シンプルかつ強力なモジュール機構を有するオブジェクト指向言語 MixJuice の提案

MixJuice : An object-oriented language with simple and powerful module mechanism

一杉 裕志
Yuuji ICHISUGI
科学技術振興事業団さきがけ研究 21 ／電子技術総合研究所
PRESTO, JST / Electrotechnical Laboratory

Abstract

ソースコード 全体に対する差分を情報隠蔽・再利用の単位として用いるオブジェクト指向言語 MixJuice を設計・実装した。この言語においては、オブジェクトの雛型であるクラスと、情報隠蔽・再利用の単位であるモジュールは、完全に直交している。これにより、観点の分離がサポートされる。MixJuice は Java をベースにした言語であるが、モジュールに関する機構が Java よりシンプルかつ強力になっている。MixJuice のプログラマは、 Java 言語の全てのライブラリと実行環境を利用することができる。

1 背景

現在のオブジェクト指向言語 (C++, Java など) は、“class” という構文にモジュールとしての役割を持たせている。モジュールとは情報隠蔽・再利用の単位であるが、これは、オブジェクトの雛型であるクラスとは、本来は独立した概念である。大規模の現実的なプログラムにおいては、クラスをモジュールとして用いると、様々な問題が生じる。

第一に、クラスは情報隠蔽の単位としては不適切である [9]。「クラス＝情報隠蔽の単位」という等式は、小規模のシステムにおいて近似的に成り立つにすぎない。大規模なシステムで、この近似の精度を少しでも高めるために、friend, protected, namespace などの機構が C++ に、package, inner class などの機構が Java に導入された。しかし、これらの機構が導入された現在においても、C++ や Java の情報隠蔽の機構は十分ではない。例えば、あるソフトウエアが有する機能が増えると、それに伴って各クラスが有する field, method はどうしても増えていく。これはスコープのサイズが肥大化していくことを意味し、システムの保守性を悪くする原因になる。

第二に、クラスは再利用の単位としては不適切である。一般にある特定の機能に関連した部分は、複数のクラスにまたがって存在する [6]。この部分を再利用の単位として、クラスとは独立して記述できるように、いわゆる「観点の分離 (separation of concerns)」を、プログラミング言語が支援する必要がある。実際に観点の分離をサポートするプログラミングスタイルやプログラミング言語がいくつか提案されている（例えば [6, 8]）。しかし、いずれの提案も、型チェックがない、分割コンパイルができないなどの問題がある。

これらの問題に対処するために、１つのクラスを複数のコンポーネントに分けて記述するという方法がある。確かにこの方法によってクラスを複数に分割できるが、「１種類のオブジェクトの雛型を１つのクラスで表現する」という素直な記述ができなくなるという欠点がある。それに伴い、メソッドの実行を単に他のオブジェクトに委譲するだけのメソッドが、ソースコード中に多く現れるようになる。このことは、システムの保守性を逆に悪くする原因になる。また、委譲することにより実行性能のオーバーヘッドも生じる。

以上のような背景から、筆者は、クラスとモジュールとを完全に分離したオブジェクト指向言語 MixJuice を設計・実装した。

2 MixJuice のモジュール機能

2.1 「モジュール ＝ 差分」

MixJuice では、「オリジナルのシステムに対する差分」をモジュールとして扱う。差分とは、 patch file のようなものであり、オリジナルのシステムに対する変更部分だけを抜き出したものである。従来のオブジェクト指向言語における継承との違いは、差分の追加の対象が単一のクラスではなく、クラス階層全体である、という点である。

プログラマは複数の class にまたがった機能を１つのモジュールとして表現することで、「観点の分離」が達成できる。

各モジュールは分割コンパイルして、それぞれソー

```
module m  // このモジュールの名前
  extends m1, m2 // 差分の追加の対象となるモジュール
  uses m3, m4  // このモジュールから参照するモジュール
{  // モジュール本体（追加する差分）
   delta S {...} // 既存のクラスへの差分の追加
   class A extends S {...} // 新たなクラスの追加
   ...
}
```

図 1: モジュール定義例

```
abstract class S {
  abstract boolean equals(S x);
  boolean equalsToA(A x) {
    return false; }
  boolean equalsToB(B x) {
    return false; }
}
```

```
class A extends S {
  boolean equals(S x) {
    return x.equalsToA(this); }
  boolean equalsToA(A x) { ... }
}
```

```
class B extends S {
  boolean equals(S x) {
    return x.equalsToB(this); }
  boolean equalsToB(B x) { ... }
}
```

図 2: Java によるプログラム例

スコード無しで配布することができる。コンパイル時には、モジュール単位で型チェックが行われる。

プログラムを実行する際には、プログラムのユーザが、使用するモジュールを指定する。指定されたモジュールは、以下のように１つのプログラムにリンクされる。まず各モジュールは、それぞれの依存関係に従って topological sort される。そして、その topological sort 結果の順番にしたがって、次々に差分の追加が行われる。そして最終的にできあがったプログラムがリンク結果となる。リンク時にも、型の整合性が検査されるため、リンクされたプログラムの安全性が保証される。

MixJuice で書かれたプログラムのユーザは複数のモジュールの中から自分が必要とする機能を選択して組み合わせて利用することができる。つまり従来の条件コンパイル (C プリプロセッサの `#ifdef`) や patch に相当することを、MixJuice は言語レベルでより安全な方法でサポートしている。

2.2 モジュール定義の構文

MixJuice では、モジュール m の定義は図 1のように記述する。

“extends” 宣言は、モジュール m が差分を追加する対象となるモジュールの名前を宣言する。従来のオブジェクト指向言語における super class に似ているため、指定されたモジュールを super module と呼ぶ。

“uses” 宣言は、このモジュール内から参照するモジュールを宣言する。この宣言は、従来のプログラミング言語のモジュール機構における “import” 宣言にほぼ相当する。“uses” 宣言で指定されたモジュールから参照可能な全ての名前（クラス名、フィールド名、メソッド名）は、このモジュールに継承され、このモジュール内から参照可能になる。なお、 “extends” 宣言は、“uses” 宣言の機能も兼ねており、super module 内から参照可能な名前は submodule からも参照可能である。

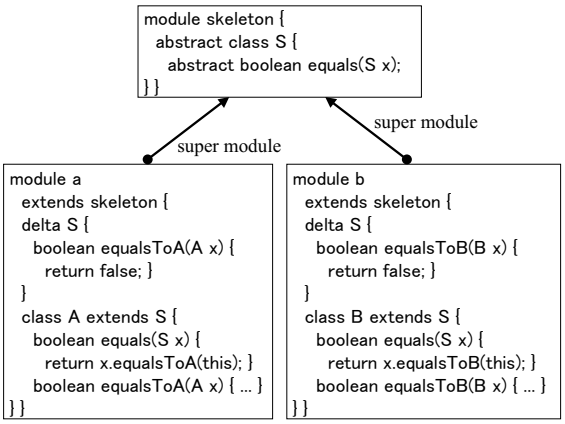

図 3: MixJuice によるプログラム例

モジュール本体には、このモジュールが super module に追加する差分部分を定義する。“delta” 構文は、既存のクラスに対する差分を記述する構文である。これにより、オリジナルのシステム内に存在するクラスに対して、フィールドやメソッドを追加することができる。

2.3 プログラム例

MixJuice を用いたプログラム例として、ダブルディスパッチと呼ばれる技法を用いたプログラムについて説明する。

まず、図 2は、普通の Java 言語によるプログラムの記述である。抽象クラス `S` のメソッド `equals` は、`S` のサブクラスのインスタンスの２つの値が等しいかどうかを調べるメソッドである。`x1.equals(x2)` が呼び出されると、このメソッドは `x1` のクラスに応じてディスパッチされ、その結果 `equalsToA` または `equalsToB` が呼び出される。そしてこれらのメソッドは、`x2` のクラスに応じてディスパッチされる。もし２つのオブジェクトが異なるクラスであれば、これらのメソッドは false を返し、同じクラスであれば、 `A` か `B` の実装に依存した方法で同値性が判断され、 true か false が返される。従来のオブジェクト指向言語でダブルディスパッチを記述すると、抽象クラス `S` の定義中にサブクラス `A`、 `B` の名前が現れるため、モジュラリティが悪い。そのため、新たなサブクラス `C` を追加するためには、 `S` の定義を編集しなければならないという問題がある。

全く同じプログラムを MixJuice を使って記述した

ものが図 3である。クラス名 A、 B に依存したコードを S の定義から分離して記述することができるため、よりモジュラリティが高い。また、同様にしてモジュールを追加することで、既存のソースを変更することなしに、S にサブクラス C を追加できる。このように、 MixJuice では、すでに存在するクラスに対してメソッドを追加することが可能であるため、従来のオブジェクト指向言語で書かれたプログラムよりも拡張性が高くなる。

2.4 protected, package, inner class の廃止

protected, package, inner class という機能は、もともと「クラス＝情報隠蔽の単位」という近似式の精度をより高めるための補正項として導入されたものである。MixJuice では、名前空間の多重継承を用いることにより、任意の名前空間が表現可能になっている。そのため、これらの機能はもはや不要である。

MixJuice が Java よりも柔軟にモジュール分割が可能であることを示すために、JDK1.2 に付属するクラスである java.util.HashMap のソースを例にとり説明する。HashMap は、 inner class を巧みに用いることにより、外部には必要最低限の名前しか公開されておらず、その意味ではモジュラリティが高い実装になっている。しかし、 HashMap の内部は、逆にモジュラリティの高い実装にはなっていない。図 4は、ソースファイル HashMap.java に含まれる３つのクラス（そのうち２つは inner class ）の間の依存関係を示したものである。３つのクラスは相互に依存している上、お互いの field や method が直接参照できる関係にあり、コードの一部の修正が、ファイル内のどの部分に影響を与えるかが非常にわかりにくくなっている。

HashMap.java はコメントを除いて約５００行であるが、より高機能なクラス TreeMap.java は約１０００行もあり、内部の依存関係はいっそう複雑である。これらのソースコードは、従来のオブジェクト指向言語が有するモジュール機構が、高機能なクラスの内部をモジュール化する能力を有していないことを示している。

図 5は、HashMap のソースコードを MixJuice で書き直し、７つのモジュールに分割した場合の、モジュール間の依存関係を示したものである。このように、依存関係にサイクルがなく、しかもサイズが小さく安定したモジュールにのみ実装が依存する形に、モジュール分割することが可能である。これにより、プログラムの保守性は大きく向上すると考えられる。

2.5 static field, static method の廃止

MixJuice では “Shared” という名前の特殊なクラスのフィールドとメソッドによって、従来の static field, static method に相当するものを表現する。プログラムの起動時に、クラス Shared のインスタンスが１つ

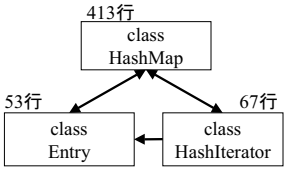

図 4: HashMap.java 内部のクラス間の依存関係

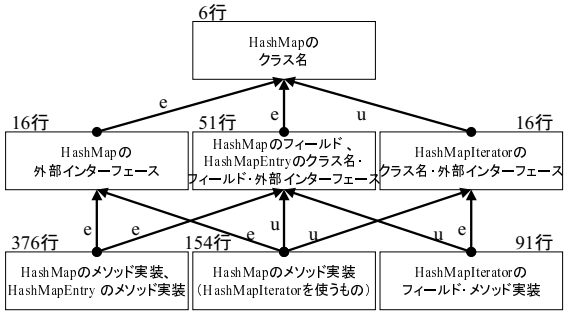

図 5: MixJuice を用いて書き直した HashMap.java の内部のモジュール間の依存関係（矢印の e は “extends”、 u は “uses” していることを表す。）

生成される。このインスタンスは他のすべてのオブジェクトから、いつでも参照することが出来る。クラス Shared の定義は、 delta 構文を用いて複数のモジュールに分散させることができるため、クラス Shared の各フィールドと各メソッドの可視範囲を制御することができる。

なお、クラス Shared は “void main(String[])” というメソッドを持っており、これが起動時に呼び出される。

Java では static method は拡張不可能だが、MixJuice では、submodule が “delta” を追加することで拡張可能になっている。これにより、プログラムの拡張性・柔軟性が増している。

3 実装

MixJuice で書かれたプログラムのソースコードは、プリプロセッサ（ソースコード変換）、 Java コンパイラ、ポストプロセッサ（バイトコード変換）によって処理され、最終的に Java 言語のバイトコードに変換される。

delta 構文による差分の追加を実現するには、JavaVM が本来持っている継承メカニズムを利用する（図 6）。module 本体に含まれる各 delta は、プリプロセッサによって、ダミーの super class を持つクラ

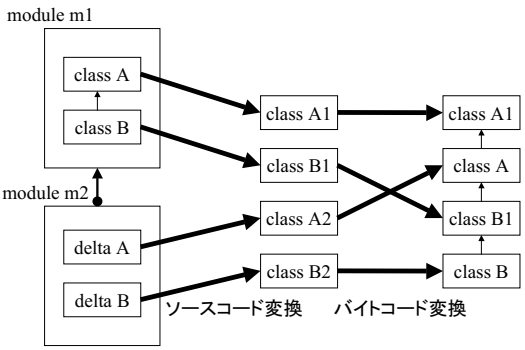

図 6: プリプロセッサとポストプロセッサによる実装

スに変換される。変換結果のソースコードは、 Java コンパイラによって普通にコンパイルされる。リンク時には、各 delta が一列に linearize されるように、各 delta の super class がポストプロセッサによって書き換えられ、適宜クラス名も変更されて、「差分を追加した結果」に相当するクラス階層が構築される。

なお、プリプロセッサとポストプロセッサは、Java コンパイラによる型チェックと、 JavaVM による verifier のチェックを通るように、注意深く変換を行っている。

4 関連研究

筆者は MixJuice の前身である Ld-2 という型チェックを行わない言語を既に実装し、それを用いて EPP (拡張可能 Java プリプロセッサ) [3, 4] という約１万５千行のプログラムを記述している。EPP は差分をモジュールとするプログラミングスタイルで書かれており、モジュールを追加することで振る舞いを拡張することができる。また、独立して開発された複数のモジュールを同時に１つのシステムに追加して動作させることもできる。MixJuice は汎用言語であるが、EPP のような拡張性の高いソフトウエアの構築に、特に適した言語である。

Mixin Layers[8] は、特殊な言語を用いず、 C++ の template を駆使して差分を記述する手法である。しかしこの手法は、差分の分割コンパイルができず、デバッグも困難であるという問題がある。

BCA[5] は、バイトコード変換により既存のクラスライブラリの再利用性を高めるためのシステムである。delta file と呼ぶ差分を記述することで、ソースコードのない既存のクラスファイルを変更することができる。しかし、 delta file の型チェック機能は実装されていない。また、専用の JavaVM を必要とする。

Cecil[1], Dubious[7], MultiJava[2] は、いずれも multi-method をサポートするオブジェクト指向言語であるが、MixJuice のものと非常に近いモジュール機構を有し、モジュール単位で型チェック・分割コンパイルが可能である。既存のクラスのソースコードを編集せずに、クラスにメソッドを追加する機能も有しており、この機能を Chambers らは open class と呼んでいる [7]。MultiJava[2] は、 Java 言語に open class と multi-method の機能を追加した言語である。MultiJava は、バイトコード変換ではなく、Chain of Responsibility pattern を用いて open class を実現しているため、クラスに差分を多く追加するほど メソッド呼び出しは遅くなる。一方 MixJuice の実装方法では、差分をいくら追加しても、メソッド呼び出しが遅くなることはない。

5 まとめ

クラスではなく差分をモジュールとして扱うプログラミング言語を提案した。この言語は Java が持つ procected, package, inner class, static field/method という言語機構を不要にし、なおかつ Java よりも強力な名前空間制御、高い拡張性・再利用性をサポートすることを示した。

謝辞

MixJuice のテストに協力していただいた MIT の学生 My Le Hoang さんに感謝します。

参考文献

[1] Chambers, C. and Leavens, G.: ”Typechecking and Modules for Multi-Methods”, ACM TOPLAS, Vol. 17, No. 9, November, 1995.

[2] Clifton, C., Leavens, G. T., Chambers, C. and Millstein, T.: ”MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java”, In Proc. of OOPSLA 2000, October 15-19, 2000.

[3] Ichisugi, Y. and Roudier, Y.: ”The Extensible Java Preprocessor Kit and a Tiny Data-Parallel Java”, In Proc. of ISCOPE’97, LNCS 1343, pp153–160, California, Dec, 1997.

[4] Ichisugi, Y.: EPP, http://www.etl.go.jp/˜epp/

[5] Keller, R. and Hölzle, U., ”Binary Component Adaptation”, In Proc. of ECOOP’98, LNCS 1445, pp.307–329.

[6] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.M. and Irwin, J.: ”Aspect-Oriented Programming.” Invited Talk. In Proc. of E-COOP’97, LNCS 1241, pp.220-242, 1997.

[7] Millstein, T. and Chambers, C.: ”Modular Statically Typed Multimethods”, In Proc. of ECOOP 99, LNCS 1628, June 1999, pp. 279-303.

[8] Smaragdakis, Y. and Batory, D.: ”Implementing Layered Designs with Mixin Layers”, In Proc. of ECOOP 98, LNCS 1445.

[9] Szyperski, C.A.: ”Import is Not Inheritance, Why We Need Both: Modules and Classes”, In Proc. of E-COOP’92, pp.19–32, 1995.