平成18年度 学士論文

開発者との対話を活かした
横断的構造の表現

東京工業大学 理学部 情報科学科

学籍番号 03-1728-6

戸部　敦

指導教員

千葉 滋 助教授

平成19年2月28日

概要

オブジェクト指向のモジュール化を補完する技術として、アスペクト指向の研究が注目されてきている。アスペクト指向言語によるソフトウェア開発では、プログラムの持つ横断的構造を視覚化するツールが必要不可欠となる。アスペクト指向言語に obliviousness という性質があるため、プログラムの持つ横断的構造が理解しにくい。横断的構造を理解せずにソフトウェア開発を行うと、クラスの変更によってアスペクトによるジョインポイントが選択されなくなることがある。これは fragile pointcut problem として知られている。

本研究では、アスペクトとクラスの両方から、横断的構造をクラス図として視覚化できるツール C&Y を提案する。利用者が選択したクラスに関連するアスペクトを表示させられるので、obliviousness によってソースコード上からは読み取れないアドバイスの織り込みを把握することができる。したがって、クラスの変更をする際にアスペクトが受ける影響を把握でき、fragile pointcut problem を回避できる。また、ポイントカット記述にワイルドカードを使用した場合は、意図しないジョインポイントを選択してしまう可能性がある。C&Y では、利用者がアスペクトを選択して、そのアスペクトに関連するクラスを表示することもできるので、アスペクトによって選択されるジョインポイントを把握することができる。

C&Y では、クラス図を用いて横断的構造の視覚化を行った。クラス図には使用関係という形でジョインポイントを表現できるため、アスペクトによるジョインポイントの選択を容易に可視化できる。また、クラス図は継承関係も表現できるため、アスペクトがクラス間の継承関係に与える変更も視覚化できる。クラス図を使うことのデメリットは、横断的関心事の視覚化に伴って図が複雑化してしまうことである。本システムでは、クラス図の複雑性を軽減するために、必要に応じて簡易表示ができる。メンバーの表示・非表示を切り替える機能を実装し、必要のないメンバーを非表示にすることで、図の簡易化を行うことができる。また、アスペクトをクラスのメンバーを拡張するものとみなし、アイコンでアスペクトによるアドバイスの織り込みを表現する。アイコン化されたアドバイスの織り込みは、利用者からの要求があるまで詳細表示を待つ。

クラス図によって横断的構造の視覚化を行うツールとして ActiveAspect

が挙げられる。ActiveAspect は、利用者がアスペクトを選択すると、そのアスペクトがプログラムにもたらす横断的構造を視覚化する。しかしC&Y とは異なり、クラスから関連するアスペクトを表示できないため、クラスの変更に伴う fragile pointcut problem は解決されない。AJDT も横断的構造の視覚化を行うツールとして挙げられる。しかし AJDT は複数のツールを併用して使うことを前提としているため、開発者は複数のビューに渡る情報を自分で統合しなければならない。

謝辞

本研究を進めるにあたり、研究が始まってから論文の作成まで様々な助言をくださった千葉滋助教授に感謝いたします。また、本稿のスタイルファイルを作成してくださった光来健一助手に感謝いたします。

アスペクト指向についての基礎知識から教えていただいた西澤無我氏、概要の構成について相談に乗っていただいた柳澤佳里氏に感謝いたします。堀江倫大氏には、関連研究を紹介していただき、論文の作成まで手伝っていただきました。心より感謝いたします。また、いつも明るい笑顔で励ましてくれた研究室の先輩の皆様、そしてともに研究に励んだ学部生の方々に感謝いたします。

目 次

第 1 章 はじめに **8**

第 2 章 横断的構造を可視化するツール **10**

2.1 アスペクト指向 10
2.2 横断的構造をもつプログラムの問題 13
2.2.1 アスペクトがクラスに及ぼす影響 13
2.2.2 クラスがアスペクトに与える影響 14
2.2.3 fragile pointcut problem 14
2.2.4 視覚化ツールの必要性 16
2.3 既存の視覚化ツール 16
2.3.1 AJDT 16
2.3.2 ActiveAspect 19
2.3.3 問題点のまとめ 22

第 3 章 C&Y **23**

3.1 クラス図 23
3.2 簡易表示 24
3.2.1 Event-Based Approach 26
3.2.2 Extension-Based Approach 26
3.3 詳細表示 27

第 4 章 実装 **29**

4.1 Eclipse プラットフォーム 29
4.1.1 アーキテクチャ 29
4.1.2 ワークベンチ 30
4.1.3 マニフェストファイル 32
4.2 GEF 33
4.3 GluonJ 34
4.4 クラスファイルの解析 36
4.4.1 クラスファイル 37
4.4.2 静的な依存関係 37
4.4.3 使用関係 38

4.4.4 ポイントカット 38
4.4.5 インタータイプ宣言 39
4.5 表示方法の実装 39
4.5.1 簡易表示の実装 39
4.5.2 アイコン表示の実装 40
4.5.3 詳細表示の実装 41

第 5 章 応用例 **43**
5.1 Observer アスペクト 43
5.2 C&Y を用いない場合 44
5.3 C&Y を用いた場合 45
5.4 検証結果 46

第 6 章 まとめ **48**
6.1 今後の課題 48

図 目 次

2.1 従来のポイントカットと model-based なポイントカット . 15
2.2 AspectJ エディタ 17
2.3 AJDT Outline 18
2.4 AJDT Visualizer 20
2.5 AJDT Cross References 20
2.6 AJDT Cross References 21
2.7 ActiveAspect 21

3.1 クラス図によるジョインポイントの表現 24
3.2 caller 側の拡張と考えた場合の表示 25
3.3 callee 側の拡張と考えた場合の表示 25
3.4 イベントとしてのジョインポイント 26
3.5 アスペクトによって選択されたジョインポイントの表現 . . 28

4.1 Eclipse のアーキテクチャ 30
4.2 ワークベンチウィンドウ 31
4.3 プラグインマニフェストエディタ 33
4.4 アスペクトの間接的な依存関係 38
4.5 簡易表示されたクラスとアスペクト 40
4.6 アイコン化されたアスペクトによるメンバーの拡張 41
4.7 Caller クラスの詳細表示 42

5.1 サンプルソフトウェアの実行結果 43
5.2 変更前のソフトウェアの構造 (簡易表示) 44
5.3 クラス変更後のソフトウェアの構造 (簡易表示) 45
5.4 変更前のソフトウェアの構造 (詳細表示) 45
5.5 変更後の Display クラスの public メソッド 46
5.6 変更前の Display クラスに注目し横断的構造を表示 47

表 目 次

4.1 GluonJ のポイントカット例 35

第1章　はじめに

今日、オブジェクト指向の普及によって高度なモジュール化が達成されている。しかし、オブジェクト指向ではモジュール化ができない関心事も存在する。例えば、ロギングのコードは複数のクラスに散在してしまうため、ロギングが必要なくなったときにログ処理を行っている全てのクラスを変更しなければならない。これではロギングという関心事をモジュール化できているとは言えない。このように複数のクラスにまたがる関心事を横断的関心事と言う。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モジュール化する技術である。ロギングのような横断的関心事をアスペクトにまとめることで、既存のプログラムへの機能の追加、削除が容易になる。アスペクト指向言語によって、横断的関心事のソースコード上の分離が達成される。

横断的関心事をソースコード上から分離したため、アスペクト指向言語は、ソースコードからクラスとアスペクトの関係性が読み取れない obliviousness という性質を持つ。アスペクト指向言語の obliviousness によって、プログラムの横断的構造は理解しにくいものとなる。横断的構造を理解せずにプログラムを変更しようとすると、新たな問題が生じる危険性がある。クラスの変更がアスペクトの機能に影響を与える代表的な例は fragile pointcut problem である。fragile pointcut problem は、クラスの変更によって選択されるべきジョインポイントが選択されなくなったり、開発者の意図しないジョインポイントが選択されてしまう問題である。アスペクト指向言語による開発では、横断的構造を視覚化するツールを使うことによって、fragile pointcut problem のような問題を避けることができる。

本研究では、クラス図によって横断的構造を視覚化できるツール C&Y を提案する。C&Y は、利用者がクラスとアスペクトのどちらに注目したとしても、関連する横断的構造を表示できるという点を重視して開発した。クラスに注目したときに関連するアスペクトとの関係性を表示できることで、obliviousness によってソースコード上からは読み取れないアドバイスの織り込みを把握できる。利用者は、 C&Y を使ってクラスから関連するアスペクトを表示することで、obliviousness によってもたらされる fragile pointcut problem を避けることができる。また、C&Y ではア

スペクトに注目をすると関連するクラスとの関係性も表示できる。多くのアスペクト指向言語ではポイントカットにワイルドカードを含めることができるため、アスペクトが影響を及ぼすクラスを全て把握するのは困難である。利用者は C&Y を使うことでポイントカットの記述をより的確なものにできる。

以下、 2 章では視覚化ツールの必要性を示し、既存のツールの問題点について述べる。3 章は C&Y による視覚化の手法、 4 章は C&Y の実装について述べる。さらに 5 章で C&Y を用いたソフトウェア開発の例を上げ、最後に 6 章で本論文をまとめる。

第2章　横断的構造を可視化するツール

アスペクトを含むシステム開発では、プログラムの横断的構造を理解しなければばならない。開発者は視覚化ツールを使うことで、一目ではわかりにくい横断的構造を理解することができる。視覚化ツールは、ツールを使用することによって開発コストを削減し、同時に必要な情報を開発者にわかりやすく表示しなければばならない。しかし開発のコストを意識し、1 つのビューで横断的構造を表示をしようとするとビューが複雑化してしまう可能性がある。また横断的構造を表示する開発ツールでは、アスペクトから関連するクラスとの関係性を表示することだけでなく、クラスから関連するアスペクトとの関係性を表示できる必要がある。

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) の普及によって高度なモジュール化が達成された。しかしロギングなどのコードは複数のオブジェクトにまたがってしまうため、オブジェクト指向で全ての関心事が分離されているとは言えない。ロギングのように複数のモジュールにまたがってしまう関心事を横断的関心事と言う。アスペクト指向プログラミング (AOP) は横断的関心事を 1 つのモジュールにまとめるプログラム技法である。アスペクト指向言語ではアスペクトと呼ばれるクラスとは異なったモジュールを提供し、アスペクトに横断的関心事をまとめることで関心事の分離を行っている。代表的なアスペクト指向言語 AspectJ [3] は、Java 言語を言語拡張することによってアスペクト指向プログラミングを実現している。以下にアスペクト指向言語で用いられる用語を AspectJ による例とともに取り上げる。[11]

- **アスペクト**
  横断的関心事をまとめたモジュールのこと。ポイントカットとアドバイスの組やインタータイプ宣言を持つ。AspectJ では以下のように記述する。

```
public aspect LoggingAspect {
  ...
}
```

- **ジョインポイント**
プログラムの実行時にアスペクトによってアドバイスを織り込むことができる位置を指す。アスペクト指向言語によってジョインポイントとして採用している位置は異なるが、一般的にはメソッドの呼び出し時やフィールド参照時などがこれに当たる。AspectJ ではメソッドの実行時や例外ハンドラの実行時もジョインポイントとして採用している。

- **ポイントカット**
ジョインポイント全ての集合から、実際にアドバイスを織り込むジョインポイントの部分集合を選択する記述のこと。一般的にポイントカットにはワイルドカードを使用できる。public メソッドの呼び出し時を選択する callPublicMethod ポイントカットは、AspectJ で以下のように記述される。

```
public aspect LoggingAspect {
  pointcut callPublicMethod : call(public * *(..));
  ...
}
```

- **アドバイス**
プログラムの実行がアスペクト内の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。ジョインポイントの直前やジョインポイントの直後などコードが実行されるタイミングを指定する。また、アドバイス内にはアスペクト指向言語が提供している特殊変数を用いることができる場合が多い。AspectJ ではポイントカットと組み合わせて以下のように記述される。before() を記述することでコードの実行されるタイミングとしてジョインポイントの直前を指定している。thisJoinpoint は AspectJ の提供している特殊変数である。

```
public aspect LoggingAspect {
  pointcut callPublicMethod : call(public * *(..));
  before() : callPublicMethod {
    System.out.println(thisJoinPoint.getSignature());
  }
}
```

- **インタータイプ宣言**
  アスペクトからプログラムの静的構造を変更する機能のこと。継承関係の変更や新たなメンバーの追加、フィールドの初期値の変更などを行うことができる。AspectJ では以下のように記述される。

```
public class Point {
  int x, y;
  public int getX(){ return x; }
  public int getY(){ return y; }
}

aspect PolarAspect {
  // Point のインターフェースに PolarFigure を追加
  declare parents : Point implements PolarFigure;

  // Point に フィールド (radius, theta) を追加
  int Point.radius;
  int Point.theta;

  // Point にメソッド (getRadius(), getTheta()) を追加
  public int Point.getRadius(){ return radius; }
  public int Point.getTheta(){ return theta; }
}
```

アスペクトはオブジェクトによって分離されたモジュールを横断するものであり、アスペクト指向言語を用いて開発されたプログラムは横断的構造を持つ。ここで言う横断的構造とは、アスペクトによって横断的関心事をまとめたプログラムの構造を指す。

2.2　横断的構造をもつプログラムの問題

AspectJ を始めとするアスペクト指向言語によってソースコード上の横断的関心事の分離は達成される。アスペクト指向言語のソースコード上からクラスとアスペクトの関係性が読み取れない性質を obliviousness という。しかしソースコード上からクラスとアスペクトの関係性が読み取れないからといって、クラスとアスペクトの依存関係がなくなったわけではない。横断的構造をよく理解せずにプログラムを変更しようとすると、開発者の予期せぬ問題が生じる可能性がある。

2.2.1　アスペクトがクラスに及ぼす影響

1 つのアスペクトに変更を加えたとするとアドバイスが織り込まれるクラスにも影響を及ぼす。AspectJ によるロギングの例を以下に示す。

```
class Factorial {
  int calc(int n){
    if(n <= 1) return 1;
    return n * calc(n-1);
  }
}

aspect FactorialLogger {
  pointcut callCalc(int n) :
    call(int Factorial.calc(int)) && args(n);
  before(int n) : callCalc(n){
    System.out.println("before: calc(" + n + ")");
  }
}
```

callCalc ポイントカットは calc メソッドの呼び出し時をジョインポイントとして選択している。args(n) は calc メソッドの引数に名前をつけ、アドバイス内で引数の値を参照できるようにするための記述である。アスペクト指向言語が持つ obliviousness によって、アドバイスが織り込まれるクラス側からはアスペクトの情報を得ることができない。このようにソースコード上ではクラスとアスペクトの依存関係は除去されているのにも関わらず、アドバイスの織り込みによってクラスとアスペクトの間に依存関係ができる。実際にアドバイス織り込み後の Factorial クラスは、簡単に書くと以下のようになる。

```
class Factorial {
  int calc(int n){
    if(n <= 1) return 1;
    int $1 = n-1;
    System.out.println("before: calc(" + $1 + ")");
    return n * calc($1);
  }
}
```

クラスとアスペクトの依存関係によって、アスペクトの変更がクラスに影響を与えることは明らかである。もし FactorialLogger のアドバイスを書き換えれば、クラス内に織り込まれるコードも変わってしまう。さらにポイントカット内にワイルドカードを使用した場合はアスペクトの影響は多くのクラスに波及する。したがって開発者はアスペクトを変更する際にどのクラスに影響を与えるかを把握する必要がある。しかしワイルドカードを含んだポイントカットが実際にどのジョインポイントを選択するかを知ることは困難である。

2.2.2　クラスがアスペクトに与える影響

アスペクトの変更がクラスに影響を与えるだけではなく、クラスの変更がアスペクトに影響を与える場合も存在する。代表的な例として織り込まれるクラスの変更に伴って、意図をしないジョインポイントが選択されたり、選択されるべきジョインポイントが選択されなくなるという問題がある。

2.2.3　fragile pointcut problem

fragile pointcut problem は、クラスのソースコードを変更したときに、ポイントカットによって選択されるべきジョインポイントが選択されなくなったり、意図をしないジョインポイントが選択されてしまう問題である。このようにクラスの変更に弱いポイントカットを fragile なポイントカットという。

例えば AspectJ で記述された次のポイントカットを考える。

```
pointcut selectAccessors() :
  call(void Buffer.set(Object))
   || call(Object Buffer.get())
```

selectAccessors() ポイントカットは Buffer クラスの accessor メソッドを選択している。しかしこのようにジョインポイントを列挙した場合、Buffer クラスの accessor メソッドのシグネチャが変わる度にポイントカットの記述を変更しなければジョインポイントが選択されなくなってしまう。したがって selectAccessors() ポイントカットは fragile であると言える。

同様にワイルドカードを用いたポイントカットも fragile である。

```
pointcut selectAccessors() :
  call(* set*(..)) || call(* get*());
```

例えば setting() というメソッドを定義したとする。accessor メソッドとして定義したつもりがなくても、上記のコードでは setting() メソッドの呼び出しも選択してしまう。

fragile pointcut problem を解決するために、言語仕様を変更しポイントカットの記述言語を開発するアプローチがある。[5] Model-based Pointcut [2] はその 1 つであり、model-based なポイントカットの記述を提案している。図 2.1 の右側が model-based なポイントカットの概念図である。これはソースコードの構造を抽象化したものを概念モデルとして定義し、ポイントカットをその概念モデルを用いて記述するものである。概念モデルを実装から切り離すことで fragile pointcut problem の解決を試みている。これは annotation-property や annotation-call [6] など、アノテーションを用いてモデルを作成する方法と類似している。

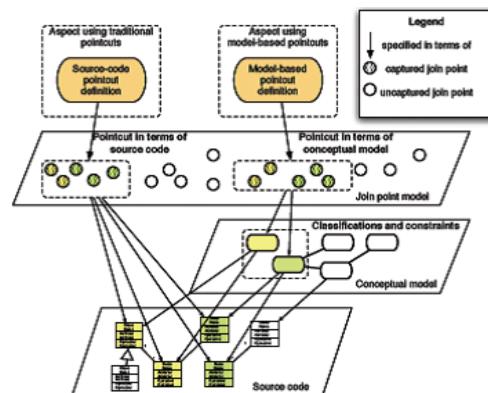

図 2.1: 従来のポイントカットと model-based なポイントカット

しかし AspectJ などのアスペクト指向言語で記述できるポイントカットが fragile であることには変わりない。言語仕様を変えずに fragile pointcut problem を避けるには、開発者がクラスを変更する際に関連するアスペクトに注意を払う必要がある。しかしクラスのコードにはアスペクトの情報

が一切記述されないため、クラスから関連するアスペクトを把握することは困難である。

2.2.4　視覚化ツールの必要性

このようにアスペクトとクラスは互いの情報をソースコード中に含んでいない場合でも影響しあっている。したがって開発者が横断的構造を理解せずにアスペクトやクラスに変更を加えたとすると、プログラム中の他方の役割が達成できなくなってしまう可能性がある。その問題を避けるためには開発者がプログラムの持つ横断的構造を理解しなければならない。しかしアスペクトとクラスの関係は一目ではわかりにくく、開発者がプログラムの横断的構造を理解することは困難である。アスペクト指向言語を用いた開発において視覚化ツールを使用することで、開発者が横断的構造を理解しやすくなり、上記の問題を避けることができる。

2.3　既存の視覚化ツール

AspectJ は有名なアスペクト指向言語であり、AspectJ で書かれたプログラムの横断的構造を視覚化するツールは数多く存在する。視覚化ツールによってアスペクト指向言語での開発は比較的容易なものになる。しかしアスペクトがプログラム全体に与える影響と、クラスがアスペクトから受ける影響を同時に把握しようとすると、既存の視覚化ツールでは表現できないことがある。ここでは AspectJ の視覚化ツールのいくつかを取り上げるとともに、それぞれのツールの問題点について言及する。

2.3.1　AJDT

AJDT(AspectJ Development Tools) [1] は AspectJ での開発を支援する統合開発環境 Eclipse [4] のプラグインである。AJDT は AspectJ の開発を支援するためにさまざまなツールを提供している。

Eclipse

Eclipse はオープンソースの統合開発環境 (IDE:Integrated Development Environment）である。統合開発環境とはエディタ、コンパイラ、デバッガなどソフトウェア開発に必要な機能を備えた環境のことで、統合開発環境でソフトウェア開発を行うことによって開発コストを大幅に減らすことができる。さらに Eclipse はプラグインを追加することで機能を後から追加できる。Eclipse の詳細については 4.1 で説明する。

AspectJ エディタ

Eclipse 上で AspectJ のプログラムを作成する場合には AJDT によって拡張されたエディタ (AspectJ エディタ) を使用する。AspectJ エディタでは、インタータイプ宣言の有無やアドバイスが実行される位置が左端のルーラーのマーカーを見ることによって一目でわかるようになっている。マーカーからポップアップメニューを開き、「Aspect Declarations」を選択するとインタータイプ宣言されたメンバーが、「advised by」を選択すると実行されるアドバイスがわかる。同様にアスペクトの側からもマーカーから、「Declared On」を選択するとインタータイプ宣言によって拡張するクラス名が、「advises」を選択するとアドバイスが何処に織り込まれるかが表示できる。図 2.2 は AspectJ エディタの使用例である。

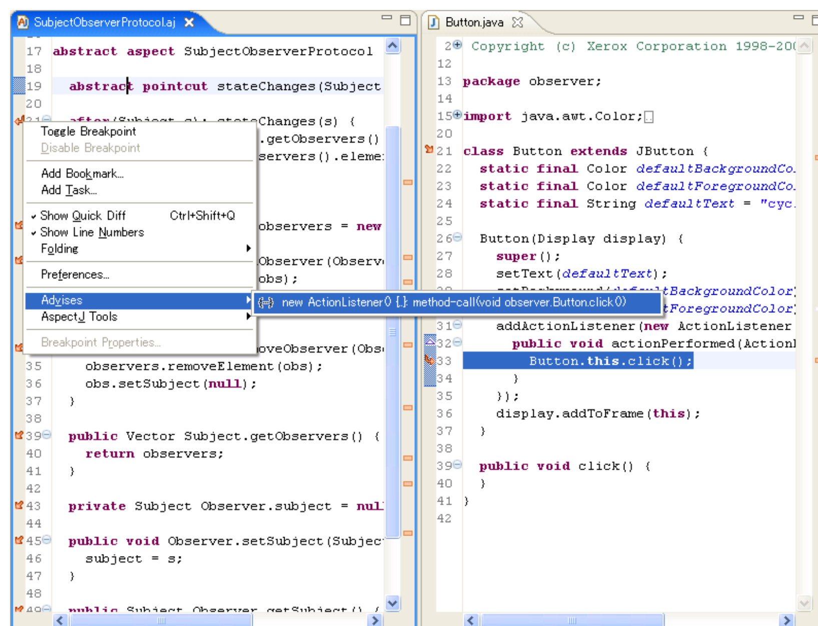

図 2.2: AspectJ エディタ

AspectJ エディタによって開発者はアスペクトが作用する位置を知ることができる。しかしその位置でどのアドバイスが実行されるのかなどの情報は一目ではわからず、開発者がその都度ポップアップメニューを開き表示しなければならない。さらに AspectJ エディタを見ただけではプログラム全体のアスペクトの織り込みを把握することはできない。これを把握するためには AJDT の他のツールを併用しなければならない。

Outline

Outline は現在アクティブになっているファイルの情報を表示するツールである。Outline にはメンバーのシグネチャのみが表示される。図 2.3 は Outline の使用例である。Outline を見るとアクティブになっている

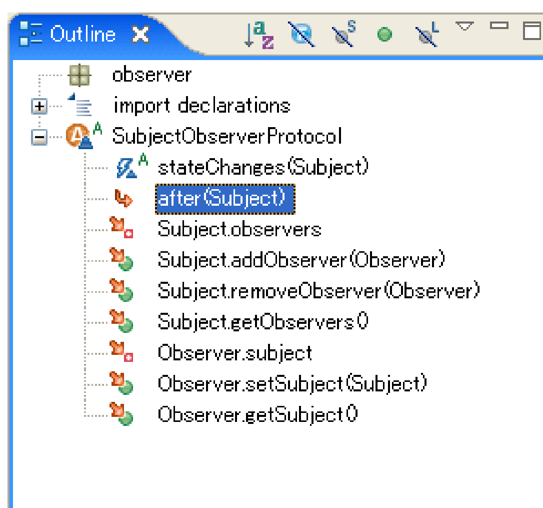

図 2.3: AJDT Outline

ファイルの概要を知ることができる。しかし Outline にはアドバイスが何処に挿入されるかが記述されないため、Outline を見ただけではクラスとアスペクト間の関係性を知ることはできない。

Visualizer

Visualizer はクラスのコードに対してアスペクトが作用する箇所を色で塗り分けるツールである。アスペクトによって塗り分けられる色が異なり、どの色がどのアスペクトを表しているかは Visualizer Menu によって把握できる。また、色のついたバーをダブルクリックすることでエディタ上のアスペクトが作用する位置にジャンプできる。図 2.4 は AJDT Visualizer の使用例である。

開発者は Visualizer によってプロジェクトの全てのクラスとアスペクトとの関連性の有無を見ることができ、アスペクトが作用するクラス内のおおまかな位置を知ることができる。しかしそれ以上の情報を得ようとするとビュー内のアスペクトが作用している箇所をダブルクリックしてエディターにジャンプしなければならない。さらに Visualizer にはクラス間の関係が表示されないといった問題がある。

Cross References

Cross References は現在アクティブなファイルの内容を表示する。例えばクラスが定義されたファイルを開くと図 2.5 のように表示される。図 2.5 を見ると、アスペクトによって宣言されたメンバーや、アスペクトによってアドバイスが織り込まれる箇所が表示されていることがわかる。アスペクトが定義されたファイルを開いた場合は、図 2.6 のように表示される。このようにアスペクトがインタータイプ宣言によってどのクラスに作用するか、またアスペクトがどのジョインポイントにアドバイスを織り込むかを表示できる。

Cross References を使うとクラスから作用するアスペクトとの関連、アスペクトからアスペクトが作用するクラスとの関連を表示できる。しかしインタータイプ宣言によって宣言されたメンバーが、どのアスペクトから追加されたのかはわからない。さらに図 2.6 のようにアドバイスが織り込まれるジョインポイントを知ることができるが、呼び出し側の情報はクラス名しか表示されていない。呼び出し側のどのメンバーから呼び出しが行われているかを知りたいときには、呼び出し側のファイルを開くしかない。

2.3.2　ActiveAspect

ActiveAspect[8] は、UML のクラス図にアスペクトを含めて表示する。クラス図にアスペクトを含める場合、図が複雑になりやすいという問題がある。ActiveAspect は次の方法で図が複雑化するのを防いでいる。まず ActiveAspect は開発者に注目したいアスペクトを 1 つ選択させる。ActiveAspect はそのアスペクトのアドバイス本文やポイントカットの記述から関連のあるクラスを自動で収集する。収集したクラスをアスペクトとの関係によって抽象化し、図から無駄な情報を自動で削除する。抽象化の際にはクラスのメンバーのアスペクトとの関係性に応じて重みを計算し、計算した結果が小さいものから順にまとめていくというアルゴリズムを実装している。最終的に抽象化を行った図が開発者に表示される。図 2.7 は Billing アスペクトを選択したときに ActiveAspect が表示する図である。

ActiveAspect を用いることで開発者はアスペクトがクラスに与える影響を把握することができる。また ActiveAspect では 1 つのアスペクトとプログラム全体のクラスとの関係性を表示するため、注目したアスペクトがプログラム全体でどのような働きをするのかという情報を得ることができる。しかし ActiveAspect では関連のあるクラスの収集・抽象化を全て自動で行っているため、アスペクトの変更に影響を受けるクラスを収集

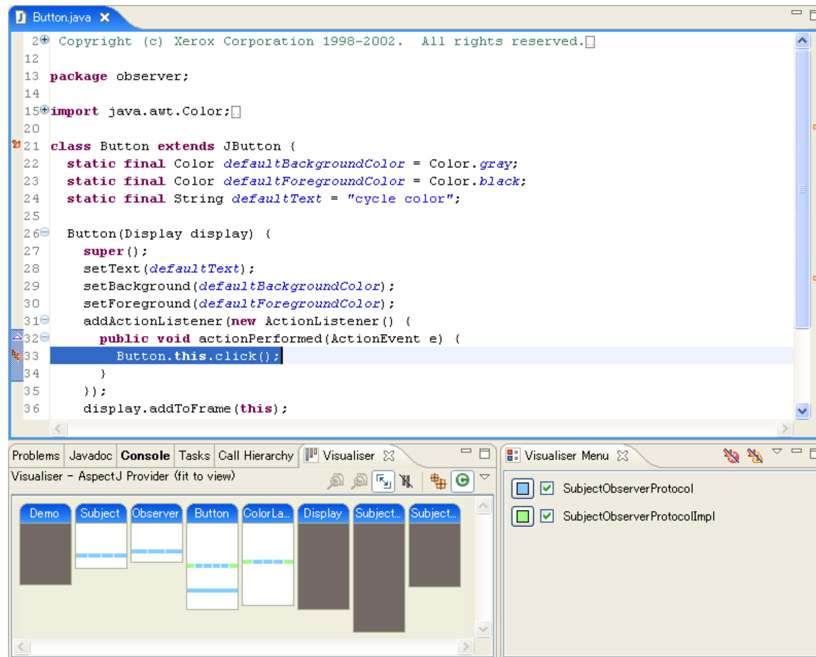

図 2.4: AJDT Visualizer

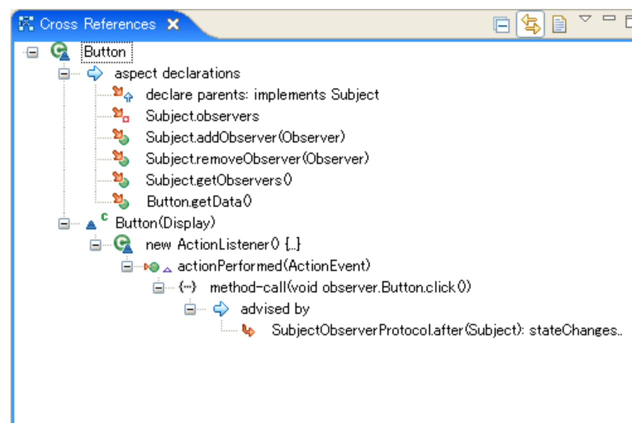

図 2.5: AJDT Cross References

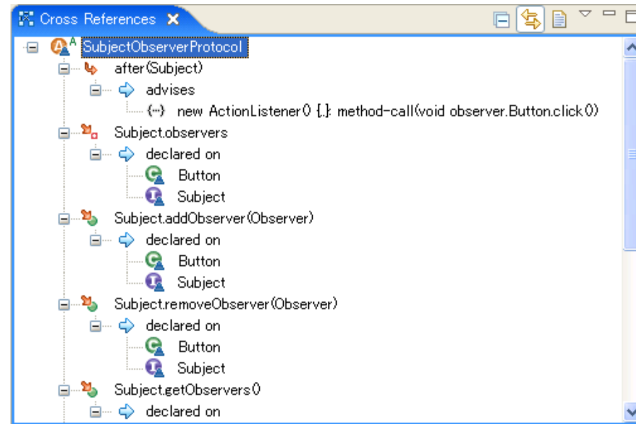

図 2.6: AJDT Cross References

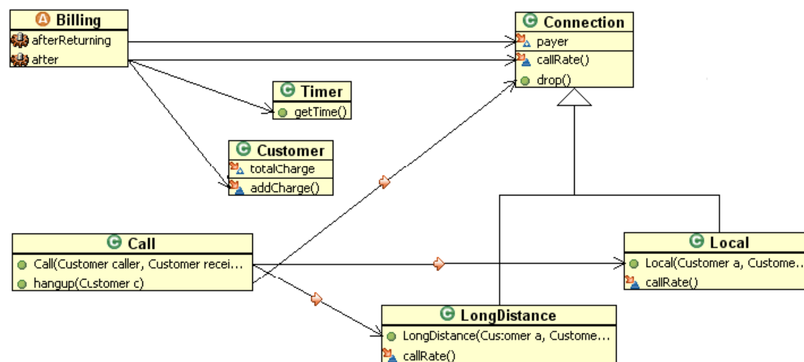

図 2.7: ActiveAspect

できなかったり、開発者によって重要なクラスの情報が抽象化の際に失われてしまう可能性がある。さらに ActiveAspect ではクラスから関連するアスペクトとの関係を表示できない。したがってクラスの変更がアスペクトに影響を及ぼす場合は ActiveAspect ではその影響を把握しきれないため、fragile pointcut problem は解決されない。

2.3.3　問題点のまとめ

AJDT に含まれるそれぞれのツールは複数のツールを併用して使うことを前提としている。そのため開発者はいくつものビューに渡る情報を自身で統合し、横断的構造を理解しなければばならない。したがって横断的構造を理解するためには多くの時間を要し、アスペクト指向言語の開発における開発コストは増大してしまう。ActiveAspect は 1 のビューでアスペクトとアスペクトに関連するクラスとの関係性を表示する。しかし ActiveAspect には、クラスから関連するアスペクトとの関係性を表示できないため、fragile pointcut problem を回避することができない。

第3章　C&Y

本研究では、クラス図を用いて横断的構造を視覚化するツール C&Y を Eclipse プラグインとして実装した。クラス図を用いることのメリットはジョインポイントの可視化が容易であることである。メンバー間の使用関係はジョインポイントそのものであり、ポイントカットによるジョインポイントの選択は使用関係に向かう矢印として表現できる。逆にクラス図を用いることのデメリットはビューが複雑になりがちなことである。ポイントカットの記述にワイルドカードを用いると、多くのジョインポイントが選択されてしまうためにビューが複雑になってしまう。C&Y ではアスペクトをメンバーを拡張するものとみなし、拡張されたメンバーに拡張アイコンをつけることでアドバイスの織り込みを表現してる。

C&Y はアスペクトとクラスのどちらに注目してもよいという部分に重点をおいて開発した。一般的にソフトウェアの変更の際には開発者はどのモジュールに注目すべきかはわかっていない。ソフトウェアの変更は開発者がビューを見てその全体構造を把握し、その上で 1 つまたは複数のモジュールに注目をしてなされるべきである。

3.1　クラス図

C&Y では UML のクラス図を用いて横断的構造を表現している。UML のクラス図を用いることで、ジョインポイントを可視化が容易となる。例えば、ジョインポイントとしてメソッド呼び出しを考える。メソッド A がメソッド B を呼び出していたとすると、「メソッド A がメソッド B を使用している」という使用関係をクラス図で表示することができる。またフィールドの参照も同様にクラス図に表示できる。さらに UML のクラス図には継承関係も表示することができる。アスペクト指向言語ではインタータイプ宣言によって継承関係が変更される可能性があり、クラス図は変更した継承関係をわかりやすく表示できるという利点がある。図 3.1 は次のコードにおけるジョインポイントの表現である。

```
class Callee {
  int field;
  void method(){}
}

class Caller {
  Callee obj;
  void access(){ obj.field = 1; }
  void call(){ obj.method(); }
}
```

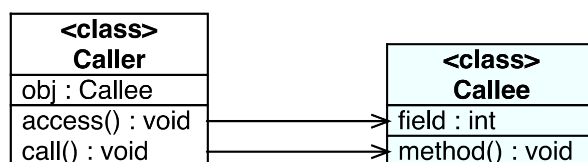

図 3.1: クラス図によるジョインポイントの表現

しかしクラス図を用いて横断的構造を表現しようとするとビューが複雑化してしまう恐れがある。ActiveAspect は 1 つのアスペクトに注目し、アスペクトとの関連性に応じて自動的に抽象化を行うことでこれを防いでいた。しかし、 C&Y ではクラスとアスペクトどちらに注目するかを制限しないため、ActiveAspect とは違った簡易化を図っている。本システムはまずアスペクトとクラスの関係を簡易化して表示する。開発者は簡易表示されたモジュール (ここではクラスとアスペクトを指す) を見て注目するモジュールを決める。開発者によって注目するモジュールが選択されると、本システムはそのモジュールに関連するモジュールとの関係性を表示する。

3.2　簡易表示

本システムはまずアスペクトとクラスの関係を簡易化して表示する。簡易化の方法としてアスペクトはクラスを拡張するものと考え、アスペクトによって拡張されるクラスにはアスペクトによって拡張されたことを示すアイコンをつけて表示する。しかしメソッド呼び出しやフィールド参照な

どでは、caller 側のクラスが拡張されたと考える場合と callee 側のクラスが拡張されたと考える場合がある。例えば上記の Caller クラスと Callee クラスに対して、次のようなアスペクトを適用したと考える。

```
aspect Aspect {
  pointcut callMethod() : call(Callee.method());
}
```

caller 側の拡張と考える場合の表示は図 3.2 のような表示にすべきである。逆に callee 側の拡張と考える場合の表示は図 3.3 のような表示となる。 ど

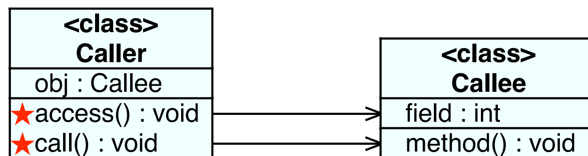

図 3.2: caller 側の拡張と考えた場合の表示

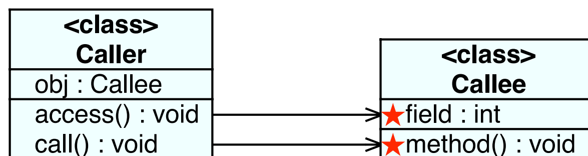

図 3.3: callee 側の拡張と考えた場合の表示

ちらの拡張と考える方が有用であるかはアスペクトによって実現したい機能によって異なる。Event-Based Approach の立場に立つとアスペクトを caller 側の拡張と考えることができ、Extension-Based Approach の立場に立つと callee 側の拡張と考えることができる。Event-Based Approach と Extension-Based Approach については後で詳しく述べる。

さらに注目する必要のないメンバーをビューから削除する機能も作成すべきである。例えば private メソッドは同一クラスにしか影響を与えることはない。そのクラス自身に注目するときは private メソッドの情報も必要であるが、他のクラスに注目しているときに private メソッドまで表示してしまうと必要のない情報のせいでビューが大きくなってしまう。

3.2.1　Event-Based Approach

Event-Based Approach ではジョインポイントをプログラム実行中に生じるイベントと考える。イベントがポイントカットによって指定された制約を満たしているときにアドバイスが実行される。この立場ではメソッド呼び出しにおいてイベントは caller 側で生起するので、アスペクトによって拡張されるクラスは caller 側のクラスである。ロギングの場合、メソッドが呼ばれた場所やタイミングの情報が重要であるが、callee 側の拡張と考えてしまうとメソッドを呼び出す箇所の情報が失われてしまう。このためロギングアスペクトの場合は Event-Based Approach の方が有用となる。

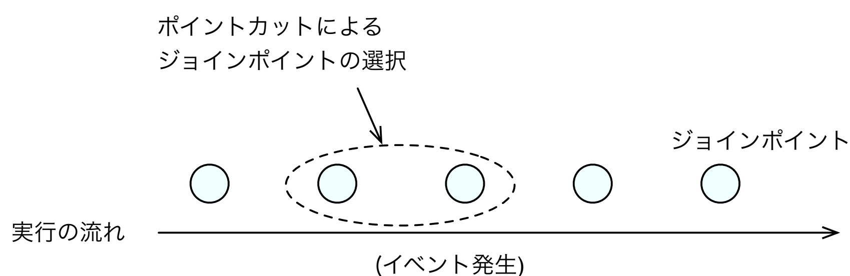

図 3.4: イベントとしてのジョインポイント

3.2.2　Extension-Based Approach

Extension-Based Approach ではアスペクトはクラスのインターフェースを拡張するものであると考える。この立場ではアスペクトによって callee 側のメソッドのインターフェースが拡張されたとみなすことができる。Observer アスペクトでは Extension-Based Approach の方が Event-Based Approach よりも有用となる。Observer アスペクトを AspectJ で書くと次のようなコードになる。

```
class Line extends Shape {
  public void setP1(Point p1){
    ...
  }
  ...
}

aspect ObserverAspect {
  pointcut change() : call(Shape+.set*());
  after() : change() {
    Display.update();
  }
}
```

上記のコードで change() ポイントカットは、Shape クラス (または Shape のサブクラス) の setter メソッドが呼ばれたときを選択している。例えば Line クラスが Shape クラスを継承していたとすると、Line クラスの setP1(Point) メソッドは ObserverAspect によって拡張される。拡張された setP1(Point) メソッドの振る舞いは、setP1(Point) が呼び出されると setP1(Point) の元々の処理の後で画面が再描画されるというものである。Observer アスペクトの場合は拡張されたクラスのインターフェースが直感的なものとなる。

3.3　詳細表示

開発者が 1 つのアスペクトまたはクラスに注目すると、本システムは注目したノードと他のノードとの関係性を詳細表示する。通常の UML でも見られるクラス間の使用関係の他に、ポイントカットによって選択されたジョインポイントと、アスペクトによって拡張されるメンバーにはアスペクトからの矢印を表示する。図 3.5 はアスペクトによって選択されたジョインポイントの表現である。さらにビューが過度に複雑になることを防ぐために、メンバー間の関係のみを表示できるようにもするべきである。

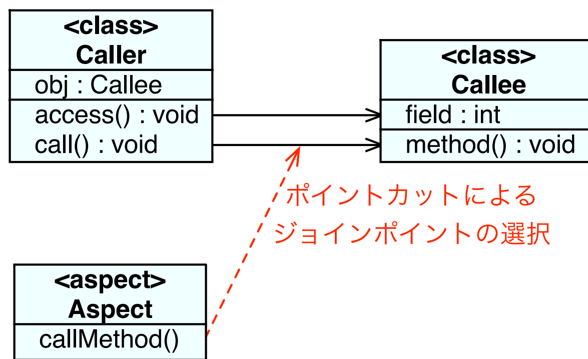

図 3.5: アスペクトによって選択されたジョインポイントの表現

第4章　実装

C&Y は Eclipse プラットフォーム上で動くプラグインとして開発した。対象とするアスペクト指向言語には、ソースコードが公開されていて Java 言語で容易に拡張可能な GluonJ を選択した。

4.1　Eclipse プラットフォーム

前章でも述べたように Eclipse は統合開発環境である。Eclipse のほぼ全ての機能はプラグインによって構成されている。Eclipse にはプラグイン開発用のプラグインも用意されているため、プラグイン開発者は Eclipse 上で簡単にプラグインを作成し Eclipse に新たな機能を追加することができる。ここでは Eclipse の説明とともに新たなプラグインの作成方法を示す。

4.1.1　アーキテクチャ

Eclipse プラットフォームは Eclipse プラットフォームランタイムのほか、ファイルなどのリソースの管理を行うワークスペースリソースプラグイン、SWT や JFace などユーザーインターフェースを構成するためのツールキットや、新たなユーザーインターフェースを追加するための拡張ポイントを提供するワークベンチプラグインなどが含まれている。Eclipse の Eclipse プラットフォームランタイムを除く機能は全てプラグインによって構成されている。Eclipse プラットフォームランタイムは Eclipse の起動やプラグインの管理を行っている。

プラグインは Eclipse の機能を構成する最小単位である。プラグインを追加することによって Eclipse に機能を追加できる。各プラグインはマニフェストファイルを持つ。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によって他のプラグインから新たな機能を取り入れることができる。plugin.xml にはプラグインが提供する拡張ポイントやプラグインがコントリビュートする拡張ポイントを記述する。

Eclipse の機能はプラグインによって構成されるため、Eclipse はいくつかのプラグインを含んだ Eclipse SDK (Eclipse Software Development

Kit) という形で配布されている。Eclipse SDK には Java 言語による開発をサポートする JDT (Java Development Tools) やプラグイン開発をサポートする PDE (Plug-in Development Environment) が含まれる。

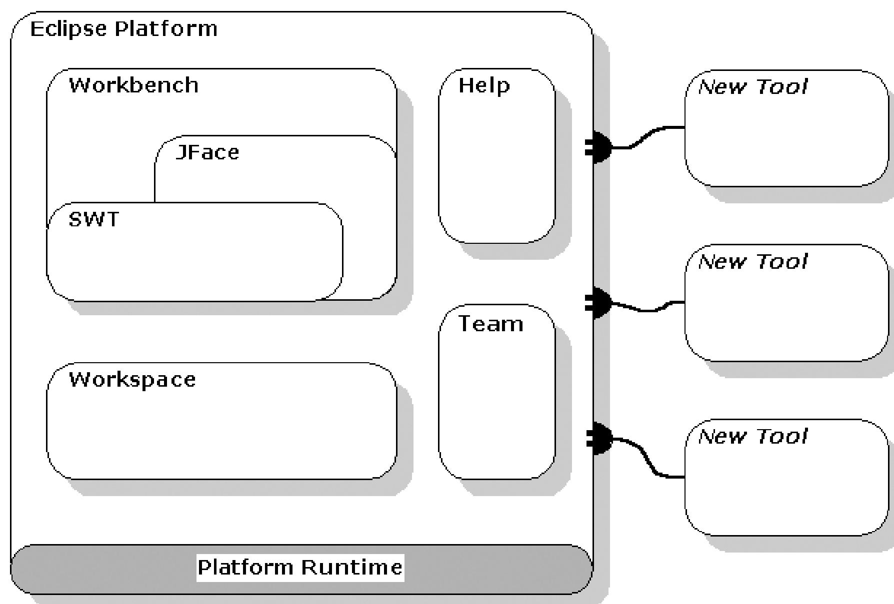

図 4.1: Eclipse のアーキテクチャ

4.1.2　ワークベンチ

多くのプラグインは Eclipse プラットフォームのワークベンチプラグインが提供する拡張ポイントをコントリビュートすることによって Eclipse に新たなユーザーインターフェースを追加する。ワークベンチウィンドウはトップレベルのウィンドウであり、ワークベンチはワークベンチウィンドウの中のエディタ、ビュー、パースペクティブを管理する。エディタは文章の編集や、ブラウザとして使用される。ビューは、各種情報のナビゲーションや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。パースペクティブは画面のレイアウト、ビューやアクションを定義する。

ワークベンチは、既存のビューおよびエディタへの振る舞いをコントリビュートできる拡張ポイントや、新規のビューおよびエディタをコントリビュートできる拡張ポイントを定義するワークベンチのエディタは EditorPart クラス、ビューは ViewPart クラスに対応している。つまりエ

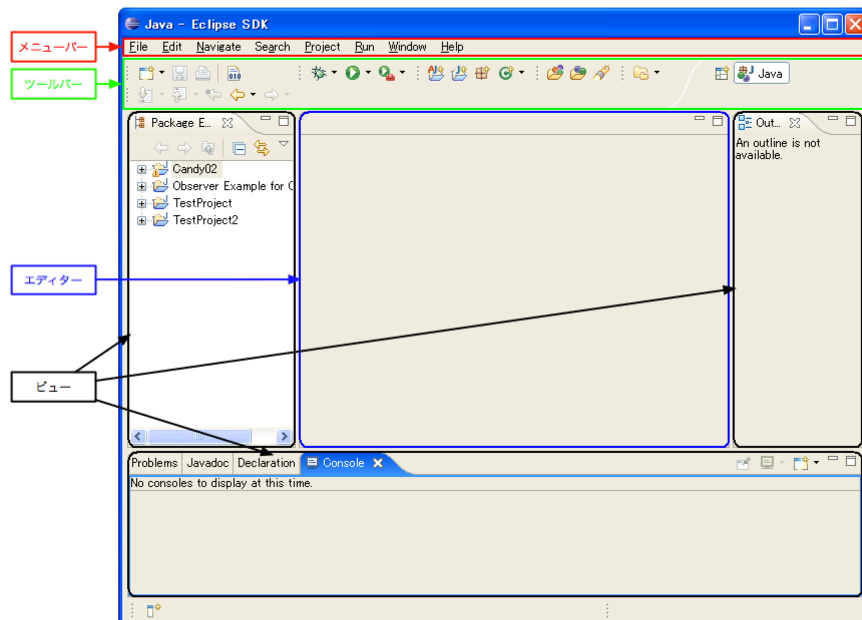

図 4.2: ワークベンチウィンドウ

ディタを新たに追加する場合は適切な拡張ポイントを選択し EditorPart のサブクラスを定義すればよい。

org.eclipse.ui.editors

Eclipse に新たなエディタを定義する。エディタには拡張子やファイルのパターンを関連付けることができる。

org.eclipse.ui.editorActions

エディタのツールバーまたはメニュー選択でアクションを定義する。アクションを追加するエディタは ID によって指定される。

org.eclipse.ui.views

Eclipse に新たなビューを定義する。定義したビューはメニューバーの Window から開くことができる。

org.eclipse.ui.viewActions

ビューのツールバーまたはプルダウンメニューにアクションを定義する。エディタにアクションを追加する場合と同様に、アクションを追加するビューは ID によって指定される。

4.1.3　マニフェストファイル

新たなプラグインを作成するにあたりマニフェストファイルの作成は必須となる。マニフェストファイルには次のようにコントリビュートする拡張ポイントなどの情報が含まれる。

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension point="org.eclipse.ui.editors">
    <editor class="candy.part.CandyEditor"
            filenames="candy.diagram"
            id="CandyEditor"
            name="CandyEditor"/>
  </extension>
</plugin>
```

実際にはマニフェストファイルは PDE の提供しているプラグインマニフェストエディタを用いて編集するため、プラグイン開発者は XML の編集法を知らずとも開発を行うことができる。図 4.3 はプラグインマニフェストエディタの使用例である。エディタにはプラグインの基本情報を編集するための Overview、他のプラグインとの依存関係を編集する Dependencies などのタブがある。Dependencies には、作成するプラグインに必須となるプラグインの情報などを定義する。例えばワークベンチに新たに機能を追加したい場合は必須プラグインに org.eclipse.ui を追加することになる。必須プラグインに選んだからといってコントリビュートする必要はないため、フレームワークを提供するプラグインを必須プラグインに追加できる。次に Extensions タブを開いてコントリビュートする拡張ポイントを選択する。こうして開発者は選択した拡張ポイントにコントリビュートすることが可能となる。

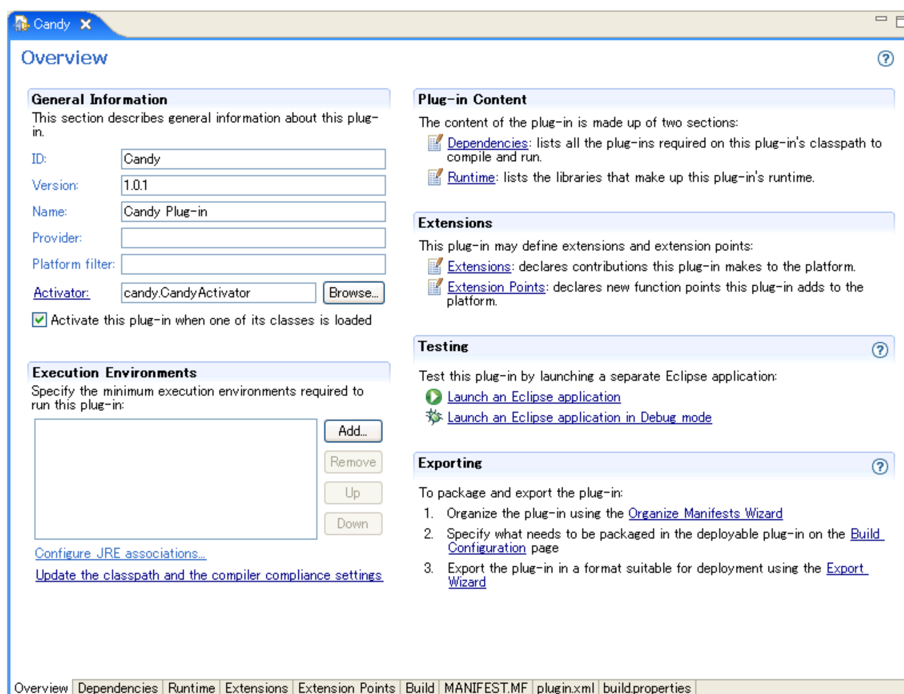

図 4.3: プラグインマニフェストエディタ

4.2　GEF

GEF(Graphical Editing Framework) はグラフィカルなエディタの作成を助けるフレームワークである。プラグイン開発者は org.eclipse.gef を必須プラグインに追加することによって GEF の提供するフレームワークを使用できる。GEF は MVC(Model-View-Controller) アーキテクチャの概念に基づいて設計されている。MVC アーキテクチャとは、プログラムをモデル・ビュー・コントローラに分類して設計を行う概念である。モデル・ビュー・コントローラの役割を以下に示す。

- **モデル**
  アプリケーションが扱う領域のデータモデルを表現したもの。通常モデルにはアプリケーションの主となる計算部分を含める。GEF ではモデルは Object 型であり、プラグイン開発者に都合のよいものをモデルとして選択することができる。

- **ビュー**
  プログラム中に含まれるユーザーインターフェースの要素。画面への出力などを扱う。GEF ではビューは org.eclipse.draw2d.IFigure を実装したクラスが担当する。

- **コントローラ**
  プログラム中でユーザーからの入力をモデルに通知する部分。ビューとモデルを繋ぐものと捉えることもできる。GEF ではコントローラは EditPart を実装するクラスが担当する。

GEF には Eclipse のワークベンチプラグインが提供する IEditorPart を実装した GraphicalEditor クラスが含まれていて、プラグイン開発者は GraphicalEditor クラスを継承したクラスを定義することで Eclipse にグラフィカルなエディタを追加することができる。

GEF を用いたプラグイン開発ではまずモデルを定義し、モデルに対応した EditPart を実装するクラスを定義する。EditPart ではモデルから対応するビュー (IFigure オブジェクト) を作成する。ユーザーからの入力は EditPart がコマンドとして受け取って処理をする。

C&Y ではモデルはプログラムを構成する要素であり、クラス、アスペクト、メソッド、ポイントカットなどに対応するモデルを作成した。さらに依存関係もモデルに含まれるため、クラス間の継承関係、アスペクト間の依存関係、メンバー間の使用関係、ポイントカットによるジョインポイントの選択なども対応するモデルを作成した。クラスやアスペクトに対応するビューはノードであり、メンバーはノード内にラベルをビューとして定義する。依存関係に対応するビューは矢印である。

4.3 GluonJ

GluonJ [7, 9] は Java の文法内でアスペクト指向を実現しているアスペクト指向言語である。@Glue アノテーションが付いたクラス (以下、@Glue クラス) をアスペクトとみなす。@Glue クラスのフィールドにポイントカットとアドバイスの組を記述し、@Refine アノテーションがついた内部クラス (以下、@Refine クラス) を定義することでインタータイプ宣言を実現している。

GluonJ ではポイントカットとアドバイスが 1 対 1 で対応していて、アドバイスは Pointcut 型のフィールドのアノテーションとして記述される。@Glue クラスには Pointcut 型ではないフィールドを作成することはできず、さらに Pointcut 型のフィールドには必ずアドバイスが対応していなければならない。Pointcut 型のオブジェクトは、 Pointcut または Pcd クラスで宣言されたファクトリメソッドによって取得できる。Pointcut と Pcd クラスの提供しているファクトリメソッドを表 4.1 に示す。GluonJ では表 4.1 で取得できるポイントカットを、Pointcut オブジェクトが持つフィールド、and, or, not を用いて組み合わせることができる。アドバイスを実行するタイミングは Before, After, Around のどれかを指定する。

表 4.1: GluonJ のポイントカット例

ポイントカット指定子	選択されるジョインポイント
call(String method)	method にマッチする メソッドの呼び出し時
get(String field)	field にマッチする フィールドの値を読み込むとき
set(String field)	field で指定された フィールドの値を書き込むとき
within(String clazz)	実行スレッドが clazz に マッチするクラス内にいる間
within(String method)	実行スレッドが method に マッチするメソッド内にいる間
when(String expression)	Java 言語で書かれた expression が true であるとき
annotate(String annotation)	annotate にマッチする アノテーションがついた メソッドまたはフィールドが アクセスされている間
cflow(String method)	method にマッチする メソッドを実行している間

Before, After, Around はそれぞれジョインポイントの事前、事後、その時点を表す。

GluonJ では @Refine クラスによりインタータイプ宣言を実現している。@Refine クラスにメンバーを宣言すると、GluonJ は @Refine クラスのスーパークラスにそのメンバーを追加する。@Refine クラスがスーパークラスのメンバーをオーバーライドすると、オーバーライドされたメンバーがフィールドならば初期値が変更され、メソッドならばメソッド本文が上書きされる。GluonJ によるアスペクトの記述は以下のようになる。

```
@Glue class LoggingGlue {
  // アドバイスとポイントカット
  @Before("{ System.out.println(‘call‘); }")
  Pointcut pc = Pcd.call("Point#getX()");

  @Refine static class ExPoint extends Point{
    int newField = 0; // Point クラスに newField を追加
  }
}
```

また、GluonJ では @Glue クラス内から他の @Glue クラスを読み込むことができる。この場合は @Glue クラスに読み込みたい @Glue クラス型のフィールドを宣言すればよい。Logging アスペクトと Tracing アスペクトを読み込む AllGlue アスペクトは以下のように記述される。

```
@Glue class AllGlues {
    @Include Logging glue0;
    @Include Tracing glue1;
}
```

上記の AllGlues アスペクトの織り込みを行うことで、Logging アスペクトと Tracing アスペクトも織り込まれる。このように GluonJ ではアスペクト同士の静的な依存関係も存在する。

GluonJ は Javassist [10] を用いてバイトコード変換を行うことでアスペクト指向を実現している。GluonJ ではユーザーによって定義された @Glue クラスの情報を保持する Gluon オブジェクトが作成される。作成された Gluon オブジェクトに対して、RefineWeaver がインタータイプ宣言の織り込みを行い、AdviceWeaver がアドバイスの織り込みを行っている。AdviceWeaver は @Glue クラスを表す Gluon オブジェクトと織り込み対象のクラスを表す CtClass オブジェクトからアドバイスの織り込みを行う。CtClass のジョインポイントを全て検査して Gluon オブジェクトのポイントカットとマッチするかを検査し、マッチした場合はその箇所にアドバイスを織り込んでいる。

4.4　クラスファイルの解析

C&Y では Javassist を使って、ビューに含められたクラスやアスペクトのクラスファイルを解析する。クラスファイルの解析が終了すると、ビューに表示されているノードとの依存関係を全て解析する。依存関係によって

は最初から表示するものもあるが、メンバー間の使用関係はビューの複雑化を防ぐために最初は非表示となっている。

4.4.1　クラスファイル

@Glue クラスは、@Glue アノテーションが付いているだけの通常のクラスである。そのため GluonJ ではアスペクトは通常のクラスファイルに保存される。C&Y の実装では、まずビューにドロップされたクラスファイルを Javassist を使って読み込む。クラスファイルからアノテーションを取得し、@Glue アノテーションが付いていればアスペクトとみなし、付いていないものは通常のクラスとみなす。クラスファイルに通常のクラスが宣言されている場合は、クラスを表すモデルを作成し、クラスファイルに定義されたメンバーを追加する。クラスファイルにアスペクトが宣言されている場合は、アスペクトを表すモデルを作成し、内部クラスのクラスファイルから @Refine クラスやポイントカット記述を追加する。さらにクラスファイルで宣言されたフィールドから、@Include アノテーションによって読み込まれる子アスペクト名とポイントカットをアスペクトに追加する。

4.4.2　静的な依存関係

C&Y ではクラスファイルの解析が終了すると静的な依存関係の解析に移る。ここでいう静的な依存関係とは、クラス間の関係ならば継承関係を、アスペクト間の関係ならば @Include によって生じた依存関係を指す。したがって通常のクラスであればその親クラスを、@Glue クラスであればフィールドを解析することで静的な依存関係を調べることができる。

静的な依存関係においては間接的に依存が生じる場合も表示すべきである。例えば図 4.4 のようなモジュール間の依存関係を考える。例えばモジュールをアスペクト、依存関係が @Include の依存関係だとすると、アスペクト B がビューに読み込まれていなかったとしても、アスペクト A のアドバイスを織り込むとアスペクト C のアドバイスも織り込まれる。モジュールがクラス、依存関係が継承関係だったとすると、クラス A にアドバイスが織り込まれた場合にはクラス C のメンバーの振る舞いも変化する。このように静的な依存関係は間接的であってもモジュールに大きく作用する。したがって静的な依存関係では間接的な依存関係も表示するべきである。

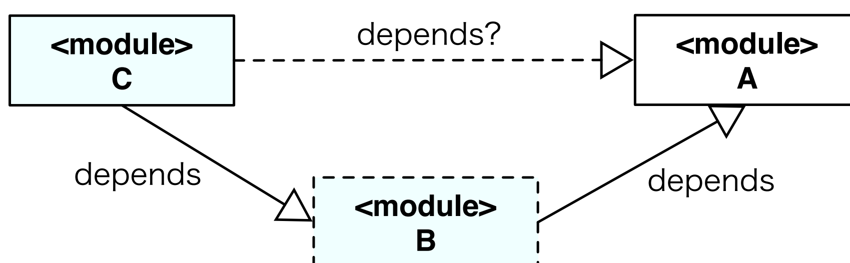

図 4.4: アスペクトの間接的な依存関係

4.4.3　使用関係

読み込まれたメソッドは使用関係を解析するために CtBehavior 型の変数を保持する。C&Y は Javassist が提供する CtBehavior を解析するビジターを使い、メソッド呼び出しとフィールド参照を取得する。callee 側のクラスが既にビューに含まれていた場合は使用関係を作成する。

使用関係は静的な依存関係とは異なり、間接的な使用関係はビューに含めない。C&Y において使用関係はジョインポイントとして用いられる。C&Y の実装どおり Event-Based Approach の立場に立つと、イベントの発生箇所はビューに表示されていないクラスであるのにもかかわらず、間接的な使用関係がアスペクトに選択されてしまう。Event-Based Approach ではイベントの発生箇所が重要視されるため、この場合にジョインポイントの選択は行われるべきではない。したがって C&Y では間接的な使用関係はビューに含めていない。

4.4.4　ポイントカット

ポイントカットが読み込まると、そのポイントカットがジョインポイントを選択するかどうかが解析される。GluonJ では @Glue クラスの全てのポイントカットを一度に検証してしまうため、どのポイントカットがどのジョインポイントを選択しているかを把握することができない。C&Y では、 GluonJ のポイントカットを読み込む処理を内部で実装し、単一のポイントカットから Gluon オブジェクトを作成する。このようにすると、Gluon オブジェクトによって選択されたジョインポイントは、対象となるポイントカットによって選択されることがわかり、GluonJ の織り込み器と同等の処理を行うことができる。

4.4.5　インタータイプ宣言

@Refine クラスが読み込まれると、RefineWeaver を作成し織り込み対象のクラスを特定する。織り込み対象のクラスがビューに存在した場合は、@Refine 内で定義されているメンバーがクラスに宣言されているかどうかを調べる。メンバーがクラスに存在しなかった場合はクラスにメンバーを追加する。

4.5　表示方法の実装

C&Y はメインビューとアウトラインビューからなる。アウトラインビューはメインビューが複雑になったとしてもメインビューの全体像を表示する。アウトラインビューを作成したのは、多くのモジュールをビューに含めた場合にビューが肥大化してしまう恐れがあるためである。肥大化したビューから開発者が見たい部分を探すには、アウトラインビューでメインビューに表示したい部分にフォーカスをあてればよい。

4.5.1　簡易表示の実装

3.2 節で AspectJ を用いて記述したアスペクトは GluonJ で次のように書ける。ここで Before アドバイスを指定しているのは、GluonJ ではポイントカットのみの記述ができないためである。

```
@Glue class Aspect {
  @Before("{}")
  Pointcut callMethod = Pcd.call("Callee#method()");
}
```

C&Y では Aspect, Caller, Callee をビューにドロップすると、最初に図 4.5 のように表示される。まず開発者は変更を行うシステム全体の構造を把握するために多くのモジュールをビューに含めると予測される。その際にメンバーを表示してしまうだけでビューがかなり大きくなってしまうため、この段階では全てのメンバーが非表示になっている。しかしこの時点で Caller クラスに拡張アイコン (■) がついていることがわかる。C&Y では Caller クラスのメンバーで拡張されているものが 1 つでもあった場合に Caller クラスが拡張されていることをアイコンで表示している。これによって開発者は Caller クラスのメンバーが拡張されていることを把握できるため、アスペクトによる影響を調べるときは Caller クラスのメンバーを表示すればよい。

図 4.5 からは読み取れないが静的な依存関係はこの段階で表示する。静的な依存関係はメンバー単位ではなくクラスやアスペクトという単位で発生するので、表示したとしてもビューが複雑化する可能性が低いためである。

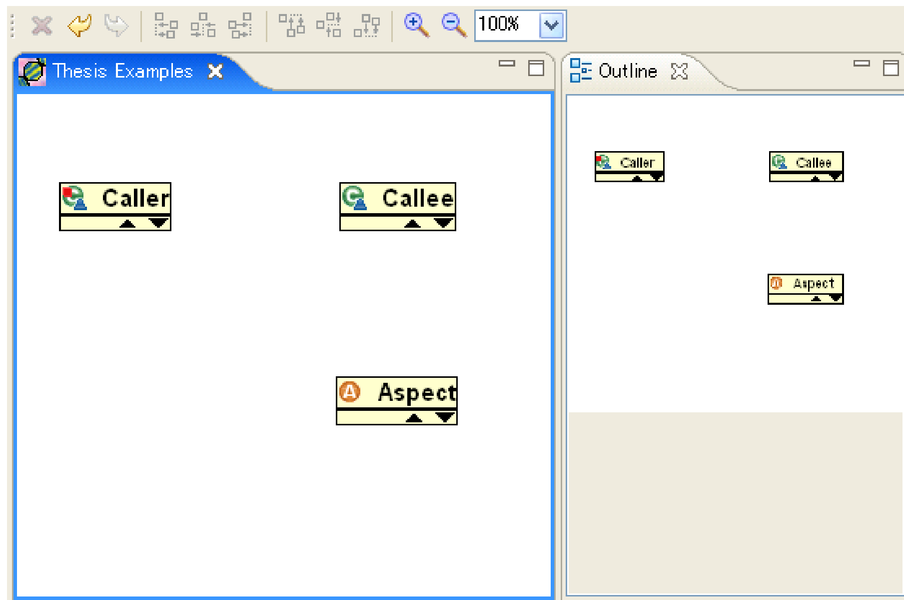

図 4.5: 簡易表示されたクラスとアスペクト

4.5.2　アイコン表示の実装

次に各モジュールのメンバーを表示したビューが図 4.6 である。Aspect によって拡張された call() メソッドが表示されたため、アイコンが Caller クラスから call() メソッドに移動している。もし開発者が call() メソッドを変更しないのであれば、これ以上ビューを詳細化する必要はない。しかし call() メソッドを編集する場合は、Caller とその他のモジュールとの関連性を理解する必要がある。

メンバーが拡張されているかどうかの判断は、メンバーが選択されたジョインポイントを持っているかどうかで判断している。選択されないジョインポイントはアスペクトにいつ選択されても良いように読み込んではいるが、アスペクトによって選択されないジョインポイントは表示しない。これは選択されないジョインポイントを表示することによってビューが複雑になってしまうためである。また C&Y の目的は横断的関心事の表現であるから、選択されないジョインポイントを表示する必要もない。

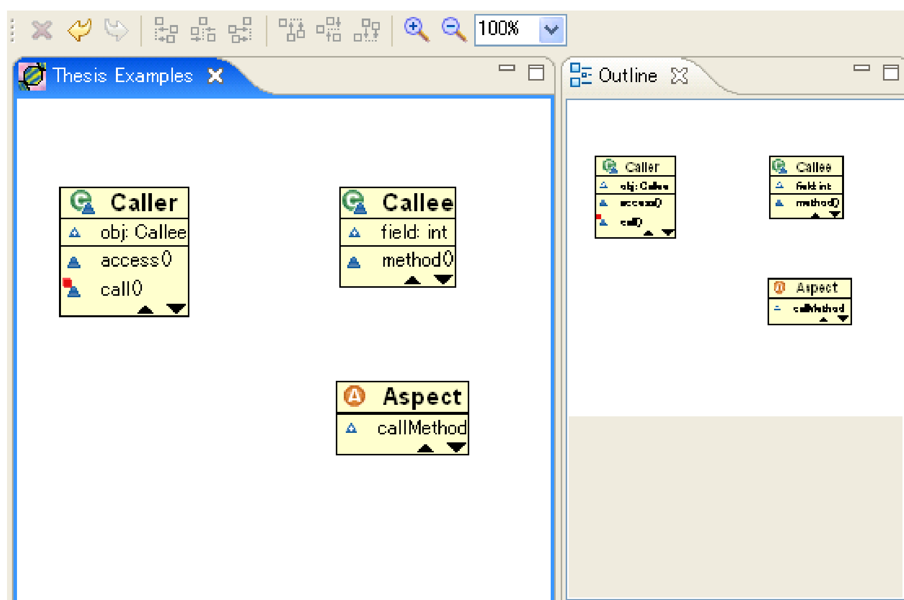

図 4.6: アイコン化されたアスペクトによるメンバーの拡張

4.5.3　詳細表示の実装

図 4.7 は Caller クラスと他のモジュールとの関係性を表示したビューである。これを見ると call() メソッドを変更する際には Callee の method() はもちろん、Aspect の callMethod ポイントカットにも注意を払う必要があるとわかる。このように開発者はビューから注意を払わなければならない箇所をすぐに把握することができる。

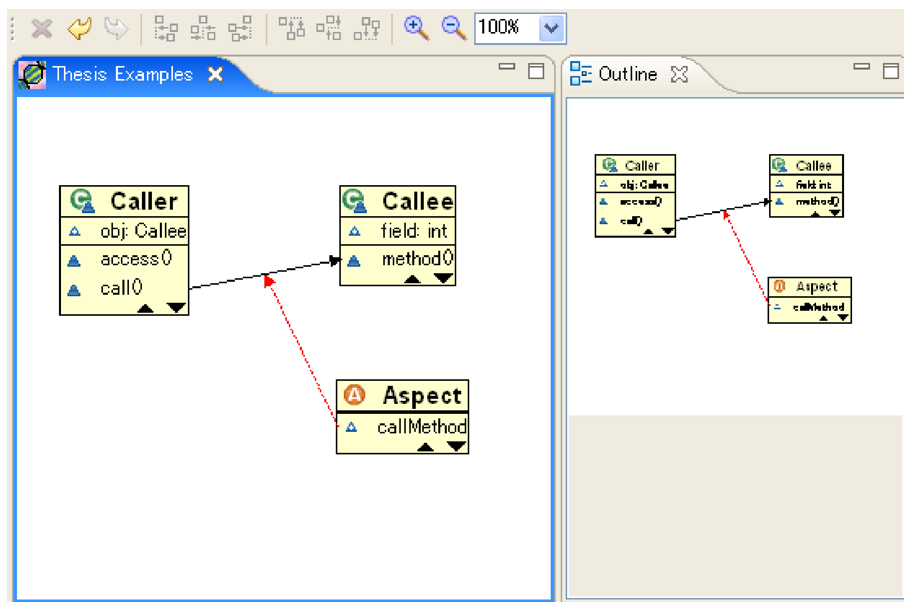

図 4.7: Caller クラスの詳細表示

第5章　応用例

今回は簡単な Observer アスペクトを含むソフトウェアを作成し、 C&Y の検証を行った。この章では開発者がこのソフトウェアを変更することを想定し、 C&Y の有用性を示す。

5.1　Observer アスペクト

図 5.1 は変更前のサンプルソフトウェアの動作である。現在このソフトウェアは点の描画しかサポートしていない。マウスを押したとき、ドラッグしたとき、クリックしたときに点が作成され、作成された点が描画される。

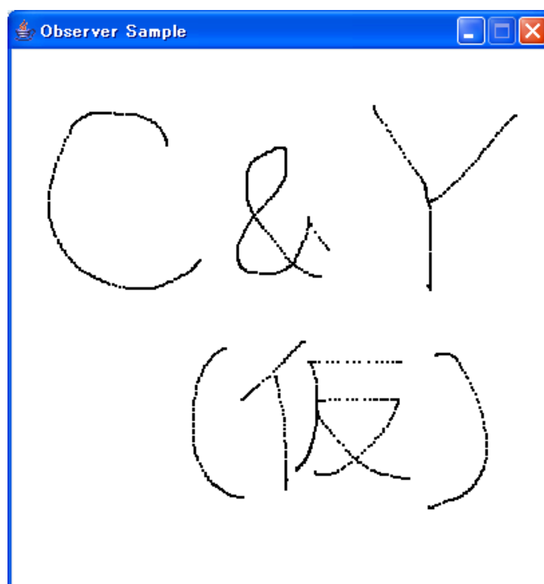

図 5.1: サンプルソフトウェアの実行結果

このソフトウェアでは、画面の再描画を UpdateDisplay アスペクトで行っている。変更前のソフトウェアの構造を C&Y で表現した図が、図 5.2 である。まだ簡易表示しかされていないが、アスペクトによって Display

クラスの継承関係が変更され、Display クラスが Observer インターフェースを実装していることがわかる。この段階で Observer クラスの変更がアスペクトや Display クラスに影響を及ぼすことが把握できる。

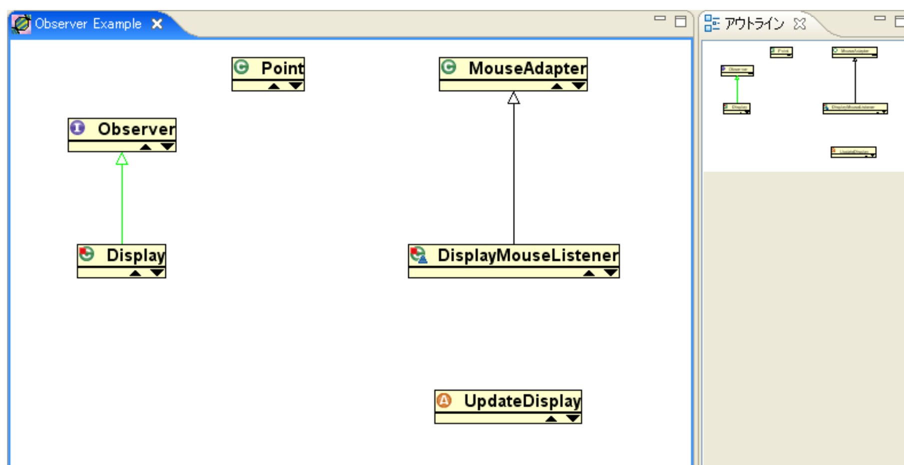

図 5.2: 変更前のソフトウェアの構造 (簡易表示)

5.2　C&Y を用いない場合

このソフトウェアを拡張して、点だけでなく線や長方形、円を書けるようにする。これらのオブジェクトは 2 点をクリックすることによって作成され、描画される。また、さらに図形を追加する可能性を考えて、図形クラスは Shape インターフェースを実装することにする。Shape インターフェースの定義は次のようになる。

```
interface Shape {
  void paint(java.awt.Graphics g);
}
```

Shape インターフェースを用いてソフトウェアを変更した。現在ソフトウェアに含まれるモジュールのうち、変更したモジュールは、 Display, Point, DisplayMouseListener の 3 つである。図 5.3 は変更後のソフトウェアの構造を C&Y で表現した図である。簡易表示の段階ではあるが、これを見ると DisplayMouseListener にあった拡張アイコンが消えていることが分かる。実際にアスペクトを織り込み、ソフトウェアを動かしてみても何も表示されない。これは、 UpdateDisplay アスペクトがソフトウェアにうまく動作していないことを示している。

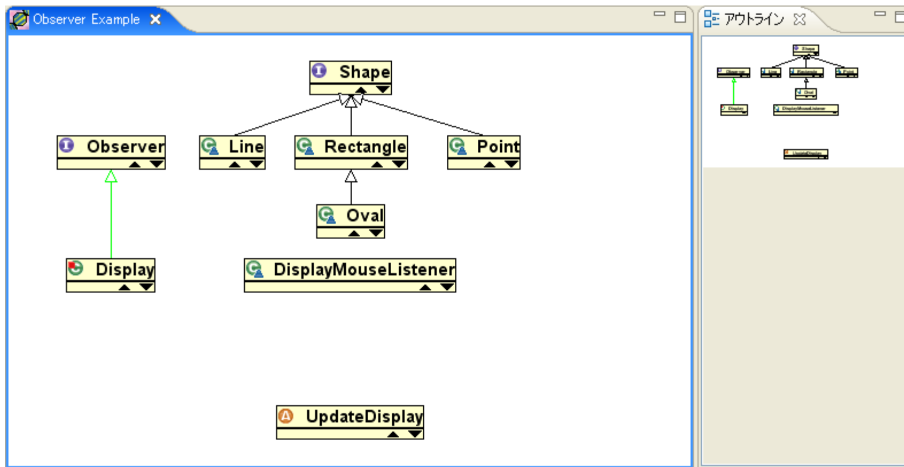

図 5.3: クラス変更後のソフトウェアの構造 (簡易表示)

5.3 C&Y を用いた場合

C&Y を用いた上でソフトウェアの拡張を行う場合を考える。図 5.4 は C&Y を用いて、変更前のソフトウェアの構造のアスペクトに関わる部分を表示した図である。完全に詳細を表示した図ではないため、ポイントカットからジョインポイントに矢印はのびていないが、アスペクトが 1 つしかなく、ポイントカットも 1 つしかないため、この図で全ての情報が得られている。また、 Display クラスの update() メソッドの上にマウスを乗せると、図のようにどこから拡張されているかがわかる。

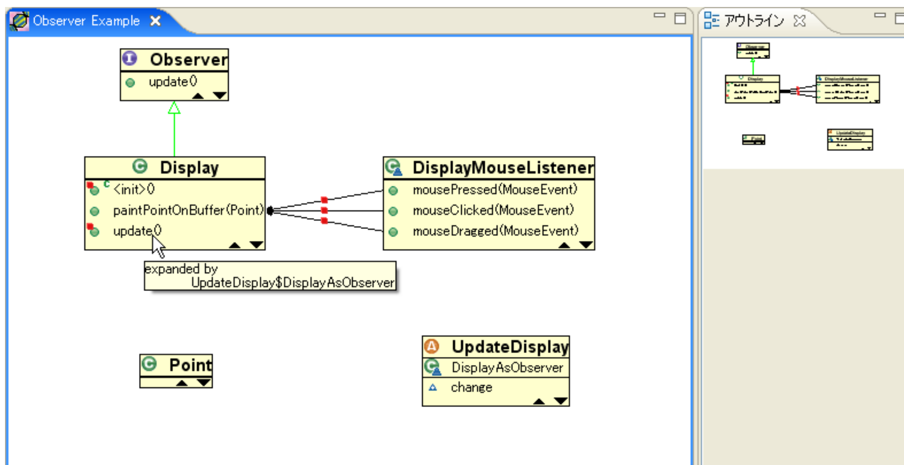

図 5.4: 変更前のソフトウェアの構造 (詳細表示)

この図を見ると、今回の変更で注意をすべきは、Display クラスの paint-PointOnBuffer(Point) の変更、DisplayMouseListener クラスの 3 つのメソッドの変更である。Point クラスはアスペクトと何の関わりもないため、変更の際にアスペクトに注意を払う必要はない。図 5.5 は、変更後の Display クラスの public メソッド一覧である。paintPointOnBuffer(Point) メソッドが削除されていることがわかる。

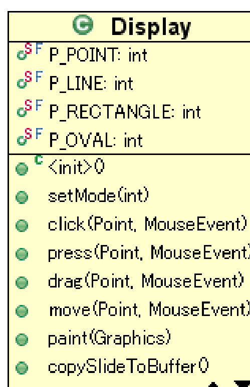

図 5.5: 変更後の Display クラスの public メソッド

Display クラスを変更する際に、C&Y を用いて図 5.6 のような図をみていれば、paintPointOnBuffer(Point) の変更時にポイントカットの記述に注意を払うことができただろう。もしそれでポイントカットの記述を変更しなければならなかったとしても、ソフトウェアを変更した後に描画されないのを見て、デバッグ作業をする手間は省くことができる。

5.4　検証結果

今回は例として Observer アスペクトを用いたソフトウェアを例に取り上げた。前述したように Observer アスペクトは Extension-Based Approach の立場に立って callee 側の拡張と考えた方が有用である。例えば Display クラスの paintPointOnBuffer(Point) メソッドに拡張アイコンが付いていれば、選択されるジョインポイントを表示しなくてもアスペクトによって影響を受けることがわかる。しかし、今回の例でも Display クラスに関連する横断的構造を表示しさえすれば、caller 側の表示だけでも DisplayObserver アスペクトの change ポイントカットによって paintPointOnBuffer(Point) メソッドが影響を受けることがわかった。また、アスペクトによる影響を考慮せずにソフトウェアを拡張した場合に比べると、デバッグ作業を省けるために開発コストが削減されることもわかった。

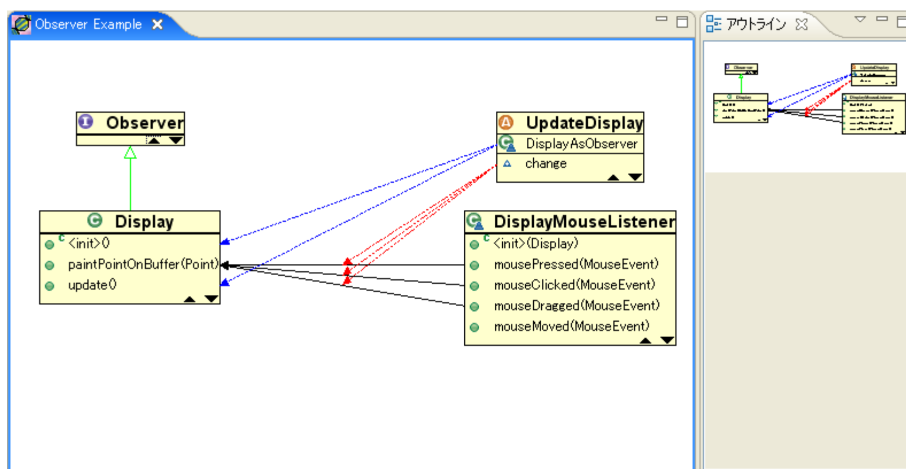

図 5.6: 変更前の Display クラスに注目し横断的構造を表示

第6章　まとめ

本論文では、利用者がクラスとアスペクトのどちらに注目したとしても、関連する横断的構造をクラス図によって視覚化できるツール C&Y を提案した。アスペクトに注意を払わずにクラスの変更を行うと、fragile pointcut problem によってアスペクトが機能しなくなってしまう。しかし、変更したクラスをどれだけ眺めても、アスペクト指向言語が持つ obliviousness によって原因を把握することはできない。また、クラスに注意を払わずにアスペクトを変更した場合にも、ワイルドカードの使用などを要因として、意図しないジョインポイントを選択してしまう可能性がある。C&Y を利用すると、クラスの変更がアスペクトに与える影響と、アスペクトの変更がクラスに与える影響を事前に把握できる。

最後に、現在の実装の問題点を今後の課題として、本論文のまとめとする。

6.1　今後の課題

現在の実装では、 cflow ポイントカットや if ポイントカットのように、ジョインポイントの選択が動的に定まるものについて考慮していない。動的なポイントカットによるジョインポイントの選択には、色を付けて表現したり、ツールチップなどを用いてその旨を開発者に知らせるなどの工夫が必要である。

また、検証で使用したソフトウェアは、ポイントカットによって選択されるジョインポイントが少なく、アスペクトも 1 つしか含まない。今後は複数のアスペクトを含むようなある程度大きなソフトウェアについて C&Y を使用し、簡易表示機能の検証や改良を行いたい。

GluonJ では @Glue クラスに含まれる全てのポイントカットがアドバイスを持つため、現在はポイントカットがジョインポイントに作用しているように表示されている。これは、アドバイスが Before, After, Around の 3 種類の名前しか持たないのに対し、ポイントカットには自由に名前がつけられることを要因としている。ジョインポイントに作用するポイントカットの名前を見ることで、選択されたジョインポイントの大まかな特徴が把握できる。しかし、 AspectJ ではアドバイスを持たないポイント

カットを定義することができる。現在の実装を AspectJ に適用することを考えると、プログラムの挙動に関係のないポイントカットとの関連性も表示してしまい、図が複雑化してしまう可能性がある。したがって、本来はアスペクトをジョインポイントに作用させるよう表示すべきである。

参考文献

[1] AJDT: AJDT project, http://www.eclipse.org/ajdt/.

[2] Andy Kellens, Kim Mens, J. B. K. G.: Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts., *Proceedings of the 20th European Conference on Object-Oriented Programming*, pp. 501–525 (2006).

[3] AspectJ: AspectJ project, http://www.eclipse.org/aspectj/.

[4] Eclipse: Eclipse project, http://www.eclipse.org/.

[5] Gybels K., B. J.: Arranging language features for more robust pattern-based crosscuts., *In Proceedings of the 2nd International Conference of Aspect-Oriented Software Development, AOSD '03*, pp. 60–69 (2003).

[6] Kiczales G., M. M.: Separation of concerns with procedures, annotations, advice and pointcuts., *In Proceedings of the 19th European Confrerence on Object-Oriented Programming*, pp. 195–213 (2005).

[7] Shigeru Chiba, R. I.: Aspect-Oriented Programming beyond Dependency Injection., *In Proceedings of the 19th European Conference on Object Oriented Programming*, pp. 121–143 (2005).

[8] Wesley Coelho, G. C. M.: Presenting Crosscutting Structure with Active Models, *Proceedings of the 5th International Conference on Aspect-Oriented Software Development AOSD '06*, pp. 158–168 (2006).

[9] 千葉滋: GluonJ Home Page, http://www.csg.is.titech.ac.jp/projects/gluonj/.

[10] 千葉滋: Javassist Home Page, http://www.csg.is.titech.ac.jp/~chiba/javassist/.

[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).