Kide: 開発環境によるオブジェクト指向言語での アスペクト指向開発の支援

金澤 圭　　堀江 倫大　　千葉 滋

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
www.csg.is.titech.ac.jp/~{kanazawa,horie,chiba}

概 要　オブジェクト指向言語などで発生する横断的関心事を解決するための技術としてアスペクト指向（AOP）がある。しかし、既存の AOP 言語では、ある関心事の視点から見て最適なプログラム分割を一度行ったら、他の視点に変更しづらいという問題がある。そこで本稿では、AOP 言語が提供するプログラムの分割機能のこの問題点を解決するため、対話的なプログラムの分割機能を統合開発環境 (IDE) 上で実現したシステム Kide を提案する。Kide は、ソースコードだけでなくドキュメントも、元のファイル分割とは異なる開発者の関心事に合わせた見せ方で表示する。また、特定の関心事に関連する部分に対して、同一のコードを挿入するなどといった共通の編集を容易に行うための支援も提供する。これらの機能は全て IDE の機能として提供しているため、新しい言語を習得せずとも用いることが出来る。

1　はじめに

今日、高品質のソフトウェアを短期間で作成することが求められる。その際、既存のソフトウェアに修正を加え新しい製品を作成する手法をとり、開発期間を短くすることがしばしばある。この手法は、既存のソフトウェアに綿密な設計が施されており、その実装が関心事ごとに分割されていた場合に特に有効である。その半面、関心事ごとに十分に分割管理されていないと、修正すべき箇所を手作業で探さなければならず、開発期間を短縮するのが困難になる。また、一般的に開発者が修正する箇所を探す時間は作業時間全体の 6 割から 9 割にのぼるといわれている [6]。本質である修正作業に開発者が集中するためには、修正箇所を手作業で探す時間を極力短くできるような保守性の高いプログラムが望まれる。

一般に、開発者はプログラミング言語が提供する機構を用いてプログラムを関心事ごとにモジュールとして分割し、それをファイル単位で管理することで保守性を高める。オブジェクト指向プログラミング (OOP) では、関心事ごとにクラスというモジュールにまとめて管理する。例えば、OOP が提供する継承機構を利用すると、共通の機能をまとめた親クラスと個別の機能を持つ子クラスとにファイル分割することができる。共通の機能に変更を加える際には親クラスを含むソースファイルのみを修正すればよいため、保守性が向上する。しかし、イベント通知処理や同期処理、ロギングといった関心事は、それ以外の関心事に合わせたプログラム分割としばしば相性が悪い。それ以外の関心事に合わせた分割では、イベント通知処理などの関心事に属するコードがあちこちのモジュールに散らばってしまう。そのような関心事を横断的関心事と呼ぶ。横断的関心事は、逆にそれに合わせてプログラムを分割すると、今度はそれ以外の関心事に属するコードがあちこちのモジュールに散らばってしまう。いずれの場合も、コードが複数のモジュール間にまたがって散在してしまい、ソースコードの保守性を低下させてしまう。

アスペクト指向プログラミング (AOP) [12] はこの横断的関心事に属するコードだけを抜き出してアスペクトと呼ばれる別のモジュールにまとめられるようにする。しかし AspectJ [11] のような

既存の AOP 言語には、ある関心事の視点からアスペクトを決め、関連するコードをそこへ集めてしまうと、他の視点からの修正が行いにくいという問題がある。しばしば、ひとつのコード断片は単一の関心事だけに属すのではなく、複数の関心事に属す。AOP でそのコード断片を特定の (横断的) 関心事のためのアスペクトの中に移すと、そのコード断片が属すそれ以外の関心事の視点から見た際、依然としてコードがあちこちのモジュールに散らばっていることになる。これでは十分なモジュール化が達成できたとは言い難い。

そこで、我々は統合開発環境（IDE）によって柔軟なプログラムの分割あるいはモジュール化を支援するためのツール Kide を提案する。Kide は、開発者が現在修正中、閲覧中の関心事に関わるコードをその都度集め、あたかも同一のソースファイルに含まれる同一モジュール内のコードであるかのように閲覧し、修正できるようにする。Kide を用いることで開発者は、開発者の関心事に合わせてその都度最適にモジュール化された形、見え方でプログラムを IDE 上で閲覧し、修正できる。さらに、Kide は、ソースコードだけでなくドキュメントも、元のファイル分割とは異なる開発者の関心事に合わせた見せ方で表示する。また、特定の関心事に関連する部分に対して、同一のコードを挿入するなどの共通の修正を容易に行うための支援も提供する。これらの機能は従来、AOP 言語のようなプログラミング言語の機能として提供されてきたものだが、Kide の場合は全て IDE の機能として提供している。このため、新しい言語を習得せずとも用いることができ、開発者の負担を軽減できる。

以下、2 章では既存のモジュール化手法の問題点を具体的な事例を用いて考察する。3 章では、本研究で提案する Kide について述べる。4 章では、実アプリケーションを用いたアスペクト指向開発の例を示し、Kide の有用性を検討する。5 章では IDE の機能でモジュール化を支援する先行研究と Kide との比較を行う。6 章でまとめと今後の課題について述べ、本稿をまとめる。

2　プログラミング言語によるモジュール化

一般的に、関心事をモジュール化するためにプログラミング言語の機構が利用される。特定のプログラミング言語でうまくモジュール化できない場合には、モジュール化のための機構が強化された別のプログラミング言語を使用する。しかし、新しいプログラミング言語を習得することは容易ではないため、開発者の負担になってしまう。例えば、OOP は広く利用されているプログラミング技法であるが、横断的関心事をうまくモジュール化することができない。AOP を用いることで横断的関心事をアスペクトという別のモジュールに分割することができるが、AOP 言語の習得は難しく、使いこなすのが困難である。

また、言語機構を利用し、ある視点に最適なモジュール化を行ったとしても、ファイル単位で静的に分割されてしまうため、それ以外の視点からの分割に変更することが難しくなる。その一方で、開発者の関心はその時々に応じて推移するため、見たい視点・見やすい視点は状況によって変わる。関心事の分割が静的に決まっているため、適切なプログラム分割とは異なる視点からの、モジュール間をまたがる作業を強いられてしまうことがある。

これらの問題に対応するために、現在では広く利用されるようになった統合開発環境（IDE）の機能を用い、ファイル分割に基づく本来のモジュール化とは異なるモジュール化を必要に応じて利用できるようにするのが理想である。また、今まで慣れ親しんできた OOP 言語を使い続けながら、AOP で実現できた横断的関心事のモジュール化を行えるようにし、プログラムだけでなく、ユーザーガイドや Javaodc [2] などのドキュメントも同じ関心事を扱う構成要素として一緒にモジュール化できるようにするのが望ましい。しかし、現在このような理想的な開発環境はない。

```
[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) {
          radius = r;
          screen.repaint();
      }
  }

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

      public void setWidth(double w) {
          width = w;
          screen.repaint();
      }

      public void setHeight(double h) {  /* sets hight */  }
  }
```

図 1. Java による図形エディタの実装

2.1 図形エディタの例

簡単な図形エディタの例を通して既存のプログラミング言語では十分なモジュール化ができないことを述べる。一般に図形エディタの編集対象である各図形を、図形の種類、という関心事でモジュール化することは容易である。図 1 にそのような関心事でプログラムを分割して得られるクラス定義の例を示す。円を表す Circle クラスには、半径を表すフィールド radius やそれを操作するための setRadius メソッドが定義されている。長方形を表す Rectangle クラスには、幅と高さを表すフィールド width、height やこれらを操作するための setWidth メソッド、setHeight メソッドが定義されている。図形の位置を表すフィールド xPoint、yPoint やこれらを操作する setPosition メソッドは円や長方形に共通の事柄であるため、親クラスである Shape クラスにまとめる。

しかし、図形エディタの実現に必要な機能という視点に立って見ると、図形の種類という関心事に沿ったモジュール分割は適切とは言えない。ユーザが図形をエディタ上で移動させたときには、画面の再描画処理が必要になるが、この処理に関連するコード断片は、異なるクラスのメソッドに散らばって実装される。例えば Circle クラスの setRadius メソッドや Rectangle クラスの setWidth メソッドは、screen オブジェクトの repaint メソッドを最後に呼んでいるが、これは再描画処理に関連するコードである。再描画処理を関心事ととらえると、図 1 のプログラムでは、それに関連するコードがモジュール化できていないといえる。これは典型的な横断的関心事であり、この関心事に属するコード断片はあちこちに散らばっている。再描画処理を行っている部分に着目して閲覧したいとき、その部分を修正したいときには、プログラムの各所から散らばった関連コードを探し出して閲覧、修正しなければならない。

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

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

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

図 2. AspectJ を用いた図形エディタの実装

2.2 AOP 言語の利用

再描画処理のような横断的関心事の問題に取り組んだのが AOP 言語である。図 2 は、代表的な AOP 言語のひとつである AspectJ [11] を用いて、図 1 中の再描画処理に関連するコードを Repainter アスペクトへと移した例である。各クラスの、戻り値が void で名前が “set” から始まるメソッドが実行された直後に Repainter アスペクトの after アドバイスが呼ばれ、Screen オブジェクトの repaint メソッドが呼び出される。再描画処理に関連するコードが各クラスのメソッドから、この Repainter アスペクトに集められ、元のクラスのメソッドからは該当するコードが削除される。さらに、集められたコードは同一なので、Repainter アスペクトの中では 1 個の短い after アドバイスにまとめられている。このため、再描画処理を関心事としてプログラムを修正するときは、Repainter アスペクトに注目して修正を行える。集められたコードも 1 個にまとめられているので、理解も容易である。

しかし、AOP を用いてプログラムを分割し、ある視点から横断的である関心事をアスペクトの中にうまく分離できても、その分割が他の関心事の視点からプログラムを閲覧、修正するときにも適しているとは限らない。コードの断片はしばしば複数の関心事に属す。例えば Circle クラスの setRadius メソッドの中に含まれる再描画に関連するコード断片は、図形の種類別の関心事である Circle に関連するともいえる。少なくとも Circle クラスの挙動の全体像を知るためには、そのコード断片を知ることが必要である。しかし図 2 のアスペクトを用いたプログラムの分割では、そのコード断片は Circle クラスの中にはなく、Repainter アスペクトの中に移されている。このため、Circle クラスを一見しても再描画関連の処理も行っていることはわからない。何らかの方法でプログラム全体を調べて Repainter アスペクトの存在に気づき、両者のソースファイルを並べて見ることで初めて理解できる。このようなプログラムの分割は、再描画処理という関心事からは適切だが、Circle という関心事からは関連するコード断片が 1 個のモジュール (すなわち Circle クラス) にまとめられているとは言い難く、適切とは言えない。任意の関心事の視点から見て適切なプログラムの分割はこのように一般には困難である [10]。

さらに、再描画処理のような横断的関心事をモジュールに分離するために上の例では AspectJ 言語を用いたが、アスペクトに関する構文は元となった Java 言語の構文とは大きく異なっており、プログラムをうまくモジュール化するためには開発者はそれに習熟しなければならず、実利用の障壁となる。AspectJ ではポイントカットと呼ばれる、Java のプログラマにとって特殊な文法を用いてアスペクトを定義する必要がある [5]。ポイントカットを定義するには、プログラム全体を細部まで

理解する必要がある上、将来的なプログラムの修正まで配慮しなくてはならない。将来、本来意図していなかったプログラム要素を選択してしまったり、意図していたプログラム要素を選択しなくなってしまうことがあるからである [7]。

3 Kide

前章で述べた問題の解決を支援するために、我々は開発者の関心の推移に応じて IDE によって対話的にプログラムのモジュール化を実行できるように支援するためのツール Kide (図 3) を提案する。Kide を使うと、開発者は編集中に現在取り組んでいる関心事に関連するコードを、対話的にメソッド単位で仮想的なソースファイルに集め、そのソースファイルを専用のエディタで閲覧、編集することができる。仮想的なソースファイル上の変更は、元のソースファイルにも同時に反映される。Kide は同時に複数の仮想的なソースファイルを扱うことができるので、開発者は様々な関心事に対応する仮想的なソースファイルを同時にいくつか定義し、それらを切り替えながら異なる側面からモジュール化されたプログラムとして編集することができる。

例えば図 1 のプログラムは、図形の種類、という関心事でソースファイルに分割、モジュール化されている。Kide を使うと、再描画処理、という関心事に関連する `setPosition` メソッド、`setRadius` メソッド、`setWidth` メソッド、`setHeight` メソッドをそれぞれのクラスから抜き出し、1 つの仮想的なソースファイルに集めることができる。定義した仮想的なソースファイルの一覧は図 3 (1) に示した Concern ビューアで確認することができる。図は再描画処理用に定義した仮想的なソースファイルを `Repainter` としたときの例である。ここで編集したい仮想的なソースファイルを Kide エディタで図 3 (2) のように開くと、集められたメソッドが上から順にならんでいる仮想的なソースファイルを見ることができ、通常のソースファイルと同様に編集可能である。

プログラミング言語を使ったモジュール分割と異なり、Kide の場合、元のソースファイルはそのままである。例えば `Shape.java` をエディタで開けば、`setPosition` メソッドは元の位置に含まれており、編集可能である。対話的に定義した視点、側面から、必要に応じてプログラムを仮想的に分割し、分割のやり方を切り替えながらプログラムの編集を行える点が Kide の特徴である。

3.1 関心事の定義

Kide では GUI を用いて対話的にプログラム要素を選択肢、関心事を定義する。プログラム要素の粒度は、フィールド、コンストラクタ、メソッドである。例えば、再描画処理についての関心事 *Repainter* に `setWidth` メソッドを加えるには、Java エディタ上で *setWidth* メソッドをマークし、右クリックからポップアップメニューを表示する。その中の「関心事に定義する」というメニューをクリックすると関心事名を入力するためのテキストボックスが出てくるので、ここで任意の関心事の名前を入力することで（ここでは *Repainter* とする）、その関心事に追加される。定義された関心事は、図 3 (1) の Concern ビューアで閲覧することができる。例えば、図 3 (1) の場合、関心事 *Logging* に `init` メソッドと `setHeight` メソッドが、関心事 *Repainter* に `setPosition` メソッド、`setRadius` メソッド、`setWidth` メソッド、`setHeight` メソッドが含まれていることがわかる。

大規模な開発では、上記の方法で関心事を定義するには労力がかかる。そこで Kide は、適当な規則に該当するプログラム要素を探し、見つかった要素を一括して関心事に加える機能を提供する。規則として「呼び出し関係」と、「メソッドのオーバーライド関係」の二種類を現在は提供している。前者は、特定のメソッドを呼び出しているメソッド群を関心事としたいときに用いる。例えば、`repaint` メソッドを呼び出しているメソッド群を関心事としたい場合、Java エディタ上で `repaint` メソッドをマークし、ポップアップメニューから「呼び出し関係」を選択する。すると、図 4 のようなウィザードに、該当するメソッド群が表示される。このウィザード上で、チェックを入れたプログラム要素の集まりを 1 つの関心事として定義できる。後者の「メソッドのオーバーライド関係」を

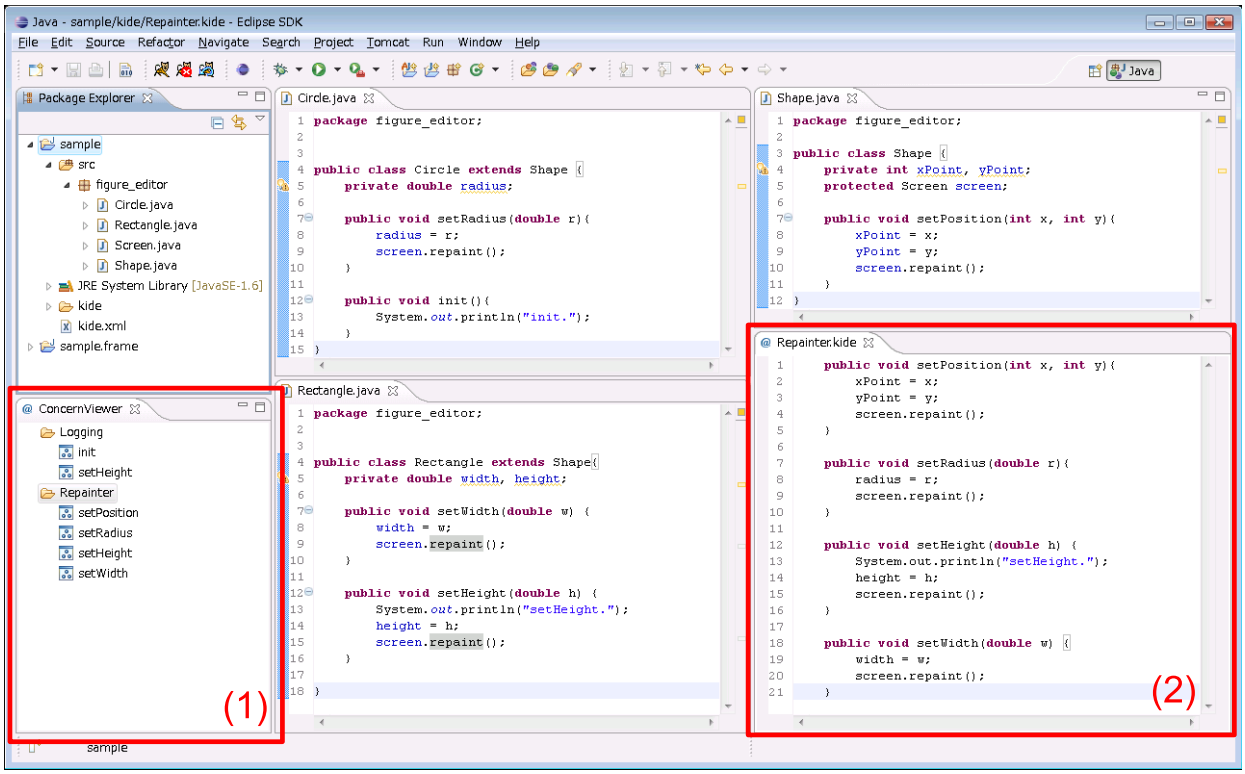


図 3. 開発中の図形エディタを Kide の機構を用いて表示した例 (Concern ビューアと Kide エディタ)

用いるときは、同様にメソッドを指定し、メニューから「オーバーライド関係」を選択する。すると、その指定したメソッドをオーバーライドしているメソッド群がウィザードに表示される。このウィザード上でチェックを入れたプログラム要素の集まりを 1 つの関心事とすることができる。現在の実装では、プログラムの修正によって、これらの呼び出し関係やオーバーライド関係に変更があった場合、その変更が自動的に関心事に含まれるプログラム要素に反映されることはない。

3.2 定義した関心事の使用方法

Concern ビューア上で関心事をクリックすると、その関心事に含まれるプログラム要素が図 3 (1) のようにエディタ上に並べて表示される。この通常のファイル分割とは異なる分割で編集が行えるエディタを Kide エディタと呼ぶ。Kide エディタ上でのプログラムの修正は元のプログラムに即座に伝わるため、これまで複数のソースファイルのエディタを切り替えながらおこなっていた編集作業を、1 つの Kide エディタ上だけで行えるようになる。

また、関心事に合わせたプログラム分割を可能にするだけでなく、同じ関心事に関連するコードに同一の修正を加える機能も Kide は提供する。その機能は以下の通りである。

- メソッドボディの冒頭に、予め開発者が記述したコードを一斉挿入する。
- return 文の直前、返り値がない場合はメソッドボディの最後に、予め開発者が記述したコードを一斉挿入する。

これらの機能は、Concern ビューア上で特定の関心事を右クリックで選択し、ポップアップメニューから「コードの一括挿入」を選択する。すると、テキストボックスが表示されるので、そこに一括挿入したいコードを記入し、挿入する。現在の実装では、この機能で一括挿入されたコードを後に一括して修正したり、一括して削除する機能は提供されない。

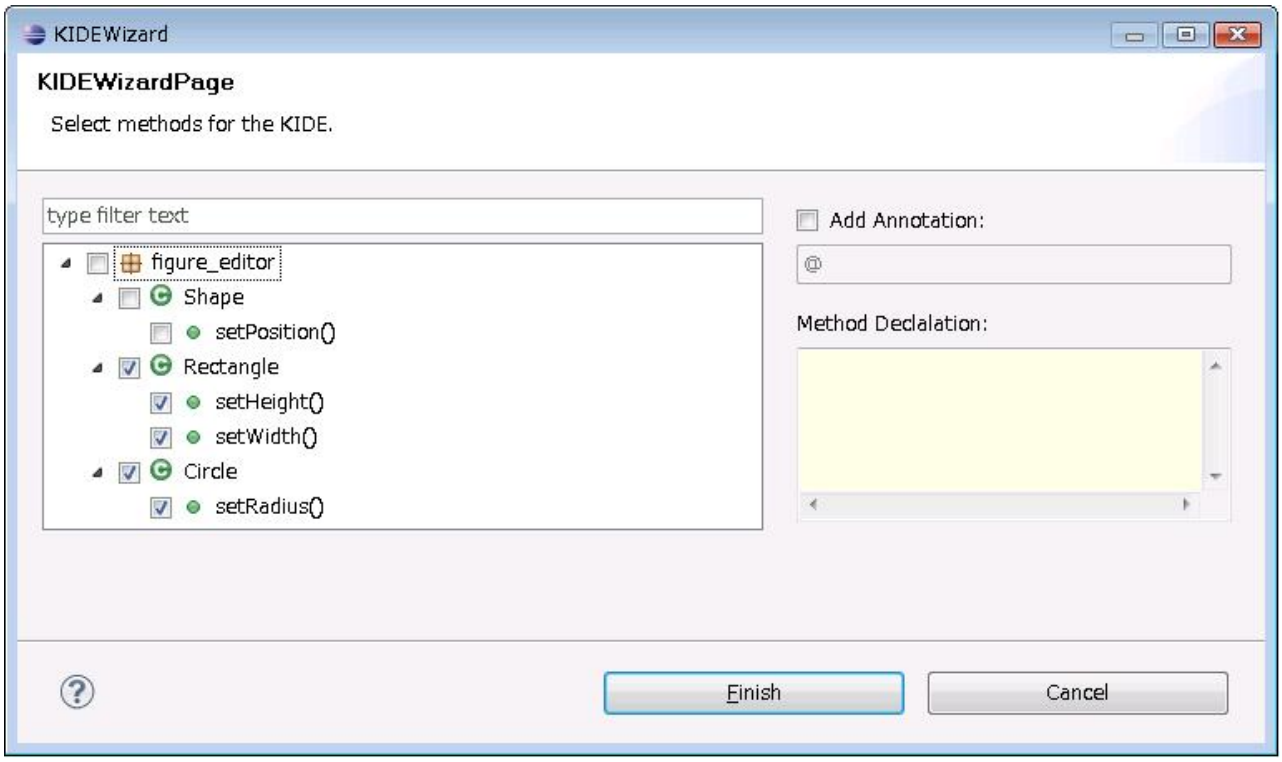


図 4. Kide の関心事定義を支援するウィザード

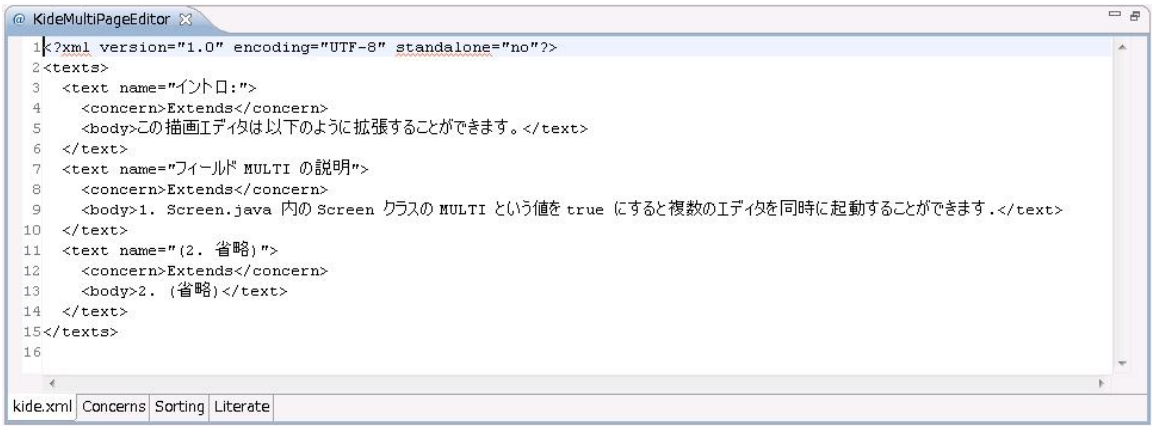
```
[Screen.java]
  public class Screen {
      private boolean MULTI = false;
      public void repaint() {  /* 省略 */  }
  }

[README.txt]
この描画エディタは以下のように拡張することができます。

1. Screen.java 内の Screen クラスの MULTI という値を
   true にすると複数のエディタを同時に起動することができます。

2.（省略）
```

図 5. 図形エディタのプログラムとドキュメント

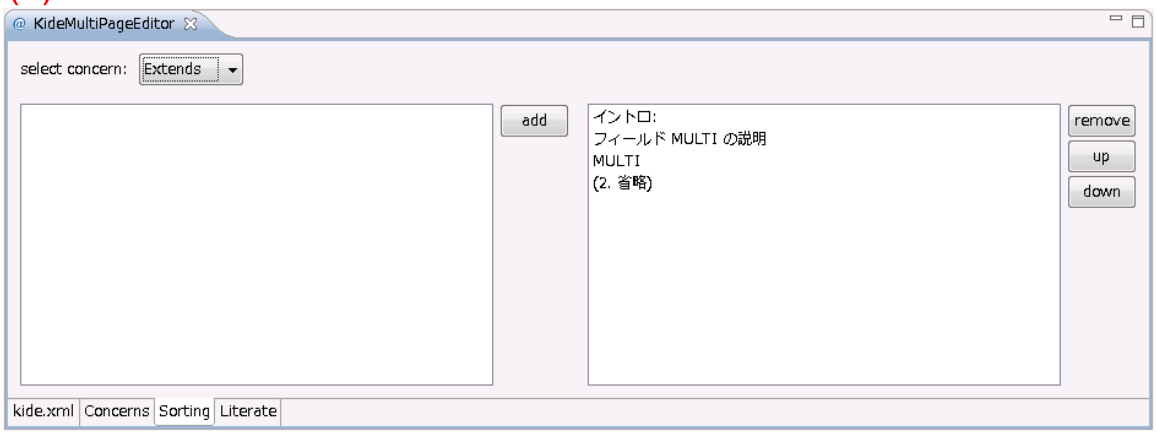
3.3 ドキュメントのモジュール化

Kide が扱えるソースファイルは、プログラムのソースファイルだけはない。ソフトウェアの付属文書のテキストファイルもプログラムのファイルと同様に扱える。したがって Kide ではプログラムと文書とをまたがる擬似的なモジュールを生成することもできる。例えば、図 5 のようなプログラムとチュートリアル文書があったとする。フレームワークのようなソフトウェアでは、このようなチュートリアル文書が欠かせないが、この文書 README.txt を読む開発者は文書の他にプログラムのソースファイルである Screen.java も交互に見ながら内容を理解しなければならない。我々は、

(1)



(2)



(3)

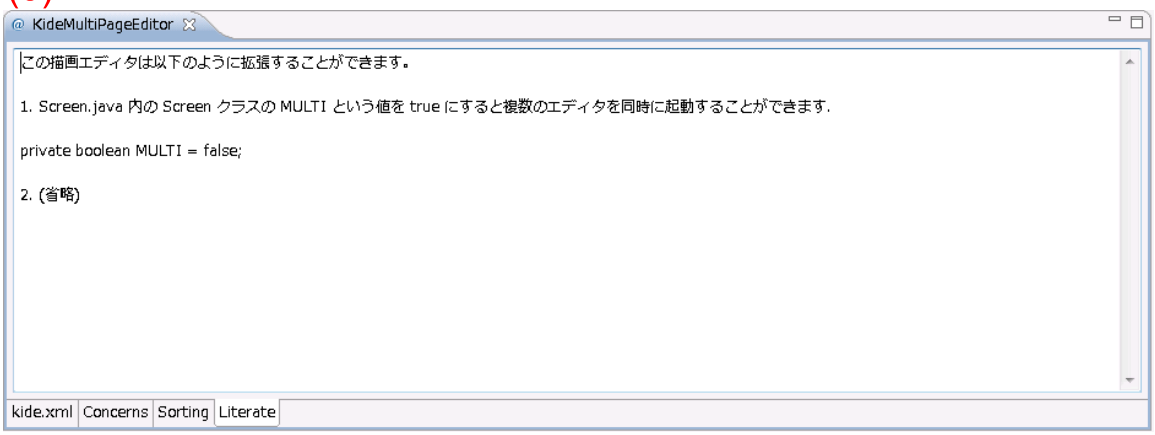


図 6. プログラムとその他ドキュメントのモジュール化

このような事例も横断的関心事の一種であると考える。例の場合、「複数のエディタを同時に起動できるように描画エディタを拡張する」という関心事に関連するコードやテキストが、README.txt と Screen.java の 2 つのソースファイルに分散してしまっている。

Kide を用いると、このような場合でも、関心事に関連するコードやテキストを集め、Kide エディタを使って閲覧、編集できる。Kide エディタで見る仮想的なソースファイルは、ちょうどプログラ

ムの一部を切り取ってチュートリアル文書の中にはめこんだようなものとなる。

図 6 に具体的な手順を示す。まず、チュートリアル文書は (1) のように XML 形式で記述する。文書の段落ごとに分けて、名前と本文、そして関連する関心事を記述する。次に (2) の画面で関心事を選択すると、左の画面にその関心事に該当するプログラム要素とドキュメント要素が表示される。その中で、必要なものを選択し、右画面に表示する順番で並べる。その後、(3) の画面に移ると (2) の右画面の内容にしたがってプログラム要素とドキュメント要素を並べたものを見ることができる。前述の通り、Kide エディタ上での編集は元のプログラムに即座に反映されるため、Kide エディタ上でチュートリアル文書を修正すると、修正は元の `README.txt` ファイルに反映される。`README.txt` と Screen.java の 2 つのソースファイルの間を行ったり来たりしながら修正を加える必要はない。

3.4 AOP との比較

Kide は従来 AOP 言語が提供してきたモジュール化についての支援のかなりの部分を同様に提供しているといえる。AOP 言語によるモジュール化の支援が提供するものは大きく次の 3 つである。

- モジュラーな視点 (関連するコードを一カ所に集める)
- ホモジニアス性 (プログラムの複数箇所にかかわる同一の修正を一括して行える)
- オブリビアスネス性 (元のソースコードを修正せずに、プログラムの振る舞いを変えられる)

Kide はこのうち、モジュラーな視点とホモジニアス性を提供する。モジュラーな視点は関心事を定義し、それれに含まれるプログラム要素を Kide エディタで閲覧することで得られる。ホモジニアス性は、関心事に含まれるプログラム要素に対する一括編集操作機能により提供される。提供される機能は AspectJ 言語が提供する before アドバイスにほぼ相当する機能と、after アドバイスに相当する機能の大半と同じである。Kide はオブリビアスネス性は提供しないが、AOP 言語や AOP システムの中にはオブリビアスネス性を提供しないものもあり、これは必ずしも AOP に必須の機能であるとはいえない。

4 Kide を用いたソフトウェア開発

Kide の有用性が発揮されるのは、ソフトウェアの保守改良時である。ソフトウェアはいったん完成してリリースされた後も、不具合の修正や機能拡張などを繰り返しながら保守されてゆく。このような時期には、しばしばバグ管理システムや issue tracker などを用いて個々の不具合や機能拡張要求といった「課題」を管理し、プログラムの修正も課題単位で行われる。個々の課題の改善を繰り返し、一定数の課題の処理が終わったところで次のリリースを実施する。

以下では、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 をどのように利用できるかを、具体的なシナリオを用いて説明する。例として 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用いる。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きた。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。

4.1 シナリオ 1: メモリ消費量の低減

利用者よりメモリ消費量が大きすぎるとの不具合 [13] が報告された。開発チームは検討の結果、単純なメモリリーク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改造をおこなうべきであるとの結論に達した。

コードレビュー用資料

JASSIST-28 に関する修正方法は以下の通り。

- メモリ消費を軽減するために、クラスファイルから不必要な部分を削ぎ落とす 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、その操作が出来るかどうかのチェックを付加した。

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

  public void addMethod(..) throws .. {
    checkModify(ct.isFrozen(), ct.getName(), ct.wasPruned);
    /* 以下省略 */
  }

  public void removeMethod(..) throws .. {
    checkModify(ct.isFrozen(), ct.getName(), ct.wasPruned);
    /* 以下省略 */
  }
  ..
```

図 7. レビュー用報告書の例

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改造が計画された。そのため、情報を保持するいくつかのクラスに、情報の破棄を実行する prune (枝刈り) メソッドを追加することになった。

ここで Kide が利用できれば、あちこちのクラスに追加した prune メソッドのために *pruning* という関心事を定義し、全ての prune メソッドを一覧表示できるようにできる。それぞれの prune メソッドが正しく連係して動作するか確認し、必要に応じて修正することが容易になる。

また prune メソッドを実行した後は、情報が削除されるので、クラス定義の変更はそれ以上できなくなる。誤って変更してしまうことを防ぐため、クラス定義を変更する各メソッドの冒頭に対象のクラスが変更可能かどうかを検査する checkModify メソッドの呼び出しを追加し、変更できないときは例外を投げるように修正することとなった。ここでも、Kide が利用できれば、checkModify メソッドの呼び出しを追加する全てのメソッドを含む関心事 *needCheckModify* を定義し、それらのメソッドの冒頭に checkModify メソッドの呼び出しを一括して挿入できる。

最後に、以上で述べたプログラムの修正について、修正後のプログラムを Subversion にチェックインする前に、コードレビューを受けるとする。そのためには、レビュー用にコードの変更点についての報告書を図 7 のように書く必要があるが、Kide を利用できれば、必要に応じてプログラムの断片を引用した報告書を作成できる。コードレビューでは、実際のプログラムを見る必要があるので、プログラムの断片の引用は非常に有用である。またコードレビューによって、プログラムの間違いが発見されたときでも、Kide を利用していればレビュー用の報告書を表示した Kide エディタから直接プログラムの修正ができる。

コードレビューの結果、修正方法が承認されれば、修正されたプログラムはチェックインされ、本課題の一連の処理は終了である。本課題を処理する間に定義した Kide の関心事は当面不要になるが、保存して将来の利用に備える。例えば本課題の処理として行ったプログラムの修正に不具合があった場合、将来、別の課題としてそれが報告され、さらなるプログラムの修正が行われるかもしれない。その際、今回の処理で定義した関心事を残しておくと、将来のプログラムの修正の際に役立つだろう。

```
  aspect ModifyChecking {
    ..
    before(CtClassType ct) :
     (execution(void CtClassType.addConstructor(..))
     || execution(void CtClassType.removeConstructor(..))
     || execution(void CtClassType.addMethod(..))
     || execution(void CtClassType.removeMethod(..))
                || ...
)
     && this(ct) {
     checkModify(..);
    }
    ..
  }
```

図 8. checkModfy メソッドの呼び出しを抜き出したアスペクト

4.2 シナリオ 2: キャッシュ機能の拡張性

Javassist は元々、バイトコードを操作するための `CtClass` オブジェクトを、一度クラスファイルを読み込んで作成した後は、それをキャッシュに保存していた。同じクラスファイルを何度も読み込む無駄を避けるためである。しかし、Javassist を利用するソフトウェアの都合で、このキャッシュ機構を拡張可能にする機能追加要求が出された。

開発チームは対策として、キャッシュにアクセスしている部分のコードを `cacheCtClass` メソッドとして切り出し、必要に応じて上書き可能にする修正を考えた。しかしキャッシュにアクセスしているコードは、あちこちのメソッドに散らばっている。ここで Kide が利用できれば、該当するコードを含むメソッドを全て集めた関心事 *needCacheCtClass* を定義し、正しく `cacheCtClass` メソッドの呼び出しが挿入されているか、一覧表示して容易に確認できる。また、このような修正内容のコードレビューを受ける際も、シナリオ 1 でも述べたように、Kide を使えば容易にプログラムの一部が引用された文書を作成できる。

4.3 AOP 言語を使った場合のシナリオ

比較のため、AspectJ 言語を用いて上記シナリオ 1 を実施する例を示す。AspectJ 言語を用いる場合、checkModify メソッドの呼び出しは直接該当するメソッドの中に挿入するのではなく、`ModifyChecking` アスペクトとして独立したソースファイルの中に記述することになる。図 8 にアスペクトの定義例を示す。

`ModifyChecking` アスペクトを独立したソースファイル内に定義することで、`checkModify` メソッドを呼ぶ、という関心事に関係するコードはモジュール化され、閲覧性も向上する。しかしながら、逆に実際に `checkModify` メソッドを呼ぶ様々なメソッド、例えば `CtClassType` クラスの addMethod メソッドなどは、そのメソッドの定義をエディタで見ても中で `checkModify` メソッドを呼んでいることがわからない。それを知るためには、AspectJ 言語用の統合開発環境を用い、ツールの支援を受けて開発者自身が考えなければならない。AspectJ 言語の場合、addMethod メソッドなどの閲覧性は逆に悪化しているといえる。

また、AOP 言語ではアスペクト内のコードを織り込む箇所をポイントカットで定義しなくてはならない。そのためには開発者は AOP 言語特有の文法を覚える必要がある。さらに、ポイントカットの定義が正確で、コードを正しい場所に織り込めるかを確認することも容易ではない。この確認のためには、コードが織り込まれるべき全てのメソッド群を精査し、統合開発環境の助けを借りて織り込みの有無を確認しなくてはならない。また AspectJ 言語は、プログラム内部の横断的関心事は扱えるが、プログラムと文書の両方にまたがるような関心事には対応できない。

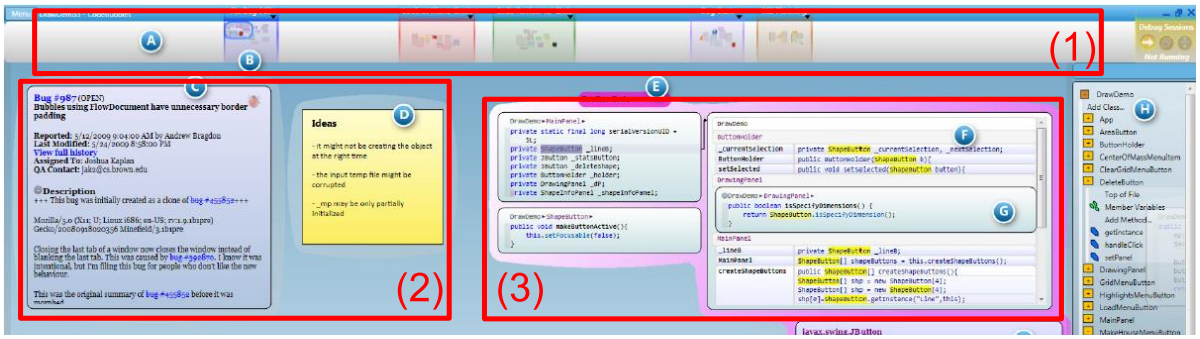


図 9. Code Bubbles IDE

このように本節のシナリオで示したような状況では、AspectJ 言語のような AOP 言語は Kide の置き換えにならない。Kide の方が有用性が高いといえる。一方、当該ソフトウェアの開発が始まったばかりの段階で、設計段階から横断的関心事をとらえており、それをアスペクトとして実装することを決められる状況では、AspectJ 言語の有用性は高い。逆に、そのような状況では Kide はあまり有用ではない。Kide と AspectJ 言語は互いに補完し合う関係にある。

5 開発環境の機構によるモジュール化

開発環境の機能を用いてソフトウェアのモジュール化を支援する研究は以前から存在する [1, 3, 4, 6, 8, 9]。本章では、それらと本研究との比較を行う。

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 IDE のソースコードエディタ上に表示することができる。AOP には、一部のコードをアスペクトに抜き出してしまうと元々の分割の視点からの閲覧がしにくくなる、という問題があった。AJDT を用いればこれを解決できるが、プログラムを編集をする際には、元のプログラムとアスペクトを互いに行き来するため、ファイル間を横断する煩雑な作業を行うことになる。

Fluid AOP[3] は、AOP の機能を IDE サポートによって実現する研究である。Fluid AOP では、3 種類のジョインポイントモデル（JPM）を提供する。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するビューアを作成することができる。これらの機構を用いることで本論文で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、関心事の指定方法に関する支援が乏しいことである。Fluid AOP では、関心事をポイントカットを用いて抽出するため、Fluid AOP を大規模システムの開発に用いるとき、プログラマの関心のあるコードを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 では対話的に関心事の指定出来る機構を提供しているため、容易にプログラマの関心事を網羅することが出来る。さらに、Fluid AOP には、コーディング支援が乏しいという問題がある。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。

Code Bubbles[6] は、プログラム断片を図 9 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 9(3) のように、開発者にとって必要なプログラム断片のみを表示することができるので、編集作業に集中することができ、開発効率が向上する。また、IDE 上で開いている bubble の集合を図 9 (1) の workspace に登録することができるので、2.1 節で挙げた問題を

解決することができる。また、図 9(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。しかし、Code Bubbles には関心事を画面上で集める支援がないため、開発者が bubble を手作業で集めなくてはならない。Kide では、呼び出し関係や継承関係など、関連するコードの探索を支援している。また、Code Bubbles では、関連するメソッドの冒頭に同一のコードを織り込む、などの AOP の機能を提供していない。Kide では、OOP の知識のみで AOP を利用できる機構を提供することを目標としている。

CIDE[8] は、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の欠点と、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の欠点の両方を補う研究である。手法としては annotative 寄りの手法を採用しているが、annotative な手法の欠点であるコードの見にくさを補うために、タグを用いる代わりに色付けを行うことでモジュール化を行う。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を意味し、それらをアスペクトとして出力することが可能である。ただし、CIDE では、プログラムの AST を用いることでこれらの機構を提供している。そのため、特定のプログラミング言語に強く依存しており、本研究のように複数言語間や異なる種類の文書間をまたがるモジュール化は行うことができない。

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、あ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。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 の高い複数の要素をひとつのエディタ上で編集したり、DOI の高い一つの要素にフォーカスを当てることは出来ない。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。

6 まとめと今後の課題

6.1 まとめ

OOP の開発環境で AOP の機能を提供するシステム 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 OOP 言語や AOP 言語での開発における問題点を以下のように解決した。

- 一度決めたプログラム分割とは異なる視点からの編集を行うことができる
- 異なる言語間や異なる文書間にまたがる関心事を抽出することができる
- これらの支援は IDE の機構として提供しているため、特別な知識を習得する必要がない

Kide を用いることでプロジェクトの各工程で動的なモジュール化を行うことができ、高い品質のソフトウェアを短期間で実装することが出来る。

6.2 今後の課題

- 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。Kide を用いることで、開発者のモジュール化コストを減らすことができるのかを検証するとともに、より良い GUI を提供するためのインタビューを行いたいと考えている。

- ホモジニアス性

  3.4 節で述べたように、本研究では AOP のモジュラーな視点とホモジニアス性という利点を開発環境によって提供したいと考えている。しかし、現在の Kide では AspectJ のホモジニアス性の一部しか実現できていない。今後、AspectJ と同等の能力を提供できるようにしたいと考えている。

- 直感的な UI

  3.3 節で述べたように、本システムはプログラムとドキュメントのモジュール化を支援している。開発環境の機能として提供することで、特別な知識なしでこの機能を用いることができる。しかし、実際に開発者に使用してもらう上で現在の UI は直感的とは言い難い。今後行う評価をふまえた上で、より使いやすい UI を提供したいと考えている。

参考文献

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

[2] Sun Microsystems. Javadoc 5.0 Tool. http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/

[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] Ye, Lingdong and De Volder, Kris. Tool support for understanding and diagnosing pointcut expressions. In AOSD '08 - Aspect-oriented software development, pages 144-155. Brussels, Belgium, 2008.

[6] Bragdon, Andrew and Reiss, Steven P. and Zeleznik, Robert and Karumuri, Suman and Cheung, William and Kaplan, Joshua and Coleman, Christopher and Adeputra, Ferdi and LaViola,Jr.and Joseph J. Code bubbles: rethinking the user interface paradigm of integrated development environments. In ICSE'10 - IEEE International Conference on Software Engineering, pages 455-464. New York, NY, USA, 2010.

[7] Christian Koppen and Maximilian Stoerzer. Pcdiff: Attacking the Fragile Pointcut Problem.

[8] Kästner, Christian and Apel, Sven and Kuhlemann, Martin. Granularity in software product lines. In ICSE '08 - International Conference on Software Engineering, pages 311-320. New York, NY, USA, 2008.

[9] Coppit, David and Painter, Robert R. and Revelle, Meghan. Spotlight: A Prototype Tool for Software Plans. In ICSE '07 - International Conference on Software Engineering, pages 754-757. Washington, DC, USA, 2007.

[10] Tarr, Peri and Ossher, Harold and Harrison, William and Sutton,Jr., Stanley M. N degrees of separation: multi-dimensional separation of concerns. In ICSE '99 - International Conference on Software Engineering, pages 107-119. Los Angeles, California, United States, 1999.

[11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J.,Griswold, W.G.: An overview of AspectJ. In: ECOOP 2001 Object-Oriented Programming. pp. 327 353. LNCS 2072, Springer (2001)

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspectoriented programming. In ECOOP, pages 220 242, 1997.

[13] JBoss Community - JBoss Issue Tracker - Javvasist https://issues.jboss.org/browse/JASSIST-28