メタレベル ・アーキテクチャを利用した オブジェクト管理システムのバージョン管理機構の構成法

Organization of a Version Management Mechanism for an Object Management System with Metalevel Architecture

藤枝 和宏　　落水 浩一郎
Kazuhiro FUJIEDA　　Koichiro OCHIMIZU

北陸先端科学技術大学院大学 情報科学研究科
School of Information Science, Japan Advanced Institute of Science and Technology

概 要

オブジェクト指向言語の実行環境のカスタマイズを可能にする機構—メタレベル ・アーキテクチャが、オブジェクト管理システム (OMS) の実現に有効であることを論じる。特に OMS のバージョン管理機構の実装にメタレベル ・アーキテクチャを利用して、組み込みのバージョン管理機構が持つような管理機構の透過性を、アプリケーションレベルで記述したバージョン管理機構に与えることについて検討する。

1 はじめに

オブジェクト管理システム (Object Management System—OMS) は、ソフトウェア開発に用いられるツールと生じる生成物 (オブジェクト) を格納、管理するためのデータベースシステムである。OMS に対する要件は数多く、データモデル、管理機構、アプリケーションインターフェイスなどについてさまざまな研究が行われている。

OMS を実現する一つの方法は既存のオブジェクト指向データベース管理システム (OODBMS) をベースにするものである。OODBMS は複雑な階層構造を持つオブジェクトの記述力やロングトランザクションのサポートなど OMS として有効な性質を多く備えている [3]。しかし、既存の OODBMS にはデータモデルや管理機構の面で OMS として不十分な点があるため、それを補うシステムを追加したり、システムを拡張したりする必要がある [4] [7]。

さらに、OODBMS を OMS として利用する場合にはクラスが生成物やツールの記述の他に、管理スキーマや開発プロセスの記述にも用いられる。そのため、クラス定義をデータベース上で管理できること、クラス定義の修正が容易であることは重要である。しかし、既存の OODBMS のスキーマ管理機構の多くはこの要求を満たしてはいない。

そこで、私達はデータモデルの拡張や管理機構の拡張を容易にしたり、強力なスキーマ管理機構を実現するために、OMS の実装と OMS 上の記述言語にメタレベル ・アーキテクチャ(以降メタアーキテクチャと略記する) を持つオブジェクト指向言語を利用することを提案し、基礎的な考察を行った [9]。

本論文では、OMS の管理機構の実装におけるメタアーキテクチャの有効性をバージョン管理機構を例にして具体的に考察する。以下、メタアーキテクチャを持つオブジェクト指向言語の特徴と、OMS に利用することの利点を述べ、次に OMS のバージョン管理機構の問題点とメタアーキテクチャを用いた解決策について述べる。

2 メタレベル ・アーキテクチャ

オブジェクト指向言語の実行環境を言語自身のオブジェクトで表現し、そのオブジェクトをアプリケーションから操作したり、置き換えたりできるようにインターフェイスを公開することで、アプリケーションに応じた実行環境のカスタマイズを容易にするアーキテクチャをメタレベル ・アーキテクチャと呼ぶ。メタアーキテクチャの構成は、その用途や言語によってさまざまである [8]。

2.1 メタクラスを持つメタアーキテクチャ

本論文では、クラスやメソッドなどの言語の構成要素が自分自身のオブジェクトで表現されていて、その振る舞いがより上位のクラスで定義されている言語を利用する。このタイプの言語の代表的なものに CLOS(Common Lisp Object System) [6] と

Dylan [1] がある。

言語の構成要素を表現するオブジェクトをメタオブジェクトと呼び、その振る舞いを定義するクラスをメタクラスと呼ぶ (図 1)。メタクラスが定義しているのは、インスタンス生成や継承、メソッドディスパッチなどの言語処理系そのものの振る舞いである。

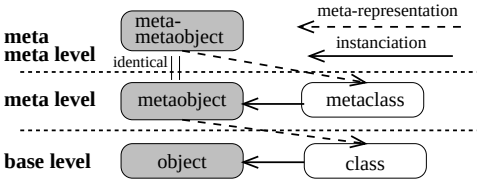

図 1: メタクラスを持つメタレベル ·アーキテクチャ

このようなアーキテクチャを持つ言語の利点として以下の 2 点が挙げられる (図 2)。

1. メタオブジェクトにアクセスすることでクラスやメソッドに関する情報を取得できる
2. メタクラスを継承を利用して拡張することで言語処理系の拡張が容易に行える

さらに、このような言語ではクラスやメソッドがオブジェクトなので、クラスやメソッドの動的な追加、削除、変更の実現が容易であり、CLOS や Dylan ではこれらの操作が言語仕様に含まれている。

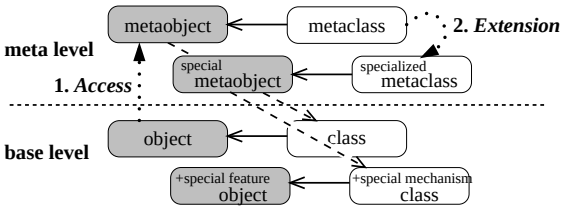

図 2: メタクラスの利用方法

3 メタアーキテクチャを利用した OMS

OODBMS の実装方法の一つに、既存のオブジェクト指向言語の言語仕様と実行環境を ad-hoc に拡張して、DBMS として必要な機能を提供する backend のシステムと統合する方法がある。この方法では、backend のシステムのインターフェイスが実行環境に埋め込まれてしまうため、既存のシステムのインターフェイスを利用した新たな機能の記述や、システムの追加、変更などが困難である。これは OMS として必要な機能を追加する際の障害になる。

backend のシステムと統合する言語としてメタアーキテクチャを持つオブジェクト指向言語を用いれば、メタクラスのカスタマイズを利用して統合に必要な言語仕様や実行環境の拡張を行うことができる。実際に、CLOS の MOP を利用してオブジェクトの永続性を実装したシステムとして PCLOS [5] がある。

メタクラスを持つ言語ではクラスがオブジェクトなので、スキーマ管理のための特別なデータベースではなく、クラスをオブジェクトと同じデータベースで管理することが可能である。クラスに対する操作もスキーマデータベースのための特別なインターフェイスではなく、通常のオブジェクトに対する操作と同様に行うことができる。この性質を利用すれば、OMS のためのスキーマ管理機構を実現することができる。

また、メタクラスのカスタマイズを利用するとデータモデルの拡張も可能である。これは OMS のデータモデルに必要な関連や制約の記述を容易にするためのデータモデルの拡張に利用できる。

このようにメタアーキテクチャを持つオブジェクト指向言語を利用してデータベースシステムとして基本的な部分を実装した上で、メタアーキテクチャを利用して OMS として必要な管理機構やデータモデルを導入することで、非常に拡張性に富んだ OMS を実現することができる。

次に実例として、OMS のバージョン管理機構の実装を取り上げてメタアーキテクチャの利用方法とその利点を具体的に示す。

4 OMS のバージョン管理機構

オブジェクトのバージョン管理機構が OMS に必要な管理機構であることは、すでに一般的な認識になっている。OMS の管理するオブジェクトでは、オブジェクトの各バージョンに付随させる情報や、バージョン管理の対象となるオブジェクトの粒度が、プロジェクトの規模や管理されるオブジェクトの性質によって異なっている。したがって、OMS のバージョン管理機構はこのような多様性に対応できなければならない。

バージョン管理機構の実装方法には、システムに組み込みの機構として実装する方法と、OMS の記述言語を利用してアプリケーションとして実装する方法の 2 通りある [2]。一般に OODBMS がシステムに組み込みの機構として提供しているバージョン管理機構は、管理単位が各オブジェクトごとであり、各バージョンに付随する情報などもすべてのオブジェクトについて同じである。

このような OODBMS を OMS として利用するためには、組み込みの機構を利用するか、あるいはまったく独自にアプリケーションレベルでバージョン管理機構を記述する必要がある。

アプリケーションレベルでバージョン管理を記述する場合には、管理機構の柔軟性を損なわないようにするために、以下のような管理機構の透過性に注意して記述しなければならない。

- 実装の透過性

同じクラスのインスタンスについて、管理機構で管理されているインスタンスと、管理されていないインスタンスにアプリケーションから全く同等にアクセスできる。

- インターフェイスの透過性
  定義済みのクラスも含めて任意のクラスに管理機構を適用できる。また、管理されるクラス、あるいは管理機構に合わせて特別なクラスを定義する必要がない。

このような透過性は、システムに組み込みの管理機構であれば、言語仕様や実行環境を拡張することで容易に満たすことができる。実際に、既存のOODBMS の管理機構はこれらの条件を満たすように設計されている。

5 メタアーキテクチャによるバージョン管理機構の実装

OMS 上の記述言語がメタアーキテクチャを持つオブジェクト指向言語であれば、アプリケーションとして記述したOMS のバージョン管理機構に、組み込みの管理機構の持つような透過性を持たせることができる。ここでは、記述言語としてCLOS を用いてその方法を示す。

5.1 バージョン管理機構の実装

オブジェクトのバージョン管理機構は、ユーザに指定された時点でインスタンスのコピーを作成し、インスタンス間の派生関係を表すグラフ構造で管理する (図 3)。

各バージョンのインスタンスには、バージョン番号や日付などの情報を付加する必要があり、バージョン管理されているインスタンスには、現在のバージョン番号などの管理情報を付加する必要がある。また、バージョン管理されているインスタンスについて、新しいバージョンを発生させる、古いバージョンのインスタンスを取り出すなどの操作を実行できなければならない。

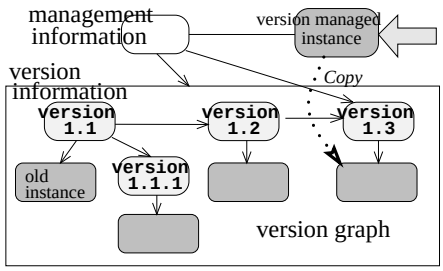

図 3:バージョン管理機構の概念図

上で示した機構をオブジェクト指向言語で実装するのは非常に容易である。ただし、管理機構の柔軟性や管理機構の透過性を損なわないようにするためには

- バージョン管理されているインスタンスにどのように管理情報を付加するか
- 新しいバージョンを発生させる、古いバージョンを取り出すなどの操作を行なうメソッドをどのクラスに持たせて、そのように実行するか

について考慮する必要がある。

例えば、必要な管理情報と操作を一つのクラスで定義して、バージョン管理が必要なクラスに継承させるという実装も可能である。しかし、この方法ではクラスを定義した時点でそのクラスのすべてのインスタンスの管理方式が決ってしまうため、管理機構の柔軟性と透過性が損なわれてしまう。

5.2 CLOS による実装

CLOS ではクラス定義の変更を容易に行なうことができる。しかし、変更の影響は変更したクラスのすべてのインスタンスの他に、すべてのサブクラスの定義およびそのインスタンスに及ぶため、継承を利用した実装は現実的ではない。

5.2.1 インスタンスに管理機構を付加する

バージョン管理機構は、ひとつのクラスの一部のインスタンスを対象に適用されることが多い。クラス定義とは独立にインスタンスに管理情報や管理機構へのインターフェイスを提供することが可能なら、管理機構の透過性を高めることができるはずである。

CLOS では任意のインスタンスに対して動的にメソッドを付加することができる。これを利用して、管理機構のインターフェイスを管理する必要があるインスタンスだけに設定することが可能である。また、それぞれのインスタンスに付加されたメソッドはオブジェクトであり、インスタンスと一対一に対応しているので、管理情報をメソッドに持たせることも可能である。

任意のクラスのインスタンスに適用可能で、適用されたインスタンスをバージョン管理可能にするメソッド `make-versioned` の定義を以下に示す。

```
 1:(defmethod make-versioned (obj)
 2:  (let ((info (make-instance
 3:                'management-information
 4:                ...)))
 5:    (defmethod check-in
 6:      ((inst (eql obj))(tag version-tag)
 7:       &optional management-info info)
 8:       メソッドの実行部)
 9:    (defmethod check-out
10:      ((inst (eql obj))(tag version-tag)
11:       &optional management-info info)
12:       メソッドの実行部)
13:    ...))
```

2~4 行で管理情報を定義したクラス `management-information` のインスタンスを生成し、引数を介して `check-in`,`check-out` の両方のメソッドに共有させている。この二つのメソッドの実行では、管理されているインスタンスを変数 `inst` で、管理情報を変数 `management-info` で参照することができる。

この方法は管理機構の透過性という点では優れているが、実行部を含むメソッドオブジェクトが、インスタンスに一つずつ付加されるため、メモリ効率 (ディスク効率) が非常に悪い。また、完全にインスタンス単位で管理されるため、管理機構の適用状況を管理するためには、さらに工夫が必要である。

5.2.2 隠しスロットで管理情報を保持する

一つのインスタンスに一つずつ管理情報を付加する最も簡単な方法は、クラスで一つ余計にスロットを定義しておくことである。しかし、普通にスロットを追加したのではサブクラスに継承されてしまうので、管理機構が適用されないインスタンスにまで余計なスロットが付加されてしまう。

メタクラスのカスタマイズで言語仕様を拡張すると、サブクラスに継承されないスロットを定義することもできる。そのスロットを利用して管理情報とインターフェイスをインスタンスに対応づけることもできる。以下に、定義したクラスに継承されないメソッドが必ずひとつ付加されるメタクラス `hidden-slot-metaclass` の定義を示す。

```
1:(defclass hidden-slot-metaclass
2:  (standard-class) ())
3:(defmethod initialize-instance :around
4:  ((class hidden-slot-metaclass)
5:   &rest all-keys &key direct-slots)
6:  (let*
7:   ((slotd (append direct-slots
8:       '((:name ++hidden++
9:        :readers (hidden-slot)
10:       :writers ((setf hidden-slot))))))
11:   (instance
12:    (apply #'call-next-method class
13:       :direct-slots slotd
14:       all-keys)))
15:    (setf (class-direct-slots class)
16:     (remove-if #'(lambda (slotd)
17:       (eq (slot-definition-name slotd)
18:           '++hidden++))
19:      (class-direct-slots class)))
20:   instance))
21:
22:(defclass artifact ()
23:  スロットの定義...
24:  (:metaclass hidden-slot-metaclass))
```

メタクラスのメソッド `initialize-instance` は、インスタンスとしてクラスが生成されたときに、クラスが持つ情報を設定するためにインスタンスを生成したメソッドから必ず呼び出される。上の例では、このメソッドを override して、最初に隠しスロットを加えて必要な情報を設定したあと (7 ~ 10)、継承されたときに用いられる情報から隠しスロットの情報を削除している (15 ~ 19)。また、カスタマイズしたメタクラスを利用する場合には、クラスを定義する際に明示的にメタクラスを指定する必要がある (22 ~ 24)。

5.2.3 実装の評価

ここでは、バージョン管理に必要な情報とインターフェイスを、メタアーキテクチャを利用してインスタンスに付加する方法として、動的なメソッドの生成を利用する方法と、メタクラスのカスタマイズを利用する方法を示した。どちらの方法も、管理情報とメソッドを動的に変更することが可能であり、また既存のクラス定義に全く影響を及ぼさないという点で、管理機構の透過性も満たしていると言える。

これらの方法はバージョン管理機構だけでなく、インスタンスに動的に情報を付加する必要があるような管理機構の実装に広く利用することができる。どちらの方法を用いるかは、管理機構に応じて選択することが可能であり、必要に応じてカスタマイズすることも可能である。

6 おわりに

本論文では、メタアーキテクチャを持つオブジェクト指向言語を OMS の実装と記述言語に用いることを提案した。また、OMS の管理機構をアプリケーションレベルで記述する際に、管理機構の透過性を高めるためにメタアーキテクチャを利用することが有効であることをバージョン管理を例に確認した。

本論文ではメタアーキテクチャの利用法そのものについては深く触れなかったが、実際にメタアーキテクチャを利用するためには、メタアーキテクチャの構成すなわち言語の実装について深く理解する必要であり、利用するのが非常に難しい。

今後の課題として、メタアーキテクチャ容易に利用できるようにするために、いくつかのメタクラスの拡張例をライブラリとして提供すること、および拡張の間の整合性をどのように維持するための枠組について考えたい。

参考文献

[1] Apple Computer Inc.: *Dylan^{tm} Interim Reference Manual*, 1994.

[2] Björnerstedt, A. and Hultén, C.: Version Control in an Object-Oriented Architecture, *Object-Oriented Concepts, Databases, and Applications*(Kim, W. and Lochovsky, F. H.(eds.)), ACM Press, 1989, pp. 451–485.

[3] Brown, A. W.: *Object-Oriented Databases Applications in Software Engineering*, McGraw-Hill Book Company, 1991.

[4] Harrison, W., Ossher, H., and Kavianpour, M.: Integrating Goarse-Grained and Fine-Grained Tool Integration, *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering (CAES '92)*, 1992, pp. 23–25.

[5] Paepcke, A.: PCLOS: A Flexible Implementation of CLOS Persistence, *Proceedings of the European Conference on Object-Oriented Programming*, 1988, pp. 374–389.

[6] Paepcke, A.: *Object-Oriented Programming: The CLOS Perspective*, The MIT Press, 1993.

[7] Team, T. G.: The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes, *Proceedings of Asia-Pacific Software Engineering Conference*, 1994, pp. 410–419.

[8] 渡部卓雄: チュートリアル リフレクション, コンピュータソフトウェア, Vol. 11,No. 3(1994), pp. 5–14.

[9] 藤枝和宏, 落水浩一郎: メタレベル ・アーキテクチャを利用したオブジェクト管理システムの構成法について, 情報処理学会研究報告 *94-SE-101*, 1994, pp. 41–48.