ソフトウェア工学（B1）
（2013年度）

桑原 寛明
情報理工学部 情報システム学科
kuwabara@cs.ritsumei.ac.jp
http://www.ritsumei.ac.jp/~hkuwa/class/2013/se/

ソフトウェア工学とは

ソフトウェア

- データ処理システムを機能させるための、プログラム、手順、規制、関連文書などを含む知的な創作（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、運用マニュアル
  - ✓ プログラム、プログラムを作成する過程で得られるシステム設計書、フローチャートをはじめとする設計書、および、プログラム説明書などの関連資料
- cf. ハードウェア(hardware)
  - ✓ コンピュータ装置

ソフトウェアとハードウェアの比較

- ソフトウェア
  - ✓ 経年劣化なし
  - ✓ 導入後に修正可能
  - ✓ 製品の量産コスト、配布・流通コストが低い
    - → 機能拡張、性能改善、環境適合に関する要求
    - → ソフトウェア進化が前提
- 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あり
    - → 機能や性能を維持することに対する要求

ソフトウェア開発

- 顧客の要求をソフトウェア製品に変換する作業
  - ✓ よいソフトウェアを効率的に構築
  - ✓ 品質（出来の良さ）とコスト（時間/費用）のバランスが重要
- 開発技術の歴史
  - ✓ 1960年代（機能の時代）
    - 既存業務の情報システム化
  - ✓ 1970年代（スケジュールの時代）
    - 開発計画やライフサイクルの登場
  - ✓ 1980年代（コストの時代）
    - 開発における生産性が重要視
  - ✓ 1990年代（品質の時代）
    - 社会の依存度の増大
    - 使い勝手の良さに対する要求
  - ✓ 21世紀（複雑さへの挑戦）

ソフトウェア工学

- コンピュータソフトウェアを対象として、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)
- ソフトウェア危機(software crisis)を打開するためのソフトウェア開発の技術体系および学問体系
- ソフトウェア工学の目的: よいソフトウェアを開発すること
  - ✓ よいソフトウェアとは？
    利用者のニーズを満たす、高信頼、使いやすい、高速、保守が容易、…
    ソフトウェアの品質特性（ISO9126）
- ソフトウェア工学の実践
  - ✓ ソフトウェアを開発するための理論、原理、技術に基づく方法論や、表記法、ツールを活用して、ソフトウェアを構築、運用、保守する活動
  - ✓ 大規模・高信頼ソフトウェアの開発 ≠ プログラミング

ソフトウェア危機

- 技術者不足、納期遅れ、品質低下、開発費増大
  - ✓ 1968年のNATO会議で指摘
- 生産物の増加
  - ✓ ハードウェアの大型化に伴うソフトウェアの規模の拡大
  - ✓ コンピュータの普及に伴う開発ソフトウェアの数の増加
- 人の増加
  - ✓ ソフトウェアの開発と利用に関わる利害関係者（ステークホルダ）の多様化
    - 利用者、顧客、販売担当者、企画担当者
    - プロジェクト管理者、設計担当者、開発担当者、運用担当者、保守担当者
  - ✓ ソフトウェアの規模が大きくなるとステークホルダも増える
    - ステークホルダ間の意思疎通や調整が難しくなる

ソフトウェア危機

- 期間の長期化
  - ✓ 開発期間、利用期間の長期化
  - ✓ 担当者の交代、動作環境や社会的要求の変化
- 社会的役割の変化
  - ✓ コンピュータで扱う分野の拡大
    - 社会のインフラを担うコンピュータシステム
  - ✓ 社会的に重要なコンピュータシステムに対する信頼性の要求
  - ✓ ソフトウェアの大衆化に伴う使い勝手の向上

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
   - ✓ 合目的性：利用者の目的にあっているか
   - ✓ 正確性：仕様に対して正しく動作するか
   - ✓ 相互運用性：他のシステムとやり取りできるか
   - ✓ セキュリティ：不当なアクセスを排除できるか
   - ✓ 標準適合性：ソフトウェアの機能が法規、規格、業界標準を遵守しているか

2. 信頼性(reliability)：機能が正常に動作し続けるか
   - ✓ 成熟性：障害時にソフトウェアが停止しないか
   - ✓ 障害許容性：障害時に機能を提供し続けられるか
   - ✓ 回復性：故障したときに素早く復旧できるか
   - ✓ 標準適合性：ソフトウェアの信頼性が法規、規格、業界標準を遵守しているか

ソフトウェアの品質特性（ISO9126）

3. 使用性(usability)：利用者にとって使いやすいか
   - ✓ 理解性：使い方が理解しやすいか
   - ✓ 習得性：初めてでもすぐに使えるようになるか
   - ✓ 操作性：ユーザインタフェースが使いやすいか
   - ✓ 魅力性：利用者にとって魅力があるか
   - ✓ 標準適合性：ソフトウェアの使用法が法規、規格、業界標準を遵守しているか

4. 効率性(efficiency)：目的達成のために使用する資源は適切か
   - ✓ 時間的効率性：応答時間が短いか、処理速度が速いか、指定されたスループットが確保できるか
   - ✓ 資源効率性：メモリやネットワークなどの資源を余計に消費しないか
   - ✓ 標準適合性：ソフトウェアの性能が法規、規格、業界標準を遵守しているか

ソフトウェアの品質特性（ISO9126）

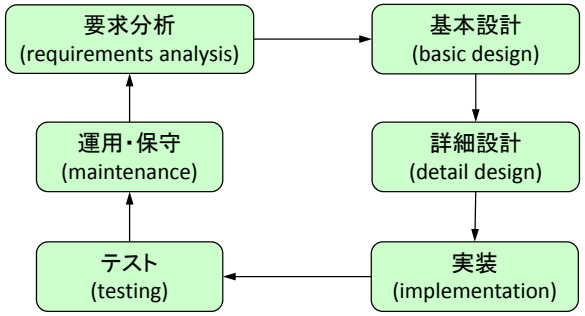
5. 保守性(maintainability)：改訂作業に必要な労力は少ないか
   - ✓ 解析性：変更箇所を特定しやすいか
   - ✓ 変更性：プログラムが変更しやすいか
   - ✓ 安定性：変更時にその影響が予想外の箇所に及ばないか
   - ✓ 試験性：変更時にテストがしやすいか
   - ✓ 標準適合性：ソフトウェアの保守性が法規、規格、業界標準を遵守しているか

6. 移植性(portability)：他の環境へ移しやすいか
   - ✓ 順応性：別の環境に移す際の手間は少ないか
   - ✓ 設置性：インストールしやすいか
   - ✓ 共存性：同じ環境で他のソフトウェアと共存できるか
   - ✓ 置換性：他のソフトウェアに置き換え可能か
   - ✓ 標準適合性：ソフトウェアの可搬性が法規、規格、業界標準を遵守しているか

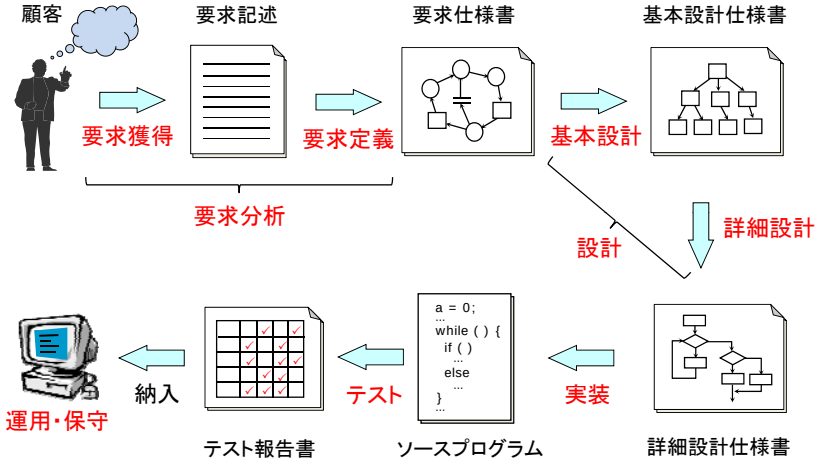
ソフトウェア開発モデル

ソフトウェア開発モデル

- ソフトウェア開発の進め方の規定
  - ✓ 開発における作業：プロセス(process)
  - ✓ 各作業においてプロダクト(product)を生成
    - 文書、図表、プログラム、マニュアルなど
- ソフトウェアプロセスモデル(process model)
  - ✓ ライフサイクル(life cycle)ともいう



ソフトウェア開発プロセス



ソフトウェア開発プロセス（要求分析）

1. 要求獲得
   - ✓ 利用者（ユーザ）が要求するシステムを整理し、自然言語で文書化
     - 要求記述
     - システム要求書(system requirements)
2. 要求定義
   - ✓ どのようなシステムを作成するのかを決定
   - ✓ システム要求書を分析し、開発するシステムを形式的に文書化
     - 要求仕様書(requirements specification)
   - ✓ 分析者(analyst)が実施

ソフトウェア開発プロセス（設計）

3. 基本設計
   - ✓ システム設計(system design)、外部設計(external design)ともいう
   - ✓ 外部から見たシステム全体をどのように作成するのかを決定
   - ✓ ソフトウェアアーキテクチャやユーザインタフェースを決定
     - 基本設計仕様書
   - ✓ 設計者(designer)が実施
4. 詳細設計(detailed design)
   - ✓ プログラム設計(program design)、内部設計(internal design)ともいう
   - ✓ 個々のモジュールのアルゴリズムやデータ構造を決定
     - 詳細設計仕様書
   - ✓ プログラマ(programmer)が実施

ソフトウェア開発プロセス（実装、テスト）

5. 実装
   - ✓ コーディング(coding)ともいう
   - ✓ 詳細設計仕様をプログラムに変換
   - ✓ 具体的なプログラミング言語(programming language)による記述
     - ソースプログラム(source program)
   - ✓ プログラマが実施

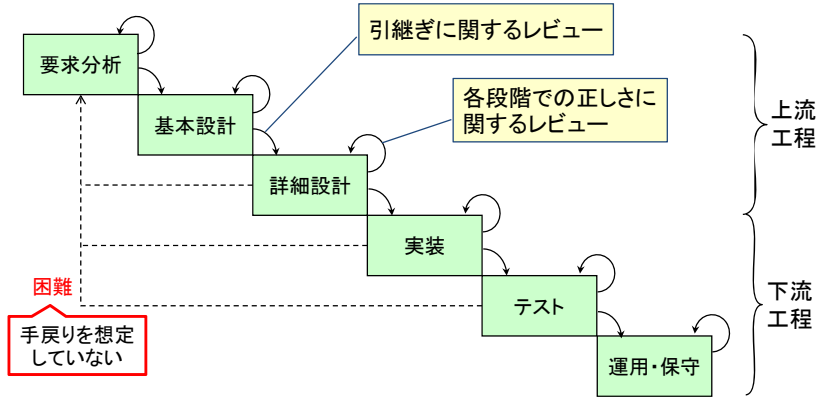
6. テスト
   - ✓ 仕様書通りにプログラムが動作するかどうかを検査
     - テスト報告書(test report)
   - ✓ 試験者(tester)が実施
     - ソフトウェア作成と独立の組織を用意
   - ✓ デバッグ(debug)
     - エラー(error) の修正は設計者やプログラマが実施

ソフトウェア開発プロセス（運用・保守）

7. 運用・保守
   - ✓ 納入後のソフトウェアを維持・管理
   - ✓ 運用段階で検出された故障（残存エラー）の修正
   - ✓ 変更要求への対応
     - 新機能の追加
     - 既存機能の変更
     - 新しい環境への適合
   - ✓ 保守者(customer engineer)が実施

ウォーターフォールモデル

- トップダウンな開発プロセス：滝(waterfall)



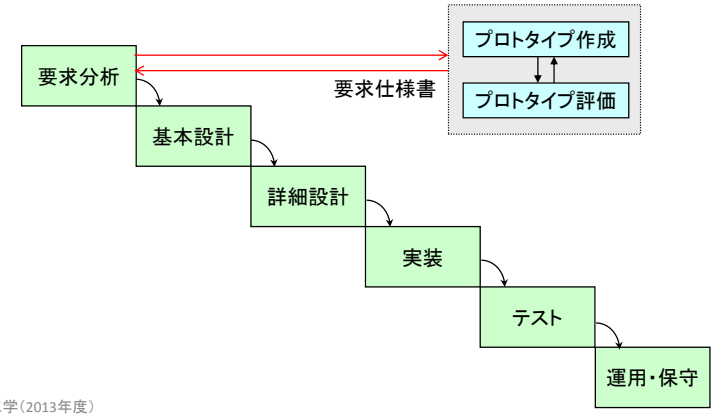
利点：工数見積もりや進捗管理が容易
欠点：仕様変更に弱い、誤り発見の遅れ、修正コストの増大

ソフトウェア開発モデルの推移

- 1960年代：流れ図(flowchart)を利用
  - ✓ 開発方法論なし
- 1970年代：構造的なソフトウェア開発
  - ✓ ウォーターフォールモデル
- 1980年代：ソフトウェアライフサイクル有害説
  - ✓ 新しいソフトウェア開発モデルの必要性
- 進化型（成長型、発展型）プロセスモデルの登場
  1. 開発の初期段階から実行可能なプログラムを部分的に作成
  2. プログラムを実際に動作させて機能を確認
  3. プログラムを改善
  4. 2. と 3. を繰り返し
  - ✓ 使い捨てプロトタイピング(throwaway prototyping)
  - ✓ スパイラルモデル(spiral model)
  - ✓ 進化型プロトタイピング(evolutionary prototyping)
  - ✓ アジャイルプロセスモデル(agile process model)

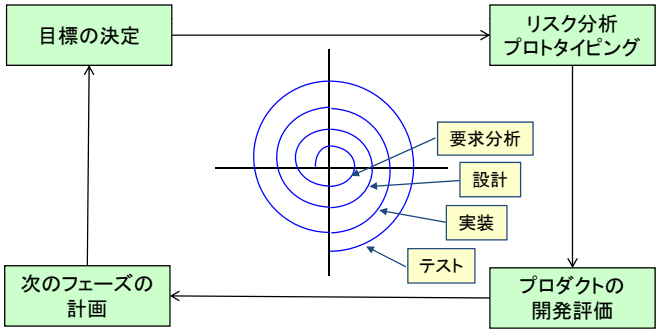
使い捨てプロトタイピング

- システム設計時にプロトタイプ（試供品：prototype）を構築
  - ✓ 問題点の早期発見と要求の曖昧さの解消
    - プロトタイプに基づき要求仕様書を作成
  - ✓ 大幅な手戻りを減少



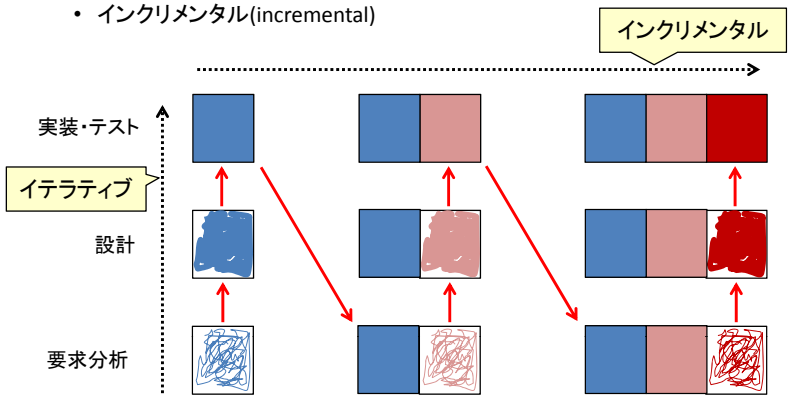
スパイラルモデル

- プロトタイプを作成し、利用者からのフィードバックに対応しながら、徐々にシステムを作成
  - ✓ リスク（開発に失敗）を分析することで、リスクを軽減
- プロセスモデルを改善する
  - ✓ 他のプロセスモデルと組み合わせて用いる



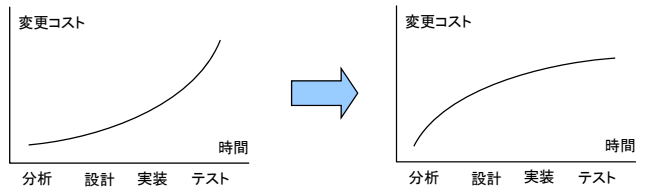
進化型プロトタイピング

- プロトタイプを少しずつ修正し、そのまま完成システムとして利用
  - ✓ 機能が明確な部分から開発
  - ✓ 反復型開発モデル
    - イテラティブ(iterative)
    - インクリメンタル(incremental)



アジャイルプロセスモデル

- 変化に迅速に対応
  - ✓ 軽量(lightweight)プロセス
    - XP(extreme programming)、Scrumなど
- 開発対象を多数の小さな機能に分割して、各機能を短期間で開発
  - ✓ 1つの反復(iteration)で1機能を実現
  - ✓ 実行可能なソフトウェアを随時提供
  - ✓ 究極の反復型開発（進化型プロトタイピング）

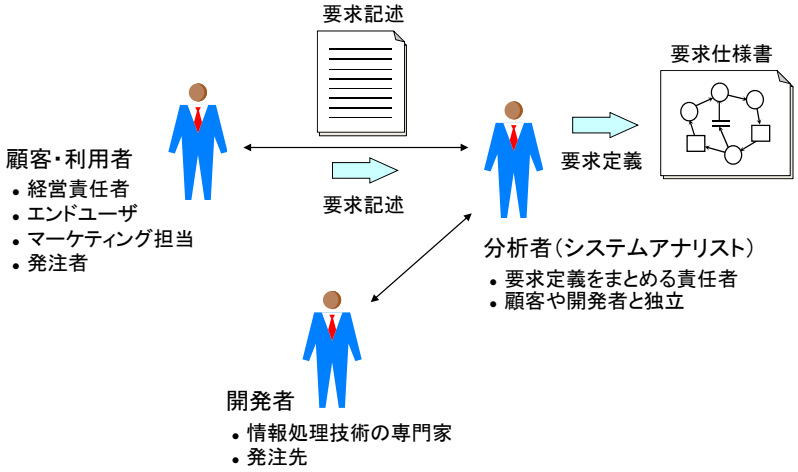


要求分析

要求分析

- 顧客や利用者の要求を見つけ仕様化する作業
  - ✓ 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(requirement)
  - ✓ 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：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

要求分析の関係者



要求分析の作業

1. 要求獲得(requirements acquisition/elicitation)
   - ✓ 利用者が真に望むものを引き出し、要求記述としてまとめる
     - 問題分析技法の適用
2. 要求仕様化(requirements specification)
   - ✓ 要求記述から要求仕様を作成する
     - 誤りや冗長を除去
     - 曖昧さ、矛盾を排除
     - 不足する情報を補足
3. 要求確認(requirements evaluation/verification/validation)
   - ✓ 作成された要求仕様の正しさを検査する
     - 要求レビューの実施

要求分析の課題

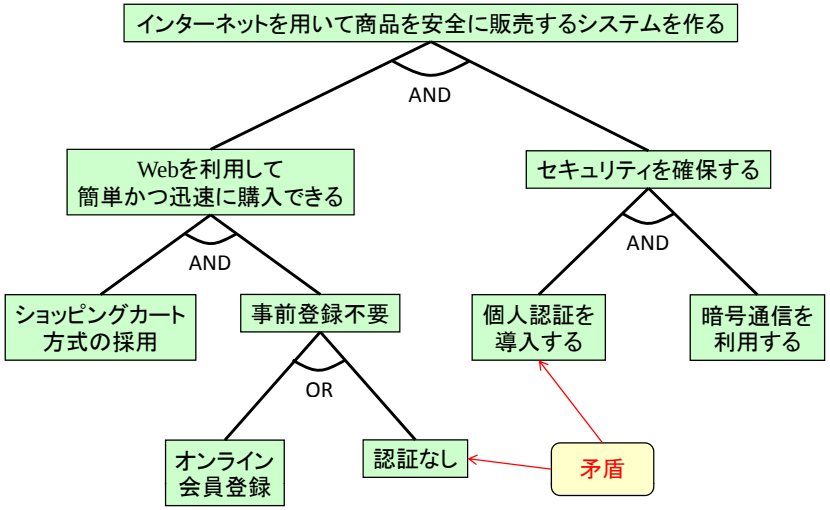
- 顧客や利用者の要求を正確かつ厳密に仕様に反映させることは困難

1. 顧客や利用者の要求の曖昧さ
   - ✓ 真の利用者を特定することが困難
   - ✓ 現状の業務形態やその問題に対する認識が不十分
2. 顧客や利用者の要求の変化
   - ✓ 初めから要求を完全に把握することは無理
     - 利用者自身も要求を明確に認識していない可能性
   - ✓ 際限ない要求
   - ✓ 市場や社会情勢、業務環境の変化
3. 利用者と開発者のコミュニケーションギャップ
   - ✓ 背景、知識、言葉の問題
     - 利用者は業務の専門家、開発者はシステム開発の専門家
   - ✓ 同じ仕様に対する解釈の違い

要求獲得

- 要求の抽出
  - ✓ 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の整理
  - ✓ 現場観察: 業務の体験や観察
  - ✓ プロトタイピング: 試供品の提供
  - ✓ シナリオ: 利用者の行動とそこから得られる事象の記述
  - ✓ ユースケース: 典型的な利用事例、シナリオを一般化したもの
  - ✓ 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): 問題のパターン化、機械の仕様と顧客や利用者の要求の明確な分離
- 要求の取捨選択
  - ✓ 交渉(negotiation): 合意形成による要求間の矛盾の解消
  - ✓ トリアージ(triage): 要求のふるい落とし

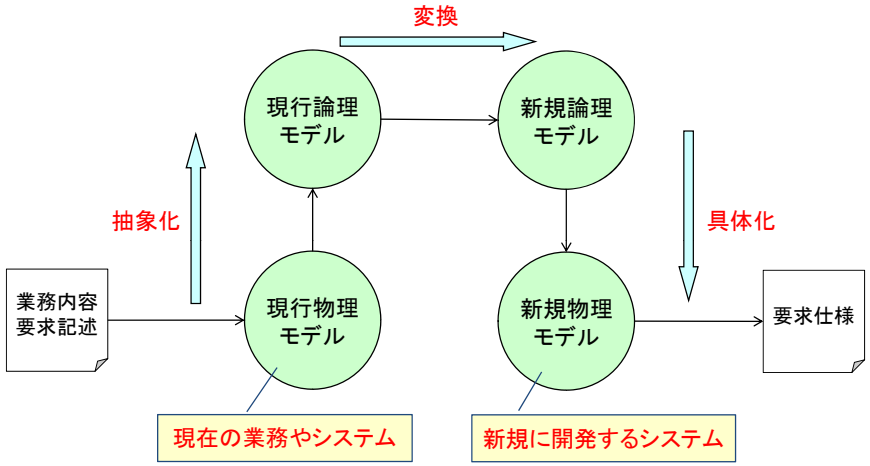
ゴール木の例



要求仕様化

- 図的表現
  - ✓ データフロー図(DFD: data flow diagram)
    - システム内部のデータの流れに着目した表記
  - ✓ IDEF0
    - 業務におけるアクティビティ(入力、出力、制約、機構)に着目した表記
  - ✓ UML (unified modeling language)
    - オブジェクト指向分析・設計で利用する統一的表記
- 形式仕様
  - ✓ 文法や意味が形式的(数学的)に定義された言語や図を利用
    - 誤り(漏れや曖昧さ)の排除
    - 機械的処理(解析や検査)の実現
  - ✓ 論理的仕様: 入出力条件を論理式で記述
  - ✓ 関数的仕様: 入出力データの変換を関数で記述
  - ✓ 代数的仕様: データとデータに対する演算の組で記述

構造化分析技法



論理的: どのような情報が必要であるかという要件(本質的)
物理的: その情報を得るための仕組みに対する要件(具体的)

構造化分析の手順

1. 現行物理モデルの構築
   現状の業務や現行システムの要求記述を分析
   - ✓ ありのまま(as-is)に具体的に表現
2. 現行論理モデルの構築
   本質的な機能や問題の洗い出し
   - ✓ 本質的でない部分(慣例的な処理や特定の事情による処理)を除外
     - 人名、媒体、日時、金額、数量など
3. 新規論理モデルの構築
   新たに開発するシステムで解決すべき問題の洗い出し
   - ✓ 追加あるいは削除する機能の特定
   - ✓ あるべき姿(to-be)を表現
4. 新規物理モデルの構築
   システムが満たすべき制約や性能を付与
5. 要求仕様の記述

要求確認

- 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: 要求仕様に基づいて作成された設計文書、ソースコード、マニュアルが容易に参照可能

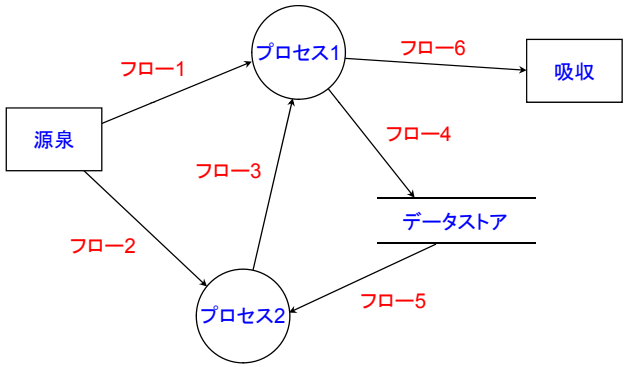
構造化分析

構造化分析

- ソフトウェアが動作するシステムの機能に着目した仕様化技法
- システムを段階的に詳細化
  - ✓ データフロー図(DFD: data flow diagram)
    - システム内の機能を、データを処理するプロセスで表現
    - 各プロセス間のデータの流れを表現
    - システムを階層的かつ図的にモデル化
  - ✓ 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

- 構文: 4つの基本記号のグラフ表現
- 意味: 各記号に付加された名称(情報)に依存



データフロー図

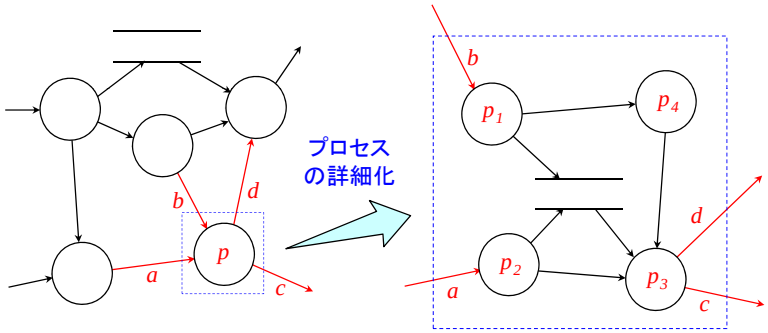
- プロセス(process)＝バブル(bubble)
  - ✓ 入力データから出力データへの変換処理を表現
  - ✓ 個々のプロセスにはその処理内容を表す名前を付与
- フロー(flow)
  - ✓ プロセス間の定常的なデータの流れ(移動)を表現
  - ✓ 矢印にデータの意味を表す名前を付与
- データストア(data store)＝ファイル(file)
  - ✓ データを格納する場所
  - ✓ 格納するデータの名称(入出力フローと同じ名称)を付与
    - 省略可
- エンティティ(entity)
  - ✓ システムの外部に存在する機器、組織、人間などを表現
  - ✓ 内容を表す名称を付与
    - 源泉(source): データの発生元
    - 吸収(sink): データの最終的な行き先

データフロー図の階層化

- 全体文脈図(context diagram)
  - ✓ データフロー図の記述の出発点(DFDレベル0)
  - ✓ システム全体を1つのプロセスで表現
  - ✓ システムと外界(エンティティ)とのデータのやり取りを表現
- 階層化の方法
  - ✓ プロセスの詳細化
    - プロセスの内部処理を詳細化(DFDレベル1, 2, 3, ...)
    - 1つのプロセスを複数のプロセスに分割
    - 分解したプロセス間のデータの流れを明確化
  - ✓ 矢印にデータの意味を表す名前を付与
    - 1つのデータを複数のデータ要素に分割

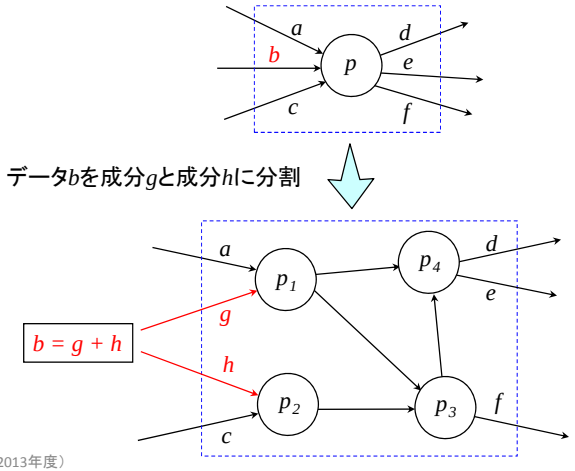
プロセスの詳細化

- 詳細化における約束
  - ✓ プロセスの入出力フローは、詳細化の前後で維持



データの詳細化

- 詳細化における約束
  - ✓ プロセスの入出力フローのデータは、詳細化の前後で維持



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能
- プロセスには1つ以上の入力フローと出力フローが必須
  - ✓ 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
- すべてのデータの流れを記述する
  - ✓ 条件による区別はできない

例題：要求記述

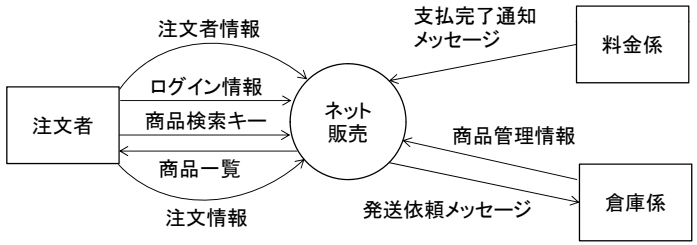
（出典：高橋，丸山：ソフトウェア工学．森北出版，2010年）

ネット販売業務の要求記述

- **注文者**は、事前に注文者情報を新規に登録する必要がある。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。新規登録において、会員管理係は、注文者情報を受け取ると、その注文者がすでに会員であるかを検査する。もし会員でなかったら、会員番号を発行し、注文者情報とともに会員情報ファイルに登録する。
- **注文者**は、会員情報を変更することができる。その際、**注文者**は会員管理係にログイン情報を送付して、会員認証に成功していなければならない。ログイン情報とは、注文者の会員番号とパスワードを合わせたデータをいう。
- 注文受付係は、**注文者**から受け取ったログイン情報により会員認証を行う。会員認証に成功した**注文者**だけが商品の検索や注文ができる。
- 注文受付係は、**注文者**から商品検索キーを受け取ると、商品管理ファイル内部の商品より該当する商品を検索し、それら商品一覧を**注文者**に返送する。
- **注文者**は、注文受付係に注文情報を送ることで希望する商品を注文できる。
- 注文受付係は、注文情報を受け取ると、その注文に関する支払完了通知メッセージの到着を待つ。**注文者**が商品の購入代金を**料金係**に支払うと、**料金係**は支払完了通知メッセージを注文受付係に送る。
- 注文受付係は、支払完了通知メッセージを受け取ると、それに該当する商品の発送情報を商品管理係に送る。
- 商品管理係は、発送情報を受け取ると、**倉庫係**に発送依頼メッセージを送る。
- **倉庫係**は、発送依頼メッセージを受け取ると、**注文者**に商品を送付する。
- 商品管理係は、**倉庫係**から商品管理情報を適時受け取り、商品管理ファイルに格納する。商品管理情報には、取扱商品情報や、それらの商品の入庫情報および出庫情報が含まれる。

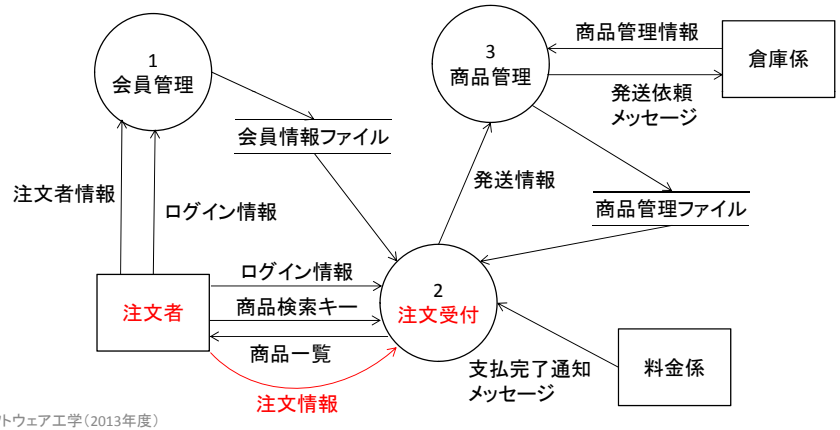
例題：全体文脈図

- プロセス
  - ✓ ネット販売
- エンティティ
  - ✓ 注文者、料金係、倉庫係



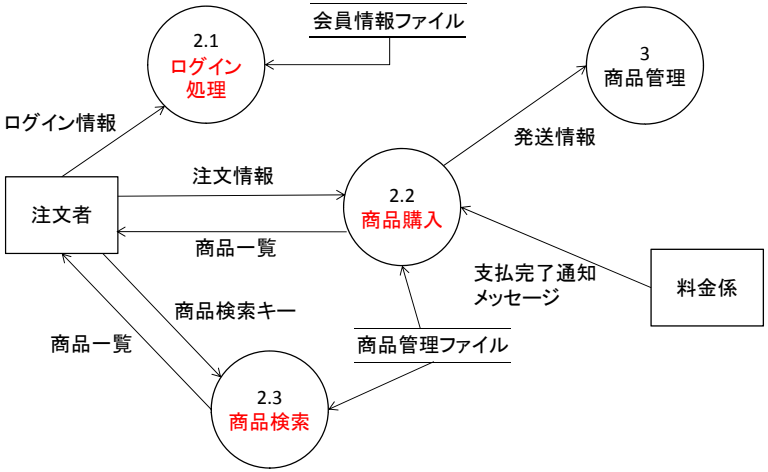
例題：DFDレベル1

- 「注文者は、注文受付係に注文情報を送ることで、希望する商品を注文することができる」
- 注文者 → 注文受付（データ：注文情報）



例題：DFDレベル2

- 「2 注文受付」プロセスを詳細化



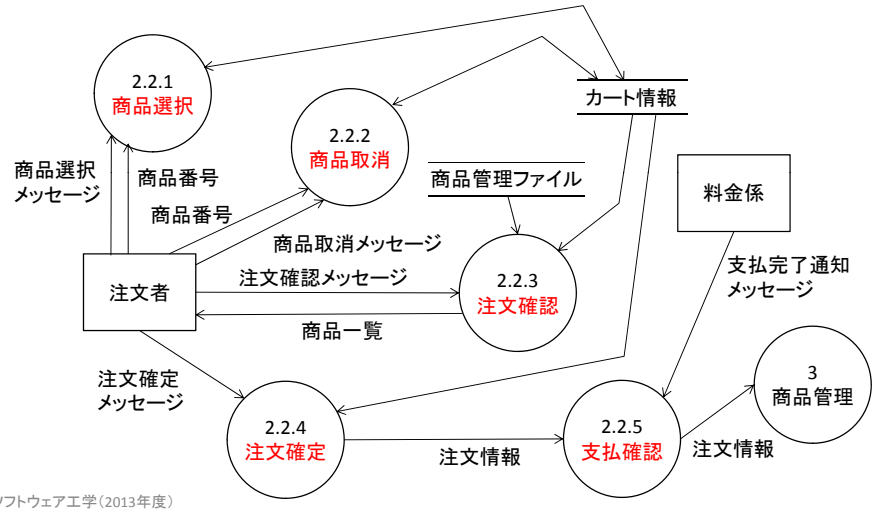
例題：要求記述の詳細化

ネット販売業務の要求記述（商品購入時の手順に関する追加）

- 注文情報は、商品選択情報（選択した商品の商品番号と商品選択メッセージ）、商品取消情報（選択を取り消す商品の商品番号と商品取消メッセージ）、注文確認メッセージ、注文確定メッセージに分けられる。
- 商品選択係は、注文者から商品番号を受け取ると、それをカート情報に追加する。
- 商品取消係は、注文者から商品番号を受け取ると、それをカート情報から削除する。
- 注文確認係は、注文者から注文確認メッセージを受け取ると、カート情報に存在する商品群に関する商品情報を商品管理ファイルから取り出し、注文者に商品一覧として送る。
- 注文確定係は、注文者から注文確定メッセージを受け取ると、カート情報に存在する商品群の商品番号を取り出し、それらを注文情報として支払確認係に送る。
- 支払確認係は、料金係から送られる支払完了通知メッセージを待つ。支払完了通知メッセージを受け取ると、注文確定係から受け取った注文情報を商品管理係に送る。

例題：DFDレベル3

- 「2.2 商品購入」プロセスを詳細化



プロセス仕様

- プロセス仕様(process spec) ＝ ミニ仕様(mini spec)
  - ✓ プロセス分割の終着点
  - ✓ DFDの最下層のプロセスの基本処理の内容を表現
    - 構造化言語：連接、選択、反復の3つの構造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

構造化言語による例（2.2.1 商品選択）

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

- 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(is equivalent to)
  - ✓ 連接：a+b; aとbからなる(and)
  - ✓ 選択：[a|b]; aまたはbのどちらかである(either-or)
  - ✓ 任意：(a); aはあってもなくてもよい(optional)
  - ✓ 反復：{a}; aを0回以上繰り返す(iterations of)
    - 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
    - （a, b：データ要素、n, m：整数）

データ辞書の例

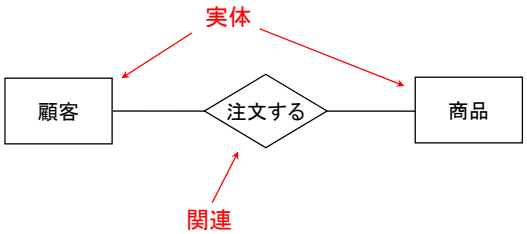
注文者情報 = 氏名 + 住所 + (電話番号) + 初期パスワード

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]

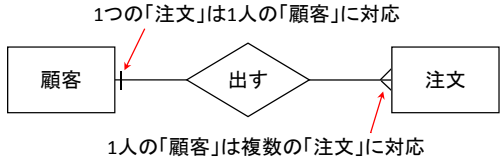
実体関連図

- データ構造やデータ間の関係を表現
- 実体(entity)
  - ✓ システム内に存在する管理対象（人、もの、金、場所など）
  - ✓ 名前を持つ長方形で記述
  - ✓ 抽象的な概念や目に見えないものでもよい
- 関連(relationship)
  - ✓ 実体間の相互の結びつき
  - ✓ 関連名を持つ菱形で表示

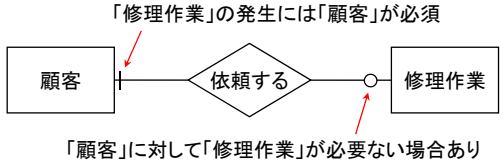


関連

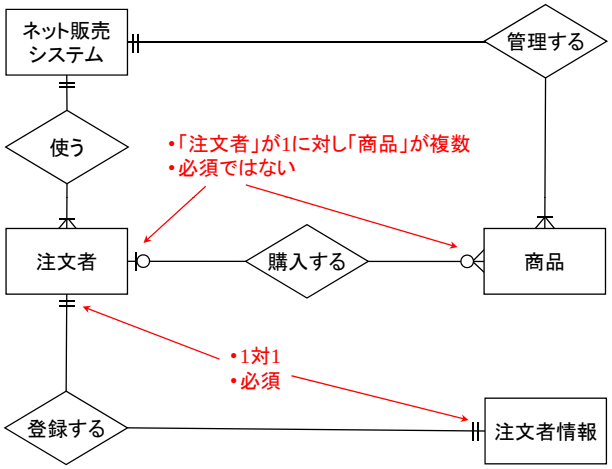
- カーディナリティ(cardinarity)
  関連するオブジェクトの数を表現（1:1, 1:N, M:N）



- モダリティ(modality)
  関連が必須であるかどうかを表現



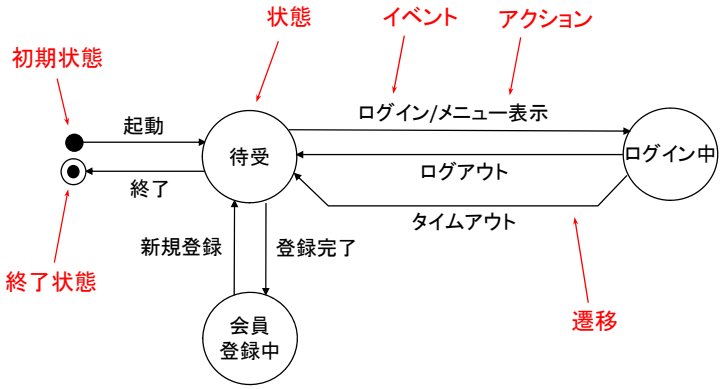
実体関連図の例



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、通信システムなどに関する要求の仕様化
- システム＝状態機械
  - ✓ 状態
    - 円あるいは四角形で表現
    - システムは必ず1つの状態に属する（複合状態を除く）
  - ✓ 遷移
    - 遷移を誘発するイベントを付与した矢印で表現
    - イベントに対する遷移先は現在の状態によって変化
    - イベントは一瞬、遷移時間は無視
  - ✓ アクション
    - 状態が遷移する際に実行される処理

状態遷移図の例



オブジェクト指向分析

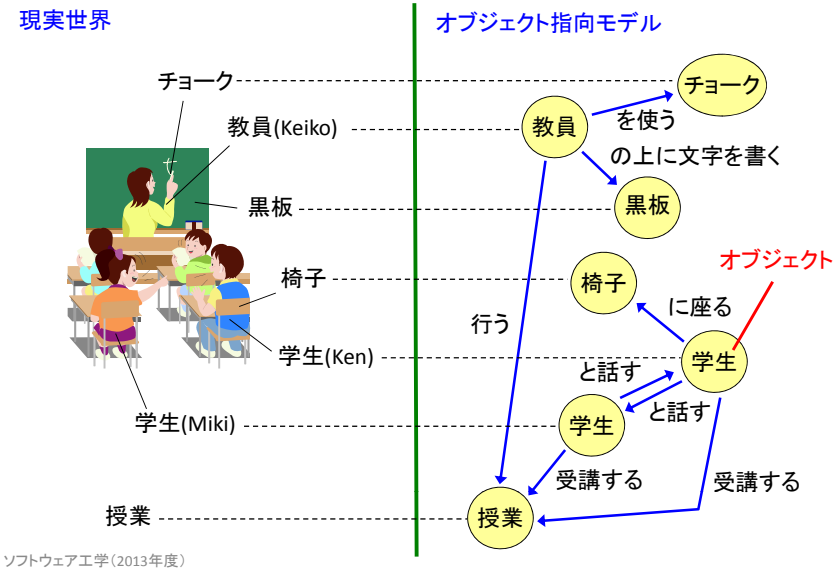
オブジェクト指向

- 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た技法
    → 人間にとって理解しやすい
  - ✓ オブジェクト指向言語を用いてそのまま実現可能

認知科学における概念
- ✓内包(intension)： 何ができるのか．機能による認知
- ✓外延(extension)： 何と似ているのか．分類による認知
- ✓属性(attribute)： 何からできているのか．構造による認知

- オブジェクトを構成単位としてソフトウェアを構築する仕組み
  - ✓ オブジェクトが中心

オブジェクト指向モデルの例

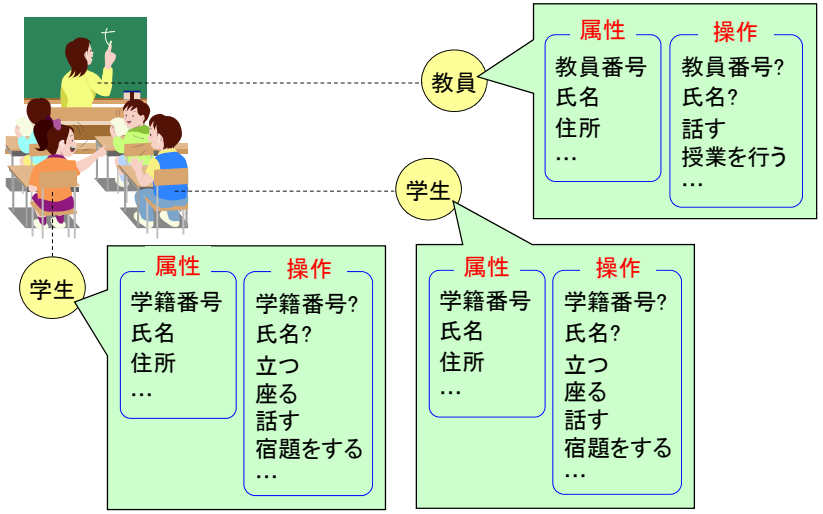


オブジェクト

- オブジェクト(Object)
  - ✓ 人間が認知できる具体的あるいは抽象的な「もの」
  - ✓ 実世界の「もの」や「役割」などの事柄(thing)を抽象化した「もの」
  - ✓ 物理的な「もの」、概念的な「もの」
  - ✓ 分析/設計者や開発するシステムに依存

- 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
    - 属性(attribute}、プロパティ(property)
  - ✓ 振る舞い(behavior)
    - オブジェクトが実行できる動作
    - 操作(operation)、メソッド(method)
  - ✓ 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



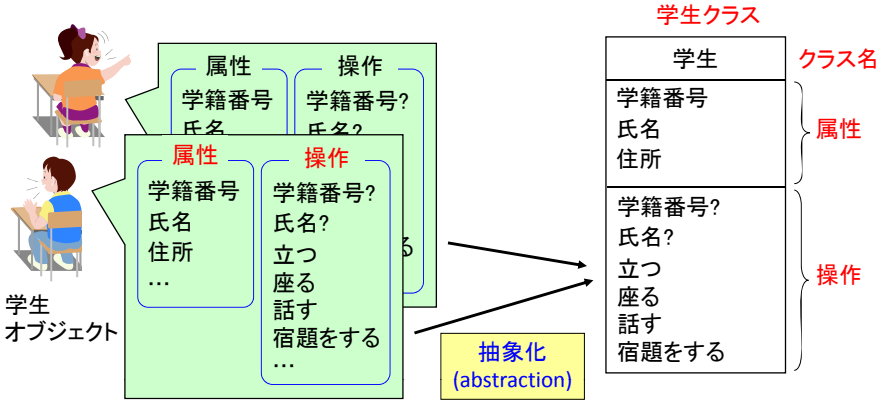
オブジェクト指向の基本概念

- クラスとインスタンス
- 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)
  - ✓ あるオブジェクトが別のオブジェクトを利用することを表す、オブジェクト間の関係
- 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)
  - ✓ ある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

など

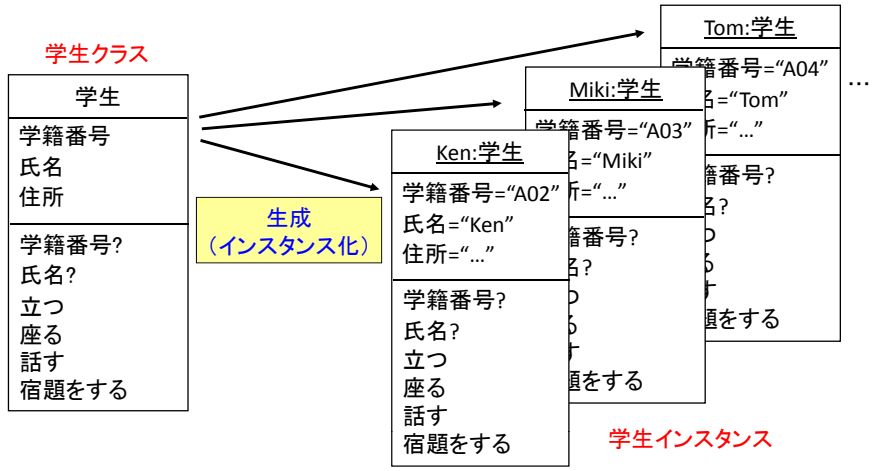
クラス

- 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
- オブジェクトの設計図

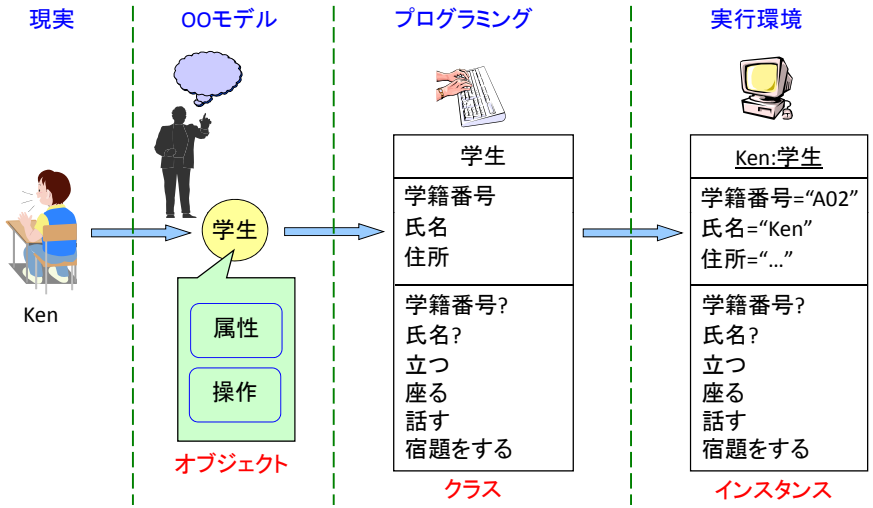


インスタンス

- クラスから生成されたオブジェクト



オブジェクト、クラス、インスタンス



オブジェクト指向開発プロセス

- オブジェクト指向では、分析、設計、実装など開発のあらゆる段階でオブジェクト(あるいはクラス)に着目→ 各工程を行き来することが容易
  - ✓ オブジェクト指向分析(OOA: OO analysis)
    - 開発するシステムを定義
    - 作成するモデルは環境に非依存
  - ✓ オブジェクト指向設計(OOD: OO design)
    - 定義したシステムをソフトウェアとして実現
    - 環境に依存する部分を考慮
  - ✓ オブジェクト指向プログラミング(OOP: OO programming)
    - プログラムの記述とテスト
- 反復型ソフトウェア開発プロセス
  - ✓ イテラティブ
  - ✓ インクリメンタル

UML(Unified Modeling Language)

- 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との関係
	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	オブジェクトの相互作用のタイミング
	相互作用概念図	シーケンス図とアクティビティ図の概要

オブジェクト指向モデリングにおける観点

- 機能的観点
  - ✓ システムの機能を定義
    - ユースケース図
- 静的観点
  - ✓ システムの構造を定義
    - クラス図、オブジェクト図
- 動的観点
  - ✓ システムの振る舞いを定義
    - シーケンス図、コミュニケーション図、アクティビティ図、状態機械図
- 物理的観点
  - ✓ システムの物理的制約を定義
    - コンポーネント図、配置図

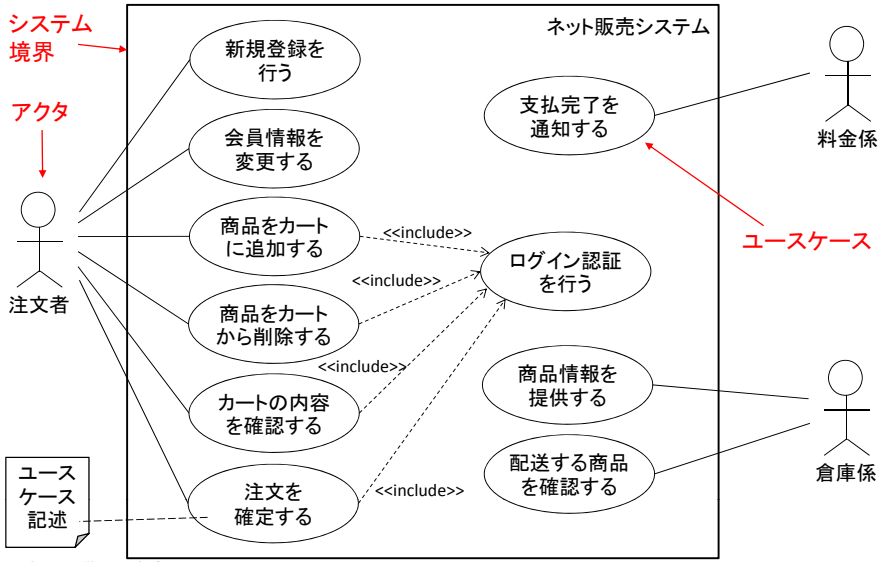
オブジェクト指向分析

- システムを構成するオブジェクトの構造や振る舞いを明確にすることで要求を仕様化する技法
- データ駆動型アプローチ
  - ✓ システム内のデータをはじめに認識する
- 責任駆動型アプローチ
  - ✓ システムの振る舞い（責任）をはじめに認識する
  1. ユースケース（利用方法）の抽出
  2. 構造の記述
     - システム内の登場人物（オブジェクト、クラス）の特定
     - クラス間の関係の特定
  3. 振る舞いの記述
     - システム全体の作業の流れの明確化
     - 個々のオブジェクト間の相互作用（メッセージの流れ）の明確化
     - 個々のオブジェクトの状態の明確化
  4. モデルの洗練

ユースケース図

- システムの機能ごとに作成
- 利用者側（アクタ）から見たシステムの使われ方を表現したもの
  - ✓ アクタ(actor)
    - システムに対して利用者が果たす役割(role)
    - 役割ごとに異なるアクタが存在
    - 外部システムでもよい
    - 受益者でなければならない
- 利用者の目的に照らして結び付けられた一群のユースケース記述（シナリオ）を持つ
  - ✓ シナリオ(scenario)
    - 利用者とシステム間の対話を表す一連の手順
    - ユースケースのインスタンス
- 要求の獲得、反復計画、妥当性検査に適用

ユースケース図の例



ユースケース記述の例

名称　　注文を確定する
開始アクタ　注文者
目的　　カートに存在する商品を購入する
事前条件　注文者はログイン認証に成功している

正常処理シナリオ
1. 注文者が、注文確定ボタンを押す。
2. システムは、カート内の商品の合計金額を計算する。
3. システムは、カートの内容と合計金額を表示する。
4. 注文者が、クレジットカード番号を入力し、購入ボタンを押す。
5. システムは、料金係に注文者の支払い状況を確認する。
6. 支払が成功すると、支払完了を通知する。
7. システムは、倉庫係に商品の発送を依頼する。
8. システムは、カートを空にする。
9. システムは、商品の在庫を更新する。

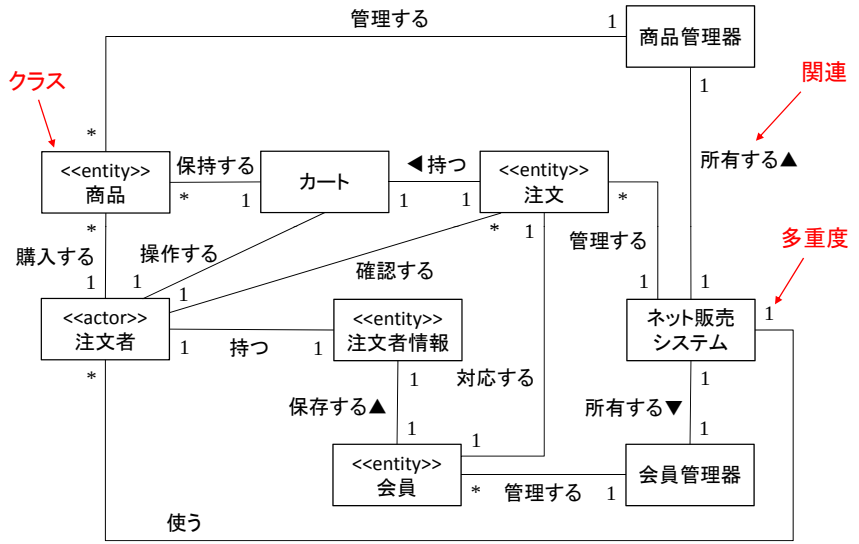
例外処理
5で支払が失敗する。
  a. システムは支払失敗メッセージを表示する。

構造の記述

- システムに登場するクラスを抽出
  - ✓ シナリオや要求仕様に出現する名詞が対応することが多い
- クラス間に存在する関連を抽出
  - ✓ シナリオや要求仕様に出現する動詞が対応することが多い
- 抽出したクラスと関連から大まかなクラス図（概念モデル）を作成
- 各クラスの属性と操作を抽出してクラス図を完成
  - ✓ 操作が重要

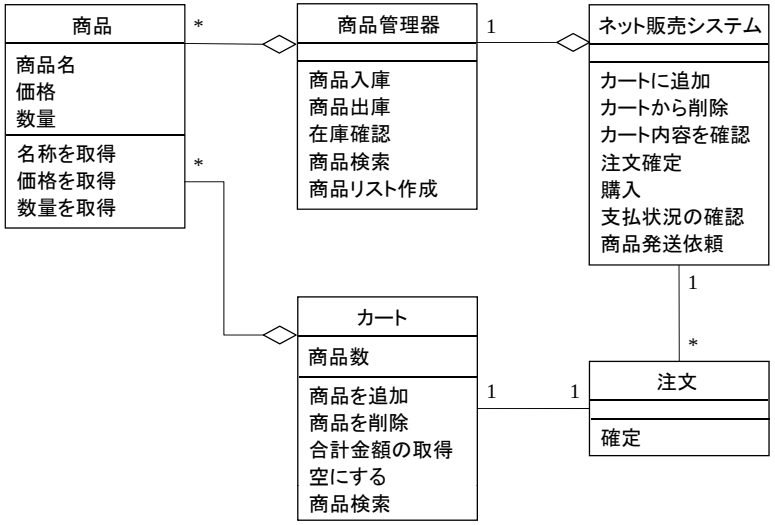
概念モデルの例

（注文に関する一部のみ）



クラス図の例

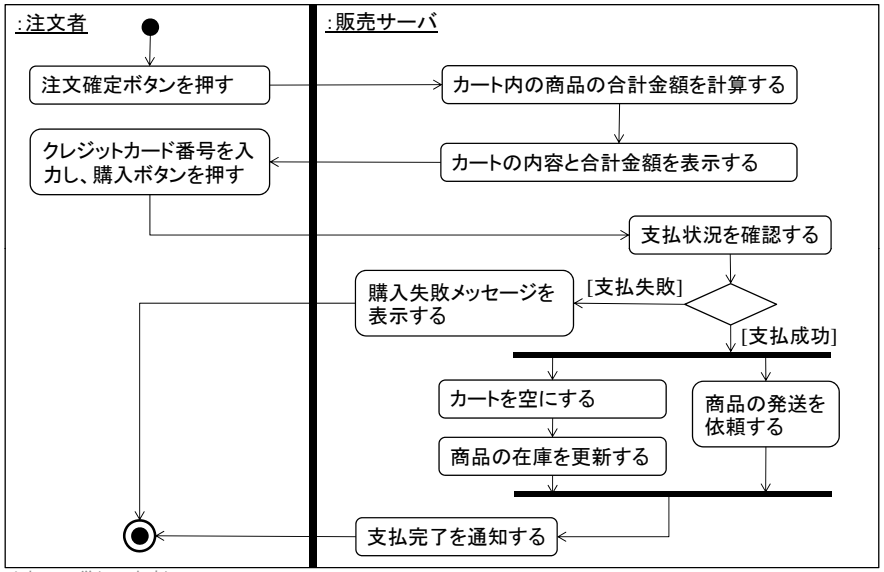
（注文に関する一部のみ）



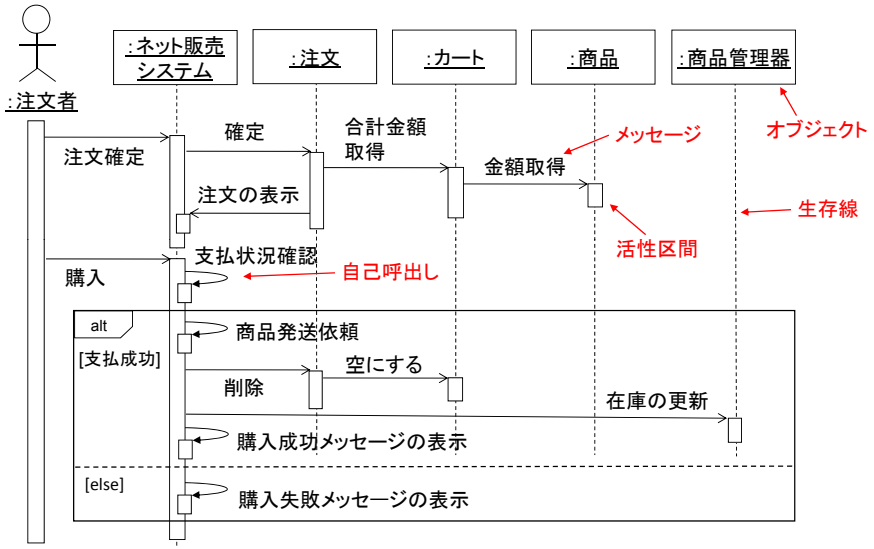
振る舞いの記述

- システム全体の作業の流れ
  - ✓ アクティビティ図
    ある機能を実現するために必要な作業の順序を示す
- オブジェクト間のメッセージ送受信の時系列
  - ✓ シーケンス図
    オブジェクト間のメッセージの流れを示す
- 個々のオブジェクトの動作、状態
  - ✓ 状態図

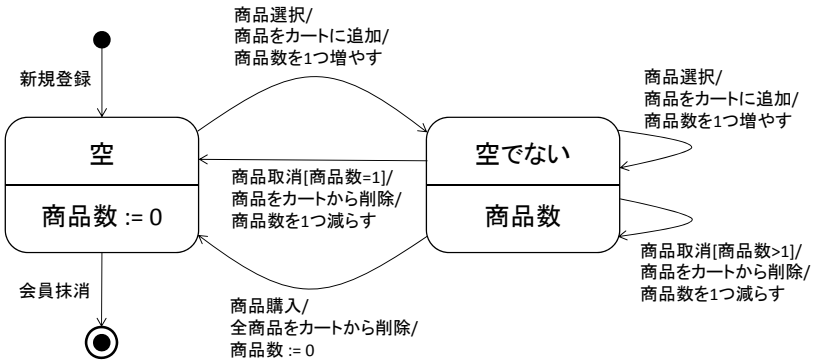
アクティビティ図の例



シーケンス図の例



状態図の例



ソフトウェア設計

ソフトウェアアーキテクチャ

- ソフトウェアの骨格となる基本構造、基本設計、設計思想
- 品質特性（非機能的な側面）に大きな影響
  - ✓（例）家の基本構造と特性

基本構造	耐火性	耐衝撃性	建設期間
わら造り	弱	弱	短
木造	中	中	中
レンガ造り	強	強	長

  - ✓ 基本構造から大きく外れる改造は困難
  - ✓ トレードオフ（セキュリティ/使用性など）の検討
- 開発の方向付け
  - ✓ 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

アーキテクチャに関わる品質特性の例

- 実行時の品質特性
  - ✓ 効率性/時間的効率性（応答時間）
    - ネットワークの通信時間 vs. データ探索時間
- 開発時の品質特性
  - ✓ 保守性/変更性
    - 商品管理ファイルの変更を局所化
- 運用時の品質特性
  - ✓ 可搬性/順応性
    - 検索コンピュータを設定ファイルで指定

アーキテクチャ設計

- ソフトウェアアーキテクチャを決定する作業
  - ✓ 性能や変更容易性などの非機能面から要求を分析
  - ✓ 設計者に強く依存する
    - 設計者の知識、経験、スキル、直感
- 設計上の課題
  - ✓ 設計の基本方針
    - 開発期間、開発費用、開発形態、利用期間、利用形態、開発後の取り扱いなど評価基準の明確化
  - ✓ 評価の視点
    - 評価する視点を定め、その視点に基づく構造の表現（ビュー）
  - ✓ 評価の追跡可能性
    - トレードオフに対する判断の記録

アーキテクチャに対する視点

- ソフトウェアアーキテクチャを捉える4つの視点
- 論理ビュー
  - ✓ システムがどのような機能を持つか
  - ✓ 機能がどのような論理構造で実現されるか
- 実行ビュー
  - ✓ システムがプロセスやタスクなどの実行単位からどのように構成されるか
  - ✓ 性能など実行時の品質特性に着目
- 開発ビュー
  - ✓ システムがどのようなファイル、ディレクトリ構造で管理されるか
  - ✓ 保守性や修正容易性などの品質特性に着目
- 配置ビュー
  - ✓ システムを配置したときの構造
    - プロセスなどの実行単位をどの計算機に配置するか
  - ✓ 信頼性、安全性、可搬性などの品質特性に着目

アーキテクチャの設計手法

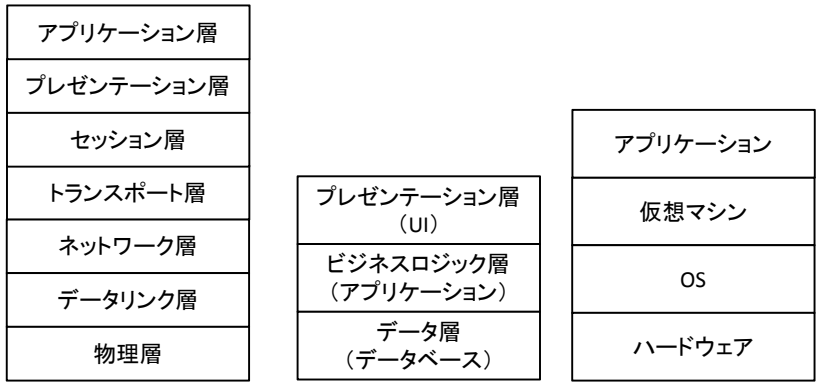
- 標準化された設計手法、定着した手法は存在しない

- Boschの手法
  1. 機能面からアーキテクチャを設計
     機能に関する要求を満たすようにアーキテクチャを設計する
  2. 品質特性を評価
     設計したアーキテクチャが非機能的な要求を満たすか評価する
  3. 要求された品質特性を満たさなければアーキテクチャ変換
     アーキテクチャスタイルを用いて要求を満たすようにアーキテクチャを変換

アーキテクチャスタイル

- 代表的なソフトウェアの基本的構造
- 多数のソフトウェアに繰り返し出現する構造
  - ✓ 抽象化、共通化、カタログ化
- 機能の分割と配置に基づく分類
  - ✓ 階層モデル
    機能を上位から下位に層状に並べて配置したモデル
  - ✓ クライアントサーバモデル
    データと処理機能をサーバとクライアントに分けて配置する分散システムモデル
  - ✓ リポジトリ（データ中心）モデル
    複数のサブシステムが共有データを介してデータ交換しながら処理を行うモデル
- データとコントロールの流れに基づく分類
  - ✓ データフローモデル
  - ✓ コントロールモデル
    - 集中型：コールリターンモデル、マネージャモデル
    - イベント駆動型：ブロードキャストモデル、割り込み駆動型モデル

階層モデル

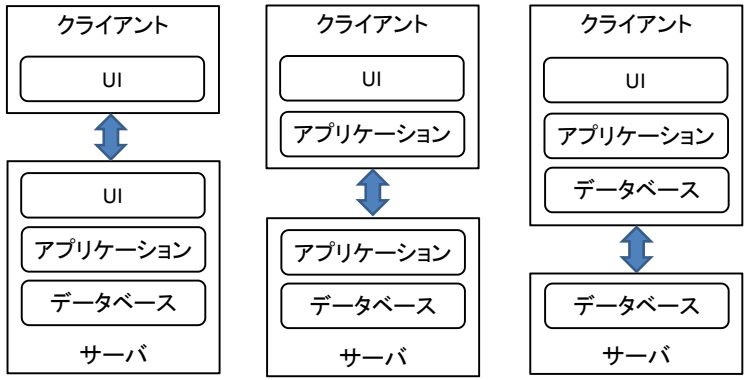


ISO 7層モデル

Webアプリケーション3層モデル

仮想マシンモデル

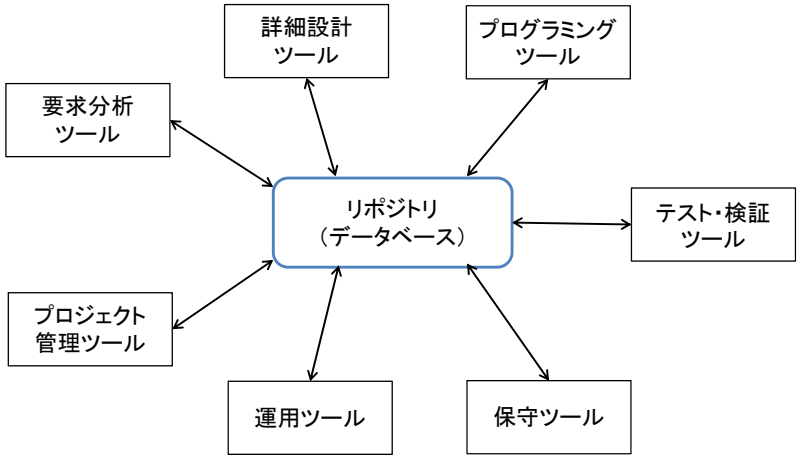
クライアントサーバモデル



シンクライアントサーバモデル

ファットクライアントモデル

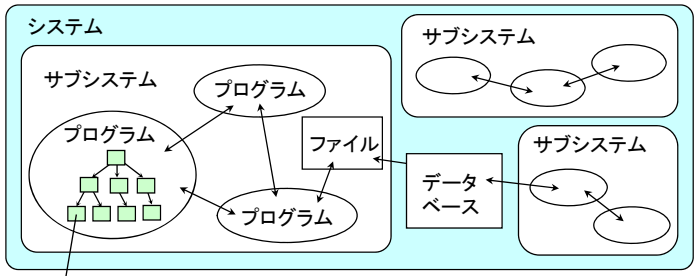
リポジトリモデル



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、クラスなど
- 利点
  - ✓ 抽象化の観点
    - 詳細を把握しなくとも利用可能
  - ✓ 開発効率の観点
    - 並行に開発可能
  - ✓ 再利用の観点
    - 既存のモジュールを再利用可能
  - ✓ 変更容易性の観点
    - 変更範囲を局所化

モジュール



モジュール

(1) 複数の文で構成され、独立して識別可能な名前をもつ

(2) コンパイルが別々にできる

(3) 決められたインタフェースを通してのみ呼出可能である

設計技法

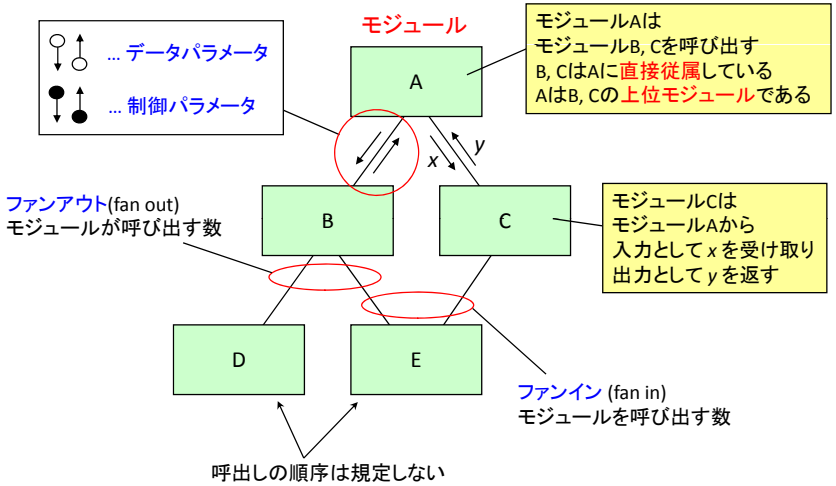
- データの流れに基づく構造化設計
  データの流れに着目して機能を分割することでプログラム構造を決定
  - ✓ 複合設計[Myers]
  - ✓ 構造化プログラミング[Dijkstra]
- データの構造に基づく構造化設計
  システムの入出力データの構造に着目してプログラム構造を決定
  - ✓ ジャクソン法[Jackson]
  - ✓ ワーニエ法[Warnier]
- オブジェクト指向設計
  データと操作をカプセル化したクラス（オブジェクト）をモジュールとしてプログラム構造を決定

構造化設計(structured design)

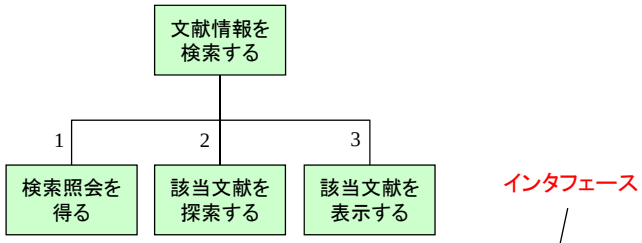
- システム機能をトップダウンで詳細化し、機能の階層構造を作成

- システム設計仕様書(system design specification)
  - ✓ モジュール構造図(module structure diagram)
    - システムを実現するモジュールの構成（静的関係）を規定
    - モジュールの呼び出し関係や入出力データのやり取りを表現
  - ✓ モジュール機能仕様書(module function specification)
    - 個々のモジュールの機能を規定
  - ✓ モジュールインタフェース仕様(module interface specification)
    - モジュールを外部から呼び出すときのインタフェースを規定

モジュール構造図



モジュール構造図の例

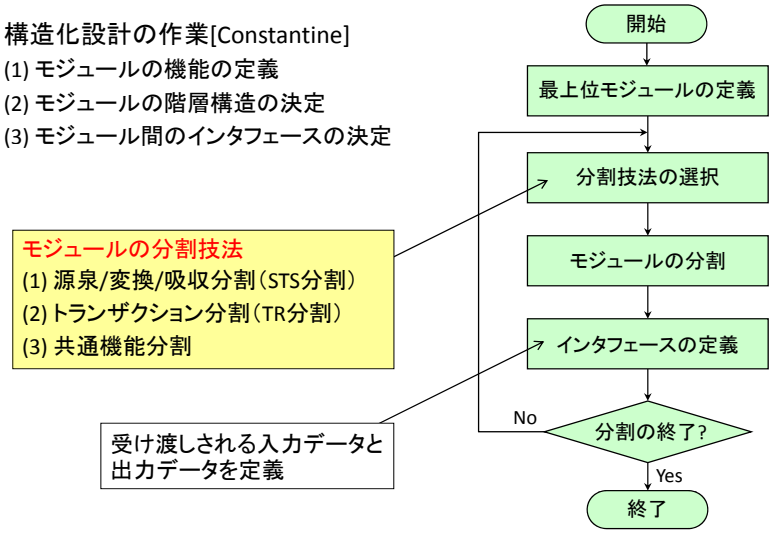


インタフェース

	入力	出力
1	なし	検索照会、端末アドレス、エラーコード
2	検索照会	該当文献、エラーコード
3	該当文献、エラーコード、端末アドレス	エラーコード

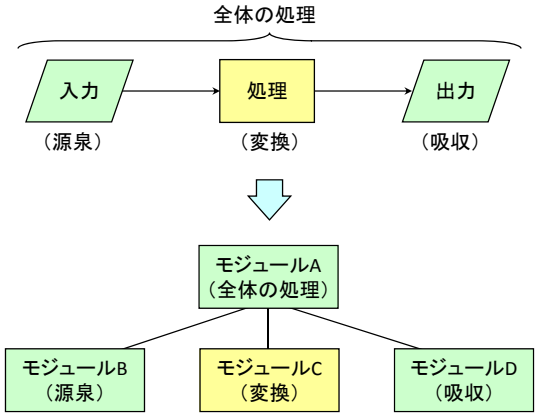
構造化設計の手順

- 構造化設計の作業[Constantine]
  (1) モジュールの機能の定義
  (2) モジュールの階層構造の決定
  (3) モジュール間のインタフェースの決定



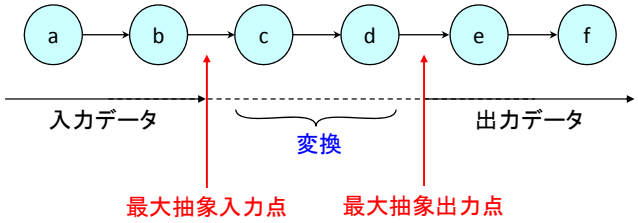
源泉/変換/吸収分割

- 源泉/変換/吸収分割（STS分割：Source/Transform/Sink decomposition）
  - ✓ 機能を入力から出力への変換とみなして分割



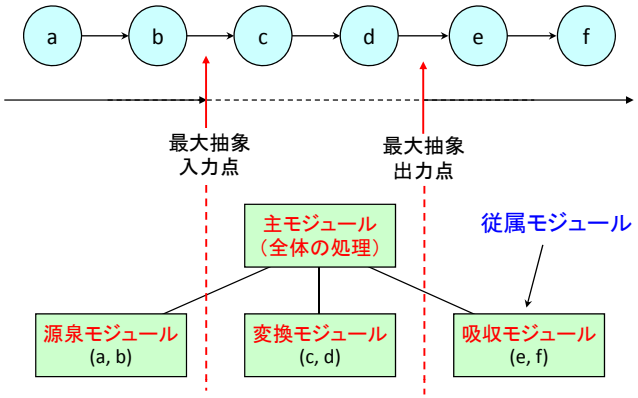
STS分割の手順(1)

1. データフロー図において主要なデータの流れを抽出
2. 最大抽象点の発見
   - ✓ 最大抽象入力点
     - 入力側から入力データを順方向にたどり、入力データとはいえなくなる点
   - ✓ 最大抽象出力点
     - 出力側から出力データを逆方向にたどり、出力データとはいえなくなる点



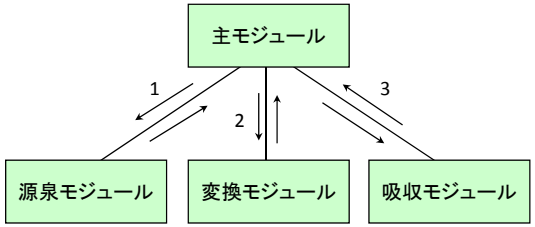
STS分割の手順(2)

3. 従属モジュールの定義
   - ✓ 最大抽象入力点及び出力点を区切りに源泉/変換/吸収部分に分割する



STS分割の手順(3)

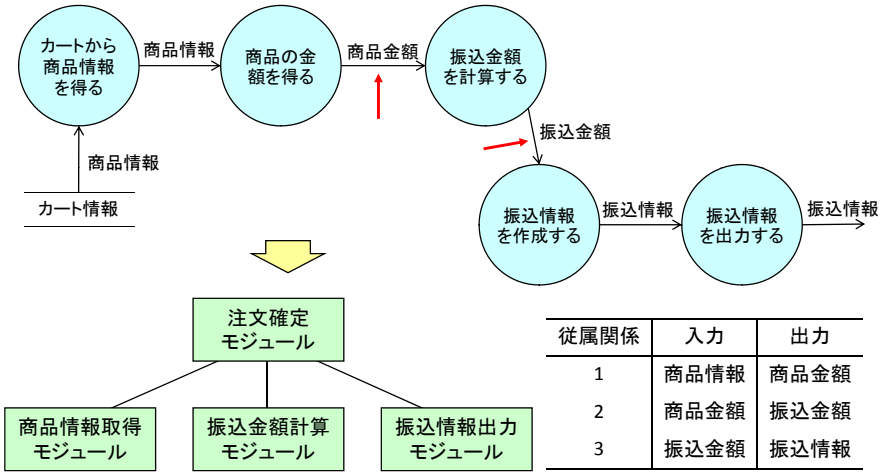
4. モジュール間インタフェースの定義
   - ✓ 下位モジュールを対象にその入出力データを決定する



従属関係	入力	出力
1	なし	最大抽象入力データ
2	最大抽象入力データ	最大抽象出力データ
3	最大抽象出力データ	なし

STS分割の例

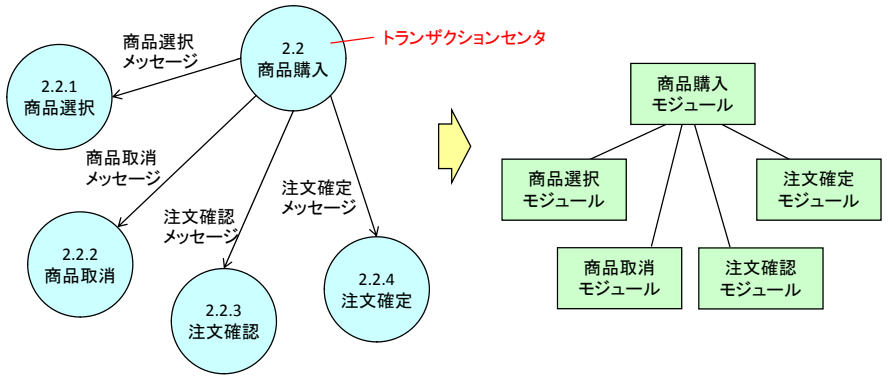
- 「2.2.4 注文確定」プロセスに着目



従属関係	入力	出力
1	商品情報	商品金額
2	商品金額	振込金額
3	振込金額	振込情報

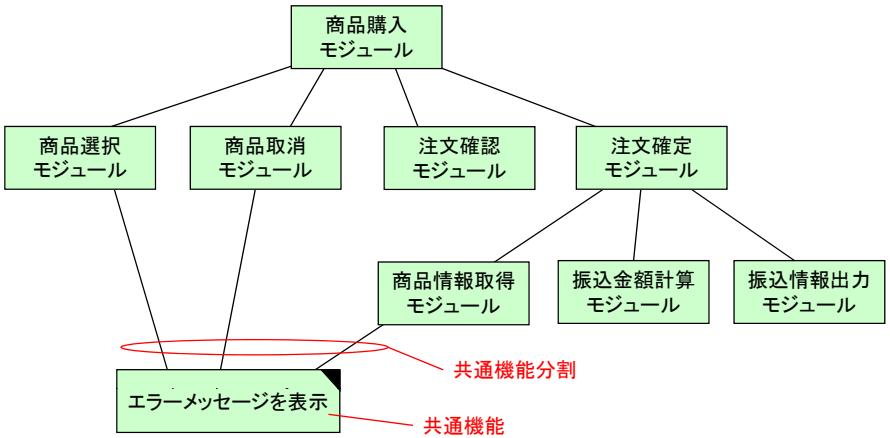
トランザクション分割

- トランザクション分割（TR分割：transactional decomposition）
  - ✓ トランザクション処理ごとに分岐先モジュールを定義する
    トランザクション：データに対する切り離せない一連の処理の単位
    （例）データベースの読み込みと更新操作のまとまり

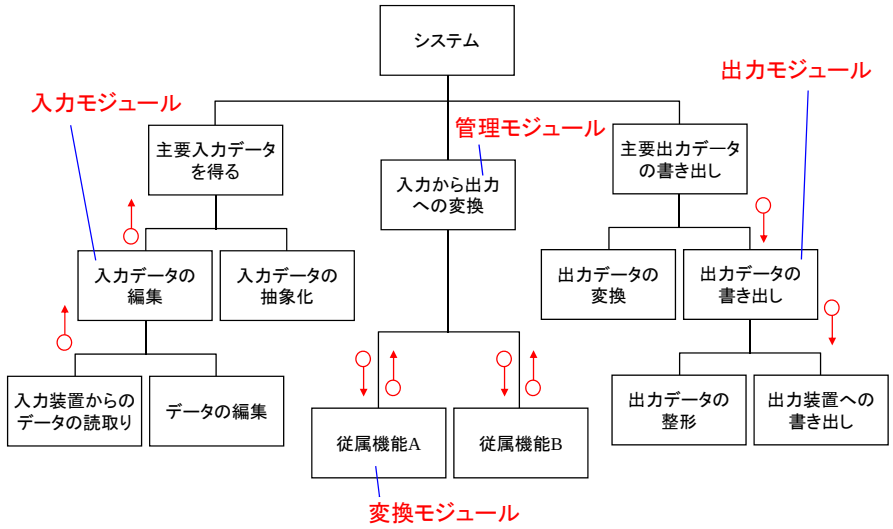


共通機能分割

- 共通機能分割(functional decomposition)
  - ✓ 複数のモジュールに含まれる共通の従属機能を取り出して定義



モジュールの一般的構造



設計の評価基準

- 分割の観点
  - ✓ モジュールの大きさ
    - モジュールを構成する文の数
  - ✓ モジュールの簡潔さ
    - 汎用化の尺度
- 独立性の観点
  - ✓ 機能独立性

    1つの目的に沿った機能のみ提供し、他のモジュールとの相互作用が少ない
  - ✓ 情報隠蔽（カプセル化）

    モジュールのインタフェースと実装を分離
  - ✓ モジュール強度（凝集度）(module strength/cohesion)
    - 同じモジュール内に存在する構成要素の関連の程度
  - ✓ モジュール結合度(module coupling)
    - 異なるモジュール間に存在する構成要素の関連の程度

モジュール強度

(1) 暗号的強度、偶発的強度(coincidental strength)
- ✓ 特定の機能を持たず、偶然に集められたモジュール

(2) 論理的強度(logical strength)
- ✓ 見かけ上は同一の機能を持つが、実際には多様な機能を集めたモジュール

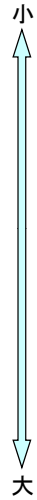
(3) 時間的強度(temporal strength)
- ✓ 実行されるタイミングが近い機能を集めたモジュール

(4) 手順的強度(procedural strength)
- ✓ 逐次的に実行される関連のある機能を集めたモジュール

(5) 連絡的強度(communication cohesion)
- ✓ 手順的強度で、同じデータを入力あるいは出力する機能を集めたモジュール

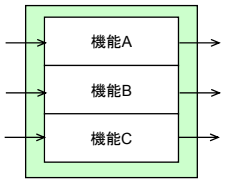
(6) 情報的強度(informational strength)
- ✓ 同じデータにアクセスする複数の機能を集めたモジュール

(7) 機能的強度(functional strength)
- ✓ 単一の機能を実行するモジュール

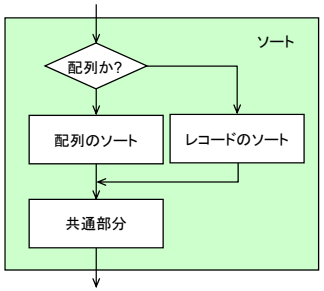


モジュール強度の例(1)～(3)

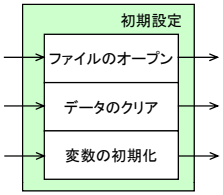
(1) 暗号的強度



(2) 論理的強度

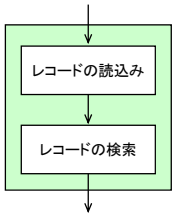


(3) 時間的強度

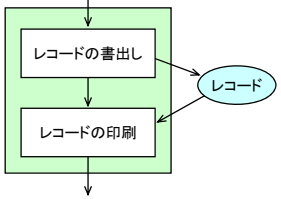


モジュール強度の例(4)～(7)

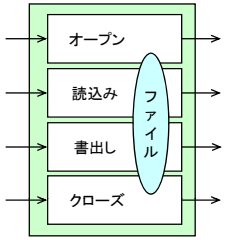
(4) 手順的強度



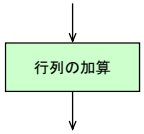
(5) 連絡的強度



(6) 情報的強度



(7) 機能的強度



モジュール結合度

(1) 内容結合(content coupling)
- ✓ 一方のモジュールが他方のモジュールの内容を直接参照する

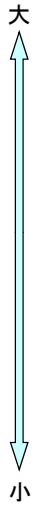
(2) 共通結合(common coupling)
- ✓ モジュール同士が共通データ領域にあるデータを参照する

(3) 外部結合(external coupling)
- ✓ モジュール同士が外部宣言されたデータを共有する

(4) 制御結合(control coupling)
- ✓ 呼出しモジュールが呼び出されたモジュールの制御を引数を通して指示する

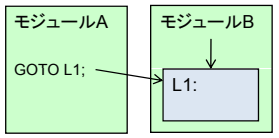
(5) スタンプ結合(stamp coupling)
- ✓ 共通データ領域にないデータの構造体を引数として受け渡す
- ✓ 呼び出されたモジュールで不要なデータが構造体に含まれる

(6) データ結合(data coupling)
- ✓ 必要なデータだけを引数として受け渡す

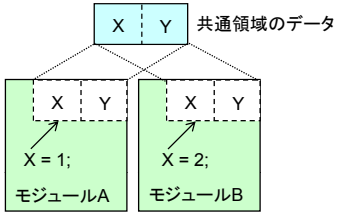


モジュール結合の例(1)～(3)

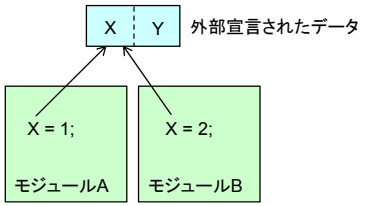
(1) 内容結合



(2) 共通結合

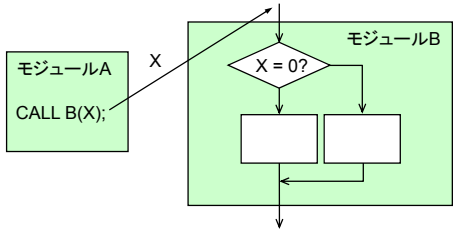


(3) 外部結合

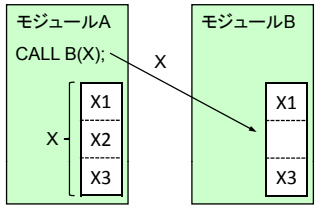


モジュール結合の例(4)～(6)

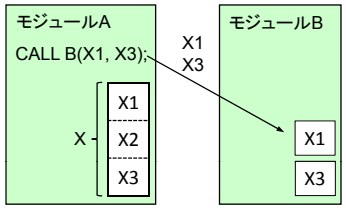
(4) 制御結合



(5) スタンプ結合



(6) データ結合



オブジェクト指向におけるクラスの設計原則

- クラス
  - ✓ 原則的に情報的強度を持つモジュール
- 単一責任の法則(SRP: single responsibility principle)
  - ✓ クラスを変更する理由は1つでなければならない
- オープン・クローズドの原則(OCP: open-closed principle)
  - ✓ 拡張に対してオープンで、修正に対してクローズでなければならない
- リスコフの置換原理(LSP: Liskov substitution principle)
  - ✓ サブクラスはそのスーパークラスで置換可能でなければならない
- 依存関係逆転の法則(DIP: dependency inversion principle)
  - ✓ 上位のモジュールは下位のモジュールに依存してはいけない
  - ✓ 抽象は実装の詳細に依存してはいけない
- インタフェース分離の法則(ISP: interface segregation principle)
  - ✓ 強い関連性を持つインタフェースのみをまとめてグループ化しなければならない

パッケージの設計原則

- パッケージの凝集度に関する設計原則
  - ✓ パッケージ
    - クラスをまとめて整理したもの
    - オブジェクト指向設計ではモジュールとみなせる
- 再利用・リリース等価の原則(reuse-release equivalence)
  - ✓ パッケージはリリースの単位で再利用されなければならない
- 全再利用の原則(common-reuse)
  - ✓ パッケージ内の全クラスが再利用されなければならない
- 閉鎖性共通の原則(acyclic-dependencies)
  - ✓ 1つの変更理由は単一パッケージに閉じ込められていなければならない
- 安定依存の原則
  - ✓ より安定しているパッケージに依存しなければならない
- 安定度・抽象度等価の原則(stable-abstraction)
  - ✓ 抽象的なパッケージほど安定していなければならない

データ構造に基づく設計

- システム内のデータの基本的な特性に着目
  - ✓ 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

    cf. プロセス中心アプローチ(POA: process-oriented approach)
    - 業務（プロセス）の変更に弱い
- データ
  - ✓ 現実世界に存在する実体
    - 「学生」データは「学籍番号」と「氏名」から構成
  - ✓ 特定のシステムや処理とは独立した存在
    - 「学生」データはその処理に無関係
- 代表的手法
  - ✓ ジャクソン法[Jackson]
  - ✓ ワーニエ法[Warnier][Orr]

ジャクソン法

- 入力データ構造と出力データ構造の対応関係からプログラムの論理構造を導出

手順

1. データ構造の定義
   - ✓ 入力データ構造と出力データ構造を分析し、4つの構成要素（基本、連続、反復、選択）でデータ構造図を作成
2. データ構造の対応付け
   - ✓ 入力データ構造図と出力データ構造図の構成要素間の対応関係を決定
3. プログラム論理構造の決定
   - ✓ 出力データ構造を基に論理構造を定義
     - 入力データから出力データへ変換する処理を記述
   - ✓ 構造が不一致のとき、中間のデータ構造を導入
     - 入出力データに対する処理を追加

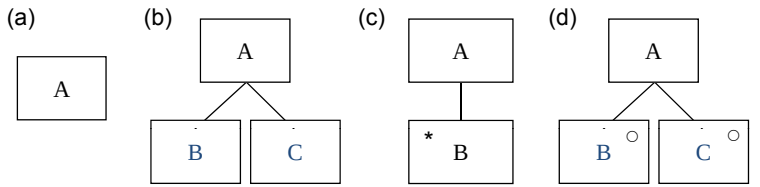
データ構造図の構成要素

(a) 基本
- ✓ これ以上分割できない構成要素（1つのデータ項目に対応）

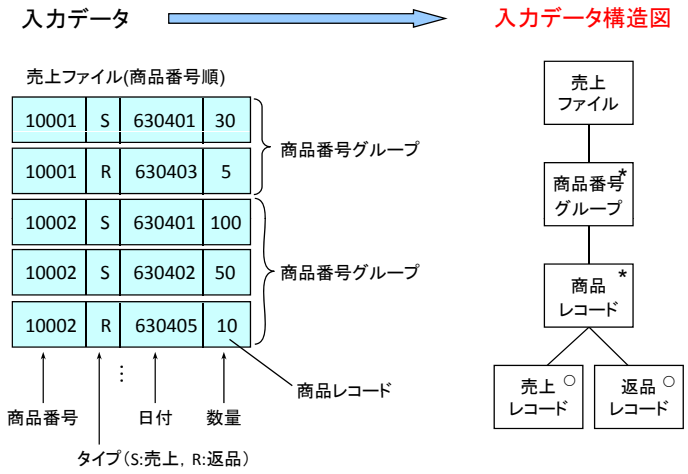
(b) 連続
- ✓ 異なる複数の基本要素からなる構成要素（複数の項目を持つレコードに対応）
- ✓ それぞれの構成要素は左から右に順に1度だけ出現

(c) 反復
- ✓ 同一の構成要素が繰り返し現れる構成要素

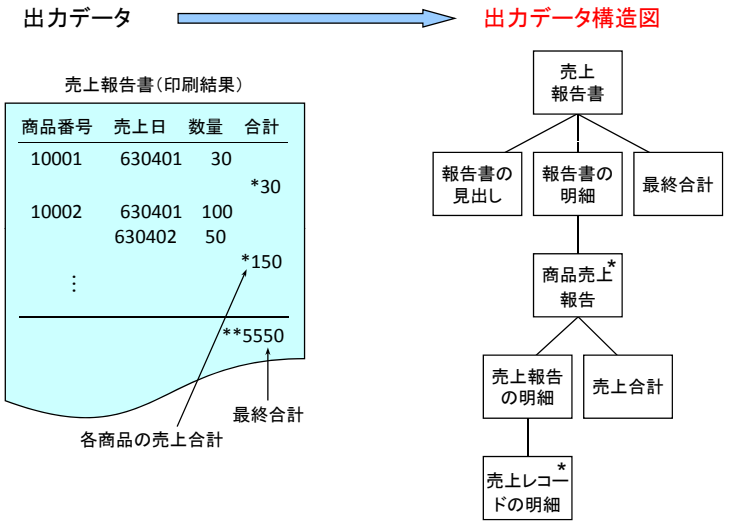
(d) 選択
- ✓ 複数の構成要素のうちどれか1つを選択する構成要素



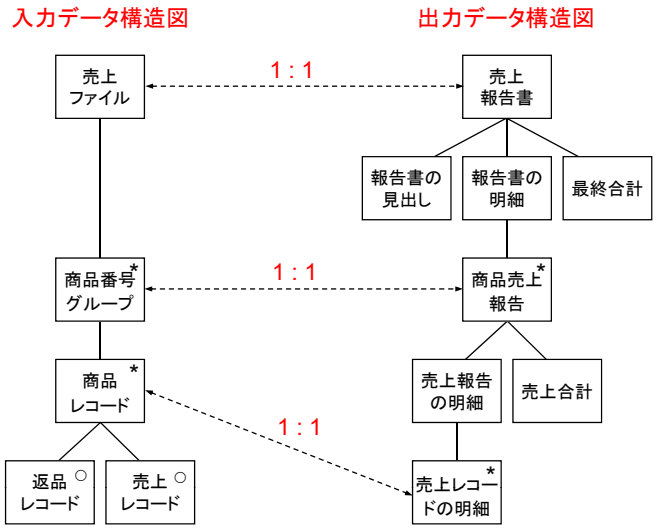
入力データ構造図の例



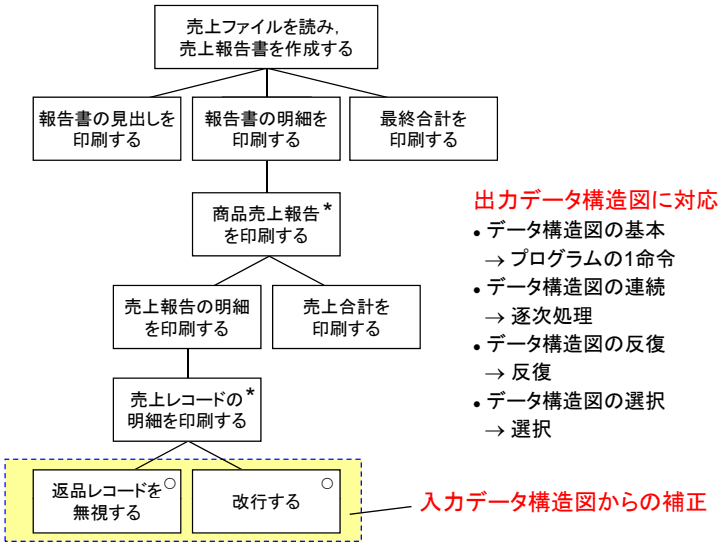
出力データ構造図の例



構成要素の間の対応



プログラム構造



ワーニエ法

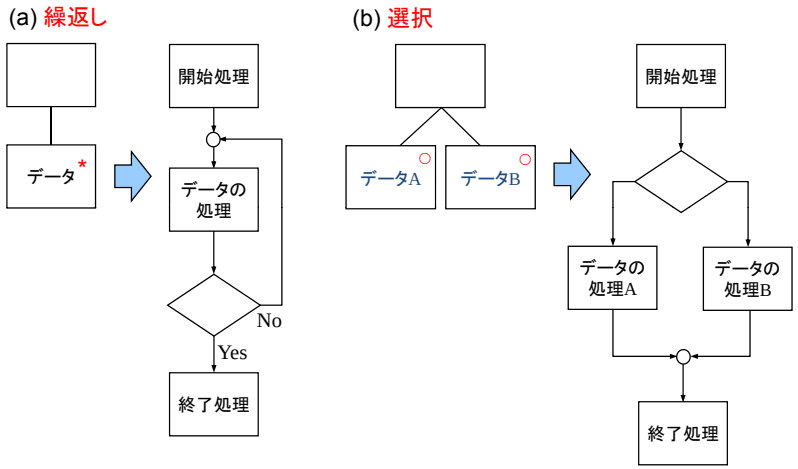
- 入力データ構造と出力データ構造から直接プログラムの論理構造を決定

手順

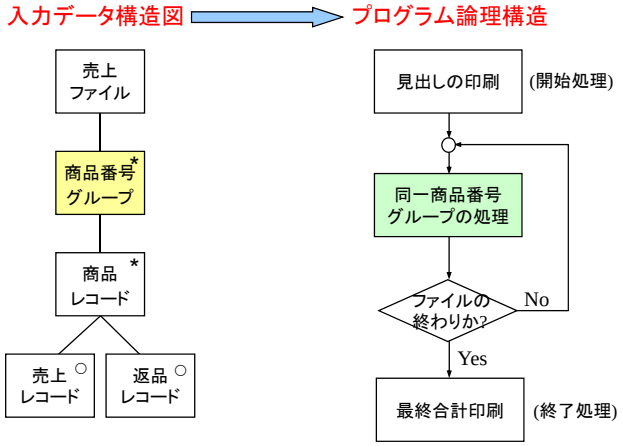
(1) データ構造の定義
- ✓ 入力データ構造と出力データ構造を分析し、4つの構成要素（基本、連続、反復、選択）でデータ構造図を作成

(2) プログラム論理構造の決定
- ✓ 入力データ構造を基にプログラムの論理構造を決定
  - 反復のデータ構造 → 反復の論理構造
  - 選択のデータ構造 → 選択の論理構造
  - 入力データ構造にあり、出力データ構造になし → 無視
  - 入力データ構造になし、出力データ構造にあり → 入力データの加工
  - 各論理構造の前後に開始部と終了部を付加

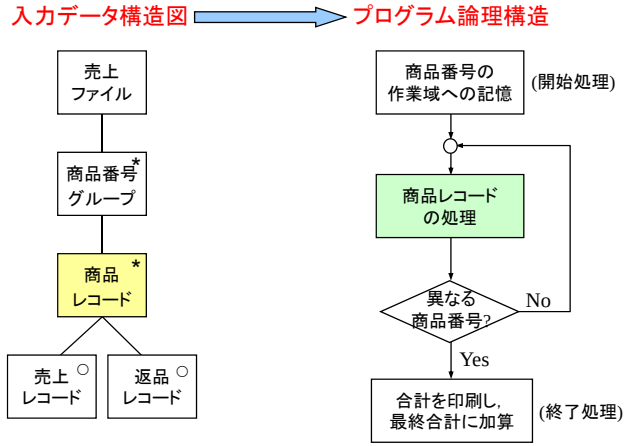
データ構造とプログラム論理構造



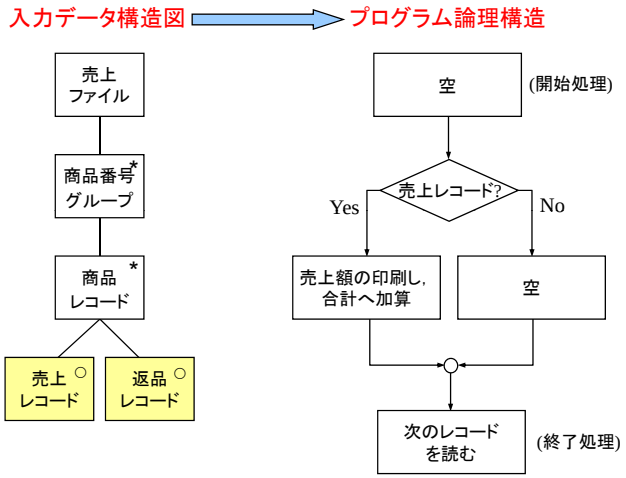
プログラム論理構造の例(1)



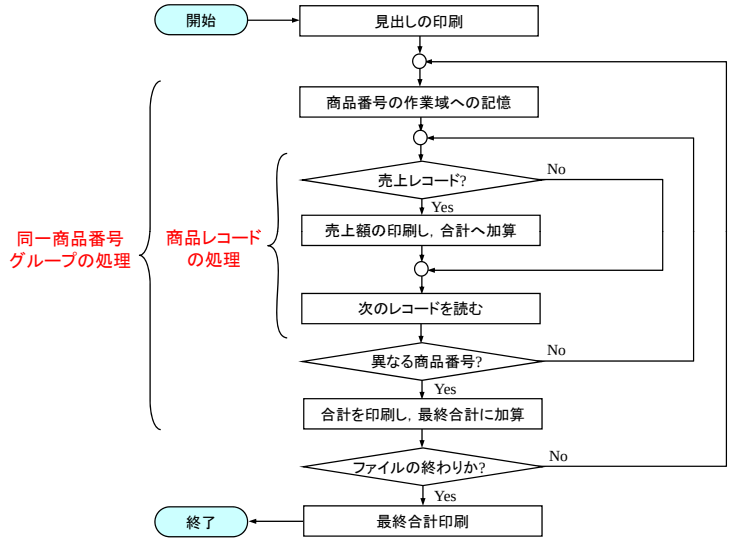
プログラム論理構造の例(2)



プログラム論理構造の例(3)



プログラム論理構造の例（全体）



プログラミング

プログラミング

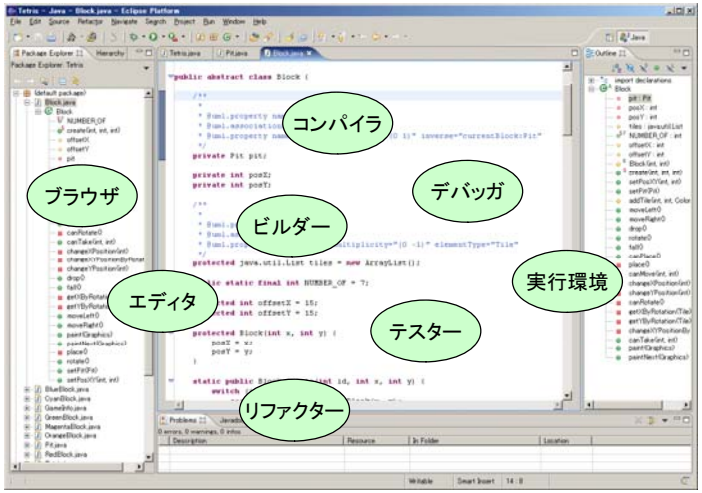
- 仕様（要求仕様や設計仕様）に基づきプログラムを作成する作業
  - ✓ プログラム
    - プログラマが記述したソースコード
    - システム上で動作するマシンコード
- 論理設計
  - ✓ 個々のモジュールの内部構造（データ構造やアルゴリズム）を決定
  - ✓ 理解しやすいプログラムの作成が重要
- コーディング
  - ✓ 具体的なプログラミング言語による記述
    - C, C++, C#, Java, Ruby, COBOLなど
  - ✓ ソフトウェア開発環境の利用
- 単体テスト
  - ✓ モジュール内に存在する論理エラーの除去

ソフトウェア開発環境

- バッチ型プログラミングツール（1950～60年代）
  - ✓ 高級言語コンパイラ
- 対話型プログラミングツール（1970年代）
  - ✓ ドキュメント作成支援、エディタ、デバッガ
- 統合プログラミング環境（1980年代後半）
  - ✓ 構造化技法支援、ビジュアル化
  - ✓ CASE(computer-aided software engineering)
- 自動化（1990年代）
  - ✓ 統合開発環境(IDE: integrated development environment)
  - ✓ リポジトリ(repository)の活用
    - ソフトウェア開発時の情報を集中管理するための保管庫
- オープン化（1990年代後半～）
  - ✓ 分散開発環境支援
    - CVS, Subversion, Mercurial, gitなど
  - ✓ プラグインの作成と公開

統合開発環境

- Eclipse, IntelliJ IDEA, MS Visual Studio, Adobe Flex, ...



プログラミングパラダイム

- プログラミング
  - ✓ 計算機を使って解くべき問題をプログラムとして記述すること
- 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
  - ✓ 設計手順やプログラム構造およびプログラムの記述方法を規定するもの
  - ✓ 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

関数fact(n)の定義

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fact(n-1) & \text{otherwise} \end{cases}$$

⇒

Lispプログラム

```
(defun fact(n)
  (if (zerop n) 1
    (* n (fact n (- n 1)))))
```

階乗n!の定義

fact(0,1)

fact(n,x) ← sub(n,1,a) ∧ fact(a,b) ∧ times(n,b,x)

⇒

Prologプログラム

```
fact(0,1).
fact(N,X) :- A is N-1, fact(A,B), X is N*B.
```


プログラミングパラダイムの例

- 手続き型プログラミング(procedural programming)
  - ✓ コンピュータの処理手順を文で記述
  - ✓ Fortran, COBOL, Algol, BASIC, PL/I, Pascal, C, Adaなど
- 関数型プログラミング(functional programming)
  - ✓ 入出力関係を表現する関数とその呼出しで記述
  - ✓ Lisp（λ算法：lambda calculus）, Scheme, ML, Haskell, OCamlなど
- 論理型プログラミング(logic programming)
  - ✓ 入出力関係を述語論理（事実と規則）で記述
  - ✓ Prolog（導出原理：resolution principle）
- オブジェクト指向プログラミング(object-oriented programming)
  - ✓ 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#など
- アスペクト指向プログラミング(aspect-oriented programming)
  - ✓ オブジェクトにまたがる横断的な関心事(cross-cutting concern)をアスペクトにまとめて記述し、あとで織り合わせ(weaving)
  - ✓ AspectJ, Hyper/J, DemeterJ(adaptive programming)など

構造化プログラミング

- IBMの記述規範
  - ✓ メモリの使用領域と処理時間の最小化
    - テスト作業や保守作業が面倒

    →理解しやすいプログラムの作成
    - プログラムの構造や振る舞いを容易に把握可能

1. 分割統治(divide and conquer)
   - ✓ 大きく複雑なプログラムを小さく簡単なモジュールで合成
     - 関心の分離(SoC: separation of concern)
2. 段階的詳細化(stepwise refinement)
   - ✓ 要求プログラムを抽象データ型を仮定して作成し、上位の抽象データ型を下位の抽象データ型で繰り返し具体化
3. 構造化定理(structured program theorem)
   - ✓ 手続き型プログラムを3つの基本制御の論理構造で構築

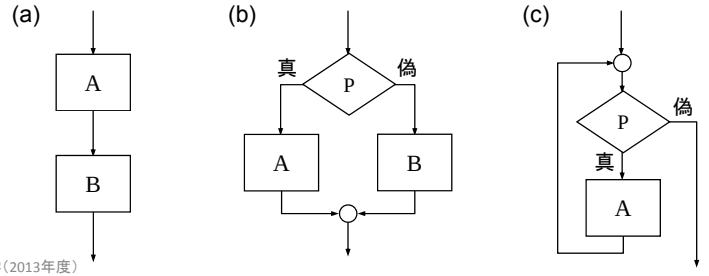
構造化定理

- すべての適正プログラム(proper program)の論理は3つの基本制御構造(連接、選択、反復)で記述可能
  - (a) 連接(sequence)

    命令(命令群) A, B を順番に実行
  - (b) 選択(selection, if-then-else)

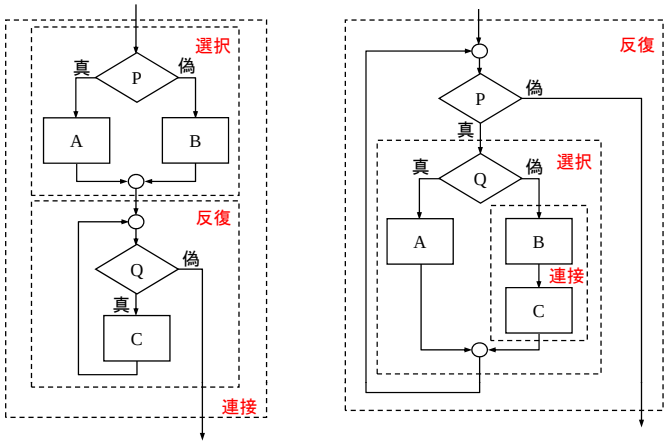
    条件 P の真偽により命令 A, B の実行を選択
  - (c) 反復(iteration, do-while)

    条件 P が真である間、命令 A を繰返し実行



適正プログラム

- プログラムの制御の流れに対し、1つの入り口と1つの出口を持つ
- プログラムの制御の流れにすべての命令が関係する



構造化プログラミングにおけるコーディング

- ソースコードをテキスト形式で入力
  - ✓ 3つの基本制御構造をそのまま命令に変換
- goto文の使用を制限
  - ✓ 例外処理(exception)
  - ✓ 関数や手続きからの脱出(return)
  - ✓ 反復からの脱出(break)
  - ✓ 重複コードの排除
- コーディング規約(coding convention/standards)
  - ✓ コーディングにおける約束
    - ソースコードの共有を促進
  - ✓ 命名規則
    - データ名、変数名、関数名の付け方
  - ✓ スタイル
    - 基本構造に基づく字下げ(indentation)や括弧の位置
    - イディオム(idiom)の活用

ソフトウェアテスト

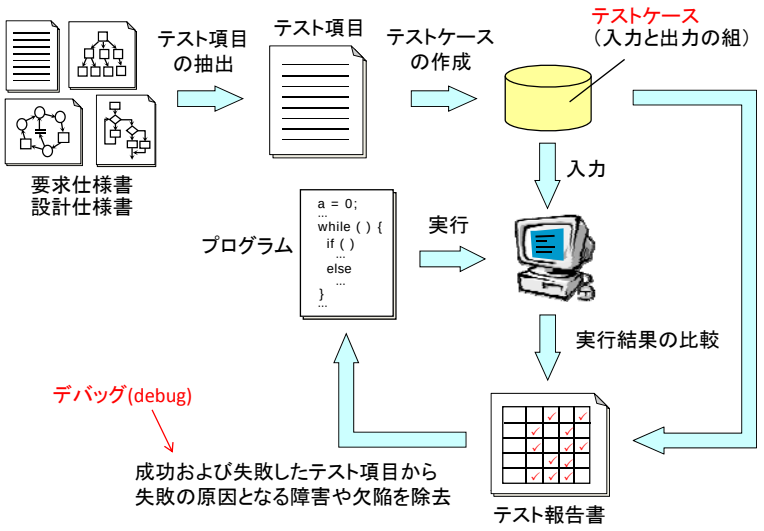
ソフトウェアテスト

- プログラムが仕様書に定義された振る舞いを満たすかどうかの検査
- 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
  - ✓ 誤り(error)
    - 実際の実行結果と予想される出力結果との相違
    - 開発において人間が起こした過ち(mistake)
  - ✓ 障害(fault)
    - ソフトウェア内部に存在する誤り
    - 欠陥(defect)、バグ(bug)
    - 誤りにより発生、故障の原因
  - ✓ 故障(failure)
    - ソフトウェアの外部から見た要求に反する振る舞い
    - すべての故障が障害を引き起こすわけではない

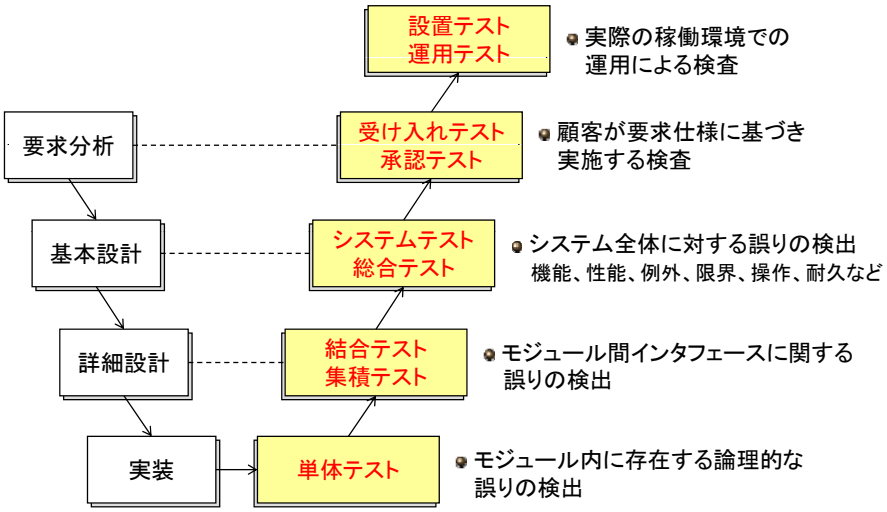
ソフトウェアテストの性質

- テストは障害や欠陥の存在を示すことはできるが、障害や欠陥が存在しないことは示せない
  - ✓ 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する作業
- テストはソフトウェアの品質を直接向上させない
  - ✓ テストは品質が維持できていないことを確認する作業
  - ✓ 設計工程や実装工程で品質を作りこむことが重要
- テストファースト(test-first)、テスト駆動(test driven)
  - ✓ テスト工程の一部を設計や実装の前に実施
    - 要求される品質に対してどのようなテストを行うのかを設計や実装の前に決定
    - 設計作業や実装作業と同時にテストを実行
  - ✓ テストの失敗と成功を繰り返すことで、要求される品質を開発者が容易に作りこみ可能

ソフトウェアテストの作業



テストエ程（Vモデル）



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
  - プログラムの詳細（内部構造や内部論理）を無視

  同値分割、境界値分析

  (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

  テスト網羅技法

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足する範囲（有効同値クラス）と満足しない範囲（無効同値クラス）に分割
  - ✓ 同値
    - 同じ範囲に属する入力データに対して同じ実行結果が得られる（プログラムが同じ動作をする）こと

（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以下あるいは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
  （例）amku5ge, X123
  (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）xy9（3文字）, abcdef（英字のみ）, 123456gh（先頭が数字）

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
  - ✓ 境界値や境界付近の値を入力データに選ぶ
  - ✓ 経験的に、多くの誤りが境界値付近で発生している

(1) 入出力条件の識別
機能仕様の入出力条件に着目し、境界を判別する

（例）

条件	1から64の数字
境界	1と64

(2) 境界に基づくテストケースの作成
（例）0, 1, 2, 63, 64, 65

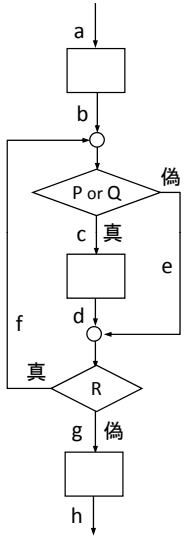
テスト網羅技法(1)

- 命令網羅、節点網羅(statement coverage, C0 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(coverage) ＝ 実行した文 / 全文

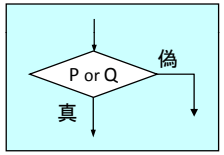
- 枝網羅、分岐網羅(edge coverage, C1 coverage)
  - ✓ プログラム中のすべての枝を1回以上実行
    （例）P or Q が真、R が真（パス：abcdf）
    P or Q が真、R が偽（パス：abcdgh）
    P or Q が偽、R が真（パス：abef）
    P or Q が偽、R が偽（パス：abegh）

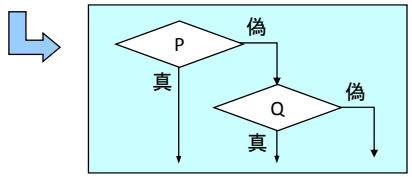
  網羅(coverage) ＝ 通過した枝 / 全枝

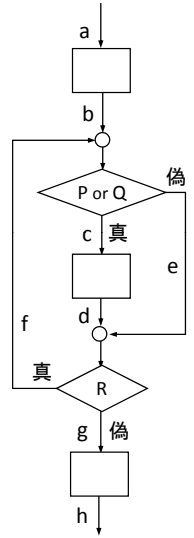


テスト網羅技法(2)

- 条件網羅(condition coverage)
  - ✓ プログラム中のすべての条件判定を1回以上実行
    （例）P と Q を区別
    P が真、Q が真 or 偽
    P が偽、Q が真 or 偽



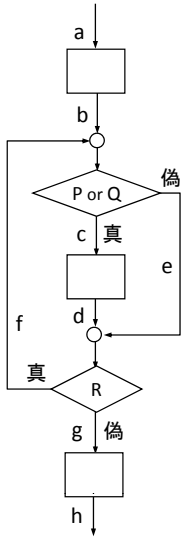




テスト網羅技法(3)

- パス網羅(path coverage)
  - ✓ 判定条件間の依存性（条件の組合せ）を考慮
  - ✓ プログラム中のすべてのパスを1回以上実行
    （例）abcdgh + abcdfcdgh + abegh
    + abefegh + abcdfegh + …

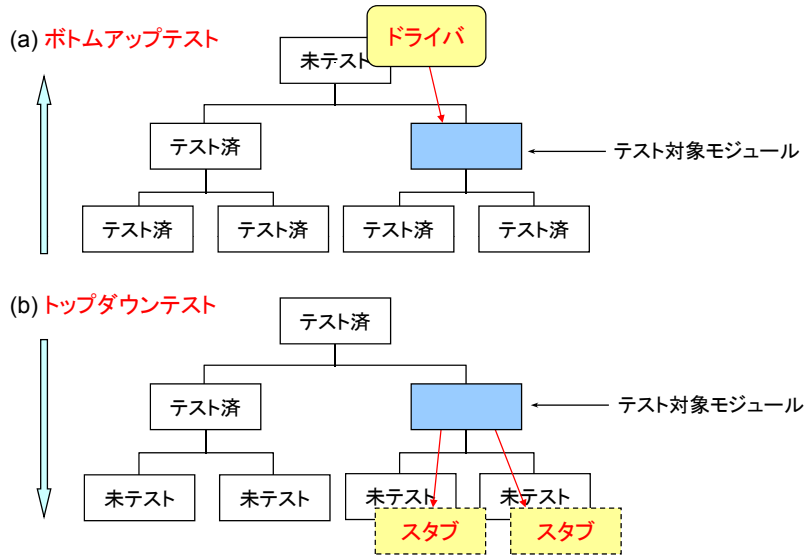
  網羅(coverage) ＝ 実行したパス / 全パス



結合テスト

- 結合テスト(integration test)
  モジュールインタフェース（引数や共有データ）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
    - ドライバ（仮のメインプログラム）が必要
    - 初期段階から並行に数多くのモジュールをテスト可能
  (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
    - スタブ（身代わりモジュール）が必要
    - インタフェースの誤りを早期に発見可能
  (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
  (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

ボトムアップテストとトップダウンテスト



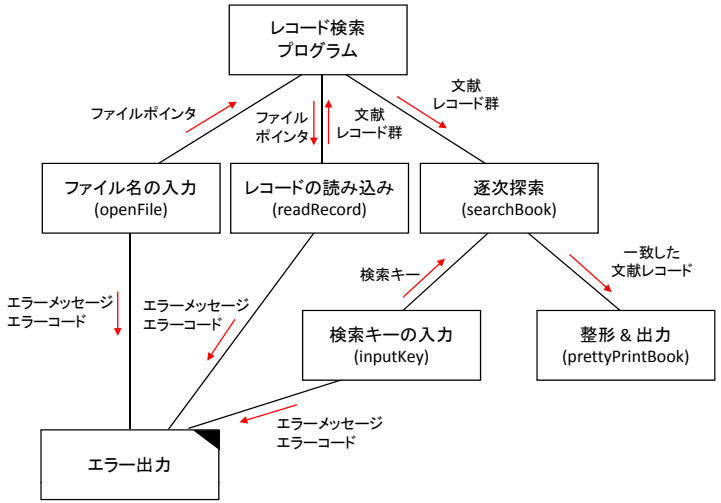
例題：トップダウンプログラミング

トップダウンプログラミング：トップダウンテストに基づくプログラミング

逐次マッチングによる検索プログラム

- 文献簡易目録ファイル名（最大100バイト）をプロンプトメッセージ出力の後、キーボードから受け取り，このファイルのすべてのレコードを配列に読み込んで、総文献件数を出力した後，入力された姓の読み（ローマ字）が前方一致で該当するすべてのレコードを検索し、姓の読み、著者、書名、出版社、出版年、ISBN番号をそれぞれ1行ずつ画面に表示するプログラムを作成せよ。検索は、会話型で行われ、終了コード（/）が入力されるまで繰り返すものとする。
- 文献簡易目録ファイルは、姓の読み、著者、書名、出版社、出版年、ISBN番号の6つの項目が空白文字で区切られてならんでおり、各項目の最大長はどれも200バイトを超えない。文献レコードの区切りは改行になっており、最大レコード数は2000件を超えないものとする。

例題：モジュール構造図



例題：ステップ(1)

データ構造の決定
メイン手続きの作成とテスト

```
#define RECORD_NUM  2000
#define FIELD_SIZE  201

typedef struct _BookRecord {
    char yomi[FIELD_SIZE];      /* 姓の読み */
    char author[FIELD_SIZE];    /* 著者 */
    …
} BookRecord;

main( )
{
    BookRecord bookTable[RECORD_NUM];
      /* 文献レコード群 */
    FILE *fp;   /* ファイルポインタ */
    int num;    /* 総文献件数 */

    fp = openFile( );
    num = readRecord(fp, bookTable);
    fclose(fp);
    searchBook(bookTable, num);
}
```

作成 & テスト対象

スタブ(stub)

```
FILE *openFile( )
{
    printf("####  Open File ####¥n");
    return(NULL);
}
```

```
int readRecord(FILE *fp, BookRecord bookTable[ ])
{
    printf("####  Read Records ####¥n");
    return(0);
}
```

```
void searchBook(BookRecord book[ ], int num)
{
    printf("####  Retrieve Books  ####¥n");
}
```

例題：ステップ(2-a)　openFile手続きの作成とテスト

```
#define FILENAME_SIZE  101

FILE *openFile( )
{
   char filename[FILENAME_SIZE];  /* ファイル名 */
   FILE *fp;  /* ファイルポインタ */

   /* プロンプトの表示  */
   printf("文献簡易目録ファイル> ");

   /* ファイル名の入力 */
   /* 本来はファイル名の長さを検査する */
   scanf("%s", filename);

   /* ファイルポインタの取得(ファイルオープン) */
   if ((fp = fopen(filename, "r")) = NULL)

      /* ファイルオープンに失敗したとき，エラー出力 */
      printError("Cannot open file %s¥n", filename, 1);

   return(fp);
}
```

```
void printError(char *msg, char *str, int no)
{
   printf("#### Error ####¥n");
}
```

スタブ

例題：ステップ(2-b)　readRecord手続きの作成とテスト

```
int readRecord(FILE *fp, BookRecord bookTable[ ])
{
   int num;  /* 総文献件数 */

   /* 総文献件数の初期化 */
   num = 0;

   /* ファイルの終わりまで */
   while (!feof(fp)) {

      /* 文献レコードを読み込む */
      if (fscanf(fp, "%s %s %s %s %d %s¥n",
        bookTable[num].yomi, bookTable[num].author, … ) == 6)
         /* 文献件数をかぞえる */
         num++;
      else
         /* 各フィールドが正常に読み込めなかったとき，エラー出力 */
         printError("Format error %s¥n", book.Table[num].yomi, 1);
   }

   /* 総文献件数の表示 */
   printf("Total number of books = %d¥n", num);
   return(num);
}
```


例題：ステップ(2-c)　searchBook手続きの作成とテスト

```
void searchBook(BookRecord book[ ], int num)
{
   char key[FIELD_SIZE];
   int i;

   while (1) {
      /* 検索キーの入力 */
      inputKey(key);

      /* 検索キーが"/"のとき，ループから脱出 */
      if (strcmp(key, "/") == 0) break;

     /* 逐次マッチング */
      printf("--------¥n");
      for (i = 0; i < num; i++) {

         /* 一致したレコードを整形して出力 */
         if (strcmp(book[i].yomi, key, strlen(key)) == 0) {
            prettyPrintBook(book[i]);
            printf("--------¥n");
         }
      }
   }
}
```

```
void inputKey(char key[ ])
{
   printf("####  Input Key ####¥n");
   strcpy(key, "/");
}
```

```
void prettyPrintBook(BookRecord book)
{
   printf("####  Pretty Print Books ####¥n");
}
```

スタブ

例題：ステップ(3)　残り手続きの作成とテスト

```
void inputKey(char key[ ] )
{
   /* 検索キーの入力 */
   printf(" 検索文字> ");
   scanf("%s", key);

   while (strlen(key) >= FIELD_SIZE) {

      /* 入力検索キーが長すぎる */
      printError("Invalid input key %s ¥n", key, 0);

      /* 検索キーの再入力 */
      printf(" 検索文字> ");
      scanf("%s", key);
   }
}
```

```
void prettyPrintBook(BookRecord book)
{
   printf("姓の読み：%s¥n", book.yomi);
   printf("著者：%s¥n", book.author);
   …
}
```

```
void printError(char *msg, char *str, int no)
{
   /* エラーメッセージの出力 */
   fprintf(stderr, msg, str);

   /* エラーコードが0でないとき，終了 */
   if (no != 0) exit(no);
}
```

システムテスト

- システムテスト(system test)
  顧客の要求をシステムが満たしているかどうかを検査

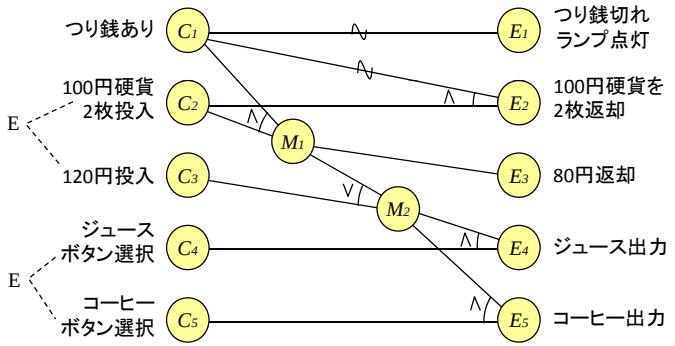
(a) 機能テスト(function test)
- システムの機能が要求仕様通りに稼働するかどうかを検査
  原因結果グラフ法

(b) 性能テスト(performance test)
- 非機能要求を評価
  過負荷テスト(stress test)、容量テスト(volume test)、
  構成テスト(configuration test)、互換性テスト(compatibility test)、
  セキュリティテスト(security test)、タイミングテスト(timing test)、
  環境テスト(environment test)、品質テスト(quality test)、
  回復テスト(recovery test)、保守テスト(maintenance test)、
  文書化テスト(documentation test)、ユーザビリティテスト(usability test)

原因結果グラフ法

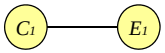
- 原因(入力)と結果(出力)の因果関係に着目し、テストケースを作成

1. 原因結果グラフ(CEG: cause-effect graph)の作成

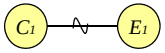


原因結果グラフ

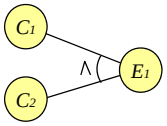
(a) 肯定(C_1であればE_1)



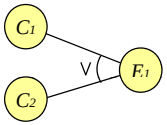
(b) 否定(C_1でなければE_1)



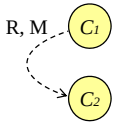
(c) 論理積(C_1かつC_2であればE_1)



(d) 論理和(C_1またはC_2であればE_1)

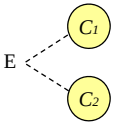


(e) 必要(R), マスク(M)



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): 同時には成立しない
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

2. 決定表(decision table)の作成

		1	2	3	4	5	6	7	8	…	16	17	18
原因	C1(つり銭あり)	×	×	×	×	×	×	×	×	…	○	○	○
	C2(100円硬貨2枚投入)	×	×	×	×	×	×	○	○	…	○	○	○
	C3(120円投入)	×	×	×	○	○	○	×	×	…	×	×	×
	C4(ジュースボタン選択)	×	×	○	×	×	○	×	×	…	×	×	○
	C5(コーヒーボタン選択)	×	○	×	×	○	×	×	○	…	×	○	×
結果	E1(つり銭切れランプ点灯)	○	○	○	○	○	○	○	○	…	×	×	×
	E2(100円硬貨2枚返却)	×	×	×	×	×	×	○	○	…	×	×	×
	E3(80円返却)	×	×	×	×	×	×	×	×	…	○	○	○
	E4(ジュース出力)	×	×	×	×	×	○	×	×	…	×	×	○
	E5(コーヒー出力)	×	×	×	×	○	×	×	×	…	×	○	×

(例)「8」: つり銭なしの状態で、200円を投入して、コーヒーのボタンを押した場合、
つり銭切れランプが点灯しており、200円が返却される。

形式手法とソフトウェア検証

ソフトウェア検証

- ソフトウェアが要求される品質を満たすことやソフトウェアに誤りが存在しないことを確認する作業
  - ✓ 正当性検証(verfication)
    - 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
  - ✓ 妥当性確認(validation)
    - ソフトウェアが利用者の要求を満たしているか
- 形式手法
  - ✓ 論理(logic)、代数(algebra)、集合論(set theory)などの数学に基づく形式化(formalization)をソフトウェア開発に取り入れること
  - ✓ 仕様の厳密性やプログラムの正しさの検証などに貢献
    - 機能の形式化
    - データの形式化

機能の形式化

- 入出力条件による形式化（論理的仕様で表現されることが多い）
  - ✓ システムの機能を入出力時に成立する条件で表現
- 関数による形式化（関数型仕様）
  - ✓ 入力を出力に変換する関数として表現

（例）整数 x と y の最大公約数 z を求める機能

入出力条件による定義
入力条件：$integer(x) \wedge integer(y) \wedge x > 0 \wedge y > 0$
出力条件：$interger(z) \wedge divide(z, x) \wedge divide(z, y) \wedge \forall w.(integer(w) \wedge (divide(w, x) \wedge divide(w, y) \Rightarrow z \geq w))$
ただし、$divide(a, b)$はbがaによって割り切れることを意味

関数$gcd(x, y)$の定義
$gcd(x, y) = gcd(x, y \textbf{ mod } x) = gcd(x \textbf{ mod } y, y)$
$gcd(x, y) = gcd(y, x)$
$gcd(x, 0) = gcd(0, x) = x$
ただし、$a \textbf{ mod } b$はaをbで割った余りを指す
関数変換と見ると、左辺を右辺に書き換えることを意味

データの形式化

- 抽象データ型(ADT: abstract data type)
  - ✓ データ構造を、それに対する演算(operation)の組により定義
  - ✓ 演算の仕様（インタフェース）と内部実装を分離し、公開演算子を通してのみデータにアクセス可能（データのカプセル化：encapsulation）
  - ✓ 内部状態を隠蔽（情報隠蔽：informatin hiding）
- 代数的仕様(algebraic specification)
  - ✓ データ型を代数とみなし、代数を公理で記述することで、演算の意味を定義

（例）スタック(stack)の代数的仕様記述

型種(sort):	公理(axioms):
Stack(integer)	*s*: Stack
	z: integer
演算子(operators):	pop(push(*z*, *s*)) = *s*
init: → Stack	pop(init) = **stack-error**
push: integer × Stack → Stack	top(push(*z*, *s*)) = *z*
pop: Stack → Stack	top(init) = **stack-error**
top: Stack → integer	empty(init) = **true**
empty: Stack → bool	empty(push(*z*, *s*)) = **false**

仕様検証

- レビュー(review)
  - ✓ 要求仕様、設計仕様、プログラムコードを精査
  - ✓ インスペクション(inspection)
    - 公式なレビュー
    - 事前に検査項目を決定しておき、正常な動作と照合
  - ✓ ウォークスルー(walk-through)
    - 非公式なレビュー
    - ソフトウェアを机上で実行し、誤りを指摘

- モデル検査(model checking)
  - ✓ ソフトウェアが時相論理式(temporal logic formula)を満たすモデルとなっているか自動的に検査
    - 安全性(safety): 望ましくない事象が決して起こらないこと
    - 活性(liveness): 望む事象がいつかは必ず起こること

プログラム検証

- 型検査(type checking)
  - ✓ 型(type)に関する不適切な演算や操作を検査することで誤りを検出

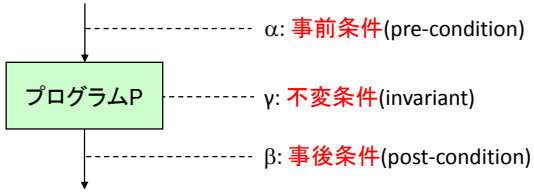
- 帰納表明法(inductive assertion method)[Floyd]
  - ✓ プログラムの部分正当性を証明する技法
  - ✓ 正当性(correctness)
    - プログラムが必ず停止する(停止性)
    - 得られた答えは必ず正しい(部分正当性)
  - ✓ Hoare論理に発展
  - ✓ 定理証明器(theorem prover)を利用

型検査

- 型
  - ✓ ものの集まり、ものを分類するための仕組み
  - ✓ 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の整数である
    - 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

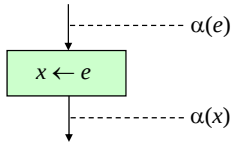
帰納表明法



- P の実行前に α が成立するならば、P の実行後に β が成立する
  (例) 10個の要素を持つ配列が入力されると、ソートされた配列
  (配列添字の大きい方が、その値が大きい)が出力される
- P の実行中は必ず γ が成立する
  (例)配列の要素の数は変わらない

検証条件(1)

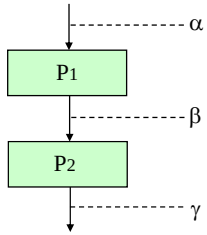
(1) 代入



$\{\alpha(e)\}\ x \leftarrow e\ \{\alpha(x)\}$

$x \leftarrow e$の実行前に$\alpha(e)$が成立するならば，
$x \leftarrow e$の実行後に$\alpha(x)$が成立する

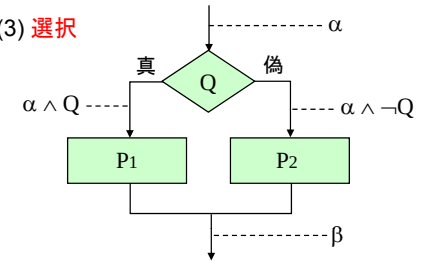
(2) 連接



$$\frac{\{\alpha\}\ P_1\ \{\beta\},\ \{\beta\}\ P_2\ \{\gamma\}}{\{\alpha\}\ P_1; P_2\ \{\gamma\}}$$

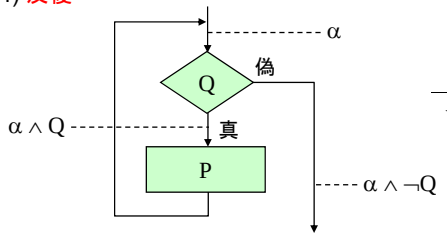
検証条件(2)

(3) 選択



$$\frac{\{\alpha \wedge Q\}\ P_1\ \{\beta\},\ \{\alpha \wedge \neg Q\}\ P_2\ \{\beta\}}{\{\alpha\}\ \textbf{if}\ Q\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ \{\beta\}}$$

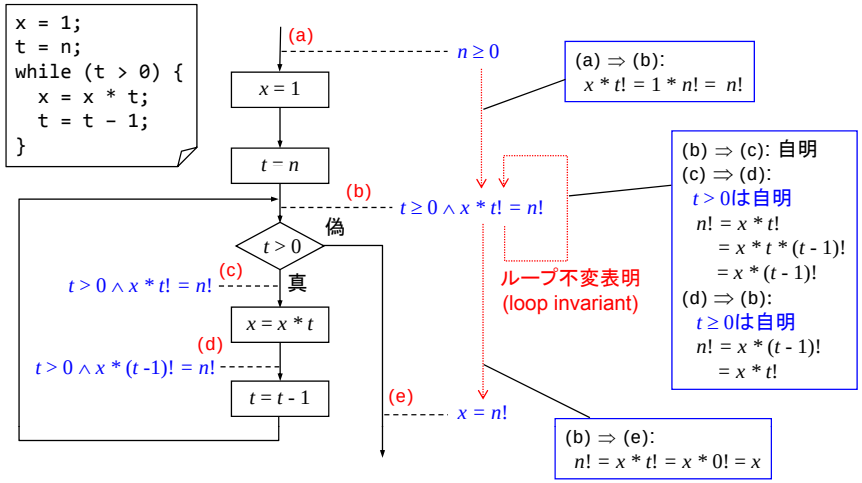
(4) 反復



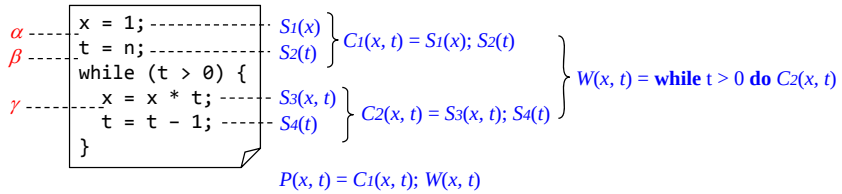
$$\frac{\{\alpha \wedge Q\}\ P\ \{\ \alpha\ \}}{\{\alpha\}\ \textbf{while}\ Q\ \textbf{do}\ P\ \{\alpha \wedge \neg Q\}}$$

帰納表明法の例

表明(assertion): プログラムの各時点で成立する変数間の関係式



Hoare論理



P は 0 以上の n に対して n! を計算する

$\{n \geq 0\}\ P(x, t)\ \{x = n!\}$

n が 0 以上ならば P の実行後に x は n! に等しい

検証条件を使って検証すると…

Hoare論理

$\{n \geq 0\}\ P(x, t)\ \{x = n!\}$

$\Leftrightarrow \{n \geq 0 \wedge 1*n! = n!\}\ C_1(x, t); W(x, t)\ \{x * t! = n! \wedge t = 0\}$

$\Leftrightarrow \{n \geq 0 \wedge 1*n! = n!\}\ C_1(x, t)\ \{\beta\},\ \{\beta\}\ W(x, t)\ \{x * t! = n! \wedge t \geq 0 \wedge \neg(t > 0)\}$

$\Leftrightarrow \{n \geq 0 \wedge 1*n! = n!\}\ S_1(x); S_2(t)\ \{\beta\},\ \{\beta\}$ **while** t > 0 **do** $C_2(x, t)\ \{x * t! = n! \wedge t \geq 0 \wedge \neg(t > 0)\}$

$\Leftrightarrow \{n \geq 0 \wedge 1*n! = n!\}\ S_1(x)\ \{\alpha\},\ \{\alpha\}\ S_2(t)\ \{\beta\},\ \{\beta\}$ **while** t > 0 **do** $C_2(x, t)\ \{x * t! = n! \wedge t \geq 0 \wedge \neg(t > 0)\}$

$\Leftrightarrow \{n \geq 0 \wedge 1*n! = n!\}$ x = 1 $\{\alpha\},\ \{\alpha\}$ t = n $\{\beta\},\ \{\beta \wedge$ t > 0$\}\ C_2(x, t)\ \{x * t! = n! \wedge t \geq 0\}$

$\Leftrightarrow \{n \geq 0 \wedge 1*n! = n!\}$ x = 1 $\{n \geq 0 \wedge x*n! = n!\},\ \{n \geq 0 \wedge x*n! = n!\}$ t = n $\{x * t! = n! \wedge t \geq 0\},$
$\{x * t! = n! \wedge$ t > 0$\}\ S_3(x, t); S_4(t)\ \{x * t! = n! \wedge t \geq 0\}$

$\Leftrightarrow \{n \geq 0 \wedge 1*n! = n!\}$ x = 1 $\{n \geq 0 \wedge x*n! = n!\},\ \{n \geq 0 \wedge x*n! = n!\}$ t = n $\{x * t! = n! \wedge t \geq 0\},$
$\{x * t! = n! \wedge t > 0\}\ S_3(x, t)\ \{\gamma\},\ \{\gamma\}\ S_4(t)\ \{x * t! = n! \wedge t \geq 0\}$

$\Leftrightarrow \{n \geq 0 \wedge 1*n! = n!\}$ x = 1 $\{n \geq 0 \wedge x*n! = n!\},\ \{n \geq 0 \wedge x*n! = n!\}$ t = n $\{x * t! = n! \wedge t \geq 0\},$
$\{x * t! = n! \wedge t > 0\}$ x = x * t $\{\gamma\},\ \{\gamma\}$ t = t – 1 $\{x * t! = n! \wedge t \geq 0\}$

$\Leftrightarrow \{n \geq 0 \wedge 1*n! = n!\}$ x = 1 $\{n \geq 0 \wedge x*n! = n!\},\ \{n \geq 0 \wedge x*n! = n!\}$ t = n $\{x * t! = n! \wedge t \geq 0\},$
$\{x * t! = n! \wedge t > 0\}$ x = x * t $\{x * (t–1)! = n! \wedge t > 0\},\ \{x * (t–1)! = n! \wedge t > 0\}$ t = t – 1 $\{x * t! = n! \wedge t \geq 0\}$

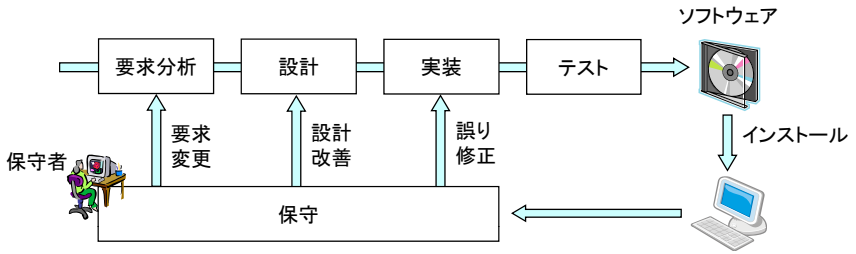
ソフトウェア保守と再利用

ソフトウェア保守

- 計算機にインストールされたソフトウェアを維持・管理する作業
  - ✓ ソフトウェアを常に正しく稼働させることが目的
- 開発の繰り返しを促進する作業
  - ✓ ソフトウェア進化(software evolution)
    - ソフトウェアを永続的に発展
    - 機能拡張、機能変更、性能改善、環境への適合
  - ✓ ソフトウェア変更の積極的な取り込み
    - ソフトウェアは本質的に変化（修正、変更）し続ける
    - 初めから完全なソフトウェアを構築するのは困難
- 保守における戦略
  - ✓ 保守して利用し続けるか vs. 破棄して作り直すか
    - ソフトウェア変更と新規開発のコストの比較
    - 特にレガシーソフトウェア(legacy software)の場合
  - ✓ プログラム進化の法則(Lehman)を考慮
    - 継続的な変更が必須、変更による複雑度の増大など

ソフトウェア保守作業

- 保守対象
  - ✓ ソースコードの変更だけではない
  - ✓ ソフトウェア開発の工程で作成されたすべての成果物
    - 要求仕様書、設計仕様書、ソースコード、テストケース、マニュアルなど
- 保守者(CE: customer engineer)が実施
  - ✓ ソフトウェア開発のすべての工程に関する知識が必要



ソフトウェア保守の分類

- 修正保守(corrective maintenance)
  - ✓ 運用段階で検出された残存エラー(誤り)の修正
  - ✓ 故障に対するソフトウェアの修正
- 適応保守(adaptive maintenance)
  - ✓ 動作環境や利用環境の変化に追従するための変更
    - 業務の拡大、ハードウェアの進歩、OSの更新、法律の改正など
- 改善保守/完全化保守(perfective maintenance)
  - ✓ 機能拡張(変更)や性能向上のための変更
    - 暗号化の導入、検索速度の向上、メニューの改変など
  - ✓ 管理のしやすさを向上させるための変更
    - 設計の改善、ソースコードの書換え、オブジェクト指向への切替え、コンポーネント化など
- 予防保守(preventive maintenance)
  - ✓ 故障を未然に防ぐための修正
  - ✓ 潜在的な誤りの除去

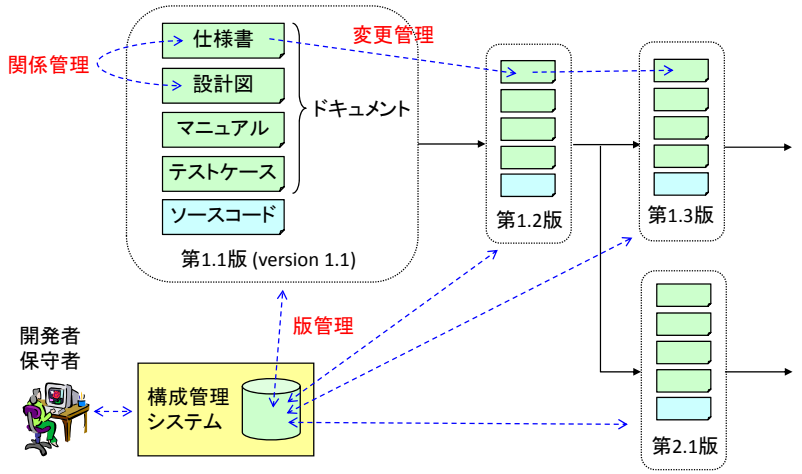
ソフトウェア保守技法

- 保守対象のソフトウェアについて知る(理解する)ことが重要
  - ✓ ソフトウェア理解とは
    - どのような要求に対して開発されたのか
    - どのような設計指針のもとで開発されたのか
    - どのようなアーキテクチャが採用されているのか
    - どのようなモジュール構成になっているのか
    - どのようなプログラムで実現されているのか
    - どのようなテストが行われたのか、など
- 保守支援の技法
  - ✓ 構成管理(configuration management)
  - ✓ 影響分析(impact analysis)
  - ✓ 回帰テスト(regression test)
  - ✓ ソフトウェア視覚化(software visualization)
  - ✓ ソフトウェアリエンジニアリング(re-engineering)
  - ✓ プログラムスライシング(program slicing)

構成管理

- ソフトウェア開発に含まれる構成要素やプロセスを一貫して管理
  - ✓ 成果物に対する付加情報(メタデータ: meta data)を管理
    - この成果物はどのプロジェクトにおいて作成されたのか
    - 開発のどの段階(要求分析、設計、実装、...)で作成されたのか
    - 誰がいつ作成したのか
    - 誰がいつどのようなテストを行ったのか
  - ✓ 成果物間の対応関係を管理
    - 仕様書の記述とそれを実現するモジュールとの関係など
- 過去に適用された変更を管理する(版管理)
  - ✓ バージョン(版: version)とリリース(release)を管理
    - バージョン: 利用者の要求を満たす特定の構成(スナップショット: snapshot)
    - リリース: 以前のソフトウェアを修正あるいは改善したもの

構成管理システム



影響分析

- 保守における変更による影響が及ぶ範囲（影響を受けるモジュール）を把握する作業
  - ✓ 保守による変更は直接変更を適用した部分にとどまらず、他の多くの部分に影響を及ぼす
- 影響範囲を変更前に解析し、必要な資源や工数を見積もる作業
- 変更後にソフトウェアの一貫性が維持されているかを検査する際にも適用
  - ✓ 波及効果解析(ripple effect analysis)
- 保守において新たな誤りを混入させる危険性や、内部に発生した矛盾によりソフトウェアが動作しなくなる可能性を減少
- 多くの影響分析手法はソースコードが対象
  - ✓ クロスリファレンス情報に基づき影響範囲を特定
    - 変数の宣言と利用、関数の呼び出し元と呼び出し先、などの関係
  - ✓ ワークプロダクト間の依存関係に基づく影響範囲を特定

回帰テスト

- 以前のソフトウェアで動作していた機能が、新規あるいは修正後のソフトウェアでも正常に動作するか検査する作業
- 未変更部分が元の通り正常に動作するか確認する作業
  - ✓ 変更前に適用されたテストケースをそのまま利用
  - ✓ 既存のソフトウェアをテスト対象ソフトウェアで置き換える場合や、稼働中のソフトウェアに修正を行った場合に実行
- 保守における新たな誤りの混入を防止
  - ✓ ソフトウェアを少しずつ変更するたびに実施
  - ✓ テストの自動化が重要
    - 同じテストの繰り返しを回避

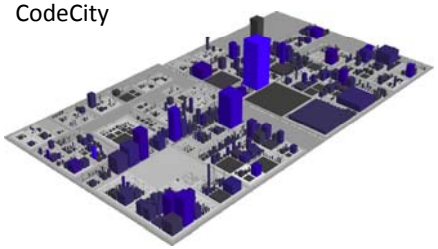
ソフトウェア視覚化

- ソフトウェアに関わる様々な情報を人間が理解しやすい形式で表現すること
- 視覚化の観点
  - ✓ 構造：プログラムを実行せずに抽出可能なソフトウェアの静的な要素やそれらの関係を可視化
    - ソースコードの色付け
    - ソフトウェアの構成要素間の関係の図示
    - 各種ドキュメントの特性（規模や複雑度など）を図示
  - ✓ 振る舞い：実際に、あるいは抽象的にプログラムを実行して取得したデータを可視化
    - 実際の経路やアルゴリズムのアニメーション化
    - 実行時の関数呼出しやメッセージのやり取りを図示
    - テスト結果を図示
  - ✓ 進化：ソフトウェアの開発プロセスを可視化
    - 過去の変更箇所を色分けして表示

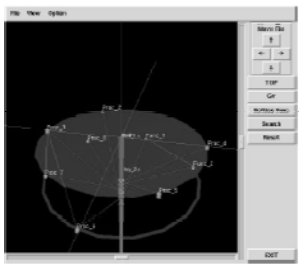
ソフトウェア視覚化の例

SeeSoft

CodeCity

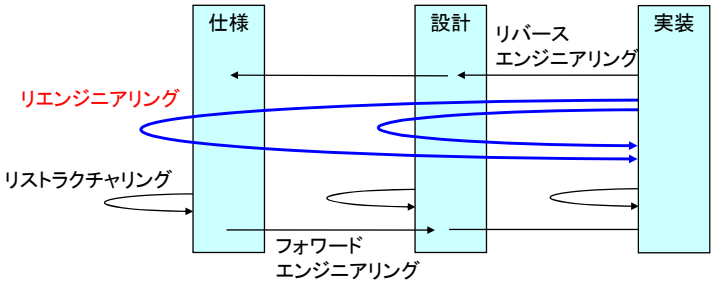


VCRGL



ソフトウェアリエンジニアリング

- 既存のソフトウェアを新たに構成しなおすことで若返らせる作業
  - ✓ 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
- リバースエンジニアリング(reverse engineering)
  - ✓ ソースコードから設計図や要求仕様を回復
- フォワードエンジニアリング(forward engineering)
  - ✓ 通常の開発

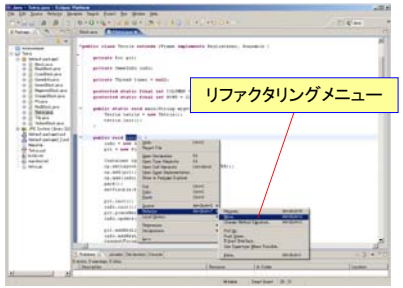


リストラクチャリング

- リストラクチャリング(再構成: restructuring)
  - ✓ 外部から見た振る舞いを変えずに、同じ抽象レベルの別の表現に変換する作業
    - 複雑な処理を単純な処理に分割して記述、実行されない命令を除去する、内部アルゴリズムを取り換える、など
  - ✓ リファクタリング(refactoring)
    - 大きな設計変更を小さな変換の繰り返しで実現することで、外部的挙動を保存したまま内部構造を改善
- リドキュメンテーション(再文書化: redocumentation)
  - ✓ プログラムコード(多くの場合ソースコード)を解析することで、そこに明示的に記述されていない情報を抽出し、人間が理解しやすい文書を作り出す作業
- デザインリカバリ(design recovery)
  - ✓ プログラムコードや設計情報から、より抽象度の高い設計情報を回復
  - ✓ 領域(ドメイン)知識や外部知識、推論を利用

リファクタリング

- 既存ソフトウェアの設計の理解性や変更容易性の向上が目的
  - ✓ 外部から見た挙動は変えない
  - ✓ 内部構造だけ変える
- 不吉な匂い(bad smell)
  - ✓ 潜在的な問題や欠陥に関する兆候
- 大きな(複雑な)設計変更を一連の小さな変換により実現
  - ✓ 関数の名前変更
  - ✓ 変数の名前変更
  - ✓ 関数の移動
  - ✓ 変数の移動、など
- リファクタリングカタログ
  - ✓ 小さな変換を集めて整理したカタログ



プログラムスライシング

- ある変数の値に関係する命令だけをもとのプログラムから抽出
  - ✓ 変数の値はプログラム全体ではなく限定された命令にのみ影響を受ける
- プログラムスライス(program slice)
  - ✓ 抽出した命令の集まり(コード)
  - ✓ 着目する変数から制御依存関係、データ依存関係をたどることによって計算
    - 制御依存関係(control dependence)

```
if (a < 0) {
    b = 10;
}
```

文「b = 10」の実行はif文の判定の結果に依存する

    - データ依存関係(data dependence)

```
a = 10;
b = a + 1;
```

変数aの値の定義が変数aの値の参照に到達

  - ✓ 逆方向スライス(backward slice): 変数の値に影響を与える命令集合
  - ✓ 順方向スライス(forward slice): 変数の値が影響を与える命令集合

プログラムスライスの例

```
 1: int func(int data[]) {
 2:   int sum = 0;
 3:   int prod = 1;
 4:   int i = 0;
 5:   while (i < data.length()) {
 6:     sum = sum + data[i];
 7:     prod = prod * data[i];
 8:     i = i + 1;
 9:   }
10:   print(sum);
11:   print(prod);
12: }
```

もとのプログラム

```
 1: int func(int data[]) {
 2:   int sum = 0;
 3:
 4:   int i = 0;
 5:   while (i < data.length()) {
 6:     sum = sum + data[i];
 7:
 8:     i = i + 1;
 9:   }
10:   print(sum);
11:
12: }
```

文10のsumに関する静的逆方向スライス

プログラム理解

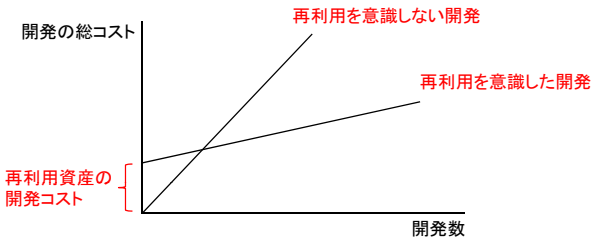
- 実際に稼働している実態はプログラム
  - ✓ プログラムが保守にとって唯一信用できる成果物
  - ✓ 実際の現場では、過去に改善された内容が要求仕様書や設計仕様書に反映されていないという状況が頻繁に発生
- 保守作業ではプログラム理解は不可欠
  - ✓ プログラム理解が必須であることが保守作業を難しくしている最大の原因

ソフトウェア再利用

- 開発されたソフトウェア（成果物）の一部や開発によって得られた知見を次の開発で活用すること
  - ✓ 再利用資産(reusable asset)
    - ソースコード、要求、仕様、設計、テストケース、マニュアル
    - ツール、開発プロセス、開発経験や知見
  - ✓ 再利用部品/コンポーネント(reusable component)
    - 再利用資産を再利用に適した形で汎用化したもの
- 再利用の分類
  - (a) 生産者側での再利用(producer reuse)：再利用資産を提供
    vs. 消費者側での再利用(consumer reuse)：再利用資産を使用
  - (b) ブラックボックス再利用(black-box reuse)：再利用資産を修正せずそのまま使用
    vs. ホワイトボックス再利用(white-box reuse)：再利用資産を一部変更して使用
  - (c) 垂直的再利用(vertical reuse)：同一プロジェクトや同一アプリケーション領域で再利用
    vs. 水平的再利用(horizontal reuse)：プロジェクトやアプリケーション領域を横切る再利用

ソフトウェア再利用の意義

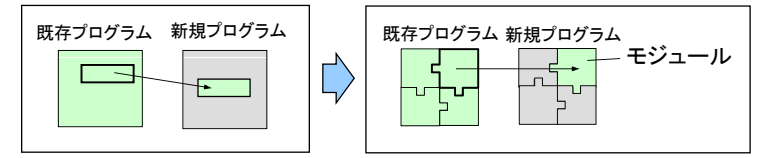
- 成果物が再利用されることを明確に意識することが重要
  - ✓ 場当たり的に行う個人活動ではない
- 過去の資産を有効に活用することが開発を成功させる鍵
  - ✓ ソフトウェアをゼロから開発することは非現実的
    - 適切な再利用部品を容易に検索し、簡単に組み合わせることで生産性を向上
    - 十分な信頼性を持つ再利用部品を組み合わせることで信頼性の高いソフトウェアを構築可能



ソフトウェア再利用技法

- オブジェクト指向におけるクラス
  - ✓ 独立性の高いモジュール
    - 実装の隠蔽（カプセル化）による置換が容易
    - 継承により既存のクラスの一部を再利用
  - ✓ 粒度が小さく再利用の効果が限定的
- オブジェクト指向フレームワーク(object-oriented framework)
  - ✓ クラスとそのインスタンス間の相互作用を内包
    - クラスだけでなく、クラスの利用方法も再利用
  - ✓ 開発者がカスタマイズ（拡張や変更）可能な版完成アプリケーション
    - アプリケーションの一部をフレームワークに組込み
- コンポーネント指向開発(component-based development)
  - ✓ コンポーネントを組み合わせることでソフトウェアを構築
- ソフトウェアパターン(software pattern)
  - ✓ ソフトウェア開発において蓄積された経験や知見の再利用

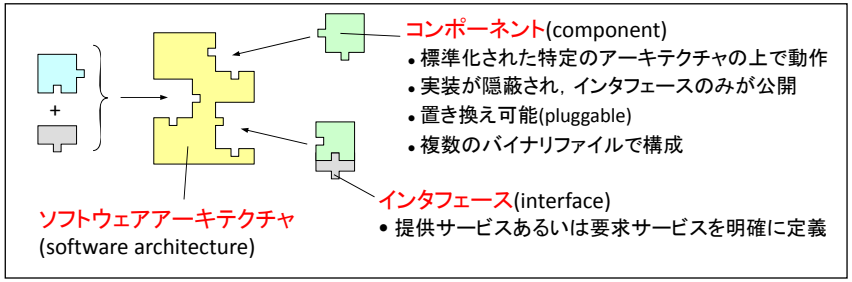
コンポーネント



コードの再利用

[コード + インタフェース]の再利用

[(コード) ＋ インタフェース + 設計]の再利用



ソフトウェア開発管理

開発管理

- ソフトウェア開発において、誰がどの段階で何をすればよいのか計画する
  - ✓ 要求を満たすソフトウェアを、決められた予算と期間で納品することが目的
- 計画通りにプロジェクトが進んでいるかの確認と、その結果に応じた対策
- 従来の管理
  - ✓ QCDの3つの項目が中心
    - 品質(quality)、コスト(cost)、納期(delivery)
- PMBOK(project management body of knowledge)
  - ✓ プロジェクト管理に必要な知識を9つの視点で体系化
    - 統合管理、スコープ管理、時間管理、コスト管理、品質管理、人材管理、コミュニケーション管理、リスク管理、調達管理

PMBOKによるプロジェクトの作業プロセス

- 立ち上げプロセス
  - ✓ プロジェクトまたはそのフェーズを定義し認可
- 計画プロセス
  - ✓ プロジェクトの目標を定義・洗練
  - ✓ 活動計画の策定
  - ✓ 資源の調達
- 遂行プロセス
  - ✓ プロジェクト計画を実施
- 制御プロセス
  - ✓ プロジェクト計画との差異を認識するために、実績報告や進捗測定を実施
- 終結プロセス
  - ✓ 契約完了手続きを行い、プロジェクトを終了

開発計画の構成要素

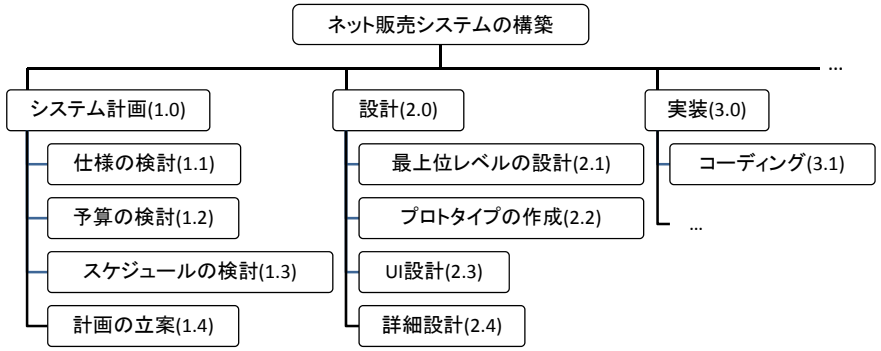
- 開発の目的
  - ✓ 何のためにこの開発が行われるか
- 開発の目標
  - ✓ システム利用者の要望、業務運営上の方針など
- 開発対象業務および運用方針
  - ✓ 開発対象の範囲と機能など
  - ✓ 業務上の制約や利用者に関する前提条件など
- 開発システムの基本構成
  - ✓ 開発するソフトウェアが実際に稼働する環境など
- 開発工数と開発コスト
  - ✓ 開発に要する工数とコスト、およびそれらの見積もり
- 開発スケジュール
  - ✓ 開発工数に応じた開発期間の決定
    - 作業明細構造(WBS: work breakdown structure)やガントチャートの活用

開発計画の構成要素

- 開発体制
  - ✓ 開発工数とスケジュールに応じた組織やチーム
  - ✓ 必要要員の割り当て計画
- 開発環境や開発方法
  - ✓ 開発支援ツール
  - ✓ 開発方法論、コーディング規約など
- 成果物の管理方法
  - ✓ 各工程で作成される成果物を管理（構成管理）する方法
- リスク管理(risk management)
  - ✓ 開発を進めていくうえで発生が予想されるリスクとその対処方法
    - リスク衝撃(risk impact)：リスクによる損失
    - リスク確率(risk probability)：リスクが発生する可能性
    - リスク制御(risk control)：リスクの影響を最小化、回避する行動

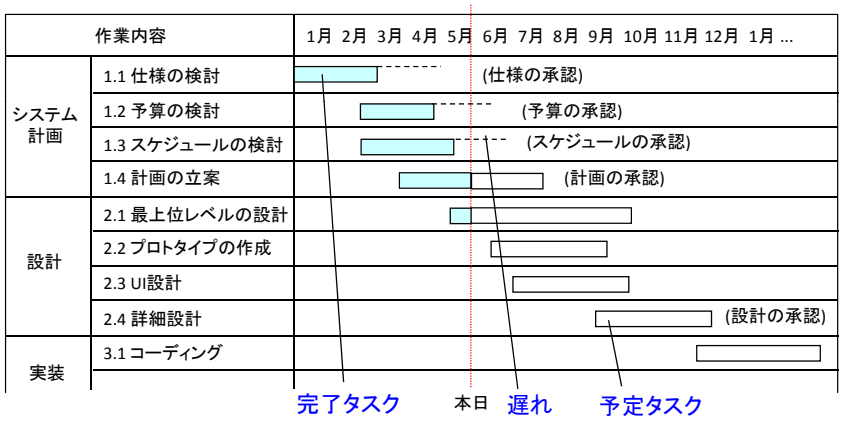
作業明細構造

- プロジェクトで実施される作業（アクティビティ）を分割し、詳細化して記述
  - ✓ 作業の明確化



ガントチャート

- 作業明細構造におけるそれぞれの作業を並列に記述
  - 各作業に対する期間を棒グラフで表示
  - 進捗の明確化



開発工数の見積もり技法

- 標準タスク法
  - ✓ ソフトウェア開発に必要とされる標準的な作業（タスク）ごとに開発工数をあらかじめ設定
  - ✓ 実際の開発に現れる作業を予測し、それらに応じて工数を積み上げることで全体の開発工数を算出
- COCOMO(constructive cost model)[Böhm]
  - ✓ 開発ソフトウェアの規模（予測規模）から開発工数を算出
    - 開発規模と開発工数の統計的モデルを利用
- COCOMO 2.0
  - ✓ オブジェクトポイントやファンクションポイントからソフトウェアの規模を予測し開発工数を算出
- ファンクションポイント法（FP法）
  - ✓ ソースコードの行数の代わりに、入力や出力などの機能数(function point)から規模を算出
  - ✓ 算出された規模から開発工数への変換方法は未定義

標準タスク法

標準タスクAの作業工数

規模＼複雑度	単純	普通	複雑
小	1	2	3
中	1.5	3	5
大	2	4	7

プロジェクトXにおける標準タスクAの工数

規模＼複雑度	単純	普通	複雑
小	1×10	2×5	3×0
中	1.5×10	3×30	5×5
大	2×0	4×10	7×10

プロジェクトXにおける標準タスクAの件数

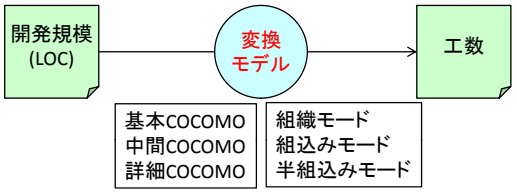
規模＼複雑度	単純	普通	複雑
小	10	5	0
中	10	30	5
大	0	10	10

10 + 10 + 0 + 15 + 90 + 25 + 0 + 40 + 70
= 260（標準タスクAの総工数）

プロジェクトXの総工数
= 標準タスクAの総工数
+ 標準タスクBの総工数
+ 標準タスクCの総工数
+ ….

COCOMO

- 基本COCOMO
  - ✓ 開発規模のみから算出
  - ✓ 計画段階で利用
- 中間COCOMO
  - ✓ 開発するソフトウェアの特徴、メンバの経験や能力で調整して算出
  - ✓ 要求定義が完了した後で利用
- 詳細COCOMO
  - ✓ モジュール構成などを考慮し、開発希望を調整して算出
  - ✓ 設計終了後に利用



COCOMO

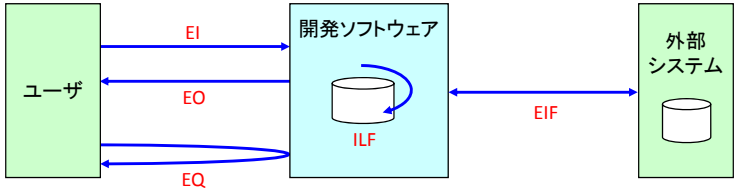
- COCOMO開発モード
  - ✓ 組織モード: 少人数で行う小規模システムの開発
  - ✓ 半組込みモード: 一般の業務システムの開発
  - ✓ 組込みモード: 厳しい制約を持つ大規模システムの開発
- COCOMO 2.0
  - ✓ アプリケーション組み立てモデル
    - GUIビルダーでの開発やプロトタイピングのような初期段階で適用
    - オブジェクトポイント法(オブジェクトの数)で規模を算出
  - ✓ 初期設計モデル
    - システム構造が決定される前に適用
    - FP法に基づき機能数で規模を算出
  - ✓ ポストアーキテクチャモデル
    - システム構造が決定された後に適用
    - FP法に基づく機能や行数で規模を算出

ファンクションポイント法における処理

- 外部入力(EI: external inputs)
  - ✓ 外部から開発ソフトウェアへの入力
  - ✓ 開発ソフトウェアの内部論理ファイルの更新あり
- 外部出力(EO: external outputs)
  - ✓ 開発ソフトウェアから外部(利用者など)への出力
- 外部照会(EQ: external inquires)
  - ✓ 外部から開発ソフトウェアへの照会
  - ✓ 開発ソフトウェアの内部論理ファイルの更新なし
- 内部論理ファイル(ILF: internal logical files)
  - ✓ 開発ソフトウェア内部に存在するファイルの規模
  - ✓ ファイル数ではなく、論理レコードの数
- 外部インタフェースファイル(EIF: external interface files)
  - ✓ 他のシステムで管理され、開発ソフトウェアが参照するファイルの規模
  - ✓ ファイル数ではなく、論理レコードの数

ファンクションポイント法(1)

- ソフトウェア内部の処理を5つの機能に分類
  - ✓ それぞれの機能に対して、複雑度ごとに機能数を数え上げ



EIの複雑度

関連ファイル数	データ項目数		
	1～4	5～15	16～
0～1	単純	単純	普通
2	単純	普通	複雑
3～	普通	複雑	複雑

EO, EQの複雑度

関連ファイル数	データ項目数		
	1～5	6～19	20～
0～1	単純	単純	普通
2	単純	普通	複雑
3～	普通	複雑	複雑

ILF, EIFの複雑度

関連ファイル数	データ項目数		
	1～19	20～59	60～
0～1	単純	単純	普通
2	単純	普通	複雑
3～	普通	複雑	複雑

ファンクションポイント法(2)

- 複雑度別の機能数に重み付け係数を掛けて合計値を算出

機能＼複雑度	単純	普通	複雑
EI	10	12	14
EO	11	13	15
EQ	1	3	5
ILF	2	4	6
EIF	3	5	7

複雑度別の機能数

機能＼複雑度	単純	普通	複雑
EI	×3	×4	×6
EO	×4	×5	×7
EQ	×3	×4	×6
ILF	×7	×10	×15
EIF	×5	×7	×10

重み付け係数

機能＼複雑度	単純	普通	複雑
EI	10×3	12×4	14×6
EO	11×4	13×5	15×7
EQ	1×3	3×4	5×6
ILF	2×7	4×10	6×15
EIF	3×5	5×7	7×10

30 + 48 + 84
+ 44 + 65 + 105
+ 3 + 12 + 30
+ 14 + 40 + 90
+ 15 + 35 + 70
= 700 (未調整FP)

ファンクションポイント法(3)

- システムの特性に応じて得点を割り当ててFP値を調整

システム特性	ポイント
1 データ通信	0
2 分散処理	0
3 パフォーマンス	4
4 高負荷環境	4
5 トランザクション量	2
6 オンラインデータ入力	1
7 エンドユーザの作業効率	2
8 マスターデータベースのオンライン更新	2
9 内部処理の複雑さ	3
10 再利用を考慮した設計	4
11 導入の容易性	1
12 運用の容易性	3
13 複数サイトでの使用	0
14 変更の容易性	2
合計	28

0: まったく関係ない
1: ほとんど影響を受けない
2: 適度に影響を受ける
3: 平均的な影響を受ける
4: 大きな影響を受ける
5: 非常に大きな影響を受ける

調整用係数
= 0.65 × (0.01 × 28)
= 0.182
FP
= 0.182 × 700
= 127.4

調整用係数 = 0.65 + (0.01 × システム特性の合計)
FP = 調整用係数 × 未調整FP

品質管理

- ソフトウェアの品質を向上させる作業(開発管理とは独立の組織で実施)
  - ✓ 品質保証(quality assurance)
    - 高品質ソフトウェアを提供するための組織的な手続きや基準のフレームワークを構築
  - ✓ 品質計画(quality planning)
    - 適切な手続きや基準の選択と開発プロジェクトへの適合
  - ✓ 品質管理(quality control)
    - 計画した手続きや基準に開発チームが従うことを保証するためのプロセスの定義や執行
- ソフトウェア開発における成果物(プロダクト)やプロセスの評価が必須
  - ✓ ソフトウェアメトリクス(software metrics)で計測
    - 開発の実態を定量的に評価する手段(数値、尺度、属性など)

ソフトウェアメトリクス

- 開発標準との比較や改善に利用
- プロダクトメトリクス(product metrics)
  - ✓ ソースコードの規模
    - LOC (lines of code): 行数
    - ステップ数: プログラムの命令だけを数えたもの
    - Halsteadの尺度: 演算子と非演算子の種類数や出現数から計測
  - ✓ 複雑さ
    - サイクロマティック数(cyclomatic number)[McCabe]
    - CKメトリクス[Chidamber & Kemerer]
- プロセスメトリクス(process metrics)
  - ✓ 作業効率を計測
    - 作業に費やした時間や資源の量
    - 特定のプロセスの実行中に発生したイベントの数(コードインスペクションで発見された誤りの数など)

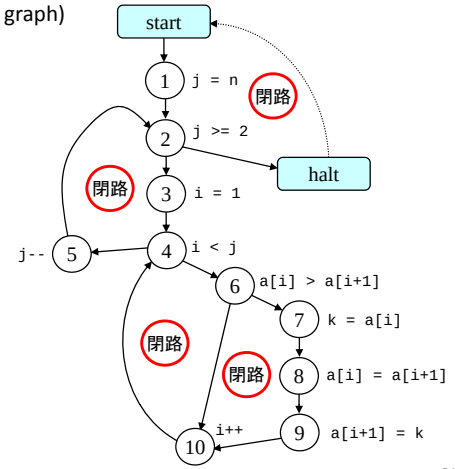
サイクロマティック数

- プログラムの流れを有向グラフ(CFG)で表現し、一次独立な閉路の数で複雑度を測定
  - ✓ CFG: 制御フローグラフ(control flow graph)

(例)ソートプログラム

```
for (int j = n; j >= 2; j--) {
  for (int i = 1; i < j; i++) {
    if (data[i] > data[i+1]) {
      int k = a[i];
      a[i] = a[i+1];
      a[i+1] = k;
    }
  }
}
```

サイクロマティック数
= 閉路の数
= 4

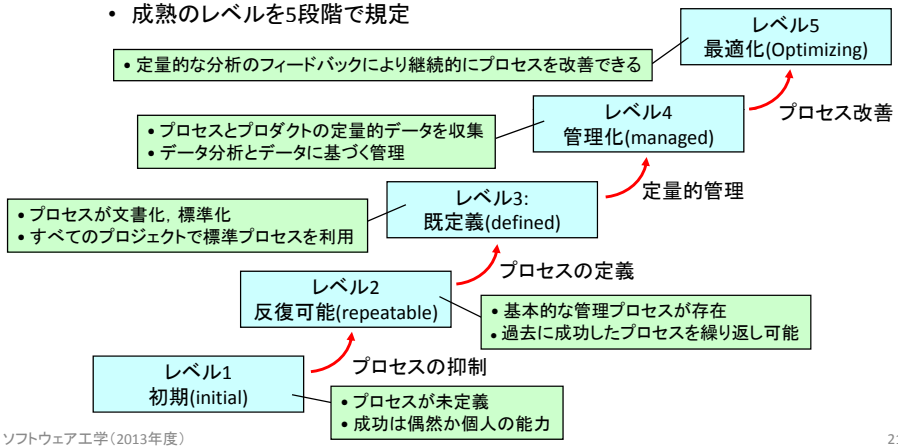


CKメトリクス

- オブジェクト指向プログラムにおけるクラスの複雑度を測定
- WMC(Weighted Methods per Class)
  - ✓ メソッドの複雑度の総和
- DIT(Depth of Inheritance Tree)
  - ✓ 継承の深さ(スーパークラスの数)
- NOC(Number of Children)
  - ✓ サブクラスの数
- CBO(Coupling Between Objects)
  - ✓ 結合(他クラスのメソッドを実行、フィールドを参照)しているクラスの数
- RFC(Response For a Class)
  - ✓ 別のクラスのインスタンスから受け取ったメッセージに対応して実行されるメソッドの数
- LCOM(Lack of Cohesion in Methods)
  - ✓ 同じフィールドを共有するメソッドの割合

プロセス成熟度

- プロダクトの品質を高めるためにプロセスの品質を改善
  - ✓ CMM(capability maturity model)[SEI]
  - ✓ SW-CMM: CMMIのうちソフトウェアに関係するもの
    - 成熟のレベルを5段階で規定



演習問題

1. ステークホルダとは何か
2. ステークホルダが増えるとどのような問題が生じるか
3. ソフトウェアの6つの品質特性とは何か

4. ウォーターフォールモデルの工程を説明せよ
5. ウォーターフォールモデルの欠点は何か
6. インクリメンタル開発プロセスモデルとイテラティブ開発プロセスモデルの違いを説明せよ

7. 要求とは何か
8. 要求分析が難しいと考えられる理由を説明せよ
9. データフロー図を構成する4要素は何か
10. データフロー図の詳細化において守るべき規則は何か
11. 実体関連図は何を表すものか

12. オブジェクト指向におけるカプセル化とは何か
13. クラスとインスタンスの関係を説明せよ
14. ユースケースとは何か

15. アーキテクチャスタイルの一つである3層モデルについて説明せよ
16. STS分割は何に着目してモジュール分割する手法か説明せよ
17. モジュールの機能独立性とは何か
18. 情報隠蔽の利点は何か
19. モジュール強度、モジュール結合度をそれぞれ説明せよ

20. プログラミングにおいて理解しやすいプログラムを作成すべき理由は何か
21. 適正プログラムとは何か
22. 構造化定理とは何か

23. テストの重要な性質を説明せよ
24. 単体テストはどのような誤りを検出するテストか説明せよ
25. ボトムアップテスト、トップダウンテストそれぞれの利点と欠点を説明せよ
26. ブラックボックステストとホワイトボックステストの違いは何か

27. ソフトウェア保守の4分類について説明せよ
28. 保守作業においてソフトウェア理解が重要であるのはなぜか

29. 開発計画を立てる理由は何か
30. 標準タスク法によって開発工数を見積もる手順を説明せよ