ソフトウェア工学 125－2
（2000．1．14）

Javaにおけるクラス定義の動的交換システムの構築

野上 耕介，山口 実靖，相田 仁，齊藤 忠夫

東京大学大学院工学系研究科

ソフトウェアシステムの拡張作業や修正作業を、そのシステムを停止させることなく実現できるシステムを現在、研究している。今回、提案する方式は、動的にそのクラス定義を変更する方法であり、それによりソフトウェアのバージョンアップという形での拡張などが実現可能となっている。実装言語としては、Java言語を採用しており、Pure Javaで、書かれているため、プラットホームに依存することはないシステムである。本稿では、提案方式の簡単な概要について述べ、その方式の特徴などについて述べる。

Kosuke Noagmi , Saneyasu Yamaguchi , Hitoshi Aida , Tadao Saito

—Department Information And Communication Engineering, Faculty of Engineering, Univers ity of Tokyo

Dynamic Exchange of Class Definition with Java

Abstract

In the recent years, network technology and infrastructure have developed tremendously and new services have been proposed.
However, in traditional network management schemes, it might be necessary to stop and even redesign the whole system to perform tasks such as reconfiguring services launched on start-up or adding new services.
In this paper, we propose a method to solve these problems. Morever, in addition to providing extended network management functionality, our proposal also enables upgrades in the form of dynamic extension of application functionality.

キーワード: クラス定義 動的交換 Java言語 継承関係
(Class Definition , Dynamic Exchange , Java Language , Inheritance)

1 はじめに

銀行の大規模なオンラインシステムや電話交換機システムなどの各種のサーバは、その一時的な停止であっても、ユーザへのサービスの提供におおきな障害をもたらす。また、ユーザへのサービスだけではなく、システム全体の停止に伴って変更対象以外のシステムに影響を与えたり、システムの再構築といったことも必要になったりする。そのため、システム上で動作しているソフトウェアシステムの拡張作業や修正作業を、そのシステムの実行を停止させることなく行える技術の開発が望まれている。

この要求に応えるために、システムが持つ機能をシステムを停止させることなく、動的に変更することが可能となるシステムを異なる2つのアプローチで提案し、それぞれについて、実装を行った。

また、今回、提案するシステムを使用すれば、アプリケーションの動的機能拡張という形式でのアップグレードといったことも可能になる。

今回、提案するシステムを実装する言語としてJava^{TM*}を選択した。

2 提案方式の概要

2.1 実装方針

本稿で提案するシステムは、オブジェクト指向の概念を基に考案したものである。オブジェクト指向では、多重継承言語や単一継承の言語があるが、今回は、ランタイム時の動的な交換に伴う自己同一性を失わないように単一継承言語に限定して考える。

オブジェクト指向言語は、クラスと呼ばれる単位で構成される。提案システムに関して言えば、その機能を動的に交換するということは、つまり、そのプログラムの振舞いを決定しているクラス定義を動的に交換するということと同一である。

提案するシステムの概要を述べていくが、前提概念として、交換可能なクラスは、親子関係にあるクラスのみとする。これは、動的に交換することにより、その意味的な同一性を保証するためである。また、この制限を設けることにより、オブジェクト指向の継承の恩恵を受けることもできる。つまり、交換したいクラスは、交換したいメソッドの部分だけを記述すればよいということになるのである。

この前提の上で、2つの異なるアプローチで、クラス定義の動的交換システムを構築した。

本稿では、まず、2つの提案システムの概要について述べ、それぞれの方式の特徴や機能の比較などを行う。さらに、それぞれの実装についても述べる。

2.2 代理クラス方式

クラスの動的交換を実現するための方法として、「代理クラス (Proxy Class)」を用いたものを提案する。この方式では、ユーザプログラムとクラスライブラリを直接リンクさせずに、代理クラスを通してリンクさせるものである。(図 1)

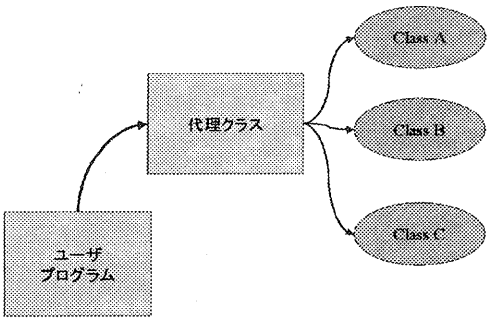

図 1: Proxy Class

代理クラスは、ユーザプログラムから呼ばれると、現在選択されているクラスに対してメッセージを投げる。これにより、メソッドを起動させることができる。この代理クラスを用いて、提案方式を実現しているが、これ以外にも、次のような機能を現在のところ用意している。

- public な フィールド値の継承
  クラスを交換しても、そのフィールド値は"public" なものに限り、その時点で所有する値がインスンスごとに引き渡される。
- テストラン機能
  交換したクラスが正常に動作するとは限らないので、交換されるクラスと交換するクラス

*Java は、米国 SunMicrosystems,Inc. の米国およびその他の国における商標である。

の両方にメソッドを投げるテストラン機能を実現している。ただし、その出力は、ファイルにその結果のみを出力するように現在は実装されている。

2.3 クラス定義の動的変更可能な ORB システムの構築

クラス定義を動的に交換することを可能とするシステムの異なるアプローチとして、サーバ側のクラス定義のみ交換可能な ORB システムを構築した。この ORB システムは、JAVA 言語において、Sun Microsystems が提供している RMI とほぼ同じシステムである。ここで、簡単に ORB のシステムについて説明する。ORB システムは、図 2のような形で形成される。

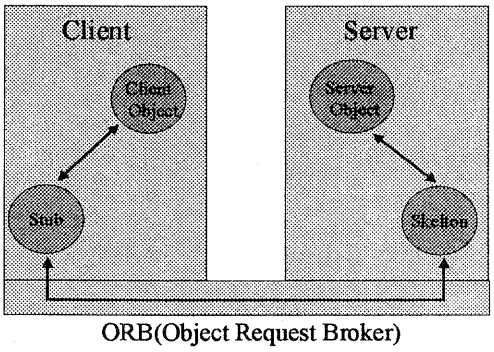

図 2: ORB System

クライアントがメソッド起動要求を投げると、その要求が図 2にあるように Stub,Skelton を経由し、サーバ側クラスにアクセスし、メソッドを起動することになる。その返値を受取り、逆に Skelton,Stub を経由し、クライアントにその結果を返す。これが ORB システムの概要である。提案する ORB システムは、概要としては、上に述べた ORB システムと同じであり、サーバ側の機能を拡張したものである。この拡張機能により、サーバ側に置かれるクラスのクラス定義を動的に変更することができるような仕様となっている。このシステムの特徴を簡単に述べる。

- サーバ側のクラス交換が可能である。
- サーバ側には、インスタンスを作成したいクラスを置いておくだけで済む。つまり、特別なサーバ用のプログラムを書く必要はなく、クライアント側の要求に従い、インスタンスを作成することが可能である。
- クライアント側は、通常のソースコードの記述とほぼ同じ記述で済む。(インスタンスの参照の獲得もしくは生成に関する部分のみが通常の記述とは異なる)
- クラスの交換は、サーバ側、クライアント側、どちらからでも指示可能である。
- 代理クラス方式と同じように、インスタンスのフィールド値の継承が可能である。

などの特徴がある。

3 実装に関して

3.1 Java 言語を採用した理由

実装の説明に入る前に、Java 言語を実装言語として選択した理由について簡単に述べる。まず Java 言語が、オブジェクト指向言語であり、かつそのクラスライブラリが豊富なことが挙げられる。また、組み込み機器やサーバ関連といった様々な分野で Java 言語が採用されはじめたことから今後 Java 言語の需要がますます増えるであろうと予測されるからである。

3.2 代理クラス方式

上に述べた実現方式に沿って、まず代理クラス方式に関しての実装を行った。
まず、この方式において中心となる代理クラスについて述べる。Java 言語で実装する上で、この代理クラスは、イベント関連の部分と、スレッド関連の部分に関して、別の代理クラスを用意している。これは、イベント関連クラスが、イベントを受け取るとメッセージを自動的に発信するという他のクラスとは異なる特殊なクラスであるためであり、また、スレッドに関しても、単純に交換するというわけにはいかないため、別の代理クラスを用意した。
代理クラスは、その中に、メソッド情報を格納したテーブルを有している。このテーブルは、インスタンスを作成するたびに、その代理クラス内部

に作成され、そのクラスが持つメソッド情報をそのメソッド名をキーとして登録していく。クラスの交換が行われると、このメソッド情報格納テーブルが更新されることになる。
ユーザプログラムがメソッドを起動すると、このメソッド情報格納テーブルを、参照し、その起動すべきメソッドを検索する。メソッドが見つかったら、引数を与えてメソッドを起動するのであるが、この際、Java 言語のリフレクションと呼ばれる機能を利用している。上に述べた通り、イベント関連クラスやスレッド関連のクラスに関しては、別の代理クラスが用意されているが、両クラスとも、この代理クラスを継承しているので、基本的な構造は同じである。異なる部分は、イベント関連用の代理クラスだとその実装するインタフェースによって必ず呼ばれるメソッドがあるのでその部分を実装した点であり、スレッド関連の代理クラスに関しては、交換の際、スレッドを一度停止し、クラスの交換を行った後、再びスレッドを開始するという実装をおこなっている。
また、クラス交換の際であるが、フィールド値の継承も行う。これも、リフレクション機能を利用している。そのため、public なフィールド値の継承しか行われない仕様となっている。このフィールド値を継承する機能を有したクラスは、代理クラスに内包されている。
テストラン機能を実装しているが、これは、内部のラッパークラス内に、交換されるクラスと交換するクラス、それぞれのインスタンスを保持し、それぞれにメソッド起動要求などのメッセージを投げることにより、実現している。しかし、メソッド起動のたびに、2 つの返値が返って来るため、その扱いに関しては、現在、ファイル出力するという形式を採用している。また、テストラン機能からの復帰という点では、「交換前の状態に戻す」と「クラスを交換し作業を続ける」という 2 つの方法を用意している。

3.3 代理クラス手法の動作

その動作を簡単に述べる。

1. インスタンス作成
   AccessServer.class のインスタンスを作成する。これが、アプリケーション作成の手助けを行う。AccessServer.class のインスタンスを利用して、交換したいクラスのインスタンスを作成する。ここで作成されるインスタンスは、そのクラスのラッパーのように機能する。このインスタンス作成時に、AccessServer.class が持っているテーブルに様々な情報が格納される。具体的には、現在、どういったインスタンスが存在しているかといった情報を持つテーブルやクラスの継承関係などを元に構築されるテーブルなどである。(図 3,4)

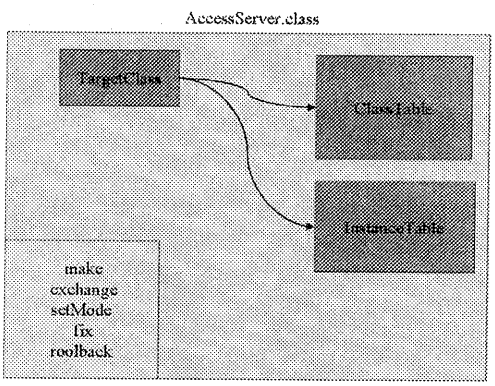

図 3: AccessServer の中身

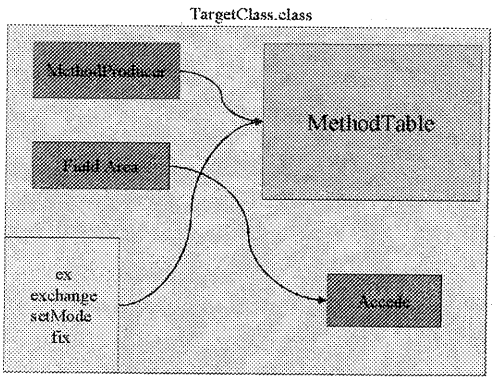

図 4: TargetClass の中身

2. メソッド起動
   作成したインスタンスのメソッドを起動させたい場合は、上で作成したラッパークラスに向かって、メッセージを投げる。この際、

ラッパークラスの中にあるメソッド情報の格納されたテーブルで、起動するメソッドを検索し、メソッドが見つかったらそのメソッドを起動する。(図5)

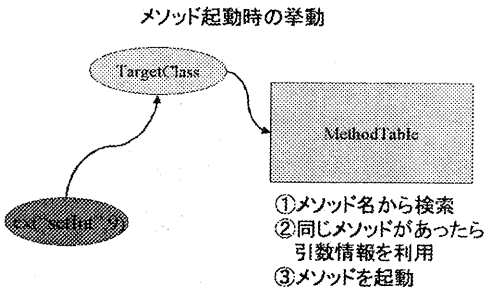

図 5: メソッド起動の際の振舞

3. クラス交換
クラスを交換する際ですが、スレッドの場合のみ、異なる挙動をとる。まずは、一般のクラスの挙動について述べる。(図6)

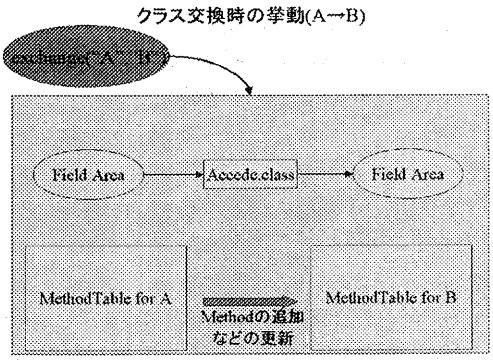

図 6: クラス交換の際の挙動

- 一般クラス
クラス交換の際に、まず、フィールド値の継承、メソッドの継承をおこなう必要があり、またオーバーライドされたメソッドに関しては、その情報を更新する必要がある。以上のことを実現するため、上に述べたラッパークラスに、その機能を実装してある。フィールド値の引渡しに関しては、上にも述べた通り、public なフィールド値のみ引渡しを行う。メソッドの継承については、テーブルごと継承していくことで先祖クラスのメソッドを継承していく。オーバーライドされたメソッドに関しては、その引数情報をもとに、メソッド情報格納テーブルにそのメソッドが存在すれば情報を上書きし、テーブルにないメソッドに関しては、そのメソッド情報を追加する。こうすることで、一般クラスのクラス交換を実現している。

- スレッドクラス
スレッドに関しては、一般のクラスとは異なり、それまでになんらかの動作を行っている可能性があるので、一度その動作を停止させる必要がある。そのため、ラッパークラスを継承し、スレッド用のラッパークラスを作成している。この中で、スレッドを停止し、一般のクラスと同じように、フィールド値、メソッド情報に関しての操作を行い、再びスレッドを開始させる機能を実装している。

一般のクラスとスレッドクラスに関しての交換時の挙動を示したが、これは、インスタンスごとにわざわざユーザが指示して行うのではなく、最初に作成した AccessServer.class のインスタンスに対して、交換したいクラス名を指定するだけで、その親クラスのインスタンスを内包するテーブルを参照し、自動的にクラスの交換を行うように実装してある。

3.4 ORB 方式の実装について

Java 言語に実装されている Serialize 機能と Reflection 機能を利用して実装している。提案システムで提供するクラスでユーザが利用するのは、次のクラスである。

- Server.class
クラス交換機能などを内包しているサーバ。サーバ側では、このクラスを実行するだけでよい。

- ORB.class

サーバへの接続を行い、インスタンスの生成および参照の獲得を行うクラスである。

- InterfaceMaker.class
  クライアントでは、サーバ側のインスタンスの参照を受け取り、インタフェースでキャストする必要がある。サーバ側のクラスからインタフェースを作成できるようにインタフェースを自動生成してくれるクラスである。

- StubGenerator.class
  クライアント側では、Stubを作成する必要があるので、Stubを自動生成するStubジェネレータを用意した。

この中で、ORBの中核となるサーバクラスとスタブクラスについて、解説する。
スタブクラスであるが、内部にサーバとのコネクションおよぶオブジェクトのシリアライズなどを行うクラスを内包している。具体的な動作は、図7に示す。

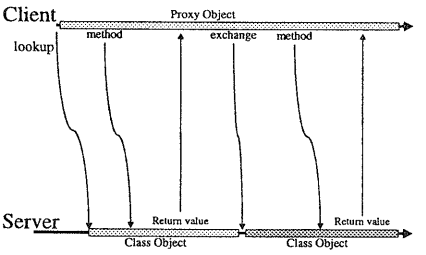

図 7: ORBの動作

サーバクラスであるが、その内部で、動的にインスタンスの入れ換えを行える機能を実装している。クライアント側からメソッドの起動要求がくると、常に最新のインスタンスを参照する。この機構により、クラス定義の動的交換をおこなっている。また、クラス交換の際、フィールド値の引渡しをすることも可能である。インスタンスの生成に関しても、サーバ側で起動時に生成しておくだけでなく、クライアント側の要求によっても、生成可能である。

クライアントおよびサーバの動作の概略を図8に示し、また、クライアント側の簡単なソースコードを載せ、動作の説明を行う。

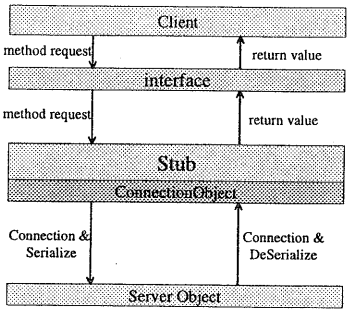

図 8: Client & Server

```
public class ClientSample{
    public static void main(String args[]){
        ORB orb = new ORB(hostname,port);
        Intf target = (Intf)orb.
                lookup("Command.class","target");
        target.method(args);
        target.exchange("NewCommand.class");
        target.methdo(args);
    }
}
```

1. サーバとの通信を行うために、ORBクラスのインスタンスを作成する。
2. lookup() メソッドを利用して、インスタンスをサーバ上に作成。クライアント側は、獲得した参照をインタフェースでキャストする。
3. 得られた参照に対して、メソッドを起動する。
4. ORBに対して。クラスの交換を指示する。
5. 参照に対して、メソッドを起動すると、新しいクラスのメソッドが起動され、その返値が得られる。

上に説明したような動作を行うが、交換したクラスに新しいメソッドが追加されており、それを利用したい場合は、そのクラスのインタフェース、およびスタブクラスを獲得し、クライアント側のコードを書き換える必要がある。

3.5 提案手法の比較

本稿で提案している2つのシステムについて簡単な比較を行ってみる。

- 適用範囲
  提案手法は、どちらも100% Javaで書かれているため、どのプラットフォームでも利用可能である。代理クラス方式では、クライアント側のアプリケーション構築やサーバ側のシステムを構築することも可能であるが、一方、ORB方式では、サーバ側のクラスのみが対象となっている点がことなる異なる。ただ、下で比較するが、システムの目的によって選択する必要がある。
- 実装の容易さ
  代理クラス方式では、交換したいクラスに関しては、そのクラスを継承し、追加したいメソッドやオーバーライドしたいメソッドを記述するという制限があるのみで、それ以外は、通常のクラス作成と違いはないため、交換クラスに関しての実装は容易であるが、それを利用するアプリケーションの記述が複雑になっている。この問題に関しては、トランスレータを用意することで、その問題を回避している。
  またORB方式では、交換したいクラスは、代理クラス方式と同様に、そのクラスを継承するだけでいいので実装は用意である。また、クライアント側は、新しいAPIを利用したければ、新しいAPIの定義されている新しいクライアント用のインタフェースを利用する必要があるため、クライアント用のコードを書き直す必要がある。新しいAPI用のインタフェースを定義する煩わしさから解放されるために、インタフェースを作成してくれるインタフェースメーカを用意した。
- 機能の豊富さ
  代理クラス方式では、フィールド値の受渡しが可能であったり、テストラン機能など機能的に豊富である。また、ORB方式でも、フィールド値の受渡しが可能などの機能を有している。
- 性能
  データを示すが、代理クラス手法は、メソッド起動にリフレクション機能を利用しているため、メソッド起動が、通常のメソッド起動と比較して、非常に遅い。一方、ORB方式では、ネットワークの遅延などがあるため、その性能測定が難しい。よって、ここでは数値測定などは行わないこととする。

3.6 性能評価

代理クラス方式は、メソッド起動に非常に時間を要します。そのデータを表3.6に示す。データは、単純に、数値を設定するメソッドと数値を獲得するメソッドのみで測定している。測定は、JRE1.2 SunEnterprise450 Memory 1024M メソッド起動回数100000回 という条件で行った。

	set メソッド	get メソッド
通常システム (ms)	8	7
提案システム (ms)	1974	1719

- 適用例本稿で提案したシステムが適用できそうな例について考えてみる。たとえば、ある金融関係のシステムで、消費税率が変更になったといった場合に、その消費税率の部分に関するメソッドを交換することで、新しい消費税率に対応することが可能になるといった適用例が考えられる。

4 まとめと今後の課題

本稿では、Javaにおけるクラス定義の動的交換システムの構築ということで、異なる2つのアプローチを提案し、その実装に関して発表した。今後の課題としては、クラス交換のタイミングなどに関しては、考慮していないので、クラスを交換しても何の問題も起こらない様に改良していこうと考えている。