

ソフトウェア工学 第10-12回

— ソフトウェアテスト —

大連理工大学・立命館大学 国際情報ソフトウェア学部

大森 隆行

講義内容

➡ ソフトウェアテスト

- 単体テスト
- 統合テスト
- システムテスト
- 受け入れテスト・設置テスト
- ソフトウェア検証

ソフトウェアテスト

■ ソフトウェアテスト

- プログラムにテストデータを与えて実行し、
その結果から開発中に紛れ込んだ誤りを検出する作業

■ ソフトウェアテストに関する用語 (cf. JIS X0014-1999)

- **誤り (error)** Error是指实际值偏离你最初的期望，往往是由人为造成的。

- 計算、観測もしくは測定された値または状態と、
真の、指定されたもしくは論理的に正しい値または状態との間の相違
→ 外部的な振る舞いとして、正しく動かないこと

- **故障 (failure)** failure是一个系统或者组件不能完成它被要求的功能。failure通常是由fault导致的。

- 要求された機能を遂行する、機能単位の能力がなくなること
→ 外部的な振る舞いとして、正しく動かなくなること

- **障害 (fault)、欠陥(defect)、バグ(bug)**

- 要求された機能を遂行する機能単位の能力の、
縮退または喪失を引き起こす、異常な状態
→ 障害は内部的に間違いがあるという状態
→ 障害が原因で故障や誤りが発生

fault通常是由于程序中不正确的步骤，过程或者数据定义，导致程序出现了非故意的、不可预料的行为。fault通常是由error导致的。

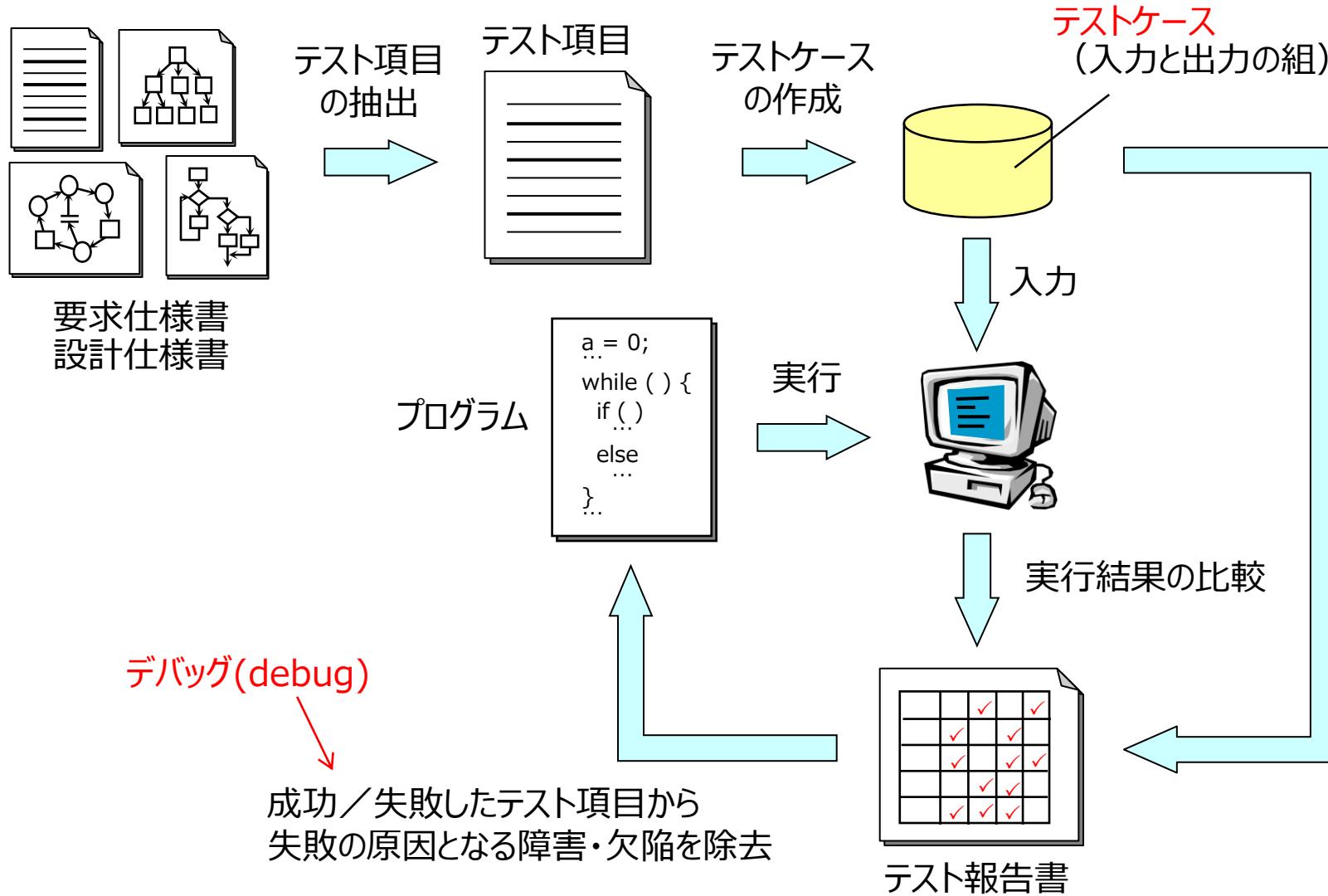
※ 人間の過ちを誤りとする考え方もある。

この場合、誤りによって、故障や障害が発生すると言える。

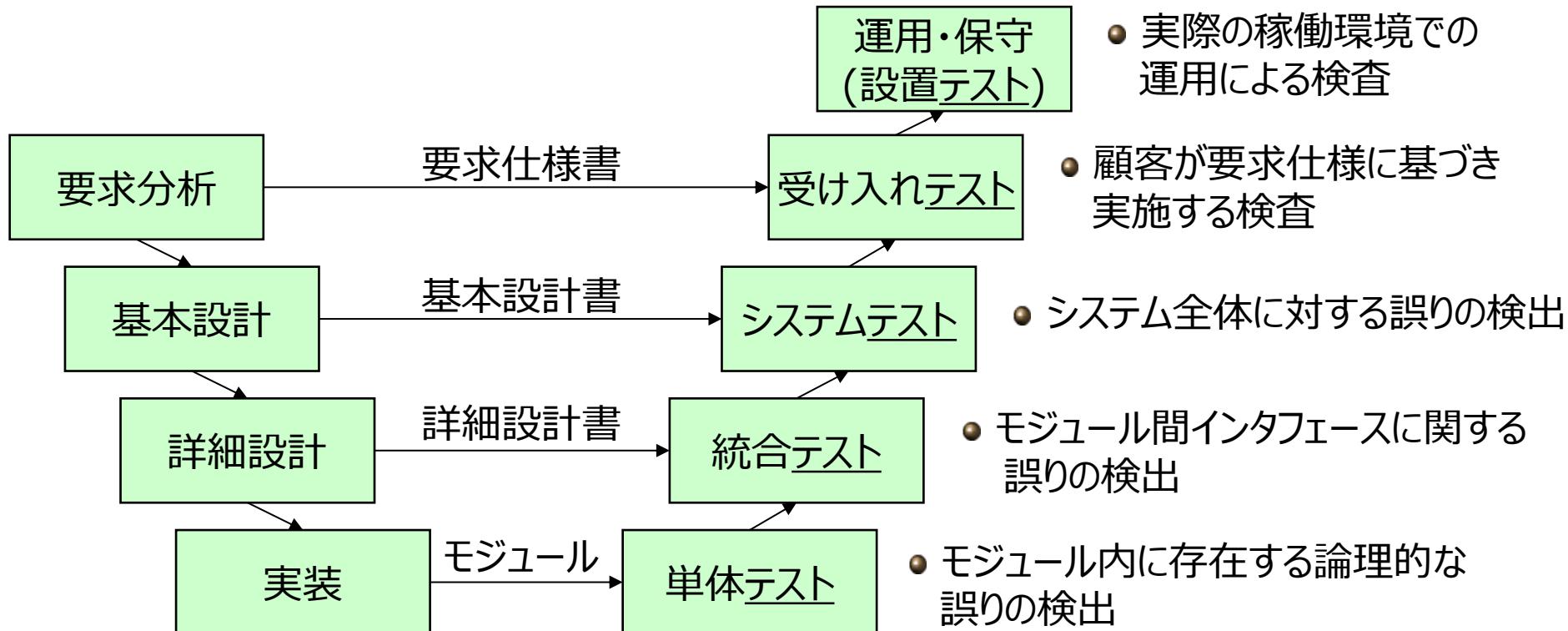
ソフトウェアテストの性質

- ソフトウェアに障害や誤りがないことを示すことはできない
 - ソフトウェアが正しく動作することではなく、正しく動作しないことを示すために実施
- 直接的にはソフトウェアの品質を向上させない
 - 設計工程、実装工程で十分な品質になるように作り込んでおくことが必要

ソフトウェアテストの作業



テスト工程 (V字モデル)



講義内容

- ソフトウェアテスト
 - ➡ 単体テスト
 - 統合テスト
 - システムテスト
 - 受け入れテスト・設置テスト

単体テスト

■ 単体テスト (unit test)

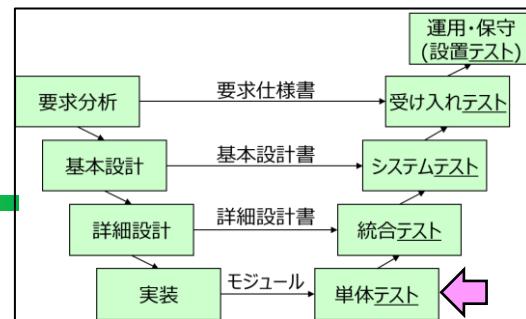
- モジュール内部に存在する誤りを検出

(a) ブラックボックステスト (black-box test)

- テストデータを与えて、実行結果を観察することで誤りを検出
- プログラムの外部仕様（機能、振る舞い）に着目
- 代表的な技法：同値分割、境界値分析、原因結果グラフ

(b) ホワイトボックステスト (white-box test)

- テストデータを与えて、実行の様子を追跡することで誤りを検出
- プログラムの内部仕様（構造や論理）に着目
- 制御の流れに基づくテスト網羅
- 代表的な技法：論理網羅



同値分割

- プログラムの入力値の領域を、機能仕様の入力条件を満たす範囲（有効同値クラス）と満たさない範囲（無効同値クラス）に分割
- 同値クラス：プログラムが同じ動作をする入力値の範囲や種類

(例)	入力条件	有効同値クラス	無効同値クラス
文字数	4以上8以下	3以下、9以上	
文字の種類	英字と数字の組合せ	英字のみ、数字のみ	
先頭文字	英字	数字	

手順

- (1) すべての有効同値クラスに属する入力データを作成
- (2) 1つの無効同値クラスと残りの有効同値クラスに属する入力データを作成

(例) amku5ge, X123

(例) xy3 (3文字),
xyxyxyxy9 (9文字),
abcdef (英字のみ),
123456gh (先頭が数字),
(数字のみは先頭が数字に包含)

境界値分析

■ 入出力条件の境界値を詳しくテストする テストケースを作成

- テストケース(test case) : プログラムのテストのために入力する値と想定される出力値の組
- 境界値やその付近の値を入力データとする
- 経験的に、多くの誤りが境界値付近で発生
 - 不等号(<と<=, >と>=)の書き間違い等

■ 手順

- (1) 入出力条件の識別

(例)	条件	1から64の数字
	境界	1, 64

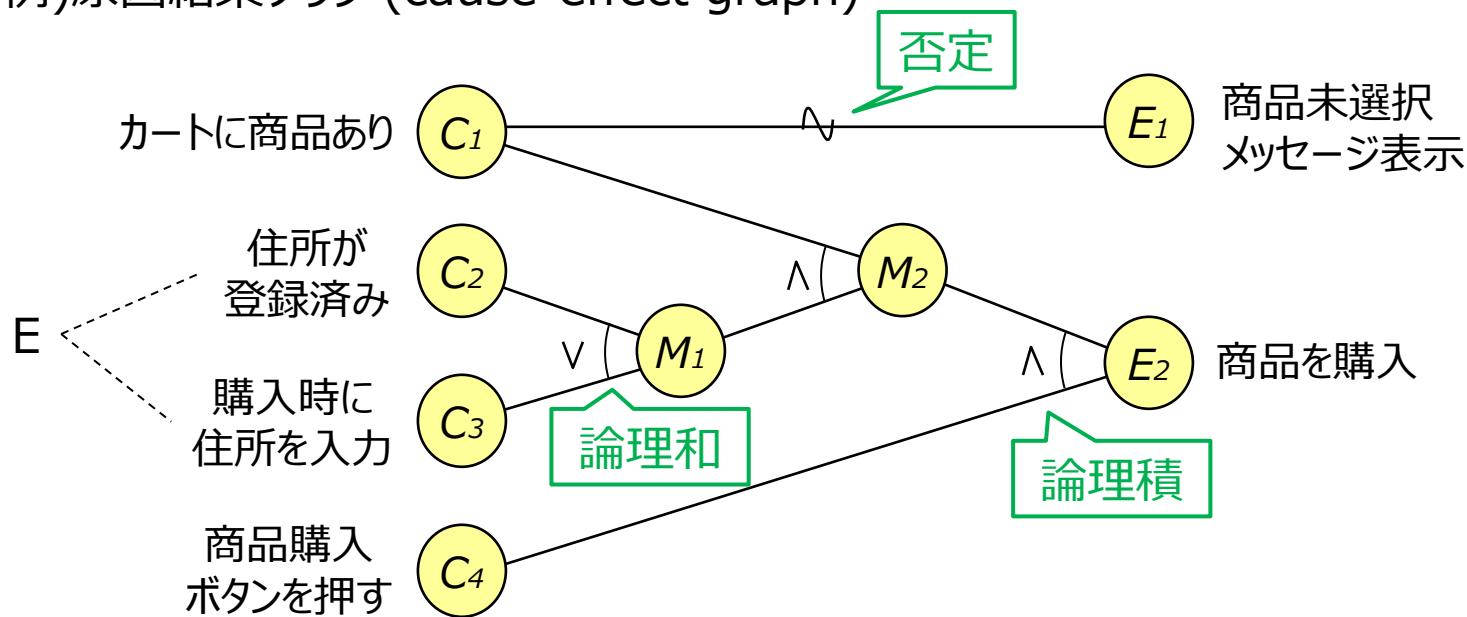
- (2) 境界に基づくテストケースの作成

- (例) 0, 1, 64, 65

原因結果グラフ(cause-effect graph)

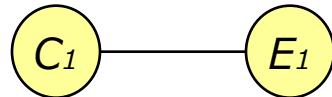
- 原因（入力）と結果（出力）の因果関係に着目し、テストケースを作成
 - 因果関係は原因結果グラフで表現
 - 原因結果グラフから決定表を作成

(例)原因結果グラフ (cause-effect graph)

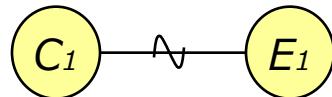


原因結果グラフ

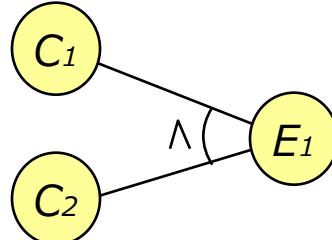
(a) 肯定(C_1 であれば E_1)



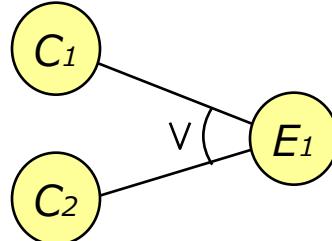
(b) 否定(C_1 でなければ E_1)



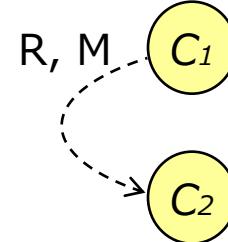
(c) 論理積(C_1 かつ C_2 であれば E_1)



(d) 論理和(C_1 または C_2 であれば E_1)



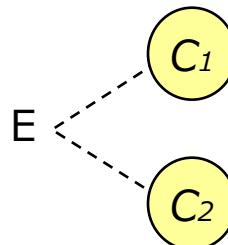
(e) 必要(R), マスク(M)



R (require): 一方が成立すれば他方も成立

M (mask): 一方が成立すれば他方は不成立

(f) 排他的論理和(E), 包含(I), 1つのみ(O)

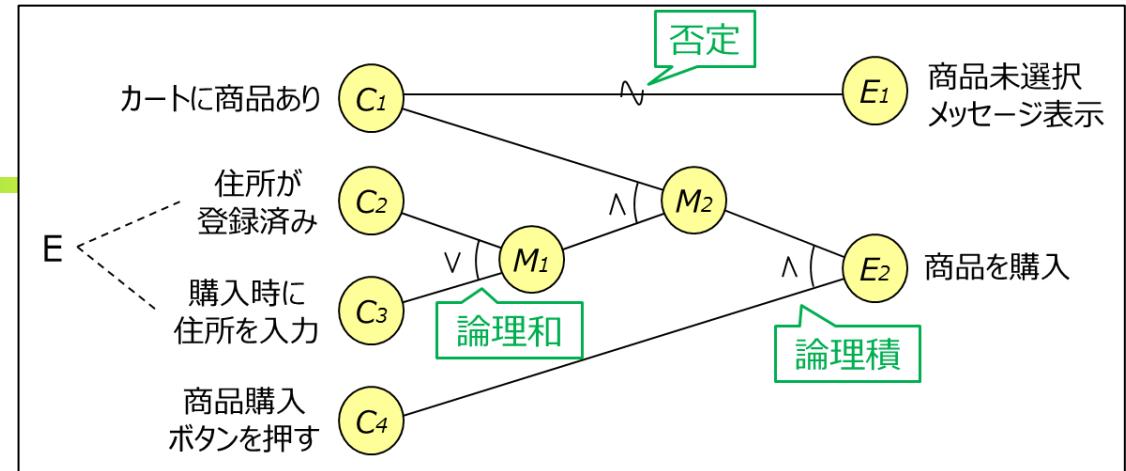


E (exclusive): 同時に成立しない

I (include): 少なくとも一方は成立

O (only one): 常に一つだけ成立

決定表



	1	2	3	4	5	6	7	8	9	10	11	12
原因	C1 (カートに商品あり)	×	×	×	×	×	○	○	○	○	○	○
	C2 (住所が登録済み)	×	×	×	×	○	○	×	×	×	○	○
	C3 (購入時に住所を入力)	×	×	○	○	×	×	×	○	○	×	×
	C4 (商品購入ボタンを押す)	×	○	×	○	×	○	×	○	×	○	○
補助	M1 (住所を入力済み)	×	×	○	○	○	○	×	○	○	○	○
	M2 (購入可能)	×	×	×	×	×	×	×	○	○	○	○
結果	E1 (商品未選択)	○	○	○	○	○	○	×	×	×	×	×
	E2 (商品を購入)	×	×	×	×	×	×	×	×	○	×	○

- C2 \wedge C3の場合は(排他関係なので)考慮しない
- 決定表に基づいて入出力をチェック

確認問題

■ 以下の各文が正しいかどうかを答えよ。

- ソフトウェアテストは、ソフトウェアに障害や誤りがないことを示すために行う。
- 単体テストでは、特にモジュール間のインターフェースの誤りに関してテストを行う。
- ブラックボックステストではモジュールの内部仕様に着目してテストデータを作成する。
- ホワイトボックステストではモジュールの内部仕様に着目してテストデータを作成する。

確認問題

■ 以下の各文が正しいかどうかを答えよ。

- ソフトウェアテストは、ソフトウェアに障害や誤りがないことを示すために行う。X
- 単体テストでは、特にモジュール間のインターフェースの誤りに関してテストを行う。X
- ブラックボックステストではモジュールの内部仕様に着目してテストデータを作成する。X
- ホワイトボックステストではモジュールの内部仕様に着目してテストデータを作成する。O

Xテストは誤りがあることを示すために行う

X単体テストはモジュール内部の機能を検査

Xブラックボックステストではモジュールの中身を考慮しない

O正しい

確認問題

- 下記のような入力条件を持つモジュールに対して、同値分割に基づく単体テストを実施する。下の(1)~(5)の各入力は、右下に示した選択肢の説明のうち、どれに該当するかを答えよ。

入力条件	有効同値クラス	無効同値クラス
文字数	4以上8以下	3以下、9以上
文字の種類	英字と数字の組合せ	英字のみ、数字のみ
先頭文字	英字	数字

- (1) DL4
- (2) 678
- (3) qwerty98
- (4) 55RITS
- (5) A12345678

■ 選択肢

- 1. 「手順(1) すべての有効同値クラスに属する入力データを作成」において使用可能
- 2. 「手順(2) 1つの無効同値クラスと残りの有効同値クラスに属する入力データを作成」において使用可能
- 3. 手順(1)(2)では使用不可能

確認問題

- 下記のような入力条件を持つモジュールに対して、同値分割に基づく単体テストを実施する。下の(1)～(5)の各入力は、右下に示した選択肢の説明のうち、どれに該当するかを答えよ。

入力条件	有効同値クラス	無効同値クラス
文字数	4以上8以下	3以下、9以上
文字の種類	英字と数字の組合せ	英字のみ、数字のみ
先頭文字	英字	数字

- (1) DL4 2
- (2) 678 3
- (3) qwerty98 1
- (4) 55RITS 2
- (5) A12345678 2

1.3 文字以下に該当→2
2.3 文字以下かつ数字のみかつ先頭数字→3
3.すべての有効同値クラスに属する→1
4.先頭が数字に該当→2
5.9 文字以上→2

■ 選択肢

- 1. 「手順(1) すべての有効同値クラスに属する入力データを作成」において使用可能
- 2. 「手順(2) 1つの無効同値クラスと残りの有効同値クラスに属する入力データを作成」において使用可能
- 3. 手順(1)(2)では使用不可能

確認問題

- (1) int型の引数xが0以上の場合にtrue、そうでない場合にfalseを返す関数がある。境界値分析に基づき、この関数の単体テストを行う。テストケース2通りを与える場合に最適なものは以下のうちどれか答えよ。
- (a) 0, 1
 - (b) -10, 10
 - (c) -1, 0
- (2) int型の引数yを入力値とする関数がある。この関数は、yが4で割り切れる場合はtrueを返す(それ以外の場合はfalse)。ただし、例外として、入力値が100で割り切れ、400で割り切れない場合はfalseを返す。この関数について、入力(原因)と出力(結果)の関係を下のような決定表としてまとめよ。

原因	4で割り切れる				
	100で割り切れる				
	400で割り切れる				
結果	trueを出力				
	falseを出力				

確認問題

以上: 包含

- (1) int型の引数xが0以上の場合にtrue、そうでない場合にfalseを返す関数がある。境界値分析に基づき、この関数の単体テストを行う。テストケース2通りを与える場合に最適なものは以下のうちどれか答えよ。
 - (a) 0, 1 → trueを返す場合しか検査できていない
 - (b) -10, 10 → 境界値から離れすぎ
 - (c) -1, 0 → 正解
- (2) int型の引数yを入力値とする関数がある。この関数は、yが4で割り切れる場合はtrueを返す(それ以外の場合はfalse)。ただし、例外として、入力値が100で割り切れ、400で割り切れない場合はfalseを返す。この関数について、入力(原因)と出力(結果)の関係を下のような決定表としてまとめよ。

原因	4で割り切れる	✓	✓	✓	✗
	100で割り切れる	✗	✓	✓	✗
	400で割り切れる	✗	✗	✓	✗
結果	trueを出力	✓		✓	
	falseを出力		✓		✓

論理網羅

coverage

■ カバレッジ基準

■ 命令網羅(文網羅)

- プログラム中のすべての文を1回以上実行

■ 分岐網羅

- プログラム中のすべての分岐を1回以上実行

■ 条件網羅

- プログラム中の分岐に関連する全ての条件判定を少なくとも1回は実行

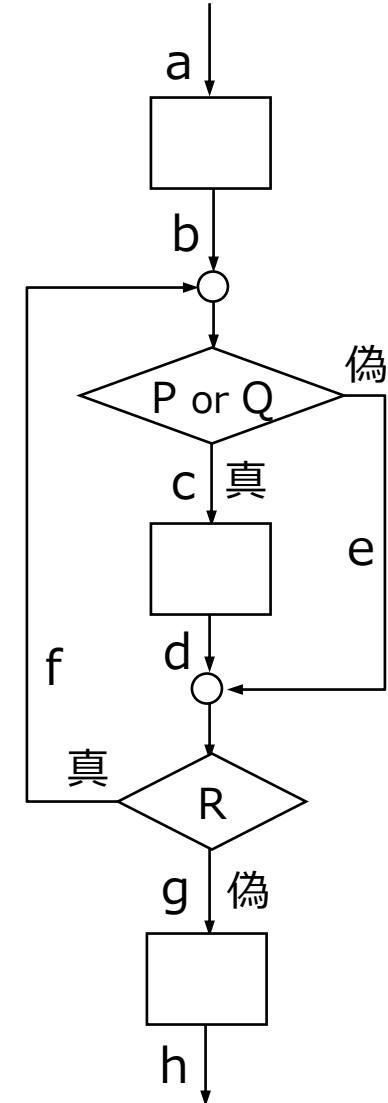
■ パス網羅

- プログラム中のすべての実行パス(経路)を実行

■ テストにかけられる工数、求められる品質に応じて、どの基準でどの程度の網羅率を達成するか検討する

命令網羅

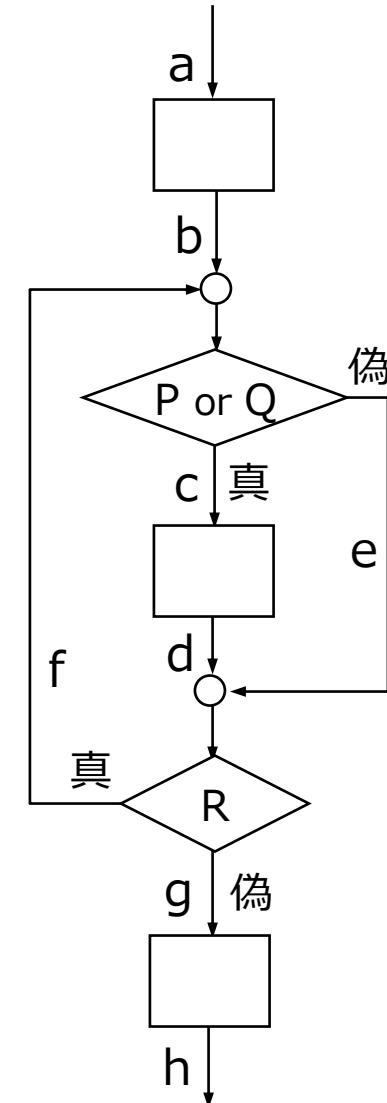
- C0, statement coverage
- プログラム中のすべての文を1回以上実行
 - (例) P or Qが真、Rが偽 (パスabcdgh)
- 網羅率
 - 実行した文 / 全文



パス(path) = 経路

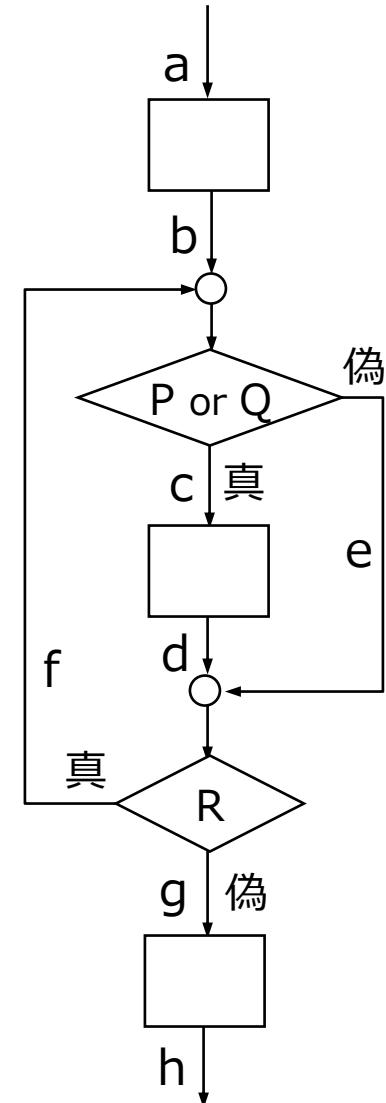
分岐網羅

- C1, edge coverage
- プログラム中のすべての分岐を1回以上実行
 - (例) Pが真、Qが偽、Rが真 (パスabcdf)
Pが偽、Qが偽、Rが偽 (パスabegh)
→ Qが真の場合のテストが抜けている
- 網羅率
 - 通過した枝 / 全枝



条件網羅

- C2, condition coverage
- プログラム中の分岐に関連する各条件判定について、真偽を少なくとも1回は取るように実行
 - 右図の例だと、Pが真、Pが偽、Qが真、Qが偽、Rが真、Rが偽の場合を網羅すればOK
 - (例) Pが真、Qが偽、Rが真 (パスabcdef)
Pが偽、Qが真、Rが偽 (パスabcdgh)
→ P,Q が偽：パス e のテストが抜けている
- 網羅率
 - 実行した条件判定結果の数の合計 / 各条件が取り得る結果の数の合計

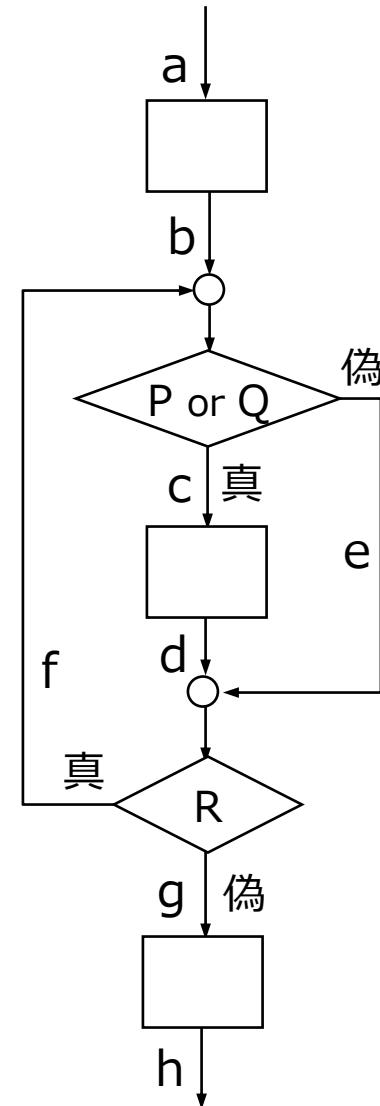


分岐/条件網羅

branch/condition coverage

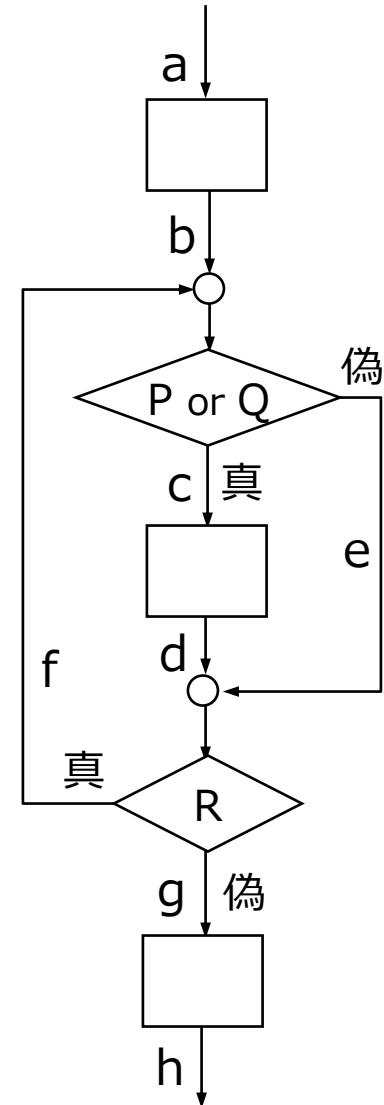
分岐網羅 + 条件網羅

- (例) Pが偽、Qが偽、Rが偽 (パス abegh)
Pが真、Qが偽、Rが偽 (パス abcdgh)
Pが偽、Qが真、Rが真 (パス abcdf)
→ PとQが真の場合 ?



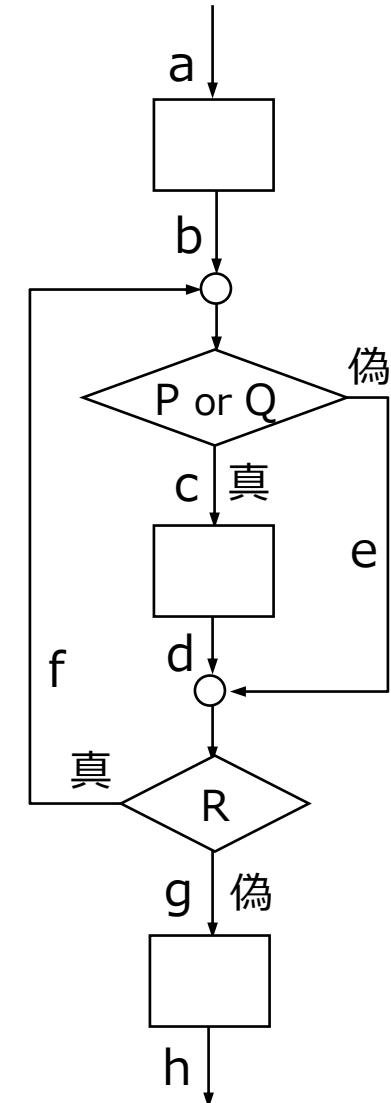
複数条件網羅

- multiple condition coverage
- すべての条件判定の真偽の組み合わせ
 - $(P, Q, R) = (\text{真}, \text{真}, \text{真}), (\text{真}, \text{真}, \text{偽}), (\text{真}, \text{偽}, \text{偽}), (\text{真}, \text{偽}, \text{真}), (\text{偽}, \text{真}, \text{真}), (\text{偽}, \text{真}, \text{偽}), (\text{偽}, \text{偽}, \text{偽}), (\text{偽}, \text{偽}, \text{真}),$
- 網羅率
 - 実行した真偽の組み合わせ / すべての組み合わせ



パス網羅

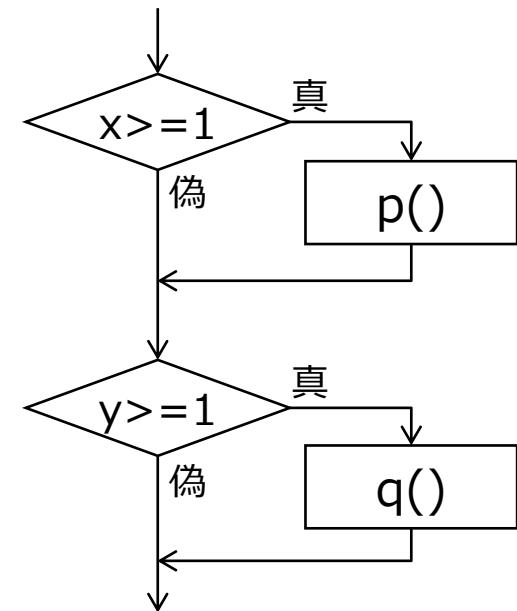
- path coverage
- プログラム中のすべての実行パス(経路)を実行
 - 条件間の依存関係を考慮
 - (例) abcdgh
 - + abcdfcdgh
 - + abegh
 - + abefcdgh
 - + abcdfegh + ... (無限に存在)
- 網羅率
 - 実行したパス / 全パス
- 通常、パス網羅100%は要求されない



確認問題

■ 右のフローチャートに示すような挙動のプログラムがあるとする。以下のような入力の組を与えるとき、それぞれの場合に、命令網羅、分岐網羅、条件網羅、パス網羅を満たすか(網羅率100%となるかどうか)どうか答えよ。

- (1) $(x, y)=(0, 0)$
- (2) $(x, y)=(1, 1)$
- (3) $(x, y)=(0, 1), (1, 0)$
- (4) $(x, y)=(0, 0), (1, 1)$
- (5) $(x, y)=(0, 0), (0, 1), (1, 0), (1, 1)$

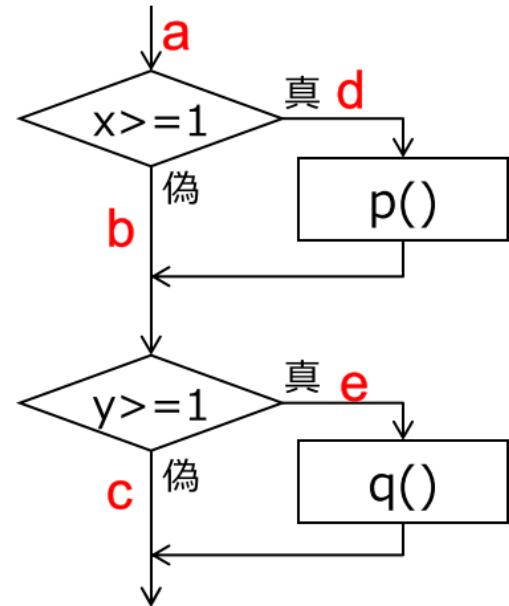


確認問題

■ 右のフローチャートに示すような挙動のプログラムがあるとする。以下のような入力の組を与えるとき、それぞれの場合に、命令網羅、分岐網羅、条件網羅、パス網羅を満たすか(網羅率100%となるかどうか)どうか答えよ。

- (1) $(x, y)=(0, 0)$
- (2) $(x, y)=(1, 1)$
- (3) $(x, y)=(0, 1), (1, 0)$
- (4) $(x, y)=(0, 0), (1, 1)$
- (5) $(x, y)=(0, 0), (0, 1), (1, 0), (1, 1)$

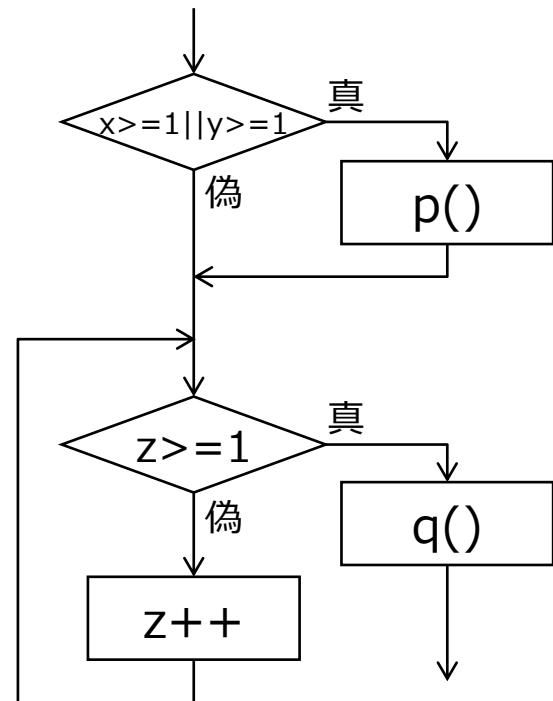
1. 実行パス: abc → いずれも満たさない
2. 実行パス: ade → 命令網羅を満たす
3. 実行パス: abe, adc → 命令網羅、分岐網羅、条件網羅を満たす
4. 実行パス: abc, ade → 命令網羅、分岐網羅、条件網羅を満たす
5. 実行パス: abc, abe, adc, ade → 命令網羅、分岐網羅、条件網羅、パス網羅を満たす



確認問題

■ 右のフローチャートに示すような挙動のプログラムがあるとする。以下のような入力の組を与えるとき、それぞれの場合に、命令網羅、分岐網羅、条件網羅、パス網羅を満たすか(網羅率100%となるかどうか)どうか答えよ。

- (1) $(x, y, z)=(0, 1, 0)$
- (2) $(x, y, z)=(0, 0, 0), (0, 1, 0)$
- (3) $(x, y, z)=(0, 0, 0), (0, 1, 0), (1, 0, 0)$



確認問題

右のフローチャートに示すような挙動のプログラムがあるとする。以下のような入力の組を与えるとき、それぞれの場合に、命令網羅、分岐網羅、条件網羅、パス網羅を満たすか(網羅率100%となるかどうか)どうか答えよ。

- (1) $(x, y, z)=(0, 1, 0)$
- (2) $(x, y, z)=(0, 0, 0), (0, 1, 0)$
- (3) $(x, y, z)=(0, 0, 0), (0, 1, 0), (1, 0, 0)$

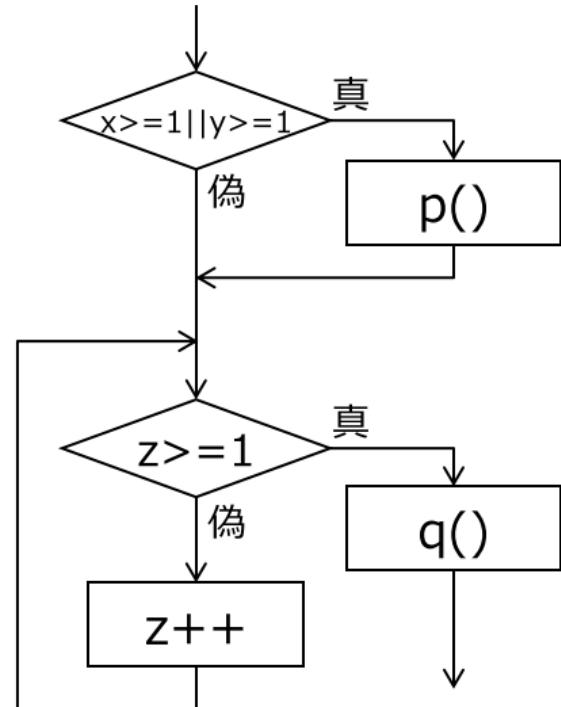
1. 実行パス: adce → 命令網羅を満たす

2. 実行パス: abe,adce → 命令網羅、分岐網羅を満たす
 $x>=1$ が真になる場合を実行していないので条件網羅は満たさない

3. 実行パス: abce,adce,adce

三つの条件 $x>=1,y>=1,z>=1$ すべてについて真偽両方実行できている
→ 命令網羅、分岐網羅、条件網羅を満たす

繰り返しの回数を網羅していないので、パス網羅を満たさない



JUnit

- Javaプログラムの**単体テスト自動化**のためのフレームワーク
 - 単体テスト実行の自動化
 - テストコードの記法を規定
 - テスト用のAPI提供
- 参考：他の言語用のフレームワークもある
→ e.g., CUnit(C言語用), CppUnit (C++用)

JUnit実用例

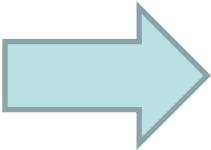
```
package testsample;

public class Sample {
    public static int div(int x, int y){
        return x/y;
    }
}
```

除算プログラム

入力	出力
4, 2	2
11, 3	3
4, -2	-2
1, 0	例外
...	...

テストケース



```
package testsample;

import org.junit.Test;

public class SampleTest {
    @Test
    public void testSample1(){
        int a = Sample.div(4, 2);
        assertEquals(a, 2);
    }

    @Test
    public void testSample2(){
        int a = Sample.div(11, 3);
        assertEquals(a, 3);
    }

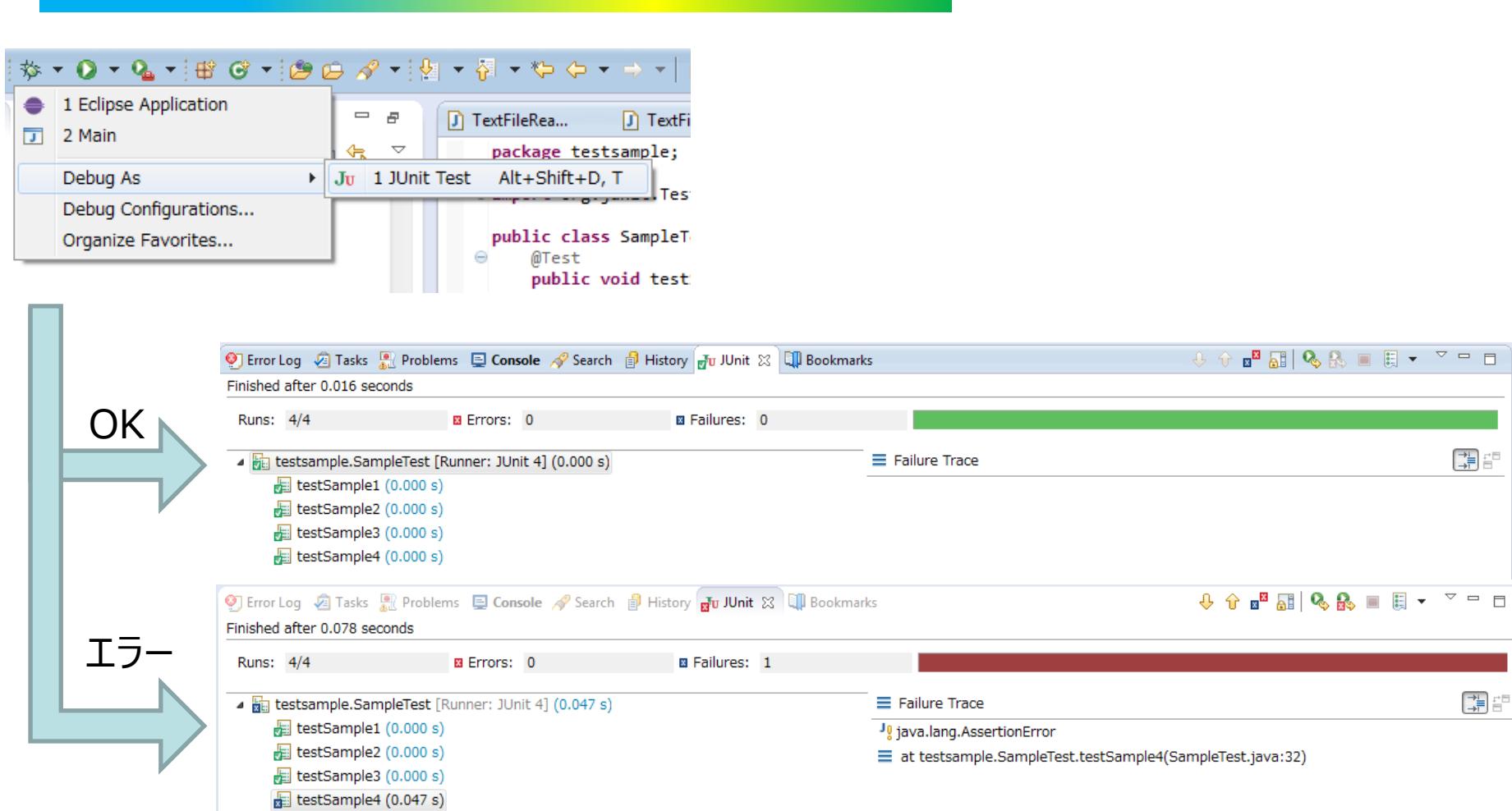
    @Test
    public void testSample3(){
        int a = Sample.div(4, -2);
        assertEquals(a, -2);
    }

    @Test
    public void testSample4(){
        try{
            Sample.div(1, 0);
        }catch(ArithmetiException e){
            return;
        }
        fail();
    }
}
```

アサーション
(assertion)

テストコード

JUnit実用例

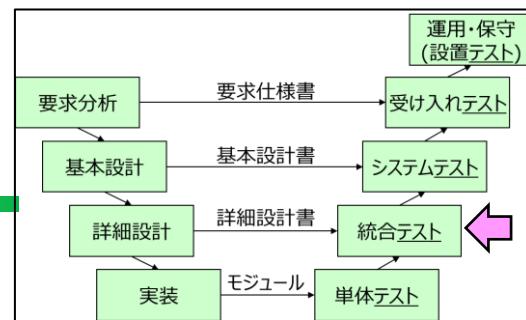


講義内容

■ ソフトウェアテスト

- 単体テスト
- ➡ 統合テスト
- システムテスト
- 受け入れテスト・設置テスト

統合テスト



■ 統合テスト(integration test) =結合テスト

- 複数のモジュールを組み合わせてテストを実施
- モジュールのインターフェースに関するエラーを検出

另一种方法是把下一个要测试的模块同已经测试好的那些模块结合起来进行测试，测试完以后再把下一个应该测试的模块结合起来测试。这种每次增加一个模块的方法称为渐增式测试。

(a) ボトムアップテスト(bottom-up test)

- モジュール構成図の最下位モジュールからテスト開始

(b) トップダウンテスト(top-down test)

- モジュール構成図の最上位モジュールからテスト開始

(c) 混合テスト(mixed integration)/サンドイッチテスト(sandwich test)

- ボトムアップテストとトップダウンテストの統合

- 基本的にはトップダウン top-down

- スタブの作成が困難な部位や、下位に存在する重要な部分に対してのみ
ボトムアップで行う

一种方法是先分别测试每个模块，再把所有模块按设计要求放在一起结合成所要的程序，这种方法称为非渐增式测试方法或一次性组装(Big Bang、 ビッグバン)

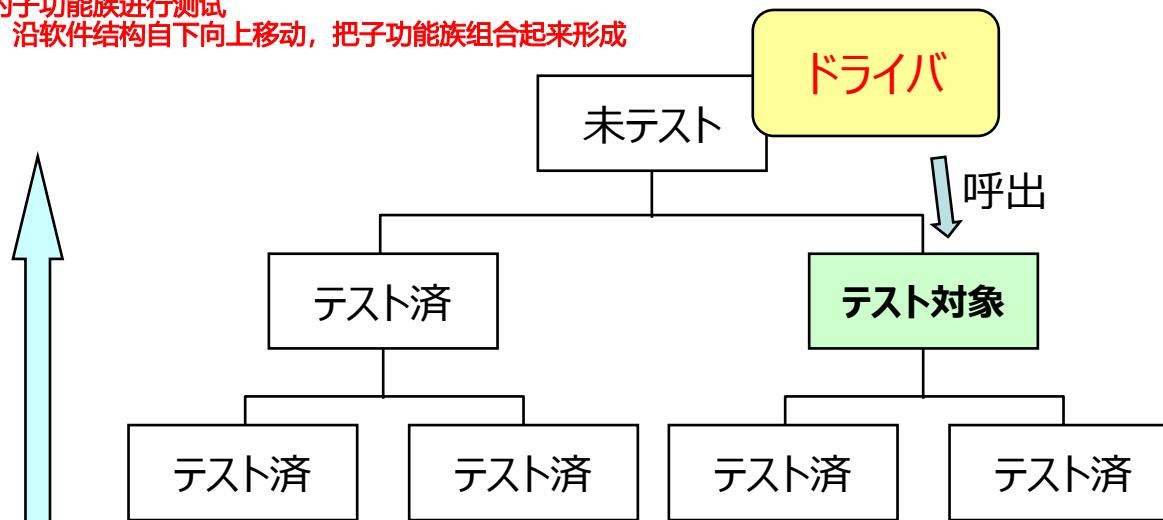
(d) ビッグバンテスト(big-bang test)

- すべての構成要素を単独でテスト後、一斉に統合してテスト
- 小規模な開発等の場合に適用

ボトムアップテスト

- モジュール構成図の最下位モジュールからテスト開始
- ドライバ (仮の呼び出しモジュール) が必要
- 長所 : 初期段階から並行に数多くのモジュールを ^{driver} テスト可能
- 短所 : 上位でエラーが見つかると大幅な再修正が必要

1. 把低层模块组合成实现某特定软件子功能的族
2. 写一个驱动程序(用于测试的控制程序), 协调测试数据的输入和输出
3. 对由模块组成的子功能族进行测试
4. 去掉驱动程序, 沿软件结构自下向上移动, 把子功能族组合起来形成更大的子功能族



トップダウンテスト

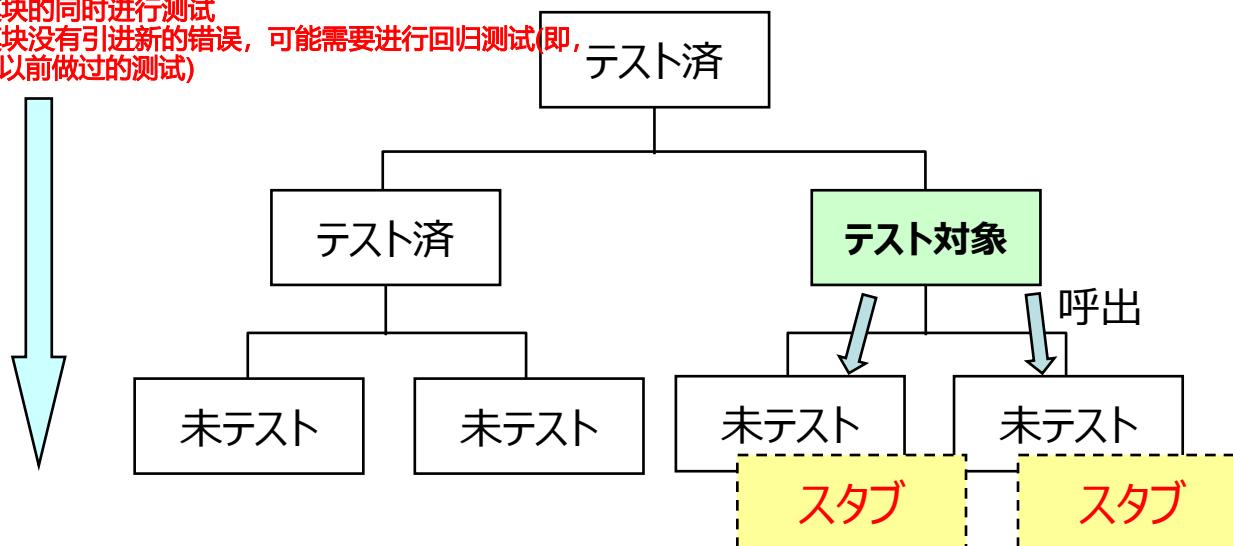
- モジュール構成図の最上位モジュールからテスト開始
- **スタブ**^{stub} (仮の被呼び出しモジュール) が必要
- 長所: インタフェースの誤りを早期に発見可能
- 短所: テストの初期段階で作業の並列化が困難

1. 对主控制模块进行测试, 测试时用存根程序代替所有直接附属于主控制模块的模块

2. 根据选定的结合策略(深度优先或宽度优先), 每次用一个实际模块
替换一个存根程序(新结合进来的模块往往又需要新的存根程序)

3. 在结合进一个模块的同时进行测试

4. 为了保证加入模块没有引进新的错误, 可能需要进行回归测试(即,
全部或部分地重复以前做过的测试)



確認問題

- 以下の説明に合う語を下の語群から選べ。
 - (1) Javaプログラムの単体テスト自動化のためのフレームワーク
 - (2) モジュール構成図の最下位モジュールからテストを行う統合テストの技法。
 - (3) モジュール構成図の最上位モジュールからテストを行う統合テストの技法。
 - (4) (2)のテストにおいて用いる仮の呼び出しモジュール。
 - (5) (3)のテストにおいて用いる仮の被呼び出しモジュール。

語群 :

JUnit Git ボトムアップテスト トップダウンテスト
ドライバ ナビゲータ スタブ

確認問題

■ 以下の説明に合う語を下の語群から選べ。

- (1) Javaプログラムの単体テスト自動化のためのフレームワーク **1.JUnit**
- (2) モジュール構成図の最下位モジュールからテストを行う統合テストの技法。 **2.ボトムアップテスト**
- (3) モジュール構成図の最上位モジュールからテストを行う統合テストの技法。 **3.トップダウンテスト**
- (4) (2)のテストにおいて用いる仮の呼び出しモジュール。**4.ドライバ**
- (5) (3)のテストにおいて用いる仮の被呼び出しモジュール。**5.スタブ**

語群 :

JUnit Git ボトムアップテスト トップダウンテスト
ドライバ ナビゲータ スタブ

講義内容

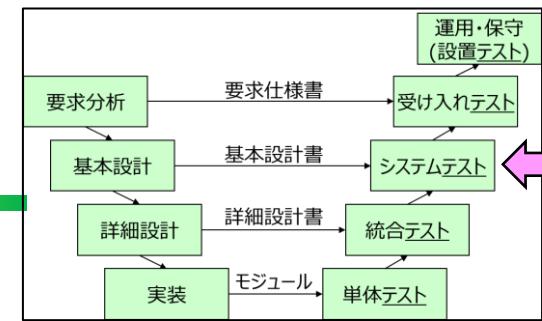
■ ソフトウェアテスト

- 単体テスト
- 統合テスト
- ➡ システムテスト
- ➡ 受け入れテスト・設置テスト

システムテスト

■ システムテスト(system test)

- システム全体の振る舞いを確認
- 実際の稼働環境にできるだけ近い環境を構築
- 基本設計仕様書に記述された機能や性能が実現できているか検査



■ 機能テスト (functional Test)

- システムの機能が正しく動作するか検査

■ 性能テスト (performance test)

- 非機能要求の評価
- 過負荷テスト、容量テスト、互換性テスト、セキュリティテスト、タイミングテスト、回復テスト、ユーザビリティテスト等

様々な性能テスト

※各性能テストの呼び方はいろいろあります

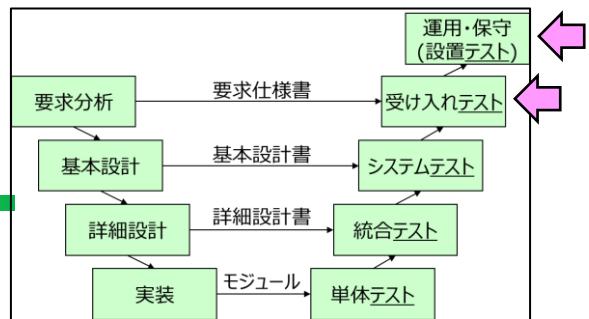
- 過負荷テスト (stress test)
 - 同時接続するユーザの人数を上限まで増やすなど、システムに負荷をかけた場合のテスト
- 容量テスト (volume test)
 - プログラムが大量のデータを扱う場合のテスト
- 構成テスト (configuration test)
 - 様々なハードウェア・ソフトウェアの構成に関してテスト
- 互換性テスト (compatibility test)
 - 他のシステムとのインターフェースに関するテスト
- セキュリティテスト (security test)
 - セキュリティ要求が満たされるかテスト
- タイミングテスト (timing test)
 - リアルタイムシステムやネットワークシステムなど、時間に関連して動作する部分をテストする

様々な性能テスト

※各性能テストの呼び方はいろいろあります

- 環境テスト (environmental test)
 - 物理的な設置場所に関するテスト(温度、電力、磁界等)
- 品質テスト (quality test)
 - 信頼性等の品質に関するテスト。システムの故障しにくさ等
- 回復テスト(recovery test)
 - システムに障害、故障が起きた際に正常に回復するかテストする
- 保守テスト(maintenance test)
 - 保守のしやすさに関するテスト。診断ツールの必要性・挙動等
- 文書化テスト (documentation test)
 - 関連文書が正確で読みやすいか
- ユーザビリティテスト(usability test)
 - UI要求に関するテスト
 - ヒューマンファクタテスト (human factors test)とも呼ぶ

受け入れテスト、設置テスト



■ 受け入れテスト

- 顧客や利用者が要求仕様書に基づき実施
- 受け入れテスト用の環境を準備
- 利用者が特定できない場合(大衆向けソフト等)、開発者が利用者の立場に立って実施

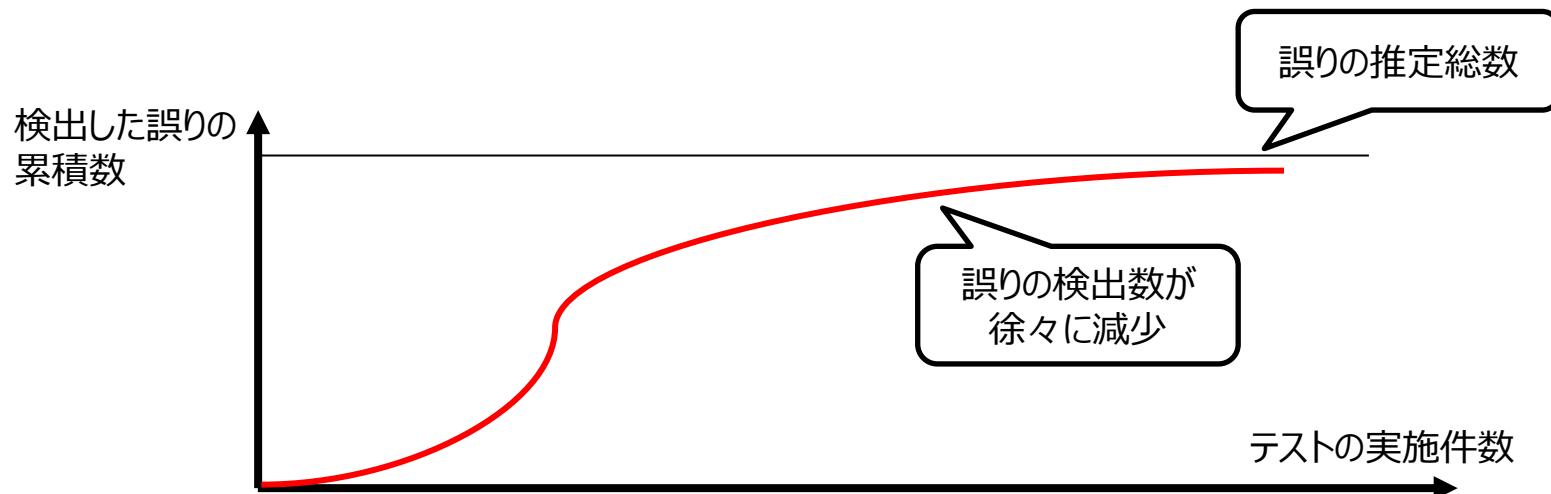
■ 設置テスト

- 実際の稼働環境にシステムをインストールして実施

信頼性成長モデル

■ 信頼性成長曲線

- テストの初期段階では多くの誤りが検出
- テストの実施件数が多くなると、誤りの累積数が収束
- 誤りの累積数が収束しない
 - モジュールが大量の誤りを含む
 - 単体テストからやり直し
- 検出される誤りが増加しない
 - テストケースが不適切な可能性大



ソフトウェア検証

■ ソフトウェア検証

- ソフトウェアに誤りが存在しないことを確認する作業
(\Leftrightarrow ソフトウェアテストでは誤りがあることを確認)
- プログラムだけではない
(仕様やその他文書に誤りがないことを確認する必要)

■ 検証の観点

- **妥当性**
 - ソフトウェアが利用者の要求を満たしているか
- **正当性**
 - 与えられた仕様に対して、プログラムが正しく実装されているか
 - 論理、代数、集合論等の数学に基づく形式的手法を用いるのが一般的
 - 部分正当性と停止性から成る
 - **部分正当性** : プログラムが停止すれば、事後条件が成り立つ
 - **停止性** : プログラムが必ず停止する

検証技法

■ 仕様の検証

- レビュー(review)

- 成果物の内容を精査
 - レビューを組織化したものにインスペクションとウォークスルーがある

- インスペクション(inspection)

- 各検査項目に関して正常な動作と照合

- ウォークスルー(walk-through)

- ソフトウェアの実行を机上でシミュレートし、正しいことを確認したり、誤りを指摘

■ プログラムの検証

- 静的型検査

- 型に関する不適切な演算や操作を検査

- 彙納表明法

- プログラムの部分正当性を証明

静的型検査

■ 型 (type)

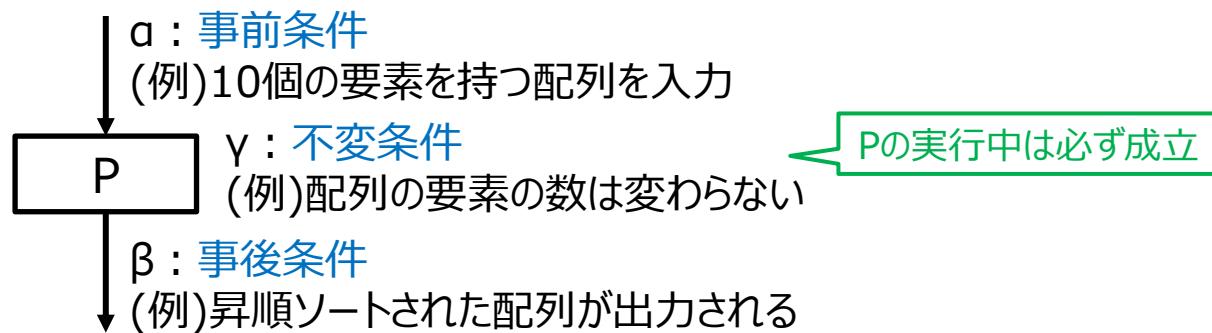
- プログラミング言語処理系では、変数や式の取りうる値を規定するもの
 - int x; 変数xの値は-2147483648から2147483647の整数
 - boolean p; 変数pの値は真(true)か偽(false)

■ 型検査 (type checking)

- 型に関して不適切な演算や操作が行われることがないプログラム
= 型安全なプログラム
 - $1+2$: 型安全である (int型とint型の加算)
 - $1+true$: 型安全でない (int型とboolean型の加算)
- 型安全でないプログラムは実行時エラーを引き起こす可能性あり
- 型推論(type inference)を利用してプログラム実行前に実行時エラーの可能性を発見

帰納表明法

- プログラムの意味を、
事前条件、不变条件、事後条件により捉える
- 条件間に成立する論理的関係を、定理証明器により
証明することでプログラムの正しさを検査



ソート(並べ替え)プログラムの検証条件例

確認問題

- 以下の説明に合うテストの名称を答えよ。
 - システム全体の振る舞いを確認するため、実際の稼働環境にできるだけ近い環境を構築して行うテスト。
 - テスト用の環境を準備し、顧客や利用者が要求仕様書に基づいて実施するテスト。
 - 実際の稼働環境にシステムをインストールして実施するテスト。
- 以下の文は正しいか。○か×で答えよ。
 - 一般的に、テストを行っても全く誤りが見つからない場合、そのソフトウェアの品質は十分に高いと考えられる。
 - ソフトウェア検証は、ソフトウェアに誤りが存在することを確認する作業である。

確認問題

■ 以下の説明に合うテストの名称を答えよ。

- システム全体の振る舞いを確認するため、実際の稼働環境にできるだけ近い環境を構築して行うテスト。**システムテスト**
- テスト用の環境を準備し、顧客や利用者が要求仕様書に基づいて実施するテスト。**受け入れテスト**
- 実際の稼働環境にシステムをインストールして実施するテスト。**設置テスト**

■ 以下の文は正しいか。○か×で答えよ。

- 一般的に、テストを行っても全く誤りが見つからない場合、そのソフトウェアの品質は十分に高いと考えられる。 **× 通常、テストの品質が悪いか、数が不足している**
- ソフトウェア検証は、ソフトウェアに誤りが存在することを確認する作業である。 **×** 誤りがないことを確認するという立場で行われる

参考文献

- 「ソフトウェア工学」
高橋直久、丸山勝久 著、森北出版、2010
- 「ソフトウェア工学 理論と実践」
シャリ・ローレンス・プリーガー 著、堀内泰輔 訳、
ピアソン・エデュケーション、2001
- 「この1冊でよくわかるソフトウェアテストの教科書」
石原一宏、田中英和 著、ソフトバンククリエイティブ、2012
- 「JUnit実践入門」
渡辺修司 著、技術評論社、2012