ソフトウェア工学 第15回
― まとめ ―

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

授業計画 (前半)

- 第1～7回
  - ソフトウェア工学概説
  - ソフトウェア開発モデル
  - 要求分析
  - 構造化分析
  - オブジェクト指向分析
  - アーキテクチャ設計、
    ユーザインタフェース設計
- 第8回 中間試験および解説

授業計画 (後半)

- 第9～14回
  - モジュール設計
  - プログラミング
  - ソフトウェアテスト
  - ソフトウェア保守と再利用
  - ソフトウェア開発管理
- 第15回 まとめ

ソフトウェア (software)

- 情報処理システムのプログラム、手続き、規則、および関連文書の全体または一部分 (JIS X0001)
  - プログラム
  - 要求定義書、外部設計書、内部設計書、データベース定義書、コーディング規約、取扱説明書、…
- cf. ハードウェア (hardware)
  - コンピュータの装置

応用ソフトウェア (application -)	メーラ、ブラウザ、ワードプロセッサ等
ミドルウェア (middleware)	データベース管理ソフト等
基本ソフトウェア (basic software)	オペレーティングシステム等

ソフトウェアとハードウェア

- ソフトウェア
  - 経年変化なし
  - 導入後に修正可能
  - 量産、配布・流通コストが低い

機能拡張、
性能改善等が
求められる

- ハードウェア
  - 経年変化あり(摩耗、部品の寿命)
  - 導入後の修正はほぼ不可能
  - 製品の量産および配布コストが高い

機能や性能の
維持が
求められる

ソフトウェア工学 (software engineering)

- コンピュータソフトウェアを対象として、その構築、運用、保守における生産性と品質の向上を実現するための技術体系や学問体系
  - 方法論(methodology)
  - 技法(technique) / 道具(tool)
  - プロジェクト管理(project management)

- ソフトウェア工学の目的
  - 良いソフトウェアを開発すること
    → ソフトウェアの品質特性(国際規格ISO9126)

ソフトウェアの品質特性 (ISO9126)

1.機能性(functionality)：
　　必要な機能が実装されているか
2.信頼性(reliability)：
　　機能が正常に動作し続けるか
3.使用性(usability)：
　　利用者にとって使いやすいか
4.効率性(efficiency)：
　　目的達成のために使用する資源は適切か
5.保守性(maintainability)：
　　改訂作業に必要な労力は少ないか
6.移植性(portability)：
　　他の環境へ移しやすいか

問題解決への取り組み

- 分割統治と構造化
- 抽象化とモデリング
- 要求分析
- 追跡可能性
- 経験の蓄積と再利用
- 系統的な評価と管理

分割統治と構造化

- 分割統治 (divide and conquer)
  - 大きな問題を独立性の高い小さな問題に分割する
  - 小さな問題をそれぞれ解く
  - 解を取りまとめて、本来の大きな問題の解を得る

- 分割統治のための手法 (例)
  - ソフトウェアの構造化：
    大きな問題をたくさんの小さな要素に分割した後に、
    それらの要素の間の関係を分かりやすく整理すること
    - 絡み合って複雑な問題も、細かく分けて考えれば
      構造を理解できる
    - ソフトウェアを“部品”に分割する
    - 構造化分析、構造化プログラミング等

抽象化とモデリング

- 抽象化 (abstraction)
  - 対象とする事物から本質的でない部分を取り除くこと
  - 検討すべき項目に関連する重要なものを抜き出して記述する（記述したものをモデルと呼ぶ）
- モデリング（modeling）
  - ソフトウェアに関する様々な項目に対して、それらの本質を浮かび上がらせるように抽出する
  - 大きく複雑なシステムであっても、詳細にとらわれずに本質的な部分に着目できる
  - 複数のモデルの存在
    - 項目ごとに別のモデルがある
    - 同じ項目でも異なる視点からモデリングできる

追跡可能性 (traceability)

- ソフトウェア開発では数多くの「判断」がなされる
  - 例えば、要求仕様や設計仕様
  - 一部は成果物(artifact)として文書化される
  - 開発プロセスに応じて、様々な成果物がある

  →「どのような判断をしたのか」
  「なぜその方法・方式を選択したのか」等を残す必要がある
- 追跡可能性
  - ある成果物から他の成果物の対応箇所や、
    大元の判断を辿ることができる性質
  - 例) プログラムの不具合を発見したとき、原因となった設計仕様書の記述箇所、要求仕様書の記述箇所、要求の理由を辿れる
  - 例) 要求仕様書を変更したとき、その要求によって実現された設計仕様書の機能や性能、プログラムの記述を辿れる
  - 利用者・分析者・設計者・プログラマが判断した
    「理由」まで辿れると良い

経験の蓄積と再利用

- 開発経験(成功、失敗)の蓄積
  - 共通化して蓄積、集約して「次の開発」に再利用
  - 方法論やツールが開発される
    → 様々なプロジェクトや企業で使われる
    → 経験として蓄積され、さらに改良され、
    さらに広く使われる
- ソフトウェア開発において
「経験的に良いと確認された方法」を
標準化(standardize)して誰もが使えるようにする
- SWEBOK (Software Engineering Body of Knowledge)
  - ソフトウェア工学基礎知識体系
  - ソフトウェア工学に関連する知識を体系的にまとめたもの
  - 最近の研究の成果を反映して、2014年にver.3策定

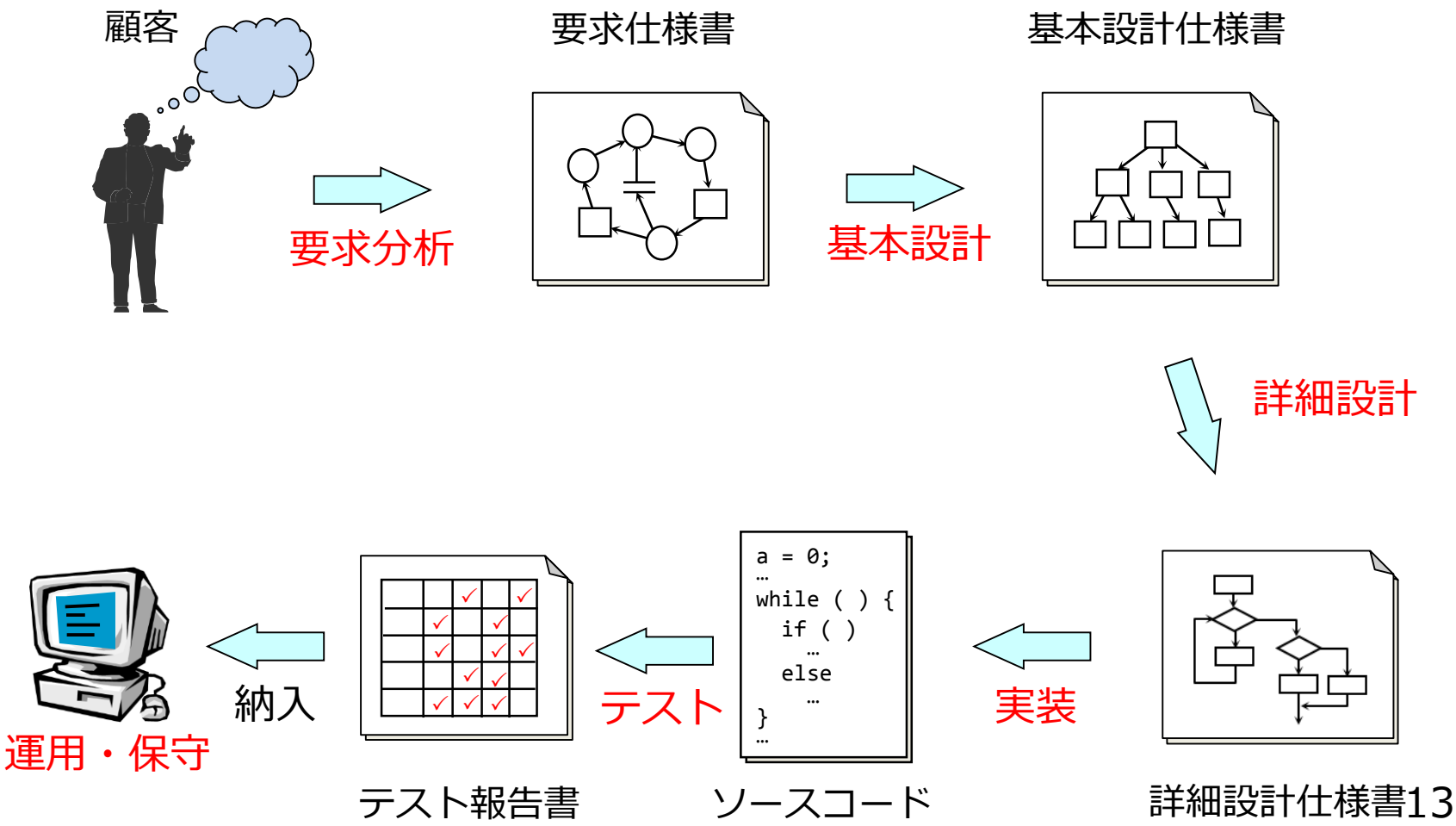
ソフトウェア開発モデル

ここではソフトウェア開発(の方法や手順)

- モデル：検討すべき項目に関連する重要なものを抜き出して記述したもの
- ソフトウェア開発モデル：
  どのようにソフトウェア開発を進めるかを
  モデル化したもの
  - 複数の開発プロセス(process, 過程)から構成される
  - 各プロセスでは、作業の成果をまとめたもの
    ＝成果物(artifact, 教科書ではproduct)を作る
- 特に、どのようなプロセスで開発するかをモデル化したものを、ソフトウェアプロセスモデル(software process model)(または ライフサイクルモデル(lifecycle model)) と呼ぶ

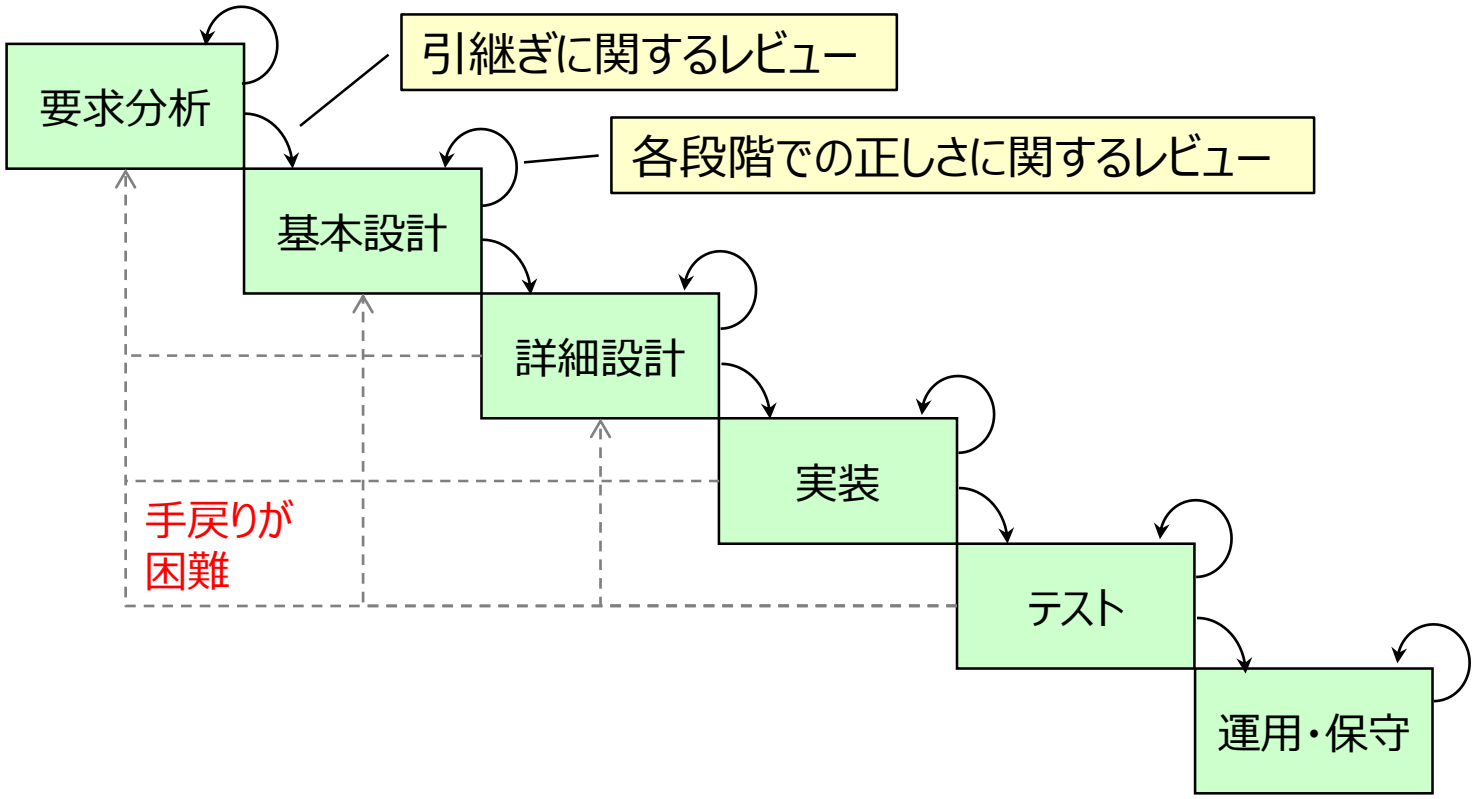
ソフトウェア開発プロセス

ソフトウェア開発の作業工程



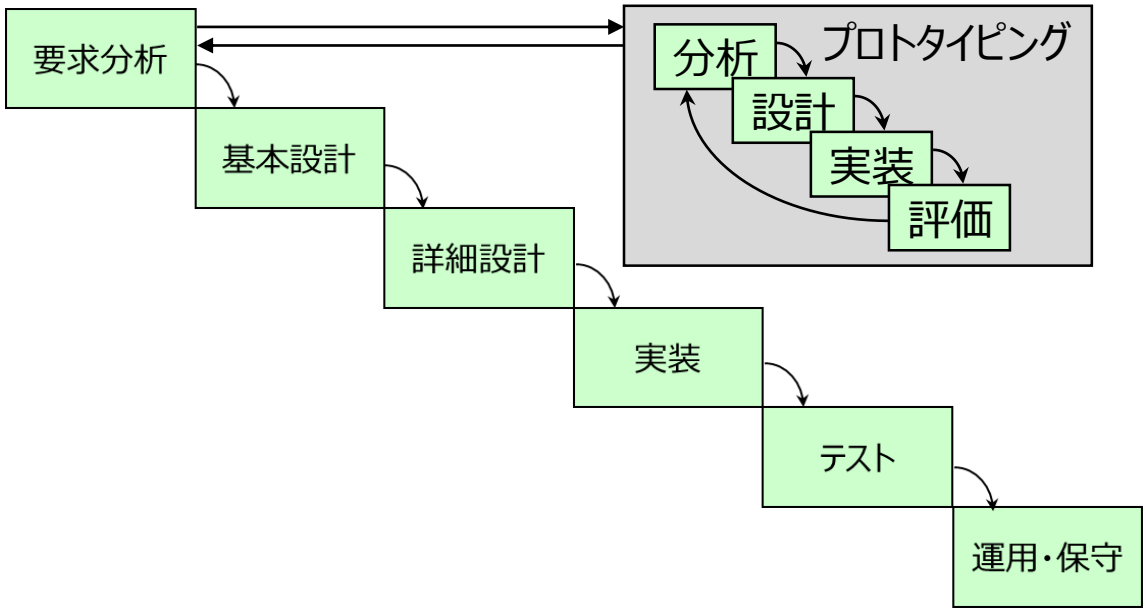
ウォーターフォールモデル

- 滝(waterfall)に例えられるプロセスモデル
- トップダウンな開発プロセス
  - 要求分析から運用・保守まで一直線に進む
  - 工程の後戻り(手戻り)は原則、想定しない



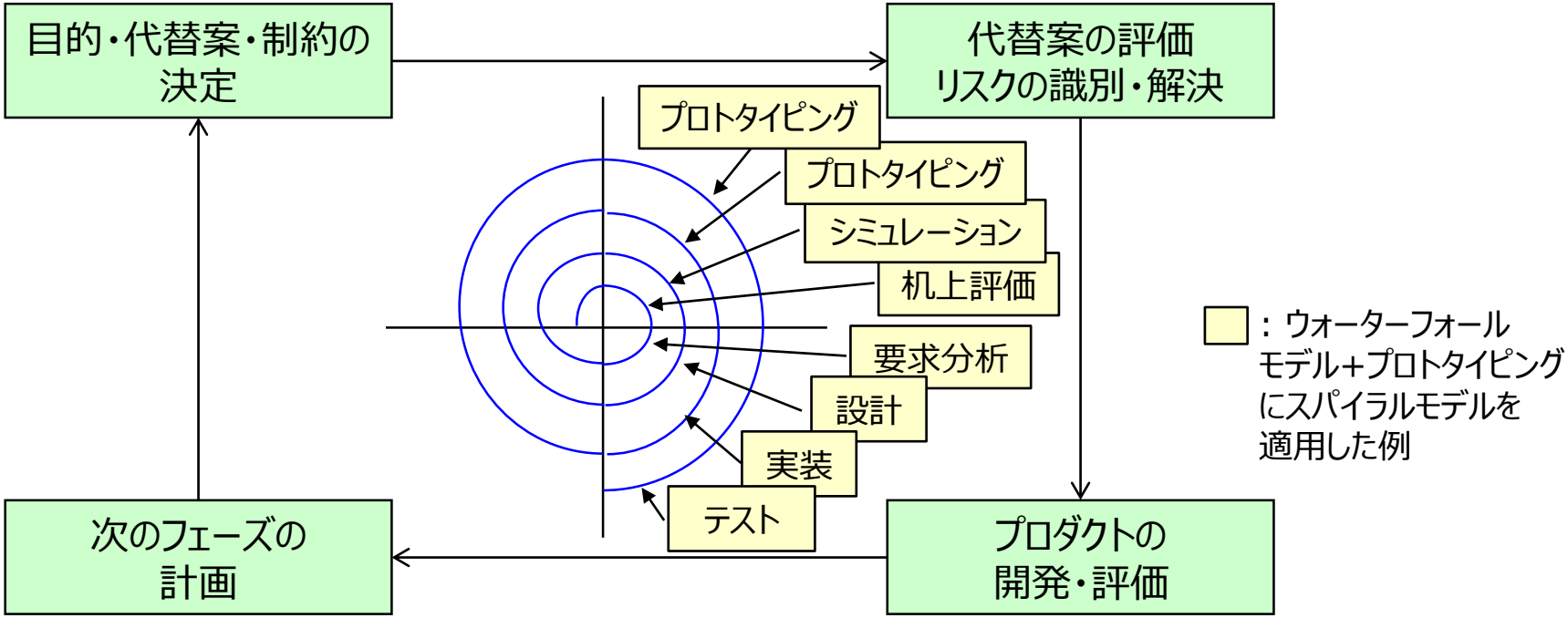
使い捨てプロトタイピング

- プロトタイプ(prototype)：品質確認等のために作る試作品
- プロトタイピング(prototyping)：
  実際にソフトウェアを構築する前に試作品を作り、
  問題点を明らかにする手法
  - UI等の確認
  - 素早く作って、素早くフィードバック
- 使い捨てプロトタイピング：
  - 開発の初期段階で作成したプロトタイプをその後使わない



スパイラルモデル

- 下の4フェーズを繰り返して、段階的に開発を進める



- スパイラルモデル自体はプロセスモデルではなく、他のプロセスモデルと組み合わせて用いる(既存のプロセスモデルを改善する)

進化型プロトタイピング

- プロトタイプに修正を加えていき、最終的なソフトウェアとするプロトタイピング手法
  - 機能が明確な部分から開発
  - すべての仕様を確認しながら作るため、大幅な手戻りを予防できる
  - 分析・設計・実装を繰り返す
    - iterative (反復的)
      - すべての部分を一度完成させ、プロトタイプとして提供
      - リリースごとに各部の完成度を高めていく
    - incremental (漸進的)
      - ソフトウェアを独立性の高い部分に分割
      - リリースごとに機能を追加する
    - 両者を組み合わせて使うのが普通

アジャイルプロセスモデル

agile [Adj]機敏な

- 変化に迅速に対応することを目指す
- 開発対象を小さな機能に分割し、各機能を迅速に開発
- 実行可能なソフトウェアを随時提供する
- 進化型プロトタイピングを突き詰めたようなもの

- 単なる開発手順というよりは、
  開発の手順や方法を考えるための枠組み
- 代表的なアジャイルソフトウェア開発手法
  - エクストリーム・プログラミング (XP: Extreme Programming)
  - スクラム (SCRUM)

要求分析

- 顧客や利用者の要求を明らかにし、仕様化する作業
  - 正確かつ厳密に定義することが重要
    - 設計工程や実装工程において誤ったソフトウェアを作成
    - 設計作業や実装作業のやり直しが発生
  - 要求定義 (Requirements definition)とも呼ぶ
- 要求 (requirements)
  - ソフトウェアを利用して実現したい内容
  - 利用する側から見たソフトウェアが実現すべき目標
    - 機能要求： 開発するシステムが何をするか
    - 非機能要求： 機能要求以外の要求。機能の実現に関する制約を表す (性能、使いやすさ、安全性、保守性、可搬性など)
- 仕様 (specification)
  - 要求を実現するために必要な機能や性能を提供する際の条件や制約
  - 要求を満たすためにソフトウェアが実現しなければならない要件

明確に機能/非機能に分けられるとは限らない

要求分析の作業

- 要求獲得 (Requirements elicitation)
  - 利用者から要求を引き出す
  - 要求定義の初めに行われる
  - 「要求記述」を作成
- 要求記述 (要求仕様化) (Requirements description/specification)
  - 「要求仕様」を実際に作成する
- 要求確認 (Requirements validation/verification/evaluation)
  - 作成された要求仕様の正しさを検査
  - 要求仕様のレビュー(review)

要求抽出の技法

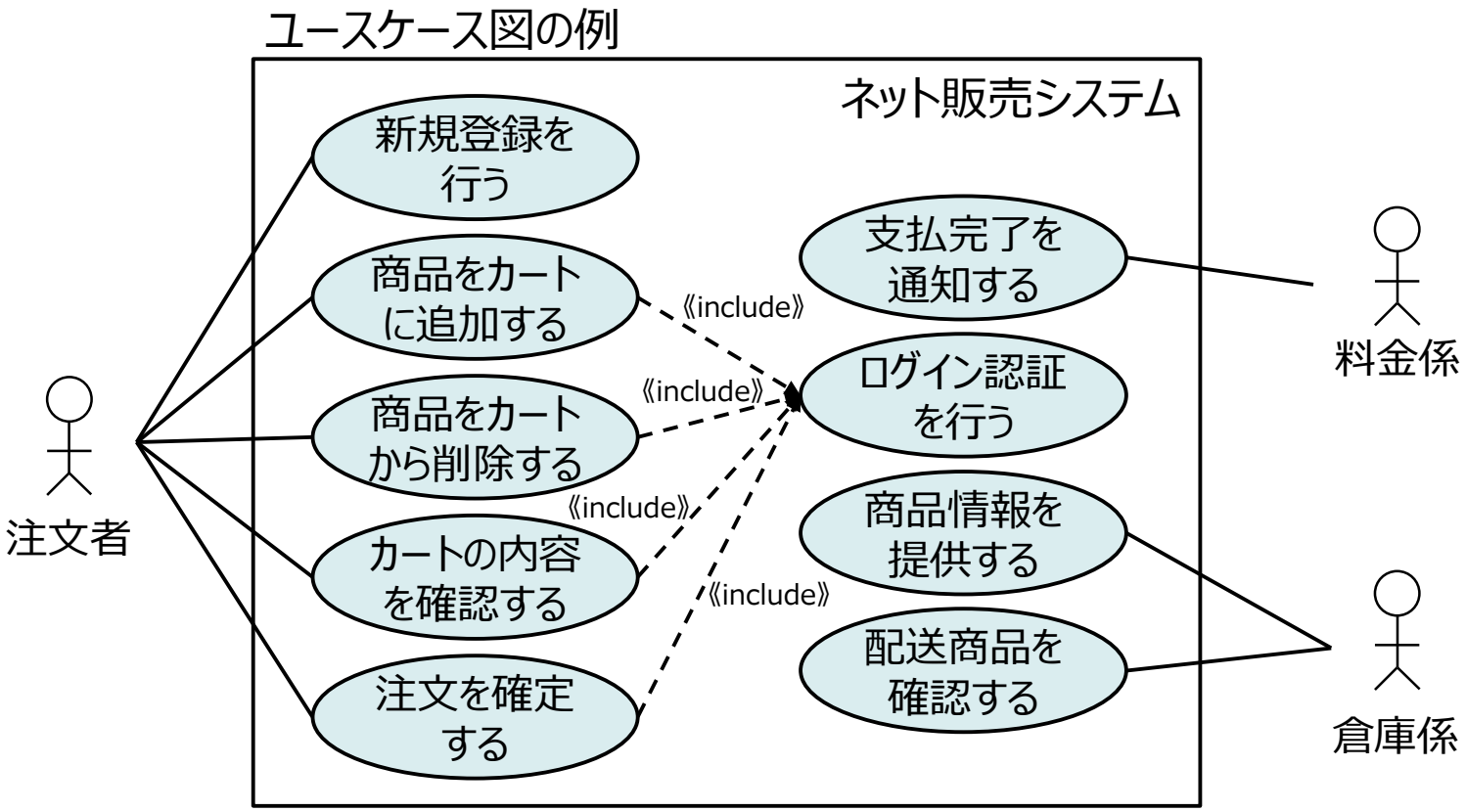
資料収集	業務やシステムに関する資料の収集
インタビュー	顧客や利用者への質疑
アンケート	質問事項に対する回答
ブレーンストーミング	自由な意見交換と意見整理
現場観察	業務の体験や観察
プロトタイピング	試作品の提供
シナリオ	状況ごとの利用例
ユースケース	典型的な利用事例
ゴール指向分析	ゴール(目標)の明確化と分割
問題フレーム (problem frame)	機械、世界(問題領域)、要求という捉え方により問題をパターン化

シナリオ

- ある状況に限定したシステムの具体的な利用例
- 利点
  - 利用時の様子を記述するものであるため、
    ソフトウェア開発の専門家でなくても理解可能
  - 設計理由の文書化・再利用
    - 機能がどのような文脈で使われるのかが分かる
    - 後工程の評価の際に再利用可能
  - 要求分析時に行われる具体例分析において
    自然に利用できる
  - ワークフロー、業務プロセスをモデル化できる
  - 未知の作業を明確化できる

ユースケース

- 利用者がどのようにシステムを使用するのかを表す典型的な事例
  - シナリオを一般化したもの



ゴール指向分析

- 手順
  - 達成すべき大きなゴール(目標)を明確にする
  - ゴールを部分ゴールに分割し、ゴール木を作成
    - AND分割：すべてを満たさなければならない
    - OR分割：いずれかを満たさなければならない
- 利点
  - ゴールや要求を系統的に整理できる

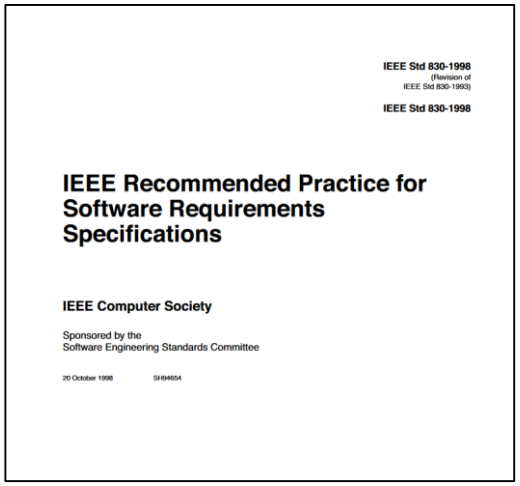
要求の取捨選択の技法

- 交渉 (negotiation)　　合意形成による要求間の矛盾解消
- トリアージ (triage)　　要求の優先順位付け
- AHP法　　解決案の階層化と点数付け
- デルファイ法
  - ステークホルダーがそれぞれ独自に意見を出し合い、
    相互参照を行って再び意見を出し合うという作業を繰り返す
  - 全体の傾向や他人の意見がフィードバックされることで、
    全体の意見が収束する(ことを目指す)

要求仕様記述言語

- 自然言語(natural language)
  - e.g., 日本語、中国語、英語
- 形式的言語(formal language)
  - e.g., Z, VDM, CafeOBJ
- 制限言語(controlled language)
  - e.g., 擬似コード、X-JRDL
- 図式言語(graphic language)
  - e.g., UML、DFD

要求確認


IEEE Std 830-1998
(Revision of
IEEE Std 830-1993)

IEEE Std 830-1998

IEEE Recommended Practice for
Software Requirements
Specifications

IEEE Computer Society

Sponsored by the
Software Engineering Standards Committee

20 October 1998 SH94654

- 要求に関する品質特性(IEEE Std. 830-1998)
  - 妥当性（正当性）： 実現すべき要求を含んでいるか
  - 非曖昧性： すべての要求が一意に解釈できるか
  - 完全性： 必要な情報がすべて含まれているか
  - 無矛盾性（一貫性）： 要求が互いに矛盾しないか
  - 重要度と安定性の順位付け： どの要求を優先的に扱うか
  - 検証可能性： 完成したシステムに対して検証可能か
  - 変更可能性： 要求の一部だけを変更可能か
  - 追跡可能性： 要求仕様とその他の成果物との対応が取れているか
    - 後方追跡可能性：
      各要求について、背景、理由、意図が容易に参照可能
    - 前方追跡可能性：
      設計文書、ソースコード、マニュアルから要求が容易に参照可能

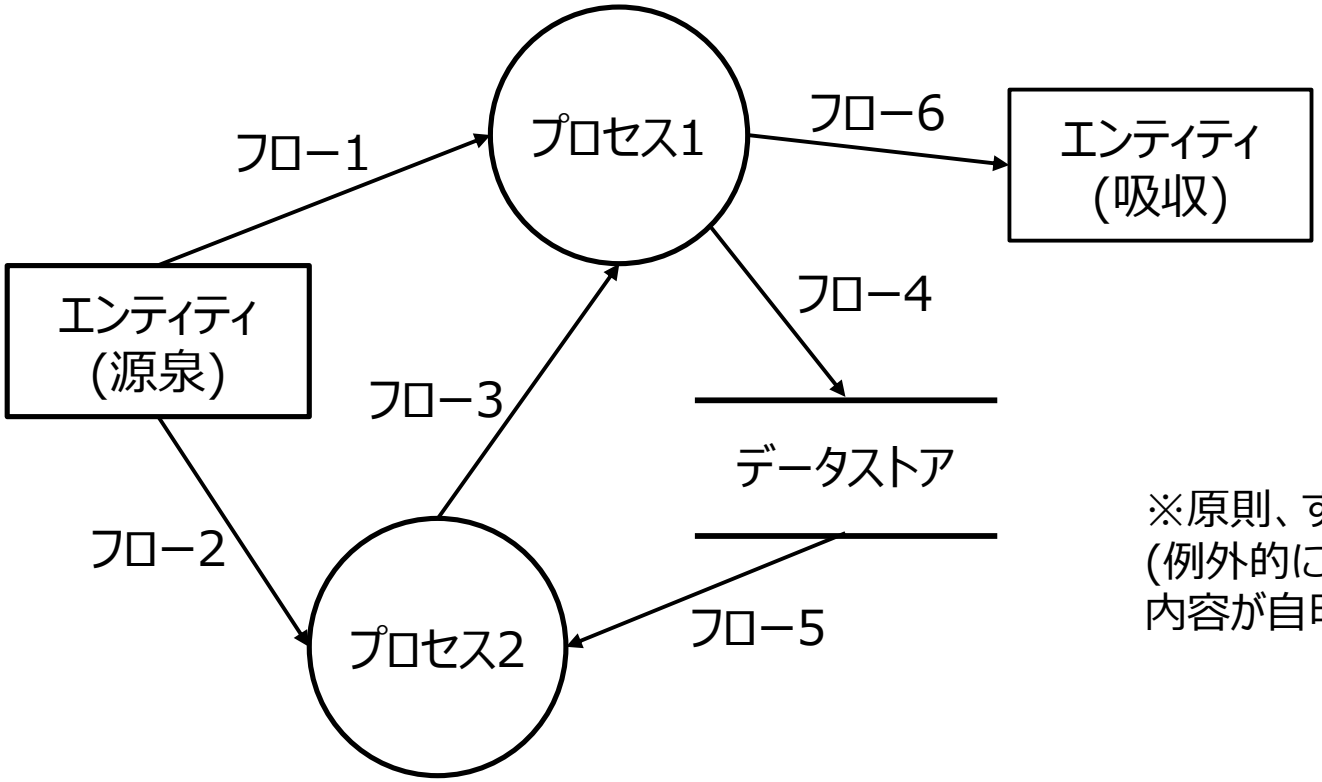
構造化分析

- システムを段階的に詳細化しつつ要求仕様書を作成する技法
- 以下のような図式表現を使用

- データフロー図 (DFD: data flow diagram)
  - システム内部のデータの流れを図示したもの
    - 複数のプロセスとその間を流れるデータを図示
  - プロセス仕様書 (process spec)
    - プロセスの基本処理の内容を表現
  - データ辞書 (data dictionary)
    - データ項目間の関係や構造を表現
- 実体関連図 (ER図: entity relationship diagram)
  - データの構造やデータ間の関係を表現
- 状態遷移図 (state-chart diagram)
  - イベントによる状態や動作の変化を表現

データフロー図

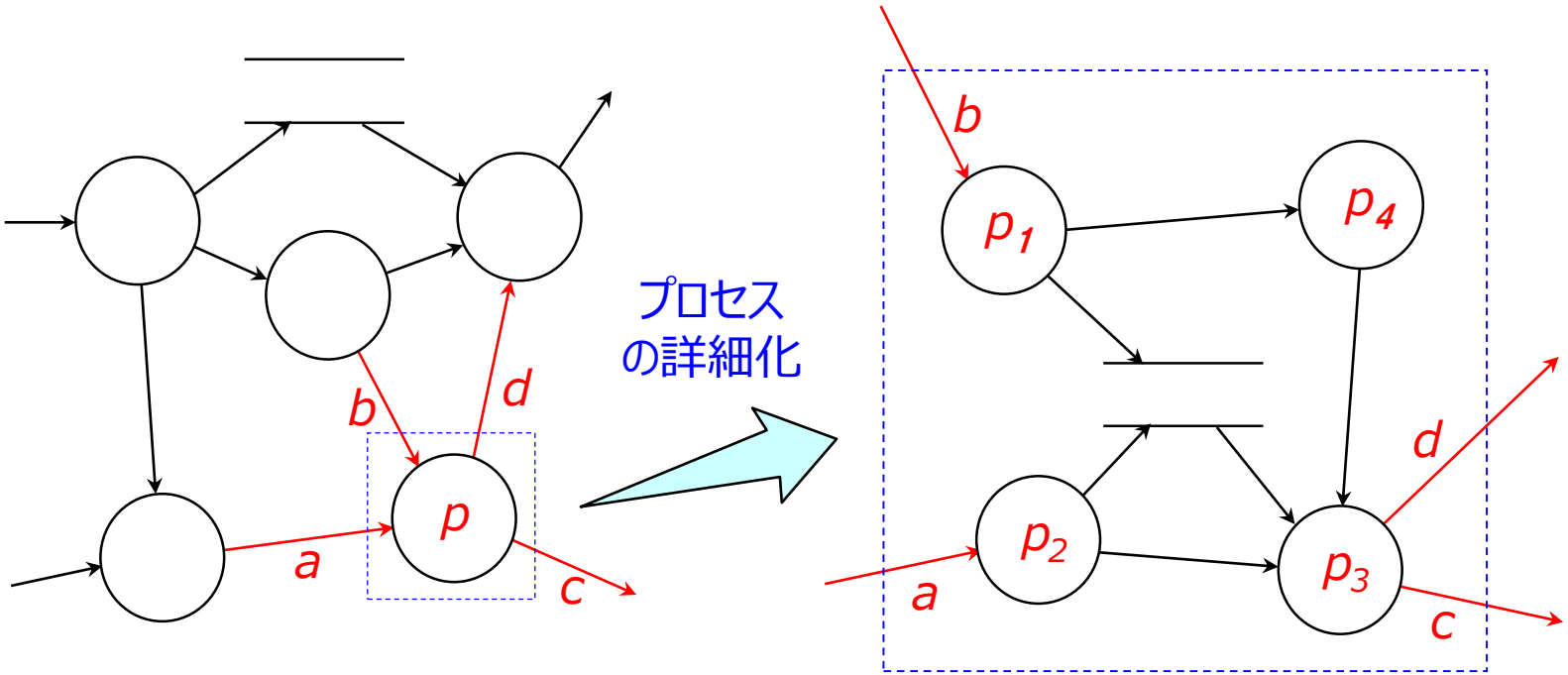
- プロセス(process)　○　データの変換処理　別名：バブル
- フロー(flow)　→　データの流れ
- データストア(data store)　＝　データの格納場所　別名：ファイル
- エンティティ(entity)　□　システム外部に存在する機器や人間等
  - 源泉 (source)：データの発生元
  - 吸収 (sink)：データの最終的な行き先



※原則、すべての要素に名前を付ける
(例外的に、データストアへの入出力は
内容が自明ならば省略することがある。)

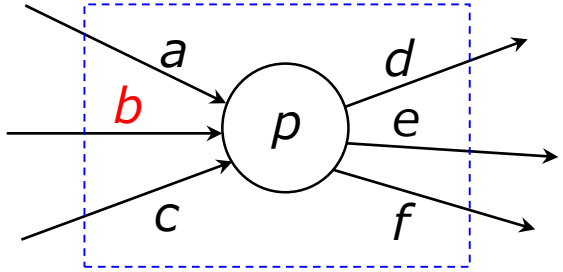
プロセスの詳細化

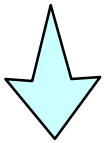
- プロセスの入出力フローは詳細化の前後で
  同一でなければならない (データの詳細化は可)



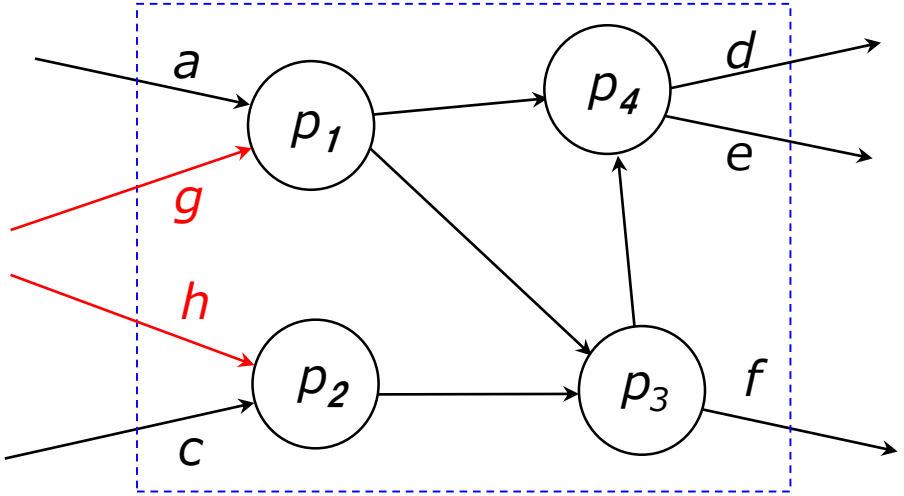
プロセス p は p_1, p_2, p_3, p_4 に分割できる

プロセスの詳細化





データディクショナリで
b = g + h
と定義されていれば、
このように分割できる



DFDの主な規則・制約

- フローはデータの流れのみを表現する
  - 制御の流れ、タイミング、時間的順序、条件選択などは記述不可
- プロセスには1つ以上の入力フローと1つ以上の出力フローが必須
  - プロセスの役割は入力から出力への変換
- データフローの入出力の間の対応関係は表現できない
  - どの入力フローがどの出力フローにつながるのか不明確
    → 表現したければプロセスを適切に分割する
- 起こりうる全てのデータの流れを記述する
  - 特定条件での区別は不可

プロセス仕様 (process spec)

- DFDの最下層のプロセスの処理内容を表現
- 別名：ミニ仕様(mini spec)
- 書式
  - 構造化言語：連接、選択、反復により機能を手続き的に記述
  - 決定表(decision table)：特定の条件と、条件が成立するときのシステムの動作の対応を表で表現
  - 計算式

連接：上から順に実行
選択：複数処理のうち
　条件を満たすものを実行
反復：条件が満たされて
　いる間繰り返す

条件A	F	T	F	T
条件B	F	F	T	T
処理1	X	-	X	-
処理2	-	X	X	-
…				

構造化言語による例

1　受け取った商品番号に該当する商品をカート情報から取得し、
　以下のいずれかの処理を行う。
　1.1　もし、該当商品が見つかった場合、以下の処理を行う。
　　1.1.1　該当商品の個数をカート情報から取得する。
　　1.1.2　該当商品の個数を1つ増加する。
　　1.1.3　新しい個数をカート情報に保存する。
　1.2　もし、該当商品が見つからなかった場合、以下の処理を行う。
　　1.2.1　該当商品をカート情報に1つ登録する。
　　…

データ辞書 (data dictionary)

- DFDに出現するデータ項目間の関係や構造を表現
  - 等価： a=b　　aはbに等しい
  - 連接： a+b　　aとbからなる
  - 選択： [a|b]　　aまたはbのどちらかである
  - 任意： (a)　　aはあってもなくてもよい
  - 反復： {a}　　aを0回以上繰り返す
    - m{a}n　　aをm回以上かつn回以下繰り返す
    - m{a}　　aをm回以上繰り返す
    - {a}n　　aをn回以下繰り返す

例：注文者情報とは、注文者の氏名、住所、電話番号、初期パスワードを合わせたデータをいう。

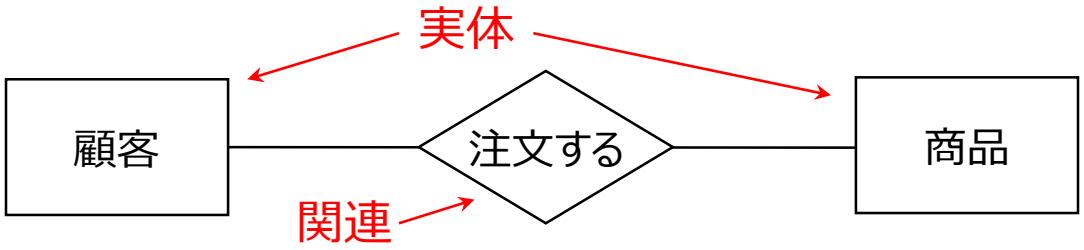
⇨注文者情報 = 氏名 + 住所 + 電話番号 + 初期パスワード

初期パスワード = 4{英数字}8

注文情報 = [商品選択情報|商品取消情報|商品確認情報|注文確定情報]

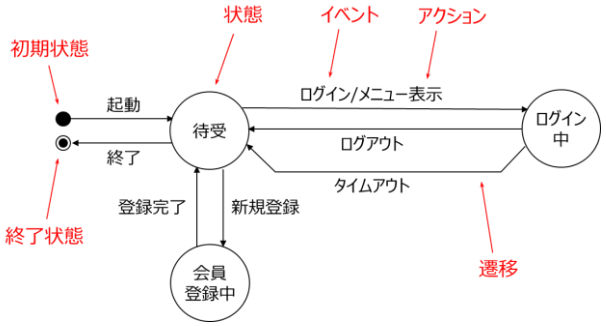
実体関連図 (ER図; entity relationship diagram)

- データ構造やデータ間の関係を表現
  - データの構造やデータ間の関係がデータ辞書で簡単に捉えられない場合
  - DFDだけでシステムの機能を捉えるのが不十分な場合
  - データベースシステム等で特に有用
- 実体
  - システムが管理する対象(e.g.,人、物、お金、場所)
  - 名前を持つ長方形で記述
  - 抽象的な概念や目に見えない物でもよい
- 関連
  - 実体間の結びつき
  - 関連名を持つ菱形で表示



状態遷移図

- システムが取りうるすべての状態(state)と、
  そのシステムに到着したイベント(event)による状態の変化を表現
- リアルタイムシステム、制御系システム、通信システムなどで
  特に有用
  - データの変換よりも状態遷移の方が重要
  - DFDやER図で表現できないタイミング制御等の表現が可能
- システムを状態機械とみなす
  - 状態
    - 名前付きの円あるいは四角形で表現
    - システムは1つの状態に属する (複合状態を除く)
    - 初期状態は　　終了状態は　　で表現
  - 遷移
    - 状態遷移を誘発するイベントの名前を付与した矢印で表現
    - イベントは一瞬、遷移時間は無視できるものとする
  - アクション
    - 状態が遷移する際に実行される処理
    - 遷移のイベント名にアクション名を付記



オブジェクト指向 (object-oriented)

- 現実世界のモデルをソフトウェアで直接的に表現する一つの方法
  - オブジェクトによるモデリング
  - 人間の認知方法にできるだけ近づけることを目指した技法

    → 人間にとって理解しやすい
  - 開発において、一貫してオブジェクトを中心に考える

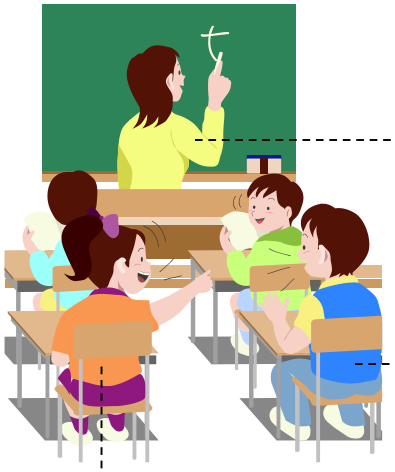
    → 分析、設計、実装が比較的容易に移行可能

オブジェクト

- object　＝(辞書的には、)物
  - 人間が認知できる具体的あるいは抽象的な「物」
  - 実世界の物体や役割、概念を抽象化した「物」

- オブジェクトの持つ特性
  - 状態 (state)
    - オブジェクトの現在の性質
    - プログラミング言語により属性(attribute)、プロパティ(property)等
  - 振る舞い (behavior)
    - オブジェクトが実行できる動作
    - プログラミング言語により操作(operation)、メソッド(method)等
  - 識別性 (identity)
    - 個々のオブジェクトを区別する手段

属性と操作



教員

属性
- 教員番号
- 氏名
- 住所
- …

操作
- 教員番号?
- 氏名?
- 話す
- 授業を行う
- …

学生

属性
- 学籍番号
- 氏名
- 住所
- …

操作
- 学籍番号?
- 氏名?
- 立つ
- 座る
- 話す
- 宿題をする
- …

学生

属性
- 学籍番号
- 氏名
- 住所
- …

操作
- 学籍番号?
- 氏名?
- 立つ
- 座る
- 話す
- 宿題をする
- …

オブジェクト指向の基本概念

- クラスとインスタンス(class, instance)
- カプセル化(encapsulation)
  - データとそれに対する処理をまとめてモジュール化
  - オブジェクトの状態を外部から隠蔽（情報隠蔽）
- メッセージパッシング(message passing)
  - オブジェクトに処理を依頼する仕組み
- 関連(association)
  - あるクラスが別のクラスを利用することを表す、クラス間の関係
- 継承(inheritance)
  - 既存のクラスに属性や操作を追加して新しいクラスを定義すること
- 集約(aggregation)
  - オブジェクトを構成する部品（部分オブジェクト）を束ねて扱う仕組み

クラス

- 同じ属性と操作を持つオブジェクトを抽象化したひな形(template)
- オブジェクトの設計図

属性
学籍番号
氏名

操作
学籍番号?
氏名?

属性
学籍番号
氏名
住所
…

操作
学籍番号?
氏名?
立つ
座る
話す
宿題をする
…

抽象化
(abstraction)

学生クラス

学生 クラス名

学籍番号
氏名
住所
} 属性

学籍番号?
氏名?
立つ
座る
話す
宿題をする
} 操作

Unified Modeling Language (UML)

- モデルを表現する統一的な図式表現法
  - グラフィカルな記法とそのメタモデル(言語の概念)を定義
  - 開発プロセスとは独立

構造	クラス図	クラスの構造(属性や操作)とクラス間の静的な関係
	オブジェクト図	ある時点でのオブジェクトの状態とオブジェクト間の関係
	パッケージ図	パッケージの構成とパッケージ間の依存関係
	複合構成図	実行時のクラスの内部構造
	コンポーネント図	コンポーネントの構造と依存関係
	配置図	システムにおける物理的な配置
振る舞い	ユースケース図	システムの提供する機能と利用者の関係
	アクティビティ図	作業の順序と並行性
	状態機械図	オブジェクトの状態とイベントによる状態遷移
	シーケンス図	オブジェクト間の相互作用の時系列
	コミュニケーション図	オブジェクト間の相互作用のリンク
	タイミング図	オブジェクトの状態遷移のタイミング
	相互作用概要図	シーケンス図とアクティビティ図の概要

ソフトウェアアーキテクチャ (software architecture)

- ソフトウェアの全体的な構造であり、システムの概念的な一貫性を提供する手段
  - モジュールの構造や構成
  - コンポーネント間の相互作用
  - コンポーネントが使用するデータの構造 等
- (広義には、システムの主要な構成要素とそれらの相互作用)

- 開発を方向付ける
  - 基本構造から大きく外れる改造は困難
  - 作業の分担や手順を検討する際の基準
  - 再利用資産の蓄積と活用

アーキテクチャスタイル

- ソフトウェアの基本的構造を表すもので、システム全体の設計に制約を課す
- システムのすべてのコンポーネントの構造を確立することが目的

- ソフトウェアの代表的な基本的構造
  - 機能分割に基づく分類
    - 階層モデル
    - クライアントサーバモデル
    - リポジトリモデル
  - 制御関係に基づく分類
    - データフローモデル
    - コントロールモデル

ユーザインタフェース設計

- ユーザインタフェース (UI: user interface)
  - ユーザとシステムの対話を仲介
- ユーザインタフェース設計
  - 設計指針に基づいてユーザインタフェースを定める作業
  - ユーザの負担を最小限にすることが目的
    - システムがユーザの意図を正しく把握する
    - ユーザに実行結果を分かりやすく提示
    - 使いやすい入力方式（どこに何を入れるのか分かりやすく）
  - 人間の認知機能を考慮する必要
    - 一度に多くの情報を提示しすぎない
    - 追従可能な速度での画面の切り替え等
    - ユーザモデルの構築も重要
      - 例：上級者、中級者、初心者ごとにUI設計

設計ガイドライン

- 観点１：対話の道具
  - 親しみやすいUI
    - 日常的な概念を使う(例：文書、フォルダ)
  - 操作手段・用語・コマンド形式の一貫性
- 観点２：対話の進め方
  - フィードバックの提供
    - ユーザの入力に対するシステムの応答
  - 驚きを最小にする
    - ユーザのメンタルモデル(mental model:ユーザの頭の中のどのように動作するかのイメージ)に沿った振る舞い
- 観点３：エラーへの対応
  - エラーからの回復を容易にする
    - 例：アンドゥ(undo)機能
  - ユーザへのガイダンス
    - 例：ヘルプ機能、オンラインマニュアル

※イメージ：頭の中で思い浮かべる姿･形。心像、形象。

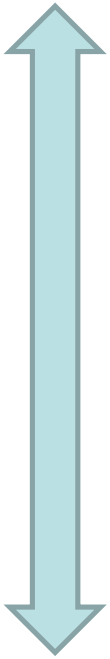
モジュール設計

- ソフトウェアをモジュールに分割し構造化する作業
  - ソフトウェア ＝ 複数のモジュールで構成
- モジュール(module)
  - 独立した機能あるいは関連する機能をひとまとめにしたプログラム単位
  - サブルーチン、関数、手続き、クラス等
- モジュール分割の利点
  - 抽象化：詳細を把握せずとも利用可能
  - 開発効率：並行して開発可能
  - 再利用：既存のモジュールを再利用可能
  - 変更容易性：変更範囲を局所化

モジュール強度(凝集度)

- 暗号的強度、偶発的強度 (coincidental cohesion)
- 論理的強度(logical cohesion)
- 時間的強度(temporal cohesion)
- 手順的強度(procedural cohesion)
- 連絡的強度(communicational cohesion)
- 情報的強度(informational cohesion)
- 機能的強度(functional cohesion)

凝集度低

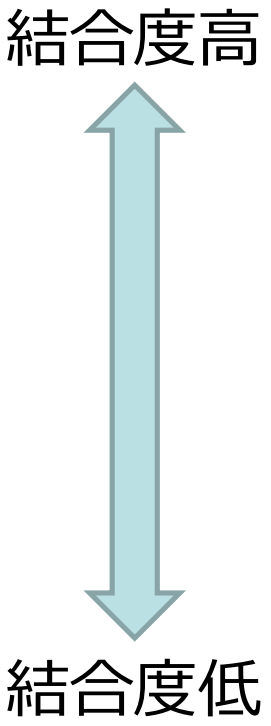


凝集度高

※モジュール強度の強さの順番は解釈によって変わることもある

モジュール結合度

- 内容結合(content coupling)
- 共通結合(common coupling)
- 外部結合(external coupling)
- 制御結合(control coupling)
- スタンプ結合(stamp coupling)
- データ結合(data coupling)



※モジュール結合度の強さの順番は解釈によって変わることもある

プログラミング

- プログラム(program)
  - プログラマが記述したソースコード
  - システム上で動作するオブジェクトコード(マシンコード)
- プログラミング(programming)
  - 仕様(要求仕様や設計仕様)に基づきプログラムを作成する作業
  - 実装(implementation) とほぼ同義
  - 要求が正しく記述されたとしても、実装工程で誤りが混入すると、システムは要求を満たさない
    → 簡潔でわかりやすいプログラムを、誤りなく確実に作ることが重要

プログラミングパラダイム

- プログラミングパラダイム(programming paradigm)
  - プログラムの作り方に関する規範
  - 設計手順、プログラム構造、プログラムの記述方法を規定
  - プログラミングの際に、何に着目して問題を整理するのか、何を中心にプログラムを構成するのかの方向付けを与えるもの
- パラダイムの例
  - 手続き型：処理手順を命令文で記述
    - FORTRAN, COBOL, BASIC, PASCAL, C, Ada等
  - 関数型：入出力関係を表現する関数とその呼び出しで記述
    - Lisp, Scheme, ML, Haskell等
  - 論理型：入出力関係を述語論理(事実と規則)で記述
    - Prolog等
  - オブジェクト指向：データとその操作をカプセル化したオブジェクトとその間のメッセージ通信で記述
    - Smalltalk, C++, Java, C#, Ruby等

プログラミングに求められるもの

- 昔の記述規範
  - メモリの使用量が少ない
  - 処理時間が早いこと

    
    But,
  - 理解しにくくなる傾向がある
- 近年のプログラミング
  - 理解しやすいプログラムの記述を目指す
    - 構造や振る舞いを容易に把握できる
    - 正しさを検査しやすい
    - バグの特定が容易
    - 構造が整っている

構造化プログラミング

- 段階的詳細化等によりプログラムを構造化するための
  プログラミング技法
- ダイクストラ(E.W. Dijkstra)により提唱

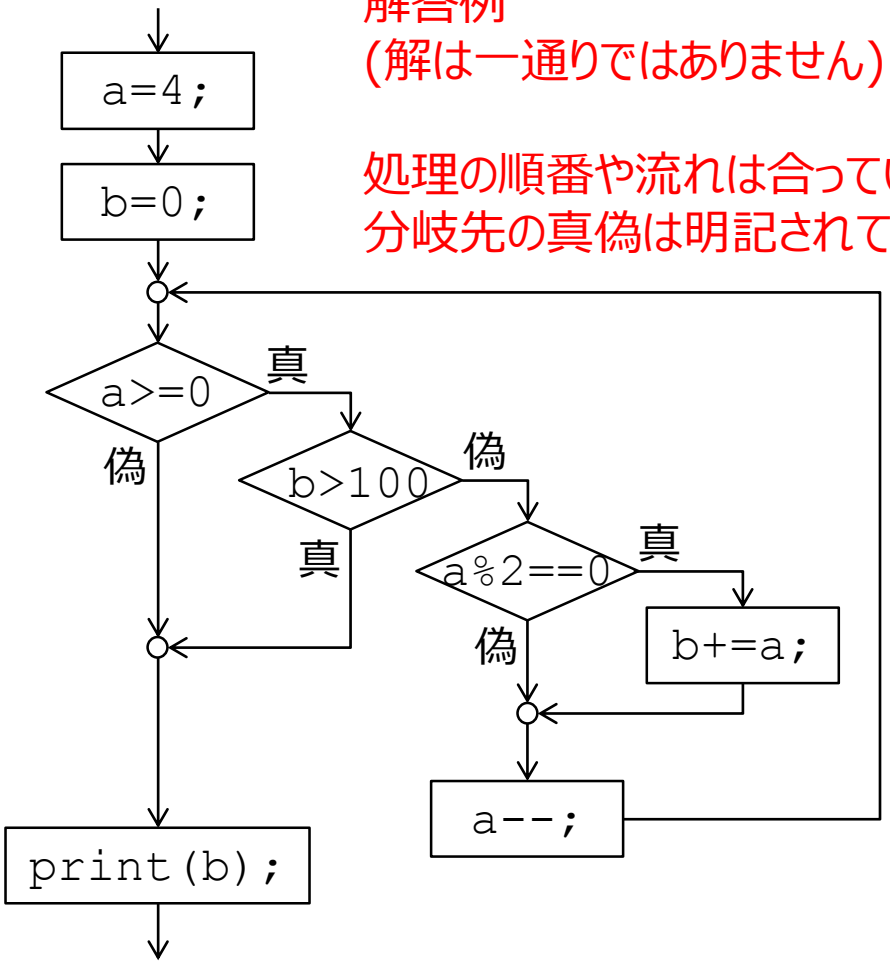
- GOTO文の是非
  - 任意の命令に制御を移動可能なGOTO文はできる限り使わない
  - 関数からの戻り(return)、繰り返しからの脱出(break)、
    例外処理等は必要
- 言語との関係
  - 多くの手続き型言語で構造化プログラミングの
    考え方が取り入れられている
  - オブジェクト指向言語等を採用した場合でも
    制御構造は記述するため、考え方として有効

確認問題

- 以下のC言語のプログラムの制御の流れを示すフローチャートを記述せよ。

```
int a=4;
int b=0;

while(a>=0){
   if(b>100){
      break;
   }
   if(a%2==0){
      b+=a;
   }
   a--;
}
print(b);
```



解答例
(解は一通りではありません)

処理の順番や流れは合っているか？
分岐先の真偽は明記されているか？

データ構造に基づく設計

- システム内で扱うデータの基本的な特性に着目したモジュール設計技法

- データ中心アプローチ (DOA: data-oriented approach)
  - 業務(プロセス)の変更に強い
  - e.g., ジャクソン法、ワーニエ法

- プロセス中心アプローチ (POA: process-oriented -)
  - 業務(プロセス)の変更に弱い
  - e.g., STS分割、TR分割

ソフトウェアテスト

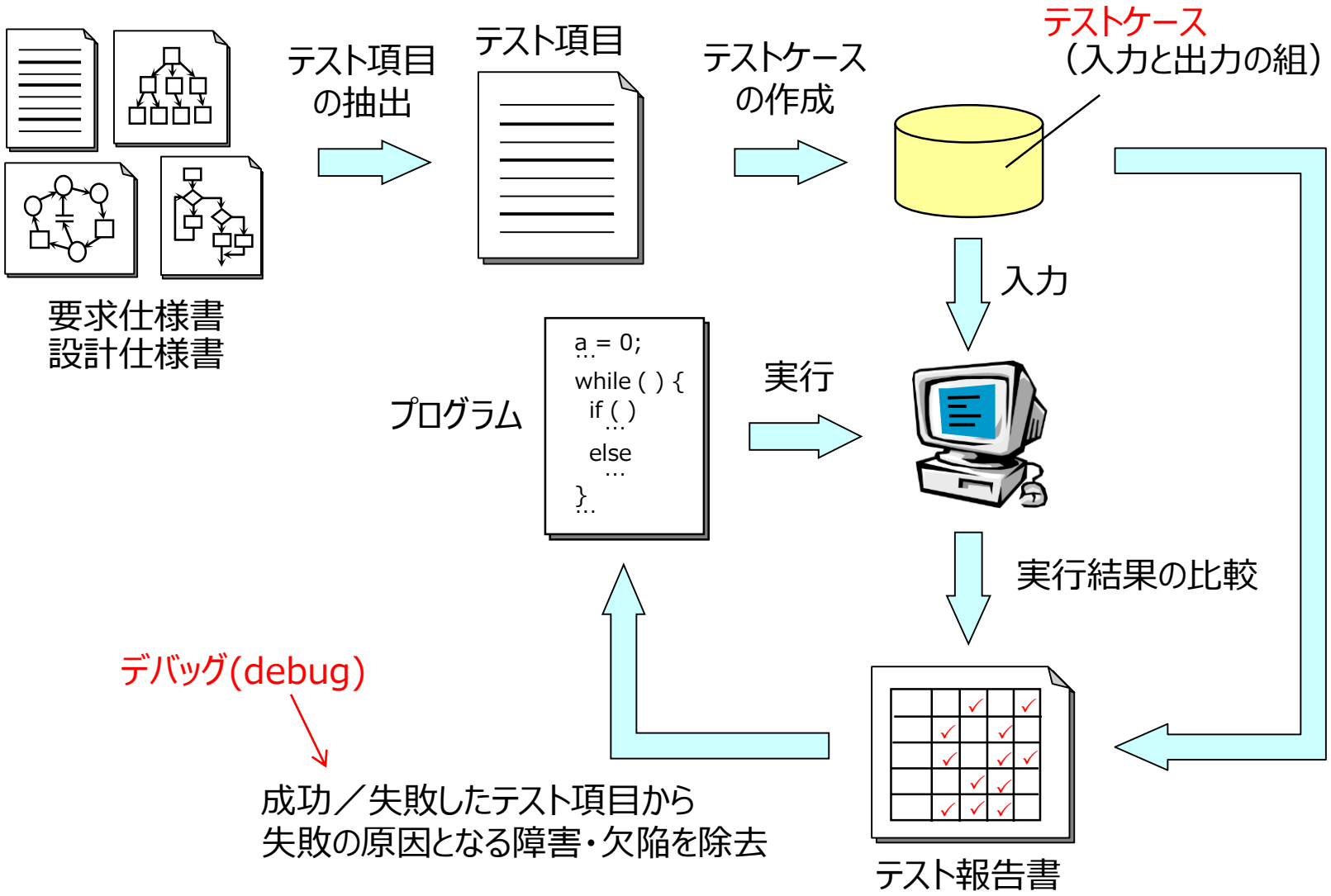
- ソフトウェアテスト
  - プログラムにテストデータを与えて実行し、
    その結果から開発中に紛れ込んだ誤りを検出する作業
- ソフトウェアテストに関する用語 (cf. JIS X0014-1999)
  - 誤り (error)
    - 計算、観測もしくは測定された値または状態と、
      真の、指定されたもしくは論理的に正しい値または状態との間の相違
      → 外部的な振る舞いとして、正しく動かないこと
  - 故障 (failure)
    - 要求された機能を遂行する、機能単位の能力がなくなること
      → 外部的な振る舞いとして、正しく動かなくなること
  - 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

※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。

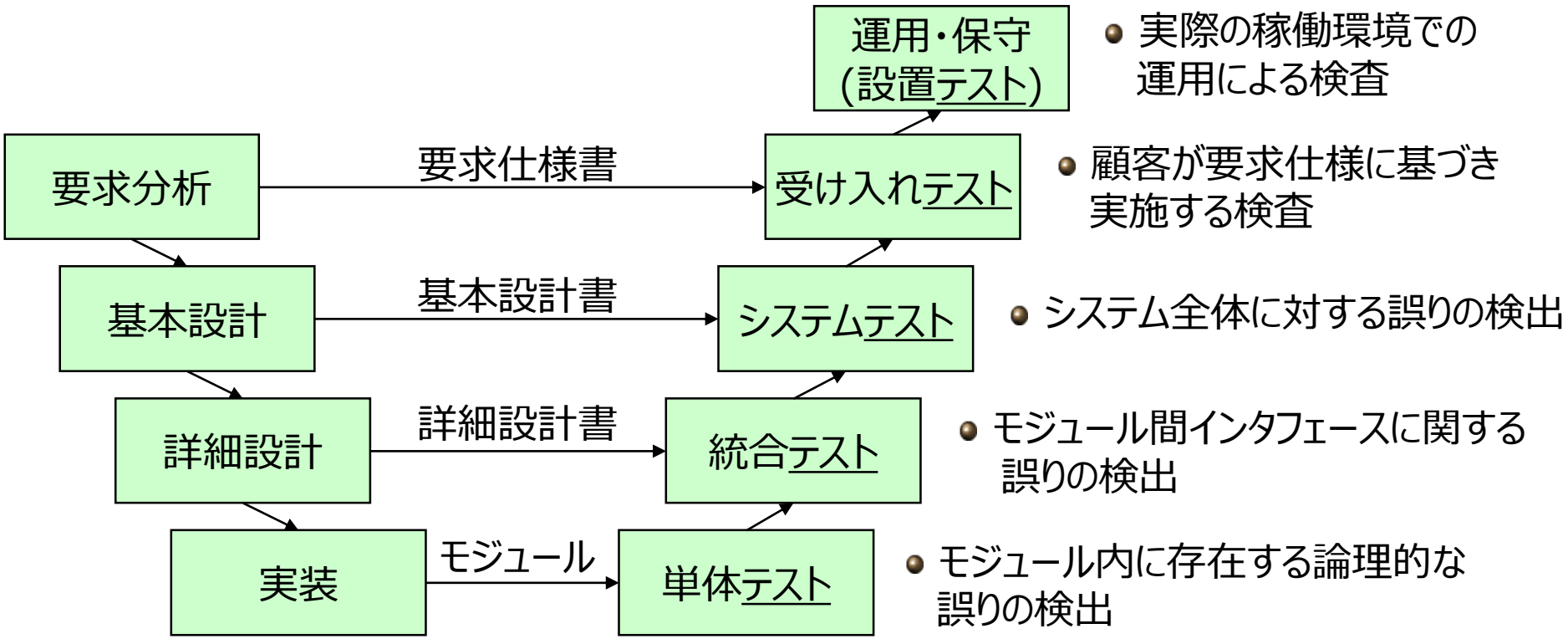
ソフトウェアテストの性質

- 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

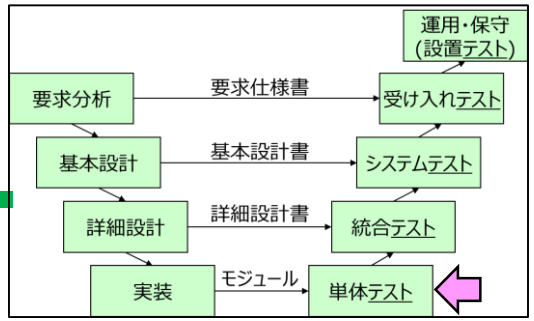
ソフトウェアテストの作業



テスト工程 (V字モデル)


運用・保守
(設置テスト)
実際の稼働環境での
運用による検査
要求分析
要求仕様書
受け入れテスト
顧客が要求仕様に基づき
実施する検査
基本設計
基本設計書
システムテスト
システム全体に対する誤りの検出
詳細設計
詳細設計書
統合テスト
モジュール間インタフェースに関する
誤りの検出
実装
モジュール
単体テスト
モジュール内に存在する論理的な
誤りの検出

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

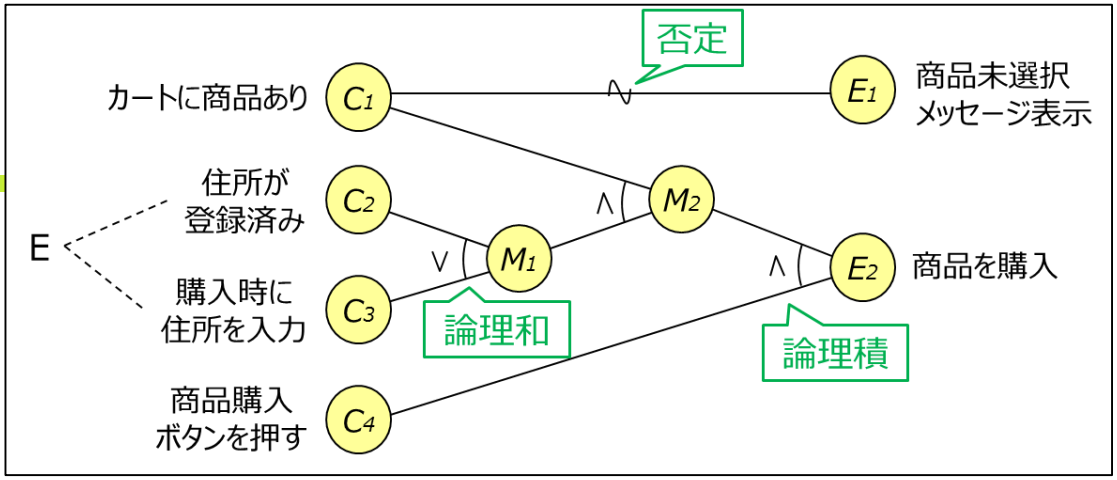
- 入出力条件の境界値を詳しくテストするテストケースを作成
  - テストケース(test case)：プログラムのテストのために入力する値と想定される出力値の組
  - 境界値やその付近の値を入力データとする
  - 経験的に、多くの誤りが境界値付近で発生
    - 不等号(<と<=, >と>=)の書き間違い等
- 手順
  - (1) 入出力条件の識別

(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＆決定表



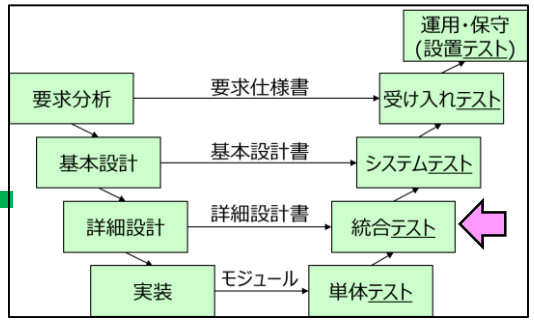
		1	2	3	4	5	6	7	8	9	10	11	12
原因	C1（カートに商品あり）	×	×	×	×	×	×	○	○	○	○	○	○
	C2（住所が登録済み）	×	×	×	×	○	○	×	×	×	×	○	○
	C3（購入時に住所を入力）	×	×	○	○	×	×	×	×	○	○	×	×
	C4（商品購入ボタンを押す）	×	○	×	○	×	○	×	○	×	○	×	○
補助	M1（住所を入力済み）	×	×	○	○	○	○	×	×	○	○	○	○
	M2（購入可能）	×	×	×	×	×	×	×	×	○	○	○	○
結果	E1（商品未選択）	○	○	○	○	○	○	×	×	×	×	×	×
	E2（商品を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

- C2∧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

論理網羅

- 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

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フェースに関するエラーを検出

(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

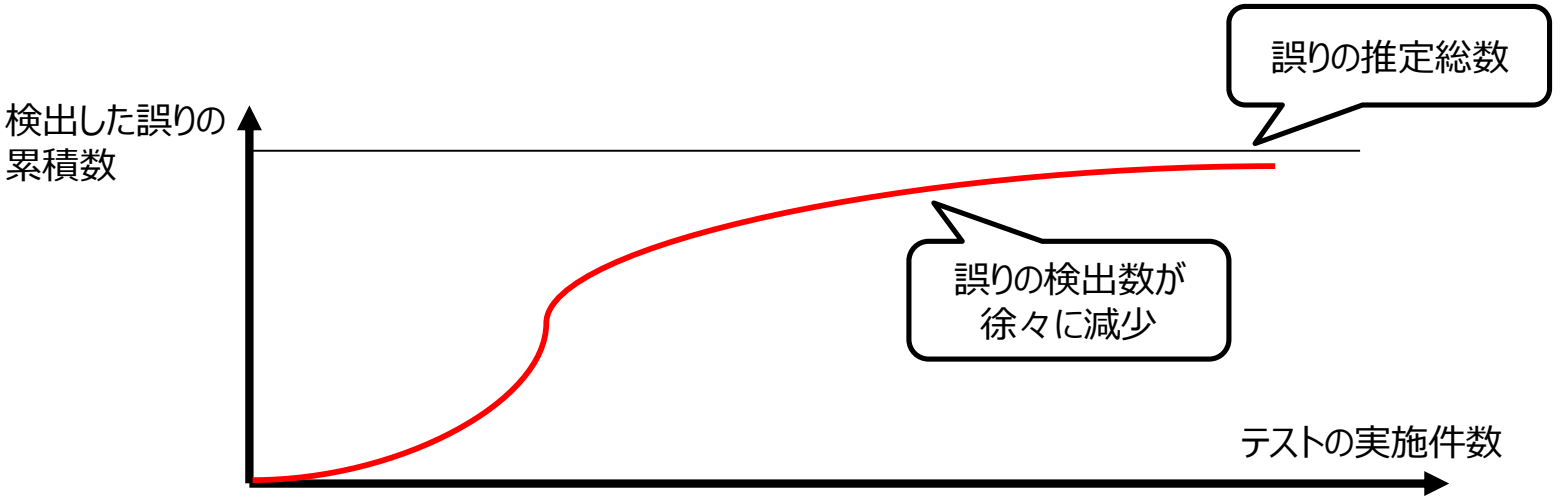
(b)トップダウンテスト(top-down test)
- モジュール構成図の最上位モジュールからテスト開始

(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
  - 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

(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

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
    (⇔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

ソフトウェア保守

- ソフトウェア保守 (software maintenance)
  - ソフトウェアを維持・管理する作業
  - 目的
    - 常に正しく稼働させる
    - 顧客の要求に合わせて変更・修正する
  - 意義
    - ソフトウェアは経年劣化がない＝利用期間が長い
    - 導入後も変更が容易
      → 欠陥の修正や新しい要求への対応が求められる
- ソフトウェア進化 (software evolution)
  - M.M. Lehmanにより提唱
  - ソフトウェアの持続だけでなく、新しい要求を受け入れ、機能拡張、性能改善を目指す

保守の対象

- 保守はソフトウェアの完成後に行われる
  → それまでに作成された全文書が保守の対象
  - 要求仕様書、基本設計書、詳細設計書、
    ソースコード 等
- 自分で作成した文書だけが対象とは限らない
  - 作成時点でわかりやすいものを作る必要がある
  - 既存の文書の理解支援も必要

保守の種類

- 修正保守
  - 出荷後に発見された誤りや障害を修正
- 適応保守
  - 利用環境の変化に追従するための修正
- 完全化保守
  - 品質向上のための修正
    - e.g., 実行速度の高速化、
      保守性向上のためのモジュール再構成
- 予防保守
  - 故障を防ぐための修正
    - e.g., defensive programmingの導入

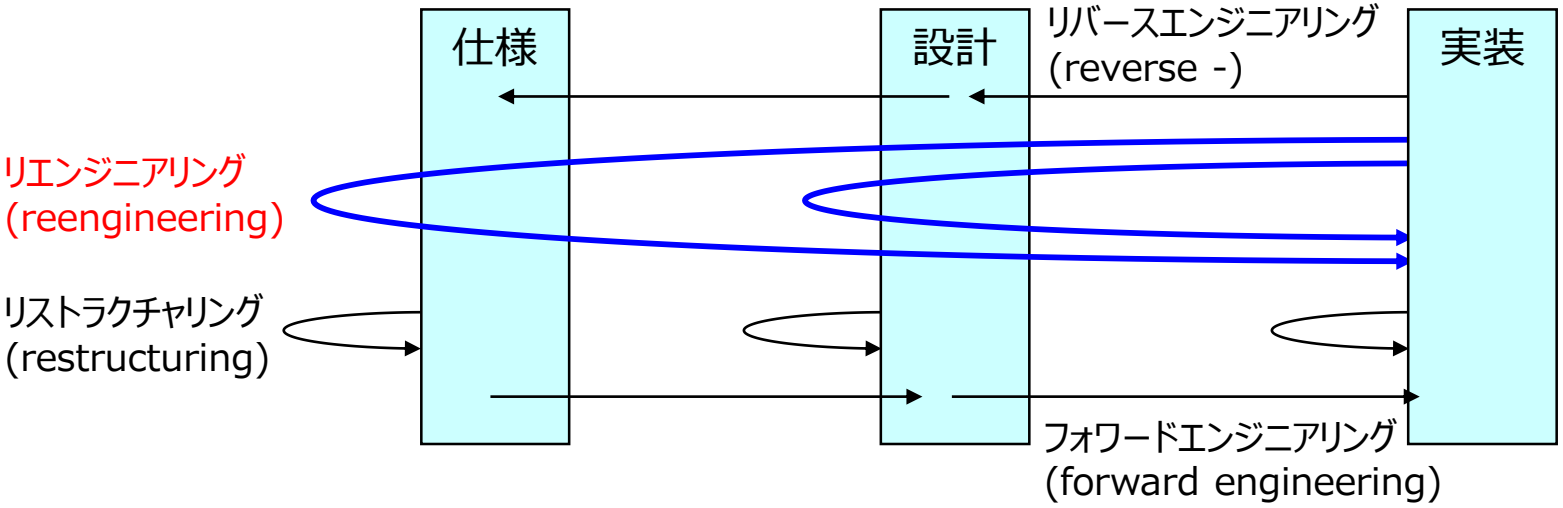
プログラム理解支援

- プログラムの構成と対応する要求は
  各種設計仕様書・要求仕様書等で
  追跡可能であることが理想
  → 実際には文書間で矛盾が発生

- プログラム理解支援 が必要
  - 実際に稼働しているプログラムの理解を支援
  - 文書間に矛盾がある場合、信頼できるものは
    稼働しているプログラム

リバースエンジニアリング

- リバースエンジニアリング
  - 後工程の成果物から前工程の成果物を作成
- フォワードエンジニアリング
  - 通常の開発
- リエンジニアリング
  - リバースエンジニアリング ＋ フォワードエンジニアリング
- リストラクチャリング
  - 特定の種類の成果物を改善 (ソフトウェアの振る舞いは不変)



- ✓ 成果物間の矛盾の解消
- ✓ ソフトウェアの再構成による設計・実装改善

リファクタリング (refactoring)

- リファクタリング
  - 既存のソフトウェアの外部的な振る舞いを変えることなく、実装を改善する作業
  - コードの理解性や保守性を向上させる
  - リストラクチャリングの一種
  - コード変換の目的や内容が「カタログ(catalog)」としてまとめられている
    - Rename: 既存の構成要素の名前を変更する
    - Extract Class: 既存クラスの一部を新しいクラスとして抽出
    - Move: クラスの構成要素を他のクラスに移動する
    - Pull Up: クラスの構成要素を親クラスへ移動する
    - Push Down: クラスの構成要素を子クラスへ移動する
    - 他多数

回帰テスト (regression test)

- 変更前のソフトウェアで動作していた機能が、変更後も正しく動作するか検査する作業
  - テストケースの内容は元のテストケースと同じで良い
  - 誤り混入時の対応を容易にするため、少し変更するたびに行うのが良い
  - 何度もテストを行う手間がかかる
    → テストの自動化が有用

プロジェクト管理

- プロジェクト管理
  - ソフトウェア開発全体の計画、
    進行状況の確認、その結果に応じた対策を行うこと
- プロジェクト管理の目的
  - 要求を満たすソフトウェアの開発
  - 決められた予算と期間で開発

- プロジェクト管理知識体系
  (PMBOK: project management body of knowledge)
  として、基本的な知識体系がまとめられている
  - 統合管理：プロジェクト計画の策定・実施・変更管理
  - 時間管理：スケジュールの作成、作業時間見積り、進捗管理
  - コスト管理：資源計画、コストの見積もりと予算化
  - リスク管理：リスクの洗い出し、対策　　等の視点がある

開発工数の見積もり

- プロジェクトの計画においては、
  まず開発工数を見積もり、
  それに基づき開発コストを見積もる

- 開発工数の単位
  - 人：何人の開発者が必要か
  - 時間(e.g., 月、日、時間)：どれだけの時間が必要か
  - 人月(にんげつ)：人×月
    - 1人月＝1人が1か月かかる作業量
    - すべての開発者が同じ能力？
    - 3人×1か月＝1人×3か月？
    - 開発者を増やすと、効率は低下する
      - コミュニケーションや管理のコストが大

ソフトウェアメトリクス

- プロジェクト管理においては、
  プロダクトやプロセスを評価する必要
  → 定量的な評価尺度(metrics)を導入
- ソフトウェアメトリクス：
  ソフトウェア開発に関する定量的な評価尺度
- ソフトウェアメトリクスの種類
  - プロダクトメトリクス
    - 成果物の評価尺度
  - プロセスメトリクス
    - 開発作業の評価尺度

参考文献

- 大森 隆行
  ソフトウェア工学 2021年度
  講義資料 第1～14回