パスカバレッジ・テキスト

オーエスエー・リミテッド　LLC
新原　俊一

目　次

パスカバレッジ

“パスカバレッジ”という用語は、ユニットテストの中で使われます。
　従って、まずユニットテストについての知識を共有しながら何故パスカバレッジが必要なのかを理解していきます。
　次に、パスカバレッジを具体的に知ることにより他のパス網羅基準との関係を理解します。
　最後にパス網羅基準が抱える問題点を整理します。

ユニットテスト

「ソフトウェア教科書　JSTQB(注)　foundation」によるユニットテストの定義

「ユニットテストは、人のエラー(誤り)によってプログラムに埋め込まれたフォールト(欠陥)を検出し、修正することで故障が発生するリスクを減らすこと」

(注)JSTQB:日本ソフトウェアテスト資格認定委員会

－　ユニットテストの課題　－

- 課題1:どうしたら、プログラムに埋め込む人のエラー(誤り)を少なくできるか？
- 課題2:どうしたら、プログラムから多くのフォールト(欠陥)を検出し、修正できるか？
- 課題3:どうしたら、課題1，2を効率的に実現できるか？

ユニットテスト

・課題1：どうしたら、プログラムに埋め込む人のエラー（誤り）を少なくできるか？

人の誤りはなぜ発生するのか？

（A）プログラムの規模が大き過ぎる

（B）プログラムが複雑過ぎる

（C）経験的に誤りやすいパターンがある

例： 日頃単文を{、}無しで記述しているため、複文を誤って単文にするケース

（正）	（誤）

while(s[i] < 8) {
s = s * 7
i++;}
	while(s[i] < 8)
	s = s * 7;
	i++;

（D）人は思い込みで作ってしまう

例：

（正）	（誤）
if (a== 5)	if (a = 5)

ユニットテスト

どうしたら誤りを少なくできるか？

－ 原 因 －		－ 対 策 －
（A）プログラムの規模が大き過ぎる	⇨	経験的に誤りが多くなるプログラムソースの行数を調べ定量化し、ソースの行数をそれ以下とする （例） コメントや空白行を除いたソース行数は、500行以下
（B）プログラムが複雑過ぎる	⇨	経験的に誤りが多くなるプログラムソースの複雑度を調べ定量化し、ソースの複雑度をそれ以下とする ＊プログラムソースの複雑度を測る代表的な方式 ① サイクロマティック複雑度 ② モジュール（関数、メソッド）の数 ③ モジュールのネストの深さ
（C）経験的に誤りやすいパターンがある	⇨	経験的に誤りやすいパターンを調べ具体的にコーディングルールを定める
（D）人は思い込みで作ってしまう	⇨	本人は思い込んでいて分からないため、第三者がコードをチェックする（コードレビュー）

ユニットテスト

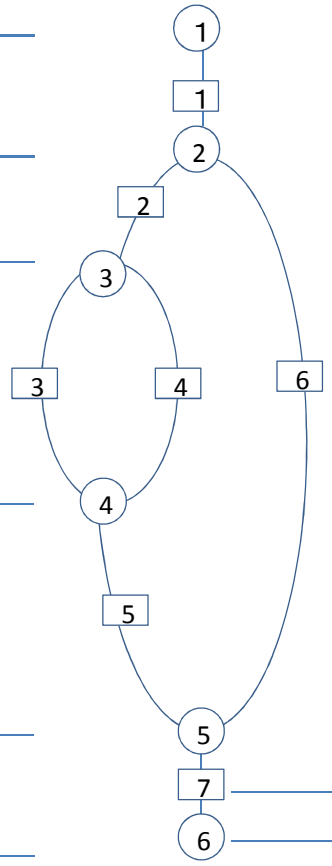
（参考）

① サイクロマティック複雑度

M＝E－N＋2P

M：サイクロマティック複雑度（Cyclomatic complexity）、E：グラフのエッジ数
N：グラフのノード数、P：連結されたコンポーネント数

```
（例） public void methodA() {
        ....
        if (条件式) {
            ....
            if (条件式) {
                ....
            } else {
                ....
            }
            ....
        } else {
            ....
        }
        ....
    }
```



サイクロマチック複雑度が
10 以下であればよい構造
30 を越える場合，構造に疑問
50 を越える場合，テストが不可能
75 を越える場合，いかなる変更も誤修正
を生む原因を作る
（McCabe‘s cyclomatic complexity）

E（基本ブロックの数）　P（コンポーネントの数）
N（分岐点の数）

サイクロマチック数＝ 7－6＋2×1 ＝ 3

ユニットテスト

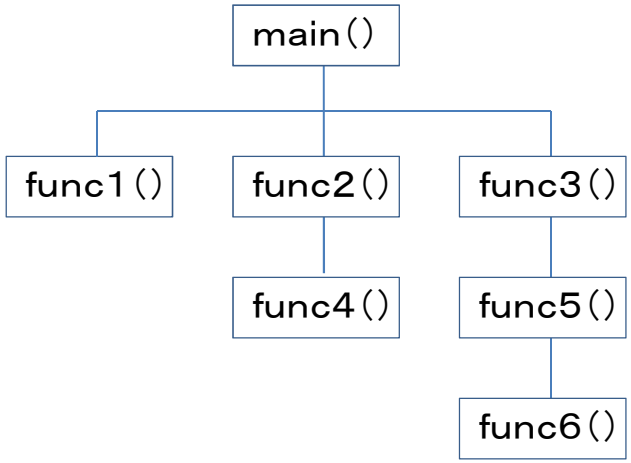
（参考）

② モジュール（関数、メソッド）の数

（例） 右図では、モジュールの数 ＝ 7

③ モジュールのネストの深さ

（例） 右図では、ネストの深さ ＝ 3



（参考）

・コーディングルール

（コーディングルールの目的：論理的なミスや欠陥に繋がる記述を減らす）

（例）「組込みソフトウェア開発向けコーディング作法ガイド[C言語版]」（IPA）

品質特性ごとに分類

＜1＞ 信頼性 ［R1．5．1］

if－else　if文は、最後にelse節を置く。

通常、else条件が発生しないことがわかっている場合は、

次のいずれかの記述とする。

（ｉ）else節には、例外発生時の処理を記述する。

（ii）else節には、プロジェクトで規定したコメントを入れる。

＊else節がないと、else節を書き忘れているのか、else節が発生しないif文なのかわからなくなる。　～

（参考）

＜2＞ 保守性 ［M3．1．1］

繰返し文では、ループを終了させるためのbreak文の使用を最大でも1つだけに留めなければならない。

＊プログラム論理が複雑にならないようにする

＜3＞ 移植性 ［P1．5．1］

＃includeのファイル指定では、絶対パスは記述しない。

＊絶対パスで記述すると、ディレクトリを変更してコンパイルするときに修正が必要となる。

＜4＞ 効率性 ［E1．1．1］

マクロ関数は、速度性能に関わる部分に閉じて使用する。

＊マクロ関数よりも関数の方が安全であり、なるべく関数を使用すべきである。

しかし、関数は呼出しと復帰の処理で速度性能が劣化する場合がある。 ～

ユニットテスト

・課題2：どうしたら、プログラムから多くのフォールト（欠陥）を検出し、修正できるか？

プログラムの欠陥とは？

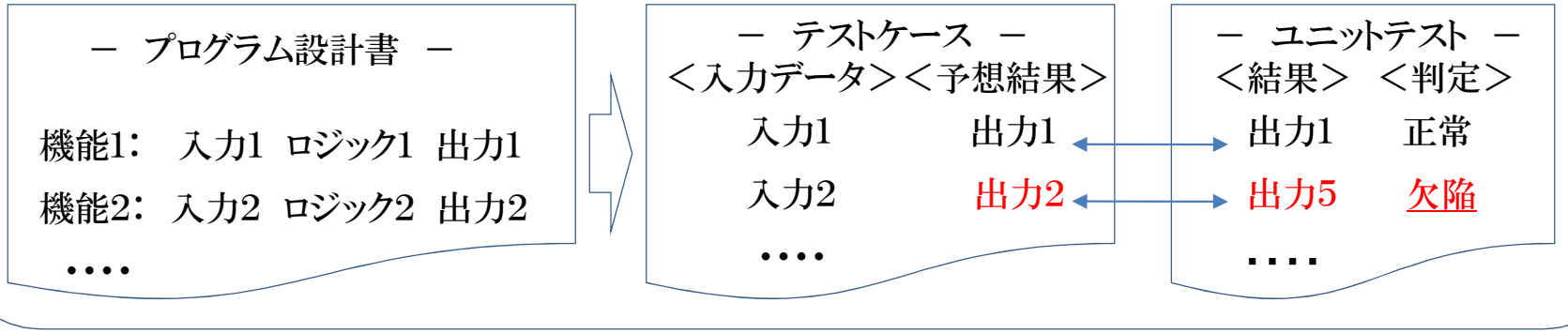
（1）プログラム設計書の機能を満足しないプログラムの欠陥

（2）想定外の動きをするプログラムの欠陥

ユニットテスト

“プログラム設計書の機能を満足しないプログラムの欠陥”を検出する方法

ブラックボックステスト：プログラムの内部構造とは無関係に、外部から見た機能を検証するテスト方法（「機能テスト」）



（参考）

＜ ブラックボックステストのテスト方式 ＞

- 同値分割法：入力データを同じ処理が行われるグループに分割し、グループ内の代表的なデータでテストを行うことで同じグループの他のデータもテストしたとみなす方式
- 境界値分析：同値分割した領域の端の値でテストする方式

（例）
- 機能1：値が3以上6以下の時処理を行う

同値グループ（3，4，5，6）

代表値5でテストを行う

境界値3、6でテストを行う

ユニットテスト

“想定外の動きをするプログラムの欠陥”を検出する方法

ホワイトボックステスト：プログラムの内部構造を理解した上で、それら一つ一つが意図した通りに動作しているかを確認するテスト方法（「構造テスト」）

「プログラムの内部構造」とは何でしょうか？

プログラムを作成する場合、まずプログラム設計書の全機能を満足するアルゴリズムを構築します。

次に、そのアルゴリズムをプログラムコードに置き換えます。

その時、プログラムの実行領域の先頭から最後に至るコードの列が複数出来上がります。

そのコードの列を“パス”といい、プログラム全体のパス（以降“全パス”）が「プログラムの内部構造」を表します。

ユニットテスト

ホワイトボックステスト(=プログラムの内部構造のテスト)には“パス”が最適です。

これを実現する理想的なテスト方式が、“パスカバレッジ”による全パステストです。

$$\text{パスカバレッジ} = \frac{\text{全実行したパス数}}{\text{全パス数}}$$

パスはプログラム中に繰返しが存在するとその数が莫大になり、現在のコンピュータ技術では処理不可能となる

ユニットテスト

では、どうやってテストするのでしょうか？

－　ホワイトボックステストにおける代表的なテスト方法　－

パスカバレッジに代わって、実行可能なパス網羅基準によるパステスト

－　パス網羅基準　－

パス網羅基準には、以下の3つの代表的な方式があります。

テストケースは各方式の網羅率を満足するよう作成します。

（1）命令網羅（C0）　命令網羅率＝実行された命令数÷全命令数

（2）分岐網羅（C1）　分岐網羅率＝実行された命令・分岐数÷全命令・分岐数

（3）条件網羅（C2）　条件網羅率＝実行された命令・分岐数（注）÷全命令・分岐数

（注）条件分岐において条件の真・偽の全ての組み合わせを含む

ユニットテスト

・課題3：どうしたら、課題1，2を効率的に実現できるか？

－　テストツールの活用　－

・静的テスト：プログラムを動作させずにテストを行う

・主な静的解析ツール

① プログラム構造分析：モジュール構造、フローチャート　他

② プログラム複雑度分析：サイクロマチック複雑度、モジュールのネスト深さ　他

③ コードチェック：ISO準拠、SECR準拠、リスクの大きいコーディングチェック　他

・動的テスト：プログラムを動作させてテストを行う

・主な動的解析ツール

① 命令網羅率計測ツール

② 分岐網羅率計測ツール

③ 条件網羅率計測ツール

パスカバレッジ

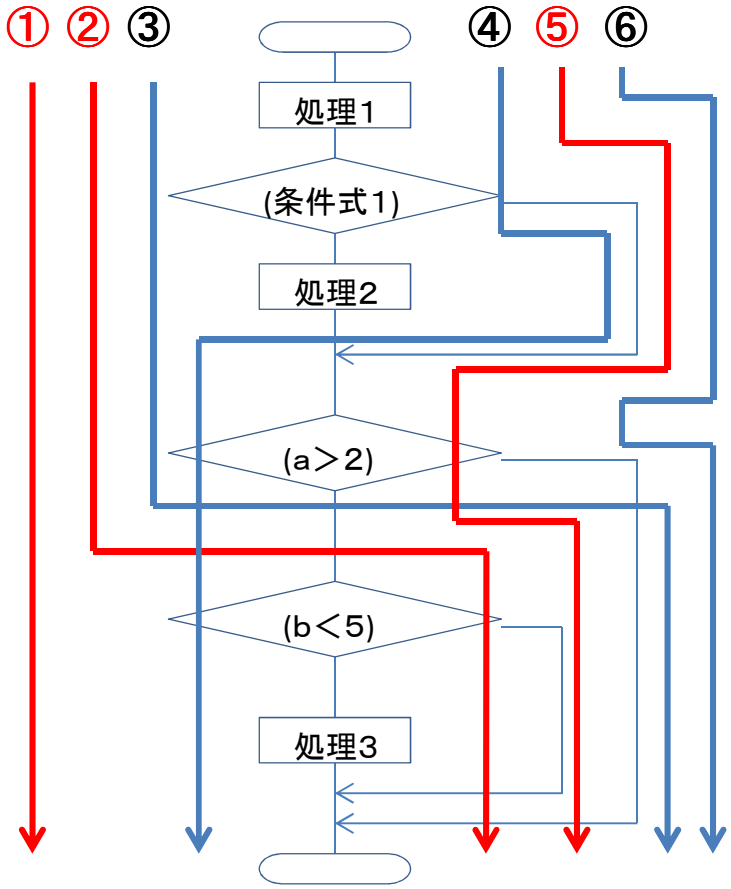
「パスはプログラム中に繰返しが存在するとその数が莫大になり、現在のコンピュータ技術では処理不可能となる」とありました

どういうことでしょうか？

パスカバレッジ

パスカバレッジ：プログラムの全パスをどのくらいテストしたかを表す指標（％）

$$\text{パスカバレッジ} = \frac{\text{全実行したパス数}}{\text{全パス数}}$$



左図には、パスが①、②、③、④、⑤、⑥の6つあります。

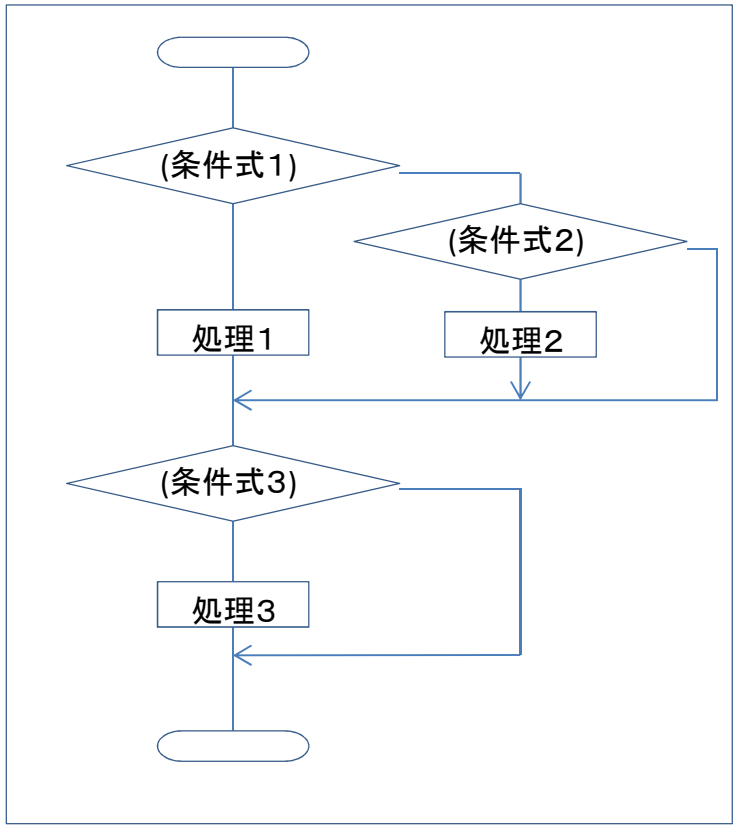
①、②、⑤を実行すると、

パスカバレッジ ＝ 3 ÷ 6 ＝ 50（％）

パスカバレッジ

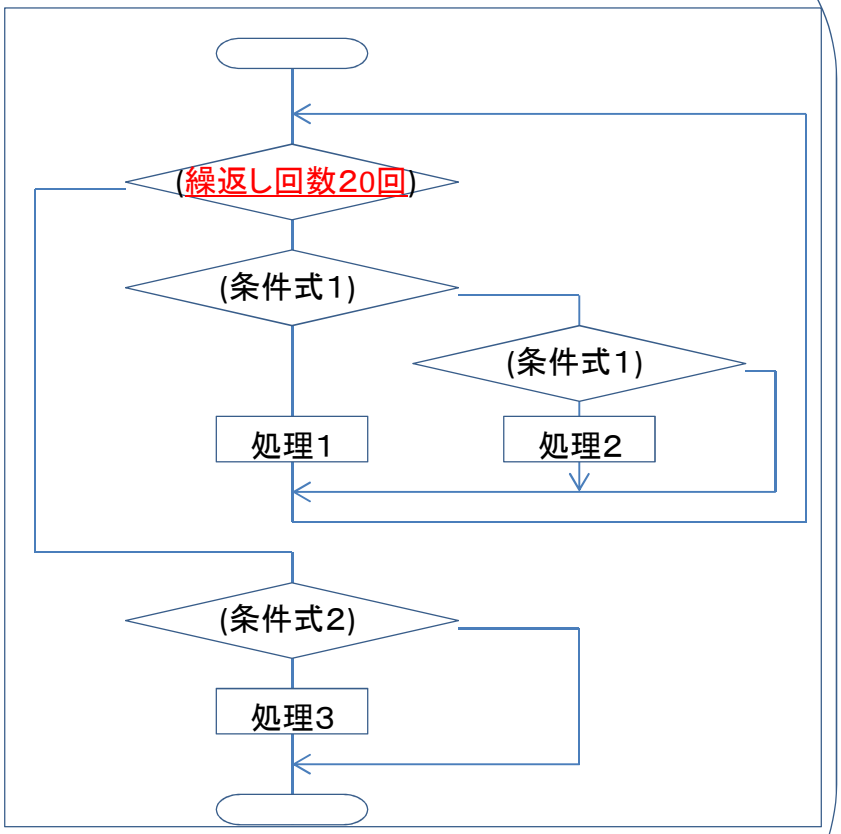
ところが、繰返しを含むプログラムの“パス”は、

・プログラム中に繰返しを含まないケース



全パス＝　3×2　＝　6

・プログラム中に繰返しを含むケース



全パス＝3の20乗×2＋2＝ 6，973，568，804
≒70億

このように、パスはプログラム中に繰返しを含むとその数は莫大なものとなり、現在のコンピュータ技術では処理できなくなります

パス網羅基準

パスカバレッジの欠点を補うのが“パス網羅基準”です。

パス網羅基準には以下の代表的な方式が有ります。

尚、パスカバレッジはパス網羅基準の中では、“経路網羅”として定義されます。

（1）命令網羅（C0）

（2）分岐網羅（C1）

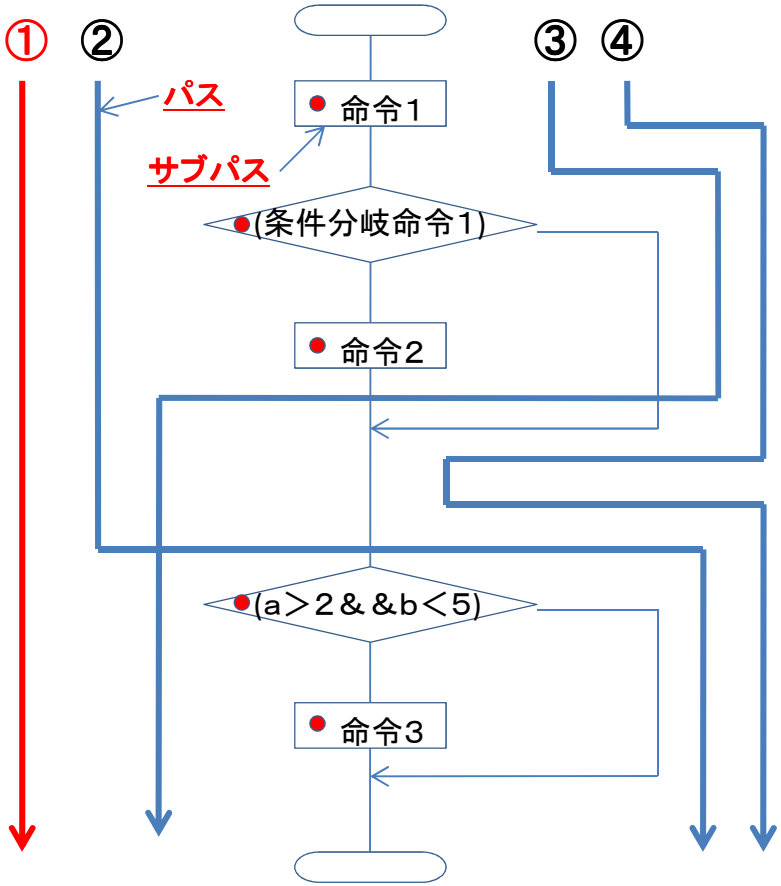
（3）条件網羅（C2）

（4）経路網羅：（経路網羅率＝パスカバレッジ）

パス網羅基準

・命令網羅の定義：ソースコード中の全ての命令を最低1回実行するサブパス(注)の網羅基準。
パス網羅基準の中では最も弱い。

$$\text{命令網羅率(=ステートメントカバレッジ)} = \frac{\text{全実行された命令数(=行数)}}{\text{全実行可能命令数(=行数)}}$$



左図には、命令網羅のサブパスが5つ、パスが4つあります。
パス①を実行すると、

命令網羅率＝5÷5＝100(％)

となり最も少ないテストケースで達成することができます。
しかし、絶対性のあるパスカバレッジで計測すると

パスカバレッジ＝1÷4＝25(％) です。

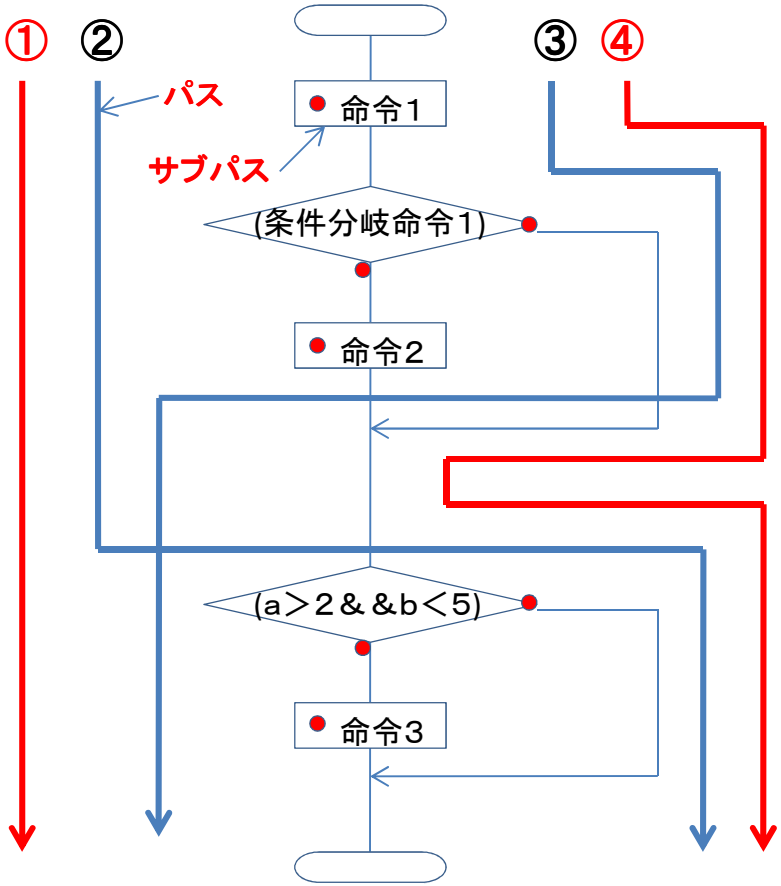
(注)「“サブパス”は、プログラムのある地点から別の地点へ至る命令の列」と定義される。サブパスもパスと呼ぶため、パスの最小単位はソースコードの一行である。(「基本から学ぶソフトウェアテスト」)

パス網羅基準

・分岐網羅の定義：ソースコード中の全ての命令と分岐を最低1回実行するサブパスの網羅基準 。

命令網羅より強力なパス網羅基準。

$$\text{分岐網羅率（=ブランチカバレッジ）} = \frac{\text{全実行された命令・分岐数}}{\text{全命令・分岐数}}$$



左図には、命令網羅のサブパスが7つ、パスが4つあります。

パス①、④を実行すると、

分岐網羅率＝7÷7＝100（％）

となり最も少ないテストケースで達成することができます。

しかし、絶対性のあるパスカバレッジで計測すると

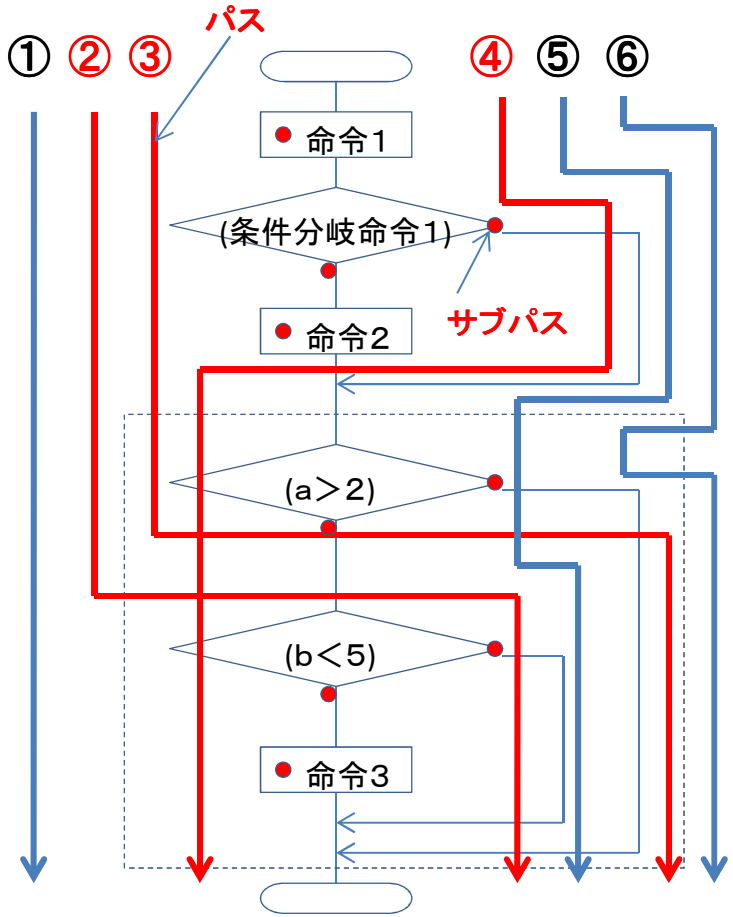
パスカバレッジ＝2÷4＝50（％） です。

パス網羅基準

・条件網羅の定義：ソースコード中の全ての命令と分岐そして、条件分岐において条件の真・偽の全ての組み合わせを最低1回実行するサブパスの網羅基準。

分岐網羅より強力なパス網羅基準。

$$\text{条件網羅率(=コンディションカバレッジ)} = \frac{\text{全実行された命令・分岐数}}{\text{全命令・分岐数}}$$



「条件の真・偽の全ての組み合わせ」とは、
・(a＞2＆＆b＜5)の例では、
(1)(a＞2)が真で(b＜5)が真　(2)(a＞2)が真で(b＜5)が偽
(3)(a＞2)が偽で(b＜5)が真　(4)(a＞2)が偽で(b＜5)が偽
の4つの組み合わせをテストすることです

左図には、条件網羅のサブパスが9つ、パスが6つあります。

パス②、③、④を実行すると、

条件網羅率＝9÷9＝100(％)

となり最も少ないテストケースで達成することができます。

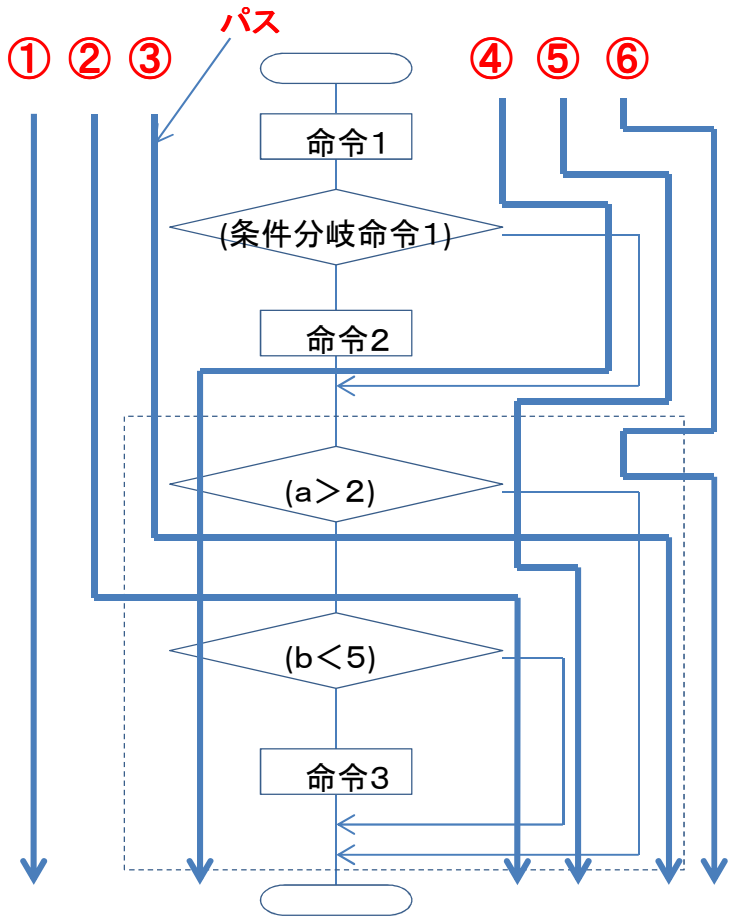
しかし、絶対性のあるパスカバレッジで計測すると

パスカバレッジ＝3÷6＝50(％)　です。

パス網羅基準

・経路網羅の定義：ソースコード中の全ての経路(=パス)を最低1回実行するサブパス(=パス)の網羅基準 。最も強力なパス網羅基準。

$$\text{経路網羅率(=パスカバレッジ)} = \frac{\text{全実行されたパス数}}{\text{全パス数}}$$



左図には、パスが6つあります。

パスカバレッジが100％になるためには、全てのパス①、②、③、④、⑤、⑥を実行する必要があります。

経路網羅率(=パスカバレッジ) ＝6÷6＝100(％)

パス網羅基準

	パスの計測精度	パスの絶対性	パスの数	生産性
命令網羅	小	無	少	高
分岐網羅		無		
条件網羅	高	無	多	低
経路網羅	完全	有	莫大	実用的でない

－　パス網羅基準の問題点　－

“絶対性が有り、実用的な指標が無い”

（参考）

「命令網羅、分岐網羅、条件網羅は実用的」とありますが、何故サブパスが実用の範囲に収まるのでしょうか？

命令網羅を例にとると、

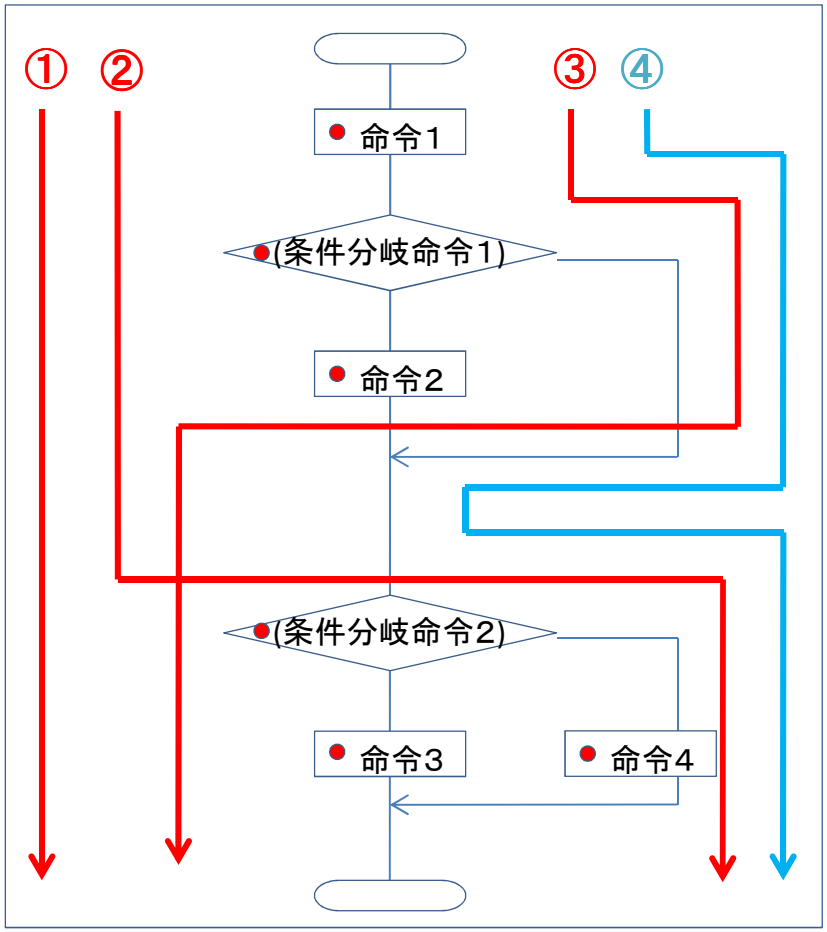
命令網羅の定義：ソースコード中の全ての命令を最低1回実行するサブパスの網羅基準

- 「ソースコード中の全ての命令」
  ：ソースコードですから繰返しを行わない状態での全命令を指し、カバレッジの分母になります。
- 「最低1回実行するサブパス」
  ：実行したパスにおいて、「同じ命令を何回繰り返しても1回としてカウントする」ということで、カバレッジの分子になります。

分母は繰返しを含まないし、分子は重複する命令を省いているので実用の範囲に収まるのです

（参考）

「絶対性が有り、実用的な指標」がなぜ必要なのでしょうか？



左図には命令が6つ、パスは4つありますが、

命令網羅率が100％になるパスの組合わせは、

①、①②、①②③、①②③④、①③、
①③④、①④、②③、②③④

9通りです。

命令網羅率が100％になるようテストした場合、

- ある時は品質が悪く
  （①②だけをテストしたケース：パスカバレッジ＝1÷4＝50％）
- ある時は品質が良い
  （①②③をテストしたケース：パスカバレッジ＝3÷4＝75％）

となります。分岐網羅、条件網羅も同様です。

これでは、ユニットテスト以降にどれだけ不具合が発生するか分かりません。

正確な納期管理を行うためには、「絶対性を持つ指標」が必要