Vol.2022-SE-210 No.29
2022/3/12

API利用パターンを用いた自動プログラム修正におけるパターン検索の予備評価

桑原 寛明[1]　渥美 紀寿[2]

概要：本稿では，API 利用パターンを用いた自動プログラム修正 [1] におけるパターン検索の予備的な評価について報告する．API 利用パターンを用いた自動プログラム修正は，API の典型的な呼び出し列を表現する API 利用パターンに従うことで API の誤った利用方法に起因するバグを自動修正する手法である．この手法では，事前に準備された API 利用パターンのデータベースから，バグの原因箇所周辺における API の利用方法と類似する API 利用パターンを検索し，見つかったパターンに合うようにソースコードを修正する．本稿では，メソッド呼び出しが 1 つ不足しているバグを対象とする予備評価の結果として，修正として適切な API 利用パターンを検索できることと，API 利用パターンのデータベースの作成が課題であることを報告する．

1. はじめに

ソフトウェア開発において大きなコストを要するデバッグを支援するために，自動プログラム修正の技術が活発に研究されている [2]．自動プログラム修正は，ソースコード中のバグをコンピュータに修正させる技術である．生成と検証に基づく自動プログラム修正は，大きくわけて (1) 修正箇所の特定，(2) 修正候補の修正，(3) 修正候補の評価の 3 ステップから構成される．(1) でソースコード中の修正すべきバグの原因箇所を特定し，特定した箇所に対する修正候補を (2) で生成する．生成された各修正候補が正しくバグを修正できるかを (3) で判定する．ソースコードの変更方法は数限りなく存在するが，その中からバグを正しく修正できる修正候補を得るために様々な修正候補生成のアプローチが研究されている．

修正候補の生成手法として，事前に準備された修正テンプレートを適用してソースコードを変更する手法があるが，様々なバグの修正には様々な修正テンプレートの準備が必要である．ソースコード片の挿入や削除の組み合わせを遺伝的アルゴリズムによって発展させながらソースコードを変更する手法も存在する．挿入するソースコード片について様々な取得元が考えられているが，取得元に含まれないソースコード片を必要とするバグの修正は難しい．

我々は，API 利用パターンを用いた自動プログラム修正を提案している [1]．これは，バグの原因箇所周辺のソースコードを API 利用パターンに合致するように変更することで，バグの修正を図る手法である．ソフトウェア開発において様々な機能を提供する API の利用は必須であるが，利用したい機能に対して呼び出すメソッドの種類と順序が決まっているなど，典型的な利用方法（以下，API 利用パターン）が存在する API も少なくない．API 利用パターンに従わない方法で API を利用することがバグの原因である場合，パターンに従うように変更することで修正できる．

API 利用パターンを用いた自動プログラム修正では，既存のソースコードをマイニングして API 利用パターンのデータベースを生成し，既存のバグ限局手法により特定された修正箇所周辺の API 利用方法と類似する API 利用パターンをデータベースから検索する．検索されたパターンと修正箇所周辺のソースコードを比較し，不足している API を追加するようなソースコードの修正候補を生成する．[1] では少数の適用例を示しているが，系統的な評価はできていない．本稿では，メソッド呼び出しが 1 つ不足している Java プログラムのバグを対象として，類似 API 利用パターンの検索に関する予備評価について報告する．

2. API 利用パターンを用いた自動プログラム修正

2.1 API 利用パターン

ソフトウェア開発では，必要とする機能を提供するライブラリやフレームワークを利用することが一般的である．ライブラリやフレームワークが提供する機能には API

[1] 南山大学 理工学部
Faculty of Science and Technology, Nanzan University
[2] 京都大学 学術情報メディアセンター
Academic Center for Computing and Media Studies, Kyoto University

(Application Programming Interface) を通してアクセスする．API の利用自体は関数呼び出しのようにシンプルであるが，実際には API が持つ制約を理解して正しく利用する必要がある．例えば，入出力系の API では，open メソッドの呼び出し 1 回に対して close メソッドの呼び出しが open メソッドの呼び出し後に 1 回だけなされるようにする，といった制約が存在することがある．このような制約を守る正しい API の利用方法は API 利用パターン [3] と呼ばれる．

必要なメソッドを呼び出していない，不要なメソッドを呼び出している，メソッドを呼び出す順序が入れ替わっている，などの API 利用パターンに従わない方法で API を利用するとバグの原因となる可能性がある．例として，API 利用パターン

```
java.util.List.stream
java.util.stream.Stream.findAny
java.util.Optional.orElse
```

と，Java コード片

```
Arrays.asList(...).stream().findAny().get();
```

を考える．asList メソッドにより配列から List を生成し，Stream API の findAny メソッドによりその中の要素を 1 つ取り出そうとしている．しかし，元の配列の長さが 0 の場合，findAny メソッドは空の Optional を返すため get メソッドの呼び出しで NoSuchElementException が発生する．Optional から値を取り出す場合は，値が確実に存在するのでなければ API 利用パターンのように orElse メソッド（あるいは orElseGet メソッド）を用いるか，isPresent メソッドで値の存在を確認することが一般的である．API 利用パターンに従って get メソッドの呼び出しを orElse メソッドの呼び出しに変更すれば例外の発生は解消される．

2.2 全体像

API 利用パターンを用いた自動プログラム修正の全体像を図 1 に示す [1]．この手法は，API 利用パターンに従わない方法で API を利用することに起因するバグは，API 利用パターンに従うように修正することで解決できる，という仮説に基づいている．手法は対象のプログラミング言語や API の利用方法に対して中立であるが，ここでは少数の必要なメソッドを呼び出していない Java プログラムの修正を対象とする．API 利用パターンを，呼び出されるメソッドの一意な名前を呼び出し式がソースコード中に出現する順に並べた列とする．

修正箇所の特定には既存のバグ限局手法を利用する．修正候補を生成するために，API 利用パターンのデータベース（以下，パターンデータベース）を事前に準備する．パターンデータベースは手作業で作成することも可能であるが，十分な規模のデータベースを得るために [1] では既存のソースコードから API 利用パターンをマイニングして構築する．修正候補の生成は，API 利用パターンの検索およびソースコード改変の 2 段階で行われる．API 利用パターンの検索では，特定された修正箇所周辺において呼び出されるメソッドの一意な名前の列（以下，バグパターン）に類似しているが同一ではない API 利用パターン（以下，正解パターン）をパターンデータベースから検索する．バグパターンを元に正解パターンを検索するため，バグパターンと正解パターンが類似していること，すなわちソースコードの大きな修正が必要ではないことが前提である．正解パターンとバグパターンを比較して修正対象のソースコードに不足しているメソッドを特定し，特定されたメソッドを呼び出すようなソースコードの改変を修正候補とする．

2.3 正解パターンの検索

正解パターンをパターンデータベースから検索するために，パターンデータベース中の API 利用パターン（以下，データベースパターン）とバグパターンの類似度を定義する．与えられたバグパターンに対して類似度の高いデータベースパターンを正解パターンとする．

定義. データベースパターン $\overline{p} = p_1, \ldots, p_m$ とバグパターン $\overline{b} = b_1, \ldots, b_n$ の類似度を，$\overline{p}$ の部分列 $\overline{p'}$ のうち以下に示す条件 $S(\overline{p'}; \overline{b})$ を満たす $\overline{p'}$ の中で $|\overline{p'}|$ の最大値を n として $n/|\overline{p}|$ と定義する．$|\overline{p}|$ は列 $\overline{p}$ の要素数である．

- $S(p; \overline{b}) = p \in \overline{b}$
- $S(p_1, \ldots, p_m; b_1, \ldots, b_n) =$
  $\exists j. p_1 = b_j \wedge S(p_2, \ldots, p_m; b_{j+1}, \ldots, b_n)$

直観的には，データベースパターンに含まれるメソッドのうちバグパターン中に順序を保ったまま出現するメソッドの割合の最大値が類似度である．なお，類似度が同一のデータベースパターンについては長い方を正解パターンとして優先する．長いパターンの方がバグパターン中に順序を保ったまま出現するメソッドの数が多く，バグパターンにより適合しているとみなす．類似度が 1 のデータベースパターンはバグパターンに含まれるメソッドのみのパターンであるため除外する．

2.4 修正候補の生成

修正候補の生成は本稿の予備評価の対象ではないため，簡単な説明にとどめる．詳細は [1] を参照されたい．正解パターンに基づいて不足しているメソッド呼び出しを追加する修正候補を以下の手順で生成する．

(1) 正解パターンとバグパターンを比較して呼び出されていないメソッドを特定する
(2) メソッド呼び出しを文として挿入できる位置を列挙する
(3) 各挿入位置についてレシーバーと引数の候補を型に基づいて列挙する
(4) レシーバーと引数の候補を組み合わせてメソッド呼び出し式を生成し，さらに返り値で初期化される変数宣

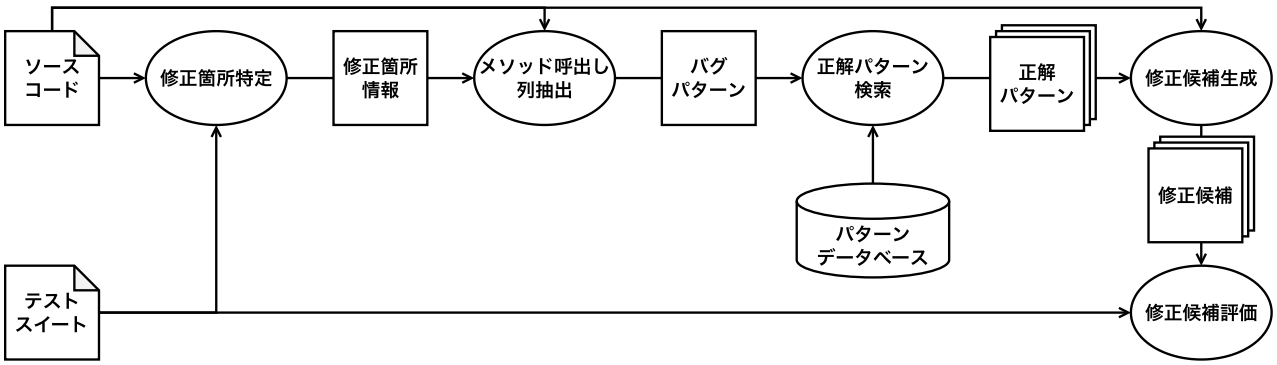

図 1 API 利用パターンを用いた自動プログラム修正

言文を生成して挿入する

(5) 挿入した変数宣言文の変数と同じ型で挿入位置以降に出現する式を宣言した変数で置換する

3. 予備評価

3.1 評価の対象と方法

本稿では，正解パターンの検索について予備評価を行う．パターンデータベースから正解パターンを検索できること，すなわちバグパターンに対する修正として適切な API 利用パターンの類似度が高くなるように，適切に類似度が定義されていることを実例を用いて確認する．

そのために，GitHub で参照可能なプロジェクトの開発履歴の中で，メソッド呼び出しを追加する修正前後のコミットのソースコードを利用する．修正後コミットのソースコード全体からパターンデータベースを構築し，修正前コミットのソースコードから抽出したバグパターンをキーとしてパターンデータベースを検索する．その結果として，修正されたソースコードに対応するデータベースパターンの類似度が高くなることを確認する．簡単のために，メソッド呼び出しが追加されるメソッド本体で呼び出されるメソッド名の列をバグパターンとする．予備評価に必要なツールは [1] とは独立に新規に実装した．

3.2 データセット

予備評価では，Defects4J[4] と MUBench[5] からメソッド呼び出しを 1 つ追加する修正を 5 種類選択して利用する．

3.2.1 Defects4J

Defects4J は，ソフトウェアテスティングの研究の支援を目的として，実プロジェクトにおけるバグを収集したデータセットである．自動プログラム修正の手法の評価にもよく利用されている．データセットには収集されたバグそれぞれについてメタデータが格納されている．プロジェクトの VCS が git の場合は git リポジトリも格納されており，メタデータに従って VCS のリポジトリを参照することでバグの修正前後のソースコードなどを入手できる．

Defects4J に含まれるプロジェクトのうち VCS に git を採用していない Chart プロジェクトを除く 17 プロジェクトを対象に，全部で 809 個のバグの修正前後のソースコードの差分を目視で確認した．その結果，あるメソッド内でメソッド呼び出しを 1 つだけ追加する差分を含む Lang プロジェクト[*1]のバグ ID38（以下，lang-38），Jsoup プロジェクト[*2]のバグ ID55（以下，jsoup-55）およびバグ ID80（以下，jsoup-80）を選択した．

```
1 @@ -869,6 +869,7 @@ public class FastDateFormat
      extends Format {
2     */
3    public StringBuffer format(Calendar calendar,
      StringBuffer buf) {
4        if (mTimeZoneForced) {
5 +          calendar.getTime(); /// LANG-538
6            calendar = (Calendar) calendar.clone();
7            calendar.setTimeZone(mTimeZone);
8        }
```

図 2 lang-38 の差分（抜粋，空白を編集）

```
1 java.util.Calendar.clone
2 java.util.Calendar.setTimeZone
```

図 3 lang-38 のバグパターン

```
1 @@ -880,6 +880,7 @@ enum TokeniserState {
2                 break;
3             default:
4                 t.error(this);
5 +               r.unconsume();
6                 t.transition(BeforeAttributeName);
7         }
8     }
```

図 4 jsoup-55 の差分（抜粋）

```
1 org.jsoup.parser.CharacterReader.consume
2 org.jsoup.parser.Tokeniser.emitTagPending
3 org.jsoup.parser.Tokeniser.transition
4 org.jsoup.parser.Tokeniser.eofError
5 org.jsoup.parser.Tokeniser.transition
6 org.jsoup.parser.Tokeniser.error
7 org.jsoup.parser.Tokeniser.transition
```

図 5 jsoup-55 のバグパターン

[*1] https://github.com/apache/commons-lang
[*2] https://github.com/jhy/jsoup

```
@@ -88,9 +88,11 @@ public class XmlTreeBuilder extends TreeBuilder {
...
-          Element el = doc.child(0);
-          insert = new XmlDeclaration(settings.normalizeTag(el.tagName()), data.startsWith("!"));
-          insert.attributes().addAll(el.attributes());
+          if (doc.childNodeSize() > 0) {
+              Element el = doc.child(0);
+              insert = new XmlDeclaration(settings.normalizeTag(el.tagName()), data.startsWith("!"));
+              insert.attributes().addAll(el.attributes());
+          } // else, we couldn't parse it as a decl, so leave as a comment
...
```

図 6 jsoup-80 の差分（抜粋）

```
org.jsoup.parser.Token$Comment.getData
org.jsoup.nodes.Comment.<init>
org.jsoup.nodes.Comment.getData
java.lang.String.length
java.lang.String.startsWith
java.lang.String.startsWith
java.lang.StringBuilder.<init>
java.lang.StringBuilder.append
java.lang.String.length
java.lang.String.substring
java.lang.StringBuilder.append
java.lang.StringBuilder.append
java.lang.StringBuilder.toString
org.jsoup.parser.Parser.xmlParser
org.jsoup.Jsoup.parse
org.jsoup.nodes.Document.child
org.jsoup.nodes.Element.tagName
org.jsoup.parser.ParseSettings.normalizeTag
java.lang.String.startsWith
org.jsoup.nodes.XmlDeclaration.<init>
org.jsoup.nodes.Node.attributes
org.jsoup.nodes.Element.attributes
org.jsoup.nodes.Attributes.addAll
org.jsoup.parser.XmlTreeBuilder.insertNode
```

図 7 jsoup-80 のバグパターン

lang-38 の差分の中でメソッド呼び出しを追加する部分を図 2，バグパターンを図 3 に示す．`org.apache.commons.lang3.time.FastDateFormat` クラスの `format(Calendar,StringBuffer)` メソッドに `java.util.Calendar.getTime` の呼び出しを追加する修正であり，図 3 の先頭に追加される．

jsoup-55 の差分の中でメソッド呼び出しを追加する部分を図 4，バグパターンを図 5 に示す．列挙型 `org.jsoup.parser.TokeniserState` の定数 `SelfClosingStartTag` が実装する `read` メソッドに `org.jsoup.parser.CharacterReader.unconsume` の呼び出しを追加する修正であり，図 5 の 6 行目と 7 行目の間に追加される．

jsoup-80 の差分の中でメソッド呼び出しを追加する部分を図 6，バグパターンを図 7 に示す．`org.jsoup.parser.XmlTreeBuilder` クラスの `insert(Token.Comment)` メソッドに `org.jsoup.nodes.Document.childNodeSize` の呼び出しを追加する修正であり，図 7 の 15 行目と 16 行目の間に追加される．

3.2.2 MUBench

MUBench は，API の誤った利用方法（API misuses，以下，API ミスユース）を検出する手法を比較あるいは評価するためのベンチマークであり，実プロジェクトおいて出現した API ミスユースを収集したデータセットである．MUBench も Defects4J と同様に，収集された API ミスユースそれぞれについてメタデータが格納されており，VCS のリポジトリから修正前後のソースコードなどを入手できる．

MUBench[*3] には，67 プロジェクトから収集された 275 個の API ミスユースが含まれている．そのうちカテゴリが missing/call に分類されており，かつ，GitHub で参照できる修正コミットが存在する 61 個の API ミスユースについて，修正前後のソースコードの差分を目視で確認した．その結果，jodatime プロジェクト[*4] の ID が 276（以下，jodatime-276），および tbuktu-ntru プロジェクト[*5] の ID が 476（以下，ntru-476）の API ミスユースを選択した．

jodatime-276 の差分を図 8，バグパターンを図 9 に示す．`org.joda.time.TestDays` クラスの `testSerialization` メソッドに `java.io.ByteArrayOutputStream.toByteArray` メソッドの呼び出しを追加する（厳密には `toByteArray` メソッドと `close` メソッドの呼び出し順を入れ替える）修正であり，図 9 の 4 行目と 5 行目が入れ替わる．

ntru-476 の差分を図 10，バグパターンを図 11 に示す．`net.sf.ntru.sign.SignaturePublicKey` クラスの `getEncoded` メソッドに `java.io.DataOutputStream.close` メソッドの呼び出しを追加する修正であり，図 11 の 6 行目と 7 行目の間に追加される．

3.3 パターンデータベースの構築

選択したバグあるいは API ミスユースごとに，PAM (Probabilistic API Mining)[6] を用いて得られる API 利用パターンの集合から長さが 2 以上のパターンのみを集めてパ

[*3] コミット ID: 44558f3fc29e3fc50f26acdda76e5b447e0a31d9
[*4] `https://github.com/emopers/joda-time`（`https://github.com/JodaOrg/joda-time` のフォークだが，MUBench ではフォーク先でのコミット ID が記録されている．修正コミットはフォーク元にプルリクエストされマージされている．）
[*5] `https://github.com/emopers/ntru`（`https://github.com/tbuktu/ntru` のフォークであり，jodatime と同様．）

```
@@ -240,8 +240,8 @@ public class TestDays extends TestCase {
         ByteArrayOutputStream baos = new ByteArrayOutputStream();
         ObjectOutputStream oos = new ObjectOutputStream(baos);
         oos.writeObject(test);
-        byte[] bytes = baos.toByteArray();
         oos.close();
+        byte[] bytes = baos.toByteArray();

         ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
         ObjectInputStream ois = new ObjectInputStream(bais);
```

図 8 jodatime-276 の差分（抜粋）

```
java.io.ByteArrayOutputStream.<init>
java.io.ObjectOutputStream.<init>
java.io.ObjectOutputStream.writeObject
java.io.ByteArrayOutputStream.toByteArray
java.io.ObjectOutputStream.close
java.io.ByteArrayInputStream.<init>
java.io.ObjectInputStream.<init>
java.io.ObjectInputStream.readObject
java.io.ObjectInputStream.close
org.joda.time.TestDays.assertSame
```

図 9 jodatime-276 のバグパターン

```
@@ -92,6 +92,7 @@ public class SignaturePublicKey {
         dataStream.writeShort(h.coeffs.length);
         dataStream.writeShort(q);
         dataStream.write(h.toBinary(q));
+        dataStream.close();
     } catch (IOException e) {
         throw new NtruException(e);
     }
```

図 10 ntru-476 の差分（抜粋）

```
java.io.ByteArrayOutputStream.<init>
java.io.DataOutputStream.<init>
java.io.DataOutputStream.writeShort
java.io.DataOutputStream.writeShort
net.sf.ntru.polynomial.IntegerPolynomial.toBinary
java.io.DataOutputStream.write
net.sf.ntru.exception.NtruException.<init>
java.io.ByteArrayOutputStream.toByteArray
```

図 11 ntru-476 のバグパターン

ターンデータベースとする．PAM は入力として与えられたメソッド名の列の集合をマイニングしてパターンの集合を出力する．PAM を実行する際のコマンドラインオプションは，入力元と出力先の指定の他はデフォルト値を用いる．

プロジェクト内で宣言される各メソッドごとに，その本体の中で呼び出されるメソッドの完全修飾名の列を抽出して PAM への入力を生成する．完全修飾名はクラスファイルから簡単に得られるため，Java バイトコードの静的解析ライブラリである SOBA[7] を用いてクラスファイルを解析し，呼び出されるメソッドの完全修飾名をクラスファイル中に出現する順序で並べる．

今回の予備評価では，呼び出されるメソッドすべてではなく，修正で追加されるメソッドの完全修飾名と接頭辞（の一部）が共通する完全修飾名のメソッドのみを抽出した．すべてのメソッドを抽出する場合，標準ライブラリなど他のパッケージのメソッドが多数呼び出されている等の理由で追加したいメソッドがマイニングの過程で消えてしまう可能性が高くなる．

表 1 呼び出し先メソッドの完全修飾名の抽出元

	コミット ID	抽出元クラスファイル	接頭辞
lang-38	825481f0	target/classes/ 以下	java.util
jsoup-55	f0f0e41e	target/classes/ 以下	org.jsoup
jsoup-80	e9feec90	target/classes/ 以下	org.jsoup
jodatime-276	0e82517a	target/test-classes/ 以下	java.io
ntru-476	4fd7cb8b	target/classes/ 以下	java.io

```
# similarity: 0.667
java.util.Calendar.getTime
java.util.Calendar.clone
java.util.Calendar.setTimeZone

# similarity: 0.333
java.util.Calendar.equals
java.util.NoSuchElementException.<init>
java.util.Calendar.clone

# similarity: 0.2
java.util.Calendar.clone
java.util.Calendar.add
java.util.Calendar.add
java.util.Calendar.add
java.util.Calendar.add
```

図 12 lang-38 の検索結果

呼び出されるメソッドの完全修飾名の抽出元のコミット ID とクラスファイルの位置，および抽出対象のメソッドの完全修飾名の接頭辞を表 1 に示す．jodatime-276 は，修正対象がテストコードであるためテストコードから抽出する．

3.4 API 利用パターンの検索結果

それぞれのバグパターンに対し，パターンデータベースから検索された API 利用パターンのうち類似度順に上位 3 個（ただし類似度 1 のパターンは除く）とそれぞれの類似度を図 13 から図 16 に示す．なお，図 14 の jsoup-80 の検索結果は，類似度が 0.5 で最も長いパターンが 6 個ある中からランダムに選択した 3 個である．

lang-38 と ntru-476 では適切なパターンが検索できてい

```
# similarity: 0.8
org.jsoup.parser.CharacterReader.consume
org.jsoup.parser.Tokeniser.transition
org.jsoup.parser.Tokeniser.transition
org.jsoup.parser.Tokeniser.eofError
org.jsoup.parser.Tokeniser.transition

# similarity: 0.8
org.jsoup.parser.CharacterReader.consume
org.jsoup.parser.Tokeniser.transition
org.jsoup.parser.Tokeniser.transition
org.jsoup.parser.Tokeniser.transition
org.jsoup.parser.Tokeniser.transition

# similarity: 0.75
org.jsoup.parser.CharacterReader.consume
org.jsoup.parser.Tokeniser.transition
org.jsoup.parser.Tokeniser.transition
org.jsoup.parser.Tokeniser.error
org.jsoup.parser.Tokeniser.eofError
org.jsoup.parser.Tokeniser.transition
org.jsoup.parser.Tokeniser.error
org.jsoup.parser.Tokeniser.transition
```

図 13 jsoup-55 の検索結果

```
# similarity: 0.5
org.jsoup.parser.ParseSettings.normalizeTag
org.jsoup.nodes.Element.nodeName

# similarity: 0.5
org.jsoup.nodes.Node.attributes
org.jsoup.nodes.Attributes.iterator

# similarity: 0.5
org.jsoup.nodes.Element.attributes
org.jsoup.nodes.Attributes.getIgnoreCase
```

図 14 jsoup-80 の検索結果

```
# similarity: 0.889
java.io.ByteArrayOutputStream.<init>
java.io.ObjectOutputStream.<init>
java.io.ObjectOutputStream.writeObject
java.io.ObjectOutputStream.close
java.io.ByteArrayOutputStream.toByteArray
java.io.ByteArrayInputStream.<init>
java.io.ObjectInputStream.<init>
java.io.ObjectInputStream.readObject
java.io.ObjectInputStream.close

# similarity: 0.75
java.io.FileInputStream.<init>
java.io.ObjectInputStream.<init>
java.io.ObjectInputStream.readObject
java.io.ObjectInputStream.close

# similarity: 0.5
java.io.ByteArrayInputStream.<init>
java.io.FileOutputStream.<init>
```

図 15 jodatime-276 の検索結果

る．ntru-476 の場合，図 11 のバグパターンと図 16 の類似度が最大の検索結果を比較すると，java.io.DataOutputStream.close の呼び出しが writeShort メソッドと toByteArray メソッドの呼び出しの間に存在しない．そこで，バグパターンの 4 行目から 8 行目の間に close メソッドの呼び出しを追加すればよいことがわかる．

```
# similarity: 0.8
java.io.ByteArrayOutputStream.<init>
java.io.DataOutputStream.writeShort
java.io.DataOutputStream.writeShort
java.io.DataOutputStream.close
java.io.ByteArrayOutputStream.toByteArray

# similarity: 0.667
java.io.DataOutputStream.<init>
java.io.DataOutputStream.writeFloat
java.io.DataOutputStream.write

# similarity: 0.071
java.io.DataOutputStream.writeInt
java.io.DataOutputStream.writeInt
java.io.DataOutputStream.writeInt
java.io.DataOutputStream.writeInt
java.io.DataOutputStream.writeInt
java.io.DataOutputStream.writeInt
java.io.DataOutputStream.writeInt
java.io.DataOutputStream.writeInt
java.io.DataOutputStream.writeInt
java.io.DataOutputStream.writeBoolean
java.io.DataOutputStream.writeBoolean
java.io.DataOutputStream.write
java.io.DataOutputStream.writeUTF
java.io.DataOutputStream.flush
```

図 16 ntru-476 の検索結果

jsoup-55 と jsoup-80 では適切なパターンが検索できていないが，そもそも適切なパターンがパターンデータベースに含まれていなかった．jsoup-55 では，org.jsoup.parser.CharacterReader.unconsume の呼び出しを追加したいが，PAM への入力となる呼び出し先メソッド名の抽出元である 1009 個のメソッド宣言のうち error と unconsume を続けて呼び出すメソッドは 2 個だけであり，パターンとしてマイニングされていない．jsoup-80 では，org.jsoup.nodes.Document.childNodeSize の呼び出しを含むパターンはデータベース中に存在するが，いずれも childNode メソッドや addChildren メソッドとともに呼び出されており，バグパターンとは異なる利用パターンであった．

jodatime-276 では，修正後のメソッド呼び出し列そのものがパターンとして検索されている．呼び出し先メソッド名の抽出元であるメソッド宣言が 63 個あるが，およそ半数のメソッドが検索されたパターンと同等のメソッド呼び出しを行っているため，パターンとしてマイニングされたと予想される．

4. 考察

4.1 API 利用パターンの類似度

メソッド呼び出しを 1 つ追加する修正に限定された小規

模なデータセットによる予備評価の範囲内ではあるが，修正として適切な API 利用パターンがパターンデータベースに含まれていれば，そのパターンをバグパターンとの類似度が高いパターンとして検索できることを確認した．一方，含まれていない場合，パターンデータベースの内容によって，データベース中のいずれのパターンもバグパターンとの類似度が低くなるか，バグパターンとの類似度は高いが修正として適切ではないパターンが検索される．

類似度の定義から，出現順が一致することを前提に，バグパターン中に出現しないメソッド呼び出しが少なく（ただし 0 個ではない），かつ長いパターンほど類似度が高くなる．これは，API の利用方法を大きく間違えているソースコードではなく少しだけ間違えていることがバグの原因となっているソースコードの修正を対象としているためであり，修正として適切な API 利用パターンを含むパターンデータベースを構築できるのであれば妥当であると言える．

実際に適用することを考えると，バグパターンとの類似度が最も高いパターンだけでなく，上位 k 件あるいは類似度が一定値以上のパターンを検索し，それぞれについて正しく修正できるか確認するという使い方になると想定される．この場合，修正として適切ではないパターンが検索結果にある程度紛れ込むことは許容できるため，プログラムの修正に有用な API 利用パターンをパターンデータベースに含めることが重要になる．

4.2 パターンデータベース

パターンデータベースは，プログラムの修正に有用な API 利用パターンを含み，修正には役立たない API 利用パターンを含まないことが理想であるが，このようなパターンデータベースを構築することは容易ではない．今回の予備評価では，修正として適切な API 利用パターンが含まれることを意図して，修正後のソースコードからパターンデータベースを構築した．しかし，意図通りに構築されない場合もあった．

パターンデータベースの構築に際して，API 利用パターンの抽出に利用するツールとソースコード群を決める必要があるが，いずれについても抽出された API 利用パターンをプログラム修正に応用するという観点から評価するための基準がなく，簡単には決められない．ツールとして本稿では PAM を利用したが，コマンドラインツールとして簡単に実行できる実装が公開されている[*6]ことと，[6] における評価では MAPO[8] や UP-Miner[9] と比較してよい結果が得られていることによる．一方，PAM よりもよい結果が得られることを主張する FOCUS[10] も存在する．FOCUS の公開されている実装[*7]は手法の評価に特化しているため本稿では利用していないが，PAM 以外のツールを活用することは今後の課題である．ソースコード群について，どのようなソースコードを収集すればよいのか明らかではない．予備評価では意図的に修正後のソースコードを利用したが，実際の開発において同じ手段は採れないため，既存のソースコード群から構築せざるを得ない．高品質である（と一般には信じられている）プロジェクトのソースコード群を利用することは考えられるが，構築されたパターンデータベースの品質を評価する手段が必要である．

パターンデータベースを手作業で構築することも可能である．例えば，API の開発者が提供する利用方法の解説文書やプログラム例から API 利用パターンを抽出することでパターンデータベースの高品質化を期待できる．ただし，解説や例が提供されていないことも多く，多数の API を網羅することは難しい．

[*6] https://github.com/mast-group/api-mining
[*7] https://github.com/crossminer/FOCUS

4.3 データセット

今回の予備評価では，メソッド呼び出しを 1 つ追加する修正に限定して 5 種類の修正を利用したが，プログラム修正に関する既存のデータセットである Defects4J と MUBench から条件を満たすものを探してこれらを発見した．この過程で，データセットから取得できる修正前後のソースコードの差分を目視で確認したが，差分の中に図 4 や図 10 のような少数のメソッド呼び出しの追加や削除のみを含む hunk が出現することは非常に少ない．

API 利用パターンの類似度の定義や構築されたパターンデータベースの品質を評価するためには，ある程度の規模のデータセットが必要である．BID3[11], ManySStuBs4J[12], BugSwarm[13] など他のデータセットの調査や，GitHub などから収集する手法の検討が必要がある．

5. 関連研究

API 利用パターンを用いた自動プログラム修正は [1] の他に [14] でも提案されている．マイニングされた API 利用パターン群から修正に適切なパターンを検索するという考え方は同じであるが，修正に適切なパターンを得やすくするために，修正箇所周辺に出現するクラス名やメソッド名を用いてマイニング対象を選別する点が異なる．この場合，修正ごとに API 利用パターンのマイニングが行われる．

何らかのパターンに着目した自動プログラム修正として，コード記述パターンから素材コード片を生成して自動プログラム修正を実現する手法 [15] や，プログラム依存グラフに基づく Systematic Edit Pattern を利用した自動プログラム修正 [16] が提案されている．

[14] でも指摘されているが，[1] や [14] の手法を API 利用パターンを用いた API ミスユースの修正とみなし，バグ限局ではなく API ミスユースの検出により修正箇所を特定する方法もあり得る．本稿の予備評価で用いた MUBench[5] に基づいて，12 種類の API ミスユース検出器を評価する研

究 [17] も行われている．大半の検出器は内部で利用パターンをマイニングしており，API ミスユースの検出と修正は近いトピックである．API ミスユースの修正に関する最近の研究として，ファジィ論理を利用した手法 [18] や機械学習を利用した手法 [19] が提案されている．

6. おわりに

本稿では，API 利用パターンを用いた自動プログラム修正におけるパターン検索に対する予備評価を行った．実際の Java プロジェクトにおいて修正された，メソッド呼び出しが 1 つ不足していたバグを対象として，修正後ソースコードを用いて構築されたパターンデータベースから，修正前ソースコードのバグパターンに類似したパターンを検索できるか確認した．

小規模なデータセットを用いた予備評価の結果ではあるが，パターンデータベースに修正として適切な API 利用パターンが含まれていれば，そのパターンとバグパターンの類似度は高くなり検索できることが確認できた．一方，修正として適切ではないパターンの中にもバグパターンとの類似度が高くなるものが存在することも確認できた．検索されたパターンに基づいて正しく修正できることはテストなどにより別途確認されるため，不適切なパターンが検索されることが直ちに問題となるわけではない．修正に適切なパターンが含まれるパターンデータベースを構築することが重要であり，その方法を明らかにする必要がある．

今後の課題として，2 つ以上のメソッド呼び出しの追加や削除を行う修正を用いて評価を行うこと，API 利用パターンの類似度の定義や構築されたパターンデータベースの品質を評価するためのデータセットを構築すること，高品質なパターンデータベースの構築手法を明らかにすることなどが挙げられる．

謝辞 本研究の一部は JSPS 科研費 JP18K11241 および 2021 年度南山大学パッヘ研究奨励金 I-A-2 の助成による．

参考文献

[1] 荒木良仁，桑原寛明，國枝義敏：API 利用パターンを用いた自動プログラム修正手法，情報処理学会研究報告ソフトウェア工学，Vol. 2021-SE-207, No. 3, pp. 1–8 (2021).
[2] Gazzola, L., Micucci, D. and Mariani, L.: Automatic Software Repair: A Survey, *IEEE Transactions on Software Engineering*, Vol. 45, No. 1, pp. 34–67 (2019).
[3] Robillard, M. P., Bodden, E., Kawrykow, D., Mezini, M. and Ratchford, T.: Automated API Property Inference Techniques, *IEEE Transactions on Software Engineering*, Vol. 39, No. 5, pp. 613–637 (2013).
[4] Just, R., Jalali, D. and Ernst, M. D.: Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs, *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, ACM, pp. 437–440 (2014).
[5] Amann, S., Nadi, S., Nguyen, H. A., Nguyen, T. N. and Mezini, M.: MUBench: A Benchmark for API-Misuse Detectors, *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, ACM, pp. 464–467 (2016).
[6] Fowkes, J. and Sutton, C.: Parameter-Free Probabilistic API Mining Across GitHub, *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, ACM, pp. 254–265 (2016).
[7] 秦野智臣，石尾　隆，井上克郎：SOBA: シンプルな Java バイトコード解析ツールキット，コンピュータソフトウェア，Vol. 33, No. 4, pp. 4_4–4_15 (2016).
[8] Zhong, H., Xie, T., Zhang, L., Pei, J. and Mei, H.: MAPO: Mining and Recommending API Usage Patterns, *ECOOP 2009 – Object-Oriented Programming* (Drossopoulou, S., ed.), Springer Berlin Heidelberg, pp. 318–343 (2009).
[9] Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T. and Zhang, D.: Mining Succinct and High-Coverage API Usage Patterns from Source Code, *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, IEEE Press, pp. 319–328 (2013).
[10] Nguyen, P. T., Di Rocco, J., Di Ruscio, D., Ochoa, L., Degueule, T. and Di Penta, M.: FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns, *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, IEEE Press, pp. 1050–1060 (2019).
[11] Li, X., Jiang, J., Benton, S., Xiong, Y. and Zhang, L.: A Large-scale Study on API Misuses in the Wild, *14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 241–252 (2021).
[12] Karampatsis, R.-M. and Sutton, C.: How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset, *Proceedings of the 17th International Conference on Mining Software Repositories*, Association for Computing Machinery, pp. 573–577 (2020).
[13] Tomassi, D. A., Dmeiri, N., Wang, Y., Bhowmick, A., Liu, Y.-C., Devanbu, P. T., Vasilescu, B. and Rubio-González, C.: BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes, *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, IEEE Press, pp. 339–349 (2019).
[14] Nielebock, S.: Towards API-specific Automatic Program Repair, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, IEEE Press, pp. 1010–1013 (2017).
[15] 安田和矢, 伊藤信治, 中村知倫, 原田真雄, 肥後芳樹：コード記述パターンに基づく素材コード片生成による自動プログラム修正手法，コンピュータソフトウェア，Vol. 38, No. 4, pp. 4_23–4_32 (2021).
[16] 野田訓広，横山晴樹，菊池慎司：プログラム依存グラフに基づく Systematic Edit Pattern を利用した自動プログラム修正，信学技報，SS2020-30, Vol. 120, No. 407, pp. 13–18 (2021).
[17] Amann, S., Nguyen, H. A., Nadi, S., Nguyen, T. N. and Mezini, M.: A Systematic Evaluation of Static API-Misuse Detectors, *IEEE Transactions on Software Engineering*, Vol. 45, No. 12, pp. 1170–1188 (2019).
[18] Nguyen, T. T., Vu, P. M. and Nguyen, T. T.: API Misuse Correction: A Fuzzy Logic Approach, *Proceedings of the 2020 ACM Southeast Conference*, ACM SE '20, Association for Computing Machinery, pp. 288–291 (2020).
[19] Nielebock, S., Heumüller, R., Krüger, J. and Ortmeier, F.: Using API-Embedding for API-Misuse Repair, *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, Association for Computing Machinery, pp. 1–2 (2020).