平成17年度 学士論文

アスペクトによる
モジュール拡張を強調した
アウトラインビューア

東京工業大学 理学部 情報科学科
学籍番号 02-2149-7
堀江 倫大

指導教員
千葉 滋 助教授

平成18年2月13日

概要

アスペクト指向プログラミングは既存のモジュールを拡張するための技術である。オブジェクト指向プログラミングでは各モジュールをオブジェクト (クラス) として分けることで、関心事の分離を行う。近年、複雑で多様化するアプリケーションに対して、この関心事の分離が十分に達成されているとはいえない。例えば、ロギングコードや同期処理、イベント通知処理といった処理はモジュール間をまたがって散在してしまい、ソースコードの再利用性を低下させてしまう。モジュール間にまたがるこれらの処理を横断的関心事と呼ぶ。アスペクト指向プログラミングはこの横断的関心事を別のモジュールにまとめて扱うことができる。

代表的なアスペクト指向言語のひとつに Java を言語拡張した AspectJ がある。AspectJ では、アスペクトを定義することでクラスに新たな処理を織り込み、クラスを拡張することができる。しかし、クラスのソースコードを見ただけではアスペクトによって、どのような拡張が行われたのか知ることができない。織り込みに関する情報を得るには、クラスに関連する全てのアスペクトのソースをプログラマが読んで把握するしかない。これは大変労力のいる作業である。そこで、ツールによる支援が必要である。統合開発環境の Eclipse プロジェクトで開発され AspectJ による開発を支援するプラグインに AJDT（AspectJ Development Tools）がある。AJDT には AspectJ の利用を支援するツールが数多く提供されている。それらのツールは、ソースコードのどの箇所でアスペクトが織り込まれるかという情報を視覚的、あるいは直感的に提供するものである。

しかし、AJDT ではアスペクトが織り込まれた位置の情報を提供するだけであり、クラスを拡張するという視点に立った情報を示すツールではない。例えば、call ポイントカットによって foo メソッドを指定した場合、foo の呼び出し側のソースコードにはアドバイスの情報が表示されるが、foo が宣言されている側のソースコードには表示されない。そのため、foo メソッドがアスペクトによって拡張されたことを知ることができない。Gregor Kiczales は「アスペクト指向とは、既存のモジュールに新しい境界を作るもので、それを定義するためのインターフェイスが存在する」と主張しており、そのインターフェイスを Aspect-Aware Interface と呼んでいる。これは織り込みの情報を提示してアスペクトの存在を表すのではな

く、モジュール自身を拡張するものとしてアスペクトを表すものである。

本研究では、この Aspect-Aware Interface の考えを基にして、アスペクトによるモジュール拡張を強調したアウトラインビューアを Eclipse のビュープラグインとして開発した。各モジュール（クラス、メソッド、フィールド）の属性を階層的に表示するときに、アスペクトの情報を重ねて表示することができる。call、set、get ポイントカットによって指定されたアドバイスの場合にも、それによって拡張されるターゲット側のモジュールに情報を重ねて表示する点が AJDT と異なる。また、このビューには選択したモジュールとそれを拡張するアドバイスの Javadoc コメントを同時に並べて表示させることもできる。これにより、それぞれ違う場所にあった関連のある Javadoc コメントを同時に見ることができるようになる。本研究では実際のアプリケーションを用いて実験をすることにより実用的な速度で Javadoc コメントの表示が可能なことを確認した。それとともに、本システムによりばらばらの場所にあった関連のあるコメントを統合することで有用な情報が得られることの検証も行った。

謝辞

本研究を進めるにあたり、研究の方向付けや論文の組み立て方について数々の有用な助言を頂き、ご指導をしてくださった千葉滋先生に大変感謝いたします。

論文のスタイルファイルを作成していただいた光来健一先生、参考になる研究について有用な助言を頂きご指導してくださった西澤無我氏、論文を構成するにあたり相談に乗っていただいた柳澤佳里氏に感謝いたします。

また、薄井義行氏には、システムの設計・実装など本研究の根源的な問題にはじまり、論文の書き方、実験方法に至るまで親身になってご指導して頂きました。石川零氏、日比野秀章氏、竹内秀行氏からは、参考となる論文について様々な助言を頂きました。心より感謝いたします。また、本研究に必要な知識をくださった青木康博氏、熊原奈津子氏、共に研究活動に励んだ皆さんに御礼を申し上げます。

目 次

図 目 次

表 目 次

第1章　はじめに

アスペクト指向プログラミングは既存のモジュールを拡張するための技術である。オブジェクト指向プログラミングでは各モジュールをオブジェクト (クラス) として分けることで、関心事の分離を行っていた。近年、複雑で多様化するアプリケーションに対して、この関心事の分離が十分に達成されているかというと、決してそうとはいえない。例えば、ロギングコードや同期処理、イベント通知処理といった処理はモジュール間をまたがって散在してしまい、ソースコードの再利用性を低下させてしまう。モジュール間にまたがるこれらの処理を横断的関心事と呼ぶ。アスペクト指向プログラミングはこの横断的関心事を別のモジュールにまとめて扱うことができる技術である。

代表的なアスペクト指向言語のひとつに Java を言語拡張した AspectJ がある。AspectJ では、アスペクトを定義することでクラスに新たな処理を織り込み、クラスを拡張することができる。しかし、クラスのソースコードを見ただけでは、アスペクトによってどのような拡張が行われるのかということを知ることができない。横断的関心事がアスペクトとしてひとつにまとめられているからである。アスペクト指向に見られるこの性質を obliviousness と呼ぶことがある。織り込みに関する情報を得るには、対象となっているクラスに関連する全てのアスペクトのソースをプログラマが読んで把握するしかない。これは大変労力のいる作業である。

そして、この obliviousness の性質を批判する声がある。この性質を持ったモジュールでは、言語レベルでの Modular-Reasoning が可能でなくなるというのがその理由である。Modular-Reasoning が可能とは、あるモジュールの挙動を把握したいときに、そのモジュールの実装やインターフェイス、さらにはそれらの中で参照されるモジュールのインターフェイスや仕様（実装ではない）を見れば十分であるということである。横断的関心事の分離を行ったクラスなどのコードでは、実行時にどこにアスペクトが織り込まれるかの情報がないので、そのコードを見ただけでは挙動を把握できない。これに対処するために、元のコードにアスペクトが織り込まれることを示す記述をしたり、アスペクトの影響の及ぼす範囲を削減する操作をする研究が存在する。しかし、これらは全てアスペクトの影響力を抑えてしまう働きがあったり、AspectJ と同等の機能を備えていなかっ

たりする。

一方で、Gregor Kiczales は「アスペクト指向とは、既存のモジュールに新しい境界を作るもので、それを定義するためのインターフェイスが存在する」と主張しており、そのインターフェイスを Aspect-Aware Interface と呼んでいる。これは織り込みの情報を提示してアスペクトの存在を表すのではなく、モジュール自身を拡張するものとしてアスペクトを表すものである。そして、Aspect-Aware Interface の存在により Modular-Reasoning が可能となり、obliviousness の性質を補うことができると主張している。本研究では、この Aspect-Aware Interface の概念を基にしている。

また、アスペクト指向の obliviousness の性質を補完するには、ツールによる支援があればよいというのが一般的な見解である。モジュールの分離という観点から見ればこの性質は好ましいものだからである。統合開発環境の Eclipse プロジェクトで開発され AspectJ による開発を支援するプラグインに AJDT（AspectJ Development Tools）がある。AJDT には AspectJ の利用を支援するツールが数多く提供されている。例えば、アウトラインビューや Cross-References ビュー、Visualiser ビューなどのツールがそうである。それらのツールは、ソースコードのどの箇所でアスペクトが織り込まれるかという情報を視覚的、あるいは直感的に提供するものである。

しかし、AJDT ではアスペクトが織り込まれた位置の情報を提供するだけであり、クラスを拡張するという視点に立った情報を示すツールではない。例えば、call ポイントカットによって foo メソッドを指定した場合、foo の呼び出し側のソースコードにはアドバイスの情報が表示されるが、foo が宣言されている側のソースコードには表示されない。そのため、foo メソッドがアスペクトによって拡張されたことを知ることができない。

本研究では、この Aspect-Aware Interface の考えを基にして、アスペクトによるモジュール拡張を強調したアウトラインビューアを Eclipse のビュープラグインとして開発した。各モジュール（クラス、メソッド、フィールド）の属性を階層的に表示するときに、アスペクトの情報を重ねて表示することができる。call、set、get ポイントカットによって指定されたアドバイスの場合にも、それによって拡張されるターゲット側のモジュールに情報を重ねて表示する点が AJDT と異なる。また、このビューには選択したモジュールとそれを拡張するアドバイスの Javadoc コメントを同時に並べて表示させることもできる。これにより、それぞれ違う場所にあった関連のある Javadoc コメントを同時に見ることができるようになる。

以降、2 章では関連のある研究や既存のツールである AJDT のツールの問題点について述べる。3 章では、本研究の基となった論文について、本システムの仕様、実装方法について順を追って解説していく。4 章、5 章

で本システムについての実験、検証を行う。実験では、実際のアプリケーションを用いて行うことにより実用的な速度で Javadoc コメントの表示が可能なことを確認した。また、検証では本システムによりばらばらの場所にあった関連のあるコメントを統合することで有用な情報が得られることを確認した。最後に、6 章で本稿をまとめる。

第2章　アスペクト指向プログラミングのための開発支援ツール

2.1　アスペクト指向

オブジェクト指向プログラミング（OOP）によりソフトウェアをモジュール化し関心事を分離する技術は十分に成功したといえるが、より複雑で多様化するアプリケーションに対しても関心事の分離が達成されているかというと決してそうとはいえない。分離できずにいる関心事がソースコードの質を劣化させているといえる。そのような例にロギングコードや同期処理、イベント通知処理が上げられる。関心事がプログラム全体に散在して互いにもつれあった状態にあるため、このような関心事が潜在しているソースコードはモジュールとしての性質に欠ていると言える。このような関心事を横断的関心事と呼ぶ。アスペクト指向プログラミング（AOP）[8] は横断的関心事を分離して別のモジュールにひとまとめにして扱うことのできる技術である。オブジェクト指向でもかなり高度なモジュール化が行えるが、モジュール同士のつなぎ換えを行おうとしたとき、プログラムの随所を変更する必要が出てくる。これは、モジュール同士の結合の度合いが高いためである。アスペクト指向とは、このモジュールの結合を緩め、再利用性を高めた技術といえる [11]。

2.2　開発支援ツールの必要性

アスペクト指向プログラミングはアスペクトの側に横断的関心事をまとめて記述することでモジュールの拡張を達成している。AspectJ [7] などのアスペクト指向言語では、クラス側には横断的関心事がアドバイスとしてどの時点で実行されるかという情報が記述されない。そのため、実行時にコード中のどの箇所にアスペクトが織り込まれるかが分からない。この性質のことを obliviousness [10] と呼ぶことがある。織り込みに関する情報を得るには、プロジェクト内の関係のあるソースコードを自分で読んで把握するしかない。これはプログラマにとって大変労力のいる作業であり、作業効率を下げてしまう。そのため、このような開発上の問題点を補完するツールが必要不可欠となるというのが一般的な見解である。

2.3　AspectJ

汎用的なアスペクト指向言語として AspectJ が上げられる。AspectJ はアスペクト指向プログラミングを Java 言語上で言語拡張し実現した言語である。AspectJ 用のコンパイラである ajc によって、AspectJ プロジェクトのコンパイル時にはアスペクトとして記述したコードは java プログラムに変換される。そのため通常の JVM で java プログラムとして実行できる。

ジョインポイント（JoinPoint）

Java クラスやインタフェースを含む AspectJ プログラムにおいて、実行時にアドバイスの実行を割り込ませることが可能なコード上の位置をのことを指す。割り込み位置としては、メソッド実行時、メソッド呼び出し時、コンストラクタの実行時、フィールドへのアクセス時といったプログラムのある時点をあらわす部分が相当する。例えば、メソッド呼び出し時、コンストラクタ呼び出し時、メソッド実行時、フィールド参照時などプログラム実行中のある時に相当する部分がジョインポイントにあたる。

ポイントカット（Pointcut）

AdpectJ プログラム内に存在する全てのジョインポイントの中から条件を指定することで作る集合を指す。AspectJ 言語があらかじめ用意しているプリミティブポイントカットと、名前付きポイントカットがある。

・call(メソッドパターン)、call(コンストラクタパターン)
メソッド呼び出し時、コンストラクタ呼び出し時
＜使用例＞ call(void Point.getX())
・execution(メソッドパターン)、exeucution(コンストラクタパターン)
メソッド実行時、コンストラクタ実行時
＜使用例＞ execution(void Point.getX())
・get(フィールドパターン)
フィールド参照時
＜使用例＞ get(int Point.x)
・set(フィールドパターン)
フィールド代入位置
＜使用例＞ set(int Point.x)
・handler(例外のタイプパターン)
例外ハンドラの実行時

＜使用例＞ handler(Exception)
・initialization(コンストラクタパターン)
＜使用例＞ initialization(Point.new())
オブジェクトの初期化時
・stataicinitialization(タイプパターン)
クラスの静的な初期化時
＜使用例＞ staticinitialization(Test)
・preinitialization(コンストラクタパターン)
オブジェクトの事前初期化時
＜使用例＞ preinitialization(Point.new())
・adviceexecution
アドバイスの実行時
＜使用例＞

```
public aspect Logger() {
  static int num = 0;
  before() : execution(Point.move(..)) {
    ...
  }
  before() : adviceexecution() && if(num++ ¡ 5) {
    ...
  }
}
```

・within(タイプパターン)
指定されたタイプパターン内の全てのジョインポイント
＜使用例＞ within(Point)
・withincode(メソッドパターン)
指定されたメソッドパターン内の全てのジョインポイント
＜使用例＞ withincode(void Point.getX())
・cflow(ポイントカット)
指定したポイントカットによって選択された全てのジョインポイントについて、各ジョインポイントの開始と終了の間にあるジョインポイント全て
＜使用例＞ cflow(call(void Point.setX())
・cflowbelow(ポイントカット)
cflow(ポイントカット) によって選択されるジョインポイントから、指定したポイントカットによって選択されるジョインポイントを除いたもの
＜使用例＞ cflowbelow(call(void Point.setX())

また下のようにポイントカットにはオペレータを使用することができ

る。
• &&
＜使用例＞ call(void Point.getX()) && within(Point)
　　Point クラス内に限る getX メソッド呼び出し時
• ||
＜使用例＞ call(void Point.getX()) || execution(void Point.getX())
　　getX メソッド呼び出しおよび実行時
• **!**
＜使用例＞ !call(void Point.getX())
　　getX メソッド呼び出し以外の時点

アドバイス（Advice）

プログラムの実行が指定するポイントカットによって選択されたジョインポイントに差し掛かった時点で実行されるコードのことである。コードを実行するタイミングには、ジョインポイントの事前、事後、戻り値を伴う事後、例外を伴う事後、その時点の 5 種類をそれぞれ before、after、after returning、after throwing、around で指定することができる。例えば、beofre アドバイスの記述は下のようになる。

```
before() : call(void Point.getX()) {
   //アドバイスの実行内容
   ...
}
```

アスペクト（Aspect）

ジョインポイントとアドバイスの組み合わせを指定するモジュール単位を指す。1 つのアスペクト内に複数のポイントカットとアドバイスを指定することが可能である。例えば、アスペクトの記述の下のようになる。

```
aspect Signal {
  before() : call(void Point.getX()) || call(void Point.getY()) {
    Signal.callSignal();
    ...
  }
  after() : execution(void Rect.test()) && within(Rect) {
    ....
```

```
  }
  ...
}
```

2.4 Modular-Reasoning

アスペクト指向言語は、オブジェクト指向によって達成された既存のモジュールを拡張することができる技術である。AspectJ の場合、横断的関心事をアスペクトとして別のモジュールにひとまとめにすることができる。そのため、AspectJ では 1 つのクラスやメソッドの挙動を理解しようと思ったときに関係のある全てのプログラムを解析する必要がある。そのクラスやメソッドの定義を見ただけでは、プログラム中のどの箇所にアスペクトが織り込まれるかが分からないからである。これは関心事を分離するという観点から見れば、アスペクト指向プログラミングのもつ良い性質である。しかし、アスペクト指向の研究者の中にも、このようなアスペクト指向の設計は実用上はあまりよくないという見解も中にはある。横断的関心事を分離してモジュールを拡張することができても、そのモジュールを解析するには関係のあるプログラムを見なければならないため、オブジェクト指向プログラミングに見られるブラックボックス（たとえ、横断的関心事が含まれていてもである）としての構造が消えてしまうという考えである。以上のような議論があるときに、Modular-Reasoning が可能かどうかというのがひとつの判断基準となることがある。

クラスやメソッドといったあるモジュールに関して、その挙動を理解したい場合がある。このとき、そのコード中の記述やその中で参照している他のモジュールのインターフェイスや仕様（例えば Javadoc コメントなど）だけを見て挙動が理解できるとき、そのモジュールは Modular-Reasoning が可能であるという。reasoning は推論という意味である。

アスペクト指向言語では、横断的関心事に関する情報がモジュールから抜き取られるために Modular-Reasoning が困難になるとされている。以下の 2 つの研究 [1, 2, 5] はアスペクト指向言語でも Modular-Reasoning が可能になるように、ある言語設計での制約を設けることにより解法を与えている。

2.4.1 Observer と Assistant

一般的に、AspectJ のようなアスペクト指向言語ではあるモジュールの実効上の仕様（effective specification）は全てのプログラムを解析することによって決定されるために、Modular-Reasoning が困難であるといわ

れている。しかし、アスペクトの中でも Modular-Reasoning が可能なものとそうでないものの 2 種類に分類することができるという考えがある [2]。それを Assistant と Observer と呼んでいる。Observer は影響するモジュールの実効上の仕様を変化させないもののことを指す。つまり、アスペクトの中でも Modular-Reasoning が維持されるものである。Assistant はその逆で、実効上の仕様を変えてしまうものである。

また、Observer と Assistant の区別には JML（Java Modeling Language）を前提として用いている。JML は契約記述言語のひとつであり、下のように記述する。

```
interface FigureElement {
/*@ model instance int xCtr, yCtr; @*/
/*@ public behavior
  @ forall int oldx, oldy;
  @ requires oldx == xCtr && oldy == yCtr
  @ && dx >= 0 && dy >= 0;
  @ assignable xCtr, yCtr;
  @ ensures xCtr == oldx + dx
  @ && yCtr == oldy + dy
  @ && \result == this;
  @ signals (Exception z) false; @*/
FigureElement moveNE( int dx, int dy );
/*@ public behavior
  @ ensures \result == xCtr;
  @ signals (Exception z) false; @*/
/*@ pure @*/ int getX();
/*@ public behavior
  @ ensures \result == yCtr;
  @ signals (Exception z) false; @*/
/*@ pure @*/ int getY();
}
```

JML では xCtr、yCtr などは model field と呼ばれオブジェクトの抽象的な状態を指定するものである。instance とは、このインタフェースを実装する全てのクラスにおいてそれらが model field になるということを意味する。represents は model fields の値は実際の変数とどのように関連しているかを規定するものである。また、assign はこのような具体的な値の割り当てを規定したものである。behavior により仕様を規定する。behavior 中にはさまざまな節を記述する。forall 節によって論理変数を導

入し、require 節は事前条件、assignable 節ではその枠組み、ensures 節は事後条件、signals 節は例外が発生したときの事後条件をそれぞれ表す。事後条件では、\return という表記を用いてメソッドの返す値を参照する。事後条件中で、$\backslash old(E)$ と書くことで E という式の以前の状態を参照することができる。例えば、下のように書くこともできる。

```
/*@ public behavior
  ...
  @ ensures getX() == \old(getX() + dx)
  @ && getY() == \old(getY() + dy)
  ...
```

Java における Modular-Reasoning

もし、FigureElement 型のオブジェクトを扱いたいときには、例えば FigureElement インタフェースの情報のみからどのようなクラスなのかが推測 (reason) できる。また、もし FigureElement を実装した Point クラスのインスタンスを扱う場合にも、Point クラスの記述を見ればよい。Point#moveNE メソッドの挙動について推測したいときには、FigureElement モジュールの仕様も見なければならないが、Point クラスは FigureElement を実装しているクラスだから、FigureElement#moveNE の仕様を参照することも依然としてひとつのモジュールの内部についての推測と考えることができる。

このように java のようなオブジェクト指向言語の場合、Modular-Reasoning が可能である。

AspectJ における Modular-Reasoning

アスペクト指向言語において、実効上の仕様が書き換わる例を以下で示す。FigureElement インタフェースを実装する Point クラスの moveNE メソッドの仕様が下のようであったとする。dx または dy が負の場合に例外を発生することが分かる。

```
public behavior
   forall int oldx, oldy;
   requires oldx == xCtr && oldy == yCtr
       && dx >= 0 && dy >= 0;
   assignable xCtr, yCtr;
   ensures xCtr == oldx + dx
       && yCtr == oldy + dy
```

```
        && \result == this;
   signals (Exception z) false;
also
   public behavior
      requires dx < 0 || dy < 0;
      ensures false;
      signals (Exception z)
         z instanceof IllegalArgException;
```

また、Point クラスの moveNE メソッドの呼び出し時に実行されるアドバイスの仕様が下のようであるとする。このアドバイスは dx、dy がともに負の場合に例外を発生することが分かる。

```
   public behavior
      requires dx < 0 && dy < 0;
      ensures false;
      signals (Exception z)
         z instanceof IllegalArgException
         && z.getMessage().equals(MOVE_SW);
```

AspectJ では、Point モジュールや Point を参照する実行モジュールからはアスペクトの挿入が明確に記述されることなくアドバイスが実行されてしまう。コンパイル後の moveNE メソッドの仕様は上の 2 つの仕様を単純に合わせたものになるが、この仕様を見ただけでは Modular-Reasoning が可能とはいえない。2 つの変数である dx、dy の論理積と論理和をとる仕様が混在しているためである。結局は、Point モジュールについての推論をするには Point クラスが対象となる全てのアスペクトを考慮しなければならない。

Assistant

実効上の仕様が書き換えられてしまうようなモジュールでは、その Assistant を明記しなければならないことにする。そこで、Assistant を適用するモジュールやそのモジュールの実行コードでは、

accept TypeName;

と記述することにする。例えば、Point クラスが moveNE に関するアスペクトを許可すれば、この Assistant は全ての Point#moveNE メソッドの呼び出し時に適用されることになる。実行コード中で許可されたときは、Assitant は実行コードからの呼び出しによるメソッド呼び出し以外はアドバイスを許可しない。下の図は、moveNE メソッドの呼び出しの制御フ

ローを図式化したものである。灰色の矢印は例外発生時の制御フローである。

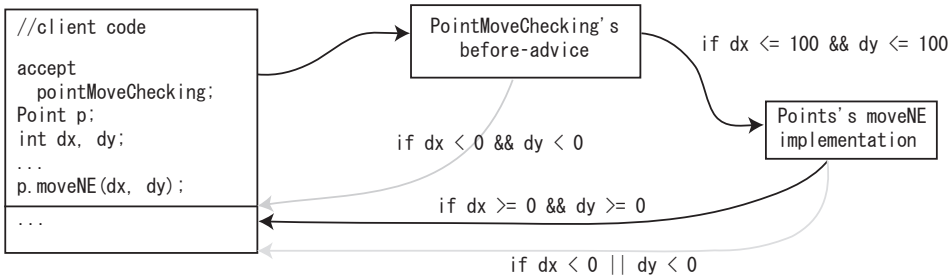


図 2.1: 制御フロー

実行コードか実装モジュールのどちらかで許可された assistant を呼び出すときには、呼び出しの挙動はある制御フローのパスに従ってシーケンシャルなものになる。それにより、メソッドと assistant の仕様を見れば、可能性のある呼び出しをいくつか推測することができるようになる。そこで、下のように PointcutMoveChecking Assistant の仕様を書き換えるべきである。

```
/*@public behavior
  @    requires dx > 0 || dy > 0;
  @    ensures true;
  @    signals (Exception z) false;
  @also
  @public behavior
  @    requires dx < 0 && dy < 0;
  @    ensures false;
  @    signals (Exception z)
  @      z instanceof IllegalArgException
  @        &&z.getMessage().equals(MOVE_SW); @*/
```

問題点

以下に、この言語設計・仕様についての問題点を考える。

- **開発効率**
  JML という契約言語を使用することを前提にしているため、AspectJ

に比べてソースコードが複雑になっている。コードの記述、解読に多くの時間を費やす。これは開発効率の低下につながる可能性がある。

- **アスペクトの種類の判断**
  開発者はアスペクトが Assistant であるか Observer であるかを判断しなければならない。その判断を間違えると、Modular-Reasoning は不可能となる。

2.4.2　Open Modules

アスペクト指向は、ロギングや同期処理、イベント通知などの横断的関心事をより綺麗にモジュール化する技術であるが、一方で従来のカプセル化のメカニズムを壊してしまっているともいえる。アスペクト指向プログラミングに基づいた TinyAspect という新しい言語を導入することでこれを解決しようという研究 [1, 2] がある。それを通して Modular-Reasoning 可能なモジュールを生成するのがこの研究の目的である。

アスペクト指向言語の特性

アスペクト指向は quantification と obliviousness の性質をもつ。quantification とは、コードを 1 箇所にまとめて記述することができるという性質のことを指す。下の例では、moveBy メソッドの実装をアスペクトにまとめて記述している。obliviousness とは、数量化が適応された部分では特別に機能強化の受け皿を用意する必要がないということである。例えば、moveBy メソッドはアドバイスに備えて特別何も用意する必要はない。

```
package shape;
public class Point extends Shape {
   public void moveBy(int dx, int dy) {
       x += dx; y += dy;
       ...
   }}
public class Rectangle extends Shape {
   public void moveBy(int dx, int dy) {
      p1x += dx; p1y += dy;
      p2x += dx; p2y += dy;
      ...
   }}

package constraints;
aspect CheckSimulationConstraints {
   pointcut moves():
      call(void shape.*.moveBy(..))
```

```
   after(): moves() {
      simulation.checkInvariants();
   }
}
```

上の例では、横断的関心事に関する情報を元のコードから上手く隠すことができるようになっている。しかしそれと同時に、上のような例な場合、コードの書き換えを難しくしているともいえる。これは、アスペクトと元のコードが強く結びついているためである。AspectJ などのアスペクト指向言語では、アスペクトがカプセル化の境界を超えて元のコードに影響してくるので、情報隠蔽の原則が壊されてしまう。このような理由から、上の例ではもしコードの内容が次のように書き換えられるとその影響を受けてしまう。例えば、Rectangle.moveBy メソッドを、

```
p1.moveBy(dx, dy);
p2.moveBy(dx, dy);
```

のように書き換えたとき、メソッド内でアドバイスは 2 度余計に呼ばれてしまうことになる。これは、アスペクトが Shape パッケージの情報隠蔽化の境界を侵害しているために起こる問題だと考えられる。論文では、この情報隠蔽の問題について、直接、言語ベースでの解法を示している。

TinyAspect

TinyAspect は新しく開発されたアスペクト指向言語である。その名前が示しているように、TinyAspect はあまり多くの機能を備えておらず、ラムダ式を用いて、unit、declaration、pointcuts、around アドバイスを定義する。下の表は、TinyAspect の文法である。Names は単純な識別子であり、Expressions は単一のラムダ式を表し、names、関数、関数適用から成る。

Names	$n ::= x$
Expressions	$e ::= n \mid \mathbf{fn} x : \tau => e \mid e1e2 \mid ()$
Declarations	$d ::= \bullet \mid \mathbf{val}\ x = e\ d \mid \mathbf{pointcut}\ x = p\ d \mid$
	$\mathbf{around}\ p(x : \tau) = e d$
Pointcuts	$p ::= n \mid \mathbf{call}(n)$
Types	$\tau, \sigma ::= \mathbf{unit} \mid \tau 1 \rightarrow \tau 2 \mid \mathbf{pc}(\tau 1 \rightarrow \tau 2)$

多くのアスペクト指向言語では、ポイントカットとアドバイスは別のクラスに宣言的に記述される。しかし、ソースレベルでの正確なモデルにな

るために、TinyAspect のプログラムは下のように連続した宣言から構成される。それぞれの宣言には、それに続く宣言を含むスコープがあり、空であったり、変数束縛、ポイントカットやアドバイスの束縛であったりする。他の宣言中で使用されたりアドバイスを受け取ったりできるように、**val** 宣言では値に static な名前を割り当てる。**pointcut** 宣言は、あるポイントカットに名前を付ける。**call(n)** の形式のポイントカットは宣言 n で定義された関数呼び出しを参照する。**n** の形式のポイントカットはポイントカット宣言の名前を付け替えるだけである。**around** 宣言はある関数への呼び出しを記述するポイントカット p を指定し、変数 x をその関数の引数に割り当て、オリジナルのコードの代わりにアドバイス **e** を実行する。また、**e** が元の関数を呼び出せるように **proceed** が用意されている。

上の文法に則って記述した例として、Cache アスペクトを利用して計算効率を上げたフィボナッチ関数を上げる。

TinyAspect は言語仕様に再帰を定義していないので、fib 関数そのものは 1 を返すだけである。その代わりに、around アドバイスを用いて再帰を実装している。fib を呼び出すときは、常に最初にこのアドバイスが実行されることになる。

$$
\begin{array}{l}
\textbf{val}\ fib = \textbf{fn} x : \textbf{int} => 1 \\
\textbf{around}\ \textbf{call}(fib)\ (x : \textbf{int}) = \\
\qquad \textbf{if}\ (x > 2) \\
\qquad\quad \textbf{then}\ fib(x-1)\ +\ fib(x-2) \\
\qquad\quad \textbf{else proceed}\ x \\
\\
(*\ advice\ to\ cache\ calls\ to\ fib\ *) \\
\textbf{val}\ inCache = fn... \\
\textbf{val}\ lookupCache = fn... \\
\textbf{val}\ updateCache = fn... \\
\\
\textbf{pointcut}\ cacheFunction = \textbf{call}(fib) \\
\textbf{around}\ cacheFunction(x : \textbf{int}) = \\
\qquad \textbf{if}\ (inCachex) \\
\qquad\quad \textbf{then}\ lookupCache\ x \\
\qquad\quad \textbf{else let} v = \textbf{proceed} x \\
\qquad\qquad \textbf{in}\ updateCache\ x\ v;\ v
\end{array}
$$

TinyAspect の文法に従うと、fib 関数を呼び出すとまず最初に Cache アド

バイスが呼び出される。もし Cache アドバイス中でが proceed を呼べば、次に再帰を定義している方のアドバイスが実行されることになる。さらに、このアドバイス中で proceed が呼ばれたときには、fib 関数が呼び出される。しかし、もしアドバイス中で再帰呼び出しをした場合には、Cache アドバイスが実行され、それゆえ全ての再帰呼び出し中で期待通りキャッシュが機能することになる。

Open Modules

TinyAspect を Open Module を用いて拡張する。これは、実行コードとモジュールの間の抽象化境界（abstraction boundary）を補強するコードをプログラム中に埋め込めるようにしたモジュールシステムのことを指す。

[**定義:Open Module**] あるモジュールから見て、外部のアスペクトに

- モジュールのインタフェースを通して外部の関数の呼び出し時にアドバイスの実行を許す
- モジュールのインタフェース内のポイントカットにアドバイスの実行を許す
- あるモジュールの内部からそのモジュール内の関数の呼び出し時にアドバイスを実行することを許さない

ようなモジュールシステムのことである。

下の表が TinyAspect の新しい文法である。Names は単純に n と表したものと修飾させた m.x がある。m とはモジュールのことである。宣言は構造に対して束縛をすることができ、Types は **sig** β **end** という形式で定義できる。β はモジュール内での変数のリストである。第一段階のモジュール表現には namae、宣言のリストを持った struct、モジュールをシグネチャで隠蔽するための m :> σ などがある。**functor**(x:σ) => m は引数としてシグネチャσ のモジュール x を取得し、x に依存したモジュールを m を返す。functor 適用は関数適用と同様に m_1、m_2 の形式で書ける。

$$
\begin{array}{lll}
Names & n & ::= \ ... \mid m.x \\
Declarations & d & ::= \ ... \mid \textbf{structure}\ x\ =\ m\ d \\
Modules & m & ::= \ n \mid \textbf{struct}d\textbf{end} \mid m\ :>\ \sigma \mid \\
 & & \quad \textbf{functor}(x:\sigma)\ =>\ m \mid m1\ m2
\end{array}
$$

$$
\begin{array}{lrcl}
Types & \tau, \sigma & ::= & ... \mid sig\ \beta\ end \\
Decl.values & d_v & ::= & ... \mid \textbf{structure}\ x\ = \bigcirc\ d \\
Module\ values & m_v & ::= & \textbf{struct}\ d_v\ end \mid \textbf{functor}(x : \sigma)\ =>\ m \\
Contexts & C & ::= & ... \mid \textbf{structure}\ x\ =\ \bigcirc\ d \mid \\
 & & & \textbf{structure}\ x\ \equiv m_v \mid \textbf{struct}\ \bigcirc\ \textbf{end} \mid \\
 & & & \bigcirc\ :>\ \sigma \mid \bigcirc\ m2 \mid m_v\ \bigcirc
\end{array}
$$

下のように、functor を用いてフィボナッチ関数を書き換えた。Cache functor は単一の要素 f を取る。f は int->int のシグネチャの関数呼び出しのポイントカットである。around アドバイスは引数 X というモジュールから実行される。fib 関数は Math モジュール内にカプセル化されている。このモジュールは具体的には、fib 呼び出しとポイントカット p を関連付ける Cache モジュールの記述を書くことでキャッシングの実装をしている。結局、fib 関数であることしか外部に公開せずに Math モジュールを構成している。

Math モジュールは外部から隠蔽されているので、Math.fib モジュールに対して外部のアドバイスは fib の外部呼出し時にしか実行されず、内部の再帰呼び出し時には影響しない。Math モジュールの呼び出し元はキャッシングが適用されるかどうかが分からないことになる。こうすることで、実行コードはモジュールの実装が変えられてしまっても影響を受けることがなくなる。

```
structure Cache =
  functor(X : sig f : pc(int-> int) end) =>
    struct
      aroundX.f(x : int) = ...
        (* same definition as before *)
    end

structure Math = struct
   val fib = fn x : int => 1
   around call(fib) (x : int) =
     if (x > 2)
        then fib(x - 1) + fib(x - 2)
        else proceed x

   structure cacheFib =
      Cache (struct
```

```
                pointcut f = call(fib)
            end)
end :> sig
  fib : int− > int
end
```

問題点

- **可読性**
  Lisp の文法を拡張しているために、Java を言語拡張した AspectJ と違って汎用性に欠ける。また、オブジェクト指向の欠点を補うものというアスペクト指向の立場とは多少着眼点がずれている。

- **機能面**
  現時点では、AspectJ で実装可能な機能で OpenModules にはないものが多い。

2.4.3　問題点のまとめ

以上に述べた 2 つの研究では、アスペクト指向言語で Mdular-Reasoning が可能になる仕様、設計を構築することに集中しているために、アスペクト指向言語のもつ良い性質を損なっている部分が多い。また、そもそもアスペクト指向には Modular-Reasoning が可能であるという主張を次章で紹介する。

アスペクトを Observer と Assistant に分類した場合 [5]、アスペクトが Observer ならばプログラマは得に何の処理もしなくて済むが、Assistant であった場合にはアスペクトの影響のあるクラス側にそれを許可するかしないかの情報を明記する必要があった。つまり、アスペクトの影響力を抑えて予想外の挙動を防ぐというものである。しかし、クラスに Assistant の影響があるかどうかはプログラマが自分で判断しなければならないことになる。

また、OpenModule [1] は、クラス側でインタフェースを定義することでアスペクトの影響力を制限してしまうというものであった。しかし、今の段階で AspectJ と同等の処理をすることはできない。

どちらの研究もオブジェクト指向にあった”Black-Box”の性質をアスペクト指向で実現するためにどうすればよいかという考えに基づくものである。しかし、前述のようにこれらの言語設計、仕様は不十分であるといえる。

2.5　AJDT(AspectJ Development Tools)

AJDT（AspectJ Development Tools）[3, 4] は AspectJ 開発を可能にするための Eclipse [6] のプラグインである。Eclipse は統合開発環境のひとつでオープンソースのソフトウェアである。AJDT には AspectJ 開発を支援するためのツールが数多く用意されている。

Eclipse とは

Eclipse は統合開発環境（IDE:Integrated Development Environment）である。統合開発環境とは、プロジェクト管理機能やコード・エディタ, デバッガなどアプリケーション開発に必要な機能を備えたソフトウエアのことである。アプリケーション開発において統合開発環境は開発の手間を大幅に削減することができる重要な要素である。詳細については、次章で述べる。

2.5.1　AJDT の提供するツール

アスペクト指向プログラミングの開発効率を十分に上げるためには、開発を補完するためのツールが必要不可欠となるというのが一般的な見解である。AJDT の場合には、前述のように AspectJ の obliviousness [10] の性質を支援するツールが数多く存在する。プログラマがソースコードのどの部分に織り込みがあるかを直感的に見て分かるようになっている。

例えば、下図のように、AJDT をインストールした Eclipse エディタには、ソースコードのどの部分にアドバイスが実行されるかが分かるように、該当箇所にはエディタ内左側にある縦ルーラーにマーカーがつくようになっている。このマーカーを「右クリック」するとポップアップメニューが表示される。このメニューの最後の項目に「advised by」というメニューがある。このメニューの上にマウスを合わせると新しいポップアップメニューが「advised by」メニューのサブメニューとして表示される。そのメニュー項目には、実行されるアドバイスに関する情報が載せてある。それは、どのアスペクトに定義されてどのポイントカットにより指定されたアドバイスかを示すものである。また、このサブメニュー項目をクリックして選択することで対応するアドバイスが記述された行へジャンプすることができるようになっている。

また、アスペクトの側にも同じような機能がある。エディタのルーラー上のマーカーをクリックすると、「advises」というポップアップメニュー項目を参照することができる。このメニューのサブメニューにはクラス側

のどの箇所でアドバイスを実行するかを示すものである。同様に、これらを選択することでクラス側の該当行へジャンプすることができる。

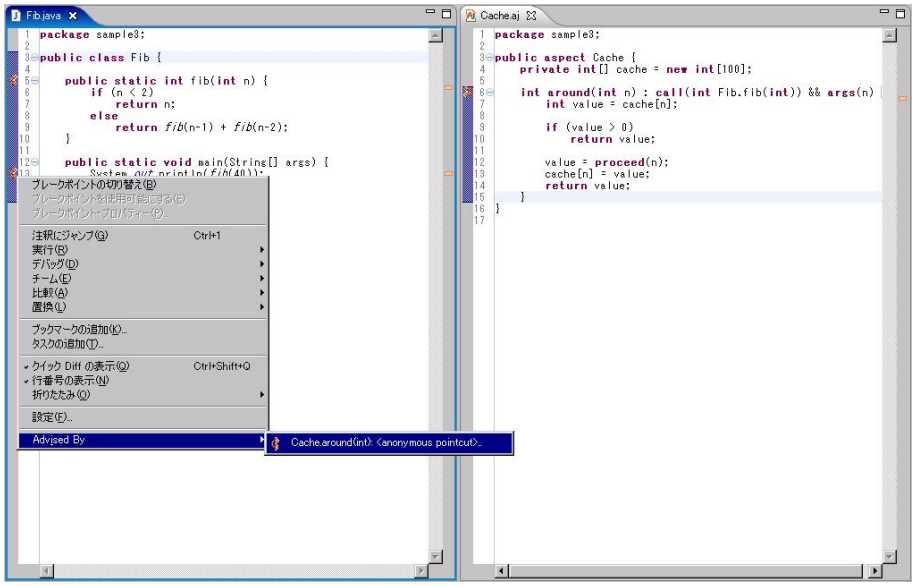


図 2.2: Eclipse エディタ

しかし、このマーカー付きのエディタを見ただけでは、どのアスペクトのアドバイスが実行されるのかという情報はユーザーがポップアップメニューを積極的に表示させなければ分からない。また、エディタを見ただけではプロジェクト内にあるアスペクト全体の織り込みの状況を把握することは不可能である。そのため、AJDT にはその他にもアスペクト指向プログラミングによる開発に役立つツールが用意されている。

Visualiser

Visualiser ビューは「ウィンドウ」→「ビューの表示」→「その他」→「Visualiser」で起動できる。Visualiser はパッケージエクスプローラーでパッケージを選択したときに、そのパッケージ内にあるファイルに対してアスペクトの影響が表示されるようになっている。クラスごとに縦の棒グラフのように表示され、その内部に横方向に色のついた線が入っている。縦方向はソースコードの相対的な長さを表していて、横線の入っている場所はソースコード中でアドバイスが実行される箇所を表している。異なるアスペクトは色の違いで区別されていて、どのアスペクトのアドバイスも実行されない場合にはクラス内の要素は黒色表示になる。アスペクトは常に黒色表示になる。クラス内の横線をクリックすることでソースコード中

の該当行へジャンプすることができる。Visualiser Menu ビューは「ウィンドウ」→「ビューの表示」→「その他」→「Visualiser Menu」で起動できる。もしくは Visaliser を起動すると一緒に起動される。このビューを使ってアスペクトの色分けをユーザーが設定できるようになっている。

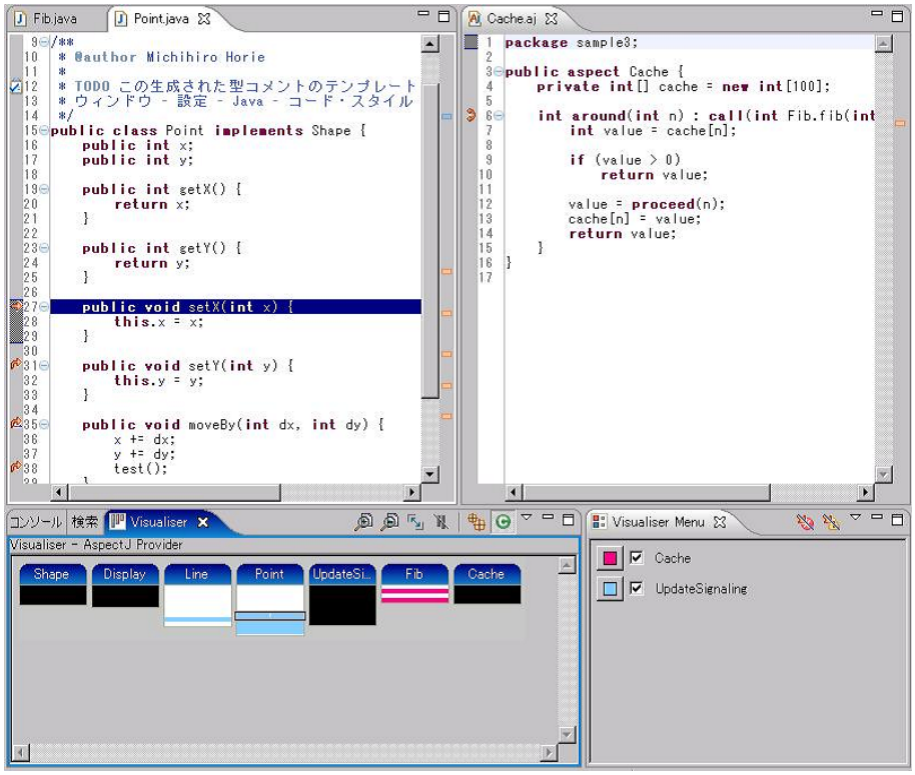


図 2.3: Visualiser、Visualiser Menu ビュー

このビューにより、アスペクトが全体としてどの箇所に織り込まれるか、どのアスペクトが織り込まれるかという全体像は把握できるが、それ以上の情報が欲しい場合には、縦の棒グラフ中の行をクリックしてエディタにジャンプしなければならない。

Outline

Outline ビューは「ウィンドウ」→「ビューの表示」→「アウトライン」で起動できる。アウトラインビューではポイントカット名、アドバイスの種類が列挙されるだけなので、どのアドバイスでどのポイントカットが利用されているかなどの情報が分からない。このビューも各要素をクリックすることでソースコード中の該当行にジャンプすることができるようになっている。また、outline ビューは現在アクティブなファイルの内容を

表示する。

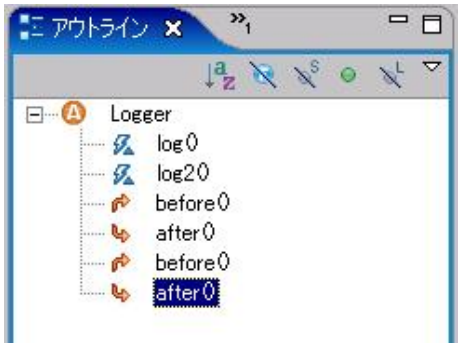


図 2.4: アウトラインビュー

Cross-References

Cross-References ビューは「ウィンドウ」→「ビューの表示」→「その他」→「AspectJ」→「cross-References」で起動できる。outline ビューと同じように現在アクティブなファイルの内容を表示するようになっている。プログラマは Cross-References ビューを利用して、どのアドバイスがどこで実行されるかの情報を得ることができる。

例えば、あるアスペクトのソースコードをエディタで開くと、そのアスペクト内に記述されたアドバイスの情報がビューを通して図 1.4 のように表示される。1 つ目の before アドバイスは Caller クラス内の Worker#work メソッドの呼び出し時に実行されることが分かる。次に、「Caller:method-call(void Worker.work(int))」の要素をクリックすると Caller クラス内の該当行へジャンプする。すると、アクティブなファイルが Caller.java に変わったから、今度は Cross-References ビューの内容は図 1.5 のように変わる。この図から、Caller コンストラクタ内の Worker#work メソッド呼び出し時にアドバイスが実行されることが分かる。ポイントカット名は「log..」と書いてあることから分かるが、これ以上の情報はソースコードにジャンプして確認しなければならない。

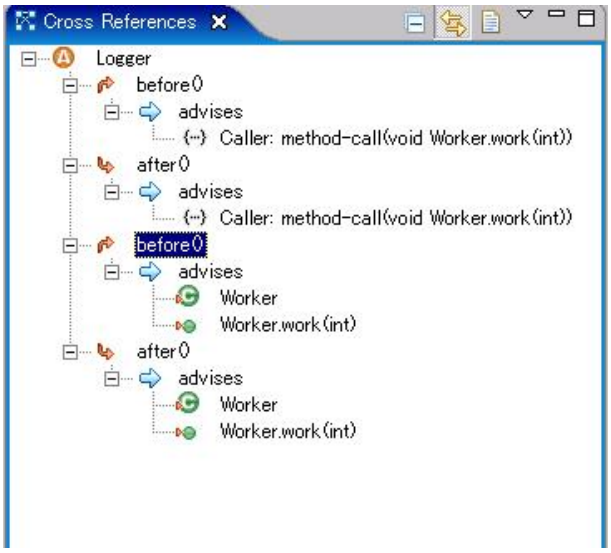


図 2.5: Cross-Reference ビュー

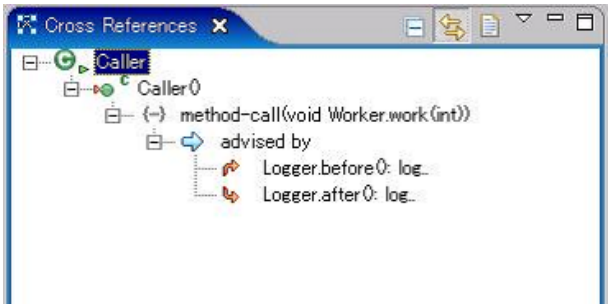


図 2.6: Cross-Reference ビュー

2.5.2　AJDT の提供するツールの問題点

これまで、AJDT をインストールすることで利用できるツールについて述べてきた。プログラマはこれらのツールを活用してアスペクト指向プログラミング開発をスムーズに行うことができるようになるが、これらは織り込みに関する情報を視覚的、直感的に分かりやすいように表示しただけである。アスペクト指向言語で Modular-Reasoning が可能なことを示しているわけではない。それは、アドバイスが実行されるということによりモジュールが拡張されるという考えに基づいていないためである。本システムでは、モジュール拡張の視点に立ったアウトラインビューアを提案する。

第3章　設計と実装

本システムはGregor氏らの論文[9]で発表されたユーザーインタフェース Aspect-Aware Interfaceを元に、アスペクトによるモジュール拡張を強調したアウトラインビューアをEclipseプラグインとして開発した。

アスペクト指向とは既存のモジュールに新しい境界を作るものだから、アスペクト指向にはそれを定義する新しいインターフェイスが存在するはずであるという主張の元に考えられたのがAspect-Aware Interfaceである。また、Aspect-Aware Interfaceは、AspectJなどのアスペクト指向言語ではModular-Reasoningが可能ではないという見解を否定するものである。

Modular-Reasoningとはモジュールの性質をそれを見ただけで理解できるかどうかという意味であった。横断的関心事をひとつのモジュールにまとめることで、クラス側にはその関心事に関する情報は記述されない。そのため、一般的にはアスペクト指向言語ではModular-Reasoningが可能ではないという意見がある。そして、前述のように、アスペクト指向言語でModular-Reasoningを可能にするためには、アスペクトの影響力を抑えるなどの言語機構を導入しなければならなかった。

この論文では、Aspect-Aware Interfaceのようなインターフェイスを利用することにより、アスペクト指向言語の能力を損なうことなくModular-Reasoningが可能になると述べている。具体的には、モジュールがどのアドバイスからの影響を受けるかということを示したユーザーインタフェイスを下図のように定義する。下図はexecutionポイントカットでafter returningアドバイスが実行されるときに、影響を受けるモジュールについてリストアップしている様子である。

```
Shape
 void moveBy(int, int) : UpdateSignaling - after returning UpdateSignaling.move();
Point implements Shape
 int x;
 int y;
 int getX();
 int getY();
 void setX(int)        : UpdateSignaling - after returning UpdateSignaling.move();
 void setY(int)        : UpdateSignaling - after reutrning UpdateSignaling.move();
 void moveBy(int, int) : UpdateSignaling - after returning UpdateSignaling.move();
Line implements Shape
```

```
 void moveBy(int, int) : UpdateSignaling - after returning UpdateSignaling.move();
UpdateSignaling
 after returning : UpdateSignaling.move() : Point.setX(int), Point.setY(int),
                                            Point.moveBy(int, int),
                                            Line.moveBy(int, int);
```

execution ポイントカットとは、メソッド本体の実行が始まってから終了するまでの間の一連の処理のことである。したがって、例えば setX(int) メソッドに execution ポイントカットによって after returning アドバイスが実行されるという挙動を上のように記述するのは理解しやすい。しかし、call ポイントカットの場合には、そのジョインポイントを含むメソッドをリストするのか、呼び出し時にアドバイスが実行されるメソッドをリストするかなどの詳しい議論は今後の課題としている。

この論文では、Aspect-Aware Interface を利用すればアスペクト指向言語でも Modular-Reasoning が可能であることを適当な例を使って示している。上の例の実際のコードの一部は下のようなものである。

```
class Point implements Shape {
 int x, y;
 public int getX() {return x;}
 public int getY() {return y;}
 public void setX(int x) {this.x = x}
 public void setY(int y) {this.y = y}
 public void moveBy(int dx, int dy) {
  x += dy; y += dy;
 }
}

class Line implements Shape {
 private Point p1, p2;
 ...
 public void movbeBy(int dx, int dy) {
  p1.x += dx; p1.y += dy;
  p2.x += dx; p2.y += dy;
 }
}

aspect UpdateSignlaing() {
 pointcut move() : {
   execution(void Shape+.moveBy(int, int));
 after() returning : move() {
   Display.update():
 }
}
```

ここで、リファクタリングの際に Point クラスのフィールド int 型 x、y の修飾子を private に変更した場合を考える。Line クラスの moveBy メソッドを次のように書き換える必要が出てくる。

```
public void moveBy(int dx, int dy) {
```

```
  p1.setX(p1.getX() + dx); p1.setY(p1.getY() + dy);
  p2.setX(p2.getX() + dx); p2.setY(p1.getY() + dy);
}
```

このとき、Aspect-Aware Inferface を見れば setX というモジュールと moveBy というモジュールには同じアドバイスが実行されるということが分かる。Cross-Reference のような従来のツールではアスペクトの織り込みがどこで実行されるかという情報を提供するだけであったが、この Aspect-Aware Interface の場合は、メソッドなどをきちんとモジュールと見なせるように moveBy メソッドの中で呼ばれるメソッドに対してアドバイスが実行されるような状況でも moveBy の中で織り込みの情報を提示するのではなく、呼ばれるメソッド自体にその情報を付すことにしている。execution ポイントカットでは、Cross-Reference との違いが分かりにくいが、call ポイントカットのときは明らかである。大きな違いは、あるモジュール M の内部にある他のモジュール M1 にアスペクトが挿入されるときに、どこにその情報を載せるかということである。Cross-Reference の場合は M の中にその情報を記述するが、Aspect-Aware Interface の場合には、M1 の方に記述するというものである。そうすることで、アスペクト指向言語は既存のモジュールを拡張するものであるという考えにより近づくことになる。

3.1　仕様

本研究では、Aspect-Aware Interface の考えに基づいたユーザーインターフェイスを Eclipse プラグインのビューとして開発することにした。アスペクト指向言語には AspectJ を用いている。開発したビューは 2 つのウィンドウから構成されている。左側はプロジェクトのアウトラインを示すウィンドウで、右側半分は Javadoc コメントを表示するためのものである。

3.1.1　要素の階層表示

クラスやメソッドなどの表示方法はパッケージエクスプローラーやアウトラインビューのように階層的なものである。別のフォルダにあるクラスへアスペクトが影響を及ぼすことがあるので、ここでは 1 つのプロジェクト以下の要素を全て表示している。つまり、プロジェクトの下には複数のフォルダがあり、その下に Java ファイルや AspectJ ファイル、またはフォルダが存在することを想定している。

なお、表示されている内容をダブルクリックすることで、それが記述されているソースファイルを開きその行にジャンプすることができる。

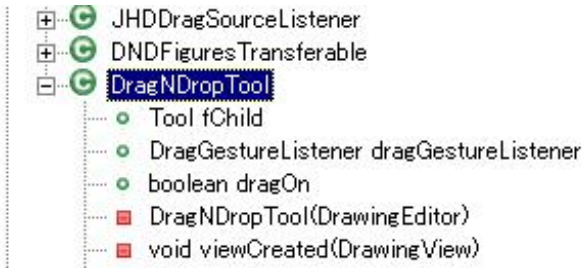


図 3.1: コンテンツの階層表示

3.1.2　アドバイスによって拡張されるモジュールの表示

アドバイスが実行されるモジュールには色をつけて、一目でそれが分かるようにしてある。また、そのモジュールは”extended by advice ...”という子を木の要素として持つ。これはアドバイスによって拡張されていることを意味する要素である。色をつけることで、そのモジュールが拡張されるということは分かるが、それ以上のことは分からない。そこで、この子の要素以下でより詳細な情報を提示している。

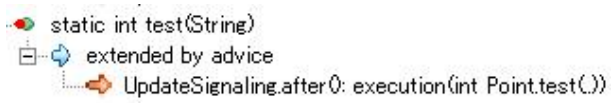


図 3.2: 拡張されたモジュールの表示

アドバイスの存在を明示する要素”extended by advice ...”は実行されるアドバイスの数や種類によってその表示形式が異なる。現段階では、下の３種類の場合がある。

1. **extended by advice**
   アドバイスは１つ

2. **extended by advices**
   アドバイスは複数

3. **extended by advice only if the Caller is 〜**
   呼び出し式が〜の場合に限ってアドバイスが実行される場合。within や withinCode ポイントカットを使ってアドバイス実行時の場所が指定されているとき。

上で述べた要素の子、つまり階層のひとつ下に、実際にどのアドバイスが実行されるかの情報が付けられる。１と２がひとつのモジュールで併用されることはないが、１または２が３と併用されて表示されることは考えられる。within や withincode を用いて指定されたアドバイスと、そうでない別のアドバイスが同じモジュールをそれぞれ拡張することがあるからである。

上の１、２、３のひとつ下の階層で表示されるアドバイスの表示内容の形式は、

アスペクト名. アドバイスの種類　：　ポイントカット名

となっている。「ポイントカット名」の部分は名前つきポイントカットの場合にのみ表示することにした。例えばアスペクト内で宣言されたポイントカットの宣言が、

```
aspect Logger {
 pointcut log() : call(void Point.setX(int));
 after () : log() { ...}
 ...
}
```

というように名前付きであれば、「Logger.after():log()」と表示する。プリミティブポイントカットや、名前付きポイントカットでも「&&」や「||」といったオペレータを用いて構成されたポイントカットの場合には、「ポイントカット名」の部分は単純に「...」としてある。オペレータ「！」の場合には、名前付きポイントカットど同様に扱うことにした。

例えば上の例のアドバイスの宣言部分が次のような場合だったときには、「Logger.after:...」と表示する。

```
 after () : log() && within(Line) { ...}
```

名前付きポイントカットを表示しているのは、拡張されるモジュールの側からは、「ポイントカット名」がその内部の実装とのインターフェイスの役割をしていると考えたためである。名前付きポイントカットの内部の実装とは上の例では「call(void Point.setX(int))」の部分である。これはアスペクト内部の実装であり、外部のモジュールからは不可視である必要があると考えた。そこで、「log() && within(Line)」のような場合には、「...」とする代わりに「only if the Caller is Line」という情報が付加されるのである。

3.1.3　call ポイントカットと execution ポイントカット

本システムでは、call ポイントカットと execution ポイントカットを反映する表示上の区別がない。つまり、アドバイスがどちらのポイントカットにより指定されてモジュールを拡張する場合も、ビューの表示からそれを区別することはない。どちらのポイントカットにしろ、ある特定のメソッドに関する拡張を意味していることに変わりはないと考えるからである。まず、それぞれの定義を見てみる。

call　call ポイントカットは呼び出し式によってメソッドが呼び出されてから、制御が移った後、元の呼び出し側に戻ってくるまでの一連の処理を表す時点をジョインポイントとして選択する。

execution　execution ポイントカットは制御がメソッド本体に移ってから終了するまでの処理を表す時点をジョインポイントとして選択する。

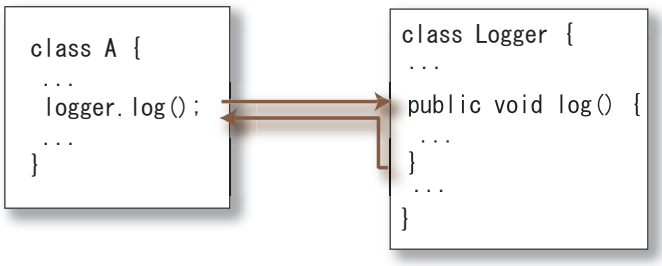


図 3.3: メソッド呼び出しの制御の流れ

この定義から分かるように、２つのポイントカットには実装上の違いがある。これにより、例えば within ポイントカットなどと組み合わせればもちろん異なるジョインポイントを選択することができる。しかし、本質的には注目しているモジュールはひとつだけであり、それを実装上の都合により２種類に分けているということがいえる。

そこで、本システムでは call ポイントカットで指定されたアドバイスによりあるモジュールが拡張されるときには、拡張を意味する情報をそのモジュールの呼び出し元ではなくモジュール自身に付けている。そうすることにより、既存のモジュールがアドバイスにより拡張されるということがより直感的に分かるようになる。これが、AJDT エディタのルーラー上でつくマーカー表示の意味するものと大きく異なる点である。マーカーは織り込みの箇所を単に表示しているだけである。

結果的に、これは execution ポイントカットで指定されたアドバイスを実行する場合と表示が同じになる。

＜表示が統一される例＞
次のように call ポイントカットと execution ポイントカットの両方で指定されたアドバイスがあったとき、AJDT のマーカーはそれぞれの場所にマーキングをする。つまり、メソッド呼び出し時とメソッド実行時を示す場所にそれぞれマーカーが付けられる。

```
aspect Logger {
 pointcut test() : call(void Test.test())
                   || execution(void Test.test());
 before() : test() {...}
}
public class Test {
  ...
  test();          //この行に call ポイントカットによる指定を示す
                   //マーカーがつく
  ...
  public void test() {    //この行に execution ポイントカットに
                          //よる指定を示すマーカーがつく
   ...
  }
  ...
}
```

本システムの場合には、前述の考えによりアドバイスをひとつにまとめた下のような表記になる。

図 3.4: Test クラス階層の一部

3.1.4　Javadoc コメントの表示

ビューの右側半分は Javadoc コメントを出力するためのものである。アドバイスにより拡張されたモジュールの Javadoc を表示させると、一緒に

そのアドバイスの Javadoc も表示されるようになっている。２つのドキュメントを１つのウィンドウで見せることにより、関連のあるそれぞれのモジュールの仕様を理解するのが容易になる。メソッドに書かれた Javadoc コメントを見ただけでは、そのメソッドの仕様しか分からない。もちろん、一般的にそこにアドバイスが実行されるかどうかの情報やアドバイスによりどのようなことが実行されるかということが書かれていることはない。また、アスペクトの側でアドバイスの Javadoc コメントを見ただけでは、やはりその仕様しか分からない。あるモジュールの Javadoc に加えて関連のあるアスペクトも同時に表示することで、そのモジュールがアドバイスにより拡張されているという観点から見ることができる。なお、Javadoc コメントは左側のビューコンテンツを選択し、右クリックでポップアップメニューを表示し、「Javadoc を表示」メニューを選択することによりビューの右側に表示される。

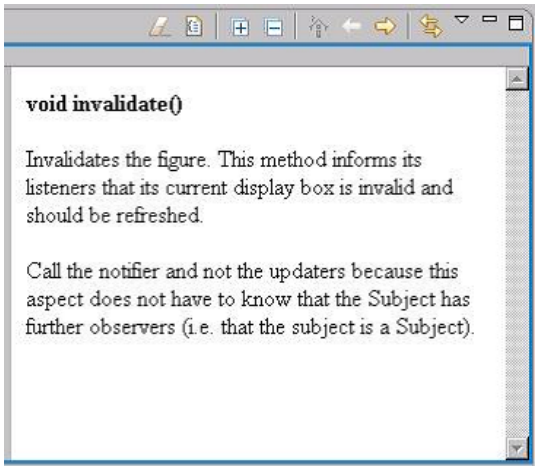


図 3.5: Javadoc コメントの表示

3.2　実装

3.2.1　Eclipse プラットフォーム

アーキテクチャ

Eclipse プラットフォームは、プラグインの概念により構造化されている。プラグインとは、新機能を定義するために中心となる構造化されたメカニズムである。プラグインは Eclipse が認識する最小単位の機能であり、Eclipse アーキテクチャではプラグインの最小サイズ、最大サイズと

もに指定していない。また、プラグインは拡張ポイントと呼ばれるコントリビューションメカニズムを使って新しい機能を取り入れる。拡張ポイントはプラグインのマニフェスト（plugin.xml）ファイルの中で宣言することで定義する。

Eclipse SDK には、基本プラットフォームとプラグイン開発に役立つ主要なツールが組み込まれている。Java 開発ツール（JDT）プラグインは、Java コードの編集、表示、コンパイル、デバックなどを提供する。JDT は SDK に組み込まれるプラグインのセットとしてインストールされる。プラグイン開発環境（PDE）は、プラグインの作成、操作、デバッグ、配置を自動化するツールを提供する。PDE を用いて拡張の開発を簡略化することができる。これも SDK に組み込まれるプラグインのセットとしてインストールされる。

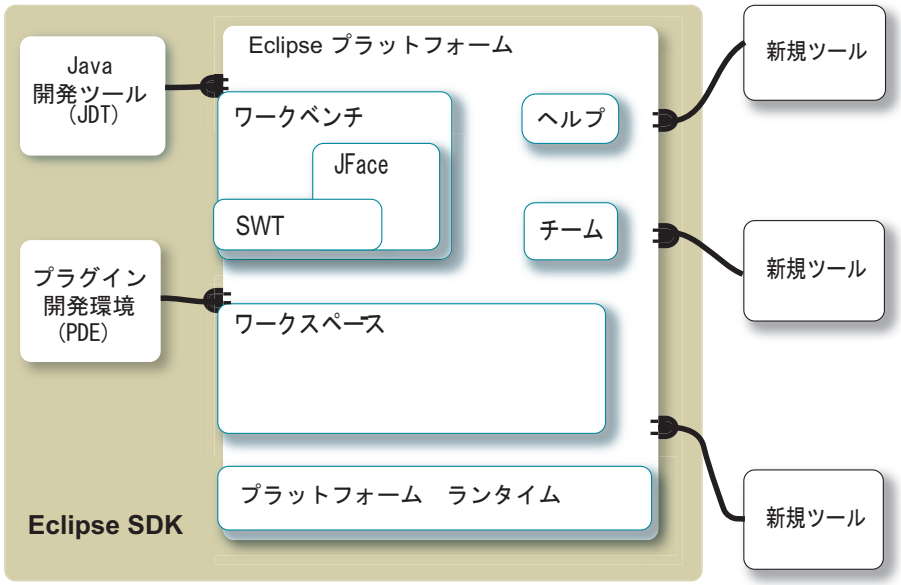


図 3.6: Eclipse のアーキテクチャ

Eclipse ランタイム

Eclipse の全機能はプラグインによって与えられていることを述べたが、Eclipse プラットフォームランタイムは例外である。これはプラグインが依存する他のプラグインや、プラグインの構造、実装の詳細の管理をするカーネルである。下図ではプラットフォームランタイムがプラグインの共通の基盤であることを表している。プラットフォームランタイムには、ロギング、デバッグトレース、アダプターなどの各種ユーティリティが用意

されている。プラットフォームランタイムはプラグインヒープの一番下に常駐し、Eclipse 全体を支えている。

ワークベンチ

ワークベンチは、エディタ、ビュー、パースペクティブなど複雑なユーザーインタフェースを構成する環境である。ワークベンチはワークベンチ・ウィンドウ（IWorkbenchWindow）のことを指す。ワークベンチ・ウィンドウは、ワークベンチにおけるトップレベルのウィンドウで、メニューバー、ツールバー、ステータスバー、ショートカットバー、およびページを含むフレームである。ワークベンチ・ウィンドウは１つのページを持つ。ページは各パーツをグループ化するためのメカニズムを持っている。パースペクティブはページ内の編集レイヤーを提供し、ビューや適当なアクション、またそのレイアウトを定義する。また、エディタはパースペクティブの直接の影響を受けることはない。ビューは、各種情報のナビゲーションやエディタのオープン、アクティブエディタのプロパティ表示などに使用される。エディターは、文章や入力オブジェクトの編集、ブラウザとして使用される。

ワークベンチは、プラグインが既存のビューおよびエディタへの振る舞いを提示、または新規のビューおよびエディタの実装を提供できる拡張ポイントを定義する。

org.eclipse.ui.views　この拡張ポイントを使用することで、プラグインはビューをワークベンチへ追加することができるようになる。プラグインは pluin.xml ファイルにビューを登録して、その上でビューの動作を定義するクラス、クラスが属するビューのカテゴリ、メニューおよびラベル上で使用する名前、アイコンなど構成情報を定義する必要がある。

org.eclipse.ui.viewActions　この拡張ポイントを使用することで、プラグインはワークベンチ内にある既存のビューへ新たなアクションを組み込むことができるようになる。具体的には、メニュー項目、サブメニュー、ツールバー項目を既存のビューのプルダウンメニューとツールバーに組み込むことができる。

org.eclipse.ui.editors　ワークベンチに新たに定義するエディタを追加したいときにはこの拡張ポイントが利用できる。plugin.xml マニフェストに記述する内容はビューと類似しており、エディタの動作を定義するクラス、メニューやラベルで使用される名前、アイコンなどである。それに加

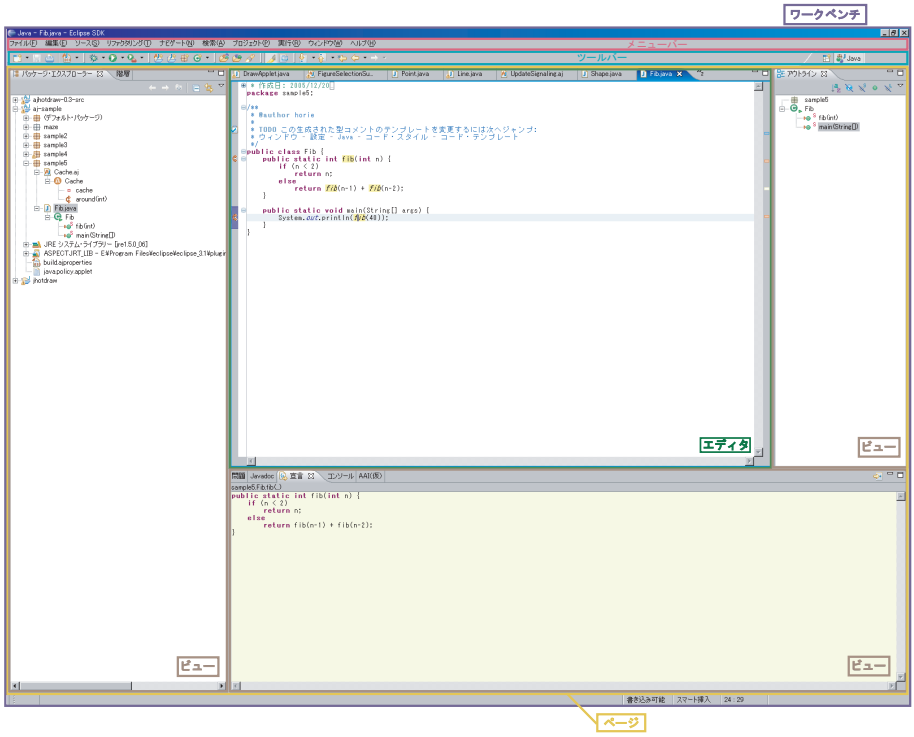


図 3.7: workbench

えて、エディタを起動する対象となるファイルの拡張子、もしくはファイルタイプのパターンを指定する。

org.eclipse.ui.editorActions　既存のエディタに対して、ワークベンチのメニューバー、ツールバーにアクションを追加をしたい場合、この拡張ポイントを利用する。

org.eclipse.ui.popupMenus　エディタやビュー内で表示されるポップアップメニューを組み込むときにこの拡張ポイントを使用する。ポップアップメニューは、特定のタイプのオブジェクトが選択された場合、指定されたビューやエディタ内で選択された場合にメニュー項目を表示する。

org.eclipse.ui.acitonSets　この拡張ポイントを利用することで、メニュー、メニュー項目、ツールバー項目をワークベンチメニューおよびツールバーに組み込むことができる。

3.2.2　プラグインマニフェスト

プラグインマニフェストはプラグイン開発の際に最初に作成しなければならないものである。plugin.xml というファイル名で、XML を用いて記述するこのファイル内に、どこで他からコントリビューションを受け取り、どこで他にコントリビュートするかを宣言する。Eclipse 起動時、Eclipse プラットフォームランタイムは最初に、どのプラグインが使用可能かを見つけるが、plugin.xml という名前のファイルについて検索している。そして、各ファイルを解析し、依存性を調べ、どのコードがプラグインを構成し、どのコードがプラグインを構成するエクステンションとそれが定義するエクステンションポイントを構成するかを調べるのである。

一般的に、プラグインは複数のファイルから構成されるが、少なくとも plugin.xml マニフェストは含んでいなければならない。また、ほとんどのプラグインは Java で記述された実行可能コードを含み、プラグインがその機能を実行するのに必要な他のリソースを含んでいる。

プラグインマニフェストはまず以下のように記述する。

```
<plugin
   id="helloworld"
   name="HelloWorld プラグイン"
   version="1.0.0"
   provider-name=""
   class="helloworld.HelloWorldPlugin">
```

id 属性は、名前空間の衝突を避けるために Java 命名規約を利用して指定する。これはプラグイン自身のプログラムからの参照を定義している。name 属性は Plug-in Registry ビューおよびダイアログ内で表示されるラベルである。次に、requires 節では依存する独立したプラグインを列挙する。

```
<requires>
    <import plugin="org.eclipse.core.runtime"/>
    <import plugin="org.eclipse.core.resources"/>
    <import plugin="org.eclipse.ui"/>
    ...
</requires>
```

runtime 節はプラグインランタイムコードを定義する 1 つまたは複数のライブラリ（JAR ファイル）を指定する。export 句ではどのパッケージまたはクラスがプラグインの外部から可視性を持っているかを定義することができる。つまり、自分のプラグインには公開されても、他のプラグインからは見えないクラスを定義できる。下のように「name=”*”」と書いた

場合には、プラグインクラス内で定義された全てのパブリッククラスが可視化されていることを示している。

```
<runtime>
      <library name="HelloWorld.jar">
         <export name="*"/>
      </library>
</runtime>
```

これらの記述をした上で、利用する拡張ポイントの定義を書いていくことにより独自のプラグインを開発することが可能となる。例えば、独自のビュープラグインを作成する場合には、先ほど述べたように org.eclipse.ui.editors 拡張ポイントを以下のように利用する。

```
<extension
    point="org.eclipse.ui.editors">
     <editor
           name="HelloWorld エディター"
           extensions="hw"
           icon="icons/helloworld.gif"
           contributorClass="org.eclipse.ui.texteditor.
                             BasicTextEditorActionContributor"
           class="helloWorld.editors.helloWorldEditor"
           id="helloWorld.editors.helloWorldEditor">
     </editor>
</extension>
```

マニフェスト・エディタ

プラグイン・プロジェクトを作成すると、マニフェスト・ファイルがプラグインのマニフェスト・エディタで開かれる。このマルチページ・エディタはプラグインを管理するためのツールであり、プラグイン開発に必要なファイルを編集するために使用する。plugin.xml を編集する際には、このマルチページを用いると便利である。各ページの編集、変更は保存しなくても、即ソースページの plugin.xml ファイルに反映される。また、plugin.xml ファイルを直接編集した場合も即それに合わせて各ページの情報が書き換えられる。以下に、マルチページ内の主なページについてその役割を説明する。

概要　「概要」ページはプラグインを作成、テスト、デプロイする方法を素早く参照できるようになっている。例えば、「一般情報」では、plugin.xml の¡plugin¿節を作成するために使用される。

依存関係　「依存関係」ページでは、作成しているプラグインを操作するために必要な他のプラグインをリスト内に記述する。リスト内のプラグインの順番によってランタイムのクラスロードの順番が変わってくる。

ランタイム　このページでは、プラグインが他のプラグインから可視になるすべてのパッケージが表示される。

拡張　「拡張」ページは plugin.xml の¡extension point¿節を編集するためのもので、プラグインの仕様を決める上で重要なページである。本システムでは、ビュープラグインを利用して情報を表示している。

拡張ポイント　拡張ポイントとは、他のプラグインを接続できる、プラットフォームに対する新しい機能を定義するものである。このページでは、作成しているプラグインで宣言される拡張ポイントを追加、除去、編集できる。

ビルド構成　このページで編集した内容は、build.properties ファイルに書き込まれる。ここでは、プラグインをビルド、パッケージ、エクスポートするために必要な情報が含まれている。「ランタイム情報」では、ビルド対象のすべてのライブラリそれぞれに対して、それを生成するためのコンパイル対象となるソースフォルダを指定する必要がある。

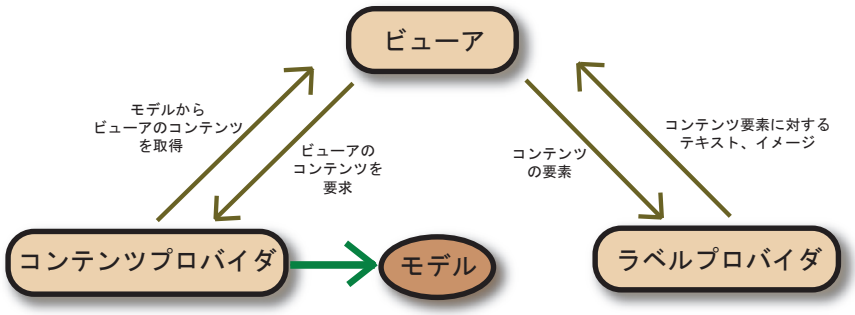


図 3.8: ビューアの構成要素

3.2.3　Java エレメント

Java エレメントとは、メソッドやクラスなど Java プロジェクトを構成する要素のことを指す。各種の Java エレメントは、JDT の IJavaElement インタフェイスのサブインターフェイスである。Java クラスやプロジェクトなどに対する操作を扱うプラグインは、JDT を使用することにより簡単に Java エレメントに対するを操作を行うことができるようになる。

3.2.4　TreeViewer の利用

本システムでは、クラスやメソッドなどの各要素を階層的に表示するために JFace のビューアクラスの一つである TreeViewer を利用している。ビューを構成する要素には、モデル、ビューア（Viewer）、コンテンツプロバイダ（ContentProvider）、ラベルプロバイダ（LabelProvider）が必要である。ビューアは今の場合 TreeViewer であるが、その他にも SourceViewer や TextViewer、TableViewer などがある。これは表示形式を指定するためのクラスである。モデルは開発者がビューに表示したいデータの内容そのものを指す。例えば、Java 要素であったり開発者が独自に定義したオブジェクトである場合もある。コンテンツプロバイダは、モデルからビューアのコンテンツを取得する方法を定義する。コンテンツプロバイダはビューアとモデル間を仲介し、ビューアの更新をトリガする。また、モデルが変更されたときにビューアに更新する必要があることを通知する。ラベルプロバイダは、ビューアが制御する各要素に文字列とアイコンを関連付ける役割を持つ。以下にビューを構成する 4 つの要素の関係図を示した。

エレメント	説明
IJavaModel	ワークスペースルートを表す Java エレメント。Java ネイチャーを持つ全てのプロジェクトの親である。
IJavaProject	Java プロジェクトを表す。（IJavaModel の子）
IPackageFragmentRoot	パッケージフラグメントのセットを表し、それをフォルダ、JAR、Zip ファイルのいづれかである基本リソースにマップする。（IJavaProject の子）
IPackageFragment	パッケージ全体に対応するワークスペースの一部、またはパッケージの一部を表す。（IPackageFragmentRoot の子）
ICompilationUnit	Java ソースファイルを表す。（IPackageFragment の子）
IPackageDeclaration	コンパイル単位ないのパッケージ宣言を表す。（ICompilationUnit の子）
IImportConrainer	コンパイル単位内のパッケージのインポート宣言のコレクションを表す。（IImportContainer の子）
IImportDeclaration	１つのパッケージのインポート宣言を表す。（IImportContainer の子）
IType	コンパイル単位のソースタイプ、クラスファイル（.class）内のバイナリー形式のいづれかを表す。
IField	型内部のフィールドを表す（IType の子）
IMethod	型内部のメソッド、コンストラクタを現す。(IType の子)
IClassFile	バイナリーの型を表す。（IPackageFragment の子）
IInitializer	型内部の静的イニシャライザ、インスタンスのイニシャライザのいづれかを表す（IType の子）

表 3.1: 各種の Java エレメント

フィールド名	意味
FIELD	フィールド
METHOD	メソッド、コンストラクタ
LOCAL_VARIABLE	ローカル変数 （メソッド呼び出しもこれに含まれる）
TYPE	クラス、インタフェース

表 3.2: Java エレメントの判別

本システムでは、現在ワークベンチ内でアクティブな状態にあるファイルのプロジェクトに対して、その中にある全ファイル中の Java エレメントをモデルとして取得している。さらに、コンテンツプロバイダでそのモデルを基に、ビューに表示するためのコンテンツに編集しなおしている。具体的には次節で述べる AJDT の marker に従って、各 Java 要素に新たに要素を追加している。追加の手順の詳細は次節で述べる。コンテンツの各要素は TreeObject 型、またはそのサブクラスの TreeParent 型のオブジェクトである。TreeParent オブジェクトは任意個の TreeParent オブジェクト、または TreeObject オブジェクトを保持することができ、プロジェクトを root としてツリー構造を構成する。インスタンスフィールドには、IJavaElement 型のインスタンス、要素名を保持する String 型の文字列、ツリーの親のインスタンスを保持する。Treeobject は IAdaptable インタフェースを実装している。

```
public class TreeObject implements IAdaptable {
  private IJavaElement element = null;
  private TreeObject parent;
  private String name;
  ...
}
```

ラベルプロバイダではコンテンツの各要素であるオブジェクトを取得後、単純にフィールド name を返すだけである。また、IJavaElement 型 element のフィールド値にしたがって、どのアイコンイメージを返すかを選択している。IJavaElement には、その特性を表すフィールドが十数種類存在するが、そのうち利用しているのは以下の 4 種類である。

コンテンツに利用する他の要素は Java エレメントではないので element

フィールドの値は null である。そのため、アイコンの決定には name フィールドや parent フィールドを使って判定している。アイコンイメージは ImageDescriptor クラスを用いて以下のように取得する。

```
//Plugin クラスのインスタンスを取得
ViewPlugin plugin = ViewPlugin.getDefault();
//gif ファイルがある場所を示す URL オブジェクトを得る
URL url = plugin.find(new Path("icons/a.gif"));
//createFromURL クラスメソッドにイメージの場所を示す URL オブジェク
//トを渡すことで ImageDescriptor オブジェクトを得る
return ImageDescriptor.createFromURL(url);
```

ImageDescriptor はイメージリソースを扱うためのクラスで、イメージを直接扱うのではなくイメージファイルの位置を特定するための情報を持ち、必要に応じてその場所からリソースを確保する。実際にイメージを作成したい場合には ImageDescriptor#createImage() メソッドを呼び出す。このメソッドは SWT の Image クラスを返す。ただし、この場合には Image クラスのオブジェクトに対して dispose() メソッドを呼び出してリソースを明示的に開放する必要が出てくる。

3.2.5　AJDT から取得するアドバイスの情報

本研究では、ビューのコンテンツの各要素を表示する際、アドバイスによって拡張される要素があった場合にどのアスペクトのアドバイスが実行されるのかという情報をその要素に追加表示する。具体的には、アドバイスによって拡張される要素以下に「expanded by advice ...」という文字列を持ったオブジェクトを持たせ、さらに、その子オブジェクトにアドバイスの情報（例えば、「advised by Logger.log()...」など）を持った子オブジェクトを持たせるという構造になっている。このとき最下位の子オブジェクトが保持するアドバイスに関する情報は AJDT から取得している。

以下のコードは AJDT からアドバイスの詳細な情報を取り出すためのものである。JavaCore クラスは各種のリソースから JDT の Java エレメントを生成したいときに便利なクラスである。ここでは、JavaCore のクラスメソッドである createCompilationUnit を使用して Java エレメントの ICompilaitonUnit を生成している。このメソッドは引数に IFile 型のインスタンスを取る。getJavaElements は、引数の ICompilationUnit からそのメンバーである IJavaElement 型の要素を持つ List を返すメソッドである。for 文の内部では、それぞれの Java エレメントに対して、アド

バイスが実行されるかどうかの情報を持っているかを検査する。クラス AJModel のメソッド getRelatedElements によってさまざまな情報を取り出すことができるが、このメソッドの第一引数に取り出したい情報の型を書く。情報の型とは、アドバイスが実行される場所なのか、またはアドバイスを実行するコードが書かれた場所なのかといった情報の種類のことである。他にも、アスペクトを宣言している場所を意味する型などがあり、全部で１２種類ある。今の場合、アドバイスがコードのどこで実行されるかの情報が欲しいので、型はクラスフィールドの ADVICES である。List 型 relationships の要素は IJavaElement 型オブジェクトで、どのクラスのどこに書かれたエレメントなのかの情報が含まれる。

```
ICompilationUnit cu = JavaCore.createCompilationUnitFrom(file);
List els = getJavaElements(cu);
for (Iterator elementit=els.iterator(); elementit.hasNext();) {
  IJavaElement javaElement = (IJavaElement) elementit.next();
  final AJRelationshipType relateType =
                  AJRelationshipManager.ADVISES;
  List relationships =
     AJModel.getInstance().
                  getRelatedElements(relateType, javaElement);
    ...
}
```

上に載せたアドバイスに関する情報を収集するコードは、ビューアを生成するときにコンテンツプロバイダで１度だけ実行され、List 型の要素として保持しておく。それ以降アドバイスに関する情報を使用したいときは、この List を利用する。例えば、ポップアップメニュー表示させる場合である。アドバイスによって拡張された Java エレメントに対してポップアップメニューを表示させると、どのアスペクトのアドバイスが実行されるかの情報がメニュー項目に表示される。この動作を実装したクラスでは、上のコードを実行するのではなく List に収められた情報を利用している。また、これはその Java エレメントが子オブジェクトにもつアドバイスの情報と同一のものである。

２章 AJDT の節で述べたように、AJDT をインストールすると、エディタのルーラー上にアドバイスに関するマーカーが表示される。１つのアドアイスにつき、アスペクト側とアドバイスが実行される Java エディタ側の２つのエディタの該当行にマーカーが付く。このアドバイスの上にマウスを乗せると、ツールチップが表示される。例えば、アドバイス側のマーカーのツールチップは、

advises Caller: method-call(void Worker.work(int))
と書かれて出力される。これは、Caller クラスの Worker#work(int) メソッドの呼び出し時にアドバイスが実行されるという意味である。一方、Caller クラスの Worker#work(int) メソッドの呼び出し部分のコードを見てみると、該当行にマーカーが記されている。このマーカーのツールチップには、
advised by Logger.after(): log...
と書かれている。これは、Logger アスペクトの after アドバイスが実行されるという意味である。また、1 箇所に複数のアドバイスが実行される場合、マーカーのツールチップには
2 AspectJ markers at this line
と書かれて出力される。これは、アスペクトのアドバイスが複数箇所に織り込まれる場合についても同様の形式である。以下は、マーカーのツールチップとマーカーのある行を出力するコード例である。

```
IFile file =
    ((IFileEditorInput) editor.getEditorInput()).getFile();
IMarker[] markers =
     file.findMarkers("org.eclipse.ajdt.ui.advicemarker",
                      true, IResource.DEPTH_ZERO);
for (int i = 0; i < markers.length; i++) {
    Map attr = markers[i].getAttributes();
    int line = ((Integer) attr.get("lineNumber")).intValue();
    String message = (String) attr.get("message");
    System.out.println("line:" + line + " " + message);
}
```

マーカーにはそれぞれ型がある。PROBLEM や BOOKMARK といった型がその一例である。また、マーカーには継承関係があり、マーカーを作成する際には plugin.xml にマーカー・スーパー型を記述する必要がある。リソース IFile のメソッド findMarkers により、必要なマーカーを取得することができる。findMarkers メソッドの第一引数には、マーカーの型を書く。今の場合、org.eclipse.ajdt.ui.advicemarker とある。このマーカーは org.eclipse.ajdt.ui.
aroundadvicemarker や org.eclipse.ajdt.ui.afteradvicemarker などの他のマーカーのスーパー型であるので、メソッドの第二引数を true にすることによりアドバイスに関するマーカーを全て取得できる。また、これがスーパー型であることは、プラグインマニフェストファイルである plugin.xml を見れば確認できる。例えば、plugin.xml 内で org.eclipse.ajdt.ui.
afteradvicemarker が定義されている箇所には、

```
<extension
      id="afteradvicemarker"
      name="After Advice Marker"
      point="org.eclipse.core.resources.markers">
   <super
        type="org.eclipse.ajdt.ui.advicemarker">
   </super>
   ...
```

と記述されていて、スーパー型が org.eclipse.ajdt.ui.advicemarker であることが分かる。findMarkers メソッドの第二引数を true にすることによって、第一引数を継承した型も対象として取得できる。第三引数では、対象となっているリソースのマーカーの検索の他に、そのリソースのメンバーの子リソースに関する情報も取得するかどうかを決めることができる。今の場合、IResource.DEPTH_ZERO となっているが、これは file 内のマーカーしか取得しないとうことである。IResource インタフェースには他にも DEPTH_ONE と DEPTH_INFINITE というフィールドがあるが、それぞれメンバーの子リソースまでを対象とするか、メンバーの子リソースに加えてそれらから派生する全てのリソースを対象とするかという意味である。

また、IFile 型 file インスタンスの取得方法は以下のようにする。PlatformUI クラスは Eclipse プラットフォームのユーザーインターフェイスへアクセスするための中核となり、主にワークベンチにアクセスのために使用される。file インスタンスを取得するためには、いくつかの手順を踏む必要がある。まず、PlatformUI クラスから IWorkbench 型のインスタンスを取得し、そこから現在アクティブなワークベンチウィンドウである IWorkbenchWindow を取得する。次に、ワークベンチウィンドウから現在アクティブなページ、IWorkbenchPage を取得する。さらに、そのページ内のアクティブなエディタを取得し、そこからファイルを取得することができる。具体的には以下のようにする。

```
IWorkbench workbench = PlatformUI.getWorkbench();
IWorkbenchWindow activeWindow =
             workbench.getActiveWorkbenchWindow();
IWorkbenchPage activePage = activeWindow.getActivePage();
IEditorPart editor = activePage.getActiveEditor();
IFile file =
    ((IFileEditorInput) editor.getEditorInput()).getFile();
```

3.2.6　Java ソースコードの解析

ビューの右側半分は Javadoc コメントを出力するためのウィンドウである。アドバイスによって拡張されたモジュールではそのモジュールとアドバイスの両方の Javadoc コメントを出力する。これらの Javadoc コメントを取得するためには、ソースコードの解析が必要となる。本システムでは、この解析に JDT の AST（Abstract Syntax Tree）を使用している。

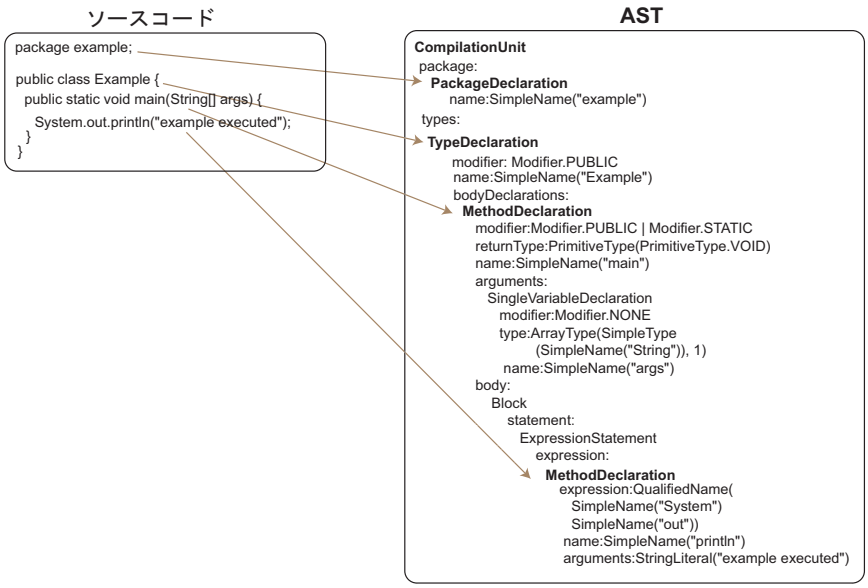
AST

Eclipse の JDT が提供する API を利用すると、Java のソースコードの操作やエラーの検出、コンパイルの実行、プログラムの起動などをすることができるが、より詳細にソースコードに対する操作を実行したいときには EclipseJDT の JDT DOM と AST を使う。JDT は JDOM と同様の独自の DOM、AST を持っている。JDOM とは Java ソースコードから XML データにアクセス、加工、出力する完全な Java ベースシステムのことである。JDT の DOM は org.eclipse.jdt.core.jdom パッケージ内で定義されているが、その役割は既に AST で置き換えられているといってよい。AST は org.eclipse.core.dom パッケージ内で定義されている。これは Java ソースの構文上の解析をサポートし、Java 言語の異なる要素を表現する 60 以上の ASTNode を定義している。JDOM がフィールド、メソッドレベルまでの解析なのに比べて、AST はノードを JDOM よりも詳細なレベルで解析する。また、AST は不完全、もしくは正しくないソースでも受け入れる。下図はソースコードとそれに対応した AST ノードを示している。太字で書かれた AST ノードは、ビジター処理の際に候補となる Java エレメントと対応している。AST を作成し、ソースコードを探索する具体的な方法は次節で述べる。

AST の探索

AST を使ったソースコード探索の方法では、まず AST クラスのクラスメソッドや ASTParser クラスのメソッドを用いてソースまたはコンパイルユニットの解析を行う。次に、ASTVisitor のサブクラスを定義してビジターを用いたノード処理の操作を定義する。最後に、必要な visit メソッドを上書きするればよい。

一般的に、低レベルノードまでアクセスしたい場合には 2 つの方法がある。1 つは、高レベルノードにアクセスした後でそのノードで定義されたメソッドを呼び出し処理をする方法である。例えば、メソッドの返り値の型を取り出したい場合などは、まずビジターでメソッド宣言ノー



ドを取得し、その後で MethodDeclaration クラスで定義されたメソッド getReturnType などを実行すればよい。２つ目は、ビジターで高レベルノードにアクセスした後で今度は低レベルノードに対するビジターを別に定義する方法である。

ビジターを用いたノード処理の定義は、必要な ASTNode 型の処理をするメソッドをオーバーライドするビジターを作成すればよい。例えば、メソッド宣言のビジターを生成したい場合には visit(methodDeclaration) メソッドをオーバーライドする。

本システムでは、メソッド宣言に対してビジターを実行する。したがって、ASTVisitor でオーバーライドするメソッドは visit(MethodDeclaration) である。このメソッドの中で、Javadoc コメントを取得するコードを記述する。具体的には下のようにする。

```
public boolean visit(MethodDeclaration node) {
    Javadoc jd = node.getJavadoc();
    if (jd != null) {
      list.add(node);
    }
    return false;
}
```

上のコードは、Java ソースコードに対しては機能するが、AJDT のソースコードには完全には機能しない。ビジターはアドバイスを methodDec-

laration として判断するが、そのアドバイスの Javadoc コメントを取得することはできずに null としてしまう。そこで、注目しているメソッドやアドバイスの Java エレメントから処理をすることでこの欠落を補っている。つまり、IResourceReference を実装している IMethod などは、getSource メソッドと getSourceRange メソッドを持っているので、これを使って javadoc コメント部分を得る。これらのメソッドを利用すると、ファイル拡張子が.java の中で記述されたメソッドの文字列を取得することができるのである。もちろん、最初からこれを利用すれば Java ファイルに対しては AST を使用する必要はなくなる。しかし、AspectJ ファイル内のアドバイスに対してこれらのメソッドを実行した場合、ポイントカットの情報を一緒に取得することができない。例えば、下のように取得した文字列はポイントカットが記述されていたはずの部分が抜けてしまっている。

```
/**
 * test javadoc comment
 */
after() :      {
System.out.println("exe test");
}
```

つまり、AST から得られる Javadoc コメントを欠いたアドバイスの宣言と、Java エレメントを利用して得られる Javadoc コメントと不正確なアドバイスの宣言を上手く組み合わせることで、正しいアドバイスとその Javadoc を生成している。

第4章　実験

以下の実験で、本システムの性能を計測する。実験対象には JHotDraw [12] と AJHotDraw [13] のソースコードを用いた。JHotDraw はオープンソースで、Java の GUI フレームワークであり、デザインパタンやフレームワークの研究などのために用いられる。そして、この JHotDraw をアスペクト指向を用いて書き換えたものが AJHotDraw である。その名前が示しているようにアスペクト指向言語には AspectJ が使われている。

計測方法は、javadoc コメントを出力させるのにかかる時間を測るものである。具体的な手順は、まずあるメソッドやフィールドを選択後、マウスを右クリックしてポップアップメニューを表示させる。次にメニュー項目の中で「Javadoc コメントを表示」メニューを選択する。これが計測の開始時間である。Javadoc コメントがビューの右側のウィンドウに表示された時間が計測の終了時間である。これを、JHotDraw と AJHotDraw の同じフィールドやメソッドに対して実行する。

4.0.7　実験環境

実験環境には以下のものを使用した。なお、ランタイムワークベンチを起動する際に、メモリ使用量を 256M に設定した。

- CPU：Pentium4 2.79GHz
- メモリ：2GB
- OS：WindowsXP SP2
- JavaSDK：5.0
- Eclipse：3.1.1
- AJDT：1.3.0

4.0.8　実験結果

以下の表に示す実験結果を得た。

[h] プロジェクト名	時間 (ms)
AJHotDraw	12.6
JHotDraw	9.1

表 4.1: 実行速度

4.0.9　考察

この実験から、AJHotDraw で Javadoc を表示させるには JHotdraw で表示させるより約 1.5 倍の時間がかかることが分かった。しかし、その速度の差は 4 ミリ秒以下である。したがって、アスペクト指向言語を導入したシステムに適用した場合も、十分実用的な速度で動作することが確認できた。

第5章　検証

本システムを実際の AspectJ プロジェクトに適用しその表示内容の有用性を検証した。アドバイスによって拡張されたモジュールの表示内容を確認するのが目的であるが、特に個々のモジュールの Javadoc コメントを結合することでどのような利便性があるかを確認した。まずは、AJHotDraw のような実際のアプリケーションに本システムを適用し、その有用性を確認した。また、AJHotDraw では確認しきれなかった本システムの機能の確認をするためにサンプルプロジェクトを作成し、その他の検証のために使用した。

5.1　AJHotDraw による検証

org.jhotdraw.framework パッケージ下にあるインターフェイス Figure で定義された invalidate メソッドを例にとる。この invalidate メソッドがアドバイスによって拡張されるということはビューの左側のウィンドウの階層を見れば分かる。invalidate 以下の階層を展開すれば、拡張するアスペクト名やアドバイスの種類、ポイントカット名の情報が見られる。
今の場合、
SelectionChangedNotification.after(StandartDrawingView):invalidateSelFigure(sdw)
と書かれている。しかし、これ以上の情報はここからは得られない。

そこで、invalidate を選択し Javadoc コメントを出力する。すると、invalidate メソッドの Javadoc コメントの下にもう 1 つ Javadoc コメントが出力される。SelectionChangedNotification アスペクトの invalidateSelFigure(sdw) ポイントカットで指定された after アドバイスの Javadoc コメントである。これら 2 つのコメントの内容をそれぞれ見てみる。invalidate メソッドの方には、figure インスタンスを無効にして、リスナに現在の表示が無効になっていることを通知するという内容のコメントが書かれている。一方、アドバイスの側には、updater ではなく notifier を呼ぶという内容のコメントが書かれている。invalidate メソッドのコメントだけでは、

figure を無効にしてリスナに通知するということまでしか分からない。しかし、実際には invalidateSelFigure アドバイスが notifier を呼ぶという挙動が figure には追加される。したがって、このメソッドとアドバイスのコメントの両方を読まなければ実際の挙動を理解できない。また、メソッドのコメントを先にウィンドウに表示し、その後でアドバイスのコメントを載せてある。これは、モジュールの仕様にアドバイスの仕様が書き加えられて、実際の挙動を示すと考えたためである。つまり、モジュールがある条件の下でアドバイスを実行して自身のモジュールを拡張するという視点に立っている。

5.2　その他の検証

AJHotDraw では、１つのモジュールに複数のアドバイスが実行される事例がなかった。また、アドバイスの宣言部で call ポイントカットと wihin や withincode ポイントカットが併用される事例もなかった。そこで、下のようなサンプルプロジェクトを作成しこれらの事例に対する表示内容の検証を行った。

ポイントカット change と move によって指定される after アドバイスが Point クラスのメソッド setX(int) を拡張する状況を想定する。

```
public class Point {
 int x, y;
 public void setX(int x) {this.x = x;}
 ...
}

aspect Update {
 pointcut change() : ...;
 pointcut move() : ...;
 after() : change() {...}
 after() : move() && within(Point) : {...}
}
```

このときのビューの該当箇所の表示は下のようになった。「extended by advice only if the Caller is Point」というのは「move() && within(Point)」というポイントカットの within(Point) の部分を反映したものである。その下の階層で、「UpdateSignaling.after():...」となっていてポイントカット名が省略されているが、これはポイントカットがオペレータで結合されている場合に省略することになっているからである。また、その下の「extended

by advice」によって change() ポイントカットで指定された after アドバイスが実行されることを表している。ここでは、ポイントカット名は省略されていない。ポイントカットの宣言をそのままアドバイス実行の条件に使用している場合は表示することになっている。

ポイントカットがオペレータで結合されている場合にもそれを表示することにすると、プリミティブポイントカットのみで生成したポイントカットの場合にもそれをそのまま表示することになる。こういったポイントカットはアスペクトの実装に関わる部分である。setX モジュールにその情報は必要なく、不可視であった方がよい。この例の場合には、プリミティブポイントカットは within(Point) だけであるので問題はないが、それを補うために「... only if the Caler is Point」という情報を提示している。実装に関わる内容で有用なものはビュー右側の Javadoc ウィンドウで表示することにしている。

第6章　まとめ

本研究では、この Aspect-Aware Interface の考えを基にして、アスペクトによるモジュール拡張を強調したアウトラインビューアを Eclipse のビュープラグインとして開発した。各モジュール（クラス、メソッド、フィールド）の属性を階層的に表示するときに、アスペクトの情報を重ねて表示することができるようになった。call、set、get ポイントカットによって指定されたアドバイスの場合にも、それによって拡張されるターゲット側のモジュールに情報を重ねて表示する点が AJDT と異なる。

また、このビューには選択したモジュールとそれを拡張するアドバイスの Javadoc コメントを同時に並べて表示させることもできる。それにより、それぞれ違う場所にあった関連のある Javadoc コメントを同時に見ることができるようになった。本研究では実際のアプリケーション（AJHotDraw）を用いて実験をすることにより実用的な速度で Javadoc コメントの表示が可能なことを確認し、それとともに、本システムによりばらばらの場所にあった関連のあるコメントを統合することで有用な情報が得られることの検証も行った。

6.1　今後の課題

今後は、拡張されるモジュールとアドバイスの Javadoc コメントの内容を意味的に結合、最適化して有用な情報を提供できるようにする。そうすることで、モジュールがどのような状況のときにアドバイスにより拡張されるのかを理解できるようにする。意味的に結合されたコメントの内容をより有用なものにするためには、そのアドバイスで使われるポイントカットの情報も必要になってくると考えている。プログラマが知りたいのはアドバイスに関する情報の他に、どの時点でそのアドバイスが実行されるかという情報である。この情報はポイントカットのコメントとコードに書かれている。

そこで、まずポイントカットのコメント部分に書かれている内容を分析する必要がある。簡単のため、まずはトレースやロギングのように数箇所のモジュールに跨っている同一のコードを１箇所にまとめる働きをするアスペクトについて考えることにする。このような場合、ポイントカットの

コメント部分に記述されるのはどの時点、条件でアドバイスを実行するかという内容であると考えられる。このとき、このコメント部分とポイントカットのコードは意味的に一致している。つまり、コメント部分にはポイントカットのコードから読み取れないことは書かれていない。そこで、コメントではなくポイントカットのコードを解析し、それをコメントビューに反映させるようなシステムが有効だといえる。上記のように、まずはトレースやロギングのように同じコードが束ねるために書かれたアスペクトを対象にこの研究を進めていきたい。

参考文献

[1] Jonathan Aldrich.: Open Modules: Modular Reasoning in Aspect-Oriented Programming, *In Foundations of Aspect Languages* (2004).

[2] Jonathan Aldrich.: Open Modules: Modualr Reasoning about Advice, *In European Conference on Object-Oriented Programming* (2005).

[3] Aspectj project: `http://www.eclipse.org/aspectj`.

[4] Andy Clement, Adrian Colyer, Mik Kersten.: Aspect-Oriented Programming with AJDT, *In ECOOP Workshop on Analysis of Aspect-Oriented Software* (2003)

[5] Curtis Clifton, Gary T. Leavens.: Observers and Assistants: A Proposal for Modular Aspect-Orieted Reasoning, *In Foundations of Aspect Languages* (2002)

[6] Eclipse project: `http://www.eclipse.org/`.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugnin, Mik Kersten, Jeffrey Palm, William G. Griswold: An Overview of AspectJ, *In European Conference on Object-Oriented Programming*, pp. 327–353 (2001).

[8] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *In European Conference on Object-Oriented Programming*, pp. 220–242 (1997).

[9] Gregor Kiczales, Mira Mezini.: Aspect-Oriented Programming and Modular Reasoning, *In International Conference on Software Engineering* (2005).

[10] Rovert E. Filman, D.P.Friedman: Aspect-Oriented Programming is Quantification and Obliviousness, *In Advanced Separation of Concerns* (2000).

[11] 千葉滋: アスペクト指向入門　Java・オブジェクト指向から AspectJ プログラミングへ, 技術評論社 (2005).

[12] JHotDraw: http://www.jhotdraw.org/.

[13] AJHotDraw:http://swerl.tudelft.nl/bin/view/AMR/AJHotDraw/.