平成15年度学士論文

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

東京工業大学 理学部 情報科学科

学籍番号 00-0348-7

薄井 義行

指導教官

千葉 滋 助教授

平成16年2月5日

概要

ソフトウェアの開発・保守を行う上で必ずデバッグ作業が行われる。デバッグ作業ではプログラムの動作確認のためコンソール (コマンドライン) に値や文字列を表示させる。その際、プリント文 (System.out.println、printf) をソースコード中に記述する方法が行われる。これによりソフトウェアのバグを発見することができ質の高いソフトウェアを開発できる。

ところが、上記のようにデバッグコードをソースコード中に直接記述した場合、ソフトウェアを配布する際に取り除く必用がある。取り忘れがあるとプログラムの実行中に不要な処理 (プリント文実行など) が行われてしまう。また、デバッグコードを挿入する位置を一ヶ所ずつ指定する必要があり効率的ではない。例えば、変数 x の値が変更される度に、その値をコンソールに出力させたいと考える。すると変数 x を変更している場所を探し出し、全ての場所にプリント文を入れなければならない。

それに対し、近年研究が行われているアスペクト指向プログラミングを使うことで効率よくデバッグコードの挿入を行えると考えられる。アスペクト指向プログラミングとは、プログラム中のさまざまな場所に散らばってしまう処理 (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。アスペクト指向プログラミングによりデバッグコードを挿入する位置をまとめて指定することができる。Java を言語拡張したアスペクト指向言語 AspectJ では、pointcut 記述 (コード挿入位置を指定するもの) によりフィールドアクセスやメソッド呼び出し、コンストラクタ実行位置などを指定する。そして、それらの条件に合う位置全てにデバッグコードを挿入することができる。また、デバッグコードを開発中のプログラムコードとは別に記述することができる。これによりデバッグのためにソースコードを変更する必要がない。既存のアスペクト指向プログラミング言語として上で述べた AspectJ がある。また、AspectJ プログラミングを統合開発環境で支援するためのツール AJDT(Eclipse のプラグイン) が開発されている。

ところが AspectJ は汎用的な言語であるためデバッグ作業には不向きな点がある。例えばデバッグ作業中、ソースコードの特定の行にデバッグコードを挿入したいと考える。しかし、AspectJ はモジュール性の観点から行単位でコードを挿入する機能を提供していない。また、デバッグコー

ドの挿入位置を指定する際、AspectJ の文法にそった pointcut 記述を行う必要がある。しかしデバッグのために pointcut 記述を行うのではプログラマへの負担が大きい。このように AspectJ にはデバッグに不向きな点がある。これらの問題は AJDT のような AspectJ プログラミング支援ツールを使っても解決することはできない。

そこで我々はこれらの問題に対処するために、Java プログラムに対し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 Bugdel を提案する。Bugdel はデバッグに特化したアスペクト指向プログラミング環境を実現しており、デバッグ作業に有効な行単位の pointcut を提供している。また、フィールドアクセスやメソッド呼び出しなどの pointcut 指定を GUI(グラフィカル・ユーザ・インターフェイス) で行える。そため複雑な pointcut 記述を行う必要がない。Bugdel を使うことでデバッグコードの挿入を効率的に行え、デバッグ作業の負担が減らされる。

Bugdel の開発にあたり統合開発環境である Eclipse 上にプラグインとして実装した。Eclipse 上に実装することで、既に提供されているエディタ機能や Java のソースコード解析機能を利用することができる。また、Bugdel ではデバッグコードの挿入をバイトコード (class ファイル) を解析し変換することで行う。バイトコードの解析、変換にはバイトコード編集ライブラリである Javassist を用いた。最後に我々はデバッグコードの埋め込みを AJDT と Bugdel を使ってそれぞれ行い、その実行時間を測定した。その結果、Bugdel は AJDT と比べて遜色のない時間でデバッグコードの埋め込みが行えることを確かめた。

謝辞

本研究を進めるにあたり、システムの提案や数々の助言をしていただいた指導教官の千葉滋助教授に感謝いたします。また、論文のスタイルファイルを作成していただいた東京工業大学の光来健一氏、アスペクト指向プログラミングや有用なライブラリの使い方を教えていただいた佐藤芳樹氏、プログラムの設計や論文の書き方などの相談に乗って頂いた西澤無我氏、参考になる文献や実験のための資料を紹介していただいた中川清志氏、論文の書き方やコンピュータの高度な使い方を教えていただいた柳澤佳里氏、松沼正浩氏に感謝いたします。また、卒業研究を行う上で励ましていただいた同研究室のみなさんに感謝します。

目 次

図 目 次

表 目 次

第1章　はじめに

ソフトウェアの開発・保守を行う上でデバッグは重要な作業であり必ず行われる。例えば、ソフトウェアの開発中、作成したプログラムが正しく動いているのかを確認するために、プリント文 (System.out.println()) などのデバッグコードをソースコードに埋め込み、変数の値や文字列をコンソールに出力させる。また、プログラムは if 文の中を実行しているのかを確認するために if 文の中に「System.out.println(”exec if”)」を挿入して確認を行う。このようにソースコードに直接デバッグコードを書いてデバッグを行うことができる。これによりプログラム中のバグを発見することができ、高品質のソフトウェアを開発することができる。デバッグの方法としてデバッガを使う方法もあるが、サーバサイドプログラミングなどではデバッガを利用することができない場合がある。この場合にはやはりログ出力という方法が用いられる。

しかし、デバッグコードはデバッグ作業が終わると必要のないものであり取り除く必要がある。取り忘れがあると、配布したソフトウェアの実行中に不要なログ出力などが実行されてしまう。配布したソフトウェアの実行中にデバッグコードが実行されるのを防ぐためにロギング API やアサーション機能、条件付コンパイルなどの技術がある。しかし、これらの技術を使った場合でもソースコード中になんらかのデバッグコードが混入してしまう。そのためソースコードを公開しているソフトウェアやチームでを開発している場合はやはりデバッグコードを取り除く必要があり、同様に取り忘れの可能性がある。

また、以上のデバッグの技術ではデバッグコードを挿入する位置を一箇所ずつ指定する必要がある。例えば、プログラム中のある変数 x が変更される度にその値をコンソールに出力させたいと考えると変数 x を変更している位置をソースコード中から全て見つけ出しプリント文を入れる必要があり効率的ではない。そこで、本研究ではアスペクト指向を利用してデバッグコードを挿入するソフトエア開発環境 Bugdel を提案する。Bugdel を使うことでデバッグコードとソースコードを分離して記述することができ、また効率よくデバッグコードを挿入することができる。

アスペクト指向プログラミングを使うことで効率よくデバッグコードを挿入できると考えられる。アスペクト指向プログラミングとはプログラ

ム中に散らばってしまう処理 (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。Java を言語拡張したアスペクト指向プログラミングではフィールドアクセスやメソッド呼び出し、コンストラクタ実行位置を pointcut 記述により指定する。そして pointcut で記述された条件に一致するプログラム中の全ての位置に、新たなコードを挿入することができる。さらにアスペクト指向プログラミングを使うことで挿入するデバッグコードを開発中のソースコードとは別に記述ですることができ、デバッグのためにソースコードを変更する必要がない。このようにアスペクト指向を使うことでデバッグコードの挿入を効率よく行うことができ、デバッグコードの取り忘れという問題も起こらない。アスペクト指向プログラミングとして上で述べた AspectJ があり、また AspectJ プログラミングを統合開発環境で支援するツールとして AJDT がある。

しかし、AspectJ は汎用的な言語であるためデバッグ作業には不向きな点がある。例えば、デバッグ作業中プログラマはソースコード中の特定の行にデバッグコードを挿入したいと考える。ところが AspectJ は行単位の pointcut を提供していない。なぜなら AspectJ は汎用的な言語であり横断的関心事のモジュール化やコード挿入によるプログラムの整合性を重視している。行単位の pointcut は挿入先のソースコードの実装に依存しすぎておりアスペクトのモジュール性を損ねてしまうため提供されていない。同じ理由により if 文の pointcut やローカル変数の pointcut も提供されていない。また、AspectJ ではデバッグコードを挿入する際 AspectJ の文法に沿った pointcut 記述を行う必要がありプログラマへの負担が大きい。また、挿入するコード内では挿入先のローカル変数や private 変数にアクセスすることができないため、それらの値を確認することができない。このように AspectJ ではデバッグ作業に不向きな点があり、これらの問題は AJDT のようなツールを使っても解決することができない。

それに対し Bugdel はデバッグに特化したアスペクト指向プログラミング環境を提供することが目的であるため AspectJ には無かった行単位の pointcut を提供している。これによりデバッグで有効な行単位のデバッグコードの挿入が行える。また、pointcut 指定を GUI(グラフィカル・ユーザ・インターフェイス) で行える。例えば、あるメソッド m() のメソッド呼び出し位置を pointcut で指定する場合、エディタ上のソースコードのメソッド名 m() をクリックすることで行える。また、行単位の pointcut は行番号の左側 (ルーラー) をクリックすることで行える。こうにデバッグ挿入のために複雑な pointcut 記述を行う必要がない。Bugdel を使うことでデバッグコードの挿入を効率的に行え、ソフトウェア配布時にデバッグコードを削除する必要もないためデバッグ作業の負担が減らされる。

Bugdel の開発にあたり統合開発環境である Eclipse 上に実装した。Eclipse

はプラグインベースのアーキテクチャであり新たなプラグインを提供することで機能を拡張できる。本システム Bugdel もプラグインとして作成されている。Eclipse にはエディタ機能や Java ソースコード解析機能が他のプラグインから提供されており Bugdel はこれらの機能を利用している。また、Bugdel ではデバッグコードの挿入をバイトコードを解析し変換することで行う。バイトコードの解析、変換にはバイトコード編集ライブラリである Javassist を用いた。最後に我々はデバッグコードの埋め込みを AJDT と Bugdel を使ってそれぞれ行い、その実行時間を測定した。その結果、AJDT と比べて遜色のない時間でデバッグコードの埋め込みが行えることを確かめた。

以下２章で、既存のデバッグ支援ツールの問題点と AspectJ を使ったデバッグコード挿入の問題点を指摘する。３章で Bugdel の仕様および実装方法を述べ、４章で実験による考察をし、５章で本稿をまとめる。

第2章　デバッグを支援するツールとその問題点

2.1　デバッグとは

デバッグとはプログラムの誤り (バグ) を取り除く作業でありソフトウェアの開発・保守を行う上で必ず行われる。デバッグを行う方法としてプログラムのソースコード中に直接デバッグコードを記述する方法がある。例えば、プログラムの実行中、ある変数の値を確認したい場合ソースコード中に System.out.println(value); を挿入したり、プログラムが if 文の中を実行しているのか確認するために System.out.println(”if exec”) を挿入したりしてコンソールにログを出力させる。このようにデバッグ用の処理をソースコード中に直接記述して行うこともできるが、ソフトウェアを配布する場合、デバッグコードを取り除く必要がある。デバッグコードの取り忘れがあるとプログラムの実行中に不要なログ出力などが行われてしまう。これらの問題を解決し、デバッグ作業をもっと効率よく行うために、さまざまツールが開発されている。この節ではデバッグを行うツールを説明し、その問題点を指摘する。

2.2　java.util.logging パッケージの利用

java.util.logging パッケージ [5] はログの出力をサポートするクラス群である。このパッケージは J2SE1.4 以降の標準 API に含まれている。

デバッグの作業やプログラムが正常に動いているのか監視するためにログを採るということがよく行われる。例えば、プログラムがどのメソッドを実行中であるのか確認するためにメソッドの名前をコンソールに出力したり、変数の値の履歴を残したりすために値をファイルに出力する。

ログ出力のプログラム例

```
public class Server{
  public void service(){
    //ログ出力コード
    System.out.println("start service");
```

```
    メイン処理
    //ログ出力コード
    System.out.println("finish service");
  }
}
```

この例ように「System.out.println();」をソースコード中に書くことでコンソールにログの出力を行うことができる。しかし、ログ出力の目的がデバッグのなど一時的なものである場合、ソフトウェアの配布時に取り除く必要があり、採り忘れがあるとソフトウェアの使用中に不要なログが出力されてしまう。また、ログ出力コードを取り除いた後、デバッグのために再びログ出力を行わせる場合には再度ログ出力コードを挿入しなければならない。ログの出力先をコンソールからファイルに変更する場合や、出力するログに時刻などの新たな情報を付加する場合は、全てのログ出力コードを変更する必要がある。

そこで、java.util.logging パッケージを利用してログの出力を行う。

java.util.logging パッケージの利用例

```
import java.util.logging.Logger;
public class Server {
  public void service() {
    //ログ出力コード
    Logger.global.info("start service");
    メイン処理
    //ログ出力コード
    Logger.global.info("finish service");
  }
}
```

上の例では Logger クラスで定義されている java.util.logging.Logger クラスの static 変数 global(Logger オブジェクト) にログを出力を行わせている。この例ではログの出力先などデフォルトのものを使用しているが、java.util.logging パッケージで用意されているクラスを使い Logger オブジェクトに対して、設定を行うことでログの出力を制御できる。

- 出力先

  System.err に出力をする ConsoleHandler クラス、ファイルに出力する FileHandler クラス、ネットワーク越しに出力を行う SocketHandler クラス、メモリのなどが用意されている。また、Handler クラスを用いて出力先を定義することが可能である。

- 出力フォーマット

  テキストベースで出力する SimpleFormatter クラス、XML ベースで出力する XMLFormatter が用意されている。また、Formatter クラスを用いて、出力されるログに時刻を付加するなど、独自のフォーマットを定義することが可能である。

- ログのレベル

  ログを採る際、ログのレベルを設定し、レベルに応じてログ出力の有無を決めることができる。デフォルトでは重要度の高い順に SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST の 7 種類が定義されている。例えば、Logger オブジェクトのログレベルを CONFIG に設定した場合、CONFIG よりも重要度の高いレベルのログが出力される。また、Level クラスを用いて独自のレベルを定義することが可能である。

java.util.logging パッケージの利用例 2

```
import java.util.logging.*;
public class Server {
  public static Logger logger;
  static{
    //出力先の設定
    Handler handler = new ConsoleHandler();
    //フォーマットの設定
    handler.setFormatter(new XMLFormatter());
    //ログレベルの設定
    handler.setLevel(Level.WARNING);
    //新規ロガーの作成
    logger = Logger.getLogger("MyLogger");
    //出力先の追加
    logger.addHandler(handler);
  }
  public void service() {
    //ログ出力コード
    logger.severe("start service");
    メイン処理
    //ログ出力コード
    logger.warning("finish service");
  }
}
```

このように java.util.logging パッケージを利用することでログ出力の制御を行い、デバッグ用のログ出力をソフトウェアの配布時に簡単に取り除くことができる。また、ログのレベルを指定することで不要なログ出力を防ぐことができる。このパッケージと同様にログ出力を支援するツールとして apache jakarta プロジェクトの log4j や Logkit などがある。

しかし、ロギング API を使った場合ソースコード中にログ出力用のコードが混入してしまい可読性の低下という問題が残る。

2.3　アサーション

アサーションとは実行中のプログラムをテストする機能である。ログ出力の場合、プログラムが正しく実行されているかどうかプログラマがログを見て判断する必要がある。それに対してアサーションを使った場合、自動的にプログラムが正しく実行されているのか判断することができる。アサーションを使うにはソースコード中にアサーションコードを書き、コンパイル時、又は実行時のオプションによってテスト用のコード実行の有無が決まる。そのため、ソフトウェア配布時にテストコードの取り忘れが無くなる。

Java では J2SDK1.4 以降に、この機能が採用されている。テストを行う際には assert 文を使う。assert 文の文法は

A　assert 式 1;

B　assert 式 1 : 式 2;

であり式 1 は boolean 型を返す式であり、その位置で成立すべき条件を書く。プログラムを実行した際、式 1 が評価され false の場合は java.lang.AssertionError が throw される。B の場合 AssertionError オブジェクトの生成に式 2 の返り値が引数として渡される。以下にアサーションの例を示す。

```
class TestAssertion{
  int amount = 0;
  void run(){
    処理
    assert amount >= 0;
    処理
    assert amount >= 0 : "amount は 0 以上";
  }
}
```

上の例では amount が 0 以上かどうかテストするために assert 文を挿入している。assert 文をバイトコード中に埋め込むにはプログラムをコンパイルする際に

```
>javac -source 1.4 TestAssertion.java
```

と入力する。また、実行の際 assert 文を有効にする場合

```
>java -ea TestAssertion
```

と-ea オプションを付けて実行する。オプションを付けない場合には assert 文は実行されない。

このようにアサーションを使うことでテストコードの実行を制御でき、配布されるソフトウェアではテストコードが実行されないようにすることができる。

しかし、アサーションを使った場合にもロギング API と同様に可読性の低下という問題がある。

2.4　条件付きコンパイルによるデバッグコードの挿入、削除

条件付きコンパイルとはコンパイル時のオプションによってプログラム断片の挿入、削除を行う技術である。前のセクションで述べたロギング API では、挿入するデバッグコードはログ出力命令、アサーションでは正誤判断をするコードのみであった。条件付コンパイルの場合はプログラマが必要に応じて、通常と同じプログラムコードを埋め込むことができる。この機能は C/C++などの言語で採用されているが Java では採用されていない。

デバッグコードを記述する際、この技術が使われる。デバッグの際に記述するデバッグコードは最終的にソフトウェアを配布する際に取り除く必要がある。そこで、条件付きコンパイルを利用してデバッグ付きオプションでコンパイルされた時にのみデバッグコードを挿入し、通常コンパイルの場合にはデバッグコードはコンパイルされないようにする。

以下に C 言語の条件付きコンパイルの例を示す。

C 言語の条件付きコンパイルの例

```
//server.c
#include <stdio.h>
void service()
{
```

```
  int amount;
  #ifdef DEBUG
    //デバッグコード
    printf("start service\n");
  #endif
  メイン処理
  #ifdef DEBUG
    //デバッグコード
    printf("end service\n");
    if(amount < 0)
      exit(1);
  #endif
}
```

このソースコードをコンパイルする際、

```
>gcc -DDEBUG server.c
```

と入力することでプリプロセッサによりデバッグコードの挿入が行われコンパイルされる。-DDEBUG を付けない場合にはデバッグコードは削除される。

条件付きコンパイルを利用することで配布されるソフトウェアに不要な処理が入ることがなく、ソースコード中に直接デバッグコードを記述した場合と異なりデバッグコードの取り忘れを防ぐことができる。

しかし、条件付きコンパイルの場合にも可読性の低下の問題がある。

2.5　デバッガ

デバッガとはプログラムを特殊な環境で実行するソフトウェアであり、プログラムの実行を特定の位置で中断させ、その時点での変数の値やメモリの使用状況を確認することができる。

デバッグの作業中、変数の値を確認する場合プリント文をソースコード中に挿入し、コンソール中に値を出力することで確認ができる。しかし、新たに他の変数を確認したい場合、その度に 1, プリント文の挿入、2, コンパイル、3, 実行、4, 値の確認、5, プリント文の削除という一連の作業を行うのは効率的ではない。また、プログラムの起動に多くの時間が掛かるアプリケーションの場合、何度もプログラムを再実行するには多くの時間がかかってしまう。

そこで、デバッガを使い実行中のプログラムを一時中断させ変数の値を確認する。デバッガを使ってプログラムを実行する際、あらかじめ実行を

中断させる位置をブレイクポイントで指定する。ブレイクポイントの設定はデバッガの使用中に変更することもできプログラムの再起動を行う必要がない。また、デバッガを使う際、プログラムのソースコードを変更する必要もない。

このようにデバッガは変数の確認やメモリの利用状況を把握するためには大変有用である。しかし、ブレイクポイントで必ず実行を中断させる必要があり大量の変数の確認を行う場合やプログラム全体からあるていどバグの位置を特定させるにはログ出力による確認の方が効率が良い。また、サーバサイドプログラミングではデバッガを利用できない場合がある。

2.5.1　jdb

jdb とは JDK(Java Development kid) に含まれる java プログラムのためのデバッガであり、コマンドラインにコマンドを入力しデバッグを行う。ブレイクポイントとしてメソッドが実行される直前やソースコードの行番号を指定することができる。

コマンド	機能	使用例
run	プログラムの実行	run Main
stop at	ソースコードの行番号にブレイクポイントを設定	stop at Point:162
stop in	メソッド名前を指定しブレイクポイントを設定	stop in Point.move
clear	ブレイクポイントの削除	clear Point:162
step	1 ステップ実行	
next	1 行実行	
cont	ブレイクポイントによる中断の再開	
print	値の出力	print point.x
dump	オブジェクトが持つフィールドの一覧を表示	dump point
locals	ローカル変数の一覧を表示	
exit	デバッガの終了	

表 2.1: jdb の主要コマンド一覧

以下に jdb の使用例のせる。

実行するプログラムのソースコード

```
1public class TestJDB {
2  public static void main(String[] args) {
3    Point point = new Point(0, 0);
4    int n = 100;
```

```
 5    int m = 200;
 6    point.move(n, m);
 7  }
 8}
 9class Point{
10  int x;
11  int y;
12  public Point(int x, int y){
13    this.x = x;
14    this.y = y;
15  }
16  public void move(int newX, int newY){
17    this.x = newX;
18    this.y = newY;
19  }
20}
```

jbd の実行例

```
>javac -g TestJDB.java #デバッグオプション付きでコンパイル
>jdb
>stop at TestJDB:5     #ブレイクポイントの設定
>stop in Point.move    #ブレイクポイントの設定
>run TestJDB           #メインメソッドの実行
  Breakpoint hit: "thread=main", TestJDB.main(), line=5 bci=13
  5               int m = 200;

main[1]print n         #変数 n の値の出力
  n = 100

main[1]dump point      #point オブジェクトのフィールド一覧の表示
  point = {
     x: 0
     y: 0
  }
main[1]cont            #中断の再開
>
  Breakpoint hit: "thread=main", Point.move(), line=17 bci=0
  17              this.x = newX;
```

```
main[1]print this.x     #this.x の値の表示
  this.x = 0

main[1]locals           #ローカル変数一覧の表示
  Method arguments:
  newX = 100
  newY = 200
  Local variables:

main[1]cont             #中断の再開
>
  The application exited
```

実行するプログラムには座標を表す Point クラスがある。Point クラスには int 型のフィールド x,y があり、move(int,int) メソッドによりフィールドの値を変更する。TestJDB のメインメソッドでは、この Point クラスを使っている。TestJDB のメインメソッドを実行する前にブレイクポイントとして TestJDB の 5 行目と Point クラスの move メソッドに設定し、それぞれのブレイクポイントでローカル変数やオブジェクトの情報を表示させている。

また、実行例ではプログラムのソースコードをオプション付きでコンパイルしている。これは、ソースコードをコンパイルして得られるバイトコード (.class ファイル) にローカル変数の変数名の情報を付加するために-g を付けてコンパイルしている。変数名の情報を付加しない場合にはデバッガの起動中、ローカル変数の値を確認することはできない。

2.5.2　統合開発環境支援のデバッガ

デバッガは、さまざまな統合開発環境に組み込まれている。この節では Eclipse に組み込まれている Java 用デバッガについて説明する。

Eclipse とは

Eclipse[3] とはフリーの統合環境 (IDE) である。統合開発環境とはソフトウェアを開発する上で必要なエディタ、コンパイラ、デバッガなどを同じインターフェイスで扱えるツールである。エディタを使って Java プログラムを開発する場合、まずエディタでプログラムのソースコードを書き、そのソースファイルを保存、コンソールに「>javac XXX.java」と入

力しコンパイル、「>java XXX」で実行などの作業をそれぞれ独立して行う。統合開発環境では、それら一連の作業を同じインターフェイスを通して行うことができる。

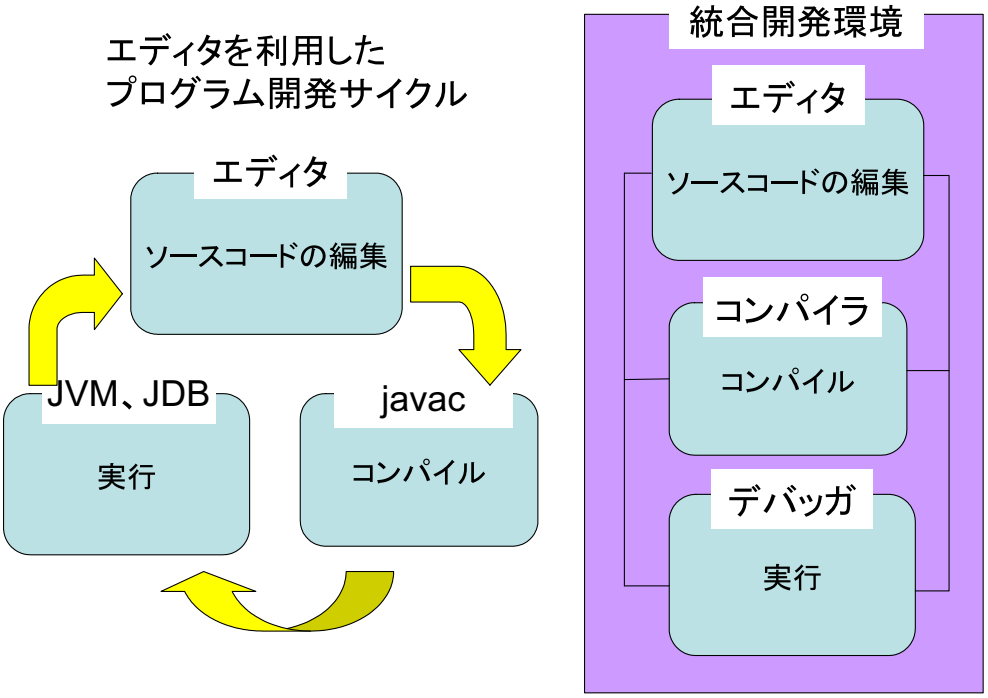

図 2.1: 統合開発環境の構成

Eclipse を扱う際インストールする Eclipse SDK には標準で Java 開発用の環境 JDT が組み込まれている。この JDT の機能を使って Java のプログラムを開発する場合、JDT に含まれる Java 用のエディタを使うことでソースファイルを保存した際にプログラムのコンパイル行う。コンパイルエラーが発見された場合、エディタ上に自動的にマークを付ける。また、GUI によりプログラムの実行やデバッグ作業を行うことができる。その他にも Eclipse SDK にはチームで開発を行う際に使われるバージョン管理ツールの CVS やビルドツールの Ant が標準で組み込まれており利用することができる。

このような機能を持った統合開発環境としては Eclipse の他に Borland 社の JBuilder やマイクロソフトの Visual C++などがある。これらの統合開発環境と Eclipse の大きな違いは以下のものである。

- オープンソース
  オープンソースでフリーであるため誰でも無料で使える。

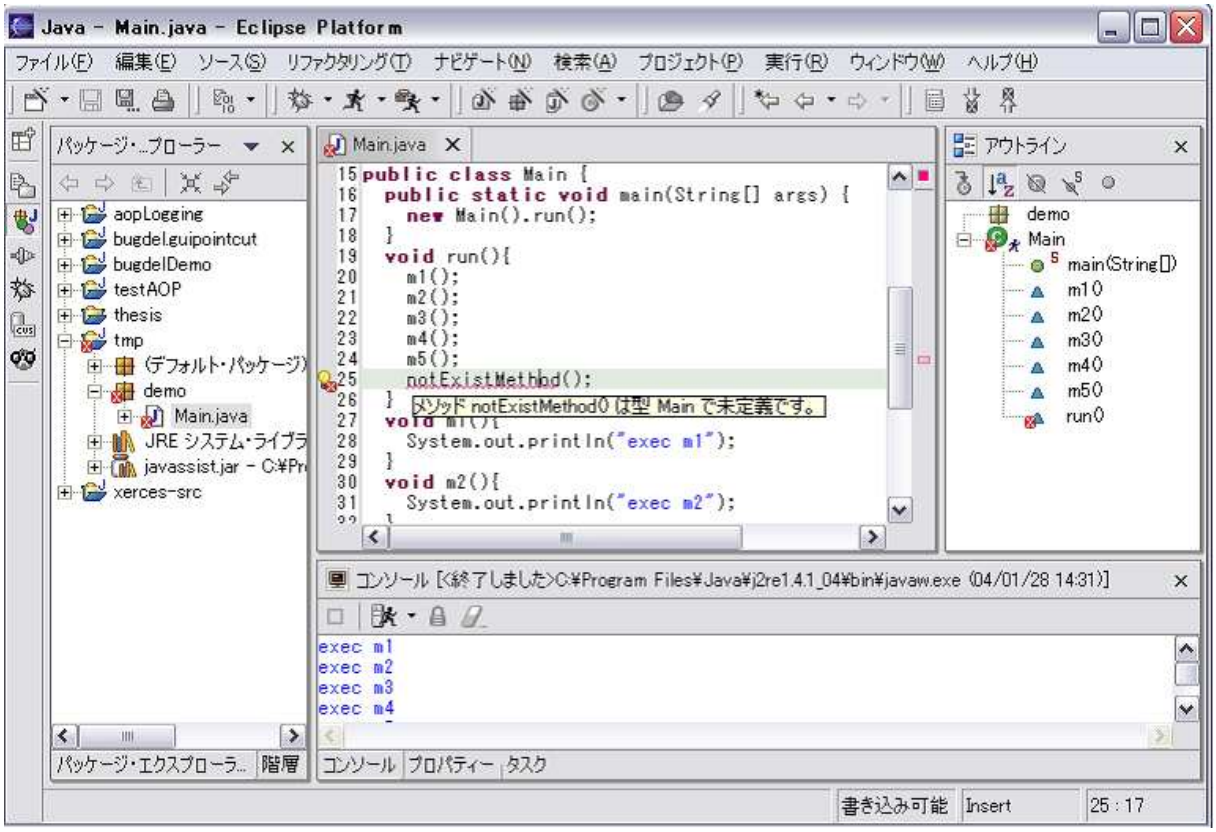

図 2.2: Eclipse の起動画面

- 拡張性
  プラグインを作成することで簡単に拡張が行える。そのため開発言語に中立であり Java 以外の開発環境を構築することもできる。

- 軽快な動作
  GUI を構成するツールとして SWT(Standard Widget Tools) を使っているため軽快に動作する。

Eclipse に組み込まれたデバッガ

jdb ではコマンドを入力しブレイクポイントの設定や変数の値の確認を行うが、Eclipse ではグラフィカルユーザインターフェイス (GUI) により、これらの作業を行える。例えば、行番号にブレイクポイントを設定する場合、JDB では「>stop at TestJDB:5」とコマンドを打つ必要があるが、Eclipse では組み込まれているエディタの行番号の左側をクリックするこ

とで行える (図 2.3)。さらに、ブレイクポイントで中断された位置での変数の一覧をツリー状に表示されマウス操作で値を確認でき難しいコマンドを入力する必要が無い。また、ブレイクポイントでは値の確認だけでなく、式を入力し評価することもできる。

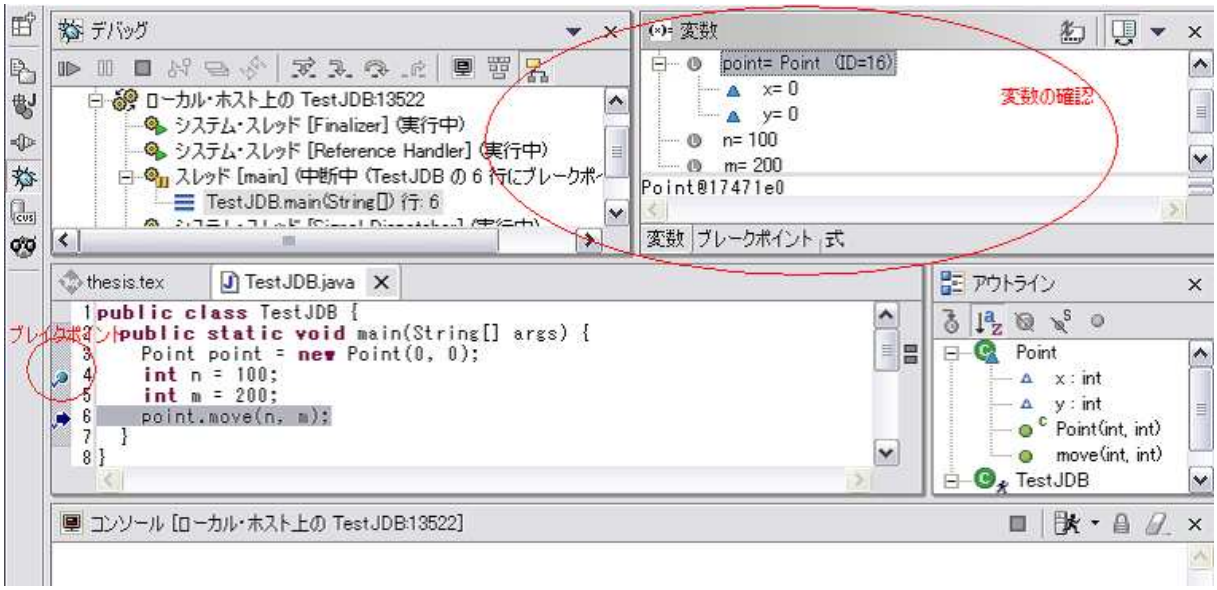

図 2.3: 統合開発環境に組み込まれたデバッガ

2.6　デバッグを支援ツールの問題点

この章ではいくつかデバッグを行うためのツールについて述べたがこれらのツールには共通して以下の問題点がある。

- デバッグコードがソースコード中に混入してしまいソースコードの可読性が悪くなる。

  ロギング API やアサーション機能、条件付コンパイルは開発したソフトウェアにログ出力などの不要な処理が実行されるのを防ぐことができるが、ソースコードに何らかのデバッグコードを埋め込む必要がある。そのため開発中のソースコードにデバッグコードが混入してしまい、ソースコードを公開しているソフトウェアやチームでソフトウェア開発を行っている場合には、それらのデバッグコードは取り除かなければならない。しかし、挿入したデバッグコードをソースコード中から探し出すのに負担が大きい場合がある。例え

ば、デバッグのためにロギング API を使ってログ出力コードをソースコード中に書いた場合、そのログ出力がソースコードのどの位置で実行されているのか判断するのは難しい。そのためデバッグコードはプログラムのソースコードとは分離されるべきであるが、上記で述べたデバッグ支援ツールでは分離することができない。

- デバッグコードを挿入する位置の指定を一ヶ所づつ指定しなければならない。

  例えば、変数 x が変更される度に変更された値をコンソールに出力させようとすると変数 x を変更している場所をソースコードから全て探し出し、ログ出力コードやブレイクポイントを挿入しなければならない。しかし、変数 x をさまざまな場所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る。また、同じものを全ての位置に挿入するのでは効率的ではない。

2.7　アスペクト指向プログラミングによるデバッグコードの挿入

2.7.1　アスペクト指向プログラミングとは

アスペクト指向プログラミング [12] とは関心事を分離するためのプログラミング技法であり、横断的な機能をモジュール化することができる。

ソフトウェアのを設計する場合、開発するシステムを小さな機能 (モジュール) へ分解し、それらを組み合わせることで開発する。プログラミング言語は、このモジュール分解の方法を提供している。現在プログラミング言語には C 言語、Fortran の手続き型、LISP、ML の関数型、prolog の論理型、Java、Smalltalk、C++のオブジェクト指向などがあり、これらの違いはモジュール分解の方法の違いである。近年、注目を浴びているオブジェクト指向では開発するシステムをオブジェクトという単位でモジュール分解され、各システムのデータや機能は各オブジェクトが担当する。そのため複数のオブジェクトに関係した処理、例えばデバッグ作業で行われるロギング処理やメモリ管理、同期処理などは一つのオブジェクトにモジュール化することができない。以下にオブジェクト指向言語である Java で書かれたロギング処理の例について説明する。

複数のクラスがあり各クラスに定義されている service メソッドが実行されたときにログを出力しようと考えた場合、次のようなプログラムになる。

```
class ServerA{
```

```
  void service(){
    Logger.log();//ログ出力用コード
    メイン処理
  }
}
class ServerB{
  void service(){
    Logger.log();//ログ出力用コード
    メイン処理
  }
}
class Logger{
  static log(){
    System.out.println("start service");
  }
}
```

ServerA、ServerB の二種類のクラスがあり、それぞれの service メソッドにログ出力用のコードを挿入している。この例では各クラスの service メソッドにログ出力用のコード (Logger.log();) が散らばってしまっている。また、新たに ServerC クラスを定義し service() というメソッドを作成した場合には作成したメソッドごとに毎回ログ出力コードを挿入する必要があり手間がかかる。これは「service メソッドが実行されたときに”start service”を出力する」という処理をモジュール化できないために起こる。

アスペクト指向プログラミングでは、このように散らばってしまう処理 (横断的関心事) をアスペクトとしてモジュール化することができる。前述の例をアスペクト指向プログラミングにより書き直すと、「service メソッドが実行されたとき”start service”を出力する」という処理をアスペクトとして記述することができ、ServerA、ServerB にはログ出力コードを記述する必要がなく、コンパイル時に自動的に埋め込まれる。

このようにアスペクト指向プログラミングでは、元のプログラムから関心のある場所を指定し、指定した位置に新たなコードを埋め込むことができる。元のプログラムにアスペクトによりコードを埋め込むことを weave すると言う。また、アスペクト指向は、既存のプログラミング技法、例えばオブジェクト指向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の弱点を補うものである。

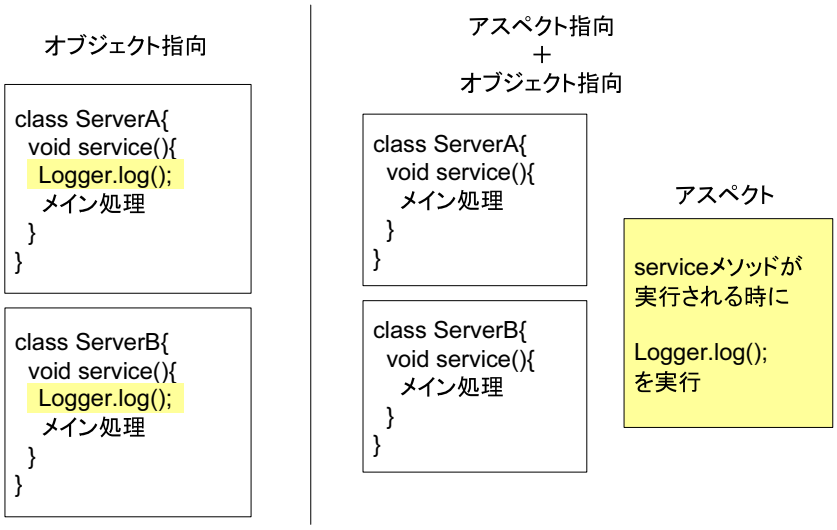

図 2.4: アスペクト指向

2.7.2　AspectJ

AspectJ[2, 11] とは Java を言語拡張したアスペクト指向言語である。言語拡張しているため使用するには専用のコンパイラ ajc が必要である。しかし、アスペクトによるコードの埋め込み (weave) はコンパイル時に行われ、コンパイルして生成されるバイトコードは標準のバイトコードである。そのため ajc によって作られたバイトコードは AspectJ の実行ライブラリ aspectjrt.jar をクラスパスに追加することで通常の JVM で実行することができる。

Java ではプログラミングを行う際、クラス又はインターフェイスを定義する。AspectJ では、これに加えアスペクトを定義することができる。アスペクトでは元のプログラム中から特定の位置を pointcut で指定し、指定した位置に新たなコードを埋め込むことができる。その際、元のプログラムのソースコードを変更する必用は無く、アスペクトによって定義された新たな処理はコンパイル時に挿入される。以下では AspectJ で重要な概念 joinpoint、pointcut、advice、aspect について説明する。

Joinpoint

結合点と訳される言葉でありメインのプログラムとアスペクトコードの結合点のことである。アスペクトによって挿入されるコードはプログラム中の任意の位置を指定できるわけではなく、この joinpoint と呼ばれる位

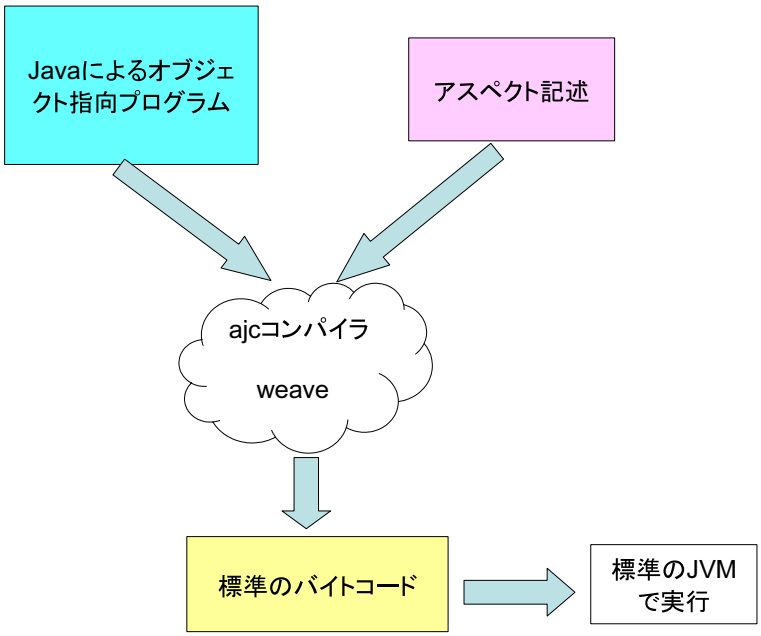

図 2.5: AspectJ の処理系

置で挿入を行う。主な joinpoint には、メソッド呼び出し時点、メソッド実行時点、コンストラクタ呼び出し時点、初期化子実行時点、コンストラクタ実行時点、静的初期化子実行時点、フィールド参照時点、フィールド代入時点、ハンドラ実行時点などがある。アスペクトによって挿入されるコードはこれらの joinpoint の位置に埋め込まれる。

```
class A{
  int n;
  void main(){
    //フィールド代入 (joinpoint)
    n = 0;
    //コンストラクタ呼び出し (joinpoint)
    String s = new String();
    if(n > 0){
      //メソッド呼び出し (joinpoint)
      foo();
    }
  }
  void foo(){//メソッド実行 (joinpoint)
    ...
```

```
  }
}
```

Pointcut

pointcut とは、いくつかの joinpoint を指定するものである。プログラム中には、メソッドコールやコンストラクタ実行、フィールド参照などさまざまな joinpoint が存在している。プログラマはそれらの中から関心のある (コードを挿入したい位置の)joinpoint を pointcut として指定する (図 2.6)。そして次節以降で説明する advice を定義することで、pointcut で指定した joinpoint の位置に新たな処理を埋め込むことができる。

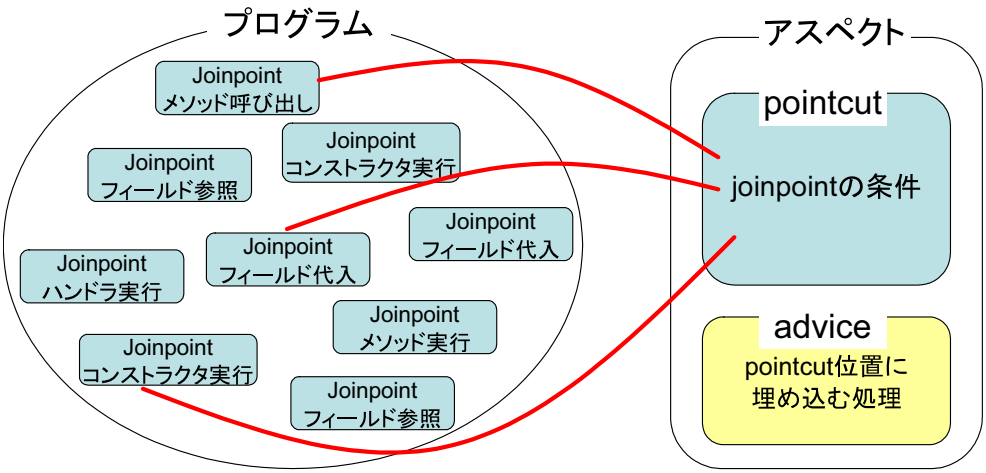

図 2.6: pointcut の概要

joinpoint を指定するための pointcut にはプリミティブなものとユーザ定義のものがある。以下にプリミティブな pointcut の使用例を示す。

- call($MethodPattern$)、call($ConstructorPattern$)

  説明:メソッド、コンストラクタの呼び出し時点
  使用例:call(public void java.awt.Point.move(int,int))

- execution($MethodPattern$)、execution($ConstructorPattern$)

  説明:メソッド、コンストラクタの実行時点
  使用例:execution(public void java.lang.String.substring(int))

- get($FieldPattern$)

説明:フィールドの参照時点
使用例:get(public int Point.x)

- set(*FieldPattern*)

  説明:フィールドへの代入時点
  使用例:set(public int Point.x)

- initialization(*ConstructorPattern*)

  説明:オブジェクトの生成時点
  使用例:set(java.lang.Object.new(int,int))

- handler(*TypePattern*)

  説明:*TypePattern* で表される例外の Exception ハンドラ (catch 文) 実行時点
  使用例:handler(java.lang.RuntimeException)

- within(*TypePattern*)

  説明:*TypePattern* で表されるタイプの中に現れる全ての joinpoint
  使用例:handler(java.lang.RuntimeException)

- args(*Type*, ...)

  説明:*Type*, ... で表される引数を持つ全ての joinpoint
  使用例:args(int,String)

- cflow(*PointcutPatern*)

  説明:*PointcutPatern* で表される joinpoint の中で呼び出される全ての joinpoint。*PointcutPatern* で指定されている joinpoint も含む (cflowbelow の場合は含まない)。

  使用例:cflow(execution(void Point.move(int,int)))
  上の pointcut を指定すると、move メソッドの中で setLocation メソッドを呼び出していて、さらに setLocation メソッドの中でフィールド変数 x,y に値を代入していた場合、それらすべての joinpoint が対象になる (図 2.7)。

  cflow はとても有効な pointcut であるが、cflow によって指定される joinpoint はコンパイル時に決定するこができない。そこで cflow を使った場合プログラム全ての joinpoint にフックが挿入され、そのフックの中で cflow の条件を判定する。そのためプログラムの実行時間が著しく低下するという問題もある。

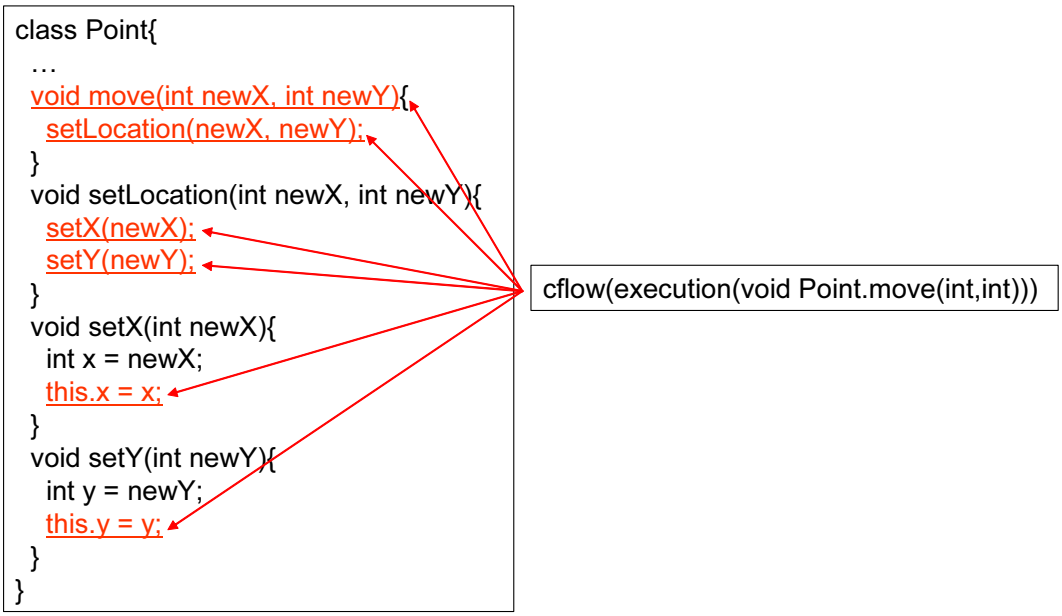

図 2.7: cflow を使った pointcut

この他にも staticinitialization、whithincode、target、if などの pointcut がある。

また、メソッドを表す *MethodPattern*、タイプを表す *TypePattern* などを記述する際に、任意の文字を表す”*”や任意の引数を表す”..”を使うことができる。

1. set(public *)
   public である任意のフィールド変数に代入する時点
2. call(* *.move(..))
   全てのクラスの move メソッドの呼び出し時点
3. handler(java.lang.*Exception)
   java.lang パッケージに含まれ、クラスの名前が Exception で終わる例外のハンドラが実行される時点

また、定義されている pointcut を and”&&”、or”‖ ”、not”!” を使って組み合わせることもできる。

1. within(Main) && call(* Point.move(..))
   Main クラスの中で Point クラスの move メソッドを呼び出している時点
2. set(* Point.x) ‖ set(* Rectangle.y)

Point クラスのフィールド x 又は Rectangle クラスのフィールド y に値を代入している時点

3. ! call(* *(..))
   全てのコンストラクタ、メソッドの呼び出し以外の joinpoint
4. within(Rectangle)&&(set(* Point.x) ‖ set(* Point.y)) Rectangle クラスの中で Point クラスのフィールド x 又は y に値を代入している時点

上記ではプリミティブな pointcut の使い方を述べた。これらの pointcut を組み合わせてプログラマが新たな pointcut を定義することもできる。

```
pointcut move(): call(* Point.setX())
              || call(* Point.setY())
              || call(* Point.move(..));
pointcut moveInRectangle() : move() && within(Rectangle);
```

Advice

advice とは pointcut で指定した joinpoint 位置で実行する振る舞いを記述するものである。advice には joinpoint で埋め込まれる処理の実行のタイミングにより before、after、around の 3 種類に分けられる。

- before
  joinpoint 位置の直前
  下の例では Point クラスの move メソッドが呼び出される直前に「System.out.println(”before call move”);」が実行される。

  ```
  before() : call(* Point.move(int,int)){
    System.out.println("before call move");
  }
  ```

- after
  joinpoint 位置の直後
  下の例では Point クラスのフィールド x に値を代入した直後に「System.out.println(”change Point.x”);」が実行される。

  ```
  after() : set(* Point.x){
    System.out.println("change Point.x");
  }
  ```

- around
  joinpoint で定義されている処理の置き換え
  下の例では Point クラスの move メソッドで実行される処理を別の処理で置き換えている。置換後の処理の中で使われている proceed() は元の処理 (ここでは move メソッドの処理) に相当する。

```
void around() : execution(* Point.move(int,int)){
  System.out.println("around before");
  proceed();
  System.out.println("around after");
}
```

Aspect

aspect は横断的関心事をモジュール化するものである。これまで pointcut、advice について述べたが、これらは aspect の中で扱われる。

aspect の例

```
public aspect LogAspect{
  //pointcut
  pointcut logging() : execution(* *.service());
  //advice
  before() : logging(){
    System.out.println("start service);
  }
}
```

このように aspect は Java の class と同じように定義される。そして、aspect を定義したファイルと class を定義したファイルを ajc でコンパイルすることで advice で記述されている処理がプログラムコード中に埋め込まれる。

このように AspectJ を使うことでデバッグコードとソースコードを別々に記述することができ、コード挿入位置をまとめて指定することができる。

2.7.3　AJDT(AspectJ Development Tools)

AJDT[1] とは Eclipse 上に AspectJ のプログラミング環境を提供するツールである。

AspectJ ではアスペクトによって挿入されるコードは、対象となる joinpoint の位置にプログラムコードを書かなくても自動的に埋め込まれる。このことは、アスペクト指向の利点の一つであるが、同時に欠点も含んで

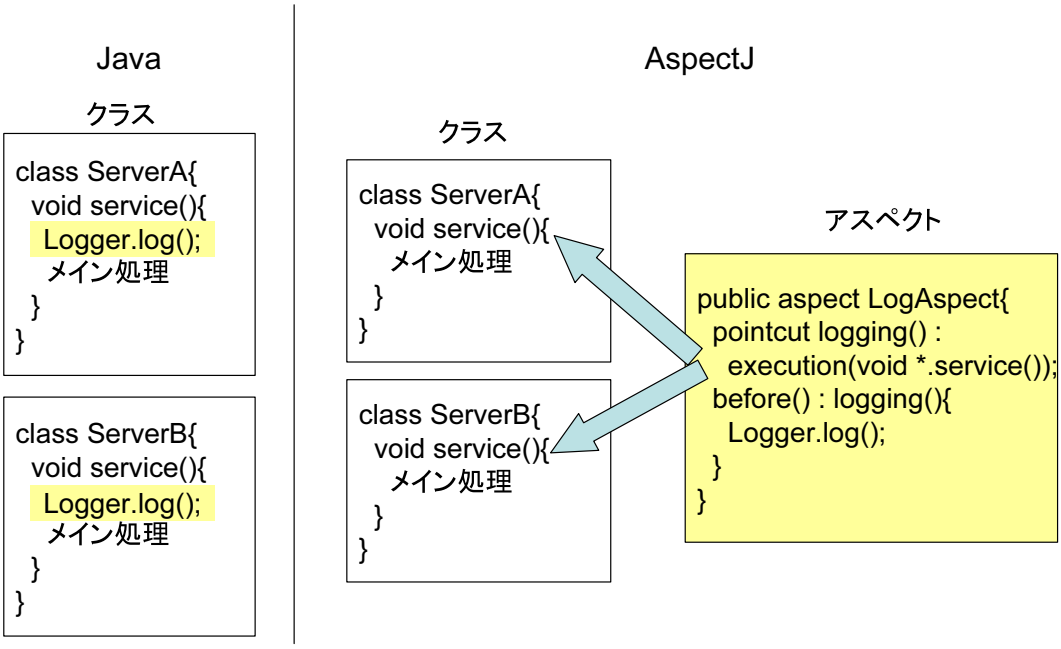

図 2.8: AspectJ によるロギング処理の例

いる。なぜならプログラマが予期してない位置にアスペクトによるコードが埋め込まれる可能性がある。また、ソースコードだけを見てもアスペクトによるコードの挿入がどの位置で行われているのか判断するのが難しい。

このようにアスペクト指向言語を使用することでアスペクトの干渉による複雑さが存在してしまう。そのため統合環境による支援が必要になる。AJDT は Eclipse 上に AspectJ プログラミング支援環境を構築し、アスペクトによってコードが挿入される joinpoint の位置にマークをつけエディタ上に表示する (図 2.9)。また、joinpoint の位置に影響を与える advice の一覧を表示することができる。

AJDT には、この他にも AspectJ 専用のエディタによるコード補完機能やアスペクトがプログラム全体どの位置に埋め込まれているのかを表示する Aspect Visualizer などが入っている (図 2.9)。

このように AJDT によりアスペクト指向プログラミング (AspectJ) による開発の支援を行うことができ、アスペクトの干渉により複雑になってしまうソースコードの解析を支援することができる。

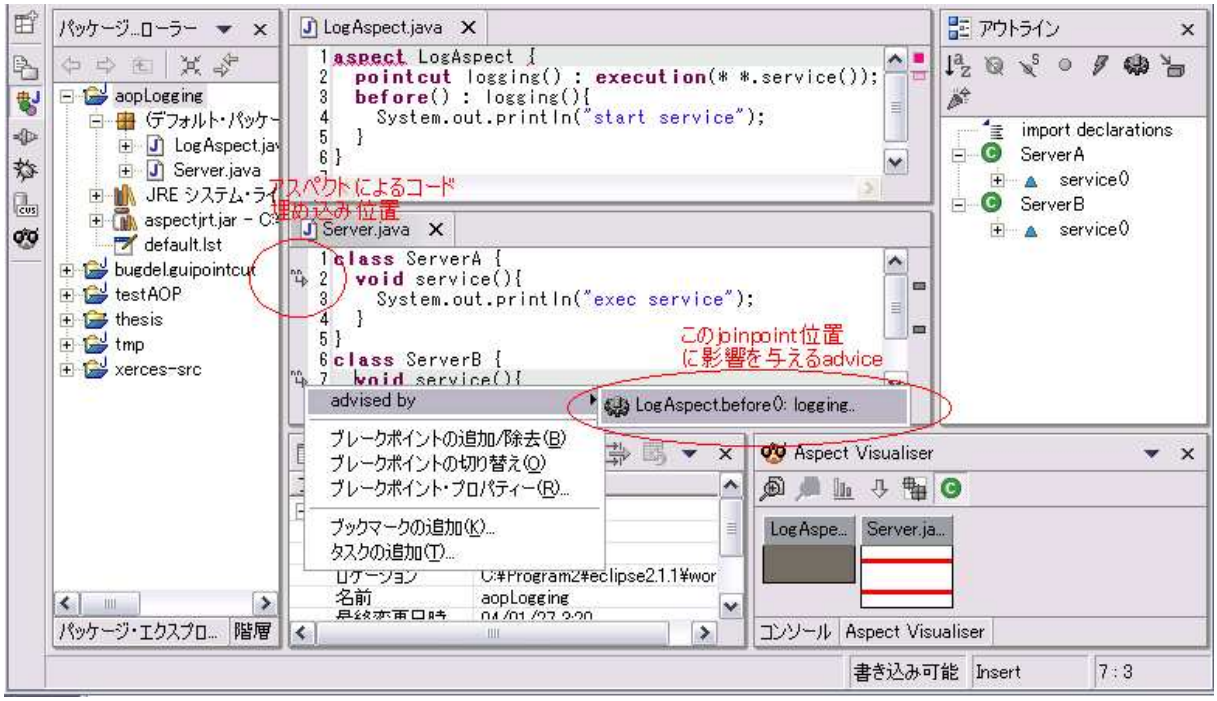

図 2.9: AJDT の利用画面

2.7.4　AspectJ を利用したデバッグコード挿入の問題点

章 2.6 で示した問題はアスペクト指向プログラミングにより解決することができると考えられる。

Java プログラミングにアスペクト指向プログラミングの利用環境を提供する既存の技術には AJDT がある。しかし、AJDT で使用する AspectJ はデバッグ専用ではなく汎用的な言語であるためデバッグ作業には不向きな点がある。

まず、デバッグ作業ではソースコード中の特定の行にデバッグコードを挿入したいと考える場合がある。しかし、AspectJ では行単位の pointcut を提供していない。なぜなら行単位の pointcut は weave 先のソースコードの実装に依存してしまいアスペクトのモジュール性やコード挿入によるプログラムの整合性を損ねてしまうからである。同様の理由で AspectJ には if 文の pointcut や while 文の pointcut、ローカル変数の pointcut が無い。そのためデバッグ作業に有効な行単位の pointcut が指定できない。また、挿入するコード内では挿入先のローカル変数や private 変数にアクセスすることができない。次に、デバッグコードは開発中のプログラムと異なりデバッグが終わると必用のないものである。デバッグコード挿入位

置を指定する pointcut は AspectJ の文法に基づいて記述する必要があり、デバッグのために pointcut の記述をおこなうのには負担が大きい。

このように AspectJ は汎用的なプログラミング言語であるためデバッグ作業に用いるには不向きな点がある。この問題は AJDT のような AspectJ プログラミング支援環境ツールを使っても解決することができない。

第3章　Bugdelの設計と実装

前章で述べたようにデバッグコードを埋め込む際、アスペクト指向を利用することで効率よく挿入ができると考えられる。しかし、既存のアスペクト指向プログラミングの利用環境である AspectJ は汎用的な言語でありデバッグには不向きな点がある。そこで本研究では Java のプログラムを対象にデバッグに特化したアスペクト指向プログラミングの利用環境を実現する Bugdel を提案する。本システム Bugdel はデバッグコードの挿入をアスペクト指向を利用して行うことができる。また、デバッグに特化したアスペクト指向プログラミングの環境を構築するのが目的であるため AspectJ には無かった行単位の pointcut を実現した。また、挿入するコード内で挿入先のローカル変数や private 変数にアクセスすることもできる。さらに Bugdel では pointcut 指定を GUI(グラフィカル・ユーザ・インターフェイス) で行えるため複雑な pointcut 記述を行う必要がない。

Bugdel の開発には統合開発環境である Eclipse 上にプラグインとして実現した。Eclipse にはエディタなどの基本的な機能の他に JDT によって Java コードを操作する API が標準で提供されている。また、Eclipse 本体にプラグイン開発の機能があるため用意に機能の拡張ができる。

この章では本システム Bugdel の使用方法と実装方法について述べる。

3.1　仕様

本システムは主に Bugdel 専用のエディタ (BugdelEditor) とビュー (Bugdel view) から成る (図 3.1)。BugdelEditor は pointcut の対象になっている位置 (デバッグコードが挿入される位置) にマークを付けたり GUI により pointcut を指定するために使われる。また、このエディタは JDT に含まれる Java エディタの機能を継承しているため Java のコード補完機能などを, そのまま利用することができる。Bugdel view は指定されている pointcut の情報を表示するものである。

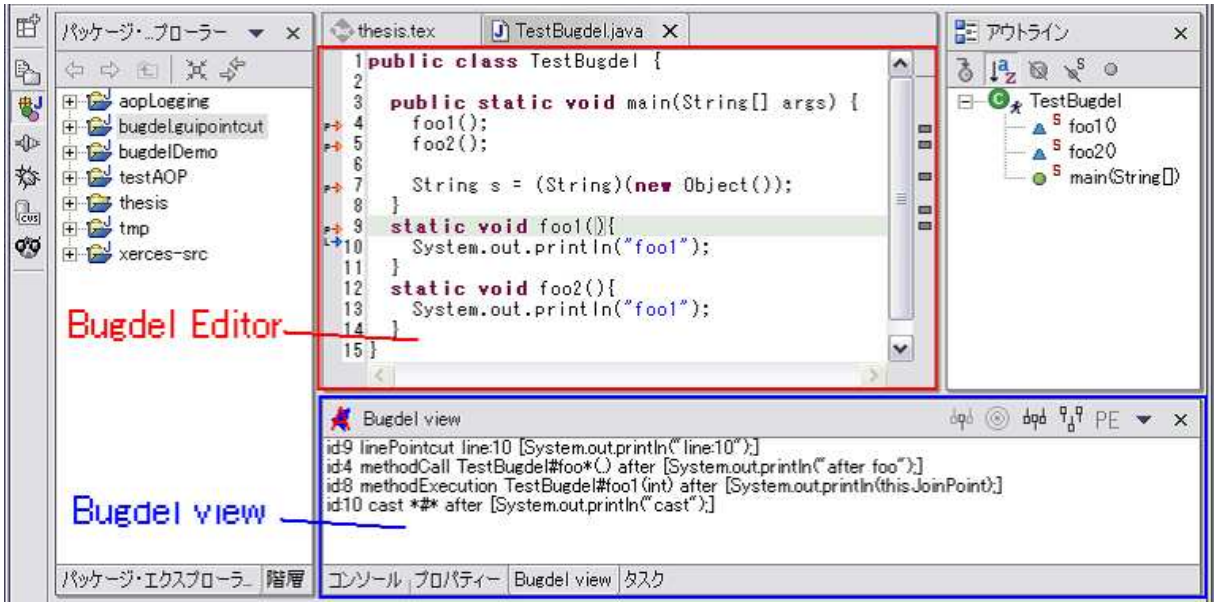

図 3.1: Bugdel 専用エディタ、ビュー

3.1.1　デバッグコードを挿入する位置の指定 (pointcut の指定)

Bugdel 専用のエディタを使用することでデバッグコードを挿入する位置の指定 (pointcut の指定) を GUI によって行える。指定できる pointcut には AspectJ と同様のメソッドコール、フィールドアクセスなどのクラスメンバに関する pointcut の他に行単位の pointcut が指定できる。

クラスメンバに関する pointcut

クラスメンバに関する pointcut では次のものが利用可能である。(表 3.1)

これらの pointcut を指定する場合にはエディタ上に表示されているソースコードから pointcut したいメンバ (クラス名、メソッド名、フィールド名) にカーソルを合わせ右クリックしポップアップメニューの「pointcut」項目を選択する (図 3.2)。その後、カーソル位置にあるメンバに関係のある pointcut の候補の一覧が表示され目的の pointcut にチェックを入れ「OK」ボタンを押す。

ユーザがクラス名とメソッド、フィールド名を直接入力し pointcut を指定することもできる。この方法で指定するにはメニューバーの一番右に表示される「bugdel」をクリックし表示される項目一覧の中から「pointcut」を選択すとダイアログが表示されるので、その中にクラス名などを記述する (図 3.3)。クラス名などを記述する際、AspectJ と同様に任意の文字列

pointcut	説明
fieldGet	フィールド参照
fieldSet	フィールド代入
methodCall	メソッド呼び出し
methodExecution	メソッド実行
constructorCall	コンストラクタ呼び出し
constructorExecution	コンストラクタ実行
instanceof	instanceof 演算の実行
cast	キャストの実行
handler	例外ハンドラの実行

表 3.1: bugdel で利用可能なクラスメンバに関する pointcut

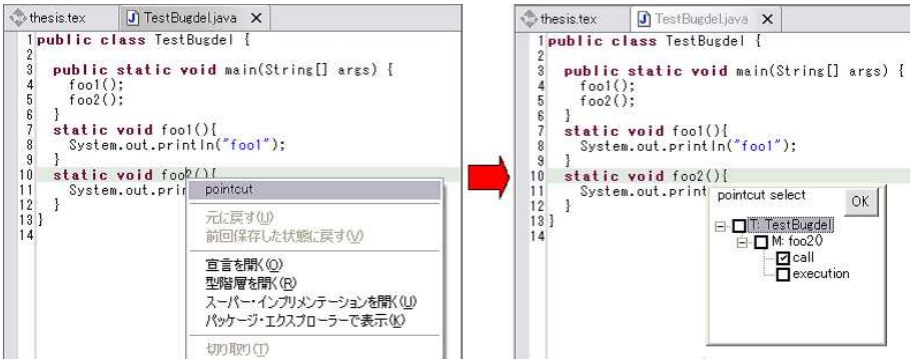

図 3.2: クラスメンバに関する pointcut1

を表す*”や任意の引数を表す”..” を使用することができる。

以上のように指定した pointcut は Bugdel 専用のビュー (Bugdel view) に表示され、クラス名などを後から編集することもできる。

また、pointcut を指定してデバッグコードを挿入する位置にはエディタ上にアイコンが表示されデバッグコードがどの位置で挿入されるのか把握することができる。このアイコンは新たな pointcut を指定したときや、編集中のソースコードを保存したときに更新される。

行単位の pointcut

行単位の pointcut はエディタの左側にあるルーラー上でポップアップメニューを表示させ「line pointcut」を選択することで行える (図 3.4)。指定した pointcut の情報はクラスメンバと同様に Bugdel ビューに表示さ

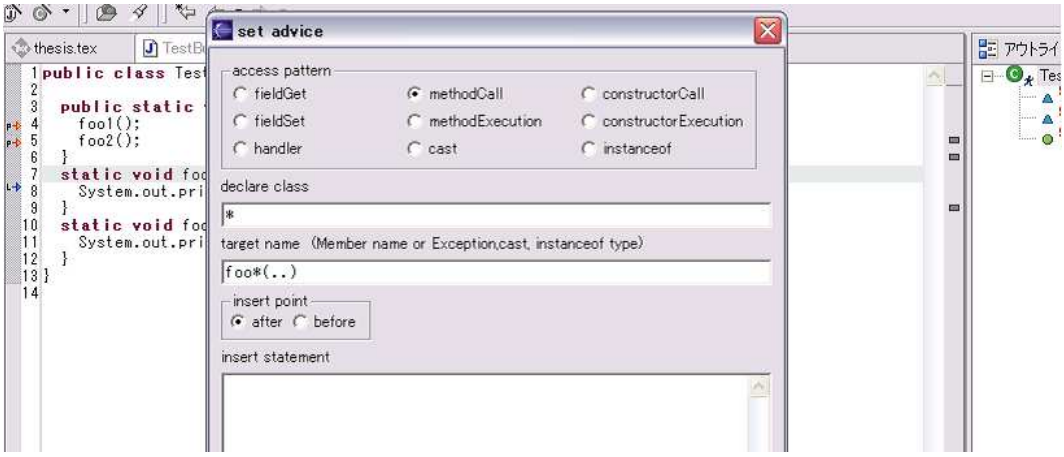

図 3.3: クラスメンバに関する pointcut2

れる。

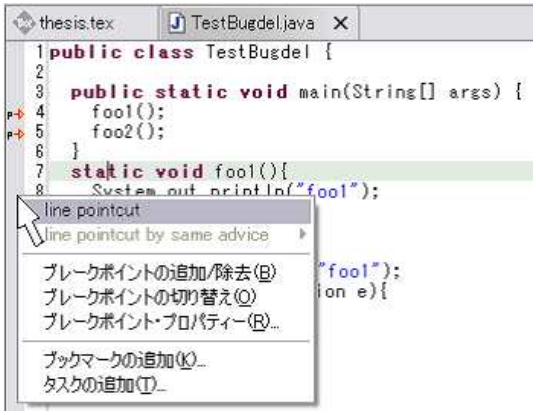

図 3.4: 行単位の pointcut

3.1.2　挿入するデバッグコードの入力

Bugdel view 上で pointcut を選び右クリックでメニューを表示され「edit」を選択 (図 3.5) するとダイアログが表示されデバッグコードを記述することができる。複数の pointcut を選択し同時にデバッグコード記述することもできる。また、メンバに関する pointcut ではデバッグコードを指定した joinpoint の前後どちらに埋め込むのかを指定することができる。

挿入するデバッグコードの中では次の特殊文字が使える。

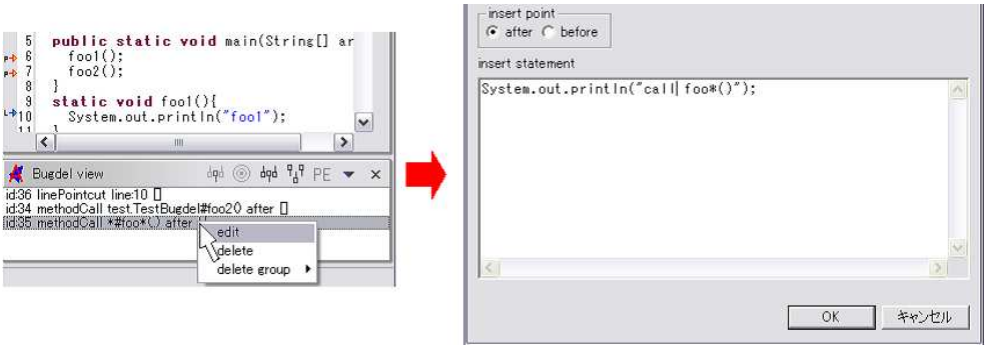

図 3.5: デバッグコードの入力

- thisJoinPoint – String 型

  joinpoint の情報 (ソースコードの行番号など) を表す。

- thisJoinPoint.line – int 型

  joinpoint の行番号を表す。

- thisJoinPoint.file – String 型

  joinpoint が存在するソースコードのファイル名を表す。

- thisJoinPoint.kind – String 型

  joinpoint の種類を表す。

- thisJoinPoint.field

  fieldSet,fieldGet の場合フィールドの値を表す。その他は null を表す。

- thisJoinPoint.target

  methodCall、fieldSet、fieldGet の場合ターゲットとなるオブジェクトを表す。その他は null を表す。

また、各 pointcut の種類に応じて以下の特殊な値を使うことができる。これらの特殊文字は以降で説明する Javassist[15] によって提供されているものである。

- fieldSet

  $0 アクセスした変数を含んでいるオブジェクトを表す。ただし static 変数の場合は null を表す。

  $1 アクセスした変数に代入する値を表す。

- fieldGet

  $0 アクセスした変数を含んでいるオブジェクトを表す。ただし static 変数の場合は null を表す。

  $_ アクセスした変数の値を表す。

- methodCall

  $0 呼び出したメソッドのターゲットとなるオブジェクトを表す。ただし static メソッドの場合は null を表す。

  $1,$2,... メソッドの引数を表す。

- constructorCall

  $_ 生成したオブジェクトを表す。

  $1,$2,... コンストラクタの引数を表す。

- instanceof

  $1 instanceof 演算子の左側のオブジェクトを表す。

  $r instanceof 演算子の右側のクラスの型を表す。

  $_ 演算の結果 (boolean) を表す。

- cast

  $1 キャストするオブジェクトを表す。

  $r キャストするクラスの型を表す。

- handler

  $1 catch 文でキャッチする例外オブジェクトを表す。

- methodExecution

  $1,$2,... メソッドの引数を表す。

- constructorExecution

  $1,$2,... コンストラクタの引数を表す。

3.1.3　デバッグコードの埋め込み (weave)

デバッグコードの埋め込みは weave 用のアクションが選択されたときに行われ、コードが埋め込まれたクラスファイルが生成され上書きされる。weave アクションはメニューバーの右端に表示される「bugdel」を選択するか専用のビュー (Bugdel view) の右上に表示されるボタンを押すことで実行される (図 3.6)。

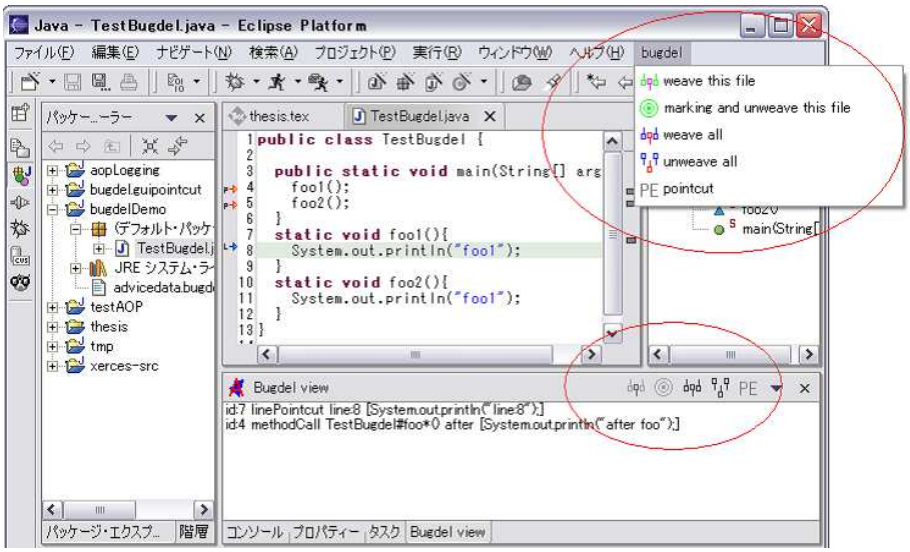

図 3.6: weave アクション

weave アクションには次の 4 種類がある。

- weave this file

  エディタで開いているソースコードをコンパイルし生成されたクラスファイルにデバッグコードを埋め込む。

- marking and unweave this file

  エディタで開いているソースコードのクラスファイルからデバッグコードを削除し、pointcut の対象となる位置に表示されるマーカーを更新する。

- weave all

  プロジェクト内すべてのファイルをコンパイルしクラスファイルにデバッグコードを埋め込む。

- unweave all

プロジェクト内すべてのクラスファイルからデバッグコードを削除する。

weave を行う際、無効なデバッグコードがある場合、エラーの情報が書かれたエラーダイアログが表示される。

3.2　実装

3.2.1　Eclipse プラットフォーム

Eclipse のアーキテクチャ

Eclipse プラットフォーム [14, 4, 8] は、プログラム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り、それ本体だけではエンドユーザ向けの機能をほとんど提供していない。つまり、統合開発環境を構築するためのフレームワークである。エンドユーザ向けの機能はプラグインを組み込むことで提供される。

Eclipse はプラグインベースのアーキテクチャでありプラグインにより機能を提供する。プラグインによって提供された機能は他のプラグインが使うこともできる。プラットフォームはユーザインターフェイスツール、リソース管理、ヘルプ開発、チーム開発などの機能をプラグインによって提供している。しかし、プラットフォームで提供されている機能は、ほとんどエンドユーザ向けの機能ではなく、他のプラグインが利用するための機能である。エンドユーザ向けの機能はプラットフォームに対してプラグインを提供することで実現される。Eclipse を扱う際インストールする Eclipse SDK には Java プログラム開発用のプラグイン JDT(Java Development Tools)[6] やプラグイン開発用のプラグイン PDE(Plugin Development Enviroment) が含まれており、これらのプラグインがエンドユーザ向けの機能を提供している (図 3.7)。

また、Eclipse に新たなプラグインをインストールすることで、さまざまなエンドユーザ向けの機能を提供することができる。主なプラグインとして、C/C++プログラム開発環境を提供する CDT、AspectJ プログラム開発環境を提供する AJDT、その他 Perl、Ruby、C#、PHP、XML プログラミングを支援するプラグインなどが作られている。さらに、Eclipse のプラグインにはプログラム開発支援を行うだけでなく、tomcat[1] を制御する tomcat プラグインやデータベースを編集するプラグイン、UML からプログラムのソースコードを生成する UML プラグインなどがある。本システムの Bugdel もプラグインとして作成されており Eclipse に組み込みこむことで機能を利用することができる。

[1] Java で作成されるサーバーサイドアプリケーション (servlet) を実行する環境

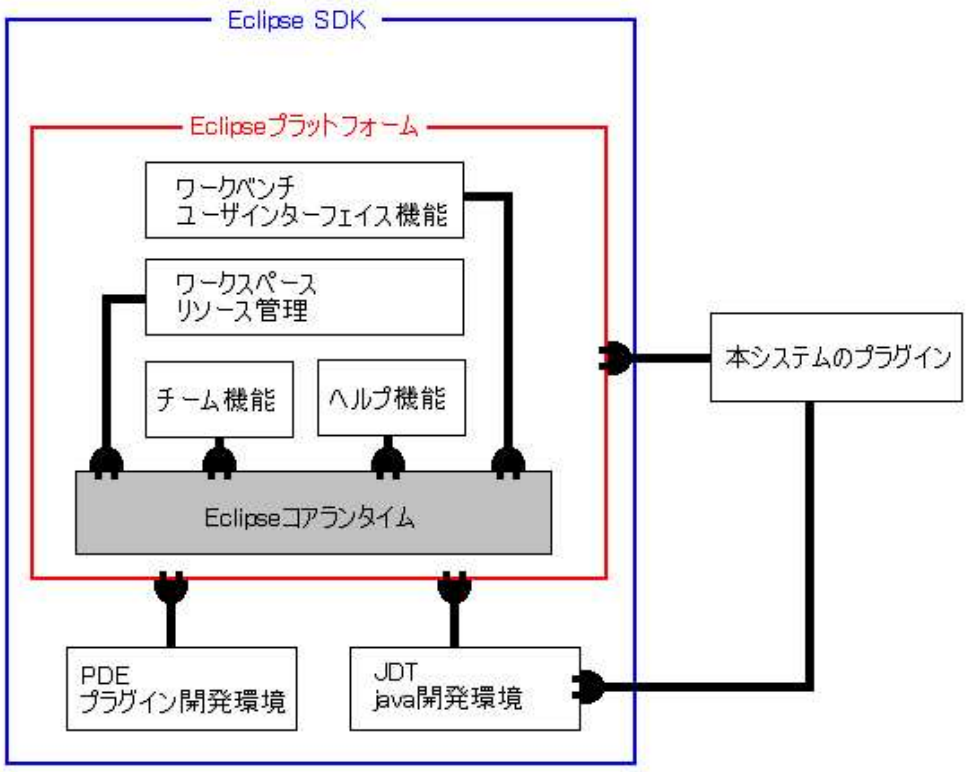

図 3.7: Eclipse SDK の構成

ワークベンチ

ワークベンチとはデスクトップ開発環境を提供するもので、リソースの管理、操作を行う際、共通のモデルを扱うことで優れた開発環境を提供する。Eclipse のワークベンチのユーザインターフェイスは主にエディタとビューから作られている。ビューは、エディタをサポートしワークベンチ内の情報を表示するものである (図 3.8)。主なビューは以下のものがある。

- ナビゲータビュー
  ワークベンチ内のリソースの階層図を表示する。
- アウトラインビュー
  エディタで開いているリソースのアウトラインを表示する。エディタの種類によって表示される内容は異なる。例えば、Java のソースプログラムを Java 専用のエディタで開いた場合には、メソッドやフィールドの一覧が表示される。テキストエディタで開いた場合には何も表示されない。

- プロパティビュー
  ファイルに関連するプロパティを表示する。テキストエディタを開いている場合、ファイルのサイズ、名前、更新日などが表示される。

- コンソールビュー
  プログラム開発の時の標準出力やエラー出力を表示する。

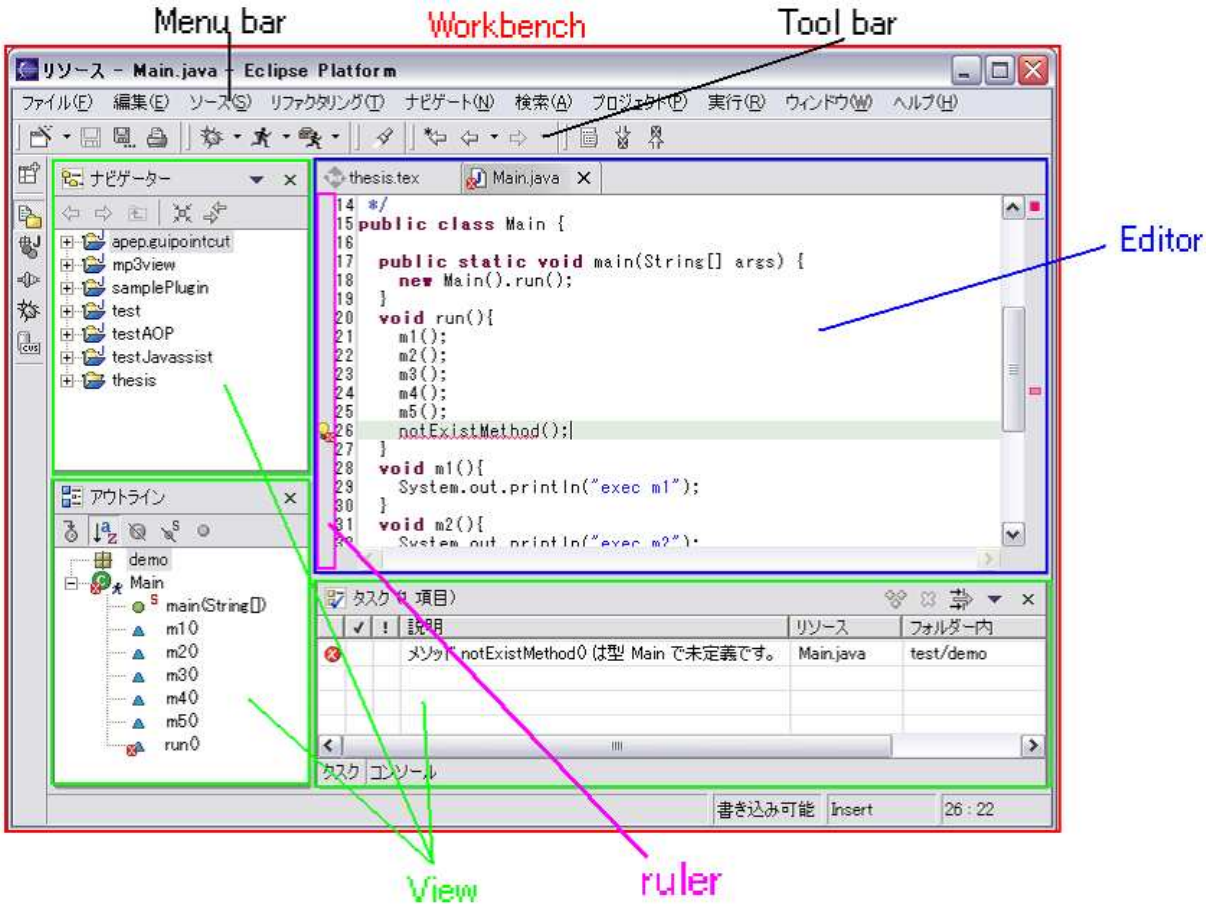

図 3.8: Eclipse ワークベンチ

Eclipse の拡張方法

Eclipse では、プラグインを作成することで新たな機能を提供する。プラグインは主に 2 つのファイル、拡張する場所 (拡張ポイント ID) を記述する plugin.xml ファイルと機能を実装した Java プログラムのバイトコードをアーカイブした jar ファイルから成る。

新たなエディタやメニューバーの項目の追加などを行う際、拡張ポイントの ID を指定することで拡張を行う。Eclipse プラットフォームのワークベンチプラグインは UI に関する拡張ポイントを、ワークスペースプラグインはリソース操作に関するさまざま拡張ポイントを定義している (表 3.2)。

拡張ポイントの ID	説明
org.eclipse.core.resources.builders	ビルド実行時の動作を定義
org.eclipse.core.resources.markers	リソースマーカーを定義
org.eclipse.ui.views	新しいビューを追加
org.eclipse.ui.editors	新しいエディタを追加
org.eclipse.ui.preferencePages	設定ダイアログ・ボックスにページを追加
org.eclipse.ui.perspectives	新しいパースペクティブを追加
org.eclipse.ui.editorActions	エディタのメニューおよびツールバーに　アクションを追加
org.eclipse.ui.viewActions	プルダウン・メニュー、およびツールバーにアクションを追加
org.eclipse.ui.actionSets	メニュー項目およびツールバー・ボタンを追加
org.eclipse.ui.popupMenus	コンテキスト・メニューに新しいアクションを追加
org.eclipse.ui.markerImageProviders	リソースマーカーに対する画像を提供

表 3.2: Eclipse プラットフォームの拡張ポイント

また、プログラマが独自のプラグインを作成し拡張ポイントを定義することもできる。Java 開発環境を提供する JDT には Java エディタにテキスト吹き出しを表示させるための拡張ポイントなどが提供されている。

拡張ポイントの情報は plugin.xml ファイルに XML 形式で記述され、各拡張ポイントに対応する Java のインターフェイスを実装することで Eclipse を拡張できる。以下に本システム bugdel の plugin.xml ファイルの一部を説明する。

拡張ポイントの指定例 (ビューの追加)

`plugin.xml` ファイルの一部

```
<extension point="org.eclipse.ui.views">
  <view
    name="Bugdel view"
```

```
    class="bugdel.guipointcut.view.bugdelView"
    id="bugdel.guipointcut.view.bugdelView">
  </view>
</extension>
```

この例はワークベンチにビューを追加するために拡張ポイント org.eclipse.ui.views を指定している。ビューの具体的な機能の実装は bugdel.guipointcut.view.bugdelView クラスが行う。また、BugdelView クラスは IViewPart インターフェイスを implements して機能を実装する。

Eclipse SDK にはプラグイン開発のための環境を提供する PDE が入っているため plugin.xml ファイルの編集や拡張ポイント ID の指定、拡張ポイントに対応するインターフェイスの選択を簡単に行うことができる (図 3.9)。

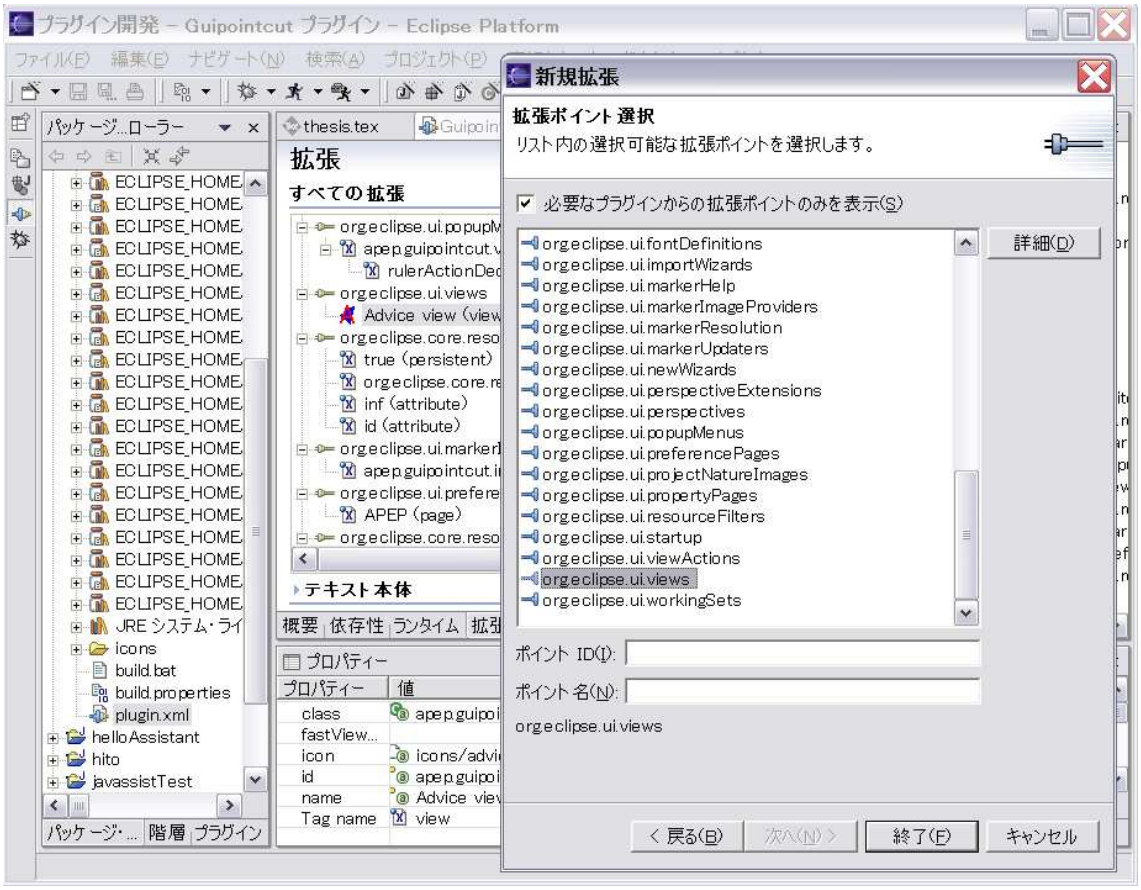

図 3.9: PDE によるプラグイン開発

SWT(Standard Widget Tools)

SWT(Standard Widget Tools)[7] とは IBM によって Eclipse 用に開発された GUI ツールである。

Eclipse プラットフォームやプラグインは Java で作られており、Java の GUI ツールとしては J2SE に標準で含まれている AWT(Abstract Window Toolkit) や Swing があるが、それぞれ問題があるため Eclipse では SWT を使っている。AWT では描画操作を OS またはウィンドウシステムが行っており動作速度は速いが、異なる OS やウィンドウシステムで描画位置ずれるなど機能上の制約がある。Swing ではウィンドウシステムがネイティブで行っていた描画操作を Java が行っているため異なるシステムで描画位置がずれるなどの問題がなく機能も豊富である。また、異なる OS でも見た目 (ルック&フィール) を統一することができ、パラメタにより変更することも可能である。通常 Java で GUI アプリケーションを作成する場合 Swing を使うことが多い。しかし、Swing ではネイティブで行っていた描画操作を Java が行なっているため動作が遅くなるという問題がある。それに対し SWT ではウィンドウシステムの API を直接呼び出しているため動作速度が早く、描画位置がずれることもない。さらに、ネイティブコードが小さいためバグも少ない (図 3.10)。Swing で作られたソフトウェアである Borland 社の JBuilder や Sun Microsystem 社の Sun One Studio と比べると SWT で作られた Eclipse は動作が速く、ボタンを押した際の反応などが早い。ただし、ウィンドウシステムの API を直接利用しているためルック&フィールはウィンドウシステムに依存する。また AWT との互換性がないため Java2D、Java Media Frameworks(JMF) といったライブラリを利用することができない。

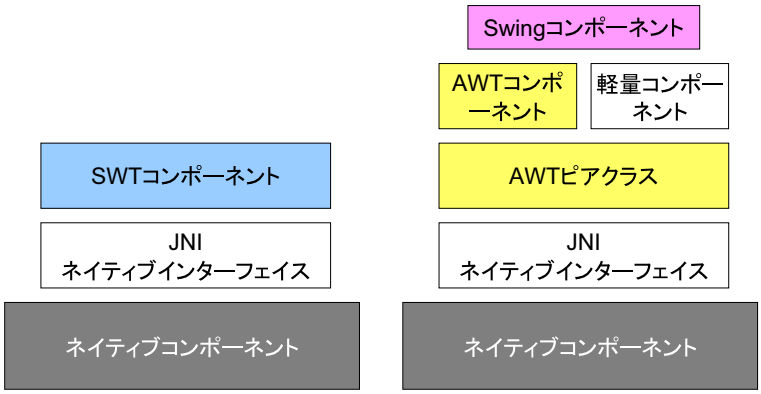

図 3.10: SWT、AWT/Swing の構成

また、SWT は Eclipse 専用ではなく、それ単独でも使用することもでき

る。しかし、現時点では OS ごとに OS 依存の SWT ライブラリ (Windows なら DLL，Linux なら so) が必要である。そのため Java の設計ポリシーである「Write Once, Run Anywhere」には反する。

	動作速度	異なる OS での描画位置のズレ	補足
AWT	早	ある	機能の制約やバグが多い
Swing	遅	なし	
SWT	早	なし	OS 依存の SWT ライブラリが必要

表 3.3: GUI ツールの比較

本システムのダイアログやビューなどは、この SWT を使って作成されている。

3.2.2　Bugdel エディタ

bugdel は Java で書かれたプログラムにデバッグコードを挿入するものであり Java 専用のエディタとして作られている。Eclipse SDK に付属している JDT には Java 専用のエディタがあり、このエディタにはコード補完機能やデバッガのブレイクポイントを指定する機能がある。そこで、bugdel エディタはこれらの機能を継承させて作成した。

以下に Eclipse に Bugdel エディタを加えるための拡張ポイントの指定と BugdelEditor クラスの説明をする。

`plugin.xml` ファイルの一部

```
<extension point="org.eclipse.ui.editors">
  <editor
    name="BugdelEditor"
    icon="icons/bugdelEditor.gif"
    filenames="*.java"
    class="bugdel.guipointcut.editor.BugdelEditor"
    id="bugdel.guipointcut.editor.BugdelEditor">
  </editor>
</extension>
```

`bugdel.guipointcut.editor.BugdelEditor` クラスの宣言文

```
public class BugdelEditor extends CompilationUnitEditor{
  ...
```

```
}
```

plugin.xml にはの中で書かれている「filenames=”*.java”」を指定することで Bugdel エディタが Java ファイルと関連付けられ、Eclipse 上で Java ファイルを開く際のエディタの候補に加えられる。

Eclipse に新たなエディタを追加する場合、拡張ポイント org.eclipse.ui.editors を指定し、エディタ機能を実装するクラスは org.eclipse.ui.IEditorPart インターフェイスを実装する。クラスを作成する際、基本機なテキストエディタ機能が実装済み org.eclipse.ui.TextEditor クラス (org.eclipse.ui.IEditorPart のサブクラス) を継承させることもできる。本システムのエディタの機能を実装している BugdelEditor クラスは JDT に含まれる Java 専用のエディタ CompilationUnitEditor クラス (TextEditor のサブクラス) を継承させている。CompilationUnitEditor クラスではカット&ペーストなどの通常のエディタの機能に加え Java のコード補完機能などが実装されているため BugdelEditor クラスにも、これらの機能が継承される。

ファイル保存時の動作

BugdelEditor クラスで新たに定義した機能は編集中のファイルが保存されたときの動作である。このエディタではファイルが保存された場合、デバッグコードが挿入される位置に表示されるマーカーの更新がされる (図 3.11)。これは CompilationUnitEditor クラスで定義されている doSave メ

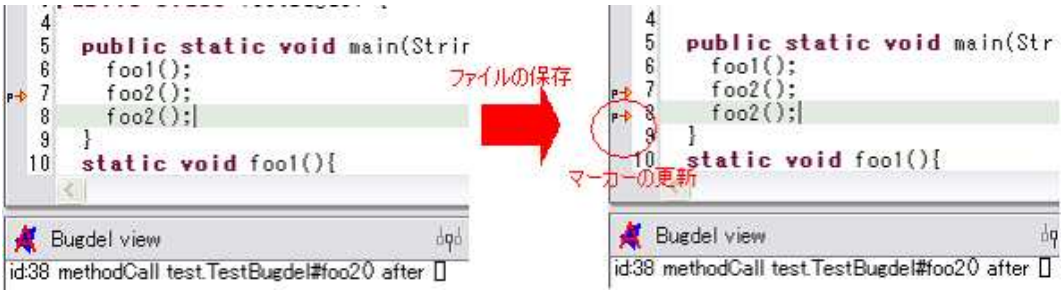

図 3.11: マーカーの更新

ソッドをオーバーライドすることで実現している。以下に BugdelEditor クラスの doSave メソッドを示す。

```
public void doSave(IProgressMonitor progressMonitor) {
  super.doSave(progressMonitor);
  pointcutMarkerUpdate();
  ...
```

```
}
```

doSave メソッドはエディタで表示されているファイルが保存されたときに呼び出される。CompilationUnitEditor クラスの doSave メソッドでは編集中の Java ファイルが保存された際、コンパイルしコンパイルエラーをエディタ上に表示する機能があり、BugdelEditor でもファイルを保存した際にコンパイルが行われる。

ポップアップメニューへの項目の追加

その他、機能追加としてメニュー項目の追加を行った。Bugdel エディタではメンバに関する pointcut を指定するためにエディタのポップアップメニューに「pointcut」項目を追加している。ポップアップメニューの項目のは editorContextMenuAboutToShow(IMenuManager menu) メソッドをオーバーライドすることで行われる。以下に BugdelEditor クラスの editorContextMenuAboutToShow メソッドの中身を示す。

```
public void editorContextMenuAboutToShow(IMenuManager menu) {
  IAction action = new PointcutSelectAction();
  menu.add(action);
  super.editorContextMenuAboutToShow(menu);
}
```

メニューバーへの項目の追加

メニューバー (図 3.8) には weave アクションのメニュー項目を追加している。メニューバーへのアクションの追加は以下のように行われる。getEditorSite メソッドは IEditorPart インターフェイス (BugdelEditor クラスのスーパーインターフェイス) で定義されているものである。

```
IAction action = ...
IActionBars actionBar = getEditorSite().getActionBars();
IMenuManager menu = actionBar.getMenuManager();
menu.add(action);
```

ルーラーアクションの動作変更

ルーラー (図 3.8) の動作は BugdelEditor クラスのスーパークラスである CompilationUnitEditor クラスで定義されている。Bugdel エディタでは行単位の pointcut を指定する際に表示されるポップアップメニューに「line

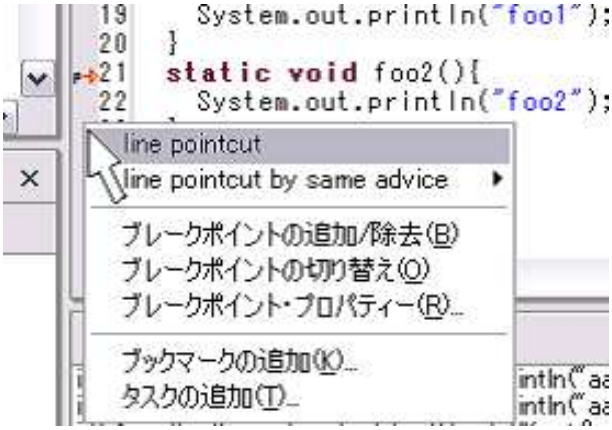

図 3.12: ルーラーの動作変更

pointcut」項目が表示されるように動作を変更している。CompilationUnitEditor クラスではダブルクリックした際デバッガのブレイクポイントが設定されるなどルーラーの動作が定義されている。このルーラー上の動作は拡張ポイント org.eclipse.ui.popupMenus を指定することで変更することができる。以下に pluin.xml ファイルの一部を示す。

```
<extension point="org.eclipse.ui.popupMenus">
  <viewerContribution
     targetID="#CompilationUnitRulerContext"
     id="bugdel.guipointcut.viewerContribution">
    <action
       ...
       class="bugdel.guipointcut.editor.action.RulerAction"
       ...
    </action>
  </viewerContribution>
</extension>
```

targetID ではルーラーの動作を変更する対象を表している。また、実際動作の定義を行うのは bugdel.guipointcut.editor.action.RulerAction クラスであり、このクラスは org.eclipse.ui.texteditor.AbstractRulerActionDelegate を継承して作成されている。bugdel.guipointcut.editor.action.RulerAction クラスでは menuAboutToShow メソッドを実装してポップアップメニューに項目を追加している。以下に RulerAction クラスのソースの一部を示す。

```
public void menuAboutToShow(IMenuManager manager) {
```

```
  ...
  IAction action = new LinePointcutAction();
  manager.add(action);
  ...
}
```

その他、デバッグコードが挿入される位置で表示されるポップアップメニューに「adviced by」項目なども追加している。

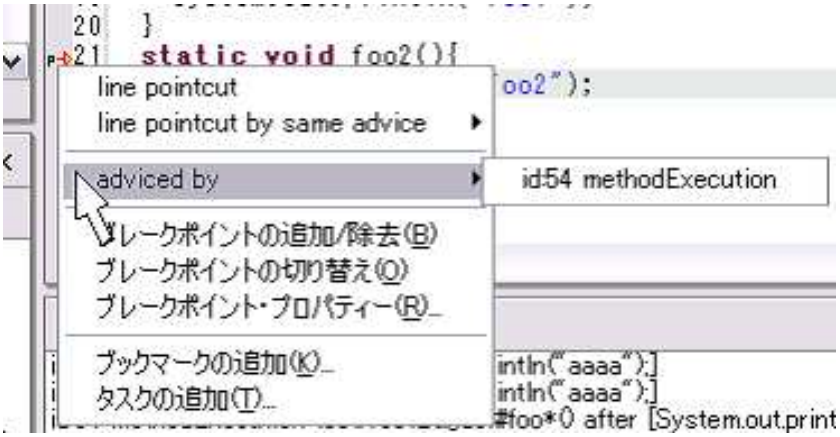

図 3.13: ルーラーのポップアップメニュー項目の追加

3.2.3　ソースコードの解析

本システムでは章 3.1.1 で述べたようにソースコード中のクラスメンバをクリックした際、pointcut の候補 (図 3.2) を表示する。この操作を行うにはカーソルで指定されているものが、クラス名、メソッド名、フィールド名であるのかを本システムが判断しなければならない。そこで、Java ソースコードの解析が必要になる。

Eclipse SDK に付属している JDT の機能は主に Java プログラミング環境を提供するエンドユーザ向けの機能であるが、他のプラグインが Java のソースコードを操作するための機能も提供している。ソースコードをモデル化して操作するには Java エレメント API を使用する方法と DOM/AST API を利用する方法がある。

Java エレメント API

Java エレメント API は org.eclipse.jdt.core パッケージに含まれる。この API は Java のリソース全体をモデル化するものである。

インターフェイス名	説明
IJavaProject	プロジェクトを表す
ICompilationUnit	1 つのソースコードを表す
IClassFile	クラスファイルを表す
IType	クラス又はインターフェイスを表す
IMethod	メソッド又はコンストラクタを表す
IField	フィールドを表す

表 3.4: Java エレメント API

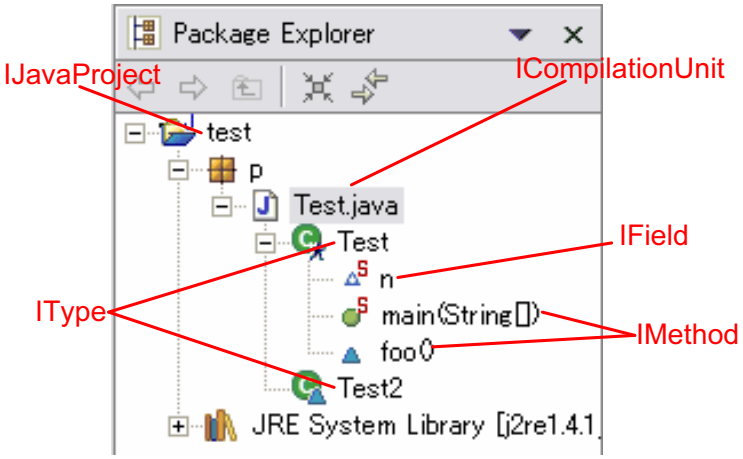

図 3.14: java エレメントの概要

Java エレメント API を利用することでソースコードの操作することができる。例えば IType には getFields、getMethods、createField、 createMethod メソッドがあり、これらのメソッドを使ってソースコード中に宣言されているメンバの検索や削除を行うことができる。以下に利用例を示す。

```
IEditorPart editor = ...
IFile file = ((IFileEditorInput)editor.getEditorInput()).getFile();
ICompilationUnit unit = JavaCore.createCompilationUnitFrom(file);
IType[] types = unit.getTypes();
```

この例はエディタ上のソースコードファイル取得して、その中で定義されているクラスとインターフェイスを調べるものである。

また、ICompilationUnit インターフェイスには codeSelect メソッドが定義されており、本システムではこのメソッドを使ってカーソル位置で指定されているクラスメンバを特定している。このメソッドは int 型の引数をとりソースコード中の文字番号 (先頭から何文字目であるのか) を渡すことで、その位置に存在する Java エレメントを返す。そこで、カーソル位置の文字番号をこのメソッドに渡すことでカーソル位置のクラスメンバを特定することができる。以下にクラスメンバを特定する本システムのプログラムの一部を示す。

```
IEditorPart editor = ...
ICompilationUnit unit = ...
ISelection selection = editor.getSelectionProvider().getSelection();
//カーソル位置の特定
int position = ((TextSelection)selection).getOffset();
//メンバの特定
IJavaElement el = unit.codeSelect(position, 0)[0];
if(el instanceof IType){
  ...
}else if(el instanceof IMethod){
  ...
}else if(el instanceof IField){
  ...
}
```

このようにクラスメンバを特定し、選択されいるメンバがメソッドならば表 3.1 の methodCall、methodExecution をフィールドならは fieldSet、fieldGet をクラスならば、そのクラスで定義されているメンバの pointcut を候補として表示する。

さらに、本システムではメソッドに関する pointcut の候補を表示するとき、指定したメッドの名前と引数の型名を表示する (図 3.15)。

メソッドの名前は IMethod の getElementName() メソッドにより取得し、引数の型名は getParameterTypes() で取得できる。しかし、IMethod はソースコードを元にをモデル化しているため getParameterTypes() で取得できる型名には制限がある。例えばソースコード中に「void foo(String s)」と記述された IMethod の getParameterTypes() で取得できる引数の型名は”String”であり”java.lang.String” ではない。同様にソースコードの先頭に import を宣言して使っているクラスについても正確なクラス名を取得できない。この問題を解決するために IType インターフェイス

図 3.15: メソッドに関する pointcut の表示

には resolveType(String unresolvedNamed) メソッドが用意されている。resolveType(String unresolvedNamed) では IType が表すソースコード中に存在する import 文を調べ unresolvedNamed が表す型の正確な名前を返す。以下にメソッドの引数の型名を取得するための本システムのコードの一部を示す。

```
IMethod method = ...
//メソッドが定義されているクラスを取得する
IType declaringType = method.getDeclaringType();
String[] parameter = method.getParameterTypes();
for(int i=0; i<parameter.length; i++){
  String unresolvedNamed = Signature.toString(parameter[i]);
  //resolveType メソッドにより正確なクラス名を取得する
  String[][] nameRec = declaringType.resolveType(unresolvedNamed);
  String resolveName = nameRec[0][0]+"."+name[0][1];
  ...
}
```

また、詳しいソースコードの検索は org.eclipse.jdt.core.search. パッケージを使って行うことができる。以下に全てのフィールドアクセス位置を検索する例を示す。

```
//コレクタの定義
class MyCollector implements IJavaSearchResultCollector{
  public void accept(IResource resource, int start, int end,
      IJavaElement element, int accuracy) throws CoreException {
    Syste.out.println(element.getElementName()+" start:"+start+" end"+end);
  }
```

```
  ...
}
//検索対象のソースコード
ICompilationUnit unit = ...;
IJavaSearchScope scope =
  SearchEngine.createJavaSearchScope(new IJavaElement[] { unit });
//アクセスパターン
int accessPattern  = IJavaSearchConstants.READ_ACCESSES
                   | IJavaSearchConstants.WRITE_ACCESSES;
MyCollector collector = new MyCollector();
SearchEngine searchEngine = new SearchEngine();
searchEngine.search(ResourcesPlugin.getWorkspace(), pattern, scope, collector);
```

このプログラムにより unit で表されるソースコードからフィールドアクセス位置を検索し、その情報がコンソールに出力される。

DOM/AST API

Java エレメント API を利用することでソースコードの中で定義されているメソッド、コンストラクタのシグネイチャの情報を取得することはできが、メソッド内部で記述されている処理コードについての情報を取得することはできない。

それに対し org.eclipse.jdt.core.dom パッケージに含まれている DOM/AST API は Java ソースコードを Java エレメントよりも詳細にモデル化しているためメソッド内部の情報も取得できる (図 3.16)。

クラス名	説明
CompilationUnit	1 つのソースファイルを表す
TypeDeclaration	クラス又はインターフェイスの宣言文を表す
MethodDeclaration	メソッド又はコンストラクタの宣言文を表す
FieldDeclaration	フィールドの宣言文を表す
MethodInvocation	メソッドの呼び出し文を表す
IfStatement	if 文を表す
TryStatement	try 文を表す

表 3.5: DOM/AST API

詳しいソースコードの検索は org.eclipse.jdt.core.dom.ASTVisitor クラスを用いて行う。以下に全てのフィールドアクセス位置を検索する例を示す。

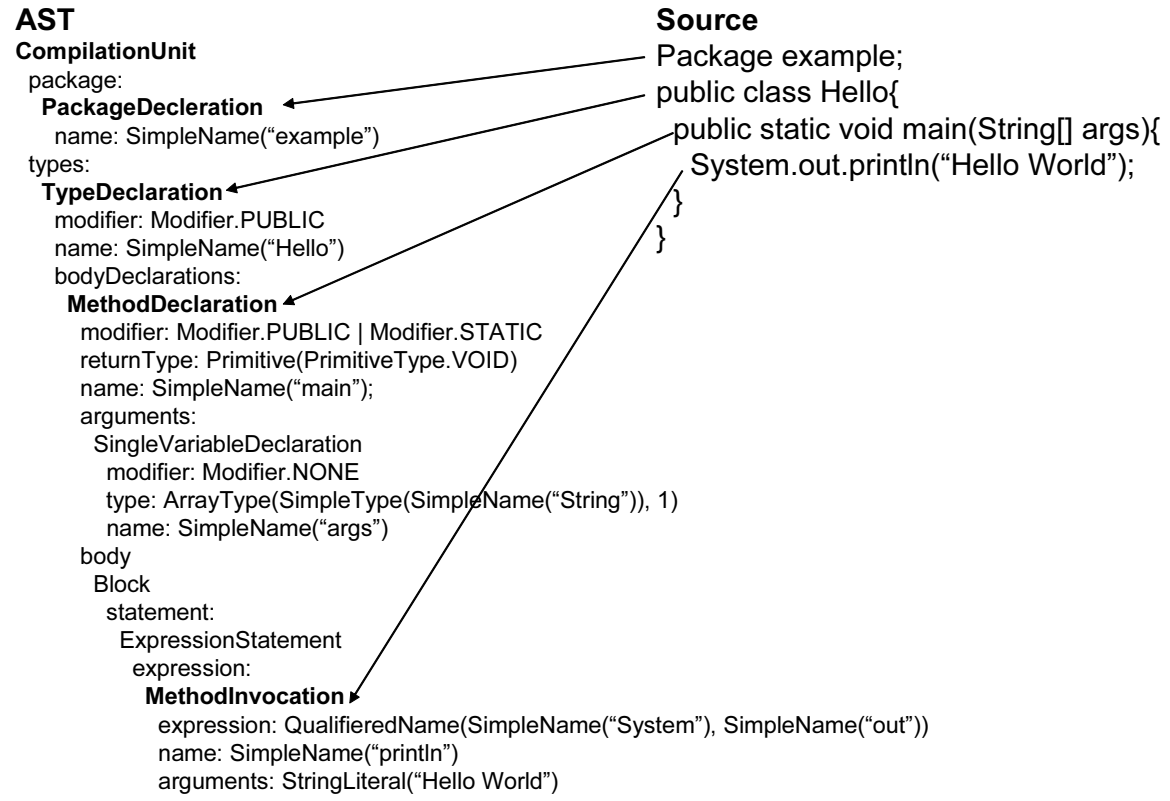

図 3.16: DOM/AST モデル

```
class MyASTVisitor extends ASTVisitor{
  public void visit(FieldAccess node){
    System.out.println(node.getName()+"start:"+node.getStartPosition());
    return true;
  }
}
CompilationUnit unit = ...
unit.accept(new MyASTVisitor());
```

このプログラムにより unit で表されるソースコードからフィールドアクセス位置を検索し、その情報がコンソールに出力される。ASTVisitor クラスで定義されているメソッドをオーバーライドすることで if 文の実行位置、メソッドの呼び出し位置など詳しい検索が行える。

3.2.4　指定した pointcut の情報の保存

本システムでは指定した pointcut を情報を保存しておき weave アクションが実行されたときに、その情報を元にデバッグコードを挿入する。指定した pointcut の情報を保存するのにはメンバに関する pointcut はシリアライズ機能を使ってファイルに保存し、行単位の pointcut はリソースマーカー機能を利用して保存している。以下でこれらの機能について説明する。

オブジェクトのシリアライズ

シリアライズとは計算機内部で扱われているデータを一次元に直列化しファイルに保存したりネットワークで通信したりする方法である。Java ではこの機能がサポートされおりオブジェクトをシリアライズすることでファイルに保存させることができる。

オブジェクトをシリアライズさせるには、そのオブジェクトのクラスが java.io.Serializable インターフェイスを implements する必要がある。しかし、このインターフェイスには宣言されているメソッドは無いため特別なメソッドを実装する必要は無い。Serializable インターフェイスを implements したクラスのオブジェクトは java.io.ObjectInputStream、java.io.ObjectOutputStream クラスを使って入出力ができる。以下に使用例を示す。

```
//シリアライズするクラス
class Point implements java.io.Serializable{
}

//オブジェクトの出力
Point p = new Point();
OutputStream outs = ...
ObjectOutputStream oouts = new ObjectOutputStream(outs);
oouts.writeObject(p);
oouts.flush();

//オブジェクトの入力
InputStream ins = ...
ObjectInputStream oins = new ObjectInputStream(ins);
Point p = (Point)oins.readObject();
```

上の例で書かれている OutputStream、InputStream オブジェクトをファイル (java.io.FileOutputStream) やソケット (java.net.Socket#getInputStream())

から取得することでローカルディスクに保存したり、ネットワークで通信したりすることができる。

本システムではメンバに関する pointcut(methodCall や fieldSet など)を指定した場合、PointcutAdvice オブジェクトが生成されシリアライズしてファイルに保存される。pointcut の内容を編集する際にはファイルから PointcutAdvice オブジェクトを読み込み、内容を変更した後、再びファイルに保存される。

リソースマーカー

Eclipse のワークスペースにはリソースマーカー機能があり行単位の pointcut はこの機能を利用して保存される。

行単位の pointcut はソースコードと密接に関係している。例えば、ソースコードの 10 行目を pointcut で指定した後、8 行目と 9 行目の間に新たなコードを付け加えた場合、指定した pointcut は一行後ろへずらさなければならない。また、8 行目が削除された場合一行前へずらさなければならない。そのためメンバに関する pointcut と同じように行単位の pointcut の情報をシリアライズしてファイルに保存した場合、ソースコードが編集される度にファイルから pointcut の情報を取り出し編集された行番号が pointcut の位置よりも上にあるのかを調べ適切に更新する必要があり複雑な操作を行わなければならない。

この問題に対して Eclipse ワークスペースに含まれるリソースマーカー機能を利用して解決することができる。リソースマーカーはソースコードの文字列や行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。また、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。本システムでは行単位の pointcut の情報を保存するためのリソースマーカーを定義している。このリソースマーカーには行番号と挿入するデバッグコードの情報を持たせている。

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

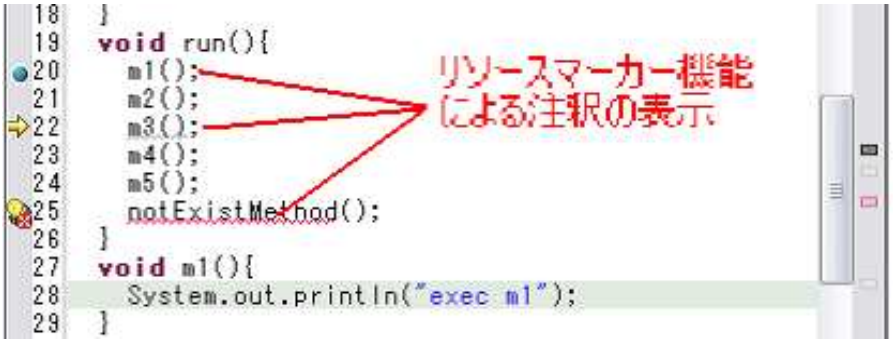

図 3.17: リソースマーカーによる注釈の表示

する。

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 3.6: リソースマーカーを操作する API

以下にリソースマーカーを操作するプログラム例を示す。

```
IResource resource = ...//ファイルシステムからリソースを取得
IMarker marker = resource.createMarker(IMarker.TASK);
marker.setAttribute(IMarker.LINE_NUMBER, 10);
IMarker[] markers = resource.findMarker(IMarker.BOOKMARK);
```

また、リソースマーカーを追加した位置の行番号に対応するルーラー上に
アイコンを表示されるためには拡張ポイントである org.eclipse.ui.markerImageProviders
plugin.xml ファイルに記述し、表示させるアイコンの画像ファイルを指定

する。本システムでは、この拡張ポイントを利用してデバッグコードが埋め込まれる位置にアイコンを表示させている (図 3.18)。

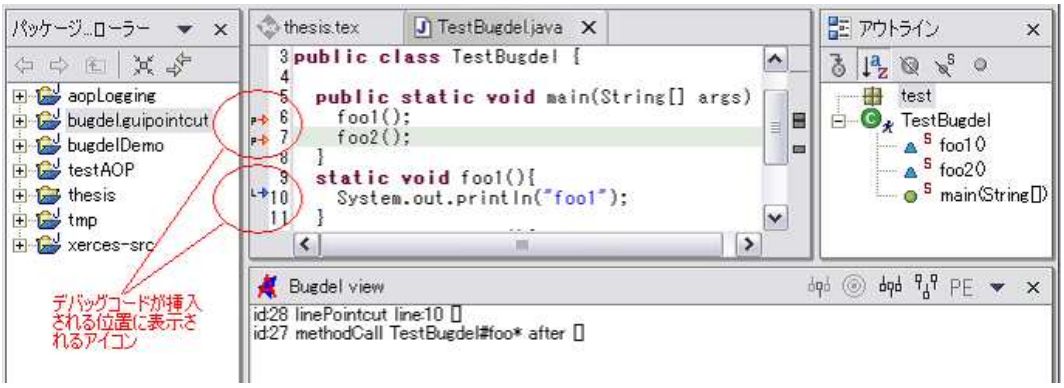

図 3.18: リソースマーカーのアイコン表示

3.2.5　デバッグコードの埋め込み

bugdel のユーザによって指定されたデバッグコードの埋め込みはソースコードを操作して行う方法とバイトコードを操作して行う方法が考えられる。ソースコードの操作を行うには章 3.2.3 で示した Java エレメント API や DOM/AST API を利用して行うことができるが問題がある。複雑なソースコード変換が必要な例を図 3.19 に示す。

このようにソースコードの変換を行いデバッグコードを埋め込むには複雑な処理を行わなければならない。また、ソースコードを変換した後クラスファイルを作成するためにコードを再コンパイルする必要がありオーバーベッドが大きくなってしまう。そこで本システムではバイトコードの変換を行うことでデバッグコードの挿入を行う。バイトコードを操作することで図 3.19 に示した変換は必要なくソースコードを再コンパイルする必要もない。

Javassist(バイトコード編集ライブラリ)

本システムではユーザによって記述されたデバッグコードの埋め込みをバイトコード (クラスファイル) の変換を行うことで実現している。バイトコードの変換には Javassist[9, 16, 15] を使用した。

Javassist とは Java のバイトコードを操作するための Java のライブラリ

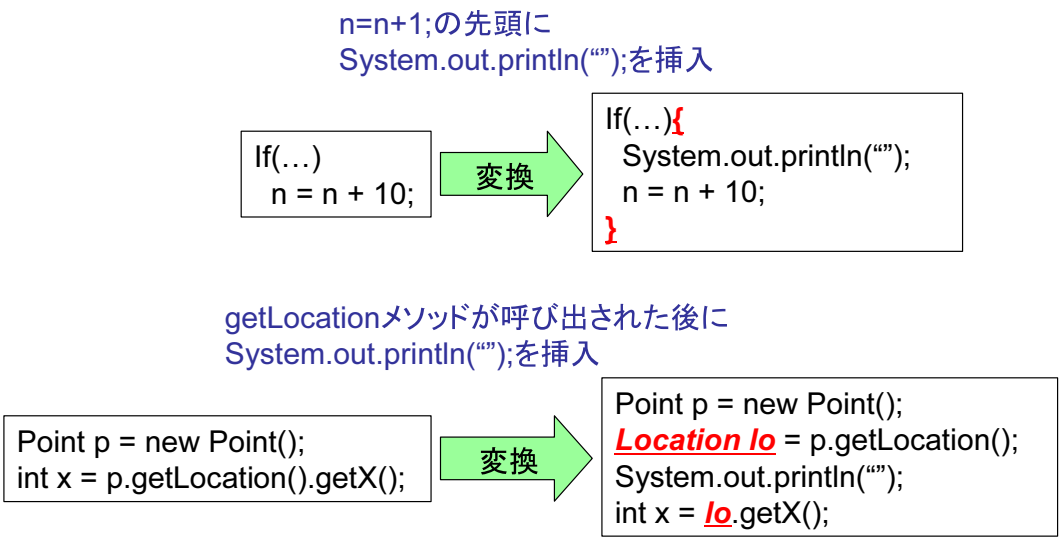

図 3.19: ソースコード変換によるデバッグコードの挿入

であり、構造リフレクションの機能を提供するものである。リフレクションとは計算システムが自分自身を構成する情報を取得して計算することである。Java の標準 API に含まれる java.lang.Class クラス、java.lang.reflect パッケージではリフレクションの一部の機能を提供しており、クラスが持つフィールドやメソッドなどの情報を取得することができる。以下に標準 API に含まれるリフレクションの使用例を示す。

```
import java.lang.reflect.Method;
public class TestReflect {
  public static void main(String[] args) throws ClassNotFoundException{
    Class clazz = Class.forName("java.lang.String");
    Method[] methods = clazz.getMethods();
    for(int i=0; i<methods.length; i++){
      System.out.println(methods[i].getName());
    }
  }
}
```

この例は java.lang.String クラスに定義されているメソッドを調べ定義されているメソッドの名前を全て出力するものである。このように java.lang.Class クラスを使いクラスで定義されているフィールドやメソッド、コンストラクタを調べることができる。しかし、標準 API に含まれるリフレクション機能ではプログラムの内観 (intropection) は行えるがプログラム自身の

変更は行えない。それに対し Javassist では標準 API に含まれるリフレクション機能に加え、バイトコードを変換することでクラスの定義を変更する構造リフレクション機能を提供している。

Javassist の特徴として次のものが上げられる。

- バイトコードの編集を行う際、BCEL [2]と異なりプログラマはバイトコードの詳しい知識を必用としない。
- メソッド呼び出し位置、コンストラクタ呼び出し位置、フィールドアクセス位置、ハンドラ実行位置などを簡単に検出できる。
- 100%Java で作られており Javassit を利用するための特別な実行環境を必要としない。

次に Javassist の使い方を説明する。リフレクション機能を利用するにはまず、クラスを表す javassist.CtClass オブジェクトを作成する。作成するにはクラスパスにある String 型でクラス名を入力する方法とストリームからバイトデータを取得する方法がある。

```
ClassPool pool = ClassPool.getDefault();
CtClass cc1 = pool.get("Point");
InputStream strem = ...
CtClass cc2 = pool.make(stream);
```

CtClass オブジェクトを操作することで、そのクラスで定義されているメソッドを表す CtMethod オブジェクト、コンストラクタを表す CtConstructor オブジェクトなどを取得しその振る舞いを変更することができる。

```
CtMethod method = ...
String statement = "System.out.println(\"start method\");";
method.insertBefore("{"+statement+"}");
```

上のプログラムにより method が表しているメソッドの本体にの先頭に statement で記述したコードが埋め込まれる。また、CtMethod、CtConstructor の insertAt メソッドを使うことでメソッドやコンストラクタの任意の行にコードを挿入することができる。

メソッド呼び出しやコンストラクタ呼び出しフィールドアクセスは javassist.expr パッケージの ExprEditor クラスを用いて検出することができ、その位置での振る舞いを変更することができる。以下にメソッド呼び出し位置に新たなコードを挿入する例を示す。

[2]Byte Code Engineering Library:Apache jakarta プロジェクトで開発されている Java 用バイトコード編集ライブラリ

メソッド名前	説明
Ctlass getSuperclass()	スパークラスを取得する
CtClass[] getInterfaces()	インターフェイスを取得する
CtConstructor[] getConstructors()	コンストラクタの一覧を取得する
CtMethod[] getMethods()	メソッドの一覧を取得する
CtField[] getFields()	フィールドの一覧を取得する
CtBehavior[] getDeclaredBehaviors()	メソッド、コンストラクタの一覧を取得する
int getModifiers()	修飾子を取得する
String getName()	クラスの名前を取得する
String getPackageName()	パッケージの名前を取得する

表 3.7: CtClass の内観用メソッド

```
CtClass clazz = ...;
clazz.instrument(new ExprEditor(){
  public void edit(MethodCall m) throws CannotCompileException {
    if(m.getMethodName().equals("move")){
      String statement = "System.out.println(\"call move\");"
      m.replace("{"+statement+"$_ = $proceed($$);}");
    }
  }
});
```

これにより move メソッドを呼び出している位置に statement で記述したコードが埋め込まれる。replace メソッドの引数の「$_ = $proceed($$)」は検出したメソッド呼び出し本体の振る舞いを表す。メソッド呼び出しを表す MethodCall オブジェクトの他にもフィールドアクセスを表す FieldAccess ハンドラの実行を表す Handler を検出することができ、それぞれの振る舞いを変更することができる。

以上のように CtClass オブジェクトを操作することでバイトコードを編集することができる。また、変換後のバイトコードは次のようにして取得する。

```
byte[] classData = clazz.toBytecode();
```

バイトコード変換によるデバッグコードの挿入

本システムでは章 3.1.1 で示した位置に対してバイトコードを変換しデバッグコードの挿入を行う。デバッグコード挿入に伴うバイトコードの変

換は対応する Javassist のメソッド [10] を実行することで行う。

bugdel の pointcut	コードの挿入を行うメソッド
methodExecution	javassist.CtMethod#insertBefore
	javassist.CtMethod#insertAfter
constructorExecution	javassist.CtConstructor#insertBefore
	javassist.CtConstructor#insertAfter
fieldGet	javassist.expr.FieldAccess#replace
fieldSet	javassist.expr.FieldAccess#replace
methodCall	javassist.expr.MethodCall#replace
constructorCall	javassist.expr.NewExpr#replace
handler	javassist.expr.Handler#insertBefore
cast	javassist.expr.Cast#replace
instanceof	javassist.expr.Instanceof#replace
行単位の pointcut	javassist.CtMethod#insertAt
	javassist.CtConstructor#insertAt

表 3.8: デバッグコード挿入のためのバイトコード変換用メソッド

また、Eclipse で Java プログラム開発する場合、生成されるクラスファイルはユーザが指定したフォルダに保存される。さらにプロジェクトが使用する jar ファイルなどのクラスパスもユーザによって指定されている。Javassist では CtClass オブジェクトを作成するためにクラスファイルの情報が必要であり、デバッグコードを Javassist がコンパイルするためにクラスパスの情報も必要である。これらの情報は章 3.2.3 で示した IJavaProject のオブジェクトにより取得することができる。

メソッド名	説明
IPath getOutputLocation()	クラスファイルの出力先を取得する
IClasspathEntry[] getResolvedClasspath(..)	クラスパスを取得する

表 3.9: プロジェクトの環境変数を取得する IJavaProject のメソッド

デバッグコードを挿入は、これまで述べた方法を使い以下の流れで行う。

1. クラスファイルの出力フォルダからクラスファイルを取得する。
2. クラスファイルからデータを取得し CtClass オブジェクトを生成する。
3. プロジェクトのクラスパスを検索し Javassist のコンパイラに追加する。
4. CtClass オブジェクトから CtMethod、CtConstructor の一覧を取得し methodExecution、constructorExecution で指定されているデバッ

グコードを挿入する。

5. CtClass オブジェクトの instrument(javassist.expr.ExprEditor) メソッドを使いメソッド呼び出しやフィールドアクセス位置を検出し fieldGet、fieldSet、methodCall、constructorCall、handler、cast、instanceof で指定されているデバッグコードの挿入を行う。
6. 行単位の pointcut の行番号を含むメソッド又はコンストラクタを検索し CtMethod、CtConstructor を取得してデバッグコードの挿入を行う。
7. CtClass オブジェクトからデバッグコードが挿入されたクラスのバイトコードデータを取得する。
8. クラスファイルの出力フォルダに変換したクラスファイルを保存する。

第4章　実験

bugdel と AJDT[1]によるデバッグコード埋め込みの時のオーバヘッドを計測するために実験を行った。実験に用いた環境は以下に示す。

- 計算機

  Sun Fire V480(Ultra SPARC ⊠ 900MHz × 4 CPU，メモリ 16GByte, Solaris 9)

- JVM

  Sun JDK1.4.0 付属の HotSpot^{TM}Client VM

- Eclipse

  Eclipse-SDK2.1.1
  実行する際に以下のコマンドによりメモリを 256MByte に設定

  ```
  >eclipse -vmargs -Xmx256m -Xms256m
  ```

- AJDT

  ajdt-0.6.3、ajde-1.1.3

実験は Xerces-J[2]2.6.0 のソースファイル (ファイル数 751、7.76MByte) をコンパイル、weave(デバッグコードの埋め込み) する時間を測定した。実行時間はコンパイル、weave の進捗状況を表示するダイアログが表示され破棄されるまでの時間を測った。weave で挿入するコードは「System.out.println();」であり、以下の 5 種類の pointcut を指定して挿入を行い、それぞれの実行時間を測定した。実験結果を図 4.1 に示す。

- org.*で表されるクラスのフィールド参照、フィールド代入、メソッド呼び出し、メソッド実行、コンストラクタ実行時点 (対応する joinpoint 51932ヶ所)
- org.apache.xerces.*で表されるクラスのフィールド参照、フィールド代入、メソッド呼び出し、メソッド実行、コンストラクタ実行時点 (対応する joinpoint 43635ヶ所)

[1]AspectJ プログラミング環境を提供する Eclipse プラグイン
[2]Java 用 XML ライブラリ

- org.apache.xerces.impl.d*で表されるクラスのフィールド参照、フィールド代入、メソッド呼び出し、メソッド実行、コンストラクタ実行時点 (対応する joinpoint 6127ヶ所)
- org.apache.xerces.impl.dtd.*で表されるクラスのフィールド参照、フィールド代入、メソッド呼び出し、メソッド実行、コンストラクタ実行時点 (対応する joinpoint 3091ヶ所)
- aaa.*で表されるクラスのフィールド参照、フィールド代入、メソッド呼び出し、メソッド実行、コンストラクタ実行時点 (対応する joinpoint 0ヶ所)

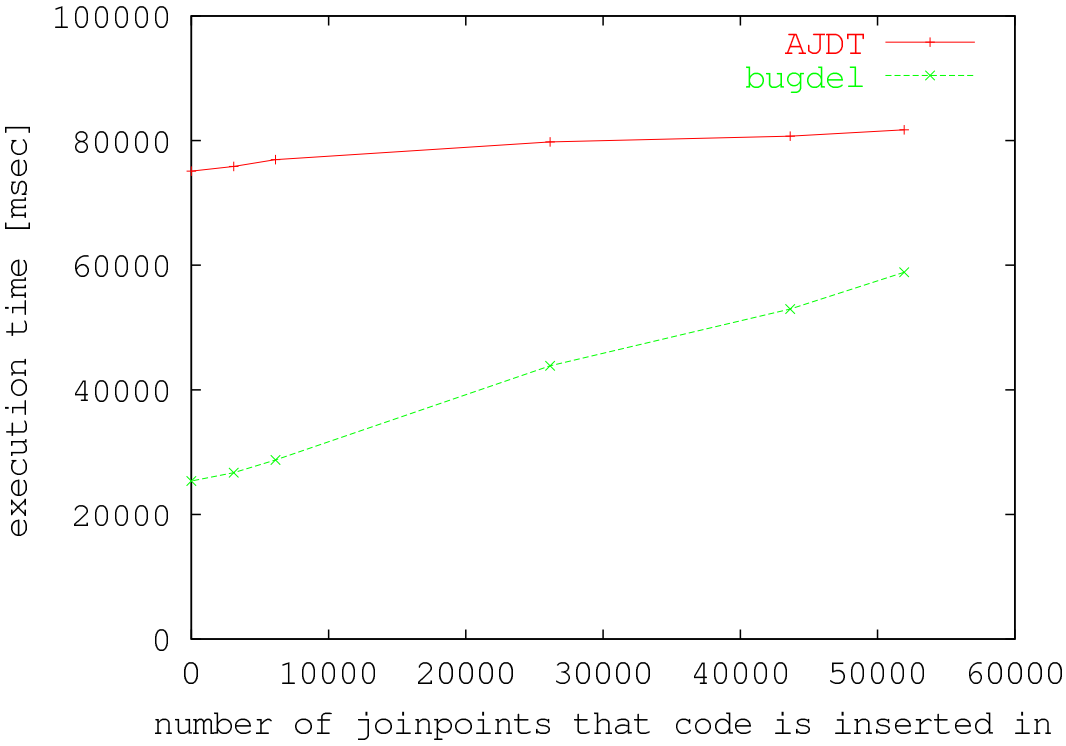

図 4.1: 実験結果

実験の結果からデバッグコードの挿入の数が増えると、それに伴う実行時間の増加の割合が Bugdel は AJDT に比べて大きいことが分かる。これは Bugdel では挿入するデバッグコードのコンパイルを各 joinpoint で毎回行っているためであると考えられる。それに対し AJDT(AspectJ のコンパイラ) では挿入するデバッグコードのコンパイルは初めに 1 度だけ実行され、joinpoint ではフックを埋め込むだけになっているため実行時間

の増加の割合が小さい。しかし、本実験で用いたファイル (総ファイルサイズ 7.76MByte) に対してデバッグコードを挿入する実行時間は AJDT と比べて遜色のない時間内でデバッグコードの挿入が行えている。

第5章　まとめ

本研究では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 Bugdel を提案した。Bugdel はデバッグに特化したアスペクト指向プログラミング環境を提供しており行単位の pointut を行うことができる。また、GUI により pointcut を指定することが可能であり複雑な pointcut 記述を行う必要がない。Bugdel を使うことによってデバッグコードの挿入を効率良く行うことができデバッグ作業の負担を少なくすることができる。また、デバッグのために挿入するコードは開発中のソースコードとは別に記述されソフトウェア配布時に取り忘れる心配がない。

本システム Bugdel は Eclipse 上にプラグインとして作成した。Eclipse 上に実装することで、Eclipse SDK によって提供されているエディタ機能や JDT の Java ソースコード解析機能を利用することができる。また、デバッグコードの挿入はバイトコードの変換を行うことで実現した。バイトコード変換にはバイトコード編集ライブラリである Javassist を用いた。

最後に Bugdel によるデバッグコードの埋め込みにかかる時間を測定し十分実用の範囲内でコードの埋め込みが行えることを確かめた。

5.1　今後の課題

今後の課題として次のことが上げられる。

- and(&) や or(||) による pointcut の連結

  AspectJ で採用されている pointcut 同士を関連付けするための and(&) や or(||) の機能を実装させる。Bugdel では pointcut 指定を GUI で行っている。そのため and(&) や or(||) の指定を GUI で行う操作方法を考える必要がある。

- バイトコードロード時のデバッグコード埋め込み

  Bugdel ではデバッグコードの埋め込みをユーザが weave アクション実行したときに行っている。そのため、実際にはデバッグで使用しないクラスにもデバッグコードが挿入されてしまい、コード挿入のための余分な時間がかかってしまう。そこで、クラスローダを使

いバイトコードのロード時にデバッグコードを埋め込む方法 [13] が考えられる。この方法を実現させるには Eclipse 上に Bugdel 専用の実行ボタンを作成し、Bugdel により生成されるクラスローダとコンソールビューを連携させる必用がある。

- 行単位 pointcut 情報の保存方法

  Bugdel ではクラスメンバに関する pointcut の情報の保存は Bugdel 専用のファイルへ保存させているため CVS(バージョン管理ツール) で管理することができる。しかし、行単位の pointcut に関しては Eclipse 内部で扱われているファイルシステムに保存されるので CVS で管理することができない。そこで、行単位の pointcut の情報も Bugdel 専用のファイルへ保存させることで CVS 管理を行えるようにしたい。

- デバッガとの連携

  デバッガと連携させることで、挿入するコード内でブレイクポイントの挿入削除を行えるようにする。また、現在はクラスファイルを変更することで静的にデバッグコードの挿入を行っているが、デバッガを利用し動的にデバッグコードを変更できるようする [17]。

参考文献

[1] AJDT aspectj development tools eclipse subproject. `http://www.eclipse.org/ajdt/`.

[2] AspectJ project. `http://www.eclipse.org/aspectj/`.

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

[4] Eclipse technical articles: technical information for tool developers. `http://www.eclipse.org/articles/`.

[5] Java(TM) Logging Overview. `http://java.sun.com/j2se/1.4/docs/guide/util/logging/overview.html`.

[6] JDT java development tools eclipse subprojectt. `http://www.eclipse.org/jdt/`.

[7] SWT standard widget toolkit. `http://www.eclipse.org/swt/`.

[8] Frank Budinsky, Timothy J. Grose, David Steinberg, Raymond Ellersick, and Ed Merks. *Eclipse Modeling Framework: A Developer's Guide*. Addison Wesley, Aug 2003.

[9] Shigeru Chiba. Load-time structural reflection in java. In *Proceedings of the European Conference on Object-Oriented Programming(ECOOP)*, pages 313–336, Jun. 2000.

[10] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the second international conference on Generative Programming and Component Engineering (GPCE)*, pages 364–376, September 2003.

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *European Conference on Object Oriented Programming (ECOOP)*, pages 327– 353, June 2001.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference for Object-Oriented Programming(ECOOP)*, pages 220–242, Jun 1997.

[13] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *European Conference on Object Oriented Programming (ECOOP)*, pages 36–44, 1998.

[14] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, and John Kellerman. *The Java Developer's Guide to Eclipse*. Addison Wesley, March 2003.

[15] 千葉 滋. Javassist home page. `http://www.csg.is.titech.ac.jp/~chiba/javassist/,http://www.jboss.org/developers/projects/javassist.html`.

[16] 千葉 滋 and 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**, 42(11):2752–2760, Nov. 2001.

[17] 佐藤 芳樹 and 千葉 滋. 効率的な java dynamic aop システムを実現する just-in-time weaver. **情報処理学会 論文誌**, 44(13):15–24, 10 2003.

付 録A　Bugdelのデモ画面

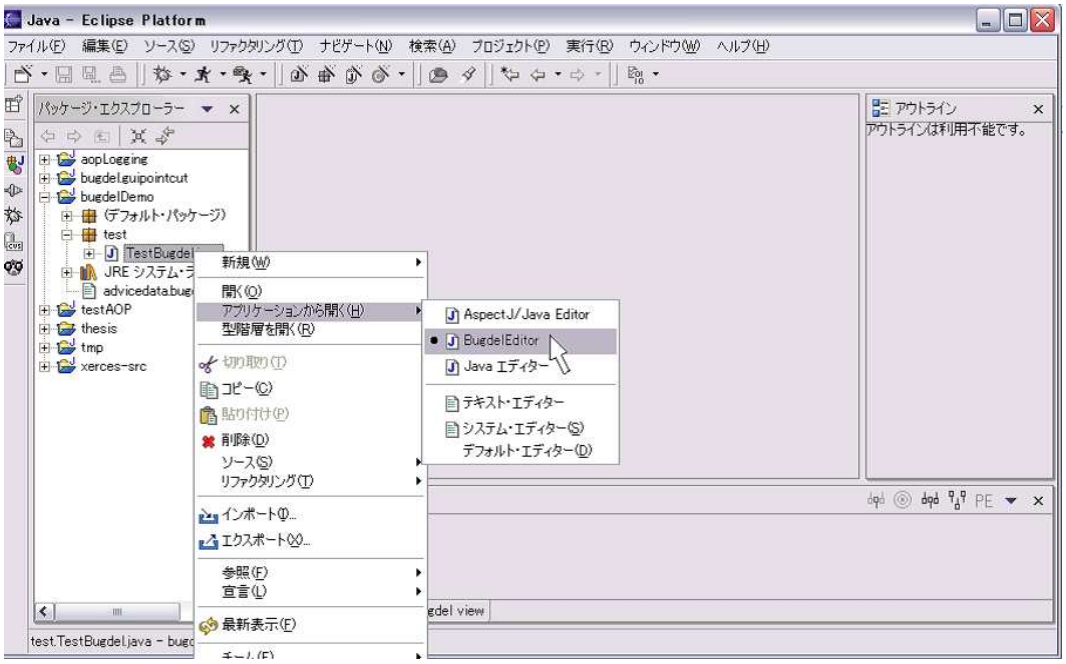

図 A.1: Bugdel エディタの起動

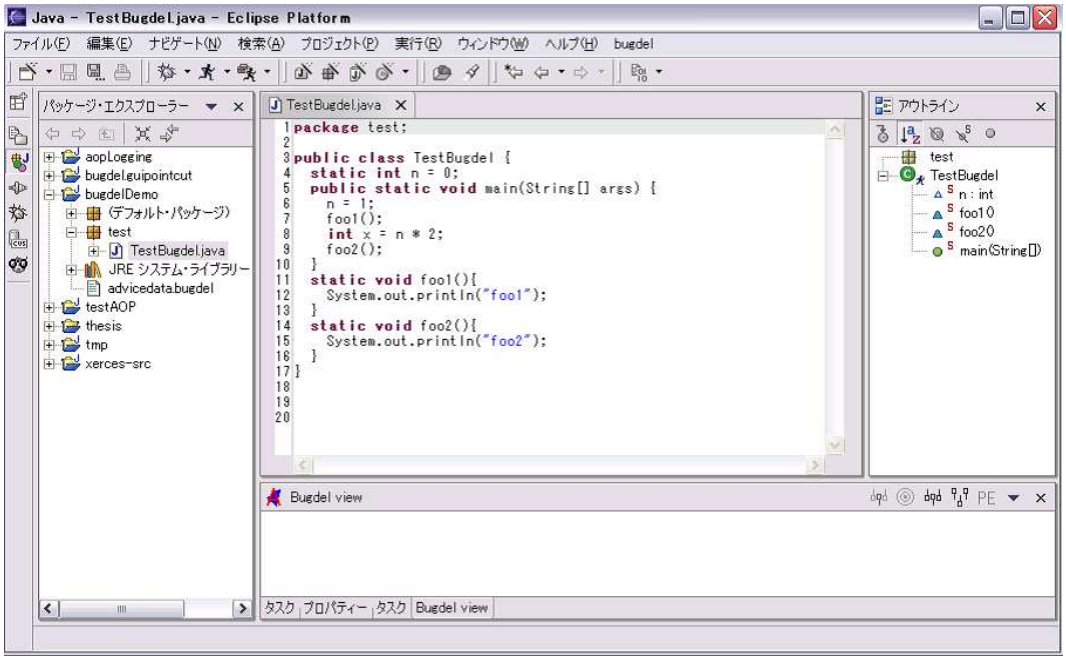

図 A.2: Bugdel エディタの起動 2

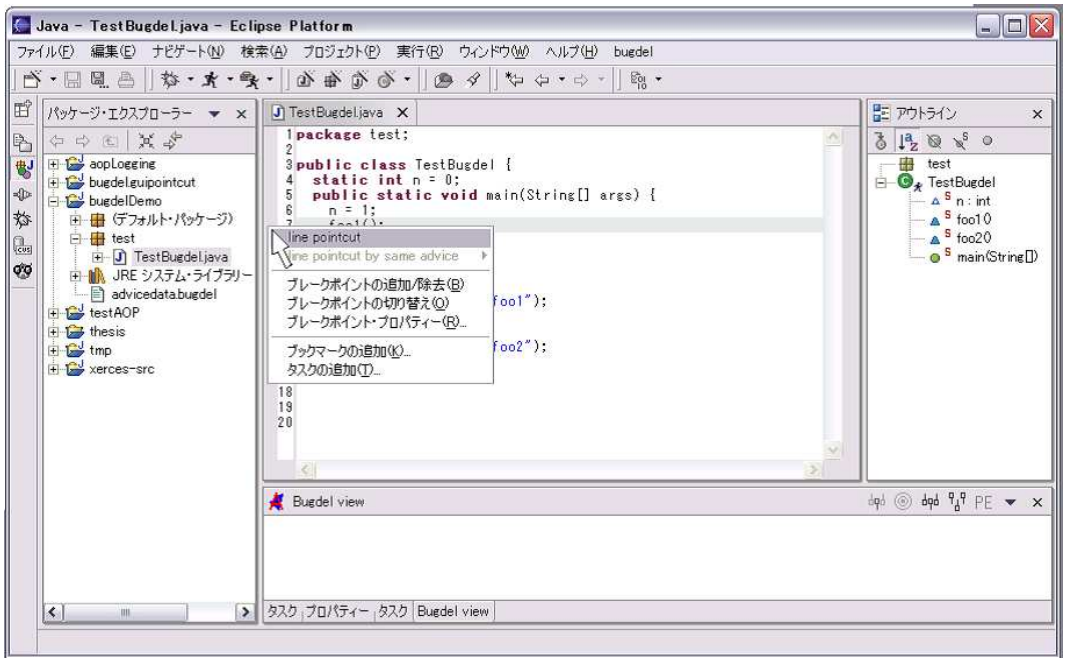

図 A.3: 行単位の pointcut 指定 (デバッグコード挿入位置の指定)

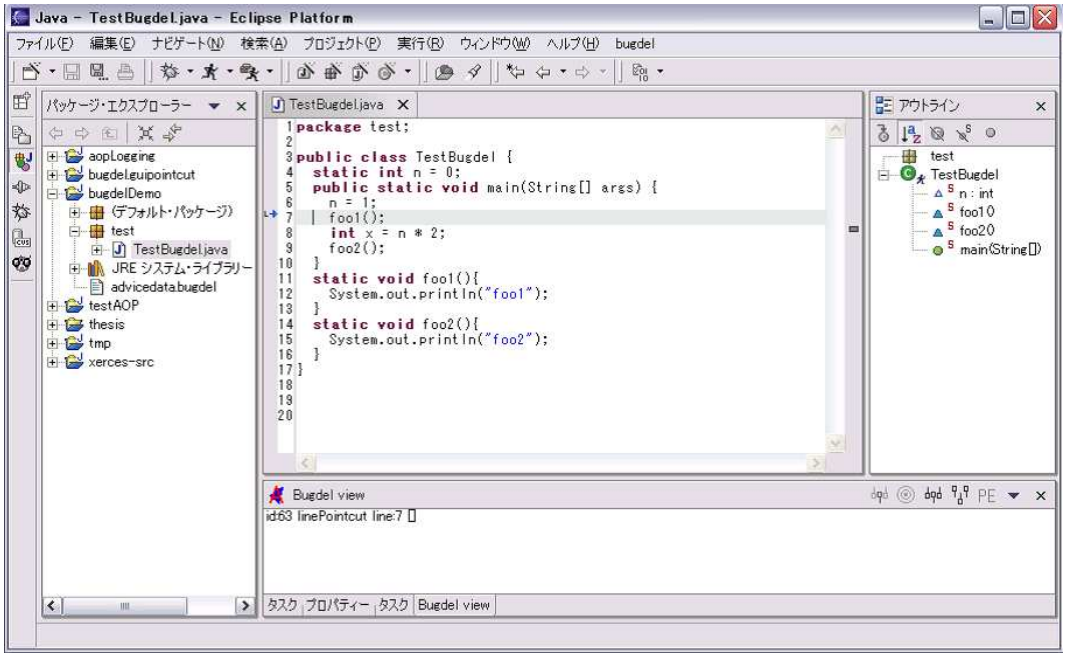

図 A.4: ルーラー (行番号の左) 上にマーカーを表示

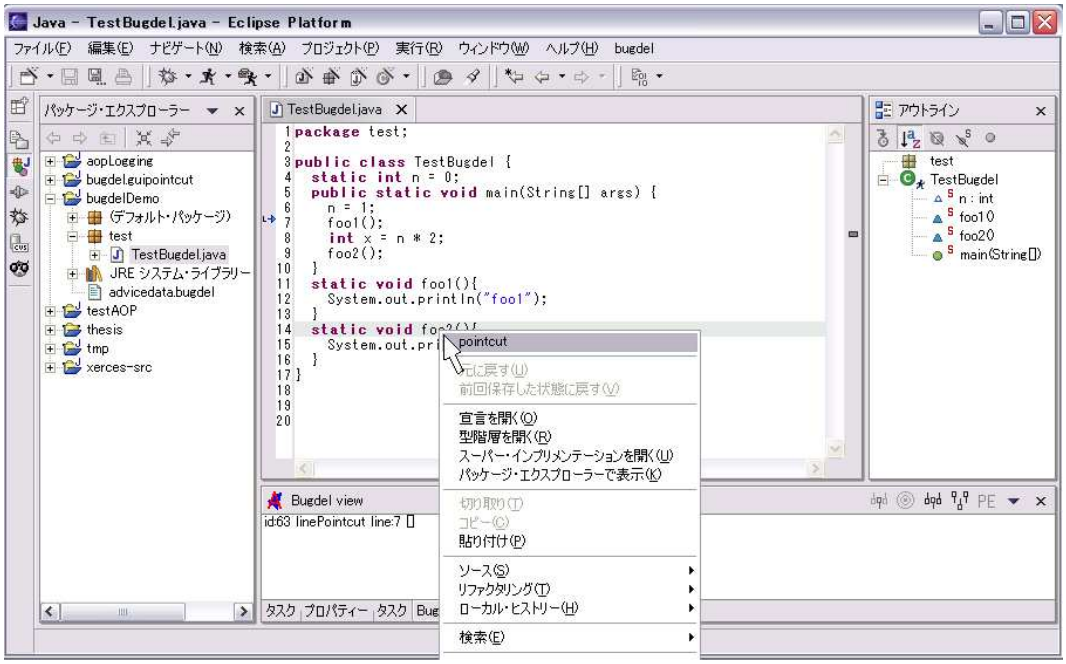

図 A.5: メソッドに関する pointcut 指定 (デバッグコード挿入位置の指定)

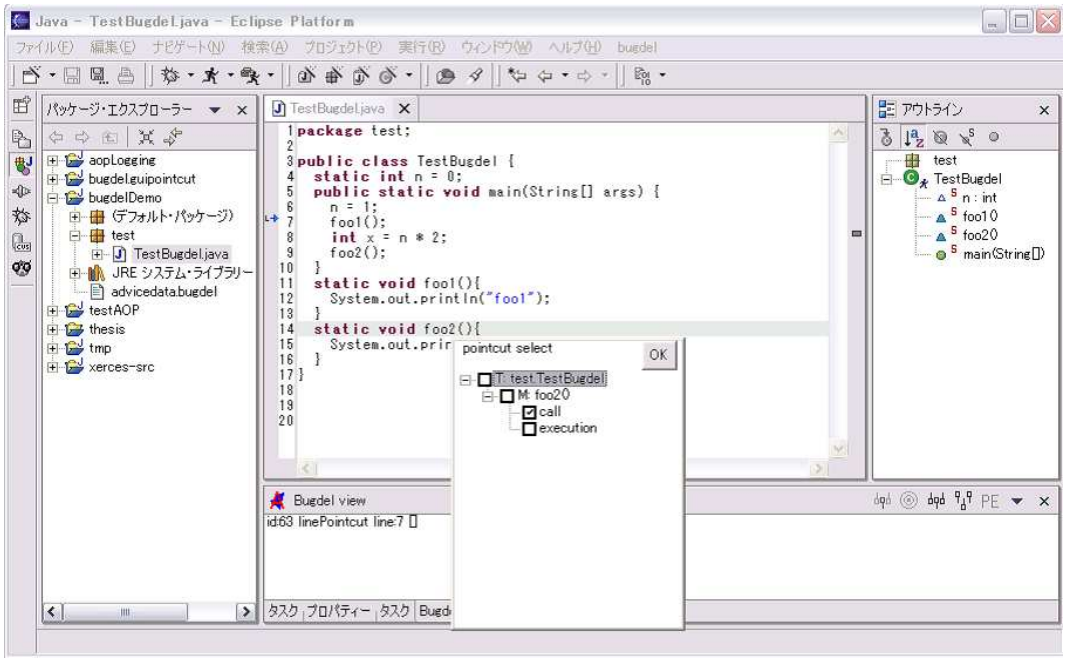

図 A.6: メソッドに関する pointocut 候補の表示

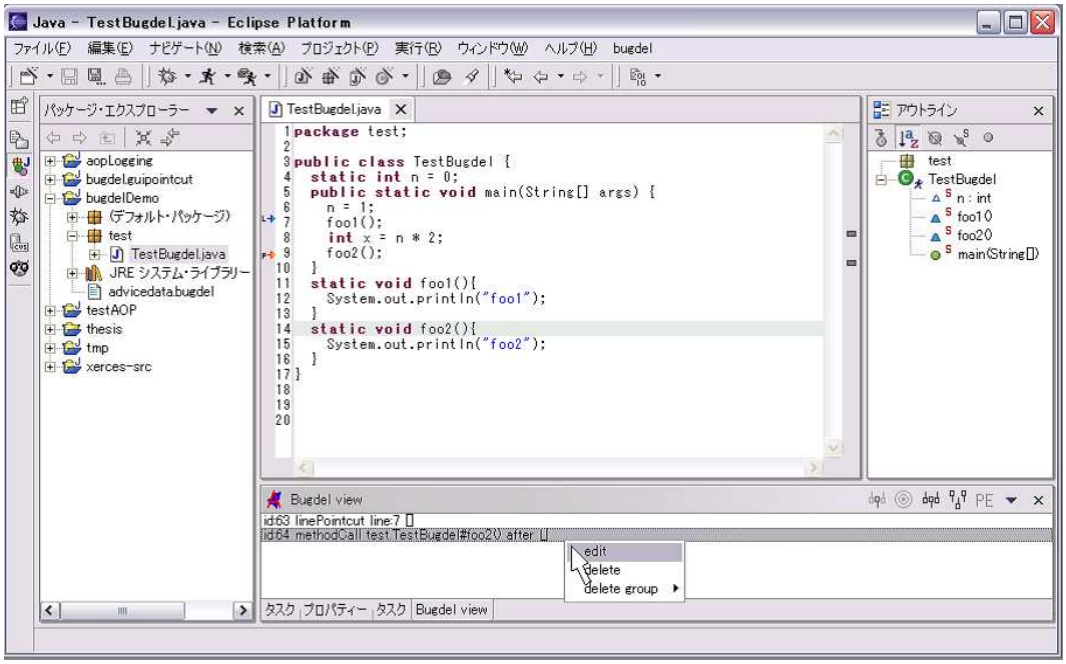

図 A.7: 指定済みの pointcut の選択

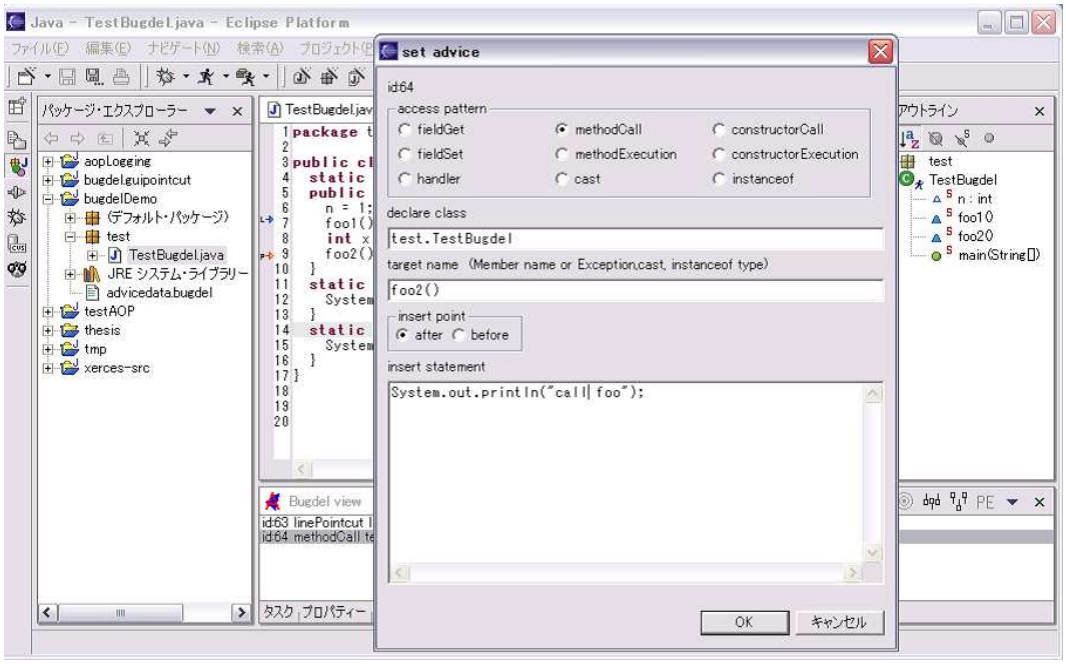

図 A.8: デバッグコードの編集

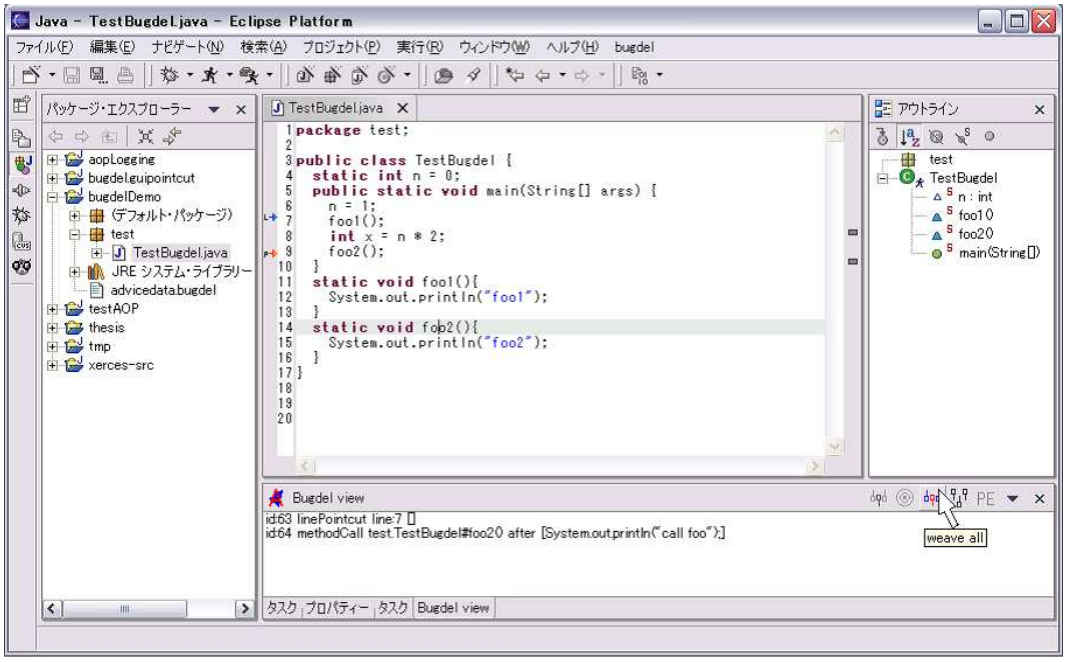

図 A.9: weave アクションの実行

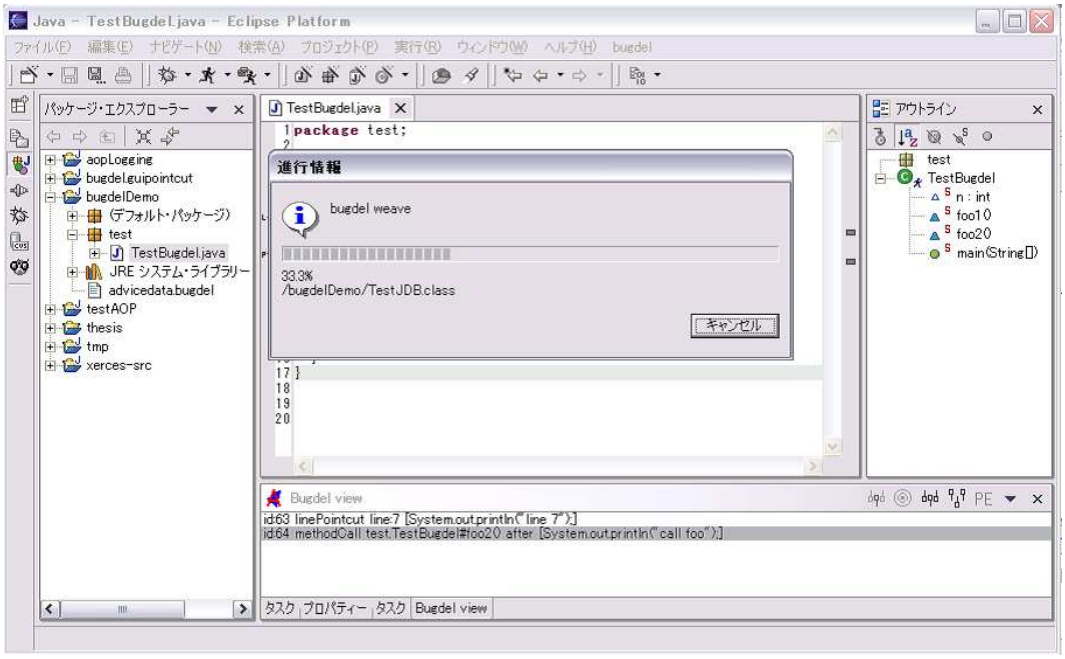

図 A.10: デバッグコードの埋め込み

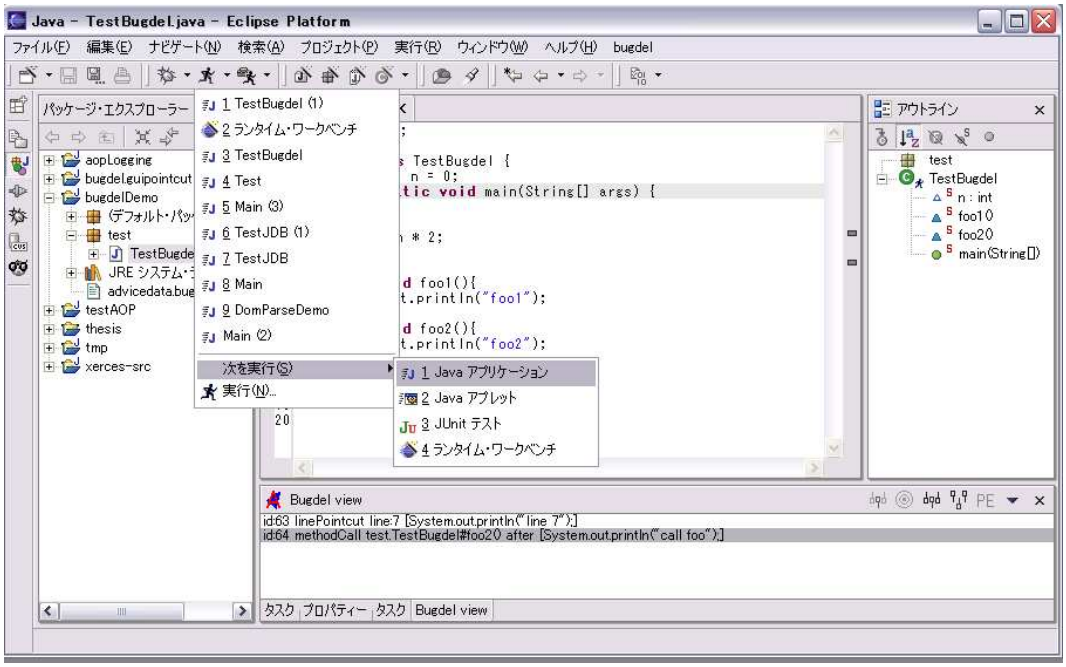

図 A.11: プログラムの実行

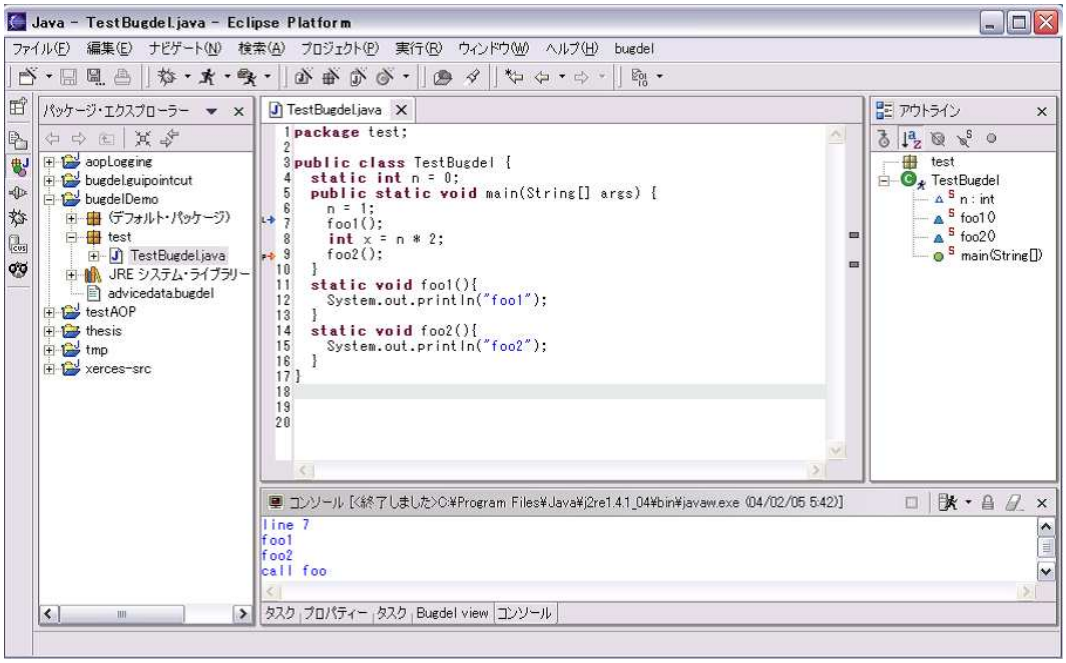

図 A.12: プログラムの実行結果

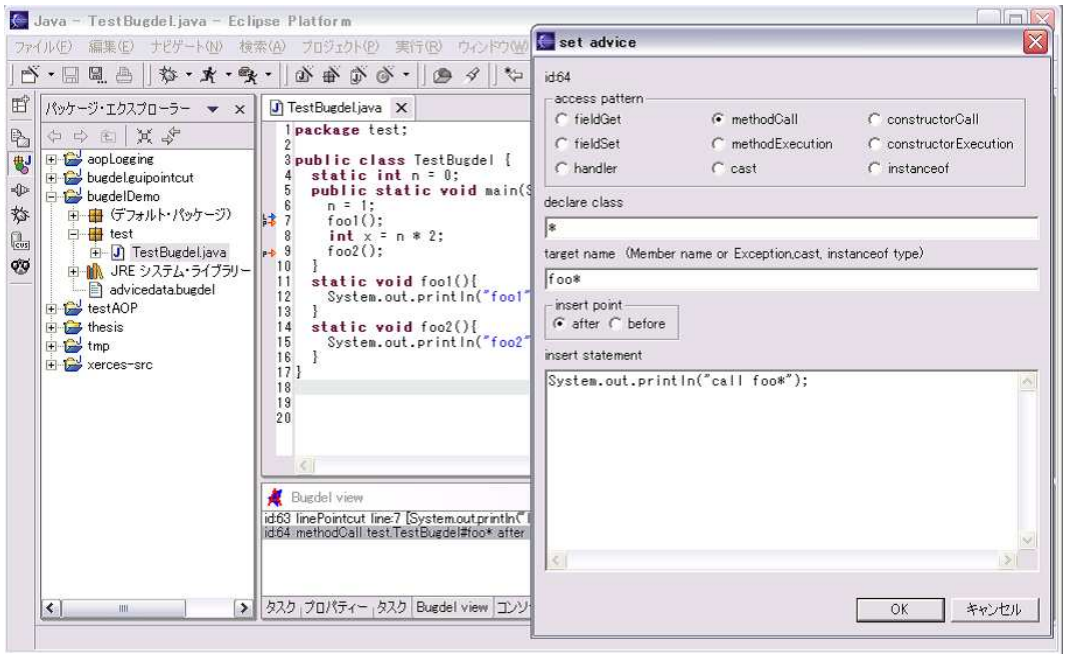

図 A.13: '*' を使った pointcut 指定