デザインパターンの使用を支援するアスペクトウィーバ

An Aspects-Weaver for Programming with Design Patterns

立堀 道昭[†] 千葉 滋[††,†††]
Michiaki TATSUBORI Shigeru CHIBA

†筑波大学大学院 工学研究科
Doctoral Program in Engineering, University of Tsukuba
††筑波大学 電子・情報工学系
Institute of Information Science and Electronics, University of Tsukuba
†††科学技術振興事業団 さきがけ研究21
Japan Science and Technology Corp.

概 要

デザインパターンを利用したプログラミングを支援する機構が望まれている．これまで，我々はクラスを基盤とした高度なマクロによる拡張機構を備えた言語処理系として OpenJava を開発してきた．しかしながら，研究を進めるにつれ，ある種のデザインパターンの使用を支援するためにはその機構は必ずしも十分でないことがわかってきた．本稿では OpenJava の経験を基にその問題点を議論する．そして，デザインパターンに関連するプログラム中の記述を他の部分から分離できる機構が重要であることを主張し，さらにその記述をプログラマがそれぞれのデザインパターンごとに特化した言語を使って行なえるようにした処理系の設計を提案する．

1 はじめに

オブジェクト指向のデザインパターンの発展とともにデザインパターンを利用したプログラミングを支援する言語機構が望まれている．我々はこれまでデザインパターンごとにそのような仕組みを容易に構築できるような処理系として，OpenJava を開発てきた [6] [7]．OpenJava では，メタクラスと呼ばれる一種のマクロを定義してクラスの記述方式を拡張する．OpenJava のマクロではクラスメタオブジェクトと呼ぶクラスの論理構造を表現するデータ構造を扱うため，オブジェクト指向プログラムに典型的な高度なソースコードの変更を記述するのに適している．また，拡張性を制限した簡潔な構文拡張の枠組とそのためのクラスライブラリを備えているために簡易的なクラスの構文拡張を容易に適用できる．デザインパターンはクラスやメソッドといったオブジェクト指向の理論的枠組に基づいており，OpenJava はプログラミングを支援する仕組みを構築する上でかなり有効であった．一方で，ある種のデザインパターンの使用を支援する場合に，そのデザインパターンに関連した記述がそれぞれのクラスに散ってしまう問題があることがわかってきた．

近年，プログラムを関心のある側面ごとに分割して記述できるようにしようという研究がなされてきている [3]．我々は， OpenJava での経験を踏まえ，この「関心ごとの分割」の指針を採り入れることにより，プログラマがそれぞれのパターンごとに特化した言語を使って，各デザインパターンに関連した

記述と各クラスに固有の記述に分割してプログラムを記述できるような処理系の必要性を提案する．このような処理系では，専用言語で分割して記述できるため，コーディングコストの軽減と視認性の向上が望める．さらに専用言語と分割の方法をデザインパターンや支援の程度に応じて，プログラマ自由に作成することができるので，高い汎用性が望める．

2 OpenJava

現在，デザインパターンの有用性が注目される一方で，実装時に生じる問題点が指摘されており [5] [1]，デザインパターンを利用したプログラミングを支援して，ソースコードの視認性を向上し，コーディングコストを軽減する言語機構が望まれている．我々はこれまで各デザインパターンごとにそのような仕組みを容易に構築できるような言語処理系を目指し，オブジェクト指向言語である Java にクラスを基盤とする言語拡張機構を追加した OpenJava を開発し，その有用性を示してきた [6] [7]．しかしながら，ある種のデザインパターンの使用を支援するためには十分でない点があることもわかってきた．

デザインパターンに沿ったプログラミング

ここで，デザインパターンを利用したファイルシステムのプログラミング例を考える．ファイルシステムの構成要素を表すために，抽象クラスとして Node，その具象クラスとして File，Directory，Link があり，それぞれ，ファイル，ディレクトリ，シンボリックリンクを表している．これらの要素への複数の操作を画一的に実現するためには VISITOR パターン [2]が使われる．パターンで定められた Visitor となる新たな抽象クラス NodeVisitor を作り， File などの各要素ごとの操作のためのメソッドを持たせる．また，Element となる Node には，NodeVisitor を受け取る抽象メソッド `accept()` を追加する．

この際，プログラマは，File などの全ての具象クラスに `accept()` を実装しなければならない．また，操作の種類ごとに具象クラスを NodeVisitor のサブクラスとして作ることになるが，継承されオーバーライドされるメソッドは 図 1 のように標準の操作として何もしないように定めることがしばしばある．これらについては，ほとんど単調なコードの繰り返しであり，自動的に実装されるような支援が望まれるところである．

```
abstract class NodeVisitor
{
   void visit(Node n) {} //default
   void visit(File n) { visit((Node)n); }
   void visit(Directory n) { visit((Node)n); }
   void visit(Link n) { visit((Node)n); }
}
```

図 1　NodeVisitor の実装例

OpenJava の言語拡張機構

OpenJava では，メタクラスと呼ばれる一種のマクロを定義してクラスの記述方式を拡張することができる．例えば， VISITOR パターンの利用を支援するメタクラス VisitorPattern を用いると，上の例のプログラムは図 2のようになる．Node の各具象クラスに散らばるメソッド `accept()` や，NodeVisitor の全てのメソッドはメタクラスによって自動的に実装されるので，プログラム中に明示する必要はない．

```
abstract class Node
   instatantiates VisitorPattern
   accepts NodeVisitor via accept
{
   ....
   void accept();
}

abstract class NodeVisitor
   instatantiates VisitorPattern
   visits Node, File, Directory, Link via visit
{}
```

図 2　OpenJava による VISITOR パターン使用例

プログラム中の `instantiates` から始まる句はマクロ VisitorPattern をクラスに適用することを指示する OpenJava の構文である．そして，`accepts` や `visits` から始まる句はメタクラス VisitorPattern によって追加された構文で，VISITOR パターンに特有な記述をする．マクロはこの構文による記述に基づいて，クラスを調べながら適切なメソッドの挿入などを行なう．

OpenJava のマクロ機構はクラスメタオブジェクトと呼ぶクラスの論理構造を表現するデータ構造を提供するので，構文木を表現するデータ構造を提供

するLisp などの従来型のマクロに較べ，オブジェクト指向プログラムに典型的な高度なソースコードの操作を記述することが容易である．また，拡張性を制限した簡潔な構文拡張の枠組と再利用性の高い構文定義のクラスライブラリを備えているために簡易的なクラスの構文拡張を容易に実現できる．

デザインパターンはクラスやメソッドといったオブジェクト指向の理論的枠組に基づいているために，プログラミングを支援する仕組み（マクロ）を構築する上で OpenJava はかなり有効であった．

問題点

一方，ある種のデザインパターンの使用を支援する仕組みのためにはOpenJava の拡張機構には欠点があることもわかってきた．具体的には，パターンのデザインの中で重要な役割をするクラスが２つ特に３つ以上である場合であり，例えば BRIDGE や VISITOR といったパターンの使用を支援する仕組みを実現しようとするとその問題点が見えてくる．これは，OpenJava がクラスを基盤とした拡張機構を提供していることに起因している．

一般的にはデザインパターンは複数のクラスによって構成される．ある複数のクラスで構成されるデザインパターンを使ったプログラミングを支援する場合，OpenJava では，クラスごとに拡張構文を用いて，そのクラスがデザインパターンで示されるどのクラス（役割）にあたり，他のクラスとどう関係するかを明示させる．例えば，図 2 の Node は `visits` 句により，このクラスは VISITOR パターンの Element にあたり，Visitor にあたる NodeVisitor をメソッド `accept()` を通して受け入れることを明示している．

Soukup は，デザインパターンに沿ったプログラミング時にソースコードからパターンが失われて視認性を悪くしているという問題点を指摘している [5]．OpenJava を用いた支援でも，各クラスはそれぞれの役割を明示しているだけであり，デザインパターンの全体構造を明示していないので，視認性の問題点を完全には解決していない．また，クラス間の関連性をもたせるために，１つのメタクラスでまったく異なる役割のクラスを扱っているが，あまり直観的とはいえず，メタクラスの記述を複雑にしている．

上の問題点は，デザインパターンに関連したプログラム中の記述を一箇所にまとめて書けるような処理系の必要性を示唆している．これは，クラスを基盤とした言語拡張機構で実現することは困難である．

3 側面（アスペクト）指向の導入

近年，プログラムを関心のある側面（アスペクト）ごとに分割して記述できるようにしようという研究がなされてきており [3]， Lopes らにより側面ごとに分割して記述されたプログラムを自動的に統合する処理系が開発されつつある [4]．我々は，この「関心ごとの分割」の指針を採り入れることにより，前節で述べた OpenJava の問題点を解決しようとしている．以下では，現在我々が開発中の SpawnJ と呼ぶ処理系を用いて，その基本的な構想を説明する．

SpawnJ

SpwanJ は OpenJava 同様，デザインパターンの使用を支援する仕組みを，プログラマが容易に実現することができる Java 言語の処理系である．しかし OpenJava と異なり，SpwanJ を使って実現された支援の仕組みは，ユーザがそのデザインパターンに関連した記述を一箇所にまとめて書くことを可能にする．これにより，プログラムの中でデザインパターンがどのように使われているかを明示することができ，視認性の問題を解決できる．

```
abstract class Node
{
   ....
}

aspect FileSystemVisitor
   from VisitorPatternAspect
{
   element Node;
   concreteElement File, Directory, Link;
   acceptance void accept(NodeVisitor);
   visitor NodeVisitor;
   visitation void visit(..);
}
```

図3　SpawnJ による VISITOR パターン使用例

前節の例題を SpawnJ で記述すると図 3 のようになる．図 2 と異なり，クラス Node の定義から `instantiates` や `accepts` などの句が消え，アスペクト FileSystemVisitor にまとめられて，視認性

が改善されている．NodeVisitorは，完全に自動定義されるのでもはや不要である．

OpenJava でメタクラスと呼んでいた，ある種のマクロ定義は，SpwanJ ではメタアスペクトと呼ばれる．これは，アスペクトを定義するための構文を規定し，さらにアスペクトをプログラムの他の部分にどのように統合するかを与える．上の例では，VisitorPatternAspect がメタアスペクトであり，element や concreteElement などの構文を規定し，この構文で記述されたアスペクトの情報をもとに，メソッド accept() を正しい位置に挿入したり，NodeVisitor を定義することを指示する．

メタアスペクトの記述は，OpenJava と同様，制限の強い簡易な再帰的下向き構文解析器やクラスメタオブジェクトを使って，容易におこなえる．OpenJava で開発された技術のうち有用であったものは，SpwanJ でもそのまま継承する．しかしながら，OpenJava では，デザインパターンに関連する情報を，拡張構文を使って，複数のクラス定義の中に分散して埋め込まなければならなかった．このような問題は SpawnJ では側指向を導入することで解決している．

複数のアスペクトの統合

側面指向プログラミングを支援する処理系において，複数に分割して記述されたアスペクトをいかに統合するかは最も重要な課題の１つである．処理系は統合時の衝突を適切に解決する機構を提供しなければならない．

統合はプログラマの意図によるところが大きいため，単純に統合を自動化することは非常に困難である．そこで， SpawnJ では，プログラマにアスペクトの統合の仕方を指示させ，その指示に従って処理系が複数の側面プログラムを自動的に統合する機構を用意することを考えている．

例えば，図 4 は，アスペクト FileSystemVisitor の統合結果に基づいてアスペクト SymbolicLinkProxy を統合することを明示しており，そのために，処理系標準の構文 after を用意している．

より高度で細粒度の統合方法をメタアスペクトが指示できるような設計のほうが汎用性が高いが，これは今後の課題である．

```
aspect SymbolicLinkProxy
   from ProxyPatternAspect
   after FileSystemVisitor
{
   ....
}
```

図 4　標準の機構を利用した合成方式の明示例

4　結論

本稿では，まず，デザインパターンに沿ったプログラミングを支援する機構として，我々の開発した OpenJava を取り上げ，クラスメタオブジェクトによる高度なマクロ機構の利点を示すとともに，クラスを基盤とした言語拡張機構の問題点を指摘した．次に，問題点の解決のためにプログラムをデザインパターンに関連した部分とそれ以外にの観点から分離して記述できる機構の必要性を議論した．その上で，それぞれのデザインパターンごとに特化した言語で側面プログラムを分離して記述する処理系の設計を，特に Java を基盤とする SpawnJ として提案した．現在，我々は SpawnJ の処理系の設計をおこなっているところである．今後は，側面プログラムの統合時の衝突の回避問題に取り組むとともに，SpawnJ の実装をおこなっていく予定である．

参考文献

[1] Jan Bosch : Design Patterns as Language Constructs, In *Journal of Object Oriented Programming*, SIGS Publications, 1997.

[2] E. Gamma, R. Helm, R. Johnson, and J.O. Vlissides : *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier and J. Irwin : Aspect-oriented programming. In *Proceedings of ECOOP'97, Lecture Notes in Computer Science, Volume 1241*, pp.220-242, Springer, 1997.

[4] C. Lopes and G. Kiczales : Recent developments in AspecJ, In *Workshop on Aspect-Oriented Programming at ECOOP'98*, 1998.

[5] Jiri Soukup : Implementing patterns, In *Patterns Languages of Program Design*, pp.395-412, Addison-Wesley, 1995.

[6] M. Tatsubori and S. Chiba : Programming Support of Design Patterns with Compile-time Reflection, In *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.

[7] Michiaki Tatsubori : *An Extension Mechanism for the Java Language*, Master of Engineering Dissertation at University of Tsukuba, University of Tsukuba, 1999.