平成17年度 修士論文

統合開発環境のための
アスペクト指向システム

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 04M37053

薄井 義行

指導教員

千葉 滋 助教授

平成18年1月30日

概要

本論文では、統合開発環境のためのアスペクト指向システム Bugdel について述べる。ソフトウェアの開発、保守を行う際、テスト用のコードやデバッグのためにトレースコードをプログラムに追加することが多い。Bugdel ではアスペクト指向を利用して、これらのコードの追加を支援する。アスペクト指向を利用することで追加コードとプログラムのソースコードを別々に記述でき、さらに pointcut を使ってパターンを記述することで追加コードの実行位置をまとめて指定することができる。

Bugdel はテスト、デバッグに特化したシステムであり AspectJ のような汎用的なアスペクト指向システムには無い機能を提供する。まず、ソースコードの行番号を pointcut で指定する Line、AllLines pointcut を提供する。これにより追加コードの実行位置をソースコードの行番号で指定できる。次に advice ボディ内から pointcut で指定した位置に存在するローカル変数へのアクセスを許可し、ローカル変数に対するリフレクション機能を提供する。これにより、advice を使ってローカル変数のトレースを行える。これらの機能はクラスのモジュール性を壊す可能性があるため汎用的なアスペクト指向システムには無い。

また Bugdel は統合開発環境である Eclipse のプラグインとして実装されており、優れたユーザインターフェイスを持つ。まず、GUI を通して選択的に pointcut を指定できる。そのため、AspectJ のように宣言的に pointcut を記述するシステムと比べて、利用するにあたって覚えることが少ない。また、pointcut に影響のあるソースコードの編集を監視する機能を持っている。従来のアスペクト指向システムではソースコードの編集によりユーザの意図に反して pointcut のターゲットを変更してしまうという問題があった。それに対して、 Bugdel では統合開発環境の機能を利用してソースコードの編集を監視し、 pointcut に影響のある編集が行われた場合には警告を出しユーザに知らせることができる。

さらに本稿では Bugdel の応用例としてデバッグモードの無い JVM 上でブレイクポイントを実現する方法を示す。

謝辞

本研究を行うにあたってあらゆる面で支えていただいた方々へ感謝の意を表します。指導教官の東京工業大学 千葉滋 助教授には大学４年生から大学院修士課程までの３年間、研究の方向性や進め方、研究発表についてご指導いただき、大変感謝しております。東京工業大 学光来健一 助手には研究内容や研究発表についてさまざまな意見をいただきました。研究室の先輩方、西沢無我氏、柳澤佳里氏には３年間多くの助言をいただきました。研究室の同輩、石川零氏、日比野秀章氏には大学４年生で同研究室に所属して以来３年間、共に研究に励みさまざまな相談を聞いていただきました。その他、同研究室に所属するみなさまのおかげで充実した研究ができました。薄井雄一氏、幸子氏には学生生活を送るにあたって経済的な面で支援していただきました。感謝しております。

目 次

図 目 次

表 目 次

第1章　はじめに

ソフトウェア開発においてテストやデバッグが行われる。その際、テスト用のコードやデバッグのためのトレースコードなどを開発中のプログラムに追加することが多い。これらの追加コードは開発中のプログラムの本来の機能には関係のないものであるため、ソースコード中に記述するとソースコードの可読性が低下してしまう。また、テストやトレースが終わると、追加コードを取り除く必要があり、取り忘れがあるとプログラムの実行中に不要な処理が行われてしまう。追加コードの挿入、削除を行う際、誤ってその周辺のプログラムを変更してしまう危険性もある。また追加コードの実行位置を一ヵ所ずつ指定しなければならず効率的でない。例えば、ある変数 x にアクセスする度にログ出力を行う場合、変数 x にアクセスしている位置をソースコード中からプログラマが見つけ出しコードを挿入しなければならない。この作業には手間がかかり、入れ忘れという問題もある。

そこで我々はアスペクト指向を利用してテスト、デバッグ用コードをプログラムに追加する Java 開発環境 Bugdel を提案する。アスペクト指向では主にコードの実行位置を指定する pointcut と実行するコードを指定する advice によって追加コードを指定する。アスペクト指向を利用することで、テストやデバッグのための追加コードと開発中のプログラムのソースコードを別々に記述することができる。さらに、pointcut を利用してパターンを記述することでコードの実行位置をまとめて指定することができる。そのため、上で述べた問題を解決することができる。既存のアスペクト指向システムとしては Java を言語拡張したアスペクト指向言語である AspectJ が有名であるが汎用的なシステムであるためテスト、デバッグには不向きな点がある。一方、Bugdel はテスト、デバッグに特化したアスペクト指向システムであり、AspectJ のような汎用的なアスペクト指向システムには無い機能を提供する。具体的にはソースコードの行番号を pointcut で指定する Line pointcut、メソッド内の全ての行番号を指定する AllLines pointcut を提供する。さらに、advice ボディ内から pointcut で指定した位置 (joinpoint) に存在するローカル変数へのアクセスを許可し、ローカル変数に対するリフレクション機能を提供する。これらの機能はクラスのモジュール性を破壊する危険性があるため AspectJ のよう

な汎用的なアスペクト指向システムでは提供していない。それに対して Bugdel ではクラスのモジュール性よりもテストやデバッグのための利便性を考え提供する。

Bugdel は統合開発環境である Eclipse のプラグインとして実装しており有用なユーザインターフェイスも提供する。まず、GUI を通して pointcut の指定が可能である。例えば、ソースコード上のクラス名やメソッド名をマウスでクリックしたり、Bugdel が提供するソースコードブラウザを利用して検索を行いながら pointcut を指定することができる。これにより Bugdel を利用するにあたって覚えることが AspectJ よりも少なくなり、pointcut を指定する手間も少なくなる。次に、pointcut に影響のあるソースコードの変更を監視する機能を持っている。従来のアスペクト指向システムでは pointcut 記述がリファクタリングなどのソースコード編集に弱いという問題があった。例えば、pointcut で setX メソッドの呼び出し位置を指定する場合、pointcut の引数には “setX” というのメソッド名を文字列で指定する。そのため、リファクタリングにより setX メソッドの名前を “setterX” に変更した場合、変更したメソッドが pointcut の対象にならなくなってしまう。また、メソッド名が “set” ではじまるメソッドの呼び出し位置を call(* *.set*(int)) pointcut で指定されている場合、ユーザが新たに setZ(int) メソッドを追加すると、setZ メソッドも pointcut の対象になる。しかし、setZ メソッドが pointcut の対象になることをユーザが期待していないかもしれない。このように、ユーザの意図に反してソースコード編集により pointcut の対象となるメソッドやフィールドが変更されてしまう可能性がある。そこで、Bugdel では pointcut に影響のあるソースコード編集を監視し、影響のある変更があった場合には警告を出しユーザに知らせる。

本稿の残りは、次のような構成からなっている。2 章でテスト、デバッグコードを記述するツールとその問題点について述べる。3 章では、Bugdel の利用方法と実装方法、4 章では Bugdel の応用例としてブレイクポイントのエミュレート、5 章では Bugdel の weaver の実験と advice の記述時間について AspectJ/AJDT との比較を述べる。そして 6 章でまとめる。

第2章　テスト、デバッグコードを記述する手法とその問題点

ソフトウェアの開発時にはテスト、デバッグ作業が行われ、その際、さまざまなコードを開発中のプログラムに追加する。この章では、テスト、デバッグコードを指定するための手法について説明し、既存のシステムの問題点を述べる。

2.1　ソースコードへの記述

デバッグを行う際にはログ出力（トレース）コードを追加する。デバッグの方法には大きく分けて２種類の方法がある。１つめはログ出力を行ってバグの位置を特定する方法、２つめはデバッガのブレイクポイントを利用する方法である。ログ出力はデバッグ作業の初期段階で利用される。初期段階ではバグの位置がプログラム全体のどこにあるのか検討が着いていない。そのため、プログラムのさまざまな位置でログ出力を行ってバグの存在するのおおまかな位置を特定する。その後、デバッガのブレイクポイントを利用してプログラムの実行、停止を繰り返してバグの詳細な位置を特定する。もしも、初期段階でブレイクポイントを利用した場合、さまざまな位置でブログラムの実行、停止を繰り返さなければならず効率的ではない。ループ文の中で変数を確認する場合も同様にプログラムの停止、再開を繰り返さなければならず、この場合にもログ出力を行う。ログ出力の場合、プログラムの実行履歴が残るため時間を逆行して調査することができるという利点もある。その他、ブレイクポイントではなくログ出力を利用する例としてマルチスレッドプログラムのデバッグがある。マルチスレッドプログラムの場合、特定のスレッドを停止させててしまうとプログラムの挙動が変わってしまい、新たなバグが発生してしまうという問題がある。そこで、ログ出力を行いスレッドを停止させずにデバッグを行う。また、デバッガ（ブレイクポイント）の利用が困難な場合にもログ出力を行ってデバッグをする。例えば、servlet のようなサーバーサイドプログラムでブレイクポイントを利用する際、ミドルウェアである servlet エンジン tomcat 自体をデバッガで起動する必要がある。プログラムを実行す

る JVM にブレイクポイントが実装されていない場合もある。以上のようにデバッグの際にはログ出力を行う必要があり、その際、プログラムにトレスコードを追加する。

テストを行う際にもプログラムにコードを追加する。例えば、例外発生時の挙動をテストするために、特定の位置に例外を投げるコードを記述したり、あるメソッドが実行されるかどうかを判断するために、boolean 変数のフラグを true にするコードをメソッドの先頭に記述する。

2.1.1　問題点

テスト、デバッグコードをソースコードに記述するためのツールに次のようなものがある。C 言語には条件付コンパイルがありコンパイル時のオプションによってテスト、デバッグコードをプログラムに追加するかどうかを決められる。Java の assert 文も同様にコンパイル時のオプションによって assert 文の追加、削除を決められる。J2SE の標準 API に含まれる java.util.logging パッケージ [28] や apache jakarta の log4j[5] などのロギング API を利用することでログ出力のオン、オフを容易に変更できる。条件付コンパイル、assert 文、ロギング API を利用することで開発したソフトウェアでテスト、デバッグコードなどの不要な処理が実行されるのを防ぐことができる。

しかし、このようなツールを使った場合でもプログラムのソースコードに追加コードを記述する必要がある。そのため、追加コードによりソースコードの可読性が悪くなるという問題がある。統合開発環境を使ってデバッグモードではデバッグコードを表示させ、通常モードではデバッグコードを表示させなくするツールもあるが、チームでソフトウェア開発を行っている場合には、さまざまプログラマがさまざまな開発環境を利用して開発を行っており、全員がデバッグコードを隠蔽する開発環境を利用できるとは限らない。そのため、追加コードを取り除く必要がある。ソースコードを公開しているソフトウェア場合にも追加コードを取り除く必要がある。ところがソースコード中から全ての追加コードを探し出すのには手間がかかり負担が大きい。追加コードの挿入、削除を行う際に誤って、その周辺のソースコードを変更してしまう可能性もあり、新たなバグの原因となりえる。

さらに、追加コードを挿入する位置を一ヶ所づつ指定しなければならないという問題もある。例えば、ある変数 x が変更される度に変数の値をログ出力させようとすると、変数 x にアクセスしている場所をソースコード中から探し出し、ログ出力コードを挿入しなければならない。変数 x をさまざまな場所（モジュール）で変更している場合、それら全ての位

置を探し出すのは困難であり効率的ではない。

2.2　アスペクト指向を利用したテスト、デバッグコードの記述

アスペクト指向の利用例としてデバッグ、テストコードの記述が有名である。アスペクト指向を利用することでデバッグ、テスト対象プログラムのソースコードとは別に追加コードを指定することができる。さらに、パターンを記述することで効率よく追加コードの実行位置を指定できる。この節ではアスペクト指向の概要と既存のアスペクト指向システムについて述べる。

2.2.1　アスペクト指向とは

アスペクト指向 [23] とは横断的関心事をモジュール化するプログラミング技法である。プログラミング言語には C 言語、Fortran などの手続き型言語、LISP、ML などの関数型言語、prolog などの論理型言語、Java、Smalltalk、C++ などのオブジェクト指向型言語がある。これらの違いはモジュールの分解の方法である。オブジェクト指向言語では各モジュールをオブジェクトという単位で分解する。

しかし、各言語を使ってある観点からモジュールを分解する場合、モジュールとして分解しきれないものが存在する。例えば、オブジェクト指向言語を使ってシステムを構築する場合、オブジェクトとう観点でモジュールを分解していくが、同期処理やロギング処理などは単一のオブジェクトとしては分解することが困難であり、処理がさまざまなオブジェクト (モジュール) にまたがって存在してしまう。このようにさまざまなモジュールにまたがる処理を横断的関心事と言い、横断的関心事をモジュール化する技法がアスペクト指向である。

アスペクト指向は単体で利用する概念ではなく、既存のプログラミング技法と共に利用し、既存のプログラミング技法に新たにアスペクトというモジュール化の手法を提供するものである。例えば、アスペクト指向言語である AspectJ は Java によるオブジェクト指向開発にさらにアスペクトというモジュール単位を提供する (図 2.1)。このようにアスペクト指向とは既存のプログラミング技法に取って代わるものではなく、既存のプログラミング技法を補完するものである。アスペクト指向システムにはさまざまなものが提案されており、汎用的なアスペクト指向システム [6, 11, 29] と特定の領域に特化した専用のアスペクト指向システム [18, 30, 7] がある。また、プログラムのソースコード中からアスペクトを抽出するための

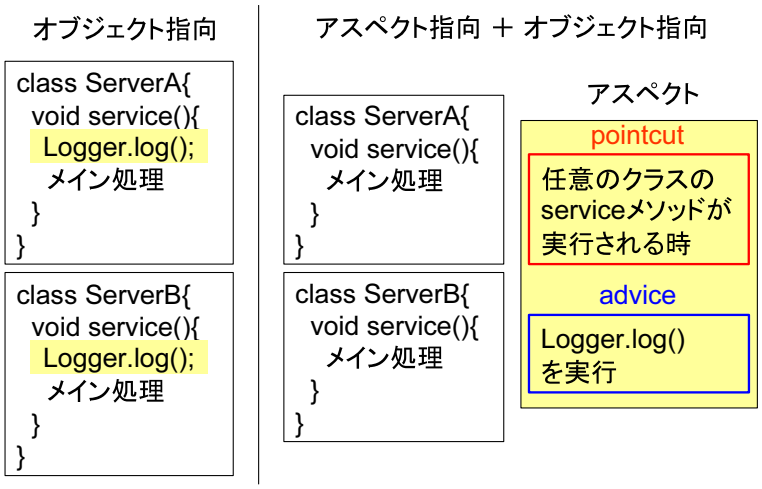

図 2.1: AOP

支援ツール [34, 15] も開発されている。

2.2.2　AspectJ

AspectJ[22, 24] とは Java を言語拡張した汎用的アスペクト指向システムである。AspectJ では class というモジュールの他に aspect というモジュール単位を提供する。aspect には pointcut、advice、inter-type という要素が含まれる。以下に aspect を記述したコード例を示す。

```
01 public aspect LogAspect{
02   int figure.Point.z = 100;
03   public int figure.Point.getZ(){
04     return z;
05   }
06   pointcut p(): call(void figure.Point.setX(int));
07   after() : p(){
08     log();
09   }
10   public static void log(){
11     ...
12   }
13 }
```

上記のコードでは 2 行目でフィールドの inter-type 宣言、3 〜 5 行目ではメソッドの inter-type 宣言、6 行目では pointcut、7 〜 9 行目では advice を定義している。また、aspect には通常のクラスと同様にメソッドやフィールドの定義も行える。以上のようにアスペクト指向では aspect を定義することで class モジュールにさまざまな処理を追加することができる。

AspectJ のコードをコンパイルするには専用の ajc コンパイラが必要になるが生成されるバイトコードは標準のバイトコードであるため通常の JVM で実行が行える。そのため、AspectJ を利用するにあたって JVM の改造は必要ない。AspectJ のコンパイラは初めに class モジュールのバイトコードを生成し、次に aspect モジュールで指定したコードをバイトコード変換を行い class モジュールのに埋め込む (図 2.2)。この埋め込み

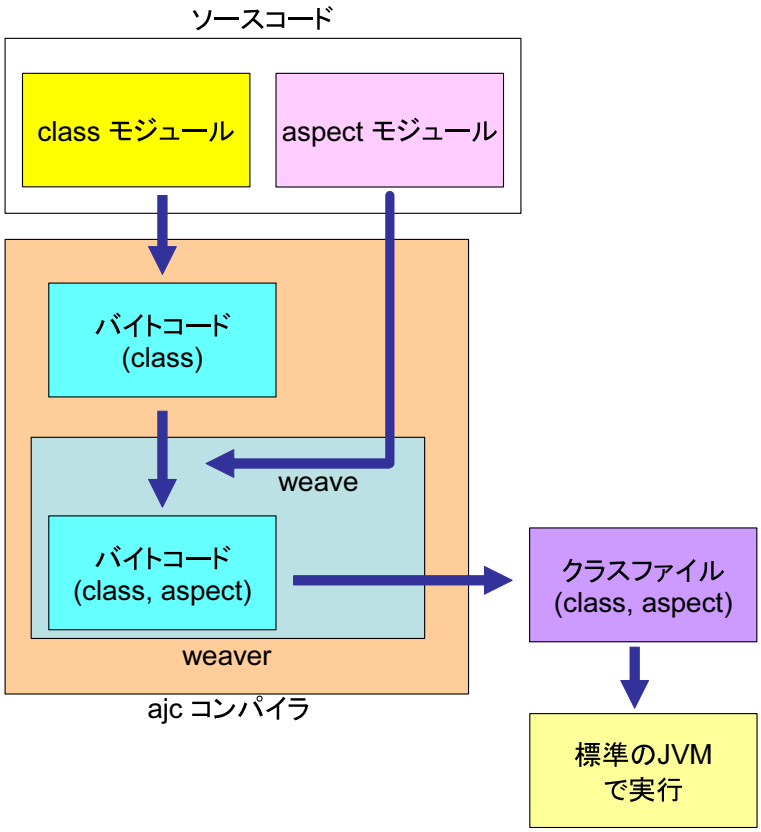

図 2.2: ajc コンパイラのプロセス

の処理を weave という。つまり、ajc コンパイラには class モジュールを生成する通常の Java コンパイラと weave を行う weaver が含まれている。以降この節では、AspectJ のプログラミングモデルの要素である

joinpoint、pointcut、advice、inter-type について詳しく説明する。

Joinpoint

joinpoint とは advice コードを実行することが可能なプログラム中の位置を表す。AspectJ では advice ボディを任意の位置で実行できるわけではなく、フィールドアクセスやメソッド呼び出し、コンストラクタの実行位置などオブジェクト指向プログラム (Java プログラム) において主要なイベントが joinpoint になり、それらの位置で advice ボディを実行することができる。joinpoint はプログラムの実行時に存在するものであるが、ソースコードやバイトコードには各 joinpoint に対応する場所が存在する。これらの場所を joinpoint shadow[16] と言う。例えば図 2.3 プログラム中には、さまざまな joinpoint shadow が存在する。

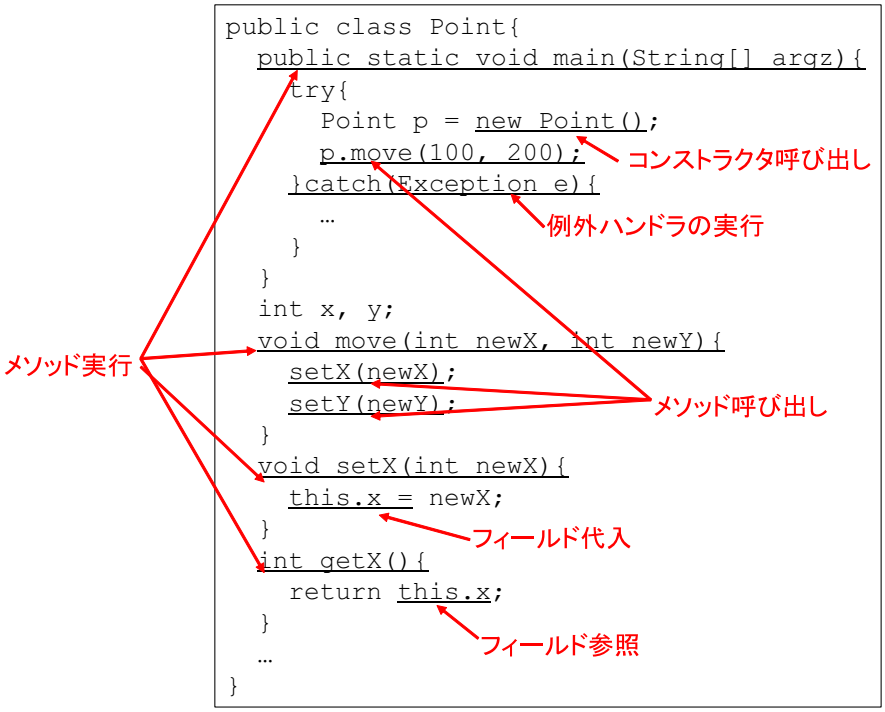

図 2.3: ソースコード中の joinpoint shadow

Pointcut

pointcut とはプログラム中の joinpoint を抽出するものである。プログラム中にはさまざまな joinpoint が存在している。その中から pointcut を

使って条件をもとに必要な joinpoint を抽出する (2.4)。例えば figure.Point

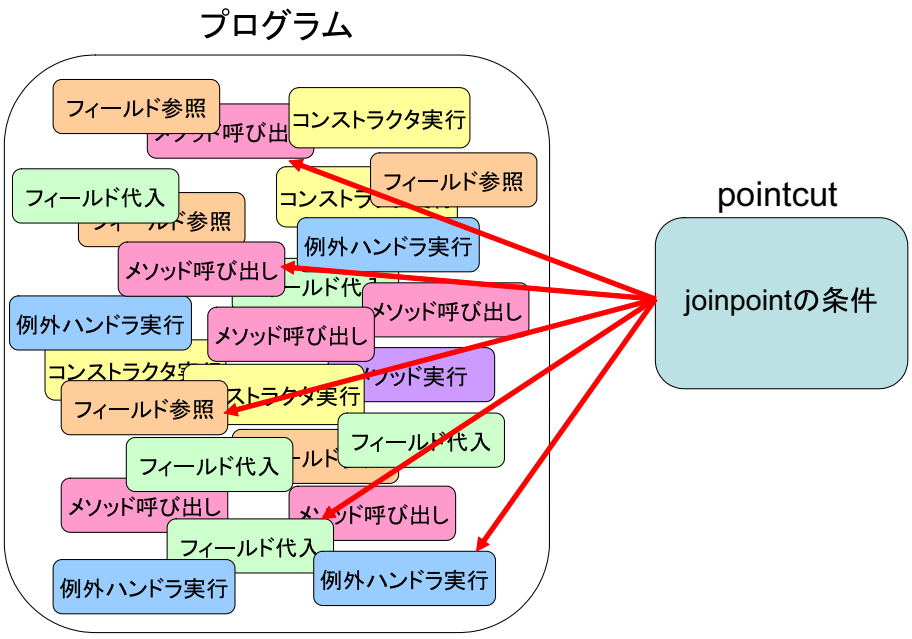

図 2.4: pointcut による joinpoint の抽出

クラスの setX(int) メソッドの呼び出しを指定する pointcut 宣言は以下のコードである。

```
call(void figure.Point.setX(int))
```

AspectJ ではプリミティブな pointcut と複数の pointcut を組み合わせた pointcut が指定可能である。また、pointcut を指定する際の引数に任意の文字列を表すワイルドカード “*” や任意のサブクラスを表す “+”、任意の引数、パッケージを表す “..”、否定を表す “!” が利用可能である。以下にプリミティブな pointcut の利用例をいくつかを説明する。

call(void Foo.m())
: Foo.m() メソッドの呼び出される時点

call(Foo.new(..))
: Foo クラスのコンストラクタ呼び出される時点

execution(* Foo.*(..) throws java.io.IOException)
: Foo クラスのメソッドで宣言文に java.io.IOException の例外発生の可能性が記述されているメソッドが実行される時点

execution (public Foo.new(..))
Foo クラスの public なコンストラクタが実行される時点

initialization(Foo.new(int))
Foo(int) コンストラクタによって生成されるオブジェクトの初期化子実行される時点

preinitialization(Foo.new(int))
Foo(int) コンストラクタによって生成する際の super コンストラクタが呼び出される前の初期化子実行される時点

staticinitialization(Foo)
Foo クラスがロードされた後に Foo クラスの静的初期化子が実行される時点

get(int Point.x)
Point クラスのフィールド x が読み出しアクセスされる時点

set(private * Point.*)
Point クラスの privete な変数に値が代入される時点

handler(java.io.IOException+)
java.io.IOException とそのサブクラスの例外ハンドラが実行される時点

adviceexecution()
advice ボディが実行される時点

within(abc.def.*)
abc.def パッケージ内で宣言されているコード内全ての joinpoint

withincode(void Figure.move())
Figure.move() メソッド内全ての joinpoint

cflow(call(void Figure.move()))
Figure.move() メソッドが呼び出される場合のコントロールフロー上にある全ての joinpoint (Figure.move() メソッドの呼び出しも含む)

cflowbelow(call(void Figure.move()))
Figure.move() メソッドが呼び出される場合のコントロールフロー上にある全ての joinpoint (Figure.move() メソッドの呼び出しは含まない)

if(Tracing.isEnabled())
: Tracing.isEnabled() メソッドが true を返す全ての joinpoint (if 文の条件式には任意の式を入れることができる。また、joinpoint の情報を表す thisJoinPoint 変数の利用が可能である。)

this(Point)
: 実行中のオブジェクトが Point クラスのインスタンスである全ての joinpoint

target(java.io.InputPort)
: ターゲットオブジェクトが java.io.InputPort クラスのインスタンスである全ての joinpoint

args(java.io.InputPort, int)
: 引数が 2 つ存在し一つめの引数が java.io.InputPort 型、2 つめの引数が int 型の全ての joinpoint

args(*, int)
: 引数が 2 つ存在し 2 つめの引数が int 型の全ての joinpoint

args(short, .., short)
: 最初と最後の引数の型が shor t 型である全ての joinpoint

さらに、pointcut を組み合わせて新たな pointcut の定義を行うことができる。pointcut の演算子には論理積を表す “||”、論理和を表す “&&”、否定を表す “!” がある。以下に pointcut の演算子の利用例を示す。

get(int Point.getX()) || get(int Point.getY())
: Point クラスの getX() メソッドの呼び出し時点又は Point クラスの getY() メソッドの呼び出し時点

set(int Point.x) && within(Point)
: Point クラスの中でフィールド x にアクセスしている時点

!within(Point)
: Point クラスのコード以外に存在する joinpoint

Advice

advice は pointcut で抽出された joinpoint で新たな処理コードを実行するものである。advice には実行のタイミングに応じて、before:joinpoint に達する直前、after:joinpoint に達した直後、around:joinpoint で指定した処理の置き換えの 3 種類がある。以下にコード例を示す。

```
before() : execution(void Server.run()){
  Logger.log();
}

after(int newX) : call(void Point.setX(int)) && args(newX){
  System.out.println(newX);
}

void around(int newX, int newY)
       : call(void Point.move(int,int)) && args(newX,newY){
  long start = System.currentTimeMillis();
  proceed(newX, newY);
  long end = System.currentTimeMillis();
  System.out.println("interval: "+(end-start));
}
```

１つめの advice は Server クラスの run() メソッドが実行される直前にログを採るものである。２つめは set(int) メソッド呼び出しの直後に引数の値を出力するものである。３つめは move メソッドの呼び出しの処理を置き換えるものである。around advice の中で使われている proceed(..) は特殊メソッドで pointcut で抽出した元々の処理を表す。また、advice 内で指定した実行コードを advice ボディと言う。

Inter-type

既存のクラスにメソッドやフィールドを追加したり、継承関係を変えたりすることを inter-type 宣言と言う。以下にコード例を示す。

```
01 int figure.Point.z = 100;
02 public int figure.Point.compareTo(Object o){
03   ...
04 }
05 declare parents : figure.Point implements java.lang.Comparable;
```

上記のコードでは１行目で figure.Point クラスに int 型の z というフィールドを追加している。２～４行目では Point クラスに compareTo(Object) メソッドを追加している。５行目では Point クラスに java.lang.Comparable インターフェイスを追加している。inter-type 宣言によりクラスに追加できるメンバはフィールド、メソッドとコンストラクタである。また、declare parents により継承関係を変更することができる。declare 句はこの他に

declare warning、declare error がありコンパイル時にエラーや警告を発生させることができる。これらはプログラムが制約を満たしているかどうかなどの検査するために使われる。以下に例を示す。

```
declare warning
  :(set(int Point.x) || get(int Point.x)) && !within(Point)
  :"bad access to Point.x, use setter and getter methods";
```

上記のコードは Point クラス以外でフィールド x に直接アクセスしている場合に “bad access to Point.x, use setter and getter methods” という警告を出すための inter-type 宣言である。これにより Point.x に対する不正なフィールドアクセスを検知できる。

2.2.3　AJDT

AJDT (AspectJ Development Tools) [3] とは AspectJ の利用を統合開発環境 Eclipse[12] で支援するためのツールである。AspectJ を利用してソフトウェアを開発する場合、advice の挙動をソースコードだけを見て把握するのは困難である。advice を宣言すると pointcut で指定した joinpoint で自動的に advice ボディが実行されるが指定した joinpoint に対応する joinpoint shadow が存在するソースコードの位置には advice ボディを呼び出すための明示的な表記はない。そのため、joinpoint shadow が存在するソースコードを見ただけではどのような advice 実行されるのかを判断するのが難しい。

そこで、AJDT では統合開発環境の機能を利用して、advice が実行されるソースコードの位置にマークを着けユーザに知らせる。図 2.5 では、Point クラスの setX(int)、setY(int) メソッドの呼び出しが pointcut で指定されているため、Line クラスのソースコード内で対象となる joinpoint shadow の位置に advice ボディが呼び出されることを示すマーカーが着いている。この他、AJDT では、advice がクラス全体のどの位置に影響をあたえるのかを示す Aspect Visualiser や Cross Reference ビューなどを提供している。図 2.5 の Aspect Visualiser を見ると Line クラスと Rect クラスの色が変化している。これは UpdateAspect の advice が Line クラスと Rect クラスから呼び出されることを表している。また、Cross Reference ビューに表示されているエレメントは advice が呼び出されるソースコードの位置情報の一覧を表している。その他に AspectJ 用のソースコードエディタがあり、advice や pointcut を定義する際のコード補完機能などが実装されている。

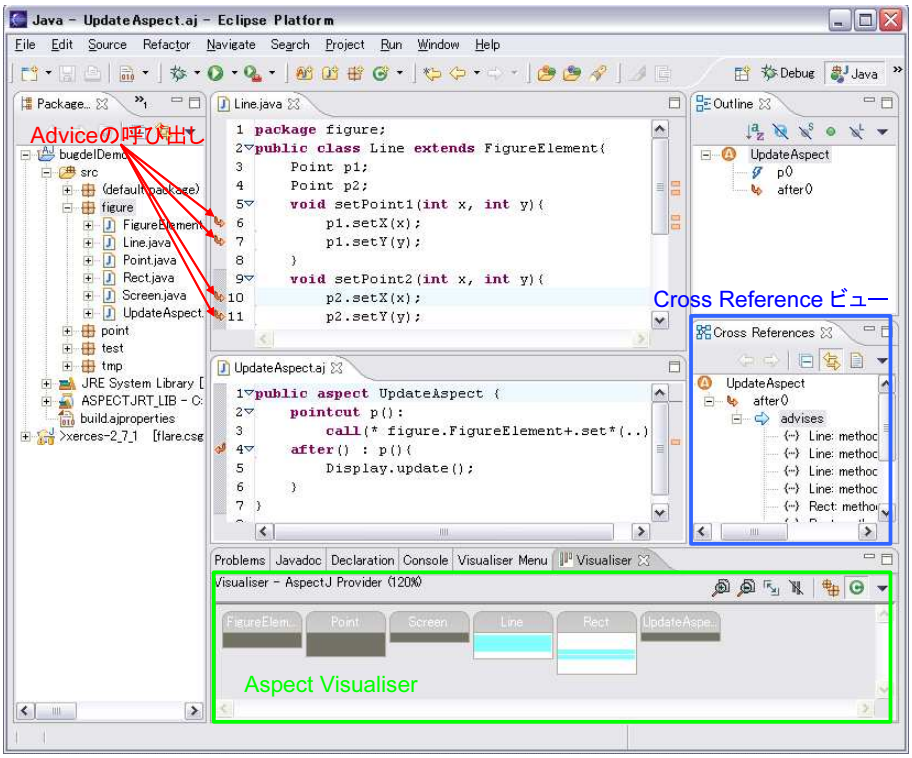

図 2.5: AJDT のよる AspectJ の利用画面

2.2.4　アスペクトの抽出を支援するシステム

JQuery

JQuery[20] とはクエリーを使ってソースコードを検索するシステムであり、Eclipse のプラグインとして実装されている。クエリーを実行することでソースコードの構成をさまざまな観点から表示させることができる。検索結果はツリー構造で表示される。図 2.6 は JQuery を利用している画面のスクリーンショットである。“?” マークのアイコンが着いているものがクエリーであり各エレメントとクエリー、検索結果が表示されている。図 2.6 上ではハイライト表示されている setPosition(int,int) メソッドに対して「Outgoing Calls」というクエリーを使ってメソッド内で呼び出しているエレメントの検索を行っている。JQuery ではクエリー言語に TyRuBa[35, 36] という論理方言語と専用の検索エンジンを利用している。Eclipse を使って検索する範囲 (ワーキングスペース) を定義すると、ワーキングスペース内にある Java のエレメント (クラス、メソッド、フィールドなど) の依存関係を表すデータベースを生成する。生成された

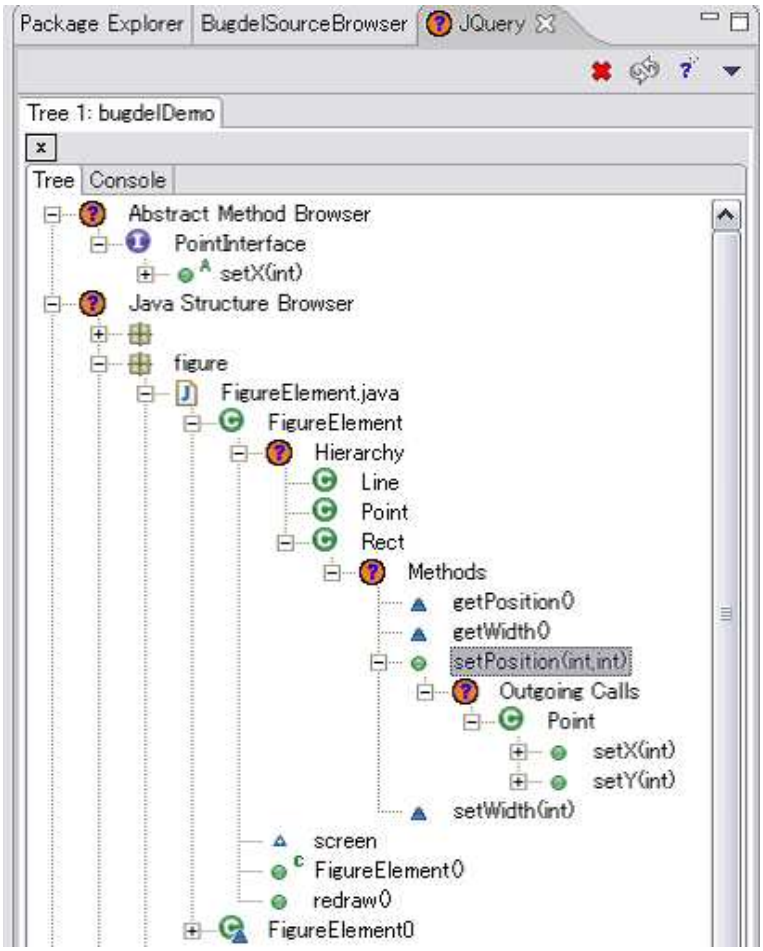

図 2.6: JQuery の利用画面

データベースに対して TyRuBa 言語を使ってクエリーを投げる。JQuery にはデフォルトで TyBuBa 用の predicate (述語) と predicate を利用したライブラリが準備されている。また、それらを組み合わせてユーザが独自に新たなクエリーを定義することもできる。新たに定義したクエリーは JQuery の GUI に組み込まれマウス操作により実行される。図 2.6 上で利用している「Outgoing Calls」はデフォルトで用意されているクエリーの一つである。JQuery を利用することでソースコードの検索が柔軟に行うことができアスペクトとして抽出するべきコードを発見するのに役に立つ。

2.3　既存のアスペクト指向システムの問題点

アスペクト指向を利用することで開発支援用コードを効率よくプログラムに追加することができる。しかし、既存のアスペクト指向システムには問題もある。この節では既存のアスペクト指向システムを使ってテスト、デバッグ用コードを指定する際の問題点を述べる。

2.3.1　クラスのモジュール性

AspectJ のような汎用的なアスペクト指向システムの場合、aspect によってクラスのモジュール性が失われないように設計されている。そのためデバッグに利用するためには問題点がいくつかある。一つめは、ある特定の位置を pointcut で指定できない点である。例えば、デバッグの際にデベロッパーはソースコードの１０行目に制御が達したときにデバッグコードを実行させたいと考える場合がある。しかし、AspectJ のような汎用的なアスペクト指向システムではソースコードの行番号を joinpoint として指定する pointcut を提供していない。なぜなら、ソースコードの行番号というのはメソッドの実装に依存するものであるからである。汎用的なアスペクト指向システムの pointcut で指定できる joinpoint はフィールドアクセスやメソッド呼び出しなどであり、pointcut を定義する際、引数として指定するものはフィールド名やメソッド名などのクラスやメソッドの実装に依存しないモジュールのインターフェイス（概観）である。そのためクラスやメソッドのモジュール性を pointcut、advice で壊すことはない。一方、ソースコードの行番号はメソッドの実装に依存するため、行番号を pointcut で指定できるように言語を設計した場合、メソッドの実装を用意に変更できなくなっしまい、クラスやメソッドのモジュール性を壊してしまう。そのため、汎用的なアスペクト指向システムでは行番号を pointcut で指定して advice コードを実行させることはできない。

二つめは、advice ボディから各 joinpoint に存在するローカル変数へアクセスができない点である。例えば、図 2.7 で示したような advice を定義することができない。そのため、advice を使ってローカル変数をトレースするコードを定義することができない。このような制約があるのは、ローカル変数はメソッドの実装に依存するものであり、メソッド内部でのみ参照されるべきのもであるからである。もしも、advice ボディからローカル変数へのアクセスを許可するとメソッドのモジュール性を壊してしまう。AspectJ では private フィールドや private メソッドに対するアクセスにも制限があり、これらのメンバに advice ボディからのアクセスを許可する場合には privileged 修飾子を付けた aspect を定義しなければならない。このようなアクセス制限はデバッグの際に問題となる。

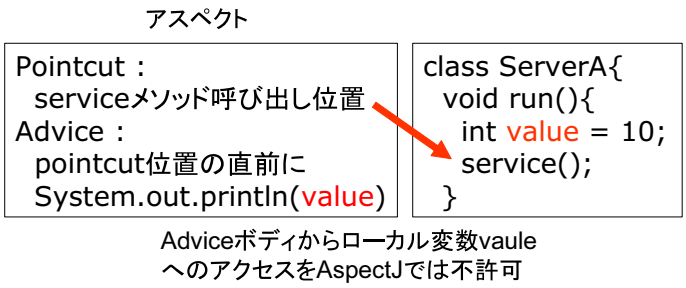

図 2.7: ローカル変数へのアクセス

以上のように汎用的なアスペクト指向システムではクラスのモジュール性、カプセル化を強く意識しているため pointcut や advice には制約があり、デバッグの際に問題となる。これは言語やシステムの設計が悪いのではなく汎用的に利用されるシステムには必要な制約である。しかし、デバッグのためにはモジュール性の制約を緩める必要がありデバッグ専用のアスペクト指向システムが必要である。

2.3.2　ユーザインターフェイス

AspectJ のようなアスペクト指向言語の場合、アスペクトをテキストベースで記述しなければならない。長期間に渡って利用されるログ出力コードをアスペクトとして記述する場合には問題にはならないが、一時的にしか利用しないデバッグコードをアスペクトとして記述するにはデベロッパーへの負担が大きい。例えば、デバッグの際、ログ出力を行う位置を変更する度に pointcut の定義を変更しなければならない。さらに、デバッグを行うためだけに専用のプログラミング言語を覚える必要があり、プリミティブな pointcut 指定子も覚えなければならない。AspectJ には統合開発環境で利用を支援するツール AJDT があるが、AJDT を使っても pointcut の指定はテキストベースで宣言的に行う必要がありデベロッパーへの負担は大きい。

ソースコードの検索と pointcut の指定を別々に行わなければならいという問題もある。デバッグの際にはソースコードの検索を行い、バグの原因となっているメソッドやフィールドを探す。そして、疑いのあるメソッドやフィールドに対して pointcut を指定してログ出力などを行う。JQuery を使うことで柔軟にソースコードの検索を行うことはできるが、pointcut を指定する作業と検索を行う作業を別々に行う必要があり検索と pointcut の指定を同時に行う機能はない。

2.3.3　ソースコードの編集

既存のアスペクト指向システムの場合、pointcut 記述がソースコードの変更に弱いという問題がある。例えば、AspectJ で以下のように pointcut で setX メソッドの実行位置を指定する “execution(void Point.setX(int))” が宣言されている場合を考える。

```
01 public aspect LogAspect{
02   pointcut p1(): execution(void Point.setX(int));
03   after() : p1(){
04     ...
05   }
06 }
07 public class Point{
08   int x, y;
09   public void setX(int newX){
10     this.x = newX;
11   }
12   ...
13 }
```

上記のコードでは 2 行目の pointcut p1() で 9 行目の Point クラスの setX(int) メソッドの実行位置を指定している。ここでリファクタリングにより setX の名前を setterX に変えた場合、setterX は pointcut p1() の対象にならなくなる。そのため、 setterX メソッドの実行位置では 3 〜 5 行目の advice ボディは実行されなくなってしまう。

次に、pointcut の引数にワイルドカード “*” を利用した場合を考える。

```
01 public aspect LogAspect{
02   pointcut p2(): execution(void Point.set*(..));
03   after() : p2(){
04     ...
05   }
06 }
07 public class Point{
08   int x, y;
09   public void setX(int newX){
10     this.x = newX;
11   }
12   public void setY(int newY){
13     this.y = newY;
```

```
14   }
15 }
```

上記のコードでは 2 行目の pointcut p() で Point クラスの名前が set で始まるメソッドの実行位置を指定している。p2() によって対象となるメソッドは Point クラスの 9 行目の setX メソッドと 1 2 行目の setY メソッドである。ここで Point クラスに新たにフィールド z と setZ メソッドを追加した場合、setZ は pointcut p2() の対象になる。しかし、ユーザは setZ メソッドが pointcut p2() で指定されることを意図してない可能性がある。ワイルドカード “*” を利用した場合と同様に、任意のサブクラスを表す “+” や任意の引数を表す “..” を利用した場合にも同様の問題が起こる可能性がある。

以上のようにユーザの意図に反してソースコードの編集により pointcut の対象が変更されてしまう可能性がある。そのため、テストコードなど長期間に渡って利用するアスペクトに記述する pointcut では問題となる。

第3章　Bugdel

前章で述べた問題点を解決するために、本章では Java 用、統合開発環境のためのアスペクト指向システム Bugdel[37, 33] の提案を行う。Bugdel はテストやデバッグのためのコードをアスペクト指向を利用して埋め込むシステムであり、このような追加コードはソースコードとは別に指定することができる。また、テスト、デバッグに特化したアスペクト指向システムであり AspectJ のような汎用的なアスペクト指向システムとは設計方針が異なる。具体的には、AspectJ ではアスペクトのモジュール性やクラスのカプセル化を重視し設計されている。そのため advice ボディから joinpoint に存在するローカル変数へアクセスできない、ある種の位置を pointcut で指定できない。これらの制約はテストやデバッグを行う際に問題となる。それに対して Bugdel ではアスペクトのモジュール性やカプセル化を阻害するものであってもテスト、デバッグに有用な機能は取り入れており、advice ボディから joinpoint に存在するローカル変数へアクセスを可能にし、Line pointcut によりソースコードの行番号を pointcut で指定できる。

Bugdel は統合開発環境である Eclipse のプラグインとして実装されており、pointcut の指定を Bugdel エディタやダイアログ、ソースコードブラウザなどの GUI を通して行える。そため、AspectJ のように pointcut の指定を専用のプログラミング言語を使って行うよりも、Bugdel の方が利用するにあたって覚えることが少ない。また、 AspectJ のような既存のアスペクト指向システムの場合 pointcut がソースコードの編集に弱いという問題がある。それに対し Bugdel では統合開発環境の機能を利用し pointcut で指定されているクラス、メソッド、フィールドの編集を監視する。そして、ソースコードの編集により pointcut の対象となるクラス、メソッド、フィールドがが変更される場合には警告を発しユーザにしらせる。ソースコードブラウザを利用し検索ベースで pointcut を指定した場合には指定した pointcut と検索の手順を対応づける。検索の手順に影響のあるソースコードの編集が行われた場合、対応する pointcut に対して警告を出す。

3.1　Bugdel の概要

Bugdel は Eclipse[13, 14] のプラグインとして実装されておりアスペクトを指定するためのユーザインターフェイスと指定したアスペクトをクラスファイルに埋め込む処理系 (weaver) を提供する。GUI 構成は主に Bugdel エディタ、Bugdel ビュー、ソースコードブラウザから成る (図 3.1)。Bugdel エディタは Java のソースエディタであり、ソースコード中のクラス名、メソッド名、フィールド名をクリックすることで pointcut の指定が行える。Bugdel エディタは Eclipse にデフォルトで含まれている Java のソースコードエディタ (org.eclipse.jdt.internal. ui.javaeditor.CompilationUnitEditor) を継承しているため、デフォルトの Java エディタの機能、例えばコード補完機能などは全て利用できる。Bugdel ビューは指定された pointcut、advice、inter-type の情報を表示するものである。Bugdel ソースブラウザビューはソースコードの構成を検索するブラウザである。検索を行いながら pointcut を指定することが可能である。

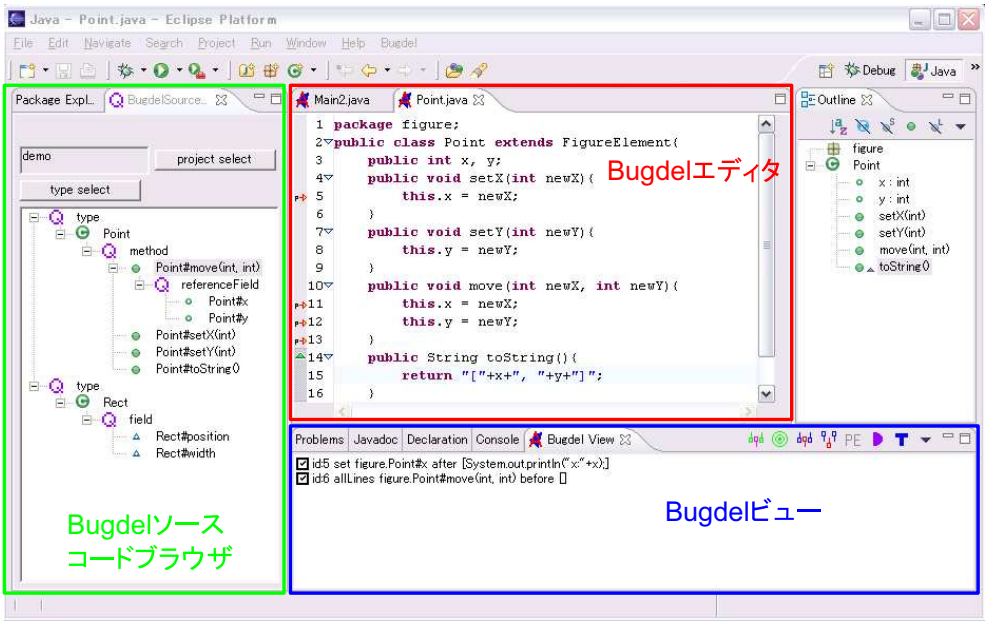

図 3.1: Bugdel のワークベンチ構成

指定されたアスペクトは Bugdel 専用の weaver によってクラスファイルに埋め込まれる。クラスファイルにアスペクトが埋め込まれるため、プログラムの実行には Eclipse や Bugdel は必要なく、通常の JVM で実行が可能である。Bugdel のシステム概要を図 3.2 に示す。Bugdel の主な特徴は以下である。

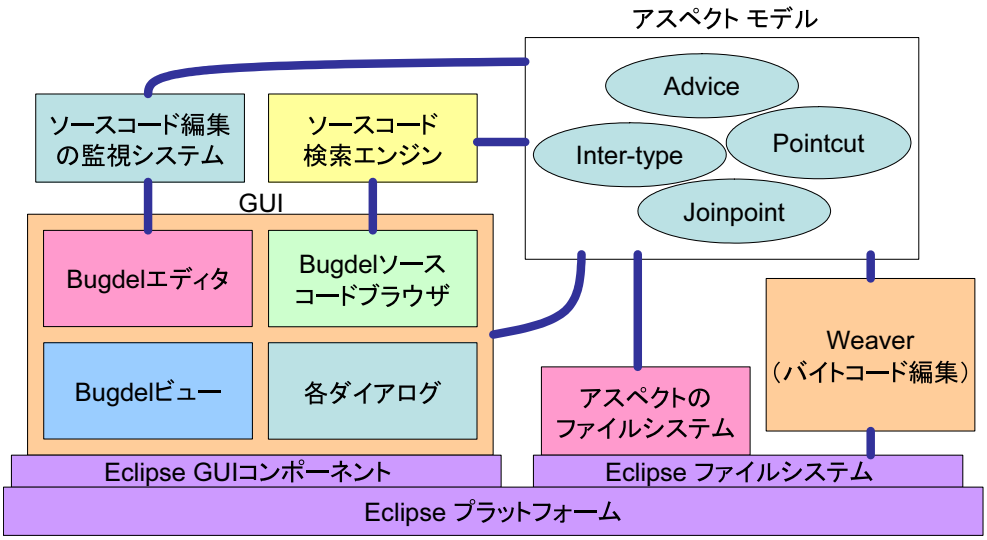

図 3.2: Bugdel のシステムの構成

- GUI を通して pointcut を指定できる。
- ソースコードブラウザを利用することでソースコード検索を行いながら pointcut を指定できる。
- Line pointcut、AllLines pointcut を提供する。
- advice ボディから pointcut で指定した joinpoint に存在するローカル変数へのアクセスが可能である。
- pointcut に影響のあるソースコードの変更を監視し、変更があった場合に警告を出す。

3.1.1　Eclipse アーキテクチャ

Eclipse はアプリケーション開発ツールを構築するために設計されたプラットフォームである。プラットフォーム自体はエンドユーザ向けの機能は、ほとんど提供していない。Eclipse はプラグインを追加することでさまざまな機能を提供する。Eclipse プラットフォームには Eclipse のコアランタイムの他に、ツールによって作成され、ファイル・システムに保管されるプロジェクト、ファイル、フォルダーなどの作成と管理を行うための API を提供するリソース管理プラグイン、UI を構成するためのツールキット (SWT と JFace) や UI 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 UI プラグイン、ヘルプのための拡張ポイントを提供するヘルププラグイン、リソースのバージョン管理を行うためのチームプラグインなどが含まれている (図 3.3)。しかし、Eclipse プラットフォームが提供するプラグインもエンドユーザ向けの機能ではない。そのため Eclipse プラットフォームだけでは、ユーザは何も利用できない。そこで、Eclipse はエンドユーザ向けの機能を含めた Eclipse SDK として配布されている。Eclipse-SDK には Eclipse プラットフォームの他に Java 開発のためのプラグイン JDT(Java Development Tools)[21]、プラグイン開発のためのプラグイン PDE(Plugin Development Environment)、ビルドツールである Ant を利用するための Ant プラグインが含まれている。これらのプラグインがエンドユーザの機能を提供している。Bugdel は Eclipse プラットフォームと JDT の機能を利用して実装されている (図 3.3)。

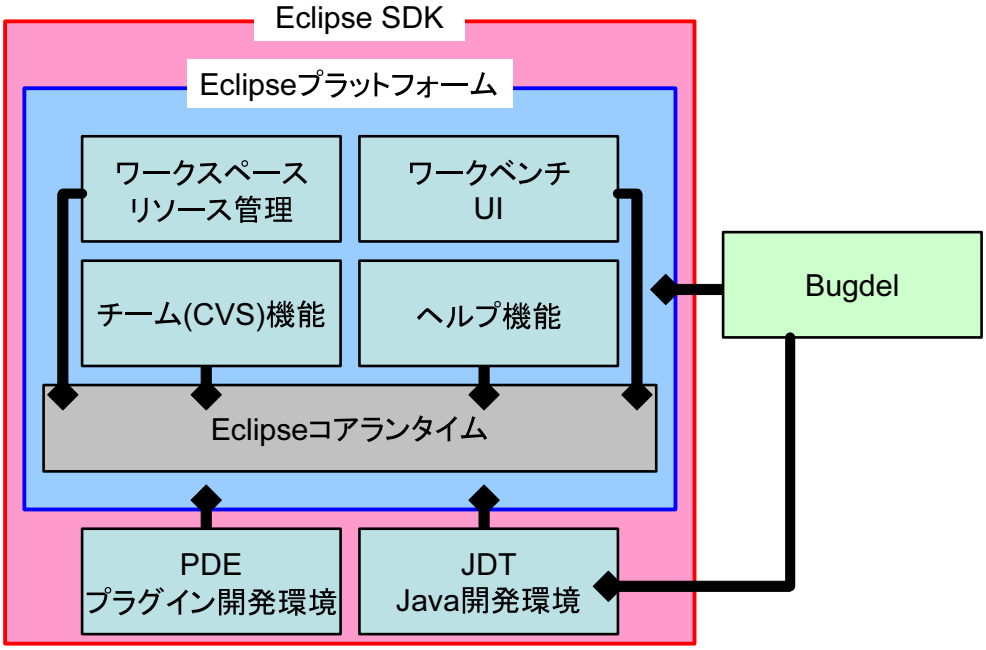

図 3.3: Eclipse SDK の構成

プラグインの構成ファイル

プラグインは主にマニフェスト・ファイルである plugin.xml ファイルと機能を実装したクラスファイルを含む jar から構成される。以下に Bugdel プラグインの構成ファイルを示す。

```
-plugins
  -bugdel_1.5.1
    -plugin.xml
    -bugdel.jar
    -bugdelrt.jar
    +javassist/
    +icons/
    +docs/
    ...
```

bugdel_1.5.1 の plugins.xml の内容の一部を示す。

```
<plugin
   id="bugdel"
   name="BugdelPlugin"
   version="1.5.0"
   provider-name="Yoshiyuki Usui"
   class="bugdel.BugdelPlugin">

   <runtime>
      <library name="bugdel.jar">
         <export name="*"/>
      </library>
      <library name="javassist/javassist.jar">
         <export name="*"/>
      </library>
   </runtime>
   <requires>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="org.eclipse.core.resources"/>
      ...
   </requires>
   <extension
      id="bugdel.ui.editor.BugdelEditors"
      name="BugdelEditors"
      point="org.eclipse.ui.editors">

      <editor
        filenames="java"
        class="bugdel.ui.editor.BugdelEditor"
```

```
   ...
</plugin>
```

上記のように plugin.xml にはプラグインの固有 ID を表す id、プラグインのユーザ表示名を表す name、プラグインのバージョン番号を表す version、プロバイダー名を表す provider-name を含める。class で指定している bugdel.BugdelPlugin は bugdel.jar に含まれる bugdel.BugdelPlugin クラスを表している。このクラスは Bugdel プラグインのライフサイクルを管理するものである。<runtime> タグにはクラスローダが参照しプラグインが提供するライブラリを指定する。上記の例では bugdel.jar と javassist.jar を指定している。これらのライブラリはプラグインの構成ファイルに含める。<requires> タグにはプラグインが利用する外部プラグインを指定する。Bugdel の場合、リソースを操作するため org.eclipse.core.resources プラグインを指定したり、Java のソースコードを操作するため org.eclipse.jdt.core プラグインなどが指定される。<extension> タグには Eclipse を拡張するための拡張ポイントを指定する。Bugdel の場合、Bugdel エディタや Bugdel ビューを追加するための記述を行っている。

bugdel.jar ファイルには plugin.xml で指定した拡張ポイントの実装クラスが含まれている。例えば、新たなエディタを追加する拡張ポイント org.eclipse.ui.editors に対しては bugdel.ui.editor.BugdelEditor クラスがビューを追加する拡張ポイント org.eclipse.ui.views に対しては bugdel.ui.view.BugdelView クラスが実装クラスとして bugdel.jar に含まれる。その他、Bugdel を利用して指定した Aspect を埋め込むための weaver の実装クラスなども含まれている。

3.2　Pointcut

Bugdel では AspectJ で提供されているフィールドアクセスやメソッド呼び出しなどのイベントを指定する pointcut に加えて、ソースファイルの行番号を指定する Line pointcut、メソッド中の全ての行番号を指定する AllLines pointcut を提供する。表 3.1 に Bugdel が提供する pointcut の一覧を示す。

Line pointcut

Line pointcut はソースファイルの行番号の先頭の joinpoint を指定する。これにより、ほぼ任意の行番号の先頭に Adivce コードを挿入することが可能である。Line pointcut を指定するには、Bugdel エディタのルーラー（エディタの左）でマウスを右クリックしポップアップメニューを開

pointcut	説明
FieldGet(field name)	フィールド参照位置
FieldSet(field name)	フィールド代入位置
MethodCall(method name)	メソッド呼び出し位置
MethodExecution(method name)	メソッド実行位置
ConstructorCall(constructor name)	コンストラクタ呼び出し位置
ConstructorExecution(constructor name)	コンストラクタ実行位置
Instanceof(class name)	instanceof 演算の実行位置
Cast(class name)	キャストの実行位置
Handler(class name)	例外ハンドラの実行位置
Line(file name, line number)	ソースコードの行番号の先頭
AllLines(method name)	メソッド内全ての行番号
within(class name)	クラス内全ての joinpoint
withincode(method name)	メソッド内全ての joinpoint

表 3.1: Bugdel の pointcut 指定子

き、「Line pointcut:[行番号]」アイテムを選択する。[行番号] には指定させている行番号が表示される（図 3.4）。デバッガのブレイクポイントと

図 3.4: Line pointcut の指定

同様に Line pointcut では任意の行番号を選択できるわけではなく、実行コードが存在する行番号だけである。空白の行や変数の宣言文だけの行は Line pointcut では指定できない。また、以下のプログラムの 3 行目や 9 行目のような super コンストラクタや this コンストラクタの存在する行番号も指定できない。

```
1 class Point extends FigureElement{
2   public Point(int x, int y){
3     super();
4     this.x = x;
5     this.y = y;
6     ...
7   }
8   public Point(){
9     this(0, 0);
10  }
11...
```

Line pointcut で指定できない行番号でポップアップメニューを表示させた場合には「Line pointcut:[行番号]」アイテムは表示されず、代わりに「Invarid Line:[行番号]」アイテムが表示させる。「Invarid Line:[行番号]」アイテムを選択した場合にはビープ音が発生し、Line pointcut は指定されない。

Line pointcut で指定するソースファイルの行番号はメソッドの実装に依存する。そのためクラスのモジュール性を考えた場合、Line pointcut 機能を提供するとメソッドの実装を安易に変更できなくなる。しかし、クラスのモジュール性を阻害してもテスト、トレース、デバッグには有用であると考え Bugdel では提供する。また、Line pointcut を提供すると行番号がずれてしまうという問題がある。例えば、ソースコードの１０行目を Line pointcut で指定し、その後、５行目に改行文字が記述された場合 Line pointcut で指定された位置を１１行目に変更しなければならない。この問題に対して Bugdel では Eclipse のリソースマーカーを利用して、行番号の変更を監視し Line pointcut で指定されている行番号を更新する。

リソースマーカーはソースコードの文字列や行番号、ファイル、プロジェクトなどに対してマーク付けをするものであり行番号などのさまざまな情報を保存することができる。リソースマーカーが持つ行番号の情報は Eclipse 内部に保存され、ソースコードが編集される度に自動的に更新される。リソースマーカーを使うことでエディタ上の文字列に波線を表示させたり、行番号の近くにアイコンを表示させたりすることができる。この機能はデバッガのブレイクポイントの位置情報を保存したり、ソースコード中のコンパイルエラー位置を表示させたりするために使われている。新たなリソースマーカーを定義することも可能であり、Eclipse の拡張ポイントである org.eclipse.core.resources.markers を指定しマーカーを定義する。Bugdel では Line pointcut の情報を保存するためのリソースマーカー、Line pointcut マーカーを定義している。Line pointcut マーカー

には行番号と advice の ID を持たせている。Line pointcut を指定すると Bugdel エディタで開いている IFile オブジェクトに対して Line pointcut マーカーを付ける。そして、ファイルをセーブする際や weave を行う際に、Line pointcut マーカーから行番号を取得する。

リソースマーカーの追加、削除を行うには Eclipse のファイルシステムからファイルを表すオブジェクトを取得し org.eclipse.core.resources パッケージの IResource や IMarker で定義されているメソッドを使って操作する。表 3.2 にリソースマーカーを操作するための API を示す。

IResource のメソッド	説明
IMarker createMarker(String type)	type で指定された型のリソースマーカーをリソースに追加する
void deleteMarkers(..)	指定された型のリソースマーカーを全て削除する
IMarker[] findMarkers(..)	指定された型のリソースマーカーを検索する
IMarker のメソッド	**説明**
Object getAttribute(String name)	name にマッピングされた属性を取得する
void setAttribute(String name, ...)	name にマッピングする属性を指定する
Map getAttributes()	属性の一覧を取得する
String getType()	マーカーの型を取得する
void delete()	マーカーを削除する

表 3.2: リソースマーカーを操作する API

AllLines pointcut

AllLines pointcut はメソッド又はコンストラクタ内に存在する全ての行番号を指定する pointcut である。ただし、Line pointcut で指定できない行番号は AllLines pointcut でも対象にならない。各行番号は joinpoint として抽出され、AllLines pointcut と結び付けられている advice コードは各行番号で実行される。これによりステップ実行が行える。例えば、Point クラスの move メソッドを AllLines pointcut で指定し advice コードとして Point クラスの変数 x の値を出力するコードを記述すると、move メソッドの各行番号で変数 x の値が出力され、プログラムの実行中に x がどのように変化してくのかを一行づつ確認することができる。

3.2.1　GUI を通した pointcut の指定

Bugdel では GUI を通して pointcut の指定を行う。pointcut の指定を行うには、まず Bugdel エディタで表示されているソースコード上でフィールド名やメソッド名を選択しする。そして、マウスをクリックしてポップアップメニューを表示させる。ポップアップメニューには「pointcut」アイテムが表示されるのでクリックする。すると選択したフィールドやメソッドに関係のある pointcut が候補として表示されるので目的のものを選ぶ。図 3.5 では Point クラスの変数 x が選択されているので Set（フィールド代入位置）、Get（フィールド参照位置）が候補として表示されている。図 3.6 では Point クラスの setX メソッドを選択しているので AllLines、Call（MethodCall）、Execution（MethodExecution）が候補として表示されている。

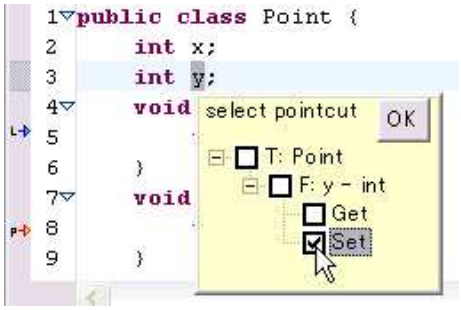

図 3.5: pointcut の候補の表示 1

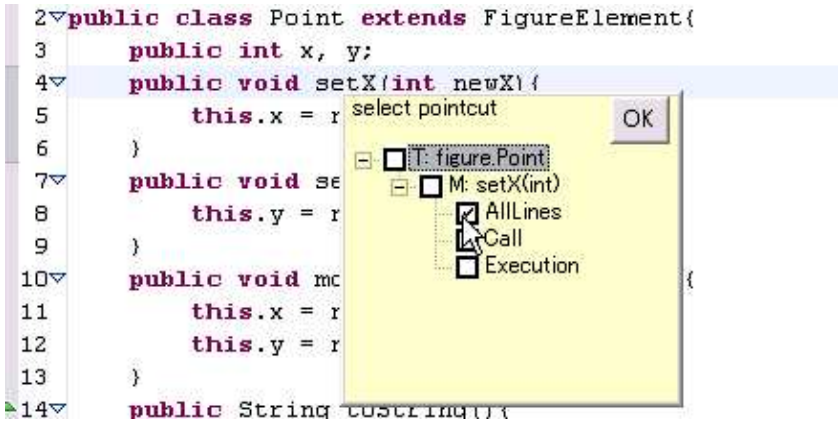

図 3.6: pointcut の候補の表示 2

このように、エディタ上フィールドを選択した場合には FieldSet と

FieldGet、メソッドを選択した場合には AllLines と MethodCall と MethodExecution、コンストラクタを選択した場合には AllLines と ConstructorCall と ConstructorExecution、タイプ（インターフェイス、クラス）を選択した場合には、タイプに含まれるフィールドとメソッドに関係する pointcut の一覧と、Handler、Cast、Instanceof が表示される。同様に Java のアウトラインビューや検索ビューなどに表示されるクラスやメソッド、フィールドなどのメンバをマウスで右クリックした場合に「Set advice by Bugdel」という項目が表示されるので選択すると、選択したメンバに関係する pointcut の候補が表示される。

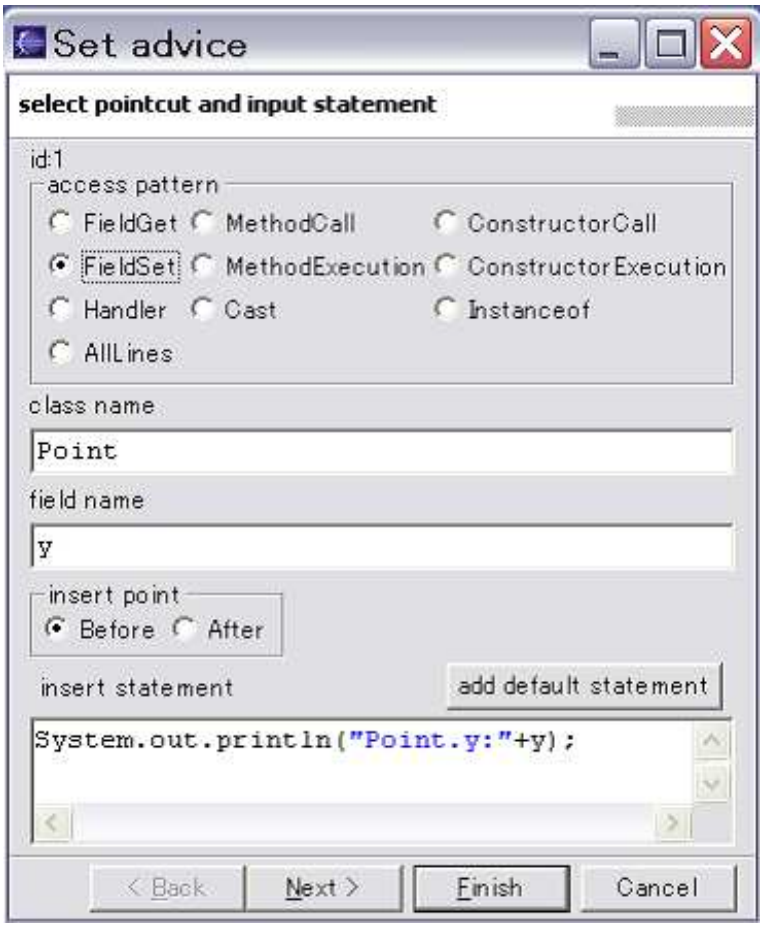

図 3.7: advice ダイアログ

ダイアログを利用して pointcut の指定を行うこともできる。Eclipse のメニューバーから「Bugdel」→「pointcut」を選択すると advice ダイアログが表示されるので、その中に pointcut 記述する。ダイアログの中には

pointcut 指定子を選択して、クラス名、メソッド名、フィールド名などを入力する。その際、“*”、“+” や “!” を利用することができる。“*” は任意の文字列を表すワイルドカードであり、“+” は任意のサブクラスという意味である。“!” は否定を表す。例えば、pointcut 指定子として MethodCall を選択し、クラス名に「Figure+」、メソッド名に「set*(int)」と入力すると Figure クラスのサブクラスのメソッド名前が set で始まるメソッドの呼び出し位置が対象となる。図 3.7 では advice ダイアログを利用して Point クラスのフィールド x への代入位置を pointcut で指定している。

指定された pointcut の情報は Bugdel ビューに表示される。また、pointcut で指定される joinpoint に対応するソースコードの位置には Bugdel エディタのルーラー上に adviced マーカーが表示される。これによりユーザは advice コードで記述されたコードがいつ実行されるのか把握することができる。adviced マーカー上でマウスを右クリックすると「adviced by」項目の中に対応する advice コードの項目が表示される。この項目をクリックすると advice ダイアログが開く。図 3.8 では setX(int) メソッドの呼び出し位置に挿入される advice の情報がルーラー上のポップアップメニューに表示されている。

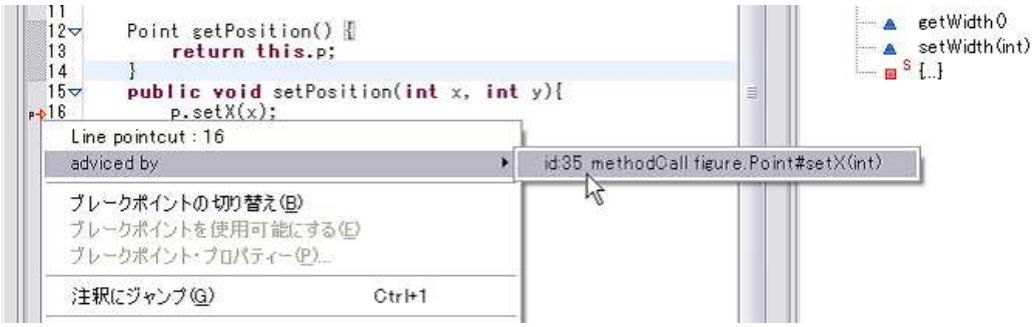

図 3.8: adviced マーカー

以上のように Bugdel では pointcut の指定を GUI を通して対話的に行えるため、Bugdel のユーザは pointcut を宣言するための複雑な文法を覚える必要がなく、set、get、call などの pointcut 指定子の種類を覚える必要も無い。これにより、AspectJ などのようなテキストベースで poiutcut を指定するアスペクト指向システムと比べて容易に pointcut の宣言を行うことができ、利用するために覚えるとこが少ない。

within、withincode pointcut の指定

within、withincode の指定は advice ダイアログ中に行う。advice ダイアログを開きトップの画面の一番下に表示される「Next (次へ)」を押すと within を入力するページに変わる。within available には within の利用を有効にする場合、チェックを付ける。within には joinpoint を含むクラス名前を入力する。withincode には joinpoint を含むメソッド名前を入力する。名前を入力するのに前節で説明した “*”、“+” や “!” を利用することが可能である。図 3.9 では within、withincode pointcut を利用して figure.Point クラスの move(int,int) メソッド内の joinpoint を指定している。

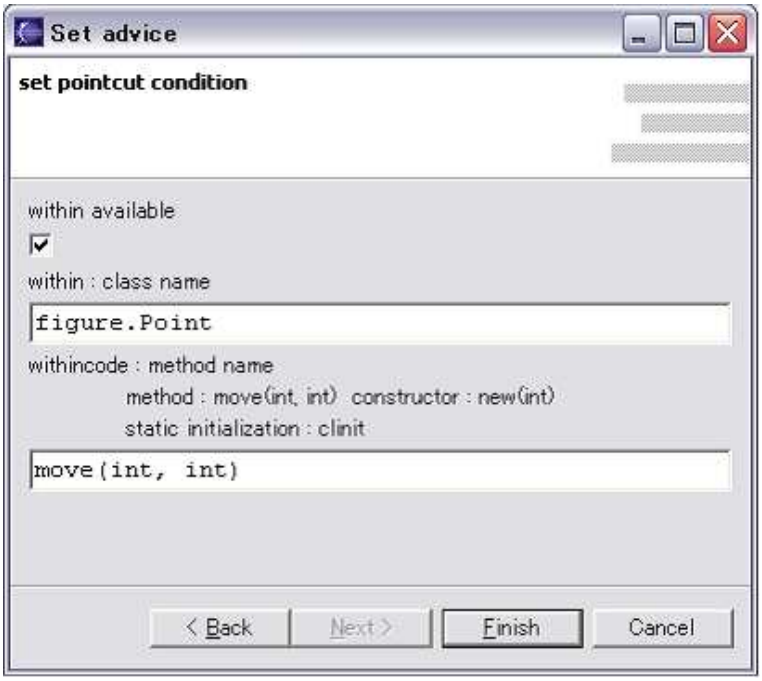

図 3.9: advice ダイアログ (within, withincode の入力)

また、Bugdel エディタを利用して pointcut の候補を表示させて pointcut を指定した場合、ソースコードをクリックした位置に応じて自動的に within に情報が記述される。例えば、figure.Line クラスの setPosition1(int,int) メソッド内に記述されている figure.Point クラスの setX(int) メソッドの呼び出しステートメントをクリックして pointcut を指定した場合、within には figure.Line、withincode には setPosition1(int,int) が自動的に記述される。ただし、within available にはチェックが付かないので within の条件を利用する場合にはユーザがチェックを入れる。

3.2.2　ソースコードブラウザ

Bugdel ではソースコードの構成をツリーで表示するソースコードブラウザ (図 3.10) を提供する。ソースコードブラウザを使うことによって検索をを行いながら pointcut の指定が行える。

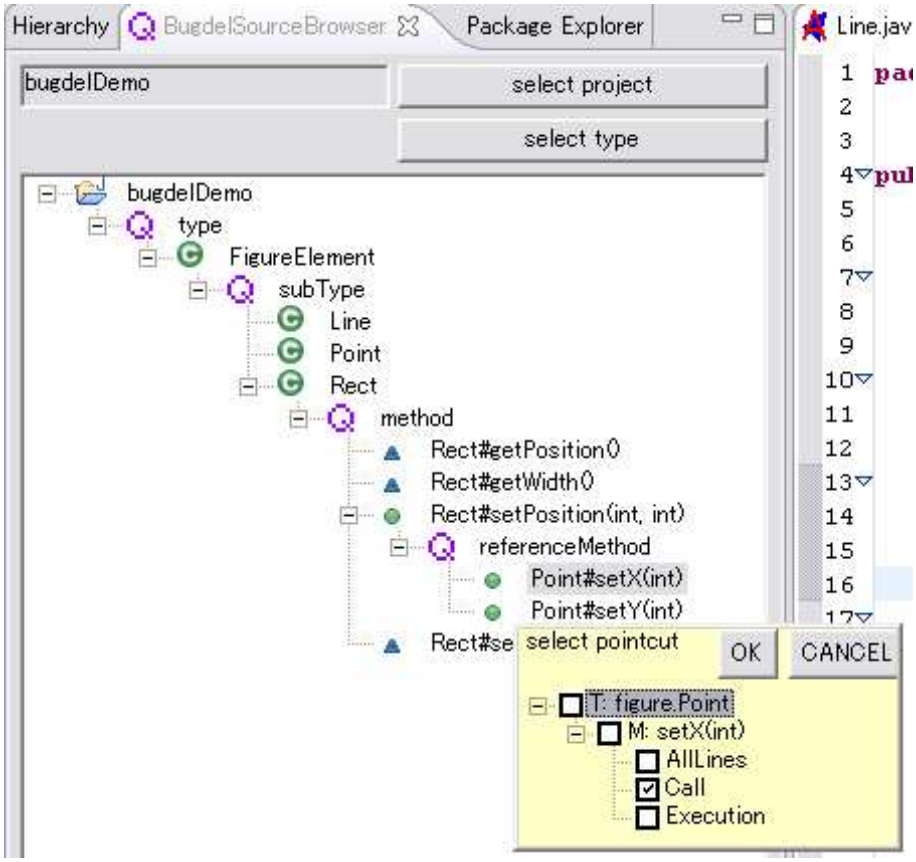

図 3.10: Bugdel のソースコードブラウザ

ソースコードブラウザを利用するにはEclipse メニューバーから「Window」→「Show View」→「Other」を選択し、BugdelSourceBrowser を選択する。するとソースコードブラウザビューが表示される。検索を行うには、初めにターゲットとなるプロジェクトとクラス (インターフェイス) を選択する。次にツリービュー上に表示されるクラスやメソッド、フィールドを表す Java エレメントをマウスでクリックしクエリーを選択する。利用可能な検索クエリーの一覧は表 3.3 である。

図 3.10 では、bugdelDemo プロジェクト内の FigureElement クラスに対して、サブクラスを検索している。次に、Rect クラスのメソッドを検索し、

ターゲット	クエリー	検索内容
クラス インターフェイス	field	宣言されているフィールド
	method	宣言されているメソッド
	constructor	宣言されているコンストラクタ
	subType	サブタイプ
	superType	スーパータイプ
	referenceType	参照しているタイプ
	referenceMethod	参照しているメソッド
	referenceField	参照しているフィールド
	declaring	宣言されているタイプ
メソッド コンストラクタ	referenceType	参照しているタイプ
	referenceMethod	参照しているメソッド
	referenceField	参照しているフィールド
	declaring	宣言されているタイプ
フィールド	declaring	宣言されているタイプ

表 3.3: 検索クエリーの一覧

その後 setPosition クラスの中で参照されているメソッドを検索している。ここで、setPosition メソッド内で参照しているメソッド Point#setX(int) の呼び出し位置を pointcut で指定するには、ブラウザ上に表示されている Point#setX(int) メソッドエレメントを選択しポップアップメニューを表示させ「pointcut」アイテムを選択する。すると、Point#setX(int) に対して pointcut の候補が表示されるので目的の pointcut を選択する。

このようにソースコードブラウザを利用することで検索を行いながら pointcut の指定を行うことができる。その際、ビューの切り替えは必要なくソースコードブラウザビューのみで検索を続けることができる。

検索エンジンの実装

ソースコードブラウザ上のタイプに対して行うクエリーの field、method、constructor と各ターゲットエレメントの declaring は JDT の org.eclipse.jdt.core パッケージの Java モデル API を利用して実装されている。Java モデル API はソースファイルを抽象化するモデルである。ICompilationUnit は１つのソースファイル、IPackageFragment はパッケージ、IType はクラス又はインターフェイス、IMethod はメソッド又はコンストラクタ、IField はフィールドを表す (図 3.11)。Java モデル API の一部を表 3.4、3.5、3.6 に示す。IMember は IType、IMethod、IField のスーパーインターフェ

イスである。IType で宣言されているメソッドを利用することで、field、method、constructor クエリーを実現している。subType、superType クエリーは IType の型階層を操作する newSupertypeHierarchy(..) メソッド、newTypeHierarchy(..) メソッドを利用して実現している。

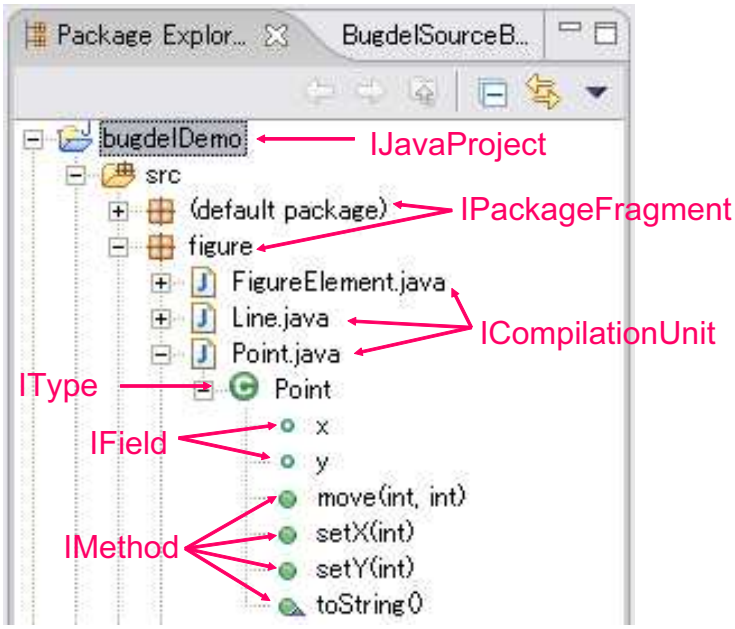

図 3.11: JDT の Java モデル

返り値の型	メソッド名前	説明
String	getFullyQualifiedName()	タイプの名前を取得
IField[]	getFields()	宣言されているフィールドを取得
IMethod[]	getMethods()	宣言されているメソッドを取得
IPackageFragment	getPackageFragment()	パッケージを取得
ITypeHierarchy	newSupertypeHierarchy(..)	スーパータイプの階層を取得
ITypeHierarchy	newTypeHierarchy(..)	サブタイプを含むタイプ階層を取得

表 3.4: IType のメソッド

タイプとメソッドに対して行うクエリーのうち referenceType、referenceMethod、referenceField は org.eclipse.jdt.core.search パッケージを利用して実装している。以下に org.eclipse.jdt.core.search パッケージを利用して検索を行うコード例を示す。

返り値の型	メソッド名前	説明
IType	getDeclaringType()	メンバが宣言されているタイプを取得
int	getFlags()	修飾子のフラグを取得

表 3.5: IMember のメソッド

返り値の型	メソッド名前	説明
boolean	isConstructor()	コンストラクタの場合 true を返す

表 3.6: IMethod のメソッド

```
01 IMember member = ...;
02 IJavaSearchScope scope =
03   SearchEngine.createJavaSearchScope(new IJavaElement[] { member });
04 int searchFor = IJavaSearchConstants.METHOD;
05 int limitTo = IJavaSearchConstants.REFERENCES;
06 int matchRule = SearchPattern.R_PATTERN_MATCH;
07 SearchPattern pattern =
08   SearchPattern.createPattern("*", searchFor, limitTo, matchRule);
09 SearchRequestor requestor = new SearchRequestor(){
10   public abstract void acceptSearchMatch(SearchMatch match)
11                                                    throws CoreException{
12     if(match instanceof TypeReferenceMatch){
13       ...
14     }else if(match instanceof MethodReferenceMatch){
15       ...
16     }else if(match instanceof FieldReferenceMatch){
17       ...
18     }
19   }
20 };
21 SearchParticipant[] paticipaints = new SearchParticipant[] {
22   SearchEngine.getDefaultSearchParticipant()
23 };
24 new SearchEngine().search(pattern, paticipaints, scope, requestor, null);
```

上記のコードでは１～３行目で検索範囲を指定している。あるクラス内を検索する場合には、クラス表す IType オブジェクトを member 変数に代入する。４～８行目では検索のパターンを指定している。まず、４行目で検索

対象を決める値を指定する。上記の例ではメソッドを検索するように変数を指定してる。この他、検索対象を表す変数にはクラスを表す IJavaSearchConstants.CLASS やフィールドを表す IJavaSearchConstants.FIELD などがある。５行目では検索の内容を参照に限定している。この他、宣言を表す IJavaSearchConstants.DECLARATIONS やインターフェイスの実装を表す IJavaSearchConstants.IMPLEMENTORS などがある。６行目で文字列マッチングのルールを決めている。上記の例ではパターンマッチを指定している。４〜７行目の条件をもとに８行目で検索パターンを生成している。createPattern(..) メソッドの第一引数で任意の文字列を指定している。９〜２０行目の SearchRequestor クラスは２４行目で検索を実行した際に通知を受け取るクラスである。検索にマッチするコードが見つかると SearchMatch オブジェクトがこのクラスに通知される。このようにしてソースコード中のクラス、メソッド内で利用しているクラスメンバを検索する。

3.2.3　pointcut に影響のあるソースコード編集の監視

Bugdel では pointcut に影響のあるソースコード編集が行われた場合に警告を発生させユーザに知らせる。例えば、pointcut 指定子として MethodExecution を選択し、クラス名に「Point」、メソッド名に「set*(int)」を入力した pointcut が指定されている場合を考える。Point クラスの定義は以下のものと仮定する。

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

この場合、指定した pointcut では setX と setY メソッドの実行位置が対象になる。次にユーザがリファクタリングにより setX の名前を setterX に変えた場合、指定済みの pointcut では setterX は対象にならない。この際、Bugdel は Bugdel エディタの背景を赤に変え、さらに Bugdel ビューに表示されている pointcut のラベルの色を赤に変える。このように、GUI により警告を出しユーザに知らせる。Point クラスに setZ メソッドのような新たなメソッドを追加し、そのメソッドが指定済みの pointcut の対

象になるような場合にも、pointcut の対象が追加されたことを示すために Bugdel エディタや Bugdel ビューの色を変えユーザに知らせる。

ソースコードブラウザを利用して pointcut を指定した場合には、検索の手順（検索パス）を pointcut に登録される。そして、その検索パスに影響のあるソースコードの編集が行われた場合にも警告を発生させユーザに知らせる。例えば、Rect クラスの setPosition メソッドの中で参照しているメソッドを検索し setX メソッドを見つけ、その setX に対して MethodCall を pointcut で指定した場合を考える。このとき、setPosition の中で参照している setX メソッドという検索の手順が pointcut に登録される。その後、リファクタリングにより setPosition メソッド内部の「p.setX(newX)」というコードを全て「p.x = newX;」に書き換えた場合、setPosition メソッドの中では setX メソッドを参照しなくなる。この場合、pointcut に対応付けられている setPosition の中で参照している setX メソッドという検索の手順に影響があったと判断され警告を出す。このように Bugdel では pointcut に影響のあるソースコードの編集を監視し、変更があった場合 Bugdel エディタの背景の色を薄い赤に Bugdel ビュー上の pointcut の色を赤に変えることでユーザに知らせる。

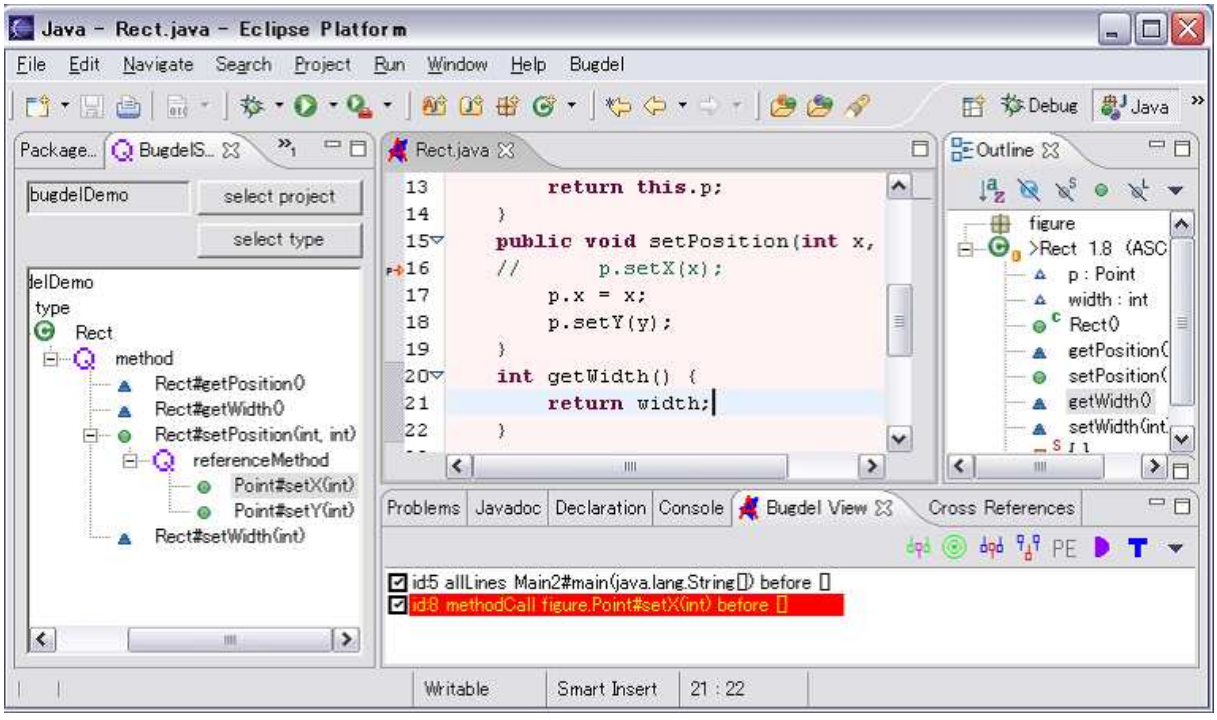

図 3.12: pointcut に影響のあるソースコード編集の監視

図 3.12 では setPosition の中の Point#set(int) メソッド呼び出しを編集しているため Bugdel ビュー中の影響のある pointcut MethodCall(Point#set(int)) のバックグラウンド赤色になっている。この pointcut はソースコードブラウザを利用して setPosition メソッドの中で参照しているメソッドを検索した結果、得られたエレメント Point#set(int) メソッドをクリックすることで指定した pointcut である。また、Bugdel ビュー上で pointcut を選択してポップアップメニューを表示させ「search path」アイテムを選択するとダイアログが表示される。ダイアログにはソースコードブラウザを利用したときの検索パスと pointcut に影響のあるソースコード編集のログの情報が書かれている。図 3.13 では検索パスに figure.Point クラス内のメソッドを検索し、次に setPosition(int,int) メソッド内で参照しているメソッドを検索し、得られた setX(int) メソッドに対して pointcut を指定したとう情報が表示されている。真ん中のテキストボックスには figure.Point.setPosition(int,int) メソッド内で参照しているメンバが変更されたという情報が表示されている。

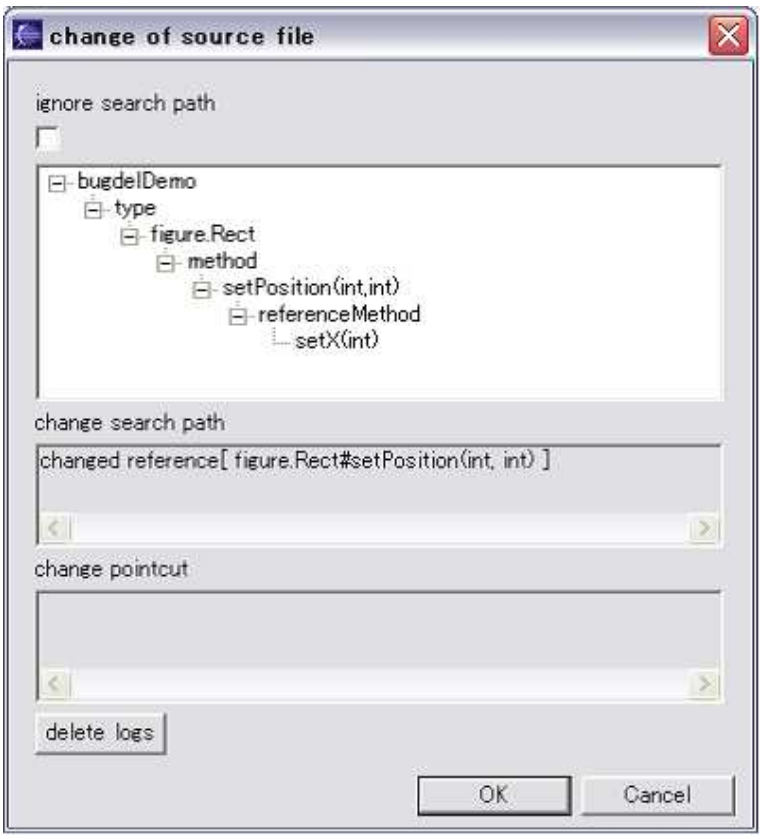

図 3.13: pointcut に影響のあるソースコード編集のログ

クラスメンバの追加、削除

フィールドやメソッドなどのクラスメンバの追加、削除の監視は org.eclipse.jdt.core.IElementChangedListener オブジェクトを利用して行う。JavaCore クラスの addElementChangedListener メソッドを利用して IElementChangedListener を登録すると、クラスメンバが追加、削除される際に、その変更をイベントとして受け取ることができる。以下にクラスメンバの追加、削除の通知を受け取るリスナのコードを示す。

```
public class MyElementChangeListener implements IElementChangedListener {
  public void elementChanged(ElementChangedEvent event) {
    IJavaElementDelta delta= event.getDelta();
    traverse(delta);
  }
  void traverse(IJavaElementDelta delta) {
    IJavaElementDelta delta= event.getDelta();
    int kind = delta.getKind();
    IJavaElement element = delta.getElement();

    if(kind == IJavaElementDelta.REMOVED){
      ...
    }else if(kind == IJavaElementDelta.ADDED){
      ...
    }else if(kind == IJavaElementDelta.CHANGED){
      if(delta.getFlags() == IJavaElementDelta.F_SUPER_TYPES){
        ...
      }else{
        ...
      }
    }
    IJavaElementDelta[] children = delta.getAffectedChildren();
      for (int i = 0; i < children.length; i++) {
        traverse(children[i]);
      }
    }
  }
}
```

IJavaElementDelta には変更が行われた IJavaElement オブジェクトと変更の種類が含まれる。

メソッドボディの編集

IElementChangedListener ではクラスメンバの追加、削除の監視は行えるがメソッドボディの編集の監視は行えない。代わりに org.eclipse.jdt.core.IBufferChangedListener オブジェクトを利用してメソッドボディの編集の監視を行う。IBufferChangedListener をソースコードを表す ICompilationUnit オブジェクトに含まれる IBuffer オブジェクトに登録すると、ソースコードが変更された場合に、通知が行われ変更された位置 (ファイルの先頭からの文字数) を取得できる。その後、通知された位置に存在するメソッドを調べる。そして、変更されたメソッドボディ内の再検索を行い参照されているメンバに変更が変更されていないかを調べる。以下にメソッドボディが変更された際に、変更されたメソッドを取得するリスナのコードを示す。

```
public class BodyChangeListener implements IBufferChangedListener{
  ICompilationUnit unit;
  public BodyChangeListener(ICompilationUnit unit) {
    this.unit = unit;
  }
  public void bufferChanged(BufferChangedEvent event) {
    int offset = event.getOffset();
    IJavaElement element = unit.getElementAt(offset);
    if(element instanceof IMethod){
      IMethod changedMethod = (IMethod) element;
      ...
    }
  }
}
```

上記の IMethod はメソッド、コンストラクタを表すクラスである。このリスナは以下のコードによって、Bugdel エディタが開かれるときにソースコードに対応する ICompilatiomUnit オブジェクト追加される。

```
ICompilationUnit unit = ...;
unit.getBuffer().addBufferChangedListener(new BodyChangeListener(unit));
```

3.3　Advice

Bugdel ビューに表示されている advice、pointcut を選択し、ポップアップメニューを表示させ「edit」アイテムを選択する。すると advice ダイアロ

グ（図 3.7）が表示されるので、その中に pointcut で指定された joinpoint で実行する advice コードを記述する。advice ダイアログの中には advice コードと advice を実行するタイミングも指定する。joinpoint に達する直前で advice コードを実行するなら before、joinpoint に達した直後で advice コードを実行するなら after を insert point に指定する。ただし、pointcut の種類が Handler と AllLines の場合には insert point は before のみ選択可能である。

3.3.1　局所変数へのアクセス

Bugdel では advice コード内から pointcut で指定した位置に存在するローカル変数へのアクセスを許可する。ローカル変数はメソッドの実装に依存するものであり、クラスのモジュール性を低下させるため AspectJ のような汎用的なアスペクト指向システムではアクセスを許可していない。デバッグの際にはローカル変数の値を確認する場合がある。そのため advice コード内からのローカル変数へのアクセスを Bugdel では許可する。図 2.7 で示したような advice が定義可能である。また、クラスの private フィールドに対しても無条件にアクセスを許可する。このような機能を提供すると存在しない変数名を advice コードに記述する可能性がある。そこで、存在しない変数を記述した場合、警告マーカーを Bugdel エディタ上に表示させユーザに知らせる。

3.4　Inter-type

Eclipse のメニューバーから「Bugdel」→「Inter-type Declaration」を選択するとインタータイプ用のダイアログが表示されるので、その中にインタータイプの追加先のクラス名とインタータイプのコードを記述する。図 3.14 では、test.*で表されるクラスに long 型のフィールド loadTime を追加している。loadTime 変数は System.currentTimeMillis() の値で初期化される。

メソッドを追加する場合にも同様に、ラジオボックスかた「method」を選択し、メソッドの宣言を declare statement に記述する。インタータイプによって生成するフィールドやメソッドは advice コード内で利用することができる。

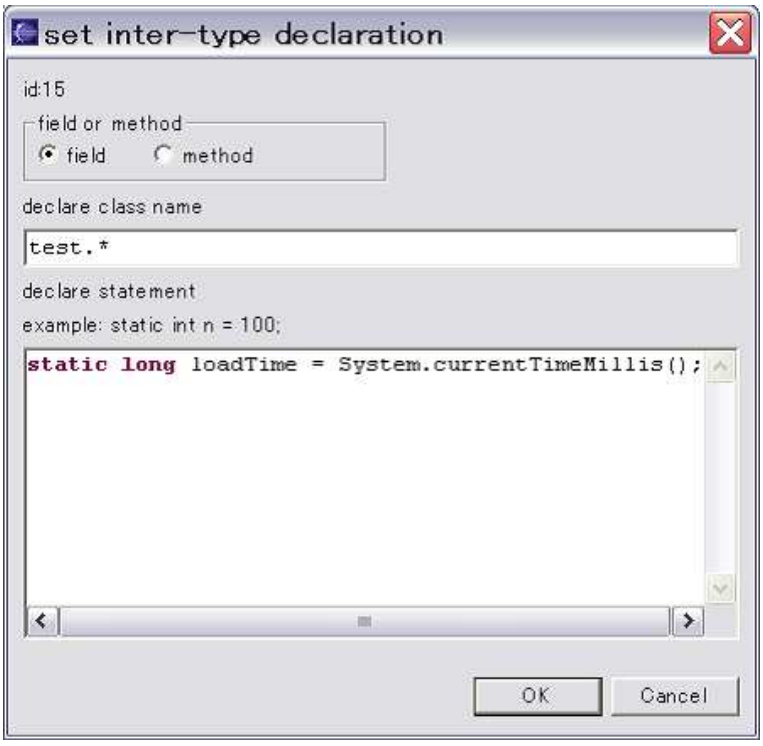

図 3.14: Inter-type ダイアログ

3.5　Weave

advice、inter-type の実現はバイトコードを変換することで行う。weave の実行はメニューバーの項目から「Bugdel」→「weave this file」又は「weave all」を選択した際に実行される。「weave this file」は現在開いているソースコードに対応するクラスファイルに、「weave all」はプロジェクト全体のソースコードに対応するクラスファイルに Aspect で指定したコードを埋め込む。

advice コードや inter-type コードに不完全なコードがある場合には weave 時にダイアログを開いてエラーを表示させる。また、メソッドに advice コードを埋め込んだ結果、バイトコードコードサイズが JVM の仕様の 66635byte を超えた場合にも weave エラーを表示させる。

「weave all」、「weave this file」を選択した際の処理の流れを図 3.15 に示す。まず、.java ファイルを JDT のコンパイラでコンパイルし.class ファイルを生成する。生成した.class ファイルに対して指定された Aspect を Bugdel 専用の weaver を使って埋め込む。weaver はバイトコードを操作するためのライブラリである javassist[9, 10, 8, 38] を利用し実装されて

いる。

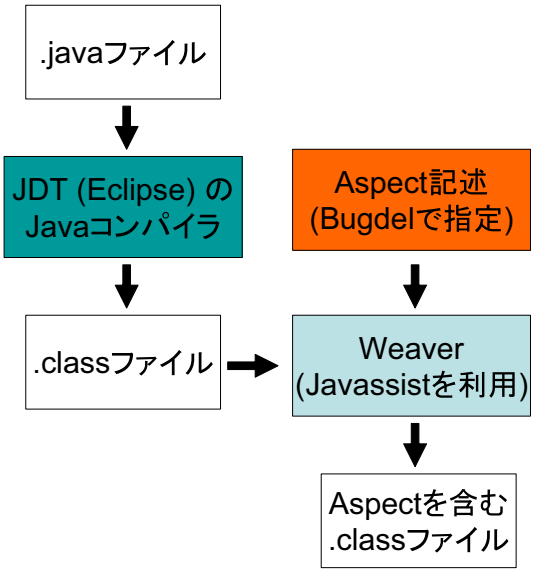

図 3.15: weave プロセス

Bugdel ではコンパイル時にクラスファイルを変換してアスペクトを埋め込むが、weave のタイミングは大きく分けて 3 種類が考えられる。一つ目は Bugdel が採用したコンパイル時 weave、二つ目はクラスを JVM にロードする際に weave を行うロード時 weave 三つ目は PROSE[25] や Wool[26] などで採用されている実行時 weave であり、それぞれ長所、短所がある。コンパイル時 weave はクラスファイル中にアスペクトを埋め込むためプログラムを実行するにあたって特殊な実行環境を必要としない。しかし、実際には利用されないクラスに対しても weave を行うため、不必要な weave 処理によるコンパイル時のオーバーヘッドがある。ロード時 weave はロードされるクラスに対して weave を行うため、利用されないクラスには weave は行われず不必要な weave 処理が無い。しかし、プログラムを実行する際にはクラスローダーを差し替える必要がある。また、weave エラーが実行時エラーとなってしまう。実行時 weave はプログラムの状態によって柔軟にアスペクトの切り替えが行える。しかし、プログラムを実行するには特殊な実行環境（JVM）が必要である。PROSE や Wool では JVM にデバッガーのサポートが必要である。

Bugdel では Eclipse との連携や JVM の可搬性を重視しコンパイル時 weave を採用した。Eclipse-SDK には JDT から提供されている Java プログラムの main メソッドを実行するためのボタンが提供されている。ボタンを押すとプロジェクトのクラスファイルの出力フォルダがクラスパスに

追加されてプログラムが実行される。Bugdel はコンパイル時 weave であるのでアスペクトを weave したクラスファイルが出力フォルダに生成される。そのため、weave 済みのクラスファイルを JDT の実行ボタンから実行することもできる。もしも、ロード時 weave を採用した場合、クラスローダーを差し替えて main メソッドを実行する必要があり、そのためのボタンを新たに Eclipse に組み込まなければならない。コンパイル時 weave を採用することで、その他の JDT で提供されているさまざまな機能を利用できる。実行時 weave を採用した場合にも同様に Eclise、JDT との連携が難しくなる。また、プログラムの実行にはデバッグモードなどが付いた特殊な JVM が必要になる。一方、Bugdel ではプログラムを実行するのにデバッグモードなど特殊な実行環境を必要としない。Bugdel と Eclipse も必要なく通常の JVM で実行できる。

Java ファイルのコンパイル

Bugdel の weaver は Aspect を埋め込む前にソースコードを一度コンパイルしてクラスファイルを生成する。コンパイルには Eclipse 内部の Java コンパイラを利用する。「weave this file」を実行した際は指定したソースファイルだけをコンパイルする。その際、コンパイルするには JDT に含まれる org.eclipse.jdt.internal.compiler.batch.Main クラスの compile(String[]) メソッドを利用する。このメソッドを利用してコンパイルするコードを以下に示す。

```
01 boolean compile(IFile file) {
02   IJavaProject jProject = JavaCore.create(file.getProject());
03   //プロジェクトに登録されているクラスパスを取得する
04   String[] classpath = JavaProjectUtil.getAllClassPath(jProject, true);
05   List argz = new LinkedList();
06   String classpathArg = classpath[0];
07   for (int i = 1; i < classpath.length; i++) {
08     classpathArg += File.pathSeparatorChar + classpath[i];
09   }
10   argz.add("-classpath");
11   argz.add(classpathArg);
12   //クラスファイルの出力フォルダを調べる
13   String outPath = JavaProjectUtil.getClassFileOutDir(jProject);
14   argz.add("-d");
15   argz.add(outPath);
16   argz.add("-g");//デバッグ情報を全て残す
```

```
17   argz.add("-preserveAllLocals");//ローカル変数を全て残す
18
19   String src = file.getLocation().toOSString();
20   argz.add(src);
21
22   Main compiler = new Main(null, null, false);
23   String[] tmp = new String[argz.size()];
24   argz.toArray(tmp);
25   boolean status = compiler.compile(tmp);//コンパイル
26   return status;
27 }
```

このコードでは 2 行目でファイルオブジェクトが存在するプロジェクトを取得している。そして、4 行目でプロジェクトに登録されているクラスパスを全て取得して、5-11 行目で取得したクラスパスをコンパイルオプションに追加している。JavaProjectUtil クラスは Bugdel が提供するライブラリである。13 行目でプロジェクトで指定されているクラスファイルの出力フォルダを検索し 14-15 行目でコンパイルオプションに追加している。Bugdel では advice 内から各 joinpoint に存在するローカル変数にアクセスできるようにしている。そのためクラスファイル中にローカル変数の情報を残さなければならない。そこで 16 行目ではローカル変数情報を残すために-g オプションを追加している。また、Bugdel ではソースファイルの行番号を pointcut で指定する Line pointcut も提供している。そのため、行番号の情報もクラスファイルに残す必要がある。-g オプションを付けることで行番号の情報もクラスファイルに残すことができる。これらのコンパイルオプションは JDK に含まれる javac コンパイラとほぼ同じものである。17 行目では-preserveAllLocals オプションを追加している。-preserveAllLocals オプションはメソッド中で利用しないローカル変数もバイトコード中に残すというものである。このオプションが無い場合、コンパイル時の最適化により利用されないローカル変数はバイトコードから削除されてしまう。例えばソースコードの中のメソッドの最後に「int n = 10;」という記述をした場合、変数 n はメソッド内で一度も利用れない。そのため-preserveAllLocals オプションを付けないでコンパイルした場合、「int n = 10;」は無いものとしてコンパイルされてしまう。すると変数 n の情報やその行番号がバイトコード中に残らなくなってしまい、Bugdel の weaver でローカル変数の情報を利用できなくなってしまう。そこで、17 行目で-preserveAllLocals オプションを付けてローカル変数の情報を残すようにしている。22-25 行目では org.eclipse.jdt.internal.compiler.batch.Main クラスを利用し生成したコンパイルオプションを使ってコンパイルを行

う。コンパイルが成功すれば返り値として true を受け取る。コンパイルが失敗すれば false を受け取る。

「weave all」を実行した場合にはプロジェクト全体をコンパイルする必要がある。プロジェクト内すべてをコンパイルするためには org.eclipse.jdt.core.IJavaProject オブジェクトを操作して行う。以下にコンパイルするコードを示す。

```
void compile(IJavaProject jProject, IProgressMonitor pm) {
  jProject.setOption(JavaCore.
      COMPILER_LOCAL_VARIABLE_ATTR, JavaCore.GENERATE);
  jProject.setOption(JavaCore.
      COMPILER_LINE_NUMBER_ATTR, JavaCore.GENERATE);
  jProject.setOption(JavaCore.
      COMPILER_CODEGEN_UNUSED_LOCAL, JavaCore.GENERATE);
  jProject.getProject().build(
      IncrementalProjectBuilder.FULL_BUILD, pm);
}
```

このコードではローカル変数と行番号情報を残すための設定を行ってプロジェクト全体のコンパイルを行うものである。コンパイルのオプションはクラスファイルにソースファイル名のアトリビュートを付けるオプションや finally ブロックをインライン化するオプションがある。

返り値の型	メソッド名前	説明
IClasspathEntry[]	getRawClasspath()	クラスパスの一覧を取得
IPath	getOutputLocation()	クラスファイルの出力フォルダを取得
String	getOption(String,boolean)	コンパイルオプションを取得
Map	getOptions(boolean)	コンパイルオプションの一覧を取得
void	setOption(String,String)	コンパイルオプションを設定
void	setOptions(Map)	コンパイルオプションの一覧を設定
IProject	getProject()	ワークスペース用のプロジェクトを取得

表 3.7: IJavaProject のメソッド

3.5.1　Advice の埋め込み

MethodExecution、ConstructorExecution pointcut の実現

pointcut が MethodExecution、ConstructorExecution に対応する advice コードの埋め込みは javassist ライブラリの CtBehavior クラスを操作

することで行う。CtBehavior オブジェクトは CtClass オブジェクトから取得できる。以下にコード例を示す。

```
01 ClassPool pool = ...;
02 CtClass clazz = pool.get("Point");
03 CtBehavior[] behaviors = clazz.getDeclaredMethods();
04 if(int i=0; i<behavior.length; i++){
05   CtBehavior cb = behaviors[i];
06   if(cb.getName("move")){
07     cb.insertBefore("{System.out.println(\"before exec\")}");
08     cb.insertAfter("{System.out.println(\"after exec\")}");
09   }
10 }
```

ClassPool はクラスパスを管理するクラスである。Bugdel では weave 対象の Java プロジェクトからクラスパスの一覧を取得して ClassPool オブジェクトに追加する。２行目では ClassPool から Point クラスを表す CtClass オブジェクトを取得している。3 行目では Point クラスの中で宣言されている全てのメソッドを取得している。CtBehavior クラスはメソッドを表す CtMethod クラスとコンストラクタを表す CtConstructor クラスの共通のスーパークラスである。CtClass にはこの他、フィールドの一覧を取得する getDeclaredFields() メソッドなどがある。６行目でメソッドの名前を調べている。メソッドの名前が move の場合、７行目でメソッドのボディの直前に、８行目で move メソッドのボディの直後に新たな処理を追加している。insertBefore、insertAfter メソッドに渡しているコードは javassist の Java 部分コンパイラに渡されバイトコードに変換され挿入される。

FieldGet、FieldSet、MethodCall、ConstructorCall、Instanceof、Handler、Cast,、Instanceof pointcut の実現

pointcut が FieldGet、FieldSet、MethodCall、ConstructorCall、Handler、Cast,、instaceof に対応する advice ボディの埋め込みは javassist.expr パッケージ内の ExprEditor クラスを利用して行う。以下にコード例を示す。

```
CtBehavior cb = ...;
ExprEditor editor = new MyExprEditor();
cb.instrument(editor);
```

instrument メソッドは引数の ExprEditor オブジェクトを使ってメソッドボディの振る舞いを変更する。instrument メソッドが実行されると、メ

返り値の型	メソッド名前	説明
String	getName()	クラス名を取得
CtClass	getSuperclass()	スーパークラスを取得
CtClass	getInterfaces()	インターフェイスを全て取得
CtClass	getDeclaringClass()	このクラスを宣言しているクラスを取得
CtField[]	getDeclaredFields()	宣言されているフィールドを取得
CtMethod[]	getDeclaredMethods()	宣言されているメソッドを取得
CtConstructor[]	getDeclaredConstructors()	宣言されているコンストラクタを取得
void	addField(CtField)	新たなフィールドを追加
void	addMethod(CtMethod)	新たなメソッドを追加
void	addConstructor(CtConstructor)	新たなコンストラクタを追加
void	makeNestedClass	新たなコンストラクタを追加

表 3.8: CtClass のメソッド

返り値の型	メソッド名前	説明
String	getName()	メソッド名、コンストラクタ名を取得
CtClass	getDeclaringClass()	このメソッドを宣言しているクラスを取得
int	getModifiers()	アクセス修飾子を取得
CtClass[]	getParameterTypes()	パラメーターの型を取得
CtClass[]	getExceptionTypes()	投げられる例外の型を取得
void	insertBefore(String)	ボディの直前に処理を追加
void	insertAfter(String)	ボディの直後に処理を追加
void	instrument(ExprEditor)	ボディ内の演算子を編集

表 3.9: CtBehavior のメソッド

ソッドボディの中の演算を順に調べフィールドアクセスやメソッド呼び出し、例外ハンドラの実行を発見すると ExprEditor クラスの edit メソッドに処理が移る。その際、各演算子の情報が edit メソッドの引数に渡される。ExprEditor クラスのメソッドは以下のようなものが定義されている。

```
public class ExprEditor{
  public void edit(NewExpr e) throws CannotCompileException {}
  public void edit(NewArray a) throws CannotCompileException {}
  public void edit(MethodCall m) throws CannotCompileException {}
  public void edit(ConstructorCall c) throws CannotCompileException {}
  public void edit(FieldAccess f) throws CannotCompileException {}
  public void edit(Instanceof i) throws CannotCompileException {}
```

```
  public void edit(Cast c) throws CannotCompileException {}
  public void edit(Handler h) throws CannotCompileException {}
  ...
}
```

各演算子の動作を変更するには、各 edit メソッドをオーバーライドしたクラスを定義する。例えば以下のようなクラスを定義する。

```
01 public MyExprEditor extends ExprEditor{
02   public void edit(MethodCall m)
                            throws CannotCompileException {
03     String clazz = m.getClassName();
04     String method = m.getName();
05     if(clazz.equals("Point") && method.equals("move")){
06       String src =
07          "{"+
08          "  if($1 < 0 || $2 < 0)"+
09          "    throw new RuntimeException();"+
10          "  $_ = $proceed($$);"+
11          "}";
12       m.replace(src);
13     }
14   }
15 }
```

MyExprEditor クラスは ExprEditor クラスを継承して edit(MethodCall m) メソッドをオーバーライドしている。MyExprEditor オブジェクトが CtBehavior の instrument メソッドの引数に渡されるとメソッド呼び出し演算子を検索し、見つかると上記の 3 行目に処理が移る。3 行目では呼び出されたメソッドが宣言されているクラスの名前を取得している。4 行目では呼び出されたメソッドの名前を取得している。そして、呼び出されたメソッドが Point クラスの move メソッドの場合、replace メソッドによりメソッド呼び出しの振る舞いを変更している。１２行目に渡される src 引数の説明をする。$1、$2 はメソッド呼び出しの際の引数を表す。この場合、メソッド呼び出しで渡された引数が 0 以下の場合 RuntimeException を投げる。$_ = $proceed($$) は、もともとメソッド呼び出しの処理を表す。この他、$で始まる特殊変数がいくつか提供されている。例えば、MethodCall オブジェクトに対して、$0 はターゲットオブジェクトを表し、$r は呼び出しの戻り値の型を表す。これらの特殊変数は演算子を表すクラスによって意味が変わる。

advice ボディを埋め込むには MethodCall オブジェクトや FieldAccess オブジェクトのから、ターゲットオブジェクトのクラス名などの情報を取り出し pointcut の条件にマッチするのかを調べる。そして、replace メソッドに渡す引数の $proceed($$) の前にコードを記述すれば before advice が、$proceed($$) の後にコードを記述すれば after advice が実現できる。以上の処理を行い Bugdel では演算に関する Adivce コードの埋め込みを実現している。

Line pointcut の実現

Line pointcut は CtBehavior クラスの insertAt(int line, String src) メソッドを利用して実現している。以下にコード例を示す。

```
CtBehavitor cb = ...;
cb.insertAt(10, "{System.out.println(\"\");}");
```

このコードは CtBehavitor で表されるメソッド又はコンストラクタの１０行目に「System.out.println(””);」で表されるコードを挿入している。insertAt メソッド内ではクラスファイル内に保持されている行番号情報を表す LineNumberTable アトリビュートを利用してコードを挿入する。そのため、挿入先のメソッドのバイトコード（クラスファイル）内には LineNumberTable アトリビュートが必要である。

AllLines pointcut の実現

AllLines pointcut で指定されているメソッド内部に存在する行番号は、クラスファイル中の LineNumberTable アトリビュートを利用して取得する。LineNumberTable アトリビュートは各バイトコードインデックスとソースコードの行番号を対応させるものである。LineNumberTable アトリビュートのフォーマットは次のようになっている。

LineNumberTable アトリビュートのフォーマット

```
LineNumberTable_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 table_length;
  {
    u2 start_pc;
    u2 line_number;
  } line_number_table[table_length];
}
```

u2 は 2 バイトの数値、u4 は 4 バイトの数値を表す。line_number_table のエントリーの start_pc から次のエントリーの start_pc までのバイトコードインデックスが line_number で表される行番号に対応する。ただし、line_number の型が u2 であるので、2 バイトで表せる最大の数６５５３５を超える行番号は行番号アトリビュートには入らない。Bugdel では、これらの情報を取得するために javassist.bytecode パッケージを利用している。以下にメソッド内全ての行番号を取得するコードを示す。

```
01 List getAllLineNumbers(CtBehavior cb){
02   MethodInfo mi = cb.getMethodInfo();
03   CodeAttribute ca = mi.getCodeAttribute();
04   LineNumberAttribute attr = (LineNumberAttribute)
                 ca.getAttribute(LineNumberAttribute.tag);
05   int afterConstructor = ca.iterator().skipConstructor();
06   List lineNumbers = new LinkedList();
07   for (int i = 0; i < attr.tableLength(); i++) {
08     int start_pc = attr.startPc(i);
09     int line_number = attr.lineNumber(i);
10     if (start_pc > afterConstructor)
11       lineNumbers.add(new Integer(line_number));
12   }
13   return lineNumbers;
14 }
```

このコードでは、まず 4 行目でメソッドに対応する LineNumberTable アトリビュートを取得している。Bugdel の仕様では this()、super() の存在する行番号は AllLines pointcut で指定でないようにしいる。そのため、5 行目のコードは this()、super() のコンストラクタ呼び出しが終わるバイトコードインデックスを調べている。メソッド内にコンストラクタ呼び

出しが無い場合には skipConstructor() は-1 を返す。８行目で start_pc の値を取得して、９行目で start_pc に対応する line_number の値を取得している。１０行目で start_pc がコンストラクタ呼び出しの後かどうかを調べ、コンストラクタ呼び出しの後の場合、１１行目で start_pc に対応する行番号を追加している。これにより、コンストラクタ呼び出し以降の行番号を全て取り出す。

以上のようにしてメソッド又はコンストラクタ内の行番号を取得して javassist.CtBehavior クラスの insertAt(int line, String src) メソッドを利用してバイトコードに Adivce ボディを埋め込む。

3.5.2　Inter-type の実現

Inter-type の実現は javassist の CtClass オブジェクトを操作することで行っている。フィールドの追加には CtClass の addField(CtField) メソッドを利用する。以下にコード例を示す。

```
01 CtClazz clazz = ...;
02 String src = "int z = 0;";
03 CtField newField = CtField.make(src, clazz);
04 clazz.addField(newField);
```

このコードでは２行目のソースから３行目で CtField オブジェクトを生成している。そして、４行目でクラスに対して新たなフィールドを追加している。同様にメソッドの追加には CtClass の addMethod(CtMethod) を利用する。以下にコード例を示す。

```
01 CtClazz clazz = ...;
02 String src = "public int getZ(){return z};";
03 CtMethod newMethod = CtNewMethod.make(src, clazz);
04 clazz.addField(newMethod);
```

このコードでは２行目で与えられたソースコードを元に３行目で CtMethod オブジェクトを生成している。そして、４行目でクラスに対して新たなメソッドを追加している。

3.6　Advice 内の特殊機能

3.6.1　thisJoinPoint によるリフレクション

advice コード内で特殊変数 thisJoinPoint を利用することができる。これはリフレクションのための変数である。advice コード内ではこの変数

を使って、各 joinpoint に存在する実行コンテキストにアクセスすることができる。例えば、thisJoinPoint.location 変数は weave 時に joinpoint が存在するソースコードの位置情報に変換される (図 3.16)。

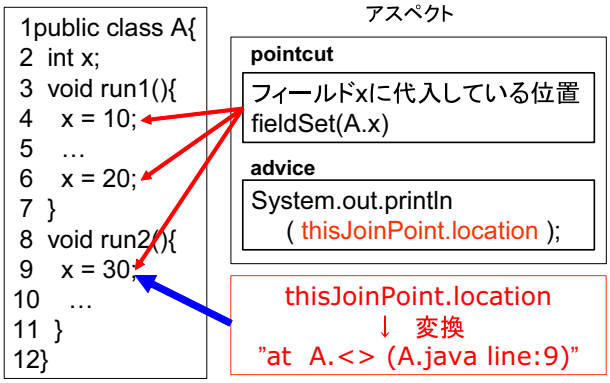

図 3.16: thisJoinPoint.lcoation 変数

以下に thisJoinPoint でアクセス可能な情報を説明する。

thisJoinPoint – java.lang.String 型
: joinpoint の情報 (ソースコードの行番号など) を表す。

thisJoinPoint.location – java.lang.String 型
: joinpoint のソースコードの位置情報 を表す。図 3.16 の例では A クラスが含まれる A.java ファイルの 9 行目を表す文字列に変換されている。また、この文字列を Eclipse のコンソールビューに出力した場合、文字列がソースファイル A.java の 9 行目とハイパーリンクされる。文字列をクリックすると Java エディタで A.java が開かれ 9 行目がハイライト表示される。

thisJoinPoint.line – int 型
: joinpoint が存在する行番号を表す。

thisJoinPoint.file – java.lang.String 型
: joinpoint が存在するソースファイル名を表す。

thisJoinPoint.filePath – java.lang.String 型
: joinpoint が存在するソースファイルのパスを表す。

thisJoinPoint.kind – java.lang.String 型
: joinpoint の種類を表す。

thisJoinPoint.field – java.lang.Object 型
joinpoint の種類がフィールドアクセスの場合フィールドの値を表す。その他は null を表す。

thisJoinPoint.target – java.lang.Object 型
joinpoint の種類がフィールドアクセスまたはメソッド呼び出しの場合ターゲットオブジェクトを表す。その他は null を表す。

thisJoinPoint.within – java.lang.String 型
joinpoint が存在するクラス名を表す。

thisJoinPoint.withincode – java.lang.String 型
joinpoint が存在するメソッド名又はコンストラクタ名とその引数を表す。

thisJoinPoint.variables – java.lang.Object[][] 型
joinpoint に存在する変数の一覧を表す。この値にはローカル変数とクラス変数の情報が含まれる。詳しくは次節せ説明する。

thisJoinPoint.weavehost – java.lang.String[] 型
weave を実行した Eclipse のホスト IP アドレスの一覧を表す。

thisJoinPoint.variables – java.lang.Object[][] 型

thisJoinPoint.variables 変数は joinpoint でアクセス可能なローカル変数とクラス変数の一覧を取得するものである。thisJoinPoint.variables は各変数の名前、値、型、アクセス修飾子の情報をもつ配列であり thisJoinPoint.variables[i][0] は変数の名前、thisJoinPoint.variables[i][1] は変数の値、thisJoinPoint.variables[i][2] は変数の型とアクセス修飾子である。変数の型がプリミティブ型の場合にはプリミティブ型に対応するラッパークラスのオブジェクトが値として含まれている。ローカル変数の変数のアクセス修飾子に local というキーワードが含まれる。例えば thisJoinPoint.variables には以下のような内容が含まれている。

thisJoinPoint.variables の内容

```
thisJoinPoint.variables[0][0] = "x";
thisJoinPoint.variables[0][1] = x;
thisJoinPoint.variables[0][2] = "local int";
thisJoinPoint.variables[1][0] = "MyClass.i";
thisJoinPoint.variables[1][1] = i;
thisJoinPoint.variables[1][2] = "static Integer";
thisJoinPoint.variables[2][0] = "this.y";
thisJoinPoint.variables[2][1] = this.y;
thisJoinPoint.variables[2][2] = "public final long";
```

thisJoinPoint.variables を使って変数の一覧を表示させるには以下のような advice ボディを記述すればよい。

```
01 Object[][] vs = thisJoinPoint.variables;
02 for(int i=0; i<vs.length; i++){
03   String name = (String) vs[i][0];
04   Object value = vs[i][1];
05   System.out.println(name+"="+value);
06 }
```

このように、joinpoint でアクセス可能なローカル変数とフィールド変数の一覧が含まれる。ただし pointcut 指定子が MethodExecution、ConstructorExecution を利用した after advice の advice ボディで利用する場合には、メソッド内で宣言されているローカル変数は含まれない。

thisJoinPoint.variables と実際の変数の同期をとるための特殊メソッドも提供している。$writeVariables() は thisJoinPoint.variables の状態を実際の変数に反映させるメソッドであり、$readVariables() は逆に実際の変数の状態を thisJoinPoint.variables に反映させるメソッドである。例えば int 型の変数 x の値が１００の場合に以下の advice ボディを実行することを考える。

```
01 System.out.println(x);//100 が出力
02 thisJoinPoint.variables[0][1] = new Integer(9999);
03 $writeVariables();//thisJoinPoint の値を反映
04 System.out.println(x);//9999 が出力
```

この場合１行目では１００が出力される。次に２行目で thisJoinPoint.varibales の内容を変更して、３行目で、その変更を反映させている。３行目が実行されると変数 x の値が９９９９になる。そのため４行目では９９９９が出力される。

同じ状況で今度は以下の advice コードを実行することを考える

```
01 System.out.println(thisJoinPoint.variables[0][1]);//100が
出力
02 x = 9999;
03 $readVariables();
04 System.out.println(thisJoinPoint.variables[0][1]);//9999が
出力
```

この場合１行目では１００が出力される。次に２行目で変数 x の値を９９９９に変更している。３行目が実行されると、その変更が thisJoinPoint.variables に反映される。そのため４行目では９９９９が出力される。

thisJoinPoint.variables に含まれるローカル変数の取得の実装方法は、まず weave 時にバイトコードを検査して各 joinpoint でアクセス可能なローカル変数の一覧をデバッグ情報を元に調べる。次に、thisJoinPoint.variables を表すオブジェクトにローカル変数の情報を代入するバイトコードを生成する。

各 joinpoint でアクセス可能なローカル変数の一覧の取得は javassist ライブラリの低レベルバイトコード API、javassist.bytecode パッケージを利用して、クラスファイル中の LocalVariableTable アトリビュートをもとに行っている。クラスファイルの中には各メソッドごとに LocalVariableTable アトリビュートを持っている。ただし、ソースファイルをコンパイルする際にオプション (-g:vars) を付けてコンパイルする必要がある。Bugdel の場合 weave を行う際に -g:vars オプションを付けて、ソースファイルをコンパイルする。LocalVariableTable アトリビュートのフォーマットは次のようになっている。

LocalVariableTable アトリビュートのフォーマット

```
LocalVariable_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 table_length;
  {
    u2 start_pc;
    u2 length;
    u2 name_index;
    u2 descriptor_index;
    u2 index;
  } local_variable_table[table_length];
}
```

各ローカル変数に対して変数が有効な範囲は start_pc から start_pc + length までのインデックスである。変数の名前はクラスのコンスタントプールの name_index 番目から変数の型は descriptor_index 番目から取得することができる。以下に javassist.bytecode パッケージを利用して、メソッド中の、あるバイトコードインデックスで参照可能なローカル変数の情報の一覧を取得するコードを示す。

```
List getLocalVariables(CtBehavior cb, int index) {
  MethodInfo minfo = cb.getMethodInfo();
  CodeAttribute ca = minfo.getCodeAttribute();

  //LocalVariableTable アトリビュートを取得
  LocalVariableAttribute attr = (LocalVariableAttribute)
              ca.getAttribute(LocalVariableAttribute.tag);
  List variables = new ArrayList();
  for (int i = 0; i < attr.tableLength(); i++) {
    int startPC = attr.startPc(i);
    int length = attr.codeLength(i);
    //index がローカル変数の有効範囲に入っているのかを調べる
    if (startPC <= index && index < startPC + length) {
      String name = attr.variableName(i);
      String descriptor = attr.descriptor(i);
      variables.add(new LocalVariable(name, descriptor, cb));
    }
  }
  return variables;
```

```
}
```

thisJoinPoint.variables 変数が記述されている advice ボディを埋め込む際、Bugdel の weaver が挿入先 (joinpoint shadow) のバイトコードインデックスを調べ、以上の方法でそのバイトコードインデックスでアクセス可能なローカル変数の名前と型名の一覧を取得する。次に、「thisJoinPoint.variables の内容」で示したような、各ローカル変数の名前、値、型名を thisJoinPoint.variables 変数へ代入するソースコードを生成し、javassist の Java 用部分コンパイラに渡しコンパイルする。生成したバイトコードを pointcut で指定された位置 (バイトコードインデックス) にコードを挿入する。

thisJoinPoint.variables に含まれるクラス変数の情報の取得の実現は javassist の CtClass オブジェクトを使ってクラスのフィールドを取得する。以下にフィールドの一覧を取得するコードを示す。

```
List getFieldVariables(CtClass clazz) {
  List variables = new LinkedList();
  CtField[] fields = clazz.getDeclaredFields();
  for (int i = 0; i < fields.length; i++) {
    String name = fields[i].getName();
    String signature = fields[i].getFieldInfo().getDescriptor();
    variables.add(new FieldVariable(name, signature, clazz));
  }
  //スーパークラスの変数を追加する
  if (!clazz.getName().equals("java.lang.Object")) {
    CtClass superClass = clazz.getSuperclass();
    variables.addAll(getFieldVariables(superClass));
  }
  return variables;
}
```

上記のコードでは、CtClass の getDeclaredFields() メソッドを使って、そのクラス内で宣言されているフィールドの一覧を取得し、同様にスーパークラスで宣言されているフィールドの一覧も取得する。このフィールドの一覧の情報から変数を読み込むためのメソッドを作成し joinpoint が含まれるクラスに追加する。joinpoint には、生成したメソッドの呼び出しが挿入される。

以上のように thisJoinPoint.variables 変数の実現では Java のリフレクション API である java.lang.reflect パッケージは利用していない。そのため、thisJoinPoint.varibales 変数を利用するにあたって advice ボディを実行する JVM 側に java.lang.reflect パッケージの実装は必要ない。た

だし、ローカル変数をダンプするコードがクラスファイルに挿入されるためローカル変数の数が多い場合、クラスファイルのサイズが大きく増加してしまう。この問題に対して、メソッド内で変数をダンプするためのサブルーチンを作ることで解決できると考えられるが、現在のとことは実装していない。

3.6.2　ビルトインメソッド

Bugdel ではデバッグ、テストのためのビルトインメソッド (特殊メソッド) をいくつか提供している。

bugdel.Bugdel.jump(int line)

このメソッドは引数で指定された行番号へプログラム実行の制御を移すメソッドである。引数で指定する行番号は weave 時にメソッド内で静的に決まる値でなければならない。また、このメソッドが呼び出される同じメソッド内の行番号でなければならない。例えば、Line pointcut によりソースコードの１０行目の joinpoint を指定し、advice ボディとして以下のコードを指定する。

```
bugdel.Bugdel.jump(15);
```

すると１０行目から１４行目に記述されているプログラムは実行されず、１５行目のプログラムから実行される。

C 言語には、このメソッドと同様の機能をもつ goto 文がある。ソースコードの任意の位置にラベルをつけ goto 文により、そのラベルの位置に制御を移すことができる。goto 文はとても強力な機能であるが、安易に利用するとプログラムの構造化が崩れプログラムが分かりにくくなる。また、デバッグが行いにくくなり、バグの原因になってしまうこともある。そのため、Java 言語などのプログラミング言語には取り入れられていない。しかし、デバッグなど一時的にしか使わない場合には有用であると考え Bugdel では提供する。

bugdel.Bugdel.jump メソッドを Bugdel の weaver により埋め込む場合、weave 時にクラスをロードしベリファイヤを実行する。もしも、ベリファイヤによって不正なバイトコードであると判断された場合には weave エラーをダイアログで表示する。例えば、以下のソースコードの２行目を Line pointcut で指定し、Line pointcut に対応する advice ボディとして「bugdel.Buglde.jump(5);」を記述することを考える。

```
1 void foo(){
```

```
2   ...
3   int n = 10;
4   ...
5   ...
6   System.out.println(n);
7 }
```

この場合、３行目のローカル変数 n は初期化されずに６行目で利用されてしまう。以上のように bugdel.Bugdel.jump を利用することにより不正なバイトコードが生成される可能性がある。そこで、weave 時に Java (JVM) のベリファイヤを実行する。不正なバイトコードが生成されたと判断された場合にはベリファイヤにより例外がなげられエラーがダイアログに表示される。ベリファイヤの実装方法は以下のように Class クラスや Method クラス、Constructor クラスのオブジェクトを JVM にロードすることで実現している。

```
01 void verify(String classname) throws java.lang.VerifyError{
02   ClassLoader loader = ...;
03   Class c = loader.loadClass(classname);
04   c.getDeclaredConstructors(); //コンストラクタのチェック
05   c.getDeclaredMethods(); //メソッドのチェック
06 }
```

上記のコードでは３行目でチェックするクラスの Class オブジェクトを生成し、４行目でコンストラクタ、５行目でメソッドに不正なバイトコードが無いかを JVM のベリファイヤを呼び出しチェックしている。不正なバイトコードが発見されると java.lang.VerifyError が発生する。このとき、Eclipse を起動している JVM のベリファイヤを呼び出すため、Eclipse を起動する際にベリファイヤを切っている場合にはチェックは行われない。

bugdel.Bugdel.jump メソッドはバイトコード中に無条件分岐（ワイド・インデックス）を行うバイトコード命令 goto_w を挿入することで実現している。bugdel.Bugdel.jump(int line) メソッドのボディは空であり、weave 時に jump メソッドの呼び出しを Bugdel の weaver が検出し goto_w 命令をメソッド呼び出しの直前に挿入する。goto_w 命令のフォーマットは図 3.17 である。goto_w 命令が JVM で実行されると分岐オフセットが branchbyte1、branchbyte2、branchbyte3、branchbyte4 から、(branchbyte1 << 24) | (branchbyte2 << 16) | (branchbyte3 << 8) | branchbyte4 の値として生成される。そして、goto_w 命令のバイトコードインデックスから分岐オフセット分離れた位置から実行が開始される。

goto_w 命令の埋め込みは javassist ライブラリの低レベルバイトコード API、javassist.bytecode パッケージを利用し以下の手順で行っている。

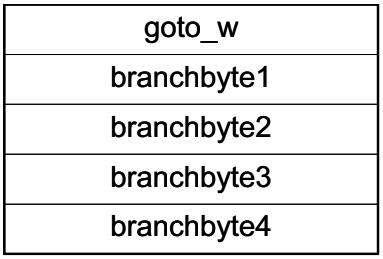

図 3.17: goto_w 命令のフォーマット

Bugdel の weaver が jump メソッドの呼び出しを検出すると、まず、引数 (line) の値を調べる。jump メソッドの仕様では引数は静的に決まる値としている。バイトコード中では jump メソッド呼び出し命令の一つ前の命令は引数の値をスタックに積む命令である。プッシュする命令が bipush、sipush ならば、それらの命令のオフセットから引数の値を計算する。ldc、ldc_w ならばクラスのコンスタントプールから値を取得する。iconst_0、iconst_1、iconst_2、iconst_3、iconst_4、iconst_5、iconst_m1 ならば引数の値は ０、１、２、３、４、５、－１である。iload、iload_0、iload_1、iload_2、iload_3 の場合、これらの命令に対応する istore 命令でストアされる値を istore 命令の一つ前の命令を調べることで再帰的に計算する。引数の値が取得できなければ weave エラーを発生させる。次に、このようにして取得した jump メソッドの引数（ジャンプ先の行番号）に対応するバイトコードインデックスを調べる。その際、クラスファイル中の LineNumberTable アトリビュートを利用する。以上のようにして、ジャンプ先のインデックスを計算し、jump メソッドの呼び出し命令のインデックスとの差をオフセットとした goto_w 命令を作成し jump メソッド呼び出し命令の直前に挿入する。

bugdel.Bugdel.jump(int) メソッドと同様のメソッドである bugdel.Bugdel.jump2(int) メソッドを実装している。bugdel.Bugdel.jump(int) は引数の行番号に制御を移す絶対ジャンプであり、bugdel.Bugdel.jump2(int) はメソッド呼び出しが記述されている行番号に引数の番号を足した行番号へジャンプする相対ジャンプである。

bugdel.Bugdel.breakpoint()

bugdel.Bugdel.breakpoint() メソッドは Eclipse に含まれる Java デバッガでプログラムを実行した場合にスレッドを停止（ブレイク）させるメソッドである。スレッドが停止するとデバッガが停止した位置に対応する

ソースコードをハイライト表示する。Eclipse の Java デバッガ以外で起動した場合には何も行われない。

このメソッドを利用することで pointcut を使ってブレイクポイントの位置を指定できる。また、advice ボディに条件文を記述することで条件付きブレイクポイントを可能にする。例えば、以下のようなコードを advice ボディとして記述する。

```
if(counter >= 10){
  bugdel.Bugdel.breakpoint();
}
```

この advice ボディは counter の値が１０以上の時にスレッドを停止させるものである。

bugdel.Bugdel.breakpoint() メソッドの実装は主に JDT の org.eclipse.jdt.debug.core パッケージを利用して実現されている。Eclipse の Java デバッガでアプリケーションが起動されると Bugdel はデバッガに接続し bugdel.Bugdel.breakpoint() メソッドの呼び出しを監視してスレッドを停止させる。しかし、breakpoint() メソッドにスレッドが達したときにスレッドを止めた場合、breakpoint() メソッドの呼び出し位置では無く breakpoint() メソッド内部でスレッドが停止してしまう。そこで、Bugdel ではスレッドの停止位置をメソッドの呼び出し位置にするため、一度ステップリターンを実行する必要がある。以下にステップリターンを行う Bugdel のプログラムを示す。

```
01 class BugdelBreakpointListener implements IJavaBreakpointListener {
02   public int breakpointHit(
03             final IJavaThread thread, IJavaBreakpoint breakpoint) {
04     if (breakpoint instanceof IJavaMethodBreakpoint) {
05        IJavaMethodBreakpoint jmb = (IJavaMethodBreakpoint) breakpoint;
06        if (jmb.getTypeName().equals("bugdel.Bugdel")
07            && jmb.getMethodName().equals("breakpoint")) {
08          Runnable r = new Runnable() {
09            public void run() {
10              thread.stepReturn(); //ステップリターンを実行
する
11            }
12          };
13          DebugPlugin.getDefault().asyncExec(r);
14        }
15     }
16     return IJavaBreakpointListener.SUSPEND;
```

```
17   }
18   ...
19 }
```

上記のプログラムでは主に JDT に含まれる org.eclipse.jdt.debug.core パッケージ内のライブラリを利用している。引数の IJavaThread はスレッドを表すクラスである。IJavaBreakpoint は Java のデバッガのブレイクポイントを表すクラスである。IJavaBreakpoint のサブクラスにはメソッドの実行直前のブレイクポイントを表す IJavaMethodBreakpoint、フィールドアクセスのブレイクポイントを表す IJavaWatchpoint、ソースコードの行番号のブレイクポイントを表す IJavaLineBreakpoint などがある。上記のプログラムでは５行目で引数の breakpoint が IJavaMethodBreakpoint かどうかを判断している。６行目でブレイクしたメソッド名、宣言されているクラス名を調べて、メソッドが bugdel.Bugdel.breakpoint の場合、８行目以降の処理を実行する。１０行目ではスレッドに対してステップリターンを一度実行している。Eclipse ではデバッガを操作するためのスレッドが決められている。そのため、１３行目ではデバッガ用のプラグインに対して、ステップリターンの要求を出している。このように BugdelBreakpointListener は全てのブレイクポイントを監視し、bugdel.Bugdel.breakpoint のメソッドブレイクポイントが発生した場合にはステップリターンを実行する。BugdelBreakpointListener のインスタンスは Bugdel の起動時に Eclipse の Java デバッガに登録される。

bugdel.Bugdel.openEditor(String host, int port, String filePath, int line)

このメソッドはソースファイルを Bugdel エディタで開くメソッドである。このメソッドを実行するとプログラムを実行している JVM からソケット（TCP プロトコル）を使って Eclipse にアクセスし Bugdel エディタを開く (図 3.18)。host は Eclipse と Bugdel が起動しているホスト名前又は IP アドレス、port は Bugdel がエディタを開く要求を待っているポート番号、filePath は Bugdel エディタで開くソースファイルのパス、line はソースファイル中でハイライトする行番号を指定する。例えば以下のコードを advice ボディに記述すると advice が実行される joinpoint 位置をハイライト表示することができる。

```
String file = thisJoinPoint.filePath;
int line = thisJoinPoint.line;
bugdel.Bugdel.openEditor("localhost", 5555, file, line);
```

上記のコードでは thisJoinPoint 変数を使い、各 joinpoint が存在するファイルのパスと行番号を取得する。取得したファイルの情報を元に Bugdel エディタを開き指定された行番号をハイライト表示するための要求を出す。

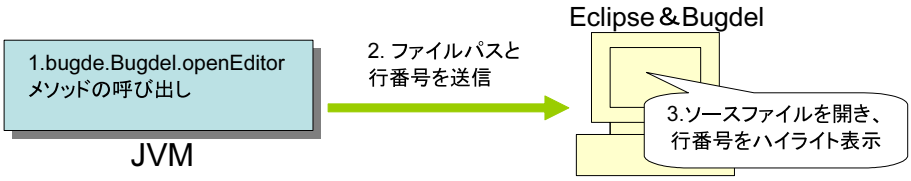

図 3.18: bugdel.Bugdel.openEditor(..) メソッドの概要

Bugdel がソースファイルを開くの要求を待つポート番号はデフォルトでは５５５５であるが、Eclipse のメニューバーから→「ウィンドウ」→「設定」→「Bugdel」を選択して Bugdel preference を開き設定することが可能である。また、プログラムを実行している JVM からソケットを使って Eclipse にアクセスするため、その JVM ではソケット (java.net.Socket クラス) が利用可能な状態でなければならない。サードパーティが独自の JVM を作成し、java.net.Socket クラスを実装していない場合には利用できない。以下に openEditor メソッドの定義の一部を示す。

```
01 public static void openEditor(String host,
02                       int port, String filePath, int line){
03   java.net.Socket sock = ...;
04   java.io.OutputStream outs = sock.getOutputStream();
05   String data = filePath + "[" + line + "]"; //protocol
06   outs.write(data.getBytes());
07   outs.flush();
08   ...
09 }
```

上記の５～６行目で示されているように送信されるデータは、[ファイルパス]”[行番号]”の文字列をバイト配列にした形になっている。そのため java.net.Socket クラスが実装されていなく利用できない場合でも TCP プロトコルによるデータ通信が可能ならば [ファイルパス]”[行番号]”というデータを Bugdel (Eclipse) に送信すれば openEditor メソッドや Socket クラスを利用せずに Bugdel エディタを開くことができる。

次に openEditor メソッド用の Eclipse (Bugdel) 側の実装を説明する。Bugdel は Eclipse が起動すると、指定されたポート番号を開けて Bugdel エディタを開くための要求を待つ。Bugdel が要求を受けるとエディタを

開く要求を Eclipse に出す。しかし、Eclipse では全てのスレッドが UI を操作できるわけではなく UI を操作するスレッドが決められている。そこで、UI を操作するスレッドに対して要求を出すことになる。以下に指定されたソースファイルを開くためのコードを示す。

```
01 public void openBugdelEditor(final String filePath, final int line) {
02   Runnable request = new Runnable() {
03     public void run() {
04       try {
05         //ソースファイルのオブジェクトを取得
06         IWorkspace workspace = ResourcesPlugin.getWorkspace();
07         IFile file = workspace.getRoot().getFile(new Path(filePath));
08         IFileEditorInput input = new FileEditorInput(file);
09
10         //Bugdel エディタでファイルを開く
11         IWorkbench w = PlatformUI.getWorkbench();
12         IWorkbenchPage p = w.getActiveWorkbenchWindow().getActivePage();
13         String editorID = "bugdel.ui.editor.BugdelEditor";
14         ITextEditor editor = (ITextEditor) p.openEditor(input, editorID);
15
16         //指定された行番号をハイライト表示する
17         IDocumentProvider provider = editor.getDocumentProvider();
18         IDocument doc = provider.getDocument(input);
19         IRegion region = doc.getLineInformation(line-1);
20         editor.selectAndReveal(region.getOffset(), region.getLength());
21       } catch (Exception e) {
22         e.printStackTrace();
23       }
24     }
25   };
26   //UI を操作するスレッドを取得する
27   Display display = Display.getCurrent();
28   if (display == null) {
29     display = Display.getDefault();
30   }
31   //UI を操作するスレッドに要求を出す
32   display.syncExec(request);
33 }
```

このメソッドでは６、７行目で filePath で表されるパスに存在する IFile オブジェクトを Eclipse のワークスペースから取得している。その IFile オブジェクトを元に８行目でエディタで開くための FileEditorInput オブジェクトを生成する。１１、１２行目で現在アクティブになっているページを取得する。１４行目で Bugdel エディタを開く。その際、Bugdel エディタの ID である “bugdel.ui.editor.BugdelEditor” を渡している。この ID は Bugdel の plugin.xml ファイルに記述されているものである。１７～２０行目は引数で指定された行番号をハイライトする。IEditorInput オブジェクトはバイナリファイルを含めたファイルを表すオブジェクトなので、行番号を調べるためのメソッドは提供されていない。そこで、１７、１８行目で IDocument オブジェクトを取得し、１９行目で引数で表される行番号に対応するソースコード範囲を調べる。そして、２０行目で調べた範囲をハイライト表示する。Eclipse では UI を操作するスレッドが決められているため、２７～３０行目で UI を操作するスレッドを取得し３２行目でエディタを開く要求を出す。

3.7　設定ファイル

3.7.1　ライブラリの設定

bugdel_x.x.x.zip を解凍して生成されるフォルダ plugins の中の plugins/bugdel_x.x.x/ config/ copyliblary フォルダ内のファイルは、Bugdel で weave を実行する際にクラスファイルの出力フォルダにコピーされる。出力フォルダにコピーされるため Bugdel で指定するデバッグ用 advice や inter-type の中で利用することができる。bugdel_x.x.x.zip をダウンロードした際には copyliblary フォルダには初めから bugdel フォルダがあり、その中に Bugdel.class ファイルが含まれている（図 3.19）。Bugdel.class には 3.6.2 節で説明したビルトインメソッドを利用するためのメソッドの定義が含まれている。

3.7.2　デフォルト advice ボディの設定

bugdel_x.x.x.zip を解凍して生成されるフォルダ plugins の中の plugins/bugdel_x.x.x/ config/ defaultInsertStatement フォルダの中のファイルを編集することでデフォルトの advice コードを指定できる。例えば、Line pointcut を指定した際に自動的に LINE_POINTCUT.txt に記述されているコードが advice ボディに指定される。ALL.txt の内容は全ての pointcut に対応する advice ボディに記述される（図 3.19）。その他、各

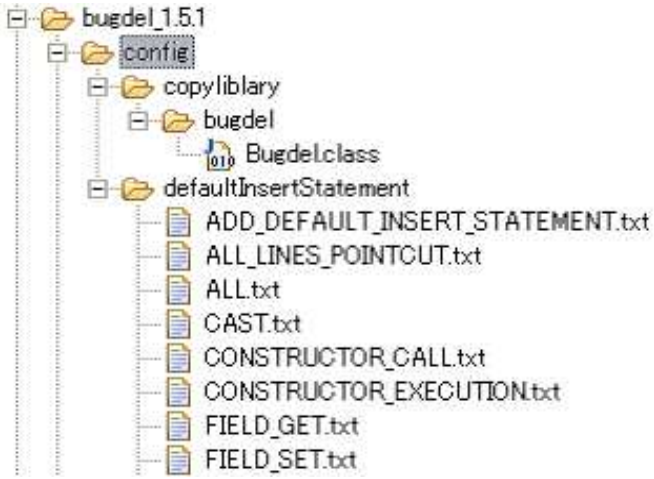

図 3.19: Bugdel の設定ファイル

pointcut 指定子に対応する設定ファイルがフォルダに含まれている。

第4章　応用(ブレイクポイントのエミュレート)

デバッガ（ブレイクポイント）を実現するためのミドルウェアとして Bugdel を利用することができる。デバッガを作成するアーキテクチャとしては JPDA (Java Platform Debugger Architecture) [27] が有名である。しかし、JPDA の場合、デバッグ対象プログラムを JPDA がサポートされている JVM 上で動かすため JVM には JPDA のバッグエンドである JVMDI (Java Virtual Machine Debug Interface) の実装が必要である。JVMDI の実装を JVM に行うには多くの労力がかかるため、JVMDI の実装が無い JVM ではデバッガを作成するのが困難であるという問題がある。一方、Bugdel を利用することで JVM の変更せずにデバッガ（ブレイクポイント）を実現することができる。JVM の変更が必要ないため JPDA よりも容易にブレイクポイントを実現することができる。この章では JPDA の問題点と、Bugdel を使ったデバッグ機能の実現、特にブレイクポイントの実現方法について述べる。

4.1　JPDA を利用したデバッガ作成の問題点

4.1.1　JPDA とは

JPDA は複数のプラットフォームのさまざまな JVM 実装に対するデバッガアプリケーションを容易に作成できるようにするためのデバッガアーキテクチャである。JPDA の構成の概要は図 4.1 である。JPDA には次の 3 つのインターフェイスがある。

Java Virtual Machine Debugger Interface (JVMDI)
: JVMDI は VM が提供するデバッグサービスを定義するものであり、JVM に実装されているネイティブインタフェースである。デバッグするときに JVM に必要なサービスを定義する。現在のスタックフレームなどの情報、ブレークポイントの設定などのアクション、ブレークポイントに到達したときなどの通知 に対する要求を含む。

Java Debug Wire Protocol (JDWP)

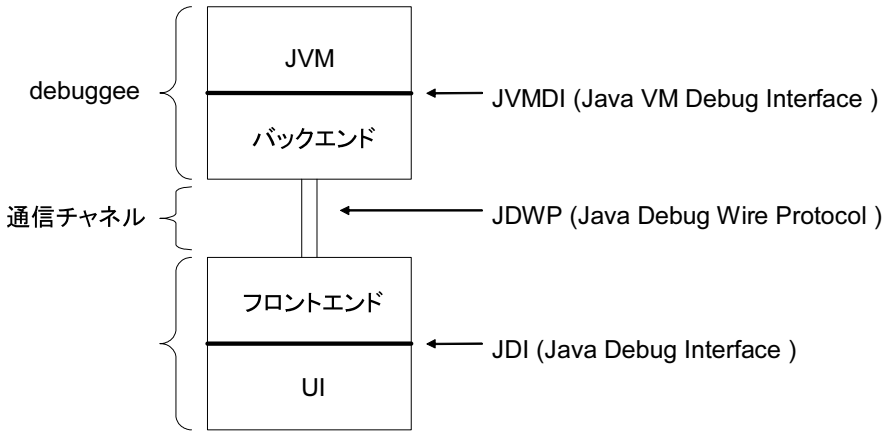

図 4.1: Java Platform Debugger Architecture

デバッギー とデバッガのプロセス間で行われる通信を定義する。デバッギー プロセスとデバッガのフロントエンドとの間で転送される情報および要求の形式を定義する。このプロトコルの仕様によって、デバッギー とデバッガのフロントエンドを、異なる実装の JVM で実行できるようになる。さらに、Java 以外の言語で記述されたフロントエンド、またはネイティブ以外の デバッギー も使用できるようになる。

Java Debug Interface (JDI)
フロントエンドによって実装される 100% Pure Java インタフェースである。ユーザコードレベルでの情報および要求を定義する。デバッガの実装側では、JDWP が直接使用されるか JVMDI が使用される。これを使用することにより、ツール開発者は、リモートデバッガアプリケーションを容易に記述でき、デバッグ機能を開発環境に容易に統合することができる。

デバッギーはデバッグを実行しているプロセスで、デバッグしているアプリケーションとアプリケーションが動作している JVM、デバッガのバックエンドから成る。VM はデバッグしているアプリケーションが動作している JVM である。JVM の実装はさまざまなベンダーが提供しており、JPDA を利用する場合には JVM に応じた JVMDI の実装が必要になる。バックエンドはデバッガのフロントエンドから VM に対して要求が送信され、要求に応じた情報がフロントエンドに返信される。バックエンドと VM 間の通信は JVMDI を利用して行われる。通信チャネルはデバッガのフロントエンドとバックエンドの通信を表す。通信されるデータの形式やセマンティックスは JDWP に指定される。フロントエンドはバッグエ

ンドから受け取る JDWP からの情報が含まれる。JDWP からの情報は JDI を通して利用される。UI はデツールベンダから、実装が提供されるものであり GUI デバッガなどを作成できる。JDK にはコマンドラインベースの JBD が実装されている。

Sun Microsystems は JVMDI、JDWP、JDI インタフェースの仕様に加え、リファレンス実装も提供している。リファレンス実装は、次のようなコンポーネントで構成される。

- 複数プラットフォームの Sun VM 用 JVMDI 実装リファレンス。
- JVMDI を使って JDWP の デバッギー 側を実装するバックエンド。
- JDWP のデバッガ側を使って JDI を実装するフロントエンド。
- JDI 上に構築した 2 つのサンプルデバッガアプリケーション。

4.1.2　問題点

JDPA を利用してブレイクポイントを実現することが可能であるが問題がいくつかある。この節では JPDA を利用したデバッガ作成の問題点を述べる。

JVM に合わせた JVMDI 実装

JPDA のフロントエンドの実装 (JDI の実装リファレンス) は Java で作成されるため、どのプラットフォーム、どの JVM でも動作する。そのため、通常、デバッガアプリケーションを作成するサードパーティは JDI のみを参照して実装を行う。統合開発環境である Eclipse に含まれている Java 用の GUI デバッガも JDI を参照して作成されている。バックエンドの JVMDI リファレンス実装を新しいプラットフォームに移す場合でも、多くの場合、ソースにわずかの変更を加えるか、ソースをまったく変更せずに、再コンパイルするだけで済む。

しかし、Sun VM と大きく異なる JVM には新たな JVMDI の実装を行う必要がある。例えば、サードパーティが独自に実装した特殊な JVM には、それぞれ独自の JVMDI の実装が必要である。JVMDI の実装には、JVM のデータ構造に深く踏み込む必要があり、イベントを取得するために JVM 実装の中にフックを設定する必要がある。 そのため JVMDI のサポートがない JVM に JVMDI の実装を追加するには、JVM の複雑さと、実装する JVMDI のオプション機能の量に応じて、多くの作業が必要になる。J2sdk1.4.2 に含まれる Soraris 用 JPDA のバッグエンド

の JVMDI 実装は、ヘッダファイル、実装ファイル合わせて 83 ファイル、総行数 21706 行にものぼる。このように、JVMDI の実装が無いプラットフォームや JVM のデバッガを作成するには、JVM の実装を理解し、その上で JVMDI の実装を作成する必要があり、多くの時間と労力がかかる。

JIT コンパイラによるコード変換

JIT コンパイラとはプログラムを実行する際にバイトコードをプラットフォームに依存するネイティブコードに変換するコンパイラである。JIT コンパイラをサポートした JVM ではプログラムを実行する際、初めはバイトコードを一命令ずつインタープリタ方式で実行する。プログラムの実行中にプロファイリングを行いメソッドの呼び出し回数を数える。呼び出し回数の多いメソッドは JIT コンパイラによりネイティブコードに変換される。バイトコードからネイティブコードへの変換にはある程度、時間がかかるが、生成されたネイティブコードはインタープリタ方式よりも高速に実行できるため、プログラム全体としては実行速度を向上させることができる。

JIT コンパイラはプログラムの実行速度を向上させるという利点があるが、JPDA と併用すると問題になる場合がある。JIT コンパイラはコードの速度向上を目的としているため、バイトコードからネイティブコードへ変換する際、さまざまなコードの最適化を行う。最適化にはメソッドのインライン展開などがある。メソッドがインライン展開されると、もともとのプログラムの中にはメソッド呼び出しが無くなってしまう [19]。そのため、JPDA を利用してメソッドの呼び出し位置をブレイクポイントで指定していても、メソッドの呼び出しが行われないためブレイクしないとう問題がある。また、デバッグ情報が利用できなくなるという問題がある。デバッグ情報はソースファイルからバイトコードに変換されるときにソースコードコンパイラによって付け加えられる。デバッグ情報にはソースコードの行番号情報やローカル変数情報などが含まれる。これらの情報は行番号を指定してプログラムをブレイクさせる場合やローカル変数の値を評価する際に利用される。ソースコードコンパイラによって付け加えられるデバッグ情報はソースコードとバイトコードをマッピングするものである。JIT コンパイラによりバイトコードがネイティブコードに変換された後もデバッグ情報を利用する場合には、バイトコードからネイティブコードに変換する際に、バイトコード用のデバッグ情報をネイティブコード用のデバッグ情報に変換しなければならない [31]。サードパーティが作成した JIT コンパイラの中にはデバッグ情報をネイティブコード用に変換しないものもある。そのため、JVM によってはデバッグモード (JPDA) と JIT

の併用を禁止しているものがある。

デバッガによる計算資源の消費

プログラムの通常モードでの実行に比べてデバッグモードでの実行では CPU やメモリなどの多くの計算資源を利用する。デバッグモードでプログラムを起動する場合、デバッギー側の JVM ではメソッド呼び出しやフィールドアクセス、例外処理などのイベントを監視しする。デバッガのエンドユーザが指定していないイベントに対しても処理を行うため、CPU やメモリなどの計算資源を必要以上に消費してしまう。これにより、実行中のプログラムが不安定になったり、デバッガ自体の挙動が不安定になったりする可能性がある。そのため、実際にプログラムが利用される場合と異なる状況でデバッグすることになる。

4.2　Bugdel を利用したブレイクポイントの実現

Bugdel を利用してスレッドを停止させるメソッドをプログラム中に埋め込むことでブレイクポイントをエミュレートすることができる。その際、JVM を変更する必要はなく、デバッグモード (JPDA) のサポートも必要ない。例えば以下のようなコードをプログラムを停止させたい位置に埋め込む。

ブレイクポイントを実現するコード

```
1 Object[][] variables = thisJoinPoint.variables;
2 showVariables(variables);
3 String file = thisJoinPoint.filePath;
4 int line = thisJoinPoint.line;
5 bugdel.Bugdel.openEditor("localhost",5555,file,line);
6 suspendThreadAndOpenResumeDialog();
```

このコードでは 1 行目で変数の一覧を取得する。2 行目は取得した変数の一覧を表示させるものである。上記のコードでは showVariables メソッドを記述したが、これは擬似的なものである。変数の一覧を表示させるメソッドはデバッガの作成者が自由に実装するものである。デバッガの作成者が変数をどのように表示するのかを決めて実装する。例えば、コマンドライン上に表示させたり、GUI で表示させたりする。場合によっては通信を行い他のホストに変数の情報を送信し、他のホスト上に変数の情報を表示させる。通信を行うために JVM 側では TCP プロトコルによる通信が利用可能でなければならない。3 〜 5 行目はブレイクする位置に対応する

ソースコードを Eclipse の Bugdel エディタで開くコードである。これにより、デバッガのユーザにブレイクした位置を視覚的に知らせることができる。６行目はスレッドを停止させ、停止させたスレッドを再開させるためのダイアログを開くものである。suspendThreadAndOpenResumeDialog メソッドも擬似的なものであり、デバッガの作成者が実装を行う。スレッドを停止させるメソッドは場合によってはプラットフォーム、JVM に依存するネイティブメソッドになる。

上記のコードを Bugdel のプラグイン設定ファイルの /config /default-InsertStatement フォルダ内にある設定ファイルに記述すれば、ブレイクポイントを実現する advice コードを毎回、記述する必要は無く、コードの挿入位置を指定するだけでブレイクポイントが実現できる。図 4.2 は上記のコードを利用した場合のスクリーンショットである。

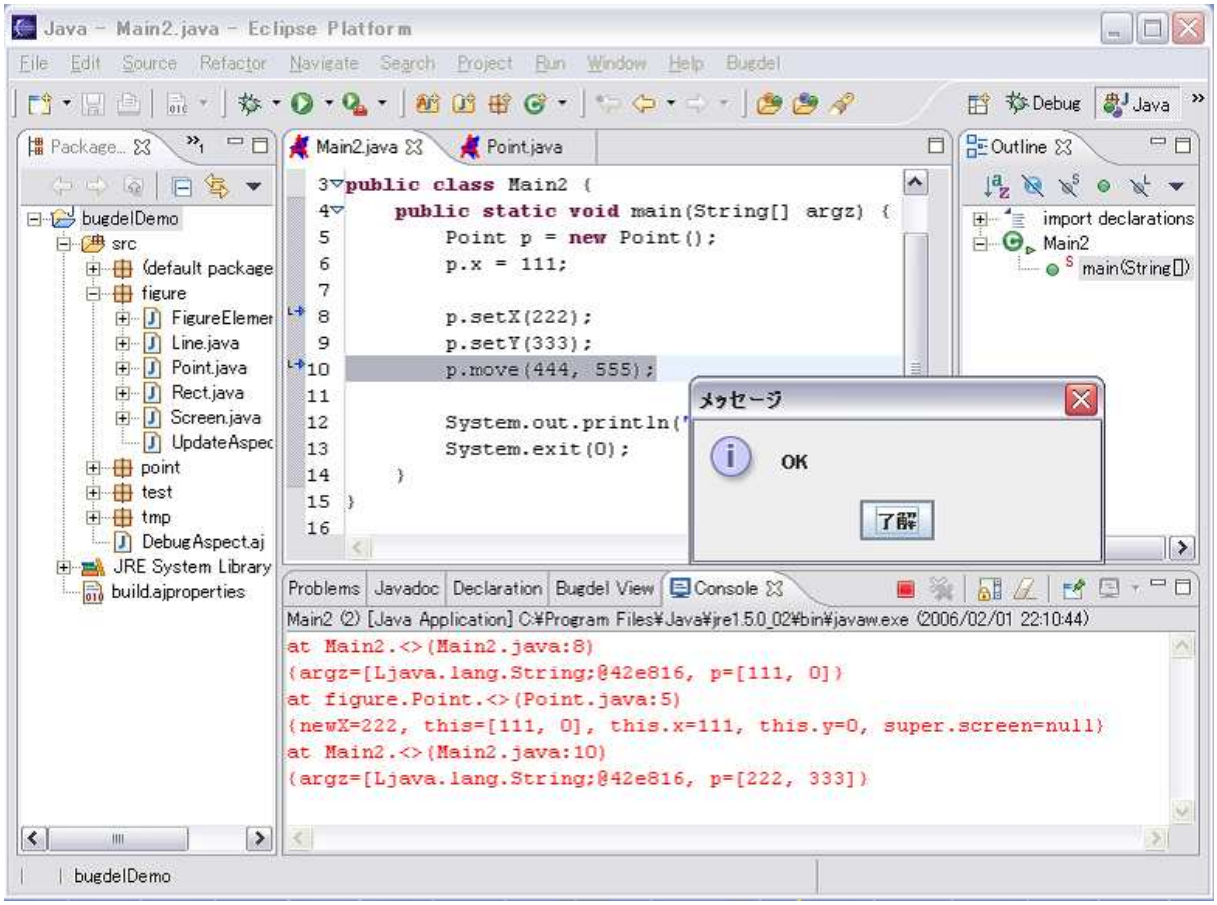

図 4.2: ブレイクポイントのエミュレート

ブレイクポイント位置の指定

ブレイクポイントの位置は pointcut を利用して指定する。メソッドの入口や出口でブレイクさせたい場合には MethodExecution pointcut、フィールドアクセス位置でブレイクさせたい場合には FieldSet、FieldGet pointcut を利用する。ソースコードの特定の行番号でブレイクさせたい場合には Line pointcut を利用する。また、メソッドの中を一行ずつステップ実行させたい場合には AllLines pointcut を利用する。

JDI (Java Debugger Interface) で提供されているブレイクポイントイベントを Bugdel の pointcut でエミュレートする場合の対応を表 4.1 に示す。これらの pointcut 指定は Bugdel のユーザインターフェイスを利用することで簡単に指定することができる。

	JDI のイベントクラス	Bugdel の pointcut
メソッドの入口	MethodEntryEvent	MethodExecution (before)
メソッドの出口	MethodExitEvent	MethodExecution (after)
フィールドアクセス	AccessWatchpointEvent	FieldGet
フィールド代入	ModificationWatchpointEvent	FieldSet
ファイルの行番号	BreakpointEvent	Line
例外の発生	ExceptionEvent	(Handler)
ステップ実行	StepEvent	(AllLines)
クラスのロード	ClassPrepareEvent	MethodExecution (clinit, before)
クラスのアンロード	ClassUnloadEvent	-

表 4.1: JDI (com.sun.jdi パッケージ) と pointcut の対応

利点

Bugdel を利用したブレイクポイントの実現では、クラスファイルにブレイクポイント用のメソッドを埋め込む。そのため、デバッグ対象プログラムを実行するために特殊な JVM は必要なく、ブレイクポイントを実現するために JVM を改造する必要もない。JIT コンパイラの最適化によってブレイクポイントで指定したイベントのコードが削除されることもない。さらに、メソッド呼び出しやフィールドアクセスなどのイベントのうち、pointcut で指定していない位置ではブレイクポイント用のコードば

実行されないため、不必要なイベント箇所でメモリや CUP などの計算資源を消費することがない。そのため JPDA を使った場合と比べると、ブレイクポイント以外の位置では実際にプログラムが利用される環境に近い状況で実行しデバッグすることができる。

限界

Bugdel を利用したブレイクポイントの実現では weave 時に静的にブレイクポイント用のコードを挿入する必要がある。そのため、ブレイクポイント位置を静的に決める必要がある。そのため、JPDA を利用して作成したデバッガのようにプログラムの実行をブレイクポイントにより停止させて、状況に応じて新たなブレイクポイントをインタラクティブに指定するということはできない。ステップ実行については AllLines pointcut によりブログラムのソースコードを一行づつ実行させることはできるが、ステップオーバーやステップリターンを行うには制御が移る先のメソッドをあらかじめ pointcut で指定しておく必要がある。

システムクラスなどクラスファイルの改変ができないクラス内に対してはブレイクポイントを埋め込むことができない。しかし、システムクラスなどソースコードが無いクラスをデバッグする必要性は低いと考えられる。

4.3　その他のデバッグ機能

この章では特にブレイクポイントを実現するための方法を示した。その他、Bugdel を利用することで、さまざまなデバッグ機能を実現することができる。例えば、methodExecution pointcut を使って全てのメソッドの前後にメソッドの情報を追加、削除するような advice を実行することでスタックトレースを実現することができる。以下にスタックトレースを実現するコード例を示す。

スタックトレースを実現するコード

```
public class StackTraceMgr extends ThreadLocal {
  protected synchronized Object initialValue() {
    return new Stack();
  }
  private static ThreadLocal tl = new StackTraceMgr();
  public static void push(Object obj) {
    ((Stack) tl.get()).push(obj);
  }
  public static void pop() {
    ((Stack) tl.get()).pop();
  }
  public static Stack getStack() {
    return (Stack) tl.get();
  }
}

実行する Advice 1
pointcut:[methodExecution]
class name:[!StackTraceMgr]
method name:[*]
insert point:[before]
insert statement:[
  String clazz = thisJoinPoint.within;
  String method = thisJoinPoint.withincode;
  StackTraceMgr.push(clazz+"#"+method);
]

実行する Advice 2
pointcut:[methodExecution]
class name:[!StackTraceMgr]
method name:[*]
insert point:[after]
insert statement:[
  StackTraceMgr.pop();
]
```

上記の StackTraceMgr クラスはスレッドに応じて Stack オブジェクトを割り当て、Stack オブジェクトに対してプッシュ（オブジェクトの追加)、

ポップ（オブジェクトの削除）をするクラスである。Advice 1 によって全てのメソッドの実行直前に実行中のメソッドを含むクラスの名前とメソッドの名前をスタックにプッシュされる。Advice 2 によって Advice 1 でスタックに積んだ情報がポップされる。

「StackTraceMgr.getStack()」コードを実行することで実行中のスレッドのスタック情報を含む Stack オブジェクトを取得することができる。取得した Stack オブジェクトを利用してスレッドのスタック情報を表示する。また、上記の Advice 1 では実行中のクラス、メソッド名をスタックにプッシュするコードを示したが、表示するスタック情報に応じて insert statement を変える。例えば、変数の一覧を表示するためには thisJoinPoint.variables を利用して変数の一覧を取得して StackTraceMgr にプッシュする。

4.4　SuperJEngine 用デバッガとしての利用

Bugdel は SuperJEngine [17] 用デバッガのミドルウェアとして採用されている。SuperJEngine とは日立ソフトウェアエンジニアリング（株）が開発しているマイコン用（J2ME）の実行環境である。SuperJEngine の JVM にはデバッグモード (JVMDI 実装) がサポートされていないため、JPDA を利用したデバッガを作成することができない。そこで、Bugdel を利用してブレイクポイントなどのデバッグ用の機能を実現している。日立ソフトが DCT（デバッグ用クラスツール群）を作成し、DCT を利用したデバッグ用コードを Bugdel を使ってクラスファイル中のさまざまな箇所に埋め込むことでデバッグ機能を実現している。

第5章　実験

5.1　Weaver

Java Grande Forum ベンチマークプログラム

Java Grande Forum [2] の Sequential Benchmark を使ってベンチマークテストを行った。比較対象は通常コンパイルのみを行い advice の実行を行わないプログラムとカウンターの値をインクリメントする advice コード（Test.n++）を全てのメソッド呼び出し位置に Bugdel と AspectJ/AJDT を使って挿入したものである。実験環境は以下である。

ベンチマークプログラム Java Grande ForumSequential Benchmark バージョン 2.0, Section 3, Size B (1)

weave するアスペクト 全てのメソッド呼び出し位置でカウンターの値をインクリメントする advice（Test.n++）を実行

コンパイル、weave 環境 AspectJ:version1.5.0

コンパイルしたプログラムの実行環境 OS:Solaris 9、CPU:UltraSPARC-III Cu 900MHz x 4、メモリ:16GB、JVM:Java HotSpot(TM) Server VM (build 1.5.0_05-b05)

Java Grande ForumSequential Benchmark の Euler、Molecular、Monte Carlo、Ray Tracer、Search のそれぞれのプログラムに対して advice が無い場合、Bugdel を使って advice コードを埋め込んだ場合、AspectJ を使って埋め込んだ場合の結果と Bugdel により生成したプログラムを実行した際の advice の実行回数を表 5.1 に示す。

実験結果から AspectJ と Bugdel による実行時間の差はほとんど無いことが分かる。

	Euler	Molecular	Monte Carlo	Ray Tracer	Search
advice 無し (秒)	48	182	153	103	37
Bugdel (秒)	48	184	155	148	40
AspectJ (秒)	48	185	155	149	39
advice の実行回数 (回)	43,357,901	2,279,161	185,845,851	670,583,894	305,175,833

表 5.1: Java Grande によるベンチマーク

Xerces のソースコード

Xerces[4] のソースコードを利用して Bugdel と AJDT によるコンパイル、weave の速度を測定した。コンパイル、weave には Eclipse を利用しコンパイル用のダイアログボックスが生成され破棄されるまでの時間を測定した。実験環境は以下である。

ベンチマークプログラム xerces-j2.7.1 のソースファイル (ファイル数 942、ファイルサイズ 11.8MB) に対してコンパイル、weave を行う。コンパイル、weave したプログラムを利用して約 1MB の xml ファイルをパーズして検索を行う。

コンパイル、weave 環境 OS：WindowsXP、CPU:Pentium4 2.66GHz、メモリ:1.0GB JVM:Java HotSpot(TM) Client VM (build 1.5.0_05-b05)、

Eclipse:version3.0.2、AJDT:version1.2.0、AspectJ:version1.5.0、

Eclipse の起動オプション:-vmargs -Xmx512m -Xms512m

コンパイル時にデバッグ情報（ローカル変数アトリビュート、行番号アトリビュート、ソースファイルアトリビュート) を加える

コンパイルしたプログラムの実行環境 OS:Solaris 9、CPU:UltraSPARC-III Cu 900MHz x 4、メモリ:16GB、JVM:Java HotSpot(TM) Server VM (build 1.5.0_05-b05)

weave するアスペクトは以下の 5 種類を用意し測定した。

1. * で表されるクラスの全てのフィールド代入、フィールド参照、メソッド呼び出し、メソッド実行、コンストラクタ実行位置でカウンターの値をインクリメントする advice（Test.n++）を実行する。対象となる型 (クラス、インターフェイス) は 892 個。
2. org..* で表されるクラスの全てのフィールド代入、フィールド参照、メソッド呼び出し、メソッド実行、コンストラクタ実行位置カウン

ターの値をインクリメントする advice（Test.n++）を実行する。対象となるクラス、インターフェイス (匿名クラスを含む) は 875 個。

3. org.apache.xerces..* で表されるクラスの全てのフィールド代入、フィールド参照、メソッド呼び出し、メソッド実行、コンストラクタ実行位置カウンターの値をインクリメントする advice（Test.n++）を実行する。対象となるクラス、インターフェイス (匿名クラスを含む) は 550 個。
4. org.apache.xerces.impl.d*..* で表されるクラスの全てのフィールド代入、フィールド参照、メソッド呼び出し、メソッド実行、コンストラクタ実行位置カウンターの値をインクリメントする advice（Test.n++）を実行する。対象となるクラス、インターフェイス (匿名クラスを含む) は 100 個。
5. org.apache.xerces.impl.dtd..* で表されるクラスの全てのフィールド代入、フィールド参照、メソッド呼び出し、メソッド実行、コンストラクタ実行位置カウンターの値をインクリメントする advice（Test.n++）を実行する。対象となるクラス、インターフェイス (匿名クラスを含む) は 30 個。
6. aaa..* で表されるクラスの全てのフィールド代入、フィールド参照、メソッド呼び出し、メソッド実行、コンストラクタ実行位置カウンターの値をインクリメントする advice（Test.n++）を実行する。対象となるクラス、インターフェイス (匿名クラスを含む) は 0 個。

通常コンパイル、Bugdel と AJDT による weave 速度、Bugdel による advice コードの挿入ヶ所の結果は表 5.2、weave 後のプログラムの実行時間の結果は表 5.3 である。

weave するアスペクト	1	2	3	4	5	6
通常コンパイル (秒)	8	8	8	8	8	8
Bugdel (秒)	66	58	52	25	24	20
AJDT/AspectJ (秒)	180	121	102	25	20	17
advice の挿入 (ヶ所)	70,146	59,930	53,575	6,992	3,093	0

表 5.2: Xerces のコンパイル、weave 時間

Bugdel では advice コードが各 pointcut shadow でインライン展開され挿入される。それに対し AspectJ では advice を実行するためのメソッドが各 pointcut shadow に挿入される。そのため、AspectJ の方がメソッド呼び出しを行う分、実行時間が長くなると考えられる。

weave するアスペクト	1	2	3	4	5	6
通常コンパイル (秒)	3.9	3.9	3.9	3.9	3.9	3.9
Bugdel (秒)	6.7	6.5	6.5	3.9	3.9	3.9
AJDT/AspectJ (秒)	7.6	7.5	7.2	4.2	4.2	3.9

表 5.3: Xerces の実行時間

5.2　Advice コードの記述時間

ユーザが AspectJ (AJDT) と Bugdel を利用して、同じ効果の advice コードを記述したときの記述する時間を測った。記述する advice は figure.Point クラスのフィールド x の代入位置を pointcut として指定し、before　advice を宣言する。advice ボディは何も行わないコードである。

AspectJ (AJDT) を使ってユーザが入力するコードは以下である。

```
before():set(* figure.Point.x){
}
```

このコードはスペース文字、改行文字を含めて全 35 文字である。コードの入力はあらかじめ用意されている DebugAspect.aj ファイルに行う。advice を入力する前に DebugAspect.aj ファイルには以下の内容が記述されている。

```
public aspect DebugAspect{

}
```

DebugAspect.aj が Eclipse の Java/AspectJ エディタ で開かれアクティブになっている状態で、ユーザが advice コードを入力する時間を測った。

Bugdel では Bugdel エディタを利用する方法（1）と advice ダイアログに入力する方法（2）の 2 種類を測定した。Bugdel エディタを利用する方法（1）では、figure.Point クラスが定義されている Point.java ファイルが Bugdel エディタで開かれアクティブになっている状態で時間を測りはじめる。そして、Point.java ファイル中のフィールド x にカーソルを合わせ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、「pointcut」アイテムを選択し pointcut の候補をダイアログに表示させ FieldSet を選択して、ダイアログの OK ボタンを押すまでの時間を測った。Bugdel ではデフォルトで before アドバイスが指定されるので、今回の実験では advice の insert point に対して before 又は after をユーザが選択する必要はない（図 5.1）。

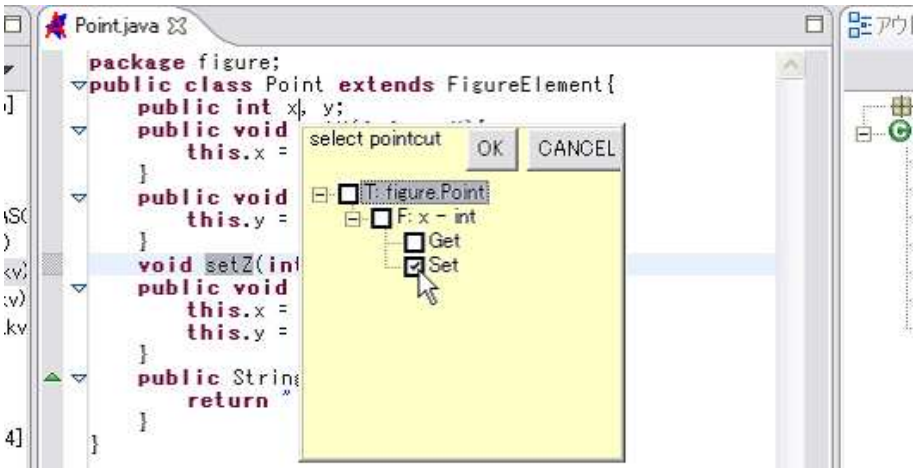

図 5.1: Bugdel エディタの利用

advice ダイアログに入力する方法（２）では Bugdel ビューのメニューバーから pointcut ボタンを押し始めてからの時間を測りはじめる。pointcut ボタンを押すと advice ダイアログが表示される。ユーザはダイアログの中で pointcut 指定子として FieldSet をラジオボタンで選択し、class name テキストフィールドに “figure.Point” を入力し、field name テキストフィールドに “x” を入力し、insert point で before を選択する。ユーザが advice ダイアログの各項目を入力して、OK ボタンを押すまでの時間を測定した。(図 5.2)

１２人のユーザにそれぞれ advice の指定を行ってもらい、各方法での時間を測りその平均を出した。測定の結果は表 5.4 である。

AspectJ/AJDT	Bugdel(1)	Bugdel(2)
9.9 (秒)	5.9 (秒)	9.8 (秒)

表 5.4: advice 記述の時間

AspectJ/AJDT を使った場合９．９秒に対して Bugdel エディタを利用した場合（１）では５．８秒、advice ダイアログを利用する方法（２）では９．８秒で指定が行えた。今回の実験では pointcut を指定する際に “figure.Point.x” を引数に記述した。AspectJ/AJDT を使って pointcut 記述を行う時間はクラスやフィールドの名前の長さに比例すると考えられる。それに対し Bugdel エディタを利用した方法（１）の場合、ユーザは変数名をクリックして pointuct の候補を表示するので指定する時間はクラス名やフィールド名の長さに関係なく一定である。そのため、クラス名

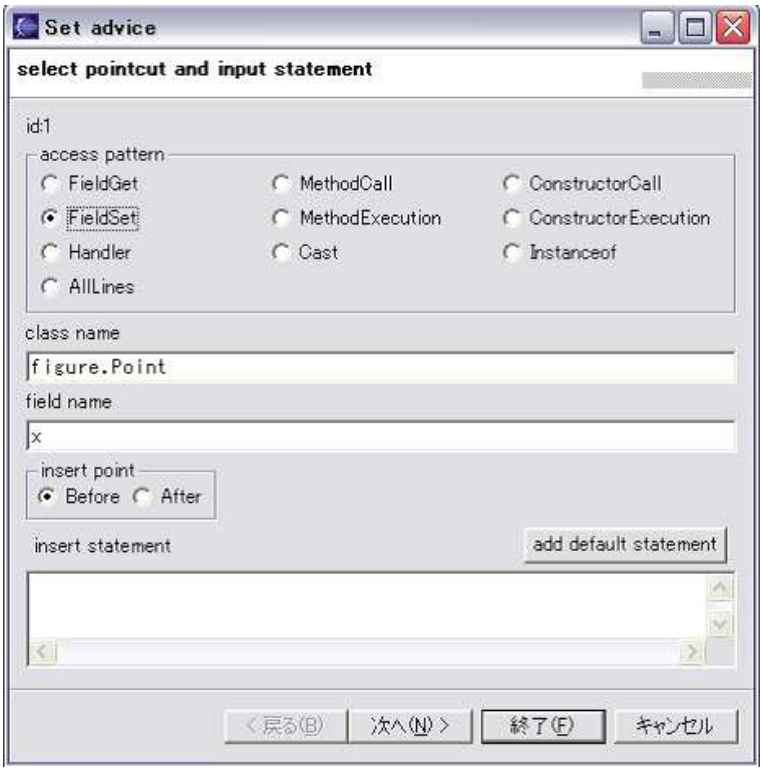

図 5.2: advice ダイアログの利用

やフィールド名が長くなると、AspectJ/AJDT を利用した場合と Bugdel エディタを利用した場合の差が大きくなると考えられる。

第6章　まとめ

本稿ではテスト、デバッグ作業を支援するための Java 用開発環境 Bugdel を提案した。Bugdel ではアスペクト指向を利用してテスト、デバッグ用の追加コードを指定する。テスト、デバッグ作業を行う際には開発中のプログラムにさまざまなコードを追加する。これらの追加コードはプログラム本来の機能とは関係ないため、ソースコード中に記述するとソースコードの可読性が低下する。さらに、追加コードの挿入、削除を行う際、誤ってその周辺のプログラムを変更してしまうという危険性もある。一方、Bugdel を利用することでプログラムのソースコードとは別にこれらの追加コードを記述できる。さらに、pointcut によりパターンを記述し追加コードの実行位置をまとめて指定することができる。

Bugdel はテスト、デバッグに特化したアスペクト指向システムであり AspectJ のような汎用的なアスペクト指向システムには無い機能を提供する。まず、Line pointcut、AllLines pointcut によりソースコードの行番号を pointcut で指定し追加コードを実行できる。次に advice ボディから pointcut で指定した joinpoint に存在するローカル変数へのアクセスを可能にし、ローカル変数のリフレクション機能も提供する。これらの機能はクラスのモジュール性、カプセル化を壊すため汎用的なアスペクト指向システムでは提供していない。しかし、Bugdel ではクラスのモジュール性よりもテスト、デバッグの際の有用性を考え提供している。Bugdel は統合開発環境である Eclispe 上に実装しており、有用なユーザインターフェイスを提供している。Bugdel エディタを使いソースコード上のクラス名、メソッド名、フィールド名をマウスでクリックすることで pointcut の指定を行え、ソースコードブラウザを利用することでクエリーを発行し検索を行いながら pointcut の指定が行える。そのため AspectJ のように宣言的に pointcut を指定するシステムと比べて pointcut を記述する手間が少なくなる。また、pointcut に影響のあるソースコード編集を監視し、変更がある場合には警告を出す機能を提供している。これにより、ユーザの意図に反した pointcut のターゲットの変更に気づくことができる。

また本稿では Bugdel の応用例としてブレイクポイントをエミュレートする方法を示した。デバッガ（ブレイクポイント）を実現する方法としては JPDA が有名であるが、JVM に対して JVMDI の実装行わなければな

らない。一方、Bugdel ではブレイクポイントをエミュレートする advice コードをクラスファイルに埋め込むため、ブレイクポイントを実現するために特殊な実行環境は必要なく通常の JVM でブレイクポイントを実現できる。

実験では Bugdel の weaver の実行速度と weaver によって生成されたバイトコードの実行速度を計測した。さらに、ユーザに Bugdel を利用してもらい advice コードの記述時間を測り AspectJ/AJDT との比較を行った。

Bugdel の配布

Bugdel をインターネット上 [32] でオープンソースソフトウェアとして公開し配布してきた。ライセンスは CPL (Common Public License v1.0)[1] である。公開して以降もユーザからの意見をもらい追加機能の実装やバグの修正を行ってきた。このようにソフトウェアの品質を高め、バージョンアップを繰り返してきた。その間 Eclipse のバージョンも 2.0、2.1、3.0、3.1 とバージョンアップがあり、それによる Eclipse アーキテクチャの仕様変更に Bugdel を対応させた。

その結果、2004 年 11 月〜2005 年 12 月のホームページ (トップページ) への総アクセス数は 14733、月平均 1052、ダウンロード総数は 3754、月平均 268 であった（図 6.1)。さらに、Bugdel は日立ソフトエンジニアリング (株) が開発している商用のマイコン用 Java 実行環境 SuperJEngine® のデバッガのミドルウェアとして採用されている。

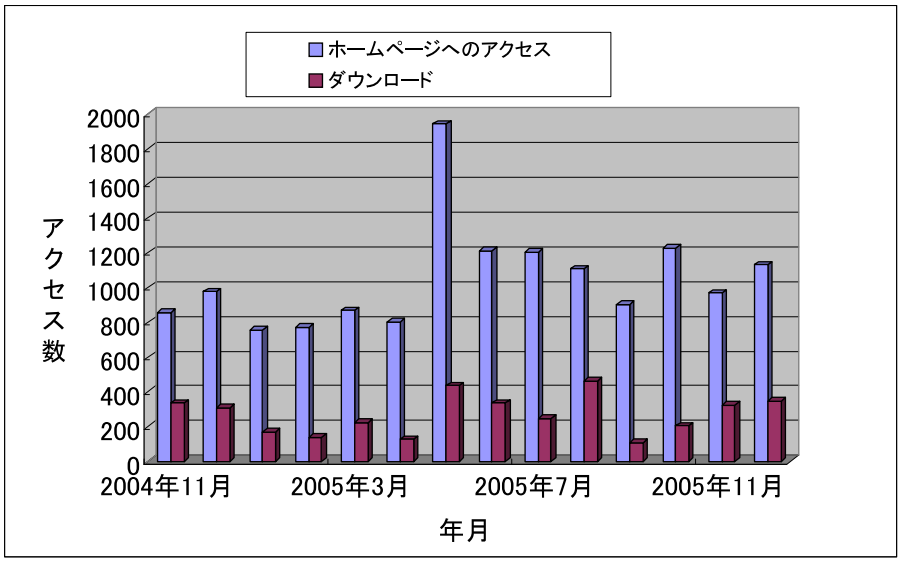

図 6.1: ホームページへのアクセス数とダウンロード数

参考文献

[1] *Common Public License - v 1.0*, http://www.eclipse.org/legal/cpl-v10.html.

[2] *The Java Grande Forum benchmark suite*, http://www.javagrande.org.

[3] AJDT Project: *AspectJ Development Tools (AJDT)*, http://www.eclipse.org/ajdt/.

[4] Apache Software Foundation: *The Apache XML Project*, http://xerces.apache.org.

[5] Apache Software Foundation: *Log4j project - Introduction*, http://logging.apache.org/log4j/docs/index.html.

[6] AspectJ Project: *The AspectJ project at Eclipse.org*, http://www.eclipse.org/aspectj/.

[7] Charfi, A. and Mezini, M.: Using Aspects for Security Engineering of Web Service Compositions, *Proceedings of the 2005 IEEE International Conference on Web Services, Volume I*, pp. 59–66 (2005).

[8] Chiba, S.: *Javassist Home Page*, http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html.

[9] Chiba, S.: Load-time Structural Reflection in Java, *European Conference on Object-Oriented Programming (ECOOP 2000)*, LNCS 1850, Springer Verlag, pp. 313–336 (2000).

[10] Chiba, S. and Nishizawa, M.: An Easy-to-Use Toolkit for Efficient Java Bytecode Translators, *2nd International Conference on Generative Programming and Component Engineering (GPCE 2003)*,, LNCS 2830, Springer Verlag, pp. 364–376 (2003).

[11] Codehaus: *AspectWerkz*, `http://aspectwerkz.codehaus.org/index-aw.html`.

[12] Eclipse Project: *Eclipse.org home*, `http://www.eclipse.org`.

[13] Fairbrother, S., Kehn, D., Kellerman, J., McCarthy, P. and D'Anjou, J.: *The Java Developer's Guide To Eclipse*, Addison-Wesley (2004).

[14] Gamma, E. and Beck, K.: *Contributing To Eclipse: Principles, Patterns, And Plug-Ins*, Addison-Wesley (2003).

[15] Harrison, W., Ossher, H., Sutton, S. and Tarr, P.: Concern modeling in the concern manipulation environment, *MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, ACM Press, pp. 1–5 (2005).

[16] Hilsdale, E. and Hugunin, J.: Advice weaving in AspectJ, *3rd international conference on Aspect-Oriented Software Development (AOSD2004)*, ACM Press, pp. 26–35 (2004).

[17] Hitachi Software Engineering Corp: *SuperJEngine®*, `http://www.hitachi-sk.co.jp/Products/SuperJ/`.

[18] Irwin, J., Loingtier, J.-M., Gilbert, J. R., Kiczales, G., Lamping, J., Mendhekar, A. and Shpeisman, T.: Aspect-Oriented Programming of Sparse Matrix Code, *International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE 1997)*, LNCS 1343, Springer, pp. 249–256 (1997).

[19] Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H. and Nakatani, T.: A study of devirtualization techniques for a Java Just-In-Time compiler, *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, ACM Press, pp. 294–310 (2000).

[20] Janzen, D. and Volder, K. D.: Navigating and Querying Code Without Getting Lost, *the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, ACM Press, pp. 178–187 (2003).

[21] JDT Java Development Tools subproject: *Eclipse Java Development Tools*,
`http://www.eclipse.org/jdt/`.

[22] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *European Conference on Object-Oriented Programming (ECOOP 2001)*, LNCS 2072, Springer, pp. 327–353 (2001).

[23] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C. V., Maeda, C. and Mendhekar, A.: Aspect-Oriented Programming, *European Conference on Object-Oriented Programming (ECOOP 1997)*, LNCS 1241, Springer, pp. 220–242 (1997).

[24] Lopes, C. V. and Kiczales, G.: Recent Developments in AspectJ, *Aspect-Oriented Programming workshop at ECOOP'98*, LNCS 1543, Springer (1998).

[25] Popovici, A., Gross, T. and Alonso, G.: Dynamic Weaving for Aspect-Oriented Programming, *Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD 2002)*, ACM Press, pp. 141–147 (2002).

[26] Sato, Y., Chiba, S. and Tatsubori, M.: A Selective, Just-In-Time Aspect Weaver, *Proc. of Generative Programming and Component Engineering (GPCE 2003)*, LNCS 2830, pp. 189–208 (2003).

[27] Sun Microsystems, Inc: *Java Platform Debugger Architecture (JPDA)*, http://java.sun.com/products/jpda/.

[28] Sun Microsystems, Inc: *Java^{TM} Logging APIs*,
`http://java.sun.com/j2se/1.5.0/docs/guide/logging/index.html`.

[29] Tarr, P., Ossher, H., Harison, W. and Jr, S. M. S.: N degrees of separation: multi-dimensional separation of concerns, *in International Conference on Software Engineering (ICSE 1999)*, IEEE Computer Society Press, pp. 107–119 (1999).

[30] Tatsubori, M., Sasaki, T., Chiba, S. and Itano, K.: A Bytecode Translator for Distributed Execution on Legacy Java Software, *Proceedings of the European Conference on Object-Oriented Programming(ECOOP2001)*, LCNS2072, pp. 236–255 (2001).

[31] Tikir, M. M., Hollingsworth, J. K. and Lueh, G.-Y.: Recompilation for debugging support in a JIT-compiler, *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, New York, NY, USA, ACM Press, pp. 10–17 (2002).

[32] Usui, Y.: *Bugdel Home Page*, Bugdel Project, `http://www.csg.is.titech.ac.jp/~usui/bugdel/`.

[33] Usui, Y. and Chiba, S.: Bugdel: An Aspect-Oriented Debugging System, *First Asian Workshop on AOSD, In Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, IEEE Press, pp. 790–795 (2005).

[34] Volder, K. D.: *JQuery: a query-based code browser*, `http://jquery.cs.ubc.ca`.

[35] Volder, K. D.: *Tyruba website*, `http://tyruba.sourceforge.net`.

[36] Volder, K. D.: *Type-Oriented Logic MetaProgramming*, PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory (1998).

[37] 薄井義行, 千葉滋: アスペクト指向を利用してデバッグコードを挿入できるソフトウェア開発環境, コンピュータソフトウェア ソフトウェア科学会, Vol. 22, No. 3, pp. 229–234 (2005).

[38] 千葉滋, 立堀道昭: Java バイトコード変換による構造リフレクションの実現, 情報処理学会 論文誌, Vol. 42, No. 11, pp. 2752–2760 (2001).

発表リスト

発表論文

- Yoshiyuki Usui and Shigeru Chiba: “Bugdel: An Aspect-Oriented Debugging System”, First Asian Workshop on AOSD, In Proc. of Asia-Pacific Software Engineering Conference (APSEC 2005), IEEE Press, pages 790-795, Taipei, Taiwan, December 15 - 17, 2005.
- 薄井 義行・千葉 滋: “アスペクト指向を利用してデバッグコードを挿入できるソフトウェア開発環境”, コンピュータソフトウェア, vol.22, no.3, pp.229-234, ソフトウェア科学会, 岩波書店, 2005.
- 薄井義行・千葉滋: “アスペクト指向を利用してデバッグコードを挿入できるソフトウェア開発環境”, 日本ソフトウェア科学会第 21 回大会, 東京工業大学, 2004 年 9 月 15 日〜17 日.

その他の発表

- 第17回 コンピュータシステム・シンポジウム, OpenSolarisChallenge, 筑波大学, 2005 年 11 月 29 日〜30 日. (ポスター, デモ発表)
- 第 4 回 SPA サマーワークショップ, 山梨県笛吹, 2005 年 8 月 23 日. (ポスター発表)
- International Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago USA, March 14-18, 2005. (ポスター発表)
- 第 8 回 プログラミングおよび応用のシステムに関するワークショップ (SPA2005), 群馬県伊香保, 2005 年 3 月 7 日〜9 日. (ポスター発表)
- AOP ワークショップ, 第 3 回 SPA サマーワークショップ, 静岡県伊東, 2004 年 8 月 23 日〜24 日. (口答発表)

- 第 299 回 Programming Tools and Techniques (PTT), 東京工業大学, 2004 年 4 月 22 日. (口答発表)
- 第 7 回プログラミングおよび応用のシステムに関するワークショップ (SPA2004), 長野県上諏訪, 2004 年 3 月 1 日〜3 日. (ポスター発表)

報道

- “日立ソフト，東工大のアスペクト指向ツールを組み込み Java デバッグ製品に応用”, Tech-On!, 日経 BP, 2006 年 1 月 24 日.
- “東工大が開発した AOP オープンソース・ソフト，日立ソフトが Java 開発環境に採用”, IT Pro, 日経 BP, 2005 年 5 月 20 日.