平成16年度 修士論文

既存Javaプログラム向け
分散化支援システムの開発

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 03M37200

須永 豊

指導教員

千葉 滋 助教授

平成17年2月7日

概要

本研究は、分散ソフトウェアの開発を支援するシステム Jacross を提案する。Java では分散ソフトウェアを開発するためのシステム・モジュールは数多く用意されている。JavaRMI を用いる事により簡単にネットワーク越しに遠隔参照するプログラムを書く事が出来るし、J2EE(Java2Platform Enterprise Edition) ライブラリを用いる事により、多階層に渡る複雑な配置のネットワークプログラムも容易に作成する事が出来る。しかし、これらの技術は一から分散プログラムを開発する際には有意義ではあるが、スタンドアロンな既存プログラムを改造し分散化する際には、修正作業が多くなってしまい、それほど有用ではない。そのため、既存プログラムを分散プログラムに自動分散する為の研究が行われており、既に Addistant、J-Orchestra に代表されるようにいくつかの実用的な自動分散化システムもある。しかし、これらの自動分散化システムも遠隔参照を自動化するという点に特化している為、現実的なアプリケーションの分散化の際には物足りない場合がある。現実的なアプリケーションの場合、分散配置に適したモジュール分割がなされているとは限らないので、プログラムを変更しなくてはユーザの望むような分散配置が出来ない可能性があるからだ。そこで、本研究で提案する Jacross は、既存システムと同様な自動分散化を行いつつ、分散化する際に生じるプログラム変更にも対応出来るようなシステムとして設計した。Jacross はプログラム変換器であり、指定された形式の分散ポリシーを基に既存プログラムを変換する事で分散プログラムを生成する。Jacross では、より現実的なアプリケーションに対応させるために、分散ポリシーの記述方法に Aspect 指向技術を取り入れ、記述力を大幅に強化し、分散配置のより細かい指示を行えるようにしてある。そのため Jacross は、既存システムを利用できない、分散化の際にアプリケーションセマンティクスの変更を伴うようなケースにも対応できる。実際に、本研究では Jacross を利用し、バイオ向けのサーチツールを分散化できることを確認した。

謝辞

本研究を支えていただいた多くの方々への感謝の意をここに表したいと思います。指導教官の、東京工業大学 千葉滋助教授には、私が研究室に所属している3年間にわたって、非常に親身に指導していただきました。本研究の方針や設計、論文の書き方や研究発表に関する指導など実に様々な点で助言をいただきました。

東京工業大学 光来健一助手には、本研究について適切な助言をいただきました。また私と同じく千葉研究室に在籍する佐藤芳樹氏と西沢無我氏には、本研究の題材について深い議論をしていただきました。そして共に学んできた千葉研究室の方々には、色々な意見や学びやすい環境を提供していただきました。これらの方々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

今日、ネットワーク技術が進化する中、分散ソフトウェアに対する期待は高まるばかりである。オブジェクト指向言語、主にJavaでは分散ソフトウェアを開発するためのシステム・モジュールが数多く用意されている。例えばJavaに標準のRMI [12] を用いる事により、簡単にネットワーク越しの遠隔参照を行うプログラムを書く事が出来る。またJ2EE(Java2Platform Enterprise Edition) [14] ライブラリを用いる事により、多階層に渡る複雑な配置のネットワークプログラムも容易に作成する事が出来る。これらの技術は分散ソフトウェアを開発する際には非常に有意義である。

しかし、これらの技術は一から分散プログラムを開発する場合と比較して、既存プログラムを分散化する際にはそれほど有用ではない。ここで言う既存プログラムとは、単一ホスト上で動作する事を想定して書かれたスタンドアロンなプログラムの事である。既存プログラムに分散化を適用する際に上記のような技術を用いようとすると、それらの技術の仕様などに応じてプログラムに手で修正を加えなくてはならない。これらの作業は非常に煩雑で、同じような作業の繰り返しであり、かつ誤りを含みやすい。このような修正作業は保守性・再利用性を維持するためにも自動化されるべき作業である。

このような既存プログラムの自動分散化を行うシステムにAddistant [21, 25] やJ-Orchestra [22] などがある。これらのシステムではプログラムの分散化作業の多くを自動化し、簡素な記述や指示のみでプログラムの分散実行を可能にしている。これらのシステムを用いれば、既存プログラムを容易に分散配置する事が可能である。しかし、現実的なアプリケーションを考えた場合、必ず全てが自動化できるわけではない。例えば、スタンドアロンな設計のアプリケーションでは、分散に適したモジュール分割がされているとは限らない。また、分散化に伴いアプリケーションのロジックの変更を行いたい場合もある。このようなケースは自動化の範疇で行われる処理ではなく、既存の自動分散化システムでは対応していない。

そこで本研究ではそのような現実的なアプリケーションの分散化に対応できるような分散支援システムJacrossの提案を行う。Jacrossは、既存プログラムを分散化する際の煩雑な修正処理は自動で行い、さらに従来の自動分散化システムでは対応しきれない箇所の支援も行うシステムである。

Jacross では Aspect 指向技術を取り入れ、分散配置に関する記述力を既存システムと比較して大幅に強化している。ユーザは、Jacross で用意する Aspect 記述を利用する事により、元プログラムのソースコードを変更せずに、元プログラムを任意の形に変更する事が出来る。Jacross では典型的な分散化の為のリファクタリング作業は自動化し、分散化の際にアプリケーションセマンティクスの変更も必要な場合は、ユーザに Aspect という形で元プログラムとの差分を用意させるアプローチを取っている。この事により従来のシステムでは対応しないアプリケーションセマンティクスの変更を伴う分散化が可能になっている。分散化の際のアプリケーションセマンティクスの変更を可能にした事で、オブジェクトより細かい粒度の分散配置や、セッション処理の追加などの機能拡張が必要な分散化にも対応できる。実際に、現実のバイオ向けサーチツールを Jacross を用いて、必要なプログラム変更を Aspect で用意し、分散化を行った。

以降、２章で関連する研究と現実のアプリケーションを例に取り、その問題点を挙げる。３章では Jacross の提案を行い、４章でその実装について説明する。５章では、実際の Jacross の利用方法とその適用例を紹介する。最後に６章で本論文のまとめを行っている。

第2章　既存Javaアプリケーションの自動分散化

この章では既存 Java アプリケーションの自動分散化作業についての説明を行い、代表的な既存研究の紹介を行う。そして、現実の Java アプリケーションへの適用を念頭に、実際の具体例を交えて既存の研究では困難な点について述べていく。

2.1　自動分散化システムの意義

オブジェクト指向、主に Java の分散ソフトウェアの開発を支援するためのシステム・モジュールは様々なもの [12, 14, 18] が存在する。例えば Java 標準の RMI [12] のようなものを利用すれば、オブジェクトをネットワーク越しにアクセスするようなプログラムを簡単に作ることが出来る。JavaRMI ではオブジェクトのインタフェースを定義し、付属する rmic コンパイラを用いるだけでネットワーク越しの通信に関するプログラムは自動生成される。他の例として J2EE（Java2Platform Enterprise Edition) ライブラリ [14] を挙げることも出来る。J2EE ライブラリ中に含まれる EJB（Enterprise Java Bean） [5, 11] ではあらかじめ決められた仕様に則り、クラスを用意していくことによって、多階層に渡るネットワークプログラムを容易に作成することが可能である。EJB では EJB コンテナがトランザクション処理などのいくつかの機能を提供することで、単に Bean 間の通信プログラムを自動生成するというだけ以上の処理を行うことも出来る。また同じく J2EE ライブラリの中の Servlet [13] や JSP [15] などを用いれば Web サーバを用いたサーバサイドアプリケーションの構築も容易に行うことができる。

しかしながら、これらの分散プログラムを支援するためのシステムは、既存プログラムを改造して遠隔ホスト上でも動作するように変更をする際にはそれほど有用ではない。ここでいう既存プログラムとは、本来単一のホスト上で動作することを意図して開発されたプログラムの事である。先に例に挙げたシステムは確かに有用であるのだが、それらを利用するためにはプログラマは手動でソースコードの修正を行わなくてはならない。

これらのシステムの中には仕様がきめ細かく決められているものもあり、そのような仕様に従うためには大幅にアプリケーションのロジックを変えなくてはならない場合もあるだろう。このような修正は時間がかかるし、誤りを犯しやすい。そもそもソースが手に入らない場合は修正自体がとても困難である。

このような背景から、分散プログラムの開発を支援するアプローチの 1 つとして既存プログラムの自動分散化 [20, 21, 22] を行うことは非常に有用である。既存 Java プログラムの分散実行への適応化を自動で行えれば、以上のようなシステムを利用してプログラムを分散実行用に修正するケースと比較し、プログラムの再利用性・保守性を高く保ち、開発にかかるコストを低く抑えることが出来る。

2.2　Addistant

まずはじめに、代表的な自動分散化ツールとして Addistant [21, 25] について述べる。Addistant は、単一の JVM 上で動作するように開発された既存の Java アプリケーションの機能分散化することを目的とし、開発されたシステムである。Addistant では分散に関する記述を「関心の分離」という視点から、通常の Java ファイルと分離して記述する必要性があるとし、分散アスペクトと呼ばれる別のファイルに記述させている。分散アスペクトでは、各クラスのインスタンスをどの JVM 上に配置するかというような分散化に伴うプログラム変更に関する記述をユーザーが行う。Addistant は、ユーザーによって定義されたこれらの分散アスペクトの指示に従い、指定されたモジュールのみを指定されたホスト上で動作できるような分散アプリケーションにバイトコード変換する。例えば、JavaSwing ライブラリを用いた既存プログラムを、遠隔にある JVM 上で動作させつつ、その GUI オブジェクトを手元にある別の JVM 上で動作させることができる。

2.2.1　自動化された遠隔参照の実装

Addistant ではプロキシ・マスタ方式を用いて、遠隔参照を実装している。この方式では遠隔メソッド呼び出しがローカルメソッド呼び出しと同様、透過的にオブジェクトへのメソッド呼び出しという形で実現される。Addistant は、JVM のクラスロード時にバイトコードを変換することによりプロキシマスタ方式を実装する。変換されたコードを利用するための特別な JVM は必要ない。Addistant ではこれらの一連の遠隔オブジェクト参照に関する実装の詳細をツールの利用者から隠して行っている。

2.2.2　クラス配置の方針

Addistant では対象プログラムの種類に応じて、クラスの遠隔配置の方針を複数用意している。これらの方針の相違点は、どのようにプロキシのクラスを定義するか、どのようにマスタクラスの変更するか、どのように呼び出し側のコード（遠隔オブジェクトにアクセスするコード）を変更するか、という実装の方法である。どの手法をとっても、それだけでは全ての種類のマスタに適用することは出来ない。各手法には適用するマスタが満たさなければならない、手法特有の制約がある。そのため、各制約を意識せずに書かれた既存プログラムに対し、単一の手法を選んでプログラム全体に適用するということは出来ない。例えば、ある手法はマスタクラスの宣言を変更する必要がある。しかし、Java では java.util.Vector のようなシステムクラスの変更を許さないため、もしもシステムクラスのインスタンスが遠隔オブジェクトならば、この手法を用いることは出来ない。この問題を回避するため、Addistant では、クラスごとにこれらの手法のうちの 1 つを利用者が選択できる。利用者がある手法を選択するために注意しなければばならない制約は、次の事項である。

- **参照渡し**
  遠隔メソッド呼び出しのパラメータとして、マスタが参照の形で渡されなければならない場合。複製の形で渡しても問題のない場合には無視してよい。

- **異種性**
  マスタに対して、ローカルと遠隔の参照の両方が同一ホスト上に存在しなければばならない場合。あるクラスの全てのインスタンスが一方のホストに存在する場合は、ローカルと遠隔が共存する必要はないので、無視してよい。

- **非可変バイトコード**
  遠隔参照の実装に必要なバイトコードが変更不可能な場合。例えば、Java では java.util.Vector のようなシステムクラスを変更することを禁じている。あるマスタクラスについて対象プログラムのバイトコード中のどの部分がこの非可変なバイトコードにあたるかにより、この制約はさらに 3 つに細分化される。
  1. マスタクラス自身のクラス宣言（クラス宣言）
  2. マスタクラス型が現れる他のクラス（参照者クラス）
  3. マスタクラスのインスタンスを生成している参照者クラス（生成者クラス）

Addistant では、以上の制約に応じて利用者が適切な手法を用いて遠隔配置を可能にする為、以下の４つの手法を用意している。

「置き換え」手法
この手法は、異種性の必要がなく、かつクラス宣言のバイトコードが可変である場合に用いる。対象クラスが遠隔参照される場合、対象クラスの中身がそのままプロキシクラスに差し替えられる。

「名前変更」手法
この手法はクラス宣言のバイトコードが非可変である場合にも適用できる。ただし参照者クラスまたは生成者クラスが非可変な場合や異種性が必要な場合は適用できない。この手法では対象クラスに対し、別の名前でプロキシクラスを生成する。そして、Addistant はその別名で定義されたプロキシクラスを遠隔参照として用いる。

「サブクラス」手法
この手法は異種性がある場合にも用いる事が出来る。この手法では、対象クラスのサブクラスとしてプロキシクラスが生成される。Addistant は、ローカルの参照には対象クラスへの参照を用い、リモートの参照にはサブクラス化されたプロキシのインスタンスを用いる。

「複製」手法
この手法は、int のようなプリミティブ型で常に用いられる。また、メソッド呼び出しによって状態の変化しない、java.lang.String などのクラスに用いることができる。この手法では、ネットワーク越しに、オブジェクト単位で浅い複製が渡される。

表 2.1 は各手法の適用可能性についてまとめたものである。

表 2.1: 各手法の適用可能性

適用上の制約	置き換え	名前変更	サブクラス	複製
参照渡し				×
異種性	×	×		
非可変クラス宣言	×		（×）	（×）
非可変参照者クラス		×		
非可変生成者クラス		×	×	

×は、左の制約に対し選択された手法が適用できないことを示す。（×）は、適用できない場合があることを示す。

```
<policy>
  <import proxy="rename" from="display">
    subclass@java.awt.-
    subclass@javax.swing.-
    subclass@javax.accessibility.*
    subclass@java.util.EventObject </import>
  <import proxy="rename" from="application">
    exactsubclass@javax.swing.filechooser.*
    exactsubclass@java.io.[InputStream|
    OutputStream|Reader|Writer] </import>
  <import proxy="subclass">
    subclass@java.util.[Dictionary
    |AbstractCollection|AbstractMap|BitSet]
  </import>
  <import proxy="writeBackCopy">
    array@- </import>
  <import proxy="replace" from="application">
    user@- </import>
  <import proxy="copy">
    - </import>
</policy>
```

図 2.1: 分散アスペクトの記述例

2.2.3　XML による簡素な分散ポリシー

Addistant では分散に関する記述を通常の Java プログラムとは分離して記述する必要性を強く述べている。分散に関する記述が通常の Java プログラムと混在すると、プログラム自体の可読性が低下し変更も困難なものになる。そこで、Addistant では分散に関わる処理がプログラム全体に拡散して入り混じることを避けるため、分散に関係のないロジックを記述した非分散プログラムとは別に、分散に関わる記述をまとめて記述させるスタンスを取っている。このまとめて別に書かれた分散に関わる記述を分散アスペクトと呼び、これらは簡素な XML 形式のファイルとして定義される。この XML 形式の分散アスペクトでは、別ホスト上で動作させたいクラスや前述したクラス配置の方針などの指定などの必要最低限の情報だけをユーザ側で記述する。

図 2.1 は Swing を用いた GUI を分散させる分散アスペクトの記述例である。GUI オブジェクトのためのクラス、すなわち java.awt、javax.swing 等のパッケージのクラスとそのサブクラスについて、マスタが display で示されるホスト側に配置されるようにしている。逆に、java.io.InputStream 等の、それ自身を除いたサブクラスについては、マスタが application 側に配置されるようにしている。これらのクラスには「サブクラス」手法を適用で

きないため、「名前変更」手法を適用している。java.util.AbstractCollection 等のサブクラス（java.util.Vector など）については、「サブクラス」手法を適用し、どちらのホストでもマスタとプロキシが混在できるようにしている。配列については、一律「書き戻し複製」手法を適用している。ユーザクラスについては、マスタが application 側に配置されるようにして、「置き換え」手法を適用している。残りの、java.util.Locale などのシステムクラスについては、「複製」手法を適用している。

2.3　J-Orchestra

次に J-Orchestra [22] について説明する。J-Orchestra も、Addistant 同様に既存プログラムを分散適用することを目的とした自動分散化ツールである。やはり、J-Orchestra でも遠隔参照の実現にプロキシ・マスタ方式を用い遠隔アクセスされるオブジェクトはプロキシ経由で間接的に参照される。J-Orchestra ではより自動化を推し進めるために Profiler や Static analysis などといった機能を用意している。これらの機能を用いることにより、Addistant ではユーザが記述しなくてはならなかった手法の選択さえもユーザから隠蔽した形で自動化している。また J-Orchestra では、実装段階でクラスの種類を更に細かく場合分けを行っている。コード変換は Addistant 同様、バイトコード変換により実現されている。

2.3.1　J-Orchestra の構成要素とその役割

- **Profiler**
  アプリケーションのクラス間の依存関係をユーザ側に知らせるための要素。Profiler から得られる情報は最適なパフォーマンスを得られる分割をユーザが思考する際の手助けになる。

- **Classfier**
  クラスの分類を行うための要素。J-Orchestra 側で対象クラスが書き換え可能であるかなどのいくつかの判定を行い、クラスの分類を行う。各クラスは J-Orchestra が定める数タイプに分類され、後述する Translator でタイプに応じた変換処理が行われる。

- **Translator**
  コードの変換を行い、分散実行を適用させる要素。Classfier で分類されたタイプに応じて、いくつかの変換パターンを用意する。Translator では、それらのパターンの中から分散実行を適用させるために

各クラスに最適なものを利用し、プロキシの実装を行う。なお、変換はバイトコード変換によって実現されている。

2.3.2　クラスの分類

前述したように J-Orchestar の Classfier では変換の対象となるクラスをいくつかのタイプに分類する。J-Orchestra では、変換対象となるコードが可変・非可変であるかを主眼に分類するだけでは不完全とし別の角度からの分類を行っている。

- **anchored unmodifiable class**
  anchored unmodifiable class とは native メソッドを持つようなクラス、あるいはそのクラスのオブジェクトの参照がアプリケーションクラスと他の anchored unmodifiable class の間で渡されているようなクラスである。ただし、J-Orchestra で想定しているプログラムは Pure Java で書かれたもので、native メソッドを含むクラスとは Java のシステムクラスに限られる。つまり、ユーザが独自に Native メソッドを利用しているような場合は考慮されていない。

- **anchored unmodifiable class**
  anchored unmodifiable class を親として持つようなクラス。

- **mobile class**
  上記以外のクラスを総称して mobile class と呼ぶ。システムクラスでも native メソッドを持たずに anchored unmodifiable class のオブジェクトを参照していない場合は mobile class となる。例えば、java.lang.ThreadGroup は java.lang.Thread クラスを参照するので anchored であるが、java.util.Vector クラスは mobile class となる。

2.3.3　変換方法

J-Orchestra では、分類された変換対象クラスをタイプ毎に異なった手法で変換を行う。以下は遠隔配置対象のクラスのタイプに応じた変換手法の違いを説明したものである。

anchored unmodifiable class
anchored unmodifiable class は J-Orchestra によって、直接変更を加える事が出来ない。そこで J-Orchestra では１つの anchored unmodifiable class に対し２つのサポートクラスを生成することで遠隔参照を行う。１つは anchored unmodifiable class を基として、元ク

ラスと同じ機能を持つプロキシクラスである。もう 1 つのクラスは application system translator と呼ばれるクラスであり、遠隔参照と実行を可能にする。例として、java.lang.Thread クラスを遠隔化する場合の説明を行う。この時、プロキシは anchored.java.lang.Thread という名前でメソッド等が java.lang.Thread と同じものが用意された状態で生成される。

```
//Original Code: Client
java.lang.Thread t = new java.lang.Thread(..);
void f(java.lang.Thread t){t.start();}

//Modified Code
anchored.java.lang.Thread t =
  new anchored.java.lang.Thread(..);
void f(anchored.java.lang.Thread t){t.start();}
```

図 2.2: プロキシへの参照の変更

図 2.2 のようにクライアント側に現れるの全ての java.lang.Thread への参照は anchored.java.lang.Thread というプロキシクラスへの参照に変更される。

一方、プロキシ内のメソッドの実装は図 2.3 のようになっている。

```
public final void setPriority(int arg0) {
  try { _remoteRef.setPriority(arg0); }
  catch (RemoteException e) {e.printStackTrace();}
}
```

図 2.3: setPriority

図 2.3 は anchored.java.lang.Thread クラスの setPriority メソッドの実装である。ここで、_remoteRef は application system translator への参照である。このようにして proxy 内部で擬似的に anchored unmodifiable class の遠隔実行を行い、anchored unmodifiable class の遠隔参照を行っている。
なお、application system translator を利用し遠隔実行を行う場合、

実際のコードが呼び出されるホストでは wrapping と unwrapping の処理が必要となる。

anchored modifiable class

anchored modifiable class は基本的には anchored unmodifiable class と同様の変換を行うことによって遠隔実行を行うこと出来る。ただし、このタイプのクラスは、他のアプリケーションクラスのフィールドに直接アクセスしてしまっているケースが考えられる。そこでこのような場合、J-Orchestra では直接フィールドを参照するのではなく getter/setter といったアクセサメソッド経由で参照するように変更を施している。

mobile class

mobile class も上記の 2 つのタイプと似たような変更を行うが、大きく 2 つの点で異なる。まず 1 つは生成されるプロキシクラスが元のクラスと同じ名前で生成されるということである。この事により、元のクラスと同じようにプロキシクラスを参照することが出来る。もう 1 つは遠隔実行のために、専用のインタフェースを備えるということである。さらに、変更される元のクラスは < クラス名 >_remote と改名され、祖先のクラスが java.lang.Object から java.rmi.server.UnicastRemoteObject へと変更される。mobile class は遠隔メソッド呼び出しの引数として渡された場合、オブジェクトの遷移が可能である。

Java Language Feature

また、J-Orchestra では変換する際に、Java に特有の問題を解決するための手法を幾つか提供している。

Static に対する扱い

遠隔配置されるクラスの静的なフィールドにアクセスしたり、静的なメソッドを利用する場合、J-Orchestra では StaticDelegator という委譲クラスの作成を行う。StaticDelegator クラスは、祖先に java.rmi.server.UnicastRemoteObject を持ち、対象クラスの static メソッド呼び出しを代行して行う。

```
class A {
  static void foo(String s){...};
}
```

```
class A_Delegator extends java.rmi.server.UnicastRemoteObject {
  void foo(String s) { A_remote.foo(s);}
}
```

static メソッドが利用されたりフィールドにアクセスしない限り StaticDelegator クラスは作成されない。

継承関係
生成されるプロキシクラスは元クラスのクラス階層と同じように生成される。

オブジェクト生成
通常のオブジェクト生成処理を行わせないため、何の処理も行わない特別なコンストラクタの呼び出しを行う。そして、ホスト毎に ObjectFactory を用意し、実オブジェクトの生成を行う。

this
this を他のオブジェクトに渡すような処理の場合、オブジェクトの実体を渡してしまう。そこで、J-Orchestra ではバイトコードレベルで this の参照をプロキシへの参照と変更することによって、この矛盾点を回避している。

標準入出力
System.in や System.out などの標準入出力は RemoteIn や RemoteOut といった代替クラスに置き換えられる。この代替クラスを用いることにより標準入出力を擬似的にブロードキャストすることが出来る。java.lang.System クラスの他の機能についても代替クラスが用意されている。

2.3.4　GUI と XML による分散配置の決定

J-Orchestra では、専用の GUI と指定された形式の XML ファイルを用いて分散配置の方針を決定する。
具体的には、図 2.4 のような GUI を用いる。上記の GUI では対象プログラムを入力とし static analyzer でクラスの分類を行い、その結果を表示する。ユーザはその結果から、最適な配置を考慮し分散配置を行う。分散配置はドラッグアンドドロップで視覚的に行う事が出来る。またユーザが行った分散配置の過程は XML 形式で出力する事も出来、指定された形式の XML をインポートすることで過程を再現することも分割を指示することも可能となっている。

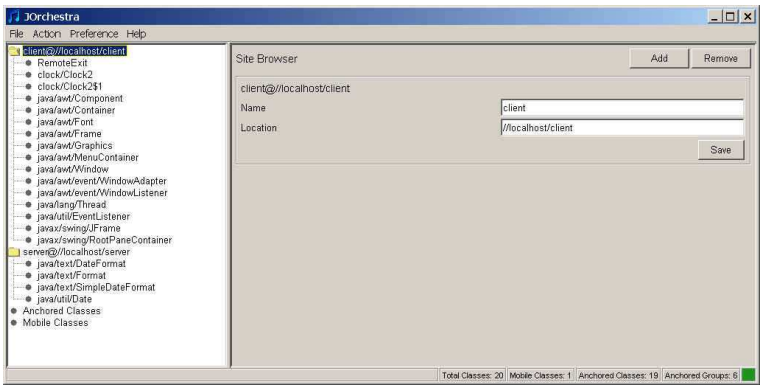

図 2.4: J-Orchestra の GUI

2.4　現実的な既存 Java アプリケーションの分散化作業

それでは現実的なアプリケーションを例にとって、自動分散化を考察してみたいと思う。現実的なアプリケーション例として、Java 言語で記述された中規模サイズのバイオ向けツール ARMSoftware [9, 10] を取りあげる。アプリケーション例として ARMSoftware を取りあげた背景には実際に ARMSoftware を分散実行したいという要求があったことを理由として記しておく。尚、本研究では問題を考察するために、本ソフトウェアの分散化をあらかじめ手動で行っている。

2.4.1　ARMSoftware

ARMSoftware は、有機化合物の経路探索を行う Java ソフトウェアである。利用者は Web サイト上からソフトウェアのダウンロードを行い、利用者のマシン上で実行する。ARMSoftware では基本的に以下の 3 つの機能を提供する。

1. **compound**
   物質名から構造を検索し構造式を表示するための機能。有機化合物では基本が同じ物質でも構造が異なる物質が数多く存在する。例えば glucose と検索すると、微妙に構造の違う数多くの glucose が検索結果として表示される。
2. **enzyme**
   有機化合物の変化を検索・表示するための機能。有機化合物は触媒等を加えることにより、複雑に別の有機化合物へと変化する。enzyme ではその変化の過程を検索し、結果を化学式と構造式により表示する。

3. **pathway**
   有機化合物の経路（変遷）を辿るための機能。有機化合物は生成されるまでに、唯一の経路で生成されるわけではない。全く同じ物質でも、生成されるまでには違う経路を辿ることがある。例えば pyruvic acid という物質から D-glucose という物質を生成する経路を検索すると、途中で加える触媒などによって 11 通りの経路が存在する。更に初期物質が違うと複数の経路が存在する。そのような物質から経路を検索する、いわば逆引きするための機能が pathway である。

ARMSoftware では、さらにこれらの検索結果を利用した描画機能も用意されている。なお、データベースはあらかじめ用意されたものがあるが、追加したり、あるいはオリジナルな形でユーザが用意する事も出来る。

2.4.2　要求

ARMSoftware を分散実行させる際の要求がどのようなものだったかを説明する。ARMSoftware は最初から遠隔配置をどのように行うかが製作者側から提示されていた。それは単にパフォーマンス的な問題の他に、運用的な問題も考慮した配置方針であった。具体的には、先に説明した 3 機能の検索機能のみをサーバ側に配置するという設計である。その他の描画機能や表示機能は全てクライアントサイド（エンドユーザ側）にまとめておく。ARMSoftware のこれらの 3 機能はデータベースにアクセスする必要がある為、ソフトウェアを利用する際は、データベースのダウンロードも行わなくてはならなかった。つまり利用者は、データベースの更新が行われる度にダウンロードを行わなくてはならない。このような設計はデータベースの更新が頻繁に行われる環境では望ましくない。そこで、データベースのダウンロードを避ける為にこのような配置が要求された。またさらに、一部の検索機能はローカルのデータベースを用いる際には敢えてサーバ側の検索機能を用いずに、ローカルでも検索が行う事が出来る設計を求められた。これは有機化合物のデータベースが、オリジナルなものであるとそれ自体に商用的な価値が発生する場合があるためである。またツールの意義を満たすために、複数のクライアントが同時にサーバにアクセスするケースも想定されなくてはならなかった。

2.5　既存技術による自動分散化の限界とその実例

この節では実際に ARMSoftware を分散化することを通して、自動分散化を行う為に従来の研究では困難な点を実例を交えて紹介する。まず、従来の自動分散化ツールが適用プログラムに対し、暗黙裡に要求する 2 つの

制約について説明する。

分散化に適したモジュール分割
: モジュール分割がクラス単位、あるいは少なくともオブジェクト単位で行われる事。

分散化はプログラムの構造を変えない
: 既存研究の目指す分散化はあくまでクラスやオブジェクトの遠隔配置を行うだけで、実行されるホストが変わるという一事を除けばプログラムの構造は変化しない。

これら 2 つの制約は自動分散化ツールの性質上、仕方のない制約と言える。自動分散化ツールはあくまで、分散実行を行えるように変換するツールとであり Java がオブジェクト指向言語である限り、最小のモジュール分割はオブジェクト単位である。また分散実行を行えるようにプログラム変換する以外にプログラムの構造を変える事は自動分散化ツールの範疇ではない。しかし、現実的にはアプリケーションを分散実行させる為には、そのアプリケーションを分散実行させても同じ性能あるいは、ツールとして同じ意義を満たして動作させる為に以上の 2 つの点を無視して分散しなくてはならない場合がある。

2.5.1　アプリケーションセマンティクスの変更を伴う分散化処理

分散実行したいプログラムは必ずしも最初から分散化を意識して書かれているわけではない。そのため、利用者が思い描くような分散化を行う際に、多少のアプリケーションセマンティクスの変更が必要となるケースがある。先に示した 2 点の制約が満たされているプログラムは言い換えれば、分散化する際にアプリケーションセマンティクスの変更が必要ないプログラム、と言う事が出来るだろう。それでは、具体的に先の 2 つの制約について抵触しなくてはならない場合、つまりアプリケーションセマンティクスの変更を伴う分散化処理を行わなくてはならない場合について例を交えて説明を行う。

メソッド単位での分散化処理

スタンドアロンな設計で作られたプログラムは、常に分散実行に適したモジュール分割をされているとは限らない。例えば、クラス Sample にあるメソッド foo が用意されているとする。foo はデータベースを利用するメソッドと仮定する。Sample に用意されているその他のメソッドはデータベースにアクセスしない。このような条件下で分散実行を行う際に、foo

はデータベースを扱うメソッドなのでサーバー側で動作させたいが、その他のメソッド呼び出しは全てクライアント側で行いたい、という要求が生じる可能性がある。この場合、オブジェクト単位で分割することは不可能で、メソッド単位での分散実行を適用しなくてはならないだろう。しかし、Java はオブジェクト指向言語なので、通常はメソッド単位では自動分散化を行うことが出来ない。そこで例えば、ローカルとリモートで同じ状態を保つようなオブジェクトを用意して擬似的にメソッドのみの遠隔実行を行わなければならない。このようなケースではリモートとローカルのオブジェクト同士で整合性を取るような何らかの処理を追加する必要があり、アプリケーションセマンティクスを変更する事になる。

マルチクライアント

遠隔配置したオブジェクト、クラスを複数のクライアントで利用したい場合があるとする。クラス Sample を遠隔配置し、複数のクライアントから Sample に対するアクセスがあるとする。仮にこのクラス Sample に静的なフィールド context が存在するとする。Sample で行われる全ての処理は、このフィールドに格納されるような設計になっているとする。スタンドアロンな環境で用いられる事を想定されたプログラムでは矛盾なく動作するかもしれないこのクラスが、分散実行を適用しようとすると、複数クライアントでこの Sample を利用しようとするため、問題が生じてしまう。この場合は、context は静的なので、クライアント A が Sample を利用した結果を誤って、クライアント B が利用してしまうというケースが容易に起こりうる。そこで例えば、このフィールド context に対しセッション情報を持たせる為に、何らかの処理を付け加えなくてはならない。このようなセッション情報を付加するような処理は、アプリケーションの構造そのものを変化させなくてはならないだろう。

2.5.2　現実的なアプリケーションによる具体例

我々は ARMSoftware を分散化する作業を通じて、実際にこのようなアプリケーションセマンティクスを伴う分散化処理に遭遇した。この節では、実際に手動で行った分散化処理と合わせて、その紹介を行う。

メソッド単位での分割

図 2.5 の説明を行う。クラス MolDoc の searchDoc メソッドは compound の検索メソッドであり、これは遠隔配置して実行したいメソッドである。

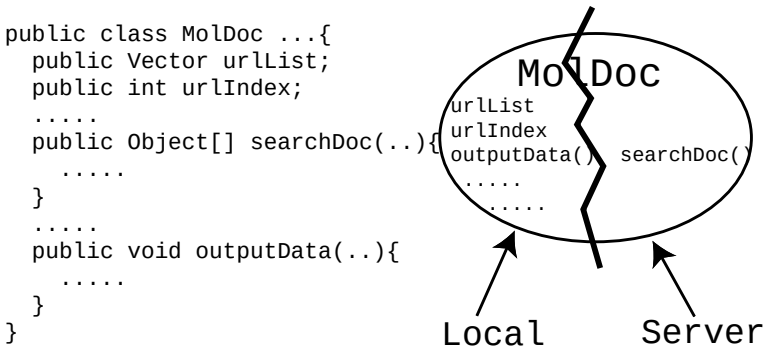

図 2.5: メソッド単位の分割

しかし、このメソッド以外のメソッドは出力用のメソッド outputData などに見られるように、全てローカル側で利用したいメソッドである。そこで我々は searchDoc メソッドのみを遠隔で動作させ searchDoc メソッドを呼び出す直前と直後でリモートとローカルのオブジェクトの整合性を取るような処理を加えた。具体的には以下の通りである。searchDoc を行う際に必要な urlList、urlIndex の二つのフィールドをメソッド呼び出しが起こる前にリモートで用意された urlList、urlIndex にセットしておく （図

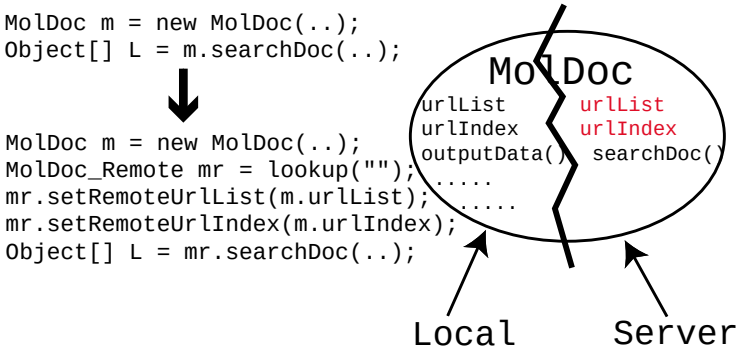

図 2.6: メソッド単位の分割例

2.6)。

セッション処理が必要な分割

図 2.7 はセッション処理が必要な分割の具体例である。クラス AbstractDoc の遠隔配置を行いたいとする。クラス AbstractDoc のフィールドには urlIndex,urlList などが static で定義されている。クライアントが 1 つであ

```
public class AbstractDoc {
  public static Vector urlList;
  public static int urlIndex;
  .....
  .....
}
```

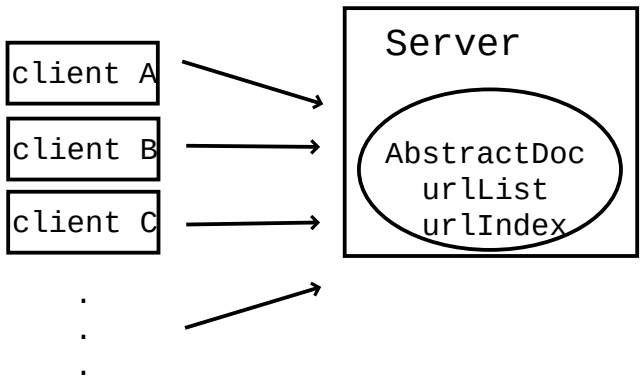

図 2.7: セッション処理が必要な分散配置

れば、J-Orchestra のように static フィールド用の委譲クラスでも作れば問題がない。しかし、アクセスしてくるクライアントが 1 つではない場合、この実装だと問題が生ずる。そこで AbstractDoc を遠隔配置する為に、これらのフィールドをクライアント毎に用意する為のリストを用意した。さらにクライアントを識別する為の ID を割り当てる為のメソッドも用意し、その ID を引数としてこれらのフィールドにアクセスするように変更した（図 2.8）。ARMSoftware には ArmView という起動クラスが用意されている。そこで、クラス ArmView に static なフィールドを持たせ、それをこのクライアントの ID とした。起動時に遠隔の AbstractDoc から getID によって ID を取得し、それ以降の static フィールドへのアクセスは全てこの ID を引き数として行う。また、リモートメソッド呼び出し時にサーバ側でこれらの static フィールドにアクセスしなくてはならない場合は、対象メソッド自体に変更を加えた。メソッドの引数に ID を追加してサーバ側にクライアントの ID を渡し、その ID を引数として各々の urlList、urlIndex を得るように変更を施した（図 2.9）。図 2.9 では、enzMap の mapping というメソッドに変更を加えている。enzMap の mapping メソッドは遠隔呼び出しされるメソッドで、内部で AbstractDoc の static フィールドにアクセスをする。そこでセッション処理を行う為に、mapping の定義を変更して int 型の引数を付け加えた。Client は各々に割り振られた

Server側

```
public class AbstractDoc ... {
  public static ArrayList urlListList;
  public static ArrayList urlIndexList;
  public static Vector urlList;
  public static int urlIndex;
  public static ID;
  .....
  public Vector getUrlList(int id) ...{
    return urlListList.get(id);
  }
  public void setUrlList(int id) ...{
    urlListList.set(urlList,id);
  }
  .....
  public int getID() ...{
    int id = ID;
    ID++;
    return id;
  }
}
```

```
Vector v = AbstractDoc.urlList;
```

Client側

```
(AbstractDoc_Remote) abs = ...;
Vector v =
   abs.getUrlList(ArmView.id);
```

図 2.8: セッション処理の 1 つ

ID を送り、サーバ側ではその ID に応じた mapping 呼び出しが行われる事になる。

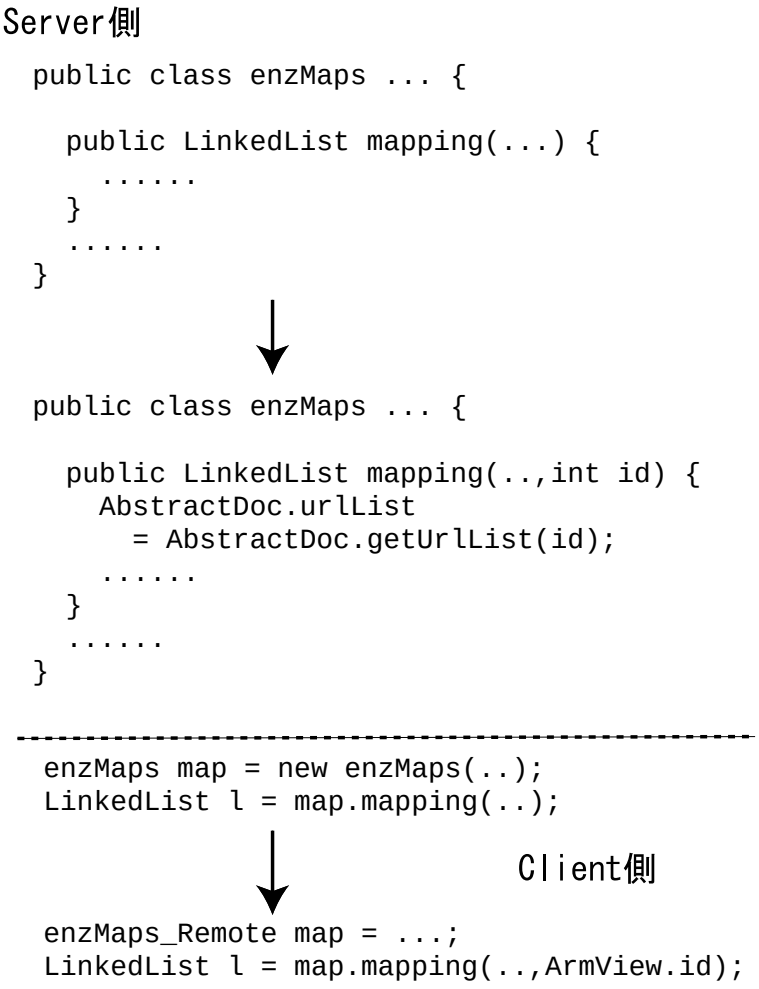

図 2.9: セッション処理の 1 つ

第3章　Jacross

前章での問題点を解決するために、本章では既存 Java プログラム向け分散支援システム Jacross の提案を行う。Jacross はプログラム変換器であり、XML で記述された分散ポリシーと元プログラムとは分離記述された分散処理用 Aspect を基として既存プログラムをプログラム変換する事で、分散プログラムを生成する。Jacross では自動分散化システムの利点を保ちつつ、現実的なアプリケーションにも対応できるような設計を目指している。それは自動分散できる箇所は全て自動化し、それ以外の自動化する事の出来ないアプリケーションに固有な箇所の支援も同時に行えるシステムである。そこで我々は、既存システムと比較して分散配置の指針を指示するための分散に関する記述を大幅に強化するアプローチを取った。記述をより詳細にする事で、ユーザが分散配置のために用意しなくてはならないプログラム（後述する分散処理用 Aspect、分散ポリシー）は増加するが、一方で色々なタイプの既存プログラムに応用する事が可能になる。ちなみに、分散に関する詳細な記述は分散配置に適していないモジュール化が行われているプログラムなどにのみ必要なもので、従来システムで分離可能なプログラムは Jacross でも同様のコストで分散化する事が可能になっている。

3.1　記述力を強化した分散化支援システム

Addistant や J-Orchestra に見られるように自動分散化システムは分散配置の指針を決定するために、XML 形式のポリシーファイルを用いる事が出来る。これらのシステムに見られる XML 記述は宣言的である。あらかじめ用意された機能を、どのクラスにどのように適用させるかということを記述するに留まっており、その分簡潔なものとなっている。しかし、用意された機能を選択的に用いるだけでは前章で挙げた問題点を解決するには物足りない。アプリケーションセマンティクスの変更を伴うような分散化処理を扱う場合、その分散化処理の内容はアプリケーションに依存する部分が出てきてしまう。Addistant や J-Orchestra では、クラスをタイプ分けすることにより遠隔配置を可能にしていたがアプリケーションセマンティクスの変更を行う場合、画一的なタイプ分けをすることは難しい。

例えば、あるプログラムのために用意したセッション情報を処理するためのプログラム変更が、他のプログラムにとっては必ずしも最適であるとは限らないからだ。そこで我々はこれらの点を解決するために Aspect 指向技術を応用することにした。

3.2　Aspect 指向技術の利用

3.2.1　Aspect 指向とは

Object 指向技術は様々な機能をカプセル化する能力に優れており、保守性、拡張性、可読性に優れた技術である。しかし Object 指向ではある機能のための処理群が複数オブジェクトに散らばってしまうことがある。例えば、ロギング・同期・永続性などの処理は各オブジェクト間に散らばってしまいがちで、上手くモジュール化する事ができない。これらの機能が上手くモジュール化出来ていないため、プログラムの修正を行う際に多くの個所を修正する必要が出てしまい、簡潔性を失った質の低いプログラムになってしまう場合がある。このように、プログラム間にまたがって散らばってしまった処理郡は横断的関心事（crosscutting-concern）と呼ばれ、上手くモジュール化できない事が問題視されている。Aspect 指向技術は、このような横断的関心事をオブジェクトとは別の側面から考慮し、それを Aspect として分離・モジュール化することを目的とした技術 [8] である。Aspect 指向技術を利用した言語 [1, 3, 4, 6, 7] では、元のプログラムから関心事を指定し、独立した 2 つのモジュール（クラスと Aspect）を合成し 2 つの機能を持ったプログラムを作りだす手法を取る。この事からも分かるように、Aspect 指向技術はあくまで既存のプログラム技術を補うための技術であり、Object 指向技術にとって変わるものでない。

3.2.2　joinpoint と pointcut

Aspect 指向技術を説明するために、joinpoint と pointcut について説明する。joinpoint とは、プログラムの動作の原始的な構成要素の事で、例えばメソッド実行、フィールド参照、インスタンス生成などがある。。Aspet 指向言語ではこの joinpoint にコードを合成する。joinpoint の種類は言語によって異なっている。 1 つ以上の joinpoint を組み合わせてプログラム中にコードを合成する箇所を具体的に指定する事を pointcut という。プログラマは数ある joinpoint の内から、うまくモジュール化出来るように pointcut を関心事に合わせて定義する。そして定義された pointcut に対し、挿入するコードを用意することで Aspect という形でモジュール

化する。

3.2.3　Aspect 技術の実現手法

Aspect 指向言語は、モジュール化されたアスペクトを通常の Java ファイルで用意するもの [1, 4, 6] と独自のフォーマットで用意するもの [7] とに分ける事が出来る。例えば、JBossAOP ではインタセプタというコンポーネントを用い、Pure Java で Aspect を表現する事で、Aspect 指向技術を体現している。後者の例としては AspectJ を挙げる事が出来る。AspectJ では、ajc 形式の特殊なフォーマットのファイルを用意する事で Aspect を利用する事が出来る。図 3.1 は、Pure Java を利用した Aspect の利用イメージである。sampleMethodA と sampleMethodC のメソッド呼び出しを pointcut し、メソッドが呼ばれる前と後でアスペクトとして用意されたコードに制御が移り、それぞれ methodA と methodB が実行される。我々

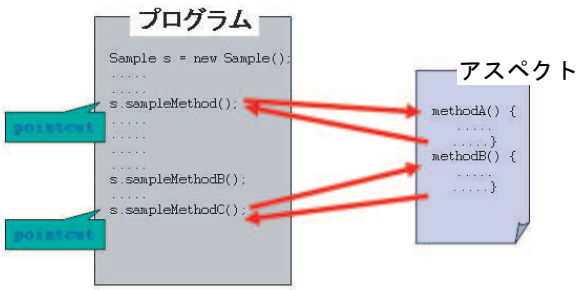

図 3.1: Aspect の利用イメージ

は独自のフォーマットを用いずに、Pure Java を利用するアプローチで分散処理用の Aspect の定義を行う事にした。

3.2.4　Jacross の Aspect イメージ

Jacross では、この Aspect 指向技術を、アプリケーションセマンティクスの変更を伴う分散化処理を支援するために応用する事にした。つまり Jacross の処理系では、図 3.2 のように元プログラム・分散ポリシー・分散処理用 Aspect の 3 つにモジュール化される事になる。前章で紹介したよ

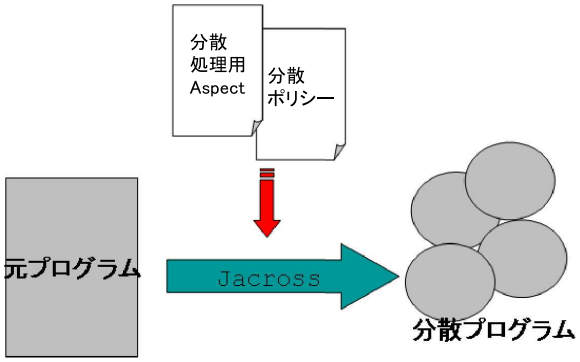

図 3.2: Jacross

うなメソッド単位での分散化処理や、セッション処理を含んだ分散化処理は、アプリケーションに依存する。このようなプログラム変更は、自動生成する事が難しい。そこで、このプログラム変更は Aspect という形で分散ポリシー・元プログラムとは更に別の形でユーザに記述させる。Jacross でアプリケーションセマンティクスの変更を伴う分散化を行う場合、ユーザに次に説明する 2 つのモジュールを用意させる。そして、分散ポリシー上でどのように元プログラムと合成するかを記述させると Jacross はそれを基に分散プログラムの生成を行う。具体的にどのような形で分散処理用 Aspect を用意するかを説明する。

Inter-type Declaraiton

Inter-type Declaration は、既存プログラムに新しくメソッドやフィールドを追加する為の機能である。特別なフォーマットに従う必要はなく、Jacross は Pure Java で書かれたクラスファイルの必要なメソッドやフィー

ルドを選択された既存プログラムに挿入する。

JacrossInterceptor

Jacross ではインタセプタを用い、Aspect 化されたモジュールの呼び出しを行う。Inter-type Declaration を用いれば、新規にメソッドやフィールドを追加する事が出来る。しかし、これだけではアプリケーションセマンティクスの変更を行うには、まだ不十分である。Inter-type Declaration は、あくまでメソッドやフィールドの追加を行う為の機能なので、アプリケーションの制御フローを変える事は出来ないからだ。アプリケーションセマンティクスの変更を行うには、新規に追加されたメソッドやフィールドあるいは、既存のメソッドやフィールドを利用する為のコードを割り込んで処理させるように埋め込まなくてはならない。そこで Jacross が提供しているのが、JacrossInterceptor である。JacrossInterceptor を用いれば、プログラム中のあらゆる pointcut で好きな呼び出しを行う事が出来る。インタセプタ用の Java ファイルは、Jacross の仕様に従わなくてはならない部分があるが、Inter-type Declaration と同様に Pure Java で自由に定義する事が出来る。

3.3　より詳細な分散配置の指定

現実的なアプリケーションの分散化作業を支援するには、より粒度の細かい分散配置の指定も必要になる。それは現実的なアプリケーションが必ずしも分散に適したモジュール分割をされているとは限らないからである。前章にも挙げたように、分散配置されたプログラムをオブジェクト単位より細かく分離したいような場合、ユーザに細かい指定をさせずに自動化することは難しい。そもそも我々が対象としているプログラムは、スタンドアロンな環境を想定して設計されたプログラムである。スタンドアロンな環境と分散実行を行う環境では様々に異なる制約があり、片方の環境に適した設計が、もう片方でも適しているという保障はない。そのため、モジュール化が分散実行に適していない場合、アプリケーションに依存したより細かい分散配置の指定を行わなくてはならない。

Jacross で用意するプログラムの分散配置の方針

Jacross では、Addistant と類似した方針で分散プログラムを配置する設計を取っている。Jacross では遠隔実行されるクラスは全てプロキシを

通して実行されるので、遠隔実行したいクラス毎にプロキシの設計方針を定めなければならない。以下が設計方針の詳細である。

Replace
Addistant の「置き換え」手法に準ずる。この方針では、対象クラスを遠隔実行が可能な形に変更されたプロキシクラスに置き換える。

Rename
Addistant の「名前変更」手法に準ずる。この方針では、対象クラスと別の名前でプロキシを定義し、参照側をプロキシクラスへの参照へと書き換える。

Subclass
Addistant の「サブクラス」手法に準ずる。この方針では、対象クラスのサブクラスの形でプロキシが定義される。この方針は１つのホスト上で対象クラスがローカルと遠隔の２つの参照を持ちたい場合に適用する。

Jacross では、これらの手法に加え**permethod** というオプションを利用する事が出来る。各方針に、このオプションを利用するとメソッド単位での遠隔配置が可能になる。この方針を用いると対象オブジェクトはローカルホスト上と遠隔ホスト上の２箇所で生成され、１組のペアとして扱われる。ローカルでのメソッド呼び出しが行われる場合は、ローカルで生成したオブジェクトを用いてメソッド呼び出しを行う。また、遠隔でのメソッド呼び出しを行う場合は、リモートで生成されたオブジェクトを用いてメソッド呼び出しを行う。このオプションではローカルと遠隔でのオブジェクトの状態の変更の自動反映まではサポートしていない。このようなオブジェクトの同期化などはプログラムに依存するものであり、自動化するのに適していないからだ。そのため、状態の変更の反映が必要な場合などは、分散処理用 Aspect を用いてユーザが記述する必要がある。

参照者クラスと対象（被参照者）クラスの関係による分類

これらの配置方針やオプションを用いるために、参照者クラスと対象クラス（被参照者クラス）の関係による場合分けを行う。

タイプ１：可変クラス→可変クラス
参照者クラス、対象クラスの両方が可変である場合をタイプ１とする。

タイプ２：非可変クラス→可変クラス

参照者クラスが非可変で、対象クラスが可変である場合をタイプ 2 とする。

タイプ 3 ：可変クラス→非可変クラス

参照者クラスが可変で、対象クラスが非可変である場合をタイプ 3 とする。

参照者クラス・被参照クラスが共に非可変の場合は対象としない。
表 3.1 にタイプと配置方針の対応関係をまとめた。○はその方針で配置可能な事を示す。△はその方針だと限定された状況で配置できない可能性がある事を示す。×はその方針では遠隔配置出来ない事を表す。Jacross で

表 3.1: タイプと配置方針の対応関係

-	Replace(permethod)	Rename(permethod)	Subclass(permethod)
タイプ 1	○ (○)	○ (○)	○ (○)
タイプ 2	○ (△)	× (×)	△ (△)
タイプ 3	× (×)	○ (△)	× (×)

は以上の表を参考にしてユーザ側が選択した配置方針を元に遠隔参照の実装を行う。

第4章　実装

この章では Jacross の実装についての説明を行う。Jacross は Addistant の文法や機能を参考に類似した設計となっており、さらに前節で説明した Jacross 独自の機能を追加している。大まかに分けると以下の 3 つの機能を実装している。

遠隔参照の実装の自動化

アプリケーションセマンティクスの変更に対応する分散 Aspect

XML による分散配置方針の決定

以下では、具体的にこれらの機能をどのように実装したかについて紹介する。

4.1　クラスのタイプに応じた配置方針の決定

Jacross では、遠隔参照の実装は自動で行われる。ユーザには遠隔参照の実装の詳細は隠蔽され、遠隔参照の実装の仕様に応じてユーザ側がプログラムを変更する必要はない。なお、遠隔参照の設計には、Remote Proxy [19] パターンとしても知られるプロキシ・マスタ方式を用いて実装した。Jacross では遠隔配置されるクラスは、全て Jacross で用意されたプロキシを通して参照される。この方式を用いる事により遠隔メソッド呼び出しがローカルへのメソッド呼び出しと同様に透過的にオブジェクトへのメソッド呼び出しという形で実現されている。Jacross では遠隔配置されるクラスに応じて以下のようにプロキシの設計方針（配置方針）を決定出来るようにする。

4.1.1　Replace

この配置方針では、遠隔配置の対象となるクラスの中身が全て Jacross で用意されるプロキシで置換される。そのため対象となるクラスが可変であることが必要条件となる。Jacross で用意されるプロキシは対象クラスと同じ構造・同じメソッドを持っている。ただし、メソッドの中身は遠隔

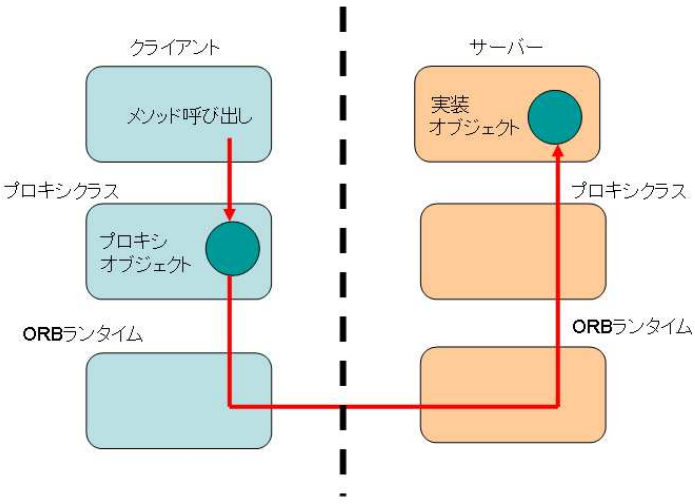

図 4.1: Remote Proxy パターン

で実行出来るようにコード変換されている。（図 4.2 参照）この配置方針は、参照者クラスが非可変でかつ、被参照者コードが可変である場合に特に有効な方針である。一方、１つのホスト上にローカルとリモートで参照を持ちたい場合には利用する事が出来ない。この配置方針では１つのホスト上の対象クラスへの参照はローカルか遠隔かどちらか一方に限られる。

4.1.2　Rename

この配置方針では、遠隔配置の対象となるクラスとは別の名前でプロキシを用意する。なお、Jacross では < クラス名 >_Proxy という名前でプロキシクラスを提供する。ここで用意されるプロキシは対象クラスと同じ構造・同じメソッドを持ちメソッドの中身のみが遠隔実行用に書き換えられている。遠隔参照を行うコードは、対象クラスを参照している箇所が全て書き換えられ、< クラス名 >_Proxy クラスへの参照に置き換えられる。（図 4.3 参照）参照の置き換えが必須となる為、参照者クラスが可変でなくてはならない。この配置方針は配置対象となるコードが非可変である場合に特に有効な方針である。また、この方針でも１つのホスト上からの対象クラスへの参照はローカルか遠隔かどちらか一方に限られる。

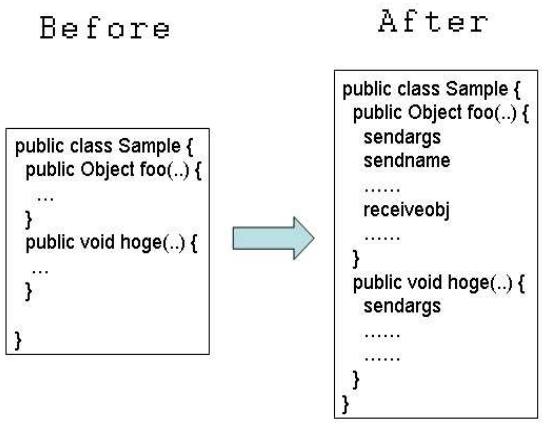

図 4.2: Replace

4.1.3　Subclass

この配置方針では、遠隔配置の対象となるクラスのサブクラスの形でプロキシを用意する。Jacross では、< クラス名 >_Proxy という名前で親クラスを対象クラスに持つプロキシクラスを提供する。メソッド・構造は親クラスと同様に作られるが、メソッド呼び出しは遠隔実行が可能なように中身が置き換えられている。この配置方針は主に 1 つのホスト上で対象クラスへの参照がローカルと遠隔の両方が混在する場合に用いられる。この場合、ローカル参照は元の対象クラスをそのまま用い、遠隔参照はサブクラス化したプロキシをインスタンス化し、そのオブジェクトを用いて遠隔実行を行う。(図 4.4) この配置方針は、他の手法と比較して制約が厳しく、参照者クラスが可変でなくてはならない上、対象クラスも可変でなくてはならなくては適用出来ない場合がある。それは対象クラスが修飾子に final を持っていたりメソッドが final だったりする場合、それを取り外さなくてはならないからだ。また対象クラスのコンストラクタ呼び出しが都合の悪い副作用を生じる場合、何もしないコンストラクタ呼び出しを追加しなくてはならない等の変更も必要になる。

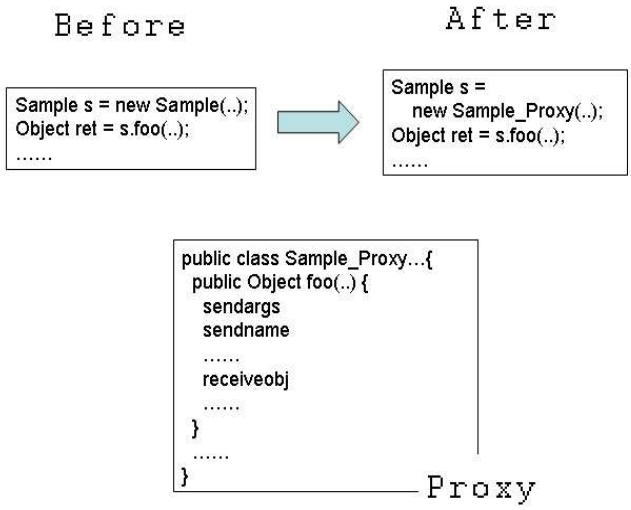

図 4.3: Rename

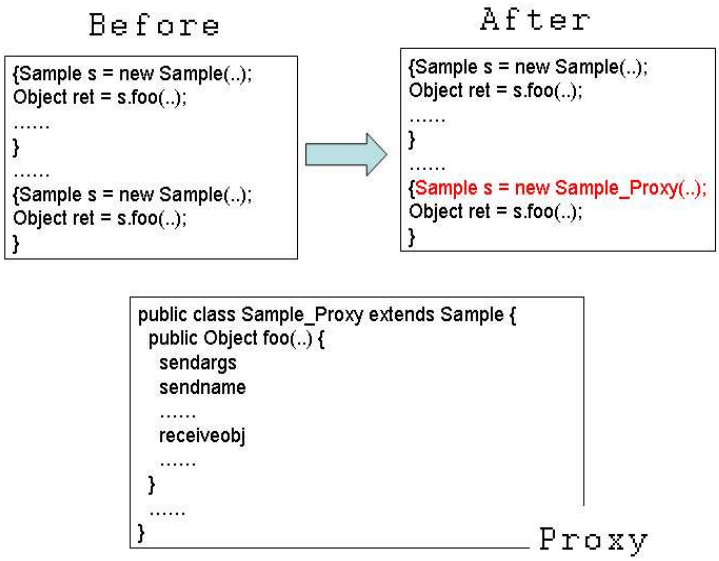

図 4.4: Subclass

4.1.4　配置方針のオプション

Jacross では上記の 3 つの方針で遠隔配置を自動化しているが、より詳細な配置方針を決定するオプションとして、以下の 2 つの機能を用意している。

permethod

メソッド単位での遠隔配置を行う為のオプション。前章で説明したように、この配置方針が用いられると対象メソッドのみが遠隔呼び出しされる。このオプションを用いると、対象オブジェクトはローカルホスト上と遠隔ホスト上にペアの形で 1 つずつオブジェクトが生成される。ローカルでのメソッド呼び出しはローカルオブジェクトを通し、遠隔でのメソッド呼び出しは、リモートで生成されたオブジェクトを通してメソッド呼び出しを行う。permethod オプションを用いる場合は、状態の反映を行わないといけないケースがあるため、対象クラス・参照クラスが共に可変である事が望ましい。Replace の場合、対象メソッドは遠隔実行できるように変更されたコードで置き換わる。またコンストラクタ呼び出しに同様のコンストラクタを用いて遠隔上にオブジェクトを生成するコードが加わる。ローカルのメソッド呼び出しはそのまま実行され、遠隔のメソッド呼び出しはペアとして用意された遠隔オブジェクトへの参照を用いて行われる。Rename の場合も同様に対象メソッドのみが遠隔実行できるように編集された < クラス名 >_Proxy の名前のプロキシクラスが生成される。Subclass の場合も同様で対象メソッドのみが遠隔実行されるようなプロキシクラスを対象クラスの子クラスとして用意する。

ORB

Jacross を用いると、ユーザは自分の好きな ORB を用いて遠隔実行の呼び出しを行う事が出来る。オリジナルな ORB を用いる場合、Jacross の仕様に基づいた形で ORB を定義すれば Jacross で用意されているものと差し替える可能である。Jacross で用意している ProxyTemplate クラスを継承し、指定のメソッドをオーバーライドする事により実装できる。また、あらかじめプロキシジェネレータが用意されているような ORB の場合、Jacross はそのプロキシジェネレータの仕様に対応する形でコードを編集する。現在の実装では、JavaRMI にのみの対応となっている。

4.1.5　具体的なコード変換処理の詳細

遠隔参照の自動化についての具体的なコード変換処理の詳細を説明する。Jacross はバイトコード変換で、これらの遠隔参照の自動化を行っておりバイトコードレベル API（Application Programming Interface）として Javassist [2, 24, 23] を用いている。Javassist は、Java の標準 API に含まれているリフレクション機能 [16] を強化したものであり、プログラムの構造を調べるだけではなく、定義を改変する事も出来るライブラリである。クラス Sample について、Javassist を使って Rename 方針でプロキシを生成する流れを説明する。以下は Jacross 中の ProxyGenerator を抜粋し、デフォルメしたコードである。

```
01 CtClass baseclass = pool.get(‘‘Sample’’);
02 CtClass proxyclass
       = pool.make(baseclass.getName() + ‘‘_Proxy’’);
03 CtMethod[] basemethods = baseclass.getDeclaredMethods();
04 CtField[] fields = baseclass.getDeclaredFields();
05 CtMethod[] proxymethods = new CtMethod[basemethods.length];
06 CtField[] proxyfields = new CtField[basefields.length];
08 for(int i = 0; i < basemethods.length; i++){
09   proxymethods[i] = makeproxymethod(basemethods[i]);
10   proxyclass.addMethod(proxymethods[i]);}
11 for(int j = 0; j < basefields.length; i++){
12   proxyfields[i] = makefieldmethod(basefield[i]);
13   proxyclass.addField(proxyfields[i]);}
```

ここで登場する CtClass、CtMethod、CtField などのクラスは、Javassist で用意されているクラスで、リフレクション API の Class,Method,Field クラスと類似した作りとなっている。01 行目は ClassPool 型の pool オブジェクトから Sample というクラスの CtClass オブジェクトを取得する処理である。ClassPool オブジェクトは、クラスを表す CtClass オブジェクトのコンテナとなっているオブジェクトである。02 行目は Sample_Proxy という名前で新たにクラスを生成する処理である。03,04 行目では Sample クラスから全てのメソッドと全てのフィールドを取得し、それぞれ CtMethod 型、CtField 型の配列に格納する。05,06 行目の処理ではプロキシ用に Sample クラスのメソッドとフィールドをそれぞれプロキシ化する為に同じサイズの配列を用意している。09 行目は、プロキシのメソッドを Sample のメソッドと同等な中身を ORB を用いた呼び出しに差し替える処理を行っている（makeproxymethod は Jacross 中の ProxyGenerator クラスのメソッド）。10 行目では、09 行目で生成された CtMethod オブジェクトをプロ

キシを表す CtClass オブジェクトに追加する処理である。CtClass では、構造リフレクションを扱う為、addMethod や addField といったメソッドが用意されている。このような構造リフレクションはメモリ上で処理が行われており、このままの状態であるとクラスファイルには反映されない。

```
14 proxyclass.writeFile();
```

そこで明示的に 14 行目のように CtClass オブジェクトに writeFile メソッド呼び出しをする事でクラスファイルに変換される。実際のプロキシ生成には、この他にも親クラスに対する処理やインターフェースに対する処理なども加わる事になる。

次に既存プログラム中の対象クラスの参照を、生成されたプロキシへの参照に変更する（Replace 方針では必要がない）。javassist.expr パッケージ内には ExprEditor というメソッドの振る舞いを変える機能を持つクラスがある。ExprEditor には引数の違う edit メソッドが 6 つ用意されている。この edit メソッドがメソッドの振る舞いを変える為のメソッドで、演算の種類ごと（メソッドコール、フィールドアクセス、インスタンス生成）に別々の処理を行う事が出来る。ExprEditor クラス自体は、java.awt パッケージのイベントリスナのアダプタクラスのように何も仕事はしないので、ユーザは適切にオーバーライドして利用しなくてはならない。

```
01 CtMethod cm = ...;
02 SampleEditor editor = new SampleEditor();
03 cm.instrument(editor);
-------------------------------------------------
04 public class SampleEditor extends ExprEditor {
05   public void edit(NewExpr n) throws ..{
06     if(n.getClassName().equals(‘‘Sample’’))
07       n.replace(‘‘{System.out.println(‘‘Sample generates’’);
08           $_=$proceed($$);};}’’);
09 }
```

上の例では、SampleEditor はメソッド内の Sample クラスのインスタンスが生成される時、インスタンス生成をする前に System.out.println(“Sample generates”); という式を行うように変更する edit 処理を行っている。06 行目の条件にマッチした場合、07 行目の replace メソッドの中の式に置き換えられる。replace メソッドは String 型の引数を取り、ここには置換する為のソースコードを記述する事が出来る。ここで、09 行目で行われている$_=$proceed($$); という処理は Javassist で用意されている特殊な構文で、replace メソッド内で元々呼び出されている処理を表している。以上のように用意された SampleEditor を使って 01〜03 行目の処理を行う

と、CtMethod オブジェクトである cm に対し、メソッドの内部を検索しインスタンス生成が行われる処理を発見したら edit(NewExpr) メソッドの処理が行われる。SampleEditor で他の引数の edit をオーバーライドする事で他の演算に対しても処理を行う事が可能である。Jacross ではこの edit(NewExpr) メソッドを用い、指定クラスのインスタンスが生成される箇所をプロキシ呼び出しに差し替えている。

4.2　分散処理用 Aspect の実装

permethod オプションを用いる場合や、セッション処理を加えたりするなどの特殊な分散化を行う場合、既存プログラムの遠隔配置を行うだけでは実現できない可能性がある。Jacross では、そのようなアプリケーションセマンティクスの変更を伴う分散化に対応する為にコード編集のための機能を提供している。

4.2.1　Inter-type Declaration の実現

アプリケーションのセマンティクスを変更する為に、既存のメソッドやフィールドだけでは足りず、新しくメソッドやフィールドを用意しなくてはならない場合がある。Inter-type Declaration は、このような場合の為の機能で通常の Java で書かれたクラスのメソッドやフィールドをバイトコードレベルで編集し、別のクラスに追加する機能である。例えば、permethod オプションを用いたとする。permethod オプションを用いる場合、ローカルのオブジェクトの状態と遠隔のオブジェクトの状態を保つためのコードを用意しなくてはならない時がある。スタンドアロンな環境を想定し設計されたプログラムでは、このようなコードは用意されていないので、新たに既存プログラムに必要なメソッドやフィールドを追加しなくてはならない。図 4.5 のように、Jacross は Inter-type Declaration 用に用意されたクラスから必要なメソッドやフィールドを抽出し、対象となるクラス（遠隔参照されるクラスとは限らない）に挿入する。

4.2.2　JacrossInterceptor

Inter-type Declaration を用いれば、新規にメソッドやフィールドを追加する事が出来る。しかし、これだけではアプリケーションセマンティクスの変更を行うには、まだ不十分である。アプリケーションセマンティクスの変更を行うには、新規に追加されたメソッドやフィールドあるいは、既存のメソッドやフィールドを利用する為のコードを埋め込まなくてはな

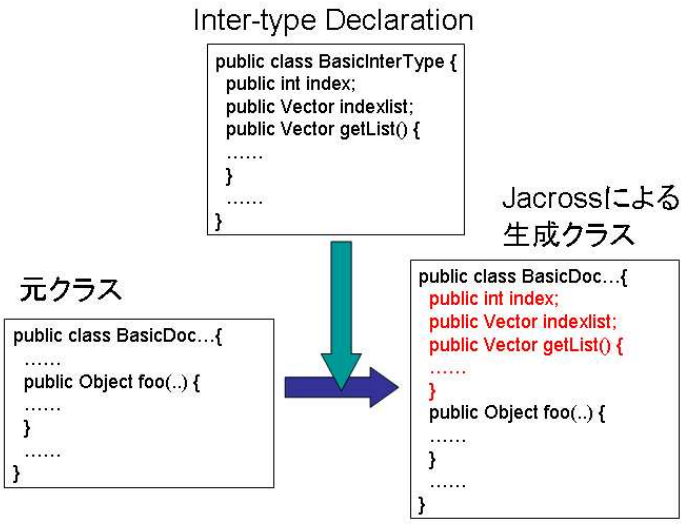

図 4.5: Inter-type Declaration

らない。Inter-type Declaration はあくまで、メソッドやフィールドの追加を行う為の機能なので、アプリケーションの制御フローを変える事は出来ない。そこで Jacross で提供しているのが JacrossInterceptor である。JacrossInterceptor は、前章で説明したインタセプタを実現する為のクラスである。インタセプタを用いる事が出来れば、プログラム中のあらゆる pointcut で好きな呼び出しを行う事が出来る。

では、具体的に JacrossInterceptor の動きについて説明する。ユーザは JacrossInterceptor を継承する形で、オリジナルのインタセプタを Pure Java で定義する。次にユーザは pointcut とインタセプタのメソッドの関連付けを行う。以上が行われると、Jacross はコード変換を行う事により指定された pointcut 箇所に対応するインタセプタのメソッド呼び出しを埋め込む。この編集されたプログラムを実行すると、pointcut 箇所で指定したインタセプタを割り込ませる形で実行する事が出来る。この JacrossInterceptor には、static メソッドとして proceed(Invocation) が用意されている。この proceed メソッドは pointcut された箇所の本来の呼び出しを行う為のメソッドである。Invocation クラスは pointcut された箇所のコンテキストを表している。proceed メソッドでは、引数として渡された Invocation オブジェクトから実行時のコンテキストを得て、リフレクションを用いる事によって、本来の呼び出しを行っている。この JacrossInterceptor の proceed メソッドを用いればインタセプタ内の好きな箇所で本来の呼び出しを行う事が出来る。

4.2.3　具体的なコード変換処理の詳細

Inter-type Declaration とインタセプタで行われているコード変換の詳細の説明を行う。プログラム変換には遠隔参照の自動化をした時と同様に Javassist を用いて実装を行った。

Inter-type Declaration

Inter-type Declaration は、指定したクラスの指定したメソッドを他のクラスに追加する機能であった。CtClass の addMethod や addField を利用する事で、CtClass オブジェクトへメソッドやフィールドの追加処理を行う事が出来る。ここでは例として、SampleIntertype 内の sampleMethod というメソッドと samplefield というフィールドを Sample クラスに追加する処理の説明をする。

```
01 CtClass intertype = pool.get(‘‘SampleIntertype’’);
02 CtMethod targetmethod
03     = intertype.getDeclaredMethod(‘‘sampleMethod’’);
04 CtField targetfield
05     = intertype.getDeclaredField(‘‘samplefield’’);
06 CtClass dclclass = pool.get(‘‘Sample’’);
07 CtMethod addedmethod
08     = CtNewMethod.copy(targetmethod, dclclass, null);
09 CtField addedfield = new CtField(targetfield, dclclass);
10 dclclass.addMethod(addedmethod);
11 dclclass.addField(addedfield);
```

まず Inter-type Declaration 用に用意された Java クラスと追加したいクラスから CtClass 型のオブジェクトを生成する（01、06 行目）。その後、追加したいメソッドやフィールドで CtMethod、CtField を生成する (02〜05 行目)。ここで取得した CtMethod オブジェクトは、そのままでは他のクラスに追加できないので CtNewMethod の copy メソッドを用いて複製を生成する（07〜08 行目）。CtField に関しても同様である（09 行目）。以上のようにして得られた CtMethod、CtField を追加する事で CtClass オブジェクトへ追加する事が出来る。後はこの CtClass ファイルをクラスファイルに変換すれば、処理は終了である。

インタセプタ

インタセプタの実装には、遠隔参照の自動化の際に用いた javassist.expr パッケージ内にある ExprEditor クラスを利用して行う。ExprEditor クラスには、上で説明したように 6 つの edit メソッドが用意されている。それらの edit メソッドをオーバーライドして処理を定義し、CtMethod の instrument(ExprEditor) メソッドの引数として呼び出す事でメソッドの中身を 1 文ずつ調べていき処理が行われていく。Jacross では、よく利用されそうな pointcut（メソッドコール、フィールドアクセス、インスタンス生成）について edit メソッドを用意し、利用出来るようにした。具体的に pointcut に対して、どのようにコード変換が行われているかの説明を行う。

```
01 public class SampleEditor extends ExprEditor {
02   public void edit(MethodCall m) throws ..{
03     if(m.getClassName().equals(‘‘Sample’’)&&
04         m.getMethodName().equals(‘‘sampleMethod’’)){
05       String src = ‘‘....’’;
06       m.replace(src);}
07 }
```

上の例では Sample クラスの sampleMethod 呼び出しが行われている箇所を String 型の src でコードを置換する作業である。03〜04 行目でクラス名とメソッド名がマッチするかの確認を行っている。尚、ここの条件式を色々と変える事によって、様々な形の pointcut（この場合はメソッドコール）に対応する事が出来る。また Jacross では、05 行目の src でインタセプタを割り込ませる処理を記述している。これを全てのメソッドに対して行えば Sample クラスの sampleMethod 呼び出しという pointcut に対してインタセプタを割り込ませる事が出来る。以上のように ExprEditor を用いる事で、メソッド呼び出し、フィールドアクセス、インスタンス生成について pointcut を定義し、インタセプタで処理を割り込ませる事が出来る。Jacross では、これらの 3 つの pointcut の他にメソッド実行をインタセプトする為の処理も行っている。このメソッド実行へのインタセプタの実装は、ExprEditor を用いずに行っている。以下のような Sample クラスの中のメソッド実行をインタセプトしたい場合の説明をする。

```
public String sampleMethod(Object[] args) {
  ......
}
```

```
01 CtMethod cm = cls.getDeclaredMethod(‘‘sampleMethod’’);
```

```
02 cm.setName(cm.getName() + ‘‘_EXECUTION_CHANGE’’);
03 CtMethod newmethod = CtNewMethod.make(cm.getReturnType(),...);
04 String src = ...;
05 newmethod.setBody(src);
```

まず 01 行目で対象となるメソッドの CtMethod オブジェクトを取得する。その後、02 行目でこの CtMethod オブジェクトの名前を < メソッド名 >_EXECUTION_CHANGE という名前に変更する。そして元の名前で戻り値、引数の型、例外の型が同じメソッドを新たに定義する（03 行目）。そのメソッドの中身を 04 行目で表す src で置換する（05 行目）。Jacross では 04 行目で指定する src 部分にインタセプタに処理させる為の記述を行っている。そして、インタセプタ内で proceed を呼ぶと、< メソッド名 >_EXECUTION_CHANGE メソッドを pointcut した箇所の引数で呼ぶ事が出来る。

4.3　XML による分散ポリシーの記述

Jacross ではプログラムをどのように分散化するかの指定する分散ポリシーに XML を採用している。分散ポリシーにユーザが記述できる情報は以下の通りである。

プロキシの配置方針の決定
: 4.2 で説明するクラスの分散配置の方針を決定する。ユーザはクラスのタイプから方針を、また対象クラスをどう遠隔配置したいかによってオプションを選択しなくてはならない。ここで選択されなかったクラスにはプロキシは生成されない。

Inter-type Declaration の指定
: Inter-type Declaration として定義された Java ファイルと、それを埋め込みたい既存プログラム中のクラスファイルの関連付けを行う。

pointcut の定義
: ユーザは、好みに応じて pointcut を定義する事が出来る。現在、pointcut としてサポートしているのはメソッド実行、フィールドアクセス、コンストラクタ呼び出しのみである。

インタセプタの設定
: 定義された pointcut とインタセプタのメソッドの関連付けを行う。

XML のパーサーには SAX（Simple API for XML）を用いて実装を行った。SAX は XML のデータを構文解析する為のイベント駆動型の API で

ある。SAX では、構文解析でタグを認識すると「タグが読み込まれた」というイベントを発生させる。このイベントのリスナーを実装する事で必要な処理を行っていく流れになる。必要な情報をイベント通知の形で受け取ることができるため、必要以上にメモリを消費しないという利点がある。Jacross で用意する XML の構造は図 4.6 のようになる。各タグの利用方法と詳細については、次章で説明する。これらのタグに対して Jacross では、それぞれにリスナーを設け各処理郡に処理を渡している。

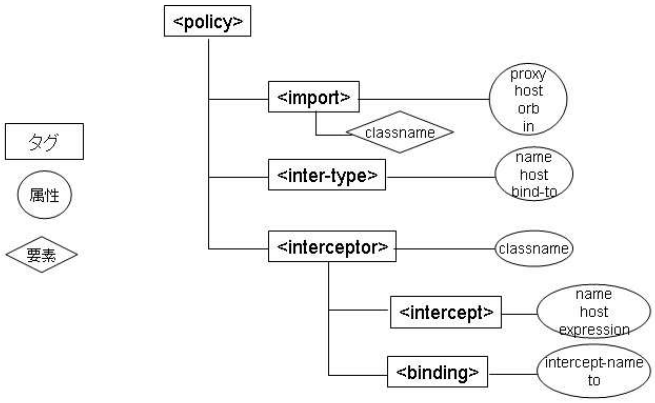

図 4.6: XML の構造

第5章　適用と利用例

この章では、Jacross の利用方法と利用例に関する説明を行う。まず最初に分散ポリシーの記述方法について説明し、次にインタセプタと Inter-type Declaration をどのように利用するかの説明を行う。利用例の節では、対象ソフトウェアとして２章で説明した ARMSoftware を用いた。

5.1　分散ポリシーの利用方法

まずは分散ポリシーをどのように記述すればいいかの説明を行う。Jacross の分散ポリシーで扱えるタグは、５つ用意されている。それぞれのタグに対して属性があり、タグに対応した設定を行わなくてはならない。

5.1.1　<import> タグ

import タグは、プロキシの配置方針を決定する為のタグである。要素としてクラスを選択し、属性によって指定された設定値を元にプロキシの生成と参照の付け替えを行う。import タグには４つの属性が用意されている

proxy
: proxy 属性では、プロキシの生成方針を設定する。ここで設定できる属性は、Replace、Rename、Subclass の３つに加え permethod オプションを追加した Replace-permethod、Rename-permethod、Subclass-permethod の合計６つである。

host
: host 属性では、遠隔配置したいクラスをどこに配置するかを決定する。

in
: in 属性は、Subclass 方針を用いる場合にのみ必要とされる。in 属性で指定したクラス内からの参照のみが遠隔参照として適用される。

orb

orb 属性は特に指定がなければ、Jacross で用意された標準の ORB が用いられる。現在のところ、専用の ORB を用いる際には Jacross の ProxyTemplate クラスを継承し、対応するメソッドの書き換えを行わなくてはならない。また、プロキシジェネレータが用意されている ORB（JavaRMI、HORB など）に対しては、JavaRMI のみのサポートを行っている。

要素として指定できるクラスは、< クラス名 > で対応するクラスが選択される。また親クラスが明示的に用意されているものについては、< クラス名 >@< スーパークラス名 > と記述しないとならない。

5.1.2　<intertype> タグ

intertype タグでは、Inter-type Declaration として用いたいクラスの設定と、それをどのクラスに挿入するかの設定を行う。intertype タグで用意されている属性は以下の 3 つである。

name
: name 属性では Inter-type Declaration として用意したクラスの名前の指定を行う。

bind-to
: bind-to 属性では、メソッドやフィールドの追加先のクラスの指定を行う。

host
: bind-to で追加先のクラスの選択を行うが、host 属性で配置先も指定する事が出来る。host を指定すると、そのホストに配置された bind-to クラスのみに Inter-type Declaration 処理が行われる。例えば、bind-to に Sample クラスを指定し、host で App サーバというホストを選択すると、App サーバ内の Sample クラスにのみ Inter-type Declaration 処理が行われ、他のホストに配置されている Sample クラスに対しては処理は行われない。

5.1.3　<intercept> タグ

intercept タグは、内部に interceptor タグと binding タグの 2 つを含み、それ以外に 1 つの属性を持つ。

name
: インタセプタとして用意したクラスの名前を設定する。

5.1.4　<interceptor> タグ

interceptor タグは intercept タグ内に含まれ、pointcut の設定をする為のタグである。どの箇所でインタセプタの呼び出しを割り込ませて行うかの決定を行う。属性としては、以下の 3 つを用意してある。

name
: name 属性では、後述する binding タグで利用する為に、この pointcut 箇所の名前を定義する。

host
: host 属性を設定すると、設定されたホスト上でのみインタセプタによる割り込み処理が行われる事になる。

expression
: 具体的に pointcut の設定を行う。Jacross では pointcut を設定する為に以下の 5 つの指定子を用意している。

表 5.1: expression で用いる事の出来る指定子

指定子	意味
call	メソッドの呼び出しを表す
execution	メソッドの実行を表す
set	フィールドへの書き込み
get	フィールドの読み出し
new	インスタンス生成

call、execution は、call/execution(戻り値の型 クラス名 メソッド名 （引数の型）) で表す。set、get は、set/get(フィールドの型 クラス名 フィールド名) で表す。new は、new(クラス名 （引数の型）) で表す。

5.1.5　<binding> タグ

binding タグも intercept タグ内に含まれる。このタグは interceptor タグで設定されたそれぞれの pointcut に対して、指定したインタセプタのメソッドを割り込ませる為のタグである。属性として以下の 2 つを含む。

intercept-name
: intercept-name 属性では、intecept-name 属性で選択した pointcut で実行したいインタセプタのメソッド名を選択する。

to

to 属性では、intercept タグで設定された pointcut を選択する事が出来る。それ以外の名前を設定するとエラーとなる。

5.2　インタセプタの利用方法

インタセプタは、基本的に Pure Java で用意する事が出来る。しかし Jacross で用いる事の出来るインタセプタには多少の制約があるので、その利用方法について説明する。

1. **インタセプタは、クラス JacrossInterceptor を継承する形で定義しなくてはならない**
2. **割り込み処理をさせるメソッドは Object 型の戻り値と Invocation 型の引数を取らなくてはいけない**

Jacross でインタセプタを用いる為の制約は、以上の 2 つである。

まず、1 の制約が必要な理由を説明する。1 の制約が必要なのは、Jacross-Interceptor で定義されている proceed(Invocation) メソッドを利用する為である。JacrossInterceptor で定義されている proceed メソッドは、Invocation オブジェクトからリフレクションを用いて pointcut 箇所の元の呼び出しが行う事が出来る。よって、割り込み処理中で proceed を使って、元の呼び出しを実行する為には、JacrossInterceptor を継承しなくてはならない。

次に 2 の制約が必要な理由を説明する。割り込み処理をさせるメソッドには pointcut 箇所の情報が必要になる（例えばメソッドコールであれば、呼び出しているオブジェクトが何であるか等）。そこで Jacross では、これらの pointcut 箇所のコンテキストを Invocation という型のクラスにして処理を行っている。なお、戻り値の Object 型のオブジェクトには、pointcut された箇所の戻り値を返すように記述しなくてはならない。クラス Invocation に用意されているメソッドは以下の表の通りである。これらのメソッドを用いてユーザは pointcut 箇所の情報を用いて処理を記述する事が出来る。pointcut 箇所によっては、用いる事が出来ないメソッド等もある（フィールドアクセスを pointcut として指定した場合、getMethodName は利用出来ない）。またインタセプタ内で Inter-type Declaration で新たに定義したメソッド等を利用する場合は、クラスの挿入先を予測してコードを記述しなくてはならない。図 5.1 は、pointcut 箇所の前後で pointcut 箇所のメソッドの名前などを表示する為のログ処理を行うインタセプタである。

表 5.2: Invocation クラスのメソッド（抜粋）

戻り値	名前	処理内容
Object	getThis()	pointcut 箇所の呼び出しを行っている オブジェクトを返す
Object	getTarget()	pointcut 箇所で行われている呼び出しの 対象オブジェクトを返す
String	getClassName()	呼び出しの対象オブジェクトの クラス名を返す
String	getMethodName()	呼び出し対象のメソッドの名前を返す
Object[]	getArgs()	呼び出し対象のメソッドあるいは コンストラクタ呼び出しの引数を返す

```
public class SampleInterceptor
    extends JacrossInterceptor {

  pulbic Object printLog(Invocation i) {
    String methodname
      = i.getMethodName();
    System.out.println(methodname + " before");
    Object ret = proceed(i);
    System.out.println(methodname + " after");
    return ret;
  }
}
```

図 5.1: ログ出力処理のインタセプタ例

5.3　Inter-type Declaration の利用方法

Inter-type Declaration として利用する Java ファイルには特に制約を設けていない。クラスファイルの中のメソッドやフィールドがそのまま、他のファイルに追加される事になる。this もそのまま挿入される為、参照先は全て挿入先のクラスのオブジェクトになる。

5.4　利用例

Jacross を ARMSoftware に適用して、手動で分散化したものとの比較実験を行った。なるべく近い環境での比較を心がける為に変換処理は手動で行ったものと、なるべく同じ変換処理を行うような分散ポリシーを用意

した。

5.4.1　ARMSoftware を Jacross を用いて変換

ARMSoftware の変換に必要な分散ポリシーを以下に示した。ワイルドカードとして*を使用している。分散処理用 Aspect は以下の通りになる

```
<policy>
    <import proxy="rename-permethod" host="server" orb="JavaRMI">
    doctype.*Doc.searchDoc(..)@util.DocType
    map.enzMaps.getMap(..)
    util.Base.getID(..)
    doctype.*Doc.remoteget*(..)
    doctype.*Doc.remoteset*(..)
    ......
    </import>

    <inter-type name="DocIntertype" host="server"
       bind-to="doctype.*Doc"/>
    <inter-type name="MapIntertype" host="server"
       bind-to="map.enzMaps"/>
    <inter-type name="ViewIntertype"
       bind-to="main.ArmView"/>
    ......
    <interceptor classname="DocInterceptor">
        <intercept name="searchremoteMol" host="server"
           expression="execution(Object[] doctype.MolDoc.searchDoc(..))"/>
        <intercept name="searchremoteEnz" host="server"
           expression="execution(Object[] doctype.EnzDoc.searchDoc(..))"/>
        <intercept name="searchLocal" host="local"
           expression="call(Object[] doctype.*.searchDoc(..))"/>
        <intercept name="setURLList" host="local"
           expression="call(void util.*.setURL(..))"/>
        ......
        <binding intercept-name="setMolSession"
                to="searchremoteMol"/>
        <binding intercept-name="setEnzSession"
                to="searchremoteEnz"/>
        <binding intercept-name="synchronizeURLList"
                to="setURLList"/>
        ......
    </interceptor>
</policy>
```

図 5.2: ARMSoftware 用分散ポリシー（抜粋）

(図 5.3、図 5.4 参照)。 この処理では、DocIntertype でセッション処理用と同期用のメソッド、フィールドの追加を行っている。searchDoc というメソッドが呼ばれる際に必要になる urlList というフィールドの同期を取る為のリモートメソッドを Inter-type Declaration で定義する。また複数のクライアントにも対応出来るように、それらのフィールドをリスト化し管理するメソッドの追加を行う。ローカル側のインタセプタでは、これらのメソッドを利用し searchDoc メソッドを呼ぶ前に遠隔にあるオブジェクトの urlList フィールドに set を行い、呼び出しが終わった後に遠隔のオブジェクトの urlList を get し、ローカルの物と置換する (synchronizeInvoke メソッド)。またサーバ側では、ユーザ ID に基づいた urlList などを得る為に searchDoc メソッドを実行時にインタセプトし、セッション処理を行っている（setSession メソッド)。

```
public class DocIntertype {
    private static LinkedList urlsession = new LinkedList();
    private static LinkedList indexsession = new LinkedList();
    public static Vector getURLlist(int id) {
        return urlsession.get(id);
    }
    public static void setURLlist(Vector urllist, int id) {
        ......
        urlsession.set(id,urllist);
        ......
    }
    public static int getURLindex(int id) {
        return indexsession.get(id);
    }
    public static void setURLindex(int urlindex, int id) {
        ......
        indexlist.set(id,urlindex);
        ......
    }
    public Vector remotegetURLlist(int id) {
        return getURLlist(id);
    }
    public void remotesetURLlist(Vector urllist, int id) {
        setURLlist(id,urllist);
    }
    public int remotegetURLindex(int id) {
        return getURLindex(id);
    }
    public void remotesetURLindex(int urlindex, int id) {
        setURLindex(id,urlindex);
    }
    ........
}
```

図 5.3: ARMSoftware 用 Inter-type Declaration（抜粋）

```
public class DocInterceptor extends JacrossInterceptor {
    public Object setMolSession(Invocation i) {
        MolDoc _this = (MolDoc)i.getThis();
        _this.urllist = AbstractDoc.getURLlist(_this.id);
        _this.urlindex = AbstractDoc.getURLindex(_this.id);
        Object ret = proceed(i);
        AbstractDoc.setURLlist(AbstractDoc.id,_this.urllist);
        AbstractDoc.setURLindex(AbstractDoc.id,_this.urlindex);
        return ret;
   }
    public Object synchronizeInvoke(Invocation i) {
        Base _this = (Base)i.getThis();
        AbstractDoc _target= (AbstractDoc)i.getTarget();
        _target.remotesetURLlist(_this.urllist,ArmView.id);
        _target.remotesetURLindex(_this.urlindex,ArmView.id);
        _target.setID(ArmView.id);
        Object ret = proceed(i);
        _this.urllist = _target.remotegetURLlist(ArmView.id);
        _this.urlindex = _target.remotegetURLindex(ArmView.id);
        return ret;
    }
    public Object synchronizeURLList(Invocation i) {
        .........
    }
    public Object synchronizeMap(Invocation i) {
        .........
    }
    ........
}
```

図 5.4: ARMSoftware 用インタセプタ（抜粋）

5.4.2　手動分散との比較

実際に ARMSoftware に Jacross を利用し適用したプログラムと、手動で分散化したプログラムと、オリジナルのプログラムの比較を行った。比較に用いた環境を以下に示す。

- ARMSoftware
  総行数：25866 行、総クラス数：286
- 計算機
  Panasonic 製 Let's Note R3（PentiumM 1.1GHz、768MB、WindowsXP）
- JVM
  Sun JDK1.4.2 付属の HotSpot^{TM}Client VM
- バイトコード API
  Javassist version 3.0 beta

比較対象となるのは、オリジナルのプログラム、手動で分散化したプログラム、Jacross を用いて分散化したプログラムの 3 つである。分散化する際に変更が必要なクラス数（新しく用意したクラスも含む)、コード数の比較を行った。Jacross を用いた場合、分散ポリシーに記述したコード数は、変更に必要なコード数に含まれている。表 5.3 の結果を得ることが出

表 5.3: コード数の比較

比較対象	変更が必要なクラス数	変更に必要なコード数（行数）
オリジナル⇔手動	33	951
オリジナル⇔ Jacross	5	349
割合（手動/Jacross）	15%	36%

来た。クラス数の比較では、ローカルと遠隔の両方に変更が必要な場合は、まとめて 1 つと換算している。コード数の比較では、ローカルと遠隔の両方に形が微妙に違う同名のクラスがある場合、それぞれについて差分を取ってある。

この結果を見ると、手動分散の方が Jacross を用いた分散化より変更点が多いことが分かる。これは、手動分散のプログラムが直接 Java コードを編集できる為か、モジュール化をうまく考えられない設計になっていた為と思われる。例えば、セッション処理を行うコードは Jacross ではインタセプタの setSession というメソッドでモジュール化し、セッション処理

が必要な複数箇所で利用している。しかし、手動で行った分散プログラムではセッション処理を行わなければならないコードに対してコードを直接書き込んでおり似たような処理内容が散在してしまっていた。また手動では、分散化を行う際に各クラスに対し遠隔参照へと参照の変更を行った箇所があったので、変更が必要なクラス数も多くなっている。

第6章　まとめ

本論文では、分散に関する記述力を強化した、既存 Java アプリケーションの為の分散化支援システム Jacross の提案を行った。Jacross では、Aspect 指向技術を利用し自動分散化処理と共にプログラムの構造を変更する事が出来る。このプログラムの構造を変化させる為の記述は、分散処理用 Aspect としてモジュール化されており、Jacross の定めるフォーマットに基づいて Pure Java の形で記述する事が出来る。スタンドアロンで動作する事を想定された現実的なアプリケーションは自動分散化に適した設計になっているとは限らない。また分散化を行う際にクライアント数を複数にする等の分散アプリケーション特有の機能拡張を施したい場合もある。そこでこのようなアプリケーションに対し、Jacross の分散処理用 Aspect を用いる事で、従来の自動分散化システムがサポートしていない、アプリケーションセマンティクスの変更を伴うような分散化にも対応する事が出来る。

ユーザは複雑な分散化を行いたい時に、分散処理用 Aspect を用いる事が出来る。分散処理用 Aspect を用いれば、元プログラムのソースコードを変更せずに、元プログラムを任意の形に変更する事が出来る。分散処理用 Aspect は付加的な扱いである為、プログラムの構造を変化させずに分散化出来る場合は、必ずしも用いる必要はない。アプリケーションセマンティクスの変更が必要ない分散化の場合は、分散処理用 Aspect を用意せずに、従来の自動分散化システムとほぼ同等の記述で分散化する事が出来る。Jacross が用意する分散 Aspect は、プログラムの制御を変える為のインタセプタ、プログラムに新しくメソッドやフィールドを追加する為の Inter-type Declaration の 2 つの Java で記述されたファイルと、それらのファイルを既存プログラムにバインドする為の XML で構成される。利用者は分散化の為に必要な機能拡張をインタセプタや Inter-type Declaration で用意し、それらを適切な箇所に XML を用いてバインドする。Jacross は、分散化処理と同時にそれらの記述に従ったプログラム変更を行い、分散プログラムを生成する。

Jacross では、従来システムにはないメソッド単位での分散化機能を提供している。現実的なアプリケーションは分散化に適したモジュール分割がされているとは限らず、クラスより細かい単位での分散化を行いたい場

合もある。このような場合、メソッド単位での分散化を用いる事により、より粒度の細かい分散配置を行う事が出来る。メソッド単位での分散化は、ローカルと遠隔に 1 つずつオブジェクトを生成し、遠隔のメソッド呼び出しの場合のみ遠隔オブジェクトにアクセスする。その為、利用する際にローカルと遠隔のオブジェクト間の同期を取る為の処理を分散処理用 Aspect で用意しなければならない場合もある。

既存の Aspect 指向言語は、Jacross で利用出来る Aspect 記述と比べて記述力が強力である。Jacross で利用出来る pointcut の指定子は基本的な 5 つの物しか用意していないが、既存の Aspect 指向言語では数多くのものが利用出来る。仮に Jacross で、AspectJ [7] の cflow や RemotePointcut [17] などの pointcut を利用する事が出来れば、より柔軟に分散処理用 Aspect を作る事が出来るだろう。より多くの pointcut 指定子を Jacross で利用できるようにする事が、今後の課題として考えられる。

参考文献

[1] "Spring AOP".
`http://www.springframework.org/`.

[2] Shigeru Chiba. Load-time structural reflection in java. In *Proceedings of the European Conference on Object-Oriented Programming*, LCNS1850, pp. 313–336, Jun. 2000.

[3] Shigeru Chiba and Kiyoshi Nakagawa. "Josh:An Open AspectJ-like Language". In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Develepment(AOSD'04)*, Mar. 2004.

[4] Codehaus. "AspectWerkz".
`http://aspectwerkz.codehaus.org/index-aw.html`.

[5] M. Fleury and F. Reverbel. "The JBoss Extensible Server". In *Proceedings of International Middleware Conference*, LCNS2672, 2003.

[6] JBoss Group. "JBoss AOP".
`http://jboss.org/products/sop`.

[7] Gregor Kiczales, Erik Hilsdale, Mik Kersten Jim Hugunin, Jffrey Palm, and William G. Grisword. "An Overview of AspectJ". In *European Conference on Object-Oriented Programming(ECOOP 2001)*, LCNS2072, pp. 327–353, 2001.

[8] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming". In *European Conference on Object-Oriented Programming(ECOOP 2001)*, LCNS1241, pp. 220–242, 1997.

[9] M.Arita. "ARMSoftware".
`http://metaboleme.jp/index.html`.

[10] M.Arita. "The Metabolic World of E.coli Is Not Small". In *Proceedings of the National Academy of Sciences USA)*, 2004.

[11] Sun Microsystems. "EnterPrise JavaBeens Technology". `http://java.sun.com/products/ejb/`.

[12] Sun Microsystems. "Java RMI". `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/`.

[13] Sun Microsystems. "Java Servlet Technology". `http://java.sun.com/products/servlet/`.

[14] Sun Microsystems. "Java2 Platform Enterprise Edition(J2EE)". `http://java.sun.com/j2ee/`.

[15] Sun Microsystems. "JavaServer Pages Technology". `http://java.sun.com/products/jsp/`.

[16] Sun Microsystems. "Reflection". `http://java.sun.com/j2se/1.5.0/docs/guide/reflection/`.

[17] Muga Nishizawa and Shigeru Chiba. "Remote Pointcut — A Language Construct for Distributed AOP". In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Develepment(AOSD'04)*, Mar. 2004.

[18] Michael Philippsen and Matthias Zenger. "JavaParty Transparent Remote Objects in Java". *Concurrency: Practice and Experience*, Vol. 9, No. 11, pp. 1225–1242, November 1997.

[19] Hans Rohnert. "The Proxy Design Pattern Revisited". In *Pattern Languages of Program Design 2*, AddisonWesley, 1995.

[20] Andre Spiegel. "PANGAEA : An Automatic Distribution Front-End for JAVA". In *Fourth IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '99)*, IPPS/SPDP Workshops, pp. 93–99, 1999.

[21] MIchiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. "A Bytecode Translator for Distributed Execution on "Legacy" Java Software". In *Proceedings of the European Conference on Object-Oriented Programming(ECOOP2001)*, LCNS2072, pp. 236–255, 2001.

[22] E. Tilevich and Y. Smaragdakis. ”J-Orchestra: Automatic Java Application Partitioning”. In *European Conference on Object-Oriented Programming (ECOOP2002)*.

[23] 千葉滋. Javassist home page. `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.

[24] 千葉滋, 立堀道昭. Java バイトコード変換による構造リフレクションの実現. 情報処理学会 論文誌, Vol. 42, No. 11, pp. 2752–2760, Nov. 2001.

[25] 立堀道昭, 千葉滋. Addistant:アスペクト指向の分散プログラム支援ツール. 情報処理学会 論文誌, Vol. 43, No. 3, pp. 17–25, Mar. 2002.