第3章　ホワイトボックステスト

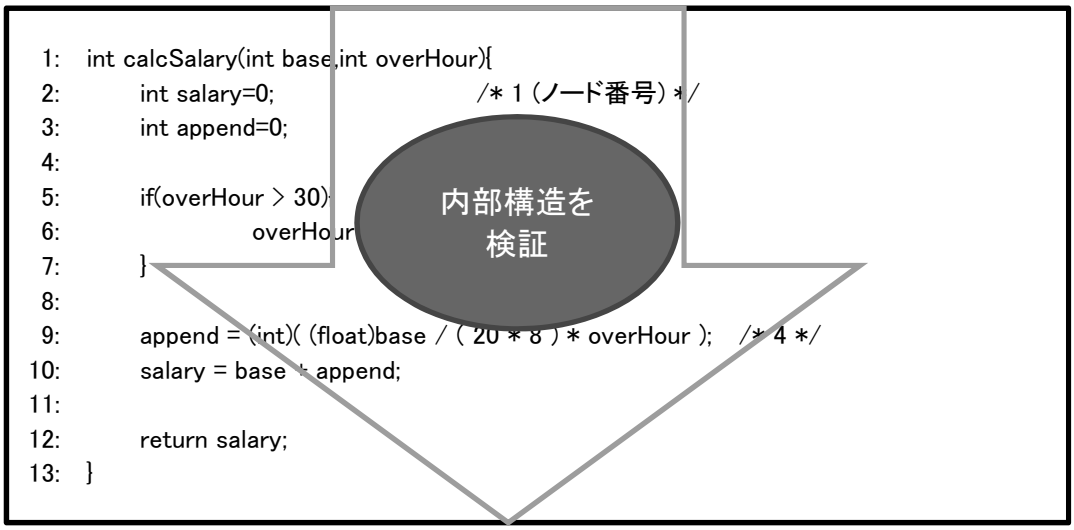
この章のねらい

この章では、ホワイトボックステストの種類と設計のポイントを紹介します。

3.1 ホワイトボックステストとは

ホワイトボックステストとは、テスト対象となるソフトウェアの内部構造に着目して、ロジックや制御の流れが正しいことを検証する手法です。通常、ホワイトボックステストでテスト設計をするためには、プログラミング言語に関する詳細な知識が必要となります。

ホワイトボックステストはランダムに実施しても効果はありません。実行していない(テストしていない)命令がないように、テストする制御パス(実行経路)を設計する必要があります。



ホワイトボックステストの技法には、以下の種類があります。

- 制御フローテスト
- データフローテスト

3.1.1　制御フローダイアグラム(CFD)

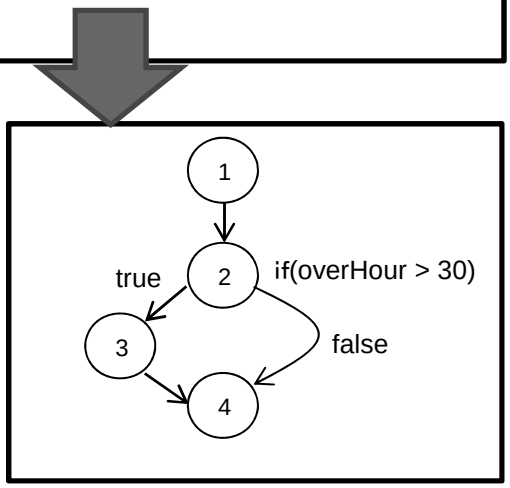
制御フローダイアグラム(CFD：Control Flow Diagram、あるいはフローグラフ)はプログラムの制御構造を図で表現したものです。制御フローダイアグラムはフローチャートとよく似ていますが、プログラムの流れを表すために記述する図なので、順番に実行する命令はブロック(ノード)としてまとめ、分岐と合流、ループを中心に描きます。

制御フローダイアグラムで使用する記号は、次のとおりです。

記号	意味
	まとめて実行する命令群および条件分岐文
	制御の流れ

例えば、以下のプログラムを制御フローダイアグラムで表すと、右下の図のようになります。

```
 1:  int calcSalary(int base,int overHour){
 2:      int salary=0;                   /* 1 (ノード番号) */
 3:      int append=0;
 4:
 5:      if(overHour > 30){              /* 2 */
 6:          overHour = 30;              /* 3 */
 7:      }
 8:
 9:      append = (int)( (float)base / ( 20 * 8 ) * overHour );   /* 4 */
10:      salary = base + append;
11:
12:      return salary;
13:  }
```



3.2 制御フローテスト

制御フローテストとは、テスト対象となるプログラムの制御の流れを検証するテスト技法です。プログラムに分岐構造や繰り返し構造があると、複数の制御パスが存在することになります。制御フローテストでは、網羅基準(カバレッジ)に基づいてテスト項目を抽出した上で、テストケースを設計します。

主な網羅基準として、以下の種類があります。

- 命令網羅(C0 網羅)
- 分岐網羅(C1 網羅)
- 全パス網羅(C∞網羅)

網羅基準によって、テスト対象となる制御パスの数は大きく変わります。

第4章　ブラックボックステスト

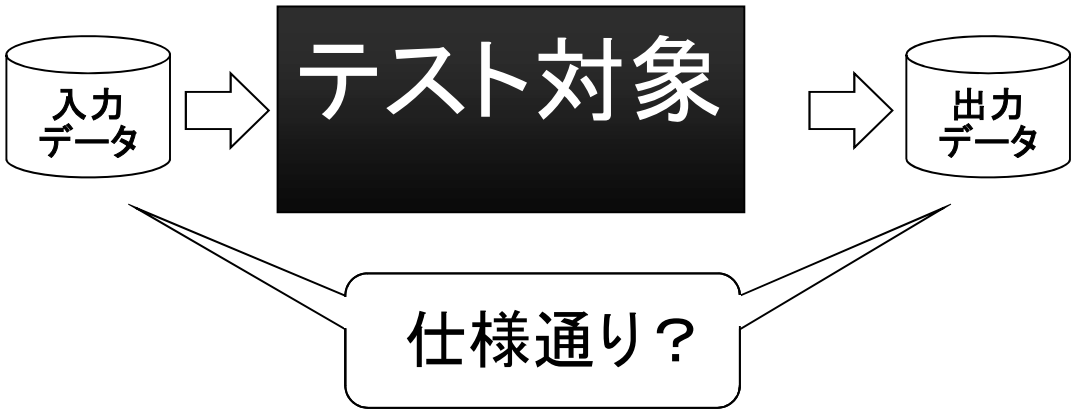
この章のねらい

この章では、ブラックボックステストの種類と設計のポイントを紹介します。

4.1 ブラックボックステストの目的

ホワイトボックステストはプログラムの内部構造を検証しますが、ホワイトボックステストでは仕様を満たすことを検証できません。仕様の検証には、ブラックボックステストを用います。

ブラックボックステストとは、プログラムをブラックボックス、つまりプログラムの内容を見ずに、入力値に対して、仕様書（または設計書）に記述された内容に合致する正しい出力値が得られるかどうかをテストする手法です。仕様書の内容に合致するとは、有効値を入力すると正しい出力値を返すことだけではなく、無効値を入力した場合に、仕様通りに無効である結果を出力することを含みます。そのためには、仕様書（または設計書）の内容を細部まで把握する必要があります。



限られたリソースの中で効率的かつ効果的なテストを実施するためには、適切なテスト技法を用いる必要があります。

ブラックボックステストの技法には、以下の種類があります。

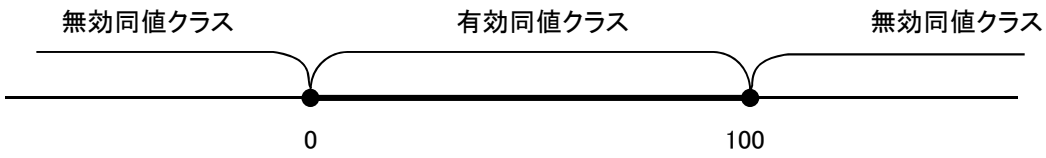
- 境界値テスト
- デシジョンテーブルテスト
- ペア構成テスト
- 状態遷移テスト
- ドメイン分析テスト
- ユースケーステスト

4.2　境界値テスト

テストにおいて、全ての入力値をテストすることは不可能です。そこで、効率よくバグを見つけるためのテストケースを検討する必要があります。そのための方法として、境界値テストがあります。

4.2.1　同値分割

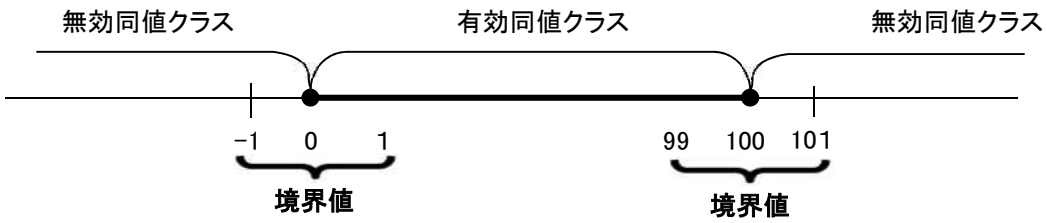
同値分割とは、同じ出力結果が得られる入力値の範囲(同値クラス)から任意のサンプルデータを抽出してテストを実施することです。たとえば、0 から 100 までの整数値が入力できる場合、0 から 100 までの全ての値をテストしようとすると、テストケース数が膨大になってしまいます。さらに、0 未満の値や 100 を超える値も全てテストすることは、もはや不可能です。同値分割では、0 から 100 までの整数値(有効同値クラス)と、0 未満の整数値(無効同値クラス)、さらに 100 を超える整数値(無効同値クラス)に分割した上で、テストケースを検討します。



4.2.2 境界値分析

境界値とは入力や出力などの境界に存在する値のことです。バグは境界に潜みやすい性格を持つため、これを使ったテストは効果的(バグの発見効率が良い)と考えられています。

入力を３つの同値類(境界内、境界外、境界上)からテストケースを選択して実施します。境界上の値の±1(整数値の場合)といったケースを追加してテストするほうが良いでしょう。うまく設計すれば、テストケースを少なくして効率的なテストを実施することが出来ますが、テストケースを少なくすることは、意図しないエラーの発見率も低くなります。たとえば、0 から 100 までの整数値が入力できる場合、「0 から 100 までの整数」がひとつの同値クラス(有効同値クラス)です。そして、この同値クラスに対する境界値は -1,0 ,1,99,100,101 です。



境界値は入力だけにあるわけではありません。出力についても境界値が存在する場合、出力の境界値もテストケースに含めます。