

2020 修論発表会

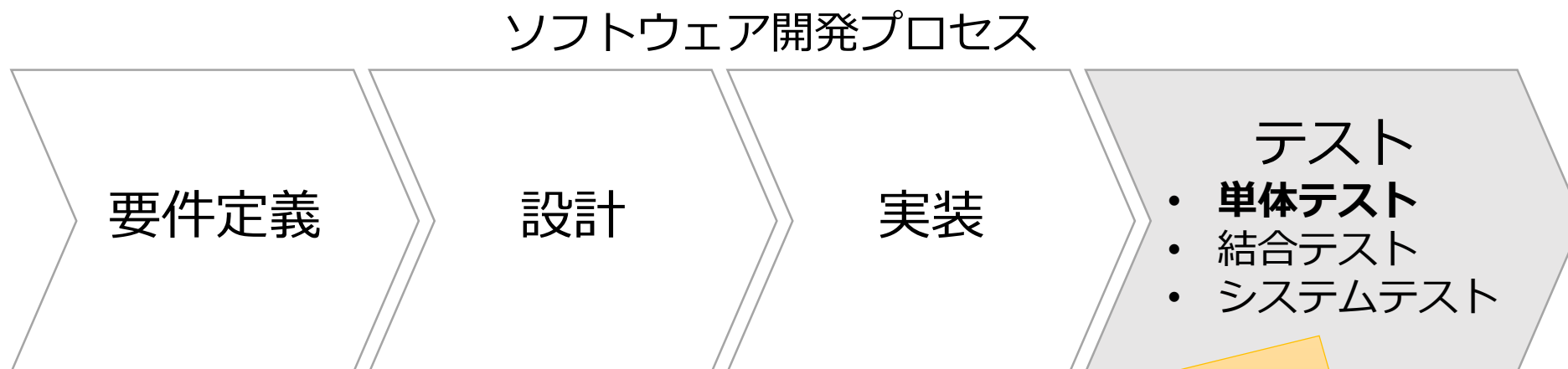
# ソースコードの類似性に基づいた テストコード自動推薦ツールSuiteRec

---

1811098 倉地亮介

# ソフトウェアテスト

- ソフトウェア開発におけるソフトウェアの品質を確かめる工程



開発全体のコストの内、30%~50%を占めると言われている[1]

[1] M. Ellims, J. Bridges, and D. C. Ince. The economics of unit testing. *Empirical Software Engineering*, 11(1):5–31, 2006.

# テストコード自動生成ツール

- テスト工程を支援するために、これまでに様々な自動生成ツールが提案されてきた

EVASUITE

TestFul

Seeker

 Randoop

 PARASOFT®  
*Jtest*®

Pex Grafter

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

# 自動生成ツールにおける課題

自動生成されたテストコードは、保守作業を困難にする[2]

- 対象コードの作成経緯や意図に基づいて生成されていない
- テストコードの保守に悪影響を与えるテストスメルが多い



テスト失敗の原因を特定するのが難しい

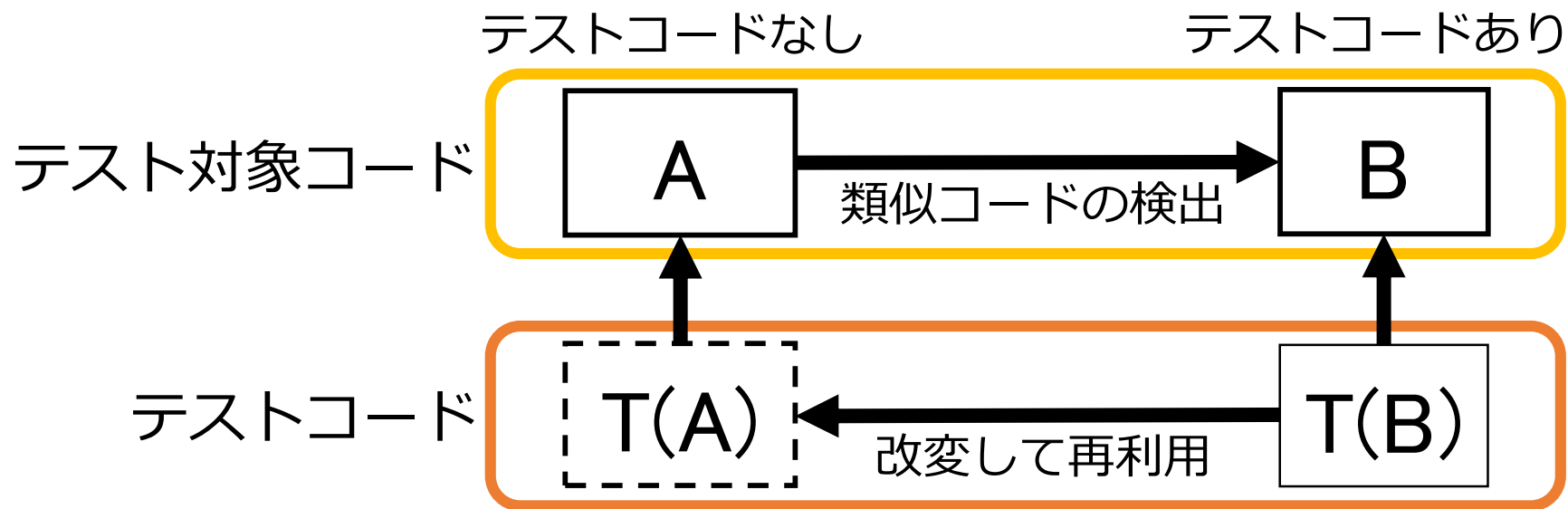
理解しやすく良質なテストコードを作成する必要がある

[2] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser. How do automatically generated unit tests influence software maintenance? In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 250–261, 2018..

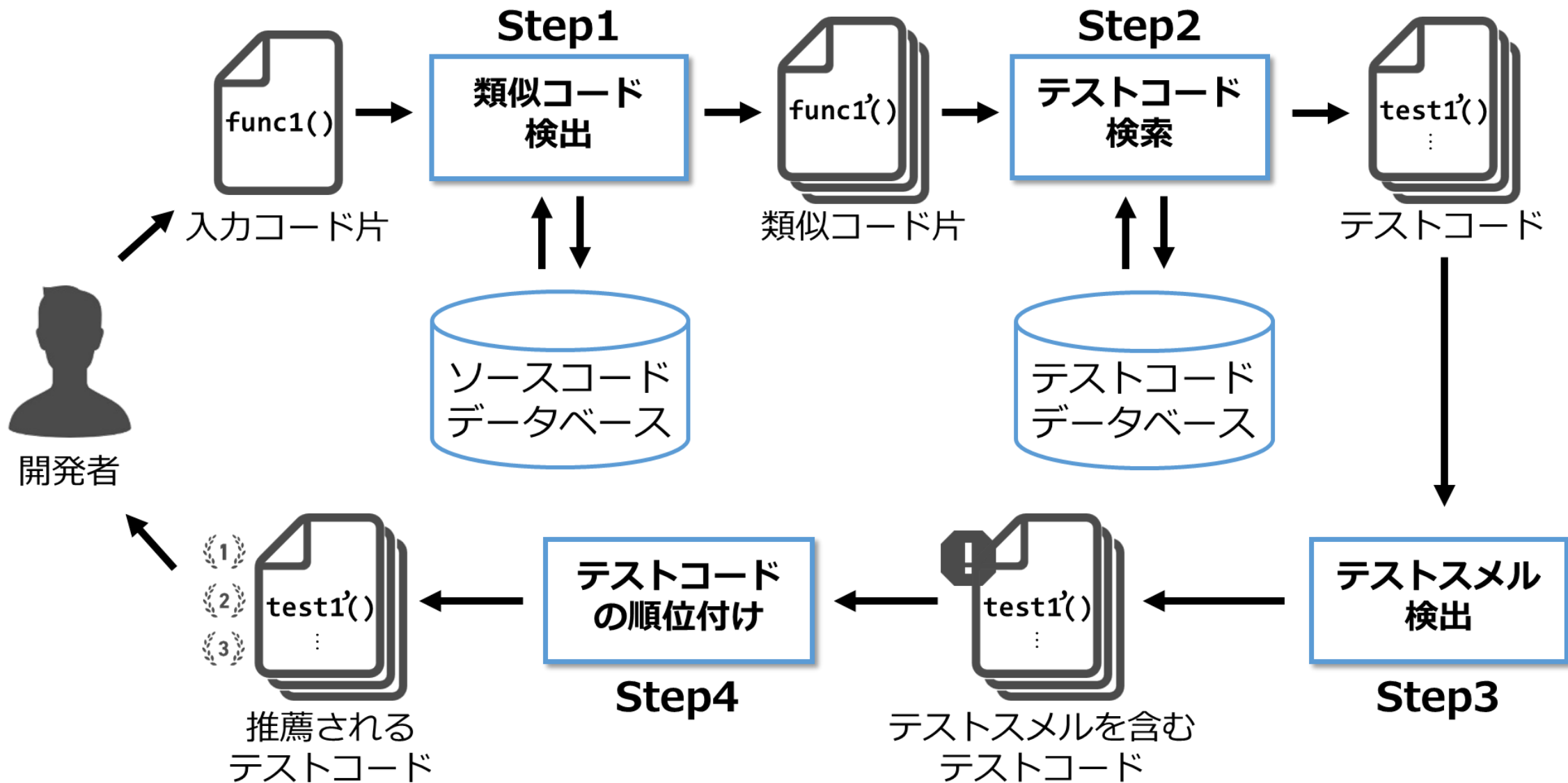
## SuiteRec: OSSに存在する高品質のテストコードを推薦

- 命名規則に従った可読性の高いテストコードを利用できる
- テストスメルの数が少なく品質の高いテストコードを推薦

## アイディア: 類似するコード間でテストコードを再利用



# SuiteRecの概要

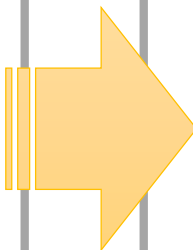


# Step1: 類似コード片の検出

- 類似コード検出ツール: NiCad[3]
  - ソース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することで類似コード片を検出
  - 高精度・高再現率で類似コード片を検出可能

```
public int countPrice(int item[]){  
    int totalprice = 0;  
  
    for(int i=0; i<item.length; i++){  
        totalprice += item[i];  
    }  
    return totalprice;  
}
```

入力コード片



```
public int calcPrice(int ...cost){  
    int totalcost = 0;  
    int num = cost.length;  
    for(int i=0; i < num; i++){  
        totalcost += cost[i];  
    }  
    return totalcost;  
}
```

類似コード片

[3] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 172–181, 2008.

## Step2: テストコードの検索

- テストコードと対象コードの対応付け
  1. 命名規則によるテストクラスと対象クラスの対応付け
  2. テストコード内のメソッド呼び出しを確認
  3. メソッド名の比較によるテストメソッドと対象メソッドの対応付け

### <CalcPriceクラス>

```
public int calcPrice(int ...cost){  
    int totalcost= 0;  
  
    for(int i=0; i < cost.length; i++){  
        totalcost += cost[i];  
    }  
  
    return totalcost;  
}
```

類似コード片(テスト対象)

### <CalcPriceTestクラス>

```
@Test  
public void testCalcPrice() throws Throwable{  
    CalcPrice sut = new CalcPrice();  
    int[] item1 = new int[0];  
    int[] item2 = new int[100];  
    assertEquals(0, sut.calcPrice(item1));  
    assertEquals(100, sut.calcPrice(item2));  
    assertNotNull(sut.calcPrice(item1));  
}
```

テストコード




## Step3: テストスメルの検出

- テストコードの良くない実装を表す指標
  - Deursenら[4]は、11種類のテストスメルを提唱した(現在19種類)

### テストスメルの例: **Assertion Roulette**

```
@Test
public void testCalcPrice() throws Throwable{
    CalcPrice sut = new CalcPrice();
    int[] item1 = new int[0];
    int[] item2 = new int[100];
    assertEquals(0, sut.calcPrice(item1));
    assertEquals(100, sut.calcPrice(item2));
    assertNotNull(sut.calcPrice(item1));
}
```



複数のassert文が存在するとテストメソッドが失敗した時、原因を特定するのが困難

テストスメル検出ツール: tsDetect[5]

- ✓ 19種類のテストスメルを検出可能
- ✓ 各テストスメルを高精度で検出可能

[4] A. Deursen, L. M. F. Moonen, A. Bergh, and G. Kok. Refactoring test code. Technical report, 2001.

[5] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the International Conference on Computer Science and Software Engineering (CASCON)*, pages 193–202, 2019.

## Step4: 推薦されるテストコードの順位付け

- 推薦されるテストコードを以下の2つの要素を基に順位付ける



類似度を優先として並び替え、類似度が同じ場合テストスメルの数で順位付ける

# SuiteRecのインターフェース

Clone Pairs 1 : 71.4% **3**

<b>1</b> Input Code		<b>2</b> Similarity Code	
<pre>public String fizzBuzz(int number) {     if (number &lt;= 0) {         return "Not Natural Number";     }     if (number % 15 == 0) {         return "fizzbuzz";     } else if (number % 3 == 0) {         return "fizz";     } else if (number % 5 == 0) {         return "buzz";     } else {         return Integer.toString(number);     } }</pre>		<pre>public String fizzBuzz(int number) {     if (number &lt;= 0) {         throw new RuntimeException();     }     if (number % 15 == 0) {         return "FizzBuzz";     } else if (number % 3 == 0) {         return "Fizz";     } else if (number % 5 == 0) {         return "Buzz";     } else {         return Integer.toString(number);     } }</pre>	

Test Suite					
Assertion Roulette	Conditional Test Logic	Default Test	Eager Test	<b>4</b> Exception Handling	Mystery Guest
Lines 8 - 13 of tcs-expt-similar/src/test/java/jp/tcs/expt02/FizzBuzz1Test.java					
<pre>@Test(expected=RuntimeException.class) public void test1() {     FizzBuzz1 fb = new FizzBuzz1();     fb.fizzBuzz(-1); }</pre>					
Lines 14 - 21 of tcs-expt-similar/src/test/java/jp/tcs/expt02/FizzBuzz1Test.java					
<pre>@Test public void returnBuzz_input5() throws Throwable {     FizzBuzz1 fizzBuzz = new FizzBuzz1();     String actual = fizzBuzz.fizzBuzz(5);     String expected = "Buzz";     assertEquals(expected, actual); }</pre>					

**5**

- 1** 入力コード
- 2** 類似コード
- 3** 類似度(UPI)
- 4** テストスメル
- 5** テストコード

## SuiteRecの有用性を定量的・定性的に評価

### 評価実験1: テストコードの作成支援に関する実験

- 被験者が作成したテストコードのカバレッジ、作成時間、テストスメルを比較して評価した

### 評価実験2: 推薦されるテストコードの順位付けに関する実験

- アンケート調査を実施し、開発者が参考にしたいテストコードを上位に推薦できるかを評価した

※本発表では、時間の都合上紹介されません

# 評価実験1

- 実験概要

- 情報科学を専攻する修士課程の学生10人に3つのタスクのテストコードを作成してもらう

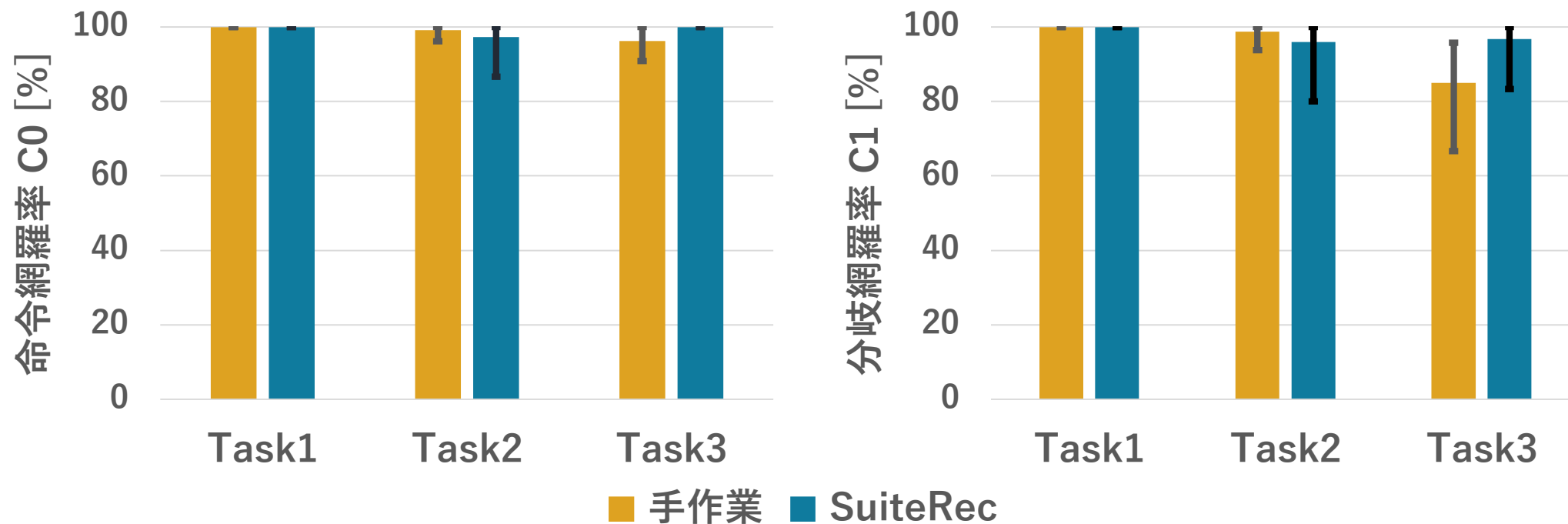
	Task1	Task2	Task3
プロダクションコードの概要	入力値に応じて, "fizz", "buzz", "fizzbuzz"を返すプログラム	第1引数に応じて, 残り3引数の計算方法(最大値, 中央値, 最小値)を変更し, 計算結果を返すプログラム	2つのスコア(0~100点)を入力し, 条件に従って試験の結果を判定するプログラム
条件分岐の数	8	16	24

- SuiteRecを使用した場合と手作業の場合で、被験者が作成したテストコードを比較し評価する
- 実験後にテストコード作成タスクに関するアンケートを実施した

# リサーチクエスチョン(RQ)

- RQ1.** SuiteRecは、高いカバレッジを持つテストコードの作成を支援できるか？
- RQ2.** SuiteRecは、テストコード作成時間を削減できるか？
- RQ3.** SuiteRecは、テストスメルの数が少ないテストコードの作成を支援できるか？
- RQ4.** SuiteRecの利用は、開発者のテストコード作成タスクの認識にどう影響するか？

## RQ1. SuiteRecは、高いカバレッジを持つ テストコードの作成を支援できるか？



- カバレッジに大きな差はない
- 条件分岐が多く複雑なプログラムのカバレッジ(C1)に多少の差が見られた

## RQ2. SuiteRecは、テストコードの作成時間を削減できるか？



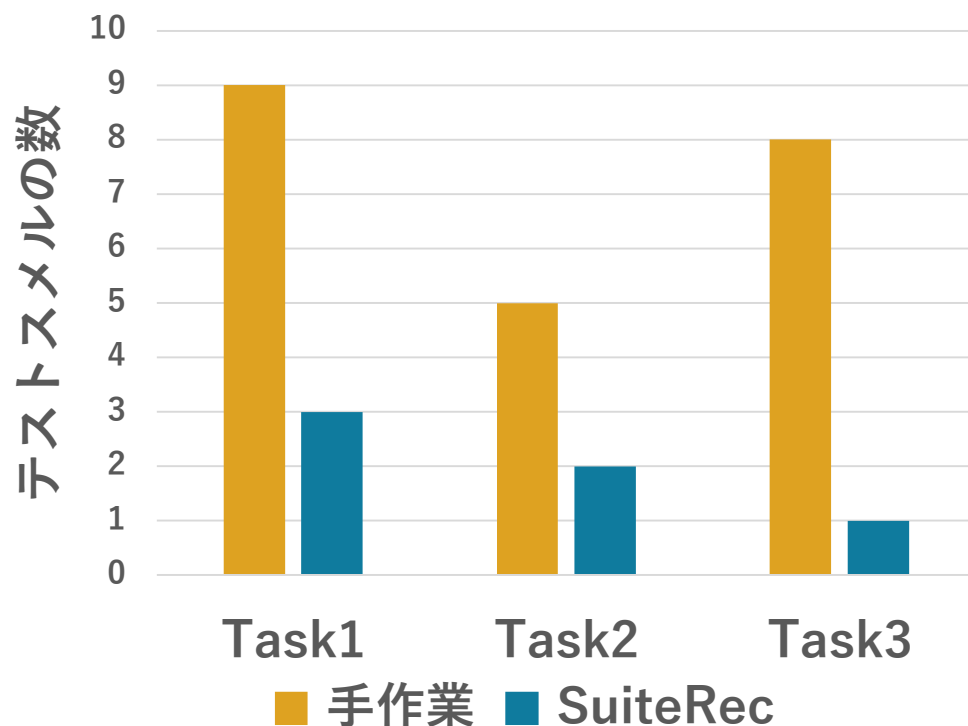
### <SuiteRecを利用した場合>

- タスク1と3は、タスク完了までの時間が手作業の場合と比べて長い
  - 考察: 推薦された複数のテストコードを理解し、変更する必要がある
- タスク2は、タスク完了までの時間が手作業の場合と比べて短い
  - 手作業の場合、余分なテスト項目を作成し、時間を費やした可能性がある

SuiteRecの利用によって、開発者はテストコード作成に多くの時間を費やす



## RQ3. SuiteRecは、テストスメルの数が少ない テストコードの作成を支援できるか？



### <SuiteRecを利用した場合>

- すべてのタスクにおいて、検出されたテストスメルの数が少ない

### <手作業の場合>

- 多く検出されたテストスメル
  - Assertion Roulette
  - Default Test
  - Eager Test
- 既存研究でも、これらのテストスメルがプロジェクト内で多く検出されたと報告

SuiteRecの利用は、テストスメルの数が少なく品質の高い  
テストコードの作成を支援できる

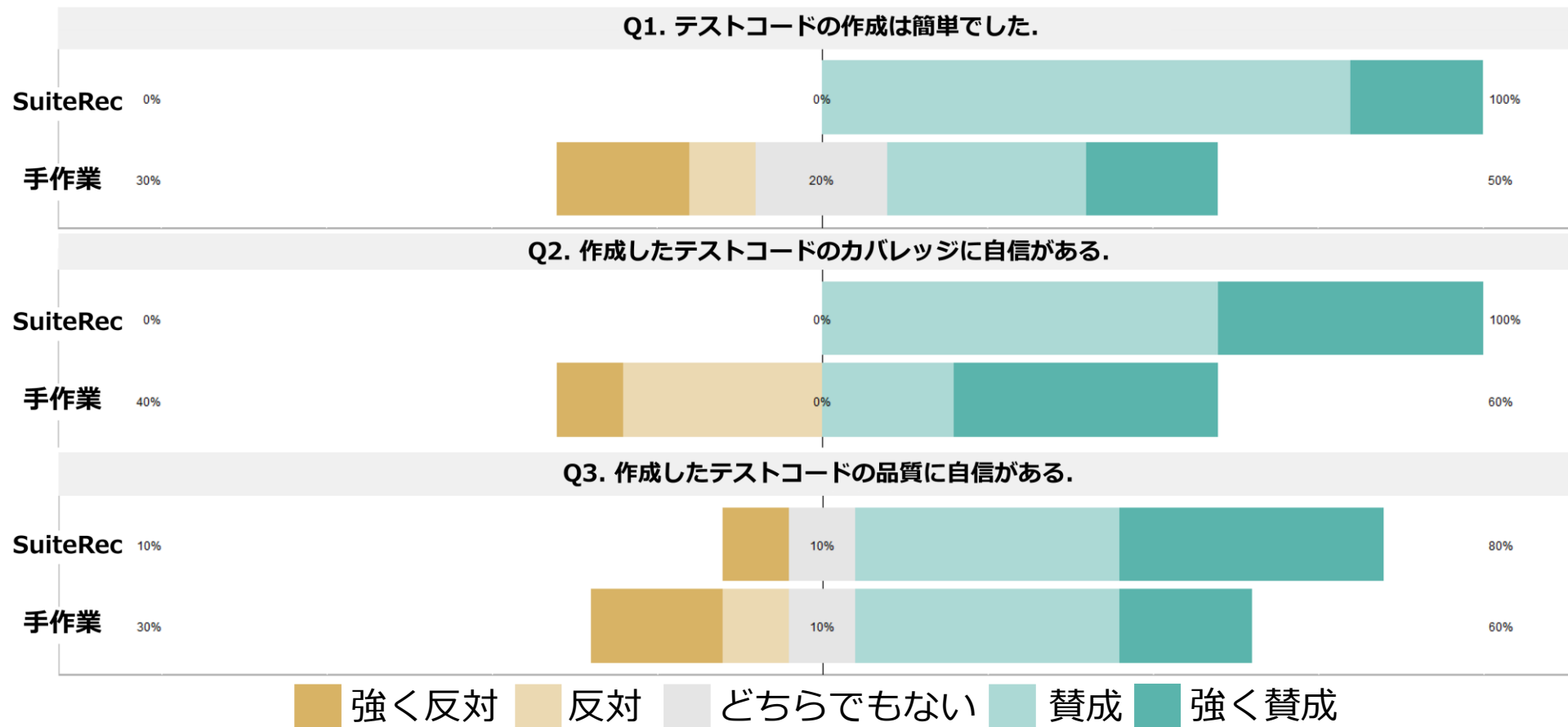
## RQ4. SuiteRecの利用は、開発者のテストコード作成タスクの認識にどう影響するか？

- 実験後の被験者に、実験タスクに関するアンケートを実施した

	項目
Q1-a	テストコードの作成は簡単でした(ツールあり)
Q1-b	テストコードの作成は簡単でした(ツールなし)
Q2-a	作成したテストコードのカバレッジに自信がある(ツールあり)
Q2-b	作成したテストコードのカバレッジに自信がある(ツールなし)
Q3-a	作成したテストコードの品質に自信がある(ツールあり)
Q3-b	作成したテストコードの品質に自信がある(ツールなし)

※5段階評価：強く反対・反対・どちらでもない・賛成・強く賛成

## RQ4. SuiteRecの利用は、開発者のテストコード作成タスクの認識にどう影響するか？



SuiteRecを利用すると、開発者はテスト作成タスクを容易だと認識し、作成したテストコードに自信が持てる

# まとめ・今後の課題

## ・まとめ

- ・ 類似コード検出技術を用いて、既存の高品質のテストコードを推薦するツールを提案
- ・ 提案ツールを利用することで、品質の高いテストコードの作成を支援できることを確認

## ・今後の課題

- ・ より実用的な利用に備えてツールを改善する必要がある
- ・ 被験者数を増やした更なる実験が必要である

# 補足資料

---

# 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%	—

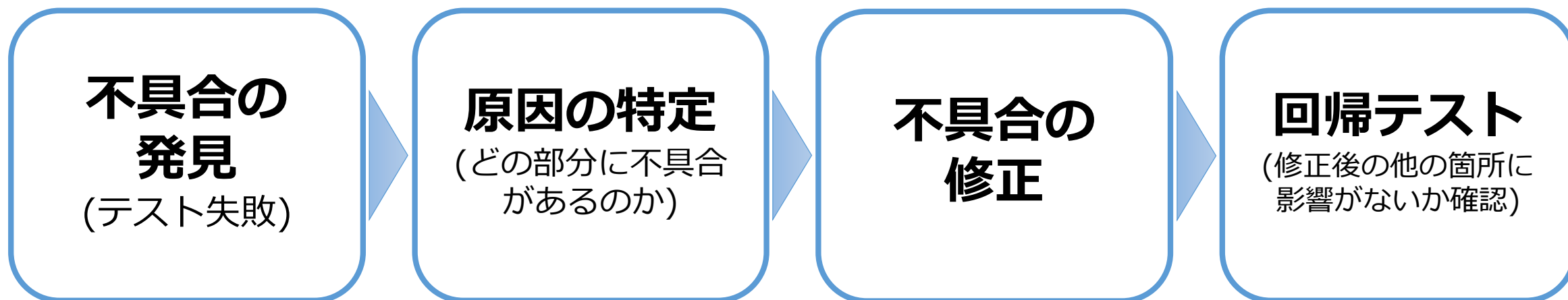
「基本から学ぶソフトウェアテスト」 Cem Kaner, Jack Falk, Hung Quoc Nguyen著, テスト技術者交流会訳, 日経BP社, ISBN4-8222-8113-2

# テストの種類

テストの種類	説明
単体テスト	プログラムを構成する比較的小さな単位の部品(関数)が、個々の機能を正しく果たしているかどうかを検証するテスト
結合テスト	個々の機能を果たすプログラムの部品(単体)を組み合わせて、仕様通り正しく動作するかを検証するテスト
システムテスト	個々の機能や仕組みを総合した全体像のシステムとして、仕様通り正しく動作するかを検証するテスト

# ソフトウェアの保守活動

## ・ソフトウェアの保守シナリオ



品質の低いテストコードは、**原因の特定と不具合の修正**が困難になり保守コストがかかる

[2] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser. How do automatically generated unit tests influence software maintenance? In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 250–261, 2018..



# 良いテストコードとは？

- 単に不具合を検出するだけでなく、開発者にどのように修正すればテストが通るのかを提示してくれる
- 対象コードの作成経緯や意図された動作から脱線することに敏感であり、結びつきが深い
- プロダクションコードの変更に対してテストコードの変更も容易に適用できるように、理解がしやすい

S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser. How do automatically generated unit tests influence software maintenance? In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 250–261, 2018..

# 自動生成されたテストコードの例

- EvoSuite[10]によって自動生成されたテストコード



## Default Test

メソッド名が初期状態



## Exception Handling

意図が分からない例外処理

```
public void test8 throws Throwable {  
    Document document0 = new Document("", "");  
    assertNotNull(document0);  
  
    document0.procText.add((Character) 's');  
    String string0 = document0.stringify();  
    assertEquals("s", document0.stringify());  
    assertNotNull(string0);  
    assertEquals("s", string0);  
}
```



## Assertion Roulette

複数のassert文が存在する

[10] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pages 416–419, 2011.

# Assertion Roulette (アサーション・ルーレット)

- **テストメソッド内に複数のassert文が存在する**

- 各assert文は異なる条件をテストするが、開発者へ各assert文のエラーメッセージは提供されない
- assert文の1つが失敗した場合、失敗の原因を特定するが困難である

```
@MediumTest
public void testCloneNonBareRepoFromLocalTestServer() throws Exception {
    Clone cloneOp = new Clone(false, integrationGitServerURIFor("small-
repo.early.git"), helper().newFolder());
```

```
    Repository repo = executeAndWaitFor(cloneOp);
```

```
    assertThat(repo, hasGitObject("ba1f63e4430bfff267d112b1e8afc1d628..."));
```

```
    File readmeFile = new File(repo.getWorkTree(), "README");
```

```
    assertThat(readmeFile, exists());
```

```
    assertThat(readmeFile, ofLength(12));
```

```
}
```

複数のassert文が存在する

# Conditional Test Logic

- テストメソッド内に複数の制御文が含まれている
  - テストの成功・失敗は制御フロー内にあるassert文に基づくため結果を予測できない

```
@Test
public void testSpinner() {
    for (Map.Entry entry : sourcesMap.entrySet()) {

        String id = entry.getKey();
        Object resultObject = resultsMap.get(id);
        if (resultObject instanceof EventsModel) {
            EventsModel result = (EventsModel) resultObject;
            if (result.testSpinner.runTest) {
                System.out.println("Testing " + id + " (testSpinner)");
                //System.out.println(result);
                AnswerObject answer = new AnswerObject(entry.getValue(), "", new CookieManager(), "");
                EventsScraper scraper = new EventsScraper(RuntimeEnvironment.application, answer);
                SpinnerAdapter spinnerAdapter = scraper.spinnerAdapter();
                assertEquals(spinnerAdapter.getCount(), result.testSpinner.data.size());
                for (int i = 0; i < spinnerAdapter.getCount(); i++) {
                    assertEquals(spinnerAdapter.getItem(i), result.testSpinner.data.get(i));
                }
            }
        }
    }
}
```

複数の制御文が存在する

# Mystery Guest (ミステリー・ゲスト)

## ・テストメソッド内で、外部リソースを利用する

- ・テストメソッド内だけでなく外部ファイルなど、外部リソースを使用すると見えない依存関係が生じる
- ・誰かが、外部ファイルを削除するとテストが失敗してしまう

```
public void testPersistence() throws Exception {  
    File tempFile = File.createTempFile("systemstate-", ".txt");  
    try {  
        SystemState a = new SystemState(then, 27, false, bootTimestamp);  
        a.addInstalledApp("a.b.c", "ABC", "1.2.3");  
  
        a.writeToFile(tempFile);  
        SystemState b = SystemState.readFromFile(tempFile);  
  
        assertEquals(a, b);  
    } finally {  
        //noinspection ConstantConditions  
        if (tempFile != null) {  
            //noinspection ResultOfMethodCallIgnored  
            tempFile.delete();  
        }  
    }  
}
```

外部にファイルを生成し、  
テストプロセスで利用している

# Eager Test (イーガー・テスト)

- **テスト対象クラスの複数のメソッドを呼び出す**

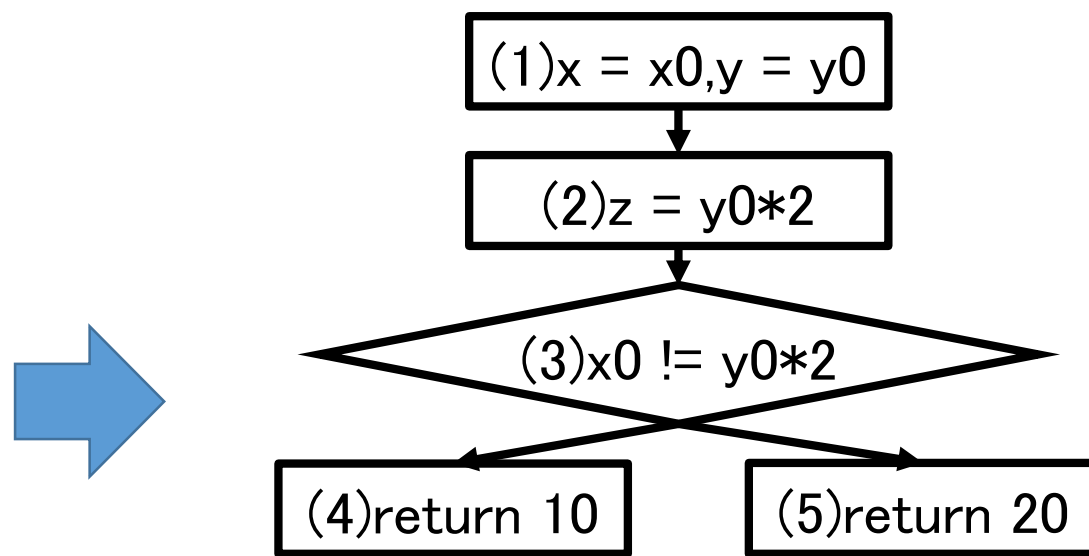
- 1つのテストメソッドで複数のメソッドをテストすると、他の開発者は何をテストしているかについて混乱が生じる

```
@Test
public void NmeaSentence_GPGSA_ReadValidValues(){
    NmeaSentence nmeaSentence = new
NmeaSentence("$GPGSA,A,3,04,05,,09,12,,,24,,,2.5,1.3,2.1*39");
    assertThat("GPGSA - read PDOP", nmeaSentence.getLatestPdop(), is("2.5"));
    assertThat("GPGSA - read HDOP", nmeaSentence.getLatestHdop(), is("1.3"));
    assertThat("GPGSA - read VDOP", nmeaSentence.getLatestVdop(), is("2.1"));
}
```

テスト対象コードのメソッド  
が複数回呼び出される

- 1.対象コードを静的解析し，プログラムを記号値で表現する
- 2.コード上のそれぞれのパスに対応する条件を抽出
- 3.パスごとにパスを通るような条件を集め，条件を満たす具体値を生成

```
public int
  testme(int x,int y)
{
  int z = y*2;
  if(x != z){
    return 10;
  }else{
    return 20;
  }
}
```



パス番号	パス条件	パス
1	$x0 \neq y0 * 2$	(1), (2), (3), (4)
2	$!(x0 \neq y0 * 2)$	(1), (2), (3), (5)

# ソースコードデータベース

## Github上に存在する3,205個のOSSプロジェクト のプロダクションコード

既存のコード検索エンジン[7]で利用されたデータセットの中から、以下の条件を満たすプロジェクトを選択

- テストフォルダが存在する
- JUnitのテストフレームワークを採用している

[7] K. Kim, D. Kim, T. F. Bissyand ´ e, E. Choi, L. Li, J. Klein, and Y. Le Traon. Facoy - a code-to-code search engine. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 946–957, 2018.



# テストコードデータベース

## ソースコードデータベースに格納されている プロダクションコードに対応するテストコード

再利用対象のテストコードとして相応しくない、以下の  
テストスメルを含むテストコードはTDBから除去される

- *Empty Test*
- *Ignored Test*
- *Redundant Assertion*
- *Unknown Test*

# 推薦プロセスの高速化

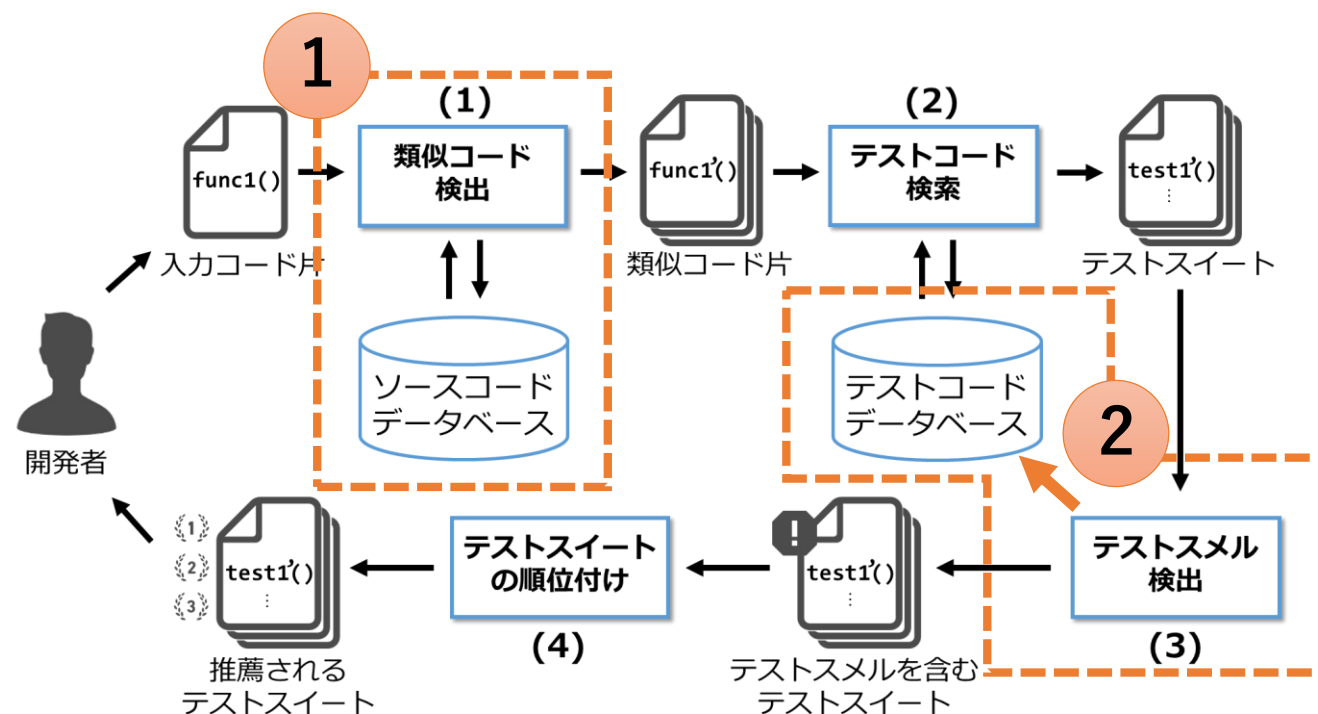
- 類似コード検索を複数並列に実行(Step1)
- データベースに必要なデータを事前に格納(Step3)

## Step1

- 前処理としてプロジェクトのサイズを調整して格納
- 類似コード検索処理を複数並列に実行

## Step3

- 事前にテストスメルを検出
- テストスメルの情報をテストコードに紐づけて格納



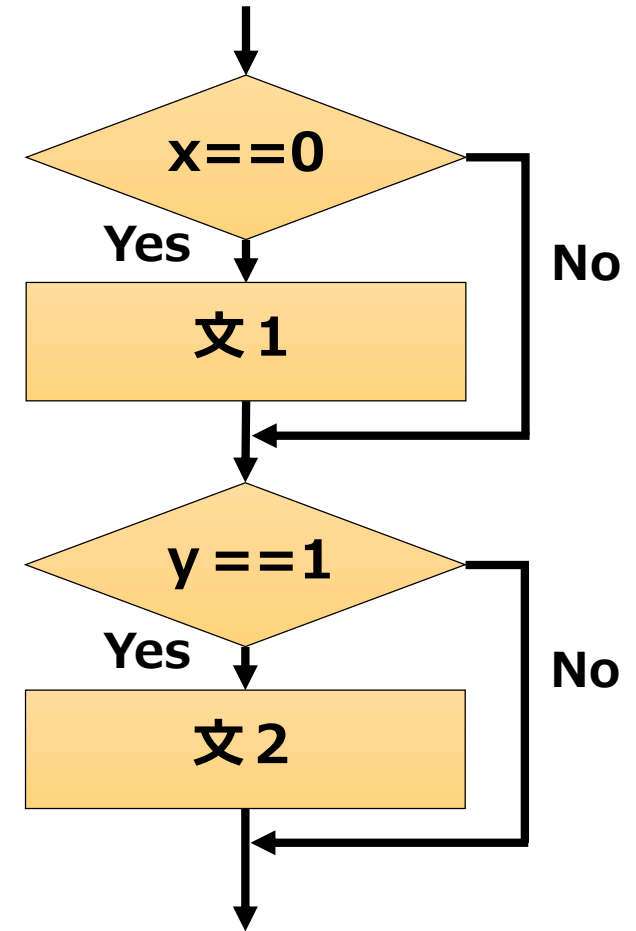
# カバレッジの種類

## ● 命令網羅 C0

- 全ての命令を最低一回実行
- フローチャートの全節点通過
  - ✓ (x=0, y=1) を入力

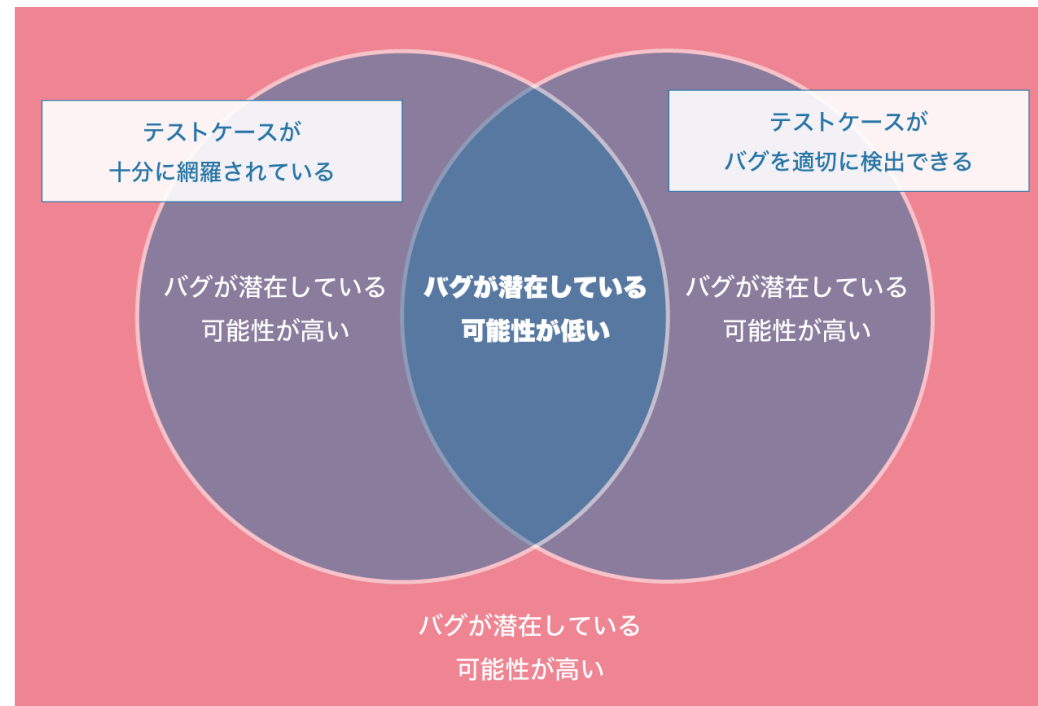
## ● 分岐網羅 C1

- 全ての条件分岐について、then、else いずれも最低一回以上実行
- フローチャートの全辺通過
  - ✓ (x=0, y=1) と (x=1, y=2) を入力

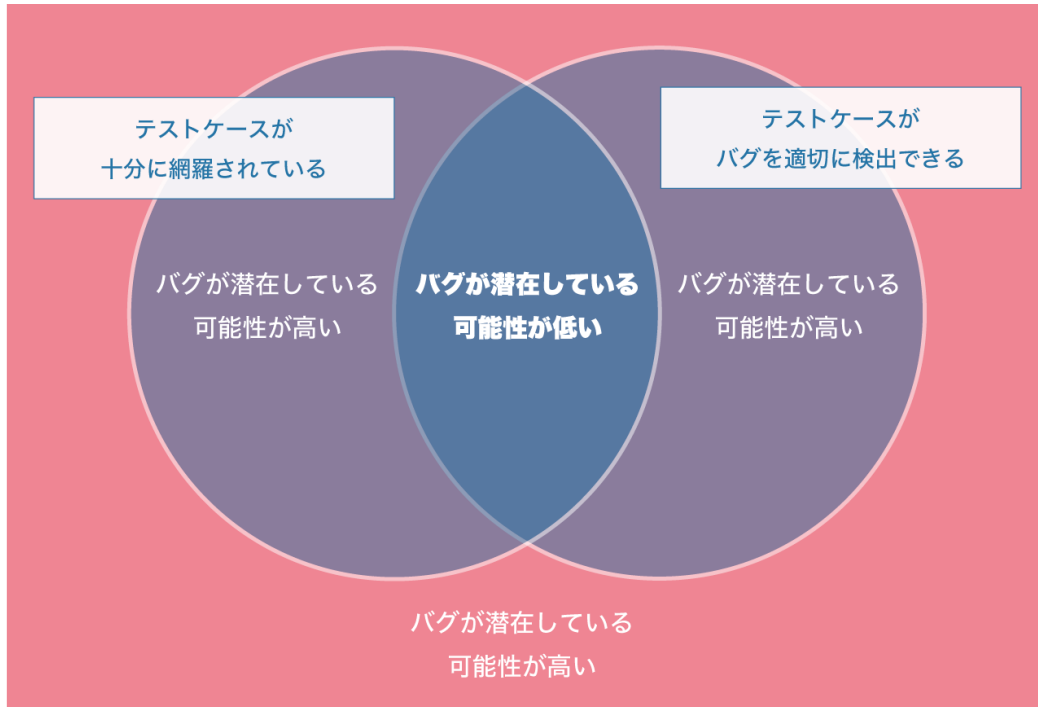


# カバレッジの議論

- カバレッジ100%を追究しても品質は高いとは言えない
  - 「カバレッジが高い」 = 「テストケースが十分に網羅されている」
  - 「ソースコードの品質が高い」 = 「バグが潜伏している可能性が低い」
  - **「テストケースの品質が高い」 = 「テストケースがバグを適切に検出できる」**



# カバレッジの議論



「テストケースが十分に網羅されている」かつ  
「テストケースがバグを適切に検出できる」=>  
「バグが潜在している可能性が低い」は「**真**」

「テストケースが十分に網羅されている」=>  
「バグが潜在している可能性が低い」は「**偽**」

## 反例

「テストケースが十分に網羅されている」かつ  
「テストケースがバグを適切に検出できない」  
=> 「バグが潜在している可能性が高い」

## RQ2. SuiteRecは、テストコードの作成時間を削減できるか？

	Task1		Task2		Task3	
	手作業	SuiteRec	手作業	SuiteRec	手作業	SuiteRec
カバレッジ (C0)	100	100	99.2	97.3	96.2	100
カバレッジ (C1)	100	100	98.8	96.0	85.0	96.7
テスト項目の数(平均)	6	6	<b>24</b>	11	<b>16</b>	12

タスク2は、カバレッジに差はないものの手作業の場合、テスト項目の数が多く無駄なテストコードを作成している

## カバレッジの目標値は何%にするべきなのか？

提唱者	カバレッジの種類	目標値
Googleの開発チーム	命令網羅率 C0	85%+
Martin Fowle氏	不明	85% - 99%

- 実験概要

- オンライン上でアンケートを実施

- 被験者にSuiteRecによって1位から10位まで順位付けられたテストコードを読んでもらい、推薦されたテストコードを参考にしたいかどうかを選択してもらう

- 評価方法

- 検索エンジンなどの評価で用いられる代表的な評価指標を利用
    - *Mean Reciprocal Rank (MRR)*
    - *Mean Average Precision (MAP)*
    - *Mean Precision@N*



## RQ5. SuiteRecは、開発者が参考にしたいテストコードを上位に推薦できるか？

MAPとMRRの評価結果

	MAP	MRR
タスクA	84.1%	83.3%
タスクB	98.3%	100%
合計	91.2%	94.1%

MP@Nの評価結果

Precision-topN	top1	top2	top5	top10
タスクA	73.3%	82.2%	85.3%	45.3%
タスクB	100%	97.8%	93.3%	53.3%
合計	86.7%	90.0%	89.3%	49.3%

- RQ1から、単純な構造のプログラムのテストコードを作成する場合、**SuiteRecの利用の有無でカバレッジに差がない**



- RQ2から、**SuiteRecを利用しない方が、開発時間を節約できる**
- **複雑な構造のプログラムのテストコード**を作成する場合、SuiteRecを使用するとカバレッジ(C1)を向上することができる

- 類似コード間のテスト再利用
  - Zhangら[8]は、クローンペア間でコードを移植を行い、移植前と移植後のテスト結果を比較しその情報を基にテストを再利用するツールGrafterを提案した
  - Sohaら[9]は、開発者が詳細な再利用計画を決めることで、コード片を再利用する際に、テストスイートの関連部分を半自動で再利用および変換を行うツールSkipperを提案した

SuiteRecは、既存ツールと2つの視点で異なる

- OSS上からテストコードを検出することができる
- クローンペア間のテスト再利用計画は開発者に委ねていること

[8] T. Zhang and M. Kim. Automated transplantation and differential testing for clones. *Proc. of ICSE*, pages 665–676, 2017.

[9] S. Makady and R. Walker. Validating pragmatic reuse tasks by leveraging existing test suites. *Software: Practice and Experience*, 43:1039–1070, 2013.

# 実験タスクの詳細概要

	Task1	Task2	Task3
言語	Java	Java	Java
プロダクション コードの概要	入力値に応じて, "fizz", "buzz", "fizzbuzz"を返す プログラム	第1引数に応じて, 残り3引 数の計算方法(最大値, 中央 値, 最小値)を変更し, 計算 結果を返すプログラム	2つのスコア(0~100点)を 入力し, 条件に従って試験の 結果を判定するプログラム
言語	Java	Java	Java
LOC	17	<b>30</b>	18
条件分岐の数	8	16	<b>24</b>
変数の数	1	<b>4</b>	2
引数の数	1	<b>4</b>	2

# 実験タスク1

```
public class Experiment01 {  
  
    public String fizzBuzz(int number) {  
        if (number <= 0) {  
            return "Not Natural Number";  
        }  
        if (number % 15 == 0) {  
            return "fizzbuzz";  
        } else if (number % 3 == 0) {  
            return "fizz";  
        } else if (number % 5 == 0) {  
            return "buzz";  
        } else {  
            return Integer.toString(number);  
        }  
    }  
}
```

言語	Java
LOC	17
条件分岐の数	8
変数の数	1
引数の数	1

# 実験タスク2

```
public class Experiment02 {  
  
    public int calcMediumMinMax(String select,int a, int b, int c){  
        if (select == "Medium"){  
            if (a < b) {  
                if (b < c) {  
                    return b;  
                } else if (a < c) {  
                    return c;  
                } else {  
                    return a;  
                }  
            } else {  
                if (a < c) {  
                    return a;  
                } else if (b < c){  
                    return c;  
                } else {  
                    return b;  
                }  
            }  
        } else if (select == "max"){  
            return Math.max(a, Math.max(b,c));  
        } else if (select == "min"){  
            return Math.min(a, Math.min(b,c));  
        } else {  
            return -1;  
        }  
    }  
}
```

言語	Java
LOC	30
条件分岐の数	16
変数の数	4
引数の数	4

# 実験タスク3

```
public class Experiment03 {  
    public String returnResult(int score1, int score2){  
        if(( score1 < 0 || score2 < 0 ) || ( score1 > 100 || score2 > 100 )){  
            return "Invalid Input";  
        }else if(score1 == 0 || score2 == 0){  
            return "failure";  
        }else if(score1 >= 60 && score2 >= 60){  
            return "pass";  
        }else if(( score1 + score2 ) >= 130){  
            return "pass";  
        }else if(( score1 + score2 ) >= 100 && ( score1 >= 90 || score2 >= 90 )){  
            return "pass";  
        }else {  
            return "failure";  
        }  
    }  
}
```

言語	Java
LOC	18
条件分岐の数	24
変数の数	2
引数の数	2

# アンケートの自由記述欄

## 自由記述欄の回答

「テスト項目を推薦されたテストコードを見て、見落としていたテスト項目に気づくことができた」

「提示されたテストスメルを理解し、それを無くすようにリファクタリングしテストコードを作成できた」

「テストスメルの存在は意識できたが、具体的にどう修正して無くすことができるのか分からなかった」