日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

Kide: 流動的なモジュラリゼーションのための IDE サポート

金澤 圭　堀江 倫大　千葉 滋

本研究では，流動的なモジュラリゼーションのための統合開発環境 (IDE) サポートを行う機構 Kide を提案し，Eclipse プラグインとして実装を行った．Kide を用いると，実際は複数のファイルをまたがって実装された関心事に関するコードを，ひとつのファイルにまとまっているかのように閲覧・編集することができる．複数のファイルに横断的に実装された関心事を修正するには，従来，ファイルの間を行き来する煩雑な編集を行わなくてはならず，新たなバグを生む遠因となっていた．そのような関心事を単体のファイルに分離する技術が幾つかあるが，関心事が横断的か否かは，どの視点からファイルに分けるかによって相対的に決まるので，全ての関心事を単一のファイルに実装することは難しい．我々は，上で挙げた問題の原因が，プログラムを編集する視点がファイル単位で固定的なことだと考えた．Kide を用いると，開発者の注目する関心事に合わせて，編集視点を流動的に変更する事ができる．

1　はじめに

プログラムは保守性や拡張性が高く，変化に強いことが重要である．このような性能を高めるために，関心事と呼ばれるひとまとまりの処理を異なるファイルに分離する．ひとつの関心事をひとつのファイル内で実装しておけば，仕様の変更などでその処理を変更しなくてはならない際に，その処理に関係のあるファイルを修正するだけで，システム全体の機能を変更できる．

しかし，関心事の中には一つのファイル内で実装することが難しく，複数のファイルをまたがって実装せざるをえない関心事が存在する．このような関心事に関連する編集を行う場合，複数のファイルをまたがる煩雑な作業が必要になるので，このプログラムは変化に弱いものになってしまう．

本研究では，関心事に合わせて編集視点を切り換えられる統合開発環境 (IDE) サポート Kide を提案する．ファイル間をまたがって実装された関心事を，従来のファイル単位で編集するのではなく，Kide の提供する機構を用いて関心事単位で編集することで，前述した問題を避けることができる．

以下，2 章では，一つのファイルに分離できない関心事の例を挙げ，それに伴う具体的な問題点について述べる．3 章では，この問題を解決するために本研究で提案する IDE サポート Kide について述べる．4 章では新たな問題例を挙げながら，関連研究について取り上げ，それらと Kide との比較を行う．5 章でまとめと今度の課題について延べ，本論文をまとめる．

2　ファイルの分割に合わない関心事

一般に，プログラムは関心事ごとに異なるソースファイルに分割してして実装される．このことで，プログラムの拡張性や保守性を上げることができるからである．しかしながら，一つのファイルに分離して実装できない関心事が存在し，その関心事に関連するコードを編集する際は，プログラマは複数のファイルを行き来しなくてはならない．

この様な関心事の例として，Eclipse プラグインのエディタに関連する実装を挙げる．図 1 は，エディタに関連する機能を拡張するためのフレームワークである．このフレームワークを用いることで，エディタ

Kide: IDE support for fluid modularization.

Kei Kanazawa, 東京工業大学大学院 数理・計算科学専攻, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology.

```
// SampleEditor.java
public class SampleEditor ... {
  public void init() {
    // TODO (Folding, Outline)
  }
  public void createPartControl(...) {
    // TODO (Folding)
  }
  public Object getAdapter(...) {
    // TODO (Outline)
  }
}
-------------------------------------
// SampleOutline.java
public class SampleOutline ... {
  public void createControl(...) {
    // TODO (Outline)
  }
}
-------------------------------------
// SampleFolding.java
public class SampleFolding ... {
  public SampleFolding(...) {
    // TODO (Folding)
  }
}
```

図 1　エディタ用フレームワークの一部

に独自のアウトラインビューアと折りたたみ (フォールディング) 機能を追加することができるとする．例えば，既存のエディタに独自のアウトラインビューアを組み込みたい場合，SampleEditor クラスの init メソッドと getAdapter メソッド，SampleOutline クラスの createControl メソッドに独自の機能を記述することになる．

このときに問題となるのが，アウトラインビューアに関連するコードが複数のファイルを横断して実装されている，ということである．先に述べたように，このような関心事を編集するには，複数のファイルをまたがる煩雑な作業が必要になる．図 1 のような小規模なフレームワークならば，プログラマが編集するべき部分を全て把握できるので問題ないが，規模が大きくなるとプログラマの負担が増え，新しいバグを生む遠因となりかねない．

また，このような関心事を新たなファイルに分離し，まとめる技術があるが，必ずしもうまくいかない．例えば，図 1 の SampleEditor クラスの init メソッドのように，複数の関心事に関連するコードがあった場合，適切なファイル分割というものはプログラマの関心に合わせて推移するため，一意には決まらない．この件に関しては 4 章の関連研究にて詳細を述べる．

図 2　図 1 に Kide を適応したときの Kide エディタ

3　流動的なモジュラリゼーションのための IDE サポートの提案

前章で述べた問題を解消するために我々は Kide という IDE サポートを提案する．Kide を用いると，実際は複数のファイルをまたがって実装された関心事に関連するコードを，一つのファイルにまとまっているかのように閲覧・編集することができる．プログラマは編集したい関心事に合わせて，編集対象を流動的に切り替えることができるため，前章で述べたような，一つのコードが複数の関心事に関連している場合にも対応することができる．本章では，具体例を用いて Kide の機能について述べる．

3.1　Kide エディタ

Kide は指定した関心事に関連するコードを複数のファイルから集め，それを Kide エディタ上に表示する．例えば，図 1 の例でアウトラインを作成するという関心事に関連するコードのみを表示したときの Kide エディタが図 2 である．Kide エディタ上での編集は，元のファイルにも伝わるため，従来の複数のファイルを行き来し行う作業を，単一のエディタ上で済ませることができる．

また，現在は予約語やコメントの文字色を変える，というサポートしか実装が済んでいないが，コーディング時の入力補完機能，エラーやワーニングの表示などのサポートも今後提供する予定である．

3.2 関心事を指定する方法

Kide エディタに集める関心事を指定する方法として，大きく分けて 2 つの方法を用意した．一つ目は，アノテーションを用いて，各コードがどの関心事に関連するかを明示的に示す方法で，この方法は現在実装段階である．もう一方は，条件を指定し，Kide がそれに合うコードをフィールド，メソッド単位の粒度で探索し，該当した断片を Kide エディタに集める方法である．

3.2.1 アノテーションを用いる方法

`@Concern` アノテーションを用いて，各コードがどの関心事に該当するかを明示的に指定できる．図 1 の例では，図 3 のようにアノテーションを用いて指定する．アウトラインビューアの作成に関連するコードには，引数に関心事の名前を表す文字列`"Outline"`を渡して，

```
@Concern("Outline")
```

を，フォールディングに関連するコードには，

```
@Concern("Folding")
```

をフィールドやメソッドの冒頭に付加する．また，いずれにも関連するコードには，

```
@Concern("Outline", "Folding")
```

のように，複数の引数を渡すこともできる．そして，`@Concern("Outline")` によって関連付けられたアウトラインに関連するコードを Kide が集め，図 2 の Kide エディタを作成できる．

3.2.2 コード上の規則性を用いた方法

Kide は，フィールドやメソッドの規則性から関連するコードを集め，Kide エディタを作成することもできる．ここでは，描画エディタの実装の一部である図 4 を用いて説明する．Shape クラスは全ての図形の親クラスで，図形が持つべき最小限の機能を実装している．そして，Shape クラスを継承した Square クラス，Circle クラスが具体的な図形を実装している．いずれのクラスも，それぞれ別のファイルに実装されている．各図形クラスは，形やスクリーン上の位置を表すフィールドを持ち，それぞれの値を変えるセッターメソッドを通して図形の形を変えたり，図形を動かしたりする．スクリーン上の表示内容の変更を伴うメソッドが呼ばれた後には，必ず画面に再描画を依頼する repaint メソッドを呼び，変更を反映させる．

```
// SampleEditor.java
public class SampleEditor ... {
  @Concern("Folding", "Outline")
  public void init() {
    // TODO (Folding, Outline)
  }
  @Concern("Folding")
  public void createPartControl(...) {
    // TODO (Folding)
  }
  @Concern("Outline")
  public Object getAdapter(...) {
    // TODO (Outline)
  }
}
-------------------------------------
// SampleOutline.java
public class SampleOutline ... {
  @Concern("Outline")
  public void createControl(...) {
    // TODO (Outline)
  }
}
-------------------------------------
// SampleFolding.java
public class SampleFolding ... {
  @Concern("Folding")
  public SampleFolding(...) {
    // TODO (Folding)
  }
}
```

図 3 アノテーションを用いて，関心事に関連するコードを定義する

この例で問題であるのは，再描画を行う処理がそれぞれの図形を表すファイルに散らばって実装されてしまっている点である．「再描画処理を行っている部分」を対象に編集を行いたい場合，前述したフレームワークの例と同様の問題が生じる．メソッド名を書き換える，といった単純な例であれば，開発環境のリファク

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

    public void setPos(int x, int y){
      xPoint = x;
      yPoint = y;
      screen.repaint();
    }
  }
-------------------------------------
// Square.java
  public class Square extends Shape {
    private double side;

    public void setSide(double s){
      side = s;
      screen.repaint();
    }
  }
-------------------------------------
// Circle.java
  public class Circle extends Shape {
    private double radius;

    public void setRadius(double r){
      radius = r;
      screen.repaint();
    }
  }
```

図 4　描画エディタの実装の一部

タリングツールを用いれば簡単に行うことができる．しかし，引数を追加したり，再描画処理の前後に新たな処理を加えなくてはならない場合，それだけでは解決しない．

このような問題を解決するにも Kide が役に立つ．Kide は以下の条件にあったコードを Kide エディタ上に集めることができる．

- メソッドの呼び出し関係
  メソッドを指定し，そのメソッドを呼び出しているメソッドを関心事として集める．Eclipse の Call Hierarchy ビューアに相当し，既存のソースコードの閲覧，編集の際の利用を想定している．
- メソッドのオーバーライド関係
  メソッドを指定し，そのメソッド自身とそのメソッドがオーバーライドしているメソッドを関心事として集める．既存のメソッドをオーバーライドする際の利用を想定している．

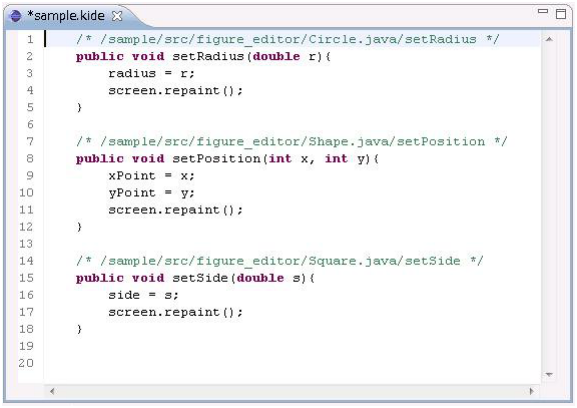

図 5　図 4 に Kide を適応したときの Kide エディタ

```
// Repainter.aj
aspect Repainter {
  after():execution(void set*(..)){
    screen.repaint();
  }
}
```

図 6　描画エディタの再描画処理をアスペクトにまとめた例

図 4 における問題を解決するには，前者の方法で repaint メソッドを指定し，コードを集めればよい．作成した Kide エディタは図 5 のようになる．

4　関連研究

ファイルによる分割に合わない関心事をひとつのファイルにまとめる技術としてアスペクト指向 (AOP) [5] [6] がある．AspectJ [1] などのアスペクト指向言語を用いると，ファイル間をまたがるコードをあらかじめアスペクトというファイルに分離することができる．例えば，描画エディタの再描画処理は図 6 のようなアスペクトに分離することができる．アスペクトに記述されたコードは，プログラム実行時の指定されたときに織り込まれるため，元のプログラムの挙動を保ちつつモジュール化できる．この例では，返り値の型が void で，かつ，set から始まるメソッドの実行後に repaint メソッドが呼ばれる．元々複数のファイル

を横断していたコードを 1 つのアスペクトにまとめることがでるので，ファイル間をまたがる編集を回避できる．しかし，元々クラスに記述されたコードの一部をアスペクトに分離するので，プログラムの振る舞いが分かりにくくなってしまう．アスペクト指向でもプログラムの見方は固定的で，関心事によっては編集しにくいことがあるという点は変わらない．Kide を用いれば，その時々のユーザーの関心事に合わせて視点を変えて閲覧や編集を行える．

AspectJ のための IDE サポートである AJDT [2] を用いることで，織り込まれるアドバイスをソースコードエディタ上に表示することができる．AspectJ と ADJT を組み合わせて用いることで，上で述べた AOP の問題を軽減することができる．しかしながら，ひとつのコードが複数の関心事に含まれるような例では AJDT を用いたとしてもうまくいかない．そのような関心事はいずれか一方の関心事を実装するファイルに含めなくてはならず，もう片方の関心事を実装するファイルには含めることができない．後者の関心事に関連する編集を行いたい場合は，複数のファイルをまたがるような編集が必要になってしまう．Kide を用いる場合，プログラマの関心事に合わせてモジュールを作成することができ，元々どのファイルにコードが実装されているかは関係ない．そのため，この様なひとつのコードが複数の関心事に関連する場合でも対応することができる．

Fluid AOP [3] は，AOP の利点を IDE によるサポートによって生み出す研究で，本提案手法は Fluid AOP に基づく．Fluid AOP は，ポイントカットを書くことにより，3 種類の編集可能なビューアを用いることができる．一つ目は Before/After/ITD JPM で，ポイントカットで該当した部分にビューア上での書き込みを反映させることができる．二つ目は Gather JPM で，ポイントカットに該当した部分を一ヶ所のビューアに表示，編集することができる．三つ目は，Overlay JPM で，ポイントカットに該当した全てのコードの中から，共通部文のみを表示し，そうでない部分は見えなくするようなビューアを作成することができる．これらの機能を用いることで本論文で例に挙げた問題を解決することができる．しかし，Fluid AOP にもいくつか問題がある．まず第一に，集めるコードの指定方法への支援がないことが挙げられる．上述した通り，Fluid AOP では，プログラマの集めたいコードをポイントカットを書くことでしか指定できず，Fluid AOP を大規模システムに対して用いたとき，本当にプログラマの関心のあるコードが全て集められているかを判断することが難しい．それに対し，Kide では関心事の指定方法の支援を充実させるべく，対話的に指定を行えるような機構を提供している．現在は GUI を用いているが，よりプログラマが容易に，直感的に指定ができるような機構にしていくつもりである．さらに，Fluid AOP では，コーディングへの支援が乏しく，プレインテキストを編集することになる．Kide では入力補完などのコーディング支援にも重点を置き，プログラマが既存の Java ソースコードエディタを用いるのと同等の感覚でコーディングを行えるようにしたいと考えている．

Mylar [4] を用いると，Java や AspectJ で書かれたプログラムをタスクごとにモジュール化できる．タスクとは，コーディングやテストなど，ユーザーによる一連の作業のことを指す．Mylar は，そのときに行っているタスクに関連するプログラム要素のみを Mylar Outline や Mylar Packege Explorer などの各ビューアに表示させる．これらのビューアは，縦スクロールなしに閲覧可能な数の要素のみを表示する．Mylar は，タスクに関連するプログラム要素を把握するために degree-of-interest(DOI) という数値基準を各プログラム要素に与える．Mylar はユーザーの作業を監視し，それに応じて DOI を変動させる．ユーザーにより選択・編集されたプログラム要素は，現在のタスクに関連している可能性が高いと考え，対応する DOI を上げる．逆にしばらく選択されなかった要素の DOI は減少させる．しかし，Mylar は DOI の高い要素の一覧をビューアに示すだけで，DOI の高い要素をひとつのエディタ上で編集したり，エディタ上で DOI の高いメソッドなどにフォーカスを当てるようなエディタ上での特別な機能はない．そのため，複数のファイルをまたがった作業を行う場合は，そのファイルの数だけエディタを開かなくてはならない．

5 まとめと今後の課題

5.1 まとめ

流動的なモジュラリゼーションのための IDE サポートを行う機構 Kide を提案し，Eclipse プラグインとして実装を行った．Kide を用いることで，従来の編集単位であったファイル以外に，関心事という単位で編集を行うことができる．これにより，プログラマが編集しやすい角度を選択しプログラムを編集することができる．

5.2 今後の課題

- 関心事の指定方法をより充実させる
  現在，関心事の指定方法はアノテーションを方法と 2 種類のコードの規則性を用いた方法のみである．この指定方法をより充実させ，プログラマが捉えたい関心事を正確に指定できるようにしたいと考えている．例えば，メソッドを指定する場合に，複数のメソッドを指定，あるいは「A というメソッドを呼び出した後に，B を呼んでいるメソッド」という指定方法を考えている．また，アノテーションを用いる方法も，アウトラインなどを用いることで，どのコードの断片がどの関心事に含まれているかが視覚的にわかりやすいような工夫をしたいと考えている．
- Kide エディタ上での支援を充実させる
  本論文中でも述べたように，Kide エディタ上でも通常のエディタと同様のコーディング支援を行いたいと考えている．また，Kide エディタの複数のファイルからコードを集めてくるという性質から，大規模システムで Kide を用いた場合，エディタが縦に伸びてしまうことが増えると考えられる．その場合でも，プログラマが編集を行いやすいように，例えば関心事として関連の薄い部分をフォールディングするなどのオプションを追加できればと考えている．
- Kide の評価
  実際にプログラマに Kide を使用してもらい評価を行いたいと考えている．実際に，Kide を用いることで生産効率を向上させることができるのかを検証したい．

参 考 文 献

[1] AspectJ. http://www.eclipse.org.aspectj/.
[2] AspectJ development tools(AJDT). http://www.eclipse.org/ajdt.
[3] T. Hon and G. Kiczales. Fluid AOP join point models. In OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pages 712-713, New York, NY, USA, 2006. ACM.
[4] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, pages 159-168, New York, NY, USA, 2005. ACM.
[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. G. riswold. An overview of AspectJ. In ECOOP '01 - Object-Oriented Programming: 15th European Conference, LNCS 2072, pages 327-353. Springer, 2001.
[6] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In ECOOP '03 - Object-Oriented Programming, pages 219-233. Springer-Verlag, 2003.