分散ソフトウェアのテストに適したアスペクト指向言語

西澤無我[†] 千葉滋[†] 立堀道昭[††]

本稿では，分散ソフトウェアのテストに適したアスペクト指向言語を提案する．分散ソフトウェア用のテストのいくつかは横断的関心事としてよく知られているが，これらのテストは AspectJ を利用しても，分散を意識させない独立したモジュールとして分離することができない．このため，そのようなテストを実装したプログラムの可読性・保守性は高くない．この問題を解決するため，我々は AspectJ を分散ソフトウェア用に拡張した DJcutter を提案する．DJcutter は複数のホストに分散した横断的関心事をモジュール化するのに適した言語機構を提供する．DJcutter の利用者はそのような横断的関心事を，単一ホスト上で動作するアスペクトであるかのように，分散を意識せずに記述することができる．本稿は，分散ソフトウェアのテストを例にこの点について説明する．

An AOP Language for Testing Distributed Software

Muga Nishizawa,[†] Shigeru Chiba[†] and Michiaki Tatsubori[††]

This paper presents an AOP language that is suitable for testing distributed software. Some of these tests are well known as typical crosscutting concerns. However, even if developers use AspectJ for implementing such a test, the implementation of the test can not be an independent module that does not include a distribution concern. Such an implementation shows low readability and maintainability. To address this problem, we propose DJcutter, which is an extension to AspectJ for distributed software. It provides a few new language constructs suitable for modularizing distributed crosscutting concerns. If developers use DJcutter, they can implement those concerns as simple aspects that does not include a distribution concern. This paper illustrates the descriptive power of DJcutter by illustrating a few examples of test code for distributed software.

1. はじめに

現在，Web アプリケーションに代表される分散ソフトウェアのテスティングの重要性・必要性が高まっている．特に，XP（Extreme Programming）[1] コミュニティが，複数の単体テスト☆を連続して自動で実行できるようにすること，すなわちテストの自動化，を提唱したことにより，多くの開発者が開発プロセスの中でテスティングを実施するようになった．テストの自動化により，テストにかかるコストが削減でき，リファクタリングなどのソフトウェアの修正・変更時にも簡単にテストの自動実行を反復することができる．その結果として，バグが早期に発見されやすくなり，ソフトウェアの開発効率が向上する．これまで，テストの自動化のため，JUnit [2] や Cactus [3] など，シンプルなテストのフレームワークが開発されてきた．

ソフトウェアのテストのいくつかが，横断的関心事であり，この関心事がテストの自動化を妨げることはよく知られている．横断的関心事とは，ロギングやセキュリティなどのモジュール間を横断する処理のことである．その実装は，いくつかのモジュール間にちらばってしまうため，保守に向いていない．ソフトウェアのテストの中では，プログラムの内部構造を確認するテストが横断的関心事にあたる．プログラムの外部仕様のテストと異なり，プログラムの内部構造をチェックするには，プログラムを構成するモジュールのいくつかにチェックのための処理を手動で挿入しなければならない．また，テストの終了時には，逆に挿入したコードを削除する操作も必要になる．このように，毎回のテストの実行時に，テスト対象プログラムを手動で編集するのは手間のかかる作業であり，テストの反

† 東京工業大学大学院情報理工学研究科
Tokyo Institute of Technology, Graduate School of Information Science and Engineering
†† IBM 東京基礎研究所
Tokyo IBM Research Laboratory

☆ 単体テストとは，狭義には，ソフトウェアを構築するモジュールそれぞれの動作確認チェックを意味する．本稿では，いくつかのモジュールを組み合わせて構築されたコンポーネントやサービスの end-to-end テストも単体テストと呼んでいる．

復を妨げてしまう．

アスペクト指向（AOP）[4)~7)] 技術の登場で，開発者は横断的なテストプログラムをテスト対象のプログラムに入り込まないように分離して記述できるようになり，テストの自動実行の反復は実施しやすくなった．しかし，汎用的な AOP 言語，特に AspectJ[8)]，を使っても，分散ソフトウェア用のテストコードは，分散を意識せずにうまくモジュール化しきれない．一般に分散ソフトウェアのプログラムであっても，その大半は分散を意識せずに書けるべきであり，分散に関する記述は，残りのプログラムから分離されているのがよい．

これらの問題を解決するため，我々は汎用的な AOP 言語である AspectJ を分散ソフトウェア用に拡張した *DJcutter* を提案する．DJcutter では，複数のホストに分散した横断的関心事をモジュール化するのに適した言語機構を備えている．特に，DJcutter が提供する**遠隔ポイントカット**は，遠隔ホスト上のジョインポイントを指定することができる．これにより，利用者は複数のホスト間にまたがる横断的関心事を，分散を意識せずに単一ホスト上で動作する簡潔なアスペクトとしてモジュール化することができる．また，DJcutter は，遠隔ホスト上のクラスのフィールドやメソッドを追加する**遠隔インタータイプ宣言**も提供する．これらの機能は，分散ソフトウェア用のテストプログラムを作成するのに有用である．

以下に本稿の残りの構成について説明する．まず，2 章で AspectJ で記述した分散ソフトウェア用のテストコードの例を示し，本研究の動機となる問題点を分析する．次に 3 章で，我々が提案する DJcutter 言語，特に DJcutter に特有である遠隔ポイントカットや遠隔インタータイプ宣言について説明する．4 章で，DJcutter を利用したテストプログラムの例を示し，DJcutter の有用性を述べる．DJcutter を使ったパフォーマンスに関する実験結果は 5 章に示す．6 章，7 章で，関連研究と議論を記述する．

2. 分散を意識したテストコード

AOP の利用により，横断的なテストコードをテスト対象プログラムから分離して記述できるようになり，テストのためだけにテスト対象プログラムを修正・変更する必要がなくなった．しかし，分散ソフトウェア用のテストコードは汎用的な AOP 言語，特に AspectJ，を利用してうまくモジュール化しきれない．この章では，まず分散ソフトウェアの単体テストの例を示す．次にこのテストコードの問題とその原因を考察する．

2.1 認証サービス・システムのための単体テスト

例として，我々は複数のホストに分散した認証サービス・システムのためのテストコードを示す．このサービス・システムの実装は，2 つのコンポーネントから構成されている．片方は，ホスト W 上で動作するフロントエンド AuthServer，もう片方は，ホスト D 上で動作するデータベース・サーバ DbServer である．これは企業向け Web アプリケーションの典型的な構成である．もしホスト T 上のクライアントがこのデータベースに新しいユーザを登録したければ，Java RMI を利用してフロントエンド AuthServer 上の registerUser() を遠隔呼び出しする必要がある．そして，その registerUser メソッドは，内部でデータベース・サーバの addUser() を遠隔呼び出しする．AddUser() が最終的にデータベースへアクセスし，ユーザリストを更新する．

RegisterUser() の動作テストの 1 つとして，registerUser() を遠隔メソッド呼び出ししたときに，データベース・サーバの addUser() が正しく実行されているか否かを確認するというものが考えられる．このようなテストは横断的関心事である．しかし，もしこの認証サービス・システムの 2 つのコンポーネントが，ともに同一のホスト上で動作するものであれば，そのテストコードは簡潔に記述することができる．AspectJ を利用して記述したそのテストコードを以下に記載する：☆

```
 1: aspect AuthServerTest extends TestCase {
 2:   boolean wasAddUserCalled;
 3:   void testRegisterUser() {
 4:     wasAddUserCalled = false;
 5:     String userId = "muga";
 6:     String password = "xxx";
 7:     AuthServer auth = new AuthServer();
 8:     auth.registerUser(userId, password);
 9:     assertTrue(wasAddUserCalled);
10:   }
11:   before(): execution(void
12:     DbServer.addUser(String, String)) {
13:     wasAddUserCalled = true;
14:   }
15: }
```

このプログラムはスペースの関係で，完全なもので

☆ そもそも，AspectJ を使用せずに，Java でこのテストコードを記述すると，テスト作成者は DbServer の addUser メソッド内に wasAddUserCalled フィールドの値を変更するためコードを挿入しなくてはならない．これはテストのためだけに，テスト対象プログラムを修正・変更する必要があるため，テストの効率を低下させてしまう．

はないが，テストコードの全体像を理解するには十分であろう．このテストコードの主要な部分は，testRegisterUser() である（3 行目から 10 行目）．それは，まずフロントエンド AuthServer の registerUser() を呼び出しており，次に wasAddUserCalled フィールドの値が true であるか否かを確認している．このフィールドの値は，addUser() が実行されたときに呼び出される before アドバイスによって true に変更される（11 行目から 14 行目）．この before アドバイス 11 行目から 12 行目は addUser メソッドの実行というジョインポイントを指定するポイントカット記述である．AspectJ では，指定したジョインポイントの直前，直後あるいはそれ自身と置き換えて，アドバイスを呼び出すことができる．この例では，addUser メソッド実行の直前（11 行目）で，wasAddUserCalled フィールドを true に変更するアドバイスコードが呼び出される（13 行目）．

2.2 AspectJ で記述されたテストコード

もしこの認証サービス・システムがもとの設定どおりに分散していたとしたら，AspectJ によるテストコードは先のプログラムよりも複雑になる．以下に分散版のテスト・プログラムを記載する☆．

```
 1: // on host T
 2: class AuthServerTest extends TestCase {
 3:   boolean wasAddUserCalled;
 4:   void testRegisterUser() {
 5:     Naming.rebind("test",
 6:                   new RecieverImpl());
 7:     wasAddUserCalled = false;
 8:     String userId = "muga";
 9:     String  password = "xxx";
10:     AuthServer auth = (AuthServer)
11:       Naming.lookup("auth");
12:     auth.registerUser(userId, password);
13:     assertTrue(wasAddUserCalled);
14:   }
15:   class ReceiverImpl
16:       extends UnicastRemoteObject
17:       implements NotificationReceiver {
18:     void confirmCall() {
19:       wasAddUserCalled = true;
20:     }
21:   }
22: }
23:
24: interface NotificationReceiver
```

☆ この分散版のテスト・プログラムも完全なプログラムではない．public のようなアクセス修飾子やコンストラクタを省略している．

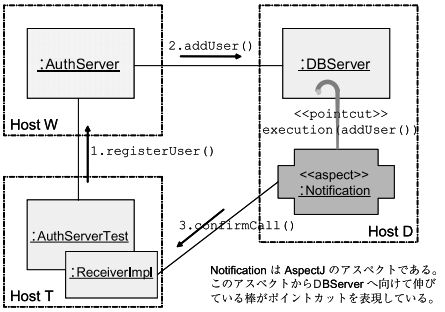

図 1 AspectJ で記述したテストコード
Fig. 1 The testing code in AspectJ.

```
25: { // on both hosts
26:   void confirmCall();
27: }
28:
29: aspect Notification { // on host D
30:   before(): execution(void
31:     DbServer.addUser(String, String)) {
32:       NotificationReceiver test
33:         = (NotificationReceiver)
34:             Naming.lookup("test");
35:       test.confirmCall();
36:   }
37: }
```

このテストコードは，3 つのサブ・コンポーネントから構成される：AuthServerTest，ReceiverImpl，そして Notification（**図 1**）．全体的なテストのアルゴリズムは先と同様である．しかし，AuthServerTest と ReceiverImpl オブジェクトはテストを実行するホスト T 上で動作する一方，Notification アスペクトは DbServer が動作しているホスト D 上で動作する．ホスト T は W や D と異なるホストである．

ホスト T 上の testRegisterUser メソッド（4 行目から 14 行目）は，まず W 上の registerUser() を遠隔呼び出しし，次に wasAddUserCalled フィールドの値が true であるか否かを確認する．このフィールドは ReceiverImpl 内で宣言されている confirmCall() が実行されることによって true に変更される．confirmCall() は，ホスト D 上で動作する Notification アスペクトの before アドバイス（30 行目から 36 行目）によって遠隔メソッド呼び出しされる．AuthServerTest クラスは UnicastRemoteObject クラスと TestCase クラスを同時に継承することはできないため，confirmCall() は AuthServerTest 内で定義できない．これは，遠隔からアクセス可能なクラスは UnicastRemoteObject クラスを継承しなければならないというのは Java RMI の

仕様である．

2.3 考　　察

上記 2.1 節と 2.2 節のテストコードを比べると，2.2 節の分散版のテストコードは，分散処理を強く意識したコードになっていることが分かる．2 つのテストコードを比べると，コンポーネントの内部構成が大きく異なっている．2.1 節のテストは，1 つのコンポーネントにまとめて設計されているが，2.2 節の分散版のテストでは，3 つのサブ・コンポーネントに分割された設計になってしまっている．

このうち，T 上のテストが AuthServerTest と ReceiverImpl との 2 つに分かれているのは Java 言語が多重継承をサポートしていないためである．しかし，仮に Java が多重継承をサポートしていたとしても，分散版のテストコードはホスト T 用のコードとホスト D 用のコードの 2 つに分かれてしまう．

2.2 節の分散版のテストが少なくとも 2 つのサブ・コンポーネントに分割された設計になってしまう原因は，分散化支援フレームワーク Java RMI の仕様にある．Java RMI をはじめ，現在広く利用されている分散化支援フレームワークの多くは，オブジェクトを単位として分散配置するという仕様である．この仕様では，1 つのメソッドは最初から最後まで同一のホスト上で，実行できなければならない．この制限は，通常のオブジェクト指向の範疇では，設計を工夫することによって回避されてきた．

しかし，アスペクト指向言語を利用して，分散ソフトウェアの横断的関心事のモジュール化を試みると，テストの例で見たように，この制限の問題が顕在化する．分散ソフトウェアのテストは，本質的に複数のホスト上のオブジェクトの間にまたがる横断的関心事である．既存のアスペクト指向言語では，それぞれのホスト上のテストに関わる関心事を，テスト対象プログラムから分離して記述することは可能であるが，分離されたテストの断片が複数のホスト上にちらばってしまう．これらのテストの断片を集めて 1 つのコンポーネントを作ろうとしても，そのテストは最初から最後まで同一ホスト上で実行可能なものではない．そのため，分散化支援フレームワークによる制限により，最初から最後まで同一ホスト上で実行可能な複数のサブ・コンポーネントに，コンポーネント全体を分割しなければならない．これはモジュール化にあたって分散を意識しなければならないことを意味する．

一般に，アスペクト指向の観点からも，分散プログラミングの観点からも，プログラムの大半は分散を意識せずに書けるべきである．

3. DJcutter

先の章の問題を解決するために，我々は分散ソフトウェア用に AspectJ を拡張した，*DJcutter* という言語を提案する．DJcutter を使用することで，開発者は異なるホスト上の複数のコンポーネント間にまたがる関心事を，分散を意識しないモジュールとして記述することができる．つまり，Java RMI などの明示的なネットワーク処理を含まないアスペクトとして実装することができる．以下では，特に DJcutter と汎用的な AOP 言語である AspectJ との相違点に絞って説明していく．

3.1 遠隔ポイントカット

AspectJ と DJcutter の最も重要な違いは，DJcutter が**遠隔ポイントカット**を提供していることである．遠隔ポイントカットは，遠隔ホスト上で動作しているプログラム内のジョインポイントを指定するための機能である．これを用いると，プログラムの実行が遠隔ポイントカットとして指定されたジョインポイントに到達したとき，ジョインポイントが存在するホストと異なるホスト上に存在するアドバイスを実行することができる．AspectJ では，アドバイスはそのジョインポイントと同一ホストになければならない．

遠隔ポイントカットは，アドバイスと同一ホスト上のジョインポイントを指定する方法と同じ仕様で，他のホスト上のジョインポイントを指定することができる．これは，分散オブジェクトに対して，手元のホスト上のオブジェクトと同様の仕様でメソッドを呼び出せるようにするフレームワーク Java RMI の機能のポイントカット版ともいえる．

遠隔ポイントカットを使うと，分散化した横断的関心事をネットワーク処理を意識しない非分散のモジュールとして，簡潔に実装することができる．以下に DJcutter 言語で記述されたアスペクトの例を示す．DJcutter は AspectJ を分散拡張したものであるため，その文法は AspectJ とほぼ同等である．

```
aspect LoggingAspect {
  pointcut setter(int x):
    args(x) && call(void Point.setX(int));
  before(int x): setter(x) {
    System.out.println("set x: " + x);
  }
}
```

このアスペクトは，任意のホスト上で setX() が呼び出されたときに，メッセージを出力する．メッセージは，どのホスト上で setX メソッドが呼び出されて

表 1 DJcutter のポイントカット指定子
Table 1 The pointcut designators of DJcutter.

指定子	ジョイントポイント
within(*TypePattern*)	*TypePattern* にマッチした型の宣言内に含まれるジョイントポイント
target(*Type or Id*)	*Type* 型もしくは *Id* の型にマッチしたインスタンスがターゲットオブジェクトであるジョイントポイント
this(*Type or Id*)	*Type* 型もしくは *Id* の型にマッチしたインスタンスによって実行されているジョイントポイント
args(*Type or Id, ...*)	引数が *Types* 型もしくは *Ids* の型にマッチしたジョイントポイント
call(*Signature*)	*Signature* にマッチしたメソッドもしくはコンストラクタ呼び出し
execution(*Signature*)	*Signature* にマッチしたメソッドもしくはコンストラクタ実行
set(*FieldPattern*)	指定された名前，型にマッチしたフィールドの値の読み出し
get(*FieldPattern*)	指定された名前，型にマッチしたフィールドの値の書き込み
handler(*TypePattern*)	*TypePattern* 型にマッチした例外のスロー
cflow(*Pointcut*)	*Pointcut* によって指定したジョイントポイントの始まりと終わりとの間に含まれるすべてのジョイントポイント
hosts(*Id, ...*)	*Id* として名付けられた DJcutter の実行時システム上で実行されるジョイントポイント

も，同一のホスト上に出力される．

LoggingAspect 内で定義されている setter ポイントカット：

```
call(void Point.setX(int))
```

は，AspectJ と同様に Point クラスの setX() の呼び出し式に対応するジョイントポイントを指定している．しかし，AspectJ のポイントカットとは異なり，その呼び出し式がどのホスト上で実行されても，このポイントカットにマッチする．

アスペクト・サーバ

アドバイスのコード：

```
System.out.println("set x: " + x);
```

は，setX メソッド呼び出しの直前に実行されるが，setX() を呼び出したスレッドが動作しているホストとは異なるホストで実行される．プログラムの実行がそのジョインポイントに到達すると，その実行スレッドは異なるホストで動作している**アスペクト・サーバ**へ，ネットワーク越しに暗黙的にメッセージを送信する☆．このメッセージを受信すると，アスペクト・サーバは対応するのアドバイスコードを実行する．メッセージを送信したプログラムの実行スレッドは，アスペクト・サーバがアドバイスコードを実行し終えるまでブロックする．

すべてのアドバイスコードは特定のホスト上で動作する単一のアスペクト・サーバによって実行される．この特徴を利用すれば，ユーザはホスト間で共有するデータの管理を，アスペクトのフィールドにデータを格納することで容易に実現できる．アスペクトのフィールドはアドバイスコードからアクセス可能である．一方，AspectJ のアドバイスコードはつねに呼び出し元のスレッドが動作しているホストと同じホスト上で実行される．それゆえ，他のホスト上で動作しているアドバイスコードとデータの共有を行いたければ，明示的なネットワーク通信をアドバイスコード内に記述しなければならない．

遠隔インタータイプ宣言

次に，我々が行った DJcutter のインタータイプ宣言（イントロダクション）の拡張について説明する．AspectJ では，必要に応じて，任意のクラスのメソッドやフィールドをアスペクト内で追加的に宣言することができる．DJcutter では，クラスのメソッドやフィールドの宣言を，異なるホスト上のアスペクト内で宣言できる．

3.2 ポイントカット指定子

DJcutter で実装されているポイントカット指定子を**表 1** に示す．多くのポイントカット指定子は AspectJ の言語仕様から受け継いでいる☆☆．

DJcutter 特有のポイントカット指定子として hosts がある．これは選択したホスト上のジョインポイントを指定する．DJcutter の各ポイントカット指定子は，デフォルトで，すべてのホスト上のジョインポイントを抽出するが，このポイントカット指定子は特定のホスト上のジョインポイントを抽出するために利用される．

たとえば，DJcutter の利用者は hosts ポイントカット指定子を，次のように利用することができる：

☆ 技術的には，アスペクト・サーバは同じホスト上で動作していてもよい．

☆☆ この表に掲載されているポイントカット指定子は，AspectJ が提供している指定子のサブセットである．現在の DJcutter の実装では AspectJ のポイントカット指定子すべてをサポートしているわけではない．

```
pointcut sample():
   call(void Point.setX(int))
   && hosts(hostId1, hostId2)
```

このポイントカットは，hostId1 と hostId2 という名前で指定されたホスト上の Point オブジェクトの setX() 呼び出しをジョインポイントとして抽出する．HostId1 と hostId2 は，DJcutter の実行時システムを各ホスト上で起動する際に，利用者が実行時パラメータとして与えることのできる実行時システムの名前である．異なるホスト上で動作する DJcutter の実行時システムには，異なる名前を与えなければならない．これらの実行時パラメータを活用することによって，開発者はソースコードに固定されたホスト名を記述することを避けられる．

DJcutter は，分散処理の実行中の制御フロー内のジョインポイントを抽出するため，cflow ポイントカット指定子を拡張している．AspectJ の cfow は，ポイントカット指定子によって指定されたメソッドの実行の始まりと終わりの間に存在するジョインポイントすべてを抽出する．つまり，スレッドが cflow によって指定されたメソッドを実行している間に，実行したジョインポイントが抽出の対象となる．AspectJ の cflow では，分散したホスト間にまたがったプログラムの制御フローを追跡することができない．

DJcutter の cflow では，カスタムソケットクラスを利用した実行時ライブラリを使用することで，ネットワーク越しに移ってしまったプログラムの制御フローを追跡し，そのフロー内のジョインポイントを指定することが可能である．たとえば，もしネットワーク通信のために Java RMI が使われていたとすると，以下のように通信が DJcutter の提供するカスタムソケットを使うように指定すれば，他のホスト上に移ってしまったプログラムの制御フローを追跡することが可能になる：

```
PointImpl p0 = new PointImpl();
Point p = (Point)
   UnicastRemoteObject.exportObject(
         p0, 40000,
         new DJCClientSocketFactory(),
         new DJCServerSocketFactory());
```

このプログラムは，PointImpl オブジェクトにネットワーク越しからアクセスするときに，DJcutter がカスタムソケットを使うようにする．DJCClientSocketFactory と DJCServerSocketFactory クラスはカスタムソケットを生成するために DJcutter が提供している実行時ライブラリである．また，DJcutter はカスタムソケット生成のためのプログラムを以下のように，シンプルに記述できるライブラリも提供している．

```
PointImpl p0 = new PointImpl();
Point p = (Point)
   DJcutter.exportObject(p0, 40000);
```

3.3 アスペクト・メソッドへのアクセス

DJcutter のアスペクトは，通常の Java クラスを実行するスレッドとは異なるアスペクト・サーバ上で実行されるが，Java クラスからアスペクト内で宣言されたメソッドを呼び出すときに，それを意識する必要はない．アスペクト内で宣言されたメソッドは，インタフェースを通じて呼び出される．したがって，アスペクトは必要なインタフェースを実装しなければならない．

たとえば，LoggingAspect 内の displayLog メソッドを通常の Java クラスから利用できるようにするには，アスペクトの定義を以下のようにする：

```
interface Logger extends RemoteAspect {
  void displayLog(Point p, int x);
}
aspect LoggingAspect implements Logger {
  void displayLog(Point p, int x) {
    System.out.println("set x: " + x);
  }
  ...
}
```

Logger インタフェースは，アスペクト内で宣言される displayLog メソッドを宣言する．通常の Java クラスからは次のようにして displayLog メソッドを遠隔呼び出しすることができる．

```
Logger logger = (Logger)
   Aspect.get("LoggingAspect");
logger.displayLog();
```

Logger は，DJcutter が提供する RemoteAspect インタフェースを拡張しなくてはならない．この RemoteAspect は java.rmi.Remote と同様に，ネットワーク越しの Java 仮想マシンから呼び出すことができるインタフェースか否かを識別するのに使われる．Aspect は DJcutter によって提供されるクラスである．Aspect の get メソッドは指定された名前（この例では LoggingAspect）のアスペクトの遠隔参照を返す．遠隔参照の型は，そのアスペクトによって実装されたインタフェース型である．この遠隔参照を使って，アスペクト・サーバ上のアスペクトのメソッドを分散透明に呼び出せる．

DJcutter では，アスペクトへの遠隔参照をリモートプロキシ・パターンで実装している．このプロキシ

オブジェクトによるアーキテクチャは，Java RMI のそれとインタフェース型を通して遠隔参照へアクセスする点で同様である[9]．このアーキテクチャは分割コンパイルが可能であり，分散ソフトウェアの開発に適している．開発者は，インタフェース型を先に定義しておけば，アスペクトを開発する前に通常の Java クラスを開発し，コンパイルすることができる．

3.4 ポイントカット引数

AspectJ と同様に，DJcutter はポイントカットによって抽出したジョインポイントの実行時コンテキストをポイントカットの引数に束縛し，アドバイスコード内で利用することができる．しかし，DJcutter の遠隔ポイントカットの場合，そのポイントカットはアドバイスが実行されるホストとは異なる遠隔ホスト上のジョインポイントを指定しているため，ポイントカット引数は遠隔ホスト上のデータを参照することになる．

たとえば

```
pointcut setter(int x):
   call(void Point.move(int,int))
   && args(x, *)
```

この setter ポイントカットの引数 x は，args によって，遠隔ホスト上の move メソッドの第 1 引数に束縛される．プログラム実行が指定されているジョインポイントに到達したとき，ポイントカット引数に束縛された実行時コンテキストは，直列化され，ネットワーク越しにアスペクト・サーバへ送信される．その後，サーバは直列化されたデータを復元し，それを利用してアドバイスコードを実行する．

3.5 thisJoinPoint によるリフレクション

DJcutter が提供する thisJoinPoint 変数も AspectJ のそれから拡張されている．ThisJoinPoint は，ジョインポイントの実行時コンテキストをリフレクティブにアクセスするための，アドバイスコード内でのみ利用できる特殊な変数である．DJcutter の場合，指定したジョインポイントとアドバイスコードとは異なるホスト上に存在するため，DJcutter の thisJoinPoint は遠隔のジョインポイントの実行時コンテキストを操作するように拡張されている．

また，DJcutter が提供する thisJoinPoint 変数は，対応するジョインポイントが存在するホストの名前を取得するためのメソッド getHost() を提供している．たとえば，以下の before アドバイスでは，Point クラスの最後に呼ばれたメソッドが配置されているホストの名前を記録することができる：

```
String lastCallerHost;
before(): call(void Point.*(..)) {
   lastCallerHost
      = thisJoinPoint.getHost();
}
```

3.6 ローカルアスペクト

分散ソフトウェア内の横断的関心事が，いつも複数のホスト間にまたがっているとは限らない．その場合，アドバイスコードをアスペクト・サーバ上で実行するよりは，対応するジョインポイントと同一のホスト上で実行した方がよい．無駄なネットワーク通信によるオーバヘッドを避けることができる．また，アドバイスコードをアスペクト・サーバ上で実行するモデルでは，AspectJ で開発されたアスペクトを正しく動作させることができない場合がある．

それゆえ，DJcutter は AspectJ のアスペクトと同等のアスペクトも提供している．開発者はアドバイスコードをジョインポイントの存在するホストと同一のホストで実行させるように指定できる．アスペクトのコピーは DJcutter によって自動的にそれぞれのホスト上へ分散配布される．このタイプのアスペクトを**ローカルアスペクト**と呼ぶ．DJcutter の利用者は，分散に関係のない横断的関心事をローカルアスペクトとして定義すればよい．ローカルアスペクトとして定義すれば，DJcutter は AspectJ で開発されたアスペクトを，正しく動作させられることを保証する．

ローカルアスペクトはそれぞれのホスト上に配布されるため，アスペクト内で宣言されたフィールドはホスト間で共有されない．ホスト間でデータをやりとりするには，データを明示的に他のホストへ（Java RMI などを利用して）渡さなくてはならない．

3.7 遠隔デプロイとロード時ウィービング

ここでは，DJcutter の遠隔ポイントカット機構を実現するメカニズムについて簡単に説明する．コンパイルされたアスペクトは，アスペクト・サーバが動作するホストと同じホスト上で動作する**アスペクト・デプロイヤ**に蓄えられ，各ホストに自動で配布される．そして，それぞれのホスト上で動作する DJcutter の拡張クラスローダが，ロード時に通常の Java クラスと配布されたアスペクトの情報とをウィーブする．この自動配布のメカニズムを**遠隔デプロイ**と呼ぶ．通常デプロイとは，プログラムを指定の場所に配布することを指す．しかし，この遠隔デプロイはローダがウィービングに必要なアスペクトの情報のみを配布することを意味している．具体的に配布される情報は，遠隔ポイントカットにより指定されたジョインポイントの情報や，遠隔インタータイプ宣言の情報である．また，蓄えられたアスペクトが 3.6 節で説明したローカルア

スペクトである場合，デプロイヤはローカルアスペクト内で定義されたすべての情報を，各ホストに配布する☆．

この遠隔デプロイの機構は，分散ソフトウェアのための単体テストに役立つ．この点については，次の 4 章で再び取り上げる．

4. アプリケーション例

この章では，DJcutter を用いて記述された 3 つのテストプログラム例を示し，遠隔ポイントカット，遠隔インタータイプ宣言，ローカルアスペクトの有用性を説明する．

4.1 遠隔ポイントカットの使用

2 章で取り上げた分散版のテストコードは非分散版のテストコードに比べ，複雑であった．まず初めに，この分散版のテストコードを DJcutter で記述した例を以下に示す．

```
 1: // on host T
 2: aspect AuthServerTest extends TestCase {
 3:   boolean wasAddUserCalled;
 4:   void testRegisterUser() {
 5:     wasAddUserCalled = false;
 6:     String userId = "muga";
 7:     String  password = "xxx";
 8:     AuthServer auth = (AuthServer)
 9:       Naming.lookup("auth");
10:     auth.registerUser(userId, password);
11:     assertTrue(wasAddUserCalled);
12:   }
13:   before(): // 遠隔ポイントカット
14:     cflow(call(void
15:       AuthServer.registerUser(String,
16:                               String)))
17:     && execution(void
18:       DbServer.addUser(String,String))){
19:     wasAddUserCalled = true;
20:   }
21: }
```

AspectJ を利用したコードと違い，DJcutter を用いたテストコードは分散したサブ・コンポーネントに分割されることはない（**図 2**）．DbServer が動作している D 上で addUser() が実行されたときに，before アドバイス（13 行目から 20 行目）が実行される．しかし，このアドバイスは D 上ではなく，testRegisterUser() が動作しているホスト T 上で実行される．それゆえ，この before アドバイスは，wasAddUserCalled フィールドの値を直接 true に変更することができる．AddUser() の実行をホスト T に通知するためのネットワーク通信を，テスト作成者が明示的に記述する必要はない．

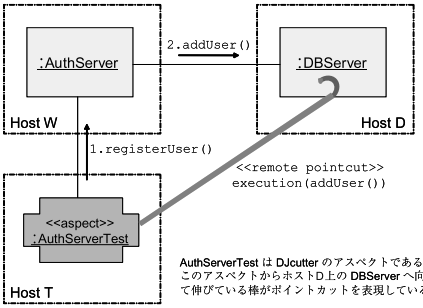

図 2 DJcutter を利用したテストコード
Fig. 2 The testing code in DJcutter.

さらに，上記の before アドバイスは cflow ポイントカット指定子を含んでいる．Java RMI を利用している場合，DJcutter が提供している cflow は，複数のホストにまたがる制御フローを扱える．これはテストコードの正確さを向上させる．上の例では単に，addUser() が実行されたか否か検査するだけではなく，addUser メソッドが registerUser() によって呼ばれているか否かも検査できている．

4.2 遠隔インタータイプ宣言の使用

テストによっては，オブジェクト内部の状態を検査するため，そのオブジェクトのアクセサー・メソッド（accessor method）を必要とするときがある．もしアクセサー・メソッドがテスト対象プログラムにあらかじめ定義されていなかった場合，インタータイプ宣言により，そのメソッドをテストのためだけに追加することができる．たとえば，開発者は registerUser() によって送られるデータが実際に addUser() によってデータベースに登録されているか否かを確認したいとする．これを実現するために，アクセサー・メソッド containsUser を DbServer クラス内に定義できる．containsUser() は，追加されたユーザ・エントリがデータベース内に格納されているか否かを private フィールドなどにアクセスしてチェックできる．

DJcutter の遠隔デプロイ機能を使えば，そのようなインタータイプ宣言を含むテストを簡単に実現することができる．一方，AspectJ の場合，開発者はコン

☆ DJcutter には，アスペクトの情報がデプロイヤにより各ホストに配布されるまで，ウィーブ対象のプログラムを各ホストに配布できない，という仕様がある．これは，DJcutter では，すでにロード済みクラスに対し，アスペクトをウィーブすることができないためである．しかし，テストの場合，開発者がこの仕様にそってテストプログラムを記述・実行することは容易である．そのため，この仕様はテストの自動実行，またはテストの実行の反復をさまたげるものではない．

パイル済みのアスペクトを手動で各ホストに配布しなければならない．

以下に，DJcutter を使って DbServer クラスに，containsUser() を追加するテストコードを示す（14 行目から 17 行目）．TestRegisterUser() は，まずユーザ muga がデータベースに格納されていないことを確認する（10 行目）．そして，次に registerUser メソッドを呼び出す（11 行目）．その後，今度はユーザ muga がデータベースに登録されていることを containsUser() を使って確認する（12 行目）．

```
 1: // on host T
 2: aspect AuthServerTest extends TestCase {
 3:   void testRegisterUser() {
 4:     String userId = "muga";
 5:     String  password = "xxx";
 6:     AuthServer auth = (AuthServer)
 7:       Naming.lookup("auth");
 8:     DbServer db = (DbServer)
 9:       Naming.lookup("db");
10:     assertTrue(!db.containsUser(userId));
11:     auth.registerUser(userId, password);
12:     assertTrue(db.containsUser(userId));
13:   }
14:   boolean DbServer.containsUser(
15:                         String userId) {
16:     ... ...
17:   }
18: }
```

4.3 Around アドバイスの使用

単体テスト時には，その単体が依存している他のプログラムから切り離してテストするため，擬似オブジェクト（mock object）が利用される．たとえば，AuthServer の registerUser メソッド内で，利用されている DbServer を擬似オブジェクトに置き換えるとする．これにより，DbServer の内部実装に依存しないテスト結果を得ることができる．

DJcutter の提供する around アドバイスを利用すれば，もとの処理と疑似処理を容易に置き換えることができる．テスト作成者は AuthServer 内で DbServer を利用している箇所を手動で擬似オブジェクトに書き換える必要はない．以下に，DbServer の利用箇所を擬似オブジェクトに差し替えるコードを示す☆．

```
 1: // on host T
 2: local aspect MockDbServer {
 3:   pointcut replacingMock():
 4:     within(AuthServer)
 5:     && call(void
 6:       DbServer.addUser(String,String));
 7:   void around(): replacingMock() {
 8:     // 疑似処理
 9:   }
10: }
```

このテストコードでは，AuthServer 内での DbServer へのアクセスを around アドバイスで置き換えている．Around アドバイスは，このテストのためだけの擬似処理を実装している．

ここで使用されているアスペクトは DJcutter のローカルアスペクトである．このテストの場合，around アドバイスは AuthServer と同じホスト上で動けばよい．したがって，ローカルアスペクトの利用により，不必要なネットワーク通信を避けることができる．

5. 実　　験

5.1 遠隔ポイントカットの通信オーバヘッド

遠隔ポイントカットの実行性能を測定するため，我々は Java RMI を利用した AspectJ のプログラムと DJcutter のプログラムを比較した．実行した Java RMI と AspectJ のプログラムは，2.2 節のテストプログラムである．DJcutter の方は，4.1 節のプログラムである．これらのプログラムは，AuthServer の registerUser メソッドが DbServer の addUser() を実行しているかどうかを確認するものである．我々はそれぞれのプログラムの testRegisterUser() の実行にかかった時間を測定した．この実験の際に，AuthServer と AuthServerTest は同じマシン Sun Fire V480☆☆上で実行させ，DbServer は他のマシン Sun Blade 1000☆☆☆で実行させた．また，Java の実行環境は Sun JDK 1.4.0_01，AspectJ は 1.2 を利用した．

表 2 に測定結果を示す．この実験結果から，DJcutter の実行時間は AspectJ のものと同等であることが分かる．実験では DJcutter の方が若干実行時間が短いが，これは以下の理由からである．2.2 節の AspectJ のプログラムでは，before アドバイスのコードが実行されるときに，遠隔参照 test が confirmCall() を呼ぶために取得される．一方，この遠隔参照は DJcutter のプログラムでは取得されない．DJcutter では，ランタイム・システムが事前に遠隔参照を取得しておく

☆ 最近では，MockServlet や MockEJB など，典型的な擬似オブジェクトをベンダが提供している場合もある．しかし，今回の例では，DbServer はそのようなサポートがされていないプログラムであると仮定している．

☆☆ UltraSPARC III Cu 900 MHz ×4 with 16 GB of memory and Solaris 9

☆☆☆ Dual UltraSPARC III 750 MHz with 1 GB of memory and Solaris 8

表 2 testRegisterUser() の実行にかかった時間（msec.）
Table 2 The elapsed time (msec.) of testRegisterUser().

Pointcut parameters	()	(String)	(String,String)
Java + Java RMI	5.2	5.3	5.4
AspectJ + Java RMI	5.3	5.3	5.4
DJcutter	4.4	4.5	4.5
DJcutter without cflow	4.2	4.2	4.3
DJcutter sending thisJoinPoint	4.9	4.9	5.1

からである．Java RMI では，この遠隔参照の取得のため，レジストリ・サーバと通信しなくてはならない．この相違により，Java RMI を利用した AspectJ のプログラムの実行時間は DJcutter のものよりも，約 1 msec 長い．

2.2 節と 4.1 節で示したプログラムはポイントカット引数を利用していない．ネットワーク越しに引数を送信することによる影響を測定するため，我々は DJcutter の before アドバイスに addUser() メソッドが実行される際の実引数を受け取らせたときの実行時間も測定した．また，AspectJ の confirmCall() メソッドの引数にも同じ変更を加えた．引数の型は String である．表 2 の実験結果より，ポイントカット引数により実効性能が受ける影響は少ないことが示された．

さらに公平な比較のため，我々は cflow ポイントカットを除いて記述された DJcutter のプログラムの実行時間も測定した．AspectJ のプログラムはもともと cflow を利用していない．表 2 の結果から，cflow を利用するオーバヘッドはそれほど大きくないことが分かる．

5.2 thisJoinPoint 変数の通信オーバヘッド

DJcutter のアドバイス内で thisJoinPoint を利用するためには，アスペクト・サーバへ暗黙的に thisJoinPoint オブジェクトを送信しなくてはならない．このオブジェクトのネットワーク通信にかかるコストは小さくない．現在の DJcutter では，アドバイス内でこの変数を利用していない場合，サーバに thisJoinPoint オブジェクトを送信しないという最適化を行っている．最適化を行わなかった場合の実行時間を 表 2 に示す．結果から，この例の場合，最適化により 0.7 msec 程度の高速化が達成されていることが分かる．

6. 関 連 研 究

Soares らは，AspectJ を使って，Java RMI により記述された分散プログラムのモジュール性を改善する方法を示している[10]．Java RMI の利用者は，Java RMI の仕様にそったプログラミングを強要される．Soares らはこの仕様を AspectJ でアプリケーションロジックから分離できることを示した．しかし彼らの研究は AspectJ を使っているので，本稿で扱っている問題には対処していない．

Java RMI は標準的なフレームワークであるが，cJVM[11]，Addistant[12]，J-Orchestra[13] など，他のシステムも開発されている．これらのシステムはネットワーク越しに結合された複数のホスト上に，1 つの Java 仮想機械を実現する．これらのシステムを利用すると，非分散ソフトウェアとした開発された元のプログラムを，自動的に分散実行できる．

AspectJ を使って書かれた 2.1 節の非分散版のテストプログラムを，これらのシステム上で分散実行すれば，遠隔ポイントカットを使って書かれた 4.1 節のプログラムと同等の結果が得られる．AspectJ のポイントカットは，これらのシステムにより自動的に，DJcutter の遠隔ポイントカットに変換されると見なせる．

しかしながら，本稿で取り上げたテストプログラムを書く用途には，この方法は適切ではない．この方法では，テスト対象のアプリケーション・プログラムも，cJVM などの上に（非分散プログラムとして）構築されている必要がある．しかし現実のアプリケーション・プログラムは，J2EE サーバなどの上に構築されており，テストのために cJVM などの上に再構築するのは非現実的である．一方，DJcutter には，そのようなプラットフォームに関する制限がない．DJcutter は遠隔デプロイとウィービングのために，専用のクラスローダを必要とするが，これを既存のプラットフォームに組み込むことは，実用上，十分可能であると思われる．

分散処理はよく知られた横断的関心事であり，これらの関心事を支援するためいくつかのシステムが提案されてきた．たとえば，D 言語[14] により，開発者は遠隔メソッドのパラメータをどのように受け渡すかを元プログラムから分離して記述することができる．これらのシステムの目的は分散処理の特定の横断的関心事を調査し，それらを分離することである．これらと同様に我々の研究も，分散ソフトウェアのテストとい

う関心事を分離するための AOP 言語を，AspectJ の拡張という形で提案している．

JAC[15),16)] は Java 言語で記述された動的 AOP のためのフレームワークである．AspectJ や他の AOP 言語と異なり，JAC はポイントカット–アドバイスをクラスライブラリとして実現している．さらに JAC は，DJcutter のローカルアスペクトのように，他ホスト上のジョインポイントを指定して，そのホスト上でアドバイスコードを実行する機能を提供している[17)]．しかし，DJcutter のような遠隔ポイントカットの機構を備えていない．そのため，本稿で示した分散ソフトウェア用のテストコードを JAC で記述すると，AspectJ と同様，モジュール実装が複数の分散化したサブ・コンポーネントから構成されてしまう．

DADO（Distributed Adaplets for Distributed Objects）[18)] は，異機種上に分散したシステム内の横断的関心事をサポートするために，CORBA に似たプログラミングモデルを提供している．このプログラミングモデルにより，開発者はクライアント側，サーバ側両方のアプリケーション・コンポーネントに現れるセキュリティやキャッシュなどの横断的関心事を分離することができる．しかし，この DADO はクライアントとサーバ間の通信のモデルに特化しており，本稿で取り上げているテストをモジュール化することは難しい．

7. 議　　論

我々は，分散ソフトウェア用のテストプログラムを簡潔に記述するための AOP 言語 DJcutter を提案した．この言語の特徴である，遠隔ポイントカットは遠隔ホスト上で実行されているプログラム中のジョインポイントを抽出する機構である．これにより，利用者は複数のホスト間にまたがった横断的関心事を，分散を意識せずに単一ホスト上で動作する簡潔なアスペクトとしてモジュール化することができる．遠隔メソッド呼び出し（RMI）が分散オブジェクト指向技術に重要であると同様に，遠隔ポイントカットは分散 AOP のための重要な言語機構であると我々は考える．

また本稿では，この DJcutter 言語が，分散ソフトウェアのテストコードを記述するために有用であることを例示した．分散ソフトウェアのテストは横断的関心事であるが，AspectJ では分散を意識せずにはうまくモジュール化できない．本研究では，遠隔ポイントカットや遠隔インタータイプ宣言，そして遠隔デプロイの機構を利用して，この横断的な実装を分散を意識しない簡潔なプログラムとして記述できることを示した．

我々は，DJcutter 言語と，DJcutter を利用した分散ソフトウェア用のテスティング・フレームワーク *DjcTest* のベータ版とを Web にて公開する予定である☆．現在の DJcutter は，アスペクト・サーバ上でアドバイスを実行するというモデルをとっている．これは DJcutter が分散ソフトウェアのテストプログラムを記述することを主目的としているためであるが，我々はそれ以外の横断的関心事を簡潔に記述できるように言語拡張していこうと考えている．本稿で述べたローカルアスペクトもその 1 つではあるが，それ以外にもアドバイスコードが動作するホストを利用者が選択できるようにするなどといった拡張も考えられる．

☆ http://www.csg.is.titech.ac.jp/~muga/mitou2004/

参 考 文 献

1) Beck, K.: *Extreme Programming Explained: Embrace Change*, chapter 4, Addison-Wesley (1999).
2) Object Mentor: *JUnit.org*, Online publishing (2001). http://www.junit.org/index.htm
3) Apache Software Foundation: Apache Jakarta Project, CACTUS. http://jakarta.apache.org/cactus/
4) Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C.V., Maeda, C. and Mendhekar, A.: Aspect-Oriented Programming, *European Conference on Object-Oriented Programming 1997* (*ECOOP 1997*), LNCS 1241, pp.220–242, Springer (1997).
5) Tarr, P., Ossher, H., Harison, W. and Jr, S.M.S.: N degrees of separation: Multi-dimensional separation of concerns, *International Conference on Software Engineering 1999* (*ICSE 1999*), pp.107–119, IEEE Computer Society Press (1999).
6) Orleans, D. and Lieberherr, K.: DJ: Dynamic Adaptive Programming in Java, *International Conference on Meta-level Architectures and Separation of Crosscutting Concerns 2001* (*Reflection 2001*), pp.73–80, Springer Verlag (2001).
7) Bergmans, L. and Aksits, M.: Composing crosscutting concerns using composition filters, *CACM*, ACM Press (2001).
8) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *European Conference on Object-Oriented Programming 2001* (*ECOOP 2001*), LNCS 2072, pp.327–353,

Springer (2001).
9) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter 4, Addison-Wesley (1995).
10) Soares, S., Laureano, E. and Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ, *Object-Oriented Programming Systems, Languages, and Applications 2002* (*OOPSLA 2002*), ACM SIGPLAN Notices, pp.174–190 (2002).
11) Aridor, Y., Factor, M. and Teperman, A.: cJVM: A Single System Image of a JVM on a Cluster, *International Conference on Parallel Processing 1999* (*ICPP 1999*), pp.4–11 (1999).
12) Tatsubori, M., Sasaki, T., Chiba, S. and Itano, K.: A Bytecode Translator for Distributed Execution of Legacy Java Software, *European Conference on Object-Oriented Programming 2002* (*ECOOP 2002*), LNCS 2072, pp.236–255, Springer (2001).
13) Tilevich, E. and Smaragdakis, Y.: J-Orchestra: Automatic Java Application Partitioning, *European Conference on Object-Oriented Programming 2002* (*ECOOP 2002*), Springer (2002).
14) Lopes, C.: D: A Language Framework for Distributed Programming, Ph.D. Thesis, College of Computer Science, Northeastern University (1997).
15) Pawlak, P., Seinturier, L., Duchien, L. and Florin, G.: JAC: A Flexible Solution for Aspect-Oriented Programming in Java, *International Conference on Metalevel Architectures and Separation of Crosscutting Concerns 2001* (*Reflection 2001*), LNCS 2192, pp.1–24, Springer (2001).
16) ObjectWeb Consortium: Online publishing. http://jac.objectweb.org/index.html
17) Pawlak, P., Duchien, L., Florin, G., Legond-Aubry, F., Seinturier, L. and Martelli, L.: JAC: An Aspect-Based Distributed Dynamic Framework, *Aspect-Oriented Software Development 2003* (*AOSD 2003*) *Tutorial* (2003).
18) Wohlstadter, E., Jackson, S. and Dvanbu, P.: DADO: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems, *International Conference on Software Engineering 2003* (*ICSE 2003*), pp.174–186, IEEE Computer Society Washington, DC, USA (2003).

(平成 16 年 10 月 25 日受付)
(平成 17 年 5 月 9 日採録)

西澤 無我

1979 年生．2002 年東京工業大学理学部情報科学科卒業．2004 年同大学院情報理工学研究科修士課程修了．同年より，同大学院情報理工学研究科博士課程在学中．プログラミング言語処理系，分散コンピューティング，基盤ソフトウェアに関する研究に従事．

千葉 滋（正会員）

東京工業大学大学院情報理工学研究科助教授．1991 年東京大学理学部情報科学科卒業．1993 年同大学院理学系研究科情報科学専攻修士課程修了．1996 年同専攻より理学博士．東京大学助手，筑波大学講師，東京工業大学講師を経て現職．プログラミング言語およびオペレーティング・システム等，システムソフトウェアの開発に興味を持っている．

立堀 道昭（正会員）

1974 年生．2002 年筑波大学大学院博士課程工学研究科修了．同年より，IBM 東京基礎研究所副主任研究員．プログラミング，分散システム，セキュリティの研究に従事．日本ソフトウェア科学会，ACM，IEEE 各会員．