pointcut に関して高い記述力を持つアスペクト指向言語 Josh

中川 清志　　千葉 滋

東京工業大学情報理工学研究科
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

要 旨

アスペクト指向プログラミング (AOP) とはロギング、同期、永続性などの複数モジュール間に散らばる処理を、アスペクトととしてモジュール化するプログラミング方法である。分離して記述されたアスペクトと、基本モジュールを合成して両方の機能を持つプログラムを作り出す処理を weave と呼び、その処理系を weaver と呼ぶ。アスペクトには合成の仕方 (pointcut) を記述するが、多くの既存 AOP 言語では限られた pointcut しか記述できない。このためプログラマの意図した合成を行えないという問題がある。本稿は pointcut のための指定子を Java で拡張できる AOP 言語 Josh を提案する。Josh は Java 言語で実装されており、プログラマが Java で定義した pointcut を実装に組み込めるように設計されている。このため、必要に応じて新しい pointcut 指定子を定義し、AspectJ のような決まった pointcut では記述できないような処理をアスペクトとしてモジュール化できる。

1　はじめに

オブジェクト指向プログラミング (OOP) では、ロギング・同期・永続性などの処理が複数モジュール間に散らばってしまい、うまくモジュール化できないことが問題視されている。アスペクト指向プログラミング (AOP) はこのような散らばってしまう処理を、アスペクトとしてモジュール化・分離することを目指している。AOP にはシステムを構成している基本モジュールであるクラスと、それを外部から改変するアスペクトという 2 種類のモジュールが存在する。分離して記述されたアスペクトをクラスに合成し、両方の機能を持つプログラムを作り出す処理を weave と呼び、その処理系を weaver と呼ぶ。この処理を言い換えるとクラスにアスペクトを埋め込む処理と言える。AOP を使うことによる利点をロギングを例に採り簡単に説明する。AOP を使わない場合、複数クラスに重複してロギングのコードを記述しなければならず、保守性や拡張性の低いプログラムになってしまう。しかしながら AOP を使う場合は、ロギングアスペクトとしてコードの分離が可能であり、ロギングの内容の変更も容易である。

アスペクトは weaver によりクラスに埋め込まれるが、その際に以下の 2 つを weaver に指定する必要がある。

(α) どこに埋め込むかの指定 (pointcut)

(β) どんなコードを埋め込むかの指定

これら 2 つを簡潔で柔軟に記述できることが必要である。

しかしながら多くの既存 AOP 言語では、(α) の pointcut を指定するための要素、pointcut 指定子が限られているという問題がある。そのためプログラマの要求する場所に、アスペクトを埋め込むことが不可能な場合がある。この理由は pointcut に特化した簡易言語を採っていることにある。これには簡潔に pointcut を記述できるという利点があるが、記述力が低くなるという弊害もある。

本稿では pointcut に関して高い記述力を持つ AOP 言語 Josh を提案する。Josh は簡潔で柔軟な pointcut 記述を目指している。簡潔さを目指すため Josh は従来の言語と同じようにアスペクトに特化した言語になっている。そして Josh のアスペクトは、weave 前にひとまず Java のコードへソースコード変換される。この際アスペクト内の pointcut 指定子は、Java の boolean 型のメソッドで表現される。そのためプログラマが新たに boolean 型のメソッドを定義すれば、それを自由に pointcut 指定子として使うことが可能になっている。Java で pointcut 指定子を記述するため、簡易言語にはできない柔軟な pointcut 指定子の定義が可能である。

pointcut 指定子の定義には Javassist[3, 9] を使う。

Javassist は構造リフレクションのためのライブラリである。Javassist などのリフレクションを使えば AOP を実現することは可能だが、煩雑でわかりにくいプログラミングをしなければならない。Josh はまた、リフレクションの強力さを保ちつつ簡潔なプログラミングを可能にしているともいえる。

以後 2 章では既存 AOP 言語での pointcut の問題点を喚起し、3 章では AOP 言語 Josh の概要について述べる。4 章において、喚起した問題を解決するための pointcut 定義について述べ、最後に 5 章で本稿をまとめる。

2 既存 AOP 言語の pointcut の問題点

AOP の言語やシステム [1, 2, 4, 6] は多数存在するが、共通した問題点がある。それはアスペクトの埋め込み先の指定 – pointcut が限られていることである。本章ではまず AOP の基本的な概念を説明をし、その次に問題点である pointcut について述べる。

2.1 AOP における基本的な概念

ここで AOP に必要な概念・語句を説明する。これらの概念は AOP 共通のものであるが、多くの言語やシステムは AspectJ[4, 5] の語彙を踏襲している。AspectJ は Java を拡張した汎用 AOP 言語であり、基本モジュールである Java クラスと独自言語で記述したアスペクトを weave する言語である。本稿でも同様に AspectJ の用語を用いる。

join-point プログラムの動作の原子的な構成要素である。例えばメソッド呼び出し、メソッド実行、フィールド参照、インスタンス生成、などがある。

pointcut 特定の条件を満たす join-point を抽出することである。例えば 「getX という名の」メソッド呼び出し、「int 型の」フィールド参照、「Point クラスの」インスタンス生成、「Point クラス内での」全 join-point、などと抽出する。抽出する条件には、その join-point の情報を使うことができる。これは join-point の情報に基づいての選別とも言える。

アドバイス pointcut された join-point に対して新たなコードを埋め込むこと、埋め込まれるコードのことである。アドバイスは join-point の前 (before) または後 (after) に埋め込むことができる。若しくは join-point の動作と置き換えることができる。アドバイスは例えば Java のような汎用言語で記述されることが多い。またアドバイス内で join-point の情報を参照することができる。

イントロダクション 既存クラスに新たな要素を追加することである。フィールド、メソッド、コンストラクタの追加などができる。また既存クラスの階層構造を変えることも含まれる。これはスーパクラスの変更や、新たなインタフェースの継承などを指している。

アスペクト 関連有るアドバイスとイントロダクションの集まりのことである。

まず AspectJ を使ったイントロダクションをコード例を示して説明する。

```
1 int Point.x;
2 public int Point.getX() {
3   return x;
4 }
```

1 行目はフィールドのイントロダクションである。int 型である x という名前のフィールドを Point クラスに追加する。2-4 行目はメソッドのイントロダクションである。public 修飾子で int 型、getX という名前の引数のないメソッドを Point クラスに追加する。

次にアドバイスのコード例を示す。

```
1 pointcut setting(): set(int Point.x);
2 before() : setting() {
3   System.out.println
4     ("-- setting x in Point --");
5 }
6 pointcut PointOrLine()
7   : within(Point) || within(Line);
```

1 行目は setting という名前の pointcut を宣言する。set はフィールド書き込みの join-point を抽出するための pointcut 指定子であり、括弧内は条件を指す。この場合は「int 型、Point クラス内、x という名」という条件のフィールド書き込みを抽出する。2-5 行目はアドバイスである。setting() が抽出する join-point の実行前 (before) にログを書くアドバイスである。6-7 行目も pointcut の宣言である。within

指定子は join-point の種類ではなく位置を指す。この例では Point クラス内での全 join-point、または Line クラス内での全 join-point を抽出する。7 行目のように論理積・論理和を使うこともできる。このように AspectJ では pointcut 指定子と条件 (引数) のペアを組み合わせることにより pointcut を構成していく。

2.2 pointcut の問題点

pointcut は直感的で容易に記述できる必要がある。そのため多くの言語では AspectJ の例のように pointcut 記述に特化した独自の簡易言語を採りアスペクトのプログラミングを促進している。簡易言語とはここでは、組み込み (built-in) 要素の組み合わせでしか記述できない言語のことを指している。例えば AspectJ では call や within などの組み込み指定子しか使うことができず、新たな pointcut 指定子を新しく定義することはできない。そのため意図した pointcut を記述できないことがある。

以下に AspectJ では記述できない例を示す。

- 例 1:画像エディタにおけるスクリーンの更新
  ある画像要素を表すクラスがあり、repaint() メソッドを使ってスクリーンを最新の状態に保つとする。repaint() を呼ぶ必要があるのは、repaint() メソッド内で参照しているフィールドに、書き込み処理が行われた場合である。そのため全てのメソッドの実行の後に repaint() を呼ぶ必要はない。この場合外部からこのクラスのメソッドを呼んだときに、そのメソッド内で repaint() に関係があるフィールドに書き込んでいるかどうかを判断する必要がある。もしそうならばそのメソッド呼び出しの後に repaint() も呼べばよく、そのメソッドが repaint() と関係がないならば repaint() を呼ぶ必要はない。このようにメソッド呼び出しを複雑な条件でふるい分ける、つまり pointcut する必要がでてくることがある。

  AspectJ ではこのような pointcut を記述できない。もし実現しようとするならば、全てのフィールド書き込みの後に repaint() を呼ぶという解決方法になるであろう。しかしながらこの解決方法だと、途中経過が見えてしまう、ちらつくなどの問題がでてきてしまう。

- 例 2:排他制御
  マルチスレッドにおいてあるフィールドの値の一貫性を保つために排他制御をしたいとする。この場合ただ単にフィールド書き込みを捉えて、synchronized ブロックでくくったとしても解決にはならない。なぜなら個々の書き込み命令が synchronized になるだけで、オブジェクトのロックは複数の命令にまたがってかかるわけではないからである。この解決には「あるフィールドにアクセスしている全ての」メソッドを synchronized にすればよい。しかしながら「あるフィールドにアクセスしている全ての」メソッド、という条件を提示することができない。

これらが不可能なのは pointcut の際に参照できる join-point の情報が少なく、また拡張できないからである。たいていの場合は修飾子、型、名前、引数などの情報が手にはいればそれで済むが、上記のようにさらに踏み込んだ情報が必要な場合もある。仮に上記の条件を提示できるような pointcut 指定子が組み込まれたとしても、一時的な解決にしかならない。なぜならプログラマの全ての要求を組み込むことは難しいからである。つまり組み込み要素の組合わせしか提供しない、簡易言語による pointcut では不充分といえる。その他の AOP 言語においても簡易言語を採っている限り同様の問題がある。

この問題を解決するために論理型言語による pointcut が提案されている。Brichau[7] らは pointcut に prolog 風言語を、アドバイスに Smalltalk を使ったシステム Soul/AOP を試作している。このシステムでは論理型言語による柔軟な pointcut 指定子を新しく定義可能である。Gybels[8] も同様に論理型言語による pointcut を提案している。しかしながら使い慣れない論理型言語を習得しなければならず、慣れるまでに多くの努力が必要である。

3 Josh の概観

この章ではアスペクト指向言語 Josh の記述方法と、実行までの主な流れについて述べる。Josh を使えば 2 章で述べたような問題を解決できるが、その具体的な方法は 4 章で述べる。

図 1 は Josh の大まかな実行の流れを示している。この中でプログラマが記述する必要があるのはア

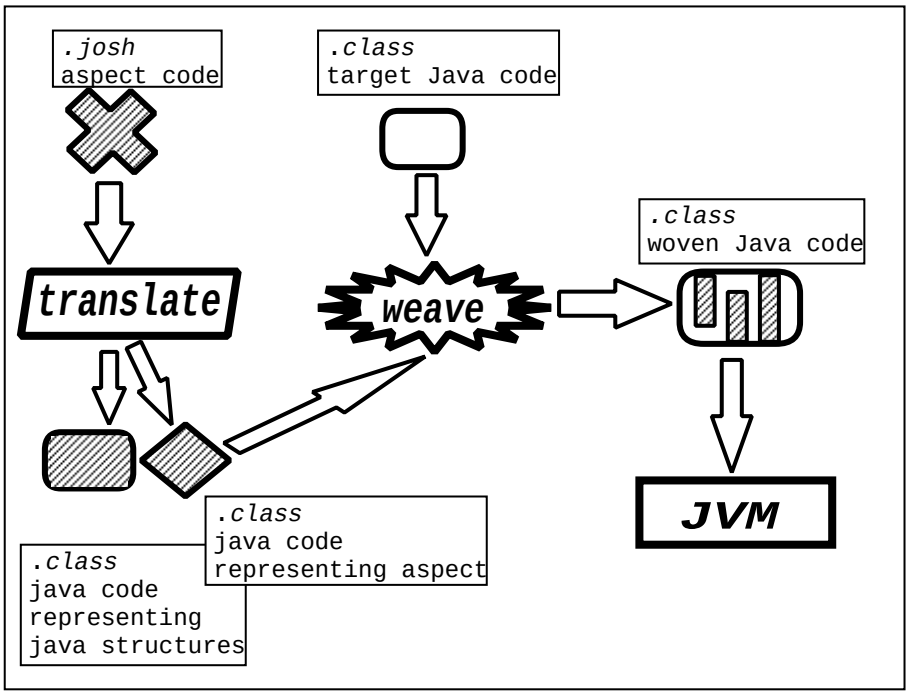

図 1: Josh 実行の主な流れ

スペクトである.josh ファイルだけである。それをコード変換することにより通常の Java ソースコードにし、さらに javac などでコンパイルしてアスペクトを表現するクラスファイルを手にいれる。こうしてできたアスペクトクラスと合成対象とする Java クラスを weave し、両方の機能を持つ Java クラスを生成する。生成されたクラスファイルの実行には特別な環境は必要なく、通常の JVM(Java Virtual Machine) があればよい。

具体的な Josh アスペクトコードの例を図 2 に示す。このプログラムは簡単なロギング処理のための

```
 1 public aspect LogAspect {
 2   static int count = 0;
 3   static void logging() {
 4     System.out.println
 5       ("before set [x]" + (count++));
 6   }
 7   pointcut setting :
 8     set("int", "Point", "x");
 9   before : setting {
10     LogAspect.logging();
11   }
12 }
```

図 2: Josh のアスペクトコード

アスペクトである。7-8 行目において pointcut 宣言をする。pointcut 指定子は set であり、これはフィールド書き込みの join-point を捉える。引数が条件をあらわしているので、この例だと int 型で Point クラスに属する x という名前のフィールド書き込みを捉える。9-11 行目がアドバイスであり、指定した pointcut の前 (before) にロギングコードを埋め込む。また 2,3-5 行目のように通常の Java のフィールドやメソッドの記述も可能である。そしてこれらをアドバイス中から参照することができる。現在の実装では static なフィールド、メソッドに限定しており、またクラス名を明示しなければならない。

Josh のアスペクトはトランスレータにより Java ソースコードへ変換される。その際に二つのコードへ分けられる。一方はアスペクト内に記述されていた通常の Java の要素を持つコード、もう一方は pointcut とアドバイスの情報を持ち、weaver に渡されるコードである。例えば先ほどのアスペクトコードは図 3 の 2 つになる。

これらのコードにおいて JoshAspect、JoshExprEditor は Josh が与える、アスペクトのための基本クラスである。また JoshExprEditor は ExprEditor のサブクラスであり、ExprEditor 及び FieldAccess は Javassist のクラスである。Javassist[3, 9] は構造リフレクションのためのライブラリであり、クラス構造の変更や各種の演算 (フィールド書き込み、メ

```
/** Java の要素をそのまま保持するクラス **/
 1 public class LogAspect extends JoshAspect {
 2   static int count = 0;
 3   static void logging() {
 4     System.out.println("before set [x]");
 5   }

/** weaver に渡されるクラス **/
 6  public class LogAspectEditor
 7      extends JoshExprEditor {
 8    public void edit(FieldAccess f) {
 9       if (set(f, "int", "Point", "x"))
10         f.replace( "System.out.println    \
11                     (\"before set [x]\"); \
12                     $_ = $proceed($$);" );
13    }
14  }
```

図 3: ソースコード変換されたアスペクトのコード

ソッド呼び出しなど) の捕捉を可能にする。Josh は Javassist を使ったソースコードへアスペクトを変換し、AOP を行う。ExprEditor クラスは演算の挙動を捉えるためのクラスである。この例では 3 行目よりフィールド参照 (FieldAccess) を対象にしていることがわかる。また 5 行目の replace メソッドは演算の挙動に対して、新たな動作を埋め込むためのものである。これを使うことによりアドバイスを実現できる。注目すべきところは 4 行目である。アスペクトファイルに記述した pointcut がほぼそのままで、この個所に再現されている。この set というメソッドが真ならばアドバイス (replace) 実行、偽ならば実行しないというコードになっており、pointcut しているといえる。

このコード変換のためのクラス、ExprEditor を使って weave を行う。weaver の仕事は weave の対象とするターゲットクラスのコードを変換することである。それにはまずターゲットクラスのコードを調べ、そして ExprEditor のコード変換対象と一致するならばコード変換をすればよい。weaver のコードは以下のようになる。

```
 1 public class JoshWeaver {
 2   public void weave(CtClass target,
 3                     JoshExprEditor editor) {
 4     CtMethod[] methods =
 5       target.getDeclaredMethods();
 6     editor.currentClass = target;
 7     for (int i=0; i < methods.length; i++) {
 8       editor.currentMethod = methods[i];
 9       methods[i].instrument(editor);
10     }
11     target.writeFile();
12   }
13 }
```

これも Javassist を使ったコードになっている。CtClass, CtMethod はそれぞれクラス、メソッド構造を表すクラスであり、4-5 行目でターゲットクラスに属する CtMethod を全て取り出している[1]。instrument(ExprEditor) はコード変換を行うメソッドである。引数の ExprEditor で定義されたコード変換の指針に従って、CtMethod オブジェクトのコードを変換する。最後に 8 行目で変換後のファイルを実際にディスクに書き込む。

Josh は以上のように Javassist のフロントエンドとして動いている。直接 Javassist を使ってアスペクトと同等の機能を持つクラスを作ることは可能であるが、アスペクトに特化した Josh の文法があることは有意だといえる。なぜなら Java でアスペクトを書くのは不可能ではないが不自然だからである。例えば C 言語のような手続き型言語を使ってオブジェクト指向風なプログラミングも不可能ではない。しかしオブジェクト指向で設計されたシステムを C 言語で実装するのは不自然である。熟練プログラマであったり細かな技法をたくさん用いたりすれば可能かもしれないが、全てのプログラマにそれらを要求できない。また C 言語のコードを見てもオブジェクト指向の設計を読み取れないという弊害もある。したがって AOP に特化した言語を使ってアスペクトの構造が明確になるようにプログラムを書けることは重要であるといえる。リフレクションを多用したプログラムは煩雑になりやすく、簡潔であるとはいいにくい。

アドバイス内での動的な値の参照

アドバイスのコードでは、join-point に関する動的な情報を参照することができる。それには'$' 記号を用いた特別な語句を使う。以下がその例である。$で始まるいくつかの語句には動的な値が束縛されている。

```
1 aspect LogAspect {
2   pointcut setting : set("int", "Point",
3                          "x");
4   before : setting {
5     System.out.println("before set [x]" +
6         "Point:" + $0 + ", arg:" + $1);
7   }
8 }
```

この例だと$0 はフィールド書き込みの対象となった Point クラスのインスタンスを表しており、$1 はそ

[1] コンストラクタへの変換処理はメソッドと同じようにできるが、本稿を通して省いている

のとき書き込もうとした値を表す。

4 pointcut の拡張

従来の AOP 言語には 2 章で述べたような問題があった。しかし Josh ではプログラマが pointcut 指定子を新たに定義できるので、この問題を解決できる。本章ではその定義方法について具体的に述べる。

4.1 boolean 型メソッドによる pointcut

Josh の poitncut 指定子の実体は boolean 型のメソッドである。これは図 2 で記述された pointcut がほぼそのまま図 3(下) の if 文の中に入っていることからもわかる。Java のメソッドであるためプログラマは pointcut 指定子を新たに定義することが可能であり、Java の記述力を活かした柔軟な pointcut 定義が可能である。しかしながら全ての pointcut 指定子を定義する必要はなく、図 2 の例の set のように汎用的な pointcut 指定子は組み込みで提供している。そのため複雑な pointcut をしたい場合にのみ必要に応じて定義すればよい。表 1 は組み込み済の pointcut 指定子の一覧である。これらは JoshExprEditor クラスで宣言されており、ソース変換後のクラスは JoshExprEditor クラスを継承するので特別な宣言をせずに使える。within 及び withincode は直接的に join-point を特定するのではなく、発生場所を特定するものである。例えば

```
within("Circle") && set("int Point.getX()")
```

とすると Circle クラス内での getX() メソッド呼び出しを捉えることになり、他のクラス内での getX() メソッド呼び出しには関与させないという様にすることができる。また'&', '|', '!' などの論理演算子と括弧'()' を使うことができる。

指定子名	対象とする join-point	使用例
set	フィールド書き込み	set("int", "Point", "x")
set	フィールド書き込み	set("int Point.x")
get	フィールド読み込み	get("int Point.y")
call	メソッド呼び出し	call("int Point.getX()")
initialize	インスタンス生成	initialize ("public Point.new(int, int)")
within	join-point の 発生場所指定 (クラス)	within("Point")
withincode	join-point の 発生場所指定 (メソッド)	withincode("public * *.*(..)")
その他	'&', '|', '!' の論理演算子、および括弧'()' の使用	

表 1: 定義済の基本的な pointcut メソッド

4.2 新たな pointcut 指定子の定義

Josh の指定子は Java の boolean 型のメソッドであるため自分で新たな boolean 型のメソッドを定義すれば、pointcut ととして使用できる。しかし約束ごとが 2 つある。1 つめは引数についてである。第一引数は、その pointcut メソッドが扱う join-point をあらわす Javassist のクラスでなければならない。Josh の処理系はこの第一引数の型を見て、定義された pointcut がどの join-point を処理するものか認識する。2 つめとしてメソッドは static でなくてはならない。表 2 は第一引数になりうる、join-point をあらわすクラスである。上から 5 つめの Instanceof までが動作に関する join-point、残りの 2 つが場所に関する join-point である。場所に関する join-point とは、例えば CtClass を第一引数にした場合には、そのクラス内でのメソッド呼び出し、フィールド参照、インスタンス生成、キャスト式、instanceof 式の全ての join-point を捉える。

join-point のクラス名	対象 join-point
MethodCall	メソッド呼び出し
FieldAccess	フィールドアクセス
NewExpr	インスタンス生成
Cast	キャスト式
Instanceof	instanceof 式
CtClass	クラス内の全 join-point
CtMethod	メソッド内の全 join-point

表 2: 抽出対象となる join-point のクラス名

メソッドの定義はアスペクト内でも Java のクラス内でもどちらでもよい。以下は組み込みの pointcut メソッドの一つ、 set を定義しているコードである。第一引数が FieldAccess であるためフィールド参照を対象にしていることがわかる。第二引数以降は pointcut の条件となっている。

```
 1 static boolean set(FieldAccess f, String type,
 2                     String dec, String name) {
 3   CtField cf = f.getField();
 4   String cf_name = cf.getName();
 5   String cf_type =
 6     cf.getType().getName();
 7   String cf_dec =
 8     cf.getDeclaringClass().getName();
 9   if (f.isWriter() &&
10       cf_name.equals(name) &&
11       cf_type.equals(type) &&
12       cf_dec.equals(dec))
13     return true;
14   else
15     return false;
16 }
```

まず 3 行目で参照しようとしているフィールドを表す CtField オブジェクトを入手する。そしてそのフィールドの名前、型、属するクラスなどを条件である引数と比べる。9 行目ではこのフィールド参照が書き込み処理かどうかを調べる。これらの処理をして真を返すか偽を返すか、つまりこの join-point を抽出するか否かを計算する。メソッドの中身を工夫することによりワイルドカードへ対応させることも可能である。

アスペクトとしての pointcut 使用時と、メソッドとしての定義時とでは引数の数が異なっている。例えば図 2 の set pointcut 使用時には

```
pointcut setting : set("int", "Point", "x");
```

と引数の数が 3 つであり定義時の 4 つより 1 つ少ない。これは pointcut の使用時には join-point に関する引数が入っていないからである。pointcut 指定子は定義時に対象とする poitncut を特定しており、アスペクト内での使用時には対象とする join-point 全てに対して動作する。そのため引数として特定の値を渡す必要はない。この例の set 指定子のような組み込み指定子の場合は必要がないが、新しく定義した pointcut メソッドを使う場合にはクラス名を明記する必要がある。

複雑な pointcut の定義例

ここでは 2.2 章の例 2 において提起した問題の解決を示す。従来の AOP 言語では「あるフィールドにアクセスしている全ての」メソッド呼び出しを特定することができなかった。Josh を使えばそのような join-point を抽出する pointcut 指定子を定義することができる。以下に callWithWrite という名前の新しい pointcut 指定子のコードを示す。既にシグネチャに応じてメソッド呼び出しを抽出する call 指定子は、定義してあるとする (表 1 参照)。callWithWrite も同様にメソッド呼び出しを捉えるが、call より引数が一つ多い。その引数があらわす名前のフィールドに、呼ばれたメソッド内で書き込み処理が起こるかどうかを調べるために使われる。

```
1 static boolean callWithWrite(MethodCall m,
2     String pattern, String fname) {
3   if(!call(m, pattern)
4     return false;  /*まずシグネチャを比べる*/
5
6   CtMethod cm = m.getMethod();
7     /*呼び出されたメソッドを表すオブジェクト*/
8   CtField[] fields = writtenFields(cm);
9     /* writtenFields() は定義済 */
10   for (int i=0; i < fields.length; i++)
11     if (fields[i].getName().
12         equals(fname))
13       return true;
14   return false;
15 }
```

3-4 行目では call メソッドを呼ぶ。これはこのとき呼ばれたメソッドのシグネチャと、引数の”pattern”を比較する。8 行目では呼ばれたメソッド内で書き込み処理が起こる全てのフィールドを得る。そしてそれらのフィールドの中に fname と同じ名前のフィールドがあるかを 10-13 行目で調べる。該当するフィールドが無い場合にのみ 14 行目に至り、false を返す。以上のように定義した pointcut メソッドのアスペクトの中での使用例は以下のようになる。ただしメソッドは Sample というクラス (アスペクト) 内で定義されたとする。

```
pointcut write_x_method :
  Sample.callWithWrite("public Point.*(..)", "x");
```

このように pointcut メソッドを定義・使用することにより、従来はできなかった pointcut をすることができる。

5 まとめ

本稿では、簡潔で柔軟な pointcut を記述できる AOP 言語 Josh について述べた。既存の AOP 言語は簡潔でアスペクトの構造をとらえやすいものであったが、組み込み (built-in) 要素の組合わせでしか pointcut を記述できないという弊害も持っていた。これは例えば AspectJ, Composition Filter[1], Hyper/J[6] に関していえる。Composition Filter はオブジェクト間のメッセージの受け渡しをフィルタリングするものである。新しくフィルタを作成することにより、フィルタの適用先及びフィルタリングの際の動作をかえることができる。しかしフィルタの構成要素は限られている。Hyper/J はある観点で Java のクラスからバイトコードをぬきとり、それを他のクラスに追加したりするものである。どこからぬきとるのか、どこに追加するかを宣言的な言語で指定することができる。しかし限られた語句の組合わせによる指定しかできない。その他にも XML(eXtensible Markup Language) でアスペクトを記述する AOP 言語・システムも有るが、同じ問題を抱えている。

Josh は 2 つの要件、簡潔さと柔軟さを持った AOP

言語を目指した。AspectJ のような簡潔な構文を保ちつつ、プログラマの必要に応じて pointcut を拡張できるようにした。Java のメソッドで pointcut を記述できるため柔軟な記述が可能であり、従来の AOP 言語ではできない pointcut を Josh でしてみせた。

現在の Josh はアスペクトの構成要素であるイントロダクションが未実装である。イントロダクションについても追加先クラスの指定は必要であり、柔軟な指定ができることは有意であると我々は考える。文法の設計と合わせて今後の課題である。

参考文献

[1] Bergmans, L. Aksit, M. and Tekinerdogan, B.: Aspect composition using composition filters, *In Software Architectures and Component Technology: The State of the Art in Resarch and Practice*, pages 357-382. Kluwer Academic Publishers, 2001.

[2] Pawlak, R. Seinturier, L. Duchien, L. Florin, G.: JAC : A Flexible Framework for AOP in Java, *Reflection'01*

[3] Chiba, S.: Load-time structural reflection in Java, *ECOOP2000*, LNCS1850, Springer-Verlag, pp.313-336(2000).

[4] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. : An Overview of AspectJ, *ECOOP2001*, LNCS2072, Springer-Verlag, pp.327-353(2001).

[5] Kickzales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *ECOOP'97*, vol.1241, Springer-Verlag, pp.220-242(1997).

[6] Ossher, H. and Tarr, P.L.: Hyper/J: multi-dimensional separation of concerns for Java, *ICSE2000*, pages734-737, 2000.

[7] Brichau, J. Mens, K. and Volder, K. D.: Building Composable Aspect-Specific Languages with Logic Metaprogramming, *GPCE2002*, LNCS2487, Springer-Verlag, pp.93-109(2002).

[8] Gybels, K.: Using a logic language to express cross-cutting through dynamic joinopints, *Workshop on AOSD2002*.

[9] 千葉滋, 立堀道昭、Java バイトコード変換による構造リフレクションの実現、情報処理学会論文誌, 42 巻 11 号, pp.2752-2760, 2001 年 11 月

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