平成20年度 学士論文

統合開発環境のための
プログラミング言語拡張
フレームワーク

東京工業大学 理学部 情報科学科
学籍番号 05-2295-1
松本 久志

指導教員
千葉 滋 教授

平成21年2月5日

概要

ソフトウェア開発の場において、既存のプログラミング言語仕様に独自の変更を行いたい場合が存在する。例えばJavaを用いてデータベースを利用するソフトウェアの開発を行う場合、O/Rマッピングを行うための独自の機構を言語拡張としてJavaに追加することで、生産性の向上が期待できる。このような言語拡張を容易に行うためのシステムとして、PolyglotやJastAddなどといった、拡張を行いやすいコンパイラの研究が存在する。これらのシステムでは、構文木や中間表現の変換を用いて拡張言語のコンパイラを作成する。

しかし、一般的にソフトウェア開発の際に用いられる統合開発環境に対しては、このような言語拡張システムは存在しない。統合開発環境とは、コンパイラやデバッガ、テキストエディタなどの機能を備えた、ソフトウェア開発のための環境である。統合開発環境ではソースの編集時にリアルタイムにエラーの報告を行うため、高速にエラー解析を行う必要がある。そのため、コンパイラに対して用いたPolyglotやJastAddのような構文木などの変換を伴う手法は、解析時間の増大につながり適切ではない。

本研究では、統合開発環境のための言語拡張システムを提案する。本フレームワークは、コンパイル時及び編集時の2種類のメタオブジェクトプロトコルを提供する。コンパイル時にはソースコードの変換を、編集時にはエラー出力の操作を、メタオブジェクトの記述に従って行う。本フレームワークのユーザーは、言語拡張をメタオブジェクトとして記述することで、統合開発環境の扱う言語の拡張が可能となる。

本フレームワークは、統合開発環境によるコンパイルの直前に、メタオブジェクトによる処理を挿入する。拡張された言語から拡張前の言語への変換処理をメタオブジェクトとして記述することで、統合開発環境上で拡張言語のコンパイルが可能となる。ソースの変換方法としては、ソースコードを文字列として変換する方法、及びソースコードの抽象構文木を用いた変換方法を提供する。抽象構文木を用いた変換では、拡張前の言語の構文解析器を用いて構文木の生成を行う。

また、ソースコードの編集時にはこのようなソース変換は行わず、メタオブジェクトに基づいたエラー出力の操作を行う。拡張言語を用いて記述

されたソースコードは、編集時にはエラーとして認識される。本フレームワークではエラー出力の前にメタオブジェクトによる処理を挿入することで、このような言語拡張に由来するエラーの除去や、新たに加えるべきエラーの追加を行うことが可能である。

本フレームワークを用いることにより、小さな言語拡張であれば独自に構文解析器を記述することなく行うことが可能である。また、メタオブジェクト内で Polyglot などを用いたソース変換を行うことにより、AspectJ のような大きな言語拡張を行うことも可能である。

本研究の実装は、Eclipse の JDT プラグインを拡張することにより行った。さらに、2 進法表記及び open class の機構の追加を言語拡張の例として用い、本フレームワークを用いて統合開発環境の拡張を行った。その結果、それぞれメタオブジェクトとして 2 進法表記には 21 行の、open class の機構には 50 行のコードを記述することでこれらの拡張を Java 言語に追加可能であり、本フレームワークを用いることで統合開発環境を容易に拡張可能であることを確認した。

謝辞

　本研究を進めるにあたり、研究の方向性や論文の構成など多くの点でご指導をしてくださった指導教員の千葉滋教授に感謝致します。

　また、論文のスタイルファイルを作成して頂いた光来健一氏、研究を進める上でさまざまな助言を頂いた今吉竜之介氏、共に研究に励んだ同研究室の皆様に感謝致します。

目 次

第 1 章　はじめに **8**

第 2 章　既存の手法とその問題点 **10**
- 2.1 AJDT . . . 10
  - 2.1.1 アスペクト指向とは . . . 10
  - 2.1.2 AspectJ . . . 11
  - 2.1.3 AJDT . . . 12
- 2.2 関連研究とその問題点 . . . 15
  - 2.2.1 GluonJ . . . 15
  - 2.2.2 extensible compiler . . . 18

第 3 章　設計 **22**
- 3.1 メタオブジェクトプロトコル . . . 22
- 3.2 本フレームワークの設計 . . . 23
  - 3.2.1 コンパイルタイムメタオブジェクトプロトコル . . . 23
  - 3.2.2 エディットタイムメタオブジェクトプロトコル . . . 24
- 3.3 例 . . . 25
  - 3.3.1 コンパイル時 . . . 26
  - 3.3.2 エディット時 . . . 27
- 3.4 本フレームワークの設計のまとめ . . . 27
  - 3.4.1 既存の手法の問題点に対する考察 . . . 28

第 4 章　実装 **29**
- 4.1 Eclipse の拡張 . . . 29
  - 4.1.1 マニフェスト・ファイル . . . 29
  - 4.1.2 マニフェスト・エディター . . . 30
- 4.2 本フレームワークの実装 . . . 30
  - 4.2.1 JDT のコンパイラの動作 . . . 31
  - 4.2.2 メタオブジェクトプロトコルの構成 . . . 33
  - 4.2.3 メタオブジェクトのロード . . . 35
  - 4.2.4 コンパイル時に用いる抽象構文木 . . . 36

第 5 章　言語拡張例 **37**
5.1 2 進法表記 37
5.1.1 文法 37
5.1.2 メタオブジェクトの記述方法 37
5.2 @Refines 40
5.2.1 文法 40
5.2.2 メタオブジェクトの記述方法 41
5.3 拡張された統合開発環境 44
5.4 メタオブジェクトの記述量 46

第 6 章　まとめと今後の課題 **47**
6.1 まとめ 47
6.2 今後の課題 47
6.2.1 抽象構文木 47
6.2.2 コンパイルの問題 48
6.2.3 メタクラス 48
6.2.4 エラーの操作 48

図 目 次

2.1 AJDT を用いた AspectJ ソフトウェア開発 11
2.2 AspectJ を用いたプログラム例 12
2.3 AJDT を用いた AspectJ のソースコード編集 13
2.4 Eclipse のアーキテクチャ 14
2.5 GluonJ を用いて記述したアスペクト例 16
2.6 Eclipse を用いた GluonJ のソースコード編集 18
2.7 Polyglot を用いてコンパイラが実現されている言語 19
2.8 Java コンパイラ及び Non-null checker の機構の JastAdd を用いた実装にかかった行数 20

3.1 コンパイルタイムメタオブジェクトプロトコルの動作 . . . 24
3.2 エディットタイムメタオブジェクトプロトコルの動作 . . . 25

4.1 plugin.xml を用いた拡張の定義 30
4.2 マニフェスト・エディター 31
4.3 JDT によるコンパイルの流れ 33
4.4 メタオブジェクトへ処理を委譲 35

5.1 2 進法表記のソース変換を行う convertSources メソッド . . 38
5.2 2 進法表記のエラー処理を行う finalizeProblems メソッド . 39
5.3 @Refines 拡張 40
5.4 Ref クラスによって拡張された HelloWorld クラス 41
5.5 CTUnit からクラス定義を取得 42
5.6 クラス定義の中から@Refines で注釈されたクラスを取得 . 43
5.7 部分木のコピー 43
5.8 ビジターパターンを用いた構文木の走査 44
5.9 Java 開発環境 44
5.10 本フレームワークを用いて拡張された統合開発環境 45
5.11 言語拡張に由来しないエラー 46

表 目 次

4.1 マニフェスト・エディター 32
4.2 主な ASTNode 36

5.1 メタオブジェクトの記述に必要となった行数 46

第1章　はじめに

近年、ソフトウェア開発の場において、言語拡張を行うためのシステムの需要が高まっている。例えば Java を用いてデータベースを利用するソフトウェアの開発を行うことを考える。このような場合、O/R マッピングを行うための独自の機構を言語拡張として追加したドメイン固有言語を作成することで、生産性の向上が期待できる。

また、オブジェクト指向やアスペクト指向といったプログラミングに対する新たな手法の導入も、既存の言語の拡張として開発される場合が多い。C++は C 言語を元にクラスなどの概念を追加した言語であり、AspectJ[9, 13] や GluonJ[3, 4] は Java 言語を元にアスペクト指向プログラミングを行うためのいくつかの新しい文法を定義した言語である。

このようにして新しく考えられた言語を実際に開発に用いるには、新しい言語のコンパイラやデバッガを作成する必要がある。また、同じ概念を実現するために考えられた新しい言語であっても、AspectJ と GluonJ のように実装は言語設計及び開発者によって異なるため、それぞれに対して個別にコンパイラを作成しなければならない。このように多岐に渡る拡張言語のコンパイラを作成するため、コンパイラを一から作成する以外にも、拡張しやすいように作られたコンパイラを元に拡張を行う手法や拡張前の言語の構文を用いて言語拡張を行うといった手法が採られている。

しかし、近年ソフトウェア開発の場においてコンパイラやデバッガなどを個別に利用することは少ない。従来テキストエディタ・コンパイラ・デバッガなどのソフトウェア開発に必要なツールが個別に行っていた処理を一つのインターフェース上で統合して扱えるようにした、統合開発環境を用いるのが一般的である。代表的な統合開発環境の例としては、Eclipse[14] や NetBeans[11]、Visual Studio[10] などが挙げられる。これらの統合開発環境では、ソフトウェアを構成するソースやリソースといった多くのファイルを、一つのプロジェクトとしてまとめて管理できる。また、テキストエディタをコンパイラやデバッガと連携させたことによって、編集しているソースコードを自動的に補完したり、ソースコード中のエラーをリアルタイムに通知したりといったことが可能になっている。このような統合開発環境を利用することにより、多くのソースやリソースを扱うような巨大で複雑なソフトウェア開発を行う際でも開発者の負担を低減すること

ができる。

　このような理由から、新しく考えられた言語に対してもこのような統合開発環境を作成することが望ましい。しかし 2 章で述べるように、コンパイラやデバッガに対して用いた既存の手法を統合開発環境に対してそのまま適用することは困難である。

　そこで本研究では、統合開発環境のための言語拡張システムを提案する。本フレームワークは、コンパイル時のソースコードの変換及び編集時のエラー操作を行うメタオブジェクトプロトコルを提供する。本フレームワークのユーザーは、言語拡張をメタオブジェクトとして記述することで、統合開発環境の扱う言語の拡張が可能となる。

　本稿の残りは、次のような構成からなっている。第 2 章は 既存の手法とその問題点について述べる。第 3 章では、本システムの特徴とその記述方法を、第 4 章では、本システムの実装方法を、第 5 章では、本システムの適用例を、そして第 6 章でまとめと今後の課題について述べる。

第2章　既存の手法とその問題点

この章では、統合開発環境の拡張に際して有用であると考えられる既存の手法と、その問題点について述べる。次の 2.1 節では AspectJ の統合開発環境である AJDT[5] がどのようにして実現されているかについて、2.2 節では拡張言語の統合開発環境の作成にあたって有用であると考えられる関連研究とその問題点について述べる。

2.1　AJDT

AJDT は、Java を拡張して作られたアスペクト指向言語である AspectJ の統合開発環境を Eclipse プラットフォーム上で提供するためのプラグインである。AJDT を用いることによって図 2.1 のように、AspectJ を用いて Eclipse 上でソフトウェア開発を行うことが可能となる。

2.1.1　アスペクト指向とは

アスペクト指向とは、従来のオブジェクト指向ではモジュール間にまたがってしまう処理（横断的関心事）をアスペクトと呼ばれるモジュールにまとめることによりうまくモジュール化するための技術である。アスペクト指向の主な概念を以下に記す。

- アスペクト
  横断的関心事をまとめたモジュール単位。ポイントカットとアドバイスを組み合わせることで他のモジュールの動作を変更可能。

- ポイントカット
  ジョインポイントの集合。アスペクトを織り込みたいポイントを条件で指定する。

- ジョインポイント
  アスペクトによって処理を追加・変更するための基点。代表的なものとして、メソッドやコンストラクタの呼び出し・実行時点、フィールドの参照や代入などがある。

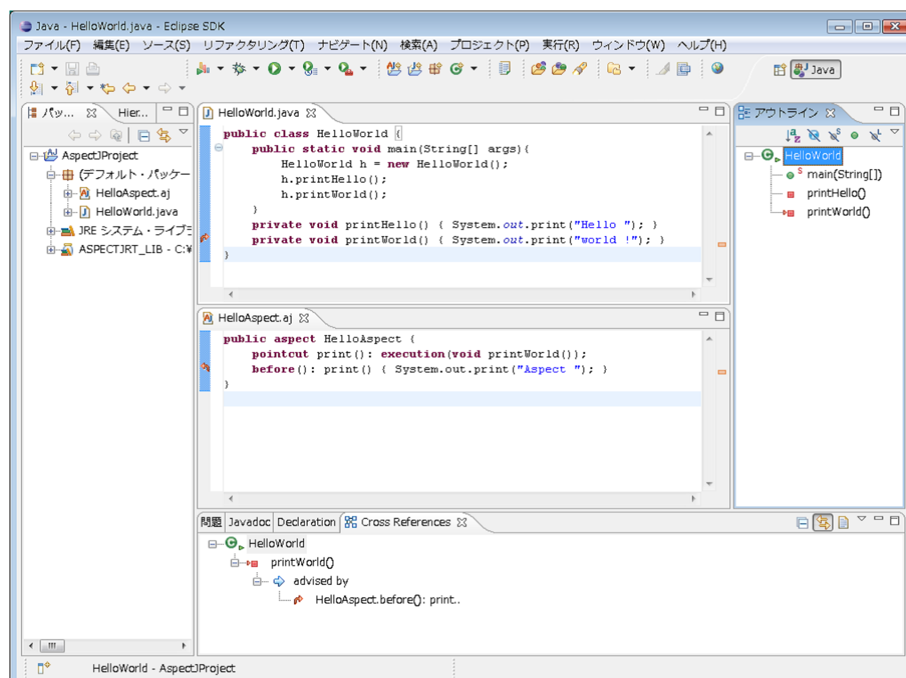

図 2.1: AJDT を用いた AspectJ ソフトウェア開発

- アドバイス
  ポイントカットに対し、追加・変更したい処理を記述したもの。
- 織り込み（weave）
  ポイントカットで指定されたジョインポイントの集合でアドバイスが実行されるように結び付けること。

2.1.2　AspectJ

AspectJ は Java 言語にアスペクト指向プログラミングを行うための機構を追加した言語実装の一つである。AspectJ を用いたプログラム例を以下に示す。

図 2.2 は AspectJ の文法を用いて記述されたプログラムである。1-10 行目に記述されている HelloWorld クラスは hello() メソッド呼び出して Hello world! と出力するクラスである。そして、9-13 行目が AspectJ で記述された HelloAspect アスペクトとなっている。10 行目の execution ポイントカットにより world() メソッドを実行するときが選択されており、そのポイントカットに対して 11 行目と 12 行目でそれぞれで before・after アドバイスが定義されている。

```
public class HelloWorld {
  public static void main(String[] args){
    HelloWorld h = new HelloWorld();
    h.hello();
  }

  private void hello() {
    System.out.println("Hello world!");
  }
}

public aspect HelloAspect {
  pointcut print(): execution(void hello());

  before(): print() {
    System.out.println("[Before hello() method]");
  }

  after(): print() {
    System.out.println("[After hello() method]");
  }
}
```

図 2.2: AspectJ を用いたプログラム例

HelloAspect アスペクトを織り込んで HelloWorld クラスを実行すると、

[Before hello() method]
Hello Aspect world !
[After hello() method]

と出力される。これは HelloAspect 内の before・after アドバイスにより、printWorld() メソッドを呼び出す直前と直後に HelloAspect 内で定義された処理が織り込まれた結果である。

2.1.3 AJDT

AJDT は AspectJ のための統合開発環境である。AJDT を用いることにより、より視覚的・直感的に AspectJ でのソフトウェア開発を行うこと

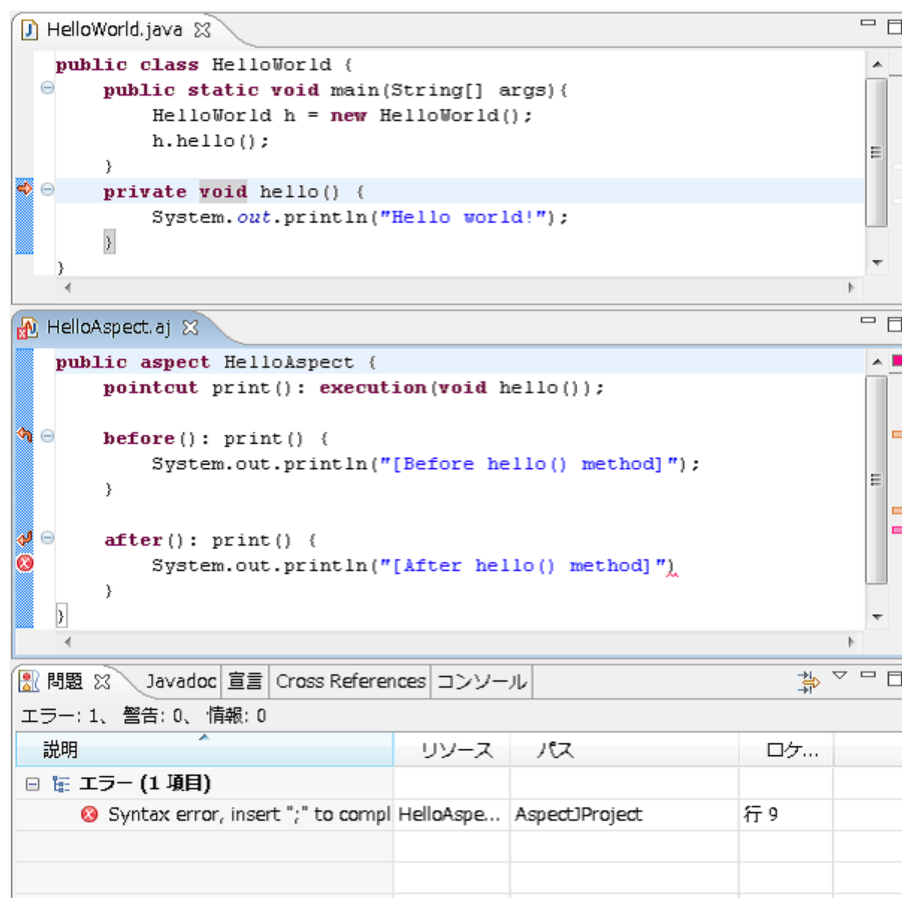

図 2.3: AJDT を用いた AspectJ のソースコード編集

ができる。図 2.3 は AJDT を用いて AspectJ のソースコードを編集しているところである。

AJDT を用いることにより、アスペクト内に記述されているアドバイスや、アスペクトが織り込まれるジョインポイントの位置などが矢印の形で表示されている。また、ソース中のエラーは編集中及びコンパイル時に発見され、ソースコード上の波線などにより報告される。ここでは、HelloAspect.aj の 9 行目にセミコロンが不足しているというエラーが報告されている。さらに、統合開発環境上から直接 HelloAspect アスペクトを織り込んで HelloWorld クラスを実行することも可能である。他にも、aspect や pointcut、execution といった AspectJ 独自のキーワードが色づけされて視覚的にソースコードを把握しやすくなっている。

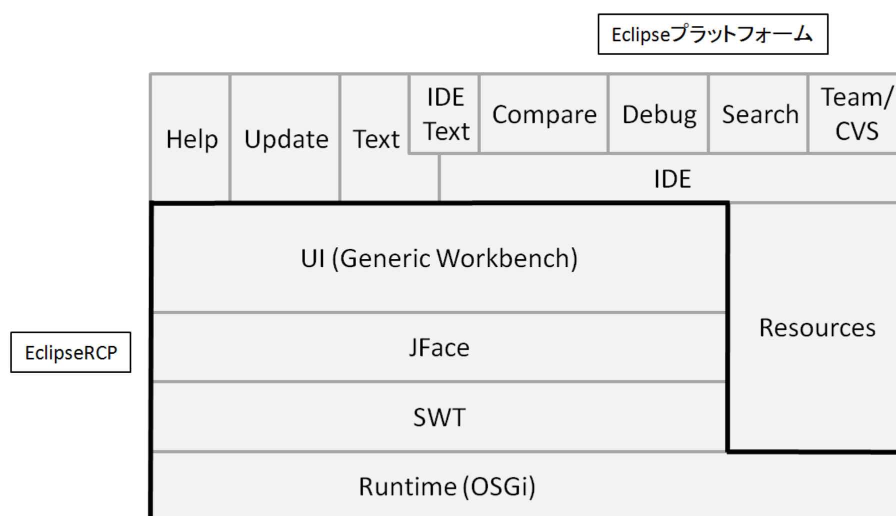

図 2.4: Eclipse のアーキテクチャ

AJDT の実装

AJDT は、JDT(Java Development Tools) を拡張することで Eclipse の拡張プラグインとして実現されている。JDT は Eclipse プラットフォームに標準搭載されている Java 用の統合開発環境である。

Eclipse のアーキテクチャは図 2.4 のように、根幹であるコアランタイムを多くのプラグインによって次々に拡張することで実現されていて、各プラグインが提供している拡張ポイントに新しいプラグインを接続することでさらなる拡張が可能である。拡張ポイントは、処理を追加するためのプログラム上の基点であり、プラグインが自分自身を他のプラグインから拡張可能にするために提供している。

AJDT も、Eclipse が提供している org.eclipse.core.resources.builders などの拡張ポイントに自身の処理をプラグインとして接続することで実現されている。

AJDT の手法の問題点

しかし、Eclipse のプラグインとして統合開発環境を作成する方法にはいくつかの問題点がある。

まず第一に、Eclipse の拡張プラグイン開発は非常に困難である。拡張言語の統合開発環境を Eclipse の拡張プラグインとして作成するためには、Java・コンパイラなどの知識に加えてプラグイン開発を行うための知識が必要である。

また、Eclipse が提供している膨大な拡張ポイントの中から自分が利用したい拡張ポイントを探すのは困難であり、場合によっては適当な拡張ポイントが存在しない場合もあり得る。例えば、Java を拡張した言語の統合開発環境について Eclipse プラグインでの実装を考えた場合、標準の Java 開発環境である JDT を拡張して作成することになる。しかし、JDT にはソースコードの字句解析中や構文解析中に処理を追加するような拡張ポイントは用意されていないため、他の org.eclipse.core.resources.builders などの拡張ポイントを利用することになる。これは、Eclipse に対して新たなコンパイラを追加するための拡張ポイントである。そのため、標準の Java に対して構文を一つ追加したいだけであっても、独自の builder を作成して処理を追加しなければならない。また文法を動的に変更することも不可能である。これは、JDT のコンパイラを実行時に再構築/再ロードする仕組みが用意されていないためである。

さらに、言語の拡張を行う場合一般的には字句解析器及び構文解析器を変更する必要がある。JDT では編集中にもパースを行いエラーやキーワードを逐次報告するため、少しでも動作を速くするためにコンパイラに対して Java に特化した最適化を行っているため、言語自体に関して新たな拡張が困難である。そのため拡張ポイントを用いて既存の JDT の字句解析器や構文解析器に処理を加えることができず、新しい言語の統合開発環境を作成する場合は字句解析器や構文解析器を独自に用意するか、拡張ポイントを使わずに JDT の字句解析器及び構文解析器を直接変更する方法を採ることになる。AJDT の場合は前者の方法を採用しているが、元の言語に対して変更が少ない場合、新しく字句解析器及び構文解析器を用意するのは無駄が大きい。また後者の方法では JDT を直接変更してしまうため、字句解析器や構文解析器に変更を加えるたびに Eclipse 全体を再ビルドしなければならない。

これらの理由から、拡張された言語の統合開発環境を Eclipse のプラグインとして開発するのには大きな労力が必要である。AJDT のように言語の仕様が確立されていて、大きなプロジェクトとして統合開発環境を作成する場合には Eclipse の提供する基盤が有用であると言えるが、小さなプロジェクトや個人単位で言語拡張を行いたい場合に新しい言語の統合開発環境を作成する手段としてはあまり現実的ではないと考えられる。

2.2　関連研究とその問題点

2.2.1　GluonJ

GluonJ は、Java をベースとしたアスペクト指向言語の、AspectJ とは異なる実装である。GluonJ では、AspectJ のように aspect などの新しい

文法を用いるのではなく、既存の Java の文法であるアノテーションを用いることによってアスペクトを実現している。

図 2.2 で扱った HelloAspect を GluonJ の文法を用いて記述すると図 2.6 のようになる。

```
1 import javassist.gluonj.*;
2
3 @Glue class HelloAspect {
4   @Before("{ System.out.println(`[Before hello() method]`); }")
5   @After("{ System.out.println(`[After hello() method]`); }")
6   Pointcut pc = Pcd.call("HelloWorld#hello()");
7 }
```

図 2.5: GluonJ を用いて記述したアスペクト例

@Glue アノテーションで修飾されたクラスが GluonJ におけるアスペクトにあたる、@Glue クラスである。ここでは 3-7 行目で HelloAspect という@Glue クラスを定義している。HelloAspect クラスの内部では、6 行目で、HelloWorld の hello() メソッドを呼ぶときをポイントカット pc として定義している。このポイントカットは 4,5 行目で@Before・@After アノテーションによって注釈されていて、それぞれアノテーションの引数として文字列を出力する命令が文字列で渡されている。GluonJ ではこのような文法で、HelloWorld の hello() メソッドを実行する直前及び直後に処理を追加するアスペクトを記述する。

GluonJ では AspectJ のようにコンパイル時にアスペクトを織り込むのではなく、プログラムの実行前、またはクラスファイルのロード時にアスペクトを織り込む手法を採用している。前者の実行前に織り込みを行う手法では、この@Glue クラスを織り込んで図 2.2 の HelloWorld クラスを実行するにはコマンドラインから

java -jar gluonj.jar HelloAspect HelloWorld.class

のようにすればよい。このようにして実行することで、HelloWorld.class の実行前に HelloAspect が織り込まれ、

[Before hello() method]
Hello Aspect world !
[After hello() method]

と出力される。

GluonJ では、追加されるキーワードはアノテーションの形で提供され

ているため、GluonJ を用いて記述されたプログラムは従来の Java 用のコンパイラや統合開発環境においてエラーとならない。これは GluonJ が Java の拡張言語でありながら構文の拡張を行っていないことに由来する。このため、GluonJ を用いて記述されたプログラムは従来の Java 用のコンパイラを用いてコンパイルを行うことができる。また統合開発環境から、例えば Eclipse から GluonJ の@Glue クラスを織り込んで実行を行うには、実行構成から JVM の引数に

```
java -javaagent:gluonj.jar=HelloAspect HelloWorld.class
```

のように渡してやればよい。このようにすることで Eclipse がクラスファイルをロードするときに HelloAspect を@Glue クラスとして織り込んで HelloWorld.class を実行することができる。

GluonJ の手法の問題点

しかし、Java 用の開発環境はあくまでも Java を用いたソフトウェア開発を目的に作られているため、GluonJ を用いた開発を行うには不満な点がいくつか存在する。

例えば前述の GluonJ におけるポイントカットの記述例では、@Before・@After アノテーションに文字列を引数として渡すことで織り込みたい処理を記述しているが、この引数は Java 用の開発環境では文字列としてしか扱われないため文法のチェックなどは行われない。そのため、@Before アノテーションに文字列型の引数として渡されている処理の中にタイプミスなどが存在しても、@Glue クラスのコンパイル時にはそのエラーは発見されず、プログラムの実行前やクラスのロード時に@Glue クラスを織り込む際になって初めてエラーとして報告される。図 2.6 ではカーソルの位置に; が不足しているが、このようなエラーは HelloAspect.java のコンパイル時には発見されない。

GluonJ のための統合開発環境であれば、このような@Glue クラスはコンパイル時にエラーとして処理されるべきである。またソースコードの編集中にも、@Before・@After アノテーションの引数をパースすることによりキーワードを色づけしたりエラーを報告したりといった処理が行われることが望ましい。図 2.6 では、@Before・@After アノテーションの引数は文字列として処理されているため全て青文字で表示されてしまっている。他にも、AJDT のようにアスペクトが織り込まれる位置などが図示する機能なども考えられる。

また、GluonJ ではアノテーションを用いて言語の拡張を行っているが、これらのアノテーション名が他のアノテーションと競合してしまう可能性もある。さらに、アノテーションは Java SE 5 から導入された機能で

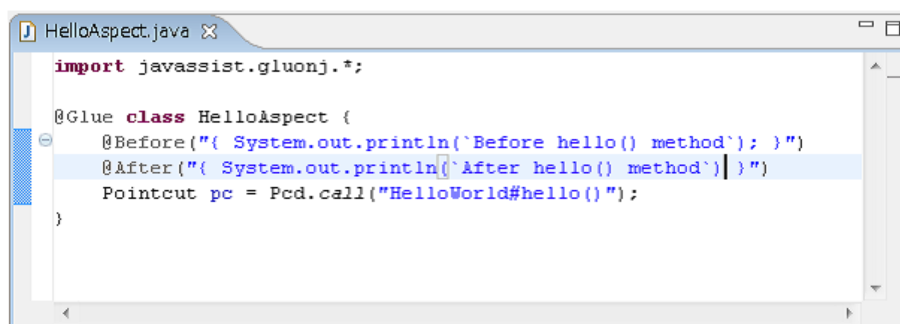

図 2.6: Eclipse を用いた GluonJ のソースコード編集

あり、それ以前の Java や他の C などの言語においては、言語拡張の際に GluonJ のような手法をとることはできない。またこの手法では元の言語の文法を拡張しないため、新しいキーワードや文法を導入するような言語拡張には対応できない。

このような問題点が存在するため、GluonJ のように拡張言語の設計を工夫するだけでは、統合開発環境を新たに作成する必要性がなくなるということにはならない。

2.2.2　extensible compiler

コンパイラを作成する場合、ソースコードを構文木に変換するために字句解析や構文解析を行うフロントエンド、及び最適化やコード生成を行うバックエンドを作成する必要がある。字句解析や構文解析を行うプログラムを手作業で生成するのは困難なため、それぞれ lex や yacc といった生成系を用いるのが一般的である。このようなコンパイラを生成するための生成系はコンパイラコンパイラと呼ばれている。

一部のコンパイラでは、対象言語の拡張を想定されたフロントエンド及びバックエンド用いることにより拡張された言語のコンパイラの生成を容易にしている。これらのコンパイラを拡張して実装されたコンパイラの例としては、AspectJ のコンパイラの１つである AspectBench Compiler[1] などが挙げられる。

AspectBench Compiler

AspectBench Compiler は AspectJ のコンパイラの１つであり、Polyglot[12] と Soot[15] という二つのフレームワークを利用して構築されている。

Polyglot はデザインパターンに基づいて構築された Java コンパイラフロントエンドフレームワークであり、構文解析器生成系として文法拡張が容易な PPG を用いている。Polyglot は Java 1.4 のためのコンパイラであるが、Polyglot を拡張することで図 2.7 のように Java 1.5 や AspectJ、及びその他の拡張言語のコンパイラが実現されている。

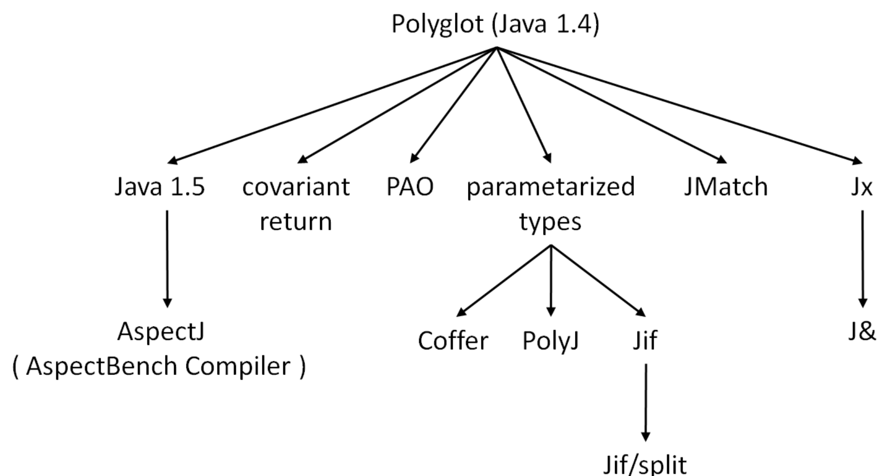

図 2.7: Polyglot を用いてコンパイラが実現されている言語

Soot は Java バイトコードの最適化などを行うためのフレームワークであり、Java バイトコードを一度 Jimple などの中間表現に変換後構文木の操作を行うことでバイトコードの変換を行うことができる。AspectBench Compiler では、Soot を用いることによりアスペクトの織り込みを実現している。

これらのフレームワークを利用することにより、AspectBench Compiler は拡張が容易なフロントエンド及びバックエンドを提供している。

JastAddJ

JastAddJ（JastAdd Extensible Java Compiler）は、JastAdd[7] というコンパイラ生成系を用いて生成された Java コンパイラである。JastAdd は、構文拡張をプラグインとして追加可能なコンパイラを生成するためのコンパイラコンパイラである。JastAdd では、拡張された言語と元の言語との差分のみを別ファイルに定義することで、拡張された言語のコンパイラを生成することができる。また JastAdd は字句解析及び構文解析に任意の生成系を用いることが可能である。前項で述べた AspectBench

Compiler は、JastAdd を用いた実装も行われている。

JastAddJ は Java 1.4 及び Java 1.5 のためのコンパイラであり、Java 1.5 のコンパイラは Java 1.4 に対する差分として実装されている。Java 1.5 のコンパイラは JastAdd を用いることで、フロントエンドの拡張には 4600 行、バックエンドの拡張には 1100 行と非常に少ない行数で実装されている。また、non-null アノテーションを用いてオブジェクトへの参照が null にならないようにする non-null checker の機構を追加したコンパイラを生成する際も、図 2.8 のように非常に少ない行数で実装が可能となっている。

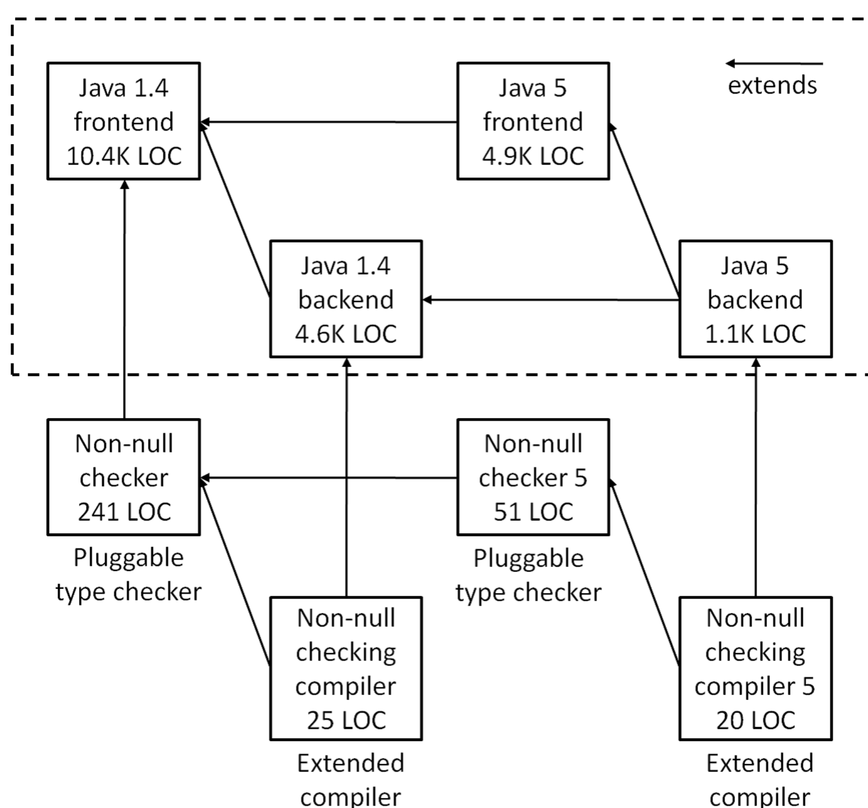

図 2.8: Java コンパイラ及び Non-null checker の機構の JastAdd を用いた実装にかかった行数

Polyglot や JastAdd の手法の問題点

しかし、統合開発環境上で用いられているコンパイラは対象言語の拡張を想定されて作られたものではない。Eclipse の Java 開発環境である JDT

のコンパイラは、さまざまな最適化を行っているためにこのような拡張性には乏しい。Eclipse であれば専用のプラグインを作成することによりこれらのコンパイラを使用した統合開発環境を作成することができるが、現在そのようなプラグインは存在しない。

また統合開発環境上では、ソースコードの編集時にもリアルタイムに構文解析を行いエラーなどを報告する必要があるため、Soot のように構文木の変換を行う手法は構文解析に要する時間を増加させてしまい適当でないと考えられる。

第3章　設計

本研究では、拡張されたプログラミング言語の統合開発環境作成を支援するフレームワークの開発を行った。本フレームワークはコンパイル時のソース変換と編集時のエラー除去及び付加を行うための、コンパイルタイム及びエディットタイムのメタオブジェクトプロトコルを提供することにより、拡張言語に適合した統合開発環境の作成を支援する。3 章では、本研究の設計について述べる。

3.1　メタオブジェクトプロトコル

メタオブジェクトプロトコルとは、クラスやメソッドといった、オブジェクトを扱う枠組みそのものをオブジェクトとして扱う仕組みのことである。これらのオブジェクトをメタオブジェクトと呼び、メタオブジェクトのクラスをメタクラスと呼ぶ。この概念は、Kiczales らの The Art of the Metaobject Protocol[8] によって Common Lisp のオブジェクトシステムに採り入れられたものである。

メタオブジェクトプロトコルを用いることによって、クラスやメソッドの挙動などのプログラム言語の仕様をその言語自身で記述及び拡張することが可能である。例として Java 言語では、クラスからフィールドやメソッドなどの情報を取得可能にするリフレクション API などにおいてメタオブジェクトプロトコルの技術が使用されている。

コンパイルタイムメタオブジェクトプロトコル

The Art of the Metaobject Protocol で用いられているメタオブジェクトプロトコルは、プログラムの実行中に動作するランタイムメタオブジェクトプロトコルである。これに対し、コンパイルタイムメタオブジェクトプロトコルはプログラムのコンパイル時に動作を行う。

従来のメタオブジェクトプロトコルは Common Lisp のようなインタプリタ言語に対してオブジェクトシステムの振舞いを変更するための技術であったが、コンパイル時にメタオブジェクトプロトコルを動作させることによりコンパイラ言語においてもメタオブジェクトプロトコルの恩

恵に与ることが可能となる。コンパイルタイムメタオブジェクトプロトコルを用いた例としては、C++にリフレクションなどの機能を追加したOpenC++[2] などが挙げられる。

エディットタイムメタオブジェクトプロトコル

エディットタイムメタオブジェクトプロトコルは、統合開発環境においてソースコードなどの編集時に動作を行うメタオブジェクトプロトコルである。エディットタイムメタオブジェクトプロトコルを用いた例としては、ソースコードの見た目の拡張を行った presentation extension[6] の研究などが挙げられる。

3.2　本フレームワークの設計

本フレームワークは、コンパイル時及びエディット時の 2 つのメタオブジェクトプロトコルから構成される。コンパイル時にはメタオブジェクトの記述に従って、拡張された言語で記述されたソースコードを既存の言語のソースコードに変換する。エディット時にはソースコードの変換は行わず、メタオブジェクトの記述に従ってユーザーに提示するエラーの操作を行う。

本フレームワークを用いることにより、コンパイル時とエディット時に行うメタプログラムをメタクラスとして外部ファイルに記述することで、元の統合開発環境の処理を変更し、拡張言語に適合した統合開発環境を作成することが可能である。

これらのコンパイルタイム及びエディットタイムのメタオブジェクトプロトコルの設計について、それぞれ 3.2.1 小節及び 3.2.2 小節で記述する。

3.2.1　コンパイルタイムメタオブジェクトプロトコル

本フレームワークは、コンパイル時にメタオブジェクトによる処理の挿入を行う。メタオブジェクト内に、拡張された言語によるソースコードから既存の言語によるソースコードへと変換する処理を記述することで、拡張された言語によるソースコードを図 3.1 のようにコンパイルし、バイトコードを出力することが可能である。

本フレームワークを用いて統合開発環境を用いてソースファイルのコンパイルを行う場合、まず本フレームワークのコンパイルタイムメタオブジェクトプロトコルによる処理を行う。コンパイルタイムメタオブジェクトプロトコルでは次のように処理が行われる。

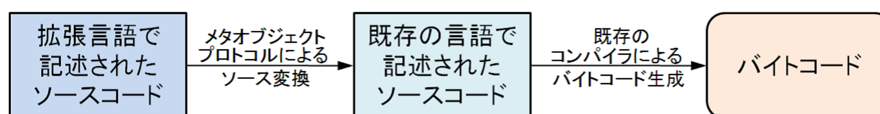

図 3.1: コンパイルタイムメタオブジェクトプロトコルの動作

1. コンパイルを行うすべてのソースファイルについて、既存の字句解析器及び構文解析器を用いて抽象構文木を生成する。
2. ソースコードと抽象構文木をメタオブジェクトに渡す。
3. メタオブジェクトによりソースコード変換を行う。変換には次の 2 種類の方法を提供する。
    - ソースコードを構文解析して生成された抽象構文木の変換
    - ソースコードの文字列としての変換

拡張された言語によるソースコードに対して用いる構文解析器は、拡張された言語に対応したものではなく既存の言語の構文解析器を用いる。そのため、拡張された言語に構文が追加されていた場合、生成される抽象構文木は不完全である可能性があるが、メタオブジェクトに対して抽象構文木とともに元のソースコードを渡すことで抽象構文木を用いない変換を可能とし、抽象構文木が不完全であってもメタオブジェクトによる変換を可能としている。

このようにしてコンパイルタイムメタオブジェクトプロトコルによるソースコード変換を行った後、図 3.1 のように既存のコンパイラを用いてコンパイルを行いバイトコードを出力する。

3.2.2　エディットタイムメタオブジェクトプロトコル

本フレームワークは、ソースコードの編集時にメタオブジェクトによる処理の挿入を行う。メタオブジェクト内に、既存のコンパイラを用いて発見されたエラーに対して操作を行う処理を記述することで、図 3.2 のように発見されたエラーの一部を除去または変更、もしくは新たなエラーを追加し、実際に報告するエラーを操作することが可能である。

本フレームワークを用いてソースファイルの編集を行う際には、まず統合開発環境に含まれる既存の言語のコンパイラによる字句解析や構文解析・型検査などが行われ、構文エラーや型エラー、及び警告が発見される。この際拡張された言語を用いて記述されたソースコードを既存の言語

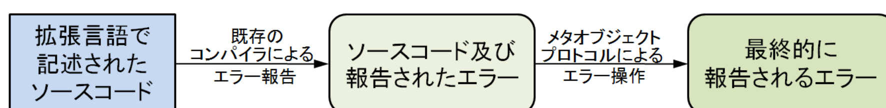

図 3.2: エディットタイムメタオブジェクトプロトコルの動作

のコンパイラで検査するため、ソースコード中に拡張された文法が存在した場合、構文エラーとして発見される。

このようにして発見されたエラー及び警告について、本フレームワークのエディットタイムメタオブジェクトプロトコルによる操作を行う。エディットタイムメタオブジェクトプロトコルでは次のように処理が行われる。

1. 発見されたエラー及び警告と、それらを含むソースコードをメタオブジェクトに渡す。
2. メタオブジェクトによりエラー及び警告の除去・変更・追加を行う。

このようにエラー及び警告の除去だけでなく変更・追加をも可能なことにより、構文を追加するような文法拡張だけでなく、元の言語の機能の一部を制限するような文法拡張にも対応が可能である。

3.3　例

ここでは Java 言語を元に言語拡張を行うことを考える。

Java 言語では数値を 8 進法及び 16 進法で表記するための記法が提供されている。数値の先頭に 0 を付加することで 8 進法を、数値の先頭に 0x または 0X を付加することで 16 進法をそれぞれ表す。例えば、

```
int x = 0124;
int x = 0x64;
```

はいずれも

```
int x = 100;
```

と同じ意味である。これは 8 進法の 124、及び 16 進法の 64 がいずれも 10 進法で 100 を表すためである。

しかし、Java では数値を 2 進法で表記するための記法は提供されていない。Perl などのいくつかの言語では、

$x = 0b1100100;

のように数値の先頭に 0b を付加することで 2 進法による表記が可能であるが、Java ではこのような表記法は提供されていない。そこで、このような表記法を Java の言語拡張として実現することを考える。

一般的にこのような言語拡張を行うためには、字句解析器や構文解析器を修正する必要がある。例えば Eclipse の Java 開発環境である JDT のコンパイラでは、字句解析の際に 0124 や 0x64 を数値を表すトークンとして評価し、構文解析時に 10 進法での値を算出することによって 8 進法や 16 進法による表記を実現している。そのため新たに 2 進数での表記を可能とするためには構文解析器および JDT プラグインの変更及び再コンパイルが必要となり、非常に困難である。また、オープンソースでないコンパイラや統合開発環境においては、このような言語拡張を行うことは不可能である。

このような言語拡張は、本フレームワークを用いることにより非常に容易に実現可能である。以下に、本フレームワークを用いてコンパイル時及びエディット時に処理を行うことでこのような言語拡張の実現が容易であることを示す。

3.3.1　コンパイル時

字句解析器や構文解析器の変更を伴わない素朴な実現方法として、コンパイルの直前に以下の文字列変換を行うことが考えられる。

1. ソースコード中の、0[bB][01]+にマッチする文字列を抽出する。
2. 抽出された文字列を 10 進数に変換する。

ここで 0[bB][01]+は正規表現であり、[] は括弧内の任意の一字を、+は直前の文字の 1 回以上の繰り返しを意味する。コンパイル前にこのような文字列変換を行うことで、

int x = 0b1100100;

というソースコードは

int x = 100;

と変換されるため、既存のコンパイラによるコンパイルが可能である。本フレームワークではこのような文字列変換を行うメタオブジェクトを記述することで、コンパイル前にソース変換を行うことができる。

このような実現方法は、ソースコード中に 0[bB][01]+にマッチする文字列やクラス名などが存在した場合にも変換を行ってしまうため、言語仕様としては適当ではない。しかし文字列やクラス名がこのような部分列を含むことは稀であり、実用上大きな問題はないと考えられる。

また抽象構文木を用いて変換を行ったり、外部の構文解析器を用いて変換を行ったりすることで、利用者の必要に応じてより正確な実装も可能である。さらにこれらの実装はメタオブジェクトとしてコンパイラの外部に記述されるため、メタオブジェクトを変更することにより、文法定義の動的な変更も可能である。

3.3.2　エディット時

エディット時には 3.3.1 小節のようなソース変換は行わない。そのため

```
int x = 0b1100100;
```

というソースコードは Java の構文として構文解析され、b1100100 の部分がエラーが報告される。

本フレームワークのエディットタイムメタオブジェクトプロトコルを利用することにより、このような言語拡張に由来するエラーの除去をが可能である。例えば先に挙げた例の場合、

1. ソースコード中のすべてのエラーを抽出する。
2. エラーの報告されている文字列が [bB][01]+にマッチする場合、そのエラーを除去する。

という処理を行うメタオブジェクトを記述することで、言語拡張に由来するエラーの除去が可能である。また、より精密なエラーの評価が必要な場合には、コンパイル時と同様にソース変換を行うことで不必要なエラーを判別することも可能である。

3.4　本フレームワークの設計のまとめ

本フレームワークを用いることで統合開発環境自体に変更を加えることなく拡張された文法に対応させることができる。本フレームワークのコンパイルタイムメタオブジェクトプロトコルを用いることで、拡張された言語のコンパイル及び実行を、統合開発環境上で行うことが可能である。またエディットタイムメタオブジェクトプロトコルにより、ソースコードの編集中に余分なエラーを除去・変更し、必要なエラーを付加することが

可能である。さらに、これらの拡張はメタオブジェクトとして記述するため、コンパイル時及び編集時に文法を動的に変更することが可能である。

3.4.1 既存の手法の問題点に対する考察

2 章で述べた既存の手法における問題点について考察を行う。

2.1 節で挙げた AJDT のような実装方法には、

- 統合開発環境の拡張が困難
- 文法を動的に変更することが不可能

という問題が存在した。
本フレームワークを用いることにより 3.3 節のように統合開発環境を容易に拡張可能であり、また動的な文法の変更も可能である。

2.2.1 小節で挙げた GluonJ のような実現方法には、

- アノテーションを用いない言語拡張が不可能
- アノテーションの存在しない言語では実現不可能

という問題が存在した。
本フレームを用いることにより 3.3 節のようにアノテーションを用いない言語拡張が可能であり、またアノテーションの存在しない言語においても同様の手法による言語拡張が可能である。

2.2.2 小節で挙げた Polyglot や JastAdd のような手法には、

- 統合開発環境のコンパイラにおいて言語拡張は想定外
- 構文木の変換は解析時間の増大につながるため編集時には利用困難

という問題が存在した。
本フレームワークを用いることにより 3.3 節のように統合開発環境の扱う言語の拡張が可能である。またエディットタイムメタオブジェクトプロトコルではソース変換を行わないため解析時間を抑えることが可能である。

第4章　実装

本研究の実装は、Eclipse の JDT を拡張することにより行った。この章では本フレームワークの実装方法について述べる。

4.1　Eclipse の拡張

Eclipse は Java を用いて開発された、オープンソースのプラットフォームである。Eclipse は Java 向けの統合開発環境として有名だが、Java の統合開発環境としての機能は Eclipse SDK に標準で含まれる JDT によって提供されているものである。JDT はいくつかのプラグインを Eclipse の提供している拡張ポイントに接続することにより実現されている。Eclipse は Java の統合開発環境としてだけではなく、Eclipse 及び追加プラグインが提供している拡張ポイントに新たなプラグインを接続することで機能の追加を行うことが可能である。

Eclipse には、プラグインの開発環境として PDE（Plugin Development Environment）が用意されている。本研究の実装もこれを用いて行った。PDE を用いることにより、プラグインを開発するために必要なファイルの管理や拡張ポイントの管理などが容易になる。

4.1.1　マニフェスト・ファイル

プラグインを開発するためには、まずマニフェスト・ファイルを記述する必要がある。Eclipse 3.1 以降のプラグインには、通常 plugin.xml と MANIFEST.MF という 2 つのマニフェスト・ファイルが含まれている。plugin.xml はプラグイン・マニフェスト・ファイルと呼ばれ、プラグインの拡張および拡張ポイントの定義などを XML を用いて図 4.1 のように記述する。MANIFEST.MF は、Eclipse 3.0 からプラグインを管理するための導入された OSGi のためのマニフェスト・ファイルで、プラグインの ID や名称・作成者、他のプラグインとの依存関係などについて記述する。

図 4.1 は JDT が Eclipse に新たなビルダーを定義しているところである。extension 節の point 属性で拡張を接続する拡張ポイントを、id で拡張を識別するための ID を、name に拡張の名前を定義する。ここで name 節

```
1  <plugin>
2    <extension
3      point="org.eclipse.core.resources.builders"
4      id="javabuilder"
5      name="%javaBuilderName">
6      <builder>
7      <run class="org.eclipse.jdt.internal.core.builder.JavaBuilder">
8      </run>
9      </builder>
10   </extension>
11 </plugin>
```

図 4.1: plugin.xml を用いた拡張の定義

の% は変数を利用することを表し、javaBuilderName は plugin.properties ファイル内で

javaBuilderName=JavaBuilder

のように定義されている。このように extension 節を定義した後に、拡張ポイントごとの定義を記述する。ここでは run class 属性として JavaBuilder クラスを登録することで、Eclipse に新たなコンパイラを追加している。

4.1.2　マニフェスト・エディター

図 4.1 のようにマニフェスト・ファイルを直接編集することは困難である。そのため PDE ではマニフェストの編集を行うために、図 4.2 のようなマニフェストエディターを提供している。マニフェスト・エディターを用いることによりこれらのマニフェスト・ファイルの編集が容易になる。

マニフェスト・エディターを用いてマニフェスト・ファイルを開くと図 4.2 のように 9 つのタブが表示される。それぞれのタブでは表 4.1 のような編集を行うことが可能である。

4.2　本フレームワークの実装

本研究の実装は、JDT のソースコードを直接変更することにより行った。一般的な、提供されている拡張ポイントに新たなプラグインを接続して Eclipse の拡張を行う方法を採らなかったのは、2.1 章で述べたように JDT にコンパイラの処理を変更するための拡張ポイントが用意されていなかったためである。

図 4.2: マニフェスト・エディター

JDT は org.eclipse.jdt.core や org.eclipse.jdt.ui といったいくつかのプラグインから構成されている。本研究では JDT のコンパイル中に処理を挿入するため、org.eclipse.jdt.core プラグインを変更することでフレームワークの実装を行った。

4.2.1　JDT のコンパイラの動作

JDT では、ソースファイルの保存時にコンパイルを行う。JDT は 4.1 に記したとおり org.eclipse.core.resources.builders 拡張ポイントに org.eclipse .jdt.internal.core.builder.JavaBuilder を登録しているため、JDT を用いてコンパイルを行うと org. eclipse.jdt.internal.core.builder.JavaBuilder #build(int, Map, IProgressMonitor) メソッドが呼ばれる。

JavaBuilder#build メソッドではフルコンパイルか部分コンパイルかの確認を行い、部分ビルドの場合はコンパイルを行うべきソースファイルのみコンパイラに渡す処理を行っている。しかし本フレームワークを用いる場合、あるソースファイルのコンパイル時に別のソースファイルのソース変換及び再コンパイルを行いたい場合が考えられるため、JavaBuilder#build メソッドで常に JavaBuilder#buildAll() メソッドを呼び、フルコンパイルを行うように変更を行った。

フルコンパイルを行う場合、BatchImageBuilder#build() メソッドでコンパイルすべきソースファイルを収集した後、org.eclipse.internal.compiler. Compiler#internalBeginToCompile(ICompilationUnit[], int) で実際のコンパイルのための準備が行われる。ICompilationUnit は Java プロジェク

ページ	説明
概要	プラグインの ID や名称などのプラグイン全体の設定、及び作成中のプラグインを読み込んだ Eclipse ワークベンチの起動や配布用アーカイブの作成などを行う。
依存関係	作成中のプラグインを実行するために必要なプラグインを指定する。
ランタイム	作成中プラグインが別のプラグインに公開するパッケージ、及びその可視性を設定する。
拡張	作成中のプラグインがどの拡張ポイントを用いて拡張を行うかについて設定する。
拡張ポイント	作成中のプラグインが他のプラグインに対して提供する拡張ポイントの定義を行う。
ビルド	ビルド時にアーカイブに含めるファイルやディレクトリの設定を行う。
MANIFEST.MF	MANIFEST.MF のソースを表示する。
plugin.xml	plugin.xml のソースを表示する。
build.properties	build.xml のソースを表示する。

表 4.1: マニフェスト・エディター

トに含まれる Java ファイルを表すクラスであり、internalBeginToCompile メソッドには引数としてコンパイルを行うべきソースファイルの配列が渡される。本フレームワークを用いた場合は常にフルコンパイルを行うため、Java プロジェクトに含まれるすべてのソースファイルが引数として渡される。

internalBeginToCompile メソッドでは各ソースファイルの字句解析及び構文解析が行われる。解析が行われたソースファイルは CompilationUnitDeclaration 型のインスタンスとなり、Compiler#process(CompilationUnitDeclaration, int) メソッドが CompilationUnitDeclaration の resolve メソッドや finalizeProblems メソッドなどを呼び出すことで型チェックなどの残りの処理が行われる。resolve は名前解決を行うためのメソッドであり、finalizeProblems メソッドは必要ない警告を除去するためのメソッドである。これは@SuppressWarning アノテーションを処理するために用意されている。

以上のコンパイルの流れをまとめると図 4.3 のようになる。

本フレームワークでは Compiler#internalBeginToCompile メソッドの最

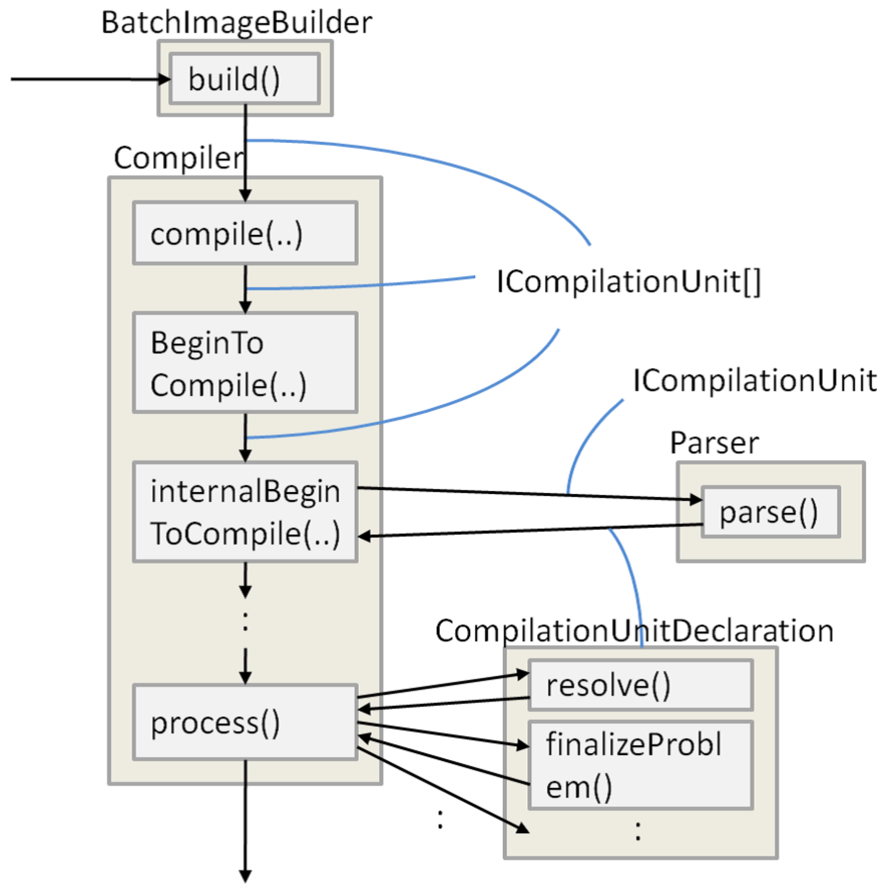

図 4.3: JDT によるコンパイルの流れ

初に処理を挿入することでコンパイル時にソースファイルの変換を行うメタオブジェクトプロトコルを、CompilationUnitDeclaration#finalizeProblems メソッドの最初に処理を挿入することで編集時にエラーの操作を行うメタオブジェクトプロトコルを実現した。

4.2.2　メタオブジェクトプロトコルの構成

コンパイルタイム及びエディットタイムのメタオブジェクトプロトコルの実装は、org.eclipse.internal.compiler.ctmop パッケージとして行った。ctmop パッケージは以下の 3 つのクラスから構成されている。

- CTMOP クラス
  このクラスはメタオブジェクトプロトコルを表すクラスである。JDT

によるコンパイル時及びエディット時に、このクラスを用いてメタオブジェクトに処理を委譲する。またメタオブジェクトがソースファイルを受け取るためのクラスプールの役割も兼ねている。

CTMOP クラスは static メソッドとして以下のメソッドを持つ。

  - void convertSources(List<SourceFile>)
    コンパイル時にメタオブジェクトに処理を委譲するためのメソッド
  - void finalizeProblem(CompilationUnitDeclaration)
    エディット時にエラーの操作を行うためのメソッド
  - List<CTUnit> getUnits()
    及びメタオブジェクトがソースファイルを受け取るためのメソッド

- CTUnit クラス
  このクラスは、本フレームワークにおけるソースファイルを表すクラスである。本フレームワークではソースコードとその抽象構文木を併せて扱うために、JDT から渡された ICompilationUnit の配列を CTUnit のリストへと変換する。メタオブジェクトはこのクラスのインスタンスを受け取り変更を加えることで、コンパイル時にソース変換を行うことが可能である。

  CTUnit クラスはソースコード及び抽象構文木を保持するための以下のフィールド、及びそれらを操作するためのメソッドを持つ。

  - SourceFile source;
    ソースファイルを保持するためのフィールド
  - CompilationUnit ast;
    抽象構文木を保持するためのフィールド
  - CompilationUnit getCompilationUnit()
    抽象構文木を得るためのメソッド
  - char[] getContents()
    ソースコードを得るためのメソッド
  - void setContents(String)
    ソースコードを変更するためのメソッド

- CompileTimeMetaobject クラス
  このクラスは、実行時に適用するメタオブジェクトのためのクラス

である。このクラスを実装したクラスのインスタンスがメタオブジェクトとなる。

CompileTimeMetaobject クラスは以下の抽象メソッドを持つ。拡張言語の統合開発環境の開発者はこれらのメソッドの実体を記述することでメタオブジェクトの実装を行う。

- void convertSources()
  コンパイル時にソース変換を行う
- void finalizeProblems()
  エディット時にエラーの操作を行う

これらのクラスは拡張された JDT アーカイブに含まれ、外部に対して公開される。ctmop パッケージの公開は、マニフェスト・エディターのランタイムタブから行った。

4.2.3 メタオブジェクトのロード

CTMOP クラスの convertSources メソッド及び finalizeProblem メソッドではクラスローダーを用いて各メタクラスをロードし、インスタンスを生成した後各メタオブジェクトに処理を委譲している。現在の実装では、プロジェクトのトップフォルダに置かれた CompileTimeMetaobject を継承したクラスのインスタンスを、実行時に適用するメタオブジェクトとしてロードしている。

メタオブジェクトのロードは、クラスローダーを利用して図 4.4 のように行っている。

```
1  URLClassLoader cmoLoader = new URLClassLoader(
2    cmoFolder, new CTMOP().getClass().getClassLoader());
3  for(String cmoFileName : cmoFileNamesList){
4    try {
5      CompileTimeMetaobject cmo
6        = (CompileTimeMetaobject)Class.forName(
7          cmoFileName, true, cmoLoader).newInstance();
8      cmo.convertSources();
9    } catch(Exception e) { e.printStackTrace(); }
10 }
```

図 4.4: メタオブジェクトへ処理を委譲

これは CTMOP#convertSources メソッドが、コンパイル時にメタオブジェクトに処理を委譲するためのコード断片である。1-2 行目でメタオブジェクトをロードするためのクラスローダーを定義している。このクラスローダーを用いて、メタオブジェクトの定義された全てのファイルについて 5-7 行目でメタオブジェクトを生成し、8 行目でメタオブジェクトに処理を委譲している。

4.2.4　コンパイル時に用いる抽象構文木

本フレームワークではコンパイル時のソース変換として、抽象構文木による変換を提供している。この抽象構文木は、ソースコードを org.eclipse.jdt.core.dom.ASTParser を用いて解析したものである。ASTParser を用いることで、char の配列を Java のソースコードとして構文解析を行った後、抽象構文木をあらわす ASTNode を生成することが可能である。

抽象構文木の各要素は ASTNode のサブクラスとして表現される。主な ASTNode としては図 4.2 のようなものが挙げられる。

ASTNode	抽象構文木の要素を表す抽象クラス
CompilationUnit	ソースファイルを表すクラス
TypeDeclaration	クラスを表すクラス
MethodDeclaration	メソッドを表すクラス
FieldDeclaration	フィールドを表すクラス

表 4.2: 主な ASTNode

メタオブジェクトの記述者は、これらの要素で表された抽象構文木を変換することにより、ソースコードの変換を行うことが可能である。抽象構文木を用いた具体的なソースコードの変換方法については、5.2.2 小節で実際の言語拡張例を用いて述べる。

第5章　言語拡張例

この章では言語拡張を行うメタオブジェクトの記述方法とその適用例について述べる。

本フレームワークではコンパイル時に行うソースコードの変換方法として、文字列としての変換と構文木の変換の 2 種類の方法を提供している。5.3 節・5.3 節ではそれぞれの方法について言語拡張の例を挙げ、メタオブジェクトの記述方法及び適用例について述べる。

また 5.3 節では、これらの言語拡張を行うメタオブジェクトを適用することにより、統合開発環境上で拡張言語を用いたソフトウェア開発が可能となることを示す。

5.1　2 進法表記

文字列変換によって実現される言語拡張の例として、3.3 節で挙げた 2 進法表記を用いる。

5.1.1　文法

文法は 3.3 節と同様に、数値の先頭に 0b または 0B を付加することにより 2 進数表記であることを表すものとする。

5.1.2　メタオブジェクトの記述方法

メタオブジェクトにはコンパイル時のソース変換及びエディット時のエラー操作について記述する。以下にそれぞれについて述べる。

コンパイル時

コンパイル時のソース変換は、メタオブジェクトの convertSources メソッドに記述する。convertSources メソッドには 3.3.1 小節で述べたように、以下の処理を記述する。

```
1  @Override
2  public void convertSources() {
3    Pattern p = Pattern.compile("0[bB][01]+");
4    for(CTUnit unit : CTMOP.getUnits()){
5      String sourceCode = new String(unit.getContents());
6      Matcher m = p.matcher(sourceCode);
7      StringBuffer sb = new StringBuffer();
8      while(m.find()){
9        String numStr = m.group().substring(2);
10       long num = Long.parseLong(numStr, 2);
11       m.appendReplacement(sb, String.valueOf(num));
12     }
13     m.appendTail(sb);
14     unit.setContents(sb.toString());
15   }
16 }
```

図 5.1: 2 進法表記のソース変換を行う convertSources メソッド

1. ソースコード中の、0[bB][01]+にマッチする文字列を抽出する。
2. 抽出された文字列を 10 進数に変換する。

この処理を行う convertSources メソッドは、Java 上で正規表現を扱うための java.util.regex パッケージを用いて図 5.1 のように記述される。これは次のように動作する。

1. 0[bB][01]+を表す正規表現を定義 (3 行目)
2. 全てのソースファイルについて以下の処理を行う (4-15 行目)
   (a) 取得したソースコードに対し、 (5 行目)
   パターンマッチを行うオブジェクトを定義 (6 行目)
   (b) 変換後のソースコードを保存する StringBuffer を定義 (7 行目)
   (c) 0[bB][01]+とマッチする部分文字列に検索し、 (8 行目)
   最初の 2 文字 (0b または 0B) を除去し、 (9 行目)
   2 進数から 10 進数に変換し、 (10 行目)
   ソースコードの置換を行う (11 行目)
   (d) マッチする部分文字列全てについて (c) を行う (8-12 行目)
   (e) 残りのソースコードを追加する (13 行目)

```
1  @Override
2  public void finalizeProblems(CompilationUnitDeclaration u) {
3    CompilationResult result = u.compilationResult();
4    for(int i=0;i<result.problems.length;i++){
5      if(result.problems[i] == null) continue;
6      if(result.problems[i].getArguments().length==0) continue;
7      String errStr = result.problems[i].getArguments()[0];
8      if(errStr.matches("^[bB][01]+$")) {
9        result.problems[i] = null;
10     }
11   }
12 }
```

図 5.2: 2 進法表記のエラー処理を行う finalizeProblems メソッド

(f) 変換後のソースコードをソースファイルに適用する (14 行目)

以上の記述により、コンパイル時に前述の処理を行う convertSources() メソッドが実装された。

エディット時

エディット時のエラー処理は、メタオブジェクトの finalizeProblems(CompilationUnitDeclaration) メソッドに記述する。finalizeProblems メソッドには 3.3.2 小節で述べたように、以下の処理を記述する。

1. ソースコード中のすべてのエラーを抽出する。
2. エラーの報告されている文字列が [bB][01]+にマッチする場合、そのエラーを除去する。

この処理を行う finalizeProblems メソッドは、図 5.2 のように記述される。これは次のように動作する。

1. ソースコードの解析結果を取得する (3 行目)
2. 全てのエラー及び警告を取得し、以下の処理を行う (4-11 行目)
   (a) エラーまたは警告が null の場合は何もしない (5 行目)
   (b) 関連付けられた文字列がない場合も何もしない (6 行目)

(c) 関連付けられた文字列を取得し、(7 行目)
[bB][01]+とのマッチングを行い、(8 行目)
マッチした場合はエラーまたは警告の報告を止める (9 行目)

以上の記述により、エディット時に前述の処理を行う finalizeProblems メソッドが実装された。

5.2 @Refines

構文木の変換により実現される言語拡張の例として、@Refines アノテーションを用いて Java に open class の機構を提供する文法を考える。open class は、クラスの定義をそのクラスの外から拡張可能な機構であり、AspectJ におけるインタータイプ宣言に相当する。

5.2.1 文法

ここでは図 5.3 のような文法を用いて Java に open class の機構を提供する。

```
1  public class HelloWorld {
2    public static void main(String[] args){
3      HelloWorld h = new HelloWorld();
4      h.hello();
5    }
6  }
7
8  @Refines(HelloWorld.class)
9  class Ref {
10   private void hello() {
11     System.out.println("Hello world!");
12   }
13 }
```

図 5.3: @Refines 拡張

この文法では、@Refines アノテーションで注釈されたクラスは他のクラスを拡張するためのクラスとなる。図 5.3 の例では Ref クラスが HelloWorld クラスを拡張することを表し、Ref クラスで記述されている hello() メソッドが HelloWorld クラスに追加される。これは HelloWorld クラスが図 5.4 のように定義されているのと同義である。

```
1 public class HelloWorld {
2   public static void main(String[] args){
3     HelloWorld h = new HelloWorld();
4     h.hello();
5   }
6   private void hello() {
7     System.out.println("Hello world!");
8   }
9 }
```

図 5.4: Ref クラスによって拡張された HelloWorld クラス

図 5.3 の HelloWorld クラスは、一般的な Java の文法では hello() メソッドを持っていないためエラーとなるが、拡張された文法では HelloWorld クラスは Ref クラスにより拡張されるため正しい構文となる。このような拡張を行ったコンパイラを作成し、図 5.3 のプログラムを実行すると、

Hello world!!

と出力される。

5.2.2 メタオブジェクトの記述方法

@Refines 拡張についても、5.3 節で挙げた 2 進法表記の拡張の場合と同様にコンパイル時のソース変換及びエディット時のエラー操作についてメタオブジェクトに記述すれば良い。

エディット時にエラーの操作を行う finalizeProblems メソッドは、2 進法表記の拡張の場合と同様に記述することで言語拡張に由来するエラーの除去が可能である。この小節では@Refines 拡張を行うために、コンパイル時に、@Refines 拡張を含むソースコードから既存の Java 言語のソースコードへソース変換を行うための convertSources メソッドの記述方法について述べる。

コンパイル時

@Refines 拡張を含むソースコードは、コンパイルの前に既存の Java 言語のソースコードへの変換を行うことにより Java コンパイラによるコンパイルが可能となる。このような変換は、全てのソースコードに対して次のような処理を行うことで実現可能である。

1. 全てのソースコードの中から@Refines アノテーションで注釈されたクラスを取得 (これをクラス A とする)
2. @Refines アノテーションの引数として記述されているクラスをクラスプールから取得 (これをクラス B とする)
3. クラス A に含まれる全てのメソッド及びフィールドをクラス B にコピー
4. クラス B をクラスプールから除去

このような処理を行うことにより、図 5.3 のように@Refines を含むソースコードを図 5.4 のような既存の Java 言語のソースコードへ変換することが可能である。

ソースファイルを表す CTUnit オブジェクトからソースファイル中で定義されているクラスを取得するには、図 5.5 のように記述する。

```
1 for(Object o : unit.getCompilationUnit().types()) {
2   if(o instanceof TypeDeclaration) {
3     TypeDeclaration type = ((TypeDeclaration)o);
4             :
5   }
6 }
```

図 5.5: CTUnit からクラス定義を取得

ここで、TypeDeclaration は、抽象構文木のクラス宣言を表す要素を意味するクラスである。

CTUnit#getCompilationUnit() メソッドによって、ソースコードを構文解析した抽象構文木が得られる。この抽象構文木は、ソースコードを org.eclipse.jdt.core.dom.ASTParser クラスを用いて構文解析することにより得られるものである。このようにして得られた抽象構文木に対して変換を行うことで、コンパイラに渡すソースコードにも変換が反映される。

同様にクラス定義の中から@Refines アノテーションによって注釈されているクラスのみを取得するには、図 5.6 のように記述する。

抽象構文木の部分木のコピーは図 5.7 のようにして行う。図 5.7 では ASTNode#copySubtrees メソッドを用いて、refinesType が持つ全てのフィールドを refinedType にコピーしている。メソッドについても同様にしてコピーを行うことで、@Refines 拡張の実装が可能である。

```
1 for(Object oo : type.modifiers()) {
2   if(oo instanceof SingleMemberAnnotation)) {
3     SingleMemberAnnotation an = (SingleMemberAnnotation)oo;
4     if(annotation.getTypeName().toString().equals("Refines")) {
5       String typeName = an.getValue().toString();
6               :
7     }
8   }
9 }
```

図 5.6: クラス定義の中から@Refines で注釈されたクラスを取得

```
1 refinedType.bodyDeclarations().addAll(
2   ASTNode.copySubtrees(refinedType.getAST(),
3     java.util.Arrays.asList(refinesType.getFields())
4   )
5 );
```

図 5.7: 部分木のコピー

以上のような構文木の変換を行うことで、コンパイル時に@Refines 拡張を含んだソースコードから既存の Java 言語のソースコードへの変換が可能である。

ビジターパターン

また、抽象構文木には org.eclipse.jdt.core.dom.ASTNode クラスを用いているため、ビジターパターンを用いて走査を行うことも可能である。図 5.5 及び図 5.6 のように@Refines アノテーションで注釈されたクラスのみに処理を行うには、ビジターパターンを用いて図 5.8 のように記述することも可能である。

org.eclipse.jdt.core.dom.ASTVisitor クラスは、ASTNode クラスのインスタンスを走査するためのビジタークラスである。

図 5.8 の例では、1-7 行目で定義されている新しい ASTVisitor を用いて AST の走査を行っている。この ASTVisitor は、アノテーションを訪問する際に、そのアノテーションが@Refines アノテーションであり、かつ親要素がクラス宣言である場合に限り、5 行目に記述される処理を行うことを意味する。よってこのような記述で、@Refines アノテーションで注釈されたクラスのみに処理を行うことが可能である。

```
unit.getCompilationUnit().accept(new ASTVisitor(){
  public boolean visit(SingleMemberAnnotation node) {
    if(node.getTypeName().toString().equals("Refines")
      && node.getParent() instanceof TypeDeclaration){
            :
    }
    return true;
  }
});
```

図 5.8: ビジターパターンを用いた構文木の走査

5.3　拡張された統合開発環境

節及び節のように記述されたメタオブジェクトを適用することによって、統合開発環境上でこれらの拡張が利用可能になることを示す。

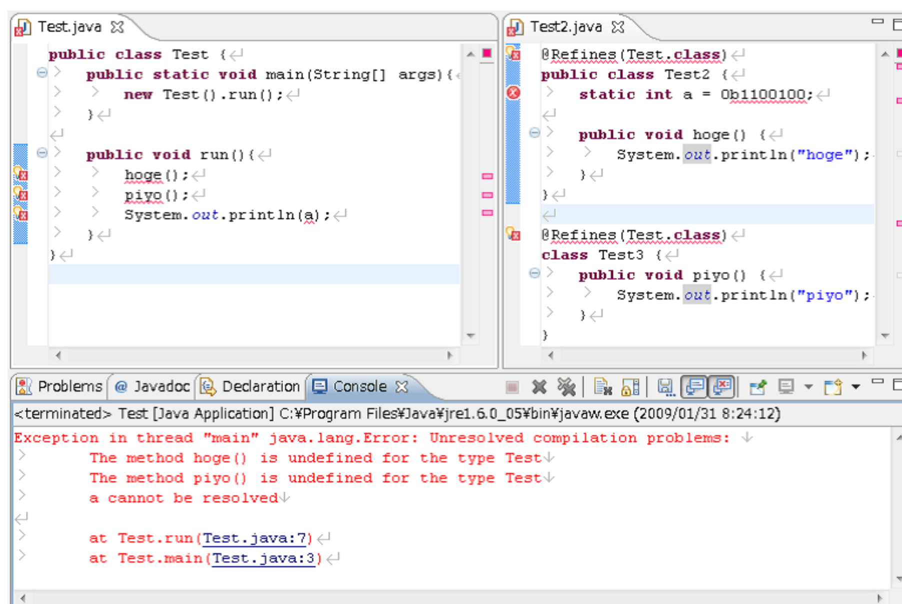

図 5.9: Java 開発環境

Eclipse 上で、既存の Java 開発環境を用いてこれらの拡張を行った言語によるソフトウェア開発を行おうとすると、図 5.9 のようにエラーが報告される。また拡張された構文が用いられているため、コンパイルや実行を行うことも不可能である。

図 5.9 の例では Test クラスの run() メソッド内で、Test クラス内に

hoge()・piyo() メソッド及び変数 a が定義されていないというエラーが報告されている。また Test2.java には、@Refines アノテーションが宣言されていないというエラーや、0b1100100 という記述が解決不能であるというエラーが報告されている。

本フレームワークを用いて、節及び節で記述したメタオブジェクトを適用することによって、図 5.10 のように Eclipse 上で拡張された言語を用いたソフトウェア開発が可能となる。

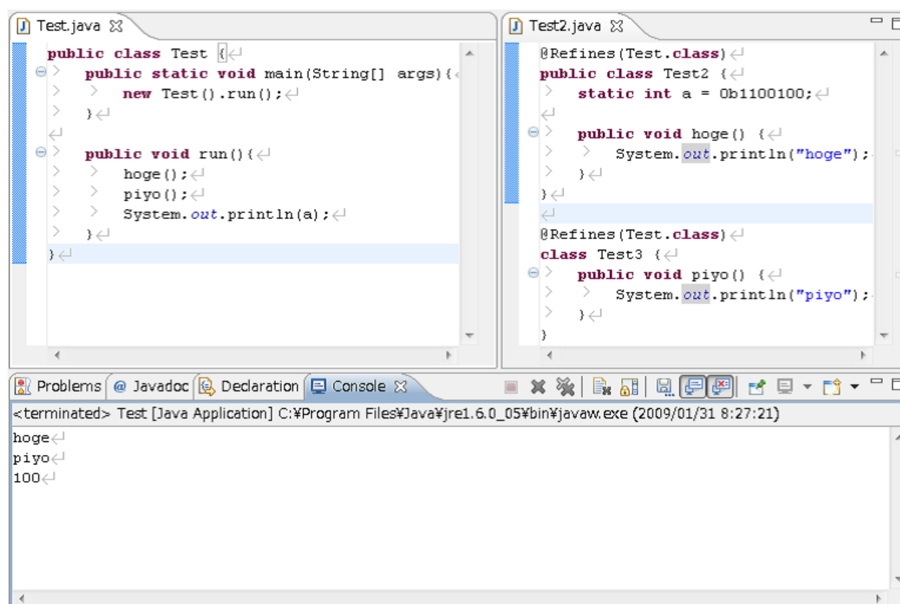

図 5.10: 本フレームワークを用いて拡張された統合開発環境

既存の Java 開発環境を用いた場合では図 5.9 のように多くのエラーが報告されていたが、本フレームワークを用いることによって図 5.10 のように言語拡張に由来するエラーの除去が行われた。また、コンパイル時にはソース変換を行うことにより Test クラスが正しく実行され、

hoge
piyo
100

と出力されている。

また言語拡張に由来しないエラーについては、図 5.11 が示す通り正しく報告が行われる。図 5.11 では、セミコロンが不足しているというエラーが報告されている。

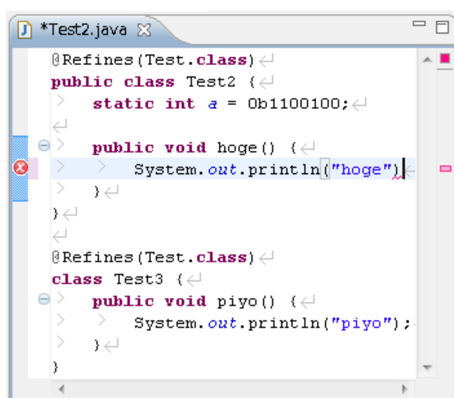

図 5.11: 言語拡張に由来しないエラー

5.4　メタオブジェクトの記述量

5.3 節で扱った 2 進法表記の拡張、及び 5.3 節で扱った@Refines の拡張を行うメタオブジェクトにおいて、コンパイル時のソース変換及びエディット時のエラー操作を行うメソッドの記述に必要となった行数は表 5.1 の通りであった。これは言語拡張の複雑さに比べて十分に小さい値であると考えられる。

	convertSources	finalizeProblems
2 進法表記	12 行	9 行
@Refines	32 行	18 行

表 5.1: メタオブジェクトの記述に必要となった行数

第6章　まとめと今後の課題

6.1　まとめ

本研究では、統合開発環境の扱う言語の拡張を行うためのフレームワークを提案した。

従来、統合開発環境は特定の言語を扱うことを前提として作成されている。そのため統合開発環境の扱う言語を拡張することは意図されておらず、拡張された言語の統合開発環境の作成は困難であった。

本フレームワークでは、ソースコードの変換を行うコンパイルタイムメタオブジェクトプロトコル、及びエラーの操作を行うエディットタイムメタオブジェクトプロトコルを提供することにより、拡張された言語の統合開発環境を、既存の統合開発環境の拡張として容易に実現可能とした。

本フレームワークの実装は、Eclipse プラットフォームの Java 開発環境である JDT プラグインを拡張することによって行った。また言語拡張の例として、2 進法表記及び@Refines の 2 つの例についてメタオブジェクトによる統合開発環境の拡張を行った。これにより、本フレームワークを用いることで統合開発環境の扱う言語を短いコードで拡張可能であることが確認できた。

6.2　今後の課題

本フレームワークにはいくつかの改善すべき問題点が存在する。以下ではそれらの問題点について確認する。

6.2.1　抽象構文木

現在コンパイルタイムメタオブジェクトプロトコルでは、org.eclipse.jdt.core.dom.ASTParser によって解析された抽象構文木を用いた変換を提供している。しかしこの構文木は、ビジターパターンを用いた構文木の探査などの目的においては優れているが、構文木の生成や変更を行うのはやや困難である。そのため、クラスやメソッドの追加や変更など、構文木を扱

うためのメソッドを本フレームワークとして提供することが望ましいと考えられる。

6.2.2　コンパイルの問題

現在の実装では、open class やアスペクトのような、あるクラスの実装が他のクラスの実装に影響を与える言語拡張を考慮するため、コンパイル時に常にプロジェクト全体のフルコンパイルを行っている。

しかしこのような実装では無駄が大きく、コンパイル時間の増大につながってしまう。常にフルコンパイルを行うのではなく、現在コンパイルを行っているクラスにより影響を受けるクラスのみコンパイルを行うようにすべきである。

6.2.3　メタクラス

現在の実装では、特定のフォルダに存在する CompileTimeMetaobject クラスの実装クラスをメタクラスと見なし、そのインスタンスにコンパイル時及びエディット時の操作を委譲している。

しかし、このような実装ではメタクラスをあらかじめコンパイルして用意しておく必要がある。また、拡張構文の利用を停止する際には、一度メタクラスを移動または削除しなければならない。

より良い実装として、メタクラスを Eclipse 上の Java エレメントとして扱うようにすることでこれらの操作を容易にすることが可能であると考えている。

6.2.4　エラーの操作

本フレームワークのエディットタイムメタオブジェクトプロトコルは、拡張された言語を既存の構文解析器で解析した後にエラーの処理を行っている。

しかし言語拡張として文法の拡張を行った場合、既存の構文解析器ではエラーとなるソースコードを解析するため、生成される抽象構文木の予想がつきづらく、またエラーの網羅も困難である。

解決策としては、構文解析器を拡張することによりエラーとなるソースコードも上手く構文解析可能とする方針や、エラーをパターンとして指定可能にすることでエラーの網羅を行いやすくするなどの方針が考えられる。これらの具体的な方法を考えることも今後の課題である。

参考文献

[1] Avgustinov, P., Christensen, A., Hendren, L., Kuzins, S., Lhotak, J., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G. and Tibble, J.: abc: an extensible AspectJ compiler, *Aspect-oriented software development: Proceedings of the 4 th international conference on Aspect-oriented software development*, Vol. 14, No. 18, Springer, pp. 87–98 (2005).

[2] Chiba, S.: A metaobject protocol for C++, *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, ACM Press New York, NY, USA, pp. 285–299 (1995).

[3] Chiba, S. and Ishikawa, R.: Aspect-Oriented Programming Beyond Dependency Injection, *LECTURE NOTES IN COMPUTER SCIENCE*, Vol. 3586, p. 121 (2005).

[4] Chiba, S., Nishizawa, M. and Kumahara, N.: GluonJ home page.

[5] Clement, A., Colyer, A. and Kersten, M.: Aspect-Oriented Programming with AJDT, *ECOOP Workshop on Analysis of Aspect-Oriented Software* (2003).

[6] Eisenberg, A. and Kiczales, G.: Expressive programs through presentation extension, *Proceedings of the 6th international conference on Aspect-oriented software development*, ACM Press New York, NY, USA, pp. 73–84 (2007).

[7] Ekman, T. and Hedin, G.: The jastadd extensible java compiler, *SIGPLAN Not.*, Vol. 42, No. 10, pp. 1–18 (2007).

[8] Kiczales, G., J.Des Rivi ＼ 'eres, Bobrow, D.: *The Art of the Metaobject Protocol*, MIT Press (1991).

[9] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.: An Overview of AspectJ, *LECTURE NOTES IN COMPUTER SCIENCE*, pp. 327–353 (2001).

[10] Microsoft: Visual Studio ホームページ, http://www.microsoft.com/japan/msdn/vstudio/.

[11] Microsystems, S.: Welcome to NetBeans, http://www.netbeans.org/.

[12] Nystrom, N., Clarkson, M. and Myers, A.: Polyglot: An Extensible Compiler Framework for Java, *LECTURE NOTES IN COMPUTER SCIENCE*, pp. 138–152 (2003).

[13] the Eclipse Foundation: The AspectJ Project, http://www.eclipse.org/aspectj/.

[14] the Eclipse Foundation: Eclipse.org home, http://www.eclipse.org/.

[15] R.Vall ＼'ee-Rai, Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot-a Java bytecode optimization framework, *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press (1999).