修士学位論文

題目

プログラム理解のための付加注釈 DocumentTag の提案と統合開発環境への統合

指導教員

井上 克郎 教授

報告者

田中 昌弘

平成 21 年 2 月 9 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻 ソフトウェア工学講座

平成 20 年度 修士学位論文

プログラム理解のための付加注釈 DocumentTag の提案と統合開発環境への統合

田中 昌弘

内容梗概

近年普及しつつあるオブジェクト指向プログラミングでは，カプセル化や継承・多相性などの機構を効果的に用いることで，プログラム理解やテスト，再利用，コーディングの効率を向上させることができる．

一方で，オブジェクト指向プログラミングを導入することにより，たとえばクラスやメソッドのような識別子間の関連が複雑かつ抽象的になり，プログラムを詳細に理解にすることが困難になるという問題が指摘されている．プログラム理解支援を目的としたツールや手法は多数提案されているが，開発者が理解した内容を記録・再利用する手段が効果的でないという課題が残されている．たとえばバグ修正を行う作業者は，プログラムの動作を分析した内容を，プログラム中のコメント，あるいはプログラムとは独立した文書に記述する．しかし，コメントとして記述する場合は，プログラム中に複数のバグ修正の内容が混在して蓄積されるため，プログラムの可読性が低下する．一方で独立した文書として記述する場合は，記述内容に対応しているプログラムの位置情報を把握することが難しい．

そこで本研究では，ソースコードの読解において理解した内容を効果的に記録するための付加注釈 DocumentTag を提案する．DocumentTag とは，識別子に対して記述する注釈であり，また，他の関連する識別子へと同一の注釈を関連付ける注釈伝播ルールを備えたものである．たとえば，あるクラスに対して，開発者がクラスの役割を DocumentTag として記述すると，そのクラスのサブクラスに対しても関連付けが行われ，開発者がサブクラスの注釈を参照する際に，関連付けられたスーパークラスの DocumentTag も同時に参照することができる．これにより開発者は，識別子間の関連を把握しながら効果的なプログラムの理解を行うことができる．本研究では，DocumentTag を統合開発環境 Eclipse の拡張機能として実装した．適用実験として，Java 言語で書かれた既存のプログラムに対して保守作業を行い，DocumentTag の注釈伝播の有効性を評価した．その結果，注釈伝播によって平均作業時間が短縮された．また，被験者に対して行ったアンケートから，注釈伝播が識別子間の関連を把握する上で効果的であったことが確認できた．

主な用語

オブジェクト指向プログラミング (Object-Oriented Programming)
プログラム理解 (Program Comprehension)
統合開発環境 (Integrated Development Environment)

目次

1 まえがき **5**

2 背景 **7**

2.1 オブジェクト指向プログラミング . . . 7

2.1.1 カプセル化によるモジュール性の向上 . . . 7

2.1.2 多相性の適用や継承によるコードの抽象化 . . . 8

2.1.3 オブジェクト指向プログラミングを用いる問題点 . . . 11

2.2 関連研究およびツールによる支援環境 . . . 12

3 付加注釈 DocumentTag の提案 **16**

3.1 付加注釈 DocumentTag の定義 . . . 16

3.2 注釈伝播（Document Propagation） . . . 19

3.2.1 変数の型からの注釈伝播 . . . 21

3.2.2 シグネチャに基づく注釈伝播 . . . 22

3.2.3 スーパークラスからの注釈伝播 . . . 23

3.2.4 オーバーライドに基づく注釈伝播 . . . 24

3.2.5 オーバーロードに基づく注釈伝播 . . . 25

3.2.6 変数初期化子に基づく注釈伝播 . . . 26

4 実装 **28**

4.1 統合開発環境への統合 . . . 30

4.2 DocumentTag のデータモデル . . . 32

4.3 注釈伝播の実現方法 . . . 33

5 実験 **35**

5.1 作業時間の比較 . . . 36

5.2 アンケート結果 . . . 38

6 考察 **40**

7 あとがき **41**

謝辞 **43**

参考文献 **44**

付録 47

1　まえがき

近年普及しつつあるオブジェクト指向プログラミングでは，カプセル化や継承・多相性などの機構を効果的に用いることで，プログラム理解やコーディング，テスト，再利用の効率を向上させることができる [3]．オブジェクト指向プログラミングは，1967 年に登場して以来，Graphical User Interface（GUI）プログラミング [7] や構文解析器生成系（Parser Generator）[9] などの分野で効果的な活用法が見出されている．また，多くのプログラムに適用可能な活用法は，デザインパターン（Design Pattern）[10] として整理されている．GUI プログラミングのためのクラス郡やデザインパターンの実装例は，Java 言語の JFrame クラスや Iterator クラス（本研究では Java 言語での参照型であるクラス，列挙型，インターフェース，アノテーションの総称をクラスと呼ぶことにする），Observable クラスなどのように，ライブラリの一部として再利用されている．

一方で，オブジェクト指向プログラミングを導入することにより，たとえばクラスやメソッド，ローカル変数などの識別子間の関連が複雑かつ抽象的になり，プログラムを詳細に理解にすることが困難になるという問題が指摘されている [6][20]．これは，オブジェクト指向プログラミングを導入することで，クラス間の継承関係や，動的束縛といった新たなプログラム内の依存関係を把握しながらプログラム理解を行うことが必要になるためであり，この依存関係を把握するには，ツールによる理解支援が不可欠であるといわれている [15][24]．

プログラム理解支援を目的としたツールや手法は多数提案されているが [1][5][14]，理解した内容を記録・再利用する手段が効果的でないことが課題として残されている [8][17][21]．これまでの記録手段としては，コメントとしてプログラムに直接記述する方法と，仕様書などのプログラムとは独立した文書として記述する方法が挙げられるが，いずれの場合にも記述する量や種類が膨大になる場合，管理上の問題が生じる．コメントとしてプログラム中に情報を記述する場合，そのようなコメントがプログラム中に散在し，プログラムの可読性を低下させ，バグを発生させる危険がある [4]．また，プログラムとは独立して記述する場合，ソースコードの変更に追従できずにプログラムとの不整合が生じることがあり，記述した内容とソースコードとの対応関係を管理することが難しくなる．

本研究では，開発者がソースコードの読解において理解した内容を効果的に記録するための付加注釈 DocumentTag を提案する．DocumentTag とはプログラムの識別子と関連付けられた注釈であり，開発者は識別子情報を入力として，注釈を参照することができる．また，DocumentTag は他の関連する識別子へと同一の注釈を関連付ける注釈伝播ルールを備えている．開発者がプログラムに含まれる識別子に対して，仕様や使用例などの理解した内容を DocumentTag として記録すると，その識別子だけでなく他の関連する識別子に対しても，記録した注釈が「関連する注釈」として付加される．たとえば，あるクラスに対して，

開発者がクラスの役割を DocumentTag として記述すると，そのクラスだけでなくサブクラスに対しても関連付けが行われ，開発者がサブクラスの注釈を参照する際に，関連付けられた DocumentTag も同時に参照することができる．この関連する注釈が付加される仕組みを，本研究では注釈伝播と呼び，識別子の種類に応じた注釈伝播ルールを定義する．そして，開発者が識別子の注釈を参照するときに，その識別子に対して記録されている注釈と，関連のある識別子に付加されたすべての注釈も同時に参照することができる環境を構築する．これにより開発者は，識別子間の関連を把握しながら効果的なプログラムの理解を行うことができる．

DocumentTag の有効性を確認するために，統合開発環境 Eclipse の拡張機能として，DocumentTag の記述・閲覧ができる機能を持ったツールの実装を行い，適用実験を行って，DocumentTag による注釈伝播の有効性を評価した．具体的には学生 4 人に対して，Java 言語で書かれた既存のプログラムの拡張作業を実施し，作業時間などを評価した．その結果，注釈伝播によって作業時間の平均が短縮された．また，被験者に対して行ったアンケートから，注釈伝播が識別子間の関連を把握する上で効果的であったことが確認できた．

以降，2 章では背景としてオブジェクト指向プログラミングやオブジェクト指向プログラミングに対応した開発環境，プログラム理解のための関連研究およびその問題点について説明を行う．3 章では提案する DocumentTag の説明をおこない，4 章では DocumentTag を実装したツールについて述べる．5 章では実装したツールを用いて行った適用実験とその結果について説明し，6 章では実験に対する考察を述べる．まとめと今後の課題をあとがきとして 7 章に記す．

2　背景

2.1　オブジェクト指向プログラミング

本節では，オブジェクト指向プログラミングを導入することの利点や問題点について説明するオブジェクト指向プログラミングを用いる利点には大きく分けて以下の 2 つが挙げられる．

- カプセル化によるモジュール性の向上
- 多相性の適用や継承によるコードの抽象化

この 2 つの機構を効果的に用いることで，プログラム理解・コーディング・テスト・再利用の効率を向上させることができる．

2.1.1　カプセル化によるモジュール性の向上

カプセル化（Encapsulation）とは，モジュールが Application Programming Interface（API）のみを外部に公開し，モジュールの内部的な動作に対しては，外部からの利用を禁止することを指す [3].

Java 言語のようにクラスを持つオブジェクト指向言語では，クラスやメンバのアクセス制御（Access Control）を行うことが，モジュールのカプセル化を行う手段である．Java 言語では予約語 private，protected，public をアクセス修飾子（Access Modifier）として，識別子の宣言部で用いることで，クラスの内部的な機能を外部から隠蔽することができる．

カプセル化されたモジュール間では，それぞれが公開している API のみを介して相互の処理の呼び出しを行う．これによって，各モジュールは，互いの実装の詳細に依存しなくなり，モジュール間のデータ的な結合度（Coupling）が低下し，処理間の凝集度（Cohesion）を上昇させることができる．以上より，カプセル化を行うことでモジュール性（Modularity）を向上できることがいえる．

カプセル化を用いて情報隠蔽を行うことは，ソフトウェアの開発工程のなかで重要な要素であり，情報隠蔽によって以下に挙げる工程での効率を向上させることが可能になると考えられている [18].

- プログラム理解
  - 他のモジュールに依存する割合が小さくなるので，各モジュールの機能や役割をモジュール単位で単独として理解することが可能となる（Modular Reasoning）.
- コーディング

  – 各モジュールを並行して開発することができ，コーディングの効率を向上させることができる．また，公開されている API さえ変更していなければ，内部の動作の変更が他のモジュールへと影響することがなくなることから，修正もより短時間に安全に行うことが可能となる．

- テスト
  – モジュールを平行して開発することができるので，テストも並行して行える部分が増え，テストの効率を向上させることができる．

- 再利用
  – 個々のモジュールが他のモジュールに依存する割合が小さくなるので，モジュール単位の再利用も容易になる．

また，情報隠蔽によってモジュール性が向上する他に，カプセル化を用いることでデータの破壊を防ぐことができるという利点がある．これはとくに高い信頼性が要求される情報を扱うソフトウェアにとって重要な概念である．実世界の情報はソフトウェアの中で最も重要な存在であり，欠陥などによって不正な値に変更されないようにデータを守ることが肝要である [19]．データを従来の手続き型プログラミングの中で扱う場合，構造体（レコード型）と呼ばれるデータ構造を用いるが，プログラム内ではこの構造体の各メンバ変数に直接アクセスできてしまうという問題がある [3]．これは欠陥が含まれるコード内で不正にデータが書き換えられても，エラーとして検出されずにデータが破壊されてしまうことを意味する．そこでオブジェクト指向プログラミングでは，カプセル化を用いることにより，プログラム内でデータに直接アクセスする代わりにアクセサメソッド（Accessor Method）を介してデータにアクセスし，データには直接アクセスできないように隠蔽することができる．そのアクセサメソッド内でエラーチェックを行うようにすることで，不正な値が読み書きされた場合のエラー検出が容易になり，データの破壊を防ぐことができる．

2.1.2 多相性の適用や継承によるコードの抽象化

継承（Inheritance）と多相性（Polymorphism）[11] は，互いに異なる概念であるが，機構としてはいずれも，ある 2 つのクラス間にスーパークラス（Super Class），サブクラス（Sub Class）という依存関係を持たせることであるという点で共通している．

たとえばクラス A がクラス B のサブクラス，クラス B がクラス A のスーパークラスであった場合，サブクラス A を処理の中で主に用いている場合は，スーパークラス B の機能を拡張（Extend）して用いているので継承であるといえる．逆にスーパークラス B を処理の中で

主に用いている場合は，スーパークラス B の未実装になっている機能をサブクラス A で実装（Implement）しているので多相性を適用しているといえる．

Java 言語ではスーパークラス，サブクラスの関係を持たせる手段として予約語に extends, implements があるが，概念的に 2 つのクラスの関係が継承なのか多相性なのか，あるいは両方なのかを特定するには，予約語の種類ではなく，上記の例のようにスーパークラスを拡張してサブクラスを用いているのか，またはサブクラスで実装してスーパークラスを用いているのかが実際の判断基準となる．以降，継承と多相性の違いを具体的に Java プログラムの例を用いて説明する．

継承を行った場合のコード例　継承を行った場合の例として Java プログラムで Template Method パターン [10] を実装したコードの例を図 1 に示す．この TabFoldingEditor クラスは，統合開発環境 Eclipse の拡張機能として組み込むことで，Eclipse 上でテキストファイルを開くと同時にインスタンス化され，40 行目で始まる createPartControl メソッドが呼び出される仕組みになっている．37 行目で示すように TabFoldingEditor クラスが TextEditor クラスを継承しており，43 行目の getSourceViewer メソッドや，46 行目，47 行目にある getAnnotationAccess, getSharedColors メソッドは TabFoldingEditor クラスのスーパークラスである TextEditor クラスで利用できるメソッドである．サブクラス内でスーパークラスの機能を利用し，スーパークラスではなくサブクラスを目的の処理の中で扱っているので，図 1 のコード例は継承を行っているといえる．このように継承を行うことで，既存のモジュールを流用することができ，コードの重複を回避できるという利点がある．しかし，継承はスーパークラスの実装の詳細に大きく依存することになるため [3]，継承を行う際は注意が必要である．継承の機構としては，あるクラスを継承したあるクラスをさらに継承することもできるが，このような連鎖的な継承は，3 回から 5 回以上行うことで理解や修正が困難になるということが実験的に示されている [13]．これは，継承を行うことによりモジュール性が低下し，カプセル化の利点を大きく損ねることが原因である．以上から，継承は必要な場合にのみ行われるべきであり，不必要な継承は避けるべきであるといえる．

多相性を適用した場合のコード例　多相性を適用した場合の例として Java プログラムで実装した場合のコード例を図 2 に示す．図 2 のコード片では，109，110 行目でリスト構造を表す型の変数 lineElementList を宣言し，112，113 行目では，lineElementList に LineElement のインスタンスをリストの各要素として複数記録し，記録したリストを用いて 114，116，117 行目の処理を行っている．ここで，110 行目の ArrayList クラスは List クラスを実装しており，ArrayList クラスは List クラスのサブクラスである．継承の場合とは異なり，サブクラスは，110 行目のようにスーパークラスの変数に代入するときにのみ用いて，以降の処理で

```
36  */
37 public class TabFoldigEditor extends TextEditor {
38
39⊖    @Override
40     public void createPartControl(Composite parent) {
41         super.createPartControl(parent);
42         ProjectionViewer projectionViewer =
43             (ProjectionViewer) getSourceViewer();
44         projectionSupport = new ProjectionSupport(
45                 projectionViewer,
46                 getAnnotationAccess(),
47                 getSharedColors());
48         projectionSupport.install();
49         projectionViewer.doOperation(
50                 ProjectionViewer.TOGGLE);
51         updateFolding();
52
53     }
54
55⊖    public TabFoldigEditor() {
```

図 1: 継承を行った場合のコード例

は 113，114，116，117 行目のようにスーパークラスの変数を主に用いているので，図 2 は多相性を適用しているといえる．このようにスーパークラスの変数にサブクラスのインスタンスを代入して以降の処理で用いることは，静的（Static）に定まる型とは異なった型が動的（Dynamic）に代入され，以降の処理では，動的に代入された型で宣言されたメンバが実際にアクセスされるので，動的束縛（Dynamic Binding）とも呼ばれる．Java 言語における動的束縛は，クラス階層（各クラスを頂点として，サブクラスからスーパークラスに有向辺を引くことで得られるクラス間の非循環有向グラフ）に従って行うことができ，継承を行う場合でも，サブクラスで宣言されていないメンバにアクセスする場合，スーパークラスで宣言されたメンバが実行時にサブクラスのメンバとして使用されるため，動的束縛を行っているといえる．このように多相性を適用して動的束縛を行うことにより，サブクラスの実装の詳細を隠蔽しながら，スーパークラスの機能として実行することができ，処理の切り替えも容易になるという利点がある．たとえば図 2 の例では，110 行目で ArrayList クラスの代わりに，List クラスのサブクラスである LinkedList クラスを用いることができる．LinkedList クラスは双方向リスト構造であり，配列構造の ArrayList クラスよりも要素の追加・削除が高速に行えるため，計算時間の短縮が期待できる．この LinkedList クラスへの変更の際には

```
108
109             List <LineElement> lineElementList =
110                 new ArrayList<LineElement>();
111
112             registerLineElementList(source,
113                     lineElementList);
114             if (lineElementList.size()==0)return;
115
116             applyFolding(lineElementList,source,model,0);
117             outputHTMLFile(lineElementList);
118
```

図 2: 多相性を適用した場合のコード例

110 行目のみを変更すればよく，多相性を適用しない場合に必要となる条件分岐を排除することができる．

以上のことから，継承や多相性の概念を効果的に導入してクラス間に依存関係を持たせることにより，コードの重複や複雑な条件文を回避することができ，カプセル化によるモジュール性の向上をさらに促進することができる．

2.1.3　オブジェクト指向プログラミングを用いる問題点

2.1.1 節で述べた通り，オブジェクト指向プログラミングを導入し，カプセル化を行うことで，モジュール性が向上し，プログラム理解やコーディング，テスト，再利用の効率を向上させることができる [3]．しかし，多相性の適用や継承によって，たとえばクラスやメソッド，ローカル変数などの識別子間の関連が複雑かつ抽象的になり，プログラムを詳細に理解にすることが困難になるという問題が指摘されている [6][20]．

具体的にクラスの場合は，継承を行うことでクラス間にスーパークラス，サブクラスの依存関係が発生し，クラスを開発者が理解するためには，そのクラスのスーパークラス，サブクラスとの関連を把握する必要がある．また，メソッドの場合も継承や多相性の適用によって，メソッド間に，オーバーライド，オーバーライドという依存関係が発生し，オーバーライドされているメソッド間，あるいはオーバーロードとなっているメソッド間で，処理内容に一貫性を保つ必要があり，クラス間，メソッド間の関連を把握しながら，プログラム理解や保守作業を行うことが必要になる．

また，クラス間やメソッド間の関連など，オブジェクト指向プログラミングを導入することで発生する識別子間の関連を，開発者が把握しながらプログラム理解を行うためには，ツールによる理解支援が不可欠であるといわれている [15][24]．つまり，プログラムの閲覧・

編集やコンパイル，デバッグを行うツールだけでなく，クラス間やメソッド間の関連を把握するためのツールを用いることで，開発者のプログラム理解の効率を向上させることが必要である．そのためには，プログラムを閲覧する際に，クラス間やメソッド間の関連を知るためのツールが同時に起動され，同期した動作を行うことが必要となる．テキストエディタやコンソールなど，個々のツールが独立するよりも，開発者が必要とする複数のツールを 1 つのシステムとして統合した統合開発環境を用いてプログラム理解や保守作業を行うことが効果的であるといえる [16]．統合開発環境内で，識別子間の関連を把握するツールを起動し，識別子間の関連を把握しながら，プログラムの閲覧・編集を行うことで，オブジェクト指向プログラミングを導入する問題点を補うことができる．以降では，この統合開発環境内で，プログラム理解を支援するツールを用いた支援環境について Eclipse を例に挙げて説明する．

2.2　関連研究およびツールによる支援環境

本節では，統合開発環境内で利用できるプログラム理解支援ツール，および関連研究について，Java プログラムの統合開発環境である Eclipse を例に説明する．

クラス階層の閲覧　2.1.2 節では，多相性や継承を用いて 2 つのクラス間に親子関係スーパークラス，サブクラスの依存関係を持たせることができることを述べたが，各クラスを頂点として，サブクラスからスーパークラスに有向辺を引くことで，クラス間の非循環有向グラフ（DAG）を構成することができる．このクラス間のグラフはクラス階層（Class Hierarchy）と呼ばれ，クラス間の依存関係をクラス階層として閲覧できるようにすることで，クラス間の把握が容易になる．統合開発環境 Eclipse では，マウス操作やショートカットキーを用いてクラス間の依存関係を閲覧することができる．図 3 では，Eclipse 上で選択した ExpandBar クラスに対して，Object クラスまでのスーパークラスを階層的に表示している．クラス階層を閲覧するツールを用いることで，クラス間の関係を把握しながらプログラムを理解することができる．しかしこのツールは，スーパークラスやサブクラスの存在を確認し，移動することが目的であり，他クラスの仕様や用途などを知るためには，クラス階層を基にクラス間を移動しながら理解する必要があり，クラス階層が複雑になるほど，開発者がプログラム理解の負担は増大する [6][13]．

参照関係の検索　Eclipse では，プログラム内で定義されている各識別子について，その識別子が参照されている箇所を簡単な操作により検索することができる．図 4 では，選択した TagEditorElementChangeListener() コンストラクタが他のメソッドで参照されている箇所を検索し，一覧で表示している．とくにメソッドやコンストラクタが参照されている箇所は，呼び出されている箇所と見なすことができるため，メソッドの呼び出し元を探す上で有用で

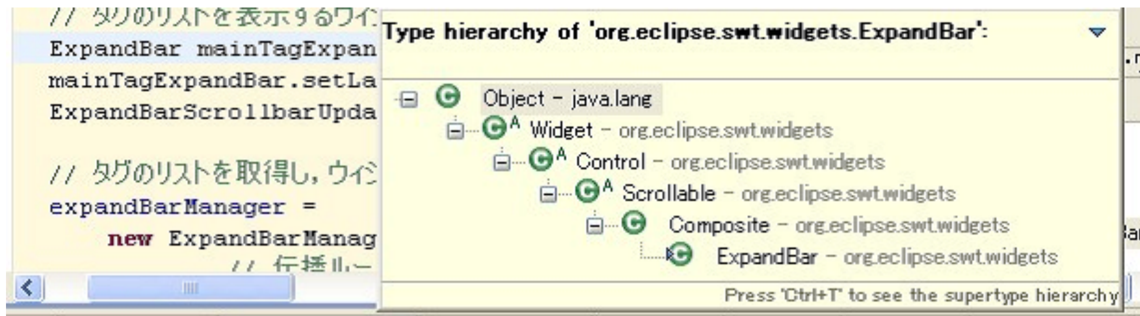

図 3: クラス階層の閲覧

ある．しかし，このツールのみでは，オーバーライドされているメソッドに移動することはできないため，クラス階層に従って，ソースコードを移動しながら，実際に呼び出されるメソッドを特定する必要がある．このため，クラス階層が複雑になるほど，参照関係の検索，とくにメソッドの呼び出し関係を調べることが困難になるといえる．

以上は，Eclipse に組み込まれている主なツールであり，オブジェクト指向プログラムに対応したツール支援が行われている．しかしこれらのツールは，クラス階層や呼び出し関係などに従ってソースコード上の移動を支援するものであり，クラス階層が複雑になると，ソースコード上の移動は困難になる．つまり，オブジェクト指向プログラミングを導入することで，プログラム理解が困難になるという問題の主要な原因は，開発者がソースコード上を移動する負担が増えることにあると考えられる. この問題に対処するために，開発者が与えた情報を基にソースコードの移動を支援する研究がいくつか行われている．以降では，これらについて説明する．

Eclipse Plug-in Mylar[14]　プログラム理解のために検索しながらソースコード上を移動することが多いことを問題として，統合開発環境 Eclipse の拡張機能として Mylar（Mylyn）が実装された．この拡張機能は，Degree Of Interest（DOI）というメトリクスを用いており，開発者が頻繁に閲覧または編集を行っているメソッドやクラスほど，DOI の値が高くなる．この DOI の値を用いて，ファイル階層やクラス階層を閲覧する画面などに現れるクラスやメソッドに対して，頻繁に利用するものは，赤くハイライトを行い，逆にほとんど利用しないものは非表示にすることで検索効率を向上させるという研究である．開発者が作業することで変化する DOI の結果を開発者間で共有・管理・再利用することが課題として挙げれらている．

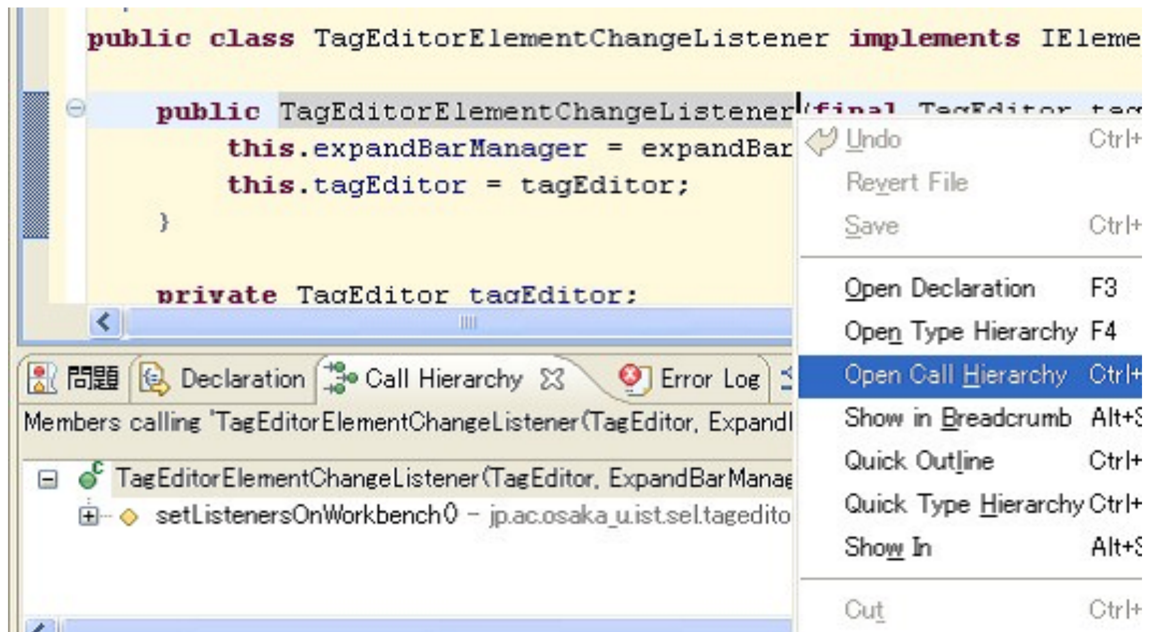

図 4: 参照関係の検索

Eclipse Plug-in TagSEA[21]　Mylar の拡張機能と同様に，ソフトウェア保守の中で頻繁に行われるコードの検索効率を向上させるために，コード上の具体的な位置を記録することを可能にする Eclipse の拡張機能として TagSEA が実装されている．この拡張機能では，プログラム中のコメントに'@tag' で始まる文字列を開発者が記述することで，記述した内容が，位置情報とともに，小画面に一覧で表示される．一覧から登録した位置に移動できるので，検索を行わずに効果的にソースコード内を移動できる．しかし適用実験の結果としては，期間の長い開発を想定した場合には，記述内容を体系化することで，効果的に移動できるのではないかというインタビュー結果も得られたが，適用実験で行った期間の短い作業では，開発者がコメントの中に情報を直接記述することや，記述した内容をグループ間で共有することに抵抗があることが分かった．原因としては記述した内容が，効果的に再利用されないことが挙げられている．

Eclipse Plug-in JTourBus[17]　プログラム理解のための，ソースコード上の移動を支援する Eclipse の拡張機能として JTourBus が実装されている．JTourBus は，開発者がコメント中に'@JTourBusStop' で始まる文字列を記述することで，記述した箇所が小画面に一覧で表示される．記録した箇所は，ソースツアーの 1 要素として登録することができ，開発者がそのソースツアーを選択することで，登録した'@JTourBusStop' の箇所に順を追って自動的に移動する．ソースコード上の移動を自動化することで移動のための検索をする作業が軽減

され，プログラム理解を効果的に行うことができる．適用実験の結果として，コメントよりも’@JTourBusStop’ として記述する方が，保守作業が効率化されるという結果にはなったが，より多くのメタ情報を記述できる Javadoc の方を利用する方が効果的であるという結果が得られた．原因としては，JTourBus で記述した内容に対して，有効な再利用方法が確立できず，’@JTourBusStop’ として記述した内容が効果的に再利用できなかったことが挙げられている．

以上より，プログラム理解を目的とした手法やツールは提案されているが，プログラムに直接記述することへの負担がある点や，記述した内容を開発者間で共有・再利用することが困難である点が課題として挙げられている．また，識別子間の依存関係を基にした移動支援は行われず，ツールを用いたプログラム理解支援を行うには課題が残されている．本研究では，この課題に対して，識別子間の関連を把握しながら，開発者の記述内容を効果的に共有・再利用する手段として，DocumentTag を提案し，Eclipse の拡張機能として実装した．3 章以降で具体的な説明を行う．

3　付加注釈 DocumentTag の提案

オブジェクト指向プログラミングを導入することにより，モジュール性の向上が行えるが，識別子間の関連が複雑かつ抽象的になり，プログラム理解にはツールの支援が必要であるとされている．既存の研究は，

プログラム理解の際に理解した内容を効果的に再利用できない点が課題として挙げられている．本研究ではこの問題に対して，理解した内容を効果的に記録・再利用し，細部にわたる理解を進めることができる付加注釈 DocumentTag を提案する．

DocumentTag は，プログラムの識別子に関連付けて記述する注釈であり，開発者は識別子情報を入力として，注釈を記録・参照する．本研究での注釈とは，プログラムに含まれる識別子に対して，仕様や使用例などを記述した文書である．また，DocumentTag は注釈伝播ルールを備えている．注釈伝播とは，注釈を DocumentTag を用いて記録することで，その識別子だけでなく関連する識別子に対しても，記録した注釈が関連する注釈として付加されることである．これにより，開発者は直接関連付けられた識別子だけでなく，依存関係を持つ識別子に対しても注釈を参照することができる．以降では，DocumentTag，および注釈伝播についての具体的な説明を行う．

3.1　付加注釈 DocumentTag の定義

DocumentTag とは，プログラムの識別子と関連付けられた注釈であり，開発者は識別子情報を入力として，注釈を記録・参照することができる．

現時点ではプログラムの内容を記録する手段として，コメントとしてプログラムに直接記述する方法と，もしくは仕様書などのプログラムとは独立した文書として記述す方法が挙げられる．

DocumentTag は，これらの手段とは異なり，コメントのようにプログラムと位置的な関連性を保ちながら，仕様書のようにプログラムに直接記述することなく，プログラムを内容を説明することができる．

記述内容　DocumentTag に注釈として記述する内容は，コメントや仕様書などに記述する内容と同様であることが想定される．具体的には，以下のように，注釈の種類は多岐に及ぶといえる．

- 識別子の仕様
  - プログラム内での役割や，一般的な扱い方，注意点，他の識別子との関連を記す
- 識別子の使用例

– 具体的な使い方をコード例を用いて説明する

- バグレポート
  – バグ ID やバグの内容，再現手順を記述する

記述項目　これらの内容の種類を区別するために，DocumentTag に記述する情報は以下に挙げる通り，注釈の記述内容に加えて，著者などのメタ情報を含めた 5 項目について，開発者は記述を行う．これにより，注釈の管理が容易になると考えられる．

- 注釈の記述内容
- 注釈の種類
- 注釈に使用している自然言語（English など）
- 記録した日時
- 著者

DocumentTag の形式的表現　以上の説明より，開発者が記録した 1 つの付加注釈 DocumentTag を，識別子情報 e，注釈情報 d，注釈伝播の種類 r の組として，式 (1) に示す形式で表現する．

$$DocumentTag \stackrel{\text{def}}{=} (e, d, r) \tag{1}$$

識別子情報 e は，その識別子がプログラム上で宣言された位置 $decl(e)$ により一意に定まる．ゆえに式 (2) に示す通り，識別子情報 e_1，e_2 が同一である必要十分条件は，プログラム上の宣言位置が一致していることである．ただし，本研究で行った DocumentTag の実装の都合上，局所変数の識別子情報のみ，式 (2) が成り立たない．この理由として，局所変数の識別子情報を一意に定めるためには，プログラム上の詳細な宣言位置を識別子情報に含める必要があり，開発者がプログラムを編集することで，位置情報が一致しなくなり，記録した注釈が参照できなくなるという問題があるためである．この問題に対処するために，局所変数の場合はプログラム上の詳細な位置情報を除外し，同じメソッド内に含まれる同名同型の局所変数は，同一の識別子と見なすことにしている．

$$e_1 \equiv e_2 \iff decl(e_1) = decl(e_2) \tag{2}$$

注釈情報 d は，複数の文字列内容から構成されており，式 (3) に示す通り，注釈の記述内容 n，注釈の種類 c，使用言語 l，記録日時 t，著者 a の組で表現される．

$$d \stackrel{\text{def}}{=} (n, c, l, t, a) \tag{3}$$

注釈伝播の種類 r は 3.2 節で説明する注釈伝播の種類を記録するための項目である．注釈伝播の種類 r を用いることで，注釈情報 d が，開発者が入力とした識別子情報に対してどのような関連があるかを提示することができる．たとえば，開発者があるクラスの注釈を参照するとき，スーパークラスの注釈も同時に参照できるが，この場合は，クラスの識別子と，スーパークラスの注釈伝播を行う注釈伝播ルールを基に，クラスの注釈をスーパークラスの注釈として開発者に提示している．すなわち注釈伝播の種類 r は，注釈伝播ルール R の種類 $rtype(R)$ と識別子情報 e の種類 $etype(e)$ から一意に定まる．この一意に定める関数を $rule$ として，式 (4) で注釈伝播の種類 r を定義する．

$$r \stackrel{\mathrm{def}}{=} rule(R, e) \tag{4}$$

この DocumentTag の例として，たとえば Integer クラスの注釈を開発者が参照したときに，注釈伝播によってスーパークラスである Number クラスの注釈も，同時に参照できる場合を考える．Integer クラスの注釈の内容は例として式 (5) の $DocumentTag_1$ のような情報が格納される．識別子情報は Integer クラスを一意に識別する情報であり，d_1 には，Integer クラスに対して記述された注釈内容が格納されている．また r_1 には，開発者が直接参照した注釈であることを示す $'MAIN'$ という情報が格納される．また，開発者が Integer クラスの注釈を参照したときに，スーパークラスとして同時に表示される Number クラスの注釈は，式 (6) の $DocumentTag_2$ の情報が開発者に提示される．r_2 には開発者が参照した注釈に対して，スーパークラスであることを示す $'SUPER'$ という情報が格納され，開発者に対して，開発者自らが参照した注釈と，どのような関係にある識別子の注釈かを示している．

$$\begin{aligned}
DocumentTag_1 &= (e_1, d_1, r_1) \qquad (5)\\
e_1 &= 'java.lang.Integer'\\
d_1 &= ('\text{整数を扱うクラス}', 'regime', 'Japanese', '\text{2009/2/9}', '\text{田中}')\\
r_1 &= 'MAIN'\\
DocumentTag_2 &= (e_2, d_2, r_2) \qquad (6)\\
e_2 &= 'java.lang.Number'\\
d_2 &= ('\text{数を扱うクラス}', 'regime', 'Japanese', '\text{2009/2/8}', '\text{田中}')\\
r_2 &= 'SUPER'
\end{aligned}$$

また式 (5)，式 (6) の例のように，本研究では，開発者に対して複数の DocumentTag を提示する場合を想定しているため，注釈伝播によって付加される複数の DocumentTag は，順序を持ち，重複を許す．すなわち，注釈伝播によって開発者に提示される DocumentTag の列

L は，式（7）で示すように，集合ではなく順序を持った列で表現される．

$$L = [DocumentTag_1, DocumentTag_2, \ldots, DocumentTag_n] \quad (7)$$

3.2 注釈伝播（Document Propagation）

注釈およびメタ情報を記録した DocumentTag をプログラム理解の中で効果的に参照できる手法として注釈伝播を提案する．注釈伝播とは，注釈を DocumentTag を用いて記録することで，その識別子だけでなく関連する識別子に対しても，記録した注釈が関連する注釈として付加されることである．この DocumentTag を用いて開発者が識別子の注釈を参照すると注釈伝播により，その識別子と関連のある注釈も同時に参照することができ，プログラム理解を効果的に行うことができる．このことをコード例を用いて説明する．

図 5 は注釈伝播の模式図を表現したものである．開発者が List クラスに注釈を記録すると，プログラム内に出現するすべての List という識別子に注釈が付加される．加えて，List クラスと関連のある識別子にも注釈が付加される．図 5 では，局所変数 lineElementList は，List 型で宣言されているため，List クラスと関連があると判断され，プログラム内の局所変数 lineElementList のすべての出現にも注釈が付加される．開発者が局所変数 lineElementList の注釈を参照すると，局所変数 lineElementList と関連のある List クラスの注釈も同時に参照することができ，プログラム内の依存関係に沿った効果的な理解を行うことができる．

また注釈伝播は，記録した 1 つの注釈がプログラム内の複数の識別子に付加されていくことと定義しているが，注釈を参照する開発者の視点で見た場合は，1 つの識別子の注釈を参照した際に，関連する識別子に付加された注釈も同時に参照できる機構であるといえる．

本研究では，識別子間の関連の種類から，注釈伝播をルールとして以下のように分類する．各注釈伝播ルールを同時に DocumentTag に適用することで，それぞれの注釈伝播ルールに従って，関連のある注釈を同時に参照することができる．

- 変数の型からの注釈伝播
- シグネチャに基づく注釈伝播
- スーパークラスからの注釈伝播
- オーバーライドに基づく注釈伝播
- オーバーロードに基づく注釈伝播
- 変数初期化子に基づく注釈伝播

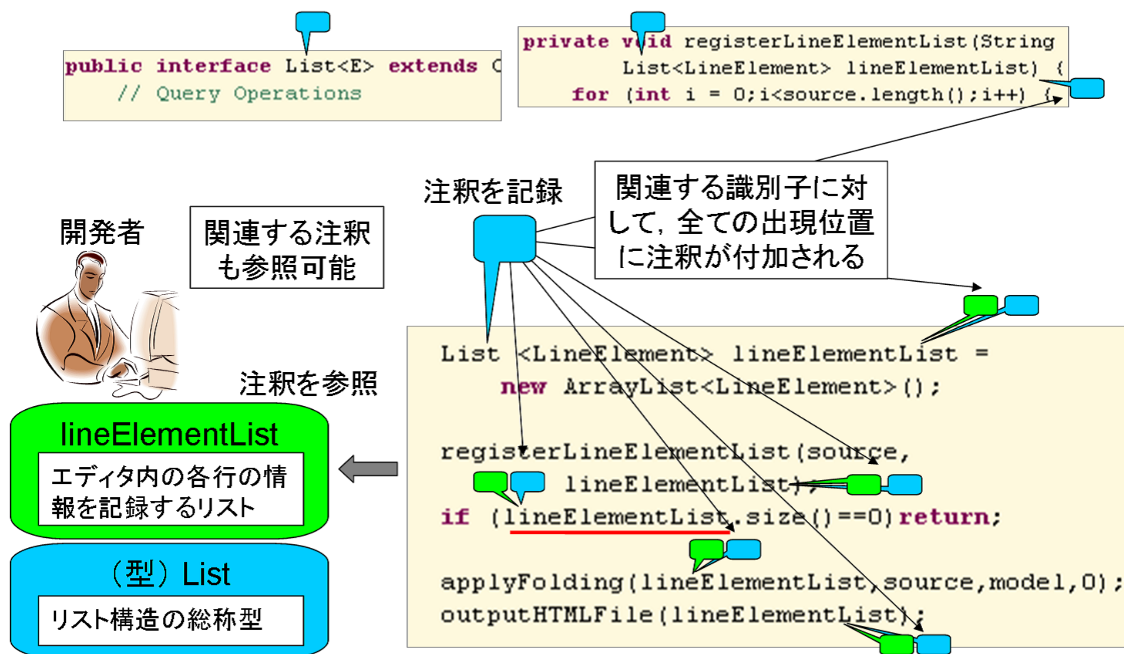

図 5: 注釈伝播

注釈伝播の形式的説明　ここで注釈を参照する開発者の視点から見た注釈伝播について，形式的な説明を行う．開発者が指定した識別子情報 e から，ある 1 つの注釈伝播ルール R によって同時に参照できる DocumentTag の列 L は，式 (8) で表現される．

$$\begin{aligned} L &= R(e) \\ &= [DocumentTag_1, DocumentTag_2, \ldots, DocumentTag_n] \end{aligned} \tag{8}$$

注釈伝播ルール R では，2 つの段階を経て DocumentTag の列 L を求める．第 1 の段階では式 (9) に示すように，操作 $prule$ を用いて開発者が指定した識別子情報 e を基に，関連する識別子情報の列 E を求める．

$$\begin{aligned} E &= prule_R(e) \\ &= [e_1, e_2, \ldots, e_n] \end{aligned} \tag{9}$$

また，識別子情報 e に対して記録されている DocumentTag の列 L_e は，操作 $getd$ を用いて式 (10) によって求められる．$getd$ から求まる $DocumentTag_{e1}, DocumentTag_{e2}, \ldots, DocumentTag_{e3}$ は，すべて識別子情報が e と等しい．

$$\begin{aligned} L_e &= getd_R(e) \\ &= [DocumentTag_{e1}, DocumentTag_{e2}, \ldots, DocumentTag_{en_e}] \end{aligned} \tag{10}$$

第 2 の段階では式 (11) に示す通り，式 (9) の結果を式 (10) の $getd$ すべてに対して適用し，式 (12) のようにべき構造をもった DocumentTag の列が求まる．ここから，式 (13) で示すように，列同士を連結してべき構造を排除する．これは，式 (12) の列は連結しても順序と数が保たれ，かつ各 DocumentTag の情報を基に，式 (13) から式 (12) に逆変換もできるためである．以上の 2 つの段階を経て，式 (13) に示す DocumentTag の列を注釈伝播の結果としている．

$$
\begin{aligned}
L &= R(e) \\
&= [getd(prule_R(e))] \\
&= [getd_R(E)] \\
&= [getd_R(e_1), getd_R(e_2), \ldots, getd_R(e_{n_e})] \quad (11) \\
&= [L_{e_1}, L_{e_2}, \ldots, L_{e_{n_e}}] \\
&= [[DocumentTag_{e_1 1}, DocumentTag_{e_1 2}, \ldots, DocumentTag_{e_1 n_{e_1}}], \quad (12) \\
&\quad [DocumentTag_{e_2 1}, DocumentTag_{e_2 2}, \ldots, DocumentTag_{e_2 n_{e_2}}], \ldots, \\
&\quad [DocumentTag_{e_n 1}, DocumentTag_{e_n 2}, \ldots, DocumentTag_{e_n n_{e_n}}]] \\
&= [DocumentTag_1, DocumentTag_2, \ldots, DocumentTag_n] \quad (13)
\end{aligned}
$$

また，複数の注釈伝播ルール $R_1, R_2, \ldots, R_n$ を適用することで求まる DocumentTag の列 L は，式 (14) で示す通り，各注釈伝播ルールの結果の列として求めることができる．式 (13) と同様に，式 (15) で示す通り，各注釈伝播ルールから求まる DocumentTag の列同士を連結することができる．ここから，L 内の順序は，注釈伝播ルールを適用する順序によって異なることがいえる．

$$
\begin{aligned}
L &= [R_1(e), R_2(e), \ldots, R_n(e)] \quad (14) \\
&= [DocumentTag_{R11}, DocumentTag_{R12}, \ldots, DocumentTag_{R1n}, \quad (15) \\
&\quad DocumentTag_{R21}, DocumentTag_{R22}, \ldots, DocumentTag_{R2n}, \ldots, \\
&\quad DocumentTag_{Rn1}, DocumentTag_{Rn2}, \ldots, DocumentTag_{Rnn}]
\end{aligned}
$$

以降は便宜上，開発者の視点で見た注釈伝播ルールの説明を行う．

3.2.1　変数の型からの注釈伝播

フィールド，局所変数，仮引数は，型を用いて宣言される．本研究の変数の型からの注釈伝播では，変数の宣言文で宣言された型を，関連する識別子と判断して注釈伝播を行う．す

なわち，変数の型からの注釈伝播によって，同時に参照できる DocumentTag の識別子は “フィールド，局所変数，仮引数の変数の型となるクラス” である．

以上より，変数の型からの注釈伝播が行われる対象は，フィールド，局所変数，仮引数であり，この注釈伝播によって同時に参照できるのは，クラスについての DocumentTag である．クラスを持たない型である基本型（int，double など）や void 型は識別子ではないため，注釈伝播が行われない．

形式的説明　変数の型からの注釈伝播のルール R_t では，変数の識別子情報 e から，その変数の型を意味する識別子情報の列 E_{R_t} を求める $prule_{R_t}$ を用いて式 (16) に示す通り，識別子情報 e を入力として，変数の型からの注釈伝播が行われ，DocumentTag の列が求まる．

$$\begin{aligned} L &= R_t(e) \\ &= [getd(prule_{R_t}(e))] \\ &= [getd_{R_t}(e_1), getd_{R_t}(e_2), \ldots, getd_{R_t}(e_{n_e})] \\ &= [DocumentTag_{R_t1}, DocumentTag_{R_t2}, \ldots, DocumentTag_{R_tn}] \end{aligned} \tag{16}$$

なお，識別子情報 e の種類はフィールドまたは局所変数，仮引数である必要がある．e がこれら以外の識別子情報であった場合，L は空集合となる．また，型は，クラスとして宣言されている参照型である必要があり，参照型以外の基本型や void 型で宣言されている場合は，L は空集合となる．

3.2.2　シグネチャに基づく注釈伝播

シグネチャ（Signature）とは，メソッドやコンストラクタを一意に特定するための情報で，Java 言語仕様 [11] では，メソッドシグネチャ（Method Signature）は，メソッド名と引数の型を指し，コンストラクタシグネチャ（Constructor Signature）は，コンストラクタ名と引数の型を指す．本研究のシグネチャに基づく注釈伝播では，メソッドやコンストラクタのシグネチャについて，その戻り値の型や仮引数，仮引数の型，例外の型を，シグネチャに関連した識別子と判断して注釈伝播を行う．すなわち，シグネチャに基づく注釈伝播によって，同時に参照できる DocumentTag の識別子は” メソッドやコンストラクタについて，戻り値の型となるクラス，仮引数，仮引数の型となるクラス，例外の型となるクラス” である．

以上より，シグネチャに基づく注釈伝播が行われる対象は，メソッドやコンストラクタであり，この注釈伝播によって同時に参照できるのは，仮引数やクラスについての DocumentTag である．メソッドやコンストラクタ以外の識別子の場合や，引数や例外の型を持たないメソッドやコンストラクタの場合は，注釈伝播は行われない．また，仮引数の型や例外の型は，

クラスとして宣言されている参照型である必要があり，参照型以外の型で宣言されている場合は，識別子ではないので，注釈伝播は行われない．

形式的説明　シグネチャに基づく注釈伝播のルール R_s では，3 つの注釈伝播ルールから構成されている．まずメソッドやコンストラクタを意味する識別子情報 e から，戻り値の型を求めるルール R_r と，引数とその型の組を第 1 引数から順に，型，引数の順で注釈を求めるルール R_p，例外の型を求めるルール R_x が順序をもって構成されている．R_p，R_x ではそれぞれ，関連する識別子を求める $prule_{R_p}$，$prule_{R_x}$ を用いている．式 (17) に示す通り，識別子情報 e を入力として，シグネチャに基づく注釈伝播が行われ，DocumentTag の列が求まる．

$$
\begin{aligned}
L &= R_s(e) \\
&= [R_r(e), R_p(e), R_x(e)] \\
&= [getd(prule_{R_r}(e)), getd(prule_{R_p}(e)), getd(prule_{R_x}(e))] \\
&= [getd_{R_r}(e_1), getd_{R_r}(e_2), \ldots, getd_{R_r}(e_{n_e}), \\
&\quad getd_{R_p}(e_1), getd_{R_p}(e_2), \ldots, getd_{R_p}(e_{n_e}), \\
&\quad getd_{R_x}(e_1), getd_{R_x}(e_2), \ldots, getd_{R_x}(e_{n_e})] \\
&= [DocumentTag_{R_s 1}, DocumentTag_{R_s 2}, \ldots, DocumentTag_{R_s n}] \qquad (17)
\end{aligned}
$$

なお，識別子情報 e の種類はメソッドもしくはコンストラクタである必要がある．e がメソッドやコンストラクタ以外の識別子情報であった場合や，仮引数や例外の型を持たないメソッドやコンストラクタの場合は，L は空集合となる．また，仮引数の型や例外の型が参照型以外の型である場合は，その型の DocumentTag は L に追加されない．

3.2.3　スーパークラスからの注釈伝播

クラスを持つオブジェクト指向言語では，クラスはさまざまな形で定義することができる．Java 言語ではクラスの用途に応じて，class や interface，enum として定義することができ，クラスの内部でメンバとして宣言する内部クラス（Member Class）や，式中に宣言する無名クラス（Anonymous Class），ブロック中に名前を持って宣言するローカルクラス（Local Class）がある．また，それぞれに定義したクラス間には親子関係を持たせることができ，継承や多相性を適用することができる．スーパークラスからの注釈伝播では，開発者がクラスの注釈を参照した場合，そのクラスの直接のスーパークラスを，クラスに関連した識別子と判断して注釈伝播を行う．すなわち，スーパークラスからの注釈伝播によって，同時に参照できる DocumentTag の識別子は “クラスの直接のスーパークラスとなるクラス” である．

以上より，スーパークラスからの注釈伝播が行われる対象は，クラスであり，この注釈伝播によって同時に参照できるのは，そのクラスから見て直接のスーパークラスとなるクラスについての DocumentTag である．直接のスーパークラスとは，クラス階層（各クラスを頂点として，サブクラスからスーパークラスに有向辺を引いた非循環有向グラフ）の上では到達可能な隣接頂点に位置するクラスのみを指し，2 階層上位にあるスーパークラスの識別子に対しては注釈伝播を行わない．また，Java 言語では Object クラスがクラス階層の最上位に位置するクラスであり，Java 言語の interface，enum のスーパークラスは Object クラスである．

形式的説明　スーパークラスからの注釈伝播のルール R_c では，クラスを意味する識別子情報 e から，そのクラスから見て直接のスーパークラスを意味する識別子情報の列 E_{R_c} を求める $prule_{R_c}$ を用いて式 (18) に示す通り，識別子情報 e を入力として，スーパークラスからの注釈伝播が行われ，DocumentTag の列が求まる．

$$\begin{aligned} L &= R_c(e) \\ &= [getd_{R_c}(e_1), getd_{R_c}(e_2), \ldots, getd_{R_c}(e_{n_e})] \\ &= [DocumentTag_{R_c1}, DocumentTag_{R_c2}, \ldots, DocumentTag_{R_cn}] \end{aligned} \quad (18)$$

なお，識別子情報 e の種類はクラスである必要がある．e がクラス以外の識別子情報であった場合や，スーパークラスが存在しない場合は，L は空集合となる．

3.2.4　オーバーライドに基づく注釈伝播

オーバーライド（Overriding）とは，スーパークラスで定義されたメソッドを，サブクラスで定義しなおすことであり，多相性を適用する場合はとくに用いられる技法である．オーバーライドに基づく注釈伝播では，あるメソッドについて，開発者が注釈を参照した場合に，そのメソッドにオーバーライドされているメソッドの注釈を同時に参照することができる．オーバーライドされているメソッドの注釈が参照できることで，オーバーライドによるメソッド間の依存関係を把握することが容易になること考えられる．以上から，オーバーライドに基づく注釈伝播を行う対象はメソッドのみであり，オーバーライドに基づく注釈伝播によって付加されてくる注釈の識別子は，“オーバーライドされているすべてのメソッド” である

なお，コンストラクタはオーバーライドの対象にはならない．また，オーバーライドは，クラスの継承と同様に，あるメソッドをオーバーライドしたメソッドをさらにオーバーライドすることができるため，あるメソッドから見てオーバーライドされているメソッドは複

数になることもある．その場合も考慮して，すべてのスーパークラス内のメソッドの中で，シグネチャが一致するメソッドをオーバーライドされているメソッドとして，開発者に提示する．

形式的説明　オーバーライドに基づく注釈伝播のルール R_r では，メソッドを意味する識別子情報 e から，そのメソッドから見てオーバーライドされているすべてのメソッドを意味する識別子情報の列 E_{R_r} を求める $prule_{R_r}$ を用いて式 (20) に示す通り，識別子情報 e を入力として，オーバーライドに基づく注釈伝播が行われ，DocumentTag の列が求まる．

$$
\begin{aligned}
L &= R_r(e) \\
&= [getd_{R_r}(e_1), getd_{R_r}(e_2), \ldots, getd_{R_r}(e_{n_e})] \\
&= [DocumentTag_{R_r 1}, DocumentTag_{R_r 2}, \ldots, DocumentTag_{R_r n}] \qquad (19)
\end{aligned}
$$

なお，識別子情報 e の種類はメソッドである必要がある．e がメソッド以外の識別子情報であった場合や，オーバーライドされているメソッドが存在しない場合は，L は空集合となる．

3.2.5　オーバーロードに基づく注釈伝播

オーバーロード（Overloading）とは，あるメソッドが，そのクラスに含まれるメソッドおよび，そのクラスのすべてのスーパークラスで定義されているメソッドの中で，メソッド名のみが一致するメソッドがある状態を指す．オーバーロードとなるメソッドが複数定義されることで，メソッドの引数の型が変更されても柔軟に呼び出されるメソッドを自動で切り替えることができる．オーバーロードに基づく注釈伝播では，開発者があるメソッドの注釈を参照したときに，そのメソッドとオーバーロードになっているメソッドの注釈を同時に参照することができる．ただし，あるメソッドとオーバーロードになっているすべてのメソッドの注釈が付加されるのではなく，クラス内でオーバーロードになっているメソッドのみを同時に参照できるものとする．これは，すべてのスーパークラスに含まれる同名のメソッドの注釈も同時に表示した場合に，一度に同時に参照される注釈の数が多くなりすぎることが考えられるためである．この注釈伝播は今後の課題として工夫の余地がある．以上から，オーバーロードに基づく注釈伝播を行う対象は，コンストラクタを含めたメソッドであり，オーバーロードに基づく注釈伝播によって付加されてくる注釈の識別子は，“メソッドが宣言されているクラス内のメソッドの中で，識別子名が一致するメソッド” である．

形式的説明　オーバーロードに基づく注釈伝播のルール R_l では，メソッドを意味する識別子情報 e から，そのメソッドが宣言されているクラス内でオーバーロードとなっているすべてのメソッドを意味する識別子情報の列 E_{R_l} を求める $prule_{R_l}$ を用いて式 (20) に示す通り，

識別子情報 e を入力として，オーバーロードに基づく注釈伝播が行われ，DocumentTag の列が求まる．

$$\begin{aligned} L &= R_l(e) \\ &= [getd_{R_l}(e_1), getd_{R_l}(e_2), \ldots, getd_{R_l}(e_{n_e})] \\ &= [DocumentTag_{R_l 1}, DocumentTag_{R_l 2}, \ldots, DocumentTag_{R_l n}] \end{aligned} \tag{20}$$

なお，識別子情報 e の種類はメソッドである必要がある．e がメソッド以外の識別子情報であった場合や，オーバーロードとなっているメソッドがメソッドが宣言されているクラス内に存在しない場合は，L は空集合となる．

3.2.6 変数初期化子に基づく注釈伝播

初期化子（Initializer）には，クラス内でコンストラクタの他に初期化を行う初期化ブロック（Instance Initializer や Static Initializer）や，フィールドや局所変数の宣言文内で行われる代入を指す変数初期化子（Variable Initializer）がある．変数初期化子に基づく注釈伝播では，後者の変数初期化子に関して注釈伝播を行う．変数初期化子に基づく注釈伝播の対象となる識別子は，フィールドと局所変数のみであり，開発者がフィールドや局所変数の注釈を参照した場合，同時に参照することのできる識別子は，“宣言文の右辺式の最初の項の中で最後に評価される識別子” である．

最初の項とは，演算が行われていた場合は，その被演算子のうち，最初に評価される項を指す．括弧を用いて入れ子構造になった式についても，内部に演算子を持たない項単位で見た場合に最初に評価される項が対象となる．その最初に評価される項の中で，制御上，最後に評価される識別子のみを関連する識別子として注釈伝播を行う．最後に評価される識別子は，メソッドやコンストラクタ，フィールド，局所変数，仮引数のいずれか 1 つになる．数値や文字列定数などのリテラルは識別子ではないので注釈伝播は行われない．

このようにして同時に参照することのできる識別子はたかだか 1 つであり，同時に表示される個数を制限している．

この変数初期化子に基づく注釈伝播が行われることで，開発者が参照したフィールドや局所変数にはどのような変数やコンストラクタ，メソッドの戻り値，フィールドが代入されているかを把握することができる．とくに，コンストラクタの代入が行われている場合は，変数初期化子に基づく注釈伝播によって動的束縛の実際の型を把握することができる．具体的には図 2 で示したコード例ような，局所変数やフィールドの宣言と同時に動的束縛が行われている場合に，局所変数 lineElementList の注釈を参照した際に，変数初期化子に基づく注釈伝播によって初期化部分の ArrayList コンストラクタの注釈が同時に表示される．これにより実際に代入されて用いられる型が，ArrayList クラスであることを把握することができる．

形式的説明　変数初期化子に基づく注釈伝播のルール R_i では，フィールドや局所変数を意味する識別子情報 e から，その識別子が宣言されている文の右辺式の最初の項の中で最後に評価される識別子を意味する識別子情報の列 E_{R_i} を求める $prule_{R_i}$ を用いて式 (21) に示す通り，識別子情報 e を入力として，変数初期化子に基づく注釈伝播が行われ，DocumentTag の列が求まる．

$$
\begin{aligned}
L &= R_i(e) \\
&= [getd_{R_i}(e_1), getd_{R_i}(e_2), \ldots, getd_{R_i}(e_{n_e})] \\
&= [DocumentTag_{R_i 1}, DocumentTag_{R_i 2}, \ldots, DocumentTag_{R_i n}]
\end{aligned}
\tag{21}
$$

なお，識別子情報 e の種類はフィールドもしくは局所変数である必要がある．e がフィールドや局所変数以外の識別子情報であった場合や，初期化部分の最後に評価される識別子を意味するトークンが識別子ではない場合は，L は空集合となる．

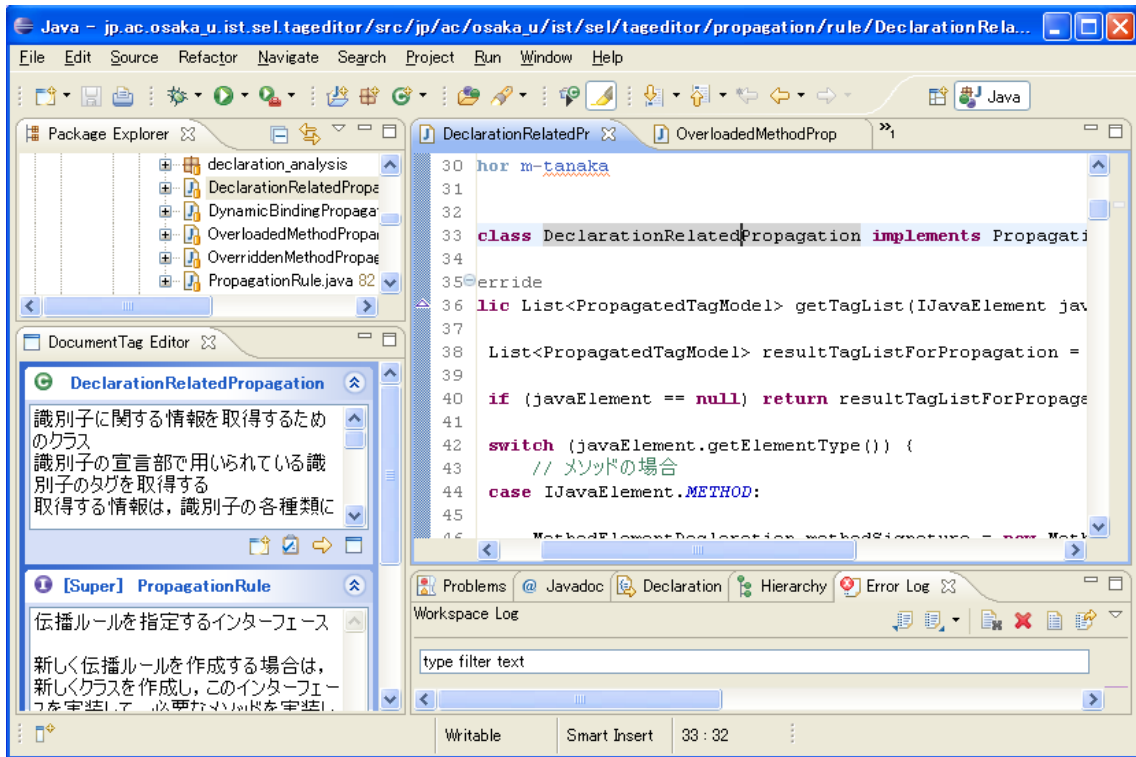

図 6: DocumentTag Editor を組み込んだ Eclipse

4 実装

本章では，提案した DocumentTag をツールとして実装した DocumentTag Editor について述べる．DocumentTag Editor は，統合開発環境 Eclipse の拡張機能である．開発者が Eclipse 上で識別子を選択することで，識別子に対応した注釈と，その識別子と関連のある識別子の注釈を同時に参照することができる．図 6 に DocumentTag Editor を組み込んだ Eclipse のスクリーンショットを示す．図 6 の左下で開かれているビューが本研究で実装した DocumentTag Editor である．テキストエディタ上の DeclarationRelatedPropagation クラスにカーソルを合わせており，DeclarationRelatedPropagation クラスに対応した注釈と，スーパークラスである PropagationRule クラスの注釈が同時に表示されている．

図 7 は，テキストエディタ上の getTagList メソッドを選択したときの図であり，DocumentTag Editor には getTagList メソッドの DocumentTag が 1 つ目に表示され，2 つ目以降には，注釈伝播によって，getTagList メソッドの型である List クラスや型パラメータである PropagatedTagModel クラス，仮引数とその型である仮引数 javaElement や IJavaElement クラス，スーパークラスの getTagList メソッドが，関連する注釈として同時に表示されてい

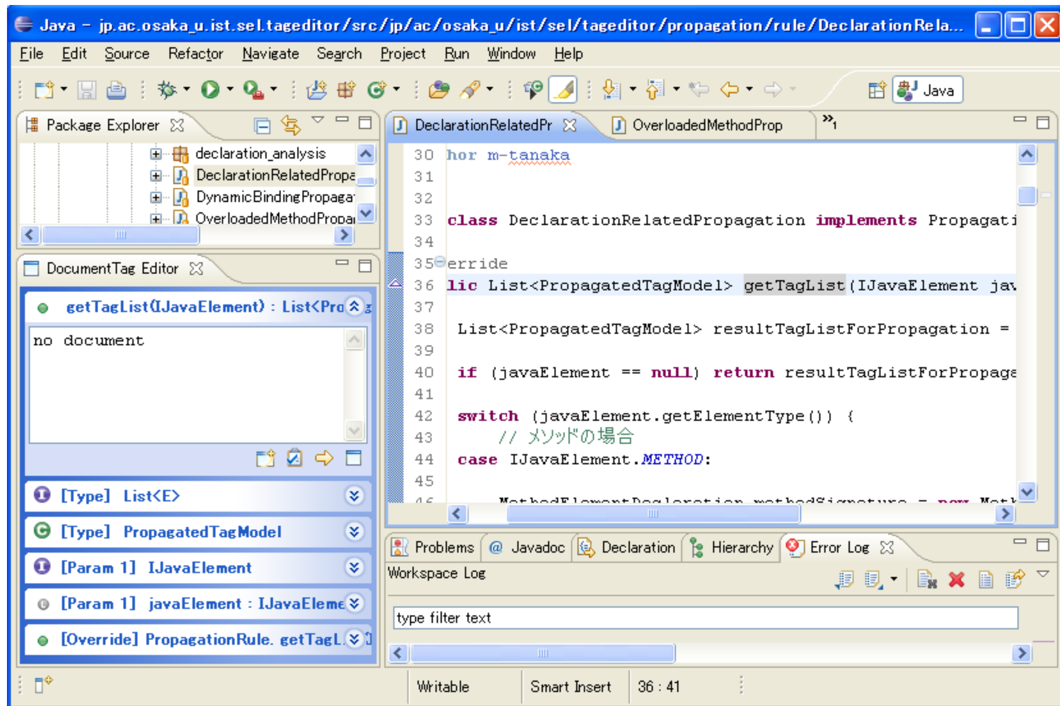

図 7: DocumentTag に記述がない場合

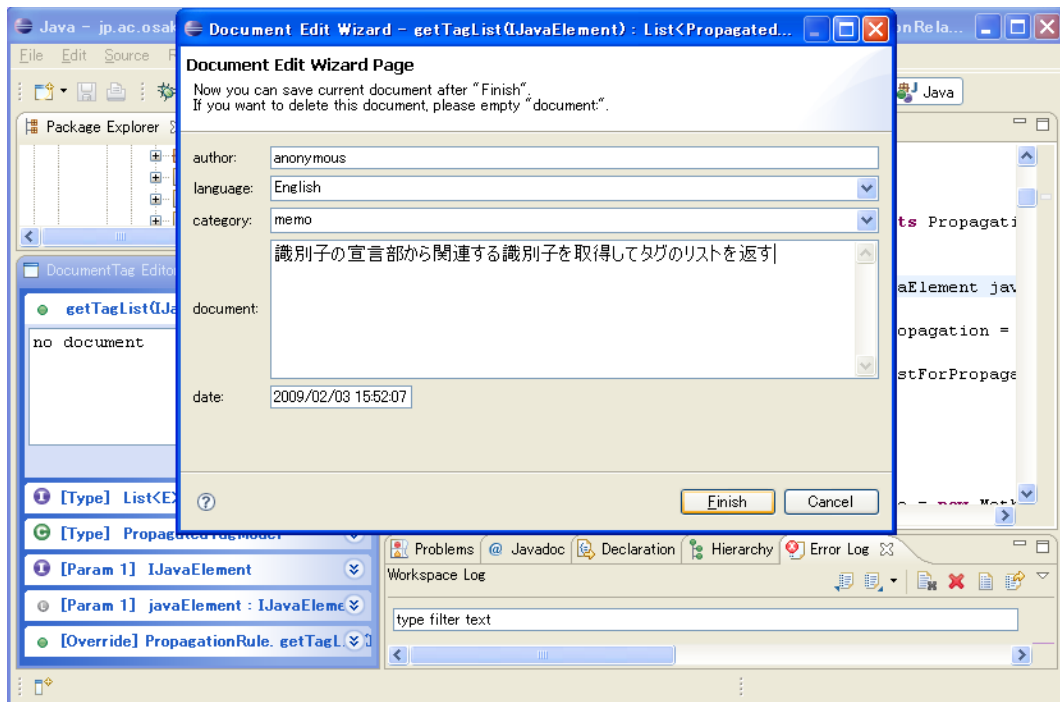

図 8: DocumentTag の編集画面

る．選択した識別子が，付加された注釈を持たない場合，図 7 のように「no document」という空の注釈が用意される．また，ソースコード中で Javadoc コメントが記述されている場合は，Javadoc の文章も DocumentTag の一種として用意される．注釈を編集するには，各 DocumentTag の表示領域にある編集ボタンを押し，図 9 に示すようなウィンドウを開く．3.1 節で示したデータ項目に対応しており，DocumentTag を記述する著者や使用する自然言語の種類，DocumentTag のカテゴリラベル，注釈の内容，記述した日時を指定することができる．

4.1　統合開発環境への統合

本節では統合開発環境への統合手段として用いた Eclipse プラグインの概要，および統合方法について述べる．

Eclipse プラグインの概要　統合開発環境 Eclipse の拡張機能は，一般に Eclipse プラグイン（Eclipse Plug-in）と呼ばれている．Eclipse に拡張機能を追加する場合は，Eclipse プラグインを Eclipse 上でプラグインプロジェクトとして作成し，作成したプラグインプロジェクトを指定の方法で Eclipse に組み込むと Eclipse に拡張機能が追加される．Eclipse のソースコードは，Java 言語で記述されているため，プラグインプロジェクトも Java 言語で作成される．また，Eclipse はすべて Eclipse プラグインで構成されており，Eclipse を起動するファイルである eclipse.exe ファイルを実行することで，指定のディレクトリにある Eclipse プラグイン（JAR ファイル）がすべて読み込まれ，機能が拡張された状態で起動される．作成した Eclipse プラグインも，対象のディレクトリに他の Eclipse プラグインと同様に配置することで，作成した拡張機能が追加された状態で起動させることができる．プラグインプロジェクトは，Java 言語で作成されるが，Eclipse で作成する Java プロジェクトとは異なり，plugin.xml という設定ファイルを保持している．この plugin.xml ファイルには，Eclipse プラグインとしての情報を記述する必要がある．プラグインを作成する上では，主に以下の項目について設定を行う．

- プラグインの概要：プラグイン名，プラグイン ID などを定める
- プラグイン依存関係：既存のプラグイン内の API を再利用するためにプラグインを指定する
- 拡張ポイント：作成するプラグインの種類を指定するために既存プラグインを指定する

1 つのプラグインプロジェクトには，拡張ポイントを複数指定することで，複数の拡張機能をもたせることができる．拡張機能 1 つに対して，その拡張機能を実現するためのクラスが

1 つ必要になる．クラスを作成し，必要なメソッドを実装し，Eclipse に組み込むことで，プラグインの種類に従って，実装した各メソッドが実行され，拡張機能が追加された Eclipse を構築することができる．

拡張ポイントの作成　本研究のプラグイン作成では，プラグイン名を DocumentTag Editor とし，Eclipse 上で DocumentTag Editor のビューを開くプラグインプロジェクトを作成するために，拡張ポイントとなる対象の Eclipse プラグインに org.eclipse.ui.editors プロジェクトを指定した．拡張ポイントを指定し，org.eclipse.ui.part.ViewPart クラスを継承した TagEditor クラスを作成し，必要なメソッドを実装し，Eclipse に組み込むことで，Eclipse にビューを開くプラグインを作成した．また，DocumentTag Editor の細かな設定を Eclipse 上で行えるようにするために，org.eclipse.core.runtime.preferences プロジェクトや org.eclipse.ui.preferencePages プロジェクトを指定した．これらの拡張ポイントを指定することで，空の注釈に用意する情報や，注釈の保存先などについて，Eclipse 上から設定ページを開き，各種設定を行うことを可能にしている．

Eclipse 内部のイベントの取得　開発者が Eclipse を操作する際に，Eclipse 内部ではさまざまなイベントが発生しており，これらのイベントに応じた処理を実装することにより，応答性の高い Eclipse プラグインを作成することができる．本研究で実装した DocumentTag Editor では，以下に説明するリスナを Eclipse に登録することで，Eclipse 上のイベントに応じた処理を行っている．まず，Eclipse 上で開発者が選択した時のイベントを取得することができるクラスとして，ISelectionListener クラスが定義されている．IWorkbenchPage クラスの変数から addPostSelectionListener メソッドを呼び出し，引数に ISelectionListener を実装したクラスのインスタンスを指定することで，テキストエディタ上で，開発者がカーソルを止めてから 500 ミリ秒後の瞬間，または，エディタ以外のビューで項目を選択した瞬間に，指定したインスタンスで実装されているメソッドが呼び出される．プラグインを終了する際には，終了する前に登録したリスナを，同じく IWorkbenchPage クラスの変数から removePostSelectionListener メソッドを呼び出すことで解除する必要がある．また，Eclipse 上で開発者が作業を行う中で，識別子の内容が変化したイベントを取得することができるクラスとして，IElementChangedListener クラスが定義されている．JavaCore クラスから addElementChangedListener メソッドを呼びだし，引数に IElementChangedListener を実装したクラスのインスタンスを指定することで，開発者がソースコードの編集により変化した識別子の情報とともに，指定したインスタンスで実装されているメソッドが呼び出される．実装するメソッド内では，変化した識別子情報や位置情報を得ることができ，抽象構文木上の変化に対応した処理を実装することができる．プラグインを終了する際には，ISelectionListener と同様に，JavaCore クラスから

removeElementChangedListener メソッドを呼び出して登録したリスナを解除する必要がある．

4.2 DocumentTag のデータモデル

1 つの DocumentTag の情報を表すクラスには，主に以下の情報をフィールドとして保持しており，DocumentTag の定義を反映している．

- IJavaElement 型の識別子情報
- 文字列型の注釈の記述内容
- 文字列型の注釈の種類
- 文字列型の注釈に使用する自然言語
- 文字列型の記録した日時
- 文字列型の著者
- 列挙型の注釈伝播の種類

識別子情報は，IJavaElement クラスのインスタンスを用いて表現している．IJavaElement クラスは，Eclipse 内部でコンパイルエラー検出や自動リファクタリングで Java プログラムの識別子情報を扱うために定義されたクラスであり，クラスやメソッドなどの各識別子に対応して，IJavaElement のサブクラスに IType クラスや IMethod クラスなどが定義されている．Java プログラムの識別子情報は，IJavaElement クラスのインスタンスとして Eclipse から取得することができ，DocumentTag が持つ識別子情報は，IJavaElement のインスタンスを保持することで実現している．また，IJavaElement クラスの識別子情報は，文字列表現に変換，または逆変換することができ，DocumentTag Editor では，1 つの DocumentTag をファイルに保存する際に，識別子情報を文字列表現に変換して保存している．具体的に，注釈情報とともに識別子情報をファイルに記録する場合には，IJavaElement のインスタンスから getHandleIdentifier メソッドを呼び出すことで，識別子情報を文字列表現が求まる．また，逆に文字列表現から識別子情報を求める場合には，JavaCore クラスの create メソッドの引数に文字列表現を指定することで，IJavaElemenet のインスタンスを求めることができる．ただし，3.1 節で示した通り，局所変数の場合には，識別子情報の文字列表現の一部に，プログラム上の位置情報が含まれており，開発者がプログラムを変更したときに，変更の前後で位置が変更された局所変数が，抽象構文木上は同一の識別子でも，識別子情報では同一と見なされず，その位置にある局所変数の注釈を参照できなくなることがある．この問題に対応するために，局所変数の識別子情報を扱う場合には，プログラム上の位置情報の比較を行わな

いことにしている．これにより，同一メソッド内で宣言されている，同型同名の局所変数は同一の識別子と見なされる．

注釈伝播の種類は，Java 言語での列挙型（enum）を用いて，伝播の種類と，識別子の種類から一意に求まるように網羅している．これらのフィールドを持ったクラスの 1 つのインスタンスを，1 つの DocumentTag として管理し，開発者に提示する DocumentTag の列は，配列構造である ArrayList クラスを用いて実現している．開発者が記録した DocumentTag は，開発者が作業しているワークスペースのディレクトリに隣接したディレクトリ内に，XML ファイル形式で保存される．記録された DocumentTag を読み出す場合は，対象の XML ファイルが読み込まれ，DocumentTag のインスタンスが生成される．

4.3　注釈伝播の実現方法

注釈伝播は，手法と同様に開発者の視点で行われるように実装されている．つまり，開発者が Eclipse 上で識別子を選択したというイベントに対して，識別子情報を基に，関連する識別子情報を求め，識別子情報の列から，DocumentTag の列を求める．関連する識別子情報は，Eclipse で提供されている API を利用して求めている．本研究で利用した以下の API について以降説明する．

- IJavaElement クラス
- ITypeHierarchy クラス
- ASTParser クラスおよび ASTVisitor クラス

まず，IJavaElement クラスは，識別子の種類を反映する形で多くのサブクラスが定義されている．そのサブクラスには，識別子に応じたメソッドが公開されており，サブクラスのメソッドを呼び出すことで，関連する識別子を求められる場合がある．本研究で実装した DocumentTag Editor では，メソッドを意味する IMethod クラスから，getDeclaringType メソッドを呼び出すことにより，メソッドの宣言されているクラスを求め，また，クラスを意味する IType クラスから，宣言されているメソッドの一覧やシグネチャの一致するメソッドを求めるメソッドが提供されており，これを基に，オーバーロードの関係にあるメソッドの識別子情報を求めている．また，IType クラスからは，スーパークラスのシグネチャを文字列表現で取得することができ，この文字列表現を IJavaProject クラスの findType メソッドの引数に指定して呼び出すことにより，スーパークラスの識別子情報を求めている．

ITypeHierarchy クラスは，クラス階層を反映したものであり，IType クラスから newSupertypeHierarchy メソッドなどを呼び出すことによりインスタンスを生成することができる．この場合は，IType クラスから見てスーパークラスとなるクラスに対してクラス階層を求める

ことができ，このクラス階層を辿ることにより，オーバーライドされているメソッドの識別子情報を求めている．

最後に ASTParser クラスおよび ASTVisitor クラスは，プログラムに対して構文解析および意味解析を行い，生成された抽象構文木に対して深さ優先探索を行うために提供されている．ASTParser クラスは，ファイルまたは，IJavaElement のサブクラスである ICompilationUnit クラスや IClassFile クラスに対して，構文解析を行い，抽象構文木（AST）を生成するクラスである．ASTParser のインスタンスから setResolveBindings メソッドを，引数に true を指定して呼び出すことで名前解決，すなわち意味解析を行った AST を生成することができる．生成された AST の識別子を意味する各ノードには，IJavaElement などの識別子情報が保持されており，識別子の型を意味する識別子情報などを得ることができる．注釈伝播の実装には，この AST のノードから得られる情報を基に，メソッドやフィールド，局所変数，仮引数の宣言部分の型や，メソッドの仮引数，メソッドの例外の型，変数の初期化部分の識別子情報を求めている．AST 上から，識別子と対応するノードを求める際に，ASTVisitor クラスを用いている．このクラスは，Visitor パターン [10] で実装されており，ASTVisitor クラスを継承したクラスを探索に使用することで，AST のノードの種類に応じた処理を行うことができる．メソッドやフィールド，局所変数，仮引数それぞれについて，宣言部を意味するノードに到達したときに，そのノードから，関連する識別子情報を求めている．

5　実験

本章では DocumentTag を実装した DocumentTag Editor を用いて行った，適用実験について説明する．具体的には学生 4 人に対して，Java 言語で書かれた既存のプログラムの拡張作業を実施し，作業時間などを評価した．その結果，注釈伝播によって作業時間の平均が短縮された．また，被験者に対して行ったアンケートから，注釈伝播が識別子間の関連を把握する上で効果的であったことが確認できた.

具体的には，DocumentTag Editor 自身のソースコードに対して，拡張課題を 2 つ用意し，それぞれを課題 a，課題 b として学生 4 人に順序を入れ替えて作業を行わせた．この実験では，本研究の手法である注釈伝播が保守作業に有効であるかを評価するため，3 章で述べた注釈伝播をすべて有効にした DocumentTag Editor を用いて作業した場合と，関連する識別子への注釈伝播を無効にした DocumentTag Editor を用いて作業した場合の作業時間を比較した．そのため，保守作業を対象にした適用実験での，学生に対する課題の割り当てや条件の割り当ては表 1 のように行った.

実験の課題説明に用いた資料は，Web サイトに配備し，DocumentTag Editor のインストールなど実験環境の準備は，Web 上の資料に従って行うこととした．また作業を行うにあたって，Eclipse の操作方法や，課題の文章の意味を確認するための質問に対応するために，実験中は巡回して各被験者に応対した．ソースコードに関する質問に対しては，一切回答を行わないこととした．課題内容の詳細については付録に示す.

また，被験者が課題 a，課題 b の作業を行う中で，課題 1 回目に比べて課題 2 回目の作業には，開発者の学習効果が影響することが想定されるため [8]，課題 1 回目の作業時間と課題 2 回目の作業時間とを別々に検定を行い，注釈伝播を有効にすることで作業時間が短縮されるかどうかを評価した.

表 1: 学生の課題の割り当て表

保守作業を行った被験者	課題 1 回目	課題 2 回目
被験者 A	課題 a（注釈伝播有効）	課題 b（注釈伝播無効）
被験者 B	課題 b（注釈伝播有効）	課題 a（注釈伝播無効）
被験者 C	課題 a（注釈伝播無効）	課題 b（注釈伝播有効）
被験者 D	課題 b（注釈伝播無効）	課題 a（注釈伝播有効）

5.1　作業時間の比較

表 1 に従って，学生 4 人を被験者 A，B，C，D に割り振り，各課題 a，b を実施したところ，各課題の作業時間は表 2 に示す結果となった．

なお，各課題 a，b の作業開始は，Eclipse を起動して Eclipse のウィンドウが開いた時刻とし，課題が成功したといえる条件として動作確認を行い，課題を満たす動作をしたと確認して Eclipse を終了した時刻を終了時刻とした．

表 2 に示すように注釈伝播が無効の場合よりも有効の場合の方が，平均作業時間が短くなるという結果が得られた．しかし作業時間にばらつきがあり，とくに被験者 A や被験者 D のように，課題 1 回目に行った課題の方が課題 2 回目に行った課題の方よりも大きく時間がかかることが分かった．

ここから，課題の順序による影響を考慮して検定を行うために，表 3 に示すように，課題 1 回目の作業時間と，課題 2 回目の作業時間とで実験結果を 2 つのグループに分割した．ここからそれぞれの場合で検定を行った結果が，表 4 である．

表 4 では，注釈伝播有効の場合の方が，注釈伝播無効の場合に比べて作業時間が短くなるという帰無仮説が棄却されないかどうかを t 検定を用いて評価を行った．

t 検定を行うには，事前条件として t 検定を行う対象の分布が正規分布かつ等分散である必要がある．そこで実験結果の数値の分布に対して，KS 検定（Kolmogorov-Smirnov test）を行って正規分布であることを確かめ，F 検定を行って等分散であることを確かめた．KS 検定では，算出される有意確率が 0.05 より大きい場合に，正規分布であるという帰無仮説が棄却されないので，正規分布であると示すことができる．また F 検定では，算出される有意確率が 0.05 より大きい場合に，等分散であるという帰無仮説が棄却されずに，等分散であることを示すことができる．また，t 検定は，2 つの数値の分布間の有意差を検定するもので，

表 2: 各被験者ごとの保守作業にかかった時間

保守作業を行った被験者	作業時間（分）	
	注釈伝播有効	注釈伝播無効
被験者 A（課題 a：有効．課題 b：無効）	131	77
被験者 B（課題 b：有効．課題 a：無効）	91	98
被験者 C（課題 a：無効．課題 b：有効）	63	87
被験者 D（課題 b：無効．課題 a：有効）	45	165
平均時間	82.5	106.75

表 3: 課題の順序を考慮して分割したグループ

課題 1 回目の作業時間（分）		課題 2 回目の作業時間（分）	
注釈伝播有効	注釈伝播無効	注釈伝播有効	注釈伝播無効
$\{131, 91\}$	$\{87, 165\}$	$\{63, 45\}$	$\{77, 98\}$

表 4: 課題の順序を考慮して分割したグループごとの検定結果

検定の種類	有意確率
注釈伝播有効の場合の KS 検定（$>$ 0.05 で正規分布）	0.9992
注釈伝播無効の場合の KS 検定（$>$ 0.05 で正規分布）	0.9992
F 検定（$>$ 0.05 で等分散）	0.3017
t 検定（$<$ 0.05 で有意差あり）	0.3824

(a) 課題 1 回目の検定結果

検定の種類	有意確率
注釈伝播有効の場合の KS 検定（$>$ 0.05 で正規分布）	0.9992
注釈伝播無効の場合の KS 検定（$>$ 0.05 で正規分布）	0.9992
F 検定（$>$ 0.05 で等分散）	0.4511
t 検定（$<$ 0.05 で有意差あり）	0.0682

(b) 課題 2 回目の検定結果

本研究で行ったt検定では，注釈伝播無効の場合に比べて注釈伝播有効の場合の方が，作業時間が短くなるという帰無仮説が棄却される確率を，有意確率として算出している．つまり本研究で行った検定の場合，t検定によって算出された有意確率の割合で，注釈伝播有効の場合の方が作業時間が長くなるという読み方ができる．この有意確率は一般的には0.05未満であるとき，有意差があると断定できる．

表4(a)に示す課題1回目の場合は，KS検定，F検定ともに0.05よりも大きく，t検定が行える分布であるといえる．t検定の結果では，1から有意確率を引いた0.6176の確率で注釈伝播有効の場合の方が，作業時間が短くなるといえるが，有意水準である0.05よりも有意確率が大きいため，作業時間に有意差があるとはいえなかった．表4(b)に示す課題2回目の場合も同様に，正規分布かつ等分散であるため，t検定が行える分布であるといえる．t検定の結果では，0.9318の確率で，注釈伝播有効の場合の方が，作業時間が短くなるといえるが，有意水準である0.05よりも有意確率大きいため，作業時間に有意差があるとはいえなかった．

5.2 アンケート結果

本節では課題を終えた学生4人に対して行ったアンケート結果について述べる．

アンケートでは，作業課題を終えた学生に，以下に挙げる内容について質問を行った．アンケートの回答を集計したところ，表5のような結果となった．

- DocumentTag Editorを使用する利点について
  - 課題a，課題b，課題全体それぞれについて利点があれば回答する
- DocumentTag Editorの改善点について
- 作業課題を行う中で時間の要した作業について
  - コーディング・課題の理解・プログラム理解・バグ修正・Eclipse上の操作のうち，時間のかかった順に覚えている範囲で回答する

表5(a)から，DocumentTag Editorを用いて関連する注釈も参照できる環境が有効であるという回答が多く得られた．しかし，表5(b)から，ツールとしての使いやすさに改善の余地があることが分かった．

また，時間のかかった作業として表5(c)，5(d)，5(e)と表5(f)，5(g)，5(h)とを比較すると，課題1回目と課題2回はともにプログラム理解に多く時間を要していることが分かった．また，課題1回目では課題2回目に比べ，とくにコーディングに時間を要する場合があることが分かった．

表 5: アンケート集計結果

項目	回答者
Eclipse 上で識別子を選択するだけで注釈を参照できる点	A,B,C,D
あるクラスを選択するとスーパークラスの仕様も同時に参照できる点	A,D
あるメソッドを選択すると各引数の仕様も同時に参照できる点	C
ソースコードに書き込まずにメモを残せる点	C

(a) DocumentTag Editor を使用する利点

項目	回答者
関連する注釈を閲覧する操作を簡単にしてほしい	B,D
注釈伝播の拡張を別プラグインから拡張できるようにしてほしい	C

(b) DocumentTag Editor の改善点

作業の種類	回答者
プログラム理解	A,D
コーディング	B
バグ修正	C

(c) 課題 1 回目で最も時間を要した作業

作業の種類	回答者
プログラム理解	C
コーディング	A
課題の理解	D
バグ修正	B

(d) 課題 1 回目で 2 番目に時間を要した作業

作業の種類	回答者
Eclipse の操作	D
バグ修正	A
課題の理解	C

(e) 課題 1 回目で 3 番目に時間を要した作業

作業の種類	回答者
プログラム理解	A,C,D
バグ修正	B

(f) 課題 2 回目で最も時間を要した作業

作業の種類	回答者
プログラム理解	B
バグ修正	A
課題の理解	D

(g) 課題 2 回目で 2 番目に時間を要した作業

作業の種類	回答者
Eclipse の操作	D
コーディング	A

(h) 課題 2 回目で 3 番目に時間を要した作業

6　考察

注釈伝播による作業時間の短縮　本研究で行った適用実験では，他識別子からの注釈伝播の機能が無効になっている場合と比べて，注釈伝播を有効にした場合の方が平均作業時間の短縮が見られた．アンケート結果にも挙げられた通り，注釈伝播によってスーパークラスやメソッドの引数の情報を効果的に理解できたことが理由であると考えられる．

被験者の学習効果　被験者 A，D の作業時間に顕著に見られるように，注釈伝播の効果よりも，課題の順序が作業時間に大きく影響する結果となった．これは，Eclipse プラグインというドメインの知識を学習したことによる効果が考えられる [8]．課題 a と課題 b で対象となるソースコードの範囲は，互いにメインクラス以外は直接の依存関係を持たないので，対象となったプログラムの知識よりも，コードの書き方を知るために多く時間を要したものと考えられる．実際にアンケート結果では，課題 1 回目でコーディングに多く時間を要した被験者が 2 名見られる．その被験者の回答では，どう書いていいのか分からなかったという理由が 2 名ともに挙げられた．課題 1 回目を終えることによって，コーディングによる拡張方法を学習し，課題 2 回目では大幅にコーディングの時間を短縮したものと考えられる．

被験者の学習効果を除外した注釈伝播の効果　被験者の学習効果による影響が大きいため，作業時間の結果を課題 1 回目の作業時間と課題 2 回目の作業時間にグループ分けを行った．それぞれの結果に対して，注釈伝播を有効にすることで，作業時間に有意差があるかを検定したところ，課題 1 回目と課題 2 回目のいずれも，有意水準の 0.05 を下回ることはなく，有意差があるとはいえなかったが，課題 1 回目に比べて，課題 2 回目の有意確率は有意水準に近い値となった．これは，アンケート結果より，課題 1 回目，課題 2 回目ともに，プログラム理解に多く時間を要す結果となったが，課題 1 回目では，コーディングに大きく時間を要した被験者が多く，注釈伝播の効果が大きく現れなかったことが考えられる．課題 2 回目では，学習効果により，プログラム理解に大きく時間をかける作業になったため，注釈伝播の効果が，有意確率として作業時間により有意にはたらく結果となったものと考えられる．今回行った適用実験は 4 名の学生を対象としたものであり，データ数が少なく，統計上の検定では，作業時間に有意差があるとはいえなかったが，実験の被験者の人数を増やすことにより，有意差を示すことができるのではないかと考えられる．

7　あとがき

本研究では，オブジェクト指向プログラミングを導入することで，プログラム内の識別子間の依存関係が複雑になることや，開発者が理解した内容を効果的に記録・再利用する手段がないことを問題として，付加注釈 DocumentTag を提案し，識別子に対応して注釈を記録できる環境を DocumentTag Editor として実装した．また，DocumentTag がもつ注釈伝播によって，識別子間の関連を把握しながら，効果的なプログラム理解ができる環境を構築した．

適用実験として，DocumentTag Editor を用いて，注釈伝播が行われることでプログラム理解が効果的になることを作業時間を基に評価した．その結果，統計的な有意差は確認できなかったが，平均作業時間が短縮される結果となった．しかし，実装したツールの使いやすさに改善の余地があり，開発者にとってより使いやすいツールにすることで，注釈伝播の効果がより向上されるものと思われる．

今後の課題としては，本研究で提案・実装した注釈伝播では，構文的な識別子間の関連のみを扱ったが，より開発者にとって有用となる関連について，注釈伝播を行って開発者に提示することが挙げられる．たとえば，デザインパターン検出ツール [23] を併用し，あるクラスやメソッドの注釈を開発者が参照したときに，デザインパターン上で関連のあるクラスやメソッドについての注釈を，デザインパターンの用語を交えて開発者に提示することで，より効果的なプログラム理解を行えることが期待できる．また，比較的計算量の小さいソースコード解析手法 [12][22] を適用し，動的束縛された変数の，プログラム実行時の型や，実際に呼び出されるメソッドを求め，スーパークラスの変数やメソッドの注釈を参照したときに，実際に代入されている型や呼び出されるメソッドについての注釈が参照できることで，実行時のプログラム把握がより効果的になると考えられる．また，定数伝播 [2] と同様にデータフロー関係に従って注釈伝播を行うことにより，値の説明を意味した注釈が，データの流れに従って伝わり，値の矛盾を導くことで検証やデバッグに利用できるのではないかと考えられる．加えて，プログラムの実行履歴 [25] として，値の変更履歴や呼び出し順序を記録するシステムとして付加注釈を利用することも考えられる．この場合は，識別子に対応した注釈ではなく，プログラム上の各トークンに対して付加する注釈となることが予想される．メソッドの戻り値や，変数の値の変化を注釈として提示することで，プログラム理解やデバッグに有効に利用できると考えられる．また，大規模ソースコードに対して行うデータフロー解析や動的解析の結果は膨大であり，注釈として提示する場合には，開発者にとって扱いやすい表現に縮約する工夫を行うことも課題として挙げられる．

また，以上のように DocumentTag の用途および注釈伝播を拡張することで，開発者側に多数の注釈が同時に提示されてしまい，ツールとしての操作性や注釈の管理に支障をきたす

ことが考えられる．これに対して個々の注釈の順位付けやフィルタリングを行うことも課題として挙げられる．

謝辞

本研究の全課程を通して，常に適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に心より深く感謝致します.

本研究を通して，適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠准教授に深く感謝申し上げます.

本研究を通して，常に適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻石尾隆助教に深く御礼申し上げます.

最後に，本研究にあたって，有意義な御指導，御助言および，適用実験へのご協力を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻ソフトウェア工学講座井上研究室の皆様に深く感謝致します.

参考文献

[1] Paul Anderson, Thomas Reps, and Tim Teitelbaum. Design and implementation of a fine-grained software inspection tool. *IEEE Trans. Softw. Eng.*, Vol. 29, No. 8, pp. 721–733, 2003.

[2] S. Blazy and P. Facon. Partial evaluation as an aid to the comprehension of fortran programs. In *Proceedings of the 2nd IEEE Workshop on Program Comprehension (IWPC '93)*, pp. 46–54, Capri, Italy, 1993.

[3] Joshua Bloch. *Effective Java (2nd Edition)*. Prentice Hall, 2008.

[4] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Chapter 5.1 Tie Code and Questions. Square Bracket Associates, 2008. `http://www.iam.unibe.ch/~scg/OORP/`.

[5] Michael Desmond, Margaret-Anne Storey, and Chris Exton. Fluid source code views for just in-time comprehension. In *Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT '06)*, Bonn, Germany, 2006.

[6] Alastair Dunsmore, Marc Roper, and Murray Wood. Practical code inspection techniques for object-oriented systems: An experimental comparison. *IEEE Software*, Vol. 20, No. 4, pp. 21–29, 2003.

[7] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, Vol. 40, No. 10, pp. 32–38, 1997.

[8] Thomas Fritz, Gail C. Murphy, and Emily Hill. Does a programmer's activity indicate knowledge of code? In *Proceedings of the the 6th joint meeting of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07)*, pp. 341–350, Dubrovnik, Croatia, 2007.

[9] E.M. Gagnon and L.J. Hendren. SableCC, an object-oriented compiler framework. In *Proceedings of the 26th International Conference on Technology of Object-Oriented Languages (TOOLS '98)*, pp. 140–154, Santa Barbara, CA, USA, 1998.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns. Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[12] Ákos Hajnal and István Forgács. A precise demand-driven def-use chaining algorithm. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR '02)*, pp. 77–86, Budapest, Hungary, 2002.

[13] R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software*, Vol. 52, No. 2-3, pp. 173–179, 2000.

[14] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pp. 159–168, Chicago, Illinois, USA, 2005.

[15] Moises Lejter, Scott Meyers, and Steven P. Reiss. Support for maintaining object-oriented programs. *IEEE Trans. Softw. Eng.*, Vol. 18, No. 12, pp. 1045–1052, 1992.

[16] P. S. Newman. Towards an integrated development environment. *IBM Systems Journal*, Vol. 21, No. 1, pp. 81–107, 1982.

[17] C. Oezbek and L. Prechelt. JTourBus: Simplifying program understanding by documentation that provides tours through the source code. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM '07)*, pp. 64–73, Paris, France, Oct. 2007.

[18] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, Vol. 15, No. 12, pp. 1053–1058, 1972.

[19] Robert Simmons. *Hardcore Java*. O'Reilly & Associates, 2004.

[20] D. Spinellis. Abstraction and variation. *IEEE Software*, Vol. 24, No. 5, pp. 24–25, Sept.-Oct. 2007.

[21] M.-A. Storey, L.-T. Cheng, J. Singer, M. Muller, D. Myers, and J. Ryall. How programmers can turn comments into waypoints for code navigation. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM '07)*, pp. 265–274, Paris, France, Oct. 2007.

[22] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pp. 264–280, Minneapolis, Minnesota, USA, 2000.

[23] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Softw. Eng.*, Vol. 32, No. 11, pp. 896–909, 2006.

[24] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Trans. Softw. Eng.*, Vol. 18, No. 12, pp. 1038–1044, 1992.

[25] 谷口考治, 石尾隆, 神谷年洋, 楠本真二, 井上克郎. プログラム実行履歴からの簡潔なシーケンス図の生成手法. コンピュータソフトウェア, Vol. 24, No. 3, pp. 153–169, 2007.

付録

適用実験の課題内容

課題 a の課題説明　注釈伝播ルール (シグネチャに基づく注釈伝播、オーバーロードに基づく注釈伝播など) は Strategy パターンで構成されている．新たに注釈伝播ルールを追加する場合には、新たにクラスを作成して，あるメソッドをオーバーライドし、そのクラスをあるメソッド内でインスタンス化してオーバーライドしたメソッドを呼び出す必要がある．

そこでこの課題では既存の注釈伝播ルールである「オーバーライドに基づく注釈伝播」(あるメソッドを選択したときに，そのメソッドがオーバーライドしているメソッドのタグを伝播する) を実装したコードを例に,「被オーバーライドに基づく注釈伝播」 (あるメソッドを選択したときに，そのメソッドをオーバーライドしているメソッドのタグを伝播する) を実装し，注釈伝播ルールとしてプラグインに追加することを行う．

なお，この場合の”オーバーライドしているメソッド (overriding method)”とは，オーバーライドされているメソッド (overridden method) が宣言されている型 (declaring type) から見たすべてのサブクラス (正しくはサブタイプ) 内で宣言されているメソッドの中で，シグネチャ(メソッド名と引数の型リスト) が一致するものを指す．

逆に，既存の注釈伝播ルールとして実装されている”オーバーライドされているメソッド”では，オーバーライドしているメソッドが宣言されている型から見たすべてのスーパークラス (正しくはスーパータイプ) で宣言されているメソッドの中で，シグネチャ(メソッド名と引数の型リスト) が一致するものを取得している．

また，この拡張を行う中でこのクラス以外にも新たに要素を追加したりすることが必要になるが，その際の名前の付け方などは自由に決めて実装してよい，

なお，課題の作業量として作成者自身が実装した場合，簡単な試用も含めて 30 分もかからない程度の作業である．既存の実装を流用することができ，自分で書くコードは全部でも 10 行程度である．

課題 b の課題説明　現在のツールで用いている GUI は，SWT の ExpandBar(グラデーションのある青い部分のウィジェット) の子要素に複数の ExpandItem(開いたり閉じたりできる水色っぽいウィジェット) をタグの数だけ配置して複数のタグを表示している．その ExpandItem の子要素にさらにテキストボックスやボタンなどを配置して，1 つのタグについての情報や機能を提供している．

本ツールでは ExpandItem の子要素にどのようなウィジェットを配置するのかを Strategy パターンを用いて構成できるようになっており，タグの種類に応じて配置するウィジェットを

選択して切り替えることができる．また，現在のツールの補足機能として，利用者がタグを記録していない場合でもソースコード上のドキュメントである Javadoc の内容をデフォルトで表示することができる．

ところが，現在 ExpandItem に配置しているウィジェットには，編集ウィザードボタンが含まれており，これはデフォルトで表示するタグには必要ないウィジェットである．(実際デフォルトで表示するタグの各属性にはデフォルトの内容が記録されているので，ツールを利用する上での問題はないが，編集ウィザードボタンの機能が新規作成ウィザードボタンとして機能することになってしまい，ツールとしての一貫性が損なわれる)

そこでこの課題では，表示するタグの情報がデフォルトで表示しているものである場合には，編集ウィザードボタンを配置しないウィジェットで表示できるようにし，編集ウィザードボタンはデフォルトでないタグ (開発者が作成したタグ) でしか表示されないようにする実装を行う．

課題で行う作業では，あるクラスを継承したクラスを新たに作成して各メソッドをオーバーライドし，ウィジェットをスイッチするコードの部分を書き換える必要がある

なお，課題の作業量として，作成者自身が実装した場合，簡単な試用も含めて 30 分かかる程度の作業であるので，大規模な実装は行わない．既存の実装を流用することができ，自分で考えて書くコードは全部でも 10 行程度である．