類似コード検出ツールを用いたテストコード再利用に向けた調査

ソフトウェア設計学研究室 M2

1811098 倉地亮介

背景

- ソフトウェアに求められる要件が高度化・多様化する一方で，ユーザからはソフトウェアの品質確保やコスト削減に対する要求も増している

- ソフトウェアテストは開発全体のコストに占める割合が大きく，品質確保の要である
  - 現状ではテスト作成作業の大部分が人手で行われており，多くのテストを作成しようとするとそれに比例してコストも増加してしまう

ソフトウェアの品質を確保しつつコスト削減を達成するために様々なテストコード自動生成ツールが提案されている

課題

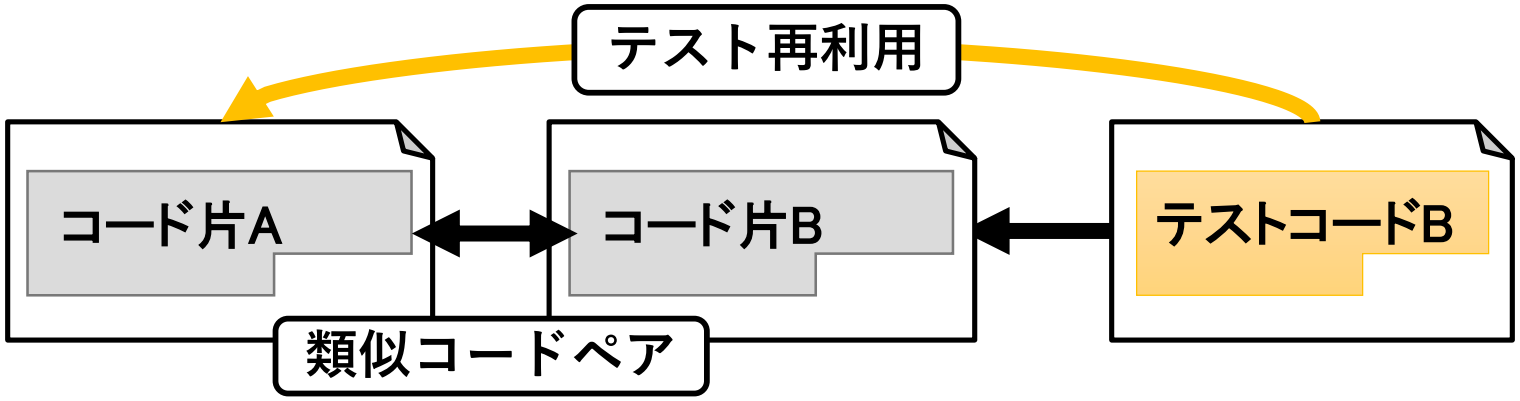
- 自動生成されたテストコードは，保守作業を困難にする[1]
  - 自動生成されたテストコードは，実際の対象コード作成の経緯や意図に基づいて生成されていないので開発者は理解しにくい
  - 開発者は自動生成されたコードを信用していない

➢テスト失敗の原因がテストコードの問題なのか，テスト対象のコードによるものなのか判断が難しい

[1] S.Shamshiri,J.Rojas,J.Pablo Galeotti,N.Walkinshaw,G.Fraser . How Do Automatically Generated Unit Tests Influence Software Maintenance?. Proc. of ICST, pp.239–249, 2018.

提案する自動生成ツール

- 既存テストの再利用によるテストコード自動生成ツールが必要である
  - 命名規則に従った保守性の高いテストコードを利用できる
  - 人によって作成された信頼性の高いテストコードを利用できる
- 類似コード間でのテスト再利用手法を提案



類似コード間でのテスト再利用手法の概要

研究動機

- ソースコードの再利用は困難な作業と考えられている[2]
  - ソースコードの内容を理解しなければならない
  - 再利用後にソースコードの修正が必要である
- テストコードの再利用も同様に難しいと考える
  - テスト対象となる類似コードペア間の関係に依存する
  - 再利用の適用対象となる類似コードペアが存在しないとできない

テスト再利用を支援するために，どのようなテストコードが
類似コード間で再利用できるのかを明らかにする必要がある

[2] Will Tracz. Confessions of a used-program salesman: Lessons learned. In Proceedings of the 1995 Symposium on Software Reusability, SSR '95, pp. 11–13, New York, NY, USA, 1995. ACM.

研究概要

- 既存プロジェクト内の類似コードのペアをテストコードの有無によって分類し，以下の調査を実施した

調査1. プロジェクト内にテストコードの再利用の適用対象となる類似コードのペアはどの程度存在するか？

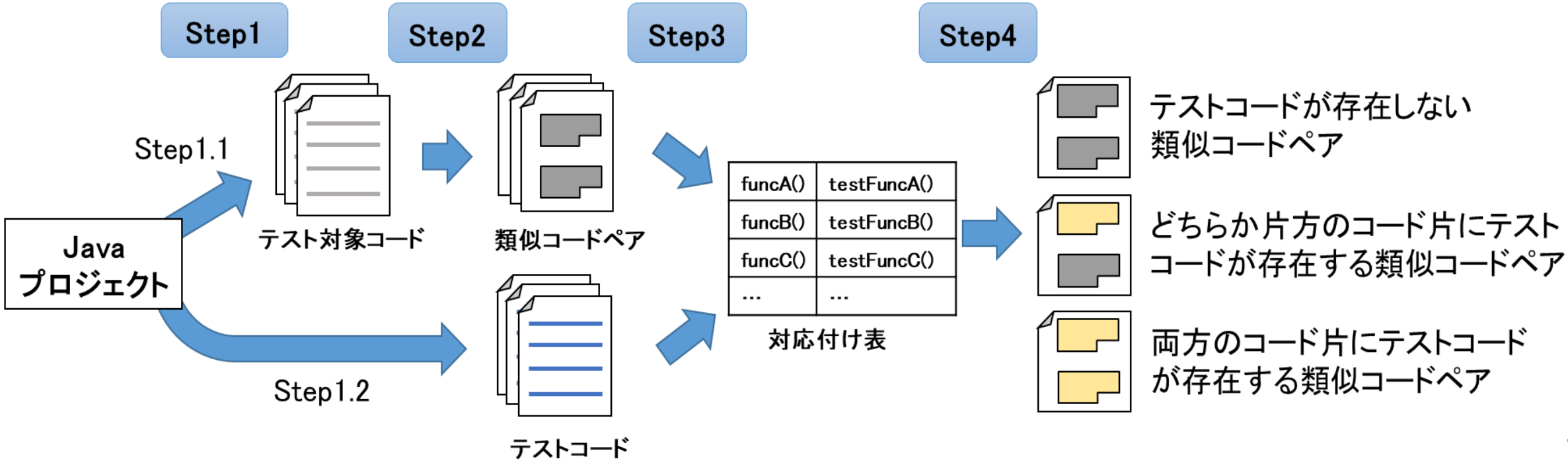
- プロジェクト内で類似コード間のテスト再利用手法がどの程度有効なのか

調査2. 類似コードペア間の類似度と対応するテストコードペア間の類似度はどのような関係があるか？

- 類似コードペア間の類似度が高いほど，テスト再利用の可能性が高くなるのではないか

類似コードペアの分類手法

Step1：プロジェクト内のテストコードとテスト対象コードを収集
Step2：類似コードペアの検出
Step3：テストコードと類似コード片をメソッド単位で対応付け
Step4：テストコードの有無によって類似コードペアを分類



Step1：テストコードとテスト対象コードの収集

- Step1.1：プロジェクト内のテストコードを収集
  - Javaのテストフレームワーク(JUnit)を用いた単体テストを対象
  - ファイル名の先頭または末尾に ”Test”が含まれるファイルを収集
    - 例：CalculatorTest.java , TestCalculator.java

- Step1.2：テスト対象コードの収集
  - テストファイル名から”Test”を除いたファイル名を持つソースコードをテスト対象コードとする
    - 例：Calculator.java

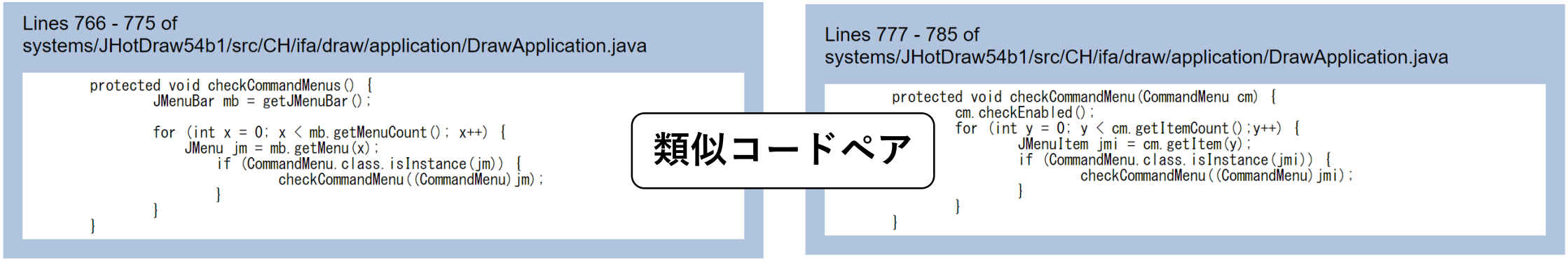
テスト対象コード
(Calculator.java)

テストコード
(CalculatorTest.java)

Step2：類似コードの検出

- 類似コード検出ツール：Nicad[3]
  - コードのレイアウトを変換させ行単位でソースコードを比較し類似コードを検出
  - 高精度・高再現率で類似コードを検出可能

➢ Nicad 標準の設定でテスト対象のコードに対して類似コードを検出する



Nicadでの検出結果

[3] Chanchal, K. R. and James, R. C.: NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, Proc. of ICPC 2008, pp. 172–181 (2008).

Step3：類似コードとテストコードの対応付け

- テスト対象となるオブジェクトのメソッド呼び出しを基にテストコードと類似コードをメソッド単位で対応付ける

  ① テストコードを静的解析し，メソッド呼び出しを確認

  ② テストメソッドを区切り文字や大文字で分割し，部分一致した時対応付ける

  - 例：testMultipyOfTwoNumbers ⇒ test + Multipy + Of + Two + Numbers

```
public class Calculator {
    public int multiply(int x, int y) {
        return x * y;
    }
}
```

類似コード片(テスト対象コード)

```
@Test
public void testMultipyOfTwoNumbers()
throws Exception {
    Calculator calc = new Calculator();
    int expected = 200;
    int actual = calc.multiply(10,20);
    assertEquals(expected,actual);
}
```

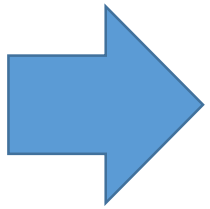
テストコード

Step4：類似コードペアの分類

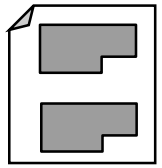
- 対応付け表を基にテストコードの有無によって類似コードペアを3種類に分類する

対応付け表

類似コードペア	テスト対象コード	テストコード
Clone Pairs 1	fuctionA	×
	fuctionB	×
Clone Pairs 2	fuctionC	testFucitonC
	fuctionD	×
Clone Pairs 3	fuctionE	testFuctionE
	fuctionF	testFuctionF

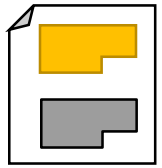


類似コードペアの分類



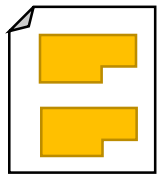
Clone Pairs 1

テストコードが存在しない類似コードペア



Clone Pairs 2

どちらか片方のコード片にテストコードが存在する類似コードペア



Clone Pairs 3

両方のコード片にテストコードが存在する類似コードペア

調査1：プロジェクト内に再利用候補になる類似コードペアはどの程度存在するか？

- 3つの有名 javaプロジェクト maven，kafka，kylin を調査

プロジェクト名	両方のコード片にテストコードが存在する類似コードペア		どちらか片方のコード片にテストコードが存在する類似コードペア		テストコードが存在しない類似コードペア		合計
	数	割合(%)	数	割合(%)	数	割合	数
Apache maven	8	2.0	139	34.2	260	63.9	407
Apache kafka	47	7.5	135	21.6	442	70.8	624
Apache kylin	7	2.9	60	24.5	177	72.2	245
合計	62	4.9	334	26.2	879	68.9	1275

プロジェクト中のテスト対象となる類似コードペアの内，約26%の類似コードペアが再利用対象になる

調査2：類似コードペア間の類似度と対応するテストコードペア間の類似度にはどのような関係があるか？

- 「両方のコード片にテストコードが存在する類似コードペア」62個の類似度と対応する153個のテストコードペアの類似度をタイプ別に分類[4]

種類	意味	類似度
タイプ1	レイアウト・空白・コメントの違いを除き完全に一致している	高
タイプ2	タイプ1に加え変数名・型の違いを除き構文的に一致している	⇑
タイプ3	タイプ2に加え文が挿入・削除・変更されている	低

Not Similar : タイプ3以上の変更・違いがあり類似していない

[4]C. K. Roy, J. R. Cordy, R. Koschke. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Science of Computer Programming, Vol. 74, No. 7, pp. 470–495, 2009.

調査2：類似コードペア間の類似度と対応するテストコードペア間の類似度にはどのような関係があるか？

- 「両方のコード片にテストコードが存在する類似コードペア」62個の類似度と対応する153個のテストコードペアの類似度をタイプ別に分類した結果

	テストコードペア間の類似度			
類似コードペア間の類似度		タイプ2	タイプ3	Not Similar
	タイプ2	77	10	6
	タイプ3	28	5	10
	Not Similar	0	0	17

類似コードペア間の類似度が高いほど，テストコードペア間の類似度も高い

調査2：類似コードペアの例

- 振る舞いが異なる類似コードペアの例

```
public V get(final K key) {
    try {
        if (allTime.shouldRecord()) {
            return measureLatency(get(key), getTime);
        } else {
            return outerValue(wrapped().get(key));
        }
    } catch (final ProcessorStateException e) {
        …
    }
}
```

```
public V delete(final K key) {
    try {
        if (allTime.shouldRecord()) {
            return measureLatency(delete(key), deleteTime);
        } else {
            return outerValue(wrapped().delete(key));
        }
    } catch (final ProcessorStateException e) {
        …
    }
}
```

同じ制御構造を持つが，最後に出力する値だけが異なる

まとめ・今後の計画

- まとめ
  - 類似コードを用いて，既存のテストコードを再利用するツールを提案
  - ツールの実現に向けて類似コードペアと対応するテストコードの関係を調査
- 今後の計画
  - 類似コードペア間の振る舞いに着目し，テストコードとの関係を調査
    - メソッド呼び出しの差異に基づく類似コードの分類手法を用いて類似コードペア間の振る舞いを確認する
  - 検出されたテストコードの保守性について評価する
    - 既存研究で定義されているTest Smell(よくない実装のテストコード)を基にテストコードの品質を評価

ソフトウェア開発にかかる費用

- ソフトウェア開発各工程での費用

	コストの割合	「運用と保守」を除いた割合
要求分析	3%	9%
仕様書	3%	9%
設計	5%	15%
コーディング	7%	21%
テスト	15%	46%
運用と保守	67%	—

「基本から学ぶソフトウェアテスト」Cem Kaner, Jack Falk, Hung Quoc Nguyen著,テスト技術者交流会訳,
日経BP社, ISBN4-8222-8113-2より

既存のテストコード自動生成ツール

- 単体テストを対象

ハイブリッド検索，動的記号実行，テスト容易化変換を統合した検索ベースのアプローチを用いたツール[1]

EVOSUITE　TestFul　Seeker　eToc


Randoop

PARASOFT Jtest®　Pex

自動生成ツールを利用することで開発者の実装コストを削減し短期間でテストコードを作成できる

[1] G. Fraser, A. Arcuri, “EvoSuite: automatic test suite generation for object-oriented software”, Proceedings of the Symposium on the Foundations of Software Engineering (FSE), pp. 416-419, 2011.

自動生成されたテストコード

- EvoSuite[5]によって自動生成されたテストコード

テスト対象コード

```
public class CalcTriangleArea{
    public double calcTriangleArea(double x, double y) {
        double area = (x * y) / 2;
        return area;
    }
}
```

テストコード

```
@Test(timeout = 4000)
 public void test1()  throws Throwable  {
    CalcTriangleArea calcTriangleArea0 = new CalcTriangleArea();
    double double0 = calcTriangleArea0.calcTriangleArea(1.8, -2.5);
    assertEquals(-2.25, double0, 0.01);
}
```

- テストメソッド名，変数名が機械的
- テスト対象コードの意図に基づいていないので不具合を検知できない

[5] G. Fraser, A. Arcuri, “EvoSuite: automatic test suite generation for object-oriented software”, Proceedings of the Symposium on the Foundations of Software Engineering (FSE), pp. 416-419, 2011.

調査対象のJavaプロジェクトの概要1

- OSS上に存在する有名プロジェクト

プロジェクト名	言語	サイズ	バージョン	テストファイル数
Apache maven	java	35257LOC	3.6.0	137
Apache kafka	java	85591LOC	2.3.0	456
Apache kylin	java	603603LOC	2.6.3	214

調査対象のJavaプロジェクトの概要2

- OSS上に存在する有名プロジェクト

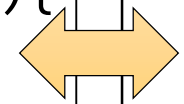
プロジェクト名	概要
Apache maven	Java用のプロジェクト管理ツール
Apache kafka	オープンソースのストーリーミング送受信処理基盤
Apache kylin	大規模なデータセットをサポートする分散型分析エンジン

タイプ1の類似コードペア

種類	意味
タイプ1	**レイアウト・空白・コメントの違いを除き完全に一致している**
タイプ2	タイプ1に加え変数名・型の違いを除き構文的に一致している
タイプ3	タイプ2に加え文が挿入・削除・変更されている

```
int sum(int[] data){
    int sum = 0;
    for(int i=0; i<data.length; i++){
        sum = sum + data[i];
    }
    return sum;
}
```

メソッドA



```
int sum(int[] data){
    int sum = 0;
    for(int i=0; i<data.length; i++){
        sum = sum + data[i];
    }
    return sum;
}
```

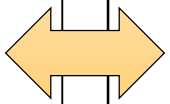
メソッドB

タイプ2の類似コードペア

種類	意味
タイプ1	レイアウト・空白・コメントの違いを除き完全に一致している
タイプ2	**タイプ1に加え変数名・型の違いを除き構文的に一致している**
タイプ3	タイプ2に加え文が挿入・削除・変更されている

```
int sum(int[] data){
    int sum = 0;
    for(int i=0; i<data.length; i++){
        sum = sum + data[i];
    }
    return sum;
}
```

メソッドA



```
double sum(double[] data){
    double sum = 0;
    for(double j=0;j<data.length;j++){
        sum = sum + data[j];
    }
    return sum;
}
```

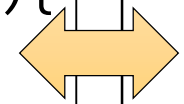
メソッドB

タイプ3の類似コードペア

種類	意味
タイプ1	レイアウト・空白・コメントの違いを除き完全に一致している
タイプ2	タイプ1に加え変数名・型の違いを除き構文的に一致している
タイプ3	**タイプ2に加え文が挿入・削除・変更されている**

```
int sum(int[] data){
    int sum = 0;
    for(int i=0; i<data.length; i++){
        sum = sum + data[i];
    }
    return sum;
}
```

メソッドA



```
int sum(int[] data){
    int sum = 0;
    for(int i=0; i<data.length; i++){
        sum += data[i];
    }
    return sum;
}
```

メソッドB

調査2：類似コードペアの類似度は，対応するテストメソッドの類似度にどのような関係があるか？

- 両方のコード片にテストコードが存在する類似コードペア62個に対応する158個のテストコードペアをタイプ別に分類

type2	type3	Not Similar	2to1	Total
105	15	33	5	158

type2：変数名・型の違いを除き構文的に一致している
type3：type2に加え文が挿入・削除・変更されている
Not Similar： type3以上の変更があり類似していない

多くの type2，type3のテストコードペアを検出

テストコード再利用パターン1

- type2 テストコードペア

テスト対象 : resolveModel(Parent parent)

```
public void testResolveParent ExistingWithoutRange() throws Exception
{
    final Parent parent = new Parent ();
    parent.setGroupId( "org.apache" );
    parent setArtifactId( "apache" );
    parent.setVersion( "1" );

    assertNotNull( this.newModelResolver().resolveModel( parent ) );
    assertEquals( "1", parent.getVersion() );
}
```

テスト対象 : resolveModel(Dependency dependency)

```
public void testResolveDependencyExistingWithoutRange() throws Exception
{
    final Dependency dependency = new Dependency();
    dependency.setGroupId( "org.apache" );
    dependency.setArtifactId( "apache" );
    dependency.setVersion( "1" );

    assertNotNull( this.newModelResolver().resolveModel(dependency ) );
    assertEquals( "1", dependency.getVersion() );
}
```

テスト対象のオブジェクトの統一的な変更により再利用可能

テストコード再利用パターン2

- type2 テストコードペア

テスト対象：get(String path)

```
public void testEmptyPathWithKey() throws Exception {
  ConfigData configData = configProvider.get("");
  assertTrue(configData.data().isEmpty());
  assertEquals(null, configData.ttl());
}
```

テスト対象：get(String path, Set<String> keys)

```
public void testEmptyPath() throws Exception {
  ConfigData configData = configProvider.get("", Collections.singleton("testKey"));
  assertTrue(configData.data().isEmpty());
  assertEquals(null, configData.ttl());
}
```

テスト対象のプロダクションメソッドに対応するように
引数の型，数を変更することで再利用可能

テストコード再利用パターン3

- type3 テストコードペア

テスト対象：parseToJobStatus(ExecutableState state)

```
@Test
public void testParseToJobStatusReturnsJobStatusPending() {
    ExecutableState executableState = ExecutableState.READY;
    JobStatusEnum jobStatusEnum
=JobInfoConverter.parseToJobStatus(executableState);

    assertEquals(1, jobStatusEnum.getCode());
    assertEquals(JobStatusEnum.PENDING, jobStatusEnum);
}
```

テスト対象：parseToJobStepStatus(ExecutableState state)

```
@Test
public void testParseToJobStepStatusReturnsJobStepStatusPending() {
    ExecutableState executableState = ExecutableState.READY;
    JobStepStatusEnum jobStepStatusEnum =
JobInfoConverter.parseToJobStepStatus (executableState);

+   assertTrue(jobStepStatusEnum.isRunable());
    assertEquals(1, jobStepStatusEnum.getCode());
    assertEquals(JobStepStatusEnum.PENDING, jobStepStatusEnum);
}
```

テスト対象のオブジェクトの統一的な変更
＋ 文の追加・削除によって再利用可能

振る舞いに着目した類似コードペアの分類

- Ishioら[6]によって提案されているメソッド呼び出しの差異に基づく類似コードの分類手法を用いる
  1. メソッドの振る舞いを説明する内容としてふさわしいプログラム文(メソッドサマリ)を抽出する
  2. 2つのプログラムのメソッドサマリが同一かどうかで振る舞いを判定する

```
1: void returnPressed() {
2:   Shell s = getShell();
3:   String input = s.getEnteredText();
4:   history.addElement(input);              // Summary
5:   String result = evaluate(input);       // Summary
6:   s.append(result);                       // Summary
7: }
```

```
1: void returnPressed() {
2:   Logger.debug("returnPressed");
3:   Shell s = getShell();                  // Clone
4:   String input = s.getInputText();       // Clone
5:   history.addElement(input);             // Clone, Summary
6:   String result = evaluate(input);       // Clone, Summary
7:   s.append(result);                       // Clone, Summary
8: }                                         // Clone
```

[5]石尾 隆, 伊達 浩典, 井上 克郎. メソッド呼び出しの差異に基づくコードクローンの分類手法. 情報処理学会論文誌, Vol. 56, No. 6, pp. 1471–1480, 2015.