開発環境と単体テスト

FU14 ソフトウェア工学概論

第12回

吉岡 廉太郎

前回の内容

- 設計の計測
  - ドキュメントを用いた計測で、規模と品質を測る
- コーディング規約
  - 開発者の作業品質を向上
  - 他工程や将来の拡張を見据えた効率化
- プログラミングの指針
  - わかりやすく、品質の高いコードを開発するためのガイドライン
- ドキュメント
  - コードを理解しやすくする内部ドキュメント
  - プログラムのことを記述する外部ドキュメント

今日の内容

- 開発環境とプロセス
  - プログラムを実行する環境を整える
- テスト手法
  - プログラムの動作を確認する観点
- 単体テスト
  - 最初に行う基本的な確認

構成管理

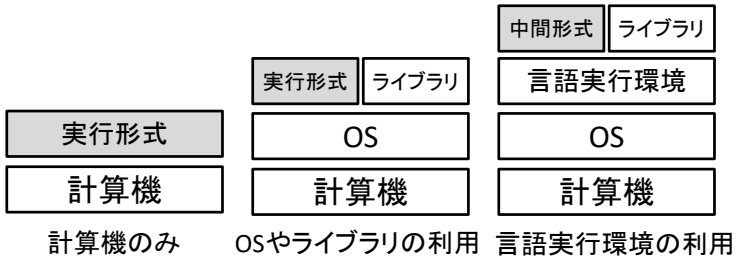
- バージョン、リリース
  - プラットフォーム別、機能別、など
- 運用システムと開発システム
  - テスト済みの安定した運用システム
  - 機能追加を行う開発システム
- 変更管理
  - テスト中発生する変更を管理する
  - 別ファイル、差分、条件付きコンパイル

Git, Subversion, CVSが一般的。
github, bitbucketなどのサービスやgitlab, gitbucketを使うのが主流。

6.1.4

ソフトウェアと実行環境

- 実行環境
  - ソフトウェアの動作に必要な環境
  - 対象の計算機やソフトウェアの内容によって環境は変わる
  - バージョンなど依存性もある



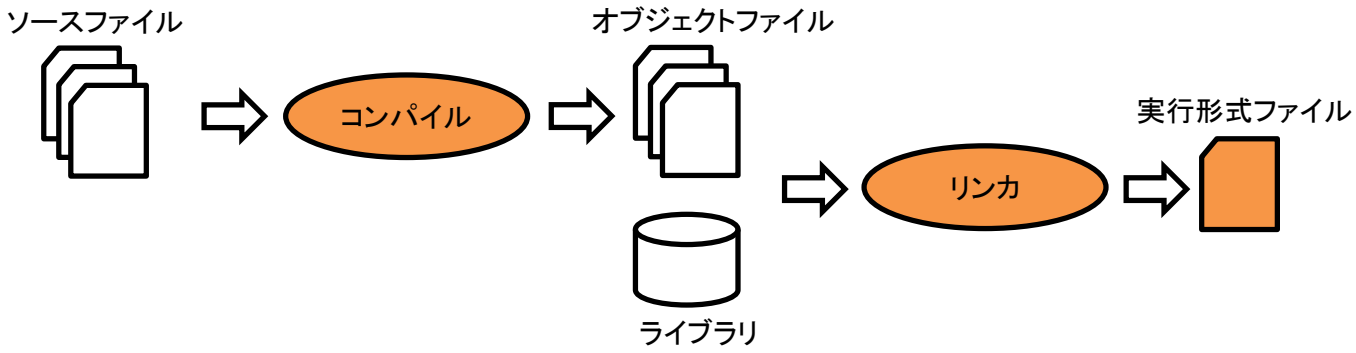
- 実行環境によるソフトウェアの種類
  - アプリケーションフレームワークの利用
  - 仮想化環境（クラウド）
  - 組み込みソフトウェア

- ソフトウェアの配置
  - 実行形式に変換されたソフトウェアを実行環境に設置して利用できるようにすること

6.2.2

ビルド作業と依存関係

- ビルド作業
  - ソフトウェアの定義を実行可能なソフトウェアへと変換する作業
- 依存関係と自動化ツール
  - ビルドには、ソースファイル、オブジェクトファイル、実行形式ファイル、ライブラリ、設定ファイルなどが関係する
  - 依存関係を管理し、適宜必要なファイルを（部分的に）更新する必要がある
- 自動化ツール
  - 管理するファイルやビルド作業の内容をあらかじめ定義しておくと、ファイルが更新されるたびに自動的にビルドする
  - 一般的なツールの例：Make, Ant, Maven, Gradle



6.2.4

バージョン管理

- バージョン
  - 実行可能なソフトウェアの単位（リリース）
  - 時系列で順序づけることができる
  - 目的別に分岐することもある
  - バージョン間の関係はツリーになる（バージョン木）

リビジョンとも呼ばれる

- バージョンの再現
  - バージョンを構成する様々な成果物や情報を“差分“で管理し、任意のバージョンをいつでも作り出すこと
  - 用途：変更、拡張、不具合の修正、目的別ソフトウェア

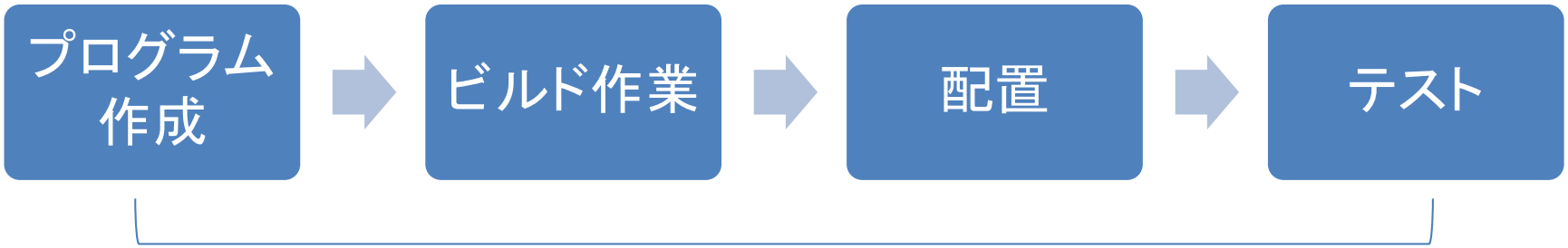
- 排他制御
  - 複数人での作業を管理する手法
  - ロック方式とマージ方式がある

ツールとしては、GIT, SVN, CVSなどが一般的

6.2.5

継続的インテグレーション

- プログラム作成以降の作業を結合する
  - プログラム作成とそのテスト作業を繰り返し頻繁に行う方法
  - 専用のツール等で自動化するのが一般的
- 効果
  - プログラムの作成や修正をより気軽に行うことができる
  - 早期に問題発見や解決ができる
  - 動作するソフトウェアを確実に手に入れることができる



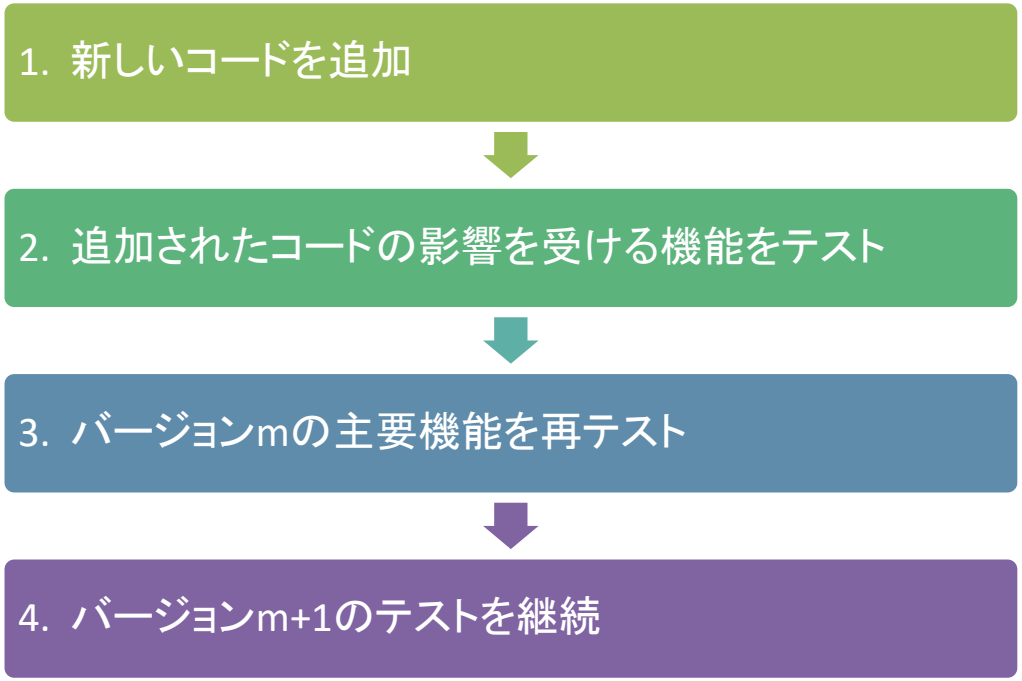
これらの手順を結合し、プログラムが常に動作する環境を構築する

回帰テスト

- 修正に伴う欠陥の発生を確認する
  - テストで発見した欠陥は修正する
  - 修正が新たな欠陥を発生させる
- 新しいバージョンやリリースが以前のものと同機能であることを検証する
  - 機能（コード）の追加に伴う欠陥の発生を発見する

回帰テストの手順

- 前提：バージョン m での機能テストは正常



ツールを使った自動テスト環境を構築するのが容易になってきている

テスト手法

- コンポーネントの動作を確認する方法
  - 入力を与え、出力を確認する

ブラックボックス

- テスト対象の機能に着目
- 外見上の振る舞い

ホワイトボックス

- テスト対象の構造に着目
- ブラックボックスに状態と手続きを追加

ブラックボックス・テスト

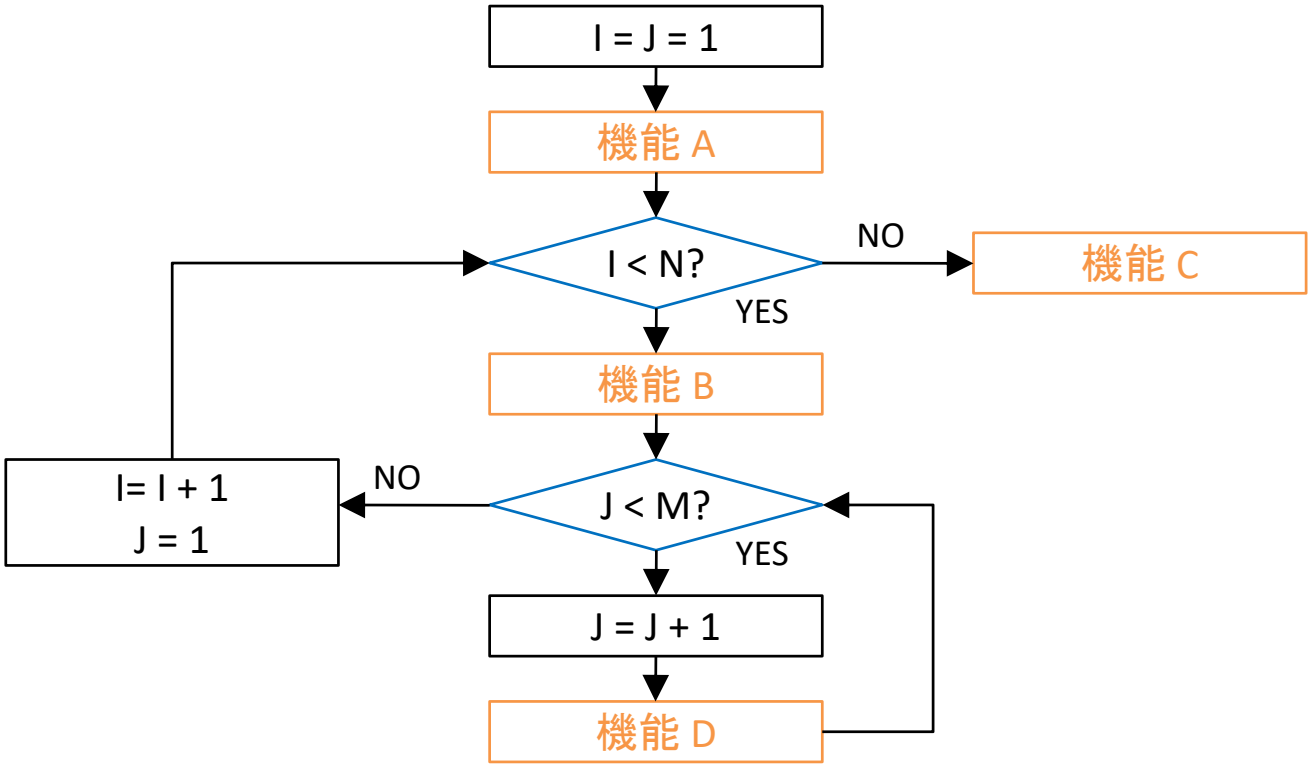
- 入力に対する出力だけを確認する

- 利点
  - 内部的構造や処理手順を考慮する必要がない

- 欠点
  - 完全なテストが不可能
    - 可能な全ての入出力の組み合わせを試すのは無理
  - 代表的な入出力の組み合わせを網羅する
    - 網羅できるかどうかが不確実

ホワイトボックス・テスト

- 内部構造に則したテストが可能
  - 例）すべての論理パスのテスト



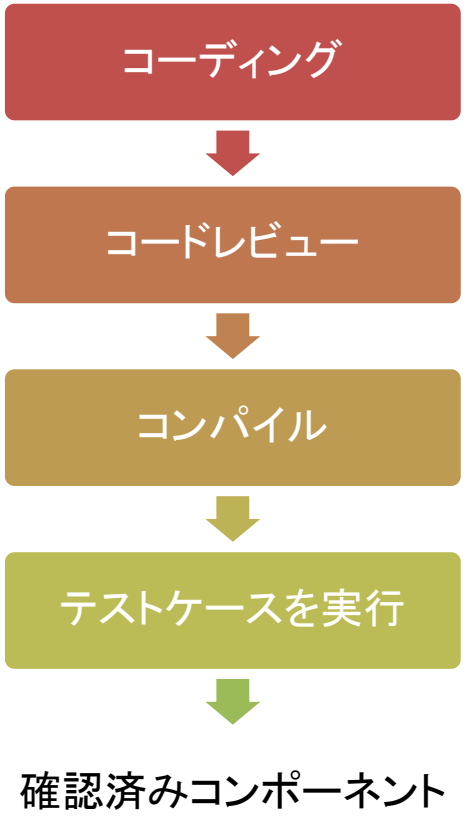
選択の規準

- コンポーネントごとに“？ボックス”を選択する
  - 適切な手法の選択が重要
- 選択時の検討事項
  - 論理パスの数
  - 入力データの性質
  - 計算量
  - アルゴリズムの複雑度

単体テスト

- コーディング時に行う最初の確認
  - コーディング単位（単体）ごと

- 単体テストの手順
  - ① コードレビュー
  - ② コンパイル
  - ③ テストケースを実行する



7.4.2

①コードレビュー

- 設計に照らし、コードが適切か確認する
- コード・ウォークスルー
  - レビューチームにコードを説明し、質問に答える形式
- コード検査
  - あらかじめ準備されたチェック項目にそってレビューチームがコードとデータを確認する

コードレビューの効果

発見手法	1000行当たりの欠陥発見数
要求レビュー	2．5
設計レビュー	5．0
コードレビュー	10．0
システムテスト	3．0
受入テスト	2．0

③テストケース

- テストの手順
  - 入力と動作条件を決める
  - コンポーネントを実行する
  - 出力を確認する
- テストケースとは
  - 特定のコンポーネントをテストするための入力データ
- あらかじめ決めた数のテストケースを実行する

テストケースの作成手順

- ①テストの目的を決定する
  - ブラックボックスかホワイトボックスを検討
- ②テストケースを選択する
  - ブラックボックス：考えられるすべての入力
  - ホワイトボックス：処理内容を考慮した選択
    - 通常値に加え、境界値や特異点を含める
  - エラーとなるテストケースも意図的に含める
- ③テストを記述する

7.2.5

ブラックボックステストの技法

同値分割法	• 入力値を同値クラス（同値領域）に分割し、各同値クラスの代表値を最低1回実行するようにテストケースを構成する
境界値分析	• 領域の境界値とその近傍で領域に含まれない値を実行するようにテストケースを構成する
決定表テスト	• 入力や条件の組合せと結果との対応関係にもとづいてすべての規則を網羅するようにテストケースを構成する
状態遷移テスト	• ステートマシン図などの状態遷移表にもとづいてテストケースを構成する
ユースケーステスト	• ユースケース記述の事前条件、事後条件、基本系列、代替系列にもとづいてテストケースを構成する

7.2.6

ホワイトボックステストの技法

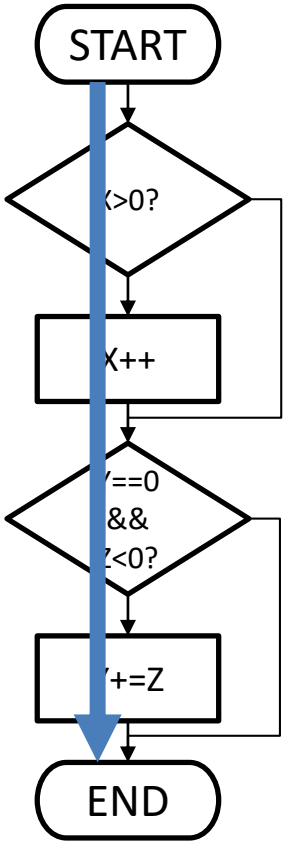
① 制御フローテスト

- プログラムの制御構造(制御フロー)上の命令や分岐を対象に、実行のパスを抽出してテストケースを構成する
- カバレッジ基準:テストの対象となる要素
  - 命令、分岐、条件など
- カバレッジ率を100%に近づける
  - (実行したアイテム要素の数)／(アイテムの総数)

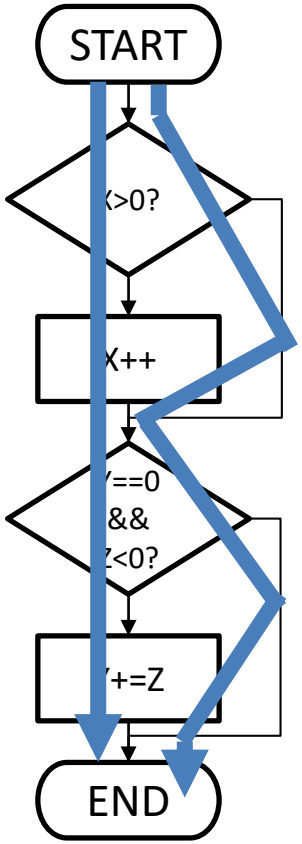
② データフローテスト

- プログラム中の変数の利用の流れを確認するようにテストケースを構成する
- 変数の宣言、利用、消滅をカバレッジ基準とするのと同じ

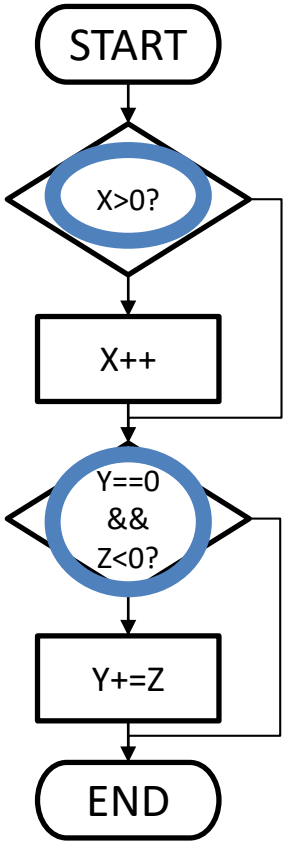
カバレッジ基準による違いの例



命令網羅



分岐網羅



① x>0
② x<=0

① y==0かつz<0
② y==0かつz>=0
③ y!=0かつz<0
④ y!=0かつz>=0

複合条件網羅

欠陥発見手法の比較

- 発生元別に見た欠陥の発見率

発見手法	要求	設計	コーディング	ドキュメント
プロトタイピング	40	35	35	15
要求レビュー	40	15	0	5
設計レビュー	15	55	0	15
コードレビュー	20	40	65	25
単体テスト	1	5	20	0

本日のまとめ

- 開発環境
  - 開発とテストを繰り返すための環境
- テスト手法
  - 目的に応じた分類：ブラック、ホワイト
- 単体テスト
  - コードレビュー
  - テストケースとカバレッジ

次回講義の事前学習：
7.1.3, 7.2.3, 7.2.4, 7.3.1