アスペクト指向プログラミングにおける テストに基づいたポイントカットの提案

櫻井 孝平　増原 英彦

本研究はアスペクト指向プログラミング (AOP) の新たなポイントカット機構として，**テストに基づいたポイントカット**を提案する．AOP 言語はアスペクトの適用する時点を指示しなければならない．既存の AOP 言語のポイントカットは，アスペクトが適用されるプログラム中の型名やメソッド名により指示を行うため，プログラムの些細な変更に応じてアスペクトの変更が必要であった．テストに基づいたポイントカットでは，テストを通じて間接的にアスペクトが適用される時点を指示する．そのため，プログラムの変更時にテストも修正される前提の下では，ポイントカット記述の変更が必要なくなる．実際にいくつかの事例に対してテストに基づいたポイントカットを利用し，ほとんどの場合に既存の AspectJ による記述よりも変更に強いポイントカット記述が可能であることを確認した．

This paper propose *test-based pointcuts*, new pointcut mechanism for aspect-oriented programming(AOP). In AOP language programmer must specify aspect applied point. Existing pointcuts of AOP language need for changes of aspects by minor modifications of the program, because these pointcuts use method or type name to specify applied points of the program. In test-based pointcuts programmer specify applied points in the program through tests indirectly. Thus it need not to change pointcut description under assumption that tests are modified for modification of the program. Actually we use test-based pointcuts for some case studies, confirm that in most cases more robust pointcut descriptions are possible than descriptions by existing AspectJ.

1　はじめに

AspectJ [11] のようなアスペクト指向プログラミング (AOP) 言語には，アスペクトとして横断的関心事のモジュール化を可能にする機構として，ポイントカットとアドバイスがある．この機構では，ポイントカットがプログラム実行中の特定の動作であるジョインポイントにマッチしたときに，アスペクトの振舞いであるアドバイスが実行される．例えば，典型的な横断的関心事であるログ出力のためには，ログを出力させたい動作にマッチするポイントカットを記述し，ログを出力する命令をアドバイスとして定義する．

既存の AOP 言語には 2 つの問題がある．1) 対象プログラムの変更により，ポイントカット記述が本来意図したジョインポイントにマッチしなくなることがある [13] [10]．2) ジョインポイントの**実行履歴**によってアスペクトに異なる挙動を行わせるのは難しい．

1 の問題は，多くのポイントカット記述がプログラム中の型やメソッドの名前に基づいていることに起因する．ポイントカット記述が一貫しているとは，対象プログラムの変更に関らず，開発者の意図したジョインポイントのみにマッチし続けることを言う．以降では，このような問題を**ポイントカット記述の非一貫性の問題**と呼ぶ†1．プログラムの振舞いに基づいたポイントカット記述，例えば cflow ポイントカットを利用する場合でも，メソッドの名前を指定する必要があるので非一貫性の問題は生じ得る．

2 の問題は以降では，ジョインポイントの実行履歴に関する問題と呼ぶ †2．例えば，あるメソッドが行

Test-based Pointcuts for Aspect-Oriented Programming
Kouhei Sakurai, Hidehiko Masuhara, 東京大学 大学院総合文化研究科, Graduate School of Arts and Sciences, the University of Tokyo

†1　文献 [13] [10] では *fragile pointcut problem* と呼んでいる．

†2　tracematches [5] などの機構により解決できる．

うに認証の成否に応じて，異なる対応したアドバイスを実行する場合を考える．メソッド自体が認証の失敗の仕方によらず正常終了するように定義されていた場合，認証のメソッド実行中にどのような動作や分岐を行ったかによって実行するアドバイスを選ばなくてはいけない．そのような場合は，認証を行うメソッド内部で呼び出されるメソッドの結果を状態変数に記録しておくなどの煩雑な方法をとらなければならない．また，このような方法はより変更されやすいメソッド定義内部に依存しているため，非一貫性の問題もより起こりやすい．

本稿では**テストに基づいたポイントカット**(*Test-based Pointcuts*) を AspectJ の拡張言語機構として提案し，前述の問題を解決する．テストに基づいたポイントカットは，アドバイスの実行時点を指定するためにメソッドのテストのためのプログラムである単体テストケースのテスト中で用いられる変数名を用いる．ここで単体テストケースのテストとは，JUnit [4] におけるテストのためのメソッドを指す．ポイントカットは参照した単体テストケースのテストがテスト対象とするメソッドの実行履歴を得るジョインポイントにマッチする．

テストに基づいたポイントカットにより，非一貫性の問題は次のように軽減される．開発者は，プログラムと単体テストケースの一貫性を常に保っているのが通例である．つまり，プログラム中のメソッド名やクラス名を変更した場合，それらを参照する単体テストケースのテストも同時に変更する．従って，テストに基づいたポイントカットが単体テストケースを通して間接的にプログラムのジョインポイントにマッチすることで，ポイントカットは修正しなくとも一貫性が得られる．

また，テストに基づいたポイントカットは，テストの異なる実行履歴ごとにジョインポイントのメソッドが同一でもマッチを行うことができる．1 つのテスト対象メソッドに対する単体テストケースのテストは，異なる状態や入力の代表値ごとに複数定義される．それらを異なる実行履歴を得るテストとして，異なった実行履歴を持つジョインポイントのマッチに利用できる．例えば，認証のメソッド呼出しにおいて，原因の異なる失敗を扱う (実行履歴の異なる) テストごとにポイントカットを記述する．これらのポイントカットが，それぞれ対応する実行履歴を持つ実際に認証の失敗するメソッド呼出しにマッチすることで，認証の失敗の原因に応じて異なるポイントカット定義が可能となる．

以降の構成は，まず 2 節では既存のポイントカットの問題を具体的な例によって説明する．続く 3 節ではテストに基づいたポイントカットについて提案し，2 節で挙げた問題の例がどのように解決されるかを説明する．4 節では実際のプログラムに対する，テストに基づいたポイントカットの有効性を確認するための実験について説明する．5 節では処理系の実現方法について説明する．6 節では将来的な課題に関する議論と，関連研究について述べる．7 節がまとめである．

2 既存のポイントカット記述の問題点

本稿では，AspectJ においてポイントカット記述の一貫性の維持が困難で，異なる実行履歴にマッチを行う必要がある例として，FTP クライアントプログラム中の通信部分を用いる．4.2 節には実際の FTP クライアントに対する事例研究がある．

本節では，まず Java による FTP 接続の記述を説明し，続いて AspectJ による横断的関心事の追加と，その問題点を説明する．

2.1 例: FTP クライアントプログラム

オブジェクト指向プログラミング (OOP) による FTP クライアントプログラムでは，ネットワークの接続を表現する BasicConnection のようなインターフェースが図 1 のように定義される．FTP プロトコルによる接続を表現する実装クラスとして，FTPConnection が定義される．クライアントはサポートするプロトコルの種類ごとに WebDAV プロトコルであれば WebDAVConnection や SMB プロトコルであれば SMBConnection など複数の BasicConnection の実装クラスを持つ．

BasicConnection で定義されるメソッドの実装はプロトコルごとに異なり，FTPConnection の定義は図 2 のようになる．図 2 の login の実装は，send,

```
public interface BasicConnection {
  boolean login(String name, String pass);
  boolean download(String file);
  .../*他のメソッド宣言が続く*/
}
```

図 1 BasicConnection の定義

```
public class FTPConnection
             implements BasicConnection {
  public boolean login(String name,
                       String pass) {
    send("USER " + name);
    if (!userOk()) return false;
    send("PASS " + pass);
    return passOk();
  }
  private void send(String data) {...}
  private boolean userOk() {...}
  private boolean passOk() {...}
  .../*他のメソッド実装やフィールド定義が続く*/
}
```

図 2 FTPConnection の login メソッドの定義

userOk 及び passOk の 3 個の private メソッドの呼出しから構成される．send は接続先にデータを送信するメソッドで，最初にユーザー名を送信する．その後 userOk を呼出し，ユーザー名が接続先に正しく受付けられたかを確認する．正しく受け付けられると，次に send によりパスワードを送信し，最後に passOk によるパスワード受付の確認が続き，結果を返す．

2.2 AspectJ による通知の実現

BasicConnection の各操作に対して，通知を行う機構を考える．例えば GUI フロントエンドを備えた FTP クライアントは，BasicConnection のログインやダウンロードなどの操作に対してユーザーに結果を通知する必要がある．その際，通知は図 3 で示す ConnectionListener インターフェースのメソッド呼出しとし，通知を受けるオブジェクトはこのインターフェースを実装しているものとする．ここでは簡単のために，BasicConnection の実装クラスの login の実行が，1) ログインに成功した場合 2) 不正なユーザー名を入力し失敗した場合，3) 不正なパスワードを入力し失敗した場合，に，通知の機構により ConnectionListener の対応するメソッド呼出しが通知されるものとする．

```
public interface ConnectionListener {
  void loginOk();
  void loginFailUser();
  void loginFailPass();
  .../*他の通知のメソッド宣言が続く*/
}
```

図 3 ConnectionListener の定義

通知の機構は AspectJ のような AOP 言語で実現することが妥当である．AspectJ を使うと ConnectionListener に関する通知の機構は，図 4 の ConnectionUpdate アスペクトのような，1 つのモジュールとして定義できるからである．その結果，保守性や可読性が高くなる．一方，Java のような OOP 言語での通知の機構の実現は，BasicConnection の各実装クラスに通知の呼出しを横断的に追加することになり，うまくモジュール化できない．

ConnectionUpdate は，login メソッドの実行中に呼出されたメソッドの結果を passSent を用いて記憶しておくことで，3 つの異なる実行履歴を識別する．ConnectionUpdate アスペクトには，login メソッドの呼出しにマッチする login ポイントカットが定義され，2 つのアドバイスで参照される．最初の after アドバイスは，login の呼出し中に passOk の呼出しが発生した場合に実行され，ユーザー受付後にパスワードが送信されたことを示す passSent フラグを true にする．aspect の修飾子として privileged を設定することで，private である passOk の呼出しを参照することが可能となる．2 番目の after アドバイスは login 終了後に実行され，戻り値と passSent の値によって，listener 変数が参照する ConnectionListener に対して適切な通知の呼出しを行う．

```
privileged aspect ConnectionUpdate {
  pointcut login():
    call(boolean login(String, String));
  boolean passSent = false;
  after(): cflow(login())
    && call(private boolean passOk()) {
    passSent = true;
  }
  after() returning(boolean result): login() {
    if (result)             listener.loginOk();
    else if (!passSent) listener.loginFailUser();
    else                    listener.loginFailPass();
  }
  .../*他の通知やリスナの管理のコードが続く*/
}
```

図 4　AspectJ による ConnectionUpdate の定義

2.3　ポイントカット記述の非一貫性の問題

ポイントカット記述の非一貫性の問題の例として，前節で説明した ConnectionUpdate の定義後に，FTPConnection クラスの login メソッド実装が改変された場合を考える．passOk と userOk が 1 つのメソッド check に統合され置き換えられたとすると，図 4 の最初のアドバイスは既に存在しない passOk の呼出しにマッチするポイントカット記述を含むため実行されず，適切な通知が呼出されなくなる．

名前に基づいたポイントカットや振舞いの性質に基づいたポイントカット記述は，ジョインポイントとなるメソッド†3の実装に強く結びつくことで一貫性を失い易くなる．例えば図 4 の制御の流れにマッチする cflow ポイントカットの利用は，login の実行中に passOk が呼出される，という実装の前提に基づいているため，その前提が前述の例のように変更されるとポイントカット記述は一貫しなくなる．

実装とポイントカット記述が強く結びついている場合，ポイントカットが一貫しているかを確認する作業は困難となる．例えば，複数人の開発プロジェクトとして ConnectionUpdate と FTPConnection の開発者が異なる場合に，ConnectionUpdate の開発者が login の内部実装の変更を認識することは困難である．

†3　ジョインポイントとなるメソッドとは，そのメソッドの実行がジョインポイントとなるメソッドを意味する．

2.4　ジョインポイントの実行履歴に関する問題

既存の AspectJ ではジョインポイントの実行履歴によって異なるポイントカットを定義できないため，アスペクトは複雑な定義になる．図 4 では passSent の情報を保つために，1 つの login の呼出しに関連する複数のアドバイスを定義している．

実行履歴をジョインポイントのマッチに利用しない解決手法として，アスペクトのためにメソッド実装を改変し，アスペクトに対して適切なジョインポイントを提供する方法が考えられるが，別の問題が発生する．例えば，ConnectionUpdate のために login メソッドを場合分けごとに公開されたメソッドを定義に分割する改変が考えられる．このような改変の作業は開発者の負担となるだけでなく，メソッド分割などが頻繁に行われることで改変されたモジュールは複雑になり，可読性や保守性の低下に繋がる．

本質的な解決には，メソッドの実行履歴をジョインポイントのマッチに利用するポイントカットが必要だが，既存の AspectJ には欠けている．例えば，AspectJ では login の実行履歴中の特定の if 分岐を参照するポイントカットを記述することができない．

3　テストに基づいたポイントカット

本稿では前節で述べた問題を解決するための新たなポイントカット機構として，**テストに基づいたポイントカット**(*Test-based Pointcuts*) を提案する．

テストに基づいたポイントカットは，ジョインポイントとなるメソッドのテストで利用される変数定義を参照することで，テスト対象メソッドの意味を反映したポイントカット記述を可能にする．例えば login の通知では，認証失敗となるユーザー名の代表値 (テストのための値) を表す変数をポイントカット記述として指定することで，ユーザー名により認証が失敗するテストの login 呼出しを参照することができる．テストは対象となるメソッドのインターフェースを通して記述されるため，テストが成功している限り，ジョ

インポイントとなるテスト対象メソッドの実装と一貫したポイントカットの定義を，インターフェースの名前や振舞いの性質を参照するよりも容易に得ることができる．

テストに基づいたポイントカットは，テストごとに異なる実行履歴をジョインポイントのマッチに利用する．テストは対象となる 1 つのメソッドに対して複数定義される．例えば認証が成功もしくは失敗する場合のように，異なった状態や入力値の代表値による場合分けごとにテストが定義され，異なった実行履歴を得る．得られたテストの実行履歴を実際のメソッド呼出しジョインポイントの実行履歴と比較することでマッチを行う．

テストに基づいたポイントカットは以下の 2 つの部分から成り立つ．

1. ポイントカットによるテストの参照．
2. テスト実行と実際の実行のマッチ．

以降の 3.1 節，3.2 節で順に説明する．

3.1 ポイントカットによるテストの参照

テストに基づいたポイントカットでは，既存の単体テストの記述を一部拡張することで，単体テストから横断的関心事のためのジョインポイントの特定を可能とする．拡張は，既存の単体テストの枠組みではテストに基づいたポイントカットが必要とする構造が完全に得られないため，既存の単体テストの枠組みに特定の構造を与えるために導入される．拡張は言語拡張ではなく，特定のライブラリと記述の規約によって達成される．

以降ではまず 3.1.1 節でテストの枠組みの拡張について説明し，続く 3.1.2 節で拡張されたテストを横断的に参照するための原始ポイントカットの導入について説明する．

3.1.1 テストの枠組みの拡張

一般にソフトウェアのテストには様々な分類があるが，本稿で扱うテストは OOP における自動化された単体テストである．図 5 は FTPConnection クラスに対する拡張された単体テストの記述である．OOP における単体テストは，JUnit [4] などの枠組みを利用する．現在のテストに基づいたポイントカットは

```
public class TestFTPConnection
                    extends TestCase {
  FTPConnection c;
  Fixtures f = new Fixtures();
  protected void setUp() {
    f.validUser = "testuser";
    .../*他の変数の初期化が続く*/ }
  public void testLoginOk() {
    Phase.test();
    boolean r=c.login(f.validUser, f.validPass);
    Phase.cond(); assertTrue(r); }
  public void testLoginUserFail() {
    Phase.test();
    boolean r=c.login(f.invalidUser,f.validPass);
    Phase.cond(); assertFalse(r); }
  public void testLoginPassFail() {
    Phase.test();
    boolean r=c.login(f.validUser,f.invalidPass);
    Phase.cond(); assertFalse(r); }
  .../*他のメソッドのテストの定義が続く*/
}
```

図 5 FTPConnection クラスのテストを定義する TestFTPConnection

JUnit(バージョン 3) を前提としている．JUnit ではクラスとしてモジュール化されたテストケースを定義することで単体テストを実施する．通常，テストケースのクラスには，図 5 中の testLoginOk などのように，test で始まる名前を持つメソッドとして，テストを定義し，1 つのテスト対象メソッドに対して複数のテストが定義される．以降ではテストケースのクラス定義をテストケース，テストケース中のテスト用のメソッド定義をテストと呼び，テストには 1 つのテスト対象メソッドが対応するとする．テストはテスト対象に与える入力 (validUser などの値)，対象のメソッド呼出し (login の呼出し)，そして事後条件を確認するコード (assertFalse の呼出し) を含む．

図 5 のテストは，テストに基づいたポイントカットのために拡張した記述を含んでいる．具体的には斜体で表されたコードで，テスト用のデータ構造体の利用

```
public class Fixtures {
    public String validUser, validPass,
                  invalidUser, invalidPass;
    .../*他のフィールドの定義が続く*/
}
```

図 6　TestFTPConnection のための Fixtures の定義

(Fixtures†4の validUser や invalidPass など) と，テスト対象メソッドを特定するための Phase の呼出しである．

- *Fixtures* はテストで利用される値を保持するフィールドの定義のみで構成され (図 6 参照)，複数のテストケースでテストを共通の意味ごとに分類するために利用する．
- *Phase* は，test() から cond() の呼出し間に呼出されるメソッドがテスト対象であることを明示するために利用する．

3.1.2　テストを参照する原始ポイントカットの導入

テストに基づいたポイントカットでは，テストで利用される値を参照する **testget**(*Type.field*) がポイントカットとして提供される．開発者はあらかじめテスト設計時に決めた変数の利用規約とその意味に従い，テストで使われるクラスの変数を testget により参照することで，関連するテストケースを横断的に参照することができる．

図 5 のテストケースをテストに基づいたポイントカットによって参照する ConnectionUpdate の定義が，図 7 である．例えば図 7 の loginOk ポイントカットは 2 つの testget ポイントカットを組み合わせ，validUser と validPass 変数を同時に利用するテストとして図 5 の testLoginOk を参照する．loginOk ポイントカットは，validUser と validPass を同時に使うテストに対して，login が成功した場合であるという意味付けを利用しており，その結果 FTPConnection の login だけではなく他のクラスの validUser と validPass 両方を使うテストも自動的に参照することができる．

```
aspect ConnectionUpdate {
  pointcut loginOk():
      testget(Fixtures.validUser) &&
      testget(Fixtures.validPass);
  pointcut loginFailUser():
      testget(Fixtures.invalidUser);
  pointcut loginFailPass():
      testget(Fixtures.invalidPass);
  after():loginOk() { listener.loginOk(); }
  after():loginFailUser() {listener.loginFailUser();}
  after():loginFailPass() {listener.loginFailPass();}
    .../*アドバイス宣言などが続く*/
}
```

図 7　テストに基づいたポイントカットを利用した ConnectionUpdate の定義

testget ポイントカットはより一般的な原始ポイントカット **test**(*Pointcut*) の導入によって以下のように定義される．

$$\mathbf{testget}(Type.field) = \mathbf{test}(\mathbf{get}(* \ Type.field))$$

test ポイントカットは与えられた *Pointcut* にマッチするジョインポイントが存在するテストを選択する．例えば，以下のようにして，特定のテストメソッドを参照したり，例外を捕捉するテストを参照できる．

```
test(execution(void testLoginOk()))
test(handler(FTPException))
```

3.2　テスト実行と実際の実行のマッチ

この節ではテストに基づいたポイントカットが，どのような実行時のジョインポイントにマッチするかを説明する．

テストに基づいたポイントカットはテストが呼出すテスト対象メソッドの実行履歴と，ジョインポイントのメソッド呼出しから得られる実際の実行履歴のマッチによって，実際の実行時のジョインポイントとのマッチを定義する．ここで実際の実行とは，例えば FTPConnection を利用する FTPClient クラス中からの FTPConnection オブジェクトへの login の呼出

†4 JUnit において fixture とは複数のテストで共有されるテストデータと協調するオブジェクトの共通の集合を指す．

しのような，テスト以外での実際の呼出しによる実行をいう．login の実際の実行は，ユーザーの入力による認証の成功や失敗によって，複数の実行履歴が考えられる．例えばログインが成功した場合は図 5 のテストである testLoginOk と同様の実行履歴となる．その結果，図 7 の validUser と validPass の参照から選択される testLoginOk を参照した loginOk ポイントカットのみがマッチする．

現在のテストに基づいたポイントカットでは，一連のジョインポイントに対応するコード上の箇所 (ジョインポイントシャドウ) の集合を実行履歴とする．これらの一連のジョインポイントは，テスト対象メソッドの呼出しジョインポイントの制御の流れの中で発生する．

テストの実行履歴中の全てのジョインポイントシャドウが実際の実行履歴に含まれる場合にマッチすると定義する．実行履歴の定義とそのマッチのためのアルゴリズムによって，テストに基づいたポイントカットのマッチの意味が決定するが，我々は実装の容易さから，ジョインポイントシャドウからなる実行履歴の定義と集合の包含関係を利用するアルゴリズムを選択した．ループや再帰呼出し中の重複したジョインポイントは実行履歴中では単一のジョインポイントシャドウとなるので，結果として繰り返し回数などはマッチの際に考慮されない．

実行履歴に含まれるジョインポイントシャドウは，if 文や while 文などの制御構造による分岐と，メソッド呼出しによるディスパッチ先のメソッド実行となる．既存の AspectJ のポイントカットは分岐をジョインポイントとして扱うことができないが，テストに基づいたポイントカットでは，新たに分岐をジョインポイントとして導入する．分岐先ごとに異なる分岐ジョインポイントが定義され，例えば if 文による分岐であれば，成功した場合と失敗した場合の 2 種類の分岐ジョインポイントシャドウが存在する．

テストに基づいたポイントカットのマッチに不要な履歴中のジョインポイントシャドウを除くため，test ポイントカットに対してフィルタリングのためのポイントカット記述を与えることができる．**test**($Pointcut_1, Pointcut_2$) として $Pointcut_1$ によりテストの選択を行い，$Pointcut_2$ としてテストの実行履歴のフィルタリングを行うためのポイントカットを与える．テストの実行履歴には疑似オブジェクト [12] などに代表される，テストでのみ利用するオブジェクトに対するメソッド呼出しが含まれることがある．また，実行履歴から除外したい特定のライブラリの呼出しなどが考えられる．例えば，

test(∗, !**cflow**(**within**(MockObject+)))

のような記述は疑似オブジェクトを表す MockObject クラスの制御の流れの下での全ての実行をテストの実行履歴から除外する．また，

test(∗, !**within**(java. ∗ .∗) || !**within**(javax. ∗ .∗)))

のような記述は Java ランタイムライブラリのパッケージに含まれる全ての実行を除外する[†5]．

テストに基づいたポイントカットの制限として，ジョインポイント後に実行される after アドバイスのみを適用することができる．AspectJ のアドバイスはジョインポイントの実行前やジョインポイント自体を置き換えることが可能だが，テストに基づいたポイントカットがマッチに利用する実行履歴はジョインポイントの実行後に判明するためである．

4 予備実験

テストに基づいたポイントカットの有効性を確認するための予備実験として，図形編集アプリケーション JHotDraw [3] と，ネットワーククライアントアプリケーション j-ftp [2] への適用事例に関して述べる．

これらのアプリケーションは Java で記述されており，ソースコードから導きだされるいくつかの横断的関心事が存在する．それらの横断的関心事をモジュール化する，既存の AspectJ によるアスペクトを記述し，同じアスペクトをテストに基づいたポイントカットを使って，どの程度一貫性を持った定義として書き直せるかを調べた．いずれのアプリケーションにもテストケースは付属していないため，テストに基づいたポイントカットを利用するため，著者が必要に応じて

†5 実装では AspectJ の制限として Java ランタイムライブラリの実行や分岐を記録することができないため，暗黙のうちにこのような定義がなされているとする．

表 1　評価結果

aspect	AJ	TB
JHotDraw		
CmdCheckViewRef	1 (1)	1
SelectionChangedNotification	2 (2)	2
j-ftp		
ConnectionUpdate	16 (15)	8

テストケースを定義した．

表 1 は全体の評価結果を示している．各行は上から，

- JHotDraw において，制約チェックを行う CmdCheckViewRef アスペクト，
- JHotDraw において，図形選択の通知を行う SelectionChangedNotificcation アスペクト，
- j-ftp において，接続の更新に関する通知を行う ConnectionUpdate アスペクト

の結果を示している．

評価の項目は以下のことを表している．

AJ ：AspectJ によるポイントカット定義の個数．括弧内の数字はテストに基づいたポイントカットで書き換えることができた定義の個数．

TB ：テストに基づいたポイントカットを利用した場合のポイントカットの定義の個数．

ここで，定義の個数とは重複を避けるように定義された名前つきポイントカットの定義の個数を意味する．

結果として，AspectJ で記述された 3 つのアスペクトの中のほとんどのポイントカット定義を，テストに基づいたポイントカットで再定義することができた．例外的に，j-ftp における ConnectionUpdate では，テストに基づいたポイントカットで定義できないポイントカットが 1 個存在した．原因はメソッドの実行履歴中に繰り返し発生する特定のジョインポイントにマッチすることができないためである．この問題点については 4.2 節で詳しく述べる．

以降では各アプリケーションのアスペクトについて順に説明する．

4.1 JHotDraw

JHotDraw における比較では，JHotDraw(バージョン 6.0b1) に対して AOP の適用を試みた AJHotDraw プロジェクト [8] [1] による実装を比較対象として利用する．AJHotDraw による実装では現在 (バージョン 0.3)，3 種類の横断的関心事がアスペクトとして実装されている．1 つはオブジェクトの永続化に関わる関心事で，アスペクトでは型間宣言のみが利用されており，対象外とする．残り 2 つはメソッド実行時の制約のチェックと図形オブジェクト選択時の通知のための関心事である．

4.1.1 制約チェック: CmdCheckViewRef

AspectJ による制約のチェックのためのアスペクトは CmdCheckViewRef として定義され，JHotDraw におけるコマンドの実行を表す execute メソッド実行時に，正しくオブジェクトの属性が設定されているかを確認する．execute の参照に関しては，名称が変更されない限り一貫しているといえる．

テストに基づいたポイントカットを利用した場合は，抽象的なコマンドオブジェクトが実行されたことを確認するテストを参照することで，execute メソッドに対するコード上の依存がなくなった．

4.1.2 選択変更の通知: SelectionChangedNotification

AspectJ による図形の選択通知のための SelectionChangedNotification アスペクトでは，図形を表す Figure オブジェクトの選択の変更にマッチする 2 つのポイントカットが定義されている．1 つは withincode と call ポイントカットを組み合わせ，図形選択のメソッド中の Figure に対する invalidate メソッドの呼出しにマッチする定義で，参照した図形の選択メソッド内での if 分岐中の invalidate の呼出しを，選択されるオブジェクトの変更と見なしている．例えば無効な選択操作として，既に選択された図形の選択の操作では，選択済み図形の集合に操作対象の Figure オブジェクトが含まれるかどうかを確認する if 分岐により invalidate が呼出されない．

テストに基づいたポイントカットを利用した場合は，Fixtures に定義された変数 selectedFigure と unselectedFigure を利用する 6 個のテストを参照するポイントカットを定義した．それらの定義は図形の選択メソッドの実装に依存せず分岐をうまく区別することができた．無効な選択操作として，変数 obsoleteFigure

を利用した 4 個のテストを適切に除外できた．

4.2 j-ftp における通知: ConnectionUpdate

ネットワーククライアントアプリケーションである j-ftp に関する実験では，バージョン 1.49-pre2 の実装に含まれる 2 つのクラス間の通知の関心事を，アスペクトによりモジュール化した．j-ftp は 2 節で紹介した FTP クライアントの例と同様の設計に基づいており，接続を表す BasicConnection インターフェースの実装クラスにより複数のプロトコルをサポートする．実験ではそれらのうち，FTP プロトコルの接続を表現する FtpConnection クラスと，ローカルのファイル操作を接続として表現する FilesystemConnection クラスに注目した．これら 2 つのクラスを横断する通知を ConnectionUpdate アスペクトとして定義することを試みた．通知には接続の成功や失敗，リモートディレクトリの変更，アップロードやダウンロードの完了などが含まれる．テストに基づいたポイントカット記述のために，2 つの接続を表すクラスのテストを定義した．

AspectJ を利用した記述では，2 つの接続のクラス間でのインターフェースや実装の違いにより，異なるポイントカット定義が必要であった．16 個の定義のうち 10 個が FtpConnection に対する記述で，6 個が FilesystemConnection に対する記述である．

また，2 つの接続クラスの異なるメソッド実装に対して，10 個のポイントカット定義が深く結びつくこととなった．4 個のポイントカット定義は cflow や withincode を利用して特定のメソッドの特定の分岐を参照する記述である．6 個のポイントカット定義はメソッド中の分岐と同様の分岐を if ポイントカットに記述することで，結果的に分岐を再現している．

テストに基づいたポイントカットを利用することで，15 個のポイントカット定義をより少数の一貫性を持つ 8 個の定義に書き換えることができた．2 つの接続クラス間で共通の Fixtures クラスの変数を利用したテストを参照することで，2 個のポイントカット定義の共通化が可能となった．AspectJ による定義では複数のポイントカット定義で参照されるポイントカットの定義が 5 個あったが，テストに基づいたポイントカットを利用することで不要になった．

書き換えられた 8 個のポイントカット定義では一貫性が得られた．これらの定義は 2 つの接続クラスへの直接の依存がなく，全て testget ポイントカットの列挙により記述された．AspectJ の記述であったようなメソッド実装の分岐の参照や再現はなくなった．

ファイルを転送するメソッド実行中に発生するストリームの書き込みごとに通知を行うためのポイントカット定義 1 個が，テストに基づいたポイントカットで書き換えることができなかった．AspectJ による定義では，cflow と call を組み合わせ，ファイル転送のメソッド中でストリームの write メソッドの呼出しにマッチを行っていた．1 回のファイル転送で発生する複数の write の呼出しにより，複数回の通知が発生する．現在のテストに基づいたポイントカットではテストの参照によって，ファイル転送のメソッド呼出しにマッチすることはできても，その中のジョインポイントである write メソッドの呼出しごとにマッチを行うことはできない．この問題の解決は将来的な課題である．

5 実現方法

テストに基づいたポイントカットを既存の AspectJ コンパイラの拡張により実装する予定である．拡張可能な AspectJ コンパイラとして abc コンパイラ [6] を利用する．既存の AaspectJ コンパイラはポイントカットが静的にマッチするジョインポイントシャドウに，実行時のマッチのためのコードとアドバイス実行のコードを織込むことでコンパイルを行う．

本節ではテストに基づいたポイントカットのコンパイルの実現方法について簡単に説明する．

テストに基づいたポイントカットのコンパイルは大まかに以下の手順からなる．

1. テストの実行
2. test ポイントカットのマッチと実行履歴のフィルタリング
3. 実行履歴のジョインポイントに対するマッチとアドバイスの呼出しの織込み

最初に，コンパイラは入力された全てのテストを実行することで，アドバイスを織込むジョインポイント

```
aspect JUnit3TestMatch {
  Set tests = new HashSet();
 before(): Pointcut &&
   withincode(public void TestCase+.test*()) {
    tests.add(thisEnclosingJoinPointStaticPart);
  }
}
```

図 8　テストを選択するために生成されるアスペクト

シャドウの特定と，マッチのためのテストの実行履歴を得る．実行履歴を記録するために，全てのテストとテストから呼出される可能性のあるメソッド中のジョインポイントシャドウに対して，実行を記録するコードを埋め込んだクラスを出力し，テストを実行する．

次に，test ポイントカットにパラメータとして与えられたポイントカットをテスト実行にマッチさせることで，テストを選択する．**test**(*Pointcut*) のテストの選択のためには，図 8 で示すアスペクトが生成されテストケースに織込まれる． このアスペクトはテストメソッドの実行中に *Pointcut* がマッチしたとき，thisEnclosingJoinPointStaticPart で取得できるテストメソッド実行のジョインポイントシャドウを，変数 tests に記録する．テストケースの実行後に変数 tests に格納されるテストメソッド実行のジョインポイントシャドウから選択されたテストが判明する．

test ポイントカットに 3.2 節で導入した右側のパラメータとして，テスト実行履歴のフィルタリングのためのポイントカットが渡された場合は，選択されたテストの実行履歴中で，マッチするジョインポイントシャドウだけにフィルタリングした結果の集合を最終的なテストの実行履歴とする．

最後に実行時のマッチのためのコードとアドバイス呼出しの織込みを行う．テストの実行履歴中のジョインポイントシャドウに対して，実際の実行履歴を得るために実行を記録するコードを追加する．選択されたテストのテスト対象メソッドの呼出しの直前には，実際の実行履歴の記録領域を確保するコードを追加する．メソッド呼出しの直後に，テストと実際の実行履歴のマッチとして，実行履歴のコードの箇所を全て通過したかを確認するコードとアドバイスの呼出しを追加する．

6　議論と関連研究

この節では現状のテストに基づいたポイントカットに関する課題を含めた議論と，関連する研究について述べる．

6.1　実行履歴の選択

3.2 節で述べたように，現在のテストに基づいたポイントカットは実行履歴をジョインポイントシャドウの集合として，集合の包含関係によりマッチを行うが，メソッド呼出しジョインポイントの順序などをマッチに利用する方法も考えられる．4 節の事例では if 分岐のマッチがほとんどで，実行回数や順序による複雑な履歴のマッチの必要性は確認できなかった．テストに基づいたポイントカットにとって，単純な分岐のマッチで充分であるかどうかの判断のためには更なる調査が必要である．

tracematches [5] は実行履歴に基づいたポイントカットとアドバイスを定義できる AspectJ の拡張言語機構である．tracematches ではジョインポイントの順序や回数によって履歴を定義することができるが，履歴中のジョインポイントは，名前に基づいたポイントカットなどの，一貫性に問題のある記述で参照する必要がある．また，分岐を履歴中のジョインポイントとして指定することはできない．

他に，異なる値の流れに基づいてジョインポイントの区別を行う場合があると予測される．テストごとに入力値などの値の分類ができれば，テストと同様の入力を受け取るジョインポイントに対するマッチが実現できると考えられるが，将来の課題とする．

6.2　開発手法

テストに基づいたポイントカットを利用するにはテストケースが必要になるため，開発手法によってはアスペクトとテスト対象のモジュール間の並行した開発が困難になる可能性がある．多くの場合，単体テストはテスト対象モジュールの実装後に記述されるため，テスト対象モジュールとアスペクトを書く開発者が異なる場合は，アスペクトを書く開発者は単体テストの

終了を待たなければならない．

モジュールを実装する前にテストを書く開発手法として，**テスト駆動開発**(*Test-Driven Development*) [7] が提唱されている．テスト駆動開発では常にテスト対象モジュールの単体テストが開発の早い段階で存在するため，テストに基づいたポイントカットを利用したアスペクトも早い段階から定義できる可能性がある．従ってテスト駆動開発はテストに基づいたポイントカットに適していると考えられるが，現状でAOP を取り入れたテスト駆動開発手法は一般的に定義されておらず，アスペクトの単体テストはどう書かれるべきかなどを明らかにする必要がある．

6.3 XPI

Griswold ら [9] は *XPI* という AOP のための新しい種類のインターフェースを導入することで，ポイントカット記述の非一貫性の問題の解決を試みているが，アスペクトの存在を想定する，安定した XPI の定義があらかじめ必要となる．テストに基づいたポイントカットでは，新たな一貫性維持の制約を加える XPI とは異なり，テストの工程における一貫性の維持を利用する．

6.4 ビューに基づいたポイントカット

Kellens ら [10] は一貫したポイントカット記述を得るために，プログラムの実装ではなく概念的なモデルを参照することでポイントカット参照を行う**ビューに基づいたポイントカット**(*View-based Pointcuts*) を提案している．ビューに基づいたポイントカットでは仕様記述と同等の概念モデルを参照することができるが，実行履歴によるジョインポイントの区別ができない．また，概念モデルを表すための内包的ビュー (Intensional views) の定義を行うために，ツールを利用して手動で対象モジュールを選定する必要があるため，実装との一貫性を保つための労力が追加的に必要となる．

7 おわりに

本稿では，AspectJ のためにポイントカット記述の一貫性を容易に得ることが可能で，メソッドの実行履歴ごとに異なったポイントカット定義が可能な言語機構として，テストに基づいたポイントカットを提案した．既存の AspectJ ではポイントカット記述の一貫性維持のために，多くの労力が必要であることが多いが，テストに基づいたポイントカットでは，一定の規約に基づいて記述された単体テストケースを参照することで，一貫性の維持の作業を，正しくテストを定義する作業に置き換えることができる．また，実行履歴の場合分けを網羅したテストを参照することで，既存の AspectJ では区別できなかった同一のメソッドの異なる実行履歴となるジョインポイントに対するマッチを可能にした．

実際のアプリケーション中のアスペクト記述の比較実験から，AspectJ で記述された既存の多くのポイントカット定義が，テストに基づいたポイントカットによって一貫性のある定義に書き換えられることを明らかにした．

一方で実験から，実行履歴中の特定のジョインポイントに対して繰り返しマッチを行うような定義が現在のテストに基づいたポイントカットでは記述できないことが明らかになった．テスト実行履歴の制御の流れを参照するような記述が必要であり，解決は今後の課題とする．

その他の今後の主な課題は，テストに基づいたポイントカットの処理系の実装と，より実践的な事例への適用による有効性の確認などが挙げられる．また，現状ではテストに基づいたポイントカットでは実行履歴として制御の流れによる分岐のみを扱うが，得られる値による場合分けや実行履歴中の値の束縛の機構が必要になる場合があると予測され，それらを扱う機構を考える必要がある．

謝辞

本稿の執筆にあたり，有益な助言を下さった PPP グループの皆様，芝浦工業大学 古宮研究室の皆様に感謝します．

参 考 文 献

[1] AJHotDraw: `http://ajhotdraw.sourceforge.net/`.

[2] j-ftp: `http://j-ftp.sourceforge.net/`.
[3] JHotDraw: `http://www.jhotdraw.org/`.
[4] JUnit: `http://www.junit.org`.
[5] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J.: Adding trace matching with free variables to AspectJ, *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, ACM Press, 2005, pp. 345–364.
[6] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J.: abc: An extensible AspectJ compiler, *AOSD '05: Proceedings of the 4th international conference on Aspect-Oriented Software Development*, ACM Press, 2005, pp. 87–98.
[7] Beck, K.: *Test-Driven Development: By Example*, Addison Wesley Longman, 2002.
[8] Deursen, A. v., Marin, M., and Moonen, L.: AJHotDraw: A Showcase for Refactoring to Aspects, *Proceedings of the Workshop on Linking Aspects and Evolution (LATE05)*, 4th International Conference on Aspect-Oriented Programming, 2005.
[9] Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., and Rajan, H.: Modular Software Design with Crosscutting Interfaces, *IEEE Software*, Vol. 23,No. 1(2006), pp. 51–60.
[10] Kellens, A., Mems, K., Brichau, J., and Gybels, K.: Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts, *ECOOP '06: Proceedings of the 20th European Conference on Object-Oriented Programming*, 2006, pp. 501–525.
[11] 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, 2001, pp. 327–353.
[12] Mackinnon, T., Freeman, S., and Craig, P.: Endo-Testing: Unit Testing with Mock Objects: *Proceedings of eXtreme Programming and Flexible Processes in Software Engineering (XP2000)*, May 2000.
[13] Stoerzer, M. and Graf, J.: Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software, *21st IEEE International Conference on Software Maintenance (ICSM)*, 2005, pp. 653–656.