平成16年度 学士論文

複数個のJoinpointの集合を 対象としたPointcutを 記述可能なアスペクト指向言語

東京工業大学 理学部 情報科学科

学籍番号 01-1457-7

竹内 秀行

指導教員

千葉 滋 助教授

平成17年2月22日

／# 概要

近年、採用されるソフトウェアのライフサイクルのモデルが変化し、一度出荷されたソフトウェアに新たな役割を追加しなければならないことがしばしばある。このとき、既存のソフトウェアと新たに導入する役割が相互に関係しあう場合、オブジェクト指向プログラミングではそれぞれのモジュールが適合しあうように設計段階から変更を加えなくてはならない。このような問題に対し、目的をあらゆる面から分離して開発を行い、最後に合成を図る、アスペクト指向プログラミング (AOP) を解決に利用することが期待されている。

既存の AOP 言語として知られる AspectJ では、アドバイス内で一つの joinpoint しか扱えない。そのため、Hyper/J で示されているような、同じ対象に対して異なるクラス階層を構成するモジュール同士の合成は難しい。AspectJ を用いてこのような合成を実現するためには、それぞれのクラス階層に属している合成したいクラスの組一つ一つのすべてのメソッドおよびフィールドに対して、互いのメソッドおよびフィールドのうち適切なものにアクセスするように変更を加えるような記述をしなければならない。このようにして実現された合成のための記述は、大規模なソフトウェアであればあるほど煩雑になり、それ自体の保守性が低下してしまう。

本研究では、複数個の joinpoint の集合を pointcut で明示的に指定し一つの advice 内でそれを利用することのできる言語 Juggler を提案する。Juggler は AspectJ の文法を基にして、pointcut 式に新しい演算子である連結演算子 (##) を追加した。連結演算子は、演算子で区切られる各々の pointcut 式で得られた joinpoint の集合を、それぞれ配列の要素として前から順番に取り出す。また、Juggler では連結演算子を使用して取り出した複数個の joinpoint の集合を扱うことのできる advice を新たに定義することができる。これを利用して、クラスの合成をするために新たに composition advice を定義し、pointcut で得られたクラスの合成方法を advice コードとして記述する。この記述は、Hyper/J の記述を基にしており、クラスの合成を簡潔に記述できる。Juggler では、このクラスの合成だけではなくほかにも、状態遷移に基づくコードの実行 (Event-based Aspect) や、Visitor の巡回戦略の分離 (Adaptive Visitor) の一部についても記述することができる。実験としてコンパイラの実装を行い、実際に

期待した実行ファイルが生成できること、また、現実的な速度でコンパイルできることを確認した。

謝辞

本研究を進めるにあたり、研究の方向付けや進め方について多くの有用な助言を頂き、指導してくださった千葉滋先生に大変感謝しております。

佐藤芳樹氏には、研究を進め方について数々の助言を頂きました。西澤無我氏には、本論文の根本に関わる問題についての相談を親身になって聞いていただきました。また、石川零氏には、研究を全般的に面倒をみていただきました。論文のスタイルファイルを作り残して頂いた光来健一先生に感謝いたします。

そして、共に研究活動をおこなった研究室の皆さんに、心から感謝いたします。

目 次

第1章 はじめに **8**

第2章 アスペクトの多様性 **10**

2.1 AspectJ とその問題点 10
2.1.1 アスペクト指向プログラミング 10
2.1.2 AspectJ 11
2.1.3 AspectJ では解決することの難しい横断的関心事 . 15
2.2 AspectJ 以外のアスペクト指向言語・システム 19
2.2.1 Hyper/J 19
2.2.2 Event-based AOP 21
2.3 拡張可能なアスペクト指向言語・システム 22
2.3.1 Josh 22
2.4 現在の AOP の問題 24
2.4.1 Joinpoint Model 24
2.4.2 複数の Joinpoint Model を使用することの弊害 . . . 26

第3章 Juggler: n-Dimensional Joinpoint Model **27**

3.1 複数個の Joinpoint の集合 27
3.1.1 Pointcut における複数個の Joinpoint の集合の扱い 28
3.1.2 Advice における複数個の Joinpoint の集合の扱い . 29
3.1.3 n-Dimensional Joinpoint Model のまとめ 30
3.2 アスペクト指向言語 Jugger 31
3.2.1 Juggler 言語の構成 31
3.2.2 Pointcut 文の構成要素 32
3.2.3 Advice の構成要素 32
3.3 実装 33
3.3.1 全体の処理の流れ 33
3.3.2 Polyglot 34
3.3.3 Javassist 35
3.3.4 Plugin 方式による言語の拡張 37
3.3.5 Pointcut の拡張 38
3.3.6 Advice の拡張 39

第4章　応用 **41**
4.1　クラスの統合 41
4.2　状態遷移に対するアドバイスコードの挿入 42
4.3　クラスへの新しいフィールドやメソッドの挿入 43

第5章　実験 **45**
5.1　コンパイル速度 45

第6章　まとめ **47**
6.1　今後の課題 47

付 録A Juggler コンパイラ **50**

図・コード目次

2.1 AspectJ による Advice の例 15
2.2 AspectJ による Introduction の例 15
2.3 給与管理を目的としたクラス構成 16
2.4 人材管理を目的としたクラス構成 17
2.5 AspectJ による人材管理の分離の例 18
2.6 状態遷移に対するアドバイスの挿入の AspectJ による例 . . 19
2.7 Hyper/J におけるアスペクトマッピングの例 21
2.8 JAsCo による StatefulAspects の表現 22
2.9 Josh による pointcut の拡張 23

3.1 複数個の Joinpoint の集合を利用した Advice の記述例 . . . 30
3.2 Juggler による記述例 (LoggingByStatefulAspect.java) . . . 31
3.3 処理のイメージ 34
3.4 PluginRegister のインターフェース 37
3.5 プラグイン登録の例 37
3.6 Pointcut Plugin のインターフェース 38
3.7 Advice Plugin のインターフェース 39

4.1 Juggler によるクラスの統合 41
4.2 Juggler による状態遷移に対するアドバイスの挿入 42
4.3 Juggler による Observer パターン 43

表 目 次

2.1 Weave Process の構成要素 24
2.2 モデルの Weave 処理の構成要素との対応 25

3.1 従来の pointcut 式の演算 28
3.2 n-Dimensional Joinpoint Model における pointcut 式の演算 28
3.3 CtClass の主なメソッド 35
3.4 CtBehavior の主なメソッド 36
3.5 ExprEditor の主なメソッド 36

5.1 コンパイル速度 46

第1章　はじめに

近年、ソフトウェアの開発手法および、運営手法が変化し、既存のソフトウェアに対する変更の要求が強くなっている。システムの開発スパンはどんどん短くなり、従来採用されていたウォータフローモデルによる売り切り型のシステムでは、要求に答えること、また、利益を追求することも難しくなってきている。現在では、中規模程度までの案件に関しては、開発手法としてスパイラルモデルを採用し、また、一度開発し終えたソフトウェアに対して要求に応じて新しい機能を付け加えることで利益をあげるような運営手法をとることが多い。

しかし、ソフトウェアに対する変更の中には、予想しない要求により、設計の初期段階からのやり直しが必要となるものがしばしばある。確かに、オブジェクト指向による設計手法の中には、変更を予想した設計を行うための技術がいくつもあるが、すべての変更の要求を予測することは不可能であるし、また、できたとしても必要以上の拡張性はコストおよびプログラムの肥大化を引き起こす。

このような問題から、ソフトウェアに対する変更を記述するための技術として現在アスペクト指向に対する期待が高まってきている。もともと、アスペクト指向はシステムをあらゆる観点から抽象化し、分離することで、モジュールの切り分けや再利用性を高める手法である。しかし、分離するだけではソフトウェアとして成り立たないため、結合するための技術もまたアスペクト指向の関連技術の中に含まれている。ソフトウェアに対する変更をすでに分離されているアスペクトだと見立てて、既存のモジュールと結合を行うことができる。

既存のアスペクト指向言語として、一般的に使用されている AspectJ がある。AspectJ は、プログラムの実行コードの中のイベントを検出し、そこに新たなコードを挿入することで、既存のプログラムの変更を行う。この、コード中のイベントを joinpoint、それを検出するための記述を pointcut、および、コードの挿入を advice と呼んでいる。これらは、joinpoint をプログラム中の特徴点、pointcut を特徴点の抽出、advice を特徴点の振る舞いの変更と見ることもできる。

この AspectJ ではアドバイス内で一つの joinpoint しか扱うことのできないという制限がある。この制限により記述することの難しいアスペクト

がいくつか存在する。例えば、異なるクラス階層を持つモジュールをそれぞれ別々のアスペクトだと見立てて合成する手法や、複数の joinpoint の実行結果にもとづいたアドバイスの実行等がある。

この結果、現在複数のアスペクト指向言語のモデルが提案され、実装も複数存在することになっている。たとえば、Hyper/J で採用されているような Class Composition Model、DemeterJ で採用されているような Traversal Model、AspectJ の intertype 宣言で採用されている Open Class Model などがある。

実査にこれらのアスペクトを利用して記述しようとした場合複数のモデル・実装を使い分ける必要がある。このことはアスペクト指向を利用使用としているユーザーに対して負担をかける結果となっている。

本研究では、これらのアスペクトの共通点をまとめ、一つのモデルの中で今までいくつかに分かれていたアスペクトのモデルを扱うことをできるようにすることを目的としている。

本稿の残りは、次のような構成からなっている。第 2 章は、アスペクトの多様性と、現在のアスペクト指向言語・システムについて言及し、第 3 章では、各種の Joinpoint Model を包括する解決法として n-Dimensional Joinpoint Model を提唱、第 4 章では、実装である Juggler を使用した応用を、第 5 章では、Juggler のコンパイラとしての評価を行い、そして、第 6 章でまとめを述べる。

第2章　アスペクトの多様性

2.1　AspectJ とその問題点

2.1.1　アスペクト指向プログラミング

2.1.1.1　アスペクト指向プログラミングとは

アスペクト指向 [8, 4] とは、あるシステムの振る舞いをその機能だけでなく、例外処理、相互の通信、相互の調整など本来の処理の目的とは異なるものを分離し、モジュール化することを目的としている。

従来のオブジェクト指向などの開発技法や言語では、基本的に単一の抽象表現（関数、制約など）によって構成されている。しかし、多くの一般的なシステムでは、いくつもの側面 (アスペクト) を持っており、それぞれに適した抽象表現を持っているため、オブジェクト指向によってシステムのある側面をうまく表現できたとしても、ほかの側面をうまく表現できるとは限らない。

以上の話を元にして、アスペクト指向プログラミング (AOP) というコンセプトが提案された。AOP とは、プログラムのさまざまなアスペクトをそれぞれの自然体で表現し、それぞれで表現されたプログラムを互いに織り込むことにより実行コードを生成するという試みである。これは一般的なプログラムの記述においても、また、特定の領域に特化された言語に対しても適用でき、それがおのおのの領域に対して寄与できるとされている。

2.1.1.2　アスペクト

ひとつの抽象表現ではシステムの中に織り込まれているすべての問題を必ずしもうまく表現できるわけではないという事実により、多くの単一の抽象表現しか持たない言語では、システムを構築する上で遠回りな表現になったり、複雑な表現になってしまう箇所がどうしても出てきてしまう。

例えば、オブジェクトの変化を伴わない操作がシステムを構築する上ではしばしば現れる。多くの一般的なオブジェクト指向言語はオブジェクトの振る舞いを明確に捕らえるのは得意ではあるがオブジェクトの変化を

伴わない構造や振る舞いを捕らえるのには不適切である。「もし pop メッセージを受け取ったなら別のオブジェクトに対して refresh メッセージを送れ」というような処理は直接的に表現することができない。

このような問題は、システムのなかのアスペクトをそれぞれコードの部分部分から抜き出すことによりうまく扱うことができるようになる。既存のものとしては関数抽象がよい例で、たとえば、メモリ確保の詳細は、malloc/free インターフェースの後ろに隠され、クライアントのコードはオブジェクトの動きだけを留意すればよいようになっている。

しかし、いくつかの状況においてはシステムにおける 2 つのアスペクトがコードのなかで互いに混じりながら存在しなければならないことがある。そのような状態のことを、「二つのアスペクトが、プログラムの中でお互いを”cross-cut”している」と呼ばれる。わかりやすい例としてオブジェクト分散コンピューティングがある。リモートメッセージとしていくつのオブジェクトを送ればよいか決定するような研究がいくつもなされてきた。それにもかかわらず、実際のプログラムを分析するとコードのいたるところで情報が入り混じっているところが出てきてしまう。その理由として、最適な戦略には、それぞれの個別の送信に関して多くの知識を必要とし、そしてまた多くのコードが実際の送信や他のサブモジュールの影響を受けてしまうということがあげられる。

この、「アスペクト同士の絡み合い」と呼ばれる現象は現存するソフトウェアシステムの多くの複雑性を押し上げているものだと考えられており、この絡まった状態の原因を追求していかなければならない。もし、この原因を突き止めることができ、うまく抽象化することができるようになれば、より直接的にシステムにおけるすべてのアスペクトを調和させ表現できるようになる。

2.1.1.3　アスペクトの結合

AOP においては、先にも述べたように、可能な限りシステムの「横断的関心事」を分離して扱うことを目標としている。しかし、ただ分離しただけでは意味がなく、それを最終的に結合し実行コードとして出力することで、初めてこのような開発手法に意味が出てくる。

このような結合を自動的に行い実行コードを出力するためのツールは、”Aspect Weaver”と呼ばれており、現在複数の実装が提案されている。

2.1.2　AspectJ

2.1.1 で述べた AOP の実装として、Java 言語 [16] にアスペクト指向を取り入れて開発された AspectJ[7] がある。これは現在もっとも幅広く利

用されているされているアスペクト指向言語である。このため、AOP 周りの概念を説明する上で必要な用語として、AspectJ の用語を用いることが多く、また、本論文では AspectJ のプログラミングモデルを対象として提案を行っているため、この節では AspectJ について簡単に紹介および解説を行う。

2.1.2.1　Joinpoint

Joinpoint とは、AspectJ においては、プログラムコードが動作する中で出てくる、操作のことを表す。AspectJ では、この操作をプログラム実行時に検出し新たなコードを挿入することで、アスペクトの結合を行うため、joinpoint「結合点」と呼んでいる。

AspectJ では joinpoint の種類として、メソッド呼び出し、フィールド参照、インスタンス生成、例外ハンドリングなどを定義している。joinpoint の種類や結合方法は言語によって異なっており、特に AspectJ のような形式をとる joinpoint の分類および結合方法は Pointcut & Advice Model として分類されている。これについては 2.2.1 において詳しく述べる。

2.1.2.2　Pointcut

Joinpoint を探し出すための方法として、AspectJ では pointcut と呼ばれるものが導入されている。AspectJ はプログラムの実行時にそれぞれの joinpoint において pointcut とマッチングを行い（実際には最適化が行われコンパイル時にマッチングされるものもある）、マッチした場合に対応したコードを挿入するという処理を行う。primitive pointcut として以下のようなものが定義されている。

`call(MethodPat)` MethodPat が表すメソッドの呼び出し位置 (caller) の joinpoint とマッチする。

`call(ConstructorPat)` ConstructorPat が表すコンストラクタの呼び出し位置の joinpoint とマッチする

`execution(MethodPat)` MethodPat が表すメソッドの実行位置 (callee) の joinpoint とマッチする。

`execution(ConstructorPat)` ConstructorPat が表すコンストラクタの実行位置の joinpoint とマッチする。

`initialization(ConstructorPat)` ConstructorPat が表すコンストラクタのクラスの初期化位置の joinpoint とマッチする。

`preinitialization(ConstructorPat)` ConstructorPat が表すコンストラクタのスーパークラスの初期化位置の joinpoint とマッチする。

`staticinitialization(TypePat)` TypePat が表すクラスの static メンバの初期化位置の joinpoint とマッチする。

`get(FieldPat)` FieldPat が表すクラスフィールドの読み出し位置の joinpoint とマッチする。

`set(FieldPat)` FieldPat が表すクラスフィールドの書き込み位置の joinpoint とマッチする。

`handler(TypePat)` TypePat が表す例外クラスを受け取る例外ハンドラの実行位置の joinpoint とマッチする。

`adviceexecution()` Advice(後述) の実行位置とマッチする。

`within(TypePat)` TypePat が表すクラス内の joinpoint すべてとマッチする。

`withincode(MethodPat)` MethodPat が表すメソッド内の joinpoint すべてとマッチする。

`withincode(ConstructorPat)` ConstructorPat が表すコンストラクタ内の joinpoint すべてとマッチする。

`cflow(Pointcut)` Pointcut が表す joinpoint の内部の joinpoint すべてとマッチする。Pointcut 自身も含む。

`cflowbelow(Pointcut)` Pointcut が表す joinpoint の内部の joinpoint すべてとマッチする。Pointcut 自身は含まない。

`if(Expression)` 実行時に Expression が true である joinpoint すべてとマッチする。

`this(Type | Var)` this が Type で表されるクラスメンバである joinpoint すべてとマッチする。Var が指定された場合は Var に this をバインドする。

`target(Type | Var)` 対象が Type で表されるクラスメンバである joinpoint すべてとマッチする。Var が指定された場合は Var に対象をバインドする。

`args(Type | Var , …)` 引数が Type で表されるクラスである joinpoint すべてとマッチする。Var が指定された場合は Var に引数をバインドする。

また、これらを式として構成した上で joinpoint とマッチングさせることもできる。pointcut 式で使用できる演算子には以下のものがある。

- `!` *pointcut*
  演算子の右の pointcut のマッチング結果の否定を式の結果とする。
- *pointcut* `&&` *pointcut*
  演算子の左右の pointcut のマッチング結果の論理積を式の結果とする。

- *pointcut* `||` *pointcut*
  演算子の左右の pointcut のマッチング結果の論理和を式の結果とする。

これらを使って、以下のように joinpoint を選別することができる。

- `within(Logger) && call(void PrintStream.println(..))`
  Logger クラス内において、PrintStream クラスの println と名前のつくメソッドを呼び出した時すべてとマッチする。

- `!within(Logger) && call(void PrintStream.println(..))`
  Logger クラス外で、PrintStream クラスの println と名前のつくメソッドを呼び出した時すべてとマッチする。

- `execute(void Figure.setX(int)) || execute(void Figure.setY(int)`
  Figure クラスにおいて、setX() もしくは setY() が呼び出された時にマッチする。

2.1.2.3　Advice

Joinpoint における振る舞いを Weaver に対して指示するためのキーワードのことを advice と呼ぶ。AspectJ においては、主に次の 3 種類の advice が定義されており、これらはすべて Java の文法で書かれた advice code をとる。

- `before()`
  Joinpoint の前に advice code を挿入する。

- `after()`
  Joinpoint の後に advice code を挿入する。

- `around()`
  Joinpoint と advice code を差し替える。アドバイスコード内で、proceed キーワードを指定することでもとの Joinpoint を実行することもできる。

以下が使用例となる。

図 2.1: AspectJ による Advice の例

```
1 before(PrintStream out) : within(Logger) && target(out)
2         && call(void PrintStream.println(..)) {
3     // 現在時刻を出力
4     out.print("[" + (new Date()) + "]");
5 }
```

このコードは、Logger クラス内において PrintStream クラスの println と名前のつくメソッドを呼び出した時に、その PrintStream オブジェクトに対して、現在の時刻を出力する。

2.1.2.4 Introduction

AspectJ には、Pointcut & Advice Model のほかに、既存のクラスに対してメソッドやフィールドなどを追加する、Class Introduction と呼ばれる機能を持つ。Open Class とも呼ばれる。

図 2.2: AspectJ による Introduction の例

```
1 public void Point.draw() {
2     Graphics.drawOval(...);
3 }
4
5 public void Line.draw() {
6     Graphics.drawLine(...);
7 }
```

以上のコードは、Point クラスおよび Line クラスに対して draw() メソッドを追加している。

2.1.3 AspectJ では解決することの難しい横断的関心事

AspectJ は、広く一般的に利用されているとはいえ、すべての横断的関心事をうまく記述できるわけではない。この項では、そのような AspectJ では扱うことの難しい横断的関心事についていくつか紹介する。

2.1.3.1 異なるドメイン・モデルを持つシステム

異なるドメイン・モデルを持つシステムにおける横断的関心事を、ある企業における社員管理システムの開発を例として明らかにする。今回は、

給与管理および人材管理の部分を作ることになっている。
　この企業は次のような運営を行っている。

要求定義

- 給与計算は部署によってまったく計算方法が異なる（例えば販売担当は売上に比例、調査担当は一定）。同じ部署内では管理職の給料は非管理職の 1.3 倍となっている。
- 毎週週間報告を提出することになっているが、管理職と非管理職で提出するべき書類が違う。

　これらはそれぞれこれから作ろうとするシステムのアスペクトであることは容易に理解できる。

オブジェクト指向のみで開発した場合　これらのアスペクトはオブジェクトとして切り出すことができるものである。それぞれクラス階層を図にすると、図 2.3、図 2.4 のようになる。

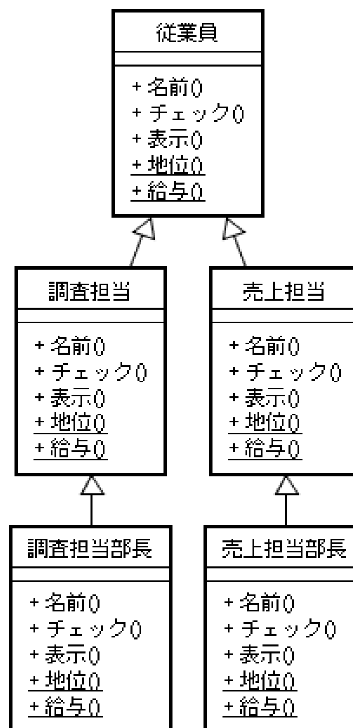

図 2.3: 給与管理を目的としたクラス構成

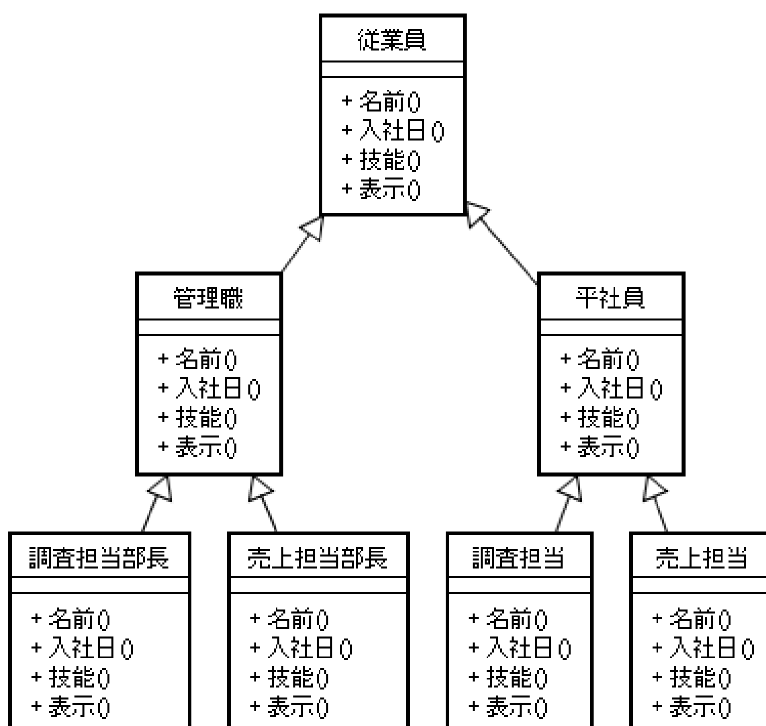

図 2.4: 人材管理を目的としたクラス構成

このような設計ではオブジェクト指向では最終的な実装に落とせないため、最終的にはどちらか一方のクラス階層を基にして、必要なメソッドなりフィールドなりを、もう一方からひとつづつ移していく必要がある。これは、明らかに似たような作業を繰り返すことになり、人為的ミスの非常におきやすい状態になる。

それに加えて、ここでもし基になっていない方のクラス階層の上位クラスにおける仕様変更があった場合、それぞれが適用されているクラスを探し出し一つ一つ手作業で変更していく必要がある。ここでもまた、変更漏れによるバグが現れる。

AspectJ で開発した場合　AspectJ でも、よくその問題点としてあげられるが、やはりどちらか一方のクラス階層を基にして開発しなければならない。そして、それに対し aspect としてもう一方のクラス階層の機能を記述することになる。例えば給与管理のクラス階層を基として人材管理の機能を追加した場合次のような記述できる。

図 2.5: AspectJ による人材管理の分離の例

```
public aspect Personnel {
    public Date payroll.Sales.getJoinDate() {
        ...
    }
    public Date payroll.SalesDirector.getJoinDate() {
        ...
    }

    ...

    after(PayrollBase t)
        : execution(void payroll..*.printOut())
          && this(t) {
        // 情報表示メソッドを変更する
        t.personnelPrintOut();
    }
}
```

これは一見うまくアスペクトを分離できているように思えるが、それぞれの Introduction 間には継承の関係が存在しないため、結局上位のクラスで変更が加えられた場合にはそれぞれの Introduction に対して変更を加えなければならない。

より良いと思われる解決法　これに対して、より良いと思われる方法として、それぞれ完全に別のクラス階層を構成できるのだから、コード上でも両方とも別々のクラス階層として構成してしまい、その後クラス階層を統合するような Aspect Weaver を使用して自動的に統合する方法がある。この方法だと、新たなクラス階層を構成する仕様が後から追加された場合でも柔軟に対応することができる。

2.1.3.2　状態遷移に対するアドバイスの挿入

GUI のアプリケーションを作るときなどに、いくつか決められた操作がされた時に、メッセージを表示したい場合がある。これも、本来の処理と混在している横断的関心事でありアスペクトとして分離できるものである。これについては、AspectJ で書くと次のようになる。

図 2.6: 状態遷移に対するアドバイスの挿入の AspectJ による例

```
1  public aspect ExecByState {
2      private int state;
3
4      after() : execute(void point1()) {
5          if (state == 0) state++;
6          else state = 0;
7      }
8
9      ...
10
11     after() : execute(void pointN()) {
12         if (state == N-1) ShowMesssage("complete!!");
13         state = 0;
14     }
15 }
```

このように確かに AspectJ でも記述できないこともないが、これには明らかにアスペクトの本筋とは関係ない冗長なコードが含まれているのが見て取れる。これではコードの見通しが悪く状態が後から追加されたときや、遷移の分岐が複数表れたときには何がなんだかわからなくなる。これは単純に、

```
point1() -> point2() -> ... -> pointN()
```

のように記述するべきである。

2.2 AspectJ 以外のアスペクト指向言語・システム

前節で紹介したように AspectJ ではうまく記述することのできないアスペクトが存在する。このような問題を解決するために、PA モデルとはまったく別の Joinpoint Model や、PA モデルを拡張したシステムが提案されている。この節ではそのようなモデルやシステムについて紹介する。

2.2.1 Hyper/J

前項で紹介したフレームワークのなかで上げられていた AspectJ とは違ったモデルを使用している Hyper/J について紹介する。Hyper/J は、Hyperspace Approach[14] と呼ばれる手法を用いて、アスペクトの記述および統合を行う。

Hyperspace Approach は、まずシステムにおいて処理する対象領域をHyperspace として定義し、その中に現れる Hyperslice(Joinpoint) を目的別に Concern としてまとめた上で、それぞれ別々に設計・実装し、Concern同士の関係を別に記述することで HyperModule と呼ばれるモジュールを構成する手法である。Hyper/J では、対象言語を Java として実装、Hyperslice をそれぞれ Java の Package, Class, Method, Field としている。

以下に、Hyper/J の使用例を示す。対象とするシステムは 2.1.3.1 で問題としてあげられていたシステムとし、それぞれを別のクラス階層として設計・実装を行ったものとする。詳細は次のとおりである。

- 給与管理システムのクラス階層をつくりパッケージ名を payroll.*とする。クラス名はそれぞれの役職名とする。
- 人材管理システムのクラス階層をつくりパッケージ名を personnel.*とする。クラス名はそれぞれの役職名とする。
- payroll.*と personnel.*をそれぞれ別の Concern として Feature.Personnel および Feature.Payroll を定義する。
- HyperModule はそれぞれの Concern の同じクラス名同士で統合することで構成する。このとき、name フィールドは同じものとして、また、同じ名前のメソッドは順番に実行されるメソッドとして、print() メソッドの戻り値を joinStrings() というメソッドで再構成する。

Hyper/J に渡すファイルの記述は以下のとおりになる。

図 2.7: Hyper/J におけるアスペクトマッピングの例

```
1  -hyperspace
2    hyperspace DemoHyperspace
3      composable class personnel.*;
4      composable class payroll.*;
5
6  -concerns
7    package personnel : Feature.Personnel
8    package payroll : Feature.Payroll
9
10 -hypermodules
11   hypermodule PersonnelPayroll
12     hyperslices:
13       Feature.Personnel, Feature.Payroll;
14     relationships:
15       mergeByName;
16       override operation Feature.Payroll.name
17         with operation Feature.Personnel.name;
18       set summary function for
19         operation PersonnelPayroll.print
20             to PersonnelPayroll.joinStrings;
21   end hypermodule;
```

2.2.2 Event-based AOP

ここでは、2.1.3.2 においてあげた AspectJ では記述することの難しい問題を解決するための方法として、Event-based AOP [3] およびその実装として JAsCo[5] を紹介する。Pointcut & Advice Model を拡張して、Aspect に対して状態を持たせることで問題の解決を行っている。また、過去の状態におけるデータを後から参照することも提案されており、この応用としてデータフローを追うためのアスペクトに関する提案もされている [10]。

以下に、JAsCo による Event-based AOP の使用例を示す。次の記述は、startJuggling() が呼び出され stopJuggling() が呼び出されるまでの間のすべてのメソッドが呼ばれるときにログを吐き出す処理を行っている。

図 2.8: JAsCo による StatefulAspects の表現

```
package test;

class ExampleLoggerBean {
  hook StatefulHook {
    StatefulHook(startmethod(..args1),
        runningmethod(..args2),
        stopmethod(..args3)) {
      start > p1;
      p1: execute(startmethod) > p3||p2;
      p3: execute(stopmethod) > p1;
      p2: execute(runningmethod) > p3||p2;
    }

    before p2 (){
      // do the logging:
      System.out.println("executing: "
        + calledmethod.getName() + " "
        + calledmethod.getClassName());
    }
  }
}

static connector testconnector {
  ExampleLoggerBean.StatefulHook hook1 =
    new ExampleLoggerBean.StatefulHook(
      * *.*.startJuggling(*),
      * *.*(*),
      * *.*.stopJuggling(*));
}
```

2.3　拡張可能なアスペクト指向言語・システム

本研究における実装は、拡張可能なアスペクト指向言語として実装されるので、ここで関連研究を紹介する。

2.3.1　Josh

Josh[2] は、豊富な情報を基にした pointcut を記述できるアスペクト指向言語として提案されている。AspectJ は、広く使われており完成度も高いが、pointcut の記述に使う、poincut 言語が不十分であるといわれている。このためユーザーの意図した joinpoint を抽出することのできない場合がある。また、汎用的なインタータイプ宣言が記述することができず、

同じようなプログラムを繰り返して書かなければならないという問題もある。

これらの問題に対して、Josh ではユーザーが独自に pointcut の定義を Java で記述することを可能とすることで対処している。この際に、リフレクションを使用してプログラム内のさまざまな情報を参照しより複雑な joinpoint を切り出すことのできる pointcut を定義することができる。定義自体には多少手がかかり技術が必要とされるが、他のプログラマが再利用することは容易である。

また、Josh ではインタータイプ宣言にも汎用性のための機能が備わっており、インタータイプ宣言の対象を pointcut 言語で指定できる。また対象に応じて、異なったメソッドやフィールドが追加されるようなインタータイプ宣言を記述可能である。

2.3.1.1　pointcut の拡張

Josh では、pointcut を Java の静的メソッドとして定義させることで比較的容易に pointcut を追加することができるようになっている。例えば、メソッド呼び出しを抽出するが、その際にクラス名とメソッド名のみが条件となる単純な指定子を定義すると以下のようになる。通常の call 指定子などと違い、返り値や引数の数などは抽出の条件としていない。

図 2.9: Josh による pointcut の拡張

```
1  static boolean simpleCall(MethodCall mc,
2      String[] args, JoshContext jc) {
3    // 上記の使用例の場合 args[0]="Point",
4    // args[1]="hello" である
5    String calleeName = mc.getMethodName();
6    String className = mc.getClassName();
7    if ( className.equals(args[0]) &&
8      calleeName.equals(args[1]) )
9      return true;
10   else
11     return false;
12 }
```

pointcut 指定子メソッドの引数は 3 つであり、型も特定のものでなければならない。第一引数は、この指定子が抽出対象とする joinpoint オブジェクトである。第二引数は String 型の配列であり、この指定子が joinpoint を抽出するか否かを決める条件に使われる。pointcut 指定子の使用の際には、カンマで区切って文字列を記述すればよく、処理系が自動的にそれ

らを配列の中に入れる。第三引数は JoshContext クラスである。このクラスのオブジェクトは weave の過程における様々な情報を保持しており、その情報を基に pointcut をすることができる。これも指定子使用の際には明記する必要なく、処理系が自動的に生成する。

2.4 現在の AOP の問題

2.4.1 Joinpoint Model

[11] の論文では、横断的関心事のモジュール化を目的として提案されている４つの Joinpoint Model について、Weave 処理をモデル化することで、これらの Joinpoint Model の核となっている意味論を捕らえている。

Weave 処理のモデル化は、まず、Weave とはあるプログラム A とあるプログラム B をあるルール META にそって、ひとつの統合されたシステム X を作ることだと定義し、

$$\mathrm{A} \times \mathrm{B} \times \mathrm{META} \rightarrow \mathrm{X}$$

その上で、A, B, X をその意味ごとにそれぞれ細かく、

$$\langle \mathrm{X}, \mathrm{X}_{\mathrm{JP}}, \mathrm{A}, \mathrm{A}_{\mathrm{ID}}, \mathrm{A}_{\mathrm{EFF}}, \mathrm{A}_{\mathrm{MOD}}, \mathrm{B}, \mathrm{B}_{\mathrm{ID}}, \mathrm{B}_{\mathrm{EFF}}, \mathrm{B}_{\mathrm{MOD}} \rangle$$

の 11 の要素に分割することで行っている。

表 2.1: Weave Process の構成要素

X	Weave 処理の結果。
X_{JP}	X の中の joinpoint。
A, B	入力となる言語。
$\mathrm{A}_{\mathrm{ID}}, \mathrm{B}_{\mathrm{ID}}$	X_{JP} の中の要素を識別するための方法。
$\mathrm{A}_{\mathrm{EFF}}, \mathrm{B}_{\mathrm{EFF}}$	識別された joinpoint において動作に影響を与える手段。
$\mathrm{A}_{\mathrm{MOD}}, \mathrm{B}_{\mathrm{MOD}}$	言語 A, B でモジュール化されたユニット。
META	Weave 処理のパラメータとなるメタ言語。

次にこの論文において上げられている４つのモデル化された Weaver の処理の特徴について次に述べる。

- PA - Plugin and Advice Model
  主に AspectJ を構成しているモデル。実行中のコードに現れる横断的関心事をあらかじめ決められたポイントで分離することができる。決められたポイントを Joinpoint とし、これを Pointcut によって判別することで、Advice として分離されたコードを呼び出す。

- TRAV - Traversals Model
  主に DemeterJ[9], DJ[13] を構成しているモデル。Visitor パターンの巡回戦略を横断的関心事として分離する。構成されたオブジェクトのグラフを、Visitor がたどる中で現れるそれぞれのオブジェクトに到達する点を Joinpoint とし、Visitor がオブジェクトに到達するつど与えられた巡回戦略と比較を行い、Visitor の継続を決定する。

- COMPOSITOR - Class Composition Model
  主に Hyper/J[15, 6] を構成しているモデル。言語の構成要素を Joinpoint としている。いくつかの Joinpoint をまとめてひとつのモジュールとして定義し、複数のモジュールの統合を決められたルールによっておこなう。

- OC - Open Class Model
  AspectJ の Introduction 等を構成しているモデル。本来のクラスの役割とは関係ないメソッドやフィールドを横断的関心事として分離する。クラスを Joinpoint とし、クラスが見つかるごとに Introduction とマッチングを行いマッチすればフィールドやメソッドを挿入する。

それぞれのフレームワークにおいて何が先ほど上げた Weave 処理の構成要素と対応しているかについて

表 2.2: モデルの Weave 処理の構成要素との対応

	PA	TRAV	COMPOSITOR	OC
X	コードの実行	巡回の実行	再構成されたプログラム	統合されたプログラム
X_{JP}	メソッドの呼び出し	オブジェクトへの到達	X 内の宣言	クラスの宣言
A	c, m, f の定義	c, f の定義	c, m, f の定義	OC を除く c の定義
A_{ID}	m の signature など	c, f の signature	c, m, f の signature	m の signature
A_{EFF}	メソッドの実行	到達可能性を提供する	宣言を提供する	宣言を提供する
B	アドバイスの定義	巡回戦略と visitor	(=A)	OC メソッドの宣言
B_{ID}	ポイントカット	巡回戦略	(=A_{ID})	影響するメソッドの宣言
B_{EFF}	アドバイスコードの実行	visitor の呼び出しと継続	(=A_{EFF})	メソッドの宣言のコピー
META	なし	なし	統合のルール	なし

表の中で出てくる c,m,f はそれぞれ class, method, field を表す。

2.4.2　複数の Joinpoint Model を使用することの弊害

前項で述べたように現在複数のアスペクトの切り口が提案され、それに対する解法としての Joinpoint が存在している。これらを使用して、異なるクラス階層を持ち、ログをとる必要があるシステムのような、複数のアスペクトを持つシステムを記述使用とすると、例えば、Hyper/J と AspectJ の両方の知識が必要となり、また複数のツールを切り替えて使う必要があるためアスペクトの必要以上の分散が起こってしまう。

今までのアスペクト指向は、既存の Joinpoint Model で対応できない問題が出てくるごとに、他の新しい Joinpoint Model を提案するか、Joinpoint Model を拡張することで対症療法的に対応してきた。しかしながら、これでは上記のような二つの Joinpoint Model にまたがるようなアスペクトが表れたときに対処が難しくなってしまう。また、またがっていない場合であっても複数の概念・ツールを使い分けなければならないことはユーザーに対して負担をかけてしまう。

第3章 Juggler: n-Dimensional Joinpoint Model

前章での問題に対処するために、この章では、複数の joinpoint を明示的に利用することのできるモデルである n-Dimensional Joinpoint Model を、Pointcut & Advice Model の拡張として提案する。

また、AspectJ の文法を拡張し、n-Dimensional Joinpoint Model を扱えるようにした言語 Juggler について解説を行う。

3.1 複数個の Joinpoint の集合

表 2.4.1 および PA モデルを拡張した言語を見ると、それぞれに joinpoint が存在し、それを抽出するためのルールを記述し、何らかの処理を加えていることがわかる。このことは、対象をより抽象化するとすべて同じ問題として扱えることを示している。

同じ問題として扱えるにも関わらず、今までのように様々なモデルが提案されてしまっている理由として、それぞれのモデルの記述能力が低いことが上げられる。本来なら、抽出 (pointcut) とそれに対する影響 (advice) という切り分けを行っている PA モデルのみで扱えるはずである。

しかし現状の Pointcut & Advice Model では他のモデルが対象にしているような問題を記述することはできない。これは、例えば Class Composition Model が 2 つのクラスを対象として処理を行っていることからもわかるとおり、問題が複数個の joinpoint に関係していることことに起因する。つまり、複数個の joinpoint の集合を明示的に扱えるような拡張を Pointcut & Advice Model に対して施せば他の問題を記述することができる。

本研究では、pointcut および advice に対して拡張を施し、joinpoint 集合の配列を pointcut と advice との間で受け渡すことで複数の joinpoint を扱う。

3.1.1　Pointcut における複数個の Joinpoint の集合の扱い

3.1.1.1　連結演算子

複数個の joinpoint を最終的に advice に対して渡して処理をするためには、pointcut 式で複数個の joinpoint を選択できるようにする必要がある。そこで、従来の pointcut 式に対して連結演算子 (##) を導入する。これは、joinpoint 集合同士の直積を意味し、演算子の左右の joinpoin 集合の配列を連結した形になる。

従来の pointcut 式の定義は、

表 3.1: 従来の pointcut 式の演算

<table>
<tr><td>pointcut :</td><td>JPS[†]</td></tr>
<tr><td>&&(積集合) :</td><td>JPS × JPS → JPS</td></tr>
<tr><td>||(和集合) :</td><td>JPS × JPS → JPS</td></tr>
</table>

となっていたが、これを joinpoint の集合に対して次元を導入することで、次のような pointcut 式の定義となる。

表 3.2: n-Dimensional Joinpoint Model における pointcut 式の演算

<table>
<tr><td>pointcut :</td><td>JPS(1)[‡]</td></tr>
<tr><td>##(連結) :</td><td>JPS(m) × JPS(n) → JPS(m+n)</td></tr>
<tr><td>&&(積集合) :</td><td>JPS(n) × JPS(n) → JPS(n)</td></tr>
<tr><td>||(和集合) :</td><td>JPS(n) × JPS(n) → JPS(n)</td></tr>
</table>

積集合および和集合は joinpoint 集合の配列のそれぞれの要素ごとに積集合、和集合を取る。それぞれの演算子の優先順位は

```
## < || < &&
```

とした。下のどちらも同じ結果となる。

- `(A() && B()) ## (C() && D())`
- `(A() ## C()) && (B() ## D())`

[†] JPS : joinpoint の集合
[‡] JPS(n) : joinpoint の集合の n 次元の配列

3.1.1.2　Pointcut の種類

pointcut については、既存の研究でもわかるとおり joinpoint の種類がいくらでも存在するので、特にはじめから定義はせずユーザー自身が必要に応じて追加できるようなプラグイン形式をとることとなる。
動作についてはすべて pointcut プラグインに依存する。

3.1.1.3　実際の連結演算子の用法

例えば、`call(void start())` と `call(void end())` で選択される二つの joinpoint をまとめたいときは、

```
call(void start()) ## call(void end())
```

が pointcut となる。これにより、配列の 1 番目に要素が `start()` を呼び出している箇所、配列の 2 番目に `end()` を呼び出している箇所がそれぞれ入る。

3.1.2　Advice における複数個の Joinpoint の集合の扱い

3.1.2.1　Advice の役割

Pointcut だけでは、ただ joinpoint 集合の配列を生成するだけなので、実際の joinpoint 集合間の関係はその処理を行う advice が入れることとなる。
例えば、Class Composition Model と同様の処理を行おうとした場合は、それぞれの joinpoint 間には同値の関係が入り、Stateful Aspect を表現しようとした場合には順序の関係を入れることになる。

3.1.2.2　Advice の種類

Pointcut & Advice Model においては、joinpoint の前か後ろという二つの関係しか存在しなかったが、複数の joinpoint を同時に扱うことにより、関係は爆発的に増加する。
このように、Pointcut と同様に、advice の種類についても制限することはできないので、はじめから定義はせずにプラグイン形式で導入することとなる。
また、それに伴い、advice code に関しても特に定義せず、プラグイン製作者が、advice code を解釈するようにした。

3.1.2.3　Advice の実際の用法

2.2.1 の例を、本モデルで記述した例を以下に示す。

図 3.1: 複数個の Joinpoint の集合を利用した Advice の記述例

```
1  identical() : class(personnel.*) ## class(payroll.*) {
2    // 文法はプラグインに依存している
3    mergeByName;
4    override void $2.name() with void $1.name();
5    summary of String print() {
6      StringBuffer ret = new StringBuffer();
7      for (int i = 0; i < $result.length; ++i)
8        ret.append($result[i]);
9      return ret.toString();
10   }
11 }
```

この例では、同一の関係を表す identical() advice を利用して Class Composition を行っている。記述の中の`$2.name()` および`$1.name()` はそれぞれ、personnel パッケージ、payroll パッケージから取り出したクラスの name() メソッドを表し、また、summary of では、それぞれの joinpoint から取り出した print() メソッドの戻り値の扱いについての記述を行っている。この例については応用の章にて詳しく取り上げる。

これらの文法については、identical() advice のプラグインにすべて依存するものであり、必ずしもアドバイスを実装するプラグインがこのような文法を取るとは限らない。

3.1.3　n-Dimensional Joinpoint Model のまとめ

上記のように、個々の pointcut および advice については特に定義せず、複数個の joinpoint の集合の明示的な利用についてのみ規定することで、既存のアスペクトを同じように抽象化することができる。

しかしながら、実際にはコンパイラの実装によってさまざまな制限がかけられる。次の節では実装の詳細について述べつつこの制限についてもあげる。

3.2　アスペクト指向言語 Jugger

3.2.1　Juggler 言語の構成

Juggler 言語は AspectJ の言語を基にして構成され、pointcut 式に対して前節で述べたような拡張が施され、pointcut および advice は、言語の要素としては提供せず、すべて言語拡張として他で定義される。また、AspectJ では Class Introduction に関する文法が存在したが、Class Introduction も n-Dimensional Joinpoint Model に取り込むことができるため、Juggler の文法としては定義しない。

図 3.2: Juggler による記述例 (LoggingByStatefulAspect.java)

```
1  package st.chimera;
2
3  import java.io.*;
4  import java.util.*;
5
6  public aspect LoggingByStatefulAspect {
7    private PrintWriter out;
8
9    public LoggingAspect(Writer out) {
10     this.out = new PrintWriter(out);
11   }
12
13   private void log(String str) {
14     out.println("[" + (new Date()) + "] ");
15   }
16
17   pointcut statefulpointcut() :
18     execution(void Juggler.start())
19       ## execution(void Juggler.end());
20
21   orderd_after() : statefulpointcut() {
22     log("juggling end.");
23   }
24 }
```

上記は、Juggler.start() と Juggler.end() が順番に呼び出された場合に Log をとるという機能を、Juggler で記述した例である。execution pointcut および、orderd_after advice は言語拡張によって追加されたもので、それぞれ次のような作用を持つものとして定義されている。

- **execution() pointcut** AspectJ の execution() pointcut と同等の機能を持つ pointcut である。指定されたメソッド本体を Joinpoint

として抽出する。

- **orderd_after() advice** 与えられた joinpoint 配列の要素である joinpoint を前から順番に通過したときに、advice code を Java の実行コードとみなして実行する。

上記のように、連結演算子があることと、pointcut および advice の名前が言語としては未定義という点を除いてはほとんど AspectJ とは変わらない。

3.2.2　Pointcut 文の構成要素

Pointcut 文の構成要素は AspectJ とほとんど変わりはない。

3.2.2.1　pointcut_decralation

```
<modifire> pointcut <identifier>(<formal_parameter_list>)
    : <pointcut_expression>;
```

- `<modifier>` アクセス指定子
- `<identifier>` pointcut 名
- `<formal_parameter_list>` pointcut 変数
- `<pointcut_expression>` pointcut 式

3.2.2.2　pointcut_expression

```
<pointcut_expression> && <pointcut_expression>
<pointcut_expression> || <pointcut_expression>
<pointcut_expression> ## <pointcut_expression>
!<pointcut_expression>
```

- `<pointcut_expression>` pointcut 式

3.2.3　Advice の構成要素

Juggler における Advice 文の構成要素は、AspectJ とは違い advice code が Java のコードとは限らず、advice plugin の実装に依存する。

3.2.3.1　advice_decralation

```
<modifire> <identifier>(<formal_parameter_list>)
    : <pointcut_expression> {
    <advice_code>
}
```

- `<modifier>` アクセス指定子
- `<identifier>` advice 識別名
- `<formal_parameter_list>` pointcut 変数
- `<pointcut_expression>` pointcut 式
- `<advice_code>` advice code 文法は advice に依存

3.3　実装

Java のコンパイラフレームワークである Polyglot[12] および Javassist を利用して Juggler コンパイラの実装を行った。

3.3.1　全体の処理の流れ

Juggler コンパイラの処理の流れについて次に簡単に述べる。

1. .java(juggler 言語) と.class(クラスファイル) を入力としてとる。
2. polyglot framework を利用して構成した、字句解析器・構文解析器・意味解析器を使用して、.java(Java 言語) への変換および Aspect の抽出を行う。
3. 変換された.java(Java 言語) は javac に対して渡され、コンパイルされ weaver に対して渡される。
4. Weaver は、polyglot framework および javac から aspect とクラスファイルを受け取る。
5. クラスファイルを Javassist を利用してスキャンし、joinpoint のリストを作る。
6. Aspect 内の pointcut 記述に応じてプラグインを呼び出し、先ほど作成した joinpoint のリストを渡し、joinpoint の選別を行う。
7. Aspect 内の advice 記述に応じてプラグインを呼び出し、pointcut が選別した joinpoint に対して、Javassist を利用して処理を行う。

図 3.3: 処理のイメージ

3.3.2　Polyglot

Polyglot とは、Java の拡張コンパイラ作成のためのフレームワークであり、実装の作業を大幅に削減してくれる。基本的には Java の拡張言語から、通常の Java 言語への変換を目的としており、polyglot 自体は Java バイトコード生成の機能は持たない。拡張コンパイラ作成者は次の作業を行うことになる。

1. 字句解析器を JFlex で記述し、自動生成する。
2. 抽象構文木のノードのうち拡張言語に属するものを作る。
3. Java 文法との差分を PPG で記述し、構文解析器を自動生成する。
4. 抽象構文木を巡回する Visitor クラスを実装し、コンパイラフロントエンドのタスクに追加する。

3.3.3 Javassist

Javassist[1, 17] は、構造リフレクションを提供するクラスライブラリである。Juggler では、Javassist によってクラスファイルに対して改変を行うことで、weave 処理を行う。

例えば、”Point”クラスに対して改変を行う場合はまず始めに、Point クラスの CtClass(compile-time Class) を次のように取得する。

```
1 ClassPool pool = ClassPool.getDefault();
2 CtClass ctpoint = pool.get(”Point”);
```

この CtClass には、クラスのフィールドやメソッドなどを参照するために次のようなメソッドが用意されている。

表 3.3: CtClass の主なメソッド

メソッド名	説明
CtConstructor[] getConstructors(void)	コンストラクタを得る
CtField[] getFields(void)	フィールドを得る
CtMethod[] getMethods(void)	メソッドを得る
int getModifires(void)	修飾子を得る
String getName(void)	クラス名を得る
CtClass getSuperClass(void)	親クラスを得る
void addField(CtField f)	f をクラスに追加する
void addMethod(CtMethod m)	m をクラスに追加する

これらの CtConstructor, CtField, CtMethod は、それぞれの名前や修飾子、引数の型などの情報を取り出すようなメソッドを持っており、CtConstructor, CtMethod については、それぞれの動作を変更する次のようなメソッドを保持している。

表 3.4: CtBehavior の主なメソッド

メソッド名	説明
void insertBefore(String src)	メソッドの開始位置に src のコードを挿入する
void insertAfter(String src)	メソッドの終了位置に src のコードを挿入する
void setBody(String src)	メソッドの処理を src のコードで置き換える
void instrument(ExprEditor editor)	処理を editor で走査する

上記の instrument() ではメソッドを捜査する中で現れる関数呼び出しやフィールドアクセスごとに、ExprEditor インターフェースのメソッドが呼ばれる。ユーザーは ExprEditor を実装するクラスを作成し、instrument() にそのオブジェクトを渡すことでメソッド内の動作を変更することができる。ExprEditor は次のようなインターフェースを持つ。

表 3.5: ExprEditor の主なメソッド

メソッド名	説明
void edit(Cast c)	クラスキャストが現れるごとに呼び出される
void edit(FieldAccess f)	フィールドアクセスが現れるごとに呼び出される
void edit(Handler h)	例外ハンドラがあられるごとに呼び出される
void edit(Instanceof i)	instanceof 命令が現れるごとに呼び出される
void edit(MethodCall m)	メソッド呼び出しが現れるごとに呼び出される
void edit(NewArray a)	配列の初期化が現れるごとに呼び出される
void edit(NewExpr e)	クラスの初期化が現れるごとに呼び出される

上記のそれぞれのメソッドに渡される MethodCall, FieldAccess 等パラメータは、コード上の状況を保持している。

3.3.4 Plugin 方式による言語の拡張

先にも述べたように、n-Dimensional Joinpoint Model においては pointcut と advice に関しては明確な定義を持たない。Juggler の処理系自身も pointcut および advice の実装は持たず、すべてプラグイン方式での追加となる。pointcut や advice を追加する場合は次の PluginRegister インターフェースを実装し、Juggler に対して渡せばよい。

図 3.4: PluginRegister のインターフェース

```
package juggler.weaver;

public interface PluginRegister {
  // によって呼び出される登録メソッドJuggler
  void regist(PluginManager loader);
}
```

PluginManager は主に登録用に次のような登録用メソッドを持ち、プラグイン実装者は regist() メソッド内でこれらを使用することで、Juggler 処理系に対して、プラグインを追加することができる。

- public void addPointcutPlugin(PointcutPlugin pointcut);
  pointcut で指定された PointcutPlugin を処理系に登録する
- public void addAdvicePlugin(AdvicePlugin advice);
  advice で指定された AdvicePlugin を処理系に登録する

通常は次のように利用する。

図 3.5: プラグイン登録の例

```
package juggler.builtin.advice;

import juggler.weaver.PluginManager;
import juggler.weaver.PluginRegister;

public class BuiltinAdviceRegister
        implements PluginRegister {
    public void regist(PluginManager loader) {
        // Before および Advice は AdvicePlugin を実装
        loader.addAdvicePlugin(new Before());
        loader.addAdvicePlugin(new After());
    }
}
```

3.3.5　Pointcut の拡張

Pointcut の拡張は次のようなインターフェースを実装して行う。

図 3.6: Pointcut Plugin のインターフェース

```
package juggler.weaver;

import java.util.List;

import juggler.PointcutArgument;
import juggler.exception.PointcutException;
import juggler.joinpoint.Joinpoint;

public interface PointcutPlugin {
  /** pointcut の名前を返す */
  String name();

  /** 与えられた joinpoint の評価を行う */
  void eval(List<Joinpoint> contexts,
    PointcutResult result, String[] args,
    PointcutArgument[] pointcutArgs)
      throws PointcutException;
}
```

name() で取得できる文字列で字句解析が行われ、トークンが pointcut-primitive であるとして処理される。eval() に渡される引数について説明する。

- contexts
  システム中の全 joinpoint のリスト。Joinpoint オブジェクトにはそれぞれの joinpoint の種類、名前、引数の型など評価に必要な情報が含まれている。

- result
  それぞれの joinpoint の評価結果を格納するためのオブジェクト。eval() では、上の contexts を評価し評価結果を result の持つ BitSet オブジェクトに対し格納する。

- args
  この pointcut 自身に渡されたパラメータ。
  `<pointcut-primitive>(<pointcut-parameter-list>)`
  における`<pointcut-parameter-list>`

- pointcutArgs
  pointcut 文の宣言で与えられる引数の配列。
  `pointcut <identifier>(<formal-parameter-list>) : ... ;`
  における`<formal-parameter-list>`

Juggler は、プログラムの中に存在するすべての joinpoint を Pointcut-Plugin を実装するオブジェクトに対して、次々に投げ込んでいくことで、Joinpoint の選別を行う。

3.3.6　Advice の拡張

Advice の拡張は次のようなインターフェースを実装して行う。

図 3.7: Advice Plugin のインターフェース

```
package juggler.weaver;

import java.util.List;

import juggler.AdviceArgument;
import juggler.exception.AdviceException;
import juggler.joinpoint.Joinpoint;

public interface AdvicePlugin {
  /** advice の名前を返す */
  String name();

  /** advice の処理を行う */
  void process(List<Joinpoint> contexts,
    PointcutResult[] results, String adviceCode,
    AdviceArgument[] adviceArgs)
      throws AdviceException;
}
```

name() で取得できる文字列で字句解析が行われ、トークンが pointcut-primitive であるとして処理される。process() に渡される引数について説明する。

- contexts
  システム中の全 joinpoint のリスト。Joinpoint オブジェクトにはそれぞれの joinpoint の種類、名前、引数の型など評価に必要な情報が含まれている。

- results
  pointcut による選別結果。Pointcut 式が連結演算子を使い joinpoint 集合の配列を構成している場合は、pointcut 式上の順番で results 配列に入る。

- adviceColde
  `<advice-code>`がそのまま渡される。

- adviceArgs
  advice 文の宣言で与えられる引数の配列。
  `<advice-name>(<formal-parameter-list>) : ... ;`
  における`<formal-parameter-list>`

第4章　応用

4.1　クラスの統合

2.1.3.1 で話題とした、異なるクラス階層の合成を行うための Advice Plugin の実装を行った。2.2.1 を Juggler で記述した例を以下に示す。

図 4.1: Juggler によるクラスの統合

```
1  public aspect PersonnelPayroll {
2    composition() :
3        class(personnel.*) ## class(payroll.*) {
4      mergeByName;
5      override void $2.name() with void $1.name();
6      String $composed.toString() {
7        StringBuffer ret = new StringBuffer();
8        for (int i = 0; i < $result.length; ++i)
9          ret.append($result[i]);
10       return ret.toString();
11     }
12   }
13 }
```

この例では、composition advice を新しく導入している。上の composition advice 内の advice code の記述について以下に述べる。基本的には Hyper/J の記述を流用している。

mergeByName　デフォルトでは、クラスそれぞれのその同じ名前を持つフィールドおよびメソッドを結合を行うことをあらわす。メソッドの結合とは、それぞれのクラスで定義されているメソッドを、前の joinpoint から順番に実行することを言う。新しく結合されたメソッドの戻り値は最後に呼び出されたメソッドの戻り値となる

override void $2.name() with void $1.name()　この記述は、name() メソッドに関しては上記の mergeByName ではなく、２番目の joinpoint の name() を１番目の joinpoint の name() で上書きすることを表す。メ

ソッドの上書きとは、上書きされるメソッドは新しいクラスにおいては呼び出されず、上書きするほうのメソッドのみを呼び出すことを示す。メソッドの戻り値は呼び出されたメソッドの戻り値となる。

**String$composed.toString() { ... }**　この記述は、結合された toString() メソッドの戻り値について定義している。通常の結合方法では最後に呼び出されたメソッドの戻り値が新しいクラスの戻り値になるが、$composed によって新たに定義することで、

4.2　状態遷移に対するアドバイスコードの挿入

2.1.3.2 で話題とした、状態遷移に対するアドバイスコードの挿入を行う Advice Plugin の実装を行った。

図 4.2: Juggler による状態遷移に対するアドバイスの挿入

```
1  public aspect ExampleLoggerBean {
2    stateful() :
3          execution(* *..*.startJuggling(..))
4            ## execution(* *..*(..))
5            ## execution(* *..*.stopJuggling(..)) {
6      before $2 {
7        System.out.println("execution: "
8          + thisJoinpoint.toString());
9      }
10   }
11 }
```

この例では、stateful advice を新たに導入している。stateful advice では、pointcut によって得られる joinpoint の配列のそれぞれの要素が順番に実行されているとき、advice code により指定されているコードを実行する。上の例では、startJuggling() と stopJuggling() が実行されている間のすべてのメソッドの実行についてログをとる。

この記述を応用することで、それぞれの joinpoint に関連付けられた引数などの変数を別の joinpoint の実行時に参照することができる。

4.3 クラスへの新しいフィールドやメソッドの挿入

Open Class の概念を intertype() Advice Plugin として実装した。Observer パターンによる pointcut の有効利用について以下に示す。

図 4.3: Juggler による Observer パターン

```
1  public aspect PointObserable extends ObservableAspect {
2    public pointcut observable() :
3      class(Point);
4    public pointcut subjectChange() :
5      call(void Point.setLocation(int, int));
6  }
7
8  public abstract aspect ObservableAspect {
9    public abstract pointcut observable();
10   public abstract pointcut subjectChange();
11
12   after(Observable observable) returning :
13       subjectChange() && target(observable) {
14     observable.notifyObservers();
15   }
16
17   intertype() : observable() {
18     implements Observable;
19
20     private List observers = new ArrayList();
21
22     public void addObserver(Observer observer) {
23       observers.add(observer);
24     }
25
26     private void notifyObservers() {
27       Iterator itr = observers.iterator();
28       while (itr.hasNext()) {
29         Observer observer = (Observer)itr.next();
30         observer.update();
31
32       }
33     }
34   }
35 }
```

上では、intertype advice を導入している。これは内部に通常の Java のクラスとほぼ同等の宣言をすることで、pointcut によって選択されたクラスに対して、新しいメソッドおよびフィールドを追加することができる。

　また、その pointcut を抽象 pointcut とすることで、ObjservableAspect に対して、Weave 対象となるクラス名を直接記述せずに再利用性を高めている。

第5章　実験

この章では、Juggler コンパイラ自身の性能を確かめる。

5.1　コンパイル速度

Juggler コンパイラが実用的な速度でコンパイルできることを示すために、Java コンパイラである javac、AspectJ コンパイラである ajc および Polyglot を利用した AspectJ コンパイラである abc とを使い、通常の Java コードのコンパイル速度および、すべてのメソッドの前後に対してのコードの挿入を行うアスペクトを一緒にコンパイルしたときの速度について Juggler と比較をおこなった。

コンパイル対象の詳細は以下の通りである。

- 対象: PureTLS Toolkits
- クラス数: 127 個
- 行数: 約 18000 行
- joinpoint 数: 6384 個
- 合成対象箇所: のべ 1052 個

また、実験環境は以下のとおりである。

- CPU: Pentium4 2.6GHz
- Memory: 1GB
- OS: WindowsXP SP2
- JavaSDK: 5.0

結果は次のようになった。単位はすべて ms。

表 5.1: コンパイル速度

コンパイラ	juggler	javac	ajc	abc
アスペクト無し	18601	2972	4443	23101†
アスペクト有り	19019		5317	N/A

javac および ajc と比較すると 5 倍以上の速度低下が見られるが、内部のこれは weaver の性能が悪いのではなく、フレームワークとして使用している polyglot の冗長性が影響していると思われる。実際 polyglot を使用して構築された AspectJ コンパイラである abc の同様の速度低下が見られる。

また、実際に polyglot による解析部、javac によるコンパイル部、Javassist による weaver 部それぞれにかかった時間を計測すると次のようになった。

- polyglot 部: 9047ms
- javac 部: 3313ms
- javassist 部: 1390ms

以上より、本手法そのものは実用的な速度で動作することがわかる。

† 内部エラーにより終了 (バイトコード変換で問題発生)

第6章　まとめ

本研究では、複数個の Joinpoint を明示的に利用するためのモデルである n-Dimensional Joinpoint Model について提案した。n-Dimensional Joinpoint Model では、pointcut 式に対して連結演算子を導入し、joinpoint 集合の配列を抽出することを可能とした。また、advice において配列の要素間の関係を導入させることで、様々なアスペクトに対して一つのモデルで対応することのできる例を示した。

6.1　今後の課題

adivce に対して、joinpoint の扱いをすべて委任してしまっており、それぞれの advice 間の整合性についてなにも考慮されていない。今後は、pointcut 記述による柔軟性を生かしつつ、すべてのアスペクト間で整合性をとれるような表現方法についての言及が必要となる。

参考文献

[1] Chiba, S.: Load-Time Structural Reflection in Java, *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, Springer-Verlag, pp. 313–336 (2000).

[2] Chiba, S. and Nakagawa, K.: Josh: an open AspectJ-like language, *Proceedings of the 3rd international conference on Aspect-oriented software development*, ACM Press, pp. 102–111 (2004).

[3] Douence, R., Fradet, P. and Südholt, M.: Composition, reuse and interaction analysis of stateful aspects, *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, ACM Press, pp. 141–150 (2004).

[4] Eclipse Project: AspectJ, `http://eclipse.org/aspectj/`.

[5] Fraine, B. D., Suvé, D. and Vanderperren, W.: JAsCo: An Aspect-Oriented approach tailored for Component Based Software Development, `http://ssel.vub.ac.be/jasco/index.php`.

[6] IBM alphaWorks: HyperJ, `http://www.alphaworks.ibm.com/tech/hyperj`.

[7] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, Springer-Verlag, pp. 327–353 (2001).

[8] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *Proceedings European Conference on Object-Oriented Programming* (Akşit, M. and Matsuoka, S.(eds.)), Vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, pp. 220–242 (1997).

[9] Lieberherr, K. J. and Orleans, D.: Preventive program maintenance in Demeter/Java, *ICSE '97: Proceedings of the 19th international conference on Software engineering*, ACM Press, pp. 604–605 (1997).

[10] Masuhara, H. and Kawauchi, K.: Dataflow Pointcut in Aspect-Oriented Programming., *APLAS '03: Programming Languages and Systems, First Asian Symposium*, pp. 105–121 (2003).

[11] Masuhara, H. and Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms., *ECOOP '03: Proceedings of the 15th European Conference on Object-Oriented Programming*, pp. 2–28 (2003).

[12] Nystrom, N., Clarkson, M. R. and Myers, A. C.: Polyglot: An Extensible Compiler Framework for Java., *CC '03: Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software*, Springer, pp. 138–152 (2003).

[13] Orleans, D. and Lieberherr, K.: DJ: Dynamic Adaptive Programming in Java, Technical Report NU-CCS-2001-02, College of Computer Science, Northeastern University, Boston, MA (2001).

[14] Ossher, H. and Tarr, P.: Multi-Dimensional Separation of Concerns and The Hyperspace Approach, *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, Kluwer (2000).

[15] Ossher, H. and Tarr, P.: Hyper/J: multi-dimensional separation of concerns for Java, *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, ACM Press, pp. 734–737 (2000).

[16] Sun Microsystems, Inc.: Java 2 Platform, `http://java.sun.com/`.

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

付 録 A　Juggler コンパイラ

```
chimera@wing /usr/local/bin
$ juggler
使い方: juggler <options> <source files>
使用可能なオプションに次のものがあります。
  -verbose
    コンパイラの処理内容の詳細を表示する
  -classpath <path>
    ユーザークラスファイルを検索する位置を指定する
  -cp <path>
    ユーザークラスファイルを検索する位置を指定する
  -sourcepath <path>
    入力ソースファイルを検索する位置を指定する
  -pluginpath <path>
    プラグインクラスを検索する位置を指定する
  -plugin <classname>
    プラグイン登録クラスを指定する
  -d <directory>
    生成されたクラスファイルを格納する位置を指定する
  -encoding <encoding>
    ソースファイルが使用する文字エンコーディングを指定する
  -version
    バージョン情報
  -help
    標準オプションの概要を出力する
```