連想アスペクトによるアプリケーション連携の記述改善評価

櫻井 孝平　増原 英彦　松浦 佐江子　古宮 誠一

本研究では**連想アスペクト**と呼ぶ言語機構の評価を行う．連想アスペクトは AspectJ のようなアスペクト指向プログラミング言語を拡張し，オブジェクトのグループにアスペクトのインスタンスを関連づける機構として提案されている．連想アスペクトはオブジェクト間の協調動作のような関心事をモジュール化するのに有用であると予想されるが，実際のアプリケーション記述でそれを確かめた例はまだない．そこで本研究では，オブジェクト指向言語で記述された複数のアプリケーションを連携させて 1 つのシステムを構築する際に連想アスペクトを利用し，その有用性を他のプログラミング手法と比較する．具体的にはテキストエディタ，コンパイラ，開発プロジェクト管理システムを連携させて統合開発環境を構築する例を用いる．

This work evaluates association aspects, which are proposed as an extension to AspectJ like aspect-oriented programming languages for supporting aspect instances associated to a group of objects. The primary application of association aspects is to modularize concerns that coordinate objects. However, there have been no practical examples that confirm this. This work applies association aspects for building systems by integrating applications written in object-oriented language and compares their usefulness against other programming styles. In particular, we build an integrated development environment consisting of a text editor, a compiler and a software project management system.

1 はじめに

アスペクト指向プログラミング (AOP) [4] は，ロギングや例外処理など複数のモジュールにまたがる横断的関心事をモジュール化するためのプログラミング手法である．

本稿では連想アスペクト [6] という AOP 言語のための機構を，複数のアプリケーション連携の進化に適用し，その有用性を評価する．

アプリケーション連携 (以下単に連携と言う) [7] [8] とは，既存の複数のアプリケーションをもとにして，それらの処理を統合することで 1 つのシステムを構築することである[†1]．本研究ではエディタやコンパイラ，開発プロジェクト管理システムをもとにして統合開発環境を構築する例を評価する．

ここでは，アプリケーションを別のものにしたり，統合させる処理を追加，変更することを連携の進化と呼ぶ．更に，本研究では逐次進化型の開発手法によって連携システムが構築されるものとする．つまり，初期の段階では単純な機能のみを持ったシステムが構築され，統合させるアプリケーションやその処理を追加，変更することで高機能なシステムが構築されてゆくものとする．

連想アスペクトはオブジェクト間の関連における関心事をモジュール化するための言語機構であり，連携のモジュール化に適していると予想される．従って，連想アスペクトを用いて記述されたシステムは連携が進化する場合に，アスペクトのみの局所的な修正，追加，削除によって容易に対応できると考えられる．こ

Evaluation of Association Aspects through Constructing Integrated Applications.
Kouhei Sakurai, Saeko Matsuura, Seiichi Komiya, 芝浦工業大学, Shibaura Institute of Technology
Hidehiko Masuhara, 東京大学, University of Tokyo
コンピュータソフトウェア, Vol.22, No.3 (2005), pp.222–228.
[小論文] 2004 年 8 月 11 日受付.

†1 システム統合 (system integration) とも呼ばれており，横断的関心事としての問題点を述べている [8].

のことを実証する目的で本研究では統合開発環境の構築における評価を行う．

以下, 2 節では連携と連想アスペクトについて簡単に紹介し, 3 節では統合開発環境におけるアプリケーション連携の進化について説明する．4 節では, 3 節で紹介した連携について各言語での実際の記述を比較する．5 節では連携の進化の一般性について議論する．6 節では関連研究, 7 節では結論を述べる．

2 アプリケーション連携と連想アスペクト

2.1 アプリケーション連携の例

ここでは簡単なアプリケーション連携の例として, テキストエディタとコンパイラによって統合開発環境を構築することを考える．

多くの統合開発環境では, テキストエディタがソースファイルを保存すると自動的にそのファイルがコンパイルされる．テキストエディタやコンパイラは既存のアプリケーションを用いることにすると, エディタの保存動作とコンパイラのコンパイル動作を統合することがここでの連携である．

テキストエディタは Editor クラスとして既に存在すると考える．Editor クラスはテキストファイルを開いて編集することができ, `save` メソッドによってファイルを保存することができる．また, テキストが編集されると, `isChanged` メソッドが `true` を返すようになる．`save` メソッドで保存されれば, `isChanged` は `false` が返るようになる．

コンパイラは Compiler クラスとして提供される．Compiler クラスは対象の言語や設定などによって別々のインスタンスが存在し, `compile` メソッドによって渡されたファイルをコンパイルする．

2.2 連想アスペクトの記述

アプリケーションの連携 [7] [8] のような横断的関心事は, 既存の AOP 言語である AspectJ [3] のアスペクトのインスタンス化機構では簡潔に記述できない [8]．アプリケーションの連携には, 複数のアプリケーションの内部オブジェクトを互いに関連づけ, あるアプリケーションの内部オブジェクトの振舞に反応して, 他のアプリケーションのオブジェクトの特定の振舞を起動するような処理の統合を行う言語機構が必要である．そこで連想アスペクトと呼ぶアスペクトのインスタンス化機構を提案し, AspectJ の拡張として設計, 実現した[†2]．連想アスペクトによって, 統合される複数のオブジェクトに関連づけられるアスペクトのインスタンスを定義することが可能になる．また, 関連づけたオブジェクトの振舞に反応するアドバイス宣言を記述することで, 連携の処理の統合を実現することができる．

Editor クラスと Compiler クラスを連携させる連想アスペクト AutoCompile の記述を以下に示す:

```
aspect AutoCompile
   perobjects(Editor, Compiler) {
  File f; //アスペクトのインスタンスフィールド
  //アスペクトのコンストラクタ
  public AutoCompile(Editor e, Compiler c)
  { associate(e, c);  f = e.getFilename();}


  //アドバイス宣言
  after(Editor e, Compiler c):
    execution(public void Editor.save())
    && target(e) && associated(e, c) {
    c.compile(f); } }
```

2 行目の `perobjects` は AutoCompile が連想アスペクトであり, Editor 型と Compiler 型のオブジェクトの組に関連づけられることを宣言している．連想アスペクトは new 演算子で明示的にインスタンス化することができる．

実際のオブジェクトに対する関連づけは `associate` メソッドで行う．AutoCompile のコンストラクタでは, `associate` メソッドを呼んで, 引数として渡された Editor と Compiler のオブジェクトに関連づける．

AutoCompile に定義されている after アドバイスは, Editor クラスの `save` メソッドの実行後の処理を定義している．アドバイス定義に現れる `associated` ポイントカットは, アドバイスを実行するアスペクトインスタンスを検索する．ここでは `target(e)` ポイ

†2 連想アスペクトのコンパイラは `http://www.komiya.ise.shibaura-it.ac.jp/~sakurai/` で公開されている．

ントカットが束縛する, save メソッドを実行したオブジェクト e に関連づけられたアスペクトインスタンスそれぞれにアドバイスの本体を実行させる．その際, 変数 c には各アスペクトインスタンスが関連づけられている Compiler オブジェクトが束縛される．アドバイスの本体では Editor が編集したファイルを Compiler の compile メソッドに渡している．

実際の AutoCompile にはエディタがファイルを開いたときに AutoCompile インスタンスを作成するアドバイスや, エディタがファイルを閉じたときに AutoCompile インスタンスの関連を解消するアドバイスなども含まれる．尚, 連想アスペクトの機能の詳細は別の論文 [6] を参照されたい．

3 統合開発環境の進化

連想アスペクトが連携という関心事の分離にどれだけ貢献しているかを評価するために, 統合開発環境における連携の進化を複数想定する．一般的に, 連携の進化は開発の過程でシステムのバージョンごとに変化すると考えられる．連想アスペクトによって関心事の分離がうまく行われていれば, バージョンアップによる連携の進化に伴って必要な記述の変更箇所が少なくなり, また局所化することが可能であると期待される．

各アプリケーションは独立して開発されており, アプリケーション間の連携は設計上, 考慮されていないとする．一般的にソフトウェアは, 要求の変化等に伴って複数のバージョンが作成されるので, 本研究では各アプリケーションの連携として以下のようなバージョンを考える．

- バージョン 1: Editor と Compiler の連携: 前節で紹介したように, エディタがファイルを保存すると, 自動的にコンパイルが行われる．
- バージョン 2: Editor と Project の連携: コンパイルの対象を 1 つのプロジェクトを構成する複数のファイルとする．つまり, エディタがファイルを保存すると自動的にプロジェクトのコンパイル (ビルド) を行う. またそれに先立ってプロジェクト内の他のエディタを保存する．
- バージョン 3: Project と Project の連携: 上に加えて, あるプロジェクトが他のプロジェクトに依存するような場合に, あるプロジェクトをビルドすると, 依存するプロジェクトを先にビルドする．

ここでは紙面の都合から Editor と Project の連携についてのみ詳しく説明する．バージョン 1 からバージョン 2 への変化は Editor と連携させるアプリケーションが Compiler から Project へ変化したことであり, これによって連携の記述を変化させる作業が生じると考えられる．

一般的な開発プロジェクトでは, 複数のソースファイルからシステムが作られているため, それらを一括してビルドしなければならない．そのための独立したアプリケーションとして, 開発プロジェクトに関連するファイルを一括してビルドを行うための Project クラスが用意されているとする†3.

統合開発環境での作業は Project が持つファイルを Editor で編集し, Project に対して build を実行することである．2.1 節で述べた Editor と Compiler の連携は, Editor と Project の連携に置き換わり, Editor が save を行った後に, Project が build を行う．また Editor で編集中のファイルを最新の状態で build を行うためには, Project が build を行う前に, その Project に含まれるファイルを編集している全ての Editor は, save を実行しなければならない．この連携を記述する上で注意しなければならないのは, save の呼び出しに反応して再帰的に build が実行され, 無限ループに陥らないようにすることである (図 1)†4.

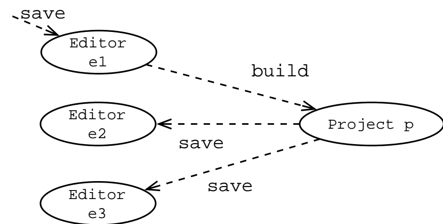

図 1 Editor と Project の連携

†3 Unix の make コマンドや, Apache ant を抽象化したものと考える．

†4 開発プロジェクトに含まれるファイルの種類ごとに異なるエディタに対応させることも可能であるが, 記述が複雑になるためここでは Editor は 1 種類とした．

```
aspect AutoBuild {
  Set Editor.assocs = new HashSet();
  Set Project.assocs = new HashSet();

  static class Relation {
    Editor e; Project p; boolean busy;
    Relation(Editor e, Project p) {
     this.e = e; this.p = p; } }

  static associate(Editor e, Project p) {
    Relation r = new Relation(e, p);
    e.assocs.add(r); p.assocs.add(p); }

  before(Project p): target(p) &&
  execution(public void Project.build()) {
   for (Iterator iter = p.assocs.iterator();
       iter.hasNext(); ) {
    Relation r = (Relation) iter.next();
    r.busy = true;
    if (r.e.isChanged()) r.e.save();
    r.busy = end; }
    ...}
```

図 2 AspectJ による AutoBuild アスペクトの記述

4 記述の比較

前節で示した連携による統合開発環境の作成を, Java, AspectJ 及び連想アスペクトによって記述し, 比較する. 以下ではそれぞれの言語でどのように連携を記述するかの概要を示し, 続いて比較する点と比較結果を示す.

4.1 各言語の記述

4.1.1 Java による記述 (*JAVA*)

Java 言語は単一継承のオブジェクト指向言語なので, アプリケーションの連携を記述する場合は, あらかじめ連携される側のクラスに連携のための機構を用意しておかなければならない.

典型的な手法は通知を受け取る汎用の Listener インターフェースを定義し, 既存の Editor クラスや Project クラスには, Listener インスタンスの管理と, Listener に対する通知のコードを侵入的に追加するというものである. 詳細は省略する.

4.1.2 AspectJ のシングルトンアスペクトによる記述 (*SNG*)

AspectJ によって, Editor と Project の連携をする AutoBuild アスペクトは図 2 のように記述できる. AspectJ のアスペクトはシングルトンが基本であるため, 型間定義により, Editor と Project にはフィールドを追加し, 関連に関する情報を表現する Relation クラスのインスタンスを複数格納する. アドバイスの本体ではそれぞれの Relation インスタンスについて, 関連する Editor の `save` を呼び出すループを行う. Relation クラスのフィールド `busy` は, 再帰的なアドバイスを防止するための, 関連ごとの状態である. `before` アドバイスは, `build` が呼ばれた Project に関連づけられている Relation についてループを行い, 対応する Editor が編集されているかどうかの判定を行って, Editor に `save` を実行する. 図では省略しているが, Editor の `save` に反応して, Project の `build` を呼び出すアドバイスも同様に記述される.

4.1.3 連想アスペクトによる記述 (*ASSOC*)

AutoBuild アスペクトは連想アスペクトによって図 3 のように定義できる.

関連ごとの状態である `busy` フィールドは, アスペクトのインスタンス変数として定義される.

`before` アドバイスは, `associated` ポイントカットによって, Project に関連づけられたアスペクトのインスタンスが検索され, 関連の相手である Editor を引数 `e` に束縛する. そして `if` ポイントカットによる判別式でエディタが編集されている場合にのみアドバイスの本体が実行されることになる. アドバイス本体

実際には例えば, ソースファイルはテキストエディタのオブジェクトと連携させ, GUI のリソースファイルは専用の GUI エディタを利用するように連携させることも可能である.

```
public aspect AutoBuild
     perobjects(Editor, Project) {
  boolean busy = false;

  before(Editor e, Project p):
     execution(public void Project.build())
      && target(p) && associated(e, p)
      && if(e.isChanged()) {
    busy = true; e.save(); busy = false;  }
   ...}
```

図 3 連想アスペクトによる AutoBuild アスペクトの記述

は関連づけられた各インスタンスにより実行され, `e` に対して `save` を呼び出す. Editor の `save` メソッドに反応するアドバイスも同様である.

4.1.4 AspectJ の抽象アスペクトによる記述 (*ABS*)

AspectJ の抽象アスペクトを用いることによって, 連想アスペクトと同様の機構を実現する手法が考えられる. 紙面の都合で詳細は略すが, その場合, 関連づけはグローバルな Map で管理され, オブジェクトの組み合わせから Iterator を返す `associated` メソッドが定義される.

この抽象アスペクトによる記述は, Hannemann らが示した Mediator パターンを利用した記述に相当する [2].

4.2 評価基準

これまでに説明した各言語による記述は, 以下のような基準によって比較できる:

非侵入性 (*non-intrusiveness*): 各アプリケーションを修正せずに連携を実現できること. Java による記述では, 連携するクラスごとに Listener を呼び出すような修正が必要となるため, これを満たさない. またこのような修正は進化に応じて追加が必要であり, アプリケーションの再利用性を妨げるので回避されるべきである.

連携の局所化 (*locality*): アプリケーションの連携の関心事が記述としてまとまっているかどうか. 通常の Java では連携の記述が各クラスに分散し, 記述が困難になり保守性の低下を招く.

関連の実現の隠蔽 (*hiding*): 関連の集合を格納するためのコレクションの操作の記述が関連を利用した処理から隠れていること. 通常の Java や AspectJ では連携ごとに必要となるが, 連想アスペクトは言語機構として隠蔽される.

状態と挙動のカプセル化 (*encapsulation*): 関連のための状態と関連の振舞を 1 つのモジュールとして定義できること. AspectJ ではシングルトンアスペクトが基本であり, 関連の状態が別のクラスとして定義される. このクラスは連携を追加する度に増えていき, 簡潔な記述の妨げになる.

アドバイスにもつれがないこと (*untangling*): 連携の処理の記述の中に, 関連を検索するための処理が混在しないこと. AspectJ ではループが混在するため保守性を低下させる. 連想アスペクトでは, `associated` ポイントカットによる検索によりアドバイス本体が複数回実行されることで, もつれはなくなる.

ポイントカットによる宣言的な条件判定 (*declarativeness*): 連携の処理を行う条件がポイントカットとしてひとまとめに記述できること. 連想アスペクトでは `associated` ポイントカットにより全ての条件をポイントカットに記述できるが, ポイントカットにかえて AspectJ では関連クラスのインスタンスごとに `if` 文を使って判別しなければならない. この判定による分岐も保守性を低下させる.

4.3 評価結果

前節の基準により評価結果は表 1 のようにまとめることができる. 表の各列は, 4.1.1〜4.1.4 節に対応しており, ○はその言語による記述が評価基準を満たしていることを示す.

表 1 評価結果

評価項目	*JAVA*	*SNG*	*ABS*	*ASSOC*
non-intrusiveness		○	○	○
locality		○	○	○
hiding			○	○
encapsulation				○
untangling				○
declarativeness				○

5 議論

本研究では連携の例として統合開発環境を取り上げ, 連想アスペクトの評価を行ったが, このような連携は特殊なものではなく, ある程度普遍的なものであると予測する. その他の例としてメールクライアントとアドレス帳の連携や, テキストエディタとスペルチェッカの連携などが考えられる. 具体的にはメール作成時のアドレス欄の入力により, アドレス帳から対応するアドレスを検索して自動補完を行ったり, メールを受け取ったらアドレス帳に自動で差出人のアドレスを追加するという連携や, エディタ上の入力に反応してスペルチェッカがスペルをチェックするような連携になるだろう. これらのアプリケーション連携は, ユーザーの要求に応じて追加, 変更が行われ, 段階的に進化していくことになる.

連想アスペクトではこれらの連携をアスペクトとしてモジュール化し, 連携するオブジェクト間に関連づけられたアスペクトのインスタンスを扱うことで簡潔に記述できる. 連携が進化した場合は, アスペクト内の書き換え, もしくはアスペクトの付け替えのみで対応することができる.

6 関連研究

Coady と Kiczales は [1] において FreeBSD OS カーネルの最適化などの非機能的関心事をアスペクトとして分離し, OS のバージョンアップにおけるコード修正に AOP が有効であることを示した点が大きく異なる. 本稿では, この研究で述べられている非機能的なアスペクトとは異なり, アプリケーションの連携について示した.

Rajan と Sullivan は連想アスペクトと同様の言語機構を持つ Eos [5] の提案において, 1 ビットの状態を持つオブジェクトを連携させる例を用いて記述の比較を行っているが, 実際的なアプリケーションに応用した事例や, 連携が進化する事例は報告されていない. 本研究ではより実践的な, 統合開発環境の進化においても連想アスペクトが有効であることを示した.

7 おわりに

本研究では, 統合開発環境のようなアプリケーションの連携が横断的関心事であることを示し, そのようなアプリケーション連携が連想アスペクトにより簡潔に記述できることを示した. 連想アスペクトによりアプリケーションの連携は, アスペクトとしてモジュール化され, 連携するオブジェクト間の状態をアスペクトのインスタンスとしてできるので, アスペクトのインスタンスの扱いに制限がある通常の AspectJ と比較して, 簡潔な記述となる.

将来的な課題としては, 統合開発環境の例における連想アスペクトの記述結果からより定量的な評価を導きだすことである. また, 他の実践的なアプリケーションの例としてセキュリティポリシーなどの記述改善を検討している.

謝辞

本研究の一部は, 科学研究費特定領域研究「IT の深化の基盤を拓く情報学研究」の組木プロジェクトの下で遂行された.

参 考 文 献

[1] Coady, Y. and Kiczales, G.: Back to the future: a retroactive study of aspect evolution in operating system code, in *AOSD 2003*, ACM Press, 2003, pp. 50–59.

[2] Hannemann, J. and Kiczales, G.: Design Pattern Implementation in Java and AspectJ, in *OOPSLA 2003*, ACM Press, 2002, pp. 161–173.

[3] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, in *ECOOP 2001*, Springer-Verlag, 2001, pp. 327–353.

[4] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, in *ECOOP 1997*, Springer-Verlag, 1997, pp. 220–242.

[5] Rajan, H. and Sullivan, K.: Eos: Instance-Level Aspects for Integrated System Design, in *FSE/ESEC 2003*, ACM Press, 2003, pp. 297–306.

[6] Sakurai, K., Masuhara, H., Ubayashi, N., Matsuura, S. and Komiya, S.: Association Aspects, in *AOSD 2004*, ACM Press, 2004, pp. 16–25.

[7] Sullivan, K.: Mediators: Easing the Design and Evolution of Integrated Systems, PhD Thesis, Dept. of Computer Science, University of Washington,

published as *TR UW-CSE-94-08-01*, (1994).
[8] Sullivan, K., Gu, L. and Cai, Y.: Non-Modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ, in *AOSD 2002*, ACM Press, 2002, pp. 19–26.