ソフトウェア工学 129－5
（2000．11．16）

オブジェクトの動的バージョン管理におけるバージョンの整合性について

新山 英児、杉山 安洋
日本大学大学院工学研究科情報工学専攻

我々の研究室では、実行中のソフトウェアを停止することなく、機能の変更や拡張を行うためのメカニズムを研究している。我々は、オブジェクトに動的バージョン管理機構を導入することによって、ソフトウェアの機能を変更、追加可能とし、それを実現するためにいくつかのツールを開発してきた。現在の方式では、複数のオブジェクトに動的バージョン管理機構を導入した場合、オブジェクト間にバージョンの不整合が生じるという問題がある。本稿では、オブジェクトの動的バージョン管理の概要と、それを用いた際に生じるバージョンの整合性に関する問題、およびそれを解決するための今後の方針について述べる。

About Consistency among Versions in the Runtime Version Management for Objects.

Eiji Niiyama and Yasuhiro Sugiyama
Nihon University

We are studying a mechanism to modify or expand software systems while they are running. Our approach is based on the runtime version management of objects. We are developing several tools to implement the mechanism. However, this mechanism has some problems about consistency among versions when two or more objects have multiple versions simultaneously. This paper will describe an overview of our runtime version management mechanism for objects, the problems about the consistency among versions, and our future plan to resolve the problems.

1. はじめに

我々の研究室では、実行中のソフトウェアを停止させることなく機能を追加、変更する機構の研究[1][2][3]が行われている。本研究では、オンラインシステム等の特別なものだけでなく、一般の利用者が使用するようなPC用のアプリケーションソフト等もターゲットとしている。我々の方式は、オブジェクトの動的バージョン管理に基づくものであり、それを実現するためにいくつかのツールを開発している。これまでは、スタンドアローンで動作するソフトウェアの動的バージョン管理を研究してきたが、現在はJavaのRMIを用いた分散システムへの対応を検討している。

しかし、現在の動的バージョン管理の機構を各オブジェクトに導入する方式には、まだいくつかの問題が存在する。特に本稿のテーマでもあるオブジェクト間でのバージョンの整合性は、分散システムでは勿論だが、スタンドアローンで動作するソフトウェアにおいても考慮すべき、重大な問題である。

本稿では、オブジェクトの動的バージョン管理の概要と、それを用いた際に生じるオブジェクト間でのバージョンの整合性に関する問題について述べる。

2. オブジェクトの動的バージョン管理の概要

オブジェクトの動的バージョン管理では、オブジェクトが複数のバージョンを持つ事を許可し、それらを実行中に選択して使用することができる。

ソフトウェアの実行中に、新しいバージョンを作成し、使用させることも可能である。ただし、この動的バージョン管理は、デバッグを目的としたものではなく、実行中のソフトウェアをデバッグの完了した新バージョンと交換することで、ソフトウェアの機能を追加、変更するためのものである。ここでは、オブジェクトの動的バージョン管理の実現方式について説明する。

2. 1. 代理オブジェクト

オブジェクトの動的バージョン管理では、使用するオブジェクトのバージョンを切り替えるためのメカニズムとして、代理オブジェクトを用いた。図1に示したように、バージョン管理対象のオブジェクトとそのオブジェクトにアクセスするクライアントの間に代理オブジェクトが存在する。

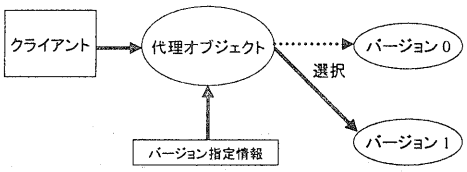

図1．代理オブジェクトの基本原理

クライアントはオブジェクトの特定のバージョンを直接参照するのではなく、代理オブジェクトを参照する。クライアントからメソッドが呼び出されると、代理オブジェクトは、どのバージョンを使用するべきかを示すバージョン指定情報を調べる。指定されたバージョンがまだ存在しない場合は、そのバージョンを代理オブジェクトが自動的に生成し、既存のバージョンの一つとして管理する。そして、複数のバージョンから指定されたバージョンのメソッドを呼び出し、そのバージョンからの戻り値をそのままクライアントに返す。代理オブジェクトから新バージョンのメソッドを呼び出す際は、初期バージョンに存在するメソッドを定義した代理インターフェースを利用している。新しいバージョンのメソッドのうち、初期バージョンにも存在するものは、代理インターフェースを用いて呼び出し、新たに追加されたメソッドは Java のリフレクションを用いてメソッドを取得し、呼び出しを行う。

我々は、代理オブジェクトのメカニズムを実現するためのツールとして **proc**(ProxyCompiler)を開発した。これは、代理オブジェクトの為のクラスである代理クラスを生成するためのツールで、バージョン管理の対象となるクラスの初期バージョンのソースコードを解析し、初期バージョンのクラス兼代理クラスのソースコード、代理インターフェースのソースコードを生成する。

2. 2. Derivation

使用するオブジェクトのバージョンを実行中に切り替える際に、既存バージョンの状態を保持するメカニズムとして Derivation を用いた。図2は Derivation の概念を示したものである。

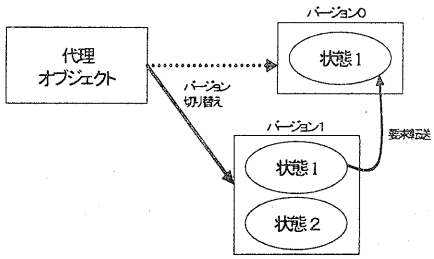

図2．Derivation の基本原理

Derivation では、オブジェクトの新しいバージョンに切り替える際、必ず旧バージョンを基に新しいバージョンを生成し、旧バージョンの状態を保持する。そして、新しいバージョンは旧バージョンの状態を共有し、新バージョンで定義されたものは直接処理し、旧バージョンで定義されたものには要求転送を行い処理する。これによって、旧バージョンの状態を保持したまま、新バージョンへの切り替えを行い、機能を変更させることが可能である。

この Derivation を用いて新しいバージョンのクラスを定義する場合、その構文は Java の言語仕様を次のように拡張して記述するものとする。

class 新バージョン **revises** 旧バージョン{……}

Java 言語の継承で用いる extends の代わりに revises を使用し、それ以外はサブクラスの記述法とほぼ同じである。但し、標準の Java コンパイ

ラはこのような記述を許可していないので、拡張した構文で記述された新バージョンのソースコードを通常の Java コンパイラによってコンパイルできるように変更するためのツールとして **derc**(Derivation Compiler)を開発した。

新バージョンのクラスを作成するには、まず上記の構文を用いてソースコードを記述し、作成したソースコードに **derc** を使用する。**derc** は Derivation を用いて記述された新バージョンのクラスのソースコードと revises で指定された旧バージョンのクラスのソースコードを解析し、Java の構文に沿った新バージョンのクラスのソースコードを生成する。このソースコードを通常の Java コンパイラでコンパイルし、対象クラスのバージョン指定情報を変更しておけば、代理オブジェクトの public メソッドが呼び出された際に新バージョンのオブジェクトが自動的に生成されバージョンの切り替えが行われる。

3. バージョンの整合性に関する問題

我々の代理オブジェクト、Derivation のメカニズムを用いたオブジェクトの動的バージョン管理では、各オブジェクトの代理オブジェクトが独立してそのオブジェクトのバージョンを管理している。したがって、複数のオブジェクトに動的バージョン管理を導入した場合などに、その複数オブジェクト間またはオブジェクト内におけるバージョンの整合性を保つことができずに、バージョンの不整合が発生する。その結果、ソフトウェアが停止することや、予想外の動作を起すことなどが考えられる。

我々の動的バージョン管理において発生するバージョンの不整合には、バージョン指定情報の間違いや、現在のバージョン指定方式では対応できないもの、ソフトウェアの開発時または変更時に組み込まれるもの、バージョン変更のタイミングによるもの等が挙げられる。

3. 1. バージョン指定情報の間違い

現在、我々の動的バージョン管理におけるバージョン指定の方法は、各クラスのバージョン指定情報をバージョン設定ファイル内に次のような構文で記述するものである。

Which.class=クラス名=バージョン:クラス名=バージョン...

"Which.class="に続きバージョンを指定したいオブジェクトのクラス名と、使用させたいオブジェクトのバージョンを示す文字列を"="で結んで記述し、複数クラスのオブジェクトのバージョンを指定したい場合は、":"で区切って指定する。したがって、ユーザが全てのクラスのバージョンを把握し、設定ファイルに直接記述する必要がある。

あるソフトウェアで使用する複数クラスに動的バージョン管理を導入した場合、そのソフトウェアにはバージョンの組み合わせに制限が生じる場合がある。例えば、図3のようにソフトウェア内でAクラスがBクラスを参照し、二つのクラス間に直接的な依存関係がある場合、二つのクラスにはバージョンの組み合わせに制限が生じることがある。

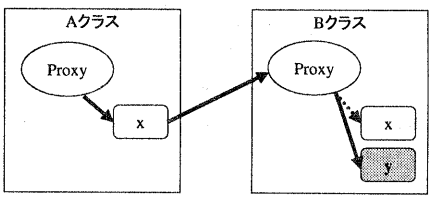

図3．直接的な依存関係

図4は、Aクラス、Bクラスのオブジェクトが Cクラスのオブジェクトを共有している例である。

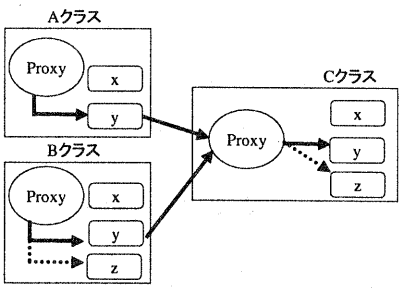

図4．オブジェクトの共有

このような構成のプログラムにおいて、Aクラスと Cクラス、Bクラスと Cクラスには参照関係

があるため、バージョンの組み合わせに制限が生じる。そして、AクラスとBクラスはCクラスのオブジェクトを共有しているため、A クラス、B クラス、C クラスのバージョンの組み合わせに制限が生じる。

図5は、BクラスとCクラスが同一のAクラスのインスタンス変数を持つ場合の例を示している。ここでは、Bクラス、Cクラスがそれぞれ A クラスを参照しているため、BクラスとAクラス、CクラスとAクラスの関係だけでなく、構造上で離れているBクラスとCクラスにも依存関係が生じる可能性がある。したがって、Aクラス、Bクラス、Cクラスのバージョンの組み合わせに制限が生じる。

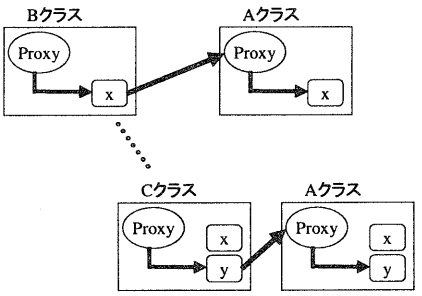

図5．クラスの共有

図6は、ソフトウェアの二つの部分に注目したクラス構成図である。

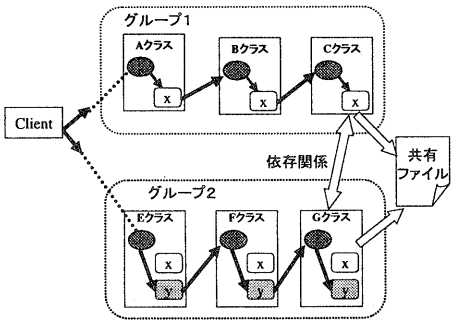

図6．間接的な依存関係

グループ1内のAクラス、Bクラス、Cクラス、グループ2内のEクラス、Fクラス、Gクラスにはそれぞれ依存関係がある。この例では、更にCクラスと G クラスがファイルを共有することによって、間接的な依存関係を持ち、グループ1、グループ2で使用している全クラスのバージョンの組み合わせに制限が生じる可能性がある。

このように、複数クラスのバージョンの組み合わせに制限が生じた場合、ユーザがこの制限に反するバージョンの組み合わせを指定することにより、バージョンの不整合が発生する。したがって、現在のバージョン指定方式を改善する必要がある。

3．2．現在のバージョン指定方式の限界

現在の動的バージョン管理におけるバージョン指定方式には、対応できない構造のプログラムが存在し、整合性を保ってソフトウェアを実行できない場合がある。

例えば、図7のように二つのパッケージ、PとQを使用したプログラムにおいて、共にAクラスが存在していたとする。ただし、パッケージPのAクラスとパッケージQのAクラスは別のものである。ここで、パッケージPのAクラスとBクラスはバージョンxで、パッケージQのCクラスとAクラスはバージョンyで動作しなければならないと仮定する。

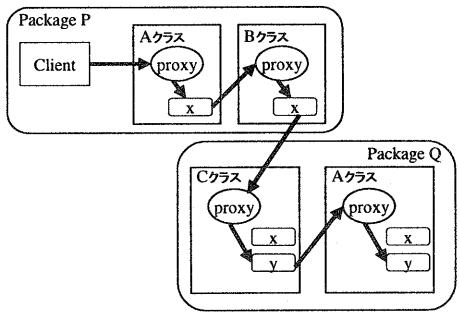

図7．複数パッケージを用いた場合

現在の動的バージョン管理は、クラス名とバージョン指定文字列を一対にしてバージョン指定するものであり、一つのファイル内にソフトウェアで使用する全てのクラスのバージョンを記述する為、パッケージPのAクラスはバージョンx、パッケージQのAクラスはバージョンyという指定はできない。パッケージPのAクラスのバージョンxを指定した場合には、パッケージQのクラス間でバージョンの不整合が生じる。

また、図5のようにソフトウェア内でAクラスを複数用いていた場合には同様の問題が生じる。ただし、ここでのAクラスは、図7の例と違って同じクラスの異なるインスタンスである。この例では、BクラスとCクラスは構造上離れているため、BクラスとCクラスに関係がまったく無い場合には、Bクラスが参照するAクラスのオブジェクトはバージョンx、Cクラスが参照するオブジェクトはバージョンyといった指定が可能かもしれないが、現在のバージョン指定方式では対応できない。

このようにクラス名が重複する構造、または、一つのクラスのインスタンスが複数存在する構造のソフトウェアでは、整合性を保持できるバージョンの組み合わせを指定する事さえできない。したがって、現在のバージョン指定方式を改善する必要がある。

3. 3. ソフトウェアの開発

オブジェクトの動的バージョン管理を導入したクラスを用いて、ソフトウェアの開発を行う場合には、当然ながらバージョンの整合性がとれるように開発する必要がある。これは、ソースコード単位の通常のバージョン管理でも同じ事である。クラス間のバージョンの整合性が崩れたソフトウェアを開発してしまえば、当然、実行時にバージョンの不整合が起こるだろう。

例えば、図4のようなクラス構成のプログラムを、既存のCクラスを用いて作成する際、Cクラスには既にバージョンx、バージョンyが存在していたとする。開発者はAクラスをCクラスのバージョンx、BクラスをCクラスのバージョンyを用いて開発した。ここでの問題は、Cクラスのオブジェクトのバージョンが同時に複数存在するような構造のプログラムを作成したことであり、このようなくバージョンの組み合わせを最初から作ってはならない。

また、複数のパッケージを用いてソフトウェアを開発する場合に、それぞれのパッケージは単体ではバージョンの整合性を保持し、問題なく動作していたとする。しかし、複数のパッケージを組み合わせてソフトウェアを開発することで、正常に動作できない可能性がある。例えば、パッケージ間で共通のオブジェクトを受け渡している場合や、図6のようにファイルを共有していた場合には、クラス関係からは判断できない間接的な依存関係があり、バージョンの不整合が発生してしまう可能性もある。

したがって、開発者は自身が開発するパッケージだけでなく、他のパッケージ内で使用している全クラスの依存関係も知る事が必要で、ソフトウェア内で使用する全てのクラスの依存関係、バージョンの組み合わせを考慮して、開発を行う必要がある。

3. 4. ソフトウェアの機能変更

動的バージョン管理では実行中のソフトウェアの機能を変更するため、最初はバージョンの整合性を保って動作していたソフトウェアであっても、実行中にバージョンの整合性が崩れてしまう可能性がある。

例えば、図3のようにソフトウェア内でAクラスがBクラスを参照し、二つのクラスはバージョンxを使用し、バージョンの整合性を保って動作していたとする。ここで、ソフトウェアの機能変更の為にBクラスのバージョンyを開発したとする。バージョンxからバージョンyへの変更がBクラス内だけのもの、例えばAクラスから受け取った値を処理する過程を変更し、返す値は同じであれば、問題はないかもしれない。しかし、Bクラスの変更が他のクラスとの依存関係を崩してしまう場合、例えばAクラスに返す値が変更された場合などは、Bクラスをバージョンyに変更することによって、バージョン整合性が崩れる可能性がある。したがって、他のクラスの依存関係を無視して、Bクラスの新バージョンを開発してはならない。Bクラスのバージョンyと整合性を保つことができるAクラスの新しいバージョンを開発しなければならない。

図4、図5の例では、Aクラス、Cクラス間の

依存関係だけを考慮して、Aクラス、Cクラスの新バージョンを開発することで、Bクラス、Cクラス間、または、Bクラス、Aクラス間の依存関係が崩れ、バージョンの不整合が起きる可能性がある。

図6の例では、最初、グループ1、グループ2内の各クラスのオブジェクトはバージョンxを使用して動作し、全クラスのバージョンの整合性は保持されていたとする。ソフトウェアの機能変更を行う際に、開発者はGクラスの変更が必要だと考え、クラス関係を考慮して、グループ2のEクラス、Fクラス、Gクラスのバージョンyを開発した。しかし、実際にはグループ1とグループ2のCクラス、Gクラスがファイルを共有しているため、間接的な依存関係がある。グループ1、グループ2の部分的な依存関係は崩すことなくバージョンの整合性が保持できているが、グループ2のバージョン変更に伴いCクラス、Gクラス間の依存関係が崩れる可能性がある。このように、ソフトウェア内での一部の依存関係のみを考慮して、クラスのバージョン変更を行ってしまうと、他の部分でバージョンの不整合を生じる可能性がある。

したがって、開発者はソフトウェアの機能を変更する際にも、二つのクラス間や部分的な依存関係だけでなく、ソフトウェアが使用する全てのクラスの依存関係を把握し、バージョンの整合性を保持したまま、各クラスの新バージョンを開発しなければならない。しかし、大規模なソフトウェアにおいて、そこまでの整合性を開発者が考慮することは、かなりの負担になるだろう。

3. 5. バージョン変更のタイミング

現在の動的バージョン管理において、各クラスの代理オブジェクトは、他のオブジェクトからのメソッド呼び出しをきっかけにして、バージョン指定情報を基に使用するオブジェクトのバージョンを変更する。したがって、バージョン変更が行われるタイミングによって、バージョンの不整合が生じる可能性がある。以下では、その基本的な例を挙げる。

3. 5. 1. 同一クラス内でのバージョン不整合

図8は、動的バージョン管理を導入した一つのクラス内でバージョンの整合性が崩れる例を示したシーケンス図である。これは、Aクラスでバージョン管理を行った場合に、Aクラスの代理オブジェクトのputメソッドが呼び出された際の処理であり、コールバック関数を用いた例を表したものである。

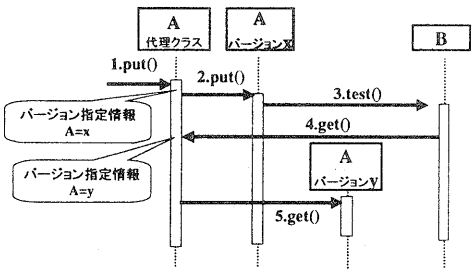

図8．同一オブジェト内の不整合

ここでの問題は、代理オブジェクトのputメソッドを呼び出した際のバージョンがxであったものが、putメソッド内での処理においてBクラスからgetメソッドを呼び出す際には、バージョンがyに変更され、Aクラスのバージョンyのgetメソッドを呼び出している事である。これによって、Aクラスのバージョンxのメソッド処理中にAクラスのバージョンyのメソッドで処理した内容をAクラスのバージョンxがBクラスからtestメソッドの戻り値として受け取ってしまう事も考えられる。このようにある処理の中で、一つのオブジェクトのバージョンが同時に複数存在することにより、バージョンの不整合が生じる。

図9も、一つのクラス内でバージョンの整合性が崩れる例を示したものであるが、LinkedList構造のプログラムのように、同じクラスのインスタンスが複数存在する場合のものである。

Aクラスの代理オブジェクトのputメソッドが呼び出された際のAクラスのバージョン指定がxであり、Aクラスのバージョンxのputメソッドを呼び出したとする。このputメソッド内で更にAクラスの代理オブジェクトのputメソッドを呼

び出したとする。ここでAクラスのバージョン指定がyに変更されていると、Aクラスのバージョンyのputメソッドを呼び出すことになり、Aクラスのバージョン不整合が生じる可能性がある。

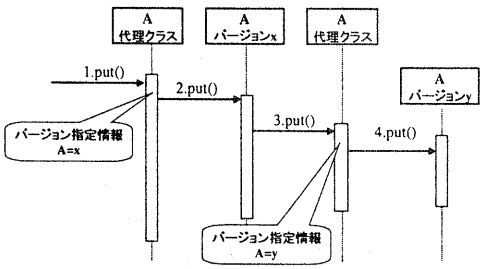

図9．同一クラス間の不整合

このように、一つのクラスに動的バージョン管理機構を導入するだけで、バージョン不整合が生じる。ここで示した例は基本的なものあり、もっと複雑なケースも考えられる。

3. 5. 2. 複数のクラス間でのバージョン不整合

ここでは、複数のクラスに動的バージョン管理を導入した場合に、クラス間のバージョンの整合性が崩れてしまう例を挙げる。

図10は、AクラスとBクラスに動的バージョン管理を導入し、Aクラスの代理オブジェクトのputメソッドを呼び出した際の処理である。ただし、前提条件としてAクラスのバージョンがxならばBクラスのバージョンもx、Aクラスのバージョンがyならば Bクラスのバージョンもyというバージョンの組み合わせに制限を持つ。

最初、Aクラス、Bクラスのバージョン指定が共にxでプログラムが動作している場合、Aクラスのputメソッドが呼び出されるとAクラスのバージョンxのputメソッドが呼び出される。さらにputメソッド内でBクラスのtestメソッドを呼び出せば、Bクラスのバージョンxのtestメソッドが呼び出される。しかし、Bクラスのtestメソッドを呼び出す際に、バージョン指定がyに変更されていた場合、Aクラスのバージョンxから Bクラスのバージョンyのtestメソッドを呼び出すことになり、前提条件であるバージョンの組み合わせに反することになる。ユーザは、前提条件に従いバージョン指定しているにも関わらず、バージョン指定情報に従って、代理オブジェクトがバージョン変更を行うタイミングにより、複数クラス間でのバージョンの不整合が起きる。

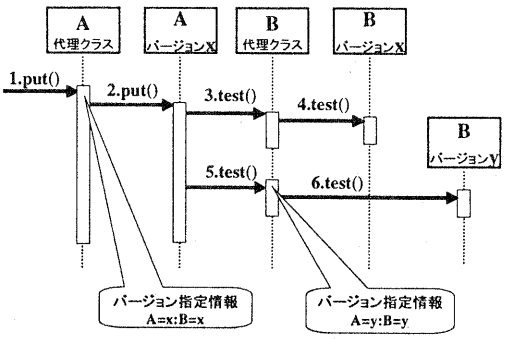

図10．複数クラス間のバージョン不整合

4. 実現方式の検討

ここでは、3章で述べた問題を解決するために必要な、現在検討中の実現方式を述べる。

4. 1. コンフィギュレーション

ユーザによるバージョン指定情報の間違いに関しては、ユーザが全てのクラスのバージョンを把握し、バージョンを指定する方式に問題がある。

そこで、Javaのパッケージのように、機能が関連するクラスやバージョンの組み合わせに制限を持つクラスだけを集め、そのバージョンを指定したものをコンフィギュレーションと呼ぶ。ソフトウェアはこのコンフィギュレーションの集合である。ユーザはバージョン設定ファイルに、自分の使用するコンフィギュレーションを記述する。コンフィギュレーションは開発者が用意し、クラスファイルと共に配布する。これで、ユーザはクラスのバージョンの組み合わせを考慮せずにバージョン変更が可能となる。

しかし、複数のコンフィギュレーションを組み合わせることによって、新たにコンフィギュレーション間のクラス関係にバージョンの不整合が生じる可能性がある。したがって、コンフィギュレーション間のクラスのバージョン整合性が保持されているかを確認するツールが必要となる。

また、システム全体で単一のバージョン設定ファイルを持つだけではなく、パッケージ毎やクラス毎、オブジェクト毎にバージョン設定ファイルが指定できるような機能も必要であると考えている。

4. 2. 開発支援ツール

3章で述べたように、開発者はソフトウェアで使用する全クラスのバージョンの整合性を考慮しなければならない。さらに、新バージョンの開発では、実行中のソフトウェアのバージョンなども考慮する必要がある。したがって、ソフトウェアの開発段階において、クラス間の依存関係を解析し、構造上のバージョン整合性をチェックするものや、実行中にあるクラスのバージョン変更を行う場合に、他クラスとのバージョン不整合が発生する可能性を開発段階で知るための開発支援ツールが必要である。

4. 3. バージョン変更のタイミング制御

3章で述べたバージョン変更のタイミングの問題を回避するためには、バージョン変更のタイミング等を制御するための情報を用意し、この制御情報を基にバージョンの制御を行う機構を動的バージョン管理に導入する必要がある。これを、バージョン管理サーバを導入することによって解決しようと考えている。前節で述べた開発支援ツールは、開発時にクラス間の依存関係情報を提供するだけでなく、バージョン管理サーバに制御情報の提供も行う。

バージョン指定情報をバージョン管理サーバに問い合わせる方式にすることで、全クラスのバージョンを一括管理し、バージョン変更のタイミング制御を行うなどの柔軟性を得る事が可能である。バージョン管理サーバは、代理オブジェクトからの問い合わせに対して、バージョン指定情報と制御情報を基に使用するオブジェクトのバージョン指定文字列を返信する。

バージョン管理サーバの機能としては、図8の例で示した同一クラス内でのバージョン不整合を回避するためのバージョンのロック機能が必要である。Aクラスのputメソッド処理中に、他のオブジェクトからAクラスのgetメソッドが呼び出された際には、同一バージョンのgetメソッドを呼び出す。また、図9の例では、Aクラスのバージョンxから、さらにAクラスのputメソッドを呼び出した場合は、同一バージョンのputメソッドを呼び出すようにしたい。

図10の例で示した複数クラス間でのバージョン不整合を回避するためには、複数の代理オブジェクトの状況を把握し、制御情報を基にオブジェクト間のバージョンの整合性を保持する機能が必要である。図10の例で、Aクラスのバージョンxからクラスのメソッドを呼び出した際には、Bクラスのバージョンyのメソッドではなくクラスのバージョンxのメソッドを呼び出せるように制御しようと考えている。

バージョン管理サーバは、トランザクションを管理し、バージョンの不整合が発生しないタイミングで、安全にバージョン変更を行う必要がある。

5. まとめ

今後はバージョン管理サーバの開発とコンフィギュレーションについて研究を行う予定である。

参考文献

[1]杉山安洋、Java クラスの動的バージョン管理の構想、情報処理学会研究報告 97-SE-115、pp.49-56,1997

[2]杉山安洋、バージョン管理に基づくオブジェクトの動的発展機構、日本ソフトウェア科学会第15回大会論文集、pp.205-208,1998

[3]Y.Sugiyama, A Mechanism for Runtime Evolution of Objects, IWPSE99, pp.27-31, 1999