アスペクト指向を用いたアジャイル分散ソフトウェア開発のための環境

西澤無我[†1] 栗田洋輔[†1] 千葉滋[†1]

本稿では，アスペクト指向を利用したアジャイルな分散ソフトウェア開発手法を提案する．従来のアジャイル開発手法と異なり，提案手法では各反復に追加する機能やリファクタリングのためのコードを，アスペクトとしてモジュール化する．アスペクト記述は他のプログラムから分離されるので，アスペクトとして実装された追加機能等は容易に既存プログラムから抜き差しできる．このような開発手法を支援する Java 用の環境として Remote GluonJ を開発した．Remote GluonJ の言語機構を用いることで，開発者は分散した横断的関心事を含む追加機能等を，1 つのホスト上で動作する単一のアスペクトとして記述できる．アジャイル開発の対象が分散ソフトウェアである場合，非分散のときよりも機能追加やリファクタリングの負担が大きいにもかかわらず，従来のアスペクト指向言語ではそれらにうまく対処しきれなかった．また Remote GluonJ によって，開発者は分散ソフトウェアを明示的に再起動することなく，アスペクトを既存プログラムに適用できる．Remote GluonJ のプログラムは既存の Java 開発環境上で開発できる．本稿は，上述した提案手法と Remote GluonJ の特徴について詳しく説明する．さらにケーススタディとして，我々は提案手法および Remote GluonJ を用い，分散レイトレーシング・アプリケーションを実際に開発した．その概要についても述べる．

An AOP Based Agile Development Environment for Distributed Software

Muga Nishizawa,[†1] Yohsuke Kurita[†1] and Shigeru Chiba[†1]

This paper proposes an AOP based method of agile development for distributed software. Unlike existing methods of agile software development, in our method, developers modularize a function and refactoring newly appended to an existing program as an aspect each iteration. Since an aspect is separated from the rest of programs, the appended function can be added/removed to/from the existing program easily. To support software development for Java with our method, we developed Remote GluonJ. Language constructs of Remote GluonJ allows implementing a newly appended function that includes a distributed crosscutting concern as a single aspect running on a single host. When a target is distributed software, in spite of the fact that developers are more burdensome for appending a new function and refactoring of distributed software than non-distributed software, they could not implement those as an aspect with existing AOP languages. Also by using it, developers can apply an aspect to distributed software without explicitly rebooting that program. Moreover, a program of Remote GluonJ can be developed with an existing Java Integrated Development Environment. This paper presents our method and these features of Remote GluonJ in detail. Moreover, as a case study of our method and Remote GluonJ, we developed distributed raytracing application. This paper illustrates the summary of that development.

1. はじめに

最近，ユーザからのフィードバックを受けながら，少しずつ新しい機能を追加実装していくアジャイルソフトウェア開発手法が注目されてきている．しかしこの手法では，実装すべきソフトウェアの全容が事前に分からないので，開発者はそのソフトウェアを適切に設計しにくい．そのようにして開発されたプログラムのモジュラリティは低くなる傾向があり，そのプログラムに追加される新機能はしばしば横断的関心事を含む．一般的なアジャイル開発では，プログラムのモジュラリティを高めるため，各反復の新機能を追加する前にそのプログラムの再設計と大規模なリファクタリングを行う．分散ソフトウェアのアジャイル開発において，そのような再設計とリファクタリングは特に負担が大きく，アジャイルな開発手法を用いる際の障害になっていた．

†1 東京工業大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology

そこで我々は，分散ソフトウェアのための，アスペクト指向を利用したアジャイル開発手法を提案する．一般的なアジャイル開発手法と異なり，提案手法では各反復において，ユーザからのフィードバックに従って追加される機能やリファクタリングのコードを，アスペクトとして実装する．アスペクトとして新機能を実装すれば，前回までの反復で開発されたプログラムに手を加える必要がない．それゆえ，仮にユーザからの評価が低くその機能を削除する場合でも，分散ソフトウェアからアスペクトを取り外すだけでよい．また，機能を追加したことによってプログラムにバグが生じても，そのアスペクトやそれがポイントカットしている箇所をチェックしてバグを探すことができる．

さらに我々は，提案手法による分散ソフトウェア開発を支援する Java 用の環境 *Remote GluonJ* を開発した．各反復で追加する機能やリファクタリングは分散した横断的関心事を含むため，これらの負担が大きいにもかかわらず，既存のアスペクト指向言語ではうまく対処しきれない．Remote GluonJ の言語機構を使うことで，開発者は分散した横断的関心事を含む追加機能やリファクタリングのコードを，1 つのホスト上で動作する単一のアスペクトとして実装できる．また開発者は，分散ソフトウェアを明示的に再起動することなく，その分散ソフトウェアにアスペクトを適用できる．Remote GluonJ のプログラムは，既存の Java 開発環境を用いて開発することができる．

本稿では，まず 2 章で既存のアジャイルな分散ソフトウェア開発における問題点を指摘し，アスペクト指向を利用したアジャイル開発手法を提案する．また，提案手法での分散ソフトウェア開発を支援する環境 Remote GluonJ の特徴について述べる．3 章で Remote GluonJ の言語機構について説明する．4 章は Remote GluonJ の実行時システムの主要な機能の仕様とその実装について説明する．5 章では，ケーススタディとして分散レイトレーシング・アプリケーションのアジャイル開発を取り上げる．6 章で Remote GluonJ の関連研究を示し，7 章で本稿をまとめる．

2. 分散ソフトウェアのアジャイル開発を支援する環境

今日，ソフトウェアを迅速に開発する手法として，**アジャイルソフトウェア開発手法**が注目されてきている．既存のウォーターフォール・モデルを用いた開発手法と異なり，アジャイルソフトウェア開発ではソフトウェアの開発期間をいくつかの**反復**（1 週間程度の短い開発期間）に分割する．各反復では，ユーザからのフィードバックを受け，前回までの反復で開発したプログラムに機能を追加し，少しずつ改良していく．開発者は 1 つの反復内で，要求分析，設計，実装，テスト，文書化の工程を行う．この反復を継続することで，開発者は目標のソフトウェアを開発していく．各反復でユーザからのフィードバックを受けることにより，ユーザのニーズに素早く適応し，リスクを軽減できることが特徴である．

アジャイル開発において，既存のプログラムへ追加する機能やリファクタリングは，しばしば横断的関心事を含む．つまり機能等を実装するために，既存プログラムの複数の箇所を編集する必要がある．アジャイル開発では，開発すべき機能の全容が事前に分からないので，開発者はソフトウェア全体を事前に適切に設計しにくい．またこの開発手法では，動作するプログラムを手早く作成することが重要視されているため，開発者は反復ごとに新しい機能をアドホックに追加実装しがちである．このようにして開発されたソフトウェアに新機能を追加する場合，新機能の実装はより横断的になる．またアジャイル開発の各反復では，機能を追加するために既存プログラムを事前に再設計，リファクタリングし，プログラムのモジュラリティを高める．この再設計と大規模なリファクタリングの負担は大きく，アジャイル開発を用いる際の障害の 1 つになっていた．

アジャイル開発の対象が分散ソフトウェアである場合，そのような機能追加とリファクタリングは分散した横断的関心事を含むので，非分散のときよりも開発者の負担が大きい．分散ソフトウェアに機能を追加したりリファクタリングしたりするために，開発者は複数ホスト上で動作する既存プログラムをしばしば修正する必要がある．さらにリファクタリングでは，既存の複数の分散コンポーネントのインタフェースをしばしば変更しなければならない．この場合，遠隔ホスト上のそのコンポーネントのインタフェースを変更するだけではなく，そのコンポーネントを利用しているクライアント側のコードも修正しなければならない．クライアント・コードを探すために，複数のホスト上で動作するプログラムをいちいちチェックしなければならない．

また，次の反復のユーザのフィードバックの工程で，実装した追加機能の評価が低ければ，その機能等をプログラムから削除しなければならないかもしれない．この場合，複数ホスト上のプログラムに加えた修正を元に戻す必要がある．また，分散コンポーネントのインタフェースを元に戻し，そのコンポーネントを利用

しているクライアント・プログラムを再度修正する必要がある．機能を追加するために行った既存プログラムの修正とそれにともなうリファクタリングが無駄になる．

さらにある反復でいくつかの機能を追加する場合，それらを並列に実装しにくい．追加するすべての機能のための再設計とリファクタリングが終了しなければ，開発者は各機能の実装を始められない．これは，ある機能のための再設計とリファクタリングが，他の機能のための再設計とリファクタリングに依存するかもしれないためである．各機能を別々のグループで開発していく場合，この問題はソフトウェアの開発効率を著しく低下させる原因となる．

2.1 アスペクト指向を利用したアジャイル分散ソフトウェア開発手法

そこで我々は，分散ソフトウェアのための，アスペクト指向を利用したアジャイル開発手法を提案する．この手法では各反復において，ユーザからのフィードバックに従って追加される機能を，アスペクトとして手早くプロトタイピングし，動作チェックを行う．その後，再度ユーザからのフィードバックを受け，評価が高かった機能だけを次の反復の直前にアスペクト抜きで実装し直す．実装にあたっては，既存のソフトウェアの再設計とリファクタリングを行い，通常のオブジェクト指向の範囲でその新機能を実装する．ただし，リファクタリングした後でも，追加機能が横断的関心事を含むのであればアスペクトのままでよい．

アスペクト抜きで実装し直す理由は，後の反復の機能追加を容易にするためである．追加機能をアスペクトのままにしておくと，反復のたびにアスペクトが実装されていく．以前の反復までに実装されたアスペクトに対して，しばしばアスペクトで拡張を加える必要も出てくる．しかし現状のアスペクト指向言語では，アスペクトの拡張をアスペクトで記述することは困難である．たとえば汎用的なアスペクト指向言語として知られる AspectJ[8] では，アドバイスそのものは無名メソッドとなっていて，そこをジョインポイントとしてポイントカットすることはできない．それゆえ，現状ではアスペクト抜きで実装し直す方が実用上適切である．

アスペクトとして実装された新しい機能は，そのソフトウェアから抜き差しされることが容易になる．実装された新機能のうち，ユーザからの評価が低いものは取り除かれなければならないが，アスペクトとして機能がプロトタイプ実装されていれば，そのアスペクトを取り除くだけでよい．アスペクトの実装や抜き差しの際に，既存のプログラムに手を加える必要はない．また，機能を追加するために必要な既存プログラムのリファクタリングも，アスペクト内で記述する．それゆえ既存のアジャイル開発と異なり，提案手法では，将来取り除かれる機能のための既存プログラムのリファクタリングを，そのプログラムに行う必要がない．さらに，アスペクトを用いれば，複数の機能を互いに独立に実装することもできる．互いに独立なアスペクトであれば，既存プログラムへ並列に機能を追加していくことが可能である．

また，新機能がアスペクトとして実装されていれば，その中に含まれるバグの発見にも役立つ．機能を追加したことによってソフトウェア全体に問題が発生したならば，少なくともそのアスペクトを加えたことによってバグが生じたことが分かる．開発者はアスペクトやそのポイントカット箇所をチェックしバグを探すことができる．一方，アスペクトを利用せずに複数の既存モジュールを書き変えて機能を追加した場合，問題の発生時に漫然とデバッグすることがしばしばある．これは，機能を追加するための修正が複数の既存プログラムに散らばっているため，開発者がその修正箇所を把握しきれなくなるからである．

2.2 Remote GluonJ

アスペクト指向を利用すればアジャイルなソフトウェア開発の機能追加やリファクタリングの負担を減らすことができるが，開発対象が分散ソフトウェアの場合，既存のアスペクト指向言語ではうまく対処できない．これは，追加機能やリファクタリングが分散した横断的関心事を含むからである．そのような開発では，非分散のときよりもリファクタリングの負担が大きいにもかかわらず，既存のアスペクト指向言語ではリファクタリング等の負担を十分に減らすことができない．

既存のアスペクト指向言語でも，分散した横断的関心事を含む追加機能等を，既存の分散プログラムから分離して記述することはできる．しかし，追加機能等を構成する各ホスト上のプログラムを，各ホスト上で動作するアスペクトとしてそれぞれ実装しなければならない．各反復の追加機能やリファクタリングのコードがこのようなアスペクトで実装されている場合，その機能を削除するには，各ホスト上の既存プログラムからそれぞれのアスペクトを取り外さなければならない．またバグが生じた場合，複数ホスト上のアスペクトをチェックしなければならず，その分バグの特定が困難になる．

上記の問題を解決するため，本稿では提案手法を

用いた分散ソフトウェア開発を支援する環境として，*Remote GluonJ* を開発した．Remote GluonJ は，アスペクト指向言語とその実行時システムであり，以下のような特徴を持つように開発された．

（a）分散化した横断的関心事の分離

Remote GluonJ は，分散した横断的関心事を 1 つのホスト上で動作する単一のアスペクトとして実装するための言語機構を提供する．提案手法では，各反復で追加する機能やリファクタリングのためのコードをアスペクトとして実装する．Remote GluonJ を利用することにより，開発者は複数ホスト上で動作する既存プログラムに散らばる追加機能等を，単一のアスペクトとして実装できる．各反復で追加機能のための修正箇所が 1 つのアスペクトとして分離されていれば，追加機能を削除するのに，そのアスペクトを分散ソフトウェアから取り外すだけでよい．また，機能を追加したことで分散ソフトウェアに問題が生じた場合，そのアスペクトの実装やポイントカットされている箇所をチェックしてバグを探すことができる．

（b）分散ソフトウェアの再起動の制御

Remote GluonJ は，その上で動作する分散ソフトウェアを明示的に再起動することなく，Remote GluonJ のアスペクトを適用できる実行時システムを提供する．分散ソフトウェアの開発時には，開発中のソフトウェアの動作をチェックするために，分散ソフトウェアを頻繁に再起動する．提案手法の各反復では，追加する機能をアスペクトとして実装するため，アスペクトを何度も編集して分散ソフトウェアの動作をチェックする．Remote GluonJ の実行時システムを利用することにより，アスペクトを変更した際に開発者が明示的に分散ソフトウェアを再起動することなく，その変更を分散ソフトウェアに適用することができる．

（c）既存の統合開発環境の利用

Remote GluonJ のアスペクトとその他の分散ソフトウェアは，Eclipse[6] や NetBeans[11] 等の既存の Java 統合開発環境を利用して開発できる．Remote GluonJ を現行の開発現場に導入できるようにするためには，アスペクト指向言語でも，既存の統合開発環境と親和性が高いものが望ましい．Remote GluonJ のアスペクトは，既存の統合開発環境を拡張することなく，その統合開発環境が提供するエディタで開発することができる．特に，統合開発環境のコード支援を受けることができる．

3. Remote GluonJ のプログラミング

Remote GluonJ の言語機構を利用することにより，分散ソフトウェアの開発者は異なるホスト上の複数のプログラムに散らばる関心事を，既存のプログラムに手を加えることなく，分散を意識しない単一のホスト上で動作するアスペクトとして実装できる．この言語機構は，2.2 節の（a）の特徴を実現する．また，Remote GluonJ の実行時システムは，開発者がその言語機構で記述されたアスペクトを修正し再コンパイルすることで，分散ソフトウェアを自動的に再起動する．この実行時システムは，2.2 節の（b）の特徴を実現する．本章では，Remote GluonJ の言語機構について詳述する．Remote GluonJ のアスペクトやその他のプログラムを動作させる実行時システムの特徴については，4 章で説明する．

既存の Java 統合開発環境を利用して Remote GluonJ のプログラムを記述できるようにするため，Remote GluonJ の言語機構は，Java の注釈を利用して設計されている．この言語設計は，2.2 節の（c）の特徴を実現する．注釈は Java の標準の文法であるため，それにより記述されたアスペクトは，Java の標準的なコンパイラでコンパイルされることができる．また，既存の Java IDE が提供するエディタで編集することもできる．開発者は，Remote GluonJ 専用の統合開発環境を利用したり，既存の Java 開発環境を拡張したりする必要はない．

3.1 遠隔ポイントカット

遠隔ホスト上で動作しているプログラム内のジョインポイントを指定するために，Remote GluonJ は**遠隔ポイントカット**を提供する[12]．この遠隔ポイントカットを用いると，プログラムの実行が遠隔ポイントカットとして指定されたジョインポイントに到達したとき，ジョインポイントが存在するホストとは異なるホスト上に存在するアドバイスを，Remote GluonJ の実行時システムが実行する．デフォルトでは，アドバイスはアスペクトが配置されているホスト上の実行時システムによって実行される．汎用的なアスペクト指向言語の多くは，アスペクトが配置されたホスト上で実行されるジョインポイントのみポイントカットとして指定できる．そして，アドバイスはポイントカットされたジョインポイントと同一のホスト上で実行される．

以下に，Remote GluonJ のアスペクトの例を示す[⋆1]．

⋆1 ここではスペースの関係上，プログラム内のアクセス修飾子は省略されている．本稿の残りで例示されているプログラムについても同様である．

```
@Glue class Logger {
  @Before("{ Logger.log(‘call setX()‘); }")
  Pointcut pc = Pcd.call("Point#setX(int)");
  static void log(String msg) {
    System.out.println(msg);
  }
}
```

上記のアスペクトとその他のプログラムとを一緒に Remote GluonJ の実行時システム上で動作させると，任意のホスト上で Point オブジェクトの setX メソッドが呼び出されたとき，ログメッセージが表示される[*1]．@Glue 注釈の付いたクラス Logger が，Remote GluonJ のアスペクトである．Logger は通常の Java クラスであり，標準の Java コンパイラでコンパイルすることができる．@Glue クラス内に宣言された Pointcut 型のフィールドとその注釈が，ポイントカットとアドバイスのペアを表す．このペアを**ポイントカット・フィールド**と呼ぶ．ポイントカット・フィールドは，アスペクト内に複数個宣言することができる．

ポイントカット・フィールド pc の初期値：

```
Pcd.call("Point#setX(int)")
```

は，任意のホスト上のプログラム実行中の Point オブジェクトの setX メソッド呼び出し，というジョインポイントをポイントカットとして指定する．Pointcut および Pcd は，Remote GluonJ が提供するライブラリである．

ポイントカット・フィールド pc の宣言には，アドバイスを表す注釈が付けられる．@Before 注釈が付けられると，指定されたジョインポイントの実行の直前で，注釈の引数に文字列として記述されたコードブロックを Remote GluonJ の実行時システムは実行する[*2]．ポイントカット・フィールドには，@Before，@After もしくは @Around 注釈のいずれか 1 つが付けられる．これらの注釈は，多くのアスペクト指向言語が提供している before，after，もしくは around アドバイスに相当する．

⋆1 Remote GluonJ の言語設計の概念により，Remote GluonJ はアスペクトのインスタンスを管理しない[5]．それゆえ，アスペクトを表すクラスには static メソッドやフィールドのみ宣言すべきである．

⋆2 Remote GluonJ は，注釈の引数の中で使われているバック・クウォートを，エスケープされたダブル・クウォートとして処理する．アドバイスのコードブロックは既存の統合開発環境の支援は受けられず，この内のエラーは，プログラムのロード時に開発者に通知される．これは現状の Remote GluonJ の限界である．現在，この問題を解決するため，我々は Remote GluonJ のアドバイスを AspectJ5 [1] のように，通常の Java のメソッドとして表現するように設計し直している．これにより，アドバイス内のコードは通常の Java コンパイラでコンパイルでき，統合開発環境の支援を受けることができるようになる．

3.1.1 ポイントカット指定子

ポイントカットを宣言するために，多くのアスペクト指向言語と同様，Remote GluonJ はポイントカット指定子を提供する．Remote GluonJ では，ポイントカット指定子としてファクトリ・メソッドを提供する．現在のバージョンの Remote GluonJ が提供するポイントカット指定子は，先述した call メソッド以外に，execution，set，get，within，そして host メソッドである．

Remote GluonJ 特有のポイントカット指定子である host メソッドは，利用者が指定したホスト上のジョインポイントを抽出する．Remote GluonJ の各ポイントカット指定子は，デフォルトで，任意のホスト上のジョインポイントを指定するが，このポイントカット指定子は特定のホスト上のジョインポイントを指定するために利用される．たとえば，Remote GluonJ の利用者は host メソッドを，次のように利用できる：

```
@Glue class Logger {
  @Before("{ Logger.log(‘call setX()‘); }")
  Pointcut pc = Pcd.host("hostId")
      .and.call("Point#setX(int)");
}
```

このポイントカット宣言は，hostId という名前で指定したホスト上の Point オブジェクトの setX メソッド呼び出しをジョインポイントとして指定する．hostId は，Remote GluonJ の実行時システムを各ホスト上で起動する際に，利用者が実行時パラメータとして与えることのできる実行時システムの名前である．異なるホスト上で動作する Remote GluonJ の実行時システムには，異なる名前を与えなければならない．これら実行時パラメータを活用することによって，利用者はソースコードに固定されたホスト名を記述することを避けられる．

3.1.2 ポイントカット引数

Remote GluonJ の利用者は，ポイントカットによって指定したジョインポイントの実行時コンテキストをポイントカットの引数に束縛し，アドバイス・コード内で利用できる．遠隔ポイントカットでは，そのポイントカットはアドバイスが実行されるホストとは異なる遠隔ホスト上のジョインポイントを指定するため，ポイントカット引数は遠隔ホスト上のデータを参照する．

Remote GluonJ では，ポイントカット引数を define メソッドで宣言する．以下に，define を利用する例を示す．

```
@Glue class Logger {
  @Before("{ System.out.println(x); }")
  Pointcut pc = Pcd.define("int", "x", "$1")
```

```
                .call("Point#setX(int)");
}
```

define メソッドの第 2 引数 x は，ポイントカット引数の名前である．第 1 引数 int は x の型を表す．第 3 引数 $1 は遠隔ホスト上で実行される setX メソッドの第 1 引数に x が束縛されることを表する．$2，$3 の場合はメソッドの第 2，第 3 引数が束縛される．プログラム実行が指定されるジョインポイントに到達したとき，ポイントカット引数に束縛された実行時コンテキストは，直列化され，ネットワーク越しにアドバイスのあるホストへ送信される．その後，直列化されたコンテキストは復元され，アドバイス・コードの中で実行される．

3.1.3 @Local アドバイスと @On アドバイス

Remote GluonJ のアドバイス・コードは，様々なホスト上で実行される．分散ソフトウェア内の横断的関心事が，いつも複数のホスト間に分散しているとは限らない．横断的関心事が 1 つのホスト上で完結している場合，対応するジョインポイントと同一のホスト上でアドバイス・コードを実行した方がよい．無駄なネットワーク通信によるオーバヘッドを避けることができる．

Remote GluonJ ではポイントカット・フィールドに @Local 注釈を付けることで，ポイントカットとして指定したジョインポイントが動作するホスト上でアドバイス・コードを実行できる．たとえば，ポイントカット・フィールドを以下のように宣言する．

```
@Glue class Logger {
  @Local
  @Before("{ System.out.println(‘setX‘); }")
  Pointcut pc = Pcd.call("Point#setX(int)");
}
```

上記のポイントカット・フィールドにより，遠隔ホスト上で実行された Point オブジェクトの setX メソッド呼び出しの直前で，指定されたジョインポイントと同じホスト上でメッセージが出力される．

また，ポイントカット・フィールドに @On 注釈を付けて，選択したホスト上でアドバイスを実行できる．以下に，@On 注釈を利用したポイントカット・フィールドの例を示す．

```
@Glue class Logger {
  @On("hostId")
  @Before("{ System.out.println(‘setX‘); }")
  Pointcut pc = Pcd.call("Point#setX(int)");
}
```

このポイントカット・フィールドにより，遠隔ホスト上で実行された setX メソッド呼び出しの直前で，hostId という名前で指定されたホスト上でアドバイス・コードが実行される．hostId は，Remote GluonJ の実行時システムの名前である．

3.2 遠隔クラス改良

遠隔ホスト上の既存のクラス定義を直接変更するために，Remote GluonJ は**遠隔クラス改良**を提供する．遠隔クラス改良では，どのように既存のクラス定義を変更するかを，元クラスのサブクラスとして記述する．そのサブクラスには @Refine 注釈が付けられる．

3.2.1 既存のメソッドの変更

既存のクラスに定義されたメソッドを変更するために，利用者は遠隔クラス改良を表現するクラスに，その変更内容を記述する．たとえば，アスペクト

```
@Glue class Logger {
  @Refine static class PointLogger
                extends Point {
    @Override void setX(int x) {
      System.out.println("x = " + x);
      super.setX(x);
    }
  }
}
```

は，任意のホスト上の既存 Point クラスの setX メソッドの定義を PointLogger に記述される setX メソッドに直接修正する．アスペクト内に宣言された PointLogger が遠隔クラス改良を表現するクラスである．遠隔クラス改良は，アスペクト内に static なネスト・クラスとして宣言されなければならない．上記の遠隔クラス改良によって，任意のホスト上で Point オブジェクトの setX メソッドが呼び出されると，代入された値がディスプレイに出力される．Java のサブクラスと異なり，Point オブジェクトの振舞いを変更するため，Point のサブクラスのオブジェクトを明示的に生成する必要はない．

また，遠隔クラス改良を表現するクラス内のメソッドは，既存のクラスに宣言されるメソッドを呼び出すために，super を利用できる．この super のセマンティクスは，Java のサブクラスと同様である．また，Remote GluonJ は既存のクラスに宣言される private メソッドを，遠隔クラス改良内で呼び出すための機構も提供する．さらに，既存クラスの private メソッドの定義も修正できる．しかしながら，本稿ではそれらについて言及しない．

標準の Java のコンパイラや既存の統合開発環境は，この遠隔クラス改良を表現するクラスを Java の通常のサブクラスと見なす．それゆえ，Remote GluonJ の利用者はそのアスペクトを記述する際に，通常の Java のクラスと同様に統合開発環境のコード支援を活用で

きる．この遠隔クラス改良の言語設計は，2.2 節の（c）の特徴を実現する．たとえば，利用者が統合開発環境のエディタで PointLogger を開発するとき，setX のメソッドの中で super. とタイプすると，元のクラスに宣言されたメソッドやフィールドの一覧が表示される．また，利用者は PointLogger クラスの setX メソッドに @Override 注釈を付けられる[⋆1]．もし変更しているメソッドの名前とシグネチャが既存のクラスのメソッドのものと異なれば，コンパイラや統合開発環境はエラーを表示する．

3.2.2 新しいメソッドやフィールドの追加

既存のクラスに新規メソッドやフィールドを追加するために，利用者は遠隔クラス改良を表現するクラスに，新しく追加するメソッドやフィールドを記述する．たとえば，以下の遠隔クラス改良 PointTracer は，任意のホスト上の Point クラスへ直接 trace メソッドと enabledTrace フィールドを追加する．

```
@Glue class Tracer {
  @Refine static class PointTracer
                extends Point {
    boolean enabledTrace = true;
    void trace(String msg) {
      if (enabledTrace)
        System.out.println(msg);
    }
  }
}
```

このように追加されたメソッドやフィールドは，同じアスペクト Tracer の中でのみ利用することができる．

新規メソッドやフィールドだけではなく，遠隔クラス改良は新しいインタフェースも既存のクラスに追加できる．インタフェースを追加するために，利用者は，そのインタフェースが遠隔クラス改良を表すクラスに実装されるように記述すればよい．

3.3 ロード時織り込み

コンパイルされたアスペクトは，Remote GluonJ の実行時システムによってロード時にその他のクラスに織り込まれる．コンパイルされたアスペクトは，まずそれが配置されたホスト上の実行時システムに蓄えられ，次に各ホスト上の実行時システムに自動で配布される．そして，それぞれのホスト上で動作する Remote GluonJ の拡張クラスローダが，ロード時に通常の Java クラスに，配布されたアスペクトの情報をバイトコード・レベルで織り込む．このバイトコード変換には Javassist [4] を利用する．

通常 Remote GluonJ のアドバイス・コードは，ロード時に define メソッドで指定されたポイントカット引数を仮引数とするメソッドに変換される．各ホスト上の拡張クラスローダは，遠隔ポイントカットとして指定されたジョインポイントを見つけると，そのジョインポイント箇所にアドバイスを表すメソッドを呼び出すためのコードを挿入する．ただし，@Local アドバイスとペアになるポイントカットで指定されたジョインポイント箇所には，アドバイス・コードが直接挿入される．

各実行時システムへ配布されるアスペクトの情報は，拡張クラスローダが織り込みに必要なもののみである．通常，遠隔ポイントカットによって指定されたジョインポイントの情報や，遠隔インタータイプ宣言の情報である．ただし，アスペクト内に @Local や @On 注釈の付いたアドバイスが宣言されていた場合，そのアドバイス・コードも指定されたホスト上へ配布される．

4. Remote GluonJ の実行時システム

4.1 ホット・デプロイメント機能

その上で動作する分散ソフトウェアの再起動を自動化するため，Remote GluonJ の実行時システムは，アスペクトの**ホット・デプロイメント**機能を提供する．ホット・デプロイメントとは，実行時システムをシャットダウンさせずに，その上で動作しているソフトウェアを自動的に再起動し，その振舞いを変更できる機能である．この機能を使えば，プログラムを編集し，その変更した振舞いをチェックするために，動作しているプログラムを再起動する開発者の手間がなくなる．また，実行時システムを再起動しない分，時間的コストの削減になる[⋆2]．

Remote GluonJ の実行時システムを起動する前に，利用者はコンパイルされたアスペクトやその他のプログラムを，Remote GluonJ の実行時システムによって指定されたディレクトリに配置する．このディレクトリ以下に置かれたアスペクトが変更されると，その Remote GluonJ の実行時システムは，分散ソフトウェアを自動的に再起動する．つまり，すべてのホストの実行時システムは，その上で動作しているプログラムを自動的に再起動する．その再起動時に，各ホスト上

⋆1 @Override は標準の Java API で提供されている注釈である．この注釈はサブクラス内で既存のメソッドを上書きするメソッドに付けられる．

⋆2 現在の Remote GluonJ の実行時システムの起動時間は長くはない．しかし我々は，アスペクトのホット・デプロイメント機能を既存の J2EE サーバに実装することを目標の 1 つとしている．既存の J2EE サーバは，その上で動作する分散ソフトウェアのために，様々なサービスやライブラリを提供する．それらのサービスやライブラリの数が増えるのにともない，J2EE サーバの起動に時間がかかるようになる．

の実行時システムは，更新されたアスペクトの内容に従って，プログラムの振舞いをロード時に変更する．

Remote GluonJ では，分散ソフトウェアに適用される新規のアスペクトが指定されたディレクトリに置かれても，そのアスペクトはホット・デプロイメントの対象になる．実行時システムは新規追加されたアスペクトに従って，その分散ソフトウェアの振舞いを変更する．同様に，すでに分散ソフトウェアに適用されているアスペクトのクラスファイルが指定されたディレクトリから削除されると，実行時システムは分散ソフトウェアを自動的に再起動し，そのアスペクトが適用されていない状態に戻す．

4.2 ホット・デプロイメント機能の実装

Remote GluonJ の実行時システムは実行中，指定したディレクトリ以下のクラスファイルを定期的に監視する．実行時システムは，そのディレクトリ以下にあるアスペクトのクラスファイルの名前と最終更新日時をチェックする．もしアスペクトのクラスファイルの最終更新日時が変更されると，実行時システムはそのアスペクトの情報を取得し，その情報を他のホスト上で動作している実行時システムに配布する．アスペクトの情報を受け取った実行時システムは，現在その上で動作しているプログラムをロードしたクラスローダとは別のクラスローダを新規作成し，既存のプログラムを再度ロードする．このロード時に，配布されたアスペクトの情報に従い，クラスローダはプログラムのバイトコードを編集する．

NFS でファイルシステムが共有された複数のホスト上でも，Remote GluonJ はアジャイルな分散ソフトウェア開発を支援できる．開発中の分散ソフトウェアのクラスファイルや設定ファイル等をホスト間で共有できるため，NFS は分散ソフトウェア開発にしばしば利用されてきた．NFS でファイルシステムが共有されたホスト間では，開発者が変更したアスペクトの実装を共有できる．このような環境の下で Remote GluonJ を使うには，アスペクト情報の各実行時システムへの配布機能をオフにする．これは実行時システムの起動オプションで設定する．このオプションをオフにすることで，指定されたディレクトリ以下のアスペクトの最終更新日時が変更されても，そのアスペクトの情報が他のホストの実行時システムに渡されることはない．変更されたアスペクトの情報に従い，各ホスト上の拡張クラスローダは，分散ソフトウェアを変更する[⋆1]．

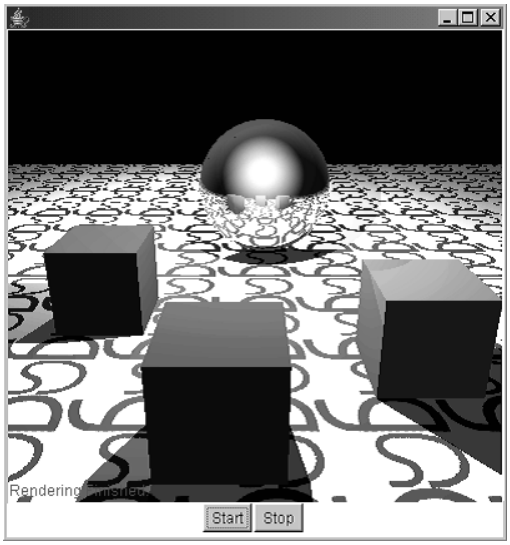

図 1 分散レイトレーシング・アプリケーションの GUI ウィンドウ
Fig. 1 GUI window of distributed raytracing program.

5. アプリケーション

提案手法のケーススタディとして，我々は分散レイトレーシング・アプリケーションの開発を実際に行った（**図 1**）．本章では，その開発時のある反復で，ユーザからのフィードバックにより追加した機能を，アスペクトとして実装した例を紹介する．その後，実装したアスペクトについて考察する．

開発した分散レイトレーシング・アプリケーションは，大まかにクライアント・ノード上で動作する GUI プログラムと，色情報をレイトレーシング法で計算するプログラムから構成される．色情報の計算の負荷を分散させるため，計算プログラムは複数の計算ノード上で動作する．各計算プログラムは，画像の指定された領域のみの色情報を計算する．

5.1 前回までの反復で開発されたプログラム

直前までの反復では，描画する物体の位置座標や計算する画像の領域等を受け取り，その領域の色情報を計算し，後で集計できるようにその計算結果をファイルに書き込むプログラムが開発されていた．この計算プログラムは，各計算ノード上で動作する．また，クライアント・ノード上で動作するプログラムは，物体のワイヤフレームが描画されている GUI ウィンドウを表示できる．GUI ウィンドウ上の物体のワイヤフレー

⋆1 ただし，このような環境の下では，アドバイスが実行されるホストを，@Local や @On 等の注釈を利用して明示的に指定するべきである．@Local や @On の付いたアドバイスについては，3.1.3 項で説明した．

ムを操作することで，ユーザは物体の位置座標を変更することができる．そのウィンドウ上のスタートボタンをマウスでクリックすると，各計算プログラムに物体のデータ等を送信し，画像の色情報を計算させる．

各計算ノード上で動作するプログラム Raytracer は，GUI クライアントからの計算開始のメッセージを待つ．GUI クライアントからウィンドウのサイズ，その中に描かれる物体のサイズ，位置座標等を受け取ると，そのプログラムは指定された領域の色情報の計算を開始する．計算プログラムは，その領域内の色情報を 1 ドットずつ計算し，1 ドットの計算を終えるたびに writeOnePixcel というメソッドを呼び出し，その色情報をファイルに書き込む．

5.2 アスペクトによる機能の実装

紹介する反復では，ユーザからのフィードバックを受け以下の 2 つの機能を追加することになった．

- 各計算ノードでの色情報の計算の途中結果を，リアルタイムに GUI クライアント上で表示する機能
- 各計算ノード上で色情報が計算された後，クライアント・ノード上に計算結果を集め，画像ファイルを生成する機能

これらの機能はともに分散した横断的関心事を含む．色情報をクライアント・ノードの GUI ウィンドウに表示するという関心事は，計算プログラムとクライアント・プログラムの 2 つを横断する．さらに，それぞれのプログラムは別々のノード上で動作する．クライアント・ノード上で，色情報を利用して画像ファイルを作成するという関心事も同様である．

我々は Remote GluonJ を使ったので，それぞれの機能をクライアント・ノード上で動作する単一のアスペクトとして実装できた．リアルタイムに 1 ドットずつ計算される色情報を GUI クライアント上で表示する機能をアスペクトで実装した例を以下に示す．

```
@Glue class RealtimeDrawing {
  @Before("{RealtimeDrawing.draw(x, y, c);}")
  Pointcut pc = Pcd.define("int", "x", "$1")
       .define("int", "y", "$2")
       .define("java.awt.Color", "c", "$3")
       .call("Raytracer#writeOnePixel(..)");

  static void draw(int x, int y, Color c) {
    synchronized (GUIWindow.g) {
      GUIWindow.g.setColor(c);
      GUIWindow.g.drawLine(x, y, x, y);
  }}
}
```

RealtimeDrawing アスペクトは，クライアント・ノードの Remote GluonJ の実行時システム上で動作する．Raytracer クラスの writeOnePixel メソッドの呼び出しを遠隔ポイントカットし，それに対応する before アドバイスでアスペクト内の draw メソッドを呼び出している．draw は，クライアント・プログラムのワイヤフレームを表示している GUI ウィンドウへ計算された色情報を描画する．また，各計算ノード上で色情報の計算が終了した後，クライアント・ノード上に画像ファイルを作成する機能も，クライアント・ノードの Remote GluonJ の実行時システム上で動作するアスペクトとして実装できた．

5.3 アスペクトの改良

テストの結果，5.2 節の RealtimeDrawing アスペクトでは，GUI ウィンドウへの描画速度が遅くなることが判明した．1 ドットの色情報の計算が終了するたびに，GUI ウィンドウへその色情報を描画する実装では，ネットワーク通信にかかるオーバヘッドが大きいからである．

そこで，リアルタイムに計算の途中結果を表示する機能の実装を，1 行分の色情報をまとめて GUI ウィンドウへ描画する実装へ変更することにした．以下に改良した RealtimeDrawing アスペクトの実装を示す．

```
@Glue class RealtimeDrawing {
  @Refine static class AddCols
              extends Raytracer {
    Color[] cols;
    void hook(int y, Color[] colors) {}
    void calculateOneLine(int y) {
      cols = new Color[WINDOW_SIZE];
      super.calculateOneLine(g, y);
      hook(y, cols);
    }
    Color calculateOnePixel(int x, int y) {
      cols[x] = super.calculateOnePixel(x, y);
      return cols[x];
  }}

  @Before("{RealtimeDrawing.draw(cols, y);}")
  Pointcut pc = Pcd.define("int", "y", "$1")
    .define("java.awt.Color[]", "cols", "$2")
    .and.call("Raytracer#hook(..)");

  static void draw(Color[] cols, int y) {
    for (int i = 0; i < cols.length; i++) {
      synchronized (GUIWindow.g) {
        GUIWindow.g.setColor(cols[i]);
        GUIWindow.g.drawLine(i, y, i, y);
  }}}
}
```

遠隔クラス改良を使用し，GUI ウィンドウへ描画する 1 行分の色情報を保持する cols フィールドと，hook メソッドを Raytracer クラスへ追加する．また，hook メソッドを呼び出すように，1 行分の色情報を計算する calculateOneLine メソッドを上書きする．さ

らに，1 ドットの色情報を計算する calculateOnePixel メソッドを，計算された色情報を cols へ格納するように上書きする．既存の calculateOneLine() の中で，calculateOnePixel() は呼び出される．そして，calculateOneLine メソッドで呼び出される hook メソッドを遠隔ポイントカットする．hook メソッド呼び出しの引数に渡された 1 行分の色情報 cols を，ポイントカット引数に束縛し，クライアント・ホスト上で実行されるアドバイス内でその色情報を GUI ウィンドウへ描画する．

5.4 考　　察

Remote GluonJ を利用したため，提案手法を用いた分散レイトレーシング・アプリケーションの開発は困難なく行えた．

5.2 節で追加した 2 つの機能はそれぞれ，クライアント・ノードの実行時システム上で動作する単一のアスペクトとして実装できた．それぞれのアスペクトは前回までの反復のプログラムに手を加えることなく，互いに独立に開発することができた．それゆえ，仮にユーザの評価によって，これらの機能を削除することになっても，開発者は削除される機能を実装するアスペクトを取り除くだけでよい．その際に，既存のアプリケーションを修正する必要はない．

5.3 節の改良されたアスペクトにあるように，リアルタイムに計算結果を表示する機能を実装するには，Raytracer クラスに cols フィールドを追加する等の設計変更が必要だった．しかし Remote GluonJ を使えば，この Raytracer クラスの設計変更を遠隔クラス改良を用いて，既存プログラムに手を加えることなく行えた．それゆえ，もう片方の機能はこの設計変更を考慮せずに実装できた．また，遠隔クラス改良による既存プログラムの設計変更であれば，アスペクトを取り除くことでその設計の変更を元に戻すことができる．

さらに，2 つの機能はアスペクトとして互いに独立しているため，機能の実装の改良もそれぞれのアスペクトにのみ着目して行えた．Remote GluonJ の実行時システムのホット・デプロイメント機能を使えば，RealtimeDrawing の改良を反映するのに，分散ソフトウェアを明示的に再起動する必要はなかった．再コンパイルした RealtimeDrawing のクラスファイルを，Remote GluonJ の実行時システムによって指定されたディレクトリへ配置するだけでよい．各ホスト上の実行時システムは，その上で動作しているプログラムを自動的に再起動する．

今回の開発で実装されたアスペクトやその他のプログラムは，すべて既存の Java の統合開発環境である Eclipse 上で実装することができた．既存のアプリケーションやアスペクトを修正する際に，特別な開発環境や Eclipse プラグイン等を利用する必要はなかった．

RealtimeDrawing アスペクトで指定したジョインポイントは，Raytracer クラスの writeOnePixel，calculateOneLine，calculateOnePixel，そして hook メソッドの 4 つである．hook 以外の 3 つのメソッドは，遠隔ポイントカットもしくは遠隔クラス改良で指定されることを見越して事前に定義しておいたメソッドではない．Raytracer クラスのモジュール性をある程度意識して設計すれば，これらは宣言されるはずのメソッドである．一方 hook は，遠隔ポイントカットとして指定するべき適切なジョインポイントが Raytracer に見つからなかったため，遠隔クラス改良を用いてアスペクトのために後から追加したメソッドである．アジャイルな分散ソフトウェア開発では，事前にソフトウェア全体の設計を把握しにくいので，将来ポイントカットするかもしれないジョインポイントを事前に用意しにくい．このような場合でも遠隔クラス改良を利用すれば，ポイントカットとして適切なジョインポイントを任意のホスト上のクラスに追加することができる．

6. 関 連 研 究

我々は以前，分散した横断的関心事をモジュール化するために遠隔ポイントカットの言語機構を提案し，それを提供する言語処理系のプロトタイプ実装を行った[12]．しかし，先の研究で開発した言語処理系だけでは，分散ソフトウェアのアジャイル開発を十分に支援できない．本研究と先の研究の差分は，アジャイルな分散ソフトウェア開発手法を提案し，その手法を利用して分散ソフトウェアを開発するために必要な環境を開発したことである．さらに本稿では，提案手法と Remote GluonJ の有用性を示すために，実際に分散レイトレーシング・アプリケーションの開発を行った．

AWED[10] や JAC[13] は，分散した横断的関心事を分離するために，遠隔ポイントカットの概念に基づいた言語機構を提供する．また，AWED は JAsCo[7] の実行時システムを流用しているため，動的にアスペクトとその他のプログラムとを織り込むことができる．しかし，Remote GluonJ の実行時システムと異なり，AWED の実行時システムはホット・デプロイメント機能を提供していない．それゆえ，アスペクトを何度も修正しながら分散ソフトウェアを開発するときに不向きである．

CaesarJ[2] は汎用的なアスペクト指向言語であるが，CaesarJ のアスペクトを遠隔ホスト上に配布する実行

時システムを持つ．CaesarJ は遠隔ホスト上のジョインポイントを指定でき，そのホスト上でアドバイスを実行できる．これは，Remote GluonJ の @Local アドバイスに相当する．また，遠隔クラス改良に相当する機能も持つ．しかし，ポイントカットとして指定されたホストとは，異なるホスト上でアドバイスを実行できない．それゆえ，5 章で例示した分散レイトレーシング・アプリケーションの追加機能をアスペクトとして実装しようとしても，CaesarJ では単一のアスペクトとして記述しきれない．

cJVM[3]，Addistant[14]，J-Orchestra[15] は，ネットワーク越しに結合された複数のホスト上に，1 つの Java 仮想機械を実現する．これらのシステムを利用すると，非分散ソフトウェアとした開発された元のプログラムを，自動的に分散実行できる．汎用的な既存のアスペクト指向言語を用いて記述された非分散版のプログラムを，これらのシステム上で分散実行すれば，遠隔ポイントカットを使って書かれたプログラムと同等の結果が得られる．

しかしながら，分散ソフトウェア開発の現状では，この方法は適切ではない．この方法では，分散ソフトウェア自体も，cJVM 等の上に（非分散プログラムとして）構築されている必要がある．しかし，現実の web アプリケーション・プログラム等は，J2EE サーバや DI コンテナ等の上に構築されており，アスペクトだけではなく分散ソフトウェア自体も cJVM 等の上に再構築するのは非現実的である．一方，Remote GluonJ にはそのような実行環境に関する制限がない．Remote GluonJ は織り込みのために，専用のクラスローダを必要とするが，これを既存の実行環境に組み込むことは，実用上，十分可能である．

分散処理はよく知られた横断的関心事であり，これらの関心事を分離するためいくつかのシステムが提案されてきた．たとえば，D 言語[9] により，開発者は遠隔メソッドのパラメータをどのように受け渡すかを元プログラムから分離して記述できる．これらのシステムの目的は分散処理の特定の横断的関心事を調査し，それらを分離することである．一方 Remote GluonJ は，分散ソフトウェア用の汎用的なアスペクト指向言語を，Java 言語の注釈を利用して提案した．

7. 議　　論

本稿では，アスペクト指向を利用したアジャイルな分散ソフトウェア開発手法を提案し，それを支援するための環境 Remote GluonJ を開発した．提案手法では，各反復時に新しく追加する機能をアスペクトとして実装する．この手法を用いることで，各追加機能の既存プログラムからの抜き差しが容易になる．追加する機能を実装するために，既存プログラムを編集する必要はない．

Remote GluonJ の遠隔ポイントカットと遠隔クラス改良を用いることで，利用者は分散した横断的関心事を，他のプログラムに手を加えることなく，1 つのホスト上で動作する単一のアスペクトとして実装することができる（2.2 節の（a））．また，Remote GluonJ の実行時システムはホット・デプロイメント機能を提供している．これにより，アスペクトを分散ソフトウェアに適用するために，利用者は分散ソフトウェアを明示的に再起動する必要はない（2.2 節の（b））．Remote GluonJ の実行時システムによって指定されたディレクトリにコンパイルしたアスペクトを配置することで，Remote GluonJ の実行時システムが自動的に分散ソフトウェアを再起動する．さらに，Remote GluonJ の言語機構は Java の注釈で適切に設計されているため，その利用者は既存の Java 統合開発環境のコード支援を受けられ，その上で開発することができる（2.2 節の（c））．

さらにケーススタディとして，我々は提案手法および Remote GluonJ を用い，分散レイトレーシング・アプリケーションを実際に開発した．

参 考 文 献

1) The AspectJ 5 Development Kit Developer's Notebook. http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html
2) Aracic, I., Gasiunas, V., Mezini, M. and Ostermann, K.: An Overview of CaesarJ, *Transactions on Aspect-Oriented Software Development I*, Rashid, A. and Aksit, M. (Eds.), Vol.3880 of Lecture Notes in Computer Science, pp.135–173, Springer (2006).
3) Aridor, Y., Factor, M. and Teperman, A.: cJVM: A single system image of a JVM on a cluster, *Proc. International Conference on Parallel Processing* (*ICPP '99*), pp.4–11 (1999).
4) Chiba, S.: Load-time structural reflection in java, *Proc. 14th European Conference on Object-Oriented Programming* (*ECOOP '00*), pp.313–336, Springer-Verlag, London, UK (2000).
5) Chiba, S. and Ishikawa, R.: Aspect-Oriented Programming Beyond Dependency Injection, *Proc. 19th European Conference on Object-Oriented Programming* (*ECOOP '05*), Black, A.P. (Ed.), Vol.3586 of Lecture Notes in Com-

puter Science, pp.121–143, Springer Berlin/ Heidelberg (2005).
6) eclipse. http://www.eclipse.org/. eclipse
7) JAsCo community. http://ssel.vub.ac.be/jasco/. JAsCo
8) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An overview of AspectJ, *Proc. 15th European Conference on Object-Oriented Programming* (*ECOOP '01*), Vol.2072 of Lecture Notes in Computer Science, pp.327–355, Springer-Verlag (2001).
9) Lopes, D.C.: A Language Framework for Distributed Programming, Ph.D. thesis, College of Computer Science, Northeastern University (Dec. 1997).
10) Navarro, L.D.B., Südholt, M., Vanderperren, W., Fraine, B.D. and Suvée, D.: Explicitly distributed aop using awed, *Proc. 5th International Conference on Aspect-Oriented Software Development* (*AOSD '06*), pp.51–62, ACM Press, New York, NY, USA (2006).
11) NetBeans community. http://www.netbeans.org. NetBeans
12) Nishizawa, M., Chiba, S. and Tatsubori, M.: Remote pointcut: A language construct for distributed aop, *Proc. 3rd International Conference on Aspect-Oriented Software Development* (*AOSD '04*), pp.7–15, ACM Press, New York, NY, USA (2004).
13) Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F., and Martelli, L.: Jac: An aspect-based distributed dynamic framework, *Software Practise and Experience* (*SPE*), Vol.34, No.12, pp.1119–1148 (2004).
14) Tatsubori, M., Sasaki, T., Chiba, S. and Itano, K.: A bytecode translator for distributed execution of legacy java software, *Proc. 15th European Conference on Object-Oriented Programming* (*ECOOP '01*), Vol.2072 of Lecture Notes in Computer Science, pp.236–255, Springer-Verlag (2001).
15) Tilevich, E. and Smaragdakis, Y.: J-orchestra: Automatic java application partitioning, *Proc. 16th European Conference on Object-Oriented Programming* (*ECOOP '02*), pp.178–204, Springer, Malaga, Spain (2002).

(平成 19 年 7 月 9 日受付)
(平成 19 年 10 月 9 日採録)

西澤　無我

1979 年生．2002 年東京工業大学第一類情報科学科卒業．2002 年より同大学院数理・計算科学専攻．2004 年同大学院理学修士課程修了．現在，同大学院情報理工学研究科在学中．プログラミング言語，分散システム，システムソフトウェアに関する研究に従事．

栗田　洋輔

1983 年生．2006 年東京工業大学第一類情報科学科卒業．現在，同大学院数理・計算科学専攻在学中．計算機アーキテクチャ，プログラミング言語に関する研究に従事．

千葉　　滋（正会員）

東京工業大学大学院情報理工学研究科准教授．1991 年東京大学理学部情報科学科卒業．1993 年同大学院理学系研究科情報科学専攻修士課程修了．1996 年同専攻より理学博士．東京大学助手，筑波大学講師，東京工業大学講師を経て現職．プログラミング言語およびオペレーティング・システム等，システムソフトウェアの開発に興味を持っている．