平成23年度 修士論文

Reserved member 方式による メンバーの実行時追加機構の 提案

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 10M37172

早船 総一郎

指導教員

千葉 滋 教授

平成24年1月27日

概要

本論文では reserved member 方式による Java 言語のためのメンバーの実行時追加機構を提案する．ここで，メンバーとはメソッド，フィールド，コンストラクタをさす．プログラムの開発では修正を行うたびに再起動を行うことが多いが，非常に大きなコストがかかってしまう．そこで，動的にプログラムの振る舞いを変更するという機能は必要性が高いとされており，標準の Java 仮想機械でも一部の機能は既にサポートされている．この機能は HotSwap と呼ばれ，実行中の Java プログラムを監視して，バイトコードレベルでクラスを定義するクラスファイルを新しいものに差し替える機能である．しかし，標準の Java 仮想機械は実行時のメンバーの追加に対応しておらず，HotSwap 機能を用いても既存メソッドの本体の書き換えしかできない．メンバーの型の変更等はできない．提案手法では，準備として，ロード時やコンパイル時など実行前に予備のメンバーをクラスにあらかじめ追加しておく．これを reserved member と呼ぶ．実行開始後にユーザーがメンバーの追加操作を行ったときには，その reserved member を変更して，追加されたメンバーの機能を実現する．追加されたメンバーを参照する式は，その reserved member を参照するように修正される．また追加されるメンバーがメソッドであるときは，その本体が reserved member にコピーされるが，このままでは型の整合性が取れず，追加されるメンバーの振る舞いを実現できない．そのため，型の整合性をとるために，実行時に型変換を行うコードが適宜挿入される．提案手法は複数のメンバーの追加を許しており，reserved member 内で実現されたメンバーの切り替えを行うことで，複数のメンバーの追加を実現している．さらに追加されたメンバーをより正確に実現するためにオーバーライドや修飾子を考慮して実装が行われている．本論文ではさらに提案方式のオーバーヘッドを計測するための実験について述べる．Reserved member の準備にかかるオーバーヘッドや型変換を必要とするメソッド呼び出しのオーバーヘッドを計測した結果によると，実行前の準備には時間がかかるが，実行中のオーバーヘッドは無視できるほど小さい．

In this paper, we present a system using a reserved member to add a member at runtime in Java. The member is method, field or constructor. Programmers often reboot programs of development after each modification, It takes a very large cost. Functions that dynamically change the behavior of programs are very required. The standard Java virtual machine supports some features. This feature is called HotSwap, to monitor running the Java programs, replaces the programs with new ones, that are defined in the class file in the bytecode level. However, the standard Java virtual machine does not allow adding members at runtime. It can only provide HotSwap for allowing the redefinition of the body of an existing method but the types of members cannot be changed. The proposed system adds a member, called a reserved member, to classes before execution, for example, at load time or compile time. When the user changes a program to add a new member to a class, All the expressions referring to the added member are also modified to refer that reserved member. If the added member is a method, the body of the added member is copied into the reserved member. For preserving the type consistency, the system also inserts type conversion code at appropriate places. The system allows adding any number of members by using a limited number of reserved members and a switch mechanism in a reserved member. The system supports method overriding and modefiers to achieve a more accurate that additional members. The performance overheads are also presented in the paper. Accoding to the results of measuring the overhead of method calls that require the typecast and preparing the reserved members, the preparation takes time before the execution, the overhead of runtime is very small.

謝辞

本研究は以下の方々なくして，存在しえなかったでしょう．研究の提案や方針などになにかと心を砕いていただき，多くの指導をしていただいた指導教員の千葉滋教授に感謝致します．

堀江倫大氏には，論文の書き方など親身になって指導をいただきました．森田悟史氏には，研究についての多大な知識を与えていただき，さらに本研究の基盤となるソフトウェアの構築に関してや論文の書き方なども指導していただきました．それ以外にも様々な意見と助言をいただきました．伊尾木将之氏，赤井駿平氏，武山文信氏には多くの意見をいただき，おかげさまで研究の質を向上させることができました．心より感謝しています．

最後に，本研究を行ううえで励ましていただいた同研究室の皆様に心から感謝いたします．

目 次

図 目 次

表 目 次

第1章　はじめに

システムを開発するにあたりプログラムを一度完成させても，まだ開発は完全に終了したわけではない．プログラムが実行中でサービスを提供している状態になっていてもメンテナンスを行う．通常プログラムの更新は実行しているプログラムを一旦停止させ，パッチを当て，再度起動する．しかし，このように停止している間はサービスを提供できないため，大きな損害となる．

そこで，実行しているプログラムを停止させることなくコードを更新する技術が必要である．このような技術は昔から研究が行われ，Smalltalk, Python, Ruby のような動的な言語では実行しているプログラムに動的な変更を行える機構を備えていることが多い．動的言語に備わっている機構ではクラス定義を変更できるだけでなく，より多くの変更を行える [3, 4, 13]．本システムで行う動的にメンバーを追加するという機能はクラス定義を変更することである．一方，Java では静的な型付け言語であるため動的な変更は難しく，実行しているプログラムを変更する機能は標準 Java 仮想機械では部分的にしか提供していない．特に新しいメンバーをクラス定義に追加することができない．

1.1　Reserved member 方式

我々は reserved member 方式によるメンバーの実行時追加機構を提案する．本システムは Java 仮想機械に手を加えていないため，既存のシステムにも導入しやすい．特に標準の HotSwap を利用している既存のシステムの実装に本システムを利用することで，新たな機能拡張を行うことができるようになる．例えば，標準の HotSwap を利用している Dynamic Aspect-Oriented Programming（DAOP）の実装に本システムを利用すれば，AspectJ [26, 16] で提案されているインタータイプ宣言を動的に実行できるなどの応用が考えられる．

Reserved member 方式では以下の 2 段階の処理により動的なメンバーの追加を実現した．一段目は，実行前に各クラスへ事前に reserved member を追加しておくことで，実際には動的な追加を行わないように準備することである．二段目は，ユーザーがメンバーの追加の操作を行ったときに，

追加したいメンバーの処理を reserved member で実現し，メンバーの呼び出しを書き換えることである．

1.2　本稿の構成

次章から，本研究の背景と提案手法について述べる．本稿の構成は以下の通りである．

2 章：動的なクラス定義の変更と関連研究

Java における既存の動的なクラス定義の差し替えについて詳細に述べ，既存の手法での問題点について述べる．動的な振る舞いの変更が可能である動的言語とその機構に関する関連研究について述べる．

3 章：Reserved member 方式

我々が提案する reserved member 方式について述べる．

4 章：バイトコード変換

Reserved member 方式を実現するにあたり， Javassist[11] を用いた必要なバイトコード変換について述べる．

5 章：実験

Reserved member 方式のオーバーヘッドを自作したマイクロベンチマークと SPECjvm2008[25] を用い，計測を行う．

6 章：まとめと今後の課題

本研究をまとめて，今後の課題を述べる．

第2章　動的なクラス定義の変更と関連研究

本章では動的なクラス定義の変更とそれに関連する研究について述べる．まずは動的なクラス定義の変更を行うための一般的なシステム構成やユーザの操作について述べ，既存の手法での問題点について述べる．

本研究で行う動的なプログラムの差し替えは結果として動的な振る舞いの変更が行われるため，関連研究として含める．この動的な振る舞いの変更を得るために，動的なプログラムの差し替え以外にも様々な方法で行われてきた．他の言語の利用している振る舞いを変更や他のモジュールや機構を利用している振る舞いを変更について述べる．また，本システムの実装に類似した実装を用いている研究も取り上げる．

2.1　動的なクラス定義の変更

Java プログラムのクラス定義を動的に変更する機能は様々な応用が考えられ，必要性が高いとされている．例えば，デバッグモードで実行中のプログラムを一時停止させ，クラス定義を更新して，プログラムを再開することができるようになる．このような変更を行うことで効率よくプログラムの開発をすることができる．また，動いている Web アプリケーションを止めずに，セキュリティパッチを当てることができる．新しいクラス定義をセキュリティパッチとして用意し，既存のクラスの定義をこの新しいクラス定義に更新すればよい．さらに，動いている Web アプリケーションを止めずに，オンサイトで性能ボトルネックを調査することにも使える．プロファイルデータを収集するコードを組み込んだクラス定義に実行中に更新することでそのような調査が可能である．

2.1.1　典型的なユーザの操作

このように動的にクラス定義を変更し実行する場合，一般的に次のような手順がとられる．まず，実行中の Java 仮想機械に対し，プロセス間通信でコネクションを張り，クラス定義を変更したいときにユーザが対話的

に新しいクラス定義のプログラムを送り込む．Java 仮想機械は送り込まれたクラス定義を利用し，その変更を既存のクラス定義に反映する．送り込むクラス定義はバイトコードにコンパイル済みで，型検査等はオフラインで実行するものとする．オフラインの準備では変更されないクラスも用意して，それら全体を合わせた上で整合性が取れているか検査を実行する．

2.1.2　既存の手法

このような動的なクラス定義の変更の必要性はよく知られているため，標準 Java 仮想機械でも一部の機能は既にサポートされている．これを用いれば Java プログラムの実行状態を監視し，クラスを定義するクラスファイルをバイトコードレベルで新しいものに動的に交換する Java プログラムを書くことができる．この機能は java.lang.instrument パッケージ [27] で提供されており，HotSwap と呼ばれる．このとき，新しいクラス定義のメソッドボディの内容を異なるものに変更することができる．しかし，変更後のクラスは型に関して既存の他のクラスと整合性がとれていなければならず，新しいメンバーを追加することはできない．これは仮想関数テーブル変更できないためであり，それに基づいた機能もあるためと思われる．

HotSwap 機能は非常に制限が大きいので，Java 仮想機械を改造し機能を拡張する研究が行われてきた．さらに Java 仮想機械を改造することでメンバーの追加や削除だけでなく，クラスの継承関係の変更も行えるシステムも存在している [4, 29] ．このように Java 仮想機械を改造することで動的にクラス定義を置換可能にする方法はよく知られている．だが，実用上は一般的に利用されている標準 Java 仮想機械を用いた上でクラス定義を動的に変更する方法が望ましい．

2.1.3　HotSwap

HotSwap は java.lang.instrument パッケージ [27] で提供されており，Java プログラムの実行状態を監視し，クラスを定義するクラスファイルをバイトコードレベルで新しいものに動的に交換する Java プログラムを書くことができる．これは本システムの実装でも利用されており，他のシステムでも広く使われている．java.lang.instrument パッケージでは，Java エージェントと呼ばれるプログラムの計測機構が提供されており，計測対象のアプリケーションと同じ Java 仮想機械上で動作するエージェントを実現する仕組みである．計測対象のアプリケーションのロード時・実行時

に，計測対象のアプリケーションのバイトコードを他のバイトコードに置き換える変換することができる．HotSwap は二つの変換のうちの実行時の変換を示している．ロード時の変換は動的なバイトコード変換を行うことができないが，クラス定義の変更に関して制限を受けない．

Instrumentation インターフェース

このクラスは，Java プログラミング言語のコードを計測するためのサービスを提供する．Instrumentation とは，ツールで使用するデータを収集することを目的としてメソッドにバイトコードを追加することである．変更は単に追加されるため，これらのツールはアプリケーションの状態や動作を変更しない．状態や動作に影響を及ぼさないこの種のツールには，監視エージェント，プロファイラ，カバレージアナライザ，およびイベントロガーなどがある．Instrumentation インタフェースのインスタンスを取得する方法は 2 つある．

1. エージェントクラスを指定する方法で Java 仮想機械を起動した場合．この場合，Instrumentation インスタンスは，そのエージェントクラスの premain メソッドに渡される．

2. Java 仮想機械の開始後にエージェントを開始する機構が Java 仮想機械に用意されている場合．この場合，Instrumentation インスタンスは，そのエージェントコードの agentmain メソッドに渡される．

エージェントが Instrumentation インスタンスを取得すると，インスタンス上のメソッドをいつでも呼び出すことができる．

ロード時の変換に用いられているメソッドとしては，addTransformer, removeTransformer などがあり，ClassFileTransformer オブジェクトを登録したり，削除を行うことができる．登録された ClassFileTransformer がクラスロードの過程で割り込み，バイトコード変換を行う．再定義による変換に用いられるメソッドとしては，redefineClasses と isRedefineClassSupported などがある．

- addTransformer メソッド

  提供されたトランスフォーマを登録する．

- redefineClasses メソッド

  引数に渡された ClassDefinition オブジェクトでクラスの定義を置換する．ClassDefinition オブジェクトとは java.lang.Class オブジェクトと新しいクラスファイルバイトをバインディングするオブジェク

トである．なお，このメソッドを使った再定義では，メソッドの本体，定数プール，属性が変更が可能である．ただし，再定義では，フィールドまたはメソッドの追加，削除，あるいは名前の変更，メソッドのシグネチャーの変更，あるいは継承の変更はできない．

- isRedefineClassesSupported メソッド

  現在の Java 仮想機械の構成がクラスの再定義をサポートしているかどうかを返す．すでにロードされているクラスを再定義する機能は，Java 仮想機械のオプションの機能である．再定義がサポートされるのは，エージェント JAR ファイルで Can-Redefine-Classes マニフェスト属性が true に設定されていてかつ Java 仮想機械がこの機能をサポートしている場合に限られる．単一の Java 仮想機械の 1 つのインスタンス生成の間に，このメソッドに複数の呼び出しを行うと，常に同じ答えが返される．

premain メソッド

Instrumentation インタフェースのインスタンスはエージェントクラスの premain メソッドの引数として取得することができる．エージェントクラスには，このメソッドを実装しておかなければならない．javaagent オプションを指定してアプリケーションを実行することで，アプリケーションの main メソッドが呼ばれる前にエージェントクラスの premain メソッドが呼ばれ，計測をすることができる．このメソッドは，以下のようなシグネチャを持つ．

```
public static void premain (String agentArgs, Instrumentation inst)
```

-javaagent オプション

java.lang.instrument パッケージではコマンドラインインターフェースで-javaagent オプションを利用する方法でエージェントを開始する．

```
-javaagent:jarpath[=options]
```

jarpath はエージェント JAR ファイルのパスである．options はエージェントのオプションである．これは一つのコマンドラインで複数回使用できるため，複数のエージェントを作成できる．複数のエージェントで同じ jarpath を使用できる．エージェント JAR ファイルは JAR ファイル使用に従う必要があり，マニフェストの記述も必要となる．

マニフェスト

エージェント JAR ファイルには次の主なマニフェスト属性が定義されています.

- Premain-Class

  Java 仮想機械の起動時にエージェントが指定される場合は，この属性でエージェントクラスを指定します．つまり，premain メソッドが含まれるクラスである．Java 仮想機械の起動時にエージェントが指定される場合は，この属性が必須である．この属性が存在しない場合，Java 仮想機械は異常終了する．これはクラス名であり，ファイル名やパスではない.

- Can-Redefine-Class

  ブール値（true または false，大文字小文字は区別しない）．クラスを再定義する機能がこのエージェントに必要かを表す．true 以外の値は false であるとみなされる．この属性はオプションで，デフォルトは false である.

transform メソッド

このメソッドの実装は，提供されたクラスファイルを変換して，新しい置換クラスファイルを返すことができる．これを用いてロード時の変換を行う．実装しているメソッドが変換不要と判定すると，メソッドは　null を返す．変換が必要と判定すると，メソッドは新しい byte[] 配列を作成し，すべての必要な変換とともに classfileBuffer 入力をその配列の中にコピーし，新しい配列を返します．classfileBuffer 入力は変更されない.

サンプルプログラム

java.lang.instrument パッケージをより詳細に解説するために図 2.1 のサンプルコードを元に解説する．まず，アプリケーションの main メソッドが呼ばれる前に 6 行目の premain メソッドが呼ばれる．8 行目のように引数の instrumentation オブジェクトをエージェントクラスに渡すことで，後のクラス定義の変更を可能にする．9 行目のように instrumentation オブジェクトに対して addTransformer メソッドを呼び出すことで，引数の agent オブジェクトの transform メソッドが各クラスのロード時に呼び出されるようになる．その後，システムが GUI（Graphical User Interface）などを作成し，そのオブジェクトにエージェントを渡すなどをすることで

```
1  public class Agent implements ClassFileTransformer {
2   private static Instrumentation inst;
3   private Agent(Instrumentation inst, String agentArgs) {
4     Agent.inst = inst;
5   }
6   public static void premain(String agentArgs, Instrumentation
7       inst) {
8     Agent agent = new Agent(inst, agentArgs);
9     inst.addTransformer(agent);
10    /* システムの準備 */
11  }
12  public byte[] transform(ClassLoader loader, String className,
13      Class<?> classBeingRedefined, ProtectionDomain
14      protectionDomain, byte[] classfileBuffer)
15      throws IllegalClassFormatException {
16    /* ロード時変換 */
17    return /* ロード時変換後のバイトコード */;
18  }
19  public static void redefineClass(String className, byte[]
20      bytecode) throws ClassNotFoundException,
21      UnmodifiableClassException, IOException,
22      CannotCompileException, NotFoundException {
23    ClassDefinition definition = new ClassDefinition(
24        Class.forName(className), bytecode);
25    ClassDefinition[] definitions = new ClassDefinition[] {
26        definition };
27    inst.redefineClasses(definitions);
28  }
29 }
```

図 2.1: java.lang.instrument パッケージのためのサンプルコード

再定義を行えるように準備する．12 行目の transform メソッドではロード時に変換を行え，その結果のバイトコードを返す．このとき主に第二引数のクラス名と第五引数のそのクラスのバイトコードを用いることで，バイトコード変換を行う．19 行目の redefineClass メソッドではクラス名と新たなバイトコードを引数として呼び出されることで実行時にクラスを再定義することができる．23 行目では提供されたクラスとクラスファイルバイトを使って，新しい ClassDefinition バインディングを作成する．それを用いて 25 行目でクラス定義を変更する．

エージェント JAR ファイルに関しても JAR ファイルを作成するための図 2.2 のサンプル XML を紹介する．ant [2] を用いることで容易に作成することができる．全体としては一つの JAR ファイルを作成するようになっており，agent.jar というファイルが作られる．そのとき，ファイルとして含むのは上のサンプルコードのクラスファイルである．また，マニフェストを以下のように記述することで，マニフェストファイルを作る

```
1  <project name="simple.jar" default="simple.jar" basedir=".">
2    <target name="simple.jar" />
3      <jar destfile="./agent.jar">
4        <fileset dir="./bin">
5          <include name="Agent.class" />
6        </fileset>
7        <manifest>
8          <attribute name="Premain-Class" value="Agent" />
9          <attribute name="Can-Redefine-Classes" value="true" />
10       </manifest>
11     </jar>
12   </target>
13 </project>
```

図 2.2: サンプル XML

ことなく，JAR ファイルを作ることができる．

2.1.4　Remote Method Invocation（RMI）

このようなシステムを構築するためには，アプリケーションを実行している仮想機械に対して，外部からクラス定義が変更されたことに関する情報を与えなくてはならない．デバッグのみであれば，Java Platform Debugger Architecture（JPDA） を元にして，システムを構築することを考えたが，分散環境にあるプログラムを更新することも考えられている．

RMI の概要は以下の通りである．Java リモートメソッド呼び出し（Java RMI）を使用すれば，Java による分散アプリケーションを作成することができ，そのアプリケーションではリモート Java オブジェクトのメソッドを他の Java 仮想機械や異なるホストから呼び出すことができる．RMI はパラメータの整列化および非整列化にオブジェクトの直列化を使用し，型を切り捨てることなく，オブジェクト指向のポリモルフィズムをサポートしている．

2.2　関連研究

2.2.1　Open Class

Open Class [13, 18] ではクラス拡張の定義をできるモジュールを提供していて，それを用いることで既存のクラスに対して，自由にメソッドを新たに定義したり上書きすることができる．Open Class があると，自由にメソッドを追加できるが，同じ名前のメソッドを追加すると，正しく動かなくなるなどの問題も存在する．

2.2.2 Classbox

そのため，Classbox [3, 4, 5, 6] という機能が提案されている．Classbox はパッケージのようなもので，Classbox を import して，その中のクラスを影響範囲を限定して拡張することができる．これにより Open Class に存在する問題を回避している．

2.2.3 Aspect-Oriented Programming （AOP）

DAOP について議論を行う前に，AOP についての知識を整理する．AOP とは Object-Oriented Programming（OOP）では解決できないモジュール間にまたがる横断的関心事（crosscutting concern）をアスペクトと呼ばれる独立したモジュールに分離するプログラミング技法である．

- アスペクト

  プログラムの横断的関心事を記述するためのモジュール．アスペクトはポイントカットとアドバイスの組で横断的関心事をまとめる．

- ジョインポイント

  プログラムのある実行点を表す．例えば，メソッドの呼び出し・実行，コンストラクタの呼び出し・実行，フィールドの参照・代入，オブジェクトの初期化，例外ハンドラの実行などがある．

- ポイントカット

  ジョインポイントの集合から目的のジョインポイントの集合を指定する．条件を指定することで，ジョインポイントの集合が作られ，アドバイスと結びつける．

- アドバイス

  ポイントカットされたプログラムの実行点で行う処理.

- 織り込み（weave）

  アスペクトで記述されたモジュールを統合する処理．ポイントカットで指定されたジョインポイントの集合でアドバイスが実行されるように結び付ける．AOP の実装によって異なるが，アドバイス本体をジョインポイントに埋め込む，アドバイス本体を呼び出すコード（フック）を埋め込む方法などがある．

AspectJ

AspectJ [16, 26] はアスペクト指向言語として最も有名であり，広く使われている言語である．AspectJ は Java 言語の拡張を行い，アスペクトの機能を利用できるようにした言語である．AspectJ で用いることができるポイントカットとアドバイスについて代表的なものを以下に示す.

- call ポイントカット

  メソッド呼び出し時，コンストラクタ呼び出し時.

- execution ポイントカット

  メソッド実行時，コンストラクタ実行時.

- before advice

  ジョインポイントが実行される直前に実行されるアドバイス.

- after advice

  ジョインポイントが実行される直後に実行されるアドバイス.

- around advice

  ジョインポイントの期間全体の処理の代わりに実行されるアドバイス.

```
1  public class Sample{
2    public static void main(String[] args){
3      Sample s = new Sample();
4      s.foo();
5      s.hoge();
6    }
7    public void foo(){
8      System.out.println(“foo”);
9    }
10   public void hoge(){
11     System.out.println(“hoge”);
12   }
13 }
```

図 2.3: Sample クラス

ここで，AspectJ を用いた簡単なプログラムを説明する．AspectJ を用いると図 2.3 のようなプログラムに対して，図 2.4 のようなアスペクトを実装することができる．元のプログラムでは foo() と hoge() を用いて，

```
1 public aspect HelloAspect{
2   pointcut helloMethod():
3     execution(void Sample.hoge());
4   before(): helloMethod(){
5     System.out.println(“Hello”)
6   }
7 }
```

図 2.4: HelloAspect アスペクト

```
foo
hoge
```

と出力するプログラムである．これに対してアスペクトでは　execution ポイントカットにより，hoge() メソッドの実行が選ばれている．そのポイントカットを用いて before advice が利用されている．このアスペクトが織り込まれた Sample クラスを実行すると，

```
foo
Hello
hoge
```

と出力されるように振る舞いが変わる．hoge() メソッドが実行される直前に HelloAspect のアドバイスが実行されるのでこのような出力になる．

アスペクトの静的な織り込み技法には主に二通りある．

- Compile-Time Weaving

  コンパイル時にアスペクトを織り込む技法であり，AspectJ で用いられている．AspectJ コンパイラは，アドバイスをメソッドに変換し，ジョインポイントの前後，またはその場所にメソッド呼び出しを挿入することで実現する．AspectJ コンパイラで作成されたクラスファイルは標準の Java 仮想機械で実行することができる．

- Load-Time Weaving

  クラスのロード時にアスペクトを織り込む技法であり，GluonJ [10] で用いられている．GluonJ では java.lang.Instrument パッケージを用いることで Java 仮想機械上で実行されているプログラムを監視することができ，クラスのロードをとらえることができる．そこでロード前のバイトコードに対してアスペクトの内容を反映する．

2.2.4　Dynamic Aspect-Oriented Programming（DAOP）

DAOP では後から横断的関心事を織り込むことで，動的に振る舞いを変更することができる．実行時に機能を追加することが出来る点は，本研究と似ている．

DAOP ではアスペクトを織り込む技法として Dynamic Weaving が可能となる．Dynamic Weaving ではプログラムの実行中にアスペクトを織り込むことができる．そのため，アプリケーションを再起動することなく，プログラムの振る舞いを変更することができる．また，一度織り込まれたアスペクトを unweave することができる機構が多い．unweave はアスペクトで記述されたモジュールを取り除く処理であり，ジョインポイントに結びついているアドバイスを取り除く．

HotSwap を用いて織り込み

この技法では，Java 仮想機械上にロードされたバイトコードを動的に新しいバイトコードに変換することで，織り込みを実現している．java.lang.Instrument API を用いることで，プログラムを監視し，変更を加えることでアスペクトの織り込みを行っている．ほとんどの DAOP システムは実装に HotSwap を用いているが，HotSwap ではメソッドボディの変更しかできないので，その範囲で実現可能な DAOP が提供されている [19, 24, 28, 15].

フックを用いた織り込み

この技法では，ジョインポイントに対してあらかじめフックを挿入しておくという準備を行う．このフックを用いてアスペクトの織り込みを行い，DAOP を実現している [21, 22]．このフックにより，各ジョインポイントがポイントカットとして選択されているかを判断し，アドバイスを実行するかを決めることができる．ロード時やコンパイル時にフックを挿入するが，すべてのジョインポイントにフックを織り込むとオーバーヘッドが大きくなり，一部のジョインポイントのみにフックを織り込むように制限すると想定していないジョインポイントを選択するアスペクトを織り込むことができない．

イベント通知を用いた織り込み

この技法では，元のコードを編集することなく，ジョインポイントで発生するようにしていたイベント通知を利用する．このイベント通知を用いてアスペクトの織り込みを行い，DAOP を実現している [24, 23]．デバッ

ガを用いてイベント通知するようにしているため，プログラムを停止させてから復帰するまでの時間的コストが大きい．

仮想機械レベルでの織り込み

この技法では，仮想機械をアスペクト指向用に拡張し，DAOP を実現している [7, 9, 1]．仮想機械を改良しているため，様々な機能に影響があり，オーバーヘッドが存在する．

2.2.5　Dynamic Virtual Machine（DVM）

Dynamic Virtual Machine（DVM）[17] では dynamic classes というモジュールを提案し，特別なクラスローダを用いて，動的な振る舞いの変更を可能にしている．これにより，クラスの再ロードとインスタンスの置き換えができる．しかし，元となる JVM のバージョンが JDK 1.2 であり，これを用いる価値を非常に下げている．

2.2.6　Dynamic Code Evolution

さらに Java 仮想機械を改造することでメンバーの追加や削除だけでなく，クラスの継承関係の変更も行えるシステムも存在している [29] ．このように Java 仮想機械を改造することで動的にクラス定義を置換可能にする方法はよく知られている．だが，実用上は一般的に利用されている標準 Java 仮想機械を用いた上でクラス定義を動的に変更する方法が望ましい．

2.2.7　Envelope-Based weaving

事前にメンバーを用意するという我々の提案手法の考えは，Envelope-Based weaving [8, 7] に似ている．Envelope-Based weaving ではコンパイル時またはロード時に各クラスのメンバーをラップするメンバーを追加しておく．この技術によりアスペクトを高速に織り込むことができる．アスペクトを織り込む時にこの追加されたメンバーを利用することで，Steamloom [9] による織り込みで必要だったオフセットの計算を行う必要がなくなり，バイトコードの操作が容易になっている．動的な変更にかかる計算のオーバーヘッドとコンパイル時またはロード時にかかる準備のオーバーヘッドとの比較で，事前に準備しておく方が優れていると主張している．Steamloom は DAOP システムの一つであり，Jikes Research Virtual Machine (RVM) という Java 仮想機械を改造して実現している．

しかし，Envelope-Based weaving の方式をそのまま用いては新規メンバーの追加が行えない．

2.2.8　Polymorphic Inline Cashes

本システムはメソッド内で複数の機能を実現しているため，メソッド内で切り替えを行っている．その実装は Polymorphic Inline Cashes [14] に似ている．この研究では処理の重い polymorphic message を行わずに，インライン展開されたキャッシュとして一つのメソッドに保持しておくことで，高速化を行っている．この一つのメソッドに保持しておく方法が類似している．しかし，キャッシュとして保持するのではなく，追加されたメンバーが実現されているため，実装としては異なる点がある．

2.2.9　SPECjvm2008

SPECjvm2008 は，Java のコア機能に焦点を当てたいくつかの現実のアプリケーションやベンチマークを含む，Java Runtime Environment（JRE）のパフォーマンスを測定するためのベンチマークパッケージソフトである．パッケージソフトには，単一のアプリケーションを実行する JRE のパフォーマンスに焦点を当ている．そのことはハードウェアプロセッサとメモリサブシステムのパフォーマンスを反映しているが，ファイル I/O の低依存性を持っており，ネットワークのマシン間で I/O が含まれていない．SPECjvm2008 の作業負荷は，一般的な汎用アプリケーションの計算を模倣している．これらの特性は，このベンチマークは，クライアントとサーバーの両方のシステムを想定し，多種多様で基本的な Java のパフォーマンスを測定するように意図している．SPEC も Java ユーザーの経験を見つけ，パッケージソフトはベンチマークのスタートアップを含んでおり，ベースと呼ばれる必要なランタイムのカテゴリがある．ボックスの性能向上のため，JVM のいずれかのチューニングを行わずに実行する必要がある．

2.2.10　JHotDraw

JHotDraw は技術化と構造化が行われたグラフィックのための Java の GUI フレームワークである．ソフトウェアデザインのために開発が行われたが，すでに十分なほど機能が備わっている．そのデザインは，よく知られているデザインパターンに大きく依存している．

オープンソースプロジェクトである JHotDraw 作成の目的は次のとおりである．

- 開発者間における，このフレームワークのための広い聴衆を得るため
- JHotDraw に基づいて新しいアプリケーションを構築するため
- アプリケーションの開発が JHotDraw の開発に影響を与えるようにするため
- 新しく高度な機能を追加するため
- 更なる発展を促すため
- 新しい Java GUI のツールキッドに JHotDraw を移植するため
- 存在するコードを強化し，リファクタリングするため
- 新しいデザインパターンとリファクタリングを特定するため
- 適切な設計と柔軟なフレームワークの例を作るため
- JHotDraw に対して新しい Java API の関連性を検討するため（例：JHotDraw に用いる Java2D API の有用性）
- 学び楽しむため

本研究で JHotDraw を用いて実験を行ったのは，オープンソフトウェアとして公開されており，様々な改良が行われているためである．様々なバージョンの JHotDraw を用いることで，ユーザによるプログラムの変更として捉えることができる．

2.2.11　Java7

Java7 [20] では動的言語サポートのために動的に型付けされるメソッドの呼び出しを可能にしている．そのため新たなバイトコード命令の invokedynamic が追加され，それにともなった新しい構成要素の method handle というリンケージの仕組みが追加されている．invokedynamic は動的言語のためにターゲットの型を指定せずにメソッド呼び出しを表現できるようにしている．このままではメソッド呼び出しを解決することができないが，method handle を用いることで呼び出すメソッドとの関連を解決している．

Java7 のサポート機能を使うことで，呼び出しを行うバイトコードを生成するための変換を簡素化することができる．しかし，それまで存在しなかったメンバーを動的に追加することはできない．このため，5 章で示したようなデバッグ中にメンバーの追加を行うような変更はできない．

第3章　Reserved member 方式

我々は reserved member 方式による Java 言語のためのメンバーの実行時追加機構を提案する．我々が提案する reserved member 方式では，reserved member を事前に追加しておき，後から実行中にメンバーが追加されたときには，reserved member を修正して追加されたメンバーを実現する．これによって，HotSwap の範囲内で動的なメンバーの追加を可能にする．標準で用意されている HotSwap ではメソッドボディの変更のみが許され，シグネチャの変更ができない．そのため，reserved member の型はどのようなメンバーでも追加できるようにし，整合性をとるため動的に型変換を行うコードを加える．さらに複数のメンバー追加に対応するため，ディスパッチを行うコードを加えて実現する．

本システムは java.lang.instrument パッケージを通して HotSwap 機能を用い，実行されるプログラムを監視する．この機能を用いると，予め用意しておいた premain メソッドが main メソッドが実行される前に呼び出される．premain メソッドは，各クラスのロード時に本システムの方式に従ってクラス定義が変換されるように Java 仮想機械を HotSwap 機能を使って設定する．クラス定義を変換するコードは Javassist [12] を用いて我々が実装した．このコードは，各クラスの定義を次に述べる reserved member を元の定義に加えたものに置き換える．この置き換えは実行時ではなくロード時に一度だけ行うため，標準の機能で実現できる．

ユーザが実行中に動的にメンバーの追加を指示すると，外部プロセスが HotSwap 機能経由で Java 仮想機械に割り込みをかける．Reserved member の中身を変更して，追加を支持されたメンバーを実現したクラス定義を作り，元のクラス定義を HotSwap 機能を使って置き換える．これにより，メンバーの実行時の追加を実現する．

3.1　Reserved member の追加

Reserved member はどのようなメンバーが任意個の追加されても対応できるようにする必要がある．標準で用意されている HotSwap ではメンバーの追加を許していないため，各クラスの実行前に reserved member を事前に追加しておく．本システムの場合，reserved member の追加は本

システムが行うため，本システムを利用するユーザーは，事前にメンバーを追加しておくことを意識する必要がない．

3.1.1　フィールドの追加

準備しておくフィールドはどのような追加，任意個の追加どちらにも対応できるように用意する必要がある．そのため，String を key とし Object を value とする HashMap を reserved field とする．この String の key を用いてディスパッチを行う．このフィールド名は他のフィールドと重複しないように準備する．今回は reserved$ という名前で始まるフィールドは各クラスに定義されていないなど仮定を設けている．そのため，reserved field の宣言は以下のようなものになる．

HashMap reserved$ = new HashMap();

3.1.2　メソッドの追加

準備しておくメソッドはどのような追加，任意個の追加どちらにも対応できるように用意する必要がある．どのような追加にも対応するために，戻り値の型は Object とし，引数の型は Object の配列と String にしている．これによって後にどのようなメソッドの追加があっても Object の配列に変換することで引数に渡すことができ，戻り値を Object として受け取ることができる．引数の String はディスパッチ機能のための準備であり，追加されるメソッドを実現する際に利用する．さらにメソッド名は各クラスで定義されているものと重複しないように定めなければならない．今回は reserved$ という名前で始まるメソッドが存在しないなど仮定を設けている．そのため，reserved method の宣言は基本的に図 3.1 のようになる．

しかし，メソッドにはオーバーライドされる可能性があるため，一概に図 3.1 のようにメソッドを準備してはならない．もし，すべてのメソッドを図 3.1 のように準備するとオーバーライドされた場合に正しい振る舞いを実現できない．そのため，親クラスが String のような組み込みのクラスでなく，クラス定義の修正が可能なクラスである場合には図 3.2 のようなメソッドをロード時に追加する．追加するメソッドは親クラスの同名のメソッドを呼ぶだけのメソッドである．より詳細なオーバーライドに関する考慮は後で議論する．

```
1 Object reserved$(Object[] objects, String key){
2   return null;
3 }
```

図 3.1: 継承関係がない

```
1 Object reserved$(Object[] objects, String key){
2   return super.reserved$(objects, key);
3 }
```

図 3.2: 継承関係がある

3.1.3　コンストラクタの追加

コンストラクタはメソッドと異なり，戻り値の型とコンストラクタ名が決まっている．従って，名前を変えることで重複を避けることができない．そこで，重複を回避するために既存のものと重複しないダミーのクラスを作成し，ダミーのクラスを引数として持つことで他のコンストラクタと区別する．今回は Reserved$ という名前で始まるクラスが存在しないという仮定を設けている．戻り値の型はそのクラス自身であるので，変えることはできないがコンストラクタを追加するにあたり問題はない．引数はメソッドと同様に Object の配列と String，そして先ほどのダミーのクラス Reserved$ を引数として受け取る．引数の String はディスパッチ機能のための準備であり，追加されるメソッドを実現する際に利用する．以上より reserved constructor の宣言は以下のようなものになる．

< *classname* >(Object[] objects, String key, Reserved$ r){}

3.2　Reserved method の変更による追加されるメソッドの実現

ユーザーが追加の指示を行ったときには，実行前に追加しておいた reserved method を利用し，追加するメソッドの機能を実現する．3 章の典型的なユーザーの操作のように新しいクラス定義が本システムに渡される．新しいクラス定義と古いクラス定義を比較し，追加されたメソッドを見つけ出し，その追加されたメソッドの機能を reserved method に実現する．Reserved method に対して，メソッドボディのコピー，引数と戻り値の型変換，追加されたメソッドの切り替えを挿入することで追加されたメソッドの機能は実現できる．

例えば，図 3.3 のように Position クラスに distance メソッドが追加さ

```
1  public class Position {
2    int x,y;
3    public Position(int x, int y){
4      this.x = x;
5      this.y = y;
6    }
7    public static void main(String[] args){
8      Position p = new Position(19, 87);
9      double d = p.distance(5, 21);
10   }
11   public double distance(int i, int j){
12     double d2 = Math.pow(x − i, 2) + Math.pow(y − j, 2);
13     double d = Math.sqrt(d2);
14     return d;
15   }
16 }
```

図 3.3: Position クラスに distance メソッドを追加

れた時を考える．ユーザーはこのソースコードをコンパイルして，新しいクラス定義であるクラスファイルを準備し，本システムに与える．そして，本システムは reserved member が追加されている Position クラスに対して図 3.4 のような変更を加えて distance メソッドの追加を実現する．

3.2.1　Reserved method の呼び出し

新しく追加したメソッドを呼び出している場所では，代わりに reserved method を呼び出すように変換を行う．この際に動的ディスパッチ及びメソッド・オーバーライドを考慮して，追加するメソッドを呼び出す可能性があるメソッドの呼び出し式を全て変換する．追加するメソッドを呼び出すのはメソッドの追加時に一緒に新規に加えられたクラスだけとは限らず，元から存在しているクラスが呼び出す場合もあるため，すべてのコードを確認し，必要に応じて変換を行って標準の HotSwap で既存のコードと置き換える．

Reserved method の呼び出しでは，引数を Object 型の配列に変換し，加えて，呼び出したい本来のメソッドのシグネチャを文字列として引数とする．Object 型の戻り値は，元々呼び出されていたメソッドの戻り値の型に変換される．追加されたメソッドの切り替えで利用するキーを引数に持たせて呼び出しが行われる．

図 3.4 では 10, 11 行目が reserved method の呼び出しである．int 型の引数二つを Object 型の配列に変換し，distance メソッドのシグネチャの String を引数に持って，reserved method を呼び出している．Reserved method の戻り値は Object 型であるので，double 型に型変換を行う．

```
1  public class Position {
2    int x, y;
3    public Position(int x, int y){
4      this.x = x;
5      this.y = y;
6    }
7    public static void main(String[] args){
8      Position p = new Position(19, 87);
9      /* reserved method call */
10     double d = (double) p.reserved$(new Object{5, 21},
11       ``distance(I,I)D'');
12   }
13   public Object reserved$(Object[] v1, String v2){
14     if(v2.equals(``distance(I,I)D''){ // dispatch
15       /* parameterCast */
16       v2 = (int) v1[1];
17       v1 = (int) v1[0];
18       /* distance の処理のコピー*/
19       double v3 = Math.pow(x − v1, 2) + Math.pow(y − v2, 2);
20       double v4 = Math.sqrt(v3);
21       /* returnCast */
22       return Wrapper.make(v4);
23     }
24     return null;
25   }
26 }
27
28 public class Wrapper {
29   public static Object make(double d) {
30     return new Double(d);
31   }
32 }
```

図 3.4: Reserved method を利用して distance メソッドを実現

3.2.2　追加されたメソッドの切り替え

Reserved method では複数の追加されたメソッドの機能を保持する必要があり，引数で渡される String を使って切り替えを行う．追加されたメソッドは，コピーされたメソッドボディや引数，戻り値の型変換を追加で実現され，一つの機能として囲まれる．そのため，この機能を囲うような if 文を作ることで切り替えを行う．戻り値の型変換によって行いたい処理は必ず return 文を持っているので，else がなくても複数の追加されたメソッドが実行されることはない．追加されたメソッドの機能を切り替えるために追加されたメソッドのシグネチャを文字列にして，比較を行う．その文字列は reserved member の呼び出しの際に実引数の二番目として渡され，追加されたメソッドの切り替えのみに利用され，選択されたメソッドボディ内では利用されない．図 3.4 では 14 行目のような if 文が作成される．第二引数と事前に用意していた文字列を比較する．この文字列はメソッドのシグネチャであり，distance メソッドで，引数が int, int 型で，戻り値が double 型であることを表している．

3.2.3　引数の型変換

引数の型に関して整合性を取るため，reseved method の引数を適切な変数に準備する必要がある．単純に追加されるメソッドのボディをコピーしただけでは，整合性が取れないアクセスを行ってしまう．プリミティブ型に変換する場合には，アンボクシングのためのバイトコードが増える．ソースコードからバイトコードにコンパイルする場合にはオートボクシングが行われるが，バイトコードレベルでの変換なので，システムがアンボクシングを行うバイトコードを生成する必要がある．

Java では処理を行う際にメソッドの引数と局所変数は同じ変数として扱われ，前から順に番号付けされている．変数の前から順に引数が利用して，残りを局所変数が利用することを考慮し，Object の配列で渡された値を展開する．この展開の際に Object 型から適切な型に変換を行う．

単純に追加されるメソッドのボディをコピーした場合，一切の変更が加えられてないため，Object の配列や String 型の変数，何も用意されてない変数に整合性の取れないアクセスをしてしまう．そのため，先頭の引数が利用する変数に Object の配列で渡された値を変数として展開できればよい．この際に reserved method 本来の引数 Object の配列と String を考慮しなければならないが，String 型の変数はディスパッチのみで利用し，処理が選択された後は利用されない．また，Object の配列も同様に渡された値を展開した後では，Object の配列を利用されない．そのため，コピーしたメソッドボディではそのものを利用することがないため上書きしても

問題はない．よって，先頭の引数が利用する変数に Object の配列で渡された値を変数として展開できればよい．この展開を行う際に Object から本来の型に変換を行う．また展開を行う際に Object の配列から値を取り出すため，Object の配列を上書きしてしまうとそれ以上値を取り出すことができない．そのため，第二引数以降を展開してから，最後に Object の配列を本来の第一引数に上書きする．

また，利用していない局所変数に展開する方法もあるが，追加されるメソッドのボディに変更を加えて引数へのアクセスをその局所変数に置き換える必要があり，利用する局所変数が増えるため，上記の方法を選択して最適化を行った．

例として，distance メソッドが reserved method を利用して実現する場合，引数の型変換は図 3.4 の 16, 17 行目のようになる．バイトコードレベルでは，引数と局所変数を同様に扱うので，説明のため先頭から番号付けして v1, v2, … とする．型変換ではバイトコードレベルで行われるので，ソースコードレベルでは正しく記述できない．そのため，図 3.4 では類似したコードで表す．distance メソッドのボディでは int 型の v1, v2 を利用する．

3.2.4　メソッドボディのコピー

メソッドを追加するときは，そのメソッドボディをあらかじめ準備しておいた reserved method にコピーする．このようなコピーを行う際に問題となるのは this や super の扱いである．this に関しては同じクラスに定義されたメソッドにコピーが行われるため，変更を加える必要がない．super に関してはオーバーライドを考慮する必要がある．後で議論するがオーバーライドを考慮する際に変更を加えなくて良いように対応している．従って，直接メソッドボディをコピーするだけでよい．

例として，distance メソッドが reserved method を利用して実現する場合，コピーされた処理は図 3.4 の 19, 20 行目のようになる．

3.2.5　戻り値の型変換

戻り値に関しても型を変換する必要があり，プリミティブ型の時のみボクシングを行う．バイトコードレベルでは適切なボクシングを行うのが難しかったため，システム側で用意しておいたメソッドを呼び出す．そのメソッドを呼び出す時にはプリミティブ型の戻り値を引数として呼び出しを行い，void もプリミティブ型なので，必ず return 文が存在することになる．

例として，distance メソッドが reserved method を利用して実現する場合，戻り値の型変換は図 3.4 の 22 行目のようになる．distance メソッドは戻り値が double なので，プリミティブ型の値をボクシングして，Object 型として戻す Wrapper クラスの make メソッドを実行時に呼び出す．

3.3　Reserved Constructor の変更による追加されるコンストラクタの実現

コンストラクターは特別なメソッドとして扱えるため，reserved constructor は reserved method とほぼ同様に利用することができる．コンストラクターとメソッドの違いとして戻り値の型が自身のクラスであるということが決まっているため，戻り値の型変換は行わなくてよい．呼び出しではメソッドと同様に Object の配列と String を持つが，加えて重複を避けるためのクラスを引数として持つ必要がある．その際にインスタンスを作成する必要はなく，null を Reserved$ クラスに変換すればよい．

3.4　Reserved Field の変更による追加されるフィールドの実現

フィールドはボディが存在しないため，引数の型変換とボディのコピーと戻り値の型変換が必要なく，アクセスのみで実現する．呼び出しでは元々のフィールド名をキーにハッシュにアクセスを行う．

3.5　詳細の考慮

3.5.1　オーバーライドの考慮

メソッドはオーバーライドがあるため，追加する reserved method は親クラスの reserved method を呼び出すようにしていた．ユーザーによってメソッドが追加された場合，対象のクラスの reserved method に追加されるメソッドが実現される．この対応で，継承関係のあるクラスで既に定義されてないメソッドの追加は実現される．

追加するメソッドが継承関係のあるクラスで既に定義されている場合は，既に定義されているメソッドと reserved method でオーバーライドの関係性が生じる．異なるシグネチャ間でオーバーライドの関係を実現するのは難しいため，既存のメソッドを reserved method で処理を実現し，reserved method でオーバーライドの関係を実現する．

```
1  class Parent {}
2
3  class Child extends Parent {}
4
5  public class OverloadTest {
6          void test(Parent o) {
7                  System.out.println("Parent");
8          }
9
10         void test(Child o) {
11                 System.out.println("Child");
12         }
13
14         private void run() {
15                 test(new Parent());
16                 Parent p = new Parent();
17                 test(p);
18                 System.out.println();
19
20                 test(new Child());
21                 test((Parent)new Child());
22                 p = new Child();
23                 test(p);
24         }
25
26         public static void main(String[] _) {
27                 new OverloadTest().run();
28         }
29 }
```

図 3.5: オーバーロードの例

組み込みのクラスでは reserved member を追加することができないため，上記のように reserved method に処理を実現することができない．しかし，そのようなクラスは限定されているため，用意すべきメソッドは有限で収まる．そのため，あらかじめ対象メソッドを reserved member の追加と同じタイミングで追加を行うことで対応する．

3.5.2　オーバーロードの考慮

メソッドはオーバーロードがあるため，同じようにメソッド呼び出しを記述しても選択されるメソッドは判断が難しい．例としては図 3.5 のようなプログラムの場合を想定する．後半の三つの例の引数は実行時の型は Child ですが，最後の 2 例については静的な型が Parent である．これらの場合，test(Parent) が実行される．

オーバーロードが静的な型に基づくため，コンパイラが解決している．

これらの呼び出しは静的な型に依存しており，動的な追加を行った場合はどのように振る舞うかを想定しにくい．従ってソースコードレベルでは同じ記述であっても，バイトコードレベルでは異なるものとなる．そのため，バイトコード変換ライブラリを活用してるため，それらの問題は回避することができる．

3.5.3　修飾子など

HotSwap では修飾子も変更できないため，事前に準備しておく reserved member は修飾子に関しても考慮されて用意される．修飾子の一部はアクセス修飾子と呼ばれ，そのメンバーの振る舞いに影響を与える．そのため，複数種類の reserved member を準備することで対応するが，それにあわせて名前を変更して追加する必要がある．

コンストラクタはアクセス修飾子によって記述を制限されるが，すべてのアクセスを許可されたコンストラクタに置き換えても，振る舞いに影響はない．フィールドは static であるかどうかによって行われる処理が異なる．そのため，二種類の reserved field を用意しなければならない．他のアクセス修飾子に関しては振る舞いに影響しない．メソッドは static であるかどうか，private であるかどうかに影響を受け，その振る舞いを変える．protected や記述を省略したメソッドは，記述に制限があるが public なメソッドと同じ振る舞いであり， public なメソッドでその機能を実現する．そのため，四種類の reserved method を用意する．アクセス修飾子以外では synchronized は reserved method を利用してバイトコード変換を行うことで実現できる．本システムでは未実装である．修飾子と似たようなものに throws 節があるが，実行時には無視されるため問題ない．

3.5.4　制限

本システムは様々な要因で制限を設けている．Java では一つのメソッドで利用できる容量として 64KB という制限がある．そのため，ディスパッチ機能で一つの reserved member が複数の処理を保持することができるが，追加される量によってはその限界を超えてしまう．対応策としては，reserved member の数を増やすことで容量を増やすことが可能である．しかし，この対応策では実行前にその判断をしなければならず，予想を上回る追加が行われれば，同様な問題が発生する．

これまで説明したように，本システムは既に実行中のクラスを新しいクラスで置き換える．この変更が反映されるタイミングはユーザーが追加を指示したタイミングによって決まるため，一貫性が取れない可能性が

ある．標準の HotSwap の仕様により，既に呼び出しが行われ，アクティブなスタックフレームが存在する場合，メソッドを再定義しても，元のメソッドのバイトコードが引き続き実行される．再定義されたメソッドは新たに呼び出しが行われたときに初めて実行される．

また，すべてのクラスを対象に reserved member の呼び出しを変換するため，Java 既存の機能に影響を与えてる．例えば，リフレクションや RMI などがあげられる．さらに，本システムで追加を行う場合には呼び出し側の変換が必要となるため，HotSwap の仕様では再定義されたメソッドが呼び出されるタイミングであっても呼び出されないことがある．呼び出し側のメソッドが既にアクティブなスタックフレームを持っている場合にはメソッドの再定義は反映されない．本システムの実装基盤として Javassist を用いてるため，その制限を受けているが，提案手法である reserved member 方式の性質そのものではない．

第4章　バイトコード変換

本章では，Javassist の基礎的な説明を行い，その後に reserved member で用いるバイトコード変換について述べる．

Reserved member 方式を実現するには HotSwap を用いてクラス定義を変更する必要があり，その際にバイトコード変換を行う必要がある．通常，HotSwap を用いる際にはメソッドボディのみが変更されたクラス定義を受け取り，そのまま変換を行うことで直接的なバイトコード変換を行わずに利用することができる．本システムでは reserved member 方式のためにバイトコードレベルの変換であるため，ソースコードレベルでは記述不可能な内容も記述することができる．バイトコード変換の際に Javassist [12, 11] を用いて必要なバイトコードを生成した．

4.1　Javassist

Javassist（Java Programming Assistant）は Java バイトコード操作を簡単にしている．Javassist によって Java プログラムで実行時に新しいクラスを定義したり，JVM がクラスファイルをロードする際にクラスファイルを変更することができる．他のバイトコードエディタと異なり，Javassist はソースコードレベルとバイトコードレベルの二つの API が提供されている．ユーザがソースコードレベル API を使用すれば，Java バイトコードの仕様についての知識なしでクラスファイルを編集することができる．API 全体は Java 言語で記述されている．さらにバイトコードの挿入はソースコードの形式で指定でき，Javassist は急いでソースコード形式で記述されたものをコンパイルする．他方でバイトコードレベルの API は，ユーザに他のエディタのようなクラスファイルの直接編集を許可している．また，Javassist は，Java のバイトコードを操作するための Java ライブラリであり，構造リフレクションの機能を持つ．これにより，クラス構造を抽象化した API があるため，バイトコードに関する知識を持たなくてもプログラマはバイトコードの変換を行うことができる．

4.1.1　サンプルプログラム

Javassist をより直感的に解説するため，図 4.1 のサンプルコードを元にバイトコード変換の例を示す．この例では，クラス名とメソッド名を用いて指定されたメソッドを取り出し，そのメソッドの先頭にバイトコードを挿入することで処理を加えてる．まず，4 行目では ClassPool オブジェクトを取得している．ClassPool オブジェクトはクラスを表す CtClass (compile-time class) のコンテナとなっている．5 行目で引数のクラス名から，CtClass インスタンスを生成している．6 - 8 行目では CtClass オブジェクトは凍結されている場合があるため，変更を行えるように凍結を解除する．凍結された CtClass オブジェクトはそれ以上変更はできない．クラスの凍結ではそのオブジェクトに含まれるデータ構造の一部を削除することで，メモリ消費を抑えることもできる．9 行目では，そのクラスで定義されているメソッドの中から引数のメソッド名であるメソッドを表す CtMethod インスタンスを生成している．10 行目でバイトコードを挿入している．最後にクラス全体を toBytecode メソッドでバイトコード配列にして返している．

```
1  public byte[] getNewByteCode(String className, String
2      methodName, String code) throws NotFoundException,
3      CannotCompileException, IOException {
4   ClassPool cp = ClassPool.getDefault();
5   CtClass cc = cp.get(className);
6   if(cc.isFrozen()){
7     cc.defrost();
8   }
9   ctMethod cm = cc.getDeclaredMethod(methodName);
10  cm.insertBefore(code);
11  return cc.toBytecode();
12 }
```

図 4.1: Javassist によるバイトコードの変換例

4.2　Reserved member 方式の実現

3 章で reserved member 方式を利用した場合の振る舞いを述べた．ここでは，その振る舞いするバイトコードどのように生成するかを述べる．図 3.1，3.3，3.4 のような変換を行う．

```
1 public byte[] addMember(String className){
2     String code = "Object reserved$(Object[] objects, "
3         + "String key){ return null; }";
4     ClassPool cp = ClassPool.getDefault();
5     CtClass cc = cp.get(className);
6     CtMethod m = CtNewMethod.make(code, cc);
7     cc.addMethod(m);
8     return cc.toBytecode();
9 }
```

図 4.2: Reserved member の追加

4.2.1　Reserved member の追加

本システムによって実行前に reserved member を追加する．コンパイラを改造することでユーザが意識せずに reserved member を追加しておくこともできるが，java.lang.instrument パッケージによって，ロード時にバイトコード変換を行えるため，本システムではこちらの方法を採用している．クラス定義に加えたいメンバーの定義は 3 章の通りであり，Javassist のソースレベルの API を用いることで簡単に追加することができる．そのため，reserved member は文字列として reserved member の宣言を与え，reserved member を作成する．クラスのロード時にクラス名も取得することができ，それを用いて図 4.2 のメソッドを呼び出す．クラス名から CtClass を取得し，新たなメンバーを追加して，そのバイトコードを返す．

4.2.2　Reserved member の呼び出し

新しく追加したメンバーを呼び出している場所では，代わりに reserved member を呼び出すように変換を行う．追加するメンバーがメソッドである場合，動的ディスパッチ及びメソッド・オーバーライドを考慮して，追加するメソッドを呼び出す可能性があるメソッドの呼び出し式を全て変換する．追加するメソッドを呼び出すのはメソッドの追加時に一緒に新規に加えられたクラスだけとは限らず，元から存在しているクラスが呼び出す場合もあるため，すべてのコードを確認し，必要に応じて変換を行って標準の HotSwap で既存のコードと置き換える．

Reserved member の呼び出し側は Javassist が提供する ExprEditor を用いて置き換える．Javassist のコンパイラでは $ から始まるいくつかの特別な意味を持つ識別子が用意されている．$_ が戻り値，$r が戻り値の型，$0 が実際の引数の 0 番目，$args が Object の配列に変換した引数を表している．これらの識別子を利用し，呼び出し側に実行時に型を変換するコードを追加する．

フィールドのアクセスは読み出しと書き込みがあり，以下のコードを用いて変換する．

$_ = ($r) $0.reserved$.get(< $fieldName$ >);

$0.reserved$.put(< $fieldName$ > , $1);

読み出しは上の，書き込みは下の文を Javassist を使ってコンパイルし，目的となる箇所に得られたバイトコードを挿入する．読み出し用に得られるバイトコードは，対象となるオブジェクトの reserved$ フィールドに対して get メソッドを呼び出して Object 型の値を得る．これを本来の型に変換して，元のフィールドの読み出しの結果の値とする．書き込み用に得られるバイトコードは，書き込む値を Object 型の値に変換し，対象となるオブジェクトの reserved$ フィールドに対して put メソッドを呼び出して書き込みを実現する．プリミティブ型の場合はボクシングを行うバイトコードを加える．

一方，メソッド呼び出しは以下のコードを用いて変換する．

$_ = ($r) $0.reserved$($args, < $methodsignature$ >);

この文を Javassist でコンパイルして得られるバイトコードは，元々のメソッド呼び出しの引数全体を Object 型の配列に変換し，それと呼び出したい本来のメソッドのシグネチャを引数として reserved$ メソッドを呼び出す．Object 型の戻り値は，元々呼び出されていたメソッドの戻り値の型に変換される．変換後の値は元々呼び出されていたメソッドの戻り値であるかのように，続くバイトコードに渡される．

最後にコンストラクタの呼び出しは以下のコードを用いて変換する．< $classname$ > とある箇所は，実際に作成するオブジェクトのクラス名で置き換えてから Javassist に渡してコンパイルする．コンストラクタには元の実引数列の他，呼び出したい本来のコンストラクタのシグネチャと Reserved$ 型の null 値が引数として渡される．

$_ = new < $classname$ >($args, < $constructorsignature$ > , (Reserved$) null);

4.2.3　追加されたメンバーの切り替え

Reserved member では複数の処理を保持する必要があり，引数で渡される String を使って切り替えを行う．コピーされた処理や引数，戻り値の型変換が追加されたメンバーの処理である．そのため，この処理を囲うような if 文を作ることでディスパッチを行う．returnCast によって行いたい処理は必ず return 文を持っているので，else がなくても複数の処理が行われることはない．追加するメンバーの機能を選択するために追加するメソッドやコンストラクタのシグネチャを文字列にして，比較を行う．そ

の文字列は Reserved member の呼び出しの際に実引数の二番目として渡され，処理の選択のみに利用され，選択された処理内では利用されない．図 3.4 のような if 文が作成される．

上記のような if 文を作成するために，切り替えのためのキーと reserved member，実現されたメンバー，実行時に渡されるキーの引数の位置を与えて，図 4.3 のメソッドを呼び出す．5 ，6 行目のように空のバイトコードを作成する．8 行目によって実行時に aloadKey 番目の引数を取り出すようになり，9 行目のようにキーを定数としてスタックに積む．この二つのバイトコードによって二つの文字列がスタックに積まれている状態になる．この状態で String クラスの equals メソッドを呼び出すようにするため，String クラスの CtClass を取得し，equals メソッドの CtMethod を取得する．addInvokevirtual で呼び出しが行えるようにクラスとメソッド名，戻り値の型，引数の型を与えることで設定する．その結果の boolean がスタックに積まれた状態になる．この結果を ifeq を追加して，次にオフセットが加わる．実現されるメンバーによってはこのオフセットが変わるため，仮の値を挿入して，現在の位置を保持し，後で変更を加える．実現したメンバーのバイトコードを挿入する．挿入し終わった位置が先ほどのオフセットの位置であるので，24 行目でオフセットを書き直す．残りの処理で，切り替えも含めた実現されたメンバーを挿入し，利用する変数やスタックの調整を行う．

4.2.4　引数の型変換

引数の型に関して整合性を取るため，動的に引数の型を変換するコードが必要である．単純に追加するメンバーの処理をコピーしただけでは，整合性が取れないアクセスを行ってしまう．例えば，図 3.4 のように distance メソッドが reserved method を利用して実現する場合の引数の型変換を考慮する．

挿入するバイトコードは図 4.4 のコードで生成する．allParameterCast は Object の配列として受け取った実引数列を本来の型の引数列に変換するバイトコードを生成するメソッドである．一番目の引数は Object の配列の場所を示す．これはメンバーが static であるかどうかで変わり，0 か 1 が渡される．残りの引数は空のバイトコードと，実引数の型の配列である．Object の配列から実引数を一つ取り出し，型変換して本来の仮引数に代入するバイトコードを生成するのが parameterCast の役割である．バイトコードレベルでは，仮引数は特別な局所変数である．型によって利用するワード数が異なるため，is2ward を用いて確認する．is2ward は型が double と long であるかどうかを判断する．

Object の配列は第一引数として reserved method に渡されるので，allParameterCast はまず追加されるメソッドの引数がなければ，引数の型変換は行われないため，空のバイトコードを返す．第一引数の利用するワード数を確認し，第二引数以降があればそれを型変換して本来の仮引数に代入するコードを生成する．その後，Object の配列からメソッドの第一引数を取り出して本来の仮引数に代入するバイトコードを生成し，生成済みのバイトコードの末尾に追加する．第一引数に対応するバイトコードレベルの局所変数には，元々 Object の配列が格納されているので第一引数の型変換は最後に実行しなければならない．実行後は Object の配列はそこから取り出した第一引数の値で上書きされてしまう．

parameterCast はまず Object の配列から指定された要素の Object を取り出す．次に変換後の型がプリミティブ型かどうかを確認して挿入するバイトコードを切り替える．プリミティブ型の場合は一度 Object 型の値をラッパークラスの型へ型変換し，プリミティブ型の値を取り出す．プリミティブ型でない場合は Object 型の値を単に目的の型へ型変換するだけである．最後に型変換後のオブジェクトを目的の仮引数に対応するバイトコードレベルの局所変数に代入する．parameterCast はこのような処理を行うバイトコードを生成する．

4.2.5　処理のコピー

メンバーを追加するときは，そのメンバーの中身をあらかじめ準備しておいたメンバーの中にコピーするだけで良いため，必要なバイトコード変換はない．

4.2.6　戻り値の型変換

同様に戻り値に関しても型を変換するバイトコードを生成し，挿入する．戻り値ではプリミティブ型の時のみ変換を行い，プリミティブ型の値をラッパークラスに変換するバイトコードを生成する．ラッパークラスを作るコードをシステムが用意しているので，プリミティブ型の戻り値を引数とし，それらのメソッドを呼び出す．void もプリミティブ型なので，必ず return 文が存在することになる．

例えば，long の場合は図 4.5 のようなコードでプリミティブ型の値をラッパークラスに変換するバイトコードを生成する．Wrapper クラスの make メソッドを実行時に呼び出すバイトコードを実行してプリミティブ型の値をラッパークラスに変換する．returnCast メソッドでは return を行っているバイトコードを見つけ出し，メソッド呼び出しを挿入するため

に隙間を作り，変換を行うためのメソッド呼び出しを挿入し，その結果を返すバイトコードを追加している．

```
1  private static void insertIf(String key,
2      CtBehavior reservedMember, CtBehavior madeMember,
3      int aloadKey) throws NotFoundException, BadBytecode {
4  // bytecode の作成
5  Bytecode bytecode = new Bytecode(reservedMember
6      .getMethodInfo().getConstPool());
7  // if stmt
8  bytecode.addAload(aloadKey);
9  bytecode.addLdc(key);
10 CtClass ctString = ClassPool.getDefault()
11     .get("java.lang.String");
12 CtMethod eqMethod = ctString.getDeclaredMethod("equals");
13 bytecode.addInvokevirtual(ctString, "equals",
14     eqMethod.getReturnType(), eqMethod.getParameterTypes());
15 bytecode.addOpcode(Bytecode.IFEQ);
16 int pc = bytecode.currentPc();
17 bytecode.addIndex(0);
18 // add body
19 byte[] bs = madeMember.getMethodInfo().getCodeAttribute()
20     .getCode();
21 for (byte b : bs) {
22     bytecode.add(b);
23 }
24 bytecode.write16bit(pc, bytecode.currentPc() - pc + 1);
25 // insert
26 CodeAttribute codeAttribute = reservedMember.getMethodInfo()
27     .getCodeAttribute();
28 CodeIterator iterator = codeAttribute.iterator();
29 int pos = iterator.insertEx(bytecode.get());
30 iterator.insert(bytecode.getExceptionTable(), pos);
31 // max local & stack
32 int maxLocals = Math.max(codeAttribute.getMaxLocals(),
33     madeMember.getMethodInfo().getCodeAttribute()
34     .getMaxLocals());
35 codeAttribute.setMaxLocals(maxLocals);
36 int maxStack = Math.max(codeAttribute.getMaxStack(),
37     madeMember.getMethodInfo().getCodeAttribute()
38     .getMaxStack());
39 codeAttribute.setMaxStack(maxStack);
40 }
```

図 4.3: 実現されたメンバーの切り替え

```
1  public Bytecode allParameterCast(int objectsLoad,
2      Bytecode bytecode, CtClass[] parameterTypes) {
3    if (parameterTypes.length == 0) {
4      return bytecode;
5    }
6    int firstStore = is2word(parameterTypes[0]) ? 2 : 1;
7    int store = firstStore + objectsLoad;
8    for (int i = 1; i < parameterTypes.length; i++) {
9      parameterCast(bytecode, objectsLoad, i, store,
10         parameterTypes[i]);
11     store += is2word(parameterTypes[i]) ? 2 : 1;
12   }
13   parameterCast(bytecode, objectsLoad, 0, objectsLoad,
14       parameterTypes[0]);
15   return bytecode;
16 }
17 public boolean is2word(CtClass parameterType) {
18   return parameterType == CtClass.doubleType
19         || parameterType == CtClass.longType;
20 }
21 public void parameterCast(Bytecode bytecode, int objectLoad,
22    int iconst,int store, CtClass parameterType) {
23   bytecode.addAload(objectsLoad);
24   bytecode.addIconst(iconst);
25   bytecode.add(Bytecode.AALOAD);
26   if (parameterType.isPrimitive()) {
27     CtPrimitiveType pt = (CtPrimitiveType) parameterType;
28     String wrapperName = pt.getWrapperName();
29     bytecode.addCheckcast(wrapperName);
30     bytecode.addInvokevirtual(wrapperName,
31       pt.getGetMethodName(), pt.getGetMethodDescriptor());
32   } else {
33     bytecode.addCheckcast(parameterType);
34   }
35   bytecode.addStore(store, parameterType);
36 }
```

図 4.4: 引数の型変換を行うバイトコードを生成する Javassist を用いたコード

```
1  package hayafune.reserved.agent;
2  public class Wrapper {
3    public static Object make(long l) {
4      return new Long(l);
5    }
6  }
7
8  public class Redefine{
9   public void returnCast(CtMethod addMethod, CtMethod
10        declaredMethod) throws NotFoundException, BadBytecode {
11    MethodInfo methodInfo = declaredMethod.getMethodInfo();
12    ConstPool cp = methodInfo.getConstPool();
13    CtClass returnType = addMethod.getReturnType();
14    if (returnType.isPrimitive()) {
15      CodeIterator ci = methodInfo.getCodeAttribute().iterator();
16      CtPrimitiveType pt = (CtPrimitiveType) returnType;
17      while (ci.hasNext()) {
18        int pos = ci.next();
19        int c = ci.byteAt(pos);
20        if (c == Opcode.LRETURN) {
21          ci.writeByte(Opcode.NOP, pos);
22          ci.insertGap(pos, 3);
23          ci.writeByte(Opcode.INVOKESTATIC, pos);
24          ci.write16bit(cp.addMethodrefInfo(cp
25            .addClassInfo("hayafune.reserved.agent.Wrapper"),
26             "make", "(J)Ljava/lang/Object;"), pos + 1);
27          ci.writeByte(Opcode.ARETURN, pos + 3);
28        }
29      }
30    }
31  }
32 }
```

図 4.5: 戻り値（long）の型変換を行うバイトコードを生成する Javassist を用いたコード

第5章　実験

Reserved member 方式では準備や呼び出し，メモリ使用量のオーバーヘッドが大きいと考えられる．そこで，我々は自作のマイクロベンチマークや SPECjvm2008 を用いて，オーバーヘッドを計測した．動作環境は，CPU は Intel Core2Duo 1.83GHz, メモリは 1.5 GB, OS は WindowsXP である．

5.1　Reserved member の追加

Reserved member 方式でオーバーヘッドが大きいと予想される実行前に reserved member を追加するオーバーヘッドを計測する．今回はロード時に全クラスに reserved member を追加するシステムを用いて計測を行った．

5.1.1　プログラム

Reserved member を追加するオーバーヘッドを計測するため，非常に多数のクラスを準備する．各クラスは図 5.1 のようなプログラムを用いてロードする．このプログラムを通常通り実行した場合と reserved member を追加せず HotSwap の機能のみ有効な場合，reserved member を 5 個追加して，新しいメンバーを追加できる場合の実行時間を比較する．ウォーミングアップとしてまず 1000 個の異なるクラスをロードした．その後，1000 個の異なるクラスをロードする時にかかる時間を計測した．従って図 5.1 の n と m は 1000 である．

5.1.2　結果

結果は表 5.1 の通りである．java.lang.instrument パッケージが提供する機能を用いて，実行されているプログラムを監視するオーバーヘッドであり，約 37 %かかっている．また reserved member の追加を行った場合は約 4 倍程のオーバーヘッドがかかっている．

```
1  class Main {
2   public static void main(String[] args) {
3     // warmup
4     // n 回ロードを行う
5     new A0();
6     new A1();…
7
8     new An−1();
9
10    // m 回ロードを行う
11    long start = System.currentTimeMillis();
12    new An();
13    new An+1();…
14
15    new An+m−1();
16    long end = System.currentTimeMillis();
17    System.out.println(end − start);
18  }
19 }
```

図 5.1: クラスをロードするマイクロベンチマーク

	平均（ms）	標準偏差（ms）
通常通り	0.410	0.015
HotSwap 機能のみ	0.561	0.007
Reserved member を追加	1.642	0.036

表 5.1: 各条件下のクラスのロード時間

5.2　Reserved method の呼び出し

Reserved member 方式では追加されたメンバーの振る舞いが正しくなるように必要な処理を加えているため，そのオーバーヘッドを計測する．オーバーヘッドを計測するため，ユーザがメンバー追加を行ったのと同様なプログラムがシステム上で実行されるように，ロード時に必要な変換を行った．図 5.2 のプログラムをロード時に図 5.3 に変換した．従ってプログラムが実行される際には reserved member の準備と呼び出し，変更の三つの処理が行われる．

この変換によって一部のメソッド呼び出しが reserved method に置き換えることができた．しかし，この変換を行う際に選択したメソッドが多く呼び出されていないと計測が行えない．そのため，ベンチマークの中でどれぐらいの回数呼び出しが行われているかを確認する必要がある．

ここでは SPECjvm2008 の一部のベンチマークに対して，ロード時に

```
1  class ExtOutputStream {
2      public static boolean isDosNewLine(byte[] b, int index){
3          return b[index] == 10;
4      }
5      public static boolean isUnixNewLine(byte[] b, int index){
6          int incIndex = index + 1;
7          return b[index] == 13 && incIndex != b.length
8                && b[index] == 10;
9      }…
10
11 }
```

図 5.2: SPECjvm2008 の XML ベンチマーク

```
1  class ExtOutputStream {
2      public static Object staticReserved$(Object[] objects,
3        String key){
4          if(key.equals(“isUnixNewLine([BI)Z”)){
5              /* parameterCast */
6              /* isUnixNewLine のコピー*/
7          }
8          if(key.equals(“isDosNewLine([BI)Z”)){
9              /* parameterCast */
10             /* isDosNewLine のコピー*/
11         }
12         return null;
13     }…
14
15 }
```

図 5.3: SPECjvm2008 の XML ベンチマークの一部を reserved method で実現

メソッドの呼び出し回数を計測するための機構を織り込み，それを用いて呼び出し回数を計測した．その結果が図 5.4，5.5，5.6，5.7，5.8 である．

これによって各ベンチマークでメソッド呼び出しを確認することができたが，XML 以外のベンチマークでは，多くの部分を占めるメソッドが，内部クラスのメソッドであったり，標準ライブラリのメソッドをオーバーライドしたメソッドであった．そのため，reserved method の性能を評価するために適さず，XML のみで変換を行い，オーバーヘッドを計測した．

XML ベンチマークでは transform.ExtOutputStream.isDosNewLine と transform.ExtOutputStream.isUnixNewLine がそれぞれ約 33 % づつを占めている．よって，通常通り実行した場合（図 5.2）と約 66 % のメソッドを reserved method で実現した場合（図 5.3）を比較する．結果は図 5.9 である．

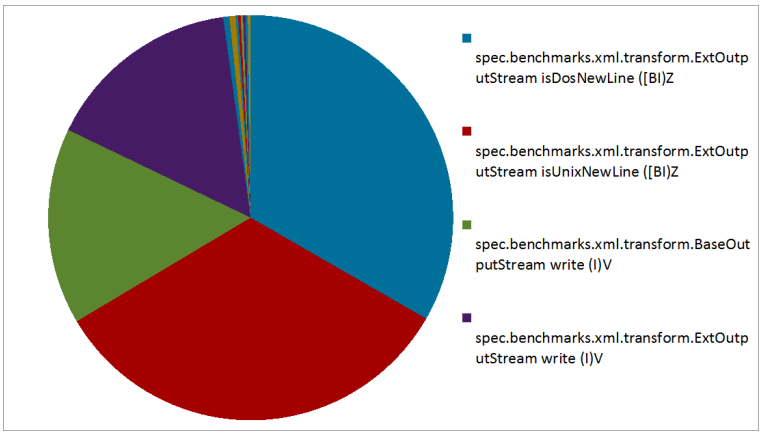

図 5.4: XML ベンチマークにおける呼び出しの割合

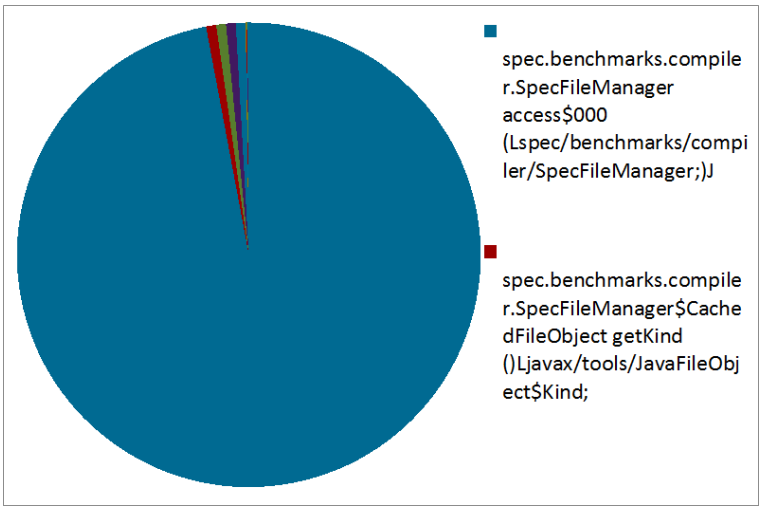

図 5.5: compiler ベンチマークにおける呼び出しの割合

通常通り実行した場合は平均 19.51 ops/m であり，標準偏差 0.386 ops/m である．Reserved method で実現した場合は平均 18.88 ops/m であり，標準偏差 0.967 ops/m である．約 3.5 % のオーバーヘッドとなり，非常に小さくエラーバーが重なるほどであることがわかる．

5.3　メモリ使用量

Reserved member を追加することでクラスファイルが大きくなり，それによってメモリの使用量が増加すると考え，通常のクラスファイルと re-

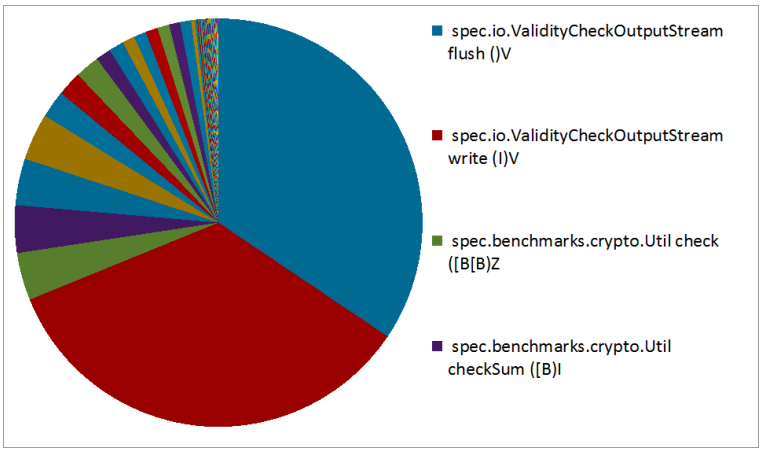

図 5.6: crypto ベンチマークにおける呼び出しの割合

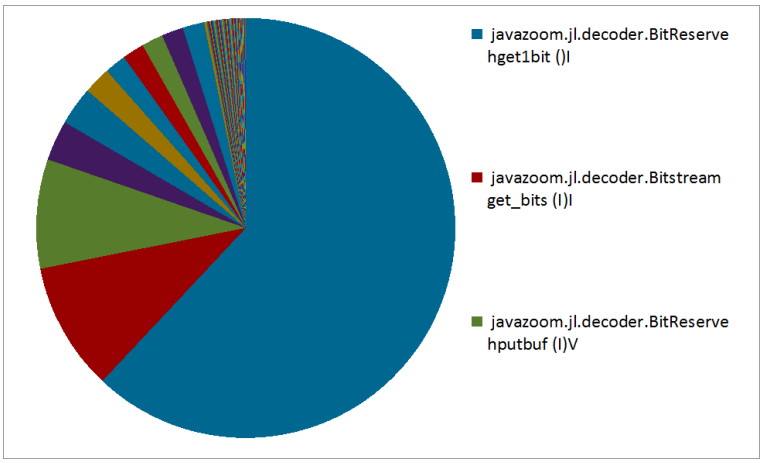

図 5.7: mpegaudio ベンチマークにおける呼び出しの割合

served member を追加したクラスファイルの大きさを比較した．SPECjvm2008 の spec.benchmarks.check ，spec.benchmarks.sunflow パッケージを除いて変換を行った．その結果，945 KB と 1038 KB となり，約 10 % の増加した．

5.4　JHotDraw を用いたコード差し替え機能の評価

Reserved member 方式を用いることでユーザが可能となる変更がどの程度であるかについて述べる．ここではユーザによる変更をオープンソフ

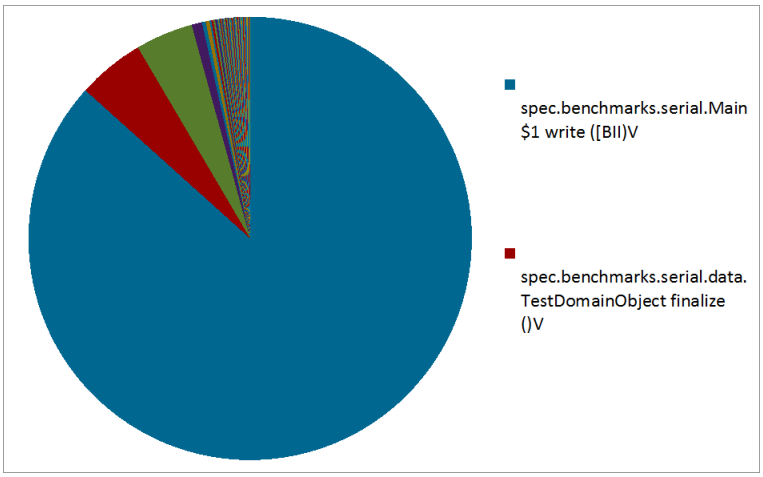

図 5.8: serial ベンチマークにおける呼び出しの割合

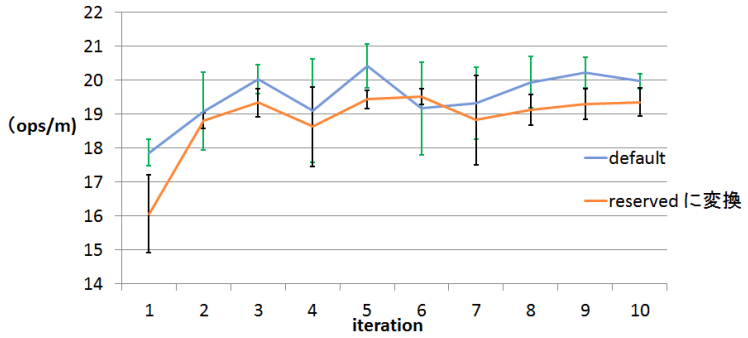

図 5.9: 各期間ごとのオペレーション速度

トウェアのバージョンアップとしてとらえ，実験を行った．JHotDraw の複数のバージョンをダウンロードしその差分を調べ，本システムを用いてその差分を変更できるかを確かめることでコード差し替え機能の評価を行った．

変更内容をまとめるために変更内容を四種類に分類しておく．一つ目がコメントやインデント，import などの変更でバイトコードレベルでは影響のない変更である．二つ目がメソッドボディに対する変更で標準の HotSwap 機能で解決できる変更である．三つ目が reserved member 方式を用いたことによって解決できる変更である．四つ目が本システムを用いて解決できない変更である．表の中で利用している（i）はコメント等の変更箇所を示し，(ii）は標準の HotSwap 機能で変更できる箇所を，(iii）は reserved member 方式を用いることで変更できる箇所を，(iv）は本シ

ファイル名	(i)	(ii)	(iii)	(iv)
AbstractCompositeFigure.java	2	1	0	0
AbstractDrawingViewAction.java	1	1	0	0
DefaultDrawing.java	1	1	0	0
DefaultDrawingView.java	4	2	0	0
Drawing.java	2	0	0	0
DrawingColorChooserHandler.java	8	1	0	0
JComponentPopup.java	3	2	1	0
NanoXMLDOMInput.java	2	0	0	0
PaletteColorSlidersChooser.java	0	1	0	0
PaletteLookAndFeel.java	4	1	0	0
QuadTreeDrawing.java	1	1	0	0

表 5.2: JHotDraw のバージョン 7.5 と 7.5.1 の変更箇所

ステムでは変更できない箇所を示す.

本システムは JHotDraw のバージョン 7.5 を実行して，その実行中に JHotDraw のバージョン 7.5.1 のクラスにすべて差し替えることに成功した．このバージョンアップにおける差分は 11 個の java ファイルが変更されている．それに伴いドキュメントの html ファイルやビルドを行う xml ファイルなどが変更されているが，動的に振る舞いを変更するのに影響を与えない．先ほどの 11 個の java ファイルに対して行われた変更をまとめるため，変更は表 5.2 の通りである．Reserved member 方式を用いたことですべての変更が反映できた.

同様に JHotDraw のバージョン 7.4 からバージョン 7.4.1 に差し替えることに成功した．java ファイルはメソッドボディの変更のみであり，標準の HotSwap 機能で解決できる変更であった.

同様に JHotDraw のバージョン 7.4.1 からバージョン 7.5 に差し替えを行おうとしたが，いくつか問題があり，すべてのクラスの差し替えは行えなかった．大きなバージョンアップでは様々な変更が含まれるため，reserved member 方式では解決できない場合が存在している．その多くが継承関係に対して変更が行われている場合であり，それに伴い多くのメンバーが変更されてる．また，java ファイル以外にも動的な振る舞いに影響があるライブラリなどのファイルにも変更があり，変更を反映することができない.

第6章　まとめと今後の課題

本論文において我々は，reserved member 方式によるメンバーの実行時追加機構を提案した．実行前に reserved member を追加することで，実際に動的な追加を行わないように準備する．ユーザーがメンバーの追加の操作を行ったときに，reserved member を利用して追加するメンバーを実現し，呼び出し側を変換する．これらによって動的なメンバーの追加を実現した．

今後の課題としては，システムの拡張を行い実用性を高めることである．特に eclipse のプラグインとして組み込むことが一つあげられる．本システムの現状ではクラス定義の変更についてユーザによる変更の指示だけであり，変更のタイミングに関して十分な API を備えていない．そこで，eclipse に備わっているデバッグ機能を拡張することで，機能性や汎用性として実用的なシステムを作り上げることができる．その際には Java Platform Debugger Architecture（JPDA）を理解し，必要な変更を加える．JPDA の一部に instrument パッケージは含まれているため，本システムを実装基盤に組み込むことは難しくないと考えられる．

同様に実際に利用されているシステムと組み合わせ実用性を高めるなら，本システムは svn などのバージョン管理システムなどと相性がよいと考えられる．クラス定義に関してより細かな変更やその変更を記憶することができるためである．

また，他のシステムの実装基盤として利用する場合，追加可能である 64 KB を超える場合がある．そのため，他のアプリケーションを reserved member で実装するようにすべて置き換えた場合，何個の reserved member を用意すれば対応できるかなど実用性を考慮したい．さらにメンバーの追加を可能にするだけでなく，メンバーの削除やクラス階層構造の変更を行えるようにしたい．

実験としてすべてのメンバーを reserved member を用いて実現することによる最悪のケースを想定する．現状の実験ではメソッドの呼び出し回数を調べており，比較的悪いケースを想定しているが，呼び出し回数が少なくてもオーバーヘッドが大きい場合があると思われる．

参考文献

[1] : The Jikes Research Virtual Machine, `http://jikesrvm.org/`.

[2] Apache Software Foundation: THE APACHE ANT PROJECT, `http://ant.apache.org/`.

[3] Bergel, A., Bergel, R., Ducasse, S. and Wuyts, R.: The Classbox Module System (2003).

[4] Bergel, A., Ducasse, S. and Nierstrasz, O.: Classbox/J: controlling the scope of change in Java, *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, New York, NY, USA, ACM, pp. 177–189 (2005).

[5] Bergel, A., Ducasse, S., Nierstrasz, O. and Wuyts, R.: Classboxes: controlling visibility of class extensions, *Comput. Lang. Syst. Struct.*, Vol. 31, pp. 107–126 (2005).

[6] Bergel, A., Ducasse, S. and Wuyts, R.: Classboxes: A Minimal Module Model Supporting Local Rebinding, *JMLC*, Vol. 2789, pp. 122–131 (2003).

[7] Bockisch, C., Arnold, M., Dinkelaker, T. and Mezini, M.: Adapting virtual machine techniques for seamless aspect support, *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, New York, NY, USA, ACM, pp. 109–124 (2006).

[8] Bockisch, C., Haupt, M., Mezini, M. and Mitschke, R.: Envelope-Based Weaving for Faster Aspect Compilers, *NODe/GSEM*, pp. 3–18 (2005).

[9] Bockisch, C., Haupt, M., Mezini, M. and Ostermann, K.: Virtual machine support for dynamic join points, *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 83–92 (2004).

[10] Chiba, S., Igarashi, A. and Zakirov, S.: Mostly modular compilation of crosscutting concerns by contextual predicate dispatch, *ACM SIGPLAN Notices*, Vol. 45, No. 10, pp. 539–554 (2010).

[11] Chiba, S.: Javassist Home Page, `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.

[12] Chiba, S.: Load-Time Structural Reflection in Java, *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, London, UK, Springer-Verlag, pp. 313–336 (2000).

[13] Clifton, C., Leavens, G. T., Chambers, C. and Millstein, T.: MultiJava: modular open classes and symmetric multiple dispatch for Java, *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, New York, NY, USA, ACM, pp. 130–145 (2000).

[14] Hölzle, U., Chambers, C. and Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches, *ECOOP'91 European Conference on Object-Oriented Programming*, Springer, pp. 21–38 (1991).

[15] Horie, M., Morita, S. and Chiba, S.: Distributed dynamic weaving is a crosscutting concern, *Proceedings of the 2011 ACM Symposium on Applied Computing*, ACM, pp. 1353–1360 (2011).

[16] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, London, UK, Springer-Verlag, pp. 327–353 (2001).

[17] Malabarba, S., Pandey, R., Gragg, J., Barr, E. and Fritz Barnes, J.: Runtime support for type-safe dynamic Java classes, ECOOP 2000—Object-Oriented Programming, pp. 337–361 (2000).

[18] Millstein, T., Reay, M. and Chambers, C.: Relaxed MultiJava: balancing extensibility and modular typechecking, *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, New York, NY, USA, ACM, pp. 224–240 (2003).

[19] Nicoara, A., Alonso, G. and Roscoe, T.: Controlled, systematic, and efficient code replacement for running java programs, *Eurosys*

'08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, New York, NY, USA, ACM, pp. 233–246 (2008).

[20] Oracle Corporation and/or its affiliates: New JDK 7 Feature: Support for Dynamically Typed Languages in the Java Virtual Machine, `http://java.sun.com/developer/technicalArticles/DynTypeLang/`.

[21] Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F. and Martelli, L.: JAC: an aspect-based distributed dynamic framework, *Softw. Pract. Exper.*, Vol. 34, No. 12, pp. 1119–1148 (2004).

[22] Popovici, A., Alonso, G. and Gross, T.: Just-in-Time Aspects: Efficient Dynamic Weaving for Java, *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pp. 100–109 (2003).

[23] Popovici, A., Gross, T. and Alonso, G.: Dynamic weaving for aspect-oriented programming, *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 141–147 (2002).

[24] Sato, Y., Chiba, S. and Tatsubori, M.: A selective, just-in-time aspect weaver, *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, New York, NY, USA, Springer-Verlag New York, Inc., pp. 189–208 (2003).

[25] Standard Performance Evaluation Corporation: SPECjvm2008 (Java Virtual Machine Benchmark), `http://www.spec.org/jvm2008/index.html`.

[26] The Eclipse Foundation: The AspectJ project, `http://www.eclipse.org/aspectj/`.

[27] The Java project team: $Java^{(TM)}$ java.lang.instrument Package., `http://download.oracle.com/javase/6/docs/technotes/guides/instrumentati%on/index.html`.

[28] Villazón, A., Binder, W., Ansaloni, D. and Moret, P.: Advanced runtime adaptation for Java, *GPCE '09: Proceedings of the eighth*

international conference on Generative programming and component engineering, New York, NY, USA, ACM, pp. 85–94 (2009).

[29] Würthinger, T., Wimmer, C. and Stadler, L.: Dynamic code evolution for Java, *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, New York, NY, USA, ACM, pp. 10–19 (2010).