プログラムスライシングに基づく関心事グラフ構築

亀田大輔[†] 滝本宗宏[††]

現在，ソフトウェアの生産性向上のための手法として，アスペクト指向プログラミング（AOP）が注目されている．AOP を導入する方法には，設計段階で導入する方法以外に，既存のプログラムに対してリファクタリングを行う方法がある．本研究では，オブジェクト指向プログラムからアスペクトを抽出するリファクタリングを支援するシステムを提案する．AOP を導入するリファクタリング手法には，関心事グラフを用いる手法がある．関心事グラフは，特定の横断的関心事に関連するプログラムの要素であるクラス，メソッド，フィールドを表す節と，それぞれの関係をラベルとする辺によって構成されるグラフである．ユーザは，関心事グラフを基に，リファクタリングを行うことで，見通し良く AOP を導入することができる．しかし，従来，関心事グラフの作成は，手動で行われており，関心事グラフの作成者は，プログラム全体を詳細に把握したうえで，関連する要素を判断する必要がある．本支援システムは，プログラムスライシングを用いることによって，各節間の情報を取り出し，関心事グラフの作成を半自動化する．また，スライシングの際，型情報や文脈情報を利用することで，さらに，関心事グラフの精度を上げることができる．

Building Concern Graph Based on Program Slicing

Daisuke Kameda[†] and Munehiro Takimoto[††]

Aspect oriented programming (AOP) makes it possible to modularize scattered concern code of a system. Such modularizing can be achieved by not only designing based on AOP but also refactoring existing programs. We propose the refactoring system which supports extracting crosscutting concerns of a system as aspects. Such aspects can easily be detected using the concern graph representation, which abstracts the implementation details of a concern and makes explicit the relation-ships between different parts of the concern. The abstraction used in a concern graph can be inexpensively and obviously mapped to corresponding source code. However, concern graph is manually built, so software developers must check whole a program in detail to build it. Our system semi-automatically generates the concern graph including a specific concern using program slicing technique. Since program slicing can extract the parts of a program that affect the values computed at some point of interest as a program slice, it enables primer concern graph to be easily refined by combining developer's knowledge acquired on the primer concern graph with slice for relevant points based on the knowledge.

1. はじめに

従来のソフトウェア開発のアプローチは，対象を機能的な視点に基づいて分割する．これは，構造化プログラミングやオブジェクト指向プログラミング（以降 **OOP**）において共通の考え方である．しかし，たとえば OOP では，システムの中心機能でない処理である，ログの記述，メッセージの通知，同期処理，セキュリティのチェックなどを 1 つのコンポーネントに局所化できず，複数のコンポーネントを横断してしまう場合があった．

これは，対象を分割するために 1 つの視点しか提供していないために生ずる問題であり，コンポーネント間の独立性を低下させる要因である．

以上の問題を解決するために，**アスペクト指向プログラミング**（以降 **AOP**）[1] が提案されている．AOP では，**横断的関心事**という複数のコンポーネントに散らばって存在する処理を，**アスペクト**として抽出することで，コンポーネント間の独立性を向上させる．その結果，プログラムの再利用性，メンテナンス性，拡張性が改善されるので，ソフトウェア開発の生産性を向上させられるものと期待されている．

AOP を用いてソフトウェアを実装する方法は，大きく分けて，アスペクト指向言語を用いる方法とフレームワークを用いる方法がある．アスペクト指向言

† エスエムジー株式会社
SMG Company, Limited
†† 東京理科大学情報科学科
Department of Information Science, Tokyo University of Science

語としては，AspectJ[2] や Hyper/J[3] などの開発が行われている．また，フレームワークとしては，AspectWerkz[4] や JAC[5] などの開発が行われている．

AOP に基づくソフトウェア開発（以降 **AOSD**）を実現するためには，このような実装方法だけでは不十分であり，AOP の導入方法の確立が必要である．AOP の導入方法としては，新規開発のソフトウェアを対象とした設計段階での導入と，既存のソフトウェアを対象としたリファクタリングによるアスペクトの抽出が考えられる．

AOP は上記のようにソフトウェアの再利用性，メンテナンス性，拡張性を向上させることが可能なので，既存の大きなソフトウェアに対して導入することができれば大きな効果が期待できる．そこで本研究では，リファクタリングによるアスペクトの導入を扱う．

AOP を導入するためのリファクタリングツールとしては，JQuery[6]，AMT[7]，FEAT[8] などが研究されている．このうち，FEAT が用いている**関心事グラフ**（Concern Graph）[9] は，次の点で優れている．

- アスペクトに関連するコードを網羅的に扱うことができる．
- プログラムの内容に基づきアスペクトを抽出することができる．

しかしながら，その作成の大部分を手動で行わなければならないという欠点を持っている．そこで本研究では，プログラムスライシングを用いることで，関心事グラフの作成を半自動化する手法を提案する．

プログラムスライシングは，制御依存関係とデータ依存関係に基づき，プログラムから，ある命令と関連する意味のまとまりを抽出することができるので，その結果を用いて，関心事グラフの作成コストを低減させることができる．しかし，OOP においては，多相性の存在やメソッド呼び出しの頻発によって，精密なスライシングが難しいので，型情報やメソッド呼び出しが実際に処理に影響を与えているかどうかの情報（以降**文脈情報**）を用いてこのグラフを最小化させる．

以降，2 章で，リファクタリングによる AOP の導入方法と関心事グラフによる手法について述べる．3 章で，プログラムスライシングについて述べる．4 章で，本研究の手法について述べ，5 章で，本手法による関心事グラフ作成の具体例を述べる．6 章で，実装と評価について述べ，最後に，7 章で，まとめと今後の課題を述べる．

2. リファクタリングによる AOP の導入

2.1 リファクタリング

リファクタリング[10] とは，ソフトウェアの外部から見た振舞いを変更せずに，理解しやすく，拡張や保守が容易になるように，内部構造を変更することである．リファクタリングを用いて AOP を導入する際には，外部的な振舞いを変更せずに，コンポーネントからアスペクトを抽出する必要がある．その際，問題となるのが，複数のコンポーネントにまたがって存在しているアスペクトをどのように発見するかという点である．

従来，アスペクトを発見するためのツールとして，以下のようなものが研究されている．

- JQuery
- AMT
- FEAT

このうち，JQuery と AMT は，指定した文字列，継承関係，メソッドの呼び出し関係などに一致するコードを検索することで，アスペクトを発見しようというアプローチである．したがって，事前にどの部分がアスペクトであるか見当がついている必要がある．また，プログラムの構造を考慮せずにアスペクトの発見を行うので，不適切なコードが含まれたり，必要なコードが欠落したりする可能性が高い．

これに対し，FEAT では，特定の関心事に関連するプログラムの要素をグラフ化した関心事グラフを作成しながらアスペクトの発見を行う．この方法によって，事前にどの部分がアスペクトか見当がつかない場合でも，プログラムの意味に基づいたアスペクトの発見が可能である．

2.2 関心事グラフ

関心事グラフは，特定の横断的関心事に関連するプログラムの要素であるクラス，メソッド，フィールドを表す節と，各要素間の関係をラベルとする辺によって構成されるグラフである．ラベルとしては，以下のようなものがある．ここで，(label a b) は a から b への辺を表す．

- (**calls** m_1 m_2)：メソッド m_1 において，メソッド m_2 を呼び出していることを表す．
- (**reads** m f)：メソッド m において，フィールド f の値を参照していることを表す．
- (**writes** m f)：メソッド m において，フィールド f に値を代入していることを表す．
- (**checks** m c)：メソッド m において，クラス c へキャスト，あるいは，型がクラス c であるか

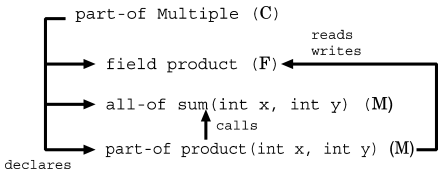


図 1 関心事グラフの例
Fig. 1 An example of Concern Graph.

チェックをしていることを表す.

- (**creates** m c)：メソッド m において，クラス c のインスタンスを生成していることを表す.
- (**declares** c $f|m$)：クラス c において，メソッド m，あるいは，フィールド f を宣言していることを表す.
- (**superclass** c_1 c_2)：クラス c_2 がクラス c_1 のスーパークラスであることを表す.

上記の辺を用いた関心事グラフの例を**図 1** に示す. ここで，C はクラス，M はメソッド，F はフィールドを表す．また，クラス名やメソッド名の前についている all-of と part-of という接頭辞は，そのクラスやメソッドの全体がアスペクトに含まれるのか，あるいは一部が含まれるのかを示す．図 1 の part-of Multiple は，クラス Multiple の一部分がアスペクトに含まれることを示している．また，all-of sum と part-of product は，メソッド sum() のすべてのコードがアスペクトに含まれ，メソッド product() の一部分がアスペクトに含まれることをそれぞれ示している．一方，フィールドには，この接頭辞は必要ない.

このように，関心事グラフは，制御フローやデータフローなどのプログラムの細かい情報は含まない．この関心事グラフの特徴は，プログラムのどの部分がアスペクトであるかを容易に判断することに役立つ．また，情報量が少ないので，グラフの追加・削除を行うのも容易である.

関心事グラフの作成は，以下のような手順で行う.

(1) グラフ作成の開始点（seed）として，抽出したい関心事に含まれるクラス，メソッド，あるいは，フィールドを指定する.
(2) クエリを用いて，上記のラベルに対応するクラス，メソッド，フィールドのコードを参照し，同じ関心事に含まれる要素をグラフに含める.
(3) 新たにグラフに追加した要素に対して，(2) の処理を実行する．以降，新たにグラフに追加する要素がなくなるまで繰り返す.

以上の手順によって，プログラムの意味に基づいたアスペクト抽出を行うことができる．しかしながら，手順 (2) にあるように，関心事に関連する処理を行っているかどうかの判断は，すべて手動で行わなければならない.

本研究では，プログラムスライシングを用いて，この問題点を解決する.

3. プログラムスライシング

プログラムスライシング[11]（以降スライシング）は，あるプログラム点 p における変数 v（以降**スライス基準**）に影響を与えるプログラムの部分集合（以降**プログラムスライス**）を求める解析手法である.

スライス基準に影響を与えるかどうかを判断するためには，プログラムの各命令の依存関係を調べる必要がある．この依存は，**データ依存**と**制御依存**とに区別される．データ依存関係，および，制御依存関係は次のとおりである.

(1) データ依存：
命令 t が命令 s に対してデータ依存するとは，ある変数 w が存在して，命令 s における変数 w の定義が，変数 w を使用している命令 t に到達することをいう.
(2) 制御依存：
命令 t が命令 s に対して制御依存するとは，命令 s が分岐命令であり，命令 t がその分岐文内に直接含まれている場合か，あるいは，命令 s がループ命令であり，命令 t がそのループ文内に直接含まれる場合をいう.

各節が各命令を表し，依存する関係にある節どうしを有向辺で結んだグラフを**プログラム依存グラフ**（以降 **PDG**[12]）という．また，複数の PDG を有向辺でつないだグラフを，**システム依存グラフ**（以降 **SDG**[13]）という．SDG は，PDG どうしが，手続き呼び出しを表す辺によって結ばれ，手続き間で受け渡されるパラメータを表す節が付加された構造を持つ.

スライス基準を起点として，SDG のデータ依存辺と制御依存辺を逆にたどることによって到達する命令集合は，**後向きのプログラムスライス**と呼ばれる．また，順方向にたどることによって到達する命令集合は，**前向きのプログラムスライス**と呼ばれる．ここで，後向きのプログラムスライスは，スライス基準に影響を与える命令の集合を意味し，一方，前向きのプログラムスライスは，スライス基準の処理が影響を与える命令の集合を意味する.

4. 本手法による関心事グラフの作成

関心事グラフとは，特定の関心事に関連するプログ

ラムの要素とその関係を示したグラフである．また，後向きのプログラムスライスと前向きのプログラムスライスを用いることで，スライス基準の命令に関連する処理を抽出することができる．スライシングによって得られるプログラム片は，同一の関心事に関連しており，そのプログラム片からプログラムの要素であるクラス，メソッド，フィールドとそれらの関係を抽出することで，関心事グラフを作成することができる．

以降で，まずスライシングについて，型情報と文脈情報を利用する方法を説明する．次に，関心事グラフを作成するための具体的な手順を説明する．

4.1 スライシング

本手法におけるスライシングは，以下の手順で行う．

(1) 各メソッドにおいて，PDG を作成する．

(2) 各クラスと，各メソッドの本体および各フィールドの宣言との間に，**宣言関係の辺**を付加する．また，クラスの継承関係として，子クラスから親クラスへの**継承関係の辺**を付加する．フィールドの参照，および変更している命令についても，メソッドからフィールドへの**フィールドの参照辺**，および**フィールドの変更辺**をそれぞれ付加する．動的な型チェックやキャストを行っている場合には，メソッドからクラスへの**型検証の辺**を付加する．

(3) スライス基準を求めるために，各 PDG を呼び出し関係によって接続し，SDG を作成する．その際に，型情報を利用して呼び出しの候補を最小にする[14]．また，文脈情報を利用して，メソッド呼び出し自体を最小にする．この際，型情報を利用して，**インスタンス生成関係の辺**も作成する．

(4) スライス基準から後向きと前向きのプログラムスライスを求める．

関心事グラフの辺のうちで，(2) と (3) で追加する辺は，プログラムスライスから直接求めることができないので，別に付加している．型情報と文脈情報については，あとでそれぞれ詳しく説明する．

本手法のグラフ作成の開始点は，従来研究の関心事グラフと扱いが異なり，全体がアスペクトに含まれる (all-of) として扱う．そして，開始点全体をスライス基準と見なす．クラスを指定した場合は，インスタンス生成を含む文，メソッド呼び出し，フィールド参照のすべてをスライス基準に追加する．メソッドを指定した場合は，メソッド呼び出しすべてをスライス基準に追加する．フィールドを指定した場合は，すべてのフィールド参照をスライス基準として追加する．

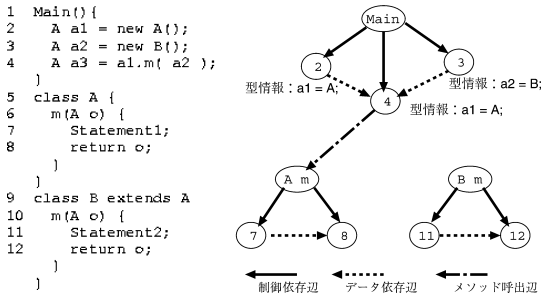


図 2 型情報の例
Fig. 2 Sample of type identification.

4.1.1 型情報の利用

呼び出されるメソッドの候補は，オブジェクト指向における多相性によって複数存在する可能性がある．

そこで型情報を用いて，呼び出すメソッドの候補（接続する PDG の候補）を少なくし，PDG 間の辺の数を減らす．

たとえば，**図 2** 左のプログラムにおいて，図 2 右の節 4 に対応する行番号 4 が多相性を持つメソッド呼び出しである．PDG をつなげ SDG を作成する際に，呼び出しているメソッドは，クラス A，クラス B のどちらのメソッドなのかを判断するためには，変数 $a1$ の型情報が必要である．そこで，変数 $a1$ についてのデータ依存辺を逆向きにたどって，変数 $a1$ の型情報を得る．結果として，行番号 2 から，$a1$ の値の型が A であることが分かる．

また，メソッドの返値にデータ依存している処理がある場合，メソッドの返値の型情報を利用する．

4.1.2 文脈情報の利用

文脈情報とは，以下の 3 つの情報のことである．

- メソッドの引数が内部で変更されているか，参照だけされているかという情報
- メソッド内部でフィールドが変更されているか，参照だけされているかという情報

メソッド呼び出しは，返値によってばかりではなく，内部での引数やフィールドの変更によっても，のちの処理に影響を与える．すなわち，文脈情報を利用することで，より厳密な依存関係に基づいたスライシングを行うことができる．本手法では，以下の条件のいずれかにあてはまる場合にだけ，そのメソッド本体とメソッド呼び出しを辺でつなげる．

- メソッドの引数が内部で変更されているか，変更されているかどうかが不明な場合
- フィールドがメソッド内部で変更されているか，変更されているかどうかが不明な場合

また，各メソッドの引数が内部で参照されているだ

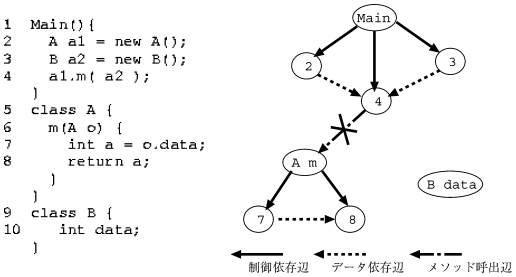


図 3 文脈情報の例
Fig. 3 Sample of context.

表 1 SDG の辺と関心事グラフの辺の対応リスト
Table 1 Maping list of SDG edge to Concern Graph edge.

本手法で作成した SDG の辺	関心事グラフの辺
メソッド呼び出し辺	(calls m_1 m_2)
フィールドの参照辺	(reads m f)
フィールドの変更辺	(writes m f)
型検証の辺	(checks m c)
インスタンス生成関係の辺	(creates m c)
宣言関係の辺	(declares c f|m)
継承関係の辺	(superclass c_1 c_2)

けの場合には，その実引数からのデータ依存辺を生成しない．フィールドが 1 つも変更されていない場合には，呼び出し変数からのデータ依存辺を削除する．

ここで，メソッドの引数についての例を示す．**図 3** 左のようなプログラムにおいて，図 3 右のような PDG が作成できたとする．このとき，行番号 4 のメソッド m の呼び出しは，引数が変数 $a2$ とデータ依存関係を持つので，メソッド Main() の PDG と，クラス A のメソッド m の PDG とをつなげる必要がある．しかしながら，メソッド m で，変数 $a2$ に対応する仮引数は参照されているだけなので，図 3 右の×印のように，行番号 4 のメソッド呼び出しは，メソッド m の本体とつなげない．

4.2 関心事グラフの作成

本手法では，プログラムスライスを用いて関心事グラフを作成する際，プログラムスライスを求める過程で新たに付加した辺を利用する．これらの各辺と，関心事グラフの辺との対応関係を，**表 1** に示す．

実際のプログラムスライスから関心事グラフを作成する手順は次のとおりである．

(1) プログラムスライスに含まれている要素を節として関心事グラフに加える．
(2) (1) で追加したクラスとメソッドについて，すべてがプログラムスライスに含まれている場合は all-of，一部だけが含まれている場合は part-of の接頭辞を付ける．

表 2 関心事グラフの辺を作成する条件
Table 2 Requirement of making Concern Graph edge.

関心事グラフの辺	条件
(calls m_1 m_2)	プログラムスライスにメソッド m_1 からメソッド m_2 へのメソッド呼び出しの辺が含まれる場合
(reads m f)	プログラムスライスにメソッド m におけるフィールド f の参照が含まれる場合
(writes m f)	プログラムスライスにメソッド m におけるフィールド f の変更が含まれる場合
(checks m c)	プログラムスライスにメソッド m におけるクラス c についての動的型チェック，あるいは，クラス c へのキャストが含まれる場合
(creates m c)	プログラムスライスにメソッド m におけるクラス c のインスタンス生成の処理が含まれる場合
(declares c f|m)	クラス c で宣言しているフィールド f やメソッド m がプログラムスライス，あるいは，関心事グラフに含まれる場合
(superclass c_1 c_2)	継承関係にあるクラス c_1 と c_2 が両方とも関心事グラフにすでに含まれている場合

表 3 従来法と本手法の比較
Table 3 Comparison of the conventional method and this method.

	従来法	本手法
開始点の指定	手動	手動
グラフに含まれる要素の判断	手動	自動＋手動で精錬
開始点の接頭辞	指定	つねに all-of

(3) 表 1 に基づいて，関心事グラフにおける各要素間に辺を付加する．ただし，辺を付加できるのは，**表 2** の条件を満たす場合だけである．

また，SDG に新たに付加した辺に対応する辺を関心事グラフに追加した際に，辺のどちか一方の要素が関心事グラフに含まれていなかったならば，その要素も関心事グラフに追加する．

以上が本手法による関心事グラフの作成手順である．従来法による作成方法と本手法による作成方法の比較を**表 3** に示す．

5. 関心事グラフの作成例

本章では，実際にどのように関心事グラフを作成するのかを，**図 4** のプログラムを用いて説明する．ここで，開始点を ErrorLogging クラスの writeLog メソッドとする．

5.1 スライシング

まず，それぞれのメソッドについて PDG を作成し

```
1  class Logging {
2    public writeLog(String logMessage){
3      System.out.println("log: " + logMessage);
     }
   }
4  class ErrorLogging extends Logging {
5    public writeLog(String logMessage) {
6      System.out.println("Error: " + logMessage);
     }
   }
7  class MethodChecker {
8    Logging logging;
9    public isThereMethod(Object obj, String methodName) {
10     logging = new Logging();
11     Class classObject = obj.getClass();
12     boolean isThere = checkMethod(classObject, methodName);
13     if( isThere == false ) {
14       logging = new ErrorLogging();
15       logging.writeLog( classObject.getName() + "."
                         + methodName + "() is not fount.");
       }
16     logging.writeLog( result );
17     return isThere;
     }
   }
```

図 4 サンプルコード
Fig. 4 Sample code.

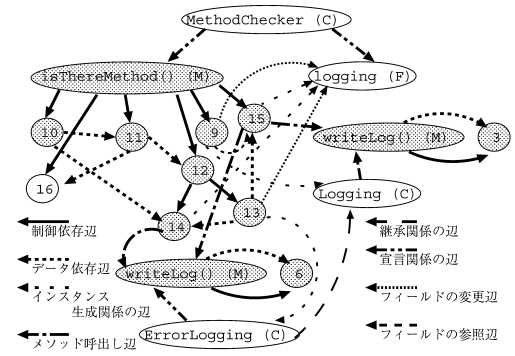


図 5 スライシングの例
Fig. 5 Example of slicing.

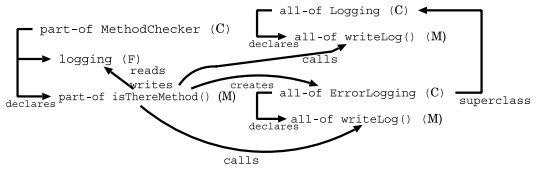


図 6 関心事グラフ変換の例
Fig. 6 Example of converting slice to Concern Graph.

たのち，宣言関係と継承関係に対応する辺を付加する．具体的には，Logging クラスの節は writeLog メソッドの節に，ErrorLogging クラスの節は writeLog メソッドの節に，MethodChecker クラスの節は logging フィールドと isThereMethod メソッドの節に，それぞれ宣言関係の辺を付加する．また，ErrorLogging クラスの節から Logging クラスの節に継承関係の辺を付加する．MethodChecker クラスの logging フィールドの参照と変更についても，行番号 10，14 の節から logging フィールドの節にそれぞれ参照関係の辺，変更関係の辺を付加する．さらに，スライス基準に対応するメソッド呼び出しを求めるために，メソッド本体とメソッド呼び出しをつなぐ必要があるので，メソッド呼び出しをすべて記録しておく．

次に，スライス基準を求めるために，PDG 間の辺を付加する．その際に，型情報と文脈情報の計算を行う．行番号 15 の writeLog メソッドの呼び出しについては，行番号 14 の型情報を用いて，行番号 15 の節から ErrorLogging#writeLog メソッドの節にメソッド呼び出しの辺を付加する．また，行番号 16 の writeLog メソッドの呼び出しは，型情報が不明なので，行番号 16 の節から Logging#writeLog() と ErrorLogging#writeLog() の両方の節にメソッド呼び出しの辺を付加する．この際，型情報を用いて，インスタンス生成関係の辺も付加する．

そして，ErrorLogging#writeLog() が開始点であることから，そのメソッド本体をスライス基準とする．さらに行番号 15 と行番号 16 のメソッド呼び出しがスライス基準となる．スライス基準が決まると，スライス基準の各要素から，後向きのプログラムスライスと前向きのプログラムスライスを求める．ただし，宣言関係の辺，継承関係の辺，フィールド参照関係の辺，フィールド変更関係の辺は利用しない．

結果として，**図 5** の網掛けの節としてスライスを得る．

5.2 関心事グラフの作成

まず，MethodChecker クラスの isThereMethod メソッド，Logging クラスの writeLog メソッド，ErrorLogging クラスの writeLog メソッドがそれぞれ関心事グラフの要素になる．そして，それぞれのメソッドがすべてプログラムスライスに含まれているかどうかを調べ，isThereMethod メソッドには part-of を，2 つの writeLog メソッドには all-of の接頭辞を付ける．

次に，SDG に含まれるメソッド呼び出し辺，フィールド参照辺，フィールド変更辺，インスタンス生成辺，宣言関係辺をそれぞれ関心事グラフの辺に置き換える．その結果，calls 辺が 2 つ，declares 辺が 3 つ，reads 辺が 1 つ，writes 辺が 1 つ，creates 辺が 1 つ作成される．

そして，追加された辺の一方の要素がない場合には，関心事グラフにその要素を追加する．図 4 の例では，MethodChecker クラスから logging フィールドへの declares 辺と ErrorLogging クラスから Logging クラスへの superclass 辺を追加する．

最終的に，**図 6** の関心事グラフを得る．

6. 評　　価

本手法の有効性を評価するために，本手法を，統合開発環境 Eclipse [16] のプラグインとして実装した．

Eclipse は，さまざまなプラグインを統合するための小規模なランタイムカーネルと，多くのプラグインが提供する機能を実現している．開発者は，Eclipse の中心的機能を構成しているプラグインが提供する拡張ポイントを用いて，提供されている機能を拡張するプラグインを開発することができる．本実装では，その拡張ポイントを用いて，グラフ作成の開始点を指定する機能を実現している．

また，Java 開発ツール（以降 **JDT**）という Java 開発のための支援ツールがプラグインとして提供されており，Java の抽象構文木を処理するためのパッケージも存在している．本実装では，JDT が提供する Visitor[17] を用いて，抽象構文木の各節を処理することで，PDG の作成と SDG に新たに追加する辺の付加を行う．SDG の作成では，作成した PDG のメソッド呼び出しとメソッド本体をあらかじめリストアップしておき，型情報と文脈情報を利用して必要なものだけを辺でつなぐ．

本手法が自動的に作成した関心事グラフは，スライシングを用いているので，意味的に欠落する要素がなく，AOP の専門家が手動で作成した場合の関心事グラフを含んだものになると考えられる．このことから，本手法で作成する関心事グラフの利用法は，次の 2 つが考えられる．

(1) 本手法で作成した関心事グラフを基にリファクタリングを行う．
(2) 本手法で作成した関心事グラフを修正して，より精密な関心事グラフを作成し，その後リファクタリングを行う．

利用法 (1) で，本手法によって作成された関心事グラフの精度が従来法によって得られるグラフに比べて不十分である場合でも，以下の条件を満たしていれば，利用法 (2) によって段階的に精密化することができると考えられる．

(1) 本手法によって作成される関心事グラフの要素数が，対象となるプログラムに含まれる要素数に比べて大幅に少ない．
(2) 従来法によって作成されるグラフに含まれない要素を，容易に削除することができる．

本手法によって作成される関心事グラフが以上の条件を満たしているかどうか，実際に使用されているアプリケーションプログラムに対して評価を行った．本評価では，IRC クライアントの EIRC[18] とコンパイラインフラストラクチャの COINS[19] を対象とする．

6.1 EIRC

IRC クライアントは，そのほとんどの機能をサーバ，クライアント間のメッセージの送受信を基に実現している．そのために，通常メッセージの送受信に関する処理が複数のコンポーネントに散らばって存在する．本評価では，メッセージ関連の処理に対応する関心事グラフの抽出を目的とし，開始点として，Message クラスを指定してグラフの作成を行った．その結果得られた関心事グラフの要素数を**表 4** に示す．

表 4 EIRC における関心事グラフの要素の数
Table 4 The number of the elements of the Concerns Graph in EIRC.

	クラス	メソッド	フィールド
プログラムの全要素数	66	759	443
関心事グラフの要素数	20	66	66
すべてが含まれる要素数	2	27	

関心事グラフとして抽出されたプログラムの要素は，元のプログラムの要素の約 12%であった．また，関心事グラフとして抽出されたクラスにおいて，すべてが関心事グラフに含まれるものは，開始点である Message クラスと MircMessage だけであった．すなわち，従来のすべて手動で行う方法に比べて，アスペクトとして判断する対象を絞り込むことができると考える．そして，接頭辞に着目することで，絞り込みはさらに容易になる．

また，本手法によって作成した関心事グラフは，そのグラフの形状を確認できるので，Message や MircMessage が直接関連する要素であるかどうかを判断することは容易であると考える．この点について，以下で，依存に基づく要素のまとまり方の点から分析する．

Message クラスや MircMessage クラス内のメソッド以外から，両クラスのメソッドを呼び出しているのは，EIRC#processMessage メソッド，EIRC#sendMessage メソッド，ServerThread#enqueueMessage メソッド，TextAttributePicker#actionPerformed メソッドだけである．そして，EIRC#processMessage メソッドだけが Message クラスの 4 つのメソッドを呼び出しており，その他は各 1 つずつであった．

Message クラスおよび MircMessage クラスからフィールドの変更や参照を行っているものの大部分は各クラス内部のものであり，そのほかに，StatusWindow クラスの msg フィールド，SimpleAppletStub クラスの parameters フィールド，ChannelItem クラスの t フィールドを参照しており，ChannelItem クラスの t フィールドについては変更も行っている．また，EIRC#sendMessage メソッドで MircMessage クラスのインスタンスが生成され，Server-

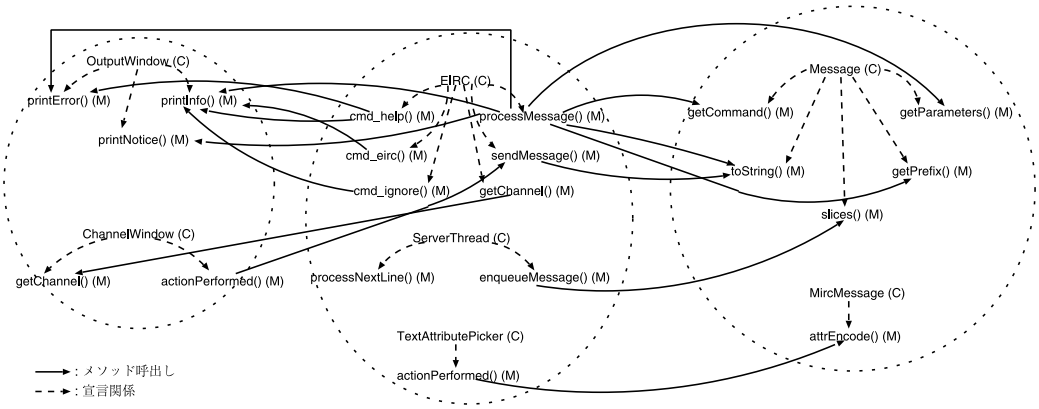


図 7 本手法で作成したメッセージについての関心事グラフ
Fig. 7 Concern Graph about Message created by our technique.

Thread#processNextLine メソッドでは，Message クラスと MircMessage クラスのインスタンスが生成されている．

一方，EIRC クラスからは，ChannnelWindow クラスと OutputWindow クラスのメソッドを多く呼び出している．それに対して，StatusWindow クラス，SimplAppletStub クラス，ChannelItem クラスの要素に対する呼び出しは，Message クラスおよび MircMessage クラスからのものしか存在しなかった．

以上の依存関係を表す関心事グラフの一部を**図 7** に示す．図 7 の関心事グラフにおいて辺による結合の仕方に着目すると，EIRC クラスを中心として，互いに疎な結合しか持たない，Message クラスおよび MircMessage クラスに関連する要素と，ChannelWindow クラスおよび OutputWindow クラスに関連する要素に分けて考えることができる．このような構造のグラフが表示されている場合，そこから Message クラスと MircMessage クラスに関連する要素を抽出することは容易であると考えられる．実際に従来手法と比較した結果については，次節で述べる．

また，この結果は，Message クラスや MircMessage クラスのようなデータに関する関心事と ChannelWindow クラスや OutputWindow クラスのような GUI に関する関心事という，複数の関心事が関連した関心事グラフの例でもある．この場合も，相互に関連する要素は相対的に疎な結合となっているので，両者を切り離すことは容易と考えられる．しかし，モジュール化が十分に行われていない場合には，関心事が密接に関連してしまい，分割が難しくなる可能性もある．

6.2 COINS

コンパイラは，四則演算，論理演算，代入，関数呼び出しなどの文法規則に関する処理と，型チェック，シンボルテーブル，最適化，目的コードへの変換などの解釈や変換に関する処理の 2 つに分割した場合，両者は直交することが知られている[20),21)]．

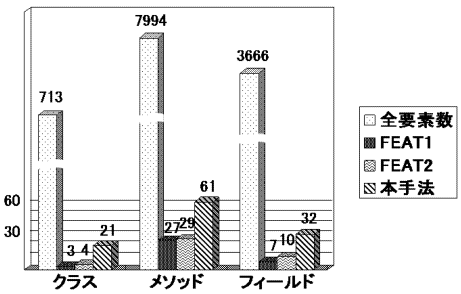


図 8 FEAT と本手法を比較したグラフ
Fig. 8 Graph which compared FEAT with this method.

本評価で対象とする COINS は，研究者向けのコンパイラ共通基盤であり，容易に新しい最適化手法や言語のパーサなどを実装することができるようになっている．COINS には，C 言語の解析を対象としたパッケージ coins.cfront パッケージがあり，ここには C 言語のシンボルテーブルを扱う SymbolTable クラスが存在する．本評価では，この SymbolTable クラスを開始点として，シンボルテーブルに関する処理を関心事グラフとして抽出した．また，COINS で開発を行っている 2 人の学生に協力してもらい，FEAT を用いて SymbolTable クラスを開始点とした関心事グラフを作成した．結果を**図 8** に示す．

ここで，対象としたプログラムに含まれるすべての要素数を全要素数，FEAT を用いて作成した 2 つの関心事グラフに含まれる要素数をそれぞれ FEAT1 と FEAT2 とした．

図 8 のグラフが示すように，COINS に含まれるすべての要素数に比べて，本手法で抽出された要素数は

```
EncodedType.insert(byte[])
Parser#topFuncDecl(EncodedType,Declarator,
                   CompoundStmnt)
```

本手法とFEAT1の結果にだけ含まれたメソッド

```
Declarator.getArgs()
Declarator.getName()
Declarator.getType()
SymbolTable$Entry.<init>(Object,Object,
                         SymbolTable$Entry)
```

本手法とFEAT2の結果にだけ含まれたメソッド

図 9 本手法と FEAT の一方の結果にだけ含まれたメソッド
Fig. 9 The method contained only in this technique and one side of FEAT.

```
EncodedType#clear()
EncodedType#get()
EncodedType#getTagName()
EncodedType#getTypeChar()
EncodedType#getTypeChar(byte[],int)
EncodedType#getTypeChar0(byte[],int)
EncodedType#increase()
EncodedType#insert(EncodedType)
EncodedType#insert(char)
EncodedType#isSigned()
EncodedType#toString()
Evaluator#make(double,EncodedType)
Evaluator#make(double,char)
Evaluator#make(long,EncodedType)
Evaluator#make(long,char,char)
Lex#getDouble()
```

```
Lex#getLong()
Lex#getString()
Lex#readIdentifier(int,Token)
Parser#compoundStatement(boolean)
Parser#declaratorList(EncodedType,int,boolean,
                      boolean,boolean)
Parser#decodeTagName(String)
Parser#initializeExprBrace(EncodedType)
Parser#initializeStruct(EncodedType,Declarator)
Parser#memberDeclarator(CompoundStmnt)
Parser#memberExpr(EncodedType,Expr,boolean)
Parser#signedOrUnsigend(char)
Parser#sizeofStruct(String)
Parser#structDecl(EncodedType,boolean,
                  CompoundStmnt)
Parser#toplevelDeclaration()
```

図 10 本手法にだけ含まれたメソッド
Fig. 10 The method contained only in this technique.

非常に小さくなっており，FEAT を用いた結果の 3 倍程度になった．

この理由を分析するために，FEAT を用いて作成した関心事グラフと比べて本手法を用いた結果が余分に含んでいる要素について調べた．メソッドについて比較した結果が，**図 9** と**図 10** である．また，本手法によって作成された関心事グラフに含まれず，FEAT による結果にだけ含まれる要素は存在しなかった．

図 9 において，本手法と FEAT1 の結果に含まれ，FEAT2 の結果に含まれなかったものは，EncodedType#insert メソッドと Parser#topFuncDecl メソッドだけである．このうち Parser#topFuncDecl メソッドは，先頭でシンボルテーブルからエントリの読み込みを行い，これを用いて処理をしている．すなわち，本来なら関心事グラフに含まれるべきであるが，誤って FEAT2 の結果には含まれなかったものと考えられる．

また，図 9 において，本手法と FEAT2 の結果に含まれ，FEAT1 の結果に含まれなかったものは，Declarator#getArgs メソッド，Declarator#getName メソッド，Declarator#getType メソッド，SymbolTable クラスの内部クラスである Entry クラスのコンストラクタである．このうち，Entry クラスのコンストラクタは，SymbolTable クラスの内部クラスであるために，本来は関心事グラフに含まれるべきであるが，誤って FEAT1 の結果に含まれなかったものと考えられる．

さらに，図 10 にあるメソッドの中でも，Evaluator クラスの make(double,char) メソッドと make(long,char,char) メソッドは，SymbolTable クラスの内部クラスである Entry クラスのフィールド value の参照を行っているので，誤って FEAT1 と FEAT2 の結果には含まれなかったものと考えられる．

以上の例から，本手法がスライシングを用いているために，すべて手動で行った場合に欠落しやすい対象を保持できると考えられる．

次に，図 9 と図 10 に書かれている言及していないメソッドについて分析する．

まず，図 9 と図 10 のメソッドは，先に述べたメソッド以外はすべて開始点である SymbolTable クラスの要素と直接結合していない．一方，本手法と FEAT による結果の両方に含まれているすべての要素は，SymbolTable のクラスの要素と直接結合している．

また，図 9 と図 10 のメソッドのうち，SymbolTable クラスの要素と直接結合している要素に対して辺を持つメソッドは，14 個である．

- EncodedType#clear()
- EncodedType#insert(char)
- Evaluator#make(double,EncodedType)
- Evaluator#make(double,char)
- Evaluator#make(long,char,char)
- Lex#getDouble()
- Lex#getLong()
- Lex#getString()
- Parser#compoundStatement(boolean)
- Parser#initializeExprBrace(EncodedType)

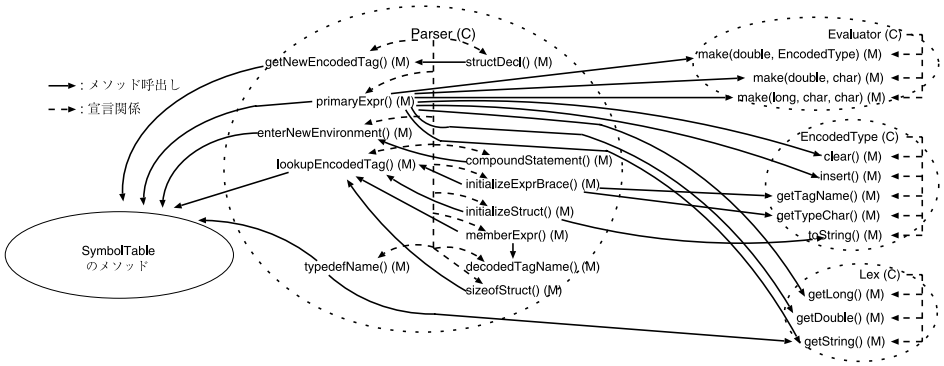


図 11 本手法で作成した SymbolTable についての関心事グラフ
Fig. 11 Concern Graph about SymbolTable created by this technique.

- Parser#initializeStruct(EncodedType, Declarator)
- Parser#memberExpr(EncodedType,Expr, boolean)
- Parser#sizeofStruct(String)
- Parser#structDecl(EncodedType,boolean, CompoundStmnt)

このうち，さらに複数の SymbolTable クラスの要素と直接結合している要素に対して辺を持つメソッドは，Lex#getString() だけである．また，その他の要素と結合する辺を持つメソッドは，

Parser#initializeExprBrace(EncodedType), Parser#initializeStruct(EncodedType,Declarator), Parser#memberExpr(EncodedType,Expr,boolean) の 3 つである．以上の依存関係を表す関心事グラフの一部を**図 11** に示す．

図 11 の関心事グラフから，Lex#getString() を除いたメソッドは，次の 2 点を判断基準として削除できるものと考える．

(1) 開始点である SymbolTable と直接結合する辺を持たない．
(2) SymbolTable に到達する辺のパスは 1 つだけである．

この基準は，前述の EIRC におけるまとまり方の基準同様開始点に対する結合が相対的に疎であることを基にしている．疎な結合は，表示されたグラフを参照しながら不要な部分を取り除く際の区切りの決定に，直感的な指針を与えるものと考える．

以上から，前述の利用法 (2) による段階的なグラフの精密化法を用いることで，従来のすべて手動で行う関心事グラフの作成方法より容易にアスペクトを抽出することができると考える．

次に，本手法と，本手法から型情報と文脈情報の利用を除いたものを比較した結果を**表 5** に示す．また，そのグラフを**図 12** に示す．

表 5 型情報と文脈情報の有効性の評価
Table 5 Evaluation of the validity of type identification and context.

	クラス	メソッド	フィールド
型情報&文脈情報あり	21	61	32
型情報&文脈情報なし	23	66	37

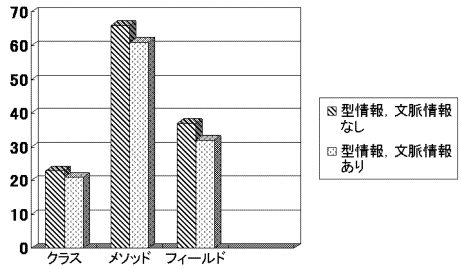


図 12 型情報と文脈情報の評価グラフ
Fig. 12 Evaluation graph of type identification and context.

型情報と文脈情報を使用しなかった場合に比べて，使用した場合は，プログラムのサイズが約 10%減少していた．この結果は，本手法による関心事グラフの作成において，型情報と文脈情報の利用が有効であることを示している．

最後に，本手法の解析範囲と深い関係にある別名とライブラリの扱いについて議論する．

引数の参照渡しや参照変数を介した間接参照が存在するプログラムは，異なる変数が同一のオブジェクトを指す可能性がある．このような同一のオブジェクトを指す変数は，**別名**と呼ばれる．本手法では，型情報の解析のほかに，次の基本的な別名解析を行っている．

(1) 参照変数どうしのコピー代入やフィールド値の代入によって生じる別名

(2) メソッドの引数の参照渡しによって生じる別名
別名であるかどうか不明な変数は，別名として扱う．

次に，本手法では，ソースコードだけを解析対象としており，フレームワークやライブラリの解析は行わない．すなわち，フレームワークやライブラリに含まれる要素は，メソッドの引数や返値を介して利用される可能性があるので，すべてプログラムスライスに含める．また，メソッドの呼び出しは，文脈情報の利用によって削除されることはない．

以上のように，本手法では，別名やライブラリについて**保守的**（conservative）な扱いをしているので，これらが使用されているプログラムにおいても，関心事グラフに必要な部分が欠落することはなく，利用法(2)の段階的なグラフの精密化法の対象として，必要な情報を含んでいる．しかしながら，より精密な別名解析とバイトコードを解析することによるライブラリの依存解析は，関心事グラフのさらなる精密化を実現する可能性がある．

7. ま と め

本稿では，スライシングを用いて，関心事グラフの作成を半自動化する手法を提案した．関心事グラフは，データとデータ，宣言と宣言，あるいは宣言とデータの間に存在する依存関係を表しており，ある関心事に含まれる節として最初に決める開始点が影響を及ぼすプログラムの要素と，影響を及ぼされるプログラムの要素によって構成される．

本手法では，この関心事グラフの性質を利用して，前向きのプログラムスライスと後向きのプログラムスライスを用いることによって得られたスライスの和集合から，関心事グラフを抽出する方法を実現した．その際，プログラムスライスから導くことのできない，宣言関係，参照関係，変更関係を新たな辺として付加した．さらに，スライシングを行う依存グラフ SDG を型情報と文脈情報とによって精密化し，最終的に得られる関心事グラフの精度を向上させた．

本手法の効果を示すために，本手法を，IRC クライアント EIRC に適用し，アスペクトになる可能性が高い関心事に含まれる要素を開始点として，関心事グラフの作成を行った．元のプログラムサイズの違いを比較したところ，元のプログラムの 12%に限定された関心事グラフが得られた．また，コンパイラ共通基盤 COINS に適用し，アスペクトになる可能性が高いシンボルテーブルを開始点として，関心事グラフの作成を行った．FEAT を用いた関心事グラフの作成結果と比較し，段階的な精密化によって，本手法が従来のすべて手動で作成する手法に比べて容易にアスペクトを抽出できることを示した．

今後，本手法を用いたリファクタリングツールの実現が重要であり，その際には，自動作成された関心事グラフをユーザの知識を用いて，容易に修正することができる機能を充実する必要がある．また，より精度を高めるために，より精密な別名解析やソースが存在しない部分のバイトコードの依存解析を採り入れることを検討する必要がある．

従来のリファクタリングツールのように事前条件を，スライシングと関心事グラフに対して定義することで，アスペクトを抽出するリファクタリングツールを作成することができるものと考える．

謝辞 貴重なコメントをいただいた，慶應義塾大学の原田賢一教授に心から感謝する．

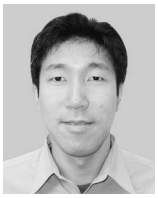
参 考 文 献

1) Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *Proc. European Conference on Object-Oriented Programming* (*ECOOP'97*), LNCS 1241, pp.220–242, Springer-Verlag (1997).
2) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: Getting Started with AspectJ, *Comm. ACM*, Vol.44, pp.59–65 (Oct. 2001).
3) Tarr, P., Ossher, H., Harrison, W. and Sutton, Jr. S.M.: N degrees of separation: multi-dimensional separation of concerns, *Proc. 21st International Conference on Software Engineering* (*ICSE*), May 16-22, pp.107–119 (1999).
4) http://aspectwerkz.codehaus.org/
5) Pawlak, R., Seinturier, L., Duchien, L. and Florin, G.: JAC: A Flexible Solution for Aspect-Oriented Programming in Java, *Proc. 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns* (*REFLECTION*), September 25-28, pp.1–24 (2001).
6) Janzen, D. and De Volder, K.: Navigating and querying code without getting lost, *Proc. 2nd International Conference on Aspect-Oriented Software Development* (*AOSD*), pp.178–187 (Mar. 2003).
7) Hannemann, J. and Kiczales, G.: Overcoming the Prevalent decomposition of Legacy Code, *Proc. Workshop on Advanced Separation of Concerns*, IEEE (2001).
8) Robillard, M.P. and Murphy, G.C.: FEAT: A Tool for Locating, Describing and Analyzing

Concerns in Source Code, *Proc. 25th International Conference on Software Engineering* (*ICSE*), pp.822–823 (May 2003).
9) Robillard, M.P. and Murphy, G.C.: Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies, *Proc. 24th International Conference on Software Engineering* (*ICSE*), pp.406–416 (May 2002).
10) Opdyke, W.F.: Refactoring Object-Orinted Frameworks, Technical Report, Ph.D. dissertation, University of Illinois (1992).
11) 下村隆夫：プログラムスライシング技術と応用，共立出版 (1995).
12) Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The program dependence graph and its use in optimization, *ACM Trans. Prog. Lang. Syst.* (July 1987).
13) Horwitz, S., Reps, T. and Binkley, D.: Interprocedural slicing using dependence graphs, *ACM Trans. Prog. Lang. Syst.* (Jan. 1990).
14) 酒井重之：型情報に基づくオブジェクト指向プログラムの効果的なスライシング，修士論文，東京理科大学 (2004).
15) Verbaere, M.: Program slicing for refactoring, MSc thesis, University of Oxford (2003). http://web.comlab.ox.ac.uk/oucl/research/areas/ progtools/projects/nate/doc/MScThesis-.pdf
16) http://www.eclipse.org/
17) Gamma, E., Helm, R., Johnson, R. and Vlisside, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).
18) http://eirc.sourceforge.net/
19) http://www.coins-project.org/
20) Appel, A.W.: *Modern Compiler Implementation in Java*, Cambridge University Press (1998).
21) De Moor, O., Jones, S.L.P. and Van Wyk, E.: Aspect-Oriented Compilers, *Generative and Component-Based Software Engineering* (*GCSE*), pp.121–133 (1999).
22) Whaley, J. and Rinard, M.C.: Compositional Pointer and Escape Analysis for Java Programs, *Proc. 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (*OOPSLA*), pp.187–206 (Nov. 1999).

(平成 16 年 12 月 22 日受付)
(平成 17 年 4 月 28 日採録)

亀田 大輔（正会員）
1978 年生．2005 年東京理科大学大学院理工学研究科情報科学専攻修士課程修了．現在，エスエムジー株式会社勤務．ACM 会員．



滝本 宗宏（正会員）
1994 年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了．現在，東京理科大学理工学部情報科学科講師．工学博士．プログラミング言語およびその処理系に興味を持つ．ACM，IEEE，日本ソフトウェア科学会各会員．