テストコードの分岐構造と カバレッジ間における関係の実証的分析

2025/2/13

肥後研究室 山中綾華

研究の背景|テストコードとカバレッジ

テストコード

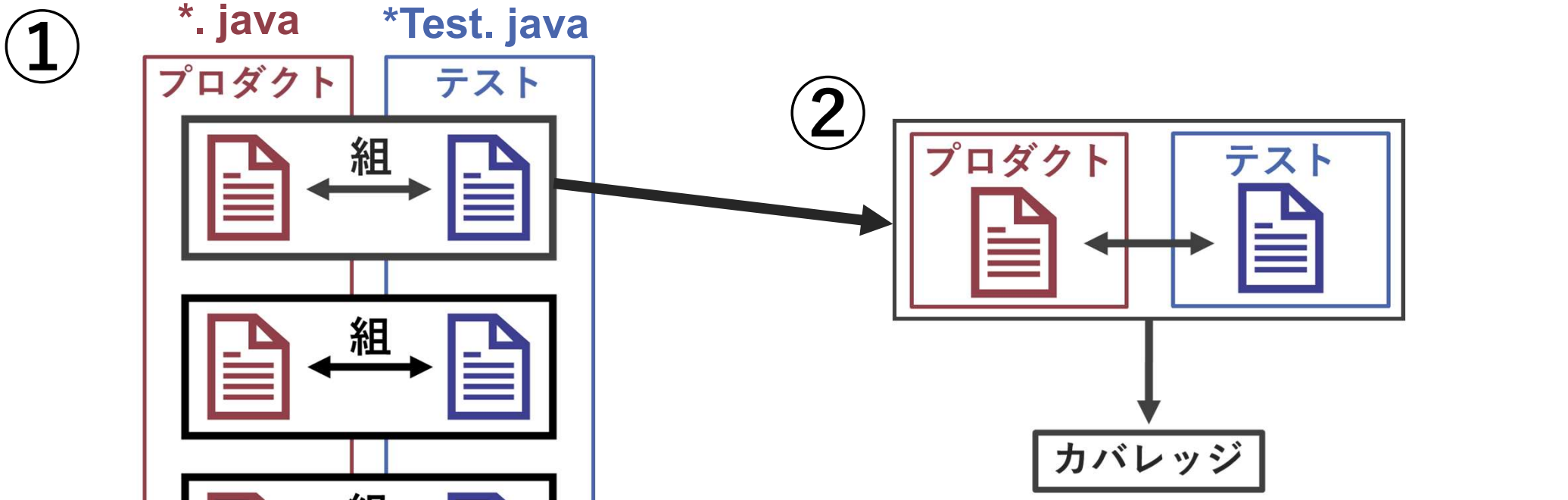
プロダクトコードが期待通りに動作するかを
自動的に検証する為のコード

良し悪しがソフトウェアの信頼性・保守性に大きく影響

カバレッジ(コードカバレッジ)

プロダクトコード全体のうち，どれだけの部分が
テストにより実行されたかを示す指標

研究の背景|カバレッジの算出方法



プロダクトコード1ファイルにつき，
テストコード1ファイルが対応

テストコードとプロダクトコードの組から，
カバレッジの値が1つ求まる

以降，組から求まるカバレッジを，
簡単のため**カバレッジ**と呼ぶ

研究の背景|高品質なテストコード

カバレッジが高いと，テストコードの品質が高い

高品質なテストコードは，分岐構造が少ない
とされている

研究の目的

高品質なテストコードは，分岐構造が少ない

という通説を検証

調査手順

1. テストコードを分岐構造の多さで分類

2. カバレッジを測定

3. 分岐構造の数とカバレッジ間の関連性を分析

調査対象

収集元: GitHub上のJavaプロジェクト

調査対象のプロジェクト数: 45個

収集した組数: 1,836組

プロジェクトの選定基準

- スター数が5,000以上
- MavenまたはGradleでビルド可能

1. テストコードを分岐構造の多さで分類

本研究におけるテストコードの複雑度(TC)

$$\text{定義:}\ \frac{\text{テストコード内の分岐構造の数の和}}{\text{テストメソッドの数}} + 1$$

分岐構造: if, else-if, switch, case

TCの値に基づき, 組を以下の6グループに分類

TC=1	TC>1				
	上位100%~80%	上位80%~60%	上位40%~60%	上位20%~40%	上位~20%
低TC	→				高TC

1. テストコードを分岐構造の多さで分類

TCを導入する利点

テストコード内の複雑度(分岐構造の数)が増加する

様々な原因を, TCの計測のみで区別可能

複雑度が増加する原因の例

- 少数のテストメソッドの中に, 分岐構造が大量に存在
- 少数の分岐構造を持つ, テストメソッドが大量に存在

2. カバレッジを測定

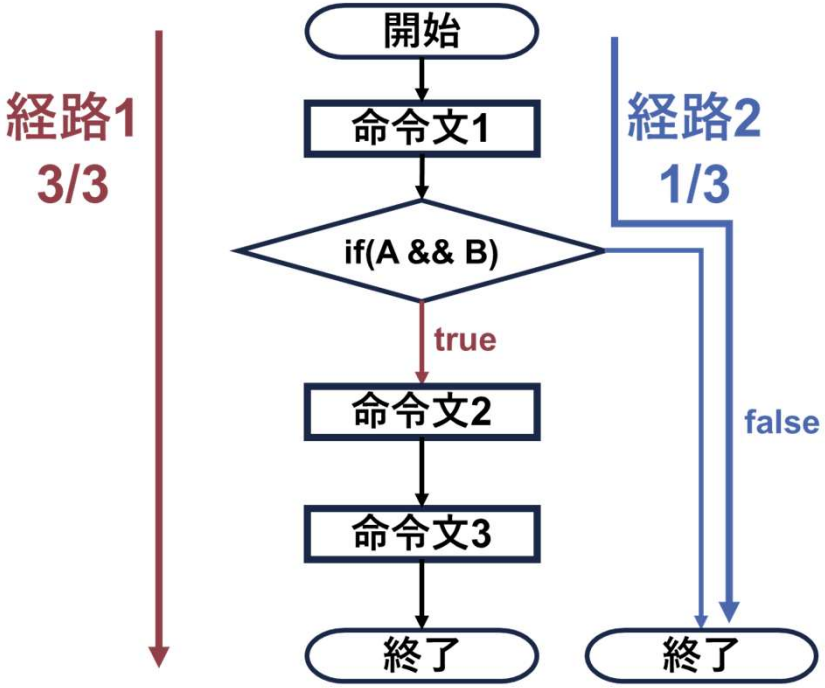
カバレッジを測定する指標は複数存在

測定するカバレッジ	命令網羅率[1]	分岐網羅率[1]
説明	全ての命令文に対する，テストで実行された命令文の割合	全ての分岐に対する，テストで実行された分岐の割合

[1] V. Antinyan, J. Derehag, A. Sandberg and M. Staron, "Mythical Unit Test Coverage, " in *IEEE Software*, vol. 35, no. 3, pp. 73-79, May/June 2018

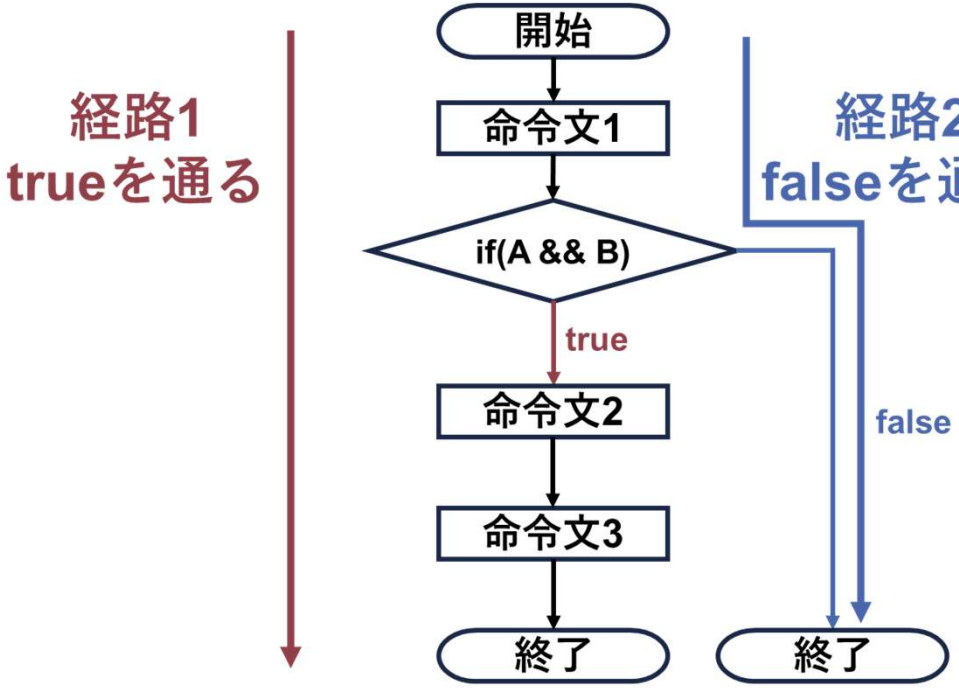
命令網羅率，分岐網羅率の詳細

命令網羅率



全命令文(経路1)を通れば100%

分岐網羅率

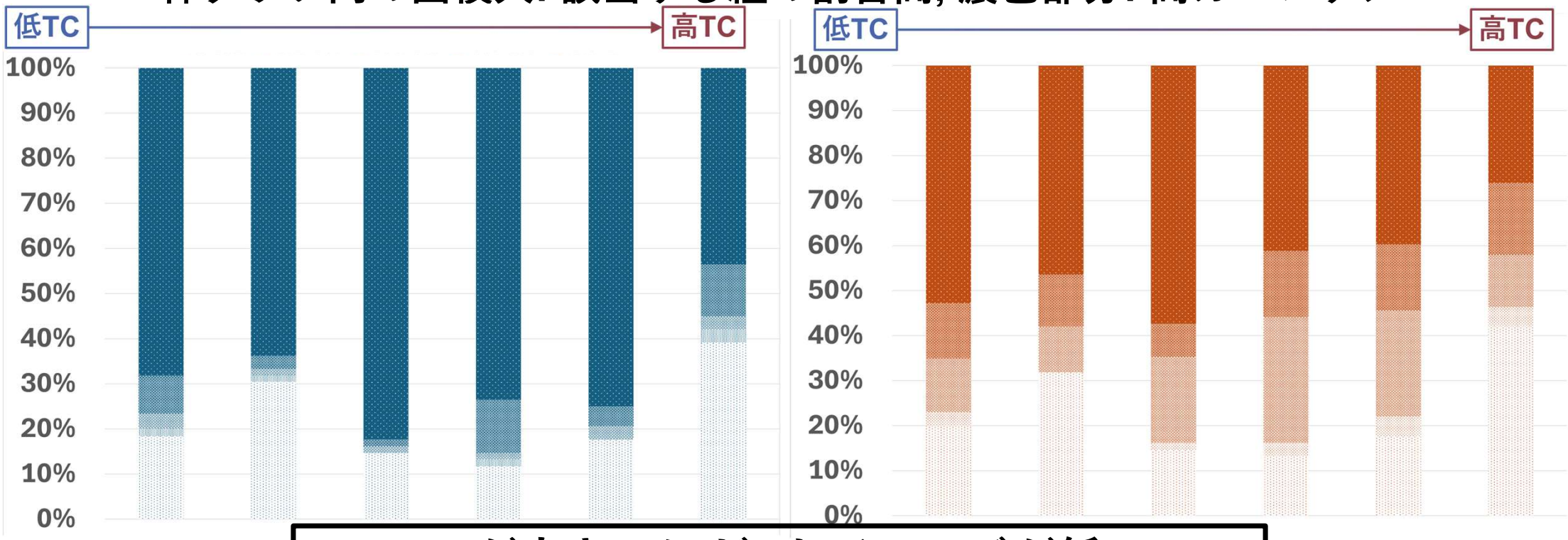


全経路(経路1, 2)を通れば100%

3. 関連性の分析結果

グループ毎の**命令網羅率(左)**, および**分岐網羅率(右)**の分布

棒グラフ内の面積大: 該当する組の割合高, 濃色部分: 高カバレッジ



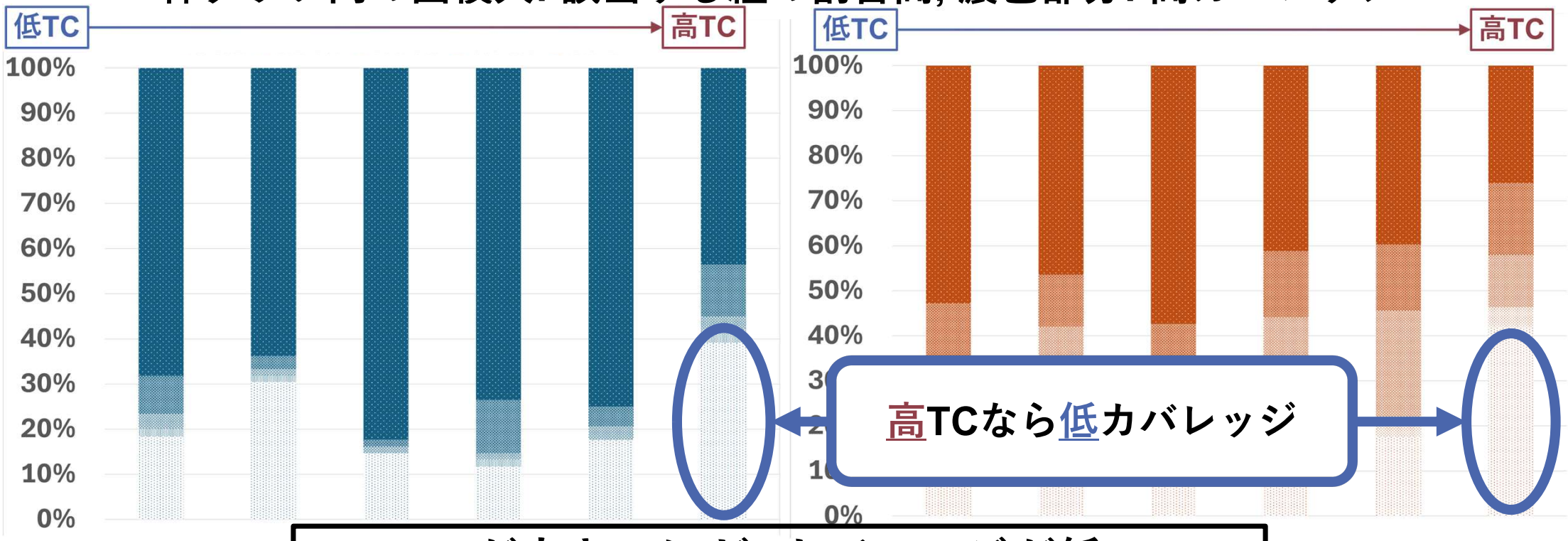
TCが大きいほど, カバレッジが低い

(≒低品質なテストコードは, 分岐構造が多い)

3. 関連性の分析結果

グループ毎の**命令網羅率(左)**, および**分岐網羅率(右)**の分布

棒グラフ内の面積大: 該当する組の割合高, 濃色部分: 高カバレッジ



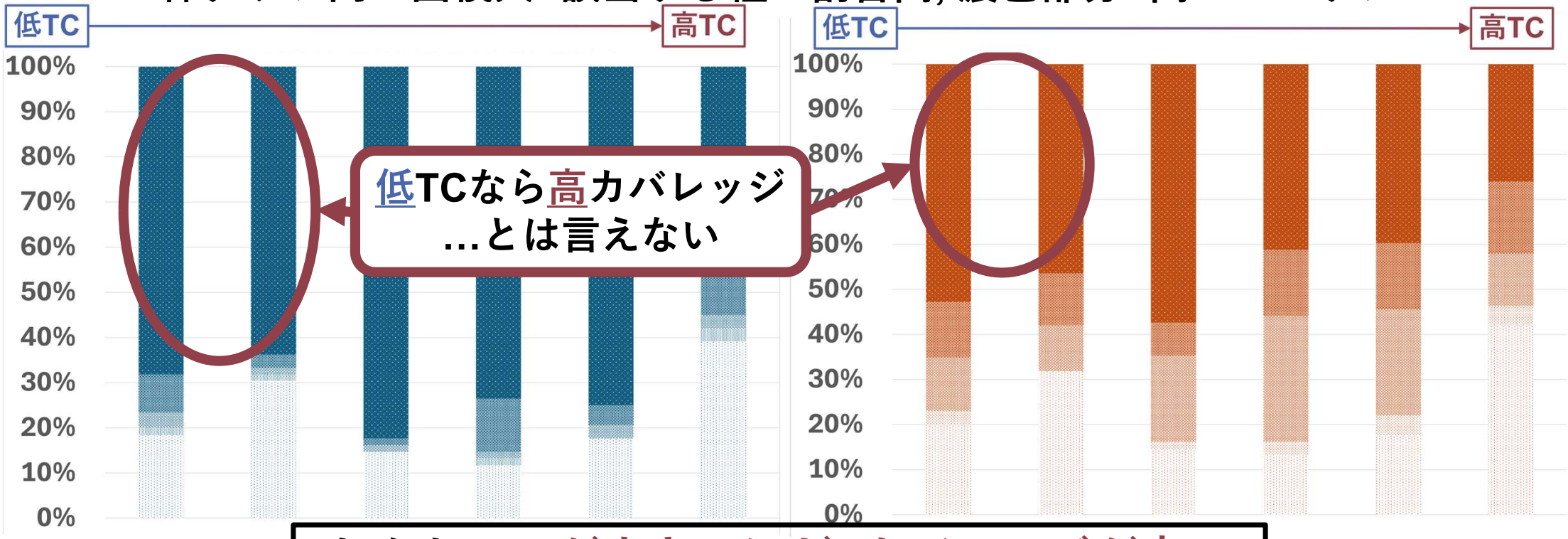
TCが大きいほど, カバレッジが低い

(≒低品質なテストコードは, 分岐構造が多い)

3. 関連性の分析結果

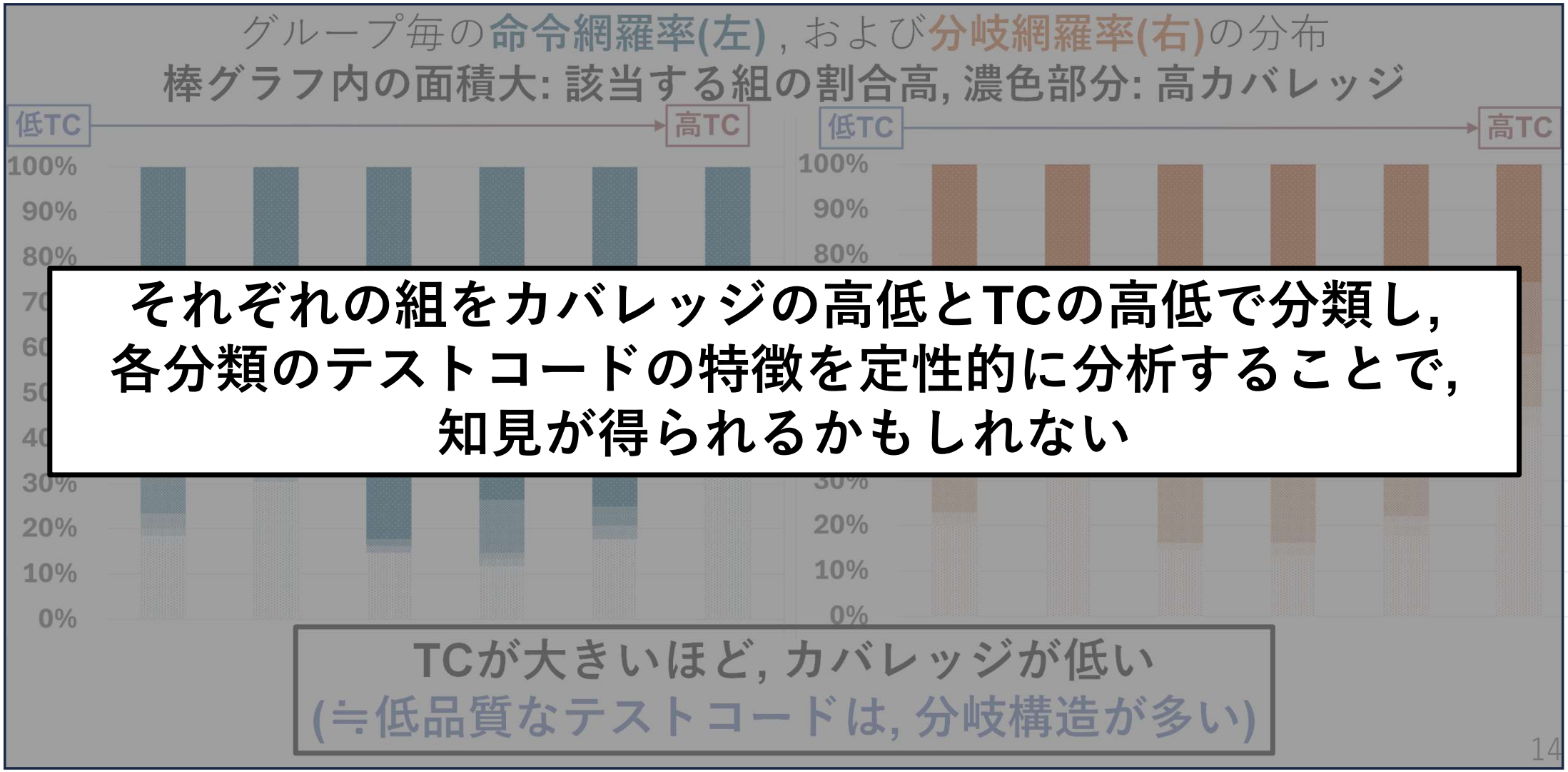
グループ毎の**命令網羅率(左)**, および**分岐網羅率(右)**の分布

棒グラフ内の面積大: 該当する組の割合高, 濃色部分: 高カバレッジ



しかし, TCが小さいほど, カバレッジが高い
とは言えない

3. 関連性の分析結果



**それぞれの組をカバレッジの高低とTCの高低で分類し,
各分類のテストコードの特徴を定性的に分析することで,
知見が得られるかもしれない**

テストコードの分析|分析結果

高カバレッジ

低TC

シンプルなテストメソッドが多数

1つの複雑なテストメソッドで,
複数パターンを過不足なく検証

高TC

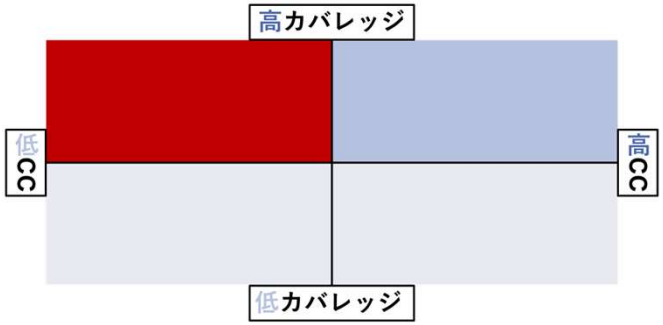
テストメソッド数や記述が不足

無意味に複雑なテストメソッド

低カバレッジ

テストコードの分析|高カバレッジ低TC

シンプルなテストメソッドが多数




```
@Test
void assertGetItemsNode() {
    assertThat(FailoverNode. getItemsNode(0), is("leader/failover/items/0"));
}
```

getItemsNode()が正しいパスを生成することのみ確認

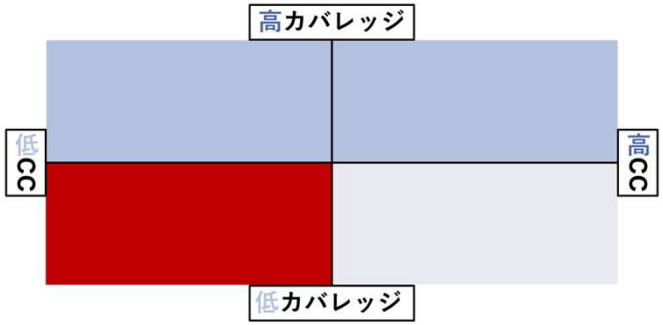
```
@Test
void assertGetExecutionFailoverNode() {
    assertThat(FailoverNode. getExecutionFailoverNode(0), is("sharding/0/failover"));
}
```

getExecutionFailoverNode()が
正しいパスを生成することのみ確認

1つのメソッドが1つの具体的な動作を検証

ソースコードの分析|低カバレッジ低TC

テストメソッド数や記述が不足




```
@Test
public void shouldAppendOtherwise() throws Exception {
    when(context. getBindings()). thenReturn(new HashMap<>());

    boolean result = sqlNode. apply(context);

    asserttrue(result);
    verify(context). appendSql(OTHERWISE_TEXT);
}
```

context.getBindings()が空の場合のみテスト
動作の妥当性の詳細を確認していない

記述は簡潔だが，テストの抜け漏れが多い

まとめ

✕ 高品質なテストコードは，分岐構造が少ない

○ 低品質なテストコードは，分岐構造が多い

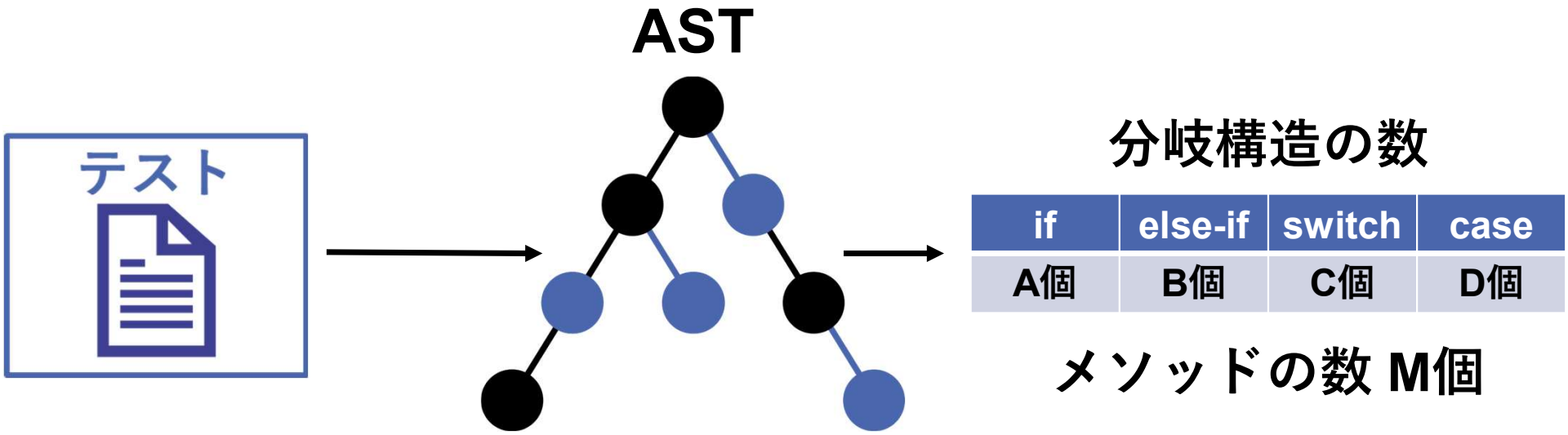
今後の課題

- プロジェクトの調査対象を拡大
- テストコードの特徴を定量的に分析
- テストコードの品質を，分岐構造の数以外の指標でも計測

Appendix

TCの計測方法の詳細

①



テストコード単位で抽象構文木(AST)を作成し,
AST内の分岐構造とメソッドの数を算出

---

②

$$\text{TCの定義:}\ \frac{\text{テストコード内の分岐構造の数の和}}{\text{テストメソッドの数}} + 1$$

①で算出した分岐構造とメソッドの数を使用し, TCの値を計算

TCとサイクロマティック複雑度の違い

TC: McCabeのサイクロマティック複雑度[3]の計測対象・単位を変更したもの

	計測対象	計測単位
テストコードの複雑度 (TC)	・分岐構造	テストコード単位 (*Test. java)
サイクロマティック複雑度	・分岐構造 ・繰り返し構造	テストコード内の関数・メソッド単位

分岐構造: if, else-if, switch, case
繰り返し構造: for, while, do-while

[3]T. J. McCabe, “A Complexity Measure, ” in *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308-320, Dec. 1976

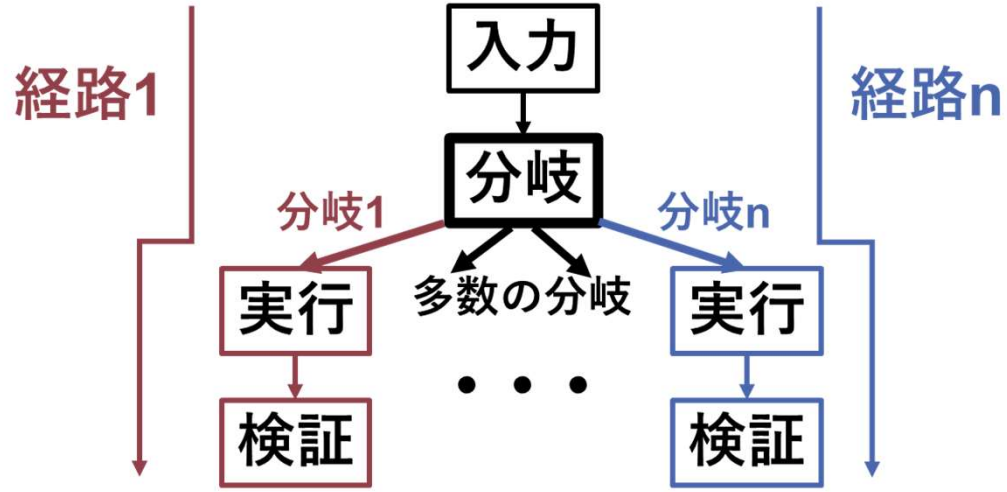
TCの区分をTC=1とTC>1で分けた理由

TC=1とTC>1では，テストコードの構造が異なるため

TC=1

1テストメソッドにつき，
1つの実行経路

TC>1



1テストメソッドにつき，
複数の実行経路

関連性の分析結果 補足

低TCのテストコード

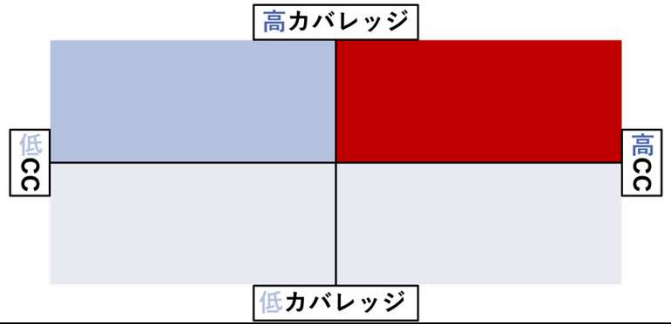
- テスト設計が簡潔
- モック・スタブ準備や前後処理が簡単

高TCのテストコード

- テストで確認すべき実行経路が増え，
  複雑なロジックにより見落としのリスクが増加
- モック・スタブ準備や前後処理が複雑

テストコードの分析|高カバレッジ高TC

1つの複雑なテストメソッドで，
複数パターンを過不足なく検証



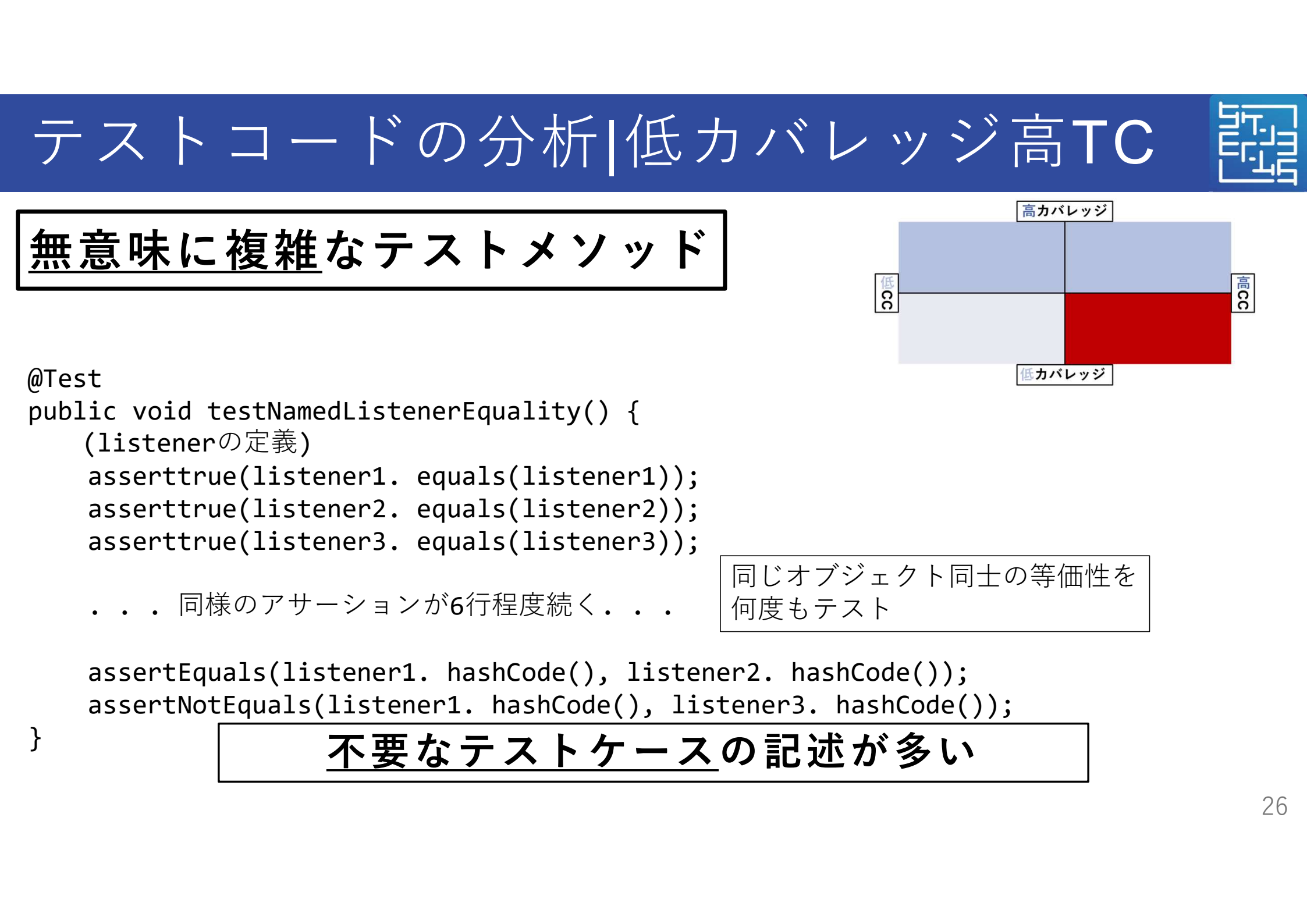

```
@Test
void assertGetShardingErrorJobBriefInfo() {
    (中略，シャード不整合を意図的に作成)
     JobBriefInfo jobBrief = jobStatisticsAPI. getJobBriefInfo("test_job");

     if (jobBrief. getStatus() == JobBriefInfo. JobStatus. SHARDING_FLAG) {
         assertThat(jobBrief. getStatus(), is(JobBriefInfo. JobStatus. SHARDING_FLAG));
     } else if (jobBrief. getStatus() == JobBriefInfo. JobStatus. OK) {
         fail("Expected status SHARDING_FLAG but got OK");
     } else {
         fail("Unexpected job status: " + jobBrief. getStatus());
     }
}
```

jobBrief. getStatus()がSHARDING_FLAGか否かを
if文で正しく反映しているか検証

分岐構造を利用して，
状態の変化に基づく動作を網羅的に検証

テストコードの分析|低カバレッジ高TC



無意味に複雑なテストメソッド


```
@Test
public void testNamedListenerEquality() {
    (listenerの定義)
    asserttrue(listener1. equals(listener1));
    asserttrue(listener2. equals(listener2));
    asserttrue(listener3. equals(listener3));

    . . . 同様のアサーションが6行程度続く. . .

    assertEquals(listener1. hashCode(), listener2. hashCode());
    assertNotEquals(listener1. hashCode(), listener3. hashCode());
}
```

同じオブジェクト同士の等価性を何度もテスト

不要なテストケースの記述が多い