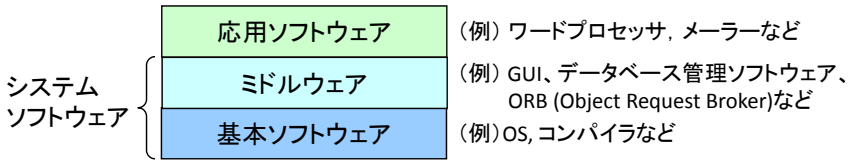
ソフトウェア工学（R1）
（2011年度）

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/2011/se/

ソフトウェア工学とは

ソフトウェア

- ソフトウェア(software)
  - ✓ 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
- 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

ソフトウェアの一般的特性

1. ソフトウェアは目に見えない製品
2. 品質の劣化なし，向上していく傾向
3. 潜在的バグが潜んでいる
4. 修正可能
5. 複写可能
6. 要求される機能は社会情勢とともに絶えず変化
7. 波及効果が生じる
8. バグは本人より第三者のほうが見つけやすい
9. 作成者の思想

ソフトウェア工学

- ソフトウェア工学(software engineering) [1968年のNATO会議]
  - ✓ 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(software development)の技術体系および学問体系
    - 方法論(methodology)
    - 技法(technique)/道具(tool)
    - プロジェクト管理(project management)
  - ✓ 大規模・高信頼ソフトウェアの開発 ≠ プログラミング

- ソフトウェア工学の目的: よいソフトウェアをうまく開発すること
  - ✓ よいソフトウェアとは？
    高信頼、保守が容易、拡張が容易、利用が簡単、高速、…
  - ✓ うまく開発するとは？
    開発期間、コスト（人件費）、…

ソフトウェア危機

- ソフトウェア危機: 1960年代後半～
  - ✓「規模」の問題（1970年代）:
    ハードウェアの大型化に伴う大規模ソフトウェアの必要性
    → 構造化プログラミング（構造化分析，設計，コーディング）
  - ✓「量」の問題（1980年代）:
    コンピュータシステムの普及に伴う開発ソフトウェア数の増加
    → 統合的ソフトウェア開発支援環境や部品化・再利用
  - ✓「質」「インタフェース」の問題（1990年代）:
    - 社会的に重要な役割を担う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要求
    - オープン化やネットワーク化に伴う接続性，移行性，互換性の要求
  - ✓「複雑さ」「変化」の問題（2000年代）:
    - 扱う対象が専門的かつ広範囲
    - 動作環境や社会的要求が流動的あるいは急激に変動

ソフトウェアの品質特性（ISO9126）

1. 機能性(functionality): 必要な機能実装の度合い
   - ✓ 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
   - ✓ セキュリティ: 不当なアクセスを排除できるか
   - ✓ 相互運用性: データやコマンドがやり取りできるか
   - ✓ 標準適合性: 企画や標準にフォーマットが合致しているか
2. 信頼性(reliability): 機能が正常に動作し続ける度合い
   - ✓ 成熟性: 故障する頻度が少なくなったか
   - ✓ 障害許容性: 障害に対して許容できるか
   - ✓ 回復性: 故障したときに早く復旧できるか
3. 使用性(usability): 分かりやすさ、使いやすさの度合い
   - ✓ 理解性: 使い方がわかりやすいか
   - ✓ 習得性: 初めてでもすぐに使えるようになるか
   - ✓ 運用性: 管理するのは楽か

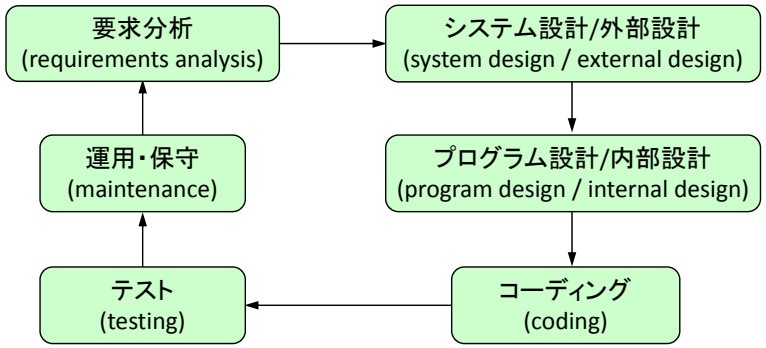
ソフトウェアの品質特性（ISO9126）

4. 効率性(efficiency)：目的達成のために使用する資源の度合い
   - ✓ 時間的効率性：処理速度が速いか
   - ✓ 資源効率性：メモリなどの資源を多く必要としないか
5. 保守性(maintainability)：改訂作業に必要な労力の度合い
   - ✓ 解析性：プログラムがわかりやすいか
   - ✓ 変更性：プログラムが変更しやすいか
   - ✓ 安定性：変更時に障害が混入しないか
   - ✓ 試験性：テストがしやすいか
6. 移植性(portability)
   - ✓ 設置性：インストールは簡単か
   - ✓ 環境適応性：いろいろな環境（OSなど）で使えるか
   - ✓ 置換性：他のソフトウェアの置き換え可能か
   - ✓ 規格準拠性：規格や規約に適合しているか

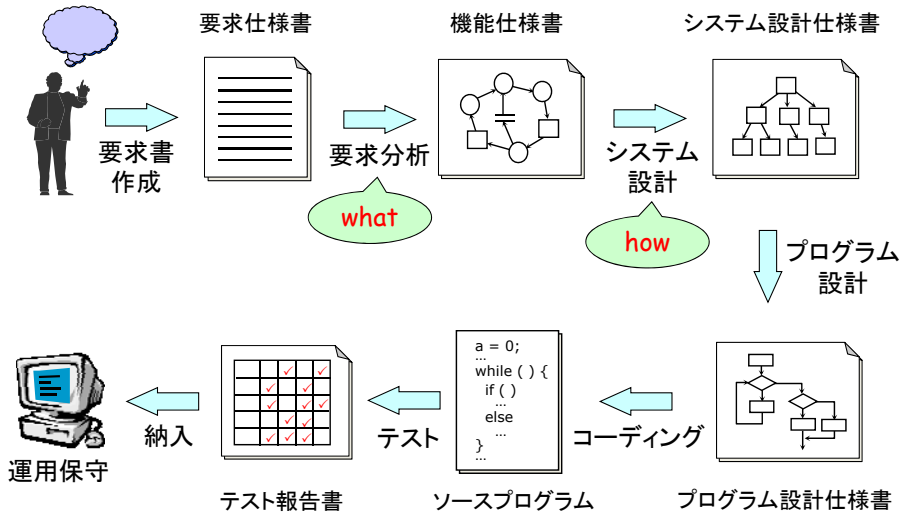
ソフトウェア開発モデル

ソフトウェア開発モデル

- ソフトウェア開発(software development)
  - ✓ 誰が(who)：プロジェクト(project)
  - ✓ 何を(what)：プロダクト(product)
  - ✓ どのように(how)：プロセス(process)
- ソフトウェアのライフサイクル(life cycle)



ソフトウェア開発プロセス



ソフトウェア開発プロセス(分析)

1. システムの要求書作成
   - ✓ 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
     - システム要求書(system requirements)
     - 要求仕様書(requirements specification)

2. システムの要求分析(requirements analysis)と
   システム定義(system definition)
   - ✓ どのようなシステムを作成するのかを決定(分析者: analyst)
     システム要求書を分析し、要求システムを形式的に文書化
     - システム機能仕様書(functional system specification)
     - 機能仕様書(functional specification)

構造化分析(structured analysis)

ソフトウェア開発プロセス(設計)

3. システム設計(system design)/外部設計(external design)
   - ✓ システムをどのように作成するのかを決定(設計者: designer)
     モジュール(module)構成、個々のモジュールの機能、
     モジュール間のインタフェース(interface)を決定
     - システム設計仕様書(system design specification)
     - 外部設計仕様書(external design specification)

4. プログラム設計(program design)/内部設計(internal design)/
   詳細設計(detailed design)
   - ✓ 個々のモジュールの内部構造を決定(プログラマ: programmer)
     アルゴリズムとデータ構造、処理手順を決定
     - プログラム設計仕様書(program design specification)
     - ロジック設計仕様書(logic design specification)

構造化設計(structured design)

ソフトウェア開発プロセス(実装)

5. コーディング(coding)
   - ✓ プログラム設計仕様をプログラムに変換(実装者: coder)
     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)

構造化プログラミング(structured programming)
- 分割・統治
- 段階的詳細化
- 3つの基本制御(逐次・選択・反復)

構造化プログラミング

- 構造化プログラミング(structured programming)[IBMの技術規範IPT]
  記述が容易 → 理解が容易な(簡単で分かりやすい)プログラムの構築技法

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プログラム(モジュール: module)で合成

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. プログラムを3つの基本制御の論理構造で構築
   - 連接、逐次(sequence)
   - 選択(selection)
   - 反復(iteration)

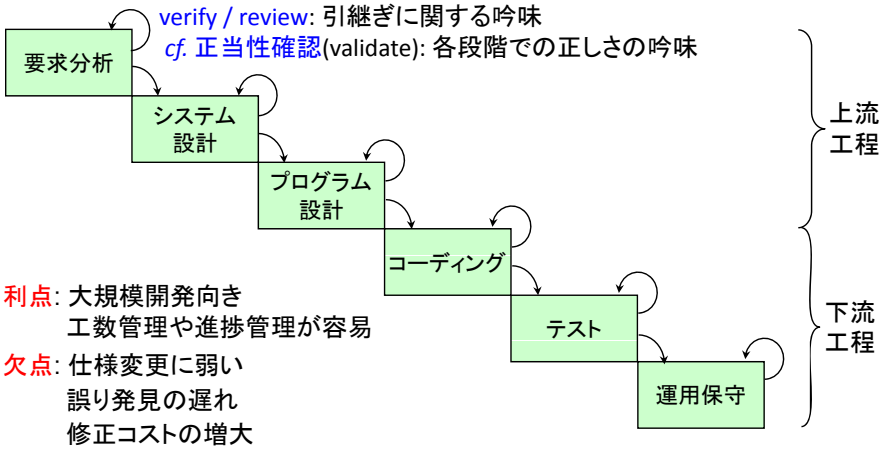
ソフトウェア開発プロセス（テスト、保守）

6. ソフトウェアテスト(software test)
   - ✓ 仕様書通りにプログラムが動作するかどうかを検査（試験者：tester）
     モジュールテスト(module test)/単体テスト(unit test)/
     集積テスト/統合テスト(integration test)/
     システムテスト(system test)/機能テスト(function test)/
     出荷テスト(shipping test)/受入れテスト(acceptance test)
   - ✓ ソフトウェア作成者と独立の組織
     - テスト報告書(test report)：エラー(error)、故障(fault)
       → デバッグ(debug)：エラーや故障の修正（設計者、プログラマ）

7. ソフトウェア保守(software maintenance)
   - ✓ 納入後のソフトウェアを管理（CE：customer engineer）
     運用段階で検出された故障（残存エラー）の修正
     手直し要求：新機能の追加、既存機能の変更、新しい環境への適合

ウォーターフォールモデル

- ウォーターフォールモデル(waterfall model)
  トップダウンな開発プロセス



利点：大規模開発向き
　　工数管理や進捗管理が容易
欠点：仕様変更に弱い
　　誤り発見の遅れ
　　修正コストの増大

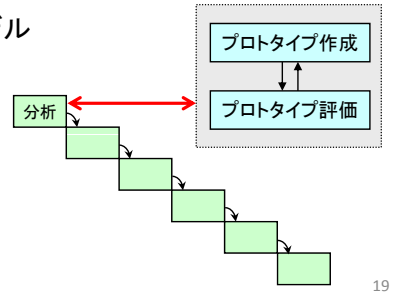
ソフトウェア開発モデルの推移

- 1960年代：流れ図(flowchart)、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パラダイム(paradigm)

1. ソフトウェアプロトタイピング(software prototyping)
   - ✓ システム設計時に試作品（プロトタイプ： prototype）を構築

2. インクリメンタル(incremental)開発プロセスモデル
   - ✓ システムを独立したサブシステムに分割し、
     サブシステムごとに開発

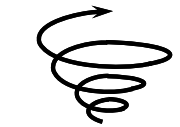
3. イテラティブ(iterative)開発プロセスモデル
   - ✓ システム全体に対して繰り返し修正を加える



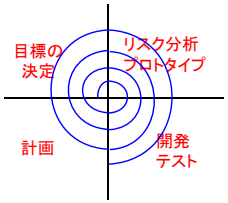
ソフトウェア開発モデルの推移

4. オブジェクト指向ソフトウェア開発(object-oriented software development)
   - ✓ オブジェクトを単位としてソフトウェアを構築
     オブジェクト：実世界の「もの」や「役割」などを抽象化したもの

5. アジャイルソフトウェア開発(agile software development)
   （例）XP(extreme programming), SCRUM, Crystal
   アジャイルアライアンス宣言(manifesto)
   - ✓ プロセスやツールよりも、個人や人同士の交流を重視
   - ✓ 包括的なドキュメントよりも、動作するソフトウェアを重視
   - ✓ 契約上の交渉よりも、顧客との協調を重視
   - ✓ 計画に従うことよりも、変化に対応することを重視
     （注）左側の項目を軽視するという意味ではない



6. スパイラルモデル(spiral model)
   - ✓ リスク（開発の失敗をもたらす可能性のある要因）の観点から
     開発プロセスを改善する

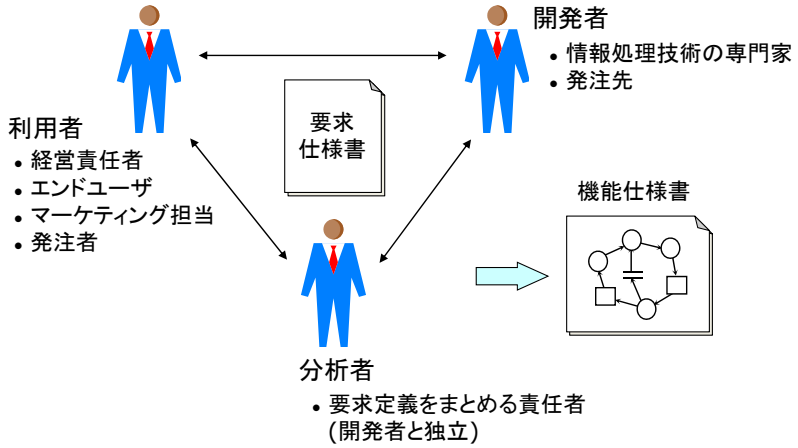


Boehm (1988)

要求分析

要求分析

- 開発すべきシステム全体の仕様をできるだけ厳密に定義



要求分析の作業

1. 要求獲得
   利用者が真に望むものを引き出し要求としてまとめる
   - ✓ 資料収集、現場観察
   - ✓ インタビュー、アンケート、ブレーンストーミング
   - ✓ プロトタイピング
   - ✓ シナリオ、ユースケース
   - ✓ ゴール指向分析、問題フレーム
2. 要求仕様化
   獲得した要求から誤りや冗長、曖昧さ、矛盾を除去し、不足を補って
   要求仕様を完成させる
3. 要求確認
   作成された要求仕様の正しさを確認する
   - ✓ 妥当性(正当性)、非曖昧性、完全性、一貫性
   - ✓ 重要度と安定度の順位付け
   - ✓ 検証可能性、変更可能性、追跡可能性

要求分析の課題

1. 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
   - ✓ 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
2. 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
3. 開発者の技術的課題認識の甘さ
   - ✓ 開発期間や開発費用の過小評価

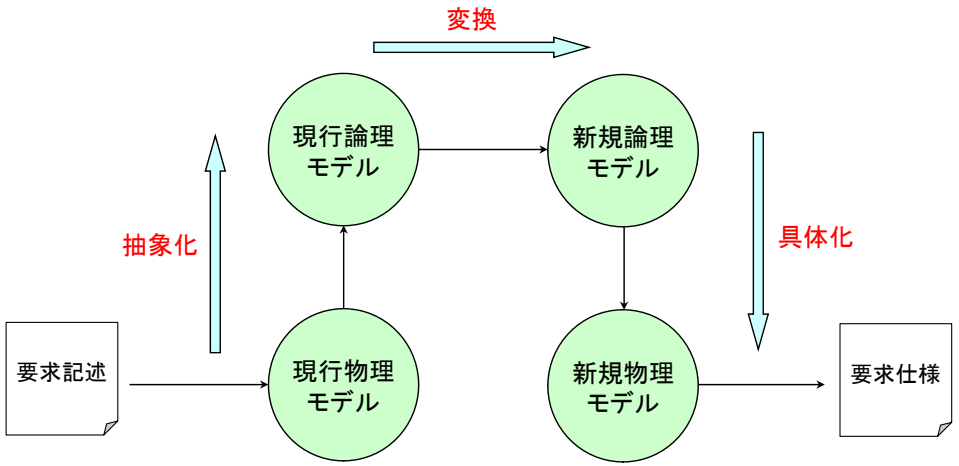
⇩

要求分析技法(モデル化技法と形式化された図式)の必要性

要求分析技法

- 分析時の観点による分類
  1. 機能:
     a. データの流れとそのデータを処理する機能に着目
        データフロー図の利用
        → 構造化分析(SA: structured analysis)
     b. ユーザの利用方法に着目
        ユースケース(use-case)とシナリオ(scenario)を利用
  2. データ: システム内のデータ構造とデータ間の制約に着目
     実体関連図を利用
  3. オブジェクト: データと機能(操作)をカプセル化して扱う
     オブジェクト指向分析
  4. プロセス: 実世界の問題の処理の流れに注目

構造化分析技法



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

構造化分析の手順

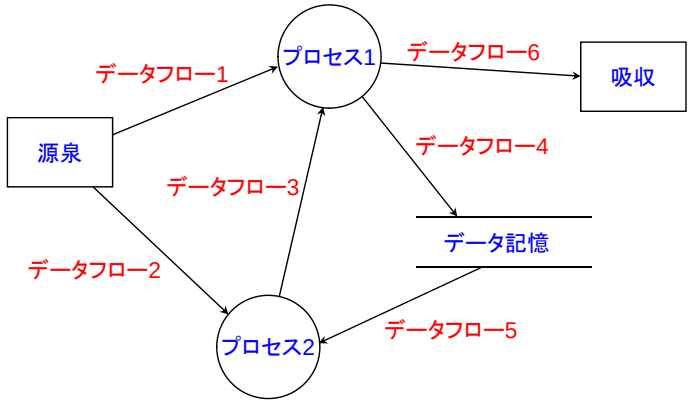
1. 現行物理モデルの構築
   現状の業務(人手部分＋機械化部分)を分析
   - ✓ 物理的なレベル(ありのまま)で正確に表現
2. 現行論理モデルの構築
   本質的な機能や問題の洗い出し
   - ✓ 本質的でない部分(人、組織、タイミング、媒体など)を除外
   - ✓ 本質的でない部分の抽象的な概念への置換
3. 新規論理モデルの構築
   新たに開発するシステムで解決すべき問題の洗い出し
   - ✓ 追加あるいは削除する機能の特定
   - ✓ 機械化部分と人手部分の境界の決定
4. 新規物理モデルの構築
   新規論理モデルを制約条件を満たすように具体化
   - ✓ 機械化部分と人手部分の境界の最終決定

仕様記述

- システム機能のモデル化
  - ✓ データフロー図(DFD: data flow diagram)
  - ✓ データ辞書(data dictionary)
  - ✓ プロセス仕様書
  - ✓ ユースケース
- 蓄積データのモデル化
  - ✓ 実体関連図(ER図: entity relationship diagram)
- 状態変化や制御手順を含む時系列動作のモデル化
  - ✓ 状態遷移図(state transition diagram)
  - ✓ リアルタイム用データフロー図

データフロー図

- システムを階層的かつ図的にモデル化
  - ✓ システム内のデータの流れを表現
  - ✓ 事象（イベント：event）のような制御は除外
    cf. フローチャート(flowchart)：制御の流れを表現



データフロー図

- 構文(syntax)：4つの基本記号のグラフ表現
  1. フロー(flow)：定常的なデータの流れを表現
     矢印にデータを表す名前を付加
  2. プロセス(process)：データの処理を表現
     個々のプロセスがどのような処理を行うのかを記述
     ＝ バブル(bubble)
  3. データストア(data store)：データを格納する場所
     格納するデータの名前（入出力フローのデータ名と同一）を記述
     ＝ ファイル(file)
  4. エンティティ(entity)：システムの外部にある組織や人を表現
     - 源泉(source)：データの発生源
     - 吸収(sink)：データの最終的な行き先
- 意味論(semantics)：各記号に付加された情報（名前）に依存

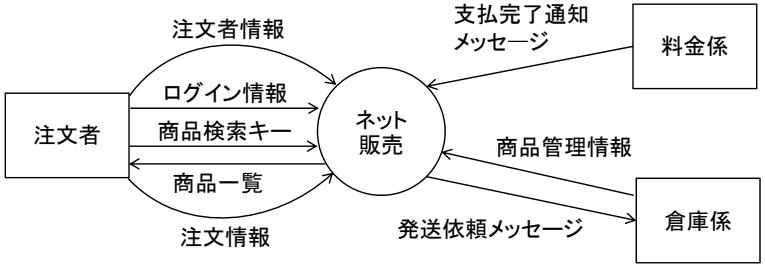
例題1：要求記述 （出典：高橋，丸山：ソフトウェア工学．森北出版，2010年）

ネット販売業務の要求記述

- （会員登録と会員情報の変更については省略）
- 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。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から該当する商品を検索し、見つかった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随時受け取り、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。

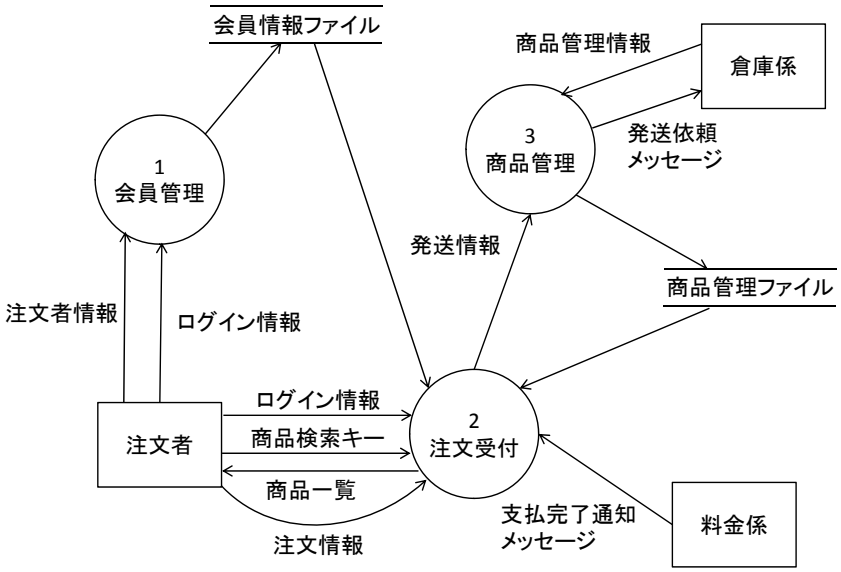
例題1：全体文脈図

- 全体文脈図(context diagram)：DFD記述の出発点
  - ✓ システム全体を1つのプロセスで表現
  - ✓ システムと外界とのデータのやり取りを表現

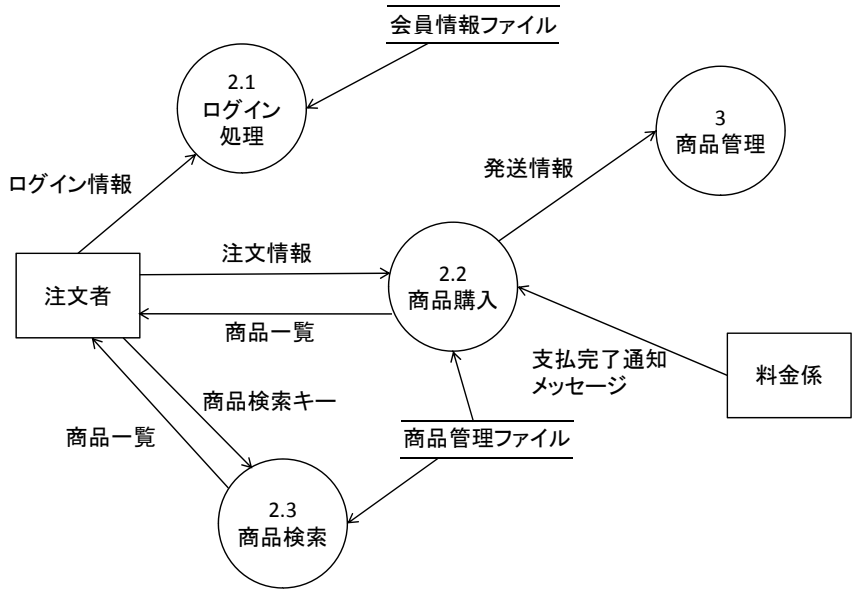


本例題における前提
- 論理レベルのDFD構築
- 業務の流れは現行モデルと同一

例題1: DFD

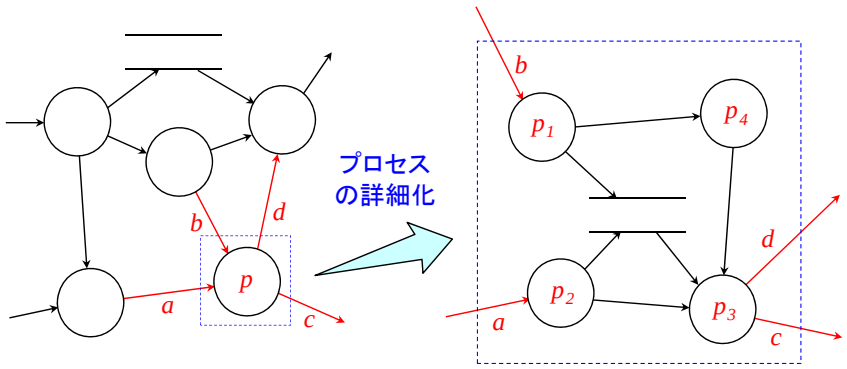


例題1: 詳細DFD

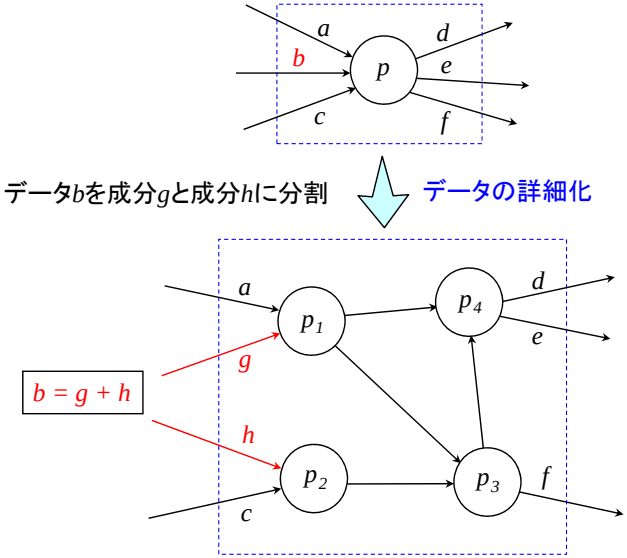


DFDの階層化

- DFDの階層化: プロセスの内部処理を詳細化
  - ✓ 着目する処理の入出力矢印は、詳細化の前後で維持



DFDのデータ詳細化



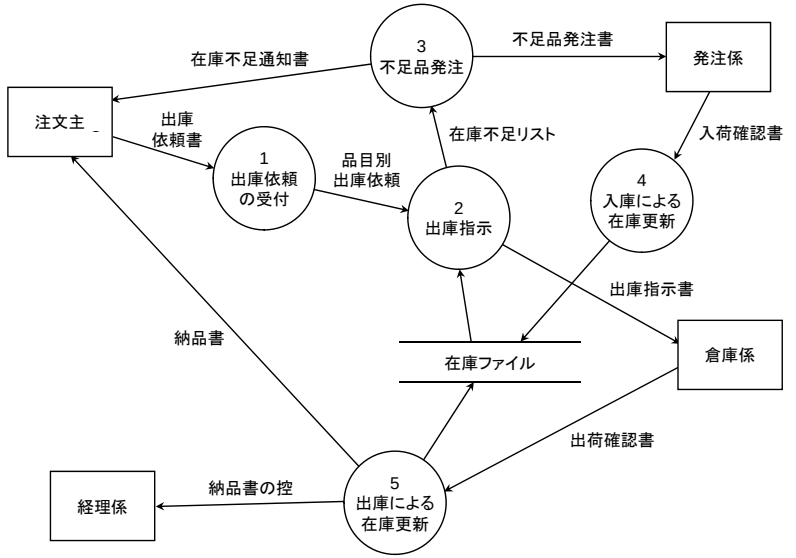
例題2：業務の記述

酒類販売の業務

- ある酒類販売会社の倉庫では、毎日数個のコンテナが搬入されてくる。その内容はビン詰めの酒で、1つのコンテナには10銘柄まで混載できる。扱い銘柄は約200種類ある。倉庫係は、コンテナを受取りそのまま倉庫に保管し、積荷表を受付係へ手渡す。また受付係からの出庫指示によって内蔵品を出庫することになっている。内蔵品を別のコンテナに詰め替えたり、別の場所に保管することはできない。
- 空になったコンテナはすぐに搬出される。
- さて受付係は毎日数十件の出庫依頼を受け、その都度倉庫係へ出庫指示書を出すことになっている。出庫依頼は出庫依頼書によるものとし、1件の依頼では、1銘柄のみに限られている。在庫が無いか数量が不足の場合には、その旨依頼者に電話連絡し、同時に在庫不足リストに記入する。そして当該品の積荷が必要量あった時点で、不足品の出庫指示をする。また空になる予定のコンテナを倉庫係に知らせることになっている。

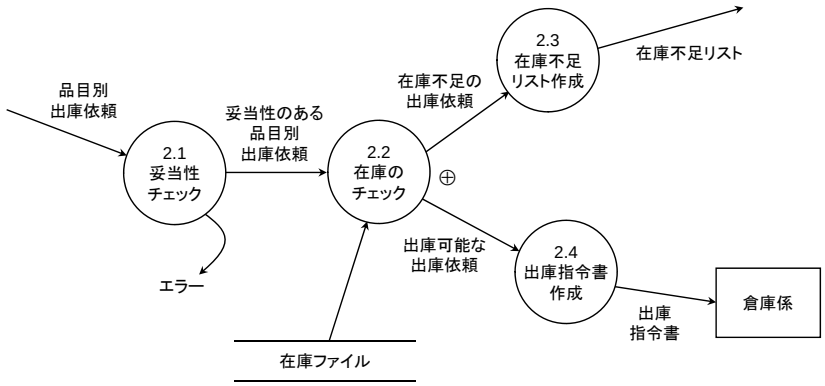
出典：山崎利治、「共通問題によるプログラム設計技法解説」
情報処理25-9、pp.934-962、1984年

例題2：DFD（レベル1）



出典：有沢誠著，「ソフトウェア工学」，岩波書店

例題2：DFD（レベル2）



*：複数のデータフローの結合を表す(AND)
⊕：複数のデータフローの分離を表す(OR)

データ辞書

- データ辞書(data dictionary)：DFDに出現するデータの構造を表現
  - ✓ 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）

データ構造の例)

注文書 = 注文番号 + 顧客名 + 送付先住所 + (電話番号) + 1{注文品目}10
　　+ 合計 + [領収書要|領収書不要]
注文品目 = 品番 + (品名) + 単価 + 数量 + 小計
品番 = 6{数字}6

プロセス仕様

- プロセス仕様：DFD記述の終了点
  - ✓ DFDの最下層のプロセスの基本処理を表現
    ＝ミニ仕様(mini spec)
    - 式
    - 原因結果グラフ(cause-effect graph)、決定表(decision table)
    - 構造化言語(e.g. PDL: program description language)

構造化言語によるプロセス仕様の例

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。
    …

実体関連図

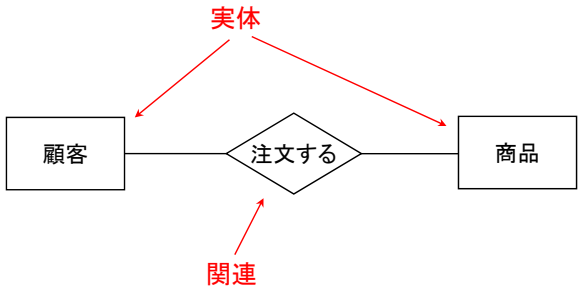
- 機能中心：同じデータを異なるファイルで重複して保存
  同じデータの仕様が異なる
  } 問題点
- データ中心：データのモデル化
  - ✓ データ(data)
    - 現実の世界に存在する実態を反映
    - 固有の特性（属性）を有する
    - 処理の仕方（処理手順）とは独立

⇩

実体関連図（ER図）

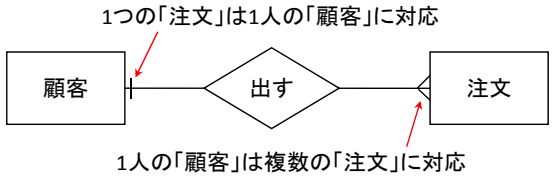
実体関連図

- 実体関連図（ER図）
  - ✓ 実体(entity)：システム内に存在する管理対象（人、物、金、場所など）
    名前を持つ四角形で表示
  - ✓ 関連(relationship)：実体間の結びつき
    関連名を持つ菱形で表示

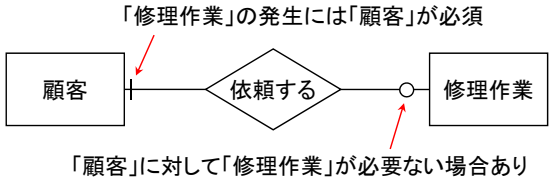


関連

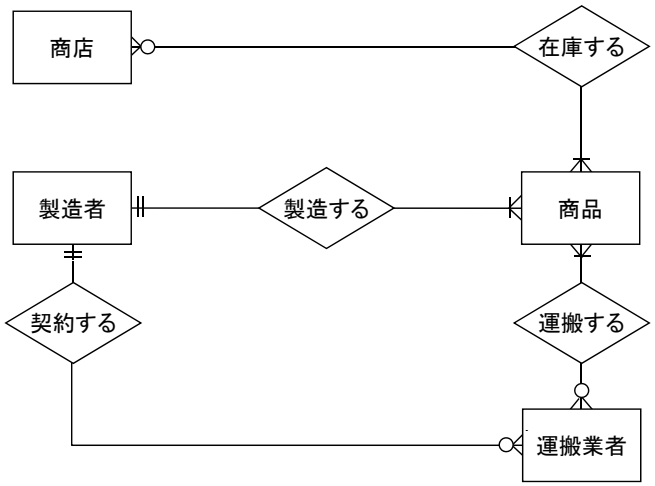
- カーディナリティ(cardinarity)
  関連するオブジェクトの数を表現（1:1, 1:N, M:N）



- モダリティ(modality)
  関連が必須であるかどうかを表現



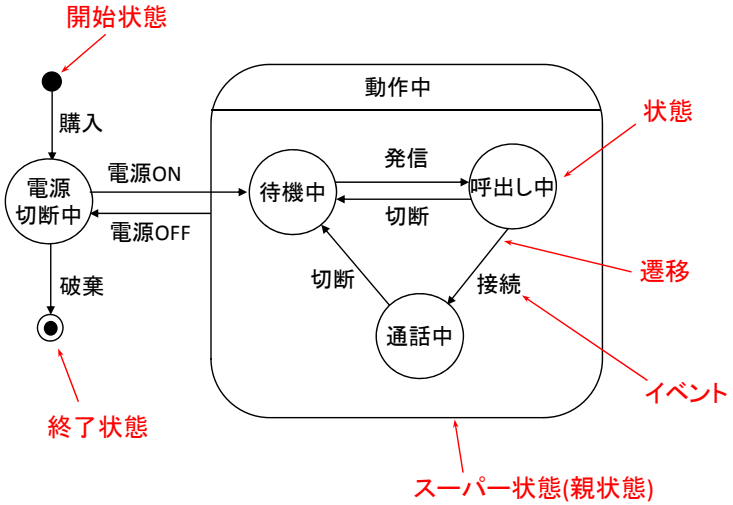
実体関連図の例



状態遷移図

- 状態遷移図/ステートチャート図(statechart diagram)
  - ✓ 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
  - ✓ 外部からのイベントに対するシステムの応答を表現
    - システムは必ず1つの状態に属する(複合状態を除く)
    - イベントは一瞬
    - 遷移時間は無視
    - イベントに対する応答は現在の状態によって変化

状態遷移図の例



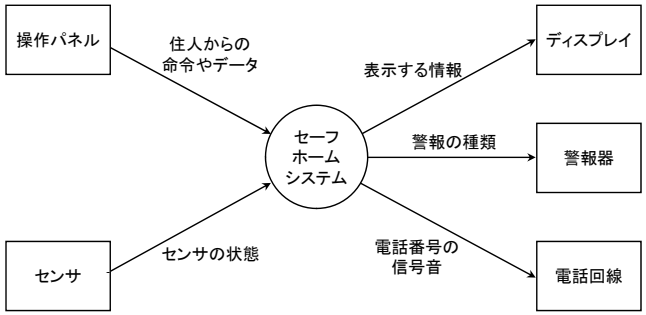
演習：システムの記述

ホームセキュリティシステムの仕様

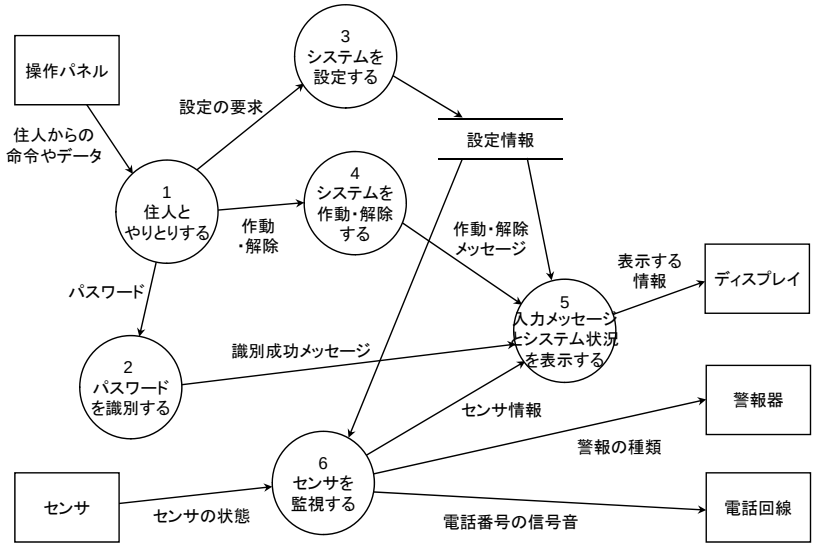
- セーフホームシステムは、設置時には住人がシステム設定をできるようにし、セキュリティシステムが接続されているすべてのセンサを監視する。操作パネルのキーパッドやキーを通して、住人とやりとりする。
- 設置時にシステムを設定するときには、操作パネルで行う。各センサに番号と種類を割りあて、システムの作動・解除を切りかえるマスタパスワードを設定し、センサイベント発生時の連絡先の電話番号を入力する。
- このソフトウェアはセンサイベントを検知した際に、システムに接続されている警報器を起動する。さらに、システム設定時に住人が指定した遅延時間を経過した時点で、監視サービスの電話番号に住所に関する情報を連絡し、検知したイベントについて報告する。電話回線への接続は、接続されるまで20秒間隔で繰り返される。
- セーフホームシステムとのすべてのやりとりは、ユーザインタフェースサブシステムによって管理される。このサブシステムは、キーパッドや特殊キーを通じて与えられた入力を読み取り、ディスプレイに入力メッセージとシステムの状況を表示する。
- (以下 略)

出典：R. Pressman著、「ソフトウェア工学の伝統的手法」、日科技連

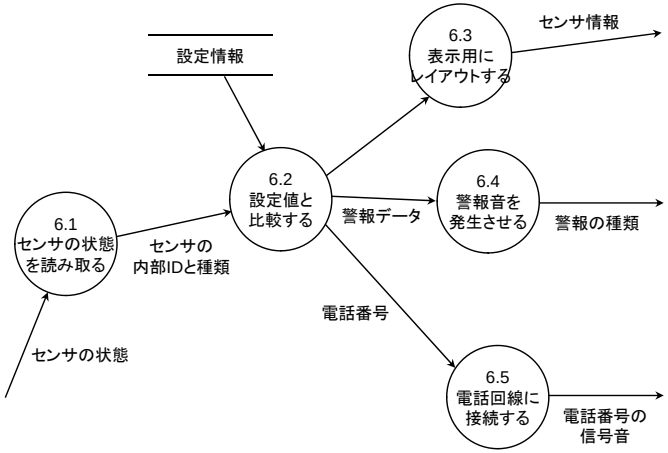
演習：全体文脈図（レベル0）



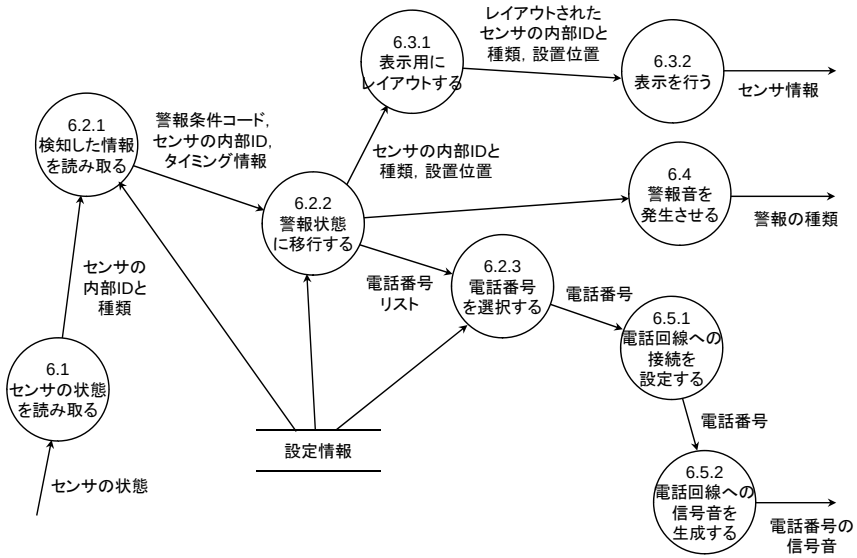
演習：DFD（レベル1）



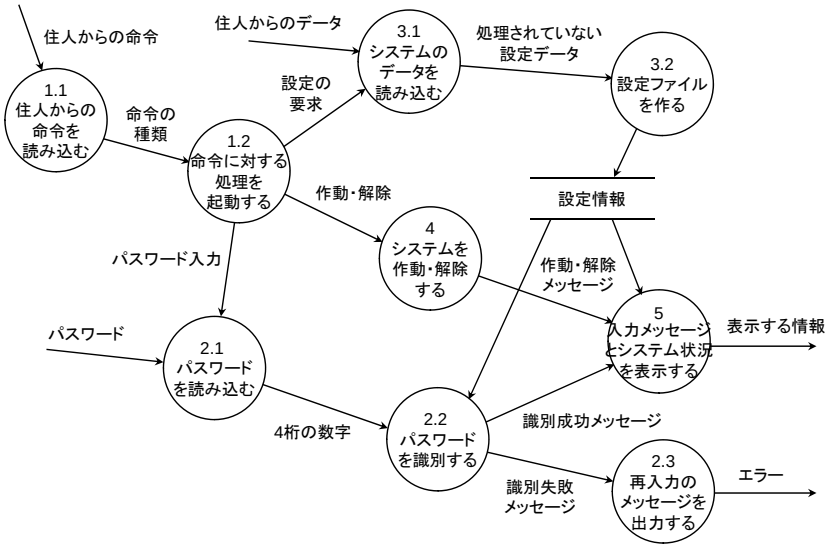
演習：DFD（レベル2）



演習：DFD（レベル3）



演習：DFD（レベル2）



オブジェクト指向分析

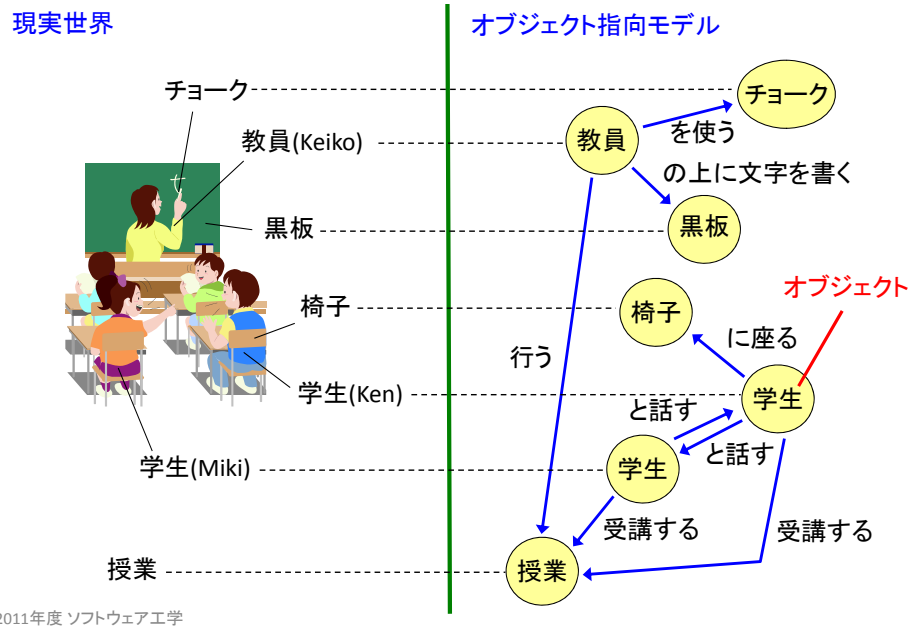
オブジェクト指向

- 現実世界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

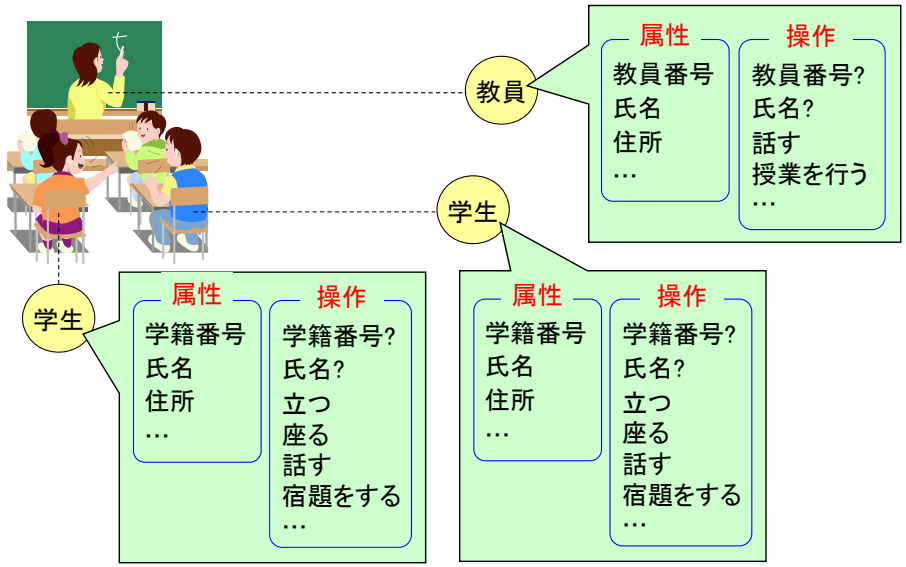
オブジェクト指向モデルの例



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」
  - ✓ 物理的な「もの」、役割や概念的な「もの」
  - ✓ データとそれに対する処理をまとめた「もの」
- オブジェクトの性質
  - ✓ 状態(state): オブジェクトの現在の様子
    ＝ 属性、プロパティ
  - ✓ 振る舞い(behavior): オブジェクトが実行できる動作
    ＝ 操作、メソッド
  - ✓ 識別性(identity): あるオブジェクトと他のオブジェクトを区別できること

属性と操作



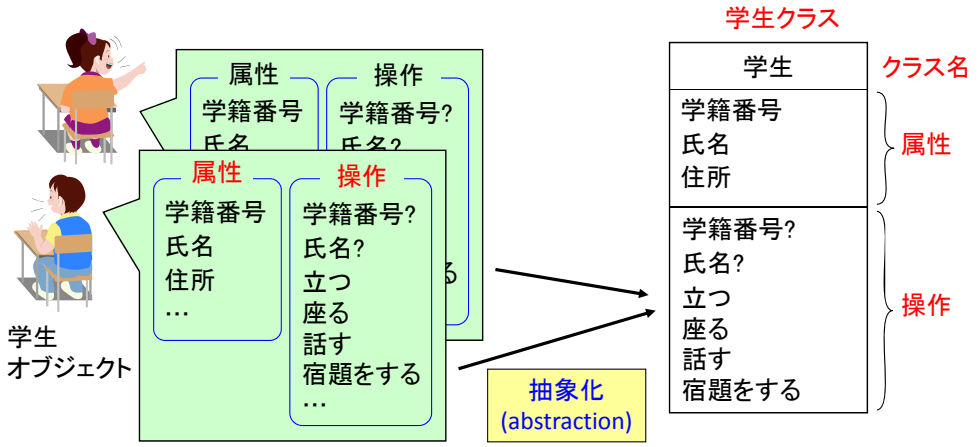
オブジェクト指向の基本概念

- クラスとインスタンス
- カプセル化
  - ✓ 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
  - ✓ 個々のオブジェクトに処理を依頼する仕組み
- 関連
  - ✓ 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
  - ✓ 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
  - ✓ 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

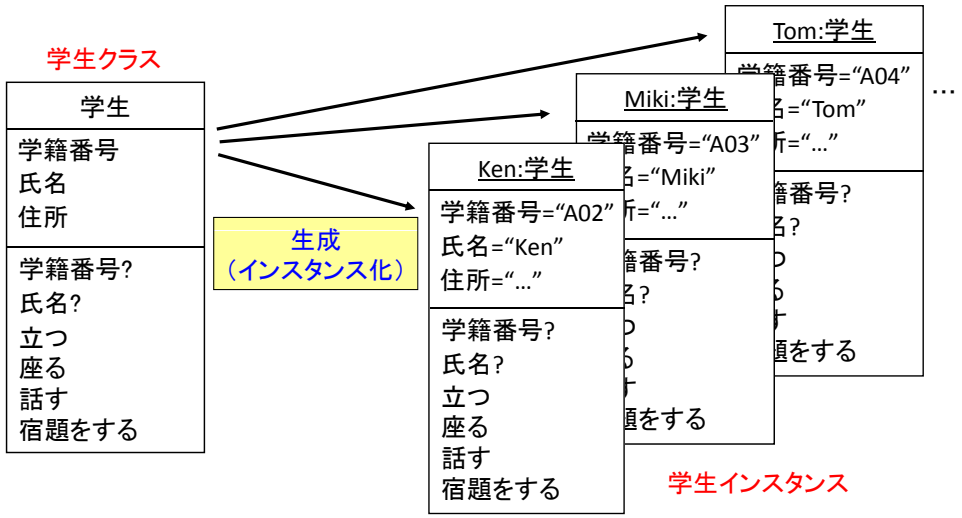
クラス

- 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

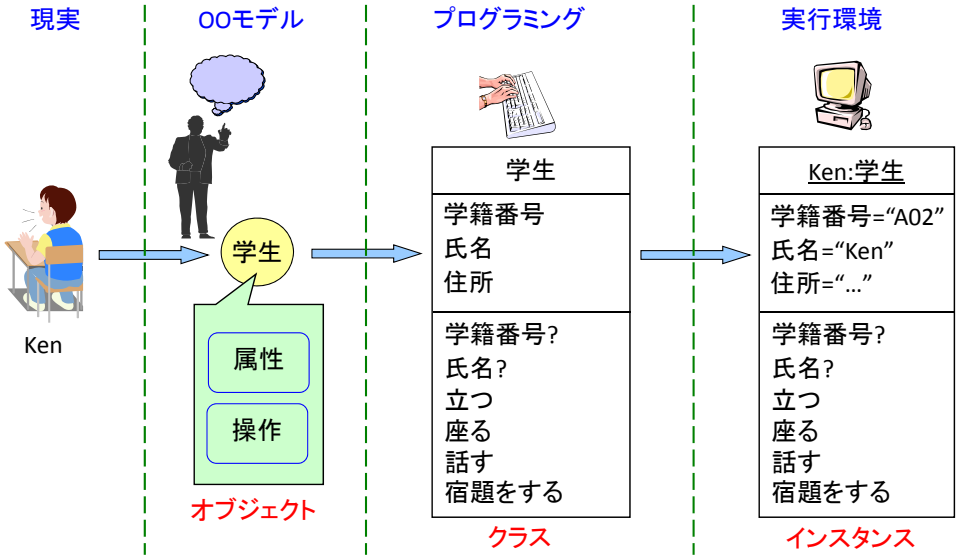


インスタンス

- 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

- 反復型ソフトウェア開発プロセス
  - ✓ イテラティブ（反復型）ソフトウェア開発
    分析、設計、実装、評価の各段階を連続的に繰り返し実施して開発を行う
  - ✓ インクリメンタル（漸増型）ソフトウェア開発
    ソフトウェアを部分に分割し、少しずつ段階的に開発を行う

UML(Unified Modeling Language)

- モデルを表現する統一的な図式表現法
  - ✓ 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

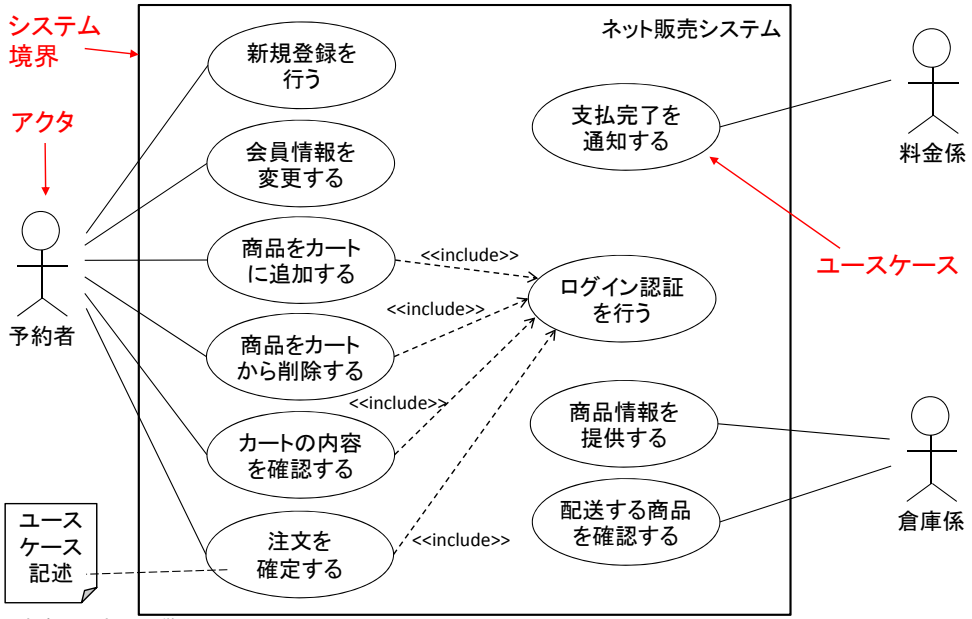
オブジェクト指向分析

- 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. ユースケースの抽出
  2. クラスの同定とクラス図の作成
  3. 振る舞いの記述
  4. モデルの洗練

ユースケース

- ユースケース(use case)[Jacobson]
  - ✓ システムの利用者側(アクタ)からみた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
  - ✓ 利用者の目的に照らして結び付けられた一群のシナリオ
    - 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
  - ✓ システムの機能ごとに作成

ユースケースの例



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。

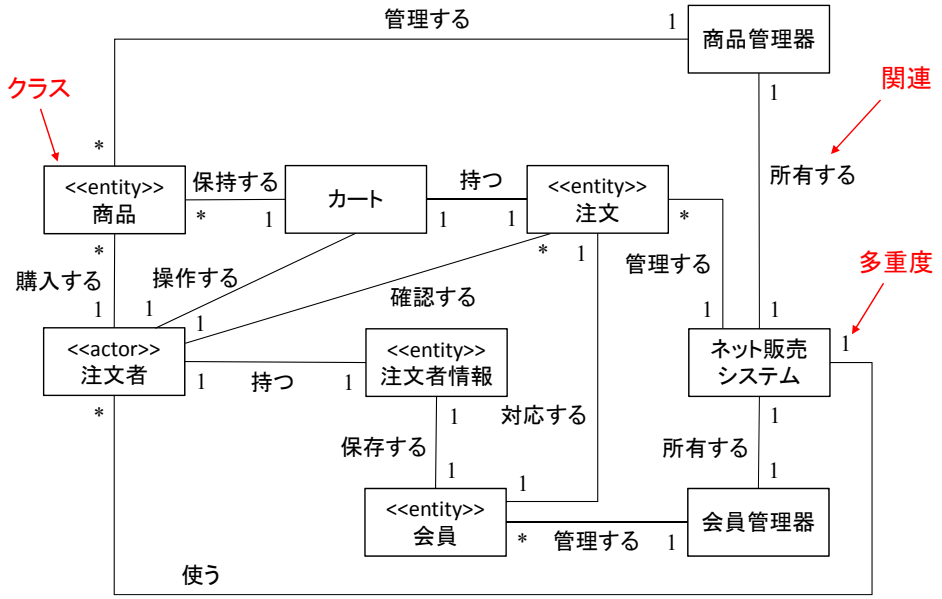
例外処理
5で支払が失敗する。
  a. システムは支払失敗メッセージを表示する。

クラスの同定

- 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

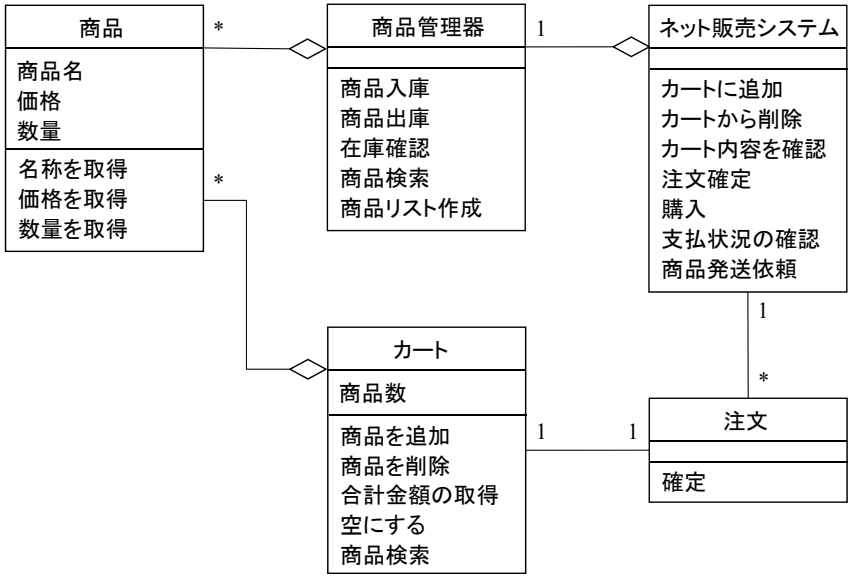
概念モデルの例

（注文に関する一部のみ）



クラス図の例

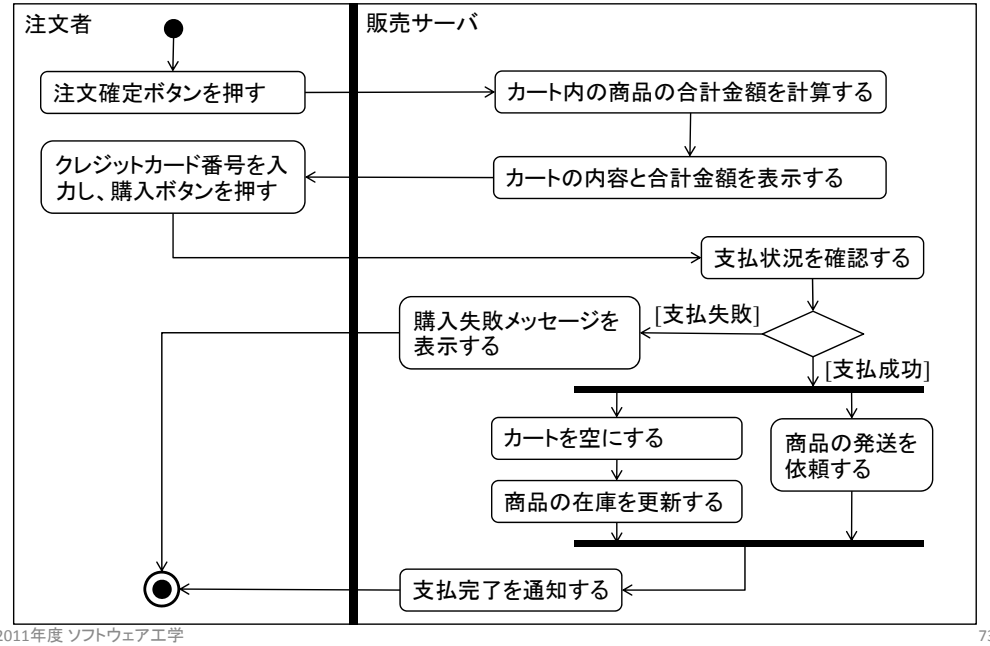
（注文に関する一部のみ）



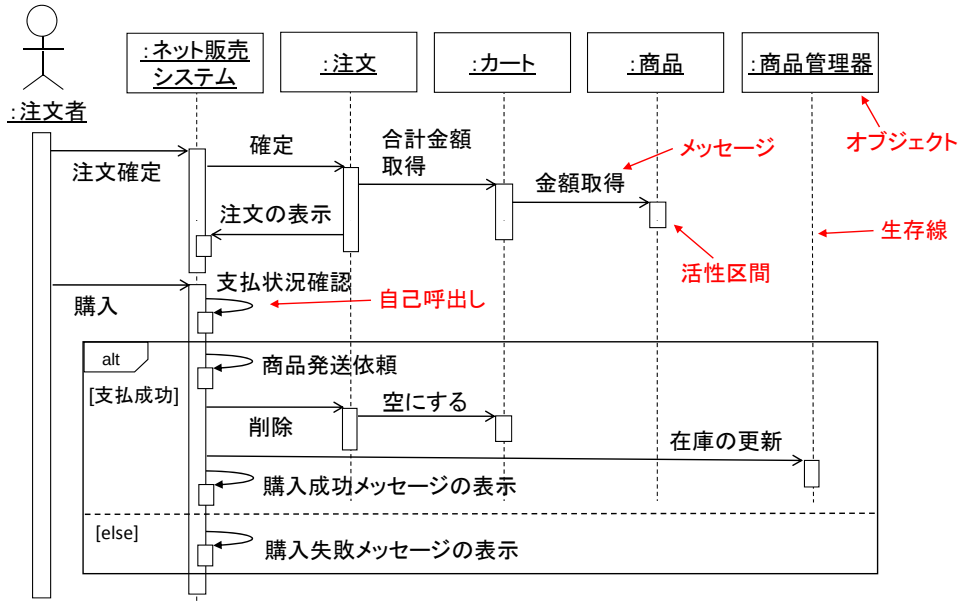
振る舞いの記述

- 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



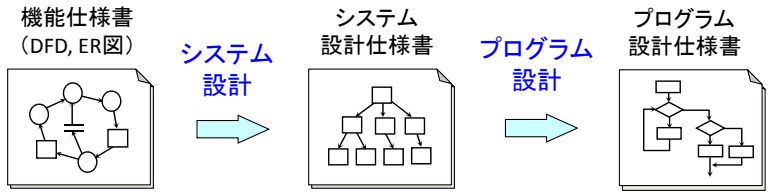
シーケンス図



ソフトウェア設計

設計

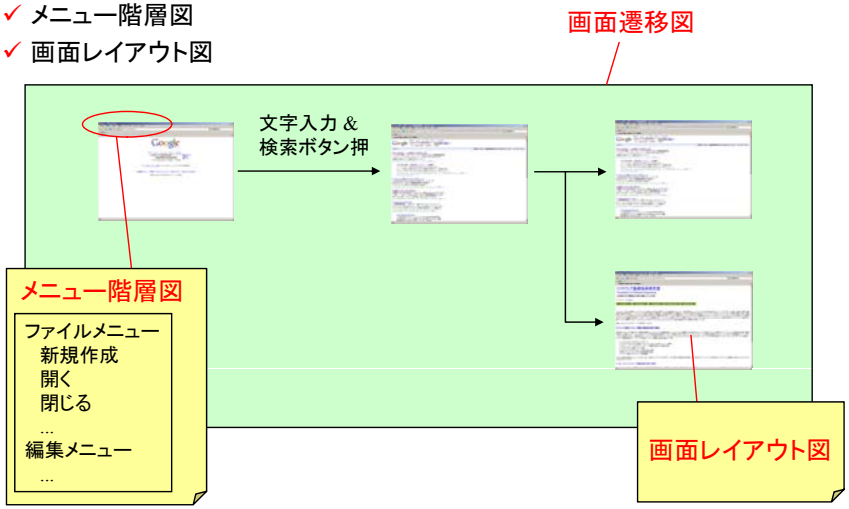
- 機能仕様書が定めるシステムをどのように実現するのかを決定



- システム設計(system design)
  - ✓ システム外部設計：システムの外部特性(環境やUIなど)を記述
  - ✓ システム内部設計：システムをサブシステムやモジュールに細分化し、それらの間のインタフェースを定義
  - ✓ データベース設計：システムに共通のデータ構造の識別と定義
- プログラム設計(program design)
  - ✓ モジュールの設計：個々のモジュールの内部構造を決定

ユーザインタフェース設計

- GUI(graphical user interface)における
  - ✓ 画面遷移状態図
  - ✓ メニュー階層図
  - ✓ 画面レイアウト図



アーキテクチャ

- 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
    - ソフトウェアの特性を決定する
  - ✓ 開発の基本事項への影響大
    - ソフトウェアの分割と構造化の進め方
    - プログラムやツールの再利用の進め方
    - 品質特性の評価の進め方
- アーキテクチャ設計
  - ✓ ソフトウェアアーキテクチャを定める作業
  - ✓ 性能や変更容易性などの非機能面から要求を分析

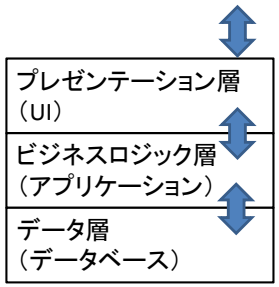
アーキテクチャの設計

- 設計者に強く依存する
  - ✓ 設計者の知識、経験、スキル、直感
- 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
- 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、利用形態など評価基準、価値基準
  - ✓ 評価の視点と追跡
    評価する視点に基づく構造の表現、トレードオフに対する判断の記録

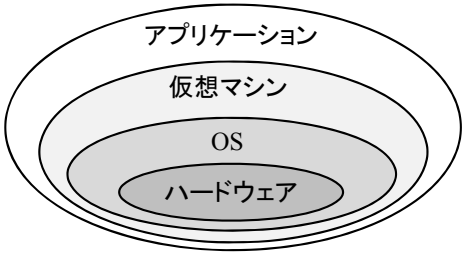
アーキテクチャスタイル

- アーキテクチャスタイル
  - ✓ 抽象化、共通化、カタログ化された代表的なソフトウェアの基本的な構造で、多数のソフトウェアに繰り返し出現する構造
- 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

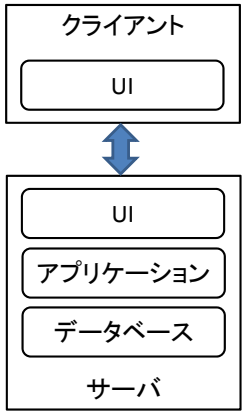


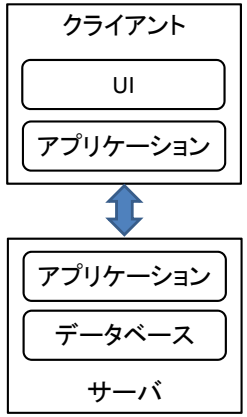
三層モデル

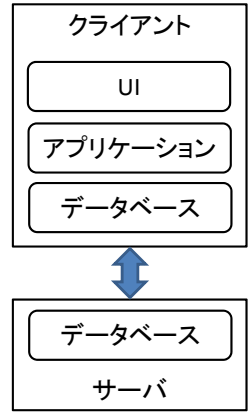


仮想マシンモデル

クライアントサーバモデル



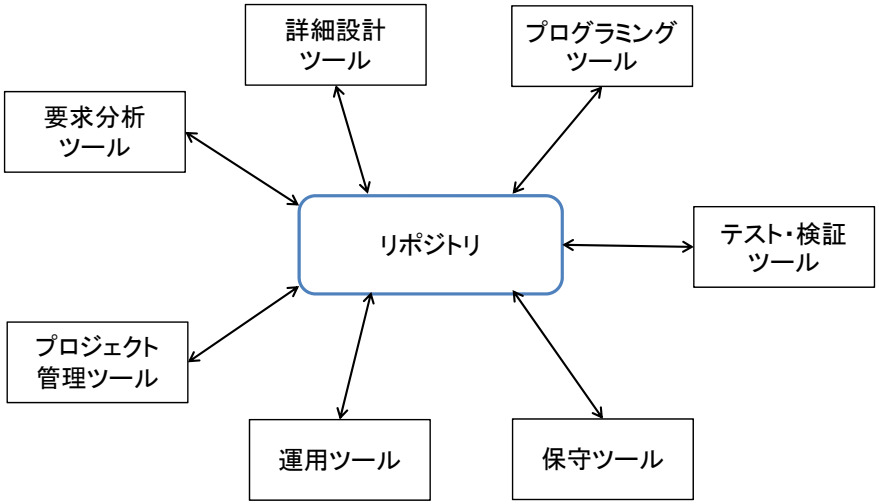




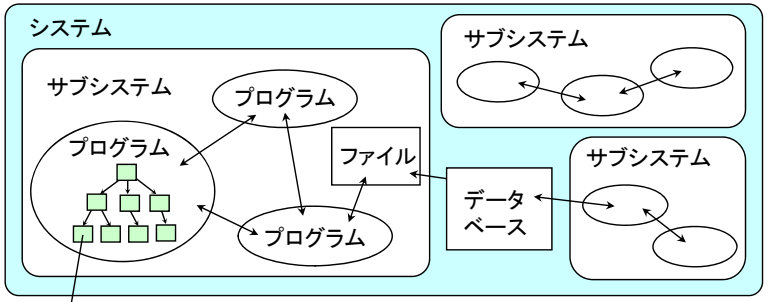
シンクライアント
サーバモデル

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



モジュール(module):
機能単体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

(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
  モジュールの代わりにデータと機能(操作)をカプセル化した
  オブジェクトを基本単位としてプログラム構造を決定
- 契約に基づく設計(DbC: Design by Contract)[Meyer]
  クライアント(client)とサプライヤー(supplier)間の義務、便益、制約を表明(assertion)で記述

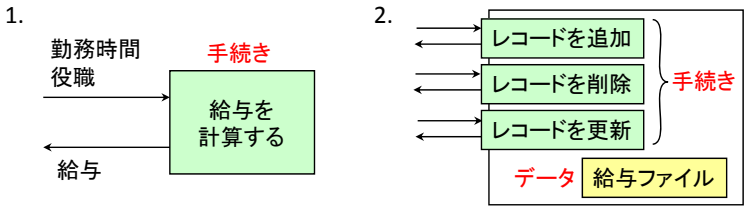
構造化設計

- 構造化設計(structured design)/複合設計(composite 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
  - ✓ 抽象化(abstraction)の概念が有効

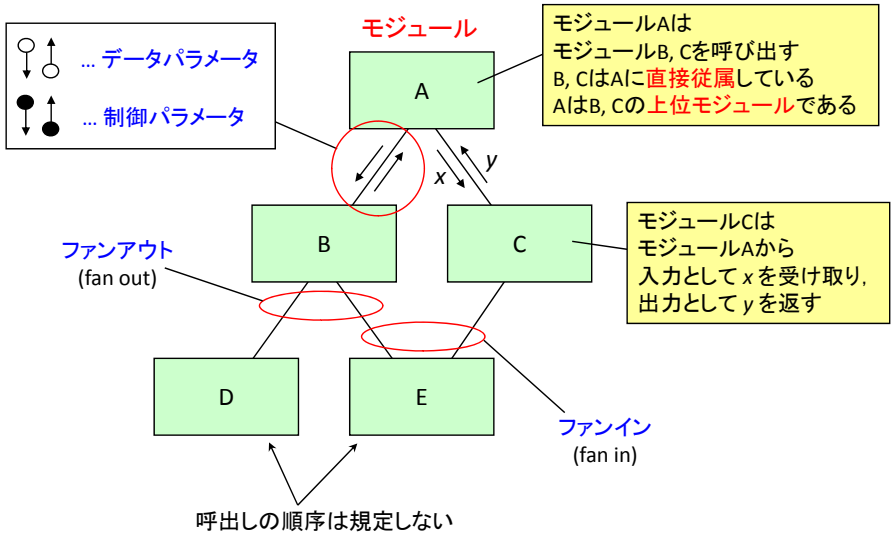
  - ✓ システム設計仕様書(system design specification)
    1. モジュール構成図(module structure diagram)
       システムを実現するモジュールの構成(静的関係)を規定
    2. モジュール機能仕様書(module function specification)
       個々のモジュールの機能を規定
    3. モジュールインタフェース仕様(module interface specification)
       モジュールを外部から呼び出すときのインタフェースを規定

抽象化

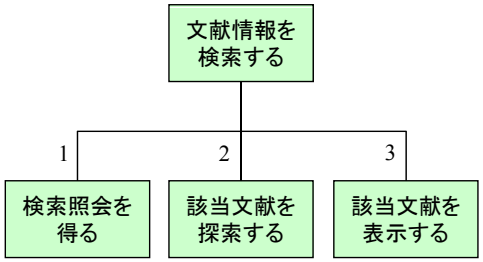
- 抽象化(abstraction) ≅ 情報隠蔽(information hiding)
  1. 手続きの抽象(procedure abstraction)
     手続きの使い方を手続きの実装から分離すること
  2. データの抽象(data abstraction)
     データの使い方をデータの実装から分離すること
     → カプセル化(encapsulation)
  3. 制御の抽象(control abstraction)
     プログラムの制御構造を内部的な詳細から分離すること
     (例)3つの基本制御構造で表現 → 構造化プログラミング



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

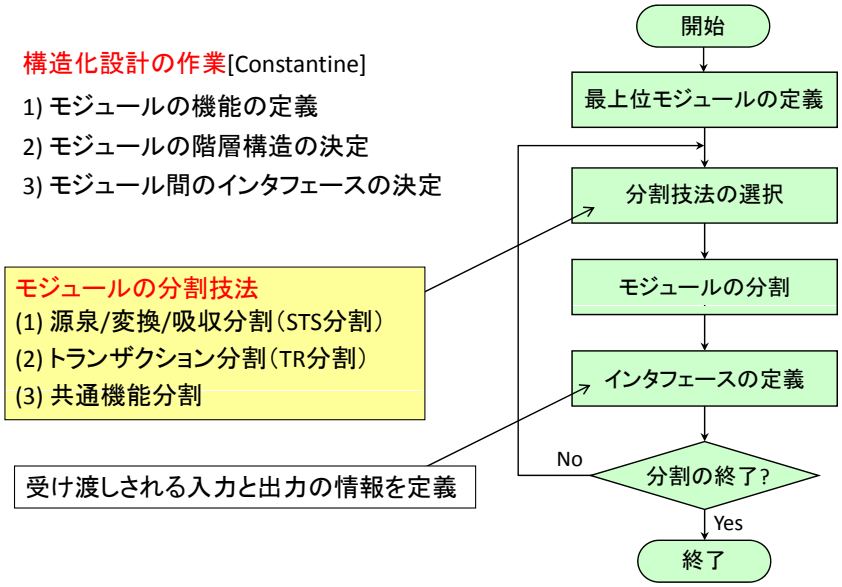
構造化設計の手順

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

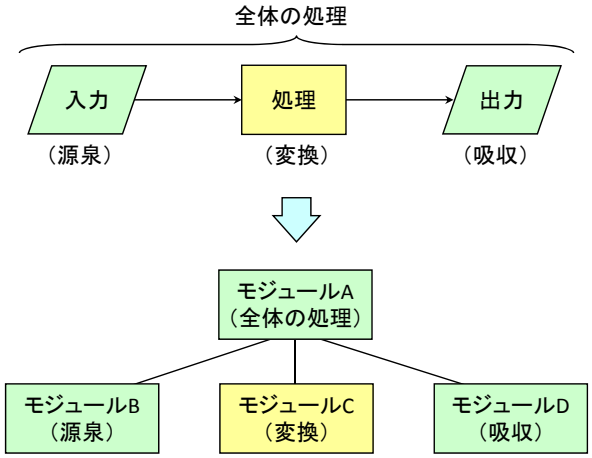
モジュールの分割技法
(1) 源泉/変換/吸収分割（STS分割）
(2) トランザクション分割（TR分割）
(3) 共通機能分割

受け渡しされる入力と出力の情報を定義



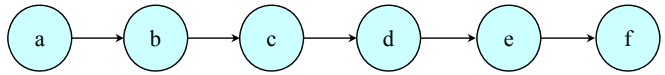
源泉/変換/吸収分割

- 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

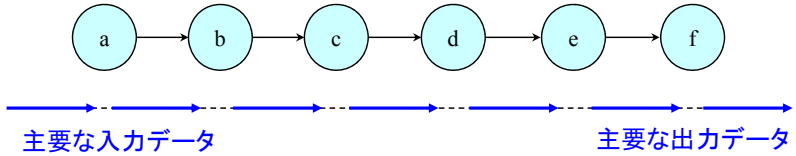


STS分割の手順(1)(2)

1. 問題構造図の記述
   - ✓ 与えられた仕様から、3〜10個の機能で問題を記述する

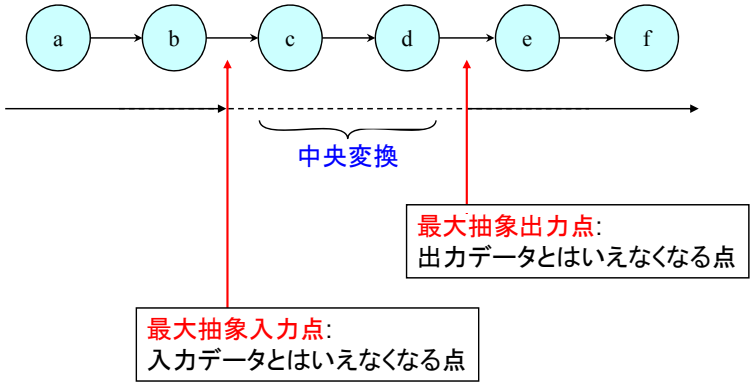


2. 主要データの識別
   - ✓ 問題のなかの主要な入出力データの流れを明らかにする



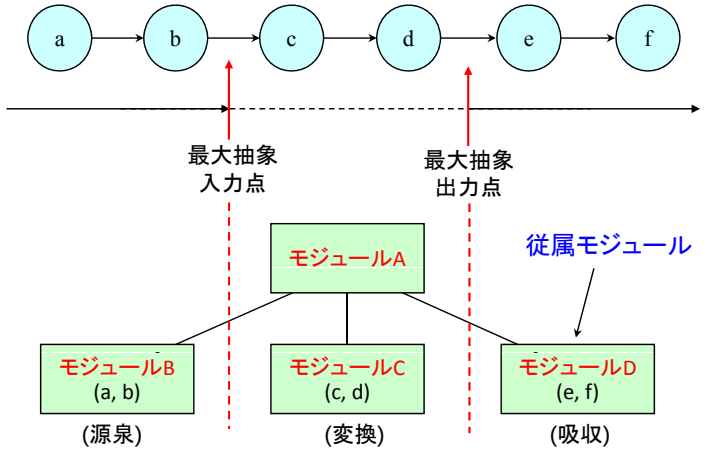
STS分割の手順(3)

3. 最大抽象点の発見
   - ✓ 入力側から入力データを順方向にトレースし最大抽象入力点を見つける
   - ✓ 出力側から出力データを逆方向にトレースし最大抽象出力点を見つける



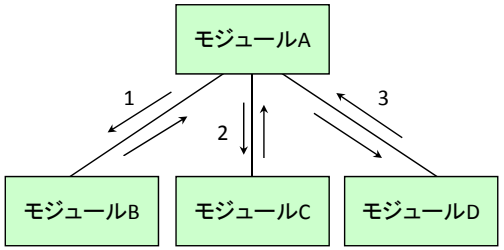
STS分割の手順(4)

4. 直接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(5)(6)

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



	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

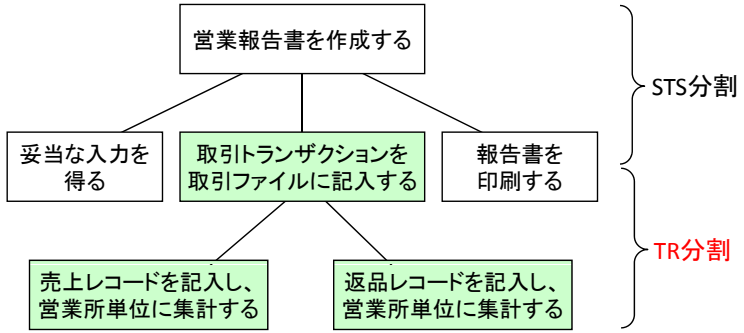
6. 分割の繰り返し
   - ✓ モジュールB, C, Dについて同様の手順を繰り返す

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）
  - ✓ 分岐するトランザクション処理ごとにモジュールを設定する

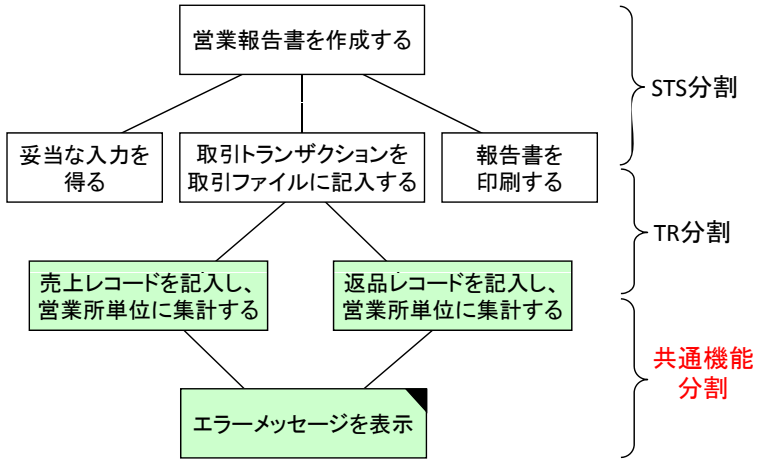
  トランザクション：1つの処理の単位

  （例）データベースの読み込みと更新操作のまとまり

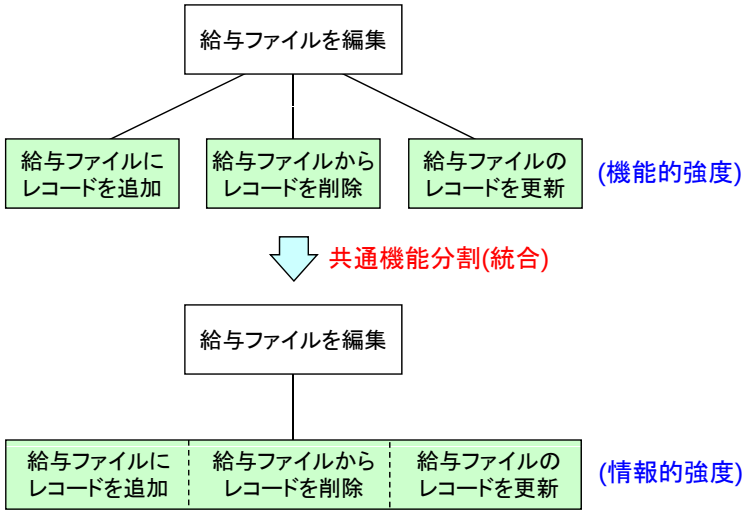


共通機能分割

- 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



共通機能分割

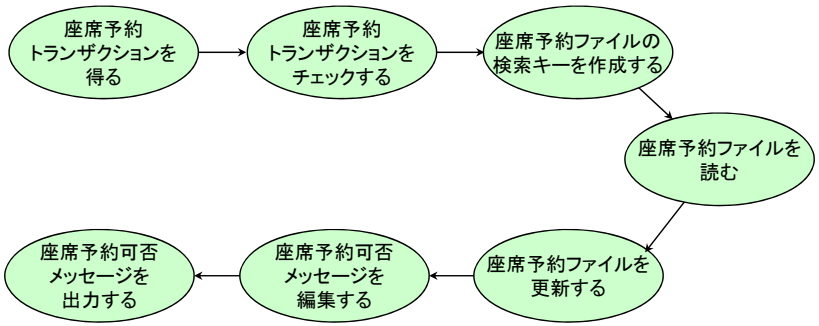


例題：問題構造図の記述

- 座席予約システム

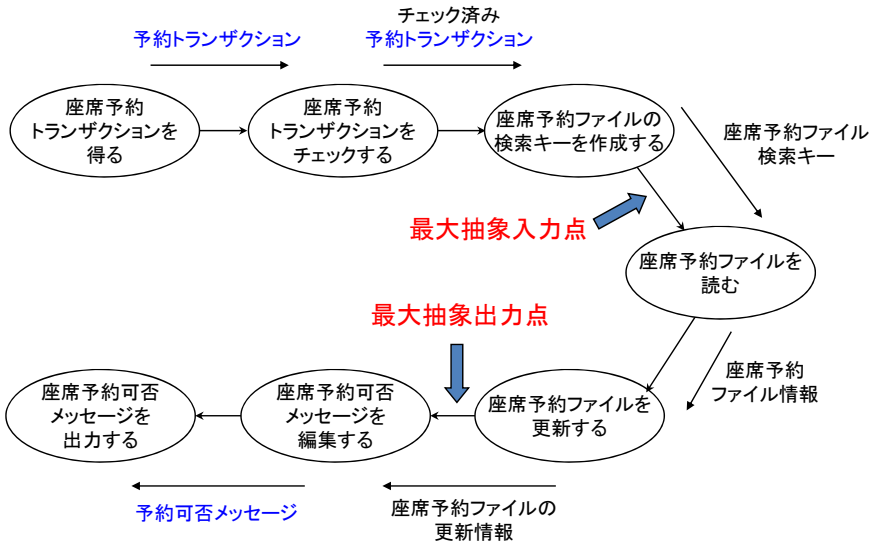
  （概要）

  端末から座席予約トランザクションを受け取り、座席予約ファイルの座席情報を基に予約の可否を調べ、座席予約ファイルの更新を行う。また、端末に予約可否のメッセージを表示する。

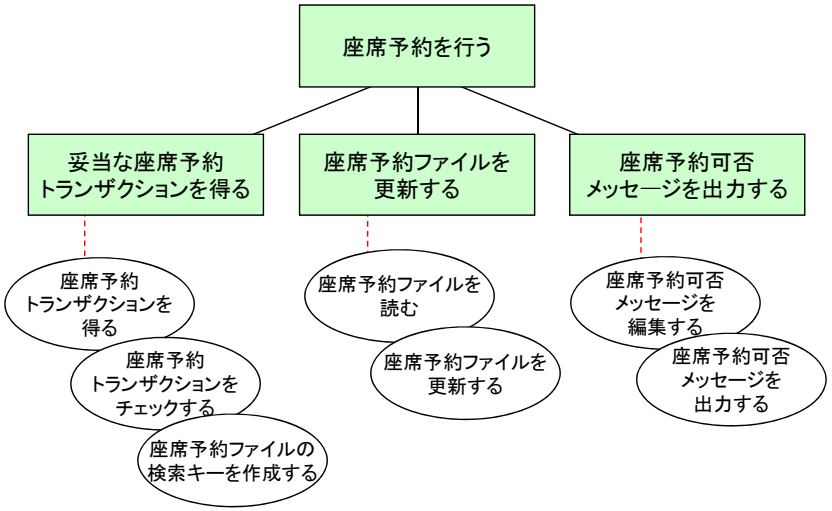


出典：電子開発学園著、「新版ソフトウェアエ学」、SCC出版局

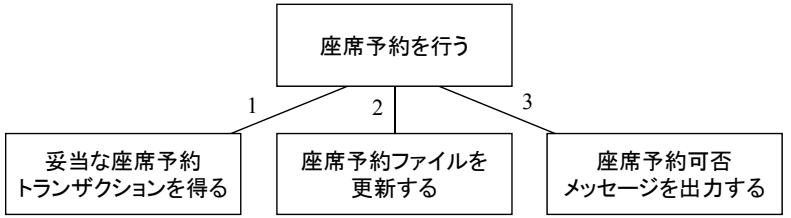
例題：主要データの識別と最大抽象点の発見



例題：従属モジュールの定義

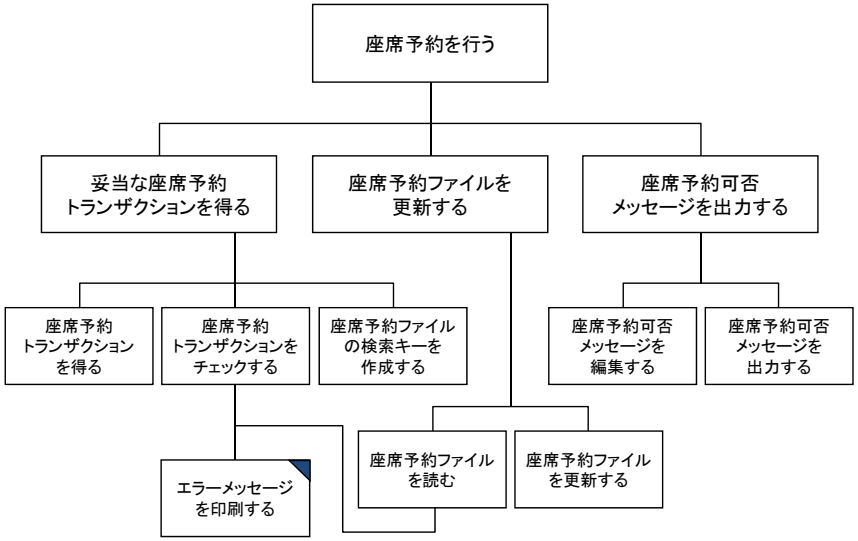


例題：インタフェースの定義

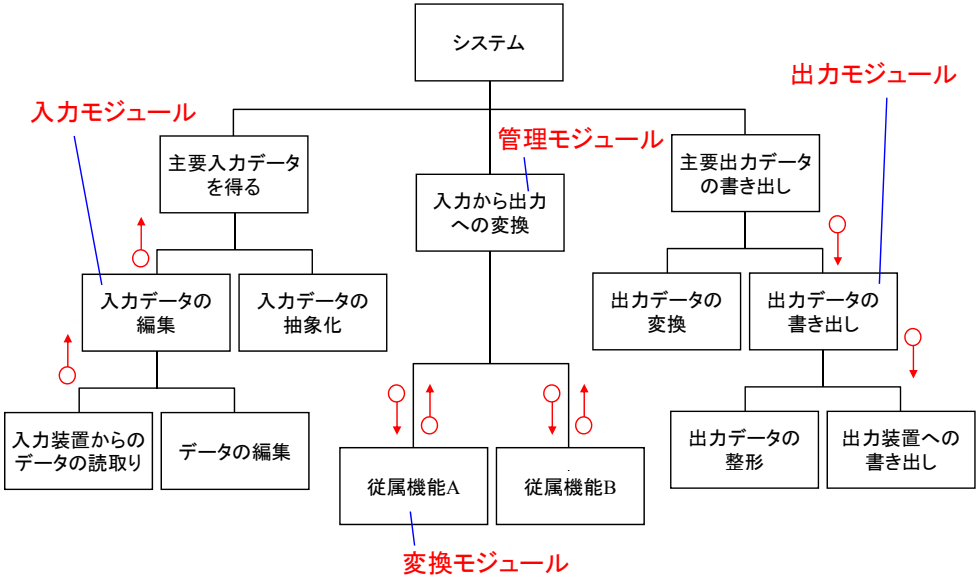


	入力	出力
1	なし	座席予約ファイル検索キー
2	座席予約ファイル検索キー	座席予約ファイルの更新情報
3	座席予約ファイルの更新情報	なし

例題：モジュール構成図



モジュールの一般的構造



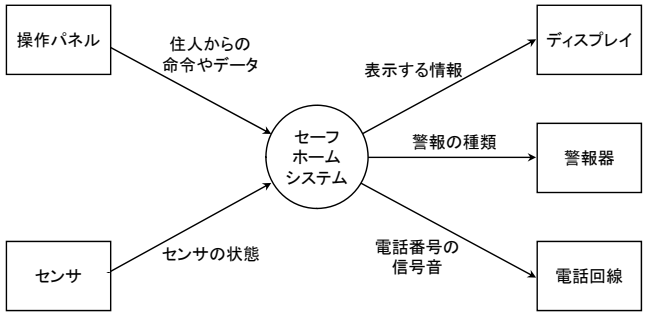
演習：システムの記述（再）

ホームセキュリティシステムの仕様

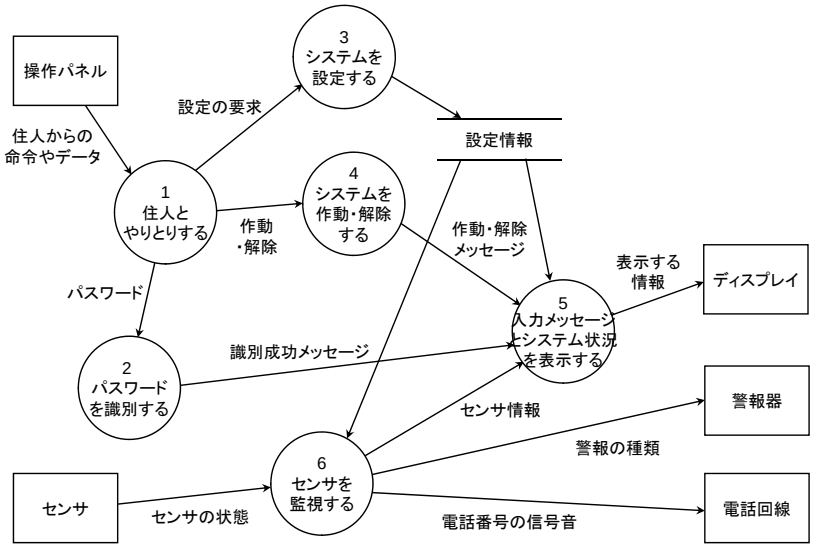
- セーフホームシステムは、設置時には住人がシステム設定をできるようにし、セキュリティシステムが接続されているすべてのセンサを監視する。操作パネルのキーパッドやキーを通して、住人とやりとりする。
- 設置時にシステムを設定するときには、操作パネルで行う。各センサに番号と種類を割りあて、システムの作動・解除を切りかえるマスタパスワードを設定し、センサイベント発生時の連絡先の電話番号を入力する。
- このソフトウェアはセンサイベントを検知した際に、システムに接続されている警報器を起動する。さらに、システム設定時に住人が指定した遅延時間を経過した時点で、監視サービスの電話番号に住所に関する情報を連絡し、検知したイベントについて報告する。電話回線への接続は、接続されるまで20秒間隔で繰り返される。
- セーフホームシステムとのすべてのやりとりは、ユーザインタフェースサブシステムによって管理される。このサブシステムは、キーパッドや特殊キーを通じて与えられた入力を読み取り、ディスプレイに入力メッセージとシステムの状況を表示する。
- (以下 略)

出典：R. Pressman著、「ソフトウェア工学の伝統的手法」、日科技連

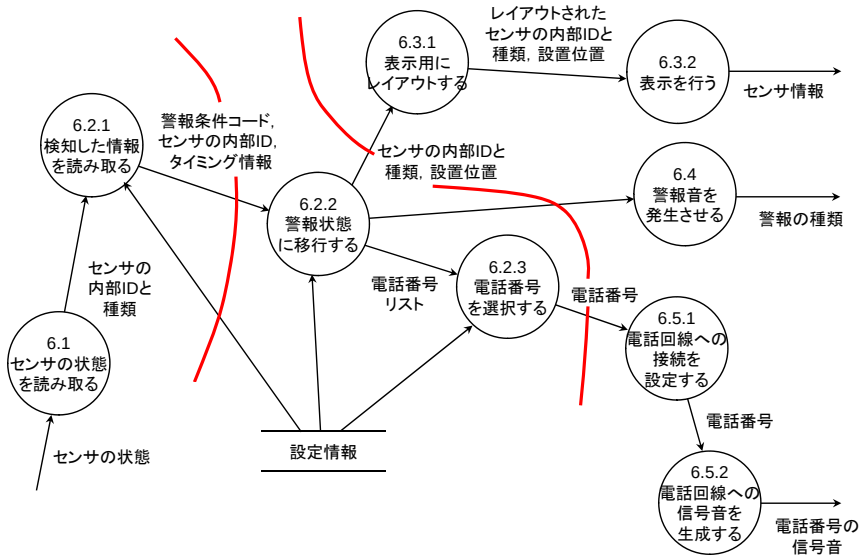
演習：全体文脈図（レベル0）



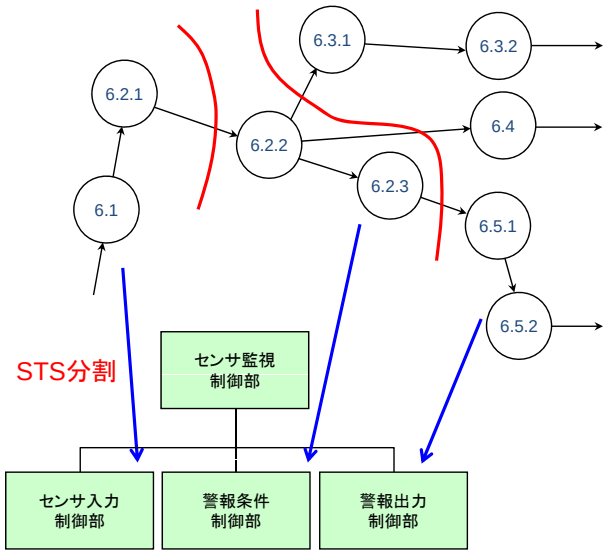
演習：DFD（レベル1）



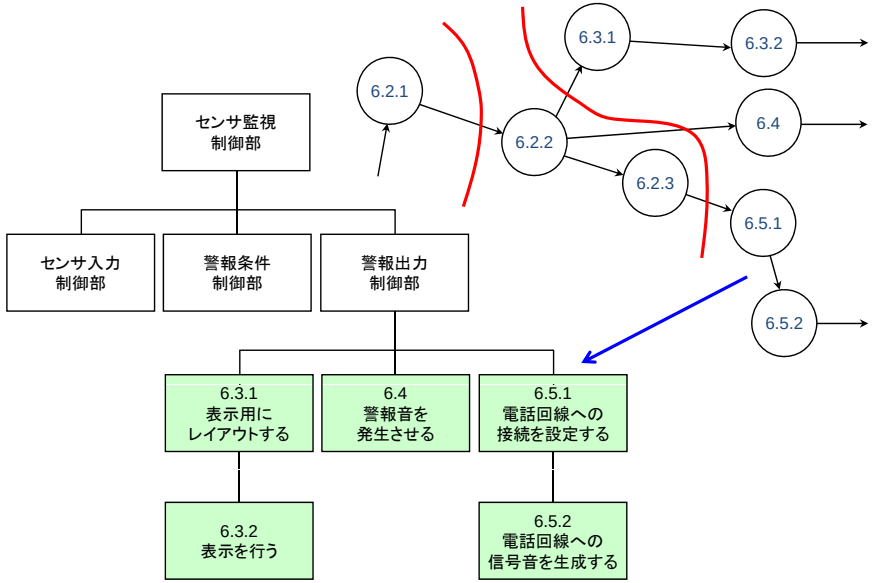
演習：DFDとモジュール分割(1)



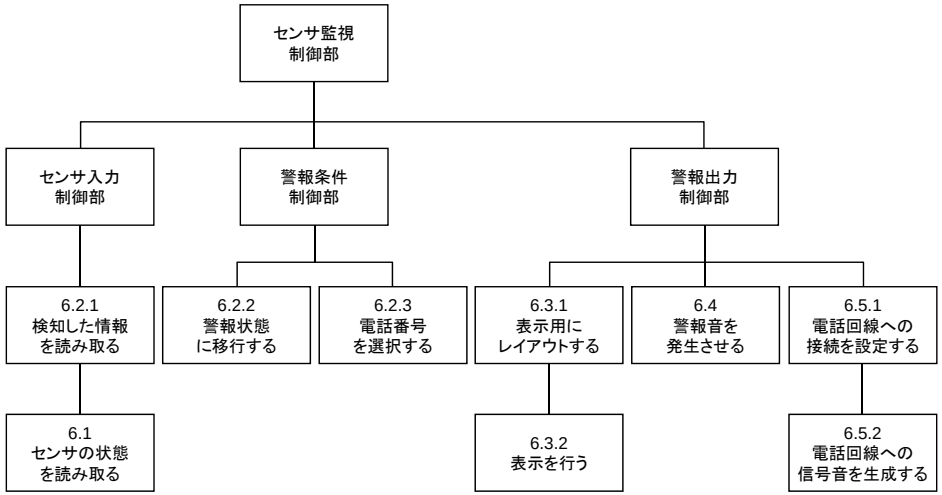
演習：モジュール構成図(1-a)



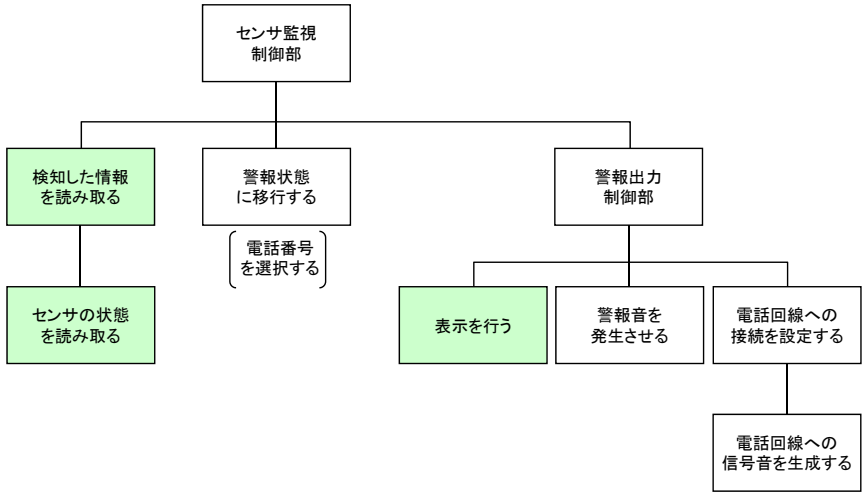
演習：モジュール構成図(1-b)



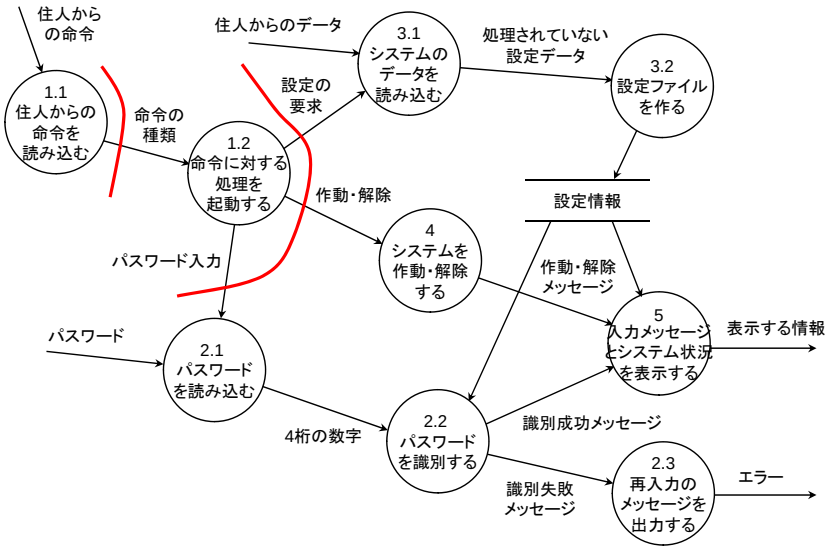
演習：モジュール構成図(1-c)



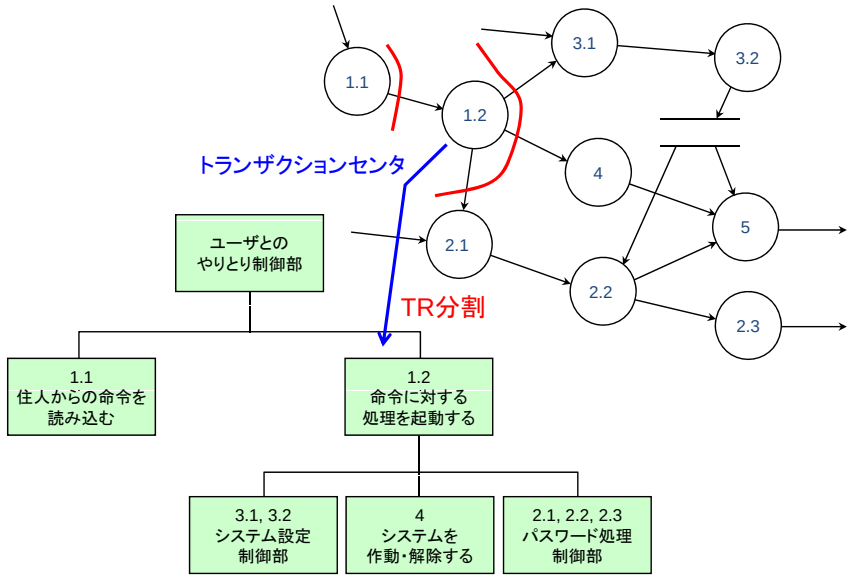
演習：モジュール構成図(1-d)



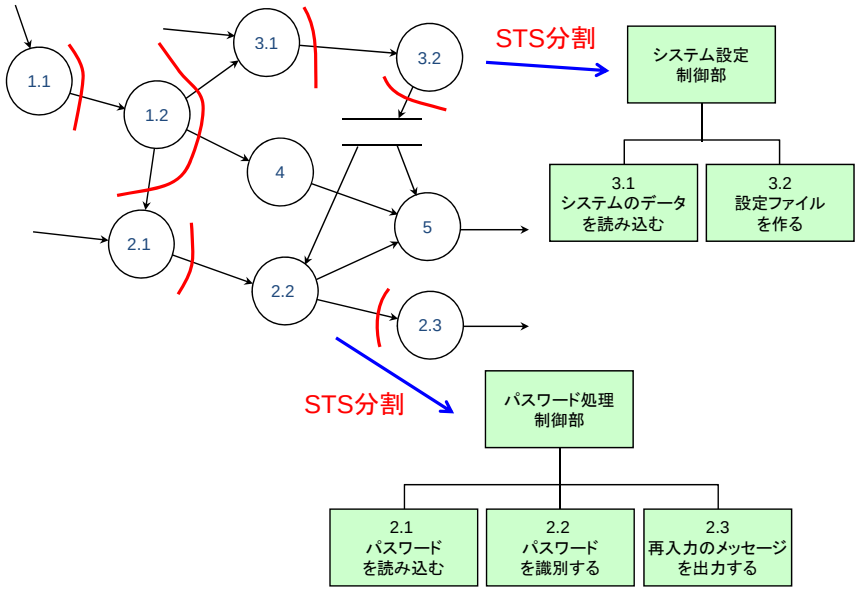
演習：DFDとモジュール分割(2)



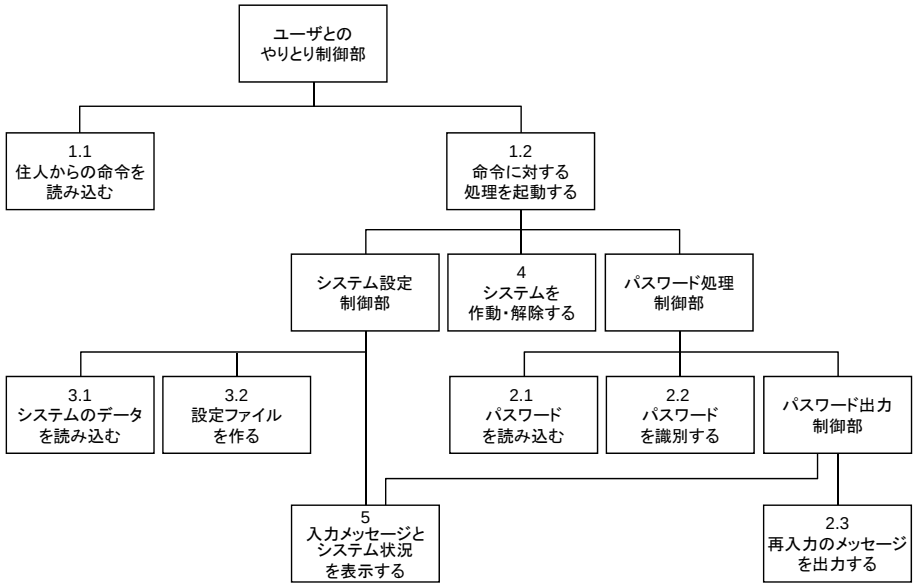
演習：モジュール構成図演習(2-a)



演習：モジュール構成図(2-b)



演習：モジュール構成図(2-c)



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
  - ✓ モジュール強度: 個々のモジュール内部での関連の強さの尺度
    ＝ モジュール凝集度
  - ✓ モジュール結合度: 異なるモジュール間の関連の強さの尺度
- 階層構造化の観点
  - ✓ 制御範囲と影響範囲
  - ✓ ファンイン(fan in)とファンアウト(fan out)

モジュール強度

- モジュール強度(module strength)/モジュール凝集度(module cohesion)
  - ✓ モジュール内に存在する構成要素(文や機能)の関連の強さ

(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、実際には関連する多様な機能を集めたモジュール

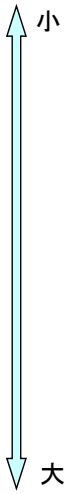
(3) 時間的強度(temporal strength)
実行されるタイミングが近い機能を集めたモジュール

(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、かつ、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

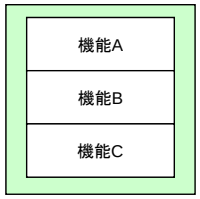
(6) 情報的強度(informational strength)
特定のデータ構造を扱う複数の機能を集めたモジュール

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

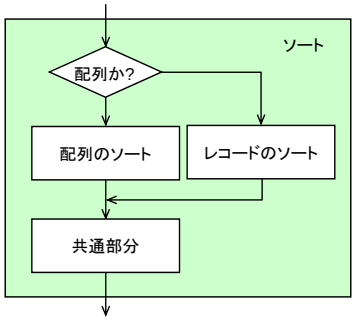


モジュール強度の例(1)～(3)

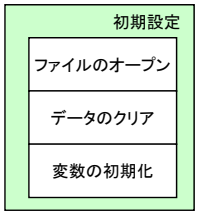
(1) 暗号的強度



(2) 論理的強度

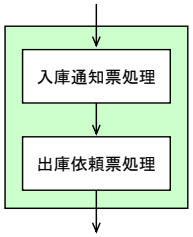


(3) 時間的強度

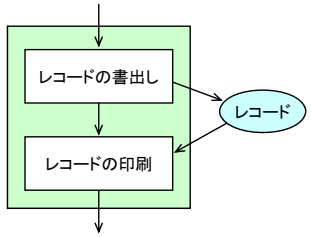


モジュール強度の例(4)～(7)

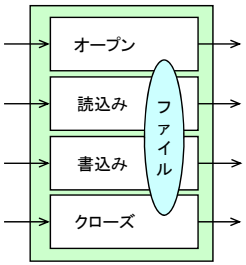
(4) 手順的強度



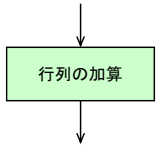
(5) 連絡的強度



(6) 情報的強度



(7) 機能的強度



モジュール結合度

- モジュール結合度(module coupling)
  - ✓ モジュール間に存在する構成要素（文や機能）の関連の強さ

(1) 内容結合(content coupling)
モジュール同士がデータや手続きを共有する
一方のモジュールが他方のモジュールの内容をインタフェースを介さず直接参照する

(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

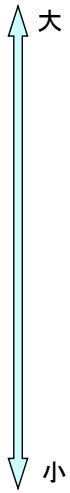
(3) 外部結合(external coupling)
モジュール同士が外部宣言されたデータを共有する

(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
データ共有はなし

(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（未使用データ含む）を引数として受け渡す

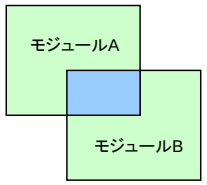
(6) データ結合(data coupling)
必要なデータだけを引数として受け渡す

(7) 無結合

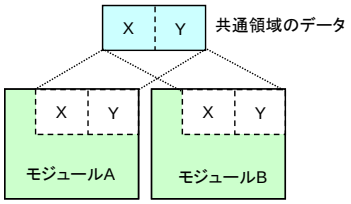


モジュール結合の例(1)〜(3)

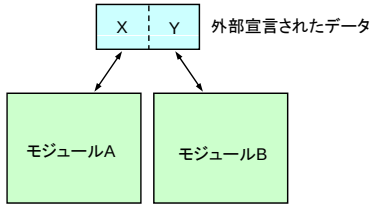
(1) 内容結合



(2) 共通結合

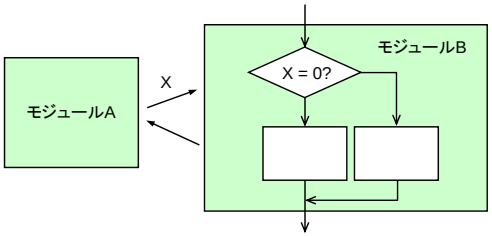


(3) 外部結合

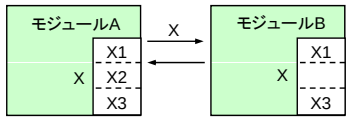


モジュール結合の例(4)〜(6)

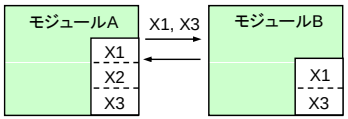
(4) 制御結合



(5) スタンプ結合

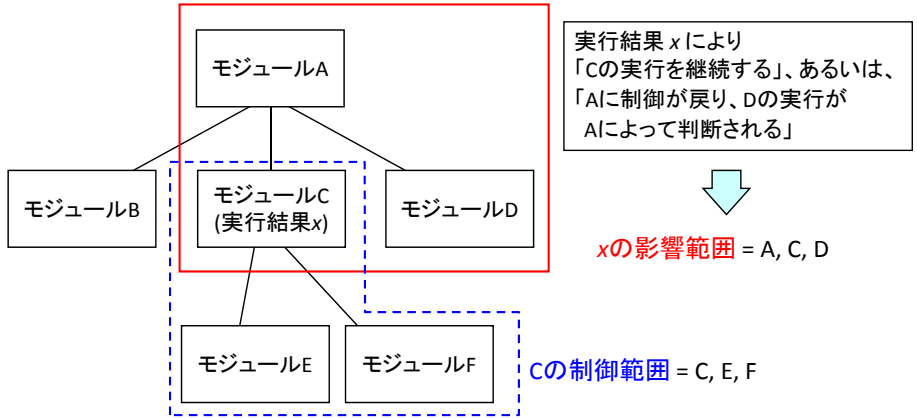


(6) データ結合



制御範囲と影響範囲

- 制御範囲: 対象モジュールとそのモジュールに従属するすべてのモジュール
- 影響範囲: 対象モジュールの実行結果により実行されるすべてのモジュール
  影響範囲 ⊆ 制御範囲 となるように修正



データ構造に基づく設計

- 入力データ構造と出力データ構造に着目し、プログラムの論理構造を決定
  - ✓ プログラム ＝ 入力データから出力データへの変換
  - ✓ データは特定の処理手順とは独立
    - → データ中心設計
  - ⇔ データの流れ（機能）に着目

1. ジャクソン構造分割（Jackson法）
2. ワーニエ法（Warnier法）

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、繰返し、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
   構造が不一致のとき、中間のデータ構造を導入
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

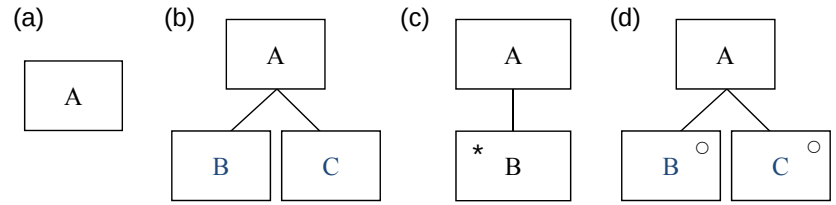
データ構造図の構成要素

(a) 基本
これ以上分割できない構成要素。1つのデータ項目が相当

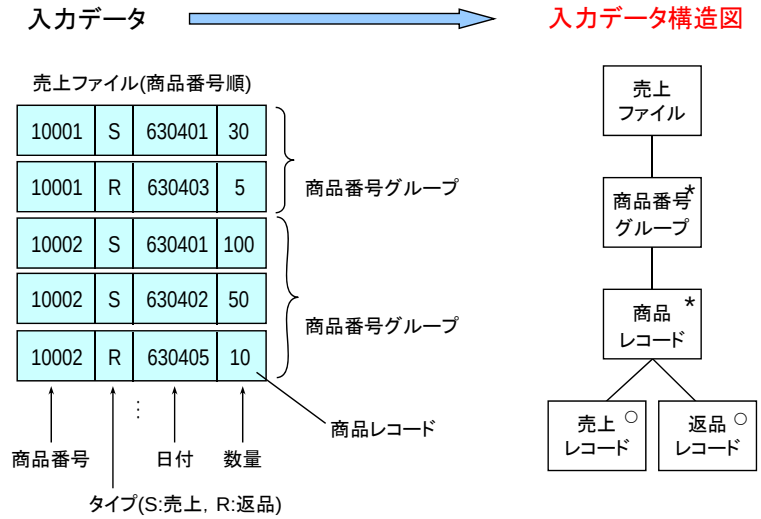
(b) 連続
異なる複数の基本要素からなる構成要素で、それぞれの構成要素は1度だけ順番に現れる。複数データ項目を持つレコードに相当

(c) 繰返し
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

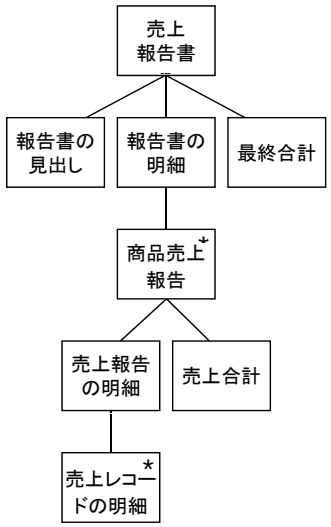
出力データ → 出力データ構造図

売上報告書(印刷結果)

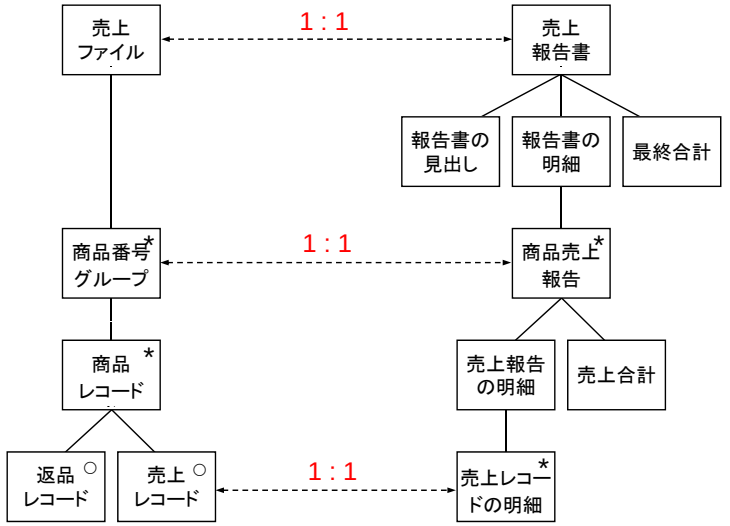
商品番号	売上日	数量	合計
10001	630401	30	
			*30
10002	630401	100	
	630402	50	
			*150
⋮			
			**5550

最終合計

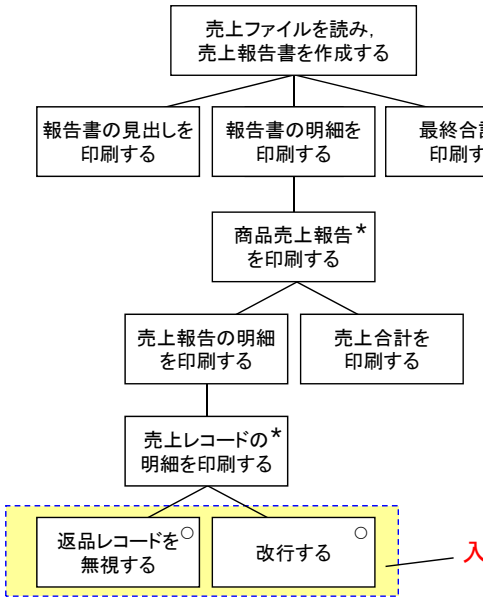
各商品の売上合計



構成要素の間の対応



プログラム構造



出力データ構造図に対応
- データ構造図の基本
  → プログラムの1命令
- データ構造図の連続
  → 逐次処理
- データ構造図の繰返し
  → 繰返し
- データ構造図の選択
  → 選択

入力データ構造図からの補正

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

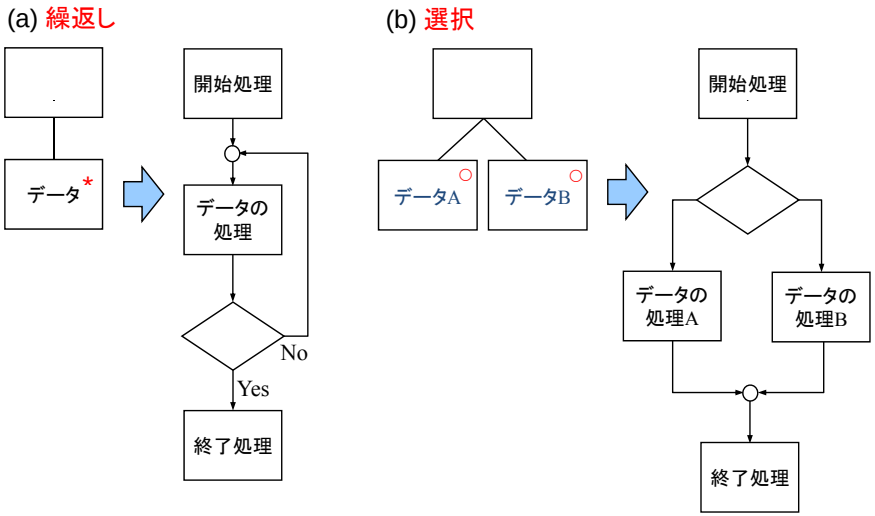
入力データ構造と出力データ構造を分析し、4つの構成要素(基本、連続、繰返し、選択)でデータ構造図を作成

(2) プログラム論理構造の決定

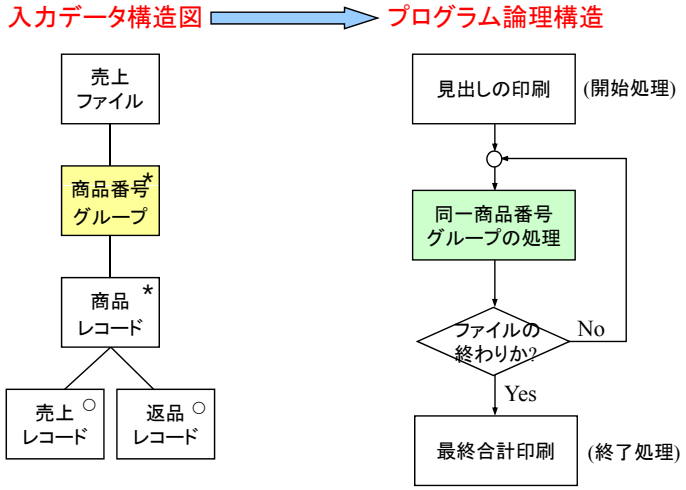
入力データ構造を基にプログラムの論理構造を決定

- 繰返しのデータ構造 → 繰返しの論理構造
- 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

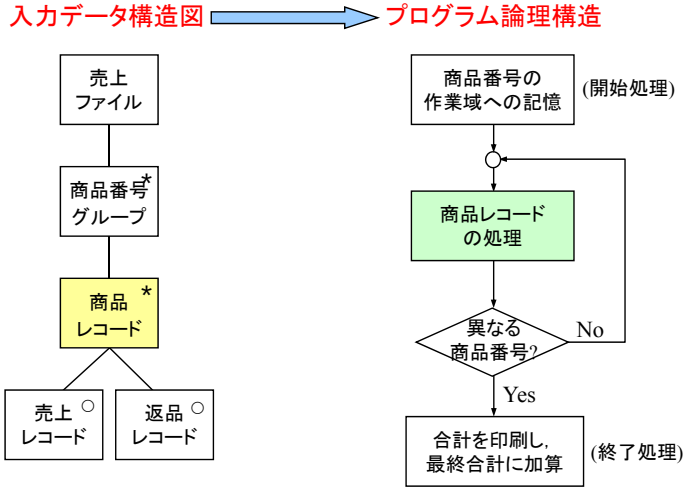
データ構造とプログラム論理構造



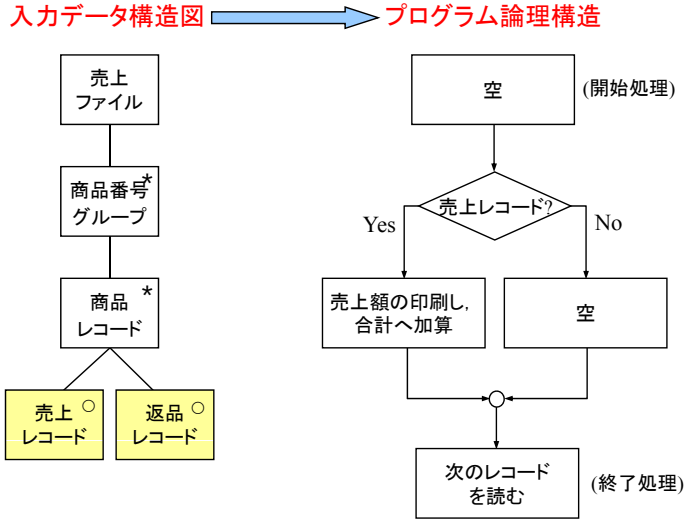
プログラム論理構造の例(1)



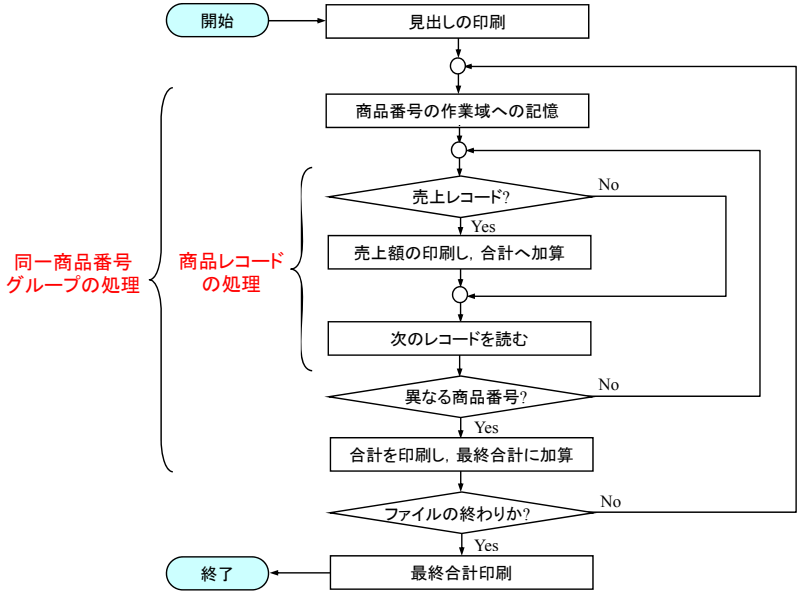
プログラム論理構造の例(2)



プログラム論理構造の例(3)

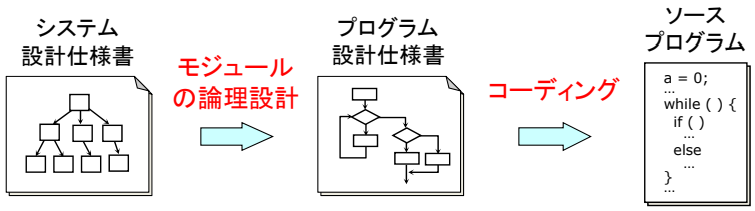


プログラム論理構造の例（全体）



プログラミング

モジュールの設計



- システム設計仕様書(system design specification)
  - ✓ モジュール構成図
  - ✓ モジュール機能仕様書
  - ✓ モジュールインタフェース仕様

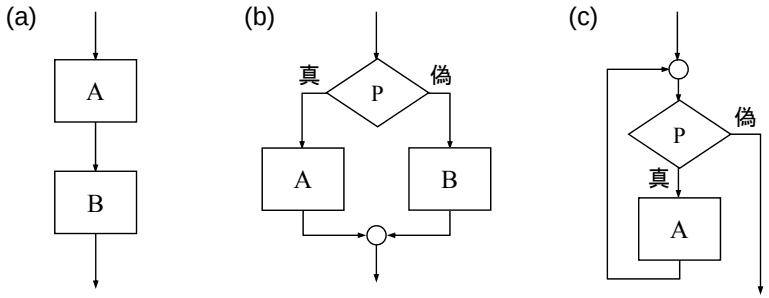
  } モジュールの外部特性の定義
- プログラム設計仕様書(program design specification)
  - ✓ モジュールの論理設計：個々のモジュールの内部構造を決定
- ソースプログラム(source program)
  - ✓ コーディング：具体的なプログラミング言語による記述

モジュールの論理設計

- モジュールの論理設計 ≅ プログラミング
  - ✓ 従来
    メモリの使用領域と処理時間の最小化
    → 生産性向上を阻害
  - ✓ 現在
    理解しやすいプログラムの作成
    → 構造化プログラミング(structured programming)[Dijkstra]

構造化プログラミング

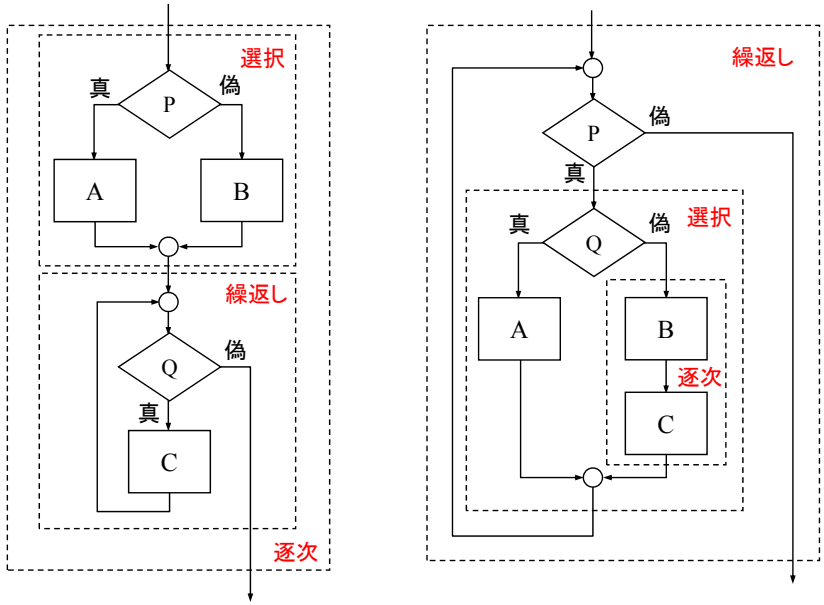
- 手続き型プログラムの論理を3つの基本制御構造の組み合わせで表現
  - (a) 逐次(sequence)
    命令（命令群）A, B を順番に実行
  - (b) 選択(selection, if-then-else)
    条件 P の真偽により命令 A, B の実行を選択
  - (c) 繰返し(iteration, do-while)
    条件 P が真である間、命令 A を繰返し実行



構造化定理

- 構造化定理[Böhm, Jacopini]
  - ✓ すべての適正プログラムの論理は3つの基本制御構造（逐次、選択、繰返し）で記述可能
- 適正プログラム(proper program)
  - ✓ 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

適正プログラムの例



構造化コーディング

- プログラムの論理を3つの基本制御構造をそのまま命令に変換
  - ✓ 逐次、選択、繰返し
- goto文の使用を制限
  - ✓ 例外処理
  - ✓ モジュールからの脱出
  - ✓ ループからの脱出
  - ✓ 重複コードの排除
- コーディング規約
  - ✓ 字下げ(indentation): 制御範囲の明確化
  - ✓ データ名や関数の名前の付け方

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

プログラム = アルゴリズム + データ構造
programs = algorithms + data structures [Wirth, 1976]

アルゴリズム = 論理 + 制御
algorithm = logic + control [Kowalski, 1979]

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
  - ✓ 構造化プログラミング

  Fortran, COBOL, Algol, BASIC, PL/I, Pascal, C, Ada
- 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
- 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#
- 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)

プログラムの記述例

- 手続き型プログラミング

Cのプログラム(繰返し)

```
int fact = 1, i;
for (i = 1; i <= n, i++)
    fact = i * fact;
```

Cのプログラム(再帰)

```
int fact(int n) {
    if (n == 0) return(1);
        return n * fact(n-1);
}
```

- 関数型プログラミング

関数 *fact ()*の定義

fact(x) = if x=0 then 1
else x × fact(x-1)

Lispプログラム

```
(DEFUN fact(N)
   (COND ((ZEROP N) 1)
   (T (TIMES N (FACT(SUB1 N)))))
```

- 論理型プログラミング

階乗 *x!* の定義

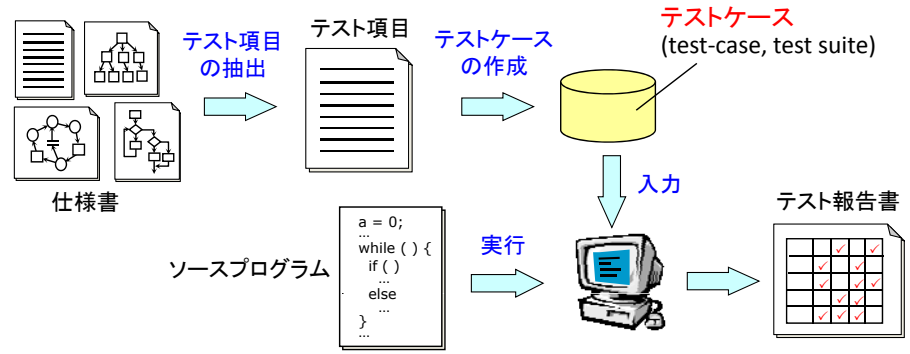
fact(0,1)
fact(x, y) ← *sub(x, 1, w)*
∧ *fact(w, z)* ∧ *times(x, z, y)*
∃ [*fact(3, y)*]

Prologのプログラム

```
fact(0, 1).
fact(X, Y) :- sub(X, 1, W),
                fact(W, Z), times(X, Z, Y).
?- fact(3, Y).
```


ソフトウェアテスト

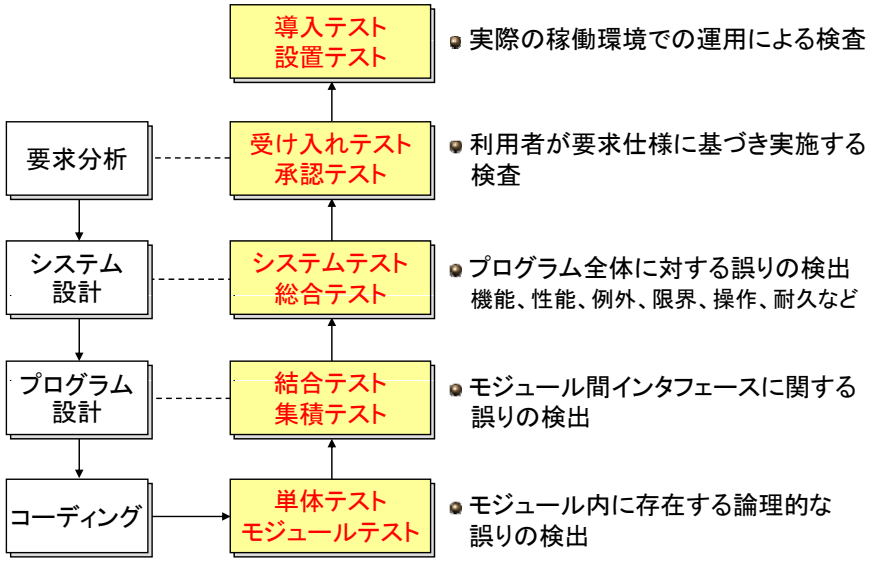
ソフトウェアテスト



- プログラムが仕様書に定義した振る舞いを満たすかどうかの検査
  - ✓ 故障(failure): 要求に反した振る舞い、間違った動作、動作不良の現象
  - ✓ 障害(fault)/欠陥(defect): 故障の原因、故障を引き起こすソフトウェア内部の誤り
  - ✓ エラー(error): ソフトウェアが障害を持つことになった開発者の誤り

「テストは誤りが存在することは示せるが、誤りが存在しないことは示せない」

ソフトウェアのテスト工程



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)に着目
  - プログラムの詳細(内部構造や内部論理)を無視

  同値分割法、限界値分析
  (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
  (c) コードレビュー(code review)
  - コードウォークスルー(walk-through): 非形式的、正当性に関するコメント
  - インスペクション(inspection): 形式的、リストとコードとの照合

同値分割法

- 同値分割法
  プログラムの入力領域を同値クラスに分類することでテストケースを作成
  (1) 同値クラスの識別
  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

(例)

入力条件	有効同値クラス	無効同値クラス
文字数	4 ~ 8	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

  (2) クラスに基づくテストケースの作成
  (2a) すべての有効同値クラスに属するテストケースを作成
  (例) amku5ge
  (2b) 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テストケースを作成
  (例) xy9, jdsi5enjcd, abcdef, 123456

限界値分析法

- 限界値分析法
  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

(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	1～64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

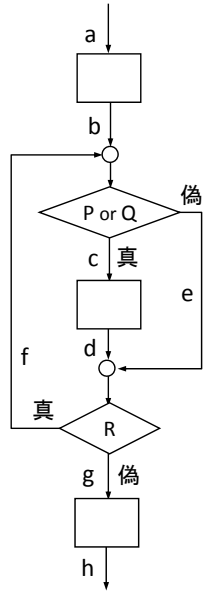
テスト網羅技法(1)

- 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

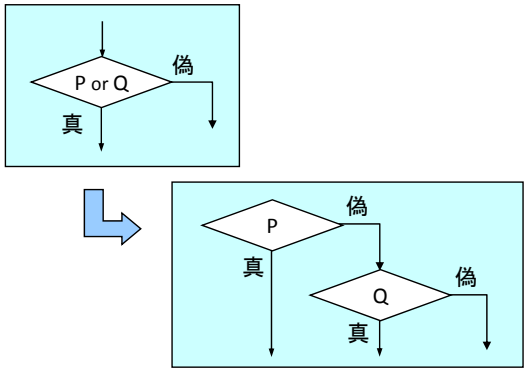
- 枝網羅、分岐網羅(edge coverage, C1 coverage)
  - ✓ プログラム中のすべての枝を1回以上実行
    （例）P or Q が真、R が真（パス：abcdf）
    P or Q が真、R が偽（パス：abcdgh）
    P or Q が偽、R が真（パス：abef）
    P or Q が偽、R が偽（パス：abegh）

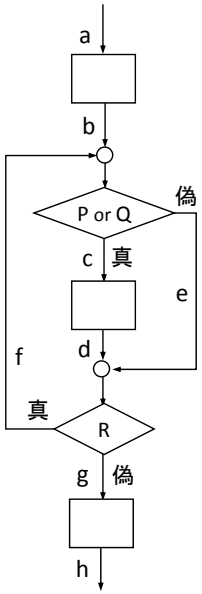
  網羅(coverage) ＝ 通過した枝 / 全枝



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

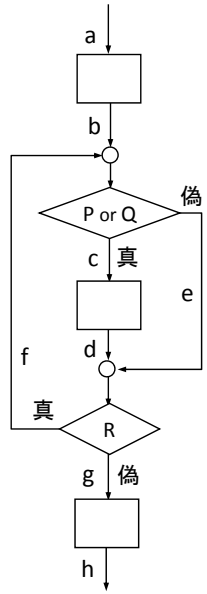




テスト網羅技法(3)

- 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 + …

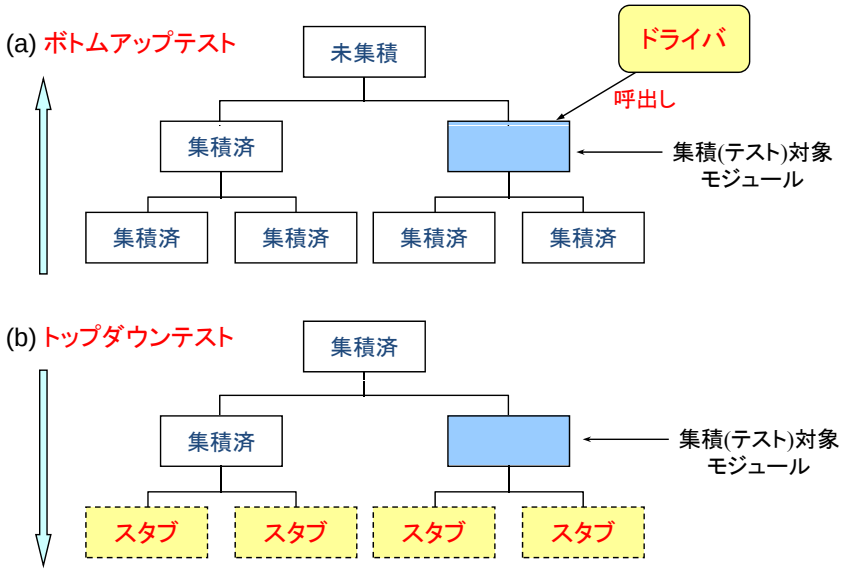
  網羅(coverage) ＝ 実行したパス / 全パス



結合テスト

- 結合テスト(integration test)
  モジュールインタフェース(パラメータや共通データ)に関するエラーを検出
  (a) ボトムアップテスト(bottom-up test)
    - モジュール階層図の最下位モジュールからテスト開始
    - テストドライバ(仮のメインプログラム)が必要
    - 初期段階から並行にテスト可能
  (b) トップダウンテスト(top-down test)
    - モジュール階層図の最上位モジュールからテスト開始
    - プログラムスタブ(身代わりモジュール)が必要
    - インタフェースエラーを早期に発見可能
  (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



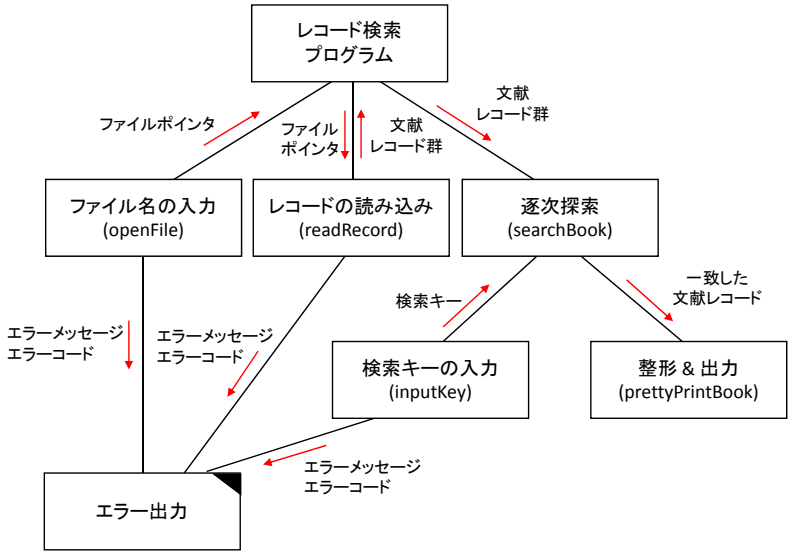
演習：トップダウンプログラミング

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。

演習：モジュール構成図



演習：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(book, num);
}
```

作成 & テスト対象

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

スタブ(stub)

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

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


演習：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

演習：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);
}
```


演習：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

演習：ステップ(3)　残り手続きの作成とテスト

```
void inputKey(char key[ ] )
{
   char key[システムが許す最大サイズ];

   /* 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

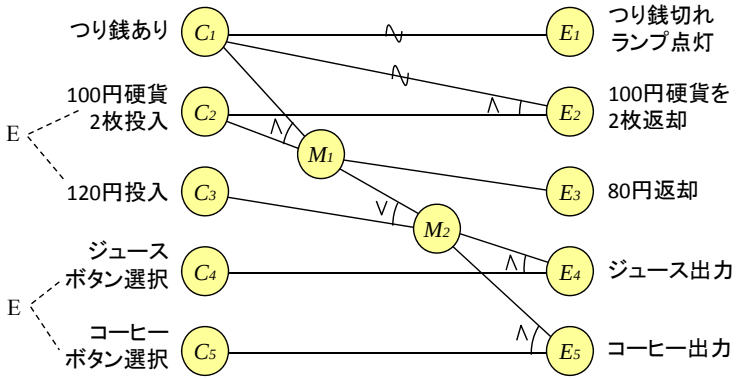
- システムテスト(system test)
  顧客の要求をシステムが満たしているかどうかを検査
  テスト計画書(test plan)を作成
  (a) 機能テスト(function test)
  - システムの機能が要求仕様通りに稼働するかどうかを検査
    原因結果グラフ法

  (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

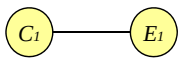
- 原因（入力）と結果（出力）の因果関係に着目し、テストケースを作成

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

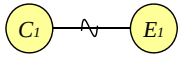


原因結果グラフ

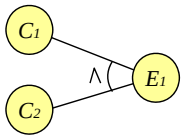
(a) 肯定(C_1であればE_1)



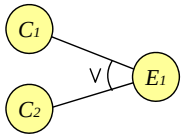
(b) 否定(C_1でなければE_1)



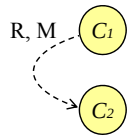
(c) 論理積(C_1かつC_2であればE_1)



(d) 論理和(C_1またはC_2であればE_1)

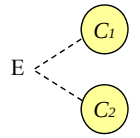


(e) 必要(R), マスク(M)



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

	原因/結果	テスト項目					
		1	2	3	4	5	6
入力	(C_1) つり銭あり	×	○	○	×	○	×
	(C_2) 100円硬貨2枚投入	○	×	○	×	○	○
	(C_3) 120円投入	×	×	×	○	×	×
	(C_4) ジュースボタン選択	○	○	○	○	×	×
	(C_5) コーヒーボタン選択	×	×	×	×	○	○
出力	(E_1) つり銭切れランプ点灯	✓			✓		✓
	(E_2) 100円硬貨2枚返却	✓					✓
	(E_3) 80円返却			✓		✓	
	(E_4) ジュース出力			✓	✓		
	(E_5) コーヒー出力					✓	

(例)「1」: つり銭なしの状態で、200円を投入して、ジュース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

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、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
  - ✓ 形式仕様記述言語 Z (Z notation)

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

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
$gcd(x, y) = gcd(x, y \bmod x) = gcd(x \bmod y, y)$
$gcd(x, y) = gcd(y, x)$
$gcd(x, 0) = gcd(0, x) = x$
ただし, $a \bmod b$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):	
init: → Stack	pop(push(z, s)) = s
push: integer × Stack → Stack	pop(init) = **stack-error**
pop: Stack → Stack	top(push(z, s)) = z
top: Stack → integer	top(init) = **stack-error**
empty: Stack → bool	empty(init) = **true** empty(push(z, s)) = **false**

形式仕様記述言語 Z

- 集合論に基づくデータの型付け

（例）基本型 NAME と DATE に対する {NAME, DATE} の仕様

BirthdayBook
known: ℙ *NAME*
birthday: *NAME* ⇸ *DATE*
---
known = **dom** *birthday*

known: 名前の集合
birthday: 誕生日の集合
A ⇸ *B*: *A*から*B*への部分関数

スキーマ(schema)

InitBirthdayBook
BirthdayBook
---
known = Ø

AddBirthdayBook
Δ*BirthdayBook*
name?: NAME
date?: DATE
---
name? ∉ *known*
birthday' = *birthday* ∪ { *name*? ↦ *date*? }

暗黙条件: *known*' = **dom** *birthday*'
a ↦ *b*: 対(*a*, *b*)

FindBirthday
Ξ*BirthdayBook*
name?: NAME
date!: DATE
---
name? ∈ *known*
date! = *birthday* (*name*?)

ソフトウェア検証

- ソフトウェア検証(verification & validation)

ソフトウェアが要求される品質を満たし、信頼できることを確認

(a) 仕様検証
- モデル検査(model checking)
  ソフトウェアが時相論理式(temporal logic formula)のモデルであるか 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起こること
- レビュー(review)

(b) プログラム検証
- 動的検証：テスト、プロファイル分析、網羅度計測、表明検査
- 静的検証：正当性検証（Hoare論理）、型検査、 記号実行、制御フロー解析、データフロー解析

プログラム検証

- テスト

エラーの存在を示すことはできるが、エラーが存在しないことは示せない

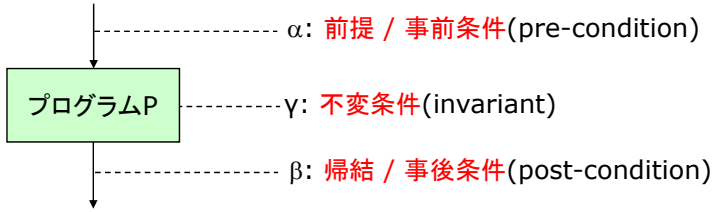
- 正当性(correctness)証明

エラーが存在しないことを数学的に証明

正当性：(1) プログラムが必ず停止する（停止性）
(2) 得られた答えは必ず正しい（部分正当性：partial correctness）

- ✓ 公理的意味論(axiomatic semantics)に基づく方法
  - 帰納表明法(inductive assertion method)[Floyd]
  - Hoare論理(Hoare logic)[Hoare]
- ✓ 定理証明器を利用

公理的意味論

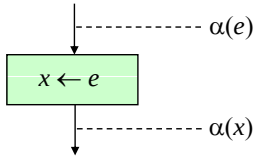


- P の実行前に α が成立するならば、P の実行後に β が成立する
  （例）10個の要素を持つ配列が入力されると、ソートされた配列
  （配列添字の大きい方が、その値が大きい）が出力される
- P の実行中は必ず γ が成立する
  （例）配列の要素の数は変わらない

契約による設計(DbC: Design by Contract)で採用
契約：事前条件を満たした状態でプログラムを実行した場合、
事後条件を満たす状態を実現することを約束

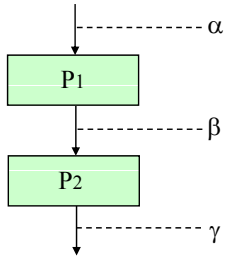
検証条件(1)

(1) 代入文



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

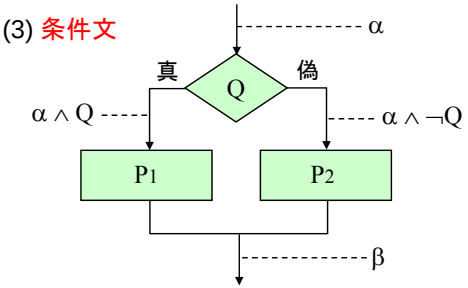
(2) 複合文



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

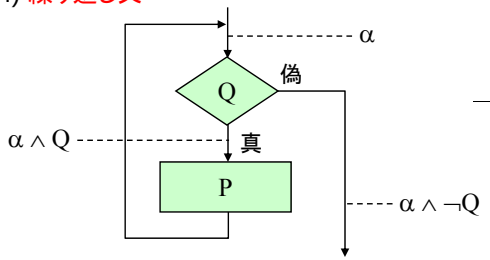
検証条件(2)

(3) 条件文



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

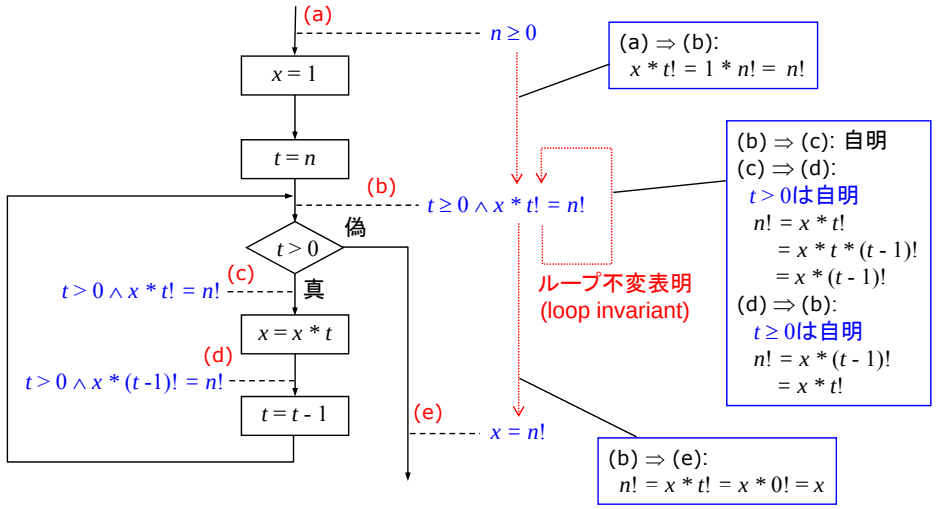
(4) 繰り返し文



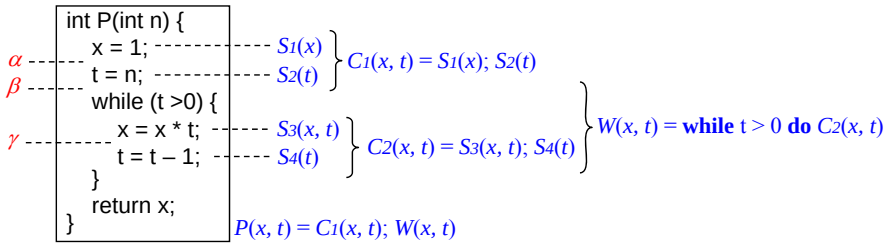
$$\frac{\{\alpha \wedge Q\}\ P\ \{\alpha\}}{\{\alpha\}\ \textbf{while}\ Q\ \textbf{do}\ P\ \{\alpha \wedge \neg Q\}}$$

帰納表明法

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! になっている

検証条件を使って検証すると…

Hoare論理

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

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

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

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

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

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

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

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

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

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

型検査

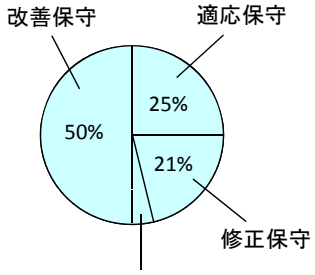
- 型(type): ものの集まり、ものを分類するための仕組み
  プログラミング言語処理系では、変数や式の取りうる値を規定するもの
  （例）int x; 変数 x の値は-2147483648から2147483647の整数である
  boolean p; 変数 p の値は真(true)か偽(false)
- 型検査(type checking)
  （例）1+2: 型安全である （int型とint型の加算）
  1+true: 型安全でない （int型とboolean型の加算）
  ✓ 型に関して不適切な演算や操作が行われることがないプログラム
  ＝ 型安全なプログラム
  ✓ 型安全で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
  → 信頼性の低下
  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
- 静的な型付けに基づく言語（強く型付けされた言語）
  すべての変数や式の型がコンパイル時に決定可能
  （例）C, C++, Java, ML

ソフトウェア保守と再利用

ソフトウェア保守

- ソフトウェア保守(software maintenance)
  現行のソフトウェアを維持・管理する作業
  ソフトウェアは変更を受け入れ可能、部品の摩耗なし
  ⇔ ハードウェア保守

  - ✓ 修正保守(corrective maintenance)
    エラーに対するソフトウェアの修正
  - ✓ 適応保守(adaptive maintenance)
    システム、ハードウェア、組織、規模、社会などの
    進化や変更に応じて発生する変更
  - ✓ 改善保守(perfective maintenance)
    機能追加や使いやすさの向上のための変更
    維持・管理のしやすさを向上させるための変更
  - ✓ 予防保守(preventive maintenance)
    故障を未然に防ぐための訂正、潜在的な障害の修正



Lientz and Swanson (1981)

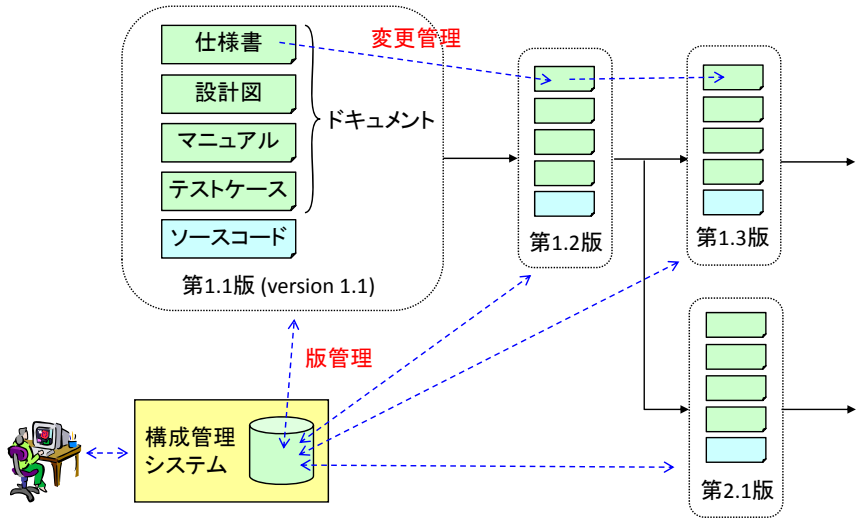
ソフトウェアの理解

- ソフトウェアを保守するためには対象のソフトウェアについて知る(理解する)必要がある
  - ✓ 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

保守技法

- 構成管理(configuration management)
  - ✓ バージョン(version)とリリース(release)を管理
- 影響分析(impact analysis)
  - ✓ 保守による変更あるいは追加による影響が及ぶ範囲(モジュール)を把握
  - ✓ 変更に関連するリスクの評価
- 回帰テスト(regression test)
  - ✓ 変更されなかった部分が元の通り正常に動作するかどうかを確認
  - ✓ 変更前に適用されたテストデータを実行
- プログラムスライシング(program slicing)
  - ✓ プログラム中の特定の文 s の値に影響を与える、あるいは、
    文 s の値が影響を与える文を抽出する手法
  - ✓ スライス(slice): 抽出された文の集合
    - 文 s の値に影響を与える文集合である逆方向スライス(backward slice)と
      文 s の値が影響を与える文集合である順方向スライス(forward slice)
    - 静的解析に基づく静的スライス(static slice)と
      実行時の情報に基づく動的スライス(dynamic slice)

構成管理



プログラムスライシング

```
a = 0;
b = a + 1;
```
データ依存関係(data dependence)
変数aの値の定義が変数aの値の参照に到達

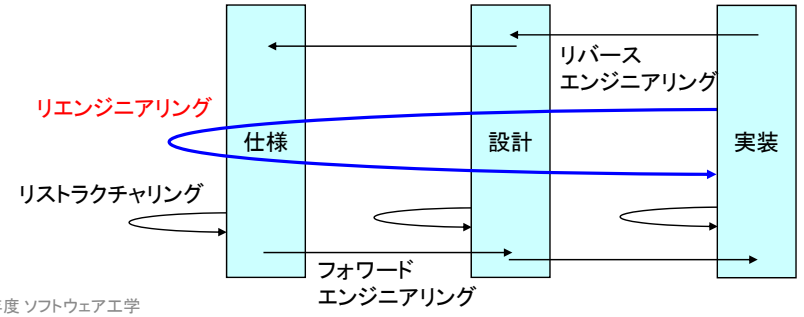
```
if (a < 0) {
  b = 10;
}
```
制御依存関係(control dependence)
文"b = 10"の実行は，if文の判定の結果に依存

```
 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++;
 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++;
 9:   }
10:   print(sum);
11:
12: }
```
文10のsumに関する静的逆方向スライス

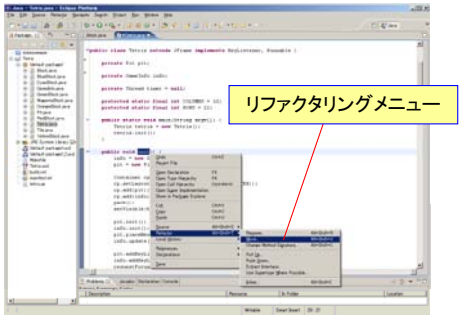
ソフトウェアリエンジニアリング

- ソフトウェアリエンジニアリング(software reengineering)
  - ✓ 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(recovery)
- フォワードエンジニアリング(forward engineering)
  - ✓ 従来の開発と同方向
- リストラクチャリング・再構成(restructuring)
  - ✓ 内部表現の単純化、構造化



リファクタリング

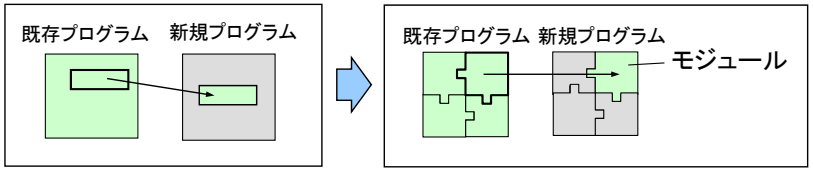
- ソフトウェアリファクタリング(software refactoring)
  - ✓ リストラクチャリングの一種
  - ✓ 既存ソフトウェアの設計の理解性や変更容易性の向上を目的として、
    そのソフトウェアの外部から見た挙動(振る舞い)を変えずに内部構造を
    再構成すること
  - ✓ 大きな(複雑な)設計変更を一連の小さな変換により実現
    (例)関数の名前変更
    変数の名前変更
    関数の移動
    変数の移動



ソフトウェア再利用

- ソフトウェア再利用(software reuse)
  開発したソフトウェアの任意の要素を繰り返し使用する作業
  - ✓ 文書、コード、設計、要求、テストケース、経験、知見…
  - (a) 生産者側での再利用(producer reuse): 再利用可能な要素を作成
    vs. 消費者側での再利用(consumer reuse): 再利用可能な要素を使用
  - (b) ブラックボックス再利用(black-box reuse): 修正なしで再利用
    vs. ホワイトボックス再利用(white-box reuse): 一部変更して再利用
  - (c) 構成的再利用(compositional reuse):
    再利用可能な要素を組み合わせてシステムを構築
    vs. 生成的再利用(generative reuse):
    実際に使用する要素を再利用可能な要素から精製
  - (d) 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の再利用
    vs. 水平的再利用(horizontal reuse):
    プロジェクトや領域を横切る再利用

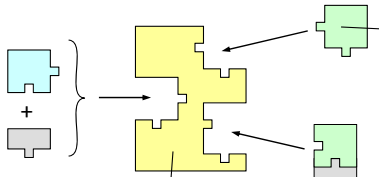
コンポーネントウェア



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



コンポーネント(component)
- 標準化された特定のアーキテクチャの上で動作
- 実装が隠蔽され，インタフェースのみが公開
- 置き換え可能(pluggable)
- 複数のバイナリファイルで構成

インタフェース(interface)
- 提供サービスあるいは要求サービスを明確に定義

ソフトウェアアーキテクチャ
(software architecture)

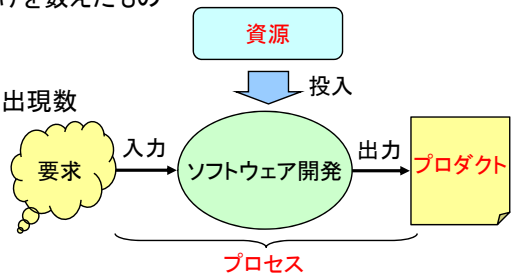
ソフトウェア開発管理

開発計画

- 開発計画の構成要素
  - ✓ 開発の目的(開発プロジェクトの目的)
  - ✓ 開発の目標(システム利用者の要望、業務運営上の方針)
  - ✓ 開発対象業務(開発対象の範囲と機能)
  - ✓ 開発システムの運用方針(管理運用者、業務上の制約)
  - ✓ 開発システムの基本構成(SW構成、HW構成、NW構成)
  - ✓ 開発工数と開発コスト
    ソフトウェアメトリクス(metrics: 定量的評価尺度)
    工数見積もり技法
  - ✓ 開発スケジュール(作業進捗管理): ガントチャート
  - ✓ 開発体制、開発環境、開発方法(方法論やツール)
  - ✓ 成果物の管理方法(構成管理方法)
  - ✓ リスク管理(risk management)
    - リスク衝撃(risk impact): 否定的事象に関連する損失
    - リスク確率(risk probability): 否定的事象が起こる可能性
    - リスク制御(risk control): 否定的事象の影響を最小化・回避するアクション

ソフトウェアメトリクス

- プロダクトメトリクス(product metrics)
  - ✓ ソースコードの規模(行数)
    - LOC (lines of code)
    - NCNB (non-comment non-blank) LOC: コメントや空行を除いた行数
    - ステップ数: プログラムの命令だけを数えたもの
  - ✓ 労力
    - Halsteadの指標:
      演算子、非演算子の種類の数や出現数
  - ✓ 複雑さ
    - McCabeの尺度
- プロセスメトリクス(process metrics)
  - ✓ 作業効率、生産性
  - ✓ プロセス成熟度
    - CMM(capability maturity model: 能力成熟度モデル)
    - SPICE (software process improvement and capability determination)
    - ISO9000: 品質の目標と制約を満たすためにとるべき動作の仕様化
      ISO9000-3: ISO9001のソフトウェア開発向け文書

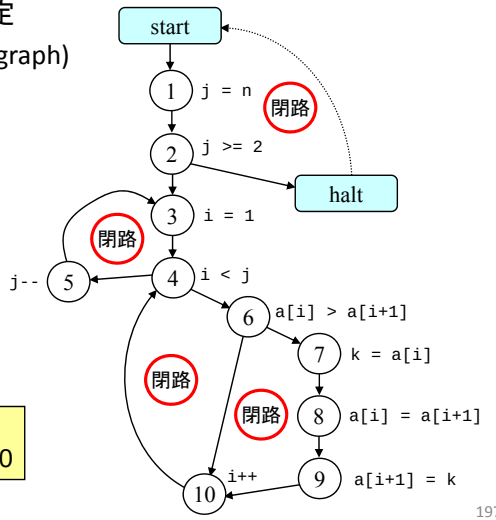


プログラムの複雑さ

- McCabeのサイクロマティック複雑度(cyclomatic complexity)
  プログラムの流れを有向グラフ(CFG)で表現し、
  一時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

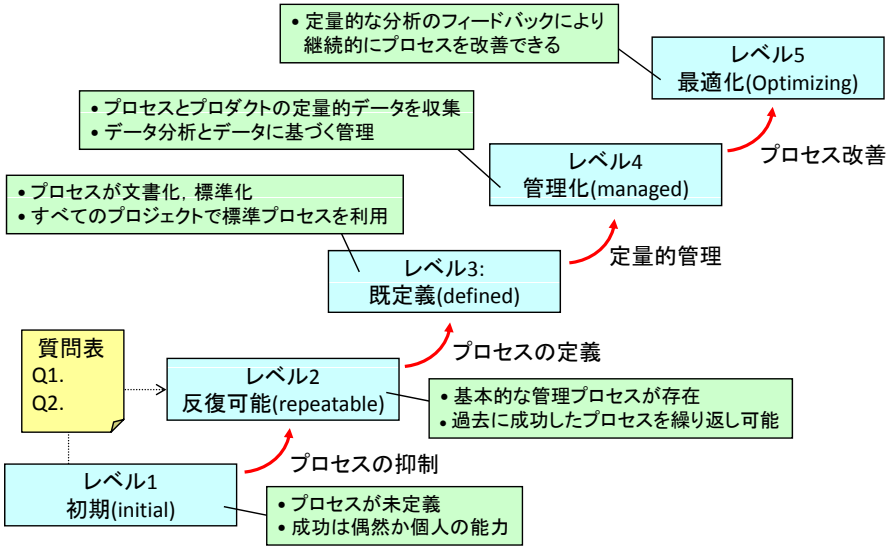
```
for (int j = n; j >=2; j--) {
  for (int i = 1; i < j; i++) {
    if (data[i] > data[i+1]) {
      int k = a[i];
      a[i] = a[i+1];
      a[i+1] = k;
    }
  }
}
```



複雑度 = 閉路の数 = 4

LOC = 9
ノード数 = 10

CMM



工数見積もり

- 類似法
  - ✓ 過去あるいは他の開発プロジェクトの実績を適用
- 標準タスク法
  - ✓ 標準的な作業（タスク）ごとに開発工数の基準を設定しておき、作業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（LOC）と開発工数の関係を統計的なモデルから推測
- COCOMO 2.0
  - ✓ FP法とソフトウェアの規模から開発工数を算出
- ファンクションポイント法（FP法：function point）
  - ✓ 入力や出力などの機能数から規模を算出
  - ✓ 開発工数への変換は未提示

標準タスク法

標準タスクAの作業日数

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

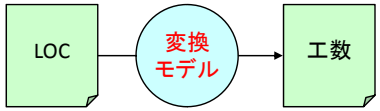
プロジェクトXにおける標準タスクAの件数

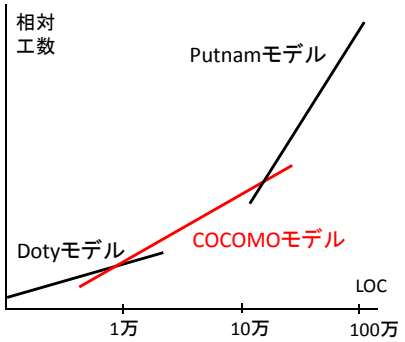
規模＼複雑度	単純	普通	複雑
小	10	5	0
中	10	30	5
大	0	10	10

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(類似法で算出)のみから算出、開発の初期段階で利用
- 中間COCOMO
  - ✓ 開発規模を要求分析結果により判明した影響要因で調整して算出
- 詳細COCOMO
  - ✓ 開発規模を設計結果により判明した影響要因で調整して算出

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

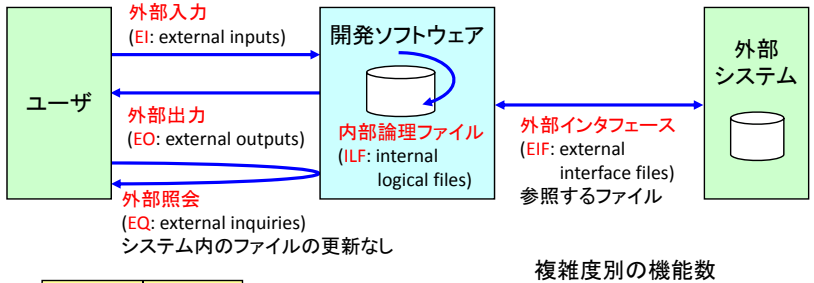
  (例)半組込みモードでの変換モデル

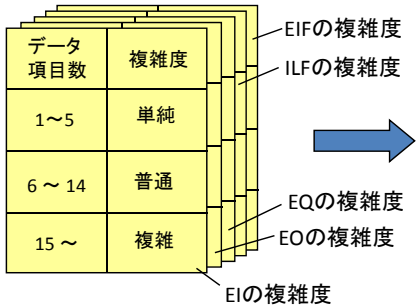
  開発工数(人月) = 3.0 × $(LOC)^{1.12}$ × 調整要因

  開発期間(月) = 2.5 × (開発工数)$^{0.35}$

  調整要因: 製品の複雑度、データベースサイズ、分析者の能力、経験、…
- 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(1)





複雑度別の機能数

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

ファンクションポイント法(2)

複雑度別の機能数

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

重み付け係数

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

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

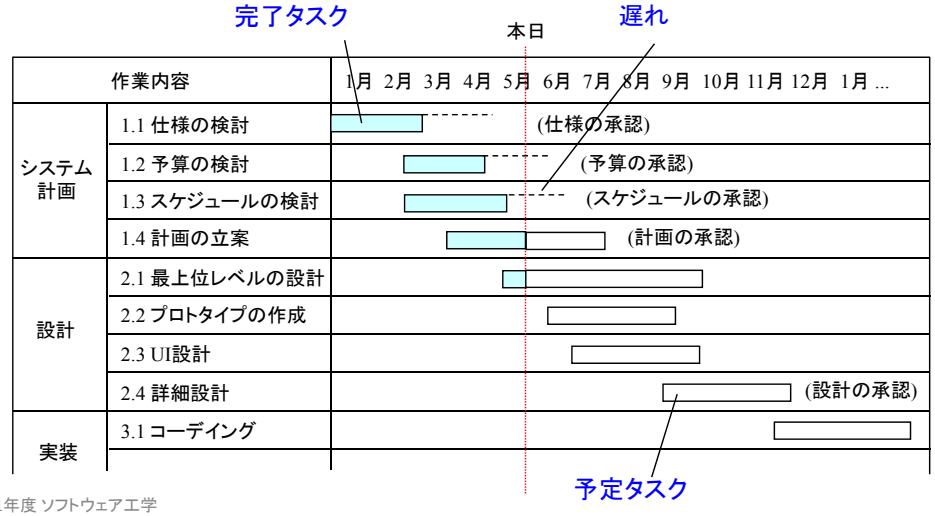
0: まったく関係ない
1: ほとんど影響を受けない
2: 適度に影響を受ける
3: 平均的な影響を受ける
4: 大きな影響を受ける
5: 非常に大きな影響を受ける

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

ガントチャート

- ガントチャート(Gantt chart)
  作業予定の明示 ＋ 作業進捗の追跡



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)
  (例)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)
  (例)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)
  (例)構造化技法支援、ビジュアル化、
  CASE(computer-aided software engineering)[1986]
- 統合開発環境/自動化(1990年代)
  (例) 統合型CASE(全行程を支援)
  リポジトリ(repository):
  開発情報(企業モデル、データモデル、DFD、モジュール構成図、状態遷移図、プログラム設計書など)を集中管理するための保管庫
  cf. ライブラリ(library): ソースコードの保管庫
- オープン化(1990年代後半～)
  (例)コンポーネントウェア、分散開発環境

ソフトウェア統合開発環境Eclipse

- 統合開発環境(IDE: Integrated Development Environment)