システムソフトウェアと
オペレーティング・システム 78－3
（１９９８．５．７）

リフレクティブアーキテクチャによる仮想空間の実現

児玉靖司　　葛西毅郎

東京理科大学 理工学部 情報科学科

Java にリフレクティブ機能を導入することにより，簡単に，各オブジェクトの挙動を動的に変更することができる．我々は，クラス MetaObject を定義し，各オブジェクトに対して動的なメソッド更新，動的継承を可能にした．さらに我々が開発中の仮想空間 Virtual Community へ適用し，このシステムを構成する部品 (特に動的に変化する部品) に対してリフレクティブタワーを形成した．結果として，各オブジェクトの動的変化を記述する際に、リフレクティブ機能を用いることにより見通しの良い記述が可能となった．

一方，我々が提案するクラス MetaObject を使えば，既存の Java のアプリケーションに簡単にリフレクティブ機能を適用できる．今後の課題として，JavaBeans コンポーネントウエアにリフレクティブアーキテクチャを導入することが考えられる．

An Implementation of a Virtual World using Reflective Architecture

Yasushi KODAMA　　Takao KASAI

Department of Information Sciences, Science University of Tokyo

The behavior of each object can be changed dynamically by introducing the reflection into the language Java. And it enables us to update a method of each object and inherit methods dynamically since we define a class MetaObject. In addtion, we applied it to a virtual world which we are now investigating. A reflective tower was formed to parts which composed this system. Consequently, when an object which always changes appearance dynamically will be described, the reflective function brought us a good description power.

While, the reflection can be adapted to any applications which written in Java using our class MetaObject. In future work, we will adapt it to JavaBeans component architecture using these facilities.

1 はじめに

柔軟性のある Java 言語により，簡単に動的に変化するアプリケーションを記述することができる．さらに，リフレクティブアーキテクチャにより，動的に変化するアプリケーションを記述する際に，大規模アプリケーションを見通しよく記述することができる．

リフレクティブアーキテクチャのプログラミング言語への導入として [5] が提案され，その後，並列オブジェクト指向言語へは [6][8]，C++ への適用として [7][2] など様々な研究がされている．我々は，Java にも同様の拡張が簡単に拡張できることを示す．最近の研究として OpenJava [9] があるが，我々は，簡単さを重視し，別のアプローチを採用する．さらに，我々が開発している仮想空間 Virtual Community への応用について議論する．

現在バージョンの Java には，リフレクション機能が備わっているが，いわゆる内省 (introspection) と呼ぶ，クラス，オブジェクトなどの情報をメソッド呼出しにより取り出す機能しか提供されていない．この問題提起は [9] によりされたが，我々は簡単さを重視し，新たにライブラリを提供することのみにより，リフレクティブ機能を提供する．リフレクティブアーキテクチャとしては，IBA(individual based architecutre) を提供し，GWA(group-wide architecture) は，今後の課題とする．

さらに，コンポーネントウエア JavaBeans への適用についても考察する．

2 リフレクション

我々は，クラス MetaObject を提供し，Java で記述する全てのオブジェクト (クラス) に対してメソッド更新 (method update)，動的継承 (dynamic inheritance) の機能を提供する．さらなる機能の提供を受けたい場合は，クラス MetaObject を更新することにより簡単に拡張することができる．各オブジェクトが，この機能の提供を受ける場合は，

- クラス MetaObject を継承する．
- クラス MetaObject のインスタンスを生成し，そのメソッドを呼出す．

が考えられるが，Java が単継承であり，既存のライブラリを継承したクラスにもリフレクティブ機能を提供したいという理由から，後者を採用した．

例えば以下のようにして，オブジェクト x(this) に対して，メタオブジェクト $\uparrow x$(m) を生成することができる．

```
MetaObject m = new MetaObject(this);
```

さらに，

```
MetaObject mm =
  new MetaObject(new MetaObject(...));
```

と繰り返すことにより，容易にメタメタ $\cdots$ オブジェクト ($\uparrow\uparrow \cdots x$) を生成し，リフレクティブタワーを形成することができる。

しかし，実際の記述では，各メタオブジェクトの機能を簡単に拡張できるようにするために，クラス定義を間に挟んでいる (図 1 および例)．クラス MetaObject

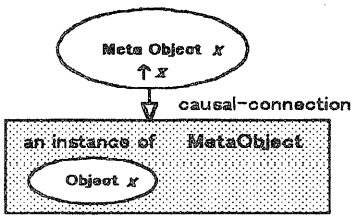

図 1: クラス MataObject の様子

のインスタンスは，オブジェクト x のメソッドおよび，フィールド情報を標準ライブラリ関数を使って得，結果として内部にオブジェクト x を持つことになる．更に，メタオブジェクト $x(\uparrow x)$ を定義し，オブジェクト x を参照することが可能となる．以上を繰り返し，リフレクティブタワーを形成する (図 2)．

クラス MetaObject は，指定したオブジェクトのメソッドリストと，フィールドリストを持つ．その結果 MataObject のインスタンスを生成した後，メソッド mcall，メソッド getField/ setField により，各々メソッド呼出し，フィールドの参照 / 更新を行うことができる．

```
public Object mcall(String name,
    Object[] args);
public Object getField(String name);
public Object setField(String name,
    Object obj);
```

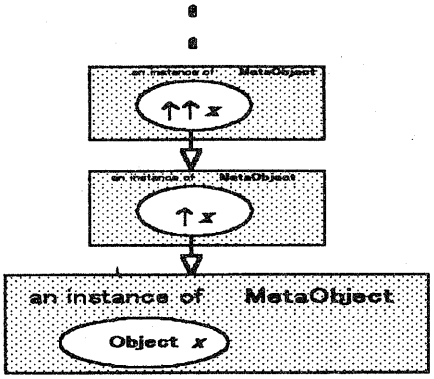

図 2: リフレクティブタワー

2.1 メソッド更新

メソッドの動的更新 (フィールド更新) は，リフレクティブ機能の代表的な機能の一つである．メソッドを動的に更新する場合は，各オブジェクトのメタオブジェクトに対して，どのメソッドを割り当てるかを指定する必要がある (図 3)．この図で明らかな

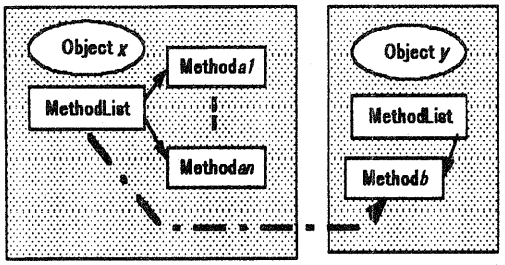

図 3: メソッド更新

ように，メソッドを動的に更新 (追加) する際は，追加するメソッドが属するオブジェクトにメッセージを委譲 (delegate) していることがわかる．

実際のメソッドの追加 / 削除は，クラス MetaObject で定義されているメソッド madd/mdel を使う．

```
public boolean madd(MetaObject obj,
    String name);
public boolean mdel(String name);
```

メソッド madd は，オブジェクト obj のメソッド name を追加し，メソッド mdel は，メソッド name を削除する．

2.2 変数のスコープ

オブジェクトのメソッド更新する際に，メソッドは定義された (元々の) クラスのインスタンス変数のみ参照可能である．実際には，更新する前のメソッドと追加されたメソッドは異なるオブジェクトで実行されているので同じインスタンス変数を参照することはできない。

それを解決するために，我々は，クラス MetaObject にメソッド getField，setField を用意した (図 4)．

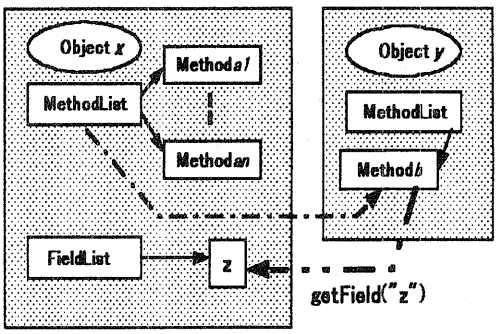

図 4: 変数のスコープ

2.3 動的継承

Java のクラスの継承関係は静的に決定されるが，クラス MetaObject により，任意のオブジェクト x のメソッド，フィールドの情報をメタオブジェクト $\uparrow x$ で操作することにより，動的に継承関係を変更することができる (図 5)．

動的継承は，メソッド更新を応用して実現することができる．親オブジェクトのメソッドすべてを追加することにより，継承と同等の機能を果たすことができる．

3 例

この節では，メソッド更新の例を紹介する．メソッド更新を用いて 100 以下の素数を求めるプログラム (エラトステネスのふるい) は以下のようになる．

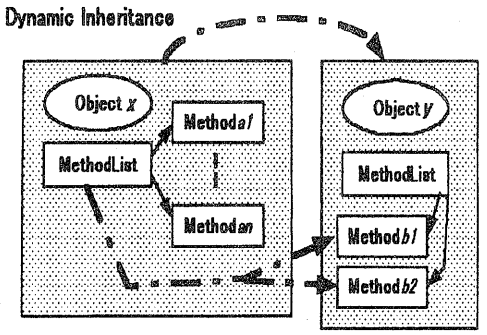

図 5: 動的継承

このプログラムの意味は [1] で紹介されたものである.

```
 1: public class Sieve {
 2:   public MetaObject mo;
 3:   public int prime;
 4:   public Sieve next;
 5:   private MetaSieve meta;
 6:   Sieve() {
 7:     mo = new MetaObject(this);
 8:     meta = new MetaSieve(mo);
 9:   }
10:   public void m(Sieve self, int n) {
11:     prime = n; next = new Sieve();
12:     meta.update();
13:   }
14:   public static void main(String a[]) {
15:     Sieve sieve = new Sieve();
16:     for(int i=2; i<=100; i++) {
17:       Object args[]=new Object[2];
18:       args[0] = sieve;
19:       args[1] = new Integer(i);
20:       sieve.mo.mcall("m",args);
21:     }
22:     System.out.println(
23:       "2'nd: "+sieve.next.prime);
24:     System.out.println(
25:       "3'rd: "+sieve.next.next.prime);
26:     System.out.println("4'th"+
27:       sieve.next.next.next.prime);
28:   }
29: }
```

図 1 のオブジェクト x は, クラス Seieve のインスタンスに相当し, メタオブジェクト $\uparrow x$ は, MetaSieve に相当する. 実際のメソッド更新は, クラス MetaSieve の メソッド update を呼出すことにより処理する. 10 ～ 13 行目のメソッド m は, 動的に更新されるメソッドである. 17 ～ 20 行目 で, 通常のメソッド呼出しとは異なり, メソッド mcall を呼出すことによりメソッド m を呼出している. この mcall により, 更新前のメソッド (通常のメソッド) および, 更新後のメソッドを区別なく呼出すことができる。MetaSieve は以下のようになる.

```
 1: public class MetaSieve {
 2:   private MetaObject mo;
 3:   private MetaObject sieve1;
 4:   MetaSieve(MetaObject i) {
 5:     mo = i;
 6:     sieve1=new MetaObject(new Sieve1());
 7:   }
 8:   public void update() {
 9:     mo.mdel("m");
10:     mo.madd(sieve1,"m");
11:   }
12: }
```

メソッド update によりメソッド更新をしている. 9 行目の メソッド mdel 呼出しによりメソッド m のエントリを削除し, 10 行目で新たに, クラス Sieve1 で定義されているメソッド m に更新している. クラス Sieve1 は以下のようになる. このクラスは更新されるメソッド m のみを定義している.

```
 1: public class Sieve1 {
 2:   public void m(Sieve self, int n1) {
 3:     if((n1 % self.prime) != 0) {
 4:       Object args[] = new Object[2];
 5:       args[0] = self.next;
 6:       args[1] = new Integer(n1);
 6:       self.next.intro.mcall("m",args);
 7:     }
 8:   }
 9: }
```

このプログラムの処理の流れを図示すると以下のようになる (図 6).

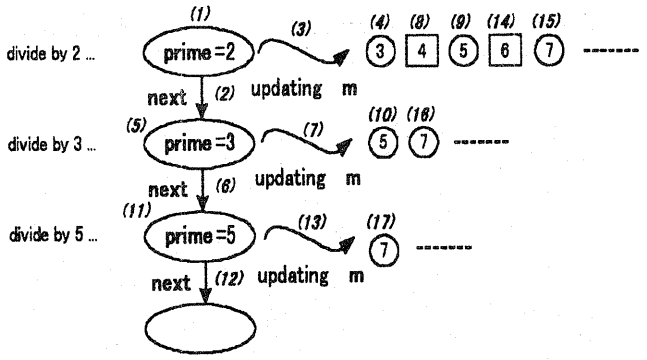

図 6: エラトステネスのふるい

結果としてメソッド main の 18 ～ 20 行目にあるように sieve.next.prime, sieve.next.next.prime とすることにより順に 2 番目, 3 番目の素数を求めることができる. プログラムを実行すると,

```
2'nd: 3
3'rd: 5
4'th: 7
```

となる.

4 仮想空間 Virtual Community

Virtual Community は, 動的に変化するアプリケーションのテストベッドとして現在開発中の仮想空間である [3]. このシステムは, 以下の特徴を持っている.

- マルチユーザシステムである.
- ユーザの化身であるアバタ (avatar) により空間を自由に動き回ることができる.
- 空間は 3 次元部品で構成される.
- チャットによりユーザ間でリアルタイムにコミュニケーションすることができる.
- アプレットとして, ブラウザから起動され, サーバと通信することによりユーザ間でデータを共有することができる.

一方, この空間では様々な, 動的に変化する要素がある. 大きく分けて, 以下の 2 種類に分けることができる.

1. 2 次元部品に関する動的変化

2. 3 次元部品に関する動的変化

2 次元部品とは, 通常, ウインドウシステムが提供する部品でフレーム, ボタンなどである. これらの部品は, Java が標準ライブラリで提供する部品のみでなく, スレッドを部品に割り当てることにより, 動的に変化し続ける部品も考えることができる[1]. 3 次元部品とは, 仮想空間を構成する部品である. 同様に, スレッドを部品に割り当てることにより, 動的に変化し続ける部品も考えることができる. さらに, 動的に追加 / 削除などの様々な変化を考えることができる.

各部品に対して, メタオブジェクトまたは, リフレクティブタワーを形成し, メタオブジェクトのメソッドを呼出すことにより, 部品の挙動 (メソッド) を動的に変化することができる.

以下の節では, 2 次元部品への応用としてフレームの下部に位置するボタンのスレッド化を紹介し, さらに, 3 次元部品への応用を考える.

[1]現在の Java では 3 次元に表現される部品も用意されているが, ここでは 3 次元部品とは, 我々の空間を構成する部品とする.

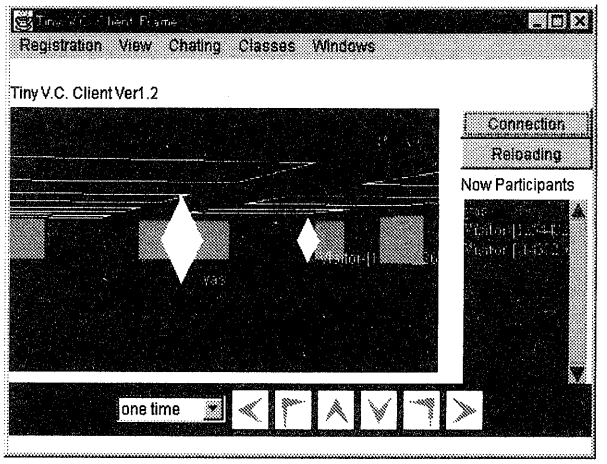

図 7: Virtual Community の様子

4.1 操作ボタンの切り替え

一般に，Java により GUI を構築する際は，標準ライブラリの部品を組み合わせるだけで，構築することができる．我々の仮想空間の GUI の部分のほとんどは，標準のクラスをそのまま用いている．しかし，これらの部品の挙動や，表現を動的に変化することはできない．各部品に対してメタオブジェクトを定義し，そのメソッドを呼び出すことにより，その挙動を動的に変化させることが容易になることが考えられる．図 7 に Virtual Community の様子を示す．この節では，まずフレームの下部にあるボタンを考える．

標準ライブラリとして用意されているボタンは，一度押す毎に，一つのイベントを発生する．イベントを多く発生しなければならない場合は，このような仕様では不便である．ここでのボタンは，アバタが広い空間を動き回るために頻繁にボタンを押す必要がある．ボタンにスレッドを割り当て，ボタンを押している間，イベントが発生するようにすると便利である．また，リフレクティブ機能を用いれば，動的にその挙動を切り替えることができる．以下のように簡単に，そのようなボタンを定義できる．

```
public class EButton extends Canvas
  implements MouseListener {
// インスタンス変数の宣言.  . . .
  EButton() {
// ボタン形状の初期化.  . . .
    m = new MetaObject(this);
    meta = new MetaEButton(m);
  }
  public void mousePressed(MouseEvent e){
// ボタンの色を変化.  . . .
    Object args[];args=new Object[1];
    args[0] = s;
// mcall による呼出し.
    m.mcall("mPressed",args);
    args = new Object[1]; args[0] = g;
    m.mcall("mPaint",args);
  }
// 更新するメソッド mPress
  public void mPressed(Sample3 s) {
    s.l.Rotate(); s.repaint();
  }
  public void mouseReleased(MouseEvent e){
//  ボタン Release の処理.
  }
// 更新するメソッド mReleased
  public void mReleased(MouseEvent e){
  }
  public void paint(Graphics g) {
```

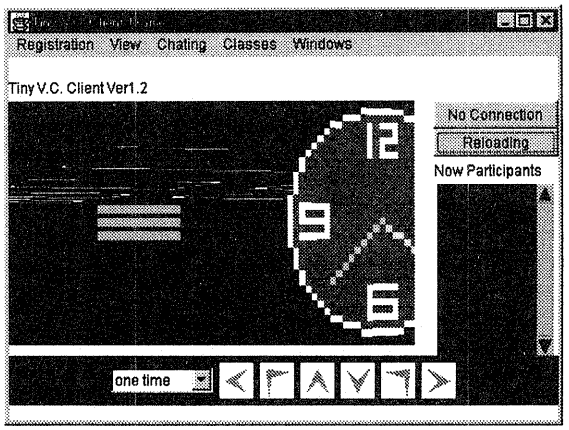

図 8: 壁を動的に変化させる

```
    Object args[] = new Object[1];
    args[0] = g;
// mcall による呼出し.
    m.mcall("mPaint",args);
  }
// 更新するメソッド.  mPaint
  public void mPaint(Graphics g) {
// mPaint の処理.
  }
  . . .
}
```

コンストラクタ EButton によりボタンの形状を初期化すると同時に，メタオブジェクトを生成する．このクラスは，以下のメソッドを更新する．

- mPressed
- mReleased
- mPaint

各メソッドは，イベントなどによりシステムから呼出される mousePressed, mouseReleased, mousePaint を再定義したものである．現在の実現では，更新するメソッドは，メタオブジェクトの mcall を使って呼出す必要があるのでこのようにする必要がある．

各ボタンはその表現と，その挙動が異なる．その違いは，メタオブジェクトを通して動的に行われる．クラス MataEButton の定義は省略する．

4.2 3 次元部品

我々の表現する空間は，全て 3 次元部品から構成されている．例えば，アバタは，サーバからのデータを基に空間を転々と動き回る 3 次元部品として表現される．最も基本的な，壁 (クラス Wall) や天井 (クラス Ceiling) など静的に固定された部品と，スレッドを割り当て動的に変化する部品に分けることができる．どちらの部品も，リフレクティブアーキテクチャを採用することにより動的に変化する部品を実現することができる．例え，静的に固定された部品でも，自分が動き回ることにより，その部品の見方が変わるため，その表現を変えなければならない．例えば，図 8 にあるように，奥にある壁は，ある一定以上の距離にあるために，その形が変化した壁である．このような場合は，部品に対して制約 (constraint) プログラミングを採用することが考えられるが，一般にリフレクションを用いた方が，柔軟性が高い表現が可能である．

我々の仮想空間では，現在 10 種類の 3 次元部品が設定され，ソースコードを閲覧したり，ネットワークを介して動的に部品を追加 / 削除可能である．

5 議論と今後の課題

本研究では，任意の Java オブジェクトにメタオブジェクトを定義可能にし，各オブジェクトのメソッド更新または，動的継承を可能にした．特に，動的に変化するアプリケーションとして仮想空間を考え，それへの応用を考えた．我々が考える部品は，2 次元部品と，3 次元部品に分け別々に扱った．また，それらの部品にスレッドを割り当て，動的に変化する部品とすることができた．動的に挙動を変化させるオブジェクトには，そのオブジェクトのメタオブジェクトを生成し，メタオブジェクトのメソッドを呼出すことによりオブジェクトの挙動を変化させる．

更なる発展として，コンポーネントウエアへの応用が考えられる．Java に対応した代表的なコンポーネントウエアとして JavaBeans [4] があるが，本論文で紹介したリフレクティブアーキテクチャを容易に適用可能である．本研究では，トランスレータを用いずにリフレクティブアーキテクチャを導入したが，JavaBeans のコード生成機能を修正し，用いることにより，簡単にリフレクティブ機能を取り入れることができる。また，ここで，動的に更新するメソッドは `mcall` を用い呼び出しているが，その問題も解決される．しかし，その際の問題点も幾つかあげることができる．

- JavaBeans の Bean には，永続性が要求される．しかし、我々の部品は，スレッドを割り当て，動的に変化する部品を主に，考えているため，永続性を保つ機能が必要である．
- 分散処理への対応する．

どちらの問題も，これまでのリフレクションに関する研究で議論されてきた．基本的には，各 Bean に対して，それを解釈し実行する簡単なインタプリタを実現することが考えられる．また，並列オブジェクト指向言語へのリフレクティブアーキテクチャ導入で議論されてきたメッセージキュー (JavaBeans では，イベントキュー) をメタオブジェクトに持たせることにより，同期に対する柔軟的な対応が可能になる．以上のような考察のもと現在開発中である．

現在，リフレクティブ機能を提供するライブラリの一つであるクラス `MetaObject` および，その機能に対する性能評価はまだ行っていない．これらも今後の課題である．

参考文献

[1] Abadi, M. and Cardelli: A Theory of Objects, *Springer-Verlag*, 1996.

[2] Chiba, S. and Masuda, T.: Designing an Extensible Distributed Language with a Meta-Lebel Architecture, *In Proc. of the 7th European Conference on Object-Oriented Programming*, LNCS 707, Springer-Verlag, 1993, pp. 482–501.

[3] Kodama, Y: Virtual Community, *In Proc. of International Symposium on Future Software Technology 97*, 1997, pp.241–246.

[4] Robert Englander: Developing Java Beans, *O'Reilly and Associates, Inc.*, 1997.

[5] Smith, B. C.: Reflection and Semantics in LISP, *Technical Report CSLI-84-8, Stanford University Center for the Study of Language and Information*, 1984.

[6] Watanabe, T. and Yonezawa, A.: Reflection in an Object-Oriented Concurrent Language, *In ABCL: An Object-Oriented Concurrent System, Yonezawa ed.*, MIT Press, 1990, pp. 45–70.

[7] 千葉滋，益田隆司: 自己反映言語 Open C++ とその分散処理への適用の実際，コンピュータソフトウエア, Vol.11, No.3, 1994, pp.33–48.

[8] 増原英彦，松岡聡，渡部卓雄: 自己反映並列オブジェクト指向言語 ABCL/R2 の設計と実現，コンピュータソフトウエア, Vol.11, No.3, 1994, pp.15–32.

[9] 立掘道昭，千葉滋，中田育男: Java 言語のための新たな自己反映機構の提案，日本ソフトウエア科学会第 14 回大会論文集，1997, pp.201–204.