

# MixJuice 言語によるデザインパターンの改善

田中 哲<sup>†</sup> — 杉裕志<sup>†</sup>

本論文は差分ベースモジュールによるデザインパターンの改善を論じる。デザインパターンはオブジェクト指向言語において高い拡張性を実現するための設計集である。しかし、既存のオブジェクト指向言語のモジュール機構の制限によって拡張性が制限される問題もあり、パターンカタログにはその問題が明示的に述べられている。本論文では、Java をベースとして差分ベースモジュールをサポートした MixJuice という言語でデザインパターンを書き直すとさまざまな問題が除去でき、より拡張性の高い設計が可能であることを示す。差分ベースモジュールでは既存のソースコードを修正せずにクラスを変更できる。これにより、デザインパターンに 5 種類の改善が行われる。1) 既存のモジュールが定義したクラスにインタフェースを追加するなどにより新しいデザインパターンに参加させることができるようになる。2) 既存のクラス階層のスーパークラス部分に抽象メソッドを追加するなどにより従来はできなかった拡張性が生まれる。3) 1 つのクラスの複数の観点を複数のモジュールに分割して定義することにより各観点ごとのスコープを実現し、より適切に情報隠蔽を行える。4) 動的な拡張性が必要な場合、リンク時のモジュール選択によりサブクラスを使わずに拡張性を実現できる。これにより、概念的に 1 つのクラスを実際に 1 つのクラスにすることができ、不要なダウンキャストを排除でき型安全性が高まる。5) リンク時の機能選択によりクラス数を減少させ動的な構造を単純にできる。本論文ではレイヤードクラス図により差分ベースモジュールによるデザインパターンの構造を図示し、これらの利点を既存のデザインパターンカタログとの比較により示す。

## Design Pattern Improvement by MixJuice Language

AKIRA TANAKA<sup>†</sup> and YUUI ICHISUGI<sup>†</sup>

This paper presents benefits of the difference based modules with design patterns. The design patterns are a catalog of designs to achieve extensible architecture in object oriented languages. But the catalog also points out several problems which is caused by the limitation of the module system of the languages. The paper presents the problems can be fixed by MixJuice which is a Java-based language with the difference based modules. The difference based modules makes it possible to modify existing classes without modifying existing source code. The design patterns can be improved in 5 ways by the class modification. 1) An existing class can be participant to a design pattern by adding an interface. 2) Adding an abstract method in the superclass of existing class hierarchy makes a pattern more extensible. 3) Several concerns of a class can be hide their information each other by implement the class by separated modules. 4) When run-time extensibility is not required, link-time extensibility by module selection can be used. It makes a system more type safe by avoiding useless downcast. 5) The link-time module selection also reduces number of classes and simplify a system. The paper presents layered class diagrams of design patterns for MixJuice and compare them to patterns for existing languages.

### 1. はじめに

デザインパターンはオブジェクト指向において高い拡張性を実現するための設計集であり、「Design Patterns」<sup>7)</sup> という書籍(以下、GoF 本 と略す)には 23 種類のパターンが載っている。ここで、各パターンにはどのように拡張性が高まるかということとともに、

それによる限界も述べられている。たとえば、Visitor パターンはオペレーションを追加できるという拡張性を実現するが、同時にオブジェクト構造を拡張することが困難であることも述べられている。

また、デザインパターンは言語に依存する<sup>9)</sup>。これは GoF 本でも次のように指摘されている。

プログラミング言語の選択は重要である。

<sup>†</sup> 産業技術総合研究所

National Institute of Advanced Industrial Science and Technology (AIST)

GoF は Gang of Four の略であり、Design Patterns の 4 人の著者を示している。

なぜなら、どの言語を使うかによってどのような観点でデザインパターンをまとめるかが違って来るからである。我々のパターンは Smalltalk/C++ レベルの言語形態を想定している。その選択によって、容易に実現できることとできないことが決まる。たとえば、もし、我々が手続き型言語を想定していれば、Inheritance(継承), Encapsulation(カプセル化), Polymorphism(ポリモルフィズム)といったデザインパターンを組み入れたであろう。また、我々のパターンの中には、あまり一般的でない言語によって直接サポートされているものもある。たとえば、CLOS はマルチメソッドを有しているので、Visitor パターン (p.353) のようなパターンは必要性がなくなるのである。

(p.16, l.3-11)

本論文は、我々の提案している MixJuice<sup>12)</sup> という言語によってデザインパターンにどのような改善が可能かを論じる。具体的には、パターンの限界が MixJuice によって除去され、GoF 本のパターンよりもさらに高い拡張性が実現できることを述べる。

また、改善点を解析し、それを実現するために必要な言語機構についても述べる。必要な言語機構はモジュール機構に関するものであり、Java のようなクラスベースモジュール(クラスを単位としたモジュール機構)に対する MixJuice の差分ベースモジュール<sup>13)</sup> の利点を述べる。

我々の長期的な研究目的は、拡張性の高いプログラムを構成可能な言語ないしモジュール機構を提案することにある。ここで、そのようなモジュール機構は、実際のプログラムで頻繁に使われるパターンを適切に支援する必要がある。つまり、MixJuice はデザインパターンを適切に支援するべきである。そのことを評価するのが MixJuice の側面から見たこの研究の意図であり、そのために GoF 本に載っている 23 種のパターンすべてについて、MixJuice による改善点をまとめたカタログを構成した。このカタログは <http://cvs.m17n.org/~akr/mj/design-pattern/> で公開している。本論文ではそのカタログの中の Abstract Factory パターンと Visitor パターンに関する詳細を述べる。

本論文は「Design Patterns<sup>7)</sup>」の日本語版の「デザインパターン<sup>8)</sup>」から引用を行っている。引用部には GoF 本からの引用であることを示してある。

本論文の構成は次のとおりである。2 章で MixJuice と差分ベースモジュールについて述べ、3 章で

MixJuice プログラムの図式表現であるレイヤードクラス図について述べる。4 章でデザインパターンの改善に使用する基本的なプログラミング技法を述べ、5 章で Abstract Factory パターン、6 章で Visitor パターンの改善を述べる。7 章で GoF 本の 23 種のパターンの改善点を概観し、8 章で改善に必要とされる言語機構について述べる。9 章で関連研究について述べ、10 章でまとめを述べる。

## 2. MixJuice と差分ベースモジュール

MixJuice は Java のモジュール機構を差分ベースモジュールに入れ換えた言語である。

差分ベースモジュールは以下の原理によって設計されたモジュール機構である。

差分定義の原理：モジュールは、オリジナルのプログラムと拡張後のプログラムとの間の差分である。ここで差分とは、新たな名前の定義の追加と、既存の名前の定義の修正の集合である。

モジュールは、再利用の単位であり、情報隠蔽の単位であり、分割コンパイルの単位である。複数のモジュールをリンクすることによって、実行可能なアプリケーションが構築される。差分ベースモジュールにおけるリンクとは、「いっさいの定義を持たない空のプログラム」に対して各モジュールで定義される差分を順番に追加していくことを意味する。

MixJuice における差分は、具体的には次のものの集合である。

- 新たなクラス・インタフェースの導入
- 他のモジュールが定義したクラスに対する新たなスーパーインタフェース・フィールド・コンストラクタ・メソッドの追加
- 他のモジュールが定義したインタフェースに対する新たなスーパーインタフェース・アブストラクトメソッド・ファイナルフィールドの追加
- 他のモジュールが定義したコンストラクタ・メソッドのオーバライド

MixJuice のプログラムはモジュールの集合であり、図 1 のように記述する。図 1 では、モジュール m1 とモジュール m2 を定義している。

モジュール定義の先頭の “extends” 宣言は、そのモジュールが差分を追加する対象となるモジュール名を宣言する。従来のオブジェクト指向言語におけるスーパークラスに似ているため、指定されたモジュール

ルをスーパーモジュールと呼ぶ。図 1 の例では、m2 は、m1 をスーパーモジュールとして指定している。またこのとき「m2 は m1 のサブモジュールである」、あるいは「モジュール m2 は m1 を継承する」という。なお、この関係にサイクルが存在してはならない。

モジュール m1 のように、“extends” 宣言を持たないものは「空のプログラム」に対する差分を定義するモジュールである。

モジュール本体(中かっこの内部)には、オリジナルのプログラムに対する差分を記述する。図 1 では、モジュール m2 は、モジュール m1 で定義されたクラス S のメソッド foo と、クラス A のメソッド foo をそれぞれオーバーライドして機能を拡張している。つまり、m2、m1 をリンクしたプログラムは図 2 の Java プログラムとほぼ等価な意味を持つ。

### 3. レイヤーダクラス図

レイヤーダクラス図(Layered Class Diagram)は MixJuice のプログラムを図示する記法であり、モジュールの組合せによる拡張を表現する。この記法は UML のクラス図の拡張となっており、クラス図で描けることはそのまま描くことが可能である。また、ソースコードから機械生成することも可能である。

レイヤーダクラス図はクラスを縦に並べて描いた UML のクラス図をモジュールとし、そのモジュールを横に並べることによってプログラムを表現する。つまり、水平方向にモジュールが並び、各モジュールの中には垂直方向にクラスが並び、具体的には次のような規則で図 3 のように記述する。

```
module m1 {
  define class S {
    define int foo() { return 1; }
  }
  define class A extends S {
    int foo() { return original() + 10; }
  }
}
module m2 extends m1 {
  class S {int foo(){return original()+2;}}
  class A {int foo(){return original()+20;}}
}
```

図 1 モジュール定義の構文  
Fig.1 Module Definition Syntax.

```
class $S1$ { int foo() { return 1; } }
class S extends $S1$ { int foo() { return super.foo() + 2; } }
class $A1$ extends S { int foo() { return super.foo() + 10; } }
class A extends $A1$ { int foo() { return super.foo() + 20; } }
```

図 2 図 1 の m2 とほぼ等価な Java プログラム  
Fig.2 Java program similar to m2 in Fig. 1.

- (1) モジュールは長方形で表し、クラス定義・拡張を囲む。
- (2) モジュールは左から右に継承関係(依存関係)に従って並べる。
- (3) クラスは上から下に継承関係に従って並べる。
- (4) クラス定義は太線の長方形で描く。
- (5) クラス拡張は対応するクラス定義と同一の垂直位置に長方形で描き、水平の点線で結ぶ。
- (6) 継承・関連などの線はそれを行っている長方形に接続する。

たとえば、図 1 のプログラムはレイヤーダクラス図として表現すると図 4 となる。

### 4. MixJuice のプログラミング技法

本章ではデザインパターンの改善に使用する MixJuice のプログラミング技法について述べる。

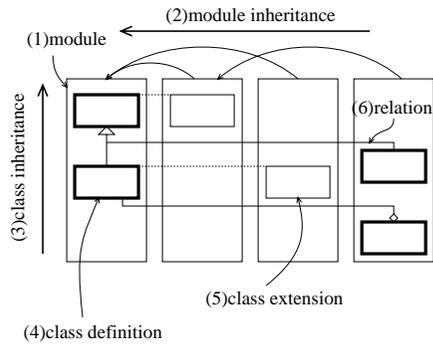


図 3 レイヤーダクラス図  
Fig.3 Layered Class Diagram.

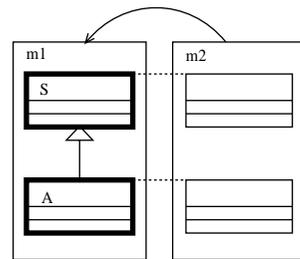


図 4 図 1 のプログラムのレイヤーダクラス図  
Fig.4 Layered Class Diagram for Fig. 1.

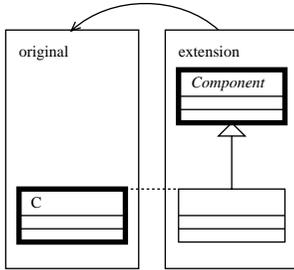


図 5 スーパーインタフェース追加のレイヤードクラス図  
Fig. 5 Layered Class Diagram for Super-interface Addition.

#### 4.1 スーパーインタフェース追加

他のモジュールが定義したクラスに対して、新たなスーパーインタフェースを追加できる。スーパーインタフェースの追加によって、クラスをそのスーパーインタフェースのサブタイプにできる。つまり、パターンにおける型的な制約を後から他のモジュールが満たすことができ、クラスの定義時には想定されていなかったパターンの構成要素にできる。たとえば、クラスが Composite パターンの木構造の一部になるためにはそのクラスが Component のサブタイプでなければならない。したがって、既存のクラスが Component のサブタイプでなければ、Java では木構造の一部にすることはできない。しかし、MixJuice では、スーパーインタフェース追加により、クラスを後から Component のサブタイプにすることができ、Composite パターンに参加させることができる。なお、GoF 本のパターンでは Component はクラスであるが、スーパーインタフェース追加を行うためにはインタフェースとする必要がある。このようなスーパーインタフェース追加は次のようなコードで記述でき、レイヤードクラス図では図 5 のようになる。

```
module original {
  define class C {...}
}
module extension extends original {
  define interface Component {...}
  class C implements Component {...}
}
```

#### 4.2 フィールド追加

他のモジュールが定義したクラスに対して、新たなフィールドを追加できる。これにより、既存のクラスに付加的な情報を保持させることができる。この情報を使って、デザインパターン内の関係を表現できる。たとえば、Observer パターンでは Subject から Observer への 1 対多の関係があり、この関係をどこか

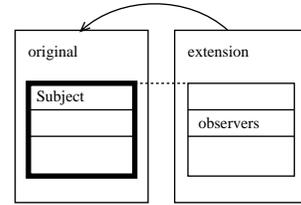


図 6 フィールド追加のレイヤードクラス図  
Fig. 6 Layered Class Diagram for Field Addition.

で実装する必要がある。通常、あるクラスが最初から Subject の役割を持っていることが想定されている場合には、この関係はそのクラスのフィールドとして実装される。しかし、そのような想定がされておらず、そのクラスに後から Subject の役割を負わせる場合には Java ではフィールドを追加できないため他の方法をとらざるをえない。しかし、MixJuice では次のようにフィールド追加を行うことにより通常どおりの実装を行うことができる。レイヤードクラス図では図 6 のようになる。

```
module original {
  define class Subject {...}
}
module extension extends original {
  class Subject {
    Vector observers;
  }
}
```

#### 4.3 メソッド追加

他のモジュールが定義したクラスにメソッドを追加し、クラスのシグネチャを拡張できる。ここで、追加対象のクラスにサブクラスが存在した場合、サブクラスのシグネチャも同様に拡張される。

追加するメソッドが抽象メソッドであった場合、アプリケーションを実行可能とするためには、そのクラス以下のすべての具象クラスに対してそのメソッドの実装を別途追加する必要がある。ここで、抽象メソッドを追加するモジュールとは別のモジュールがそのような具象クラスを追加する可能性がある。その場合、メソッドの実装は抽象メソッドを追加したモジュールと具象クラスを追加したモジュールの両方を継承する補完モジュール<sup>13)</sup>によって行われる。あるモジュールが補完モジュールであることを示すためには、モジュール継承を extends のかわりに complements を使って記述する。なお、追加するメソッドが抽象メソッドでなかった場合に

クラスのシグネチャとは、クラスが提供する API であり、メソッドのシグネチャの集合である。メソッドのシグネチャはメソッドの名前と型の対である。

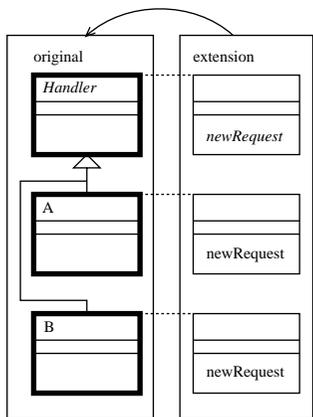


図 7 メソッド追加のレイヤードクラス図

Fig. 7 Layered Class Diagram for Method Addition.

は、アプリケーションが実行不能になることはない。

たとえば Chain of Responsibility パターンにおいて、要求を処理する Handler 以下のクラス階層に対してメソッドを追加することにより、要求の種類を増やすことができる。このようなメソッド追加は次のようなコードで記述でき、レイヤードクラス図では図 7 のようになる。

```

module original {
  define abstract class Handler {...}
  define class A extends Handler {...}
  define class B extends Handler {...}
}
module extension extends original {
  class Handler {
    define abstract void newRequest();
  }
  class A { void newRequest() {...} }
  class B { void newRequest() {...} }
}
    
```

#### 4.4 メソッド 拡張

他のモジュールが定義したメソッドを、オーバーライドできる。これにより、既存のメソッドに処理を割り込ませることができる。このような既存のメソッドに対する処理の割り込みにより、そのメソッドを使っているクライアントのソースコードを編集せずに機能を増やすことができる。たとえば、Observer パターンではイベントの発生に付随して通知を行うが、イベントの発生を検知するのにメソッド 拡張を使用できる。このようなメソッド 拡張は次のようなコードで記述でき、レイヤードクラス図では図 8 のようになる。

```

module original {
  define class Subject {
    define void m() {...}
  }
}
    
```

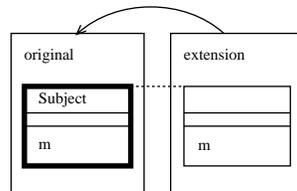


図 8 メソッド 拡張のレイヤードクラス図

Fig. 8 Layered Class Diagram for Method Extension.

```

module extension extends original {
  class Subject {
    void m() { original(); イベント通知 }
  }
}
    
```

#### 4.5 名前空間分離

名前を定義する側は、提供する名前の集合を任意にグループ化できる。名前を利用する側は、グループ化された名前の集合を任意に選んで利用できる。なお、ここでいう名前はクラス名だけでなくメソッド名・フィールド名なども含む。

名前空間分離により、複数のクラスが協調して動作する場合にクラスを横断した適切な名前空間を設定できる。たとえば、Memento パターンにおいて Memento 内の情報を適切に情報隠蔽できる。このような名前空間分離は次のようなコードで記述でき、レイヤードクラス図では図 9 のようになる。

```

module m {
  define class C { ... }
  define class Memento {}
}
module m.implementation extends m {
  class C {
    // Memento の state を参照可能
    ...
  }
  class Memento {
    // m.implementation をモジュール継承し
    // ないモジュールからは state は不可視
    int state;
  }
}
    
```

#### 4.6 仕様モジュールと実装モジュールの分離

これは名前空間分離の特殊形である。クラスを、公開仕様を定義するモジュールと、実装を定義するモジュールに分離する。これにより、Java の protected と同様に次の 2 種類の仕様を分離できる。

- 公開仕様のみ依存するユーザ向けの仕様
- 拡張を行うために実装に依存するユーザ向けの仕様

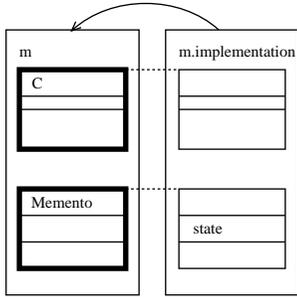


図 9 名前空間分離のレイヤードクラス図  
 Fig. 9 Layered Class Diagram for Namespace Separation.

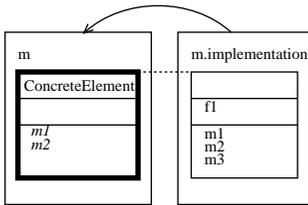


図 10 仕様モジュールと実装モジュールの分離のレイヤードクラス図  
 Fig. 10 Layered Class Diagram for Separation of Specifications and Implementations.

具体的には、公開する名前だけを定義するモジュールと、それ以外の名前を定義するモジュールに分けることによってこれを実現する。ここで、MixJuice の名前空間はクラスとは独立しているため、クラスを継承するかどうかとは独立に実装に依存するかどうかを選択できる。たとえば、Visitor パターンのオブジェクト構造の内部に特定のオペレーションだけに必要な情報があっても適切な情報隠蔽を行える。このことは 6.3 節で詳しく述べる。このような仕様モジュールと実装モジュールの分離は次のようなコードで記述でき、レイヤードクラス図では図 10 のようになる。

```

module m {
  define class ConcreteElement {
    // public methods
    define abstract void m1();
    define abstract void m2();
  }
}
module m.implementation extends m {
  class ConcreteElement {
    // impl. of public methods
    void m1() {...}
    void m2() {...}
    // protected fields and methods
    int f1;
    define void m3() {...}
  }
}

```

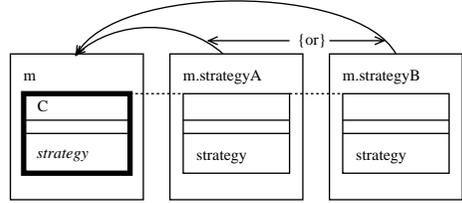


図 11 実装モジュール選択のレイヤードクラス図  
 Fig. 11 Layered Class Diagram for Implementation Selection.

#### 4.7 実装モジュール選択

あるクラスに対する複数種類の実装をそれぞれモジュールとして用意しておき、リンク時にモジュールを選択できる。これにより、静的に機能が選択可能な場合には、サブクラスを不要にできる。たとえば、Strategy パターンにおいて戦略がリンク時に決まるならば、実装モジュール選択によって戦略の実装を選択することにより、クラス構造を単純化できる。このような実装モジュール選択は次のようなコードで記述でき、レイヤードクラス図では図 11 のようになる。なお、レイヤードクラス図では、実装モジュール選択の選択肢を表現するために、モジュール継承の矢印に {or} という印を付加する。

```

module m {
  define class C { // non-abstract
    define abstract void strategy();
  }
}
module m.strategyA extends m {
  class C { void strategy() {...} }
}
module m.strategyB extends m {
  class C { void strategy() {...} }
}

```

### 5. Abstract Factory パターン

本章と次章では Abstract Factory パターンと Visitor パターンをとりあげて目的・構造・問題点および MixJuice による改善策を述べる。23 種のパターンの中からこれらを選んで取り上げるのは、これらの中に MixJuice による特徴的な改善技法がほぼ含まれているためである。

Abstract Factory パターンの問題点は 2 種類あり、2 種類の改善策を述べる。最初の改善策は GoF 本の

リンク時には提供されている実装のうち、ちょうど 1 つの実装モジュールだけをリンクするように注意する必要がある。現在のリンクの実装では、2 つ以上の異なる実装モジュールが指定されてもリンクエラーにはならない。

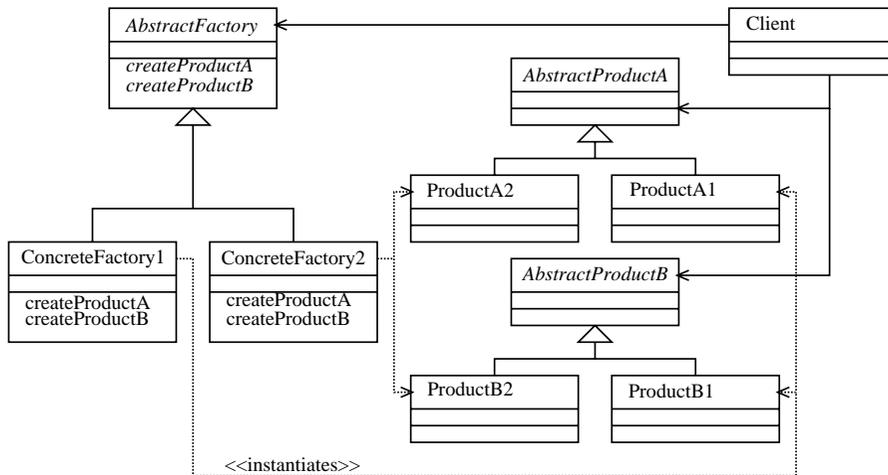


図 12 Abstract Factory パターンの構造  
Fig. 12 Structure of Abstract Factory Pattern.

パターンと同じクラス構造を使う方法で、最初の問題を解決する。もう 1 つの改善策は GoF 本のパターンとは異なるクラス構造を使う方法で、限定された状況でしか使用できないものの、両方の問題を解決する。

### 5.1 Abstract Factory パターンの目的と構造

GoF 本において Abstract Factory パターンの目的は次のように述べられている。

互いに関連したり依存し合うオブジェクト群を、その具象クラスを明確にせずに生成するためのインタフェースを提供する。

( p.95, l.3-4 )

つまり、生成すべきオブジェクト群が複数セットあり、他のオブジェクト群を後から導入できるという拡張性を実現するパターンである。GoF 本では例として Motif 用のオブジェクト群と Presentation Manager 用のオブジェクト群を選択できるウィンドウアプリケーションがあげられている。

この目的を実現するために、Abstract Factory パターンは図 12 のようなクラス構造を使用する。クライアントはまず選択したオブジェクト群に対応する ConcreteFactory のインスタンスを 1 つ生成し、それを經由してオブジェクトを生成する。この結果、最初に ConcreteFactory のインスタンスを生成するところ以外では AbstractFactory, AbstractProduct 以外のクラスを参照する必要がなく、個々のオブジェクト群の種類には依存しなくて済む。そのため、新しい ConcreteFactory を定義すれば、そのインスタンスを生成するコードを加えるだけで新しいオブジェクト群に対応できる。

### 5.2 Abstract Factory パターンの問題点

Abstract Factory パターンではオブジェクト群に含まれるオブジェクトの種類を後から増やすことが困難だという問題がある。このことは GoF 本に次のように述べられている(ここでいうインタフェースは Java のインタフェースではなくシグネチャを意味する)。

新たな種類の部品に対応することが困難である。新たな種類の部品に対応できるように Abstract Factory パターンを拡張することは容易ではない。なぜならば、AbstractFactory クラスのインタフェースは生成される部品の集合を固定しているからである。新たな種類の部品への対応は、インタフェースの修正が必要となるために、AbstractFactory クラスだけでなくそのすべてのサブクラスを修正しなければならない[実装]の節で、この問題に対する 1 つの解決策について議論する。

( p.98, l.10-15 )

つまり、AbstractFactory クラスに列挙したメソッドが部品を表すクラスに対応しているため、部品を増やすためには AbstractFactory クラスのソースコードを修正してメソッドを追加しなければならないという問題である。

なお、引用の最後に触れられている「1 つの解決策」とは、オブジェクトを生成するオペレーションに生成すべきクラスを指定するパラメータを付加するというものである。しかし、この方法を使うと戻り値のダウンキャストが必要になる。

また、ダウンキャストに関しては別の問題もある。

```

module original {
  define abstract class AbstractFactory {
    define abstract AbstractProductA createProductA();
    define abstract AbstractProductB createProductB();
  }
  define class ConcreteFactory1 extends AbstractFactory {
    AbstractProductA createProductA() {return new ProductA1();}
    AbstractProductB createProductB() {return new ProductB1();}
  }
  define class ConcreteFactory2 extends AbstractFactory {
    AbstractProductA createProductA() {return new ProductA2();}
    AbstractProductB createProductB() {return new ProductB2();}
  }

  define abstract class AbstractProductA {...}
  define class ProductA1 extends AbstractProductA {...}
  define class ProductA2 extends AbstractProductA {...}

  define abstract class AbstractProductB {...}
  define class ProductB1 extends AbstractProductB {...}
  define class ProductB2 extends AbstractProductB {...}

  define class Client {
    AbstractFactory factory;
    define void m1() {
      AbstractProductA a = factory.createProductA();
      AbstractProductB b = factory.createProductB();
    }
  }
}

```

図 13 Abstract Factory パターンの実装

Fig. 13 Implementation of Abstract Factory Pattern.

クライアントがサブクラスに特有のメソッドを呼び出すためにはダウンキャストが必要である。このことは GoF 本に次のように述べられている。

すなわち、すべての部品が、戻り値の型により与えられる同一の抽象化されたインタフェースを備えた形でクライアントに返される。したがって、クライアントは部品のクラスに関して区別をしたり、安全な仮定をおくことが不可能になる。もし、クライアントがサブクラスに特有のオペレーションを実行したくても、抽象クラスのインタフェースを通してでは、それらを実行することはできない。クライアントはダウンキャストを(たとえば C++ の `dynamic_cast` を使って)行うことができるかもしれないが、これはつねに実行可能、または安全であるとは限らない。なぜならば、ダウンキャストは失敗する可能性があるからである。(p.100, l.12-18)

さらに、クライアントではなく、個々の部品集合内部の実装におけるダウンキャストの問題もある。複数の部品を組み合わせる作業を行う場合、クライアントはある部品のメソッドを呼び出し、その引数に他の部品を渡すことになるが、受け取ったメソッドは引数として渡された部品をダウンキャストしなければならない。これは上記と同じ理由により、安全ではない。そ

して、GoF 本のウインドウシステムの例について考えても、ウインドウオブジェクトに背景画像を設定する、ウインドウの親子関係を変更するなど、複数の部品に関わる作業は数多く想像できる。たとえば、ウインドウを表すクラス( `Window`, `MotifWindow`, `PMWindow` )と画像を表すクラス( `Image`, `MotifImage`, `PMImage` )があり、`Window` の背景画像を設定するメソッドが引数として `Image` を受け取るとすると、`MotifWindow` のインスタンスに `PMImage` のインスタンスを渡すことは型的にエラーではない。つまり、これは正しくない操作であるにもかかわらず型的に許され、コンパイラはこの間違いを検出できない。実際に実行時にダウンキャストした時点で失敗することになる。

### 5.3 AbstractFactory クラスに対するメソッド追加による改善

MixJuice では `AbstractFactory` クラスのシグネチャを後から追加するモジュールで拡張できるので、クラスに対応するメソッドの列挙に他のモジュールがメソッドを追加することができ、既存のモジュールを修正せずに新しい部品を導入できる。この方法は GoF 本のクラス構造に対して直接適用でき、既存のパターン実装をモジュール分割する必要もないため、既存の Java プログラムをモノリシックなモジュールとして MixJuice に移植した場合でも使用できる。

```

module extension extends original {
  class AbstractFactory {
    define abstract AbstractProductC createProductC();
  }
  class ConcreteFactory1 {
    AbstractProductC createProductC() {return new ProductC1();}
  }
  class ConcreteFactory2 {
    AbstractProductC createProductC() {return new ProductC2();}
  }

  define abstract class AbstractProductC {...}
  define class ProductC1 extends AbstractProductC {...}
  define class ProductC2 extends AbstractProductC {...}

  class Client {
    define void m2() {
      AbstractProductC c = factory.createProductC();
    }
  }
}

```

図 14 Abstract Factory パターンの拡張

Fig. 14 Extension of Abstract Factory Pattern.

たとえば、図 13 のような Abstract Factory パターンの実装があったとする。問題は、AbstractFactory クラスで部品の集合 (AbstractProductA, AbstractProductB) がメソッドの集合 (createProductA, createProductB) として列挙されているため、Java では新しい部品を追加してもそれを AbstractFactory クラスのメソッド経由で生成できないということである。しかし、MixJuice では図 14 のようなモジュールにより、AbstractFactory クラスに新しいメソッド (createProductC) を追加できるため、新しい部品 (AbstractProductC) を問題なく導入できる。

図 13、図 14 のモジュールをリンクしたプログラムのレイヤードクラス図を図 15 に示す。

なお、新しい種類の部品を導入するには、その部品自体を実装するだけでなく、既存のすべての ConcreteFactory にその種類の部品を生成するメソッドを追加しなければならない。この追加は、4.3 節で述べたように補完モジュールを実装することによって行う。

まとめると、この改善策には次のような性質がある。

- 既存のソースコードを修正しなくても、新たな種類の部品の生成に対応することが可能になる。
- 既存のすべての ConcreteFactory に対し追加された部品を生成するメソッドを実装する補完モジュールが必要となる。
- 既存のデザインパターンの構造のままで利点が得られる。

#### 5.4 部品の実装選択による別解

どの部品集合を使うかがリンク時に決定できる場合、MixJuice では実装モジュール選択によりサブクラスを使わずに部品集合の実装を選択できる。これを使う

と、前節とは異なる方法で AbstractFactory パターンの問題を解決できる。

Abstract Factory パターンの部品集合がリンク時に決定できることは稀ではない。たとえば、GoF 本で述べられている例 (Motif と Presentation Manager の選択) においては、コンパイル対象のプラットフォームが決まった段階で使用する部品集合は決定され、リンク時に部品集合を選択することが可能である。一般に、プラットフォームに依存する機能の選択はリンク時に決定できる。

また、この方法ではダウンキャストによる型安全性の問題を解決できる。これはサブクラスが除去されるからである。ダウンキャストの問題はアプリケーションの中で概念上 1 種類しか存在しないものをサブクラスに分割したところに原因があり、概念をそのまま 1 種類のクラスで表現すればダウンキャストは不要にでき、型安全性の問題を解決できるのである。

MixJuice でこのような実装の選択を実現するには図 16 のように記述する。この記述では、AbstractFactory クラス以下のクラス階層は存在せず、GoF 本のパターンに比べてクラス数が減っている。この実装では、部品を表現するクラスの抽象的なシグネチャの定義 (product) と複数の実装の定義 (product.impl, product.impl2) をそれぞれモジュールとして表現している。これらのモジュールはリンク時の選択に応じてリンクされ、選択された機能を持つアプリケーションが生成される。ここで、シグネチャの定義ではアブストラクトコンストラクタ (ProductA(), ProductB()) を使用している。これは MixJuice の機能であり、コンストラクタのシグネチャだけを定義し、実装は与えないものである。このアブストラクトコン

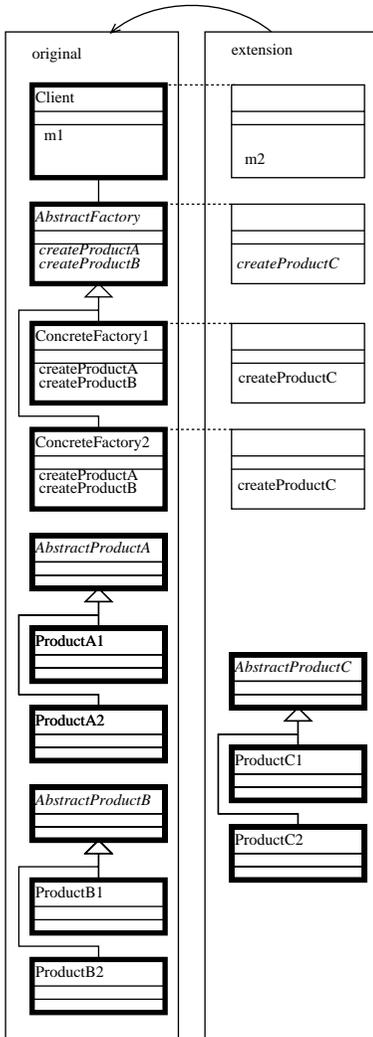


図 15 Abstract Factory の改善のレイヤードクラス図

Fig. 15 Layered Class Diagram for Improved Abstract Factory Pattern.

ストラクタにより、クライアントはどの実装が選ばれるか知らなくてもオブジェクトを生成できる。

ここで、Abstract Factory パターンが目的とする拡張性は「生成すべき部品集合を後から実装されるモジュールで導入できること」であるが、この目的は上記の MixJuice の実装でも達成できる。たとえば、既存のソースコードを変更しなくても図 17 のモジュール (product.imp3) を新しく実装すれば、部品集合を追加することになり、他の実装と同様にリンク時に選択できる。

また、Abstract Factory パターンの問題であった部品の追加も容易である。これは、AbstractFactory クラスが存在せず、部品をすべて列挙しなければならない部

```

module product {
  define class ProductA {
    define abstract ProductA();
    ...
  }
  define class ProductB {
    define abstract ProductB();
    ...
  }
}

module product.imp1 extends product {
  class ProductA {
    ProductA() {...}
    ...
  }
  class ProductB {
    ProductB() {...}
    ...
  }
}

module product.imp2 extends product {
  class ProductA {
    ProductA() {...}
    ...
  }
  class ProductB {
    ProductB() {...}
    ...
  }
}

module client extends product {
  ... new ProductA() ...
  ... new ProductB() ...
}

```

図 16 実装選択による Abstract Factory パターンの実装  
Fig. 16 Implementation of Abstract Factory Pattern using Implementation Selection.

```

module product.imp3 extends product {
  class ProductA {
    ProductA() {...}
    ...
  }
  class ProductB {
    ProductB() {...}
    ...
  }
}

```

図 17 実装選択による Abstract Factory パターンにおける実装の追加

Fig. 17 Implementation Addition for Abstract Factory Pattern using Implementation Selection.

分が存在しないためである。たとえば、図 18 のようなモジュールを記述することにより、部品 (ProductC) を追加できる。

なお、これらの 2 つの拡張を両方同時に使用した場合、補完モジュールが必要になる。この場合、補完モジュールでは、前者の拡張で導入した部品集合に属する、後者の拡張で導入した部品を実装しなければならない。具体的には図 18 の product.c.imp1 が product.imp1 で導入した部品集合に属する、product.c で導入した部品の実装である。

図 16、図 17、図 18 のモジュールをまとめたレイ

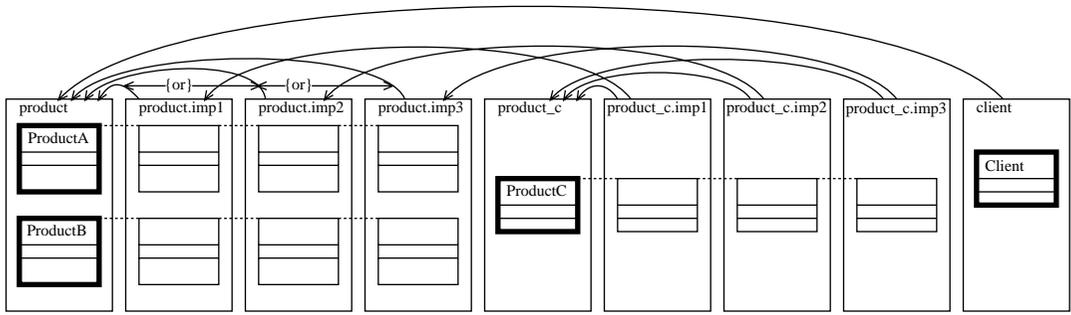


図 19 Abstract Factory の別解のレイヤードクラス図

Fig. 19 Layered Class Diagram for Another Solution of Abstract Factory Pattern.

```

module product_c {
  define class ProductC {
    define abstract ProductC();
    ...
  }
}

module product_c_imp1
complements product_c, product_imp1 {
  class ProductC {
    ProductC() {...}
    ...
  }
}

module product_c_imp2
complements product_c, product_imp2 {
  class ProductC {
    ProductC() {...}
    ...
  }
}

module product_c_imp3
complements product_c, product_imp3 {
  class ProductC {
    ProductC() {...}
    ...
  }
}
    
```

図 18 実装選択による Abstract Factory パターンにおける部品の追加

Fig. 18 Product Addition for Abstract Factory Pattern using Implementation Selection.

ヤードクラス図を図 19 に示す。

まとめると、この改善策には次のような性質がある。

- GoF パターンでは AbstractFactory クラスに生成対象の部品をメソッドとしてすべて列挙する必要があったが、そのような列挙を行うところがない。このため、部品のクラスを追加するだけで部品を増やすことができる。
- 各部品ごとにクラスが 1 つしかなく、サブクラスが存在しないため、ダウンキャストが不要になる。
- アブストラクトコンストラクタにより、抽象的な生成の方法が与えられるため、AbstractFactory クラスおよびそのサブクラスが不要になる。こ

の結果、クラス数が減少する。

- オブジェクト群と部品を同時に追加した場合、補完モジュールが必要になる。

## 6. Visitor パターン

本章では Visitor パターンの目的・問題点および MixJuice による改善策を述べる。問題点は 3 種類あり、それに対応した 3 種類の改善策を述べる。最初の 2 種類の改善策は GoF パターンと同じクラス構造を使い、最後の改善策は異なるクラス構造を使う方法である。

### 6.1 Visitor パターンの目的と構造

GoF 本において Visitor パターンの目的は次のように述べられている。

あるオブジェクト構造上の要素で実行されるオペレーションを表現する。Visitor パターンにより、オペレーションを加えるオブジェクトのクラスに変更を加えずに新しいオペレーションを定義できるようになる。  
( p.353, l.3-5 )

つまり、オブジェクト構造とオペレーションを分離し、新しいオペレーションを後から導入できるという拡張性を実現するパターンである。GoF 本では言語処理系において構文木に対するさまざまなオペレーション( 型検査、コード生成、プリティプリントなど ) を分離して定義する例が示されている。

この目的を実現するために、Visitor パターンは図 20 のようなクラス構造を使用する。クライアントはオペレーションを実行したいときにはそのオペレーションに対応する ConcreteVisitor のインスタンスを用意し、それを Element 以下のクラス階層からなるオブ

GoF 本のクラス構造では、Client クラスと Element クラスの間に ObjectStructure クラスが存在するが、説明を単純化するためにここでは省いてある。

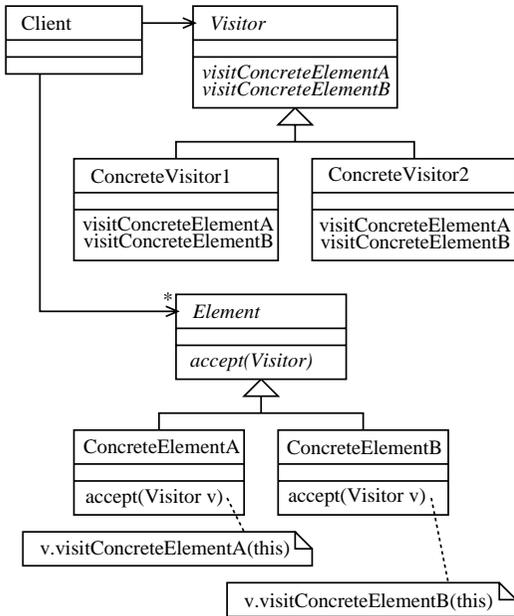


図 20 Visitor パターンの構造  
 Fig. 20 Structure of Visitor Pattern.

ジェクト構造の accept メソッドに渡す。ここで、ConcreteVisitor は後からいくつでも追加できるので、オブジェクト構造のソースコードの修正なしにオペレーションを追加できるという目的が達成できる。

6.2 Visitor パターンの問題点

Visitor パターンの問題点を次に 3 種類述べる。オブジェクト構造の情報隠蔽ができない。

このことは GoF 本に次のように述べられている。

カプセル化を破る。visitor によるアプローチでは、ConcreteElement クラスのインタフェースが、visitor が仕事を行うのに十分、強力であることを仮定している。その結果、このパターンでは要素の内部状態にアクセスする公開オペレーションを提供するように強いられることがしばしばある。したがって、カプセル化に対して妥協を与えることになるかもしれない。(p.359, l.5-8)

オブジェクト構造を拡張することができない。

このことは GoF 本に次のように述べられている。

新しい ConcreteElement クラスを加えることは難しい。Visitor パターンでは、Element の新しいサブクラスを加えることを難しくする。新しい ConcreteElement クラスを導入することにより、

Visitor クラスでは新しい抽象化されたオペレーションを宣言し、各 ConcreteVisitor クラスではそれに対応する実装を行わなければならない。デフォルトの実装を Visitor クラスに与えて、ほとんどの ConcreteVisitor クラスにこれを継承させることもときにははできるだろう。しかし、これは例外的な場合である。

(p.358, l.7-12)

Visitor クラスの存在はオブジェクト指向的ではない。オブジェクト指向パラダイムでは、クラスはデータとオペレーションの組合せを抽象化するものである。しかし、Visitor パターンはオペレーションを分離することを目的としており、Visitor クラスはオペレーションのみを表現するクラスである。つまり、Visitor クラスはオブジェクト指向的ではない。Visitor クラスは拡張性のために便宜的に導入されたクラスであり、オブジェクト指向という観点からは適切でなく、理解しにくい。また、上記の情報隠蔽できないという問題はクラスベースモジュールの名前空間ではオブジェクト指向的でないクラスをうまく扱えないということに起因する。

6.3 オブジェクト構造の仕様モジュールと実装モジュールの分離による改善

オブジェクト構造の情報隠蔽ができないという Visitor パターンの問題は MixJuice では仕様モジュールと実装モジュールの分離によって改善できる。

オブジェクト構造とそれに対するオペレーションを異なるクラスで表現しているにもかかわらず、名前空間がクラス単位であることがこの問題の直接的な原因である。クラス単位の名前空間ではオブジェクト構造の情報をオペレーションに公開するにはクラス外にすべて公開するしかないため、ある ConcreteVisitor だけに選択的に公開することなどができず、適切な情報隠蔽が行えない。しかし、MixJuice の名前空間はクラス単位ではなく、メソッド・フィールドを単位としてクラスを横断できるため、Visitor パターンのようなクラス構造でも外部に提供する情報を分割できる。

このような情報隠蔽は図 21 のようにして実現できる。ここではクラス定義を仕様モジュール (element, visitor) と実装モジュール (element.implementation, visitor.implementation) の 2 つに分割している。この分割により、名前空間が分離され、オブジェクト構造を利用する場合に仕様モジュールの名前のみを利用するか実装モジュールの名前も

```

module element {
  define abstract class Element {
    define abstract void accept(Visitor visitor);
  }
  define class ConcreteElementA extends Element {
    void accept(Visitor v) { v.visitConcreteElementA(this); }
  }
  define class ConcreteElementB extends Element {
    void accept(Visitor v) { v.visitConcreteElementB(this); }
  }
  define abstract class Visitor {
    define abstract void visitConcreteElementA(ConcreteElementA a);
    define abstract void visitConcreteElementB(ConcreteElementB a);
  }
}
module element.implementation extends element {
  class ConcreteElementA {
    int state;
  }
  class ConcreteElementB {
    int state;
  }
}
module visitor extends element {
  define class ConcreteVisitor extends Visitor {}
}
module visitor.implementation
  extends visitor, element.implementation {
  class ConcreteVisitor {
    void visitConcreteElementA(ConcreteElementA a) {
      ... int x = a.state; ...
    }
    void visitConcreteElementB(ConcreteElementB b) {
      ... int x = b.state; ...
    }
  }
}
}

```

図 21 名前空間分離による Visitor パターンの情報隠蔽の改善

Fig. 21 Improved Information Hiding of Visitor Pattern by Namespace Separation.

利用するかを利用する側が自由に選択できるようになる。図 21 では ConcreteVisitor を定義する visitor モジュールは element という仕様モジュールだけを利用し、ConcreteVisitor の具体的な実装である visitor.implementation モジュールは element.implementation という実装モジュールも利用している。レイヤードクラス図を図 22 に示す。

まとめると、この改善策には次のような性質がある。

- ConcreteElement の内部状態へのアクセス方法を、公開オペレーションとは分離できる。
- MixJuice による情報隠蔽は、情報を利用する側が利用するかどうかの選択権を持っており、通常の情報隠蔽とは異なる。このため、情報を隠蔽して完全に利用不能にすることはできない。

#### 6.4 Visitor に対するメソッド追加による改善

オブジェクト構造を拡張することができないという Visitor パターンの問題は MixJuice ではメソッド追加によって改善できる。この方法は、Abstract Factory パターンの 5.3 節の方法と同様に、既存の Java プログラムをモリシックなモジュールとして MixJuice に移植した場合でも使用できる。

この問題はオブジェクト構造の拡張に Visitor クラスのソースコードの修正が必要になるというものである。これはオブジェクト構造を表現する各クラスに対応するメソッドが Visitor クラスに列挙されているためである。

各クラスに対応するメソッドが列挙されているためにクラスを増やせないという問題は、Abstract Factory パターンと同じである。したがって、同じ改善策が有効である。つまり、MixJuice のメソッド追加の機能を使えば、この問題を解決できる。また、2 種類の拡張を同時に行ったときに補完モジュールが必要になるのも Abstract Factory パターンと同様である。ここで、Visitor パターンにおける 2 種類の拡張は、オブジェクト構造を表現するクラスの追加とオペレーションの追加である。

図 23 ではオブジェクト構造自身を定義する element モジュール、Visitor パターン (Visitor クラスや accept メソッド) を定義する visitor モジュール、オブジェクト構造を拡張する element.c モジュールが定義される。element.c モジュールでは、オブジェクト構造に ConcreteElementC を追

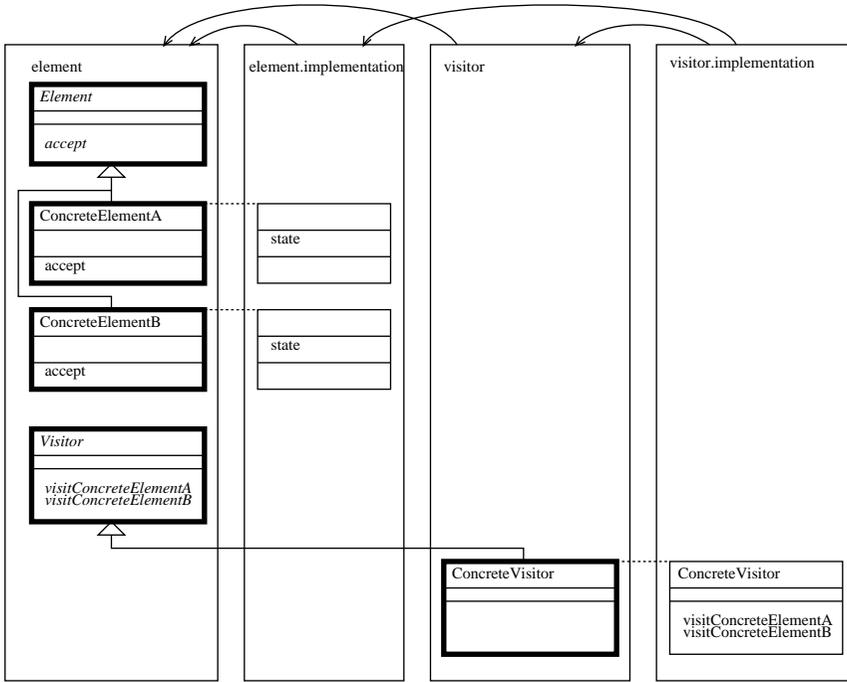


図 22 Visitor の名前空間分離のレイヤードクラス図

Fig. 22 Layered Class Diagram for Visitor Pattern using Namespace Separation.

加するという拡張を行い、同時に Visitor クラスに対して visitConcreteElementC メソッドを追加している。レイヤードクラス図を図 24 に示す。

また、図 23 では、オブジェクト構造自身 (element モジュール) と、Visitor パターン (visitor モジュール) を分離して定義している。これは、オブジェクト構造に accept メソッドを後から追加することによって行っている。つまり、メソッド追加により、Visitor パターンの拡張性が上がるだけでなく、もともと Visitor パターンの適用を考慮していなかったオブジェクト構造に Visitor パターンを適用できる。

まとめると、この改善策には次のような性質がある。

- オブジェクト構造部分のソースコードを修正することなく後から Visitor パターンを導入できる。
- 新しい ConcreteElement クラスを導入できる。
- ConcreteElement の追加とオペレーションの追加を同時に行った場合には補完モジュールが必要になる。
- 既存のデザインパターンの構造のままでも利点が得られる。

### 6.5 オブジェクト構造に対するメソッド追加による別解

Visitor クラスがオブジェクト指向的でないという問題は、Visitor クラスを作るかわりにオブジェクト

構造に対してオペレーション用のメソッドを追加することで解決できる。

この問題は Visitor パターンが新しいオペレーションを後から導入できるという拡張性を実現するために、オペレーションを (Java などでは後から導入できない) メソッドではなく (後から導入できる) クラスを使って表現したことに起因する。しかし、MixJuice ではメソッドもクラスと同様に後から導入できるため、オペレーションをメソッドで実現しても目的とする拡張性を実現できる。

この Visitor クラスが存在しない方法を使うと、Visitor クラスにオブジェクト構造の各クラスをメソッドとして列挙する必要がなくなるため、オブジェクト構造にクラスを加えることも容易になる。

これらのオペレーションの追加とオブジェクト構造へのクラスの追加という 2 つの拡張を同時に使用した場合、補完モジュールが必要になる。この補完モジュールでは前者で導入したオペレーションの、後者で導入したクラスに関する処理を実装しなければならない。

図 25 では、オブジェクト構造自身を定義する element モジュールと、オペレーションを定義する operation1, operation2 モジュールを定義する。operation1, operation2 モジュールは Visitor パターンで各 ConcreteVisitor が行っていた処

```

module element {
  define abstract class Element {...}
  define class ConcreteElementA extends Element {...}
  define class ConcreteElementB extends Element {...}
}

module visitor extends element {
  class Element { define abstract void accept(Visitor v); }
  class ConcreteElementA {void accept(Visitor v) {v.visitConcreteElementA(this);}}
  class ConcreteElementB {void accept(Visitor v) {v.visitConcreteElementB(this);}}

  define class Visitor {
    define abstract void visitConcreteElementA(ConcreteElementA elt);
    define abstract void visitConcreteElementB(ConcreteElementB elt);
  }

  define class ConcreteVisitor1 extends Visitor {
    void visitConcreteElementA(ConcreteElementA elt) {...}
    void visitConcreteElementB(ConcreteElementB elt) {...}
  }

  define class ConcreteVisitor2 extends ConcreteVisitor1 {
    void visitConcreteElementA(ConcreteElementA elt) {...}
    void visitConcreteElementB(ConcreteElementB elt) {...}
  }
}

module element_c extends visitor {
  define class ConcreteElementC extends Element {
    ...
    void accept(Visitor v) { v.visitConcreteElementC(this); }
  }

  class Visitor {
    define abstract void visitConcreteElementC(ConcreteElementC elt);
  }

  class ConcreteVisitor1 {
    void visitConcreteElementC(ConcreteElementC elt) {...}
  }

  class ConcreteVisitor2 {
    void visitConcreteElementC(ConcreteElementC elt) {...}
  }
}

```

図 23 Visitor パターンにおける ConcreteElement クラスの追加  
Fig. 23 ConcreteElement Addition for Visitor Pattern.

理を実装する。つまり、operation1 モジュールが ConcreteVisitor1 と同じオペレーションを表現するとすれば、ConcreteVisitor1 の visitConcreteElementA で行われる処理を ConcreteElementA の operation1 メソッドとして定義する。ここで、他のオペレーションが必要になった場合には、Visitor パターンで ConcreteVisitor を増やすのと同様に operation1, operation2 モジュールと同様なモジュールを追加して定義すればよく、element モジュールの変更は必要ない。つまり、Visitor パターンと同等な拡張性が実現できている。レイヤードクラス図を図 26 に示す。

さらに、オペレーションに必要なフィールドをメソッドと同じモジュールで追加することもできる。この場合、そのフィールドは他のオペレーションからは不可視になるため、情報隠蔽も可能である。

ただし、ConcreteVisitor クラスの定義において

図 24 の ConcreteVisitor2 のように継承を利用する場合には、この別解は使用できない。このような継承が必要になるのは複数のオペレーションで共通の処理を再利用する場合であり、オペレーションをメソッドとして追加する場合にはそのような再利用は困難である。

このような形でオブジェクト構造とオペレーションの分離を行うと、Visitor クラス以下の階層および accept メソッドは不要である。オペレーションを追加するためには単にオペレーションをモジュールとして実装して追加すればよい。つまり、追加可能とするための特別なことはなにもなく、そのような拡張性を MixJuice という言語自身がサポートしているといえる。なお、1 章に GoF 本から引用したように、CLOS<sup>17)</sup> でも MixJuice と同様に Visitor パターンは不要である。

まとめると、この改善策には次のような性質がある。

- Visitor パターンを使う必要がなく、Visitor クラ

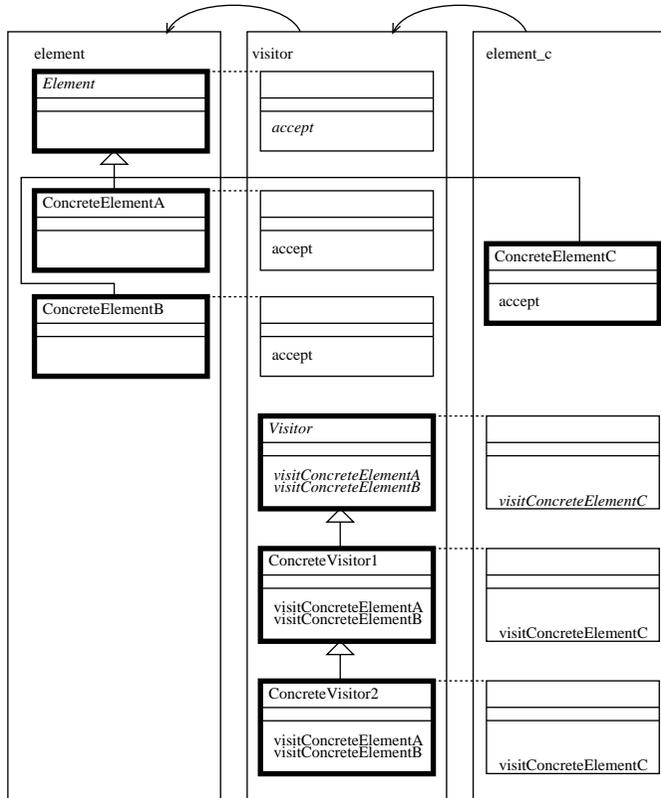


図 24 ConcreteElement クラスの追加のレイヤードクラス図

Fig. 24 Layered Class Diagram for Addition of ConcreteElement Class.

```

module element {
  define abstract class Element {...}
  define class ConcreteElementA extends Element {...}
  define class ConcreteElementB extends Element {...}
}

module operation1 extends element {
  class Element { define abstract void operation1(); }
  class ConcreteElementA { void operation1() {...} }
  class ConcreteElementB { void operation1() {...} }
}

module operation2 extends element {
  class Element { define abstract void operation2(); }
  class ConcreteElementA { void operation2() {...} }
  class ConcreteElementB { void operation2() {...} }
}

```

図 25 Visitor クラスを使わないオペレーション追加

Fig. 25 Operation Addition without Visitor Class.

スが必要なくなり、記述が単純になる。

- オブジェクト構造の拡張が可能になる。
- ConcreteElement の追加とオペレーションの追加を同時に行った場合には補完モジュールが必要になる。
- 複数のオペレーションで共通の処理を再利用することは困難である。

## 7. デザインパターンの改善点

本章では、MixJuice による GoF パターン 23 種の改善をまとめる。

### 7.1 各パターンへの影響

MixJuice による各 GoF パターンの改善を表 1 に示す。各パターンにはいくつかの改善策があり、表 1

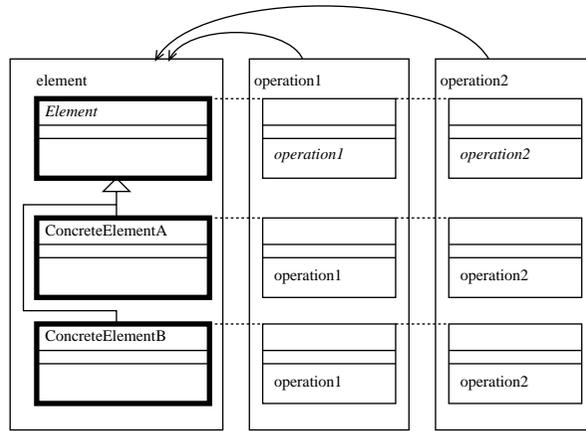


図 26 Visitor クラスを使わないオペレーション追加のレイヤードクラス図

Fig. 26 Layered Class Diagram for Operation Addition without Visitor Class.

表 1 MixJuice のデザインパターンへの影響

Table 1 Improvements for Design Patterns by MixJuice.

デザインパターン	種別	導入	拡張	情報隠蔽	型安全性	単純化	使用するプログラミング技法
AbstractFactory	改善		p.98				メソッド追加
	別解		p.98		p.100		実装モジュール選択
Builder	改善		p109				メソッド追加・メソッド拡張
	別解						メソッド拡張・実装モジュール選択
FactoryMethod	改善			p.118			名前空間分離
	別解						実装モジュール選択
Prototype	改善	p.131					メソッド追加
Singleton	別解					p.138	クラスメソッド拡張
Adapter	別解		p.153			p.152	スーパーインタフェース追加
Bridge	改善						メソッド追加
	別解						実装モジュール選択
Composite	改善						スーパーインタフェース追加
Decorator	改善		p.191				メソッド追加
	別解					p.190	クラス拡張
Facade	改善			p.201			名前空間分離
	別解						クラスメソッド追加
Flyweight	なし						
Proxy	なし						
ChainOfResponsibility	改善		p.241				スーパーインタフェース追加・メソッド追加
Command	なし						
Interpreter	改善		p.265				メソッド追加
Iterator	改善			p.280			名前空間分離
Mediator	なし						
Memento	改善			p.307			名前空間分離
Observer	改善		p.318				クラス拡張
State	改善						メソッド追加
Strategy	改善		p.339				メソッド追加
	別解					p.340	実装モジュール選択
TemplateMethod	改善		p.351				メソッド実装
Visitor	改善 1			p.359			仕様モジュールと実装モジュールの分離
	改善 2		p.358				メソッド追加
	別解		p.358	p.359			メソッド追加

( 表内のページ数は , MixJuice が解決するデザインパターンの問題について , GoF 日本語版が述べている場所を示している . )

はそれぞれの改善策について、その種別と改善点およびその改善に使った MixJuice の技法を示している。ここで、表に載せてあるのは実用上重要と思われる問題点の改善策であり、改善点の過半数は GoF 本にも載っている問題を解決したものである。

改善策の種別はクラス構造が GoF 本のもと同じかどうかを示している。「改善」はクラス構造が同じものを示しており、既存の Java プログラムを直接 MixJuice に移植すればそのまま恩恵が得られ、GoF パターンの利点を失わないことが保証される。「別解」はそうではなく、既存のソースコードを整理しなければならず、さまざまな得失がある。

改善点は次の 5 種に分類した。

**導入可能性** 既存のクラスをあるデザインパターンの新たな構成要素にするために、ソースコードの編集の必要があったものが、MixJuice を用いることによって、編集の必要がなくなるもの。たとえば 6.4 節では、Visitor パターンの適用が考慮されていないオブジェクト構造に対し、他のモジュールが accept メソッドを追加することによって Visitor パターンの構成要素にできることを述べた。

また、これは 1 つのクラスが複数のデザインパターンに参加している場合でも、個々のデザインパターンごとに別のモジュールに分離して記述可能であることを意味する。つまり、1 つのデザインパターンに関する複数のアスペクトを分離することが可能である。このことはプログラムのモジュラリティ向上に貢献する。

**拡張性** パターンを用いたプログラムに新たな機能を追加するとき、従来はソースコードの編集の必要があったものが、MixJuice を用いることによって、編集の必要がなくなるもの。たとえば、後からスーパークラスにメソッドを追加する必要がある場合がこれにあたる。5.3 節、6.4 節では、Abstract Factory パターン・Visitor パターンがメソッド追加によって拡張性が向上することを述べた。

なお、これはパターンの各部を機能単位でモジュールに分割できることを意味し、導入可能性と同様にモジュラリティ向上に貢献する。

**情報隠蔽** クラス単位の情報隠蔽機構しかない場合に比べて、MixJuice を用いることによって情報隠蔽の精度が向上するもの。これは名前を定義する側と参照する側の双方に利点がある。定義する側にとっては名前を参照するスコープのサイズが減少することにより、定義を変更する場合の影響範囲が小さくなる利点がある。参照する側にとって

は名前を使用するかどうかの選択肢が増加するという利点がある。たとえば、6.3 節では、Visitor パターンが対象とするオブジェクト構造内部の名前を隠蔽し、必要のないモジュールには非公開にできることを述べた。

なお、Java の package, nested class や C++ の friend のような機構があれば、MixJuice でなくても改善できる場合がある。ただし、Java や C++ では、あるクラスの非公開部分をさらに複数に分割してそれぞれで異なる可視性を持たせることはできない。

**型安全性** ダウンキャストの必要があったものが、不要になるもの。これは、GoF パターンではない「別解」によって可能になる。たとえば、5.4 節では、AbstractProduct クラスと ConcreteProduct クラスが融合したクラス構造を使うことにより、ダウンキャストが不要になることを述べた。

**クラス構造の単純化** クラスの数が減少し、クラス構造が単純化するもの。これは、GoF パターンではない「別解」によって可能になる。これにより、プログラムの実行時のイメージを理解やすくなる。たとえば、5.4 節、6.5 節で述べた改善策では、それぞれ AbstractFactory クラス・Visitor クラス以下のクラス階層が除去され、クラス数が減少している。なお、1 つのクラスの定義が複数のモジュールに分割される場合があるので、必ずしもソースコードが単純化するとは限らない。

## 8. モジュール機構とデザインパターン

MixJuice の差分ベースモジュールがデザインパターンに寄与する点は次の 3 つにまとめられる。

- 拡張手段の増加
- クラス独立な情報隠蔽機構
- 実装モジュール選択による静的な多態性

本章ではそれぞれについて述べ、他の言語との関連について触れる。

### 8.1 拡張手段の増加

Java などの従来のオブジェクト指向言語において、ソースコードを編集せずに可能な拡張手段はサブクラスを定義することしかない。しかも、サブクラスを使うにはアプリケーションのどこかでそのサブクラスのインスタンスを生成しなければならない。つまり、サブクラスの名前を明示的に記述してインスタンス生成を行うコードをアプリケーションに挿入しなければならず、アプリケーションの拡張には必ずソースコードの編集が必要になる。

しかし, MixJuice にはサブクラスを定義する以外の拡張手段があり, ソースコードを編集せずにさまざまな拡張が行える. また, それらの拡張はある種の規則に従うことにより, それぞれの拡張を単独で理解可能 (modular reasoning 可能) にできる<sup>14)</sup>.

たとえば, あるモジュールで定義したクラスに対し他のモジュールでメソッドを追加できる機能によって, Abstract Factory パターン・Visitor パターンの拡張性を改善できることを 5.3 節と 6.4 節で述べた. また, 6.4 節では, 同様なメソッド追加により, Visitor パターン自身が導入可能なことも述べた.

このように, 導入可能性・拡張性の改善点は拡張手段の増加によって実現されている. なお, あるモジュールで定義したクラスを他のモジュールで修正できる機能を持っている言語は MixJuice だけではなく, Cecil<sup>2)</sup>, MultiJava<sup>3)</sup>, AspectJ<sup>15)</sup>, CLOS<sup>17)</sup>, Ruby<sup>16)</sup> など, さまざまな言語がある. そのような修正が可能なクラスはオープンクラスと呼ばれる<sup>3)</sup>.

## 8.2 クラス独立な情報隠蔽機構

デザインパターンのように複数のクラスが協調して動作する状況では, クラス単位の名前空間は適切ではない. たとえば, 6.3 節では, Visitor パターンのオブジェクト構造と Visitor にまたがる処理の情報隠蔽のために, クラスを横断する名前空間が必要になることを述べた.

ここで, ある機能に対して適切な情報隠蔽を行うためには, その機能に関連するコードを含み, かつ, なるべく小さな領域を名前空間とする必要がある. しかし, 次の 2 つの問題により, クラスを最小単位とする名前空間では適切な名前空間を構成できない.

- 1 つの機能に関連するコードは複数のクラスにまたがって存在することがある.
- 1 つのクラスには複数の機能に関連するコードが存在することがある.

MixJuice のクラス独立な情報隠蔽機構はこのような状況でも適切な名前空間を構成できる. これは MixJuice ではメソッドやフィールドを最小単位とし, クラスをまたがった名前空間を構成できるからである. つまり, 情報隠蔽の改善点はクラス独立な情報隠蔽機構によって実現されている.

## 8.3 実装モジュール選択による静的な多態性

デザインパターンが提供する拡張性は「新しくサブクラスを定義することによって機能を追加できる」というものが多い. たとえば, Abstract Factory パターンでは AbstractFactory クラスのサブクラスを定義することによってオブジェクト群を追加し, Visitor パ

ターンでは Visitor クラスのサブクラスを追加することによってオペレーションを追加する. このような拡張機能は, 多態性によりクライアントのソースコードをほとんど編集せずに利用できる.

MixJuice の実装モジュール選択はこの多態性と類似的な機能を提供する. 実装モジュール選択によって「新しくモジュールを実装することによって機能を (リンク時に) 選択可能にできる」という拡張性を実現できる. また, この選択時にクライアントのソースコードを編集する必要はない. たとえば, 5.4 節では, Abstract Factory パターンと同様な拡張性を実装モジュール選択によって実現することを述べた.

ただし, 実装モジュール選択による多態性はサブクラスによる多態性とは異なり, リンク時に選択を行わなければならない. つまり, 動的に選択することはできず, 静的に選択しなければならない. サブクラスによる多態性ではすべてのサブクラスをリンクし, 1 つのプログラムの実行中にすべてのサブクラスの機能を使い分けられることができる. これに対し, 実装モジュール選択ではリンク時に 1 つのモジュールを選択しなければならないため, 実行中には 1 つのモジュールの機能しか使用できない.

しかし, リンク時に選択を行わなければならないというこの制約はクラス数を減少させる効果がある. これは実行中には 1 つのモジュールの機能しか存在しないため, サブクラスを作らず, スーパークラスへのメソッド追加などによって機能を実装することが可能だからである.

また, サブクラスを作らないことには型安全性を高める効果もある. サブクラスを作ると, 5.2 節で述べたように, スーパークラスからサブクラスへのダウンキャストが必要になることがあるが, サブクラスが存在しなければダウンキャストは不要である.

さらに, 実装モジュール選択はプログラムのバイナリサイズを減らす効果もある<sup>18)</sup>. これは 1 つの機能を実現するモジュールのみをリンクするからであり, また, 機能制限版を容易に (クラス構造を複雑にせず, 型安全性を保ったまま) 実装できるからである. なお, サブクラスを使用しつつ特定のサブクラスのみをリンクすることは可能であるが, クラス構造が複雑になり, 型安全性を失い, ビルドプロセスが複雑になるという問題がある.

そして, 実装モジュール選択などによって静的な関係をリンク時に解決することはプログラムの高速化にも有効である<sup>6)</sup>. これは, 動的ディスパッチなど, 比較的遅い機構を使用しないプログラムを書けるからである.

表 2 言語と技法

Table 2 Languages and Techniques.

技法	AspectJ	CLOS	Ruby
スーパーインタフェース追加	declare parents		include
フィールド追加	introduction		代入
メソッド追加		defmethod	def
メソッド拡張	around advice		alias/def
名前空間分離	aspect	package	
仕様モジュールと実装モジュールの分離			
実装モジュール選択	aspect 選択		require 切替え

なお、MixJuiceの実装モジュール選択を使わなくても、既存の言語処理系のオプションや環境変数を使ってソースファイルやコンパイルされたモジュールをすり替えれば、同様の効果を得ることができる。たとえば、JavaならばCLASSPATHを変更することによって、同じ名前を持つ複数のクラスを静的に選択できる。しかし、これはビルドプロセスをさらに複雑にする<sup>11)</sup>。たとえば、複数のCLASSPATHを考慮して依存関係を管理しなければならない。

このように、型安全性とクラス構造の単純化の改善点は実装モジュール選択による静的な多態性によって実現されている。

#### 8.4 他の言語への適用

デザインパターンの改善策において使用されるMixJuiceのプログラミング技法には4章のようなものがあるが、MixJuice以外にも同様な技法を使える言語がある。たとえば、オープンクラスを持っているCLOSなどの言語ではメソッド追加が可能である。そのような言語ではMixJuiceと同様な改善が可能ながある。

AspectJ, CLOS, Rubyについて各技法の対応を表2に示す。

AspectJではMixJuiceとほぼ同様の技法が使える。ただし、AspectJはJavaの名前空間を引き継いでいるため、名前空間に関しては制限がある。とくに、introductionで導入するメソッドやフィールドと同名のもので導入先のクラスに存在した場合は衝突が起るために自由に名前をつけることはできず、名前空間を完全に分離することはできない。ただし、introductionを使わずにアスペクト自身にメソッドやフィールドを定義することによって、ある程度の分割は可能である。

CLOSとRubyは型が動的であるため、型に関する技法はMixJuiceやAspectJとは異なる。動的型の言語では、任意のオブジェクトを任意の変数に束縛できるため、型的な制約が存在しない。したがって、スーパーインタフェース追加によって型的な制約を満たすという技法は基本的には必要ない。ただし、プログラ

ム自身が動的に型を検査する場合には、その検査を制御するために型情報を設定する必要がある場合がある。

## 9. 関連研究

GoF本の23種のパターンとモジュール機構の関係を網羅的に調査したものは本論文のほかにHannemannらのもの<sup>10)</sup>とEideらのもの<sup>4)</sup>がある。

HannemannらはAspectJによるデザインパターンの改善を行った<sup>10)</sup>。この論文内の表には各パターンに対する改善点と各クラスの役割がまとめられている。ただし、この改善点はモジュラリティに関するものに限られており、拡張性が改善されることについては触れていない。また、8.4節で述べたようにAspectJは本論文で述べた技法をほぼそのまま使用できるが、彼らは既存のクラスにメソッドなどを追加する方法ではなく、アスペクトという一種のオブジェクトを追加することでデザインパターンの実装を行っている。このため、GoF本のパターンとはクラス構造が異なることになり、その違いによる差異が発生する。これに対し、本論文の(別解でない)改善では、既存のクラスにメソッドやフィールドを追加することによってGoF本のパターンと同じクラス構造を構成するため、クラス構造の違いによる差異は発生しない。たとえば、実行時に存在するオブジェクトの数は本論文の方法とGoF本の方法では同じになるが、彼らの方法では多くなる。

また、彼らはいくつかのデザインパターンをライブラリ化し再利用可能としている。しかし、これらのライブラリは型的に安全ではない。たとえば、ObserverパターンのライブラリでObserverとSubjectの組合せの間違いをコンパイラが検出することはできない。これに対し、MixJuiceでは拡張対象のモジュール・クラスを明示しなければならないため、任意のモジュール・クラスを拡張できるような再利用可能なライブラリは作れないが、型的に安全である。また、彼らのライブラリはコメントを除くと各パターンについて100

行以下しかない上にインタフェースの定義が主なため、実装を共有することにはあまり貢献しない。なお、AspectJ は将来的には generics<sup>1)</sup> のサポートによって型安全性の問題を解決しようとしており、MixJuice は将来的にはパラメライズドモジュールの導入によって再利用性の問題を解決しようとしている。したがって、AspectJ と MixJuice の安全性と再利用性の違いは将来は縮まる可能性がある。

Eide らは unit モデル<sup>5)</sup> によるモジュールシステムにより、デザインパターンをモジュール化し、各パターンの静的な部分と動的な部分を分離できることを示した<sup>4)</sup>。この論文内の表には各パターンの各クラスが静的か動的かの種別がまとめられている。静的な性質を利用するのは MixJuice で実装モジュール選択が静的な選択を行うのと同様であるが、それによってクラス構造を単純化できることには触れられていない。

## 10. おわりに

我々は 23 種の GoF パターンを MixJuice で書き直し、MixJuice の機能による改善策を検討した。本論文ではそのうち Abstract Factory パターンと Visitor パターンについてそれぞれ複数の改善策の詳細を述べた。改善策のプログラム構造をレイヤードクラス図により視覚的に表現し、また、改善策の利点と欠点を明確に述べた。23 種すべてを載せたカタログは <http://cvs.m17n.org/~akr/mj/design-pattern/> で公開している。

各改善策の改善点は導入可能性、拡張性、情報隠蔽、型安全性、クラス構造の単純化に分類した。そして、導入可能性・拡張性の改善は拡張手段の増加によって実現され、情報隠蔽の改善はクラス独立な情報隠蔽機構によって実現され、型安全性の改善・クラス構造の単純化は実装モジュール選択による静的な多態性によって実現されることを述べた。

また、各改善策にはそれぞれで使用した技法をまとめた。これにより、MixJuice だけでなく、他の言語に対して改善策が適用できるかどうかを判断できる。そして、AspectJ, CLOS, Ruby の各言語について技法の対応を述べた。

## 参 考 文 献

- 1) Bracha, G.: JSR14: Add Generic Types To The Java Programming Language. <http://jcp.org/jsr/detail/14.jsp>
- 2) Chambers, C. and Leavens, G.T.: Typechecking and Modules for Multimethods, *ACM*

*Trans. Prog. Lang. Syst. (TOPLAS)*, Vol.17, No.6, pp.805–843 (1995).

- 3) Clifton, C., Leavens, G.T., Chambers, C. and Millstein, T.: MultiJava: modular open classes and symmetric multiple dispatch for Java, *Proc. OOPSLA2000*, pp.130–145 (2000). Published as ACM SIGPLAN Notices, Vol.35, No.10.
- 4) Eide, E., Reid, A., Regehr, J. and Lepreau, J.: Static and Dynamic Structure in Design Patterns, *Proc. 24th International Conference on Software Engineering (ICSE 2002)*, pp.208–218 (2002).
- 5) Flatt, M. and Felleisen, M.: Units: Cool Modules for HOT Languages, *ACM SIGPLAN Notices*, Vol.33, No.5, pp.236–248 (1998).
- 6) Friedrich, M., Papajewski, H., Schröder-Preikschat, W., Spinczyk, O. and Spinczyk, U.: Efficient Object-Oriented Software with Design Patterns, *Generative and Component-Based Software Engineering, 1st International Symposium (GCSE'99)*, LNCS 1799, pp.79–90 (1999).
- 7) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley professional computing series, Addison-Wesley, Reading, MA, USA (1995).
- 8) Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 本位伝真一, 吉田和樹(監訳): オブジェクト指向における再利用のためのデザインパターン, ソフトバンク株式会社 (1995).
- 9) Gil, J. and Lorenz, D.H.: Design Patterns vs. Language Design, *Proc. ECOOP'97*, LNCS 1357, pp.108–115 (1998).
- 10) Hannemann, J. and Kiczales, G.: Design Pattern Implementation in Java and AspectJ, *Proc. OOPSLA2002* (2002).
- 11) Harper, R. and Pierce, B.C.: Advanced module systems (invited talk): a guide for the perplexed, *Proc. fifth ACM SIGPLAN International Conference on Functional programming (ICFP '00)*, p.130 (2000). Published as ACM SIGPLAN Notices, Vol.35, No.9. <http://www.cis.upenn.edu/bcpierce/papers/modules-icfp.ps>
- 12) Ichisugi, Y.: MixJuice home page. <http://staff.aist.go.jp/y-ichisugi/mj/>
- 13) Ichisugi, Y. and Tanaka, A.: Difference-Based Modules: A Class-Independent Module Mechanism, *Proc. ECOOP 2002*, LNCS 2374, pp.62–68 (2002).
- 14) 一杉裕志, 田中 哲: 安全に結合可能なアスペクトを提供するためのルール, 日本ソフトウェア

科学会第 19 回全国大会論文集，日本ソフトウェア科学会 (2002).

- 15) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., m, J.P. and Griswold, W.: An overview of AspectJ, *Proc. ECOOP2001* (2001).
- 16) Matsumoto, Y.: Ruby home page.  
<http://www.ruby-lang.org>
- 17) Steele, G.: *Common Lisp the Language*, 2nd edition, Digital Press (1990).
- 18) 田中 哲，一杉裕志：プログラミング言語 MixJuice による HTTP server のモジュール化，日本ソフトウェア科学会第 18 回全国大会論文集，日本ソフトウェア科学会 (2001).

(平成 14 年 9 月 30 日受付)

(平成 14 年 10 月 22 日採録)



田中 哲 (正会員)

1972 年生．1994 年東京電機大学工学部電子工学科退学．2000 年北陸先端科学技術大学院大学博士後期課程修了．博士 (情報科学)．同年より電子技術総合研究所にて COE 特別研究員として勤務．2002 年産業技術総合研究所入所．拡張可能なシステム・言語処理系・関数型言語・フリーソフトウェアに興味を持つ．日本ソフトウェア科学会会員．



一杉 裕志 (正会員)

1965 年生．1990 年東京工業大学大学院情報科学専攻修士課程修了．1993 年東京大学大学院情報科学専攻博士課程修了．博士 (理学)．同年電子技術総合研究所入所．2001 年より産業技術総合研究所情報処理研究部門に勤務．オブジェクト指向言語，アスペクト指向言語，並列・分散言語，拡張可能システムに興味を持つ．日本ソフトウェア科学会，ACM 各会員．