	53	340	97
(処理に関する修正比率)	(100%)	641.5%	183.0%
CtClass.setName(..)	75	27	27
CtClass.addField(.., ..)	43	7	7
ClassPool.makeClass(.., ..)	41	27	27
(機能に関する修正合計)	159	61	61
(機能に関する修正比率)	260.7%	(100%)	100%
全体合計	212	340	158
全体比率	134.2%	215.2%	100.0%

ような関心事に着目する場合でも比較的少ないプログラム閲覧量で修正を行えたと言える。それに対し、言語機構を用いた静的なモジュール分割では、その分割に適している関心事に着目する場合は少ない行数を閲覧するだけで修正を行えるが、そうでない場合は関心事に関連しないプログラムも閲覧しなくてはならず閲覧しなくてはならないコード行数が多くなってしまったと言える。

次に、AjHotDraw の関心事に関する評価結果について考える。Javassist を対象としたときとの違いとして、処理に関する修正を行う際に、従来エディタで処理に着目した分割を行ったプログラムを閲覧した場合と Kide を用いた場合に差がほとんどないことが挙げられる。上記した通り、通常は前者の方が短くなることが多いが、この例ではポイントカットの記述が複数行にわたっているのに対し、そのポイントカットに該当し処理が織り込まれる箇所が一つしかないことがあり、このような結果となった。しかし、全体をみると Javassist の場合と同様に Kide はどのような関心事に着目する場合も比較的少ないブラウザ量で済んでいると言える。

また、全体を通して見ると Kide を用いることで閲覧するコードの行数を減らせる割合は Javassist の方が AjHotDraw よりも高いことが分かる。これは AspectJ 版 Javasisst が Java 言語版 Javasisst の横断的関心事をアスペクトに抜き出して作成されたのに対し、AjHotDraw は設計段階から各関心事を捉え、より最適なモジュール分割ができていたことが要因だと考えられる。Kide による貢献が大きくなるのは Javassist の例のよう

表 7.2: 評価結果: AjHotDraw

修正対象/分割方法	処理に着目	機能に着目	Kide
PasteCommandUndo	14	33	20
CommandObserver	5	5	5
CommandExecuteInitUndo	7	9	3
CommandExecuteCheck	9	5	5
(処理に関する修正合計)	35	52	33
(処理に関する修正比率)	(100%)	148.6%	94.3%
PastCommand.execute()	42	27	27
CTXCommandMenu.addMenuItem(..)	16	6	6
CutCommand.execute()	23	15	15
(機能に関する修正合計)	81	48	48
(機能に関する修正比率)	168.8%	100%	(100%)
全体合計	116	100	81
全体比率	143.2%	123.5%	100.0%

表 7.3: 評価結果: 折りたたみ機能の有無による Kide の効果の違い

修正対象/Kide の種類	折りたたみ機能無	折りたたみ機能有
Javassisst/処理に関する修正合計	340	158
Javassist/処理に関する修正比率	215.2%	100%
AjHotDraw/機能に関する修正合計	100	81
AjHotDraw/機能に関する修正比率	123.5%	100%
全体合計	440	239
全体比率	184.1%	100.0%

に横断してしまう処理をあらかじめ捉えない場合や、ひとつのモジュールに多くの処理が実装されているような大規模プログラムの保守時だと言える。逆に、各関心事を事前に捉えられるときや、各モジュールに少数の処理のみが実装されている場合は従来エディタを用いてもそれほどブラウザビリティは低下しないと考えられる。

表 7.1、7.2 をグラフにまとめたものが図 7.1 である。言語機構のみを用いた場合は関心事の種類によって必要なコード行数の増減が大きく、Kide はどのような関心事に着目した場合でも少ない閲覧量で済んでいることが視覚的にも明らかである。この実験では Kide と言語機構のみを用いた場合では約 1.4 倍から 2 倍程度の差しかない。しかしながら、ここで例に挙げた Javassist と AjHotdraw は比較的モジュール化が上手く出来ている例である。他のプログラムを題材にした場合はより、Kide の効果がで

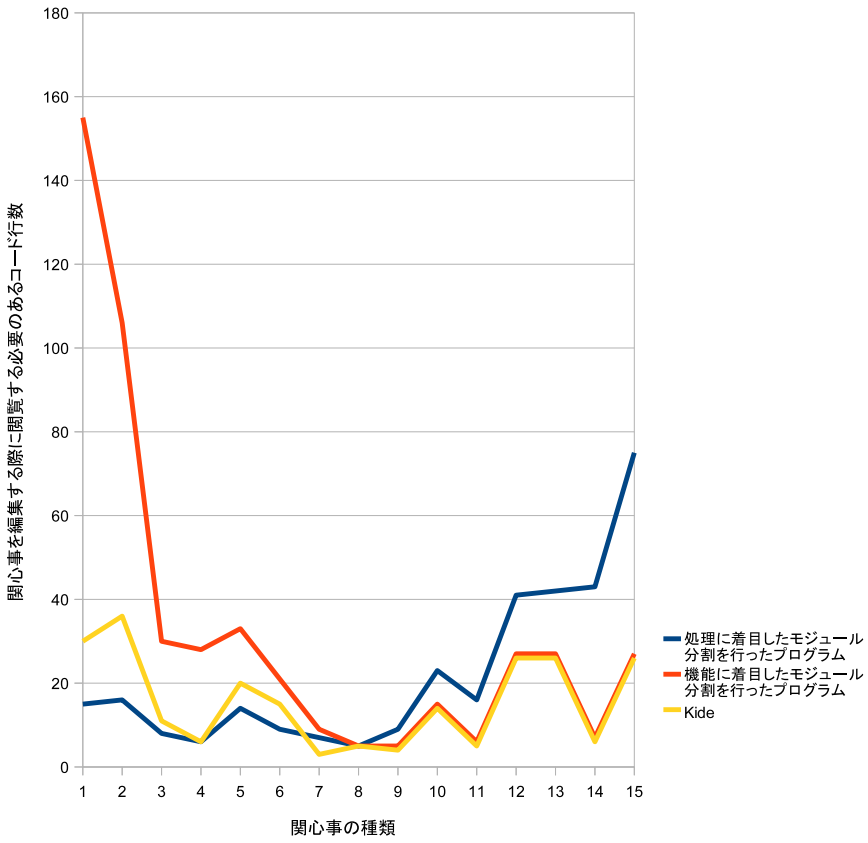

図 7.1: 評価結果

る結果を得られるのではないかと考えている。また、今回の実験では言語機構を用いてモジュール化したプログラムが事前に捉えていたであろう関心事を題材に評価を行った。そのため、ほとんどの関心事においてどちらかの言語機構による分割が Kide と同等以上の結果を得られている。しかし、実際には設計段階では予想もしなった関心事を対象にした修正を行わなくてはならないこともある。この場合は、いずれのプログラムも修正に多くのコードを閲覧しなくてはならない。それに対し、Kide は関心事を発見した段階で新たに定義を行い、Kide の機能を利用することができるため少ないブラウズ量で修正を完了することができると考えられる。

最後に、Kide の折りたたみ機能の効果について考察する。折りたたみ

機能をつけた Kide と つけない Kide の評価結果を表 7.3 に示した。まず、Kide に折りたたみ機能を実装したことで、 Javassist では 46。5%、AjHotDraw では 81。0%のコードを閲覧するだけで済むことが分かった。Javassist の方がひとつのモジュールに多くの種類の処理が実装されていたため、Kide による閲覧コード量の削減率が大きくなったと考えられる。また、従来エディタを用いて、機能に関する修正を処理に着目した分割を行ったプログラムに対して行った場合、あるいは処理に関する修正を機能に着目した分割を行ったプログラムに対して行った場合には 2 つ以上のファイルをまたがった編集が必要であった。それに対し、Kide では常に一つのエディタで編集を行えるため、この観点から見ても Kide の閲覧性は高いと言える。

第8章　まとめと今後の課題

8.1　まとめ

対話的なモジュール分割を支援する開発環境 Kide を提案し、Eclipse プラグインとして実装した。Kide は開発者の着目する関心事合わせて動的にプログラム分割を変更でき、開発者はその関心事を編集する上で最適な視点から編集を行うことができる。また、ソースコードを含む文書の作成も支援する。文書上のソースコードへの修正が元のプログラムへ伝わるので、修正の転記作業が不要になる。さらに、Kide を用いた場合と言語機構のみを用いた場合のモジュール分割について閲覧性の評価を行い、Kide の有用性を示すことができた。

8.2　今後の課題

関心事として登録したメソッドがなくなった場合への対応

現在の Kide では、関心事に登録したメソッドが消されてしまった場合の対応をしていない。実用上、このままでは問題があるので、Java Element の追加や削除をイベントとして取得し、その都度 Kide DB 内を更新していく必要がある。

人に利用してもらう形の評価

本論文では閲覧する必要のあるプログラム行数を以下に削減できたかという評価を行った。次は、実際に開発者に Kide を用いてもらう形の評価を行いたいと考えている。実際には、どのようなテストケースを用いるか、サンプル数はどれほど必要なのか、など様々な点を考慮する必要がある。

参考文献

[1] : AspectJ development tools(AJDT), http://www.eclipse.org/ajdt.

[2] : JBoss Community - JBoss Issue Tracker - Javassist, Javassist enhancement failed on deserializing hibernate proxies, https://issues.jboss.org/browse/JASSIST-28.

[3] Bragdon, A., Reiss, S. P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. and LaViola, Jr., J. J.: Code bubbles: rethinking the user interface paradigm of integrated development environments, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, New York, NY, USA, ACM, pp. 455–464 (2010).

[4] Bragdon, A., Reiss, S. P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. and LaViola, Jr., J. J.: A research demonstration of code bubbles, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, New York, NY, USA, ACM, pp. 293–296 (2010).

[5] Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. and LaViola, Jr., J. J.: Code bubbles: a working set-based interface for code understanding and maintenance, *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, New York, NY, USA, ACM, pp. 2503–2512 (2010).

[6] Christian, K. and Maximilian, S.: *Pcdiff: Attacking the Fragile Pointcut Problem*.

[7] Coppit, D., Painter, R. R. and Revelle, M.: Spotlight: A Prototype Tool for Software Plans, *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, Washington, DC, USA, IEEE Computer Society, pp. 754–757 (2007).

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

[9] Kästner, C., Apel, S. and Kuhlemann, M.: Granularity in software product lines, *Proceedings of the 30th international conference on Software engineering*, ICSE '08, New York, NY, USA, ACM, pp. 311–320 (2008).

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

[11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and G., W. G.: An overview of AspectJ, ECOOP '01, pp. 327–353 (2001).

[12] Ossher, H. and Tarr, P.: Hyper/J: multi-dimensional separation of concerns for Java, *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, New York, NY, USA, ACM, pp. 734–737 (2000).

[13] Shigeru, C., Michihiro, H., Kei, K., Fuminobu, T. and Yuuki, T.: Do we really need extending syntax for advanced modularity?, AOSD '12 (2012). To appear.

[14] Tarr, P., Ossher, H., Harrison, W. and Sutton, Jr., S. M.: N degrees of separation: multi-dimensional separation of concerns, *Proceedings of the 21st international conference on Software engineering*, ICSE '99, New York, NY, USA, ACM, pp. 107–119 (1999).

[15] Ye, L. and De Volder, K.: Tool support for understanding and diagnosing pointcut expressions, *Proceedings of the 7th international conference on Aspect-oriented software development*, AOSD '08, New York, NY, USA, ACM, pp. 144–155 (2008).

[16] 金澤圭, 堀江倫大, 千葉滋: Kide: 開発環境におけ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論文集, PPL '11, pp. 231–244 (2011).

[17] 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).