アスペクトのコメントをクラスのコメントに織り込む AspectJ 用の改良 Javadoc の提案

堀江 倫大　　千葉 滋
Michihiro HORIE　　Shigeru CHIBA
東京工業大学 情報理工学研究科 数理・計算科学専攻
{horie,chiba}@csg.is.titech.ac.jp

本論文は、アスペクトのコメントをクラスのコメントに自動的に織り込み、API ドキュメントを作成する AspectJ 用の改良 Javadoc を提案する。ライブラリやフレームワークを開発する際は、それらの API ドキュメントも作成しなければならない。しかし、AOP でライブラリやフレームワークの開発を行うと、プログラムがクラスとアスペクトに分離するので、従来の Javadoc のようなツールでは API ドキュメントをコメントの形でうまくプログラム中に埋め込めない。提案する Javadoc はこの問題に対処する。

1　はじめに

一般的に、ライブラリやフレームワークを開発する際は、それらの API（Application Programming Interface）ドキュメントも同時に作成する必要がある。ユーザーは API ドキュメントを読み、ライブラリやフレームワークを実際に使用する。Java で開発を行う場合、 API ドキュメントを作成するために、Javadoc [9] のようなツールを利用することが多い。Javadoc はソースコード中のドキュメンテーションコメント（/** ... */で囲まれたコメント、以下、API コメントと呼ぶ）として埋め込まれたドキュメントを収集し、 HTML フォーマットで Java API ドキュメントを自動的に作成する。

しかし、アスペクト指向プログラミング（AOP）でライブラリやフレームワーク開発を行うと、従来の Javadoc では API コメントをうまくプログラム中に埋め込めない。AOP では、クラスとアスペクトに実装が分離するため、API コメントもそれぞれの実装に沿って分離して書くのが理想的であるが、従来の Javadoc では分離した API コメントを API ドキュメント作成時にクラスごとにひとまとめにすることができない。AspectJ [6] 用のドキュメント作成ツールに Ajdoc [11] があるが、Ajdoc を用いても不十分である。Ajdoc は、必要な情報を公開クラスのドキュメントに必ずしも反映させられるとは限らない。

クラスとアスペクトに API コメントを分離して書くのではなく、全てをクラスの API コメントとして書く方法もある。しかし、このような方法では、API コメントのモジュール化が達成できず、API コメントの保守性が低下する。

Ajdoc や従来の Javadoc が持つ問題を解決するために、我々はアスペクトの API コメントをクラスの API コメントに織り込んでから API ドキュメントを生成する AspectJ 用の改良 Javadoc を提案する。本ツールを用いると、クラスとアスペクトの API コメントを分離しておき、API ドキュメントの作成時に公開クラスにアスペクトの API コメントを本ツールによって自動的に織り込むことができる。そのために、我々は、API コメント専用のポイントカット言語を用意した。この言語機構を用いることにより、アドバイスやポイントカットの API コメントをうまくモジュール化することができる。

以下、2 章では Ajdoc や従来の Javadoc の持つ問題点について述べる。3 章では、その問題を解決するために我々の提案する改良 Javadoc について説明する。4 章では、より具体的な例を用いて本ツールの有用性を確かめる。5 章では、関連研究について取り上げ、6 章で本論文をまとめる。

2　Ajdoc や従来の Javadoc の利用

AOP では、アスペクトの API コメントも API ドキュメントに反映させなければならない。公開クラスの API コメントに公開クラスに関することだけを書いただけでは完全な API ドキュメントにならないからである。以下では、Ajdoc や従来の Javadoc を用いた API コメントの書き方とその問題点について述べる。

2.1 Ajdoc の利用

AspectJ 用の API ドキュメント作成ツール Ajdoc を用いると、クラスとアスペクトに別々に API コメントを記述することができる。作成された API ドキュメントには、ポイントカットによって選択されたメソッドにアドバイスによって振る舞いが拡張されることを示す表示が行われる。しかし、この機能だけでは不十分である。

2.1.1 内部アスペクトがある場合

公開クラスに影響を与えるアスペクトがフレームワーク内部にあるとき、Ajdoc はその影響を必ずしも API ドキュメントに反映させられるとは限らない。公開クラスのメソッドの実装に基づいた API コメントだけではメソッドの正確な振る舞いが分からないため、それでは不十分である。

例えば、図 1 は J2SE 5.0 ライブラリ [8] にある Stack クラスの実装の一部をアスペクトとして独自に分割したものである。スタックのトップにあるオブジェクトを見る Stack クラスの peek メソッドがある。StackChecking アスペクトは、この peek メソッドに事前条件を加え、スタックのサイズが 0 のときには RumtimeException の EmptyStackException を投げる。API コメント中の @return や @exception は通常の Javadoc で使用可能なタグであり、API ドキュメントにメソッドの戻り値や例外に関する説明を加えるときに使用する。図 2 は、Ajdoc を用いて API ドキュメントを作成した例である。Stack クラスの peek メソッドは、StackChecking アスペクトの before アドバイスによって振る舞いが拡張されることが分かる。StackChecking アスペクトの before アドバイスの API ドキュメントを見ると、スタックのサイズが 0 のときに EmptyStackException が投げられることが分かる。

しかし、peek メソッドの振る舞いが拡張されることが API ドキュメントに表示されるだけでは不十分である。peek メソッドは Stack クラスの別のメソッド pop 内でも呼び出されている。そのため、pop メソッドの API ドキュメントにも、スタックのサイズが 0 のときに例外が投げられることが表示されなければならない[1]。しかし、Ajdoc ではこれを実現でき

[1] J2SE 5.0 ライブラリには、pop メソッドの API ドキュメントに EmptyStackExcpetion が起こりうる例外として明記されている。

```
public class Stack<E> extends Vector<E> {
          :
    /**
     * Removes the object at the top of
     * this stack and returns that object
     * as the value of this function.
     *
     * @return The object at the top of
     *     this stack (the last item of
     *     the <tt>Vector</tt> object).
     */
    public synchronized E pop() {
        E obj = peek();
        removeElementAt(size() - 1);
        return obj;
    }

    /**
     * Looks at the object at the top of
     * this stack without removing it
     * from the stack.
     *
     * @return the object at the top of
     *       this stack (the last item of
     *       the <tt>Vector</tt> object).
     */
    public synchronized E peek() {
        return elementAt(size() - 1);
    }
          :
}

aspect StackChecking {
    /**
     * @exception EmptyStackExcpetion
     *            if this stack is empty.
     */
    before(Stack s): execution(* Stack.peek())
                     && this(s) {
       if (s.size() == 0)
          throw new EmptyStackException();
    }
}
```

図 1: Stack クラスと StackChecking アスペクト

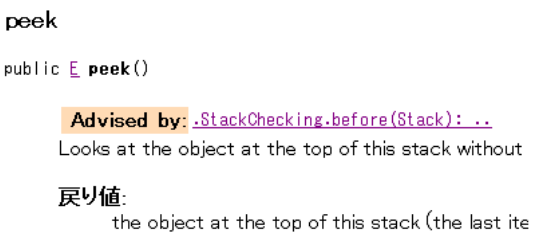

図 2: Ajdoc によって生成した Stack クラスの API ドキュメントの一部

ない。Ajdoc は、どのメソッドやクラスがアスペクトに拡張されるかという観点に立ち API ドキュメント作成するツールだからである。

また、StackChecking アスペクトはライブラリの内部に存在するアスペクトであるため、その存在をライブラリのユーザーに教える必要性は必ずしもない。ユーザーはこのアスペクトを直接に利用することはないからである。しかし、Ajdoc では内部アスペクトも必ず API ドキュメントとして作成されてしまう。

2.1.2 公開アスペクトがある場合

フレームワーク中に公開アスペクトがあるとき、Ajdoc はその影響を公開クラスの API ドキュメントに反映させることができないことがある。[12] で提案されているように、公開アスペクトは、抽象ポイントカットを持つ抽象アスペクトとしてフレームワークのユーザーに提供されることが多いため、API ドキュメントの作成時に公開アスペクトの実装がクラスに織り込まれないためである。Ajdoc は、織り込みが実際に行われていなければ、API ドキュメントにアスペクトの API コメントの表示を行わない。

しかし、公開クラスの API ドキュメントには公開アスペクトに関する記述が必要な場合もある。フレームワークのユーザーがサブアスペクトを定義しポイントカットをオーバーライドすると、公開アスペクトの影響が公開クラスに及ぶ可能性があるからである。

例えば、図 3 は図形エディタフレームワークの実装の一部である。このフレーワークでは、ユーザーが図形を定義したり、その図形に独自の動作を追加することができる。公開クラス Figure の registerFigure メソッドは、ユーザーが定義した図形をエディタに登録する。また、公開アスペクト UndoFunction は、figures ポイントカットで指定された図形に Undo の機能を追加する。フレームワークのユーザーは、UndoFunction のサブアスペクトを定義し、figures ポイントカットをオーバーライドして利用する。例えば、下のように定義すると、直線を表す Line クラスや楕円を表す Ellipse クラスに対し Undo の機能を実行できるようになる。

```
public aspect UndoableFigure
                        extends UndoFunction{
  protected pointcut figures(): target(Line) ||
                                target(Ellipse);
}
```

```
public class Figure {
     :

  /**
   * Registers the figures to a draw editor.
   */
  protected void registerFigure(DrawEditor editor) {
     editor.addFigure(this);
  }
}

public abstract aspect UndoFunction {

  protected abstract pointcut figures();

  /**
   * Adds the undo function to figures
   * that will be selected by pointcut
   * <code>UndoFunction.figures</code>.
   */
  void around(Figure fig, DrawEditor editor) :
        tools() && this(fig) && args(editor) &&
        execution(void Figure.
                    registerFigure(DrawEditor)) {
     editor.addFigure(new UndoableFigure(fig));
  }
}
```

図 3: フレームワーク内の公開アスペクト ToolUndo

このような場合は、UndoFunction の figures ポイントカットによって指定された図形に Undo の機能が加わることを公開クラス Figure の registerFigure メソッドの API ドキュメントに表示すべきである。そうしなければ、registerFigure メソッドの正確な振る舞いをフレームワークのユーザーは理解できない。しかし、 Ajdoc では、UndoFunction の around アドバイスの API コメントは、 registerFigure メソッドの API ドキュメントから読むことができない。

2.2 素朴な解決策

このような問題を避けるため、従来の Javadoc や Ajdoc を用いた場合も、アスペクトの振る舞いを事前に考慮し、全ての API コメントを公開クラスにまとめて記述する方法が考えられる。しかし、そのような方法では、 API コメントのモジュール化ができない。メソッドの実装と API コメントの内容が一致しないため、プログラムの仕様変更などの際に、API コメントをいちいち変更しなければならない。

例えば、図 1 の例では、 peek メソッドや pop メソッドの API コメントに、例外として EmptyStackException が投げられ得ることを記述しておくことに

なるが、これでは API コメントのモジュール化が達成できていない。

また、AspectJ では、ワイルドカードを使用してポイントカットを定義することで、複数のジョインポイントを一度に選択することができる。例えば、以下のポイントカットは、ワイルドカードを使用して、Servlet クラスのサブクラスの ”do” から名前が始まる全てのメソッドにアドバイスが織り込まれる。このとき、該当するすべてのメソッドの API コメントを書く際に、いちいちアドバイスの振る舞いを考慮しなければならない。

```
execution(void Servlet+.do*(..))
```

しかし、仕様変更により、仮にこのポイントカットを削除することになった場合に、該当する全てのメソッドの API コメントからアドバイスに関する部分を削除しなければならない。これは、非常に非効率的な作業である。

3 提案

我々は、アスペクトの API コメントをクラスの API コメントに織り込みながら API ドキュメントを作成する AspectJ 用の改良 Javadoc を提案する。本ツールを用いると、アスペクトとクラスの挙動についての仕様を API コメントとして、それぞれに対応するソースファイルに分けて書くことが可能になる。そのように書いても、アスペクトの影響を個々の公開クラスのドキュメントの記述に反映した API ドキュメントを生成できる。

本ツールは、Ajdoc 同様、API ドキュメントを生成する際、アスペクトが織り込まれるジョインポイントに対応する API コメントに、そのアスペクトの API コメントを織り込む。このため、例えばジョインポイントがメソッド呼び出しであるときは、そのメソッドの API コメントにアスペクトの API コメントを連結したものが API ドキュメントの記述として用いられる。

一方、Ajdoc とは異なり、本ツールは内部アスペクト用の API ドキュメントの作成はおこなわない。ライブラリやフレームワークの API ドキュメントを作成する観点から考えると、内部アスペクトの存在は実装依存の部分に属し、ライブラリの外部ユーザからは隠しておくべきだからである。さらに本ツールは、アスペクトが織り込まれるジョインポイント以外の場所に対応する API コメントにも、そのアスペクトの API コメントを織り込むことができる。本ツールは、このために専用のポイントカット言語も用意している。アスペクトの影響は、それが織り込まれるジョインポイントの挙動だけではなく、他の部分の挙動にも間接的に及ぼされるからである。例えばあるメソッドの実行をジョインポイントとして織り込まれるアスペクトは、そのメソッドを呼び出しているメソッドの挙動にも間接的に影響を与えるだろう。そのような場合は、呼び出しているメソッドの API ドキュメントの記述にもアスペクトについての API コメントを挿入すべきである。

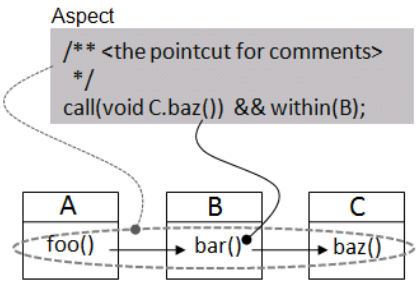

図 4: コールグラフ上の場所の選択

以下本章では、提案するツールを用いて API コメントを記述する方法の概要を述べる。

3.1 API コメント用のポイントカット言語

AOP で書かれたプログラムの API ドキュメントを作成するには、通常のジョインポイント以外にも API コメント用のジョインポイントを選択できるポイントカット言語が必要である。

通常のポイントカットが選択するジョインポイントを中心にして、そのジョインポイントのコールグラフ上の他の場所にもアスペクトの API コメントを織り込んだ API ドキュメントを生成したいことがある（図 4）。例えば、2.1.1 のように、ポイントカットが選択する Stack クラスの peek メソッドだけでなく、peek メソッドを呼び出す pop メソッドにも StackChecking アスペクトの before アドバイスの API コメントを織り込んだ API ドキュメントを生成したい。

また、コールグラフ上のメソッドの API ドキュメントの他にも、アスペクトの API コメントを織り込んだ API ドキュメントを生成したいことがある。フレームワークに公開アスペクトが存在し、クラス側に

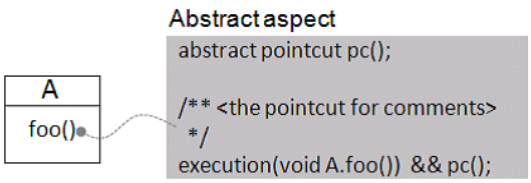

図 5: アスペクトが将来的に織り込まれる場所の選択

アスペクトの実装がまだ織り込まれていないときも、将来的に想定できるその影響を公開クラスの API ドキュメントに反映させたい（図 5）。例えば、2.1.2 のように、UndoFunction アスペクトのサブアスペクトで figures ポイントカットをオーバーライドし Figure クラスのサブクラスを指定すると、そのサブクラスが表す図形に Undo の機能が加わることを Figure クラスの registerFigure メソッドの API ドキュメントに表示したい。

本ツールが提供する API コメント用のポイントカット言語を用いると、これらの場所にもアスペクトの API コメントを織り込んだ API ドキュメントを生成することができる。

3.2 アスペクトの API コメントのモジュール化

API コメント用のポイントカット言語を用いると、ポイントカットとアドバイスの API コメントを別々に記述しておくことができる。ポイントカットにはいつアドバイスが実行されるかだけを API コメントに記述し、アドバイスには何が実行されるかだけを API コメントに記述するべきである。こうすることで、各 API コメントの再利用性が増す。API ドキュメントの生成時に、別々に記述したポイントカットやメソッドの API コメントを関連するアドバイスの API コメントに本ツールが合成する。

3.2.1 @comment

@comment は、引数にメソッド名を取り、そのメソッドの API コメントを展開する。例えば、以下のように、レポジトリのタイムスタンプの管理を行う TimeStampManagement アスペクトがあるとする。ファイルの更新が行われたとき、after アドバイスは TimeStamp クラスの update メソッドを呼び出す。このとき、before アドバイスのコメントに @comment を用いると、TimeStamp.update メソッドのコメントを Repository クラスの update メソッドに展開した API ドキュメント を生成することができる。

```
aspect TimeStampManagement {
  /**
   *   :
   * @comment (TimeStamp.update(File))
   *   :
   */
  after(File file) : args(file) &&
          execution(Repository.update(File)) {
     TimeStamp.update(file);
  }
    :
}
```

また、メソッドの API コメントにも@comment を使用できる。AspectJ によるデザインパターンの実装例 [5] には、抽象アスペクトで実装の雛形を定義しサブアスペクトでメソッドやポイントカットをオーバーライドする例がいくつか存在する。例えば、Observer パターンでは、抽象アスペクト ObserverProtocol の updateObserver メソッドが抽象メソッドとして定義され、サブアスペクトでこれをオーバーライドする。

```
public abstract aspect ObserverProtocol {
   protected abstract pointcut
                  subjectChange(Subject s);

 /**
  * @comment (updateObserver(Subject,Observer))
  */
 after(Sbuject s) : subjectChange(s) {
    Observer o = ...;
    updateObserver(s, o);
 }

 /**
  * Defines how each <i>Observer</i> is to be
  * updated when a change to a <i>Subject</i>
  * occurs. To be concretized by sub-aspects.
  */
 protected abstract void
        updateObserver(Subject s, Observer o);
     :
}
```

このとき、サブアスペクトでオーバーライドした updateObserver メソッドの API コメントに ObserverProtocol の抽象メソッドの API コメントも使用したければ、@comment の引数に super を用いて下のように記述する。ObserverProtocol の subjectChange ポイントカットによって選択された場所の API コメントには、サブアスペクトの updateObserver メソッドの API コメントが織り込まれ、API ドキュメントが生成される。

```
aspect Observer extends ObserverProtocol {
  /**
   *  @comment(
   *    super.updateObserver(Subject,Observer))
   *     :
   */
  protected void
        updateObserver(Subject s, Observer o) {
     :
  }
}
```

3.2.2 @if、@elif

@if や @elif は引数にクラス名を取り、 それがジョインポイントの型と一致するかを判定する。例えば、以下のように、バックアップ用のファイルの管理を行う BackupDeletion アスペクトがあるとする。Save クラスのファイルを保存する execute メソッドが呼ばれたとき、バックアップ用のファイルを削除する。SaveAll クラスは Save クラスのサブクラスである。execute メソッドを呼び出すのが SaveAll クラスのとき、全てのバックアップ用のファイルを削除し、そうでないときは、保存したファイルのバックアップ用のファイルのみ削除する。

このとき、@if と @elif を用いて、ジョインポイントの型によって展開する API コメントを振り分けることが可能である。API ドキュメントの生成時に、SaveAll クラスの execute メソッドの API コメントには BackupDirectory クラスの deleteAll の API コメントが展開され、Save クラスの execute のメソッドの API コメントには BackupDirectory クラスの delete メソッドの API コメントが展開される。

```
aspect BackupDeletion {
  /**
   *    :
   * @if (SaveAll)
   *   @comment(BackupDirectory.deleteAll())
   * @elif (Save)
   *   @comment(BackupDirectory.delete(File))
   * @endif
   *    :
   */
  after(Save s, File f) : target(s) && args(f)
            && call(void Save.execute(File)) {
     if (s instanceof SaveAll)
        BackupDirectory.deleteAll();
     else if (s instanceof Save)
        BackupDirectory.delete(f);
  }
}
```

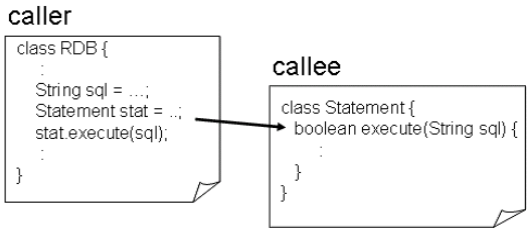

図 6: 呼び出す側（caller）と呼び出される側（callee）

3.3 アドバイスの API コメントの多重化

本ツールを用いると、ひとつのアドバイスに複数個の API コメントを記述することができる。メソッドの呼び出し側（caller）と呼び出される側（callee）で表示すべき API コメントが異なる場合があるためである。例えば、下のようなポイントカットがあるとする。

```
call(* Statement.execute(String)) && within(RDB+)
```

このとき、呼び出される側の Statement クラスの execute メソッドの API ドキュメントには、RDB クラスのサブクラスから呼び出されたときのみアドバイスが実行されることが表示されなければならない。一方、Statement クラスの execute メソッドを呼び出す RDB のサブクラスのメソッドの API ドキュメントには、そのような記述はいらない。

3.3.1 @caller(class)、@callee(class)

本ツールが提供する指定子 @callee(class)、@caller(class) を用いると、複数の異なる API コメントの織り込み先をそれぞれ指定することができる。これらの指定子は名前付きポイントカットの API コメント内に記述する。名前付きポイントカットを指定しないアドバイスに関しては、そのアドバイスのコメントに指定子を記述できる。

@caller は、ポイントカットが選択する場所のコールグラフ上にあるメソッドのうち、呼び出し側に当たるものを選択する。また、 @caller などの指定子は within や execution（表 1）を引数に取り、コールグラフにおける API コメントの有効範囲をさらにクラス側の構造によっても絞り込む。以下の例では、java.sql パッケージ内にあり、Statement クラスの execute メソッドのコールグラフ上にあるメソッドのうち呼び出し側に当たるものを@caller は選択する。選択され

名前	詳細
within(⟨*Class Pattern*⟩)	指定したクラスパターン内にあるクラスやメソッドを選択する。 [例] @callee(within(Figure+)) : Figure のサブクラスのメソッドのうち呼び出される側のメソッドに当たるものを選択する。
execution(⟨*Method Pattern*⟩)	指定したメソッドパターンに適合するメソッドを選択する。 [例] @caller(execution(Line.move(..))) : Line.move メソッドが呼び出す側のメソッドであるとき、これを選択する。

表 1: API コメントの有効範囲を絞り込むために使用する指定子

たメソッドの API ドキュメントは、validate メソッドの API コメントが織り込まれて生成される。

@callee は、ポイントカットが選択するメソッドの呼び出し先のメソッドを指定する。アスペクトの実装で execution ポイントカットが使用されているときには、ポイントカットによって指定されたメソッドを @callee は選択する。以下の例では、Statement クラスの execute メソッドの API コメントを呼ばれる側のメソッドとして @callee は選択する。execute メソッドの API ドキュメントは、*only if the caller ...* の API コメントと validate メソッドのコメントが織り込まれて生成される。

この例では、@callee(within(Statemnt)) の代わりに @callee(execution(Statement.execute(String))) と記述しても選択されるメソッドは Statement クラスの execute で変わらない。しかし、このような指定の仕方は、クラス側の仕様の変更に弱い。例えば、execute メソッドの名前が変更されたとき、@callee の引数の execution にもいちいち変更を加えなければならない。@callee の引数に execution を用いる場合には、将来的にもプログラムの仕様変更がないと考えられるものに対してだけ使用するのが望ましい。

```
/**
 * @caller (within(java.sql.*))
 *   @comment(validate(String))
 *
 * @callee (within(Statement)))
 *   only if the caller is the subclasses
 *   of <code>RDB</code>.
 *   @comment(validate(String))
 */
before(String sql) : args(sql) && within(RDB+)
         && call(* Statement.execute(String)) {
   validate(sql);
}
```

@callerclass や @calleeclass はクラスの API コメントにアドバイスの API コメントを織り込んだ API ドキュメントを生成したいときに使用する。例えば、以下のように、Servlet のサブクラスのメソッド全てを pc ポイントカットが選択するとする。このようなときには、クラスのドキュメントとして アドバイスの API コメントを表示した方が都合がよい。

```
/**
 * @calleeclass (execution(Servlet+))
 */
pointcut pc() : exeuction(* Servlet+.*(..));
```

3.3.2 @for

@for は、アドバイスの API コメントに記述し、引数に caller などの指定子名を取る。@for を用いると、ポイントカットとアドバイスのコメントのモジュール化を保ちながら複数の異なる記述を行うことができる。

ひとつの名前付きポイントカットを複数のアドバイスが指定することがある。このようなとき、@for は特に有用である。例えば、以下のように、トランザクションの管理を行う TransactionManagement アスペクトがあるとする。transactionalMethods ポイントカットの API コメントで呼び出し側と呼ばれる側の織り込み先を指定し、before アドバイスや after アドバイスの API コメントでそれぞれの織り込み先に対する記述を行える。

```
public aspect TransactionManagement {
 /**
  * @calleeclass (within(Facade+))
  *
  * @caller (within(persistence.*))
  */
  pointcut transactionalMethods() :
              execution(* Facade+.*(..));

  /**
   * @for (calleeclass)
   *    @comment (beginTransaction())
   * @for (caller)
```

```
   *    an internally invoked method is
   *    under transaction control
   */
  before() : transactionalMethods() {
      beginTransaction();
  }

  /**
   * @for (calleeclass)
   *   when returning normally
   *   @comemnt (endTransaction())
   */
  after() returning : transactionalMethods() {
     endTransaction();
  }
      :
}
```

4 例題

AspectJ で実装されたアプリケーションサーバープログラム Health Watcher [3] を利用し、本ツールを用いた場合にどのような API コメントの記述が必要か具体例を挙げて検討する。Health Watcher のプログラムは、クラスの数が 692（LOC 9591）であり、アスペクトの数は 25（LOC 1989）である。アスペクトの使用用途は主にデザインパターン、永続化、分散化、トランザクション、例外ハンドリングなどであり、全て内部アスペクトとして実装されている。

4.1 Observer パターン

3.2.1 で述べた ObserverProtocol アスペクトのサブアスペクトとして UpdateStateObserver アスペクトを Health Watcher では定義していた。本ツールを利用するために、図 7 のように API コメントを記述する必要がある。

subjectChange ポイントカットの定義により、CommandServlet のサブクラスから Subject のサブクラスの setter メソッドが呼ばれたときだけアドバイスは実行される。そのため、呼び出される側のメソッドの API ドキュメントには、限定的にアドバイスが実行されることが明示されなければならない。そこで、subjectChange ポイントカットの API コメントに @callee を用いる。@callee の引数を within(Subject+) とし、Subject のサブクラスの setter メソッドの API ドキュメントに以下のような API コメントを織り込めるようにする。

only if the caller is the subclasses of ⟨*code*⟩

```
public aspect UpdateStateObserver
                extends ObserverProtocol {
            :
  /**
   * @callee (within(Subject+))
   *   only if the caller is the subclasses
   *   of <code>ComamandServlet</code>.
   *
   * @caller (within(CommandServlet+))
   */
  protected pointcut
             subjectChange(Subject subject):
        this(CommandServlet+) &&
        call(* Subject+.set*(..)) &&
        target(subject);

  /**
   * @comment (super.
   *         updateObserver(Subject,Observer))
   *   when the actual type of observer is
   *   <code>IFacace</code>, subject will be
   *   executed as follows:
   *
   * @if (Employee)
   *    @comment (IFacade.
   *                updateEmployee(Employee))
   * @elif (Complaint)
   *    @comment (IFacade.
   *              updateComplaint(Complaint))
   * @elif (HealthUnit)
   *    @comment (IFacade.
   *            updateHealthUnit(HealthUnit))
   * @elif (Symptom)
   *    @comemnt (IFacade.
   *                  updateSymptom(Symptom))
   * @elif (MedicalSpeciality)
   *    @comemnt (IFacade.
   *  updateMedicalSpeciality(MedicalSpeciality))
   * @endif
   */
  protected void updateObserver(Subject subject,
                                Observer observer) {
    if (observer instanceof IFacade) {
      IFacade facade= (IFacade) observer;
      if(subject instanceof Employee)
         facade.updateEmployee((Employee)subject);
      else if(subject instanceof Complaint)
         facade.updateComplaint((Complaint)subject);
      else if(subject instanceof HealthUnit)
         facade.updateHealthUnit((HealthUnit)subject);
      else if(subject instanceof Symptom)
         facade.updateSymptom((Symptom) subject);
      else if(subject instanceof MedicalSpeciality)
         facade.updateMedicalSpeciality(
                      (MedicalSpeciality) subject);
    }
  }
}
```

図 7: UpdateObserver の実装

CommandServlet⟨/code⟩

また、呼び出す側のメソッドの API ドキュメントにもオブザーバーの更新に関する API コメントを明記したい。そこで、@caller の引数には within(CommandServlet+) と書き、CommandServlet のサブクラスのうち setter メソッドを呼ぶメソッドにアドバイスの API コメントを織り込むようにする。

updateObserver メソッドの API ドキュメントには、ObserverProtocol アスペクトの updateObserver メソッドの API コメントを初めに記述したい。updteObserver メソッドの汎用的な意味が記述されているためである。そこで、@comment を用いて ObserverProtocol の updateObserver メソッドの API コメントを指定する。また、updateObserver メソッドの実装では、observer オブジェクトの型が IFacade であった場合にオブザーバーの更新を行う。そのため、API コメントには以下のように書き、条件的にサブジェクトが実行されることを明記する。

when the actual type of observer is ⟨code⟩IFacade ⟨/code⟩, subject will be executed as follows:

さらに、実装に合わせて、@if を用いて展開する各 API コメントの条件分けを行う。subjectChange ポイントカットが選択するジョインポイントの型は Subject であるので、@if には Subject のサブクラスを指定する。

4.2 例外ハンドリング

例外ハンドリングを行うアスペクトが Health Watcher プログラムにはいくつか存在した。そのうちのひとつである HWPersistenceExceptionHandler アスペクトの after throwing アドバイスは、AddressRepositoryRDB クラスの search メソッドの実行期間中に PersisitenceMechanismException が投げられると、次に RepositoryException を投げる処理を行っている。

このとき、本ツールを利用するために、図 8 のように API コメントを書く必要がある。AddressRepositoryRDB クラスの search メソッドとそれを呼ぶ側のメソッドの API ドキュメントにそれぞれ after throwing アドバイスの API コメントを織り込みたい。search メソッドの呼び出し側のメソッドは、ComplaintRepositoryRDB にだけあった。そこで、@callee と@caller の

```
public aspect HWPersistenceExceptionHandler {
       :
 /**
  * @callee (within(healthwatcher.data.rdb.
  *                     AddressRepositoryRDB))
  *
  * @caller (within(healthwatcher.data.rdb.
  *                   ComplaintRepositoryRDB))
  */
  pointcut searchAddress() : execution(
           public Address AddressRepositoryRDB
                                .search(int));
 /**
  * @exception RepositoryException
  */
  after() throwing
              (PersistenceMechanismException e)
                     throws RepositoryException
              : searchAddress() {
     throw new RepositoryException(
                ExceptionMessages.EXC_FALHA_BD);
  }
}
```

図 8: HWPersistenceExceptionHandler アスペクトの実装

引数には within を用いて呼び出される側と呼び出し側のメソッドがあるクラスをそれぞれ指定する。after throwing アドバイスの API コメントには投げられ得る例外として RepositoryException があることを記述しておく。

5 関連研究

Javadoc は、ソースコード中に API コメントとして埋め込まれたドキュメントを収集し、HTML フォーマットで Java API ドキュメントを自動的に生成するツールである。AOP で書かれたプログラムの API ドキュメントを作成するには、公開クラスにアスペクトの影響を考慮した API コメントを書かなければならず、API コメントのモジュール化ができない。

Ajdoc は、AspectJ で書かれたプログラムの API ドキュメントを作成するツールである。しかし、公開クラスに影響を与えるアスペクトがライブラリやフレームワークの内部にあるとき、 Ajdoc はその影響を API ドキュメントには必ずしも反映できるとは限らない。また、 フレームワーク中に公開アスペクトがあるときにも、Ajdoc はその影響を公開クラスの API ドキュメントに反映できないことがある。

API ドキュメントの生成時にクラスへのアスペク

トの影響を反映させるわけではなく、AOP で開発を行うときにアスペクトの影響を理解するための提案がいくつかある。Aspect-Aware Interface [7] は、obliviouesness [2] を解決するアスペクト指向言語用の新しいインターフェースである。このインターフェースを通して、アスペクトが織り込まれる度にクラスへの影響を表示するべきであると提案している。これにより、モジュールプログラミングが可能になる。しかし、Aspect-Aware Interface は基本的な概念を提示しただけで、各ポイントカットの表示方法などに明確な指針はない。execution ポイントカットの場合には呼ばれる側のメソッドの仕様を変更するが、メソッド呼び出し時を選択する call ポイントカットやフィールドの参照・代入時を選択する get、set ポイントカットなどは未解決の問題としている。本ツールは、このようなポイントカットであっても、API ドキュメントの生成時に呼び出し側のメソッドや呼ばれる側のメソッドへアドバイスの API コメントを織り込むことで、その影響を反映させることができる。

open module [1, 10] や XPI（crosscut programming interface）[4] は、obliviousness を解決する別の手法を提案している。これらの手法では、アスペクトが拡張可能なジョインポイントをモジュールのインターフェースに公開する。このインターフェースで公開されているジョインポイントしかアスペクトは拡張しないため、クラスの obliviouesness が保たれる。しかし、この手法では、アスペクトによって拡張されるジョインポイントを予め開発者が予測しなければならない。拡張される可能性のあるポイントカットを予測することは困難である。予測できない場合には、拡張するジョインポイントに合わせて毎回モジュールインターフェースを更新する必要がある。open moudle や XPI を用いた開発でも、本ツールは API ドキュメントを生成することが可能である。

6 まとめと今後の課題

6.1 まとめ

本稿において我々は、アスペクトの API コメントをクラスの API コメントに織り込んでから、API ドキュメントを生成する AspectJ 用の改良 Javadoc を提案した。本ツールを用いると、クラスとアスペクトの API コメントを分離しておき、API ドキュメントの生成時に公開クラスにアスペクトの API コメントを本ツールによって自動的に織り込むことができる。そのために、コメント用のポイントカットを用意した。この言語機構を用いることにより、アドバイスやポイントカットの API コメントをうまくモジュール化することができる。

6.2 今後の課題

@caller や @callee などの指定子を用いて、呼び出す側や呼び出される側のメソッドの API コメントにアドバイスのそれぞれ異なる API コメントを織り込んだ API ドキュメントを本ツールは生成することができる。しかし、これらの指定方法で十分かを今後評価する必要がある。また、@caller などの指定子は within や execution などを引数に取り、API コメントを織り込む範囲をクラス側の構造をもとに絞り込めるが、これらの指定方法に関しても十分かの評価を今後行わなければならない。また、本ツールの実装は未完成であるが、Eclipse ワークベンチから本ツールを起動することを想定し、現在、実装を行っている。

参考文献

[1] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *ECOOP 2005 - Object-Oriented Programming, LCNS 3586*, pages 144–168. Springer-Verlag, 2005.

[2] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Technical report, 2000.

[3] Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Cláudio Sant'Anna, Sergio Soares, Paulo Borba, Uirá Kulesza, and Awais Rashid. On the impact of aspectual decompositions on design stability: An empirical study. In *ECOOP 2007 - Object-Oriented Programming, LNCS 4609*, pages 176–200. Springer-Verlag, 2007.

[4] William G. Griswold, Macneil Shonle, Kevin Sullivan, Yuanyuan Song, Nishit Tewari, Y uanfang Cai, and Hridesh Rajan. Modular Software Design With Crosscutting Interfaces. In *IEEE Software, vol.23*, pages 51–60, 2006.

[5] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM.

[6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. G riswold. An overview of aspectj. In *ECOOP 2001 - Object-Oriented Programming: 15th European Conference, LNCS 2072*, pages 327–353. Springer, 2001.

[7] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[8] Sun Microsystems. Java 2 platform stard edition 5.0 api specification. http://java.sun.com/j2se/1.5.0/docs/api/.

[9] Sun Microsystems. Javadoc 5.0 tool. http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/.

[10] Neil Ongkingco, Pavel Avgustinov, Julian Tibble, Laurie Hendren, Oege de Moor, and Ganes h Sittampalam. Adding open modules to AspectJ. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software developmen t*, pages 39–50, New York, NY, USA, 2006. ACM.

[11] AspectJ Organization. the aspectj documentation tool. http://www.eclipse.org/aspectj/doc/next/devguide/ajdoc-ref.html.

[12] André L. Santos, Antónia Lopes, and Kai Koskimies. Framework specialization aspects. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 14–24, New York, NY, USA, 2007. ACM.