社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

自動プログラム修正手法を用いた自動リファクタリングツールの試作

谷門　照斗†　肥後　芳樹†　楠本　真二†

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1-5
E-mail: †{a-tanikd,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　リファクタリングとはソフトウェアの外部的振る舞いを保ったまま，内部構造を改善する作業であり，ソフトウェアの保守性を向上させるために重要な作業である．しかし，リファクタリングは複数の手順で行われる複雑な作業であるため，手動によるリファクタリングは誤りが発生しやすく，適切に行うには高い技術と多大な時間を必要とする．そこで本研究では，リファクタリング手順を自動化する手法を提案する．具体的には，ソフトウェアメトリクスを用いてリファクタリングすべき箇所を自動的に特定し，自動プログラム修正手法を用いてメトリクスの値が改善するようにソースコードの編集を自動的に行う．本研究では，自動リファクタリングツールを試作し，そのツールで自動的にリファクタリングが行えることを確認した．

キーワード　自動リファクタリング，自動プログラム修正，ソフトウェアメトリクス

1. ま え が き

近年，ソフトウェアは社会のさまざまな分野で重要な役割を果たしており，より利便性や信頼性の高いソフトウェアが要求されている．しかし，ソフトウェアの利便性を高めるため大規模化，複雑化したソフトウェアの信頼性を高く保つには，開発や保守に大きなコストを必要とする．ソフトウェアの保守性を保つためにソースコードの品質は重要である．しかし，機能の追加や変更，バグ修正などによって，ソースコードには変更が加え続けられるため，ソースコードの品質は徐々に低下していく．

このような品質の低下を抑えるために，リファクタリングという作業が行われる．リファクタリングとは，ソフトウェアの外部的な振る舞いを変えずに内部構造を改善する作業である [1]．リファクタリングは新機能の実装やバグを修正する際に頻繁に行われる [2]．リファクタリングによって，ソースコードの保守性や可読性を向上させられる一方，手動によるリファクタリングは誤ってバグを混入させてしまう恐れがあることが指摘されている [3]．

そこで，リファクタリング支援に関する研究も行われている．Ge らは開発者がリファクタリングを手動で行おうとしていることを検知し，開発者が行おうとしていたリファクタリングの続きを自動的に行うための BeneFactor というツールを開発した [4]．Ouni らはリファクタリング推薦ツールを開発した [5]．このツールは，入力されたプログラムに対してどのようなリファクタリングを行うべきかを推薦する．推薦されるリファクタリングは，元の振る舞いを保ちつつソースコードの品質ができるだけ向上し，かつ開発履歴をより利用できるようなものである．リファクタリングの推薦に開発履歴を用いるのは，過去に同じタイミングで編集されたことのあるプログラム要素は意味的に関連があるという仮説や，過去に何度もリファクタリングされたプログラム要素は将来再度リファクタリングされる可能性が高いといった仮説に基づいている．

本研究ではリファクタリング作業を自動的に行う手法を提案する．提案手法ではまず，ソフトウェアメトリクスを用いてリファクタリングすべき箇所を自動的に特定する．次に，自動プログラム修正手法を用いてメトリクスの値が改善するようにソースコードの編集を自動的に行う．例えば，サイクロマチック数を用いることで，複雑なメソッドを複数のメソッドに分解したり，複雑な条件分岐をガード節を用いて書き換えたりするリファクタリングが行える．本研究では，提案手法の一部を実装したツールを試作し，そのツールで自動的にリファクタリングが行えることを確認した．

以降，2 章で研究背景としてリファクタリングや自動プログラム修正，ソフトウェアメトリクスを紹介し，研究目的について述べる．3 章では提案手法について述べ，4 章では提案手法の一部を実装した試作ツールについて述べる．5 章では試作ツールを用いた適用実験について述べ，6 章では今後の展望について述べる．最後に，7 章では本研究のまとめと今後の課題について述べる．

2. 研 究 背 景

2.1 リファクタリング

リファクタリングとは，ソフトウェアの外部的な振る舞いを変えずに内部構造を改善する作業である．Fowler らは，リファクタリングすべきソースコードの状態と，そのリファク

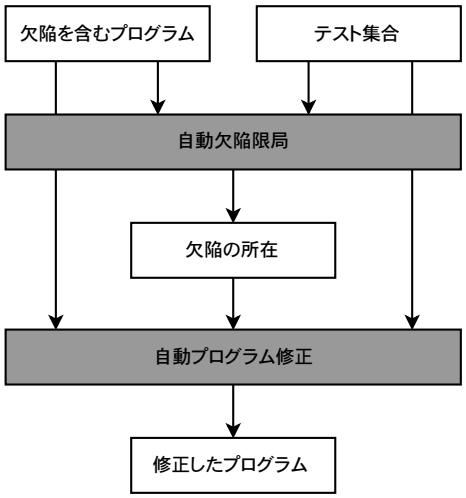

図 1　デバッグ自動化技術の概要

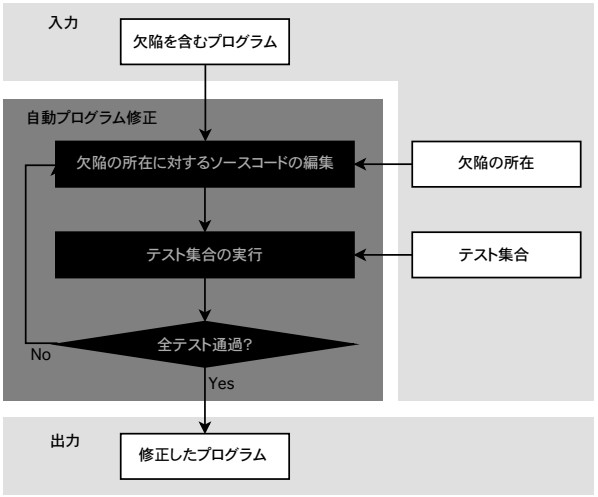

図 2　自動プログラム修正の概要

タリング方法についてまとめている [1]．リファクタリングの例としては以下のようなものがある．

メソッドの抽出　一連の手続きをメソッドに切り出し，それに目的を表すような名前を付ける

メソッドの移動　不適切なクラスで宣言されているメソッドを適切なクラスに移動する

メソッドの引き上げ　複数のサブクラスで行われている同一の処理をスーパークラスに移動する

ガード節による入れ子条件式の置換　入れ子になった `if-else` 文をガード節で置き換える

リファクタリングによって，ソースコードの保守性や可読性を向上させたり，プログラムのパフォーマンスを向上させたりすることができる [3]．その一方で，手動によるリファクタリングは誤ってバグを混入させてしまったり，元の振る舞いを破壊していないことをテストするためにコストを要するということが指摘されている [3]．また，プロジェクトの状況によってはリファクタリングを行う時間を割くことができないこともある [3]．

2.2　自動プログラム修正

近年，デバッグ支援の研究として，デバッグの自動化に向けた研究が活発に行われている．デバッグの主要な作業には以下の 2 つが挙げられる．

- 欠陥の所在の特定
- ソースコードの修正

以上の作業をそれぞれ自動化する技術として以下の 2 つがある．

- 自動欠陥限局
- 自動プログラム修正

以上のデバッグ自動化技術の概要を図 1 に示す．自動欠陥限局は欠陥を含むプログラムとテスト集合から，欠陥の所在を特定する技術である．また，自動プログラム修正は，欠陥を含むプログラムとテスト集合，欠陥の所在から，修正したプログラムを生成する．欠陥を含むプログラムとは，通過しないテストケースが少なくとも 1 つ存在するプログラムである．入力として与えるテスト集合には，欠陥を含むプログラムが通過しないテストケースと，通過するテストケースがそれぞれ少なくとも 1 つ含まれていなければならない．

次に，自動プログラム修正の処理概要を図 2 に示す．まず，自動プログラム修正は欠陥を含むプログラムにおける欠陥の所在に対し，ソースコードの変更を行う．変更方法には，同一プロジェクト内からのプログラム文の挿入や，プログラム文の削除を行う手法 [6]〜[8] や，欠陥の所在においてプログラムの分岐を行い，分岐条件をテスト集合から生成する手法 [9], [10] などがある．次に，テスト集合を実行し，全てのテストを通過する場合は修正したプログラムを出力する．通過しないテストが存在する場合は，再びソースコードの変更を行う．

2.3　jGenProg

本研究では，自動プログラム修正手法を流用し，リファクタリングを自動的に行う手法を提案する．自動リファクタリングのベースには jGenProg [7] という手法を用いた．jGenProg は遺伝的プログラミングに基づいて自動プログラム修正を行う．遺伝的プログラミングとは，プログラムの構造などの木構造で表されるデータを対象にした，解の探索を行うアルゴリズムである．遺伝的プログラミングを用いた自動プログラム修正では，欠陥を含むプログラムを基にして，ランダム性のある操作によって様々なプログラムを生成し，テスト集合を全て通過するプログラムを生成することを目指す．

jGenProg の修正手法は図 3 のようになる．初期変異プログラムの生成では，後述の変異によってプログラムに変更を加えたものを一定数生成する．次に，選択では，生成されたプログラムから修正完了に近いプログラムを一定数選択し，残りのプログラムを廃棄する．交叉では，選択で絞り込まれたプログラム集合から，2 つずつプログラムを選び，プログラム中のランダムな位置で 2 つのプログラムを入れ替える．また，変異では，欠陥の所在となる位置にプログラム文を挿入したり，削除したり，置換したりする．置換は，挿入と削除の組み合わせで表される．最後に，生成したプログラムに対しテスト集合を実行し，全てのテストを通過するプログラムを出力

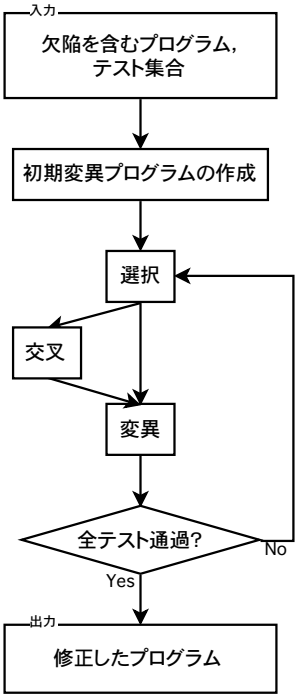

図 3　jGenProg の処理概要

する．全てのテストを通過するプログラムが存在しない場合は，選択に戻る．

2.4　ソフトウェアメトリクス

ソフトウェアメトリクスとは，ソースコードの規模，複雑さ，保守性などの性質を定量的に示す指標のことで，ソフトウェアの品質評価や工数・保守コスト予測などに用いられる尺度である [11]．代表的なソフトウェアメトリクスとしては，CK メトリクスやサイクロマチック数などがある．

CK メトリクスとは，オブジェクト指向言語におけるクラスやメソッド，フィールド間の関連性からソフトウェアの複雑度を評価するメトリクスである [12]．CK メトリクスは，WMC, DIT, NOC, CBO, RFC, LCOM という 6 つのメトリクスから構成されている．

サイクロマチック数とは，関数やメソッド内の処理の複雑さを表すメトリクスである [13]．プログラムの制御フローを有効グラフで表現したときの枝の数を e，節点の数を n としたとき，サイクロマチック数は $e - n + 2$ で表される．この値は直観的にはソースコード上の分岐の数に 1 を加えた数を表す．サイクロマチック数が大きいほど，そのメソッドの制御フローは複雑になり，保守性や可読性が低いことを意味する．

2.5　研 究 目 的

リファクタリングを行うことで，プログラムの外部的振る舞いを変更せずに，ソフトウェアの品質を向上させることができる．しかし，どのようなリファクタリングを行うかは開発者の経験や勘によるものが大きく，かえってソフトウェアの品質の低下を招いてしまうことすらある．適切なリファクタリングを行うには以下のようなことを考慮しなければならない．

- ソースコードのどこをリファクタリングすべきか
- どのようなリファクタリングをすべきか

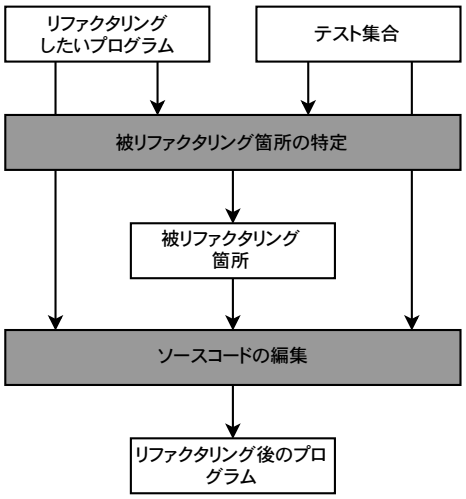

図 4　提案手法の概要

- そのリファクタリングを行うことでソフトウェアの品質はどの程度向上するか

さらに，手動によるリファクタリングは誤ってバグを混入させてしまったり，元の振る舞いを破壊していないことをテストするためにコストを要するということが指摘されている [3]．また，プロジェクトの状況によってはリファクタリングを行う時間を割くことができないこともある [3]．

このような困難を取り除くために，リファクタリング作業を自動化することが望まれる．そこで，本研究では，リファクタリングを自動で行う手法を提案する．リファクタリング自動化には，遺伝的プログラミングを用いた自動プログラム修正手法である jGenProg を利用する．遺伝的プログラミングを行うことで，より良いリファクタリングを行うことができると考える．

3.　提 案 手 法

本研究では，Java プログラムのリファクタリングを自動的に行う手法を提案する．提案手法の概要を図 4 に示す．提案手法の入力は，リファクタリングしたいプログラムとテスト集合である．出力は，リファクタリング後のプログラムである．入力として与えるテスト集合は，リファクタリングしたいプログラムの振る舞いを表しており，プログラムの仕様として解釈できる．

提案手法は以下の 2 つのステップから構成される．

ステップ 1　被リファクタリング箇所の特定

ステップ 2　ソースコードの編集

ステップ 1 とステップ 2 はそれぞれ，デバッグ自動化における自動欠陥限局と自動プログラム修正に対応する．ステップ 1 では，リファクタリングしたいプログラムのソフトウェアメトリクスを計測し，複雑で保守性が低い部分や品質が低い部分を特定する．例えば，サイクロマチック数を利用することで，複雑なメソッドや複雑な条件分岐を特定できる．使用するメトリクスは利用者が選択できるようにする．

ステップ 2 の概要を図 5 に示す．ステップ 2 では，まずス

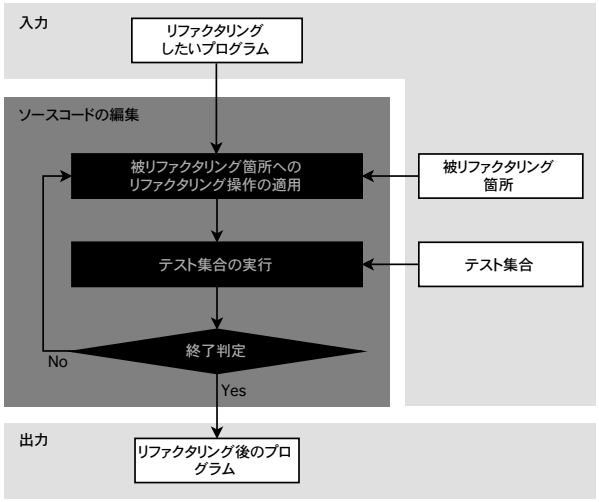

図 5　ソースコード編集の概要

テップ 1 でリファクタリングすべきと判断された箇所に対して，リファクタリング操作を行う．リファクタリング操作としては，メソッドの抽出やメソッドの移動，メソッドの引き上げ，入れ子条件のガード節による置換といった操作を予め定義しておき，それらを被リファクタリング箇所に適用する．次に，テスト集合を実行することで，リファクタリング操作を適用したプログラムが元のプログラムの振る舞いを破壊していないことを確認する．最後に終了判定では，ソフトウェアメトリクスを計測することで，リファクタリング操作の適用によりプログラムの品質が向上しているかを確認し，元のプログラムの振る舞いを保ちつつ品質が向上したプログラムを出力する．プログラムの振る舞いが破壊されるか，品質が向上していない場合は，リファクタリング操作の適用に戻る．

4.　ツールの試作

現時点では，提案手法の実装は完全に終了しておらず，一部機能のみを実装した試作の段階に留まっている．試作したツールについて述べる．

ステップ 1 に関しては，ソフトウェアメトリクスに基づいてリファクタリングすべきプログラム文を特定する処理が実装できていない．現状では，入力されたプログラムのすべてのプログラム文をリファクタリング対象とする実装になっている．

ステップ 2 に関しては，リファクタリング操作のカタログ化およびリファクタリング操作の適用処理の実装ができていない．したがって，現状では jGenProg のプログラム変更操作を用いて行えるリファクタリングのみを対象とする．つまり，被リファクタリング箇所に対して同一プログラム中に存在するプログラム文の挿入や削除，置換の組み合わせのみで表現できるリファクタリングが行える．また，ステップ 2 の終了判定では，様々なメトリクスを組み合わせることで，プログラムの品質を多角的に評価すべきであるが，現状ではプログラムの複雑さを表すサイクロマチック数が減少しているかどうかのみを判定基準に用いている．

以上の話をまとめると，試作ツールでは，同一プログラム中に存在するプログラム文の再利用の組み合わせで表現でき，かつサイクロマチック数が減少するようなリファクタリングが行える．

```
package example;

public class Example01 {
  public static int max2(int a, int b) {
    int max;

    if (a < b) {
      max = b;
    } else {
      max = a;
    }

    return max;
  }

  public static void reuseMe(int max, int a, int b)
      {
    max = Math.max(a, b);
  }
}
```

図 6　Example01.java

5.　試作ツールの適用実験

試作したツールを用いて，自動的にリファクタリングができることを確かめるために実験を行った．この実験の内容と結果について述べる．

5.1　実験対象

試作ツールでリファクタリングが行えるような 2 つのプログラム Example01.java, Example02.java を作成し，これらに対して実験を行った．実験対象のプログラムを図 6 と図 7 に示す．試作ツールでは，同一プログラム中に含まれるプログラム文の再利用の組み合わせで表現できるリファクタリングしか行えない．そのため，各プログラム中には再利用元となるプログラム文を保持した `reuseMe` メソッドを定義してある．

リファクタリングされることを期待するメソッドは `Example01#max2` および `Example02#geometricMean` である．両メソッドはそれぞれ，`if-else` 文，`for` 文を 1 つ含んでいる．そのため，両メソッドのサイクロマチック数はともに 2 である．しかし，`Example01#max2` の `if-else` 文と `Example02#geometricMean` の `for` 文は，各クラスの `reuseMe` のプログラム文で置換しても振る舞いは変わらない．さらに，このような置換を行うことで，両メソッドのサイクロマチック数はともに 1 に減少する．試作したツールによってこのような置換によるリファクタリングが行われることを期待する．

5.2　実験結果

実験結果を図 8 および図 9 に示す．両プログラムとも，期待通りのリファクタリングが行われ，サイクロマチック数が 2 から 1 へと減少している．

```java
package example;

import java.util.List;

public class Example02 {
  public static double geometricMean(List<Double>
      numbers) {
    double product = 1.0;

    for (double num : numbers) {
      product *= num;
    }

    return Math.pow(product, 1.0 / numbers.size());
  }

  public static void reuseMe(double product, List<
      Double> numbers) {
    product = Utility.product(numbers);
  }
}

class Utility {
    public static double product(List<Double>
        numbers) {
        return numbers.stream().reduce(1.0, (acc,
            succ) -> acc * succ);
    }
}
```

図 7　Example02.java

```
--- .../Example01.java
+++ .../Example01.java
@@ -4,11 +4,7 @@
 public class Example01 {
  public static int max2(int a, int b) {
    int max;
-   if (a < b) {
-     max = b;
-   }else {
-     max = a;
-   }
+   max = java.lang.Math.max(a, b);
    return max;
  }
```

図 8　Example01 の実験結果

6.　今後の展望

本研究では，Java プログラムを自動でリファクタリングする手法を提案したが，実装したツールはその一部しか実装できていない．今後実装する予定の機能を以下に述べる．

被リファクタリング箇所の特定処理

試作したツールでは，ソフトウェアメトリクスに基づく被リファクタリング箇所の特定処理が実装できておらず，すべてのプログラム文をリファクタリング対象としている．

リファクタリングの効率化やプログラムの振る舞いの破壊

```
--- .../Example02.java (original)
+++ .../Example02.java (refactored)
@@ -4,9 +4,7 @@
 public class Example02 {
  public static double geometricMean(java.util.List<
      java.lang.Double> numbers) {
    double product = 1.0;
-   for (double num : numbers) {
-     product *= num;
-   }
+   product = example.Utility.product(numbers);
    return java.lang.Math.pow(product, (1.0 / (
        numbers.size())));
  }
```

図 9　Example02 の実験結果

を避けるためにも，ソフトウェアメトリクスに基づく被リファクタリング箇所の特定処理を実装すべきである．具体的には，入力で与えられたプログラムのソフトウェアメトリクスを計測し，複雑で保守性が低い部分や品質が低い部分を特定し，そのような部分に対してのみリファクタリング操作を適用できるようにする．例えば，サイクロマチック数を利用することで，複雑なメソッドを複数のメソッドに分解したり，複雑な条件分岐をガード節を用いて書き換えたりするリファクタリングが行える．使用するメトリクスは利用者が自由に選択できるようにすることを考えている．

複数のメトリクスを組み合わせた終了条件

試作したツールでは，プログラムの品質が向上したかの判断にサイクロマチック数のみを利用している．サイクロマチック数が減少しても他の不都合が生じる可能性もあるので，複数のソフトウェアメトリクスを組み合わせて，プログラムの品質を多角的に評価すべきである．

リファクタリング操作のカタログ化・リファクタリング操作の適用処理

試作したツールでは，jGenProg のプログラム変更操作を用いて行えるリファクタリングのみを対象としている．つまり，被リファクタリング箇所に対して同一プログラム中に存在するプログラム文の挿入や削除，置換の組み合わせで表現できるリファクタリングしか行えない．

しかし，このような低レベルな操作のみで行えるリファクタリングは多くない．そこで，メソッドの抽出やメソッドの移動といった高レベルでリファクタリングに特化した操作を予めカタログ化しておき，それらを適用することを考えている．そうすることでより高品質なリファクタリングが行えると期待している．

また，以下の機能の実装も考えている．

テスト自動生成技術によるテストの強化

提案手法では，テスト集合をプログラムの仕様として解釈している．したがって，テストの質が低く，プログラムの振る舞いをきちんと表現できていなければ，元のプログラムの振る舞いを破壊するような変更を行ってしまう恐れがある．

そこで，EvoSuite [14] のようなテスト自動生成技術を用いて，自動リファクタリングを開始する前にテストを強化することで，振る舞いが破壊される可能性を下げることができるのではないかと考えている．

リポジトリマイニングによる再利用候補の収集

試作したツールでは，同一プログラム中に含まれるプログラム文の再利用の組み合わせで表現できるリファクタリングのみを対象とした．したがって，リファクタリングしたいプログラムと同様の処理が同一プログラム中に含まれている場合にしかリファクタリングが行えない．

そこで，オープンソースで開発されているライブラリのメソッドを収集しておき，それらをリファクタリングに用いることで，バリエーションに富んだリファクタリングを行える可能性がある．

遺伝的プログラミング中のコンパイル失敗の許容

試作したツールでは，遺伝的プログラミングを用いて変異プログラムを生成する際に，コンパイルに失敗する変異プログラムが生成された場合は，その変異プログラムを廃棄している．しかし，一旦コンパイルに失敗するようになったとしても，さらにプログラムに変更を加えることでコンパイルに成功し，リファクタリングに成功する可能性もある．したがって，コンパイルの失敗を許容することで行えるリファクタリングのバリエーションが増える可能性がある．

7. あとがき

本研究では，Java のプログラムを自動でリファクタリングする手法を提案した．提案手法では，ソフトウェアメトリクスに基づき被リファクタリング箇所を特定し，その箇所に対して自動的にリファクタリングを行う．リファクタリングの実行には，遺伝的プログラミングを用いた自動プログラム修正手法を流用する．また，提案手法の一部を実装したツールを試作し，適用実験を行った．その結果，期待通りのリファクタリングが行われた．

今後の課題としては，提案手法の未実装部分を実装することや，テスト自動生成技術によるテストの強化処理の実装，リポジトリマイニングによる再利用候補の収集などが挙げられる．

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：17H01725）の助成を得て行われた．

文　献

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: improving the design of existing code, Addison-Wesley Professional, 1999.

[2] E. Murphy-Hill, C. Parnin, and A.P. Black, “How we refactor, and how we know it,” IEEE Transactions on Software Engineering, vol.38, no.1, pp.5–18, 2012.

[3] M. Kim, T. Zimmermann, and N. Nagappan, “A field study of refactoring challenges and benefits,” Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software EngineeringACM, p.50 2012.

[4] X. Ge, Q.L. DuBose, and E. Murphy-Hill, “Reconciling manual and automatic refactoring,” Proceedings of the 34th International Conference on Software EngineeringIEEE Press, pp.211–221 2012.

[5] A. Ouni, M. Kessentini, H. Sahraoui, and M.S. Hamdi, “The use of development history in software refactoring using a multi-objective evolutionary algorithm,” Proceedings of the 15th annual conference on Genetic and evolutionary computationACM, pp.1461–1468 2013.

[6] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, “A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each,” Software Engineering (ICSE), 2012 34th International Conference onIEEE, pp.3–13 2012.

[7] M. Martinez and M. Monperrus, “Astor: A program repair library for java,” Proceedings of the 25th International Symposium on Software Testing and AnalysisACM, pp.441–444 2016.

[8] Z. Qi, F. Long, S. Achour, and M. Rinard, “An analysis of patch plausibility and correctness for generate-and-validate patch generation systems,” Proceedings of the 2015 International Symposium on Software Testing and AnalysisACM, pp.24–36 2015.

[9] S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference onIEEE, pp.691–701 2016.

[10] J. Xuan, M. Martinez, F. Demarco, M. Clement, S.L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, “Nopol: Automatic repair of conditional statement bugs in java programs,” IEEE Transactions on Software Engineering, vol.43, no.1, pp.34–55, 2017.

[11] N. Fenton and J. Bieman, Software metrics: a rigorous and practical approach, CRC Press, 2014.

[12] S.R. Chidamber and C.F. Kemerer, “A metrics suite for object oriented design,” IEEE Transactions on software engineering, vol.20, no.6, pp.476–493, 1994.

[13] T.J. McCabe, “A complexity measure,” IEEE Transactions on software Engineering, vol. • •, no.4, pp.308–320, 1976.

[14] G. Fraser and A. Arcuri, “Evosuite: automatic test suite generation for object-oriented software,” Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineeringACM, pp.416–419 2011.