平成18年度 学士論文

実行中のプログラムの振舞いを安全に拡張できるアスペクト指向システム

東京工業大学 理学部 情報科学科
学籍番号 02-0878-1
栗田 洋輔

指導教員
千葉 滋 助教授

平成18年2月6日

概要

近年、プログラムの振舞いをモジュール単位で動的に変更する技術として、Dynamic AOP が注目を浴びている。これは、オブジェクト指向では不可能とされていたモジュールの分割を可能とする、Aspect-Oriented Programming (AOP) において、アスペクトを動的に追加 (ウィーブ)・削除 (アンウィーブ) できる技術である。通常のシステムでは、予め設定された特定の振舞いしかプログラムの実行中に変更することができないので、任意の振舞いを変更したいならば、実行中のプログラムを停止、コンパイル、ビルド、再スタートする必要があるが、Dynamic AOP ではプログラムの実行を止めずに、任意の振舞いを変更できる。

しかし、Dynamic AOP は自由度が高い分、問題点も多い。他のアドバイスが実行された後でのみ、実行されるべきアドバイスが存在する場合などは、それらのアドバイス同士の順序関係を考慮しない単純なウィーブでは問題が発生してしまう。例えば、ショッピングサイトにログインしてからログアウトするまでの顧客をアスペクトを用いてプロファイリングするというアプリケーションをを考える。顧客のログイン時にはログイン用アドバイスが呼び出されて各種カウンタを加算し、ログアウト時にはログアウト用アドバイスが呼び出されて各種カウンタを減算するとする。この場合、ウィーブ前に既にログインしていた顧客がウィーブ後にログアウトすると、ログイン用アドバイスが呼び出されずにログアウト用のアドバイスだけが呼び出されてしまい、カウンタは加算されずに減算だけが行われてしまう。このアプリケーションには顧客の ID を単位とした順序関係が存在する例だが、他の属性を単位とした順序関係や、スレッドやオブジェクトを単位とする順序関係も考えられる。

この問題に関しては、既存の Dynamic AOP System でも、アスペクト内にアドバイスの順序関係を制御するフラグや、それらのマップなどを定義し、アドバイス内でそれらを参照・変更することでアドバイスの順序関係を元にウィービングを制御できる。しかし、そのような方法では、コードのサイズが大きくなってしまう問題がある。また、アドバイス内にアドバイス間の順序関係を考慮するコードを手続き的に記述されてしまうと、モジュール性・可読性が低下してしまい、プログラムに間違いが発生しやすくなるという問題もある。

本研究では、Java Platform Debugger Architecture (JPDA) のブレークポイントを用いる Dynamic Weaving という、既に存在する実装法を用いて Dynamic AOP System を開発し、その上でこの問題を解決した。順序関係を AND 、OR 、NOT という論理演算子を用いて宣言的に定義する API をシステムが提供し、この API によりアドバイス間の順序関係を記述することで、アドバイス間の順序関係をアドバイスの中身から分離できる。システムは、API によって定義された順序関係を元に、アドバイスを実行すべきかどうかを自動的に判断する。

このような API を実装するときには、ブレークポイントの設定が遅れると、ジョインポイントからアドバイスが呼び出されるべきときに、呼び出されないという問題を考える必要がある。この問題に関しては、全スレッドを一度止めることで、ポイントカットで指定された全てのジョインポイントからアドバイスを呼び出し可能にする Dynamic AOP System は既に存在するが、その実装法ではウィーブのためにアプリケーション全体の実行が止まってしまうため、時間コストが大きいという点で問題があった。また、スレッドを止めずに、ポイントカットで指定された全てのジョインポイントからアドバイスを呼び出し可能にする Dynamic AOP System も既に存在するが、ジョインポイントからアドバイスを呼び出し可能になるタイミングをシステムの側で検知できない。そのため、十分な時間を確保した後に、手動でトランザクションをコミットしなければならず、ジョインポイントからアドバイスが呼び出されないというリスクを回避するためには長時間待たなければならないという問題があった。

この実装上の問題に関しては、JPDA の、ブレークポイントは設置を要求した順番に実際に設置されるという仕様と、プログラム中の一つの位置に複数のブレークポイントを設置できるというの仕様を用いて対処した。ポイントカットで指定された全てのジョインポイントに１つ目のブレークポイントを設置した後、今度は同じ位置に２つ目のブレークポイントを設置する。このようにすることで、２つ目のブレークポイントからイベントが発生したときには、ポイントカットで指定された全てのジョインポイントに最低１つのブレークポイントが設置されていることが保障される。よって、ポイントカットで指定された全てのジョインポイントからアドバイスが呼び出し可能になったことを、システムが自動的に検知できるようになった。

アスペクトの外部に存在する順序関係に対処する方法は、本研究の今後の課題である。

謝辞

本研究を進めるに当たり、研究の方針や進め方について数々の助言をしてくださった指導教員の千葉滋助教授に感謝致します。そして、本稿のスタイルファイルを作成して頂いた光来健一氏に感謝いたします。また、西澤無我氏には、論文の内容について数々の助言を頂きました。柳澤佳里氏には、多くの疑問を聞いて頂き、指導して頂きました。石川零氏には、論文の構成について面倒を見て頂きました。最後に、卒業研究を行う上で励まして頂いた同研究室の皆様に心から感謝いたします。

目 次

図 目 次

表 目 次

第1章　はじめに

今日、情報システムが社会的に重要なインフラとなる中で、ソフトウェアの持続的実行や安全性がますます重要になっている。特に、システムが停止すると、企業に対する巨額の損失やユーザからの信用を失墜するといった事態を招くこととなるミッションクリティカルシステムでは、それらの性質が非常に重要になる。

しかし、従来のシステムでは、予め設定された特定の振舞いしかプログラムの実行中に拡張することができなかった。このようなシステムでは、任意の振舞いを拡張したいならば、実行中のプログラムを停止、コンパイル、ビルド、再スタートする必要がある。

プログラムの実行を止めずに振舞いを変更する技術として、近年、注目を浴びているものに Dynamic AOP (DAOP) がある。これは、オブジェクト指向を超えたモジュールの分割を可能とする、Aspect-Oriented Programming (AOP) において、アスペクトを動的に追加 (ウィーブ)・削除 (アンウィーブ) できる技術である。

DAOP はウィーブのタイミングで大きく 2 通りに分けることができる。アスペクトをロード時にウィーブできるロードタイム時の AOP と、プログラムの実行中の任意の時点でアスペクトをウィーブできるランタイム時の AOP である。ランタイム時の AOP の方がより柔軟であるが、これには問題点も多い。アスペクトをウィーブするタイミングによって、プログラムの様々な状態に不整合が発生してしまうことがあるからだ。

DAOP に対する需要が大きいミッションクリティカルシステムでは、プログラムに不整合が発生してしまうことは致命的であるが、従来の DAOP System では、このような問題についてあまり考慮されていなかった。

本論文では、アスペクトがウィーブされるタイミングによって、プログラムにどのような不整合が発生してしまうかを考え、それを防止するために DAOP System に必要な機能を考察する。そして、本システムでどのようにして、そのような機能を実装したのかを説明する。

本稿の残りは、次のような構成からなっている。第 2 章では、 既存の DAOP System と、それらを利用した場合に発生する問題について述べる。第 3 章では、本研究における成果を述べる。第 4 章では、本システムの実装方法を述べる。第 5 章では、本研究の必要性を実験結果をもっ

て実証する。第 6 章では、本研究のまとめを行う。

第2章　既存の DAOP システムと問題点

2.1　AOP の概念

Aspect Oriented Programming (AOP) とは、オブジェクト指向の欠点を補うモジュール化技術で、以下の概念をもつ。

- 横断的関心事
  複数のモジュールに散らばる、ある面 (Aspect) から見ると似ているコード
- アスペクト
  プログラムの横断的関心事をまとめたモジュール
- ウィーブ
  対象プログラムとアスペクトを結びつけること
- ジョインポイント
  ウィーブ対象プログラムにおける、アスペクトと結びつけることができるポイント
- ポイントカット
  ジョインポイントの集合から、実際にアスペクトと結びつけるポイントを選び出すための記述
- アドバイス
  プログラムの実行がポイントカットで指定されたジョインポイントに至ったときに実行される、アスペクトに含まれるコード

アスペクト指向という名前が初めて使われた AspectJ という言語では、それらは次のように記述される。

```
class Rectangle {
  void calculateArea() { /* 計算処理をする */ }
}
```

```
class Circle {
  void calculateArea() { /* 計算処理をする */}
}
```

クラス Rectangle のメソッド calculateArea と、クラス Circle のメソッド calculateArea は、計算を行うという点では共通している。そこで、これらのメソッドが呼び出される時点で、before calculate というメッセージをログに出力したいとする。

オブジェクト指向では、通常、以下のような方法が用いられる。まず、ログ出力用のクラス OOP_Logger を作る。

```
public class OOP_Logger {
  public static void println(Object obj) {
    System.err.println(obj);
  }

  ...
}
```

そして、calculateArea や calculateArea のメソッド内部の初めの行に

```
OOP_Logger.println("before calculate");
```

という記述を追加する。

しかし、この方法は、同様なコードが複数のモジュールに散らばってしまうというし、ログ出力のコードがプログラムの核となるロジックと混在してしまうので、保守性・可読性という点で問題がある。

それに対し、AspectJ では、次のようなアスペクトを定義することで、calculateArea や calculateArea のメソッド内部からログ出力のコードを抜き出すことができる 。

```
public aspect AOP_Logger {
  pointcut loggedMethods():
    execution(void Rectangle.calculateArea())
    || execution(void Circle.calculateArea());

  before():loggedMethods() {
    System.err.println("before calculate");
  }
}
```

AOP_Logger は、アスペクトであり、loggedMethods というポイントカットと、before():loggedMethods() というアドバイスを持っている。ポイントカット loggedMethods　は、Rectangle クラスの calculateArea メソッドと Circle クラスの calculateArea メソッドが実行される「とき」を指定している。また、アドバイス before():loggedMethods() は、ポイントカット loggedMethods で指定されたポイントカットに至る前 (before) に、ログ出力を行うということを定義している。

アスペクト AOP_Logger をウィーブすることで、Rectangle クラスの calculateArea メソッドが呼び出される前と、Circle クラスの calculateArea メソッドが呼び出される前に、ログ出力を行うコードが織り込まれる。

以上のようにして、ログ出力を行うという横断的関心事を一つのモジュールにまとめることができた。オブジェクト指向の例では、ログ出力の実装は OOP_Logger という一つのモジュールにまとめることができたが、ログ出力を行う「時点」は複数のモジュールに散らばっていた。しかし、アスペクト指向を用いると、それまでも一つのモジュールにまとめることができた。これはモジュール間の順序関係が逆転したから可能となったことだ。

従来のプログラミング手法では、モジュール同士を結びつけるときは、主にメソッド呼び出しを使ってきた。AOP では、これに加えて、ポイントカット・アドバイスによっても、モジュールを結び付けられるようにした。メソッド呼び出しの場合、モジュール間の呼び出しは (メソッドを) 呼ぶ側に記述される。一方、ポイントカット・アドバイスの場合、それは (アドバイス) が呼ばれる側のモジュールに記述される。

アスペクト指向では、このように順序関係が逆転させることで、呼ぶ側のモジュールに修正を加えることなく振舞いを変更できる。そのため、呼ぶ側のモジュールを再利用することも可能となる。

以上の議論は、アスペクト指向入門 [9] を参考にした。

2.2　Java ベースの AOP システムの分類

Java ベースの AOP システムは、ウィーブのタイミングによって、Compile-time weaving、Load-time weaving、Run-time weaving の 3 通りに分けられる。

Compile-time weaving は、コンパイル時にアスペクトのウィーブが行われ、プログラムの実行中にはアスペクトをウィーブ/アンウィーブすることはできない。Load-time weaving は、アスペクトがウィーブされる対象のクラスが Virtual Machine にロードされるタイミングで、アスペクト

をウィーブできる。しかし、クラスのロードが完了した後には、アスペクトをウィーブ/アンウィーブできない。Run-time weaving は、アスペクトがウィーブされる対象のクラスが Java Virtual Machine (JVM) にロードされた後でも、アスペクトをウィーブ/アンウィーブできる。

このうち、Compile-time Weaving のみを用いた AOP System を、Compile-time AOP System、それ以外のシステムを Dynamic AOP System と呼ぶことにする。特に、Run-time weaving が可能な AOP System を、ここでは Run-time AOP System と呼ぶことにする。

2.3　Run-time AOP System の分類

Run-time AOP を実現するために現在用いられているウィービングの方法は、大きく分けて以下の 5 通りである。

- Hook weaving
- JPDA event notification based weaving
- JPDA HotSwap based weaving
- Java Dynamic Proxy based weaving
- Virtual Machine level weaving
- OS level weaving

処理速度の向上のため、複数の方法を用いている AOP System が存在することに注意。

2.3.1　Hook weaving

Hook weaving とは、コンパイル時に予め全てのジョインポイントにフックを挿入する方法である。フックには、その場所のジョインポイントがポイントカットによって指定されているかを確認するコードがあり、もし指定されているのならば、アドバイスのコードへジャンプするようになっている。

この方法を用いた AOP System には、EAOP [4]、JAC [3]、PROSE [1]、等がある。EAOP では、プリプロセッサによってコンパイル時にフックが挿入され、フック挿入済みのクラスファイルが生成される。JAC、JBoss AOP では、拡張されたクラスローダによって、バイトコードのロード時にフックが挿入される。PROSE 2 は、Jikes RVM 内 の Just-In-Time(JIT)

コンパイラを拡張していて、クラスが JIT コンパイル されるタイミングで、フックが挿入される。

Hook weaving は、パフォーマンスの面で大きな欠点がある。全てのジョインポイントにフックが挿入されるため、プログラムが肥大化してしまう。また、ポイントカットで指定されていないジョインポイントでも、いちいちポイントカットされたかどうかを確認する処理を実行しなければならないので、アスペクトが挿入されていない場合でも、多大なオーバーヘッドが発生してしまう。

オーバーヘッドを減らすには、予め指定した特定のジョインポイントの集合だけにアスペクトをウィーブできるようにするという方法も考えられるが、そのようにしてしまうと、予想できなかったジョインポイントにはアスペクトをウィーブできないという制限が発生してしまう。

2.3.2　JPDA event notification based weaving

JPDA based weaving は、Java Platform Debugger Architecture (JPDA) の、イベント通知機能を用いてウィーブを行う方法である。

JPDA を用いると、実行中の Java アプリケーションに対して、メソッドの入り口や出口、フィールドの参照や変更、例外の発生、特定の行番号への到達、スレッドの開始や終了などのポイントでイベントを発生させるように要求できる。イベントの発生ポイントでスレッドや VM 全体を停止させ、アドバイスを実行した後にスレッドを再開することで、ウィービングを行うことができる。

この方法を用いた AOP System には、PROSE、Wool [8] などがある。

Hook weaving のようにウィーブの対象となるコードを変更しないため、プログラムのサイズが肥大化してしまうという問題はない。また、ジョインポイントの中でもポイントカットで指定された部分のみが処理を行うので、アスペクトが実行されない限り、オーバーヘッドは発生しない。

しかし、この方法は、イベントが発生してからイベントがハンドルされるまでのタイムラグが大きく、アドバイスが実行される回数に比例してオーバーヘッドが大きくなってしまう。

2.3.3　JPDA HotSwap based weaving

JPDA Hotswap based weaving は、Java Platform Debugger Architecture (JPDA) の、HotSwap という機構を用いてウィーブを行う方法である。HotSwap を用いると、VM 上にロードされたバイトコードを、実行時に新しいバイトコードに置き換えることができる。

この方法を用いた AOP System には、PROSE、Wool などがある。

Sun VM (Version 5.0) では、メソッドの内部のバイトコードしか置き換えることができないという制限がある。

2.3.4　Java dynamic proxy based weaving

Java dynamic proxy based weaving は、Java Reflection API 内に Version 1.3 から導入された Proxy 機能を用いてウィービングを行う方法である。

この機能を用いると、通常のメソッドをラップした proxy メソッド を生成することができる。proxy メソッド は、ラップされた通常のメソッドを呼び出す前に、任意のコードを実行することができる。

この方法を用いた AOP System には、Nanning [7] がある。

Java の Proxy 機能は、メソッドにしか適用することができないため、この方法を用いた実装では Method call pointcut しか利用できない。

2.3.5　Virtual Machine level weaving

Virtual machine level weaving は、Virtual machine (VM) をアスペクト指向言語用に拡張することで、アスペクトのウィーブを行えるようにする方法である。

この方法を用いた AOP System には、Steamloom [2] 、PROSE などがある。Steamloom と PROSE は、Jikes Research Virtual Machine (Jikes RVM) を拡張している。

2.3.6　OS level weaving

OS level weaving は、OS の カーネルを拡張して、実行中のバイトコードを直接書き換えることを許す方法である。

インタープリター方式を採用している言語ではインタープリターをアスペクト指向に対応させればよいが、コンパイル方式の言語の場合はそのような方法を用いることができないため、OS のカーネルを変更する。

2.4　Run-time Weaving の手法の分類

Java ベースの Run-time Weaving 手法を様々な観点から比較した。比較対象の手法は以下の４つである。

- Hook Weaving
  コンパイル時にフックを用いる手法
- Breakpoint Weaving
  JPDA のブレークポイントを用いる手法
- HotSwap Weaving
  JPDA の HotSwap を用いる手法
- Extended VM Weaving
  Virtual Machine を拡張する手法

比較の項目は以下の通りである。

- DefaultSpeed (DS)
  アドバイスがウィーブされない場合の実行速度
- Switch Time (ST)
  アドバイスの呼び出しと、そこからの復帰にかかる時間
- Weaving Time (WT)
  ウィーブが完了するまでにかかる時間
- Method Before/After Advice (MBAA)
  メソッドについての Before/After アドバイス
- Method Around Pointcut (MAA)
  メソッドについての Around アドバイス
- Field Pointcut (FA)
  フィールドを参照・変更するときに呼び出されるアドバイス
- Schema Modification (SM)
  クラスのスキーマの変更

図 2.1 がこれらの項目についての比較である。

- アドバイスがウィーブされない場合の実行速度 (DS)
  Hook を用いた手法が不利である。Hook Weaving では、プログラムがジョインポイントに到達するごとに、そこがポイントカットで指定されているかどうかをいちいち確認しなければならない。ここで多大なオーバーヘッドが生じる。BreakPoint Weaving や HotSwap Weaving では、プログラムをデバッグモードで動かすため、通常の実行に比べやや遅くなる。Extended VM Weaving は、どのような

表 2.1: Run-time Weaving 手法の比較

手法	DS	ST	WT	MBAS	MAA	FA	SM
Hook Weaving	×	△	○	○	○	△	×
Breakpoint Weaving	○	×	○	○	×	○	×
HotSwap Weaving	○	○	△	○	○	△	×
Extended VM Weaving	○	○	△	○	○	○	○

拡張方法を採用するかに依存するが、Steamloom の場合では、ベースとなる Jikes RVM に対して、4% だけの遅延で済んでいる。この遅延は、クラスのロード時にバイトコード操作のためのデータ構造をつくるための手続きと、メソッドのインライン化を記録する手続きのためである。

- アドバイスの呼び出しと、そこからの復帰にかかる時間 (ST)
  Breakpoint Weaving が不利である。これはアドバイスの呼び出し時に、異なるプロセス間やマシン間で通信を行うためである。Hook Weaving だと、プログラムの実行がジョインポイントに到達して、既にウィーブされたアドバイスを呼び出すときにも、そのジョインポイントが本当にポイントカットされているのかをいちいち確認するため、アドバイス呼び出しにはやや時間がかかってしまう。HotSwap Weaving や Extended VM Weaving では、アドバイスの中身を対象コード中に直接埋め込む方法と、アドバイスを呼び出すスタブだけを埋め込む方法の 2 つに分けられるが、前者の方法を用いた場合は、アドバイスを呼び出すための時間はかからない。後者の方法を用いた場合は、アドバイス呼び出しがインライン化されなければ、メソッド呼び出しのための時間がかかる。

- ウィーブが完了するまでにかかる時間 (WT)
  Hook Weaving が最も早い。フックから参照されるフラグを変更するだけだからである。Breakpoint Weaving では、ブレークポイントが設置されるまでの時間がウィーブが完了するまでの時間となる。

- メソッドについての Before/After アドバイス (MBAA)
  全ての手法で行うことができる。また、図 2.1 の比較対象にはないが、Dyncmic Proxy Weaving でもこれらのポイントカットを用いることができる。Hook Weaving や、HotSwap Weaving の場合、例外が投げられてメソッドから出る場合も After アドバイスに含めると、話が少し難しくなる。だが、PROSE の場合は、メソッド全体

を try-finally 構文の try 節で囲み、finally 節の After アドバイスを呼び出すスタブを置くことで、この問題を解決している。

- メソッドについての Around アドバイス (MAA)
  Breakpoint Weaving では実装することができない。なぜならば、ブレークポイントはスレッドを中断・再開するだけで、特定のコードを実行しないようにすることは不可能だからである。

- フィールドについてのアドバイス (FA)
  フィールドについてのアドバイスをウィーブする場合に問題なのは、ウィービングのプロセス自体よりも、プログラム中のどこからフィールドアクセスが行われるかとういう情報を知ることである。もし、フィールドの修飾子が public だった場合は、全てのクラスの全てのメソッドから、そのフィールドにアクセス部分を調べ出さなければならない。プログラムのサイズが大きい場合は、これは大きな問題である。だが、これはクラスローダーを拡張することで対応できる。クラスのロード時に、そのクラスのどの部分がどのフィールドにアクセスするかという情報を抜き出し、それを記録しておけばよい。Hook Weaving や HotSwap Weaving は、この方法を用いることで、フィールドのついてのアドバイスをウィーブできる。なお、Breakpoint Weaving の場合は、フィールドにブレークポイントを設置すれば容易に実装できる。Extended VM Weaving の場合は、VM の中のフィールドのアクセスする部分を拡張すればよい。

- クラスのスキーマの変更 (SM)
  現時点では、Sun VM に、実行時に動的にクラスのスキーマを変更する機能は備わっていない。よって、VM を拡張しなければ、そのようなことを行うことはできない。

2.5　問題点

既存の Run-time AOP には問題点がある。それを次のショッピングサイトのシステムを単純化した例を用いて説明しよう。このシステムは、主に以下の 2 つのクラスから成る。

- Shopping クラス (図 2.1)
  ショッピングサイトの入り口であり、login 、loginAsVIP 、logout という 3 つのメソッドから成る。ショッピングを開始するときには login メソッドまたは loginAsVIP メソッドが必ず呼び出される。ショッピングを終了するときには logout メソッドが必ず呼び出される。引

```
public class Shopping {

  public void login(Customer customer) {
    System.out.println("-> Login:      " + customer);
/* ログインの処理を行う　*/
  }

  public void loginAsVIP(Customer customer) {
    System.out.println("-> Login(VIP): " + customer);
    /* VIP (Very Important Person) として、
       通常とは別のログイン処理を行う */
  }

  public void logout(Customer customer) {
    System.out.println("<- Logout:     " + customer);
/* ログアウトの処理を行う */
  }
}
```

図 2.1: Shopping クラス

数にはショッピングサイトに入退場する Customer オブジェクトが代入される。

- Customer クラス ((図 2.2))
  ショッピングサイトの顧客を表すクラス。customerID 、age 、sex という 3 つの属性を持っている。

以下は、ショッピングサイトの実行例である。customerID、age、sex という属性を持った Customer が不定期にログイン・ログアウトを行う。通常のログインではなく、VIP(Very Important Person) としてログインする Customer もいる。行の最初の矢印はログイン・ログアウトを視覚的に表している。

```
-> Login:      customerID:0  age:24  sex:Male
-> Login(VIP): customerID:1  age:38  sex:Female
<- Logout:     customerID:0  age:24  sex:Male
-> Login:      customerID:2  age:43  sex:Male
```

```
-> Login:       customerID:3  age:51  sex:Female
<- Logout:      customerID:1  age:38  sex:Female
-> Login(VIP): customerID:4  age:45  sex:Female
<- Logout:      customerID:4  age:45  sex:Female
-> Login:       customerID:5  age:30  sex:Male
<- Logout:      customerID:3  age:51  sex:Female
<- Logout:      customerID:2  age:43  sex:Male
```

このショッピングサイトのシステムにおいて、プロファイリングを行うストーリーを考える。なお、以下の条件を仮定する。

- ある時点以降にセッション内に存在する顧客の総数、平均年齢、女性の割合を計算したい。
- 現在、稼動中のシステムでは、そのような値を計算することができない。
- システムを止めることはできない。

この要求を Run-time AOP を用いて実現するには、図 2.3 のようなアスペクトを考える。(本研究で開発したシステムで表記してある)
このアスペクトの、ポイントカットとアドバイスは以下のようになっている。

- loginPCut — Shopping クラスの login メソッドを指定
- vipLoginPCut — Shopping クラスの loginAsVIP メソッドを指定
- logoutPCut — Shopping クラスの logout メソッドを指定
- loginAdvice — loginPCut で指定されたジョインポイントにおいて、Customer の情報を元にカウンタを更新
- vipLoginAdvice — vipLoginPCut で指定されたジョインポイントにおいて、Customer の情報を元にカウンタを更新
- logoutAdvice — logoutPCut で指定されたジョインポイントにおいて、Customer の情報を元にカウンタを更新

このアスペクトをウィーブする際、タイミングや順序関係の問題を考慮しないと、プログラムにさまざまな不整合が発生してしまう。

2.5.1　アドバイスの同時バインディングの問題

まず、ポイントカットで指定された全てのジョインポイントと、アスペクトの全てのアドバイスが同時にバインドされない場合を考える。

ProfilingAspect の場合、loginAdvice 、vipLoginAdvice 、logoutAdvice の 3 つのアドバイスが、対応するジョインポイントにバインドされるタイミングに時差が生じてしまうと問題が発生する。

例えば、先の実行例における、customerID=0 の顧客がログインする前に、 ProfilingAspect をウィーブすることとする。このとき、以下の条件が満たされると、customerID=0 の顧客のログインはカウントされるが、ログアウトはカウントされないという問題が発生し、ある時点以降のセッション内の総顧客数、平均年齢、女性の割合の計算結果が不正となってしまう。

- customerID=0 の顧客がログインする前に、loginAdvice と、Shopping クラスの login メソッドがバインドされる。
- customerID=0 の顧客がログアウトした後に、logoutAdvice と、Shopping クラスの logout メソッドがバインドされる。

この場合の実行結果は、以下のようになる

```
-> Enrty ------ customerID:0  headCounter:1  averageAge:24.0  femaleRatio:0.0%
-> Enrty(VIP) - customerID:1  headCounter:2  averageAge:31.0  femaleRatio:50.0%
-> Enrty ------ customerID:2  headCounter:3  averageAge:35.0  femaleRatio:33.3%
   (続く)
```

図 2.4 のように、logout メソッド が呼び出された時点で、logout メソッドと logoutAdvice のメソッドのバインディングがまだ行われていないため、customerID=0 の顧客のログアウトがプロファイリングされていないのだ。

以降、ポイントカットで指定された全てのジョインポイントとアスペクトの全てのアドバイスが同時にバインドされることを**アドバイスの同時バインディング**、それを行わない場合に発生する問題を**アドバイスの同時バインディングの問題**と呼ぶことにする。

既存 Run-time AOP でのアドバイスの同時バインディングの問題

既存の Run-time AOP では、アドバイスの同時バインディングの問題を解決するために、ウィービングが完了するまで全てのスレッドを止めるという手法が用いられることが多い。しかし、このような方法では、シス

テム全体を一度停止してしまうことで、「プログラムを止めずにウィーブを行える」という Run-time AOP の長所を犠牲にしている。僅かな時間でもシステムの実行が止まることが致命的になるシステムでは、このような手法を用いることはできない。

また、Run-time AOP の実装に JPDA を利用している PROSE は、アドバイスを同時にバインディングする機能 (Atomic Weaving という) を提供している。しかし、PROSE の場合、ポイントカットで指定された全てのジョインポイントとアスペクトの全てのアドバイスが同時にバインディング可能になるタイミングをシステムが検知できない。これは、JPDA に「ストップポイントが実際に設置されたかどうかを調べる機能がない」という理由による。PROSE を使うユーザーは、「そろそろ全てのストップポイントが設置されただろう」というタイミングを推測し、手動でウィービングのトランザクションをコミットしなければならない。全てのストップポイントが設置される前にコミットしてしまうとバインディングが同時に行われないから、ユーザーはコミットするまでに十分な時間を確保する必要がある。そのため、PROSE では、ウィービングが完了するまでに時間がかかってしまう。

2.5.2　アドバイスの順序関係の問題

同時バインディングの問題を解決しても、ウィーブのタイミングによっては、他の問題が発生してしまう。バインドされたアドバイスを実際に実行するかどうかを順序関係に基づいて判断しないと、他のアドバイスに依存するアドバイスが、依存先のアドバイスが実行される前に実行されてしまうという問題である。これを、**アドバイスの順序関係の問題**と呼ぶことにする。

例えば、ショッピングサイトの実行結果における、customerID=1 の顧客がログインした後、customerID=0 の顧客がログアウトする前にアスペクトをウィーブした場合を考えてみる。この場合のプロファイリングの出力は以下のようになる。

```
<- Exit ------  customerID:0  headCounter:-1  averageAge:24.0  femaleRatio:-0.0%
-> Enrty ------ customerID:2  headCounter:0  averageAge:Infinity  femaleRatio:NaN%
-> Enrty ------ customerID:3  headCounter:1  averageAge:70.0  femaleRatio:100.0%
<- Exit ------  customerID:1  headCounter:0  averageAge:Infinity  femaleRatio:NaN%
-> Enrty(VIP) - customerID:4  headCounter:1  averageAge:77.0  femaleRatio:100.0%
<- Exit ------  customerID:4  headCounter:0  averageAge:Infinity  femaleRatio:NaN%
-> Enrty ------ customerID:5  headCounter:1  averageAge:62.0  femaleRatio:0.0%
<- Exit ------  customerID:3  headCounter:0  averageAge:Infinity  femaleRatio:-Infinity%
<- Exit ------  customerID:2  headCounter:-1  averageAge:32.0  femaleRatio:100.0%
```

プロファイリング開始後にログインしてセッション内にいる顧客の総数 (headCounter) がマイナスになっているところがある。このような不正なプロファイリングが行われてしまった原因は、アスペクトのウィーブ時に既にログインしている顧客に対してもログアウトをカウントしてしまったことにある。図 2.5 は、それを表した図である。

loginID=0 の customer と loginID=1 の customer は、loginAdvice や vipLoginAdvice が呼ばれていない (カウンタが加算されていない) のに関わらず、logoutAdvice が呼ばれて (カウンタが減産されて) いる。それで、headCouner の値がマイナスになったのである。本質的な問題は、logoutAdvice が loginAdvice や vipLoginAdvice が実行されることに依存しているというところにある。

アドバイスにコードを追加する問題回避法

このような問題については、アスペクトのアドバイス内にコードを追加することで対処することができる。例えば、次のようにすればよい。

まず、ProfilingAspect 内に、以下のフィールドを加えて初期化する。

```
private Map<Advice, Map<Integer, boolean>> dependencyMap;
```

これは、ある Advice を、ある customerID を持った Customer がを実行したかどうかの情報を格納する。

そして、loginAdvice と vipLoginAdvice 内に定義されている adviceBody メソッド内に次のようなコードを加える。これは、loginAdvice や vipLoginAdvice が実行されたことを記録している。

```
Map<Integer, boolean> thisAdviceMap = dependencyMap.get(this);
thisAdviceMap.set(getCustomerID(context), true);
```

最後に、logoutAdvice 内の adviceBody メソッドの先頭に次のコードを加える。これは、loginAdvice と vipLoginAdvice が両方とも実行されていない場合には logoutAdvice も実行しないということを意味する。

```
int customerID = getCustomerID(context);
if (!dependencyMap.get(loginAdvice).get(customerID)
      && !dependencyMap.get(vipLoginAdvice).get(customerID)) {
   return;
}
```

このようにすることで、logoutAdvice が loginAdvice や vipLoginAdvice に依存して問題に対処できた。しかし、この方法には以下のようにいくつか欠点がある。

- コードのサイズが大きくなってしまう
- アドバイス内に本質的なロジック (機能的関心事) と、制御のためのロジック (非機能的関心事) が混在している
- 順序関係は静的な情報なのに、手続き的に記述している

このような欠点があるため、これらの方法を用いることは、可読性・保守性・モジュール性・エラーの導きやすさという観点から望ましくないといえる。

```
public class Customer  {
  private static int nextID = 0;
  private int customerID;
  private int age;
  private String sex;  // "Male" or "Female"

  public Customer(int age, String sex) {
    customerID = nextID++;
    this.age = age;
    if (sex.equals("Male") || sex.equals("Female")) {
      this.sex = sex;
    } else  {
      throw new IllegalArgumentException();
    }
  }

  public int getCustomerID() {
    return customerID;
  }

  public int getAge(){
    return age;
  }

  public String getSex() {
    return sex;
  }

  public String toString() {
    return "customerID:" + customerID + "  age:" + age + "  sex:" + sex;
  }
}
```

図 2.2: Customer クラス

```
public class ProfilingAspect extends Aspect {
  private int headCounter = 0;    // Customer の総数
  private int ageSum = 0;         // Customer の年齢の合計
  private int femaleCounter = 0;  // Customer のうち、女性の総数
  private int getAge(Context context) { /* 略 */ }
  private int getCustomerID(Context context) { /* 略 */ }
  private String getSex(Context context) { /* 略 */ }
  private double calculateAverageAge() { /* 略 */ }
  private double calculateFemaleRatio() { /* 略 */ }
  private void increaseCounter(Context context) { /* 略 */ }
  private void decreaseCounter(Context context) { /* 略 */ }

  private void printStatus() {
    System.out.println("headCounter:" + headCounter + "  averageAge:"
        + roundDouble(calculateAverageAge()) + "  femaleRatio:"
        + roundDouble(calculateFemaleRatio() * 100) + "%");
  }

  public ProfilingAspect() {
    MethodEntryPCut loginPCut = MethodEntryPCut.getInstance("Shopping",
        "login(Customer)");
    MethodEntryPCut vipLoginPCut = MethodEntryPCut.getInstance("Shopping",
        "loginAsVIP(Customer)");
    MethodExitPCut logoutPCut = MethodExitPCut.getInstance("Shopping",
        "logout(Customer)");

    Advice loginAdvice = new Advice(loginPCut) {
      public void adviceBody(Context context) {
        increaseCounter(context);
        System.out.print("-> Enrty ------ customerID:"
          + getCustomerID(context) + "  ");
        printStatus();
      }
    };

    Advice vipLoginAdvice = new Advice(vipLoginPCut) {
      public void adviceBody(Context context) {
        increaseCounter(context);
        System.out.print("-> Enrty(VIP) - customerID:"
          + getCustomerID(context) + "  ");
        printStatus();
      }
    };

    Advice logoutAdvice = new DefaultAdvice(logoutPCut) {
      public void adviceBody(Context context) {
        decreaseCounter(context);
        System.out.print("<- Exit ------  customerID:"
          + getCustomerID(context) + "  ");
        printStatus();
      }
    };

    addAdvices(loginAdvice, vipLoginAdvice, logoutAdvice);
  }
}
```

図 2.3: Profiling アスペクト

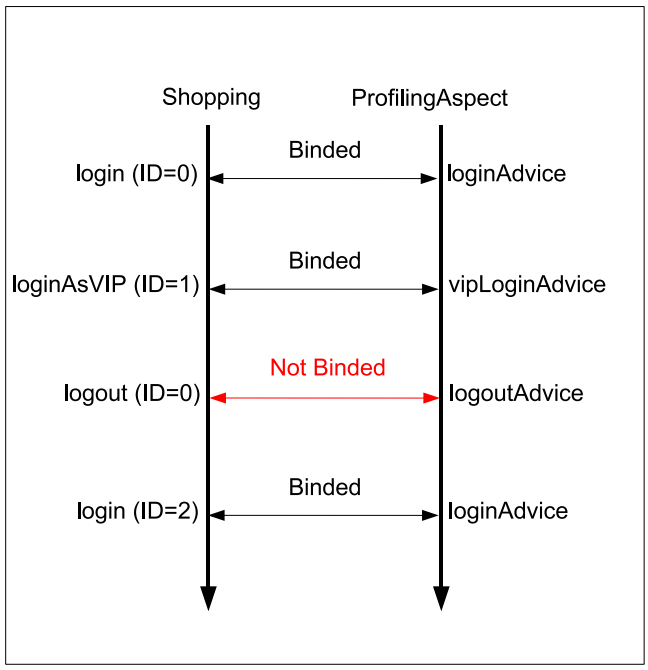

図 2.4: 同時バインディングが行われない場合

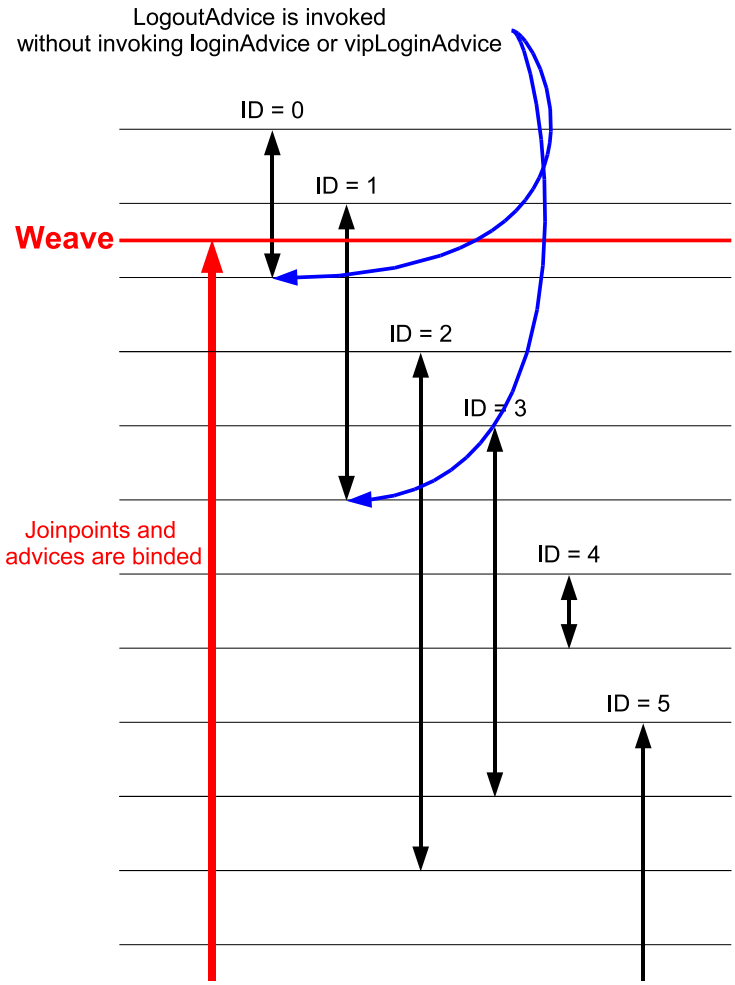

図 2.5: アドバイスの順序関係が考慮されない場合

第3章　本研究の成果

本研究では、順序関係を宣言的に記述すると、それに基づいてアドバイスが順番に活性化される Dynamic AOP System を開発した。

3.1　順序関係の文法とモデル

3.1.1　順序関係を記述する文法

本研究で開発したシステムは、順序関係を記述する API を提供している。順序関係を記述する文法は以下のようになっている。

```
<OrderRelation> := <Advice>.follows(<OrderComponent>);
<OrderComponent> := <Advice> | <OrderComposite>
<OrderComposite> := <AND_Composite> | <OR_Composite> | <NOT_Composite>
<AND_Composite> := new AND(<OrderComponent>*)
<OR_Composite> := new OR(<OrderComponent>*)
<NOT_Composite> := new NOT(<OrderComponent>)
<Advice> := "ObjectOfAnAdvice"
```

- `<OrderRelation>`
  順序関係を指定するステートメント。一つのアドバイスにつき、一つだけ `<OrderRelation>` を定義することができる。

- `<Advice>`
  本研究で開発したシステムでは、アドバイスは first-class object として表現される。具体的には、Advice クラスまたは Advice クラスのサブクラスのオブジェクトを指す変数が `<Advice>` にあたる。

- `<OrderComponent>`
  順序関係を構成する部品である。`<Advice>` であることも、`<OrderComposite>` であることもある。

- `<OrderComposite>`
  いくつかの順序関係が合成された順序関係である。AND(論理積)、OR(論理和)、NOT(論理否定) を用いることで、順序関係を合成し、

新たな順序関係をつくることができる。ただし、NOT の場合だけは、複数の順序関係を合成するのではなく、一つの順序関係を論理的に反転する。

- <AND_Composite>
  複数の順序関係を論理積で合成されてできた順序関係である。
- <OR_Composite>
  複数の順序関係を論理和で合成されてできた順序関係である。
- <NOT_Composite>
  順序関係を論理的に反転してできた順序関係である。

3.1.2　順序関係のモデル

セッションを考えない場合

時刻 t において、アドバイス A が実行されたことを A(t) と表記し、活性化されていないときは ¬ A(t) と表記することとする。ここで、アドバイス A と アドバイス B の二つのアドバイスについて考える。

アドバイス A が時刻 t で実行されるとき、アドバイス B も時刻 t で実行されることは、以下のように表記できる。

$$A(t) \supset B(t)$$

アドバイスの順序関係とは、この式が任意の t について成立することである。よって、A.follows(B) とは以下のような関係となる。

$$A.follows(B) \Leftrightarrow \forall\, t,\ A(t) \supset B(t)$$

また、同様にして、以下のような関係も成立する。

$$A.follows(new\ AND(B,\ C)) \Leftrightarrow \forall\, t,\ A(t) \supset (B(t) \wedge C(t))$$

$$A.follows(new\ OR(B,\ C)) \Leftrightarrow \forall\, t,\ A(t) \supset (B(t) \vee C(t))$$

$$A.follows(new\ NOT(B)) \Leftrightarrow \forall\, t,\ A(t) \supset (\neg\, B(t))$$

セッションを考える場合

時刻 t 、セッション s において、アドバイス A が実行されたことを A(t, s) と表記し、活性化されていないときは ¬ A(t, s) と表記するこ

ととする。ここで、アドバイス A と アドバイス B の二つのアドバイスについて考える。

時刻 t において、アドバイス A が時刻 s で実行されるとき、アドバイス B も時刻 s で実行されることは、以下のように表記できる。

$$A(s, t) \supset B(s, t)$$

セッションを考えた場合、アドバイスの順序関係とは、この式が任意の s と t について成立することである。よって、A.follows(B) とは以下のような関係となる。

$$A.follows(B) \Leftrightarrow \forall s, \forall t, A(s, t) \supset B(s, t)$$

また、同様にして、以下のような関係も成立する。

$$A.follows(new\ AND(B, C)) \Leftrightarrow \forall s, \forall t, A(s, t) \supset (B(s, t) \wedge C(s, t))$$

$$A.follows(new\ OR(B, C)) \Leftrightarrow \forall s, \forall t, A(s, t) \supset (B(s, t) \vee C(s, t))$$

$$A.follows(new\ NOT(B)) \Leftrightarrow \forall s, \forall t, A(s, t) \supset (\neg B(s, t))$$

3.1.3　セッションの記述法

本研究で開発したシステムでは、セッションごとにアドバイスの活性化を行うことができる。その際、セッションは自由に定義することが可能である。例えば、以下のようなものが考えられる。

- スレッドの ID
- プログラムの実行時の値

本研究では、セッションを定義するために用いるポリシーを、セッションポリシーと呼ぶ。ショッピングサイトの例の場合では、以下のように顧客の ID を元にセッションを決定するセッションポリシーを定義すればよい。

```
setSessionPolicy(new SessionPolicy() {
  public Object getSessionID(Context context) {
    return getCustomerID(context);
  }
});
```

SessionPolicy インターフェースには、public Object getSessionID(Context context) というメソッドが定義されているので、これを実装すればよい。アドバイスが呼び出されようとするときには、必ずこの setSessionID メソッドが呼び出される。そして、getSessionID メソッドの戻り値によってセッションが決定される。セッションの ID が同一ならば 同一セッションとみなされる。

第4章　実装

4.1　JPDA

本システムでは、Java Platform Debugger Architecture (JPDA) を用いている。

JPDA は、Java Debugger Wire Protocol (JDWP) というプロトコルと、Java Virtual Machine Tools Interface (JVMTI) と Java Virtual Machine Dubugger Intervace (JDI) という 2 つのインターフェース、それとこれらを結びつける 2 つのソフトウェアコンポーネント (バックエンドとフロントエンド) から構成されている。

1. Debugger
   デバッグする側のプロセス
2. Debuggee
   デバッグされる側のプロセス
3. UI
   デバッガーの側でのユーザーインターフェース
4. フロントエンド
   UI とバックエンドを結び付けるコンポーネント。UI とは JDI、バックエンドとは JDWP を用いて通信する。
5. バックエンド
   デバッギー側の VM とフロントエンドを結び付けるコンポーネント。VM とは JVMTI、フロントエンドとは JDWP を用いて通信する。
6. VM
   デバッギー側の VM

4.2　本システムのアーキテクチャー

本システムでは、Run-time AOP を JPDA を用いて実現している。同様の実装法を用いている Run-time AOP System には、PROSE や Wool

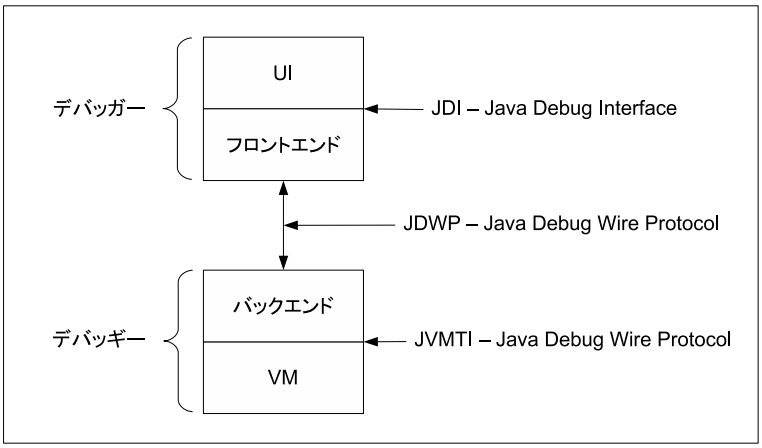

図 4.1: JPDA のアーキテクチャ

がある。この実装法では、アスペクトがウィーブされる側のプログラムは Debuggee VM で、アスペクトは Debugger VM で別々に実行される。

一般的に、AOP では、プログラムの実行がジョインポイントに到達すると、そのジョインポイントがアドバイスのポイントカットで指定されている場合は、そのアドバイスを実行してから元のコードに復帰する。この流れは以下のようになる。

1. ジョインポイントの前のコードを実行
2. プログラムがジョインポイントに到達
3. ジョインポイントがアドバイスのポイントカットで指定されているかどうかを判別 (コードが直接埋め込まれる実装ではコンパイル時に全て判別される場合もある)
4. もし、ジョインポイントがアドバイスのポイントカットで指定されている場合は、対応するアドバイスを実行する
5. ジョインポイントの後のコードを実行

本システムは、ポイントカットで指定されるジョインポイントに JPDA を用いてブレークポイントを設定する。スレッドの実行がブレークポイントに到達すると、そのスレッドは停止し、別プロセスでアドバイスが実行された後、スレッドは再開される。これを細かく書くと以下のようになる。

1. Debuggee VM のスレッドがジョインポイントの前のコードを実行
2. Debuggee VM のスレッドがジョインポイントに到達

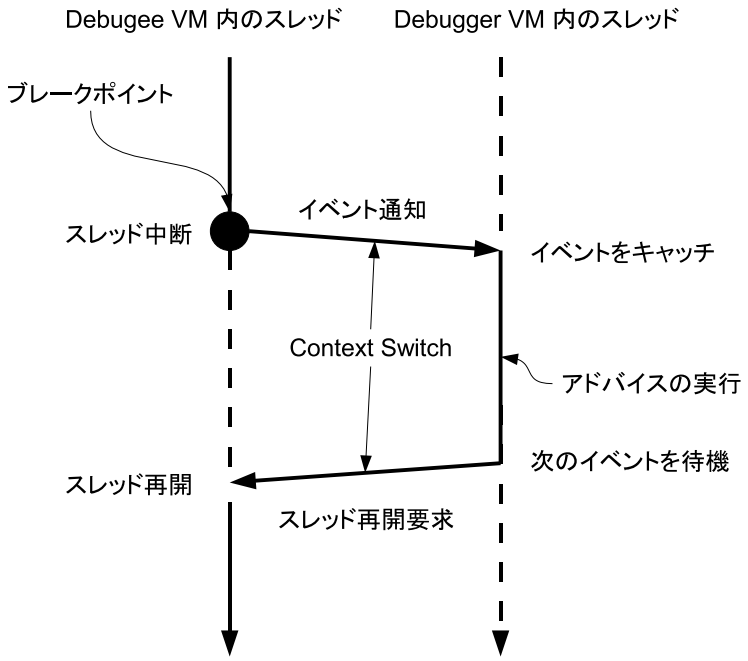

図 4.2: 本システムてのアドバイス呼び出し

3. Debuggee VM のスレッドが中断される
4. Debuggee VM から Debugger VM にイベントが通知される
5. Debugger VM で待機しているスレッドがイベントをハンドル
6. Debugger VM でイベントをハンドルしたスレッドは、イベントからジョインポイントを割り出し、対応するアドバイスを実行する (Debuggee VM のメソッドを呼び出したり、変数を参照・変更することも可能)
7. Debugger VM が Debuggee VM のスレッドを再開する
8. Debugger VM のスレッドがジョインポイントの後のコードを実行 (Debuggee VM のスレッドは次のイベントを待機する)

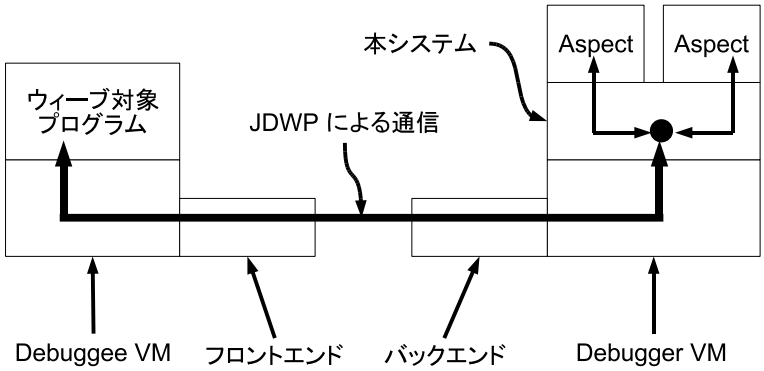

図 4.3: 本システムのアーキテクチャ

4.3　アドバイスの同時バインディングの問題の解決法

アドバイスの同時バインディングの問題に関しては、2.5.1 節 (21 ページ) で述べた。JPDA にはブレークポイントが実際に設定されたかどうかを調べる機能がないということが、この問題を解決する障害となっていた。

問題は、ブレークポイントを有効･無効にする com.sun.jdi.request.EventRequest クラスの void setEnabled(boolean val) メソッドが非同期なことである。このメソッドから復帰した時点では、ブレークポイントを有効・無効にする要求を Debugger VM が Debuggee VM に送信する (準備をする) だけであって、実際にブレークポイントが設定されているわけではない。

また、JPDA にはブレークポイントの登録状況を調べる機能が存在する。com.sun.jdi.request.EventRequestManager クラスの、breakpointRequests メソッドであり、仕様は以下のようになっている。

> 有効および無効なブレークポイント要求の、変更不可能なリストを返します。
> このリストはこれらの要求のライブビューであり、
> 要求が追加または削除されると、その変更が即時に反映されます。
>
> 戻り値:
> すべての `BreakpointRequest` オブジェクトのリスト

一見、これを用いれば、ブレークポイントが**実際に**設定されているかどうかを調べることができそうだが、問題がある。この仕様でのライブビューというのは、Debugger VM 内に存在するブレークポイントの登録情報のライブビューであって、Debuggee VM 内に実際に設定されたブレークポイントのライブビューではないのだ。そのため、このメソッドによって取得した情報ではブレークポイントは有効になっているが、実際には設定されていないということが起こり得る。

そのため、ブレークポイントが実際に設定されているかどうかを調べるには、JPDA で提供されている API に頼らずに、何かしらの工夫を行わなければならない。

本研究では、ブレークポイントを 2 重に設定することで、この問題を解決する。この方法は JPDA の以下の性質を用いている。

- 1 つの場所に複数のブレークポイントを設定することができる。
- ブレークポイント設定の要求はキューに入れられ、順番に設定される。

アスペクトの全アドバイスの全ポイントカットで指定されたジョインポイントを $J = \{j_k \mid k \in \{1, 2, 3, ...\}\}$ にとすると、本システムでは、以下の順にブレークポイントを設定する。

```
for (int k = 0; k < J.length; k++) {
    J[k] に、「1 つ目」という属性を付加したブレークポイントを設置
}
for (int k = 0; k < J.length; k++) {
    J[k] に、「2 つ目」という属性を付加したブレークポイントを設置
}
```

プログラムの実行が、「2 つ目」という属性を付加したブレークポイントに到達する時点では、既に「1 つ目」という属性を付加したブレークポイントは全て設定されている。よって、その時点で、ウィーブされるアスペクトのアドバイスを呼び出すジョインポイントには、全てブレークポイントが設定されているということが保障される。

つまり、「2 つ目」という属性が付加されたブレークポイントのイベントを Debugger VM がキャッチした時点で、ウィーブ対象のプログラムのジョインポイントと、ウィーブされるアスペクトをバインドすればよいのである。

本研究で開発したシステムでは、ブレークポイントが全て設定されたかどうかを表すフラグ (Weaving Flag) を用意した。フラグの初期値は false

であり、「2 つ目」という属性が付加されたブレークポイントからのイベントを Debugger VM がキャッチしたら、そのフラグを true にする。「1 つ目」という属性が付加されたブレークポイントからのイベントは、キャッチしてもすぐには実行せず、フラグを参照し、それが true になっていればアドバイスを実行するようにする。

ウィービングの手順を細かく記述すると、以下のようになる。

1. Debugger VM が、アスペクトを呼び出す全てのジョインポイントに、「1 つ目」という属性を付加したブレークポイントを送信する。
2. Debugger VM が、アスペクトを呼び出す全てのジョインポイントに、「2 つ目」という属性を付加したブレークポイントを送信する。
3. Debugger VM で待機しているスレッドがイベントをハンドル。
4. Debugger VM は、イベントから属性を調べ、それが「2 つ目」だった場合は、Weaving Flag を true にする。そして、Debuggee VM のスレッドを再開する。
5. イベントの属性が「1 つ目」だった場合は、イベントから対応するアスペクトを判別する。
6. 対応するアスペクトの Weaving Flag が false の場合は、Debuggee VM のスレッドを再開する。
7. 対応するアスペクトの Weaving Flag が true の場合は、アドバイスを実行した後、Debuggee VM のスレッドを再開する。

図 5.1 は、ブレークポイントを設定する順序を視覚的に表している。状態 0 から状態 2 の間は、ポイントカットで指定された全てのジョインポイントにブレークポイントが設定されていないので、この時点でアドバイスを実行してはならない。状態 3 以降は、ポイントカットで指定された全てのジョインポイントとウィーブされるアスペクトのアドバイスをバインディング可能な状態である。だが、すぐにバインディングが行われるわけではない。2 つ目のブレークポイントにまだ到達してない場合、状態 3 のジョインポイント A・B・C 、状態 4 のジョインポイント B・C 、状態 5 のジョインポイント C にスレッドが到達した場合は、アドバイスは実行されない。つまり、これらの場合はバインディング可能であるが、それが可能であるという情報を検出し、Debugger VM に送信することができないために、遅延が行われるわけである。

このようにすることで、アドバイスの同時バインディングの問題を解決することができた。

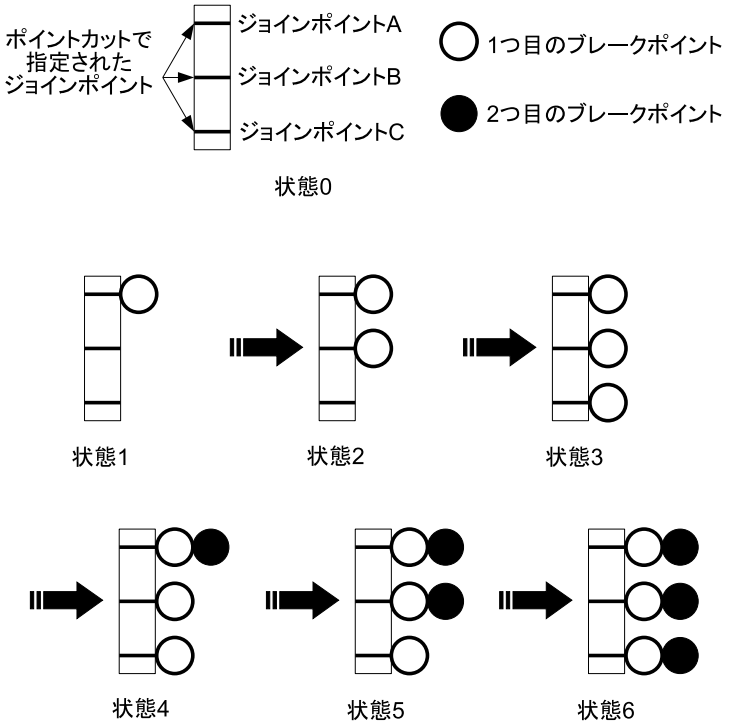

図 4.4: ブレークポイントを設定する順序

4.4　アドバイスの順序関係の問題の解決法

プロファイリングアスペクト 2.3 に問題があることは 2.5.2 節 (22 ページ) で述べた。これは、アドバイスの活性化に順序関係を定義しないことが原因だった。

既存の Dynamic AOP System でも、アスペクト内にアドバイスの順序関係を制御するフラグや、それらのマップなどを定義し、アドバイス内でそれらを参照・変更することでアドバイスの順序関係を元にウィービングを制御できる。しかし、そのような方法では、コードのサイズが大きくなってしまう問題がある。また、アドバイス内にアドバイス間の順序関係を考慮するコードを手続き的に記述されてしまうと、モジュール性・可読性が低下してしまい、プログラムに間違いが発生しやすくなるという問題もある。

そこで、本研究は、順序関係を AND 、OR 、NOT という論理演算子

を用いて宣言的に定義する API を提案する。

プロファイリングアスペクト 2.3 の例では、logoutAdvice が loginAdvice および vipLoginAdvice に依存している。この依存性に対して、2.5.2 節 (23 ページ) では、フラグへのマップを定義し、アドバイス間の順序関係をアドバイス内に宣言的に記述していた。

次は、アスペクトに追加されるフィールドである。

```
private Map<Advice, Map<Integer, boolean>> dependencyMap;
```

次は、loginAdvice と vipLoginAdvice に追加されるコードである。顧客の ID を単位にして、このアドバイスが実行されたことを、フィールドのマップに追加している。

```
Map<Integer, boolean> thisAdviceMap = dependencyMap.get(this);
thisAdviceMap.set(getCustomerID(context), true);
```

次は、logoutAdvice の先頭に追加冴えるコードである。顧客の ID を単位として、loginAdvice または vipLoginAdvice が実行されたかどうかを参照している。

```
int customerID = getCustomerID(context);
if (!dependencyMap.get(loginAdvice).get(customerID)
      && !dependencyMap.get(vipLoginAdvice).get(customerID)) {
   return;
}
```

これに対して、本システムの API を用いれば、以下のようにわずかなコードを追加するだけで同様のことを行うことができる。

アスペクト内には、ウィービングの単位を指定する以下の内部クラスを追加する。

```
Criterion criterionByID = new DefaultCriterion() {
  public Object getKey(Context context) {
    return getCustomerID(context);
  }
};
```

そして、アスペクトのコンストラクタには、以下のコードを記述する。アドバイスの順序関係を宣言し、ウィービングの単位を設定している。

```
logoutAdvice.depend(new OR(loginAdvice, vipLoginAdvice));
setCriterion(criterionByID);
```

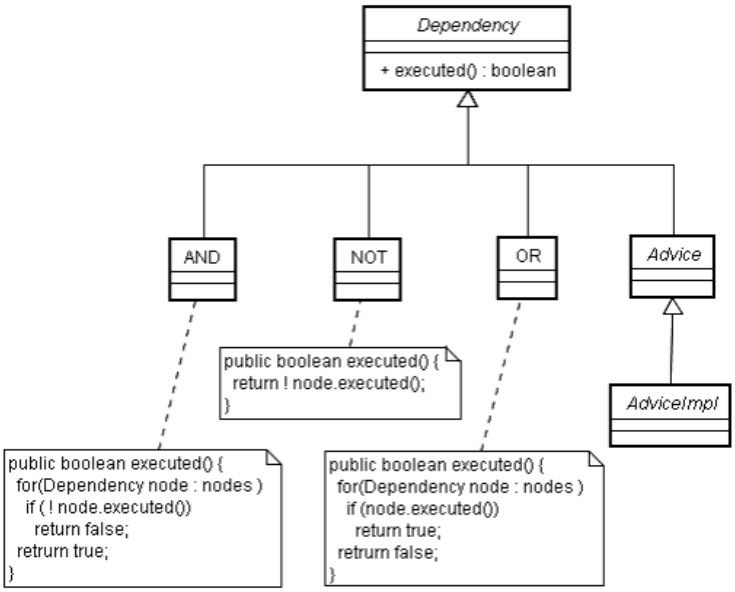

図 4.5: 順序関係をもつモジュールのクラス図

この API を利用した場合、以下のような長所がある。

- 複数のアドバイス間に散らばっていたウィービング単位を参照する部分が、Criterion オブジェクトの中に集約されている。ウィービング単位の参照が必要なときは本システムが行ってくれる。
- アドバイス間の論理的な順序関係を、そのまま論理式で記述できる。
- API を用いない場合に比べ、コード量が少なくなる。
- API を用いない場合に比べ、可読性が高い。

順序関係を解決する API は、Composite パターン [5] を用いて、図 4.5 のように実装した。順序関係を解決するために、アドバイスのツリーを根から辿り、アドバイスが既に活性化されているかどうかを調べる。

第5章　実験と検証

本研究で開発したシステムは、ウィーブ対象プログラムからアドバイスを呼び出すようにするために、ブレークポイントを設定していた。アドバイスを呼び出すために設定されたブレークポイントからのイベントをキャッチすると、無条件にアドバイスを実行するという単純な方法には問題があるのだった。ブレークポイントが設定される時間に遅延が生じることによって、呼び出されることを想定したアドバイスが実際には呼び出されないという問題だ。

また、本システムでは、ブレークポイントを二重に設定することで、アドバイスが呼び出し可能になるタイミングを調べている。よって、ウィーブにかかる時間はブレークポイントを設定する時間次第である。

このように、既存のシステムにおける単純な方法の問題点も、本システムにおける解決法も、ブレークポイントを設定する時間に強く依存している。そこで、ブレークポイントが設定される時間を調べた。実験環境は、Windows XP、AMD Athlon(tm) MP 2400+ 2.00GHz、 1.00GB RAM である。

表 5.1、図 5.1 は、設定されるブレークポイントの個数を変え、全てのブレークポイントが設定される時間を計測している。設定されるブレークポイントの数は、0 個から 1000 個まで 100 個ずつ増やして測定した。(これはイベントを通知しないブレークポイントの個数。イベントを通知するブレークポイントがこれとは別に 1 個存在するので、厳密には 1 個から 1000 個 までである。) 各々の測定は以下のような方法を 100 回行い、平均をとった。(工夫をしない場合、Debuggee VM から Debugger VM へ非常に短い間隔でデータ量が送信され、オーバーヘッドが発生してしまうので、ブレークポイントからのイベント通知に、2 回までというフィルターをかけた。)

- Debuggee VM 上の対象プログラムで空のループを回す
- Debugger VM 上で 1 回の計測を開始し、Debuggee VM にブレークポイントをセットする (最後のブレークポイントだけはループの内部、それ以外はループの外部にセットする)
- Debuggee VM から Debugger VM にイベントが通知されたら 1 回

表 5.1: JPDA におけるブレークポイントの実行時間

設定したブレークポイントの個数	時間 (ミリ秒)
0	18.042
100	36.171
200	57.476
300	72.674
400	79.759
500	94.596
600	126.15
700	133.39
800	153.46
900	162.90
1000	169.91

の測定を終了する。

図 5.1 のように、ブレークポイントの設定にかかる時間は、設定したブレークポイントの数の一次関数となっている。つまり、１つのブレークポイントの設定にかかる時間は一定である。

実験結果が一次関数であることより、１つのブレークポイントの設定にかかる時間は、以下のように計算できる。

1 つのブレークポイントの設定にかかる時間
＝（1000 個のブレークポイントの設定にかかる時間
　　－ 0 個のブレークポイントの設定にかかる時間）/ 1000

```
(169.91 (ms) - 18.042(ms)) / 1000 = 0.1519(ms)
```

ブレークポイントの設定にかかる時間が一定であるということより、本システムでのウィーブにかかる最短時間が計算できる。二重に設定されるブレークポイントのうち、最初に設定されるものが設定された直後に、プログラムの実行がそのブレークポイントに到達したときが最短なので、これは以下の式で計算できる。

本システムでのウィーブにかかる最低時間
＝（設定するブレークポイント数 + 1）× 1 つのブレークポイントをセットする時間

　また、二重に設定されるブレークポイントのうち、最後に設定されるものが設定される直前に、プログラムがそのブレークポイントを通過してしまったときが、ウィーブに最も時間がかかる場合である。よって、これは以下の式で計算できる。

　本システムでのウィーブにかかる最長時間
＝ 2 × 設定するブレークポイント数 × 1つのブレークポイントをセットする時間

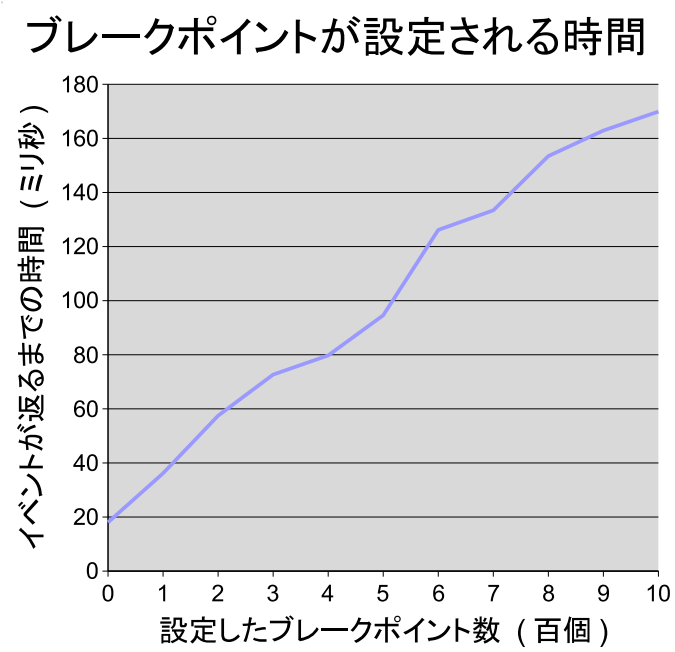

図 5.1: ブレークポイントの設定にかかる時間

第6章　まとめと今後の課題

6.1　まとめ

本研究は、従来の単純な手法でアスペクトをウィーブした場合、アドバイスの実行に問題が発生することに注目した。ウィーブする対象プログラムが特定の制御フローに沿って実行されることを前提としたアスペクトの場合、前提としている制御フローの途中でウィーブされることで、アスペクトのアドバイスが予期せぬ順序で呼び出されてしまう。

そのような問題を解決するために、本研究では、アドバイスに活性化という概念を導入した。活性化とは、アドバイスが実行可能になることをいう。アスペクトがウィーブされた場合、無条件にアドバイスを実行するのではなく、正しい順序でアドバイスが実行可能になるまで、アドバイスの呼び出しを無視する。そして、アドバイスが正しい順序で活性化されると、活性化されたアドバイスからを呼び出すことが可能となる。

また、アドバイスの活性化はセッションごとに行う必要がある。この場合のセッションとは、一般的な意味のセッションのことも含むが、広義には、ある観点から見たときに同一の属性をもつジョインポイントの集合を指す。このようなことを行わなければならないのは、一般的に、プログラム中に制御フローが複数存在することが原因である。具体的には、複数のスレッドが並列して実行される場合が考えられる。この場合は、スレッドの ID が制御フローの ID となっている。また、ある観点から見たときに同一の属性をもつジョインポイントを結んだ線も、抽象的なレベルでの制御フローと考えられる。この場合は、ジョインポイントから得られる ID が制御フローの ID となる。

アドバイスの活性化は、従来の方法でも制御することができた。以下のようにすればよい。アドバイスの外部 (例えばアスペクトのフィールド) に、どのアドバイスがどのセッションにおいて実行されたかという情報を保存する。そして、各々のアドバイス内に、その情報を参照・変更して、活性化の順序を制御するロジックを書けばよい。

しかし、そのような方法では、コードのサイズが大きくなってしまう問題がある。また、アドバイス内にアドバイス間の順序関係を考慮するコードが手続き的に記述されてしまうと、モジュール性・可読性が低下してしまい、プログラムに間違いが発生しやすくなるという問題もある。

本研究では開発したシステムでは、順序関係を AND 、OR 、NOT という論理演算子を用いて宣言的に定義する API を提供した。この API によりアドバイス間の順序関係を記述することで、アドバイス間の順序関係をアドバイスの中身から分離できる。本システムは、API によって定義された順序関係を元に、アドバイスの活性化を自動的に判断する。

このような API を実装するときには、ブレークポイントの設定が遅れると、ジョインポイントからアドバイスが呼び出されるべきときに、呼び出されないという問題を考える必要がある。アドバイスを活性化する順序関係を解決するためには、まず、この問題を解決しなければならない。

この問題に関しては、全スレッドを一度止めることで、ポイントカットで指定された全てのジョインポイントからアドバイスを呼び出し可能にする Dynamic AOP System は既に存在するが、その実装法ではウィーブのためにアプリケーション全体の実行が止まってしまうため、時間コストが大きいという点で問題があった。また、スレッドを止めずに、ポイントカットで指定された全てのジョインポイントからアドバイスを呼び出し可能にする Dynamic AOP System も既に存在するが、ジョインポイントからアドバイスを呼び出し可能になるタイミングをシステムの側で検知できない。そのため、十分な時間を確保した後に、手動でトランザクションをコミットしなければならず、ジョインポイントからアドバイスが呼び出されないというリスクを回避するためには長時間待たなければならないという問題があった。

本システムでは、ブレークポイントを二重に設定するという方法を用いた。その方法を用いることで、スレッドを止めることなく、アドバイスを呼び出し可能になった時点をシステムが検知することが可能となった。

6.2　今後の課題

JPDA のブレークポイントを用いる本研究の実装法では、ウィーブ対象プログラムに振舞いを追加することはできるが、ウィーブ前から既に存在する振舞いを取り除くことはできない。しかし、この制限は、JPDA の HotSwap を用いることで回避できる。HotSwap を用いれば、既存のメソッド内部のバイトコードを、新しいバイトコードで置き換えることができる。HotSwap を用いた場合、アドバイスは Debugger VM ではなく、Debuggee VM の中で実行されるので、Debugger VM の内部でアドバイスの活性化を制御する本研究のアイデアだけでは対応できない。そのため、HotSwap を用いた場合もアドバイスを順序関係に基づいて活性化できるような方法を考えることが、今後の課題の一つである。

また、本研究では、ウィーブ対象プログラム中で参照される値をアドバ

表 6.1: 本研究で開発したシステムと DAC++ の比較

	本システム	DAC++
考慮の対象	同一アスペクトのアドバイス	ウィーブ対象プログラム
活性化の単位	アドバイス	アスペクト
ベースとなる言語	Java	C++

イスが変更する場合を考えていない。このような場合は、アドバイスの実行が、他のアドバイスだけではなく、ウィーブ対象プログラムにも影響を与えてしまうので、衝突を回避するために、一層の工夫を行わなければならない。アスペクトをウィーブすることの、対象プログラムへの影響を考慮したシステムとしては、DAC++ [6] がある。DAC++ では、ウィーブを行うことで衝突が発生してしまうメソッドを調べ、それをメタデータとして保存し、それらのメソッドの実行が終了したタイミングでアスペクトをウィーブするという方法を用いている。

本研究と DAC++ は、表 6.1 の通り、多くの点で異なっている。アドバイスという細かい単位で活性化を行うという本システムの長所を生かしつつ、ウィーブ対象プログラムへの影響を考慮することが、真に安全なウィービングを実現する Dyanamic AOP System を実現するための重要な課題である。

参考文献

[1] Andrei Popovici, Thomas Gross and Gustavo Alonso: Dynamic weaving for aspect-oriented programming, *Proceedings of the 1st international conference on Aspect-oriented software development*, pp. 141–147 (2002).

[2] Christoph Bockisch, Michael Haupt, Mira Mezini and Klaus Ostermann: Virtual machine support for dynamic join points, *Proceedings of the 3rd international conference on Aspect-oriented software development*, pp. 83–92 (2004).

[3] Daby Suvee, Wim Vanderperren and Viviane Jonckers: JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development, *Technical Report 02/ll/INFO, Ecole des Mines de Nantes*, pp. 21–29 (2003).

[4] R Douence and M Sudholt: A model and a tool for Event-based Aspect-Oriented Programming (EAOP), *Technical Report 02/ll/INFO, Ecole des Mines de Nantes*.

[5] Erich Gamma, Richard Helm, Ralph Johnsona and John Vlissids: *Design Patterns: Element of Reusable Object-Oriented Software*, Addison-Wesley (1995).

[6] S.Almajali, T.Elrad: Coupling Availability and Efficiency for Aspect Oriented Runtime Weaving Systems, *Proceedings of the Second Dynamic Aspects Workshop (DAW05)*, pp. 47–56 (2005).

[7] Nanning Aspects, `http://nanning.codehaus.org`.

[8] Yoshiki Sato, Shigeru Chiba and Michiaki Tatsubori: A Selective, Just-in-Time Aspect Weaver, *Proceedings of Second International Conference, on Generative Programming and Component Engineering (GPCE 2003)*, pp. 189–208 (2003).

[9] 千葉滋: アスペクト指向入門, 技術評論社 (2005).

[10] S.Almajali and T.Elrad: Coupling Availability and Efficiency for Aspect Oriented Runtime Weaving Systems, *Proceedings of the Second Dynamic Aspects Workshop (DAW05)*, pp. 47—56 (2005).

付 録A　簡易 API リファレンス

本研究で開発したシステムの主要な API です。

A.1　Aspect クラス

アスペクトを実装する場合は、必ず Aspect クラスを継承しなければならない。

`public Aspect ()`

空のアスペクトを生成する。

`public void registerAdvices(Advice ... advices)`

アスペクトに複数のアドバイスを登録する。

`public void setSessionPolicy(SessionPolicy sessionPolicy)`

アスペクトにセッションポリシーを登録する。

A.2　Advice クラス

アドバイスを実装する場合は、必ず Advice クラスを継承しなければならない。

`public Advice ()`

空のアドバイスを生成する

`public void adviceBody(Context context)`

アドバイスの内部を実行する。新規アドバイスを実装する場合は、このメソッドをオーバーライドする。

`public void follows(DependencyComponent dependency)`

アドバイスの順序関係を定義する。このアドバイスが呼び出されたセッションと同一セッションで dependency が活性化された場合のみ、このアドバイスは実行される。

A.3　PCut クラス および PCut クラスのサブクラス

A.3.1　PCut クラス

ポイントカットに対応するクラス。以下のクラスは全て PCut クラスのサブクラスである。

A.3.2　FieldAccessPCut クラス

```
public static FieldAccessPCut getInstance(String className, String fieldName)
```

className クラス の fieldName フィールドのアクセス時を指定するポイントカットを生成する。

A.3.3　BreakpointPCut クラス

```
public static BreakpointPCut getInstance(String className, int lineNumber)
```

className クラス の lineNumber 行を指定するポイントカットを生成する。

A.3.4　MethodEntryPCut クラス

```
public static MethodEntryPCut getInstance(String className, String methodName)
```

className クラス の methodName メソッドの実行を開始する時点を指定するポイントカットを生成する。

A.3.5　MethodExitPCut クラス

```
public static MethodEntryPCut getInstance(String className, String methodName)
```

className クラス の methodName メソッドの実行が終了した時点を指定するポイントカットを生成する。

A.3.6　FieldModificationPCut

```
public static FieldAccessPCut getInstance(String className, String fieldName)
```

className クラス の fieldName フィールドの変更時を指定するポイントカットを生成する。

A.4 DependencyComponent インターフェース とそれを実装するクラス

A.4.1 DependencyComponent インターフェース

順序関係を定義できるインターフェース。以下のクラスは全て DependencyComponent インターフェースを実装している。

A.4.2 AND クラス

```
public AND(DependencyComponent ... dependencies);
```

dependencies が全て活性化されたときのみ、活性化される DependencyComponent を生成する。

A.4.3 OR クラス

```
public OR(DependencyComponent ... dependencies);
```

dependencies のうち、最低でも 1 つが活性化されたとき、活性化される DependencyComponent を生成する。

A.4.4 NOT クラス

```
public NOT(DependencyComponent dependency);
```

dependencies が活性化されていない場合のみ、活性化される DependencyComponent を生成する。

A.4.5 Advice クラス

A.2 を参照

A.5 Context クラス

```
public Value getLocalVariable(String varName)
```

ジョインポイントのローカル変数 varName を返す。

```
public Value getLocalVariable(String fieldName)
```

ジョインポイントのフィールド fieldName を返す。

```
public ThreadReference getThread()
```

ジョインポイントに到達したスレッドへの参照を返す。

```
public ObjectReference getThisObject()
```

ジョインポイントが含まれるオブジェクトの参照を返す。