情報処理学会論文誌 Vol.0 No.0 ??–?? (??? 1959)

Refactoring Script：複合リファクタリングを適用可能なリファクタリングスクリプトと処理系

概要：リファクタリングはコード体質改善の手法として広く知られているが，手動での実行はコストが高く欠陥を埋め込みやすいため，リファクタリングツールが多数提案されている．しかし，これらのツールは単体の単純なリファクタリングの実行を支援するものであり，リファクタリングによるデザインパターンの導入など，複雑なリファクタリングを行うのは難しい．すなわち，単体のリファクタリングを複数種類組み合わせて逐次実行したり，複数箇所に対してあるいは複数回数繰り返してリファクタリングを実行することは困難である．そこで我々は，Java ソースコードを表現可能なモデルを用いて，リファクタリング内容やその適用箇所の指定を記述できるスクリプトおよびその処理系を提案する．複雑なリファクタリングを簡潔に記述でき，少ないコストで複雑なリファクタリングを実行できること，またプロジェクト横断的に再利用できることを評価実験で確認し，本手法の有用性を示した．

キーワード：リファクタリング，コード操作

Refactoring Script: A script for composite refactoring and its processor

Abstract: Refactoring has been recognized widely as the way to improve the internal qualities of source codes. Because manual refactorings is time-consuming and error prone, many tools supporting automated refactoring have been suggested. However, because these tools are only for supporting a unit and simple refactoring, it is difficult to preform complicated refactorings such as a introduction of a design pattern. That is, it is difficult to apply a set of combined refactorings or to apply refactorings multiple times to multiple positions. In this research, we propose the script language and its interpreter that can describe how and where to refactor by using a model expressing source codes. From the results of evaluations, we concluded that our language and interpreter allow users to describe the steps of refactorings as scripts, replay and reuse them simply for multiple projects.

Keywords: Refactoring, Code manipulation

1. はじめに

リファクタリングとは，「ソフトウェアの外的振る舞いを保ったままで，内部の構造を改善してゆく作業」?と定義され，近年広く認知され，実践されている．リファクタリング手法のうち頻繁に用いられるものは，**フィールド名の変更**や**メソッドの抽出**などに代表されるようにパターンとしてまとめられている?が，手動による適用はコストが高く，欠陥を埋め込み易くなる．それを解決するため，リファクタリングの実行を支援，自動化するツールや手法が多数提案されている．

表??に主要な統合開発環境 (Integrated Develop Environment; IDE) がサポートするリファクタリング機能の数をまとめる．例えば Eclipse IDE? は Java 言語向けに，23

種類のリファクタリングを提供しており，Visual Studio? 向け拡張ライブラリである ReSharper? は C# 言語向けに，31 種類のリファクタリング機能を提供していることを表す．

表 1 主要な IDE がサポートするリファクタリングの数

IDE	言語	リファクタリング数
Eclipse 3.7	Java	23
	Scala	5
Visual Studio 2010	C#	6
ReSharper 6	C#	31
IntelliJ IDEA 12	Java	30
NetBeans 7.2.1	Java	23
XCode 4.2	Objective-C	8

ここで，リファクタリングの種類について，以下の 2 つの用語を定義する．

原始リファクタリング それ以上分解できないような小さく単純なリファクタリング単体を指す?．本論文ではとくに，Eclipse の標準機能として予め備わっているリファクタリングを指すことにする．

複合リファクタリング 原始リファクタリングの組み合わせによって構成される複雑なリファクタリングを指す?, ?．ここで組み合わせとは，(a) 特定箇所に複数種類のリファクタリングを組合せること，(b) 複数箇所にあるリファクタリングを適用すること，または (c) (a) と (b) の組み合わせを指す．

表??に示した IDE をはじめ，現在提案されているツールの多くは，予め搭載された原始リファクタリングの自動実行を支援するものであり，複合リファクタリングを定義，適用する仕組みはない．

Vakilian ら?は，各リファクタリングツールにはその動作を詳細に設定することで，より目的に沿ったリファクタリングを実現することができるが，設定ダイアログはコーディング作業の妨げになりオーバーヘッドを生み，結果的に生産性を落とす可能性があると報告した．また，Mens ら?は，オプションを細かく設定できても，対象のドメインにマッチするような設定や拡張は現状のツールでは不十分であると報告た．したがって，詳細なリファクタリングオプションの指定作業を簡素化し，簡潔にリファクタリングを実行できるインターフェースが必要であると考えられる．

Vakilian ら?は，Eclipse のリファクタリング操作の記録からその傾向を調査してまとめた．これによると，原始リファクタリングを組み合わせて，複合ファクタリングを実現する場面が多く存在する．

リファクタリングによりデザインパターンを導入する手法が，?にまとめられている．

複合リファクタリングを実行するには，開発者が適用の度に目的の場所をマウスやキーボードを利用してリファクタリング機能を呼び出す必要があり，実行コストが高い．また，適用箇所と適用内容が多くなるほど，そのすべてを正確に実行するのは困難であり，適用漏れの可能性がある．

さらに，リファクタリングに関してパターンが形成されているにもかかわらず，多くのツールではリファクタリングの適用内容を記録して再利用できない．

したがって，頻繁に用いるような複合リファクタリングを，プロジェクト横断的に適用することが困難である．Eclipse には，リファクタリングの内容をスクリプトとして記録，再生をする機能があるが，これはライブラリ配布の際に古いバージョンを使用している開発者のバージョンアップ作業を支援するものである?．この記録を任意に作成して自由にリファクタリングのステップを記述することはできない．

そこで我々は，リファクタリング内容を記述するためのスクリプトおよび処理系を提案する．

本論文では，以下の 4 点を研究課題とする．

RQ1 リファクタリング操作（適用箇所と適用内容）を簡潔かつ正確に記述でき，またそれを適用できるか？

RQ2 ツールを利用しない場合と比べて，複合リファクタリングを正確に実行できるか？

RQ3 ツールを利用しない場合と比べて，複合リファクタリングを実行するコストを軽減できるか？

RQ4 プロジェクト横断的にリファクタリング操作を再利用できるか？

本論文による貢献は以下の通りである．

- リファクタリング操作を記述するために RefactoringScript 言語を提案した
- 記述したリファクタリング操作を適用する方法としての RefactoringScript 処理系を開発した
- RefactoringScipt 言語及び処理系を Eclipse プラグインとして作成することで，広く利用可能とした
- 作成した RefactoringScript 言語および処理系に対し，その実用性を評価した

また，RefactoringScript 言語，処理系のリファクタリング機能は Eclipse JDT を用い，スクリプトのインタープリタには JRuby を用いることなどによって，低コストで実装することが出来た．

本論文の以降の構成は次のとおりである．??節で本論文の動機付けの例を示す．??節で，RefactoringScript 言語および処理系を提案し，設計と構成要素について詳細に述べる．??節で，評価実験の結果と考察を示す．??節で，リファクタリングの組み合わせや，スクリプトによるリファクタリングの記述に関する関連研究を述べる．最後に??節で結論を述べる．

2. 動機付け

本節では， 複合リファクタリングが必要な動機付けの例

として，(i) 関連する名前の変更，(ii) コーディング規約の適用，(iii) デザインパターンの導入の 3 場面を考える．

2.1 関連する名前の変更

?によると，**フィールド名の変更**を実行した後，関連する要素の名前を変更するという組み合わせが高頻度で行われる．例えば，リスト??のように，**フィールド名の変更**した後，そのフィールドのアクセサの名前も変更（**メソッド名の変更**）する場合考えられる．

リスト 1 フィールド名と対応するアクセサの名前を変更する例（上：元のソースコード，中：フィールド名の変更後，下：関連するアクセサ名の変更後）

```
private int page;
public int getPage(){
  return page;
}
```

```
private int pageCount;
public int getPage(){
  return pageCount;
}
```

```
private int pageCount;
public int getPageCount(){
  return pageCount;
}
```

既存のリファクタリングツールにおける**フィールド名の変更**は，定義とその参照の変更しか行わない．例えば，リスト??のフィールド page の名前を pageCount に変更するリファクタリングをしても，対応するアクセサの名前は getPage のまま変化しない．開発者は別途**メソッド名の変更**を実行する必要がある．

2.2 コーディング規約の適用

プロジェクトには，コーディング規約（ソースコードを作成する際のルール）が存在することがある．とくにチーム開発においては，開発メンバ間でコーディング規約を定めておくことでコード全体の保守性を高められるため，開発メンバは規約を遵守することが求められる．

? や ?は，基本となる規約をまとめたもので，自由に改変および利用することが可能である．例えば，「(27) private, protected なフィールドの名前の接頭辞や接尾辞にはアンダースコアをつける」や，「(44) メソッドのオーバーロードは避ける」などは多くのプロジェクトで採用されている規約である．

既に規模が膨らんだプロジェクトに対して，この規約を適用するには，以下の操作を行う必要がある．

- アクセス修飾子が private または protected のフィールドのうち，名前の接頭辞（または接尾辞）にアンダースコアがついていないものをすべて抽出し，それらに対して**フィールド名の変更**を実行する．
- 特定のクラスの中からメソッド名と引数の数が等しいメソッドをすべて取得し，それらに対して**メソッド名の変更**を実行する．

リスト 2 private なフィールドの名前に接頭辞を付ける例（上：リファクタリング前，下：リファクタリング後）

```
class Book{
  public static final PREFIX = "#";
  private String name;
  private String text;
  private int id;
  protected String category;
}
```

```
class Book{
  public static final PREFIX = "#";
  private String _name;
  private String _text;
  private int _id;
  protected String category;
}
```

リスト 3 パラメータ数の同じ同名のメソッドを検索して，片方の名前を変更する例（上：リファクタリング前，下：リファクタリング後）

```
class Manager{
  void register(String name, Country c){}
  void register(String name, int code){}
  void register(String name, int code,
                String address){}
}
```

```
class Manager{
  void register(String name, Country c){}
  void registerWithCode(String name, int code){}
  void register(String name, int code,
                String address){}
}
```

コーディング規約はプロジェクト横断的に利用できる．しかし，既にあるソースコードに対してコーディング規約を新たに適用もしくは変更する場合，対象となる箇所を全て選択し，リファクタリング処理を施す必要があり，実行コストが非常に高い．適用箇所が多くなるほど，手動による適用ではミスが生じやすくなる．

2.3 デザインパターンの導入

?では，図??で表されるような Visitor パターン導入の例が挙げられている．

この変形には 20 回以上のリファクタリングが組み合わ

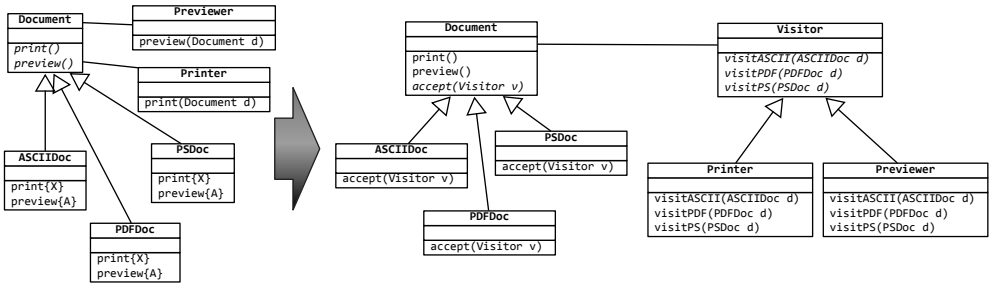

図 1 Visitor パターンの導入（左：導入前，右：導入後）

されており，例えば以下のリファクタリングが含まれる．

メソッドの移動 各 *print* メソッドを，*Printer* クラスに移動する．

メソッド名の変更 名前の衝突を避けるため，移動したメソッドの名の先頭に *visit* を追加する．

上の 2 つだけとってみても，「*Document* クラスのサブクラスから指定されたシグネチャの（*print* と名のつく）メソッドを探し出し，*Printer* クラスに移動」「移動させたメソッドを，移動元のクラス名（*ASCIIDoc*, *PDFDoc*, *PSDoc*）に基いた新しい名前（*visitASCII*, *visitPDF*, *visitPS*）に変更する」という操作が必要である．これらの操作は適用の回数が多く，手動で行うのは明らかにコストがかかる．しかし，形式的に書き下すことができる．

3. RefactoringScript 言語および処理系

本節では，要素技術である Eclipse について，そのアーキテクチャやリファクタリング機能の実装について紹介し，我々の提案する RefactoringScript 言語およびその処理系の設計等を述べる．

3.1 Eclipse プラグイン

3.1.1 プラグインアーキテクチャ

Eclipse の機能はすべて Java で実装されたプラグインで構成されており，Eclipse は基底フレームワークの上にこれらのプラグインを組み合わせる形で実装されている．

プラグインを作成することで，Eclipse ユーザは自由に Eclipse の機能を拡張することができ，逆に特定のプラグインの Java クラスを参照することで，Eclipse の機能を利用できる．

3.1.2 リファクタリングプラグイン

リファクタリング機能もこの最たる例で，プラグインとして提供されている．リファクタリングを行うライフサイクルは次のとおりである?．1) リファクタリングオブジェクトを作成し，2) 妥当性を検査する．必要であれば，3) その他のオプションを設定し，4)Change オブジェクトを作成する．最後に Change オブジェクトを 5) 実行する．

例えば**フィールドのカプセル化**リファクタリングを行うためのクラスは， org.eclipse.jdt.internal.corext.refactoring.sef.SelfEncapsulateFieldRefactoring に定義されている．これを利用して，実際に**フィールドのカプセル化**リファクタリングを行うコード例をリスト??に示す．

リスト 4 フィールドのカプセル化リファクタリングを行うコード例

```
SelfEncapsulateFieldRefactoring ref =
  new SelfEncapsulateFieldRefactoring(f); // 1
ref.checkInitialConditions(); // 2
ref.setGetterName("getX"); // 3
Change change = ref.createChange(); // 4
Change undo = change.perform(); // 5
```

3.1.3 JDT

JDT?は, Eclipse 上に構成されたプラグインであり，IDE のバックエンドでコア機能を提供するプラグインと，IDE 上でユーザインターフェースを提供するプラグインから成る?．JDT のうち，パッケージやクラスと言った Java 特有の要素と一対一に対応しているのが Java エレメント（Java モデル）で，これを可視化することにより，パッケージエクスプローラ（図??）が実現されている?．Java エレメントは `org.eclipse.jdt.core` 以下に定義されている．

また，JDT はリファクタリング機能そのものも提供する．RefactoringScript 処理系のリファクタリング機能はこれを用いて実装しているため，リファクタリング操作を一から定義する必要がなく，低コストで実装することが出来た．

3.2 RefactoringScript 言語および処理系の要件

我々が提案する RefactoringScript 言語とその処理系に求められる要件は以下のとおりである．なお以降では，RefactoringScript 言語および処理系を合わせて単に RefactoringScript, RefactoringScript 言語により記述されたスクリプトを，RefactoringScript スクリプトもしくは単にスクリプトと表現する．

R1: 解析 API とリファクタリング機能 リファクタリングの適用箇所を検索し，リファクタリング操作を適用できる．

R2: 簡潔なスクリプト表現 スクリプトにはリファクタリングの適用箇所と適用内容以外の記述を極力含ま

ない．

R3: 即時実行 コンパイル作業などを必要とせず，その場でスクリプトを実行できる．

R4: 広く利用可能 導入コストが小さく，容易に利用することができる．

なお，Java エレメントの検索と，それを対象とするリファクタリング処理は，今回のプロトタイプ開発の段階ではステートメントレベルに踏み込まない範囲に制限した．表??に各 Java エレメントと対応するクラスの名前，取得可能な属性を示す．なお，表中の○は属性を取得できることを，×は取得できないことを表す．

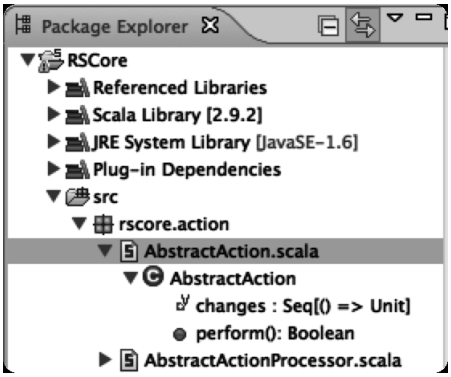

図 2 Eclipse のパッケージエクスプローラ

表 2 Java エレメントと取得可能な属性

Java エレメント	プラグイン中のクラス名	名前	修飾子	型
プロジェクト	IJavaProject	○	×	×
ソースフォルダ	IPackageFragmentRoot	○	×	×
パッケージ	IPackageFragment	○	×	×
ソースファイル	ICompilationUnit	○	×	×
クラス	IType	○	○	×
フィールド	IField	○	○	○
メソッド	IMethod	○	○	○
メソッド引数	ILocalVariable	○	×	○

3.3 RefactoringScript 言語および処理系の全体像

本節では，RefactoringScript 言語，処理系，ユーザ間のインタラクションについて述べる．

本ツールは，以下の 2 つのコンポーネントから構成される．

RSCore [*1] RefactoringScript 言語の要素やその処理系，インタープリタを含んだ，ツールの根幹部分

RSUI RSCore のインタープリタに入力する RefactoringScript スクリプト専用のエディタ[*2]や，作成したスクリプトを実行するためのメニュー[*3]など，ユーザが操作するインターフェース部分

ユーザが記述したスクリプトを作業ワークスペース（ユーザの作業領域．ソースコードやその他ファイル，ディレクトリが含まれる場所）に対して適用させる際の手順，ユーザと RefactoringScript 処理系とのインタラクションは図??のとおりである．

(1) ユーザはエディタでスクリプトを作成・編集する．
(2) ユーザはスクリプトファイルを指定してコアコンポーネントを起動する．
(3) インタープリタにスクリプトを入力する．
(4) インタープリタはユーザのワークスペースにスクリプトを実行，適用する．
(5) ユーザはスクリプト実行の成否を通知される．

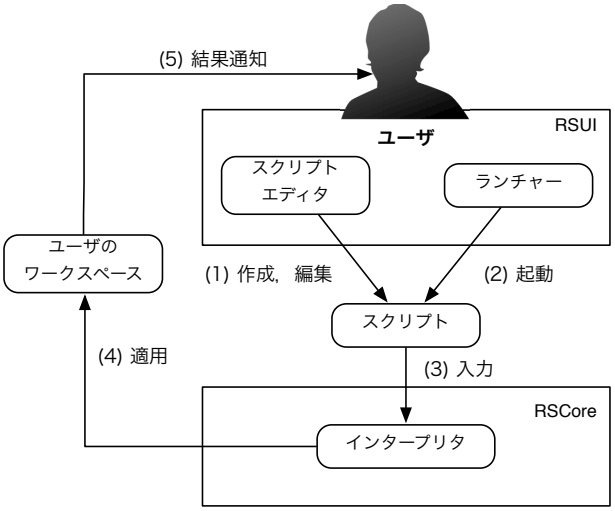

図 3 RefactoringScript 処理系，ユーザ間のインタラクション

また，スクリプトの解釈から，ユーザのワークスペースへの適用までは，図??に示すとおりである．

(1) コードエンティティを利用するスクリプトを解釈してリファクタリングの適用箇所を検索する．
(2) 適用箇所と適用内容を合わせてアクションを生成する．
(3) アクションを実行し，Java エレメントに対して変更を適用する．

Eclipse 上でスクリプトを作成し，実行する画面のスクリーンキャプチャを図??に示す．

3.4 RefactoringScript 言語

本節では，RefactoringScript 言語の要素について詳細に述べる．

3.4.1 コードエンティティとコードエンティティコレクション

JDT の Java エレメントは，ワークスペースから特定の要素を検索するのに適した API を提供する．例えば，リスト??は特定のクラス以下のすべてのメソッドやフィールドに対応する Java エレメントを取得する．

[*1] https://github.com/t3kot3ko/RSCore
[*2] https://github.com/t3kot3ko/RSEditor
[*3] https://github.com/t3kot3ko/RSLauncher

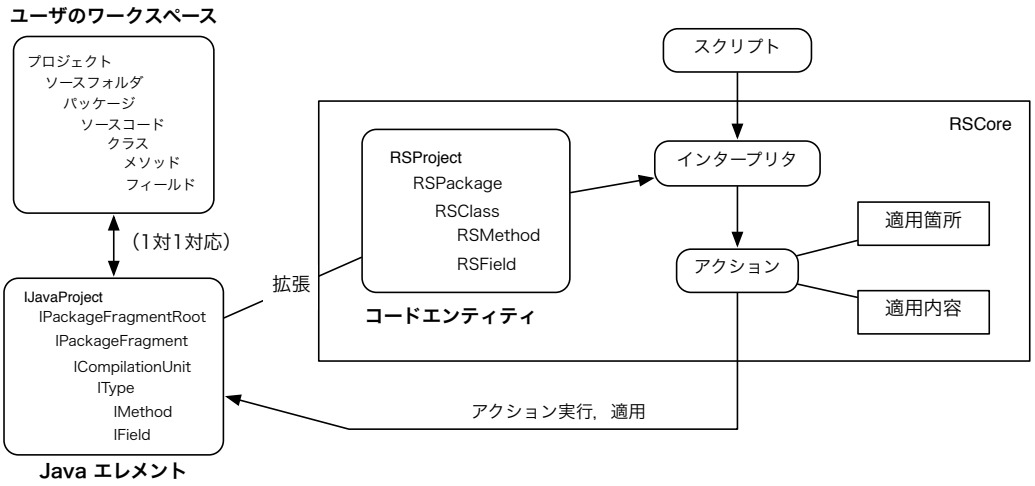

図 4 RefactoringScript 処理系の詳細

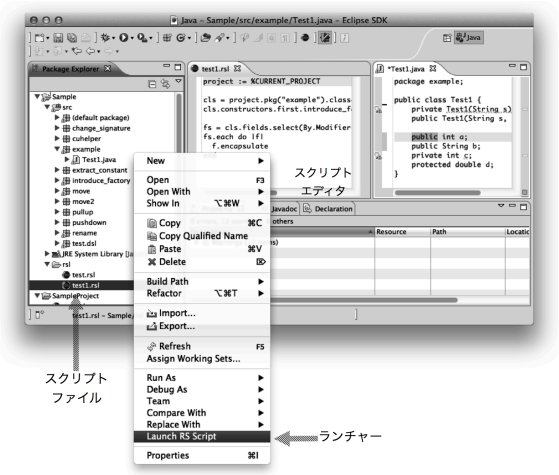

図 5 実行画面のスクリーンキャプチャ

リスト 5 クラス以下のメソッド，フィールドを取得する例

```
IType t = ...
IMethod[] methods = t.getMethods();
IField[] fields = t.getFields();
```

しかし Java エレメントは，取得する要素の条件を詳細に指定できる API を持たない．例えば，アクセス修飾子情報に直感的にアクセスできないので，「public かつ static」なメソッドだけを抽出するのが難しい．

コードエンティティ (Code Entity; CE) は，主に以下の 2 つの API を Java エレメントに追加したクラスである．

- 検索と解析に必要となる API．例えば，指定したアクセス修飾子をすべて持っているかどうかを判定できる API やスーパークラスを取得できる API．
- コードの木構造を簡潔な自然言語により近い記述でトレースできる API．例えばメソッドをすべて取得するには，`t.getMethods()` ではなく，`t.methods` を利用できる．

各コードエンティティと Java エレメントの対応関係を表??に示す．なお，表中のインデントはパッケージの包含関係，クラスの継承関係を表す．

表 3 Eclipse 上の要素と CE の対応関係

Eclipse	RSEntity
`org.eclipse.jdt.core`	
`IMember`	`RSMember`
`IType`	`RSClass`
`IField`	`RSField`
`IMethod`	`RSMethod`
`IPackageFragment`	`RSPackage`
`ILocalVariable`	`RSParameter`
`IJavaProject`	`RSProject`
`org.eclipse.core.resources`	
`ResourcesPlugin`	`RSWorkspace*`

なお，`RSWorkspace` は他の CE とはやや異なり，対象となるワークスペースへの参照を表し，他の CE を検索するための起点となる．

コードエンティティコレクション (Code Entity Collection; CEC) は，CE の集合を表し，集合に含まれた CE を検索できる API を提供する．検索を行うためのスクリプト例は次節で述べる．

3.4.2 クエリ選択子と限定子

CEC から CE を検索するには `select` メソッドを利用してクエリ選択子 *QuerySelector*，限定子 *Qualifier* および検索パラメータ *SearchParams* を組み合わせて以下の書式でスクリプトを記述する．

CEC.`select`(*QuerySelector*(*Qualifier*(*SearchParams*)))

QuerySelector ::= "`By.name`"|"`By.namereg`"
|"`By.modifier`"|"`By.typename`"

Qualifier ::= ""|"`With.or`"|"`With.and`"|"`With.out`"

クエリ選択子は検索キーを指定するキーワードである．検索キーは，CE の名前と名前の正規表現，アクセス修飾子，型名の 4 つを指す．この 4 つにより，表??に示した要素がもつ属性をキーに CE を検索することができる．各 CE と，対応するクエリ選択子，検索キーの組み合わせを表??に示す．なお，表中の○は，CE が検索キーを用いて検索できることを表す． 例えば，RSProject の集合は名前をキーに要素を検索できるが，アクセス修飾子名をキーに要素を検索できない．

限定子は与えられた検索パラメータ群を OR, AND, NOT のいずれで解釈するかを指定するキーワードである．ただし，限定子を必要としない場合（検索パラメータが 1 つしかない場合）は省略可能である．リスト??, ??, ??に，CEC から CE を検索するスクリプトの例を示す．

リスト 6 メソッド群 ms のうち，アクセス修飾子が private であるメソッドを検索するスクリプト例

```
# 限定子を省略できる
ms.select(By.modifier("private"))
```

リスト 7 メソッド群 methods のうち，アクセス修飾子が private または protected であるメソッドを検索するスクリプト例

```
# パラメータを OR で解釈する
fs.select(
   By.modifier(With.or("private", "protected")))
```

リスト 8 メソッド群 ms のうち，アクセス修飾子が public で，かつ返却値型が int または String であるメソッドを検索するスクリプト例

```
# select メソッドはチェインできる
ms.select(By.modifier("public"))
  .select(By.typename(With.out("int", "String")))
```

3.4.3 特別変数

スクリプト中で利用できる特別な変数として，以下のような変数を宣言，利用することができる．なお，変数への代入には演算子:=を用いる．

- $：現在のワークスペースへの参照を表す．RSWorkspace と等価である．
- %CURRENT_PROJECT：スクリプトが属すプロジェクトへの参照を表す．ただし，必ずしもスクリプトがプロジェクトに属す必要はなく，$.project("project_name") とすることで任意のプロジェクトを参照できる．

3.4.4 アクション

CE/CEC に対するリファクタリング操作をアクション (*Action*) と呼び，アクションパラメータ (*params*) を伴って以下の書式で表現する．

$$CE/CEC.Action(params)$$

アクションパラメータはリファクタリングを行う際に必要最低限を指定すればよい．表??で，現時点でサポートしているアクションの種類と指定できるアクションパラメータをまとめる．

3.5 RefactoringScript 処理系

3.5.1 インタープリタ

本ツールで用いるスクリプトのインタープリタに求められる要件は以下の 3 点である．

RI1: 動的解釈 ユーザが作成したスクリプトを実行時に解釈，評価できる．

RI2: Java 資産の利用 Java で作成された Eclipse プラグインの機能を利用できる．

RI3: 簡潔な記述 ユーザは CE の検索とそれに対する処理の記述のみに集中できる．

本ツールでは，以下のように RI1, RI2, RI3 を満足する JRuby? を採用した．本ツールで用いるスクリプトは，JRuby の内部 DSL とみなすことができる．また，JRuby の採用により，インタープリタを一から実装する必要がなく，低コストで処理系に組み込むことが出来た．

- ScriptingContainer[*4]により，プログラム実行時に Ruby プログラムを解釈できる (RI1).
- JRuby は Java による Ruby 実装であるため，Java 資産をシームレスに利用できる (RI2).
- スクリプト中に Ruby 表現が利用できるため，型宣言なしに変数が利用できる．関数呼び出しのカッコを省略できる ．また，Ruby の組み込み関数を用いることができるため．CEC を走査する場合 Array#each メソッドを利用して，外部イテレータを使うことなく簡潔な記述が可能になる (RI3).

3.5.2 スクリプト例

RefactoringScript 言語によるスクリプト例を示す．ここでは，??節の動機付け例の解決を考える．リスト??に「スクリプトが属しているプロジェクト内の example パッケージ内のすべてのクラスについて，private かつ static でないフィールドの名前の先頭にアンダースコアを付けるように**名前の変更**を行う．ただし，すでについている場合は何も行わない」という操作のスクリプト例を示す．

4. 評価

4.1 評価内容と結果

RefactoringScript スクリプト利用による記述可能性，正確性と実行コスト，再利用性を評価するため，以下の 4 つの複合リファクタリングを行う場面を想定して，被験者実験およびケーススタディを行った．

なお，4 つの場面は?に挙げられた複合リファクタリング

[*4] org.jruby.embed.ScriptingContainer

表 4 クエリ選択子

検索キー クエリ選択子	名前 By.name	名前の正規表現 By.namereg	アクセス修飾子 By.modifier	型名 By.typename
RSField	○	○	○	×
RSMethod	○	○	○	○
RSClass	○	○	○	×
RSParameter	○	○	×	○
RSProject	○	○	×	×
RSWorkspace	×	×	×	×

表 5 サポートされているアクションと対応するアクションパラメータ

アクション	レシーバ	対応するリファクタリング名	必須アクションパラメータ
rename	RSField	**フィールド名の変更**	変更後の名前
rename	RSMethod	**メソッド名の変更**	変更後の名前
encapsulate	RSField	**フィールドのカプセル化**	なし
introduce_factory	RSClass	**ファクトリメソッドの導入**	ファクトリメソッドの導入先クラス
introduce_factory	RSMethod	**ファクトリメソッドの導入**	ファクトリメソッドの導入先クラス
introduce_parameter_object	RSMethod	**パラメータオブジェクトの導入**	パラメータオブジェクトを定義するクラス名
pull_up	RSMethod	**メンバの引き上げ**	引き上げ先のクラス
push_down	RSMethod	**メンバの引き下げ**	なし
change_return_type	RSMethod	**返却値型の変更**	変更後の返却値型名
delete*	RSEntity	（削除）	なし
move*	RSEntity	（移動）	移動先のクラス

* delete と move は，単純変形処理（単にコード片を削除／移動する）であり，厳密にはリファクタリング操作ではない

リスト 9 private フィールドの名前を変更するスクリプト記述例

```
cp := %CURRENT_PROJECT
cp.pkg("example").classes.toRuby.each do |c|
  # private かつ static でないフィールドを取得
  fs = fp.fields.select(By.modifier("private"))
       .select(By.modifier(With.out("static")))

  # フィールドの先頭がアンダースコアでなければ変更
  fs.toRuby.each do |f|
    unless((f.name)[0] == "_")
      f.rename("_" + f.name)
    end
  end
end
```

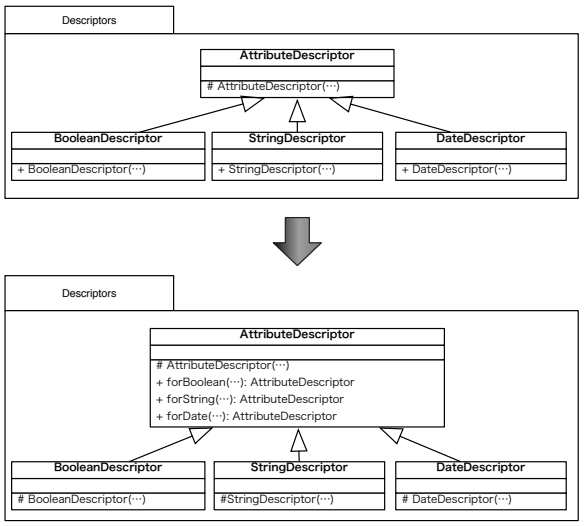

図 6 Factory によるクラス群の隠蔽?

の傾向やコーディング規約?などを考慮し選択した．

EX1 指定したパッケージ内のすべてのクラスについて，private フィールド名に接頭辞を付ける．

EX2 指定したパッケージ内のすべてのクラスについて，public フィールドをカプセル化する．

EX3 図??のように，Factory によるクラス群の隠蔽?を行う．

EX4 パッケージ内の特定のフィールドの名前を変更し，さらに対応するアクセサの名前を変更する．

4.1.1 記述可能性

EX1, 2, 3, 4 について，Java コードでリファクタリング処理を記述した場合と，RefactoringScript 言語により記述した場合のコード行数を計測した．結果を表??に示す．なお，実験対象の Java プロジェクトは RSCore のテストに利用したテストデータである．

4.1.2 正確性と実行コスト

複合リファクタリングを RefactoringScript 言語および処理系を利用した場合と手動で行った場合の正確さと実行コスト比較を比較するため，以下の被験者実験を行った．

表 6 Java による記述と RefactoringScript による記述のスクリプト行数の比較（単位：行）

	Java	RefactoringScript
EX1	42	10
EX2	33	5
EX3	107	9
EX4	48	12

実験対象は実験用に用意したサンプルプロジェクト[*5][*6]である．なお，操作難易度の類似や，予備知識の影響などを考慮して，EX1, 4 のみを実験対象とした．また簡単のため，EX4 における対象フィールドは int 型のフィールドとして，型によるフィールドの抽出をスクリプトで行った．

被験者 情報系学部生，院生計 5 名（P1〜P5）

手法 EX1, 4 を手動 → スクリプト利用の順で行うグループ，スクリプト利用 → 手動で行うグループに分け，それぞれ目的のリファクタリングが完了するまでの時間と，正確に適用された箇所を計測する．

この被験者実験結果を，表??, ??にまとめる．なお，表??は，被験者 P1 が，EX1 を手動で，EX4 をスクリプトにより適用し，それぞれ 7 分，22 分かかったことを表す．また，表??は，被験者 P1 が，EX1 を手動で，EX4 をスクリプトにより適用し，それぞれ 27 箇所，96 箇所を正しく適用できたことを表す．ただし，EX1, EX4 の適用するべき箇所の数はそれぞれ，30, 96 である．

表 7 手動とスクリプトによる実行との実行時間の比較（単位：分）

実験	P1	P2	P3	P4	P5	平均
EX1（手動）	7	-	5	-	-	6.0
EX1（スクリプト）	-	10	-	5	14	9.7
EX4（手動）	-	17	-	9	13	13.0
EX4（スクリプト）	22	-	10	-	-	16.0

表 8 手動とスクリプトによる実行との正確さの比較（単位：箇所）

実験	P1	P2	P3	P4	P5	平均
EX1（手動）	27	-	30	-	-	28.5
EX1（スクリプト）	-	30	-	30	30	30
EX4（手動）	-	95	-	96	93	94.7
EX4（スクリプト）	96	-	96	-	-	96

4.1.3 再利用性（ケーススタディ）

EX1, 3, 4 を，オープンソースプロジェクト P1[*7], P2[*8]に適用し，手動でリファクタリングを適用した場合のソースコードとの差分を機械的に取得し，その結果を目指で確認することで，目的の処理が行えることを確認した．なお，リファクタリング操作のみに焦点を当てるため，規模が中程度のプロジェクトを実験題材として選択した．表??に，プロジェクトと実験の種類，リファクタリングにより影響を受けたファイル数，行数および適用箇所を示す．

表 9 オープンソースプロジェクトに対する適用

プロジェクト	実験	ファイル数	行数	適用箇所
P1	EX1	3	68	16 フィールド
P1	EX4	2	18	6 フィールド 12 メソッド
P2	EX3	6	20	6 クラス 6 メソッド

4.2 考察

4.2.1 記述可能性

RQ1 リファクタリング操作（適用箇所と適用内容）を簡潔かつ正確に記述でき，またそれを適用できるか？

4 つのケースすべてについて，RefactoringScript で記述したスクリプトは Java で記述したスクリプトの 1/4〜1/10 程度の行数になっている．これは主に，以下の理由によると考えられる．

- CE 柔軟に検索する API により，条件文のネストになりにくい
- ワークスペースの取得などリファクタリングの実行に直接関係しない処理を記述しなくてよい

また，RefactoringScript 言語によるスクリプトは，Ex3 程度の規模であっても 10 行ほどの少ない行数で記述することができる．

RefactoringScript 言語は，リファクタリングの適用箇所の検索と適用内容を記述することに特化しているため，簡潔なスクリプト記述を実現している．

4.2.2 正確さと実行コスト

RQ2 ツールを利用しない場合と比べて，複合リファクタリングを正確に実行できるか？

RQ3 ツールを利用しない場合と比べて，複合リファクタリングを実行するコストを軽減できるか？

実行コストに関しては，いずれの実験についても手動の方がやや所要時間が短い結果となった．これは，RefactoringScript が一定の学習コストを要することが原因であると考えられる．実際，スクリプトを記述した被験者からのフィードバックには，以下の意見があった．

- Ruby イディオムの利用に戸惑った
- スクリプトの書き方を深く学習した後であれば時間を短縮できると思う．（なお本実験では，例題とその解答，および実験を行うにあたって必要なスクリプト片を資料として配布するにとどめた）

しかし逆に，手動で実験を行った被験者からのフィードバックには，

- より複雑な題材の時（例えば，適用箇所が莫大なとき）手作業では行いたくない

[*5] https://github.com/t3kot3ko/Ex1
[*6] https://github.com/t3kot3ko/Ex4
[*7] https://github.com/shigenobu/acbook-wa710
[*8] http://code.google.com/p/jslideshare/

- そもそも機械的な単純作業を手動で行いたくない

といった意見もあった．したがって，スクリプト記述に十分慣れた後であれば，開発者の負担を軽減できると考えられる．

正確さについては，スクリプトを用いた場合すべての被験者が正しく動作するスクリプトを記述することができた．一方，手動で行った場合，以下のミスを含んだ解答があった．

- 指定されていないフィールドまでリネームしている (EX1).
- フィールドは正しくリネームされているが，アクセサの名前が間違っている (EX4).

以上より，スクリプトによる統一的な適用箇所の指定は，正確に複合リファクタリングを行えることに寄与していると考えられる．

4.2.3 再利用性

RQ4 プロジェクト横断的にリファクタリング操作を再利用できるか？

??, **??**の実験に利用したスクリプトをほぼ改変することなく，そのまま各プロジェクトに適用することができた．プロジェクトごとに変更する必要があったのは，主に以下の点である：

- 適用対象のパッケージ名
- アクションパラメータ（例えば，P1 に対する EX4 では created, updated, executed というフィールド名をそれぞれ createdAt, updatedAt, executedAt に変更し，対応するアクセサの名前も変更した）

しかしこれらはあくまでプロジェクト固有な要素であり，ユーザがプロジェクトごとに指定しなければいけない最低限のパラメータである．Ruby 表現により，関数を利用できることを考慮すれば，このようなプロジェクト依存なパラメータの指定箇所を容易に局所化できると考えられる．

4.3 制限

4.3.1 遅延評価

本ツールでは，リファクタリングによる変形の影響を CE に反映させることができない．例えば，リスト**??**のように**フィールド名の変更**を行う際，ワークスペースは影響を受けても CE の状態は更新されない．

リスト 10 制限：フィールド名の変更

```
f = $.methods.pkg("example").classes
            .first.first_field
f.name  #=> a
f.rename("newname")
f.name  #=> a
$.methods.pkg("example").classes
            .first.first_field.name  #=> newname
```

したがって，同じ CE に対して複数回アクションを施す際は，その都度 CE を検索し，特定する必要があるが，遅延評価の仕組み（CE 検索用のクエリを実行の直前まで蓄えておく仕組み）を導入することにより，解決できる見込みである．なお，特定の CE に対し複数の原始リファクタリングを施すケースでは，ほとんど問題にならないと考えられる．

4.3.2 エラーハンドリング

本ツールではスクリプト実行時のエラーハンドリングを行わない．スクリプトの実行が成功したことはダイアログの表示で理解できるが，失敗した時その原因をユーザが理解できない．また，リファクタリングの前提条件が満たされない場合，内部的にはエラーを検知しリファクタリング処理を行わないが，リファクタリングが失敗していることをユーザは理解できない．これらは，ユーザへの通知機能を強化することで解決できる見込みである．

さらに，複合リファクタリングの一部が失敗した場合，リファクタリングシーケンス全体をロールバックする機能が存在しない．これは，リファクタリング操作を適用する際，逆変換用のアクションを保存し，リファクタリング失敗時にそれらを適用することで解決できる見込みである．

4.3.3 操作可能範囲

??節で述べたとおり，今回のプロトタイプ開発の段階では RefactoringScript 言語で提供する検索用の API や，リファクタリング操作は，ステートメントレベルに踏み込んだ解析に対応しない．したがって，例えば「ヌルオブジェクトの導入」**?**は，適用箇所を検索することも，それに対する処理を記述することもできない．これは，ステートメントレベルのコード要素に対応した CE および，検索 API，リファクタリング機能を実装することにより解決できる見込みである．

また，Eclipse が提供しているステートメントレベルに踏み込まないリファクタリングのうち，Convert Local Variable to Field, Infer Generic Type Arguments などには対応していない．これらは，プロトタイプ開発の段階で，動機付けの例や被験者実験に関係無かったため省略したが，他の実装済みのリファクタリング機能と同じように実装を行えば，比較的容易に対応できる見込みである．

4.3.4 妥当性の脅威

評価における 4 つのリファクタリング操作は，RefactoringScript 言語および処理系で比較的容易に実現できるものを選択し，実現不可能なものは題材に選ばなかった．したがって，例えば他の研究における評価実験題材を RefactoringScript では実現できない可能性もあり，妥当性の脅威であると言える．

また，被験者実験では，作業時間のうちスクリプトそのものの学習コストが占める割合が大きくなった．これは，今後事前にスクリプトの記法を十分に習得した上で実験に

臨むなどして，学習コストを正確に見積もった上で，それを差し引きした純粋な作業時間の計測を目指したい．

5. 関連研究

丸山ら?は， 抽象構文木を XML で表現し，その上に構築した制御フローグラフとプログラム依存グラフを用いて，リファクタリングの前提条件の検証と変形処理の生成を行うツールを提案した．彼らは抽象構文木に対する API を提供する代わりに汎用的な表現である XML を利用することで，DOM や SAX など既存技術の適用を実現した．Cardy?は，入力テキストを彼が提案する文法に基づいてソースコード解析し，取得したツリー構造に対して変形ルールを適用させることで，ソースコード変換を行う TXL というフレームワークを提案している．彼は抽象構文木の変換処理系である TXL を利用することでリファクタリングを含めた様々なソースコードの変形処理を実現した．Verbaere ら?は，ソースコード情報を独自のスキーマの抽象構文木にマッピングし，ML ベースの独自言語での変換記述を実現する JunGL というシステムを提案した．Hills ら?は，JDT による解析とメタプログラミング用言語 Rascal による変換ルールの記述と処理系を提案しており，Visitor パターンの実装を Interpreter パターンに変換するリファクタリングエンジンを作成した．

彼らの手法はいずれも，コード変形操作を定義するための手法であり，彼らの処理系が持つリファクタリング機能を呼び出す際に，我々が開発した RefactoringScipt を利用できる可能性がある．また，RefactoringScipt がサポートするリファクタリングの種類を増やす際に彼らの処理系を利用できる可能性がある．

しかし，彼らの手法は，リファクタリング操作の定義に特化しており，すでに存在するリファクタリングの組み合わせや複数箇所への適用を簡潔に記述できる RefactoringScript とは根本的にコンセプトが異なる．ただし，?と?はコードを変形するルールの定義に既存のプログラミング言語を用いており，JRuby の内部 DSL としてスクリプト記述を実現した RefactoringScript と類似している部分もある．

このように，RefactoringScript は，一般の開発者がコーディングの一環としてスクリプトを書くことで，複合リファクタリングをオンデマンドに実行することの支援を目的としているのに対して，既存研究による手法は新たなリファクタリング機能の開発者に対して，その支援を行うことを目的としている点で相違点がある．したがって，リファクタリング機能の呼び出し，特に，複合リファクタリングの呼び出しをスクリプトで表現する際は，RefactoringScript を用いることで既存研究の処理系を利用するよりも記述量を抑えられる可能性が高い．

上記の相違点を踏まえて，RefactoringScript と既存研究を比較した結果を表??で示す．なお，表中の Y は該当する性質を持つと判断されること，N は持たないと判断されることを示す．以上から，一般の開発者がリファクタリング機能の適用をプログラマブルに記述できるという点で，本手法には新規性および貢献があると結論付けられる．

表 10 既存研究と RefactoringScript の比較

	Refactoring Script	?	?	?	?
リファクタリングの新規追加が行える	N	Y	Y	Y	Y
リファクタリングの呼び出しが容易である	Y	N	N	N	N
コード変形に用いる言語	既存 (J)Ruby	既存 Rascal	独自	既存 Java	独自
リファクタリング処理系	既存	既存 独自	独自	独自	独自

Li ら?, ?は，Erlang 言語向けに Wranglar なるリファクタリング用スクリプトとその処理系を提案している．

RefactoringScript と同様に，リファクタリング箇所とリファクタリング内容を記述したスクリプトを用いて複合リファクタリングの自動実行支援を目的としている．スクリプトに記述されたリファクタリング操作そのものと前提条件のチェックを逐次的に実行している点も類似している．

しかしこれらはテンプレートベースなパターンマッチングに基づく適用箇所の記述を行なっている．そのため，Java のようなプログラミング言語を利用する開発者には馴染みがなく混乱をきたす可能性がある．

一般的に，Java 言語を始めとする純粋オブジェクト指向プログラミング言語は，パッケージやクラス，フィールド，メソッド等の包含関係を直感的に木構造で表現できる．例えば，Eclipse のパッケージエクスプローラ（図??）では，これを視覚的に表示して，ユーザのソースコードの構造把握を支援する．

RefactoringScript 言語では，JQuery?の DOM セレクタや XPath? のような，木構造をトップダウンで探索できるインターフェースによる簡潔な記述の実現を目指した．

6. おわりに

本論文では，リァクタリングの適用箇所と適用内容を記述できる RefactoringScript 言語とその処理系を提案した．CE 検索 API と，JRuby の利用による Ruby 文法の採用により，ユーザフレンドリなスクリプトを実現した．本手法の利用により，コード規約の適用など，多くの箇所にリファクタリングを適用する場合や，プロジェクトをまたいでリファクタリングを繰り返し適用する場合などに，そのコストを大きく削減すると期待できる．

現在は，サポートできるリファクタリングの種類が Eclipse で提供されているリファクタリング機能に限定さ

れているが，これを拡充することでより柔軟なコード変形が実現できる予定である．

Ruby では演算子の再定義が可能なので，例えば|| 演算子を With.or の代わりに用いるなどして，より簡潔なスクリプト記述が実現できるよう改善する予定である．なお，現状では Ruby 言語を用いることで，Java や Ruby などのオブジェクト指向言語を利用する開発者にとって馴染みのあるスクリプト記述の実現を目指しているが，?, ?, ? などで提案されている他の言語の構文との比較検討を通して，より良い言語を目指して行きたい．

また，スクリプトを蓄積し共有するすることで，リファクタリングを組み合わせて利用するべき場面と，その具体的な対処法をまとめることができ，よりリファクタリング機能を頻繁に利用されるようになると期待できる．

さらに，コード検索に RefactoringScript 言語を利用し，任意のソースコード文字列を指定した位置に差し込むエンジンを用意すると，アスペクト指向プログラミング処理系として利用できたり，リファクタリング操作そのものを記述して新しいリファクタリングを定義できる可能性がある．

今後，RefactoringScript の有用性を示すため，学習コスト，規模に対するコスト変化の測定，既存研究とのスクリプト記述量等の比較，事例研究を行い，複雑なリファクタリングに対する記述量削減効果の確認も行っていきたい．

参考文献

[1] Eclipse. http://www.eclipse.org/.

[2] Eclipse Java development tools (JDT). http://www.eclipse.org/jdt/.

[3] IBM, Explore refactoring functions in Eclipse JDT. http://www.ibm.com/developerworks/opensource/library/os-eclipse-refactoring/.

[4] IBM JDT プログラマーズガイド. http://publib.boulder.ibm.com/infocenter/iadthelp/v6r0/index.jsp.

[5] JetBrains ReSharper. http://www.jetbrains.com/resharper/.

[6] JQuery. http://jquery.com/.

[7] JRuby. http://jruby.org/.

[8] Microsoft Visual Studio. http://www.microsoft.com/ja-jp/dev/.

[9] Oracle, Code Conventions for the Java Programming Language. http://www.oracle.com/technetwork/java/codeconvtoc-136057.html.

[10] XML Path Language(XPath). http://www.w3.org/TR/xpath/.

[11] オブジェクト倶楽部 Java コーディング標準. http://www.objectclub.jp/community/codingstandard/CodingStd.pdf.

[12] James R. Cordy. Source transformation, analysis and generation in TXL. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '06, pp. 1–11, New York, NY, USA, 2006. ACM.

[13] Johannes Henkel and Amer Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pp. 274–283, New York, NY, USA, 2005. ACM.

[14] Mark Hills, Paul Klint, and Jurgen J. Vinju. Scripting a refactoring with Rascal and Eclipse. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pp. 40–49, New York, NY, USA, 2012. ACM.

[15] Huiqing Li and Simon Thompson. A domain-specific language for scripting refactorings in Erlang. In *Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering*, FASE'12, pp. 501–515, Berlin, Heidelberg, 2012. Springer-Verlag.

[16] Huiqing Li and Simon Thompson. Let's make refactoring tools user-extensible! In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pp. 32–39, New York, NY, USA, 2012. ACM.

[17] K. Maruyama and S. Yamamoto. Design and implementation of an extensible and modifiable refactoring tool. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pp. 195–204, 2005.

[18] T. Mens and T. Tourwe. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, Vol. 30, No. 2, pp. 126 – 139, feb 2004.

[19] Mel Ó Cinnéide and Paddy Nixon. Composite refactorings for Java programs. Technical report, Department of Computer Science, University College Dublin, 2000.

[20] Mohsen Vakilian, Nicholas Chen, Roshanak Zilouchian Moghaddam, Stas Negara, and Ralph E. Johnson. A compositional paradigm of automating refactorings. Technical report, University of Illinois, 2012.

[21] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pp. 233–243, Piscataway, NJ, USA, 2012. IEEE Press.

[22] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. Jungl: a scripting language for refactoring. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pp. 172–181, New York, NY, USA, 2006. ACM.

[23] ジョシュア・ケリーエブスキー. パターン指向リファクタリング入門 ソフトウエア設計を改善する 27 の作法. 日経 BP 社.

[24] マーチン・ファウラー. リファクタリング: プログラミングの体質改善テクニック. ピアソン・エデュケーション.

[25] 竹添直樹, 志田隆弘, 奥畑裕樹, 里見知宏. Eclipse 3.4 プラグイン開発 徹底攻略. 毎日コミュニケーションズ.