OOプロジェクト計画

佐原伸

プロジェクト推進グループ

SRA

質問

- ソフトウェアの生産性は？
- オフィスの生産性は？
  - 電話の中断は、どれくらい影響しているか？
  - 会議時間はどれくらいか？
- ソフトウェアの信頼性は？
  - エラーがどの工程で起こったか管理されているか？
- 個人のユニークな行動は許されているか？
  - 服装
  - 会社に出てこない
  - 勤務報告書を書かない
  - 髪型
- 分野は？
  - 事務処理
  - 制御系
  - その他
- 言語は？
  - C++
  - Smalltalk
  - その他

ソフトウェア／プロジェクトの性質

- ソフトウェア／プロジェクトの性質
  - ソフトウェアの生産性は低い (宮本)
  - 失敗したプロジェクト(DeMarco/Lister)
  - プログラマーの仕事の割合
  - 保守コストの増大 (宮本)
  - ユーザー要求の誤解
  - 上流工程のミス (SEA)
  - 下流工程中心の開発（日本）
  - 工程毎のエラー修正コスト (Boehm)
  - 生産性の限界(Lewis)
  - BoehmのTop 10 (from COCOMO)
  - BoehmのTop 10

ソフトウェアの生産性は低い(宮本)

ソフトウェア生産性とコストの例

a．生産性の例　（従来）
- ●システム・プログラミングの場合：１０～１５行/日
- ●事務処理プログラミングの場合：３０～５０行／日
- ●１人月当たりの平均プログラミング生産性：２００～１０００行／人月

b．ソフトウェア・コストの例　（従来）
- ●アメリカの例
  - １９５３年　約＄４.００／命令
  - １９７５年　約＄８.００／命令
  - １９８０年　約5.００～＄２５.００／命令
- ●日本の例
  - １９８０年　￥１０００～￥５０００／命令
- ●特殊な例　（アメリカの指令制御システム）
  - 開発コスト＝＄７５／命令
  - 保守コスト＝＄４０００／命令

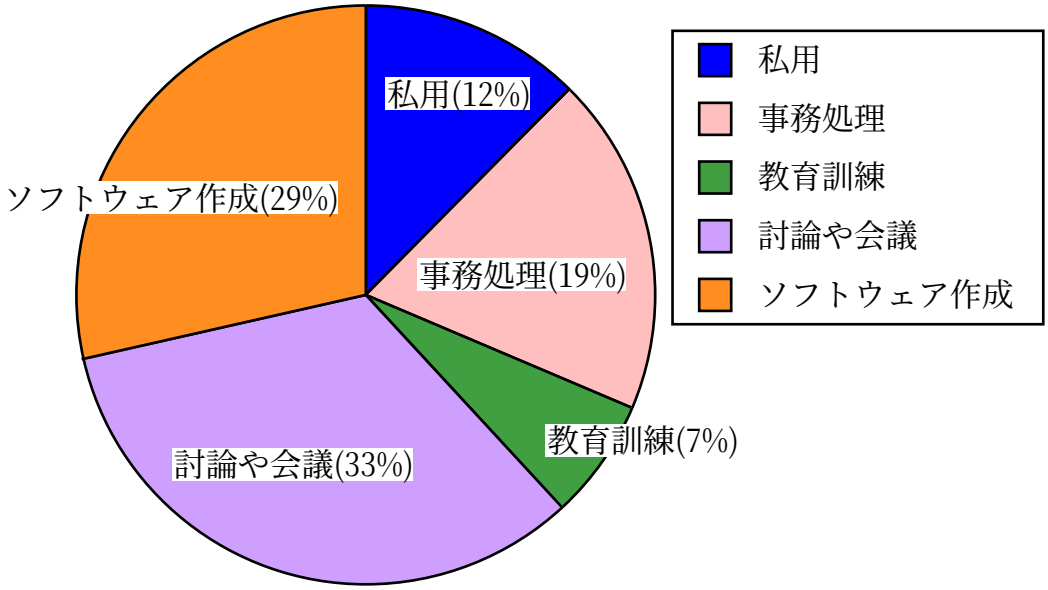
c．プログラマ契約料の例　（現在）
- ●日本の場合　約￥５００,０００／人月　（約＄２,０００）
- ●アメリカの場合　約＄５,０００／人月

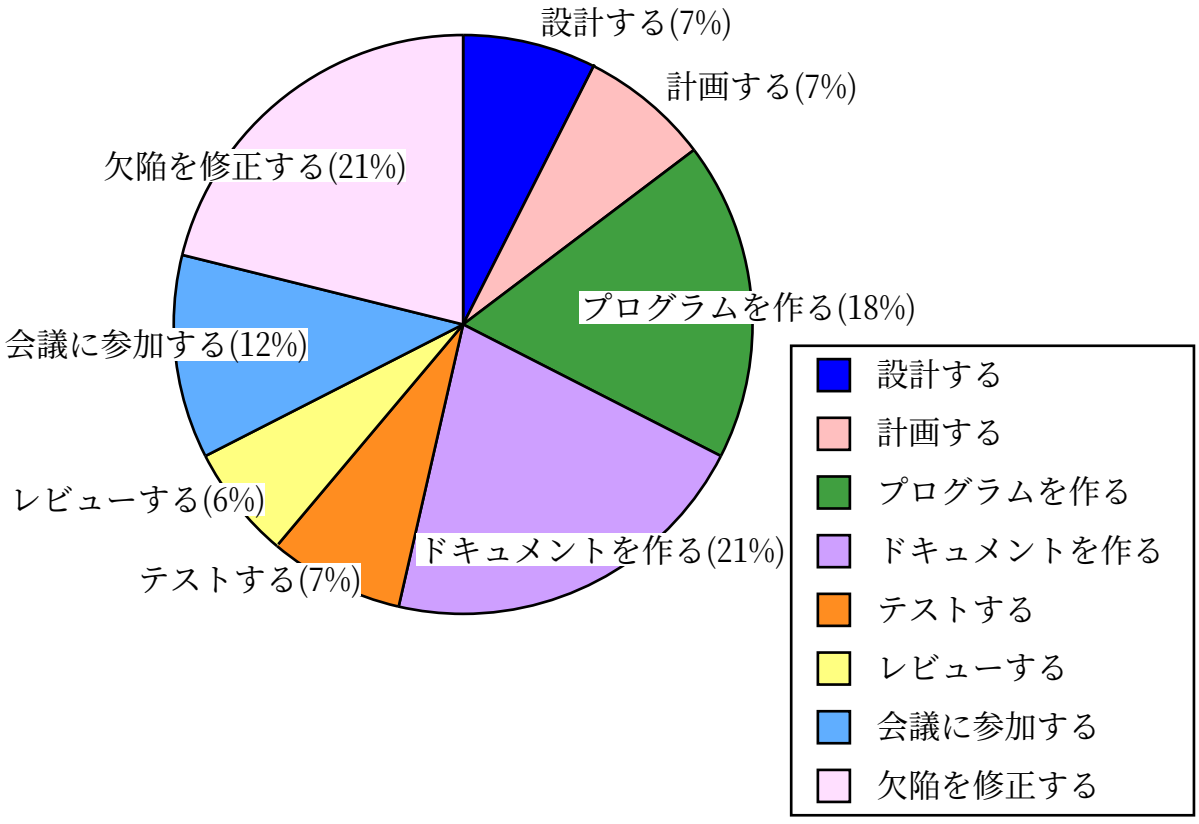
- **１０年前と変わっていない**
  - COCOMO
- **生産性を上げるためには**
  - １ステップ当たりの機能量増加
  - 再利用

失敗したプロジェクト (DeMarco/Lister)

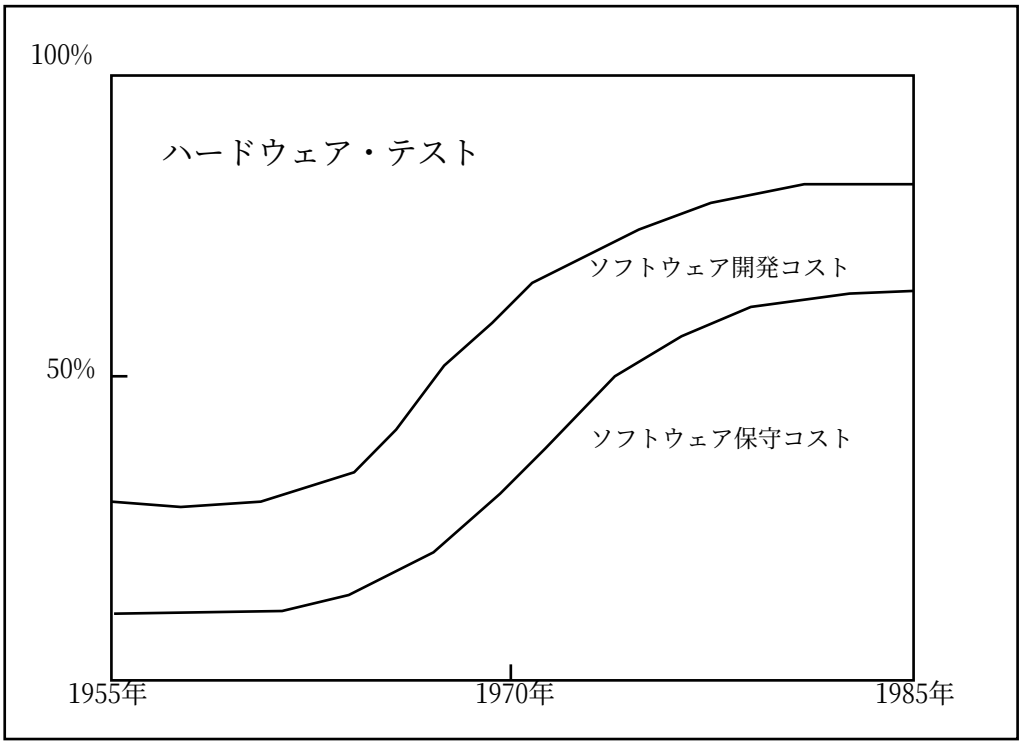
- プロジェクトの１５％が失敗
  - 中止・延期・納入されたが使われない
- 大きなプロジェクトほど失敗する可能性が高い
  - 25人年以上のプロジェクトの25%が完成しなかった
- 失敗の原因は、単なる技術的問題ではない
- 「政治的要因」による失敗と言われていた
  - 実際には...
    - 意思疎通の問題
    - 要員の問題
    - 管理者や顧客への幻滅感
    - 意欲の欠如
    - 高い退職率

プログラマーの仕事の割合





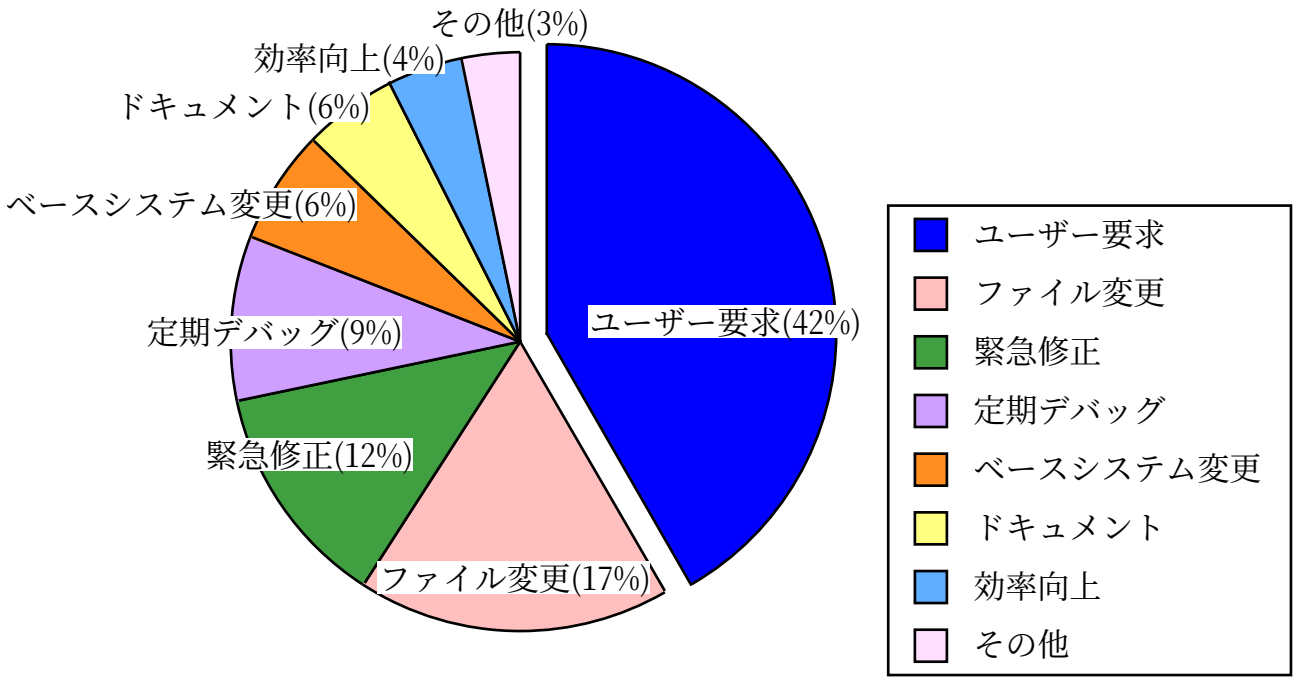
保守コストの増大 (宮本)



ハードウェアとソフトウェアのコスト比

- **保守コストが年々増大している**

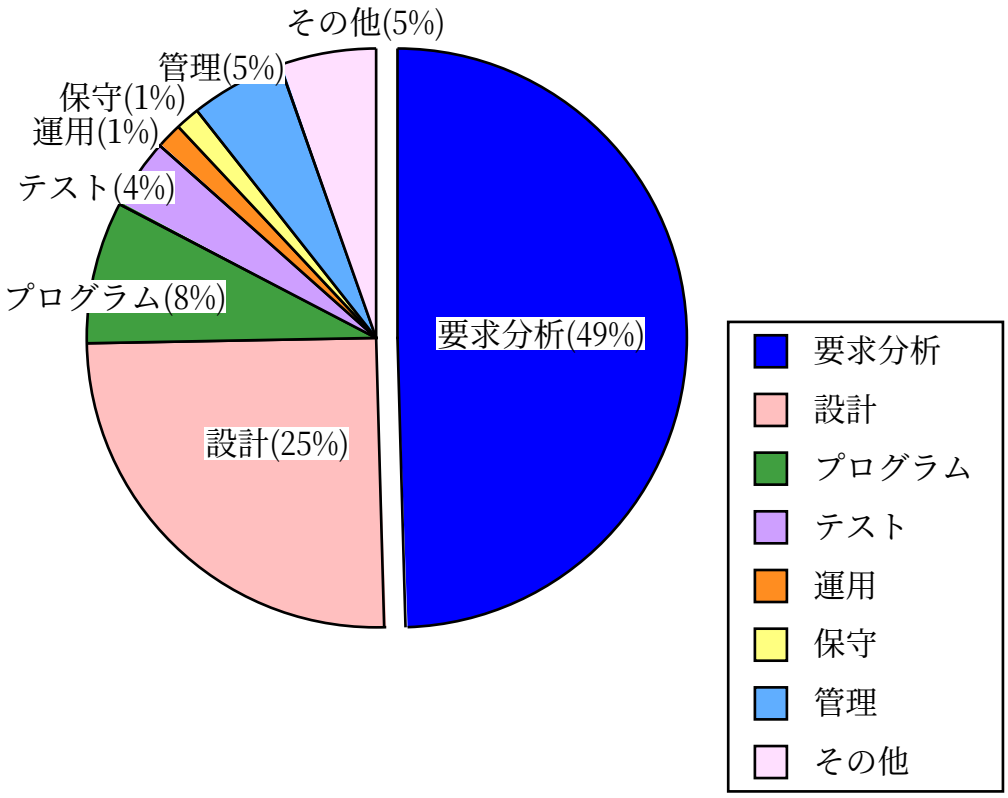
ユーザー要求の誤解



Source: B.P.Lientz and E.B. Swanson. Software Maintenance: A User/Management Tug of War. Data Management, pp.26-30, 1979

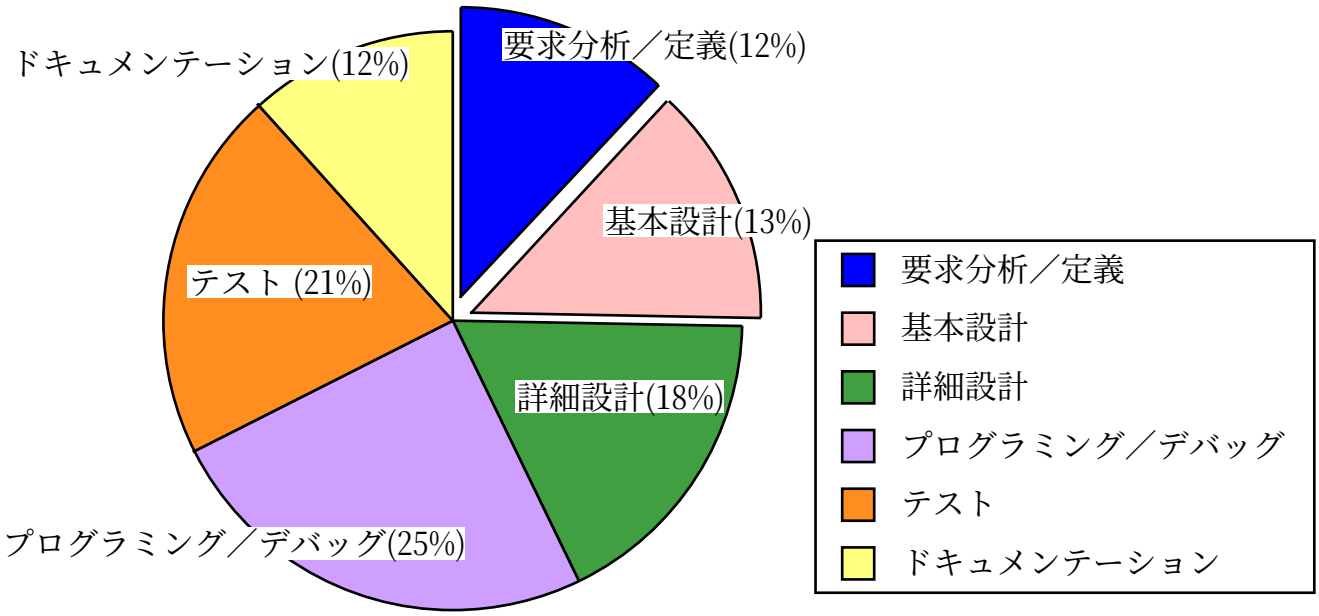
- **保守作業の内訳**

上流工程のミス (SEA)

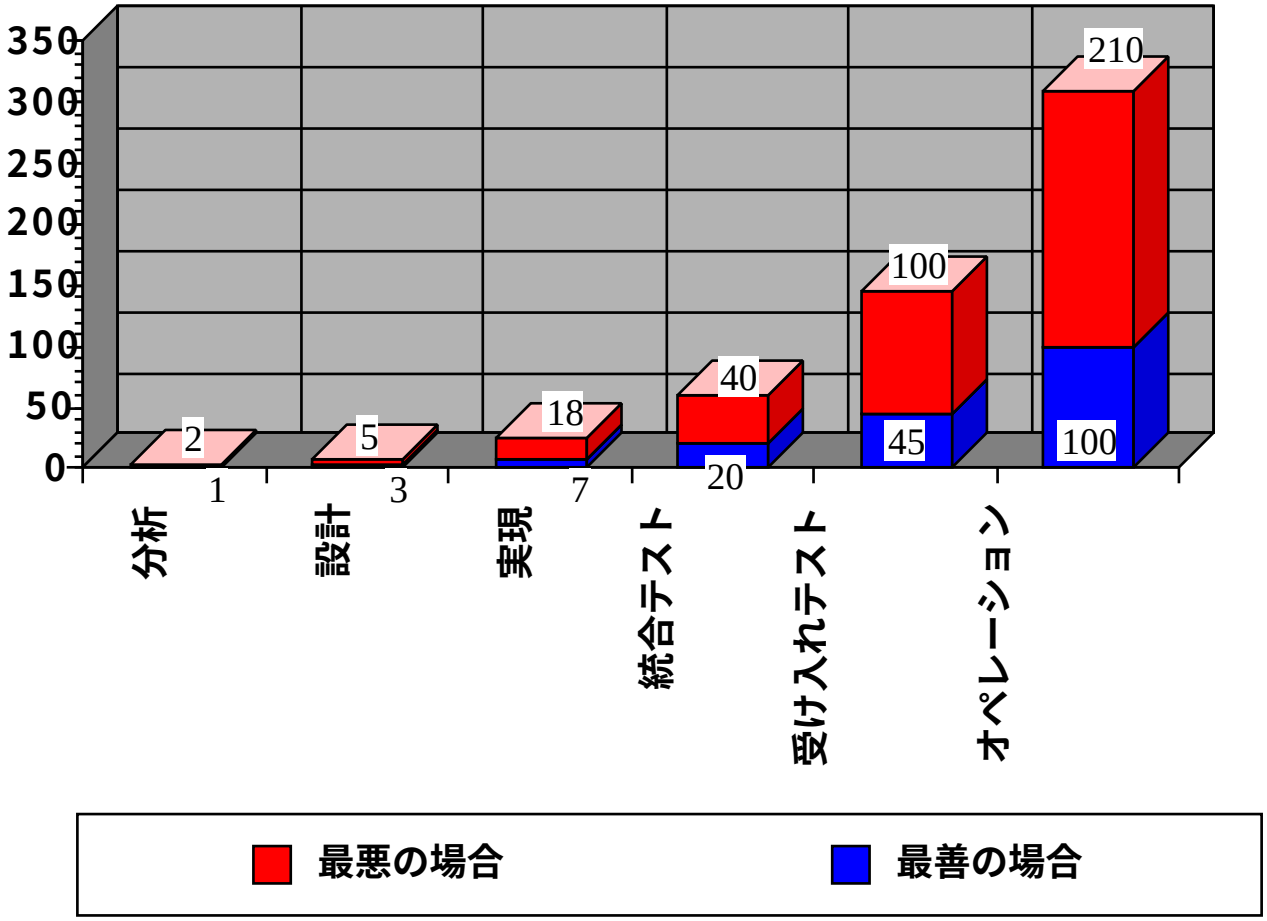


- **重大なトラブルの７０%から７５%が要求分析と設計工程で発生している**

下流工程中心の開発（日本）



工程毎のエラー修正コスト (Boehm)



生産性の限界(Lewis)

製品規模	生産性	行数/人月
超大規模 (>512K)	36 - 250	72
大規模 (64-512K)	63 - 500	166
中規模 (16-64K)	125 - 1000	250
中小規模 (2-16K)	200 - 1250	400
小規模 (<2K)	333 - 2000	667

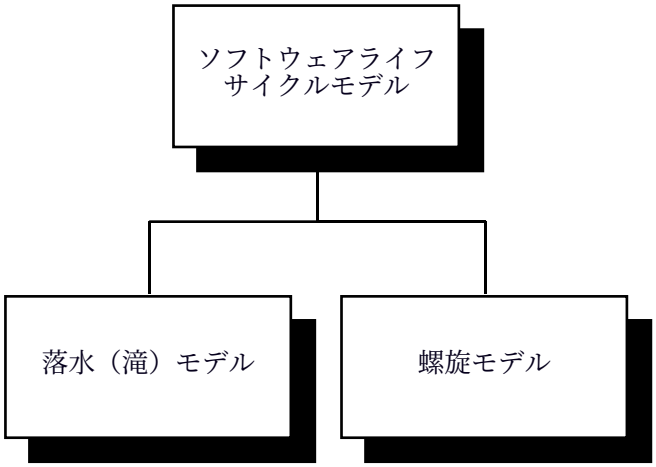
BoehmのTop 10 (from COCOMO)

- １００倍のコスト
  - 出荷したソフトウェアの欠陥は、分析工程の欠陥の１００倍のコスト
- 工期の圧縮
  - 通常のプロジェクトの工期は人や金やその他を投入することによって25%圧縮できる、がそれ以上は不可能である
    - ただし、「ブルックスの法則」に注意
      - 遅れているプロジェクトに人を投入すると、ますます遅れる
- 保守コスト
  - 保守コストは開発コストの２倍である
- 開発と保守のコスト
  - 開発と保守のコストは、主にソース行数の関数である
- 人の生産性
  - 人による生産性のバラツキは非常に大きい

BoehmのTop 10

- ソフトウェアコスト
  - ソフトウェアコスト対ハードウェアコスト
    - １９５５年は１５：８５
    - １９８５年は８５：１５
    - 今後はさらにソフトウェアコストの比重が高まる
- コーディング
  - コーディングは開発コストのわずか１５％を占めるだけである
- 製品と単体プログラム
  - アプリケーション製品はプログラムの３倍のコストであり、システムソフトウェア製品はそのまた３倍のコストがかかる
- ウォークスルー
  - ウォークスルーは、エラーの６０％を見つけ出す
- Paretoの法則
  - モジュールの２０％が、コストの８０％を消費する
  - モジュールの２０％が、エラーの８０％を含む
  - モジュールの２０％が、修正予算の８０％を消費する
  - モジュールの２０％が、実行時間の８０％を使う
    - Knuthの法則
      - コードの２～３％が、実行時間のほとんどを使う
  - ツールの２０％が、時間の８０％を使う

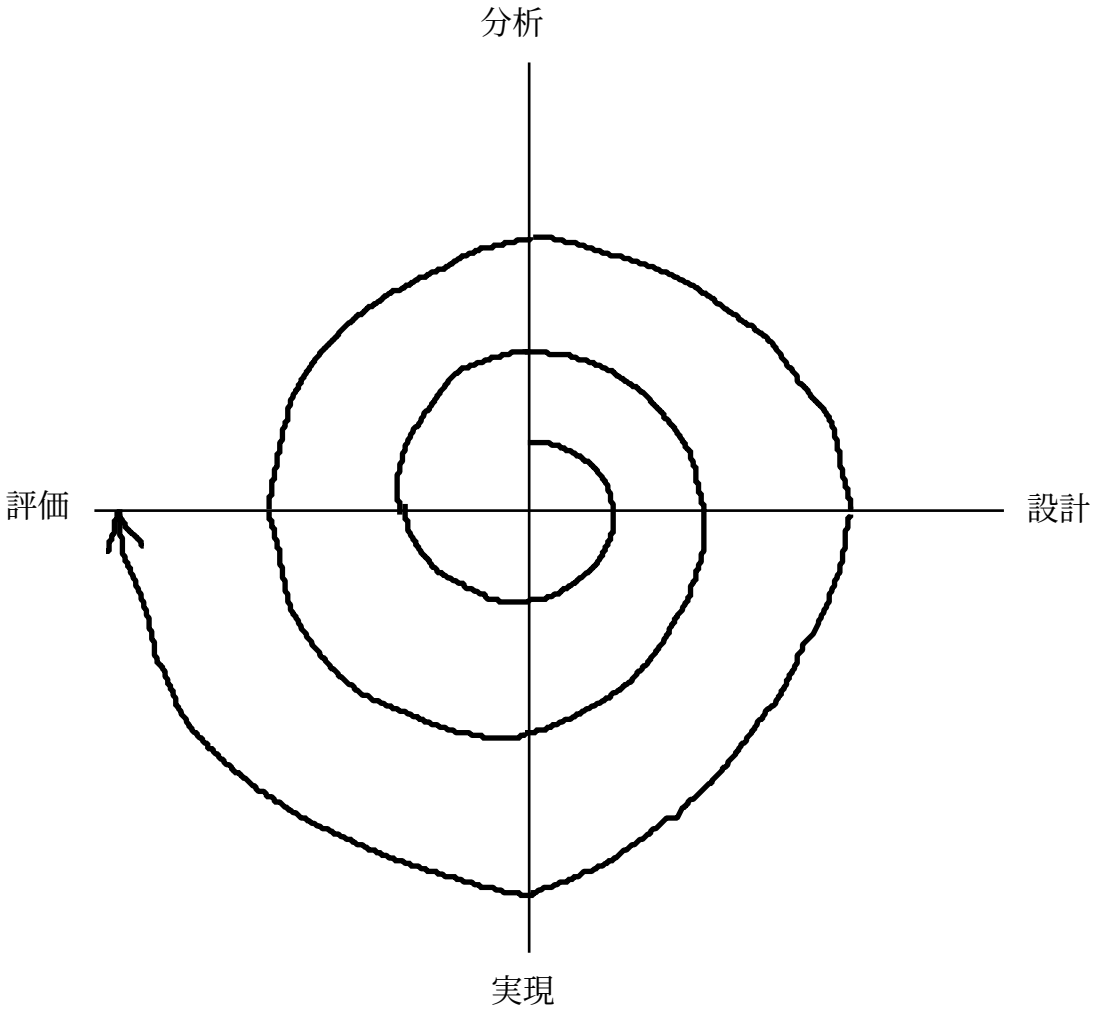
開発プロセス



落水（滝）モデル

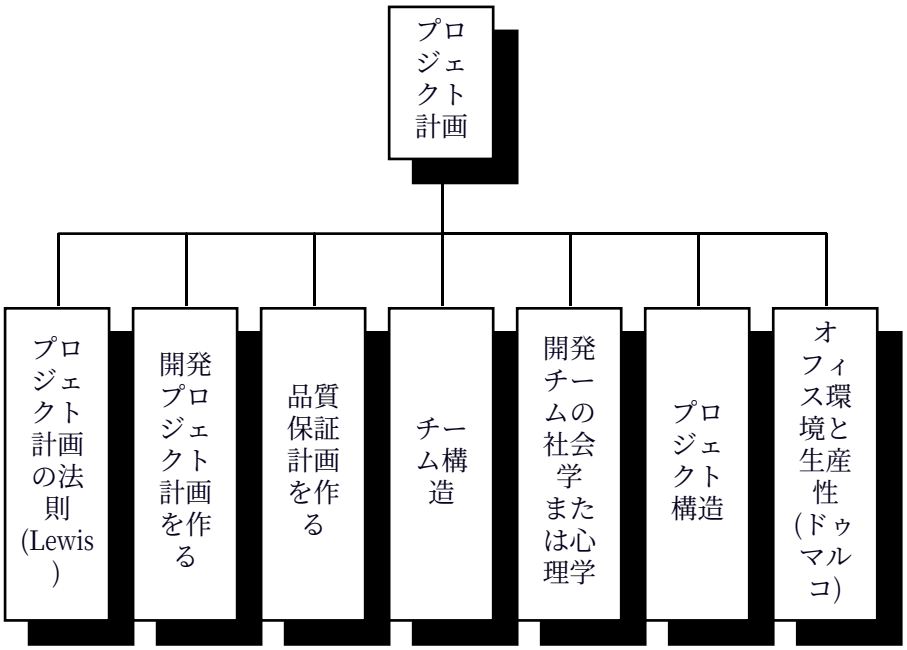
- ソフトウェア工学の３０年前のモデル
- 問題点
  - 紙の上での作業を前提にしている
  - 結果が見えるまでに多くの時間がかかる
  - 修正が少ないことを前提にしている
  - 誤りの検出が開発の終わりごろまで遅れる
  - 再利用を促進できない
  - プロトタイピングが奨励されない
  - 普通は厳密に実施されない

螺旋モデル



- **ラピッドプロトタイピング**
  - ３カ月程度で、分析／設計／実現／評価を繰り返す

プロジェクト計画


プロジェクト計画
プロジェクト計画法則 (Lewis)
開発プロジェクト計画を作る
品質証画作る
チーム構造
開発チームの社会学または心学
プロジェクト構造
オフィス環境と生産性 (ドゥマルコ)

プロジェクト計画の法則 (Lewis)

- 90%
  - プロジェクトの最初の９０％は工数の９０％かかる．最後の１０％も工数の９０％かかる。
- Murphyの法則
  - 何かがうまく行かなければ、今後もうまく行かないだろう
- L.B. Johnson
  - 成功への近道より、失敗の罠の方が多い
- H.L. Mencken
  - 全ての複雑な問題には回答がある．が、うまく動かない．
- C. Schulz
  - 手に負えない問題はない．楽勝できないだけだ．
- David Parnas
  - 誰かが「理論的には」というとき、それは「本当ではない」ことを意味する．

開発プロジェクト計画を作る

- 例
  - はじめに
    - 計画の目的
    - 計画の範囲
    - 用語の定義
    - 参考文献・ドキュメント
  - 管理
    - 体制
      - いくつかのチームに分け、リーダーを置く
      - テストチームとドキュメントチームを置く
    - 責任とコミュニケーションインタフェース
      - 分析・設計・コードのそれぞれをウォークスルーする
      - ドキュメントは開発チームが書くが、テストチームとドキュメントチームが保守する
    - 方針と手続き
      - 各リーダーがクラス分けと名前付けを行う
      - 信頼性と製品の効率を両方追求する
      - 全ての生産物は点検する
  - 活動
    - 構成管理と版管理
    - プロジェクト状態の追跡
      - 全てのプログラマーは２週間に１回以下の項目を報告する
        - どのサブシステムをやっているか
        - 前回までに何ができたか
        - どんな問題が起きたか
    - ツール・技法・方法論
      - 分析と設計はＯＭＴを使う
      - プロジェクト管理はMacProjectを使う

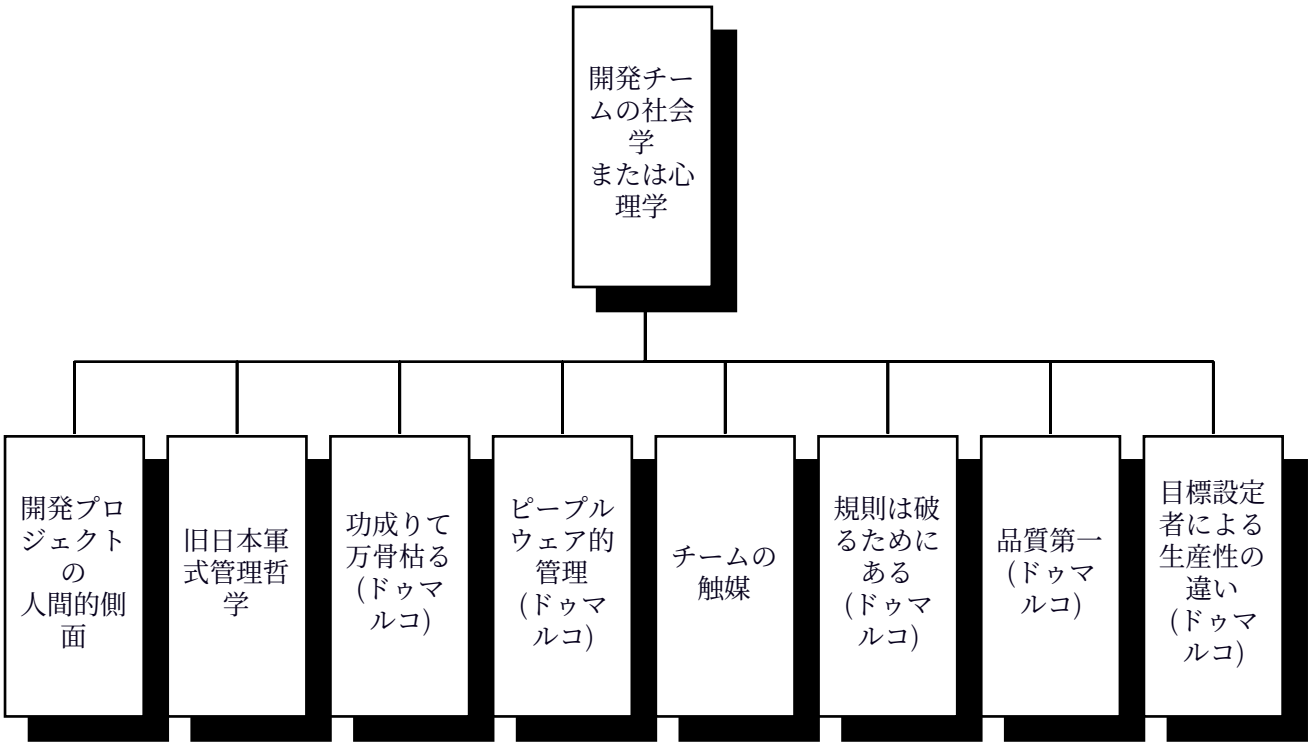
品質保証計画を作る

- 例
  - 計画の目的
  - 参考文献・ドキュメント
  - 管理
    - プログラムは本人たちがテストし、さらにテストチームがテストする
    - ユーザーを交えたウォークスルーを行う
  - 文書化
    - 以下のテスト用ドキュメントを作る
      - 要求された機能と効率の一覧
      - 設計上の主なコンポーネントとそのインタフェースの一覧
      - 検証計画
  - 標準
  - レビュー
    - 要求仕様のレビュー
    - システム設計のレビュー
    - プログラム設計のレビュー
    - 検証計画のレビュー
  - 問題報告
  - ツールと技法
- 質問
  - あなたの部署には、品質保証計画か、それに代わるものがありますか？
  - 無いとすれば、ソフトウェアの品質はどうやって維持していますか？

チーム構造

- 旧日本軍型チーム
  - 構成
    - 専務・部長・課長・主任・その他大勢
  - 特徴
    - 技術が分からない人が技術的決定をする
      - 誤った現状の把握（クックス．ノモンハン）
    - 専門家がいないため、最新の技術が取り込めない
      - 二流の火器と想像力に欠ける教理
    - コミュニケーションのロスが大きい
      - 支離滅裂な指揮関係と中央における計画性の欠如
    - 誰も責任を取らなくてよい
      - 辻政信をはじめ誰も責任を取らず、日中戦争・太平洋戦争へ
- チーフプログラマーチーム
  - 構成
    - チーフプログラマー・バックアッププログラマー・秘書・秘書・スペシャリスト
  - 特徴
    - 技術的決定をできる人間が、自分で作る
    - 専門家を置けるため、最新の技術が取り込みやすい
    - コミュニケーションのロスが少なくて済む
    - 責任は明白になる

開発チームの社会学
または心理学



開発プロジェクトの人間的側面

- ドゥマルコ
  - ソフトウェア開発上の問題の多くは、技術的というより社会学的なものである
  - 人は交換できる部品ではない
- ワインバーグ
  - 成功のほとんど全てが、少数の傑出した技術労働者の働きに依存している
    - 彼らは自分の回りのものすべては最高であることを望む人々であった。

旧日本軍式管理哲学

- 失敗をするな
- 兵隊を休ませるな
- へまをやったやつは厳罰だ
- 兵隊はいくらでも補充できる
- 決められたやり方を手早くやれ
- 作業手順を標準化せよ
- 新しいことはするな。そんなことは参謀の仕事だ

功成りて万骨枯る
(ドゥマルコ)

- 米国的管理
  - もっと長い時間働くようにプレッシャーをかける
  - 製品の開発過程を機械化する
  - 製品の品質について妥協する
  - 手順を標準化する
- 早くやれとせかせば、雑な仕事をするだけで、質の高い仕事はしない

ピープルウェア的管理 (ドゥマルコ)

- エラー大歓迎
  - 間違いを許さないと、人は消極的になるだけ
    - 失敗しそうなことには手を出さなくなる
  - 開発標準や作業規定の無理強いは悪である
- 管理とは尻を叩くこと？
  - ハンバーガーを作るにはうまくいくかもしれないが、体でなく頭を使う仕事ではうまくいかない
    - 製造作業では、作業者を部品の一つとして考えると便利なこともあるが...
  - 担当者の自発的やる気が認められないと、やる気は失せる
- 個人のユニークさを認めるか？
  - ネクタイを締めてこない
  - 自宅に高性能WS
  - 経費の使い方
    - 書籍代が異常に高い
    - ソフトウェア代が異常に高い
- 質問
  - エラーを認めないとどんな利点があるか？
  - 尻を叩く管理にはどのような利点があるか？
  - 個人のユニークさを認めないと、どんな利点があるか？

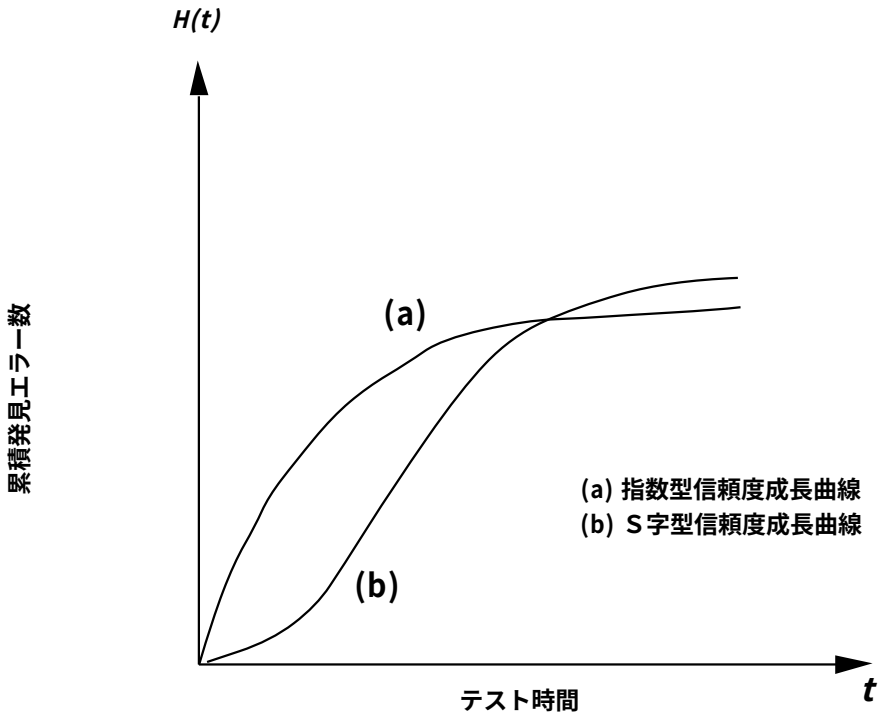
チームの触媒

- プロジェクトに人を入れる場合、静的能力を重視し過ぎる
  - コーディィング能力や設計能力やドキュメントがどのくらい書けるかなど
- 動的能力を重視する
  - チーム全体にうまくなじむか
  - チームの触媒的役割
    - ネアカ型
      - その人がいると、担当者間の意思疎通がよくなり、プロジェクトが楽しくなり、チームの結束は固くなる
    - ネクラ型
      - あるいは、普段は何をしているか分からないのだが、皆が気がつかない問題に気づく人
- 人の組合せが、高い生産性を上げることがある
  - ベルダーブレーメン
  - 鹿島アントラーズ

規則は破るためにある(ドゥマルコ)

- スカンクワークプロジェクト
  - 公には存在しないひそかな作戦
    - 例
      - DEC PDP-11
      - 某社へのUNIX導入
        - 公式決定はWANGを使った開発環境
- お遊びセッション（ヨードン）
  - 公の自由時間
    - 例
      - Object Pearl
      - 明
      - Mac OBJ
- 稟議なしの注文書
  - VAX-11 780注文
- 神保町めぐり
- 喫茶店プロジェクト
- 質問
  - 規則破りをしたことがあるか？
  - 規則破りをどう思うか？
  - 規則を守る利点は何か？

品質第一
(ドゥマルコ)



ソフトウェア信頼度成長曲線

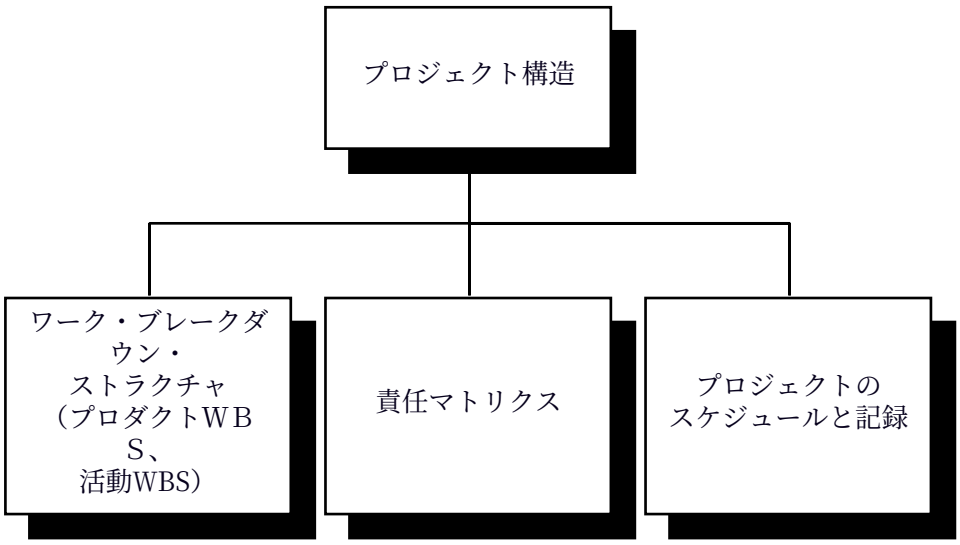
- **ユーザーの要求を超えた品質水準は、生産性を上げるひとつの手段である**
- **品質はただである。ただし、品質に対して喜んで金を出す人だけに...**
- **どうするか?**
  - 原因分析
    - 記録された欠陥をひとつひとつ分析する
      - 全部追跡しなくても、サンプルで十分
  - 信頼度成長曲線
    - 稼働可能状態の判定
      - 信頼度成長曲線と欠陥数の実績を比べて判断する

目標設定者による生産性の違い(ドゥマルコ)

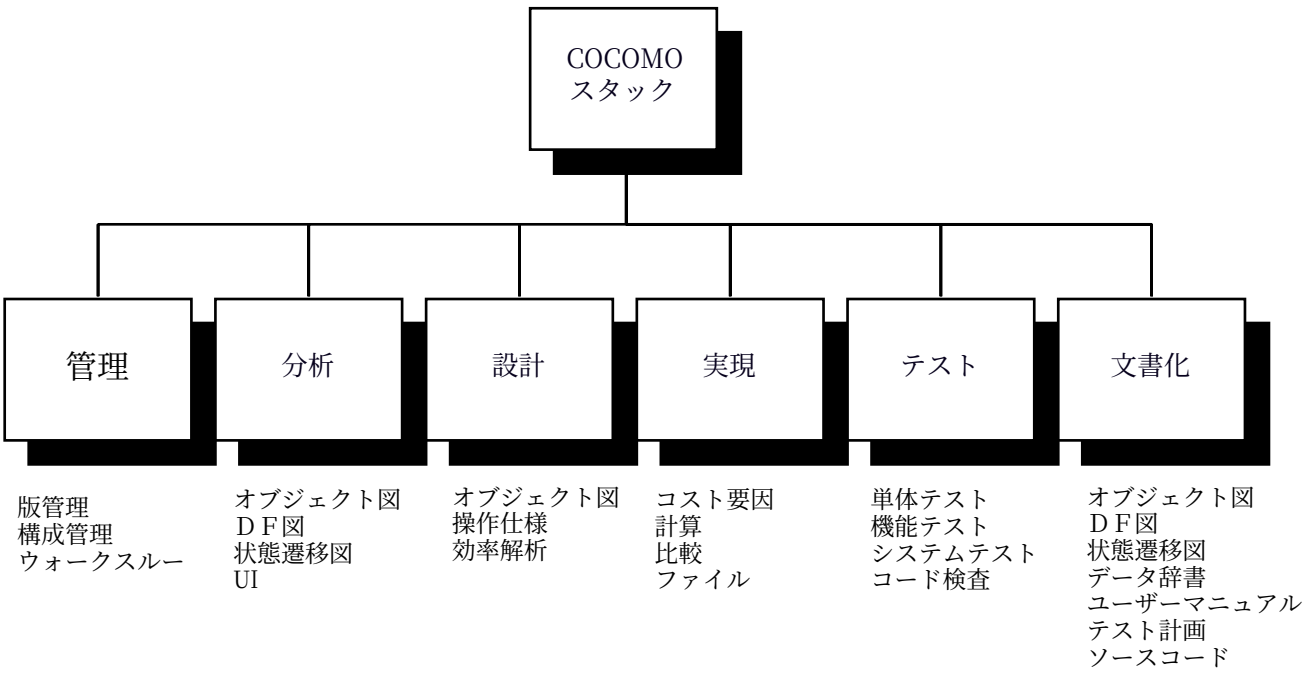
目標値設定者	平均の生産性	プロジェクト数
プログラマ	8	19
管理者	6.6	23
プログラマと管理者	7.8	16
システムアナリスト	9.5	21
目標なし	11	24

- 「目標なし」 が一番生産性が高い

プロジェクト構造


プロジェクト構造
ワーク・ブレークダウン・ストラクチャ（プロダクトWBS、活動WBS）
責任マトリクス
プロジェクトのスケジュールと記録

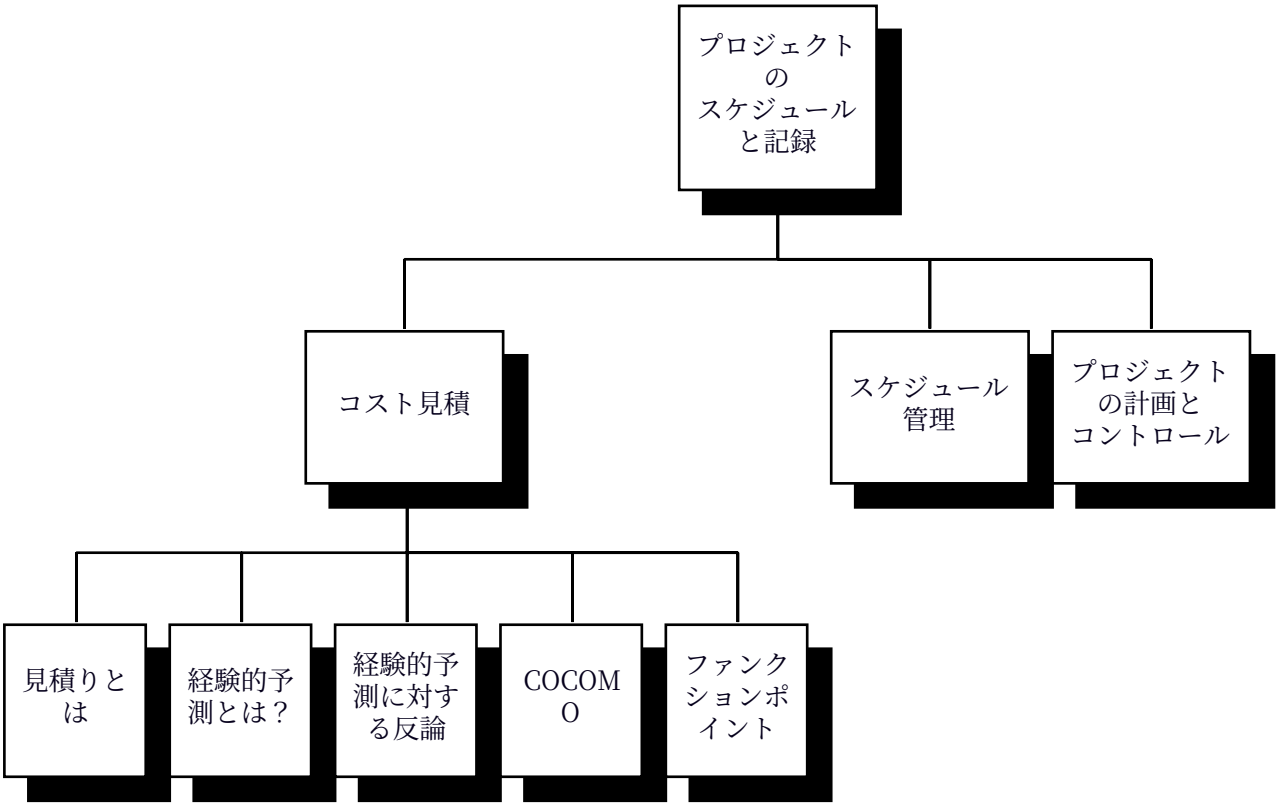
ワーク・ブレークダウン・ストラクチャ（プロダクトＷＢＳ、活動WBS）



責任マトリクス

タスク	佐原	引地	土屋	桜井
責任マトリクス	○			
チーム会議主催	○			
分析	○		○	
オブジェクト図	○			
ＤＦ図	○		○	
状態遷移図	○		○	
設計	○		○	○
システム設計	○		○	
オブジェクト設計	○		○	○
品質計画の管理	○			
設計検査	○		○	○
ログとメモとドキュメントの収集		○		○
版管理		○	○	
ユーザーズマニュアル		○		○
コード検査	○		○	○
単体テスト	○	○	○	○
機能テスト	○	○	○	○
システムテスト	○	○	○	○

プロジェクトの スケジュールと記録



見積もりとは

- 見積もりを上回るか下回る確率が等しい予測
- えせ見積もり技法（DeMarcoの分類）
  - 今回の見積り＝前回の見積もり
  - 新しい見積り＝前回の見積り＋許される遅れ
  - 過去に生じた遅れ＝将来のつけ
  - 当初の見積り＝上司の期待値
  - 見積り＝上司の期待値＋上司の許容度
- 人の偏向（DeMarcoの実験）
  - 所要時間を過小に見積もる傾向がある
- 複雑で大きな塊は見積もりが困難
  - 結合度の少ない、強度の高い塊に分ける

経験的予測とは？

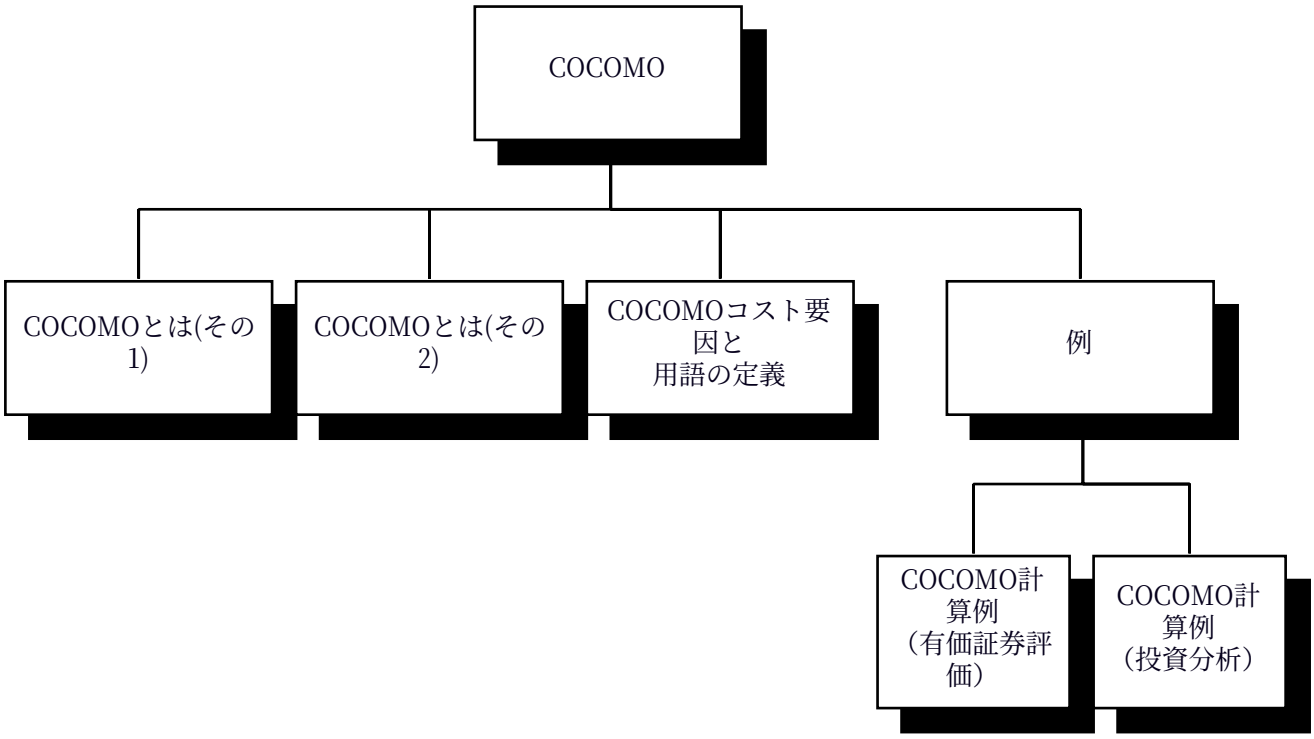
- 統計的な推測をもとに行う予測
- ソフトウェアの経験的予測にはコード数（行数）・機能量が使われる
- 対象システムの仕組みなどが大きく変わるときには使えない

経験的予測に対する反論

- データがない
  - データが十分でないことは確かだが、米国では結構測定されている
- 開発システムのコード数（行数）を早い時期に予測するのは無理
  - 今のところ他に適当な指針がない
  - システムモデル（後述）を入力とすれば、結構予測が可能
- ソフトウェアシステムは皆条件が異なる
  - 構成単位にまで分解してみれば、違いは少ない
- 予想の数字が大きすぎると言って、管理者が受け入れてくれないだろう
  - できないものは結局できない

5

COCOMO


COCOMO
COCOMOとは(その1)
COCOMOとは(その2)
COCOMOコスト要因と用語の定義
例
COCOMO計算例（有価証券評価）
COCOMO計算例（投資分析）

COCOMOとは(その1)

- COCOMO（Boehm@TRW）
  - 他種類の多くのプロジェクトから作成されたモデル
  - ソフトウェア工学国際会議最優秀論文
  - TRW社はこの後のモデルを企業秘密としている
  - コスト要因が１５個であり、見積もりしやすく、精度も高い
  - Basic, Intermediate, Detailedの３種類有り、段階的に使える
- COCOMOで分かったこと
  - 人の影響の大きさ
  - 言語の種類に依存しない

COCOMO

- COCOMOはCOnstructive COst Modelの略である。米国のTRW社のBoehmにより開発された。
- モデルは「開発作業」・「保守作業」・「コンバージョン作業」のそれぞれに「概算レベル(Basic COCOMO)」・「詳細レベル(Detailed COCOMO)」・「中間レベル(Intermediate COCOMO)」という３種類のモデルを持つ。
- 一番単純なBasic COCOMOは、ソースコードの行数からコストを求める。2番目はIntermediate COCOMOで、ソースコード行数に加えて15個のコスト因子からコストを求める。3番目はDetailed COCOMOで、ソースコード行数とコスト因子を個々の工程・サブシステム・モジュール毎に考慮してコストを求める。以下では、良く使われる開発作業のBasic COCOMOとIntermediateCOCOMO及び保守作業のIntermediateCOCOMOについて説明する。
- COCOMOの見積の誤差は大体20%以内であるが、気分によらずいつも一定の結果を出すため、定期的に見積りをやり直すことによって、さらに見積精度が上がるであろう。しかし、COCOMOは過去のプロジェクトから導きだした統計的なモデルであるから、魔法のように信じてはならない。
- 富士通ではCOCOMOを改良したFJCOCOMOを使用している。

COCOMOコスト要因と用語の定義

- ソースコード行数(DSI)
- 人月(MM)
- 期間(TDEV)
- 作業時間
- プロジェクトのモード
- 計算式
- コスト要因
- RELY 要求される信頼性
- DATA データベースの大きさ
- CPLX 製品の複雑さ
- TIME 実行時間の制約
- STOR 主記憶の制約
- VIRT 仮想マシンの安定性
- TURN ターンアラウンド時間
- ACAP アナリストの能力
- AEXP アプリケーションの経験
- PCAP プログラマーの能力
- VEXP 仮想マシンの経験
- LEXP プログラミング言語の経験
- MODP 現代的プログラミングの習慣
- TOOL ソフトウェアツールの使用
- SCED 要求される開発スケジュール

ソースコード行数(DSI)

- 　COCOMOで計算に使用する主なパラメータはソースコード行数(DSI)である。DSIはDelivered Source Instructionの略で、以下のような定義がなされている。
- 　Deliveredとは、支援ソフトウェアなどのように出荷しないものを除くことを意味する。ただし、出荷するソフトウェアと同じ支援を行なわなければならないテスト計画・ドキュメント・パラメータなどがあれば、それらも含める。
- 　Source Instructionとは、注釈や既存のユーティリティ・プログラムを除いたソースコードの行数である。ジョブ制御文やフォーマット文あるいはデータの宣言文なども含む。

人月(MM)

- 人月は、ソフトウェア作成工数を表す。
- 人数にはプロジェクトに直接関係するプロジェクト管理者やライブラリアンを含むが、コンピュータセンターのオペレータや秘書・上級管理者は含まない。
- 工数には、「設計」工程以後を含み、「計画と要求分析」工程は含まない。

期間(TDEV)

- 期間は、開発に要する期間(月数)のこと。
- 期間には、「設計」工程以後を含み、「計画と要求分析」工程は含まない。

作業時間

- 　COCOMOでは１ケ月の作業時間を19日=152時間と見なしている。異なる場合、値を修正する必要がある。
- 　作業時間には管理や文書化の時間を含むが、ユーザー教育・納入計画・コンバージョン計画などは含まない。

プロジェクトのモード

- COCOMOでは、プロジェクトの形態により、工数や期間の見積り式が少し異なる。
- 以下の３つのモードがある。
  - （１）orgranic mode(有機体モード)　比較的小規模なチームが、インハウスで、親密な関係を保ちながら開発する。コミュニケーションのためのオーバーヘッドがなく、安定した開発環境で、新しいアーキテクチャやアルゴリズムはあまり必要なく、比較的小規模(50KDSI以内)である。
  - （２）semidetached mode(長屋モード)　organicとembeddedの中間。
  - （３）embedded mode(はまりモード)　制約がきつく、密接に結合したハードウェア・ソフトウェア上で、微妙な調整と操作が必要な、電子決済システムや航空機管制システムなどを開発する場合。

計算式

Basic COCOMO計算式		
モード	工数	期間
Organic	MM=2.4(DSI/1000)^1.05	TDEV=2.5(MM)^0.38
Semidetached	MM=3.0(DSI/1000)^1.12	TDEV=2.5(MM)^0.35
Embedded	MM=3.6(DSI/1000)^1.20	TDEV=2.5(MM)^0.32
Intermediate COCOMO計算式		
モード	工数	期間
Organic	MM=3.2(DSI/1000)^1.05	TDEV=2.5(MM)^0.38
Semidetached	MM=3.0(DSI/1000)^1.12	TDEV=2.5(MM)^0.35
Embedded	MM=2.8(DSI/1000)^1.20	TDEV=2.5(MM)^0.32

- **工数および期間の計算式は以下のようになる。ここで"^"はべき乗計算を示す。**

コスト要因

- 　Intermediate COCOMOの場合は、前節で示した計算式に、以下に示す１５個のコスト因子を掛ける。すなわち、
- EMM = MM * C1 * C2 * C3 * ... * C15
  - EMM : Estimated Man-Month

RELY 要求される信頼性

- 製品に対して要求される信頼性。
  - 非常に低 = 製品のエラーを、開発者が修正すればよいだけのとき。
  - 低 = ユーザーがエラーを容易に回復できるとき。
  - 普通 = 多少の損失は招くが、ユーザーがエラーを回復できるとき。
  - 高 = エラーが大きな財政上の損失を招くか、人身事故を招くとき。

DATA データベースの大きさ

- データベースサイズとプログラムの行数(DSI)の比。
  - DATA = データベースサイズ (バイト数) / DSI。
  - 低 = DATA < 10
  - 普通 = 10 <= DATA < 100
  - 高 = 100 <= DATA < 1000
  - 非常に高 = DATA >= 1000

CPLX 製品の複雑さ

- 製品の複雑さ。
  - 非常に低 = 直線的コードと若干の構造化命令、A=B+C*(D-E)などの単純な式、単純な形式のread, write文、主記憶上の単純な配列などの特徴を持つ製品の場合。
  - 低 = 単純なネスト、大部分は単純な述語、SQRT(B**2-4.*A*Cといった普通のレベルの式、GETとPUTレベルのI/O、編集やデータ構造の変更や中間ファイルの無い単一ファイルの処理などの特徴を持つ製品の時。
  - 普通 = 大部分は単純なネストされた構造化命令、いくらかのモジュール間制御、決定表、標準の数学・統計関数の使用、基本的な行列・ベクトル演算、デバイスの選択・状態チェック・エラー処理を含む入出力処理、複数の入力・単一出力、単純な構造的変更、単純な編集などの特徴を持つ製品の時。
  - 高 = 高度にネストされた構造化命令、キューやスタックの制御、複雑なモジュール間制御、基本的な数値分析、曲線補間、普通の微分方程式、切り捨て・四捨五入、物理I/O操作、最適化されたI/Oのオーバーラップ、データ・ストリームの内容による特別なルーチンの起動、レコードレベルの複雑なデータ再編成などの特徴を持つ製品の時。
  - 非常に高 = 再入可能コードと再帰コード、固定した優先順位の割り込み制御、難解だが構造的な数値分析、通信ラインの取り扱い、汎用的なパラメータ起動によるファイルの構造化、コマンド処理、検索の最適化などの特徴を持つ製品の場合。
  - 最高 = 動的に優先順位が変わる複数のリソースのスケジューリング、マイクロコード・レベルの制御、難解で非構造的な数値分析、デバイスのタイミングに依存したコード、マイクロプログラムの命令、密に結合した動的なデータ構造、自然言語によるデータ管理などの特徴を持つ製品の時。

TIME 実行時間の制約

- 実行時間の総量の何パーセントを使うかを示す。
  - TIME = 使用する実行時間 / 実行時間の総量 * 100
  - 普通 = TIME <= 50%
  - 高 = TIME = 70%
  - 非常に高 = TIME = 85%
  - 最高 = TIME = 95%

STOR 主記憶の制約

- 　主記憶の何パーセントを使用するかを示す。
- STOR = 使用する主記憶容量 / 主記憶容量 * 100
  - 普通 = STOR <= 50%
  - 高 = STOR = 70%
  - 非常に高 = STOR = 85%
  - 最高 = STOR =95%

VIRT 仮想マシンの安定性

- 製品を開発するためのベースとなる仮想マシンの変更頻度。OSの開発であればコンピュータのハードウェアであり、DBMSの開発であればハードウェア+OSであり、DB指向のアプリケーションの開発であればハードウェア+OS+DBMSを仮想マシンと見なす。
  - 低 = 大きな変更が12ケ月に１回位、小さな変更は1ケ月に１回位の時
  - 普通 = 大きな変更が6ケ月に１回位、小さな変更は2週間に１回位の時
  - 高 = 大きな変更が2ケ月に１回位、小さな変更は1週間に１回位の時
  - 非常に高 = 大きな変更が2週間に１回位、小さな変更は2日に１回位の時

TURN ターンアラウンド時間

- 開発者が、コンピュータに入力してから結果を得るまでの平均レスポンス時間(ART)。
  - 低 = 対話型システムを使用している場合
  - 普通 = ART < 4時間
  - 高 = 4 <= ART < 12時間
  - 非常に高 = ART >= 12時間

ACAP アナリストの能力

- 　アナリスト・チームの能力。能力・能率・コミュニケーション能力・協調性などを含むが、アナリストの経験は含まない。アナリスト人口全体のなかの比率で示す。
  - 非常に低 = 下から15%程度
  - 低 = 下から35%程度
  - 普通 = 下から55%程度
  - 高 = 下から75%程度(つまり上位25%)
  - 非常に高 = 下から90%程度(つまり上位10%)

AEXP アプリケーションの経験

- 製品と同様なタイプのアプリケーションの経験。
  - 非常に低 = 4ケ月以下
  - 低 = 1年
  - 普通 = 3年
  - 高 = 6年
  - 非常に高 = 12年以上

PCAP プログラマーの能力

- 　プログラマ・チームの能力。能力・能率・コミュニケーション能力・協調性などを含むが、プログラマの経験は含まない。プログラマ人口全体のなかの比率で示す。
  - 非常に低 = 下から15%程度
  - 低 = 下から35%程度
  - 普通 = 下から55%程度
  - 高 = 下から75%程度(つまり上位25%)
  - 非常に高 = 下から90%程度(つまり上位10%)

VEXP 仮想マシンの経験

- 　プロジェクトチームの仮想マシンに対する経験。プログラミング言語の経験は含まない。
  - 非常に低 = 1ケ月以内
  - 低 = 4ケ月
  - 普通 = 1年
  - 高 = 3年以上

LEXP プログラミング言語の経験

- プロジェクトチームのプログラミング言語の経験。
  - 非常に低 = 1ケ月以内
  - 低 = 4ケ月
  - 普通 = 1年
  - 高 = 3年以上

MODP 現代的プログラミングの習慣

- 　以下の６つの現代的プログラミングの習慣(MPP)があるかいなか。
- （１）段階的詳細化、プロトタイピングを含む、構造化分析・構造化設計。
- （２）.構造化設計ダイヤグラムの使用。
- （３）トップダウン・段階的詳細化による開発。
- （４）設計とコードのウォークスルーとインスペクション。
- （５）構造化プログラミング。
- （６）プログラム・ライブラリアン。
  - 非常に低 = MPPをひとつも使わない
  - 低 = いくつかのMPPの初歩的使用。
  - 普通 = いくつかのMPPの効果的使用。
  - 高 = かなりのMPPの効果的使用。
  - 非常に高 = 全部のMPPの効果的・習慣的使用。

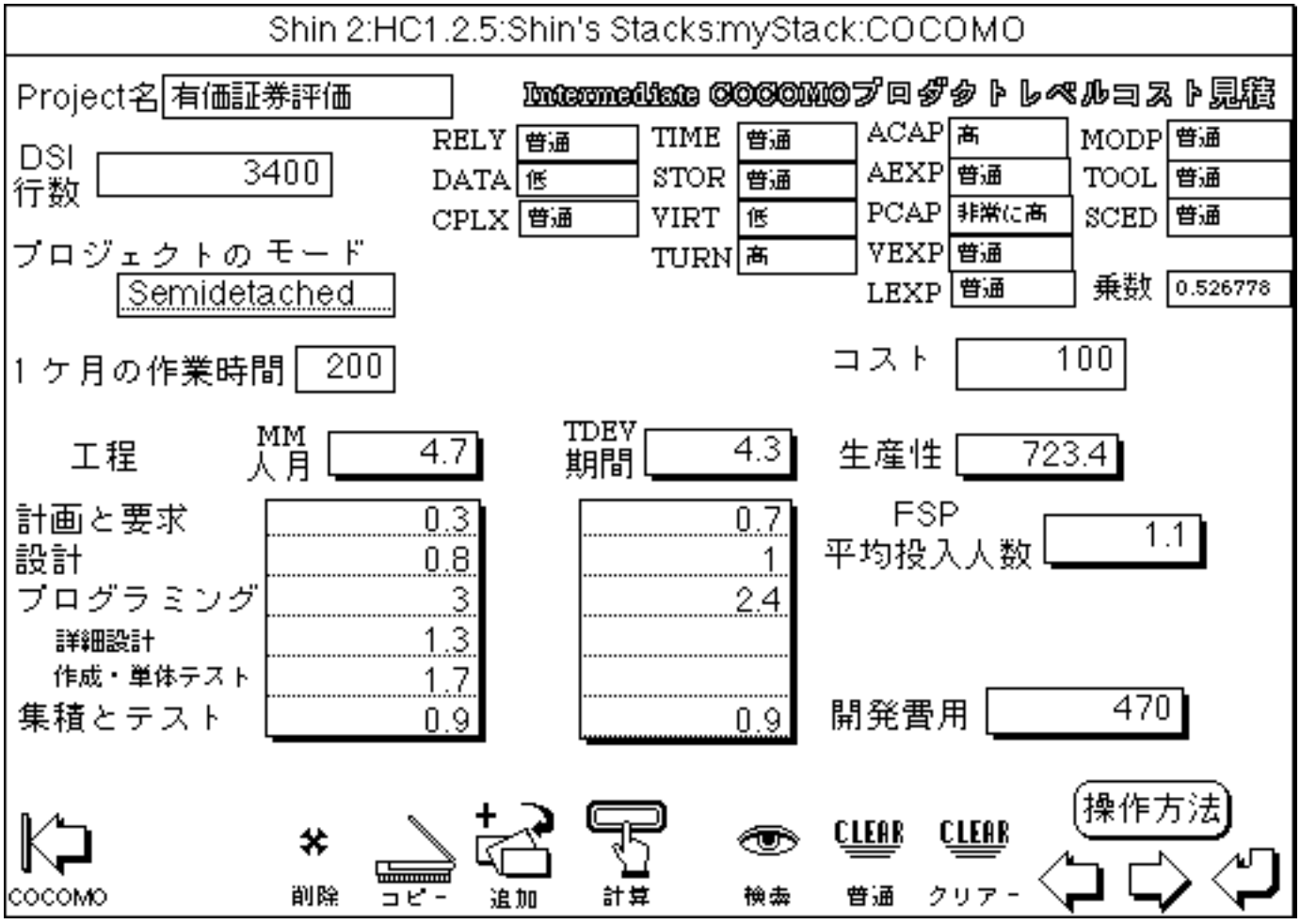
TOOL ソフトウェアツールの使用

- ソフトウェアツールの使用程度。
  - 非常に低 = アセンブラー・基本的リンカー・基本的モニター・バッチ式デバッガーなどを使用している場合。
  - 低 = 高級言語コンパイラー・マクロアセンブラー・単純なオーバーレイを行なうリンカー・言語独立なモニター・バッチ式エディター・基本的なライブラリー管理ツール・基本的なDBツールなどを使用している場合。
  - 普通 = リアルタイムまたはタイムシェアリングOS・DBMS・高度なオーバーレイリンカー・対話的デバッグツール・単純なプログラミング支援ライブラリー・対話的ソースエディターなどを使用している場合。
  - 高 = 仮想メモリーOS・DB設計ツール・単純なプログラム設計言語・効率測定分析ツール・基本的な構成管理機構を持ったプログラミング支援ライブラリー・プログラムのフローとテストケースのアナライザー・基本的なテキストエディターとテキスト管理ツールなどを使用している場合。
  - 非常に高 = 構成管理機構を持った強力なプログラミング支援ライブラリ・強力な統合化文書処理システム・プロジェクト管理システム・要求仕様記述言語と分析ツール・強力な設計ツール・自動化検証システム・必要な特殊ツール(クロスコンパイラー・命令セットシミュレータ・画面フォーマッター・通信処理ツール・データ入力処理ツール・変換ツール)などを使用している場合。

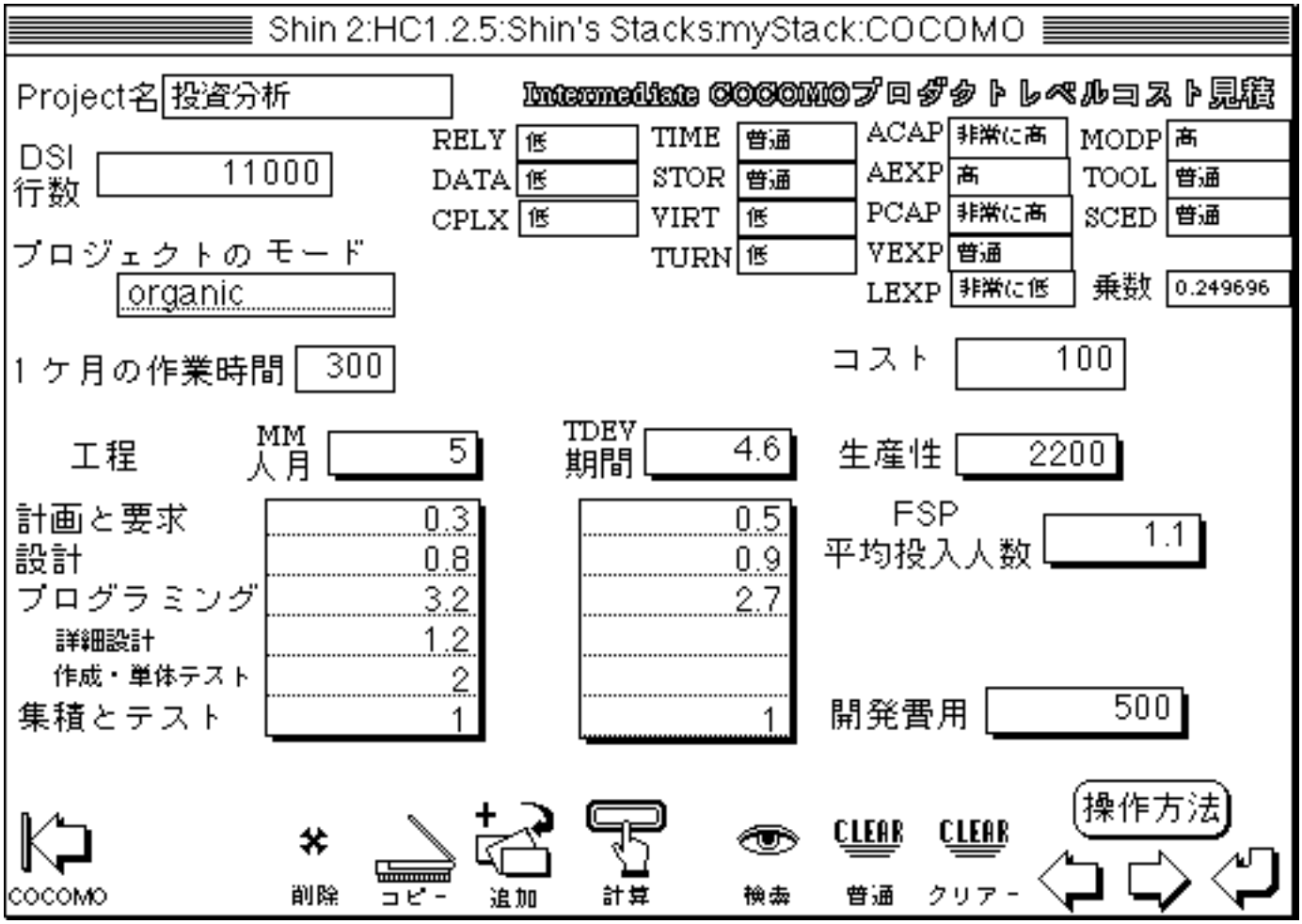
SCED 要求される開発スケジュール

- Basic COCOMOで計算された標準の開発期間(TDEV)からの乖離。標準の開発期間より短くても長くても、開発工数(MM)は標準より大きくなる!!!
  - 非常に低 = TDEVの75%の期間で開発しなければならないとき。
  - 低 = TDEVの85%の期間で開発しなければならないとき。
  - 普通 = TDEVと同じ期間で開発するとき。
  - 高 = TDEVの130%の期間で開発するとき。
  - 非常に高 = TDEVの160%以上の期間で開発するとき。

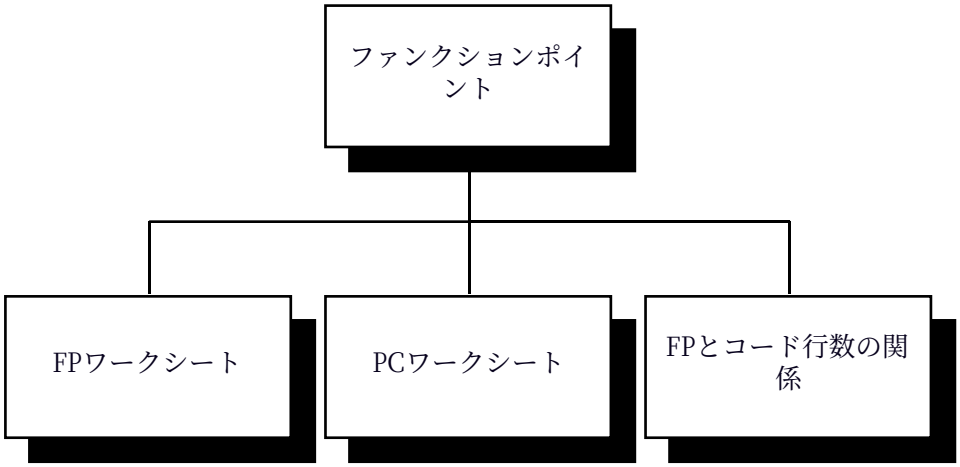
COCOMO計算例
（有価証券評価）



COCOMO計算例
(投資分析)



ファンクションポイント



FPワークシート

計算	単純			平均			複雑			合計
外部入力数	2	3	6	1	4	4	1	6	6	16
外部出力数	3	4	12	2	5	10	1	7	7	29
内部論理ファイル数	5	7	35	2	1	20	1	15	15	70
インタフェースファイル	4	5	20	3	7	21	1	10	10	51
外部要求	10	3	30	2	4	8	2	6	12	50
									To	210

- Function Pointの計算

PCワークシート

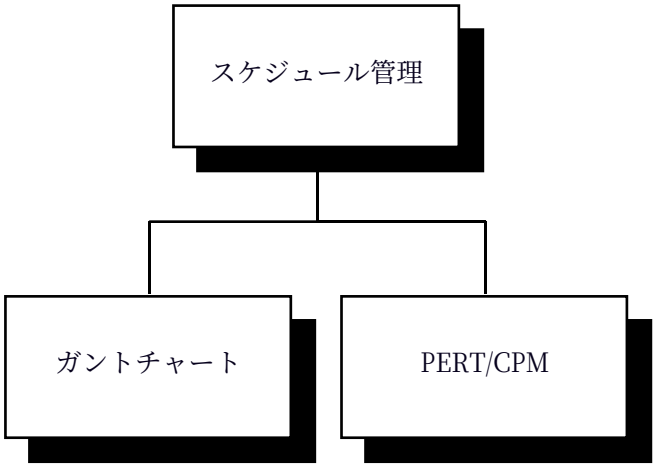
項目	値		
データ通信	0	存在しないか、影響無い	0
分散機能	0	ささいな影響	1
効率	2	並以下の影響	2
複雑な構成	1	平均的影響	3
トランザクションの割合	2	大きな影響	4
オンラインデータ入力	0	重大な影響	5
ユーザーの効率	5		
オンライン更新	0		
複雑な処理	2		
再利用性	5		
インストレーションの容易さ	4		
オペレーションの容易さ	5		
複数サイト	0		
変更の容易さ	5		
Total PC	31		
PCA	0.96		
FPA	20		

- Processing Complexityの考慮
  - FPA (Adjusted function point) 計算
    - PCA = 0.65 + (0.01 * PC)
    - FPA = FP * PCA

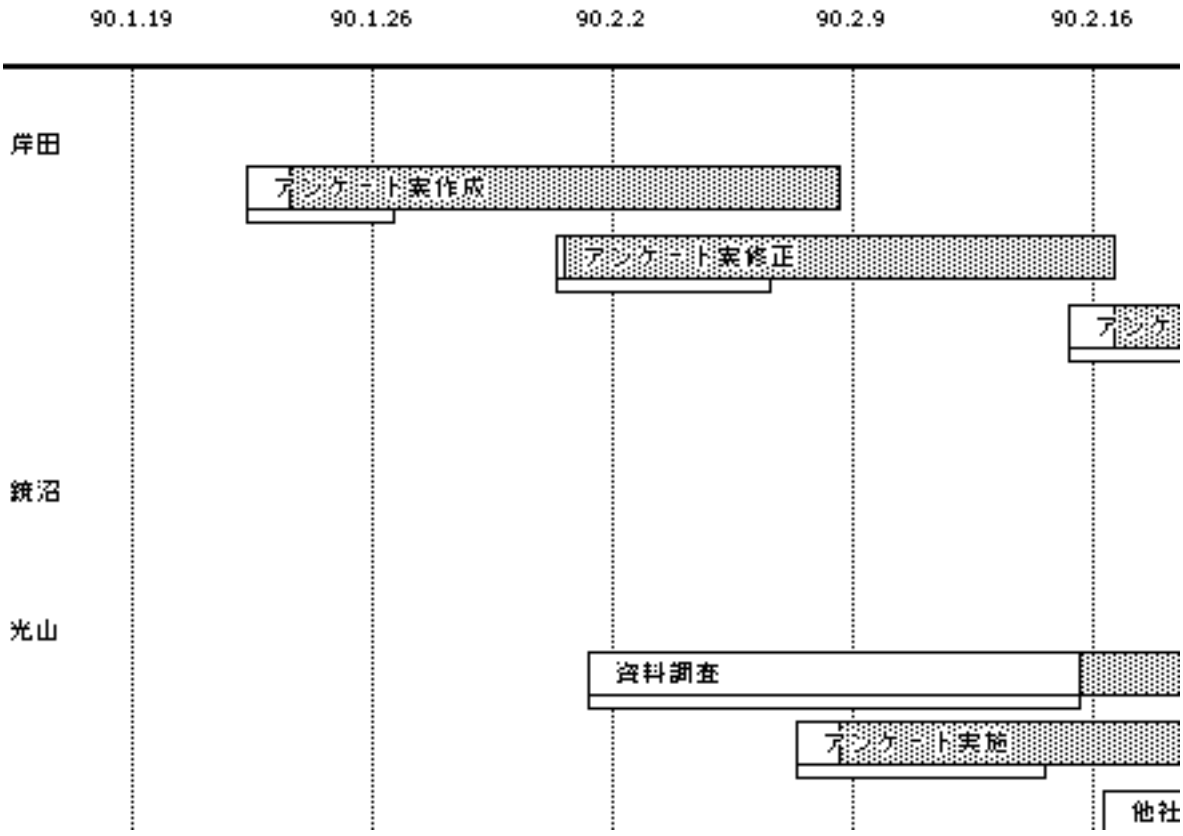
FPとコード行数の関係

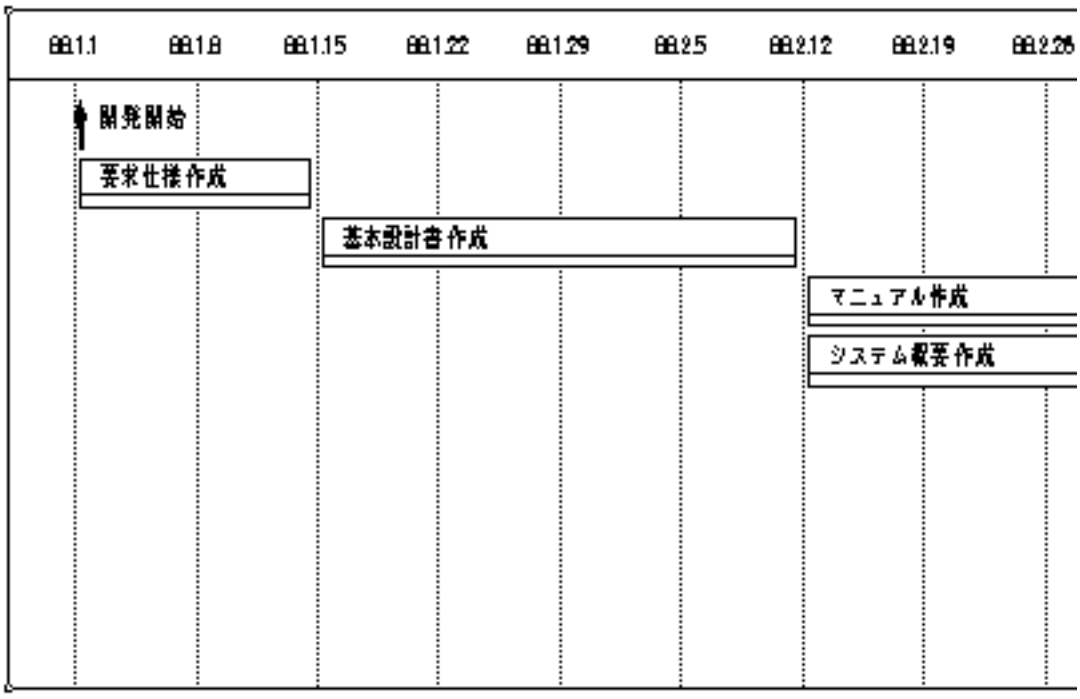
言語	コード行数/1FP
アセンブラ	32(
C	15(
COBOL	10(
FORTRAN	10(
Pascal	91
PL/I	8(
Ada	71
Prolog	64
APL	32
Smalltalk	21
スプレッドシート	6

スケジュール管理



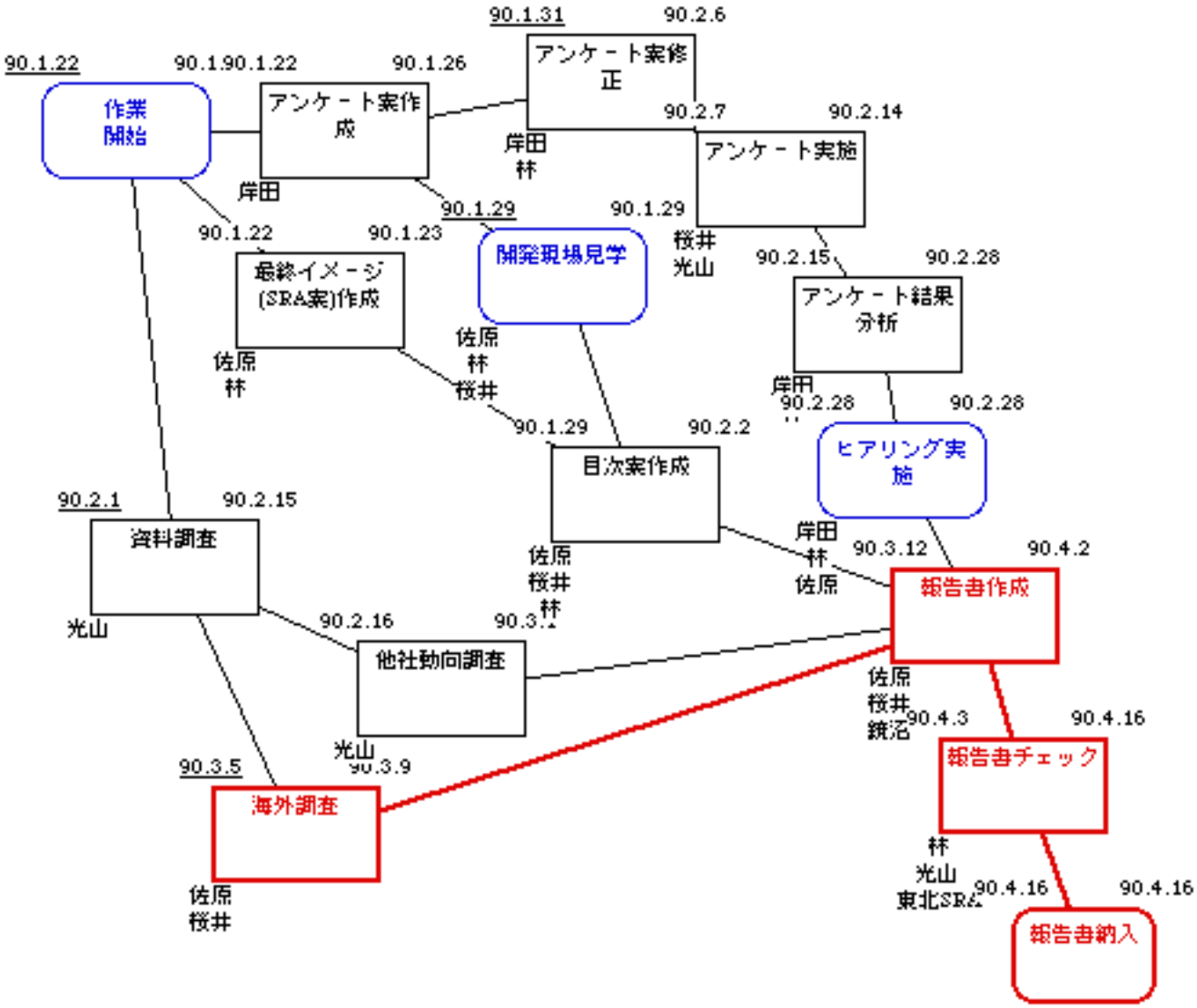
ガントチャート


90.1.19
90.1.26
90.2.2
90.2.9
90.2.16
岸田
アンケート案作成
アンケート案修正
アンケ
銃沼
光山
資料調査
アンケート実施
他社


88.1.1
88.1.8
88.1.15
88.1.22
88.1.29
88.2.5
88.2.12
88.2.19
88.2.26
開発開始
要求仕様作成
基本設計書作成
マニュアル作成
システム概要作成

47

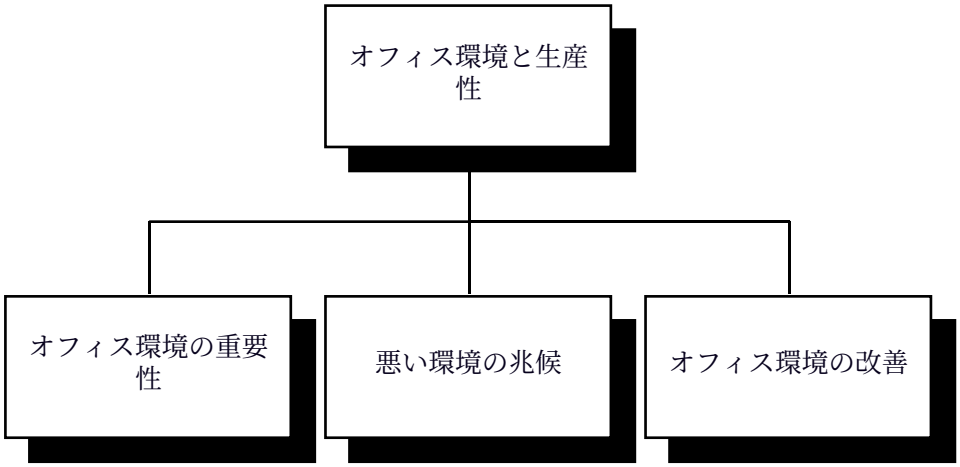
PERT/CPM


90.1.22
作業開始
90.1.90.1.22
アンケート案作成
岸田
90.1.26
90.1.31
アンケート案修正
岸田
林
90.2.6
90.2.7
90.2.14
アンケート実施
桜井
光山
90.1.22
最終イメージ(SRA案)作成
佐原
林
90.1.23
90.1.29
開発現場見学
佐原
林
桜井
90.1.29
90.2.15
90.2.28
アンケート結果分析
岸田
90.2.28
90.2.28
ヒアリング実施
岸田
林
佐原
90.1.29
90.2.2
目次案作成
佐原
桜井
林
90.2.1
90.2.15
資料調査
光山
90.2.16
90.3.1
他社動向調査
光山
90.3.9
90.3.5
海外調査
佐原
桜井
90.3.12
90.4.2
報告書作成
佐原
桜井
鏡沼
90.4.3
90.4.16
報告書チェック
林
光山
東北SRA
90.4.16
90.4.16
報告書納入

プロジェクトの計画とコントロール

- プロジェクトモデルの作成
- システムモデル（要求分析概観図）の作成
- Basic COCOMOによる評価
- 概要PERTチャートの作成
  - ガントチャートの生成
- システムモデル（要求分析）の作成
- WBS（Work Breakdown Structure）の作成
  - プロダクトWBS
  - 活動WBS
- Intermediate COCOMOによる評価
- 詳細PERTチャートの作成
- システムモデル（設計）の作成
- Detailed COCOMOによる評価
- 詳細PERTチャートの修正
- 繰り返し...

オフィス環境と生産性
(ドゥマルコ)



オフィス環境の重要性

環境要因	上位1/4グループ	下位1/4グループ
一人当たりのスペース	7㎡	4.5㎡
十分に静かか？	５７%	２９%
プライバシーは十分か？	６２%	１９%
電話の呼び出し音を消せるか？	５２%	１０%
電話を他へ転送できるか？	７６%	１９%
無意味な割込は多いか？	３８%	７６%

- プログラミングコンテストのデータから

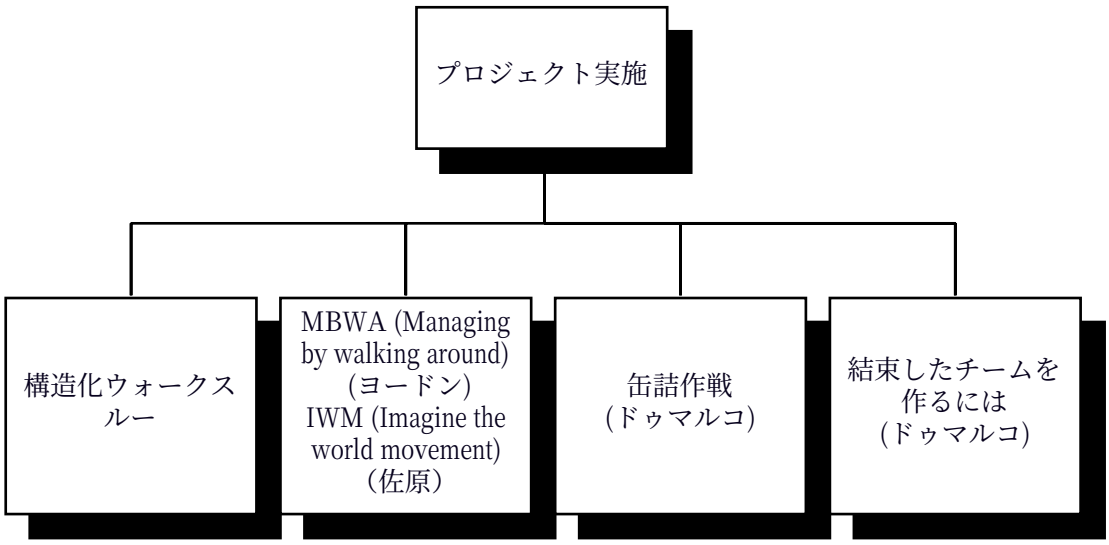
悪い環境の兆候

- プログラムは夜作られる
  - 夜になると仕事がはかどる
    - 会社の１週間分の仕事 = 自宅の１日分の仕事
- オフィスから雲隠れ
  - 会議室にこもる
  - 図書室にこもる
  - 喫茶店に行く
- 電話は鳴ってから３０秒以内に取れ
  - 電話による割込は、集中状態(フロー状態)を15分も邪魔する
    - 1時間に4回電話があると、仕事はほとんど進まない
- 机の上は、毎日きれいに整理して帰れ

オフィス環境の改善

- 頭脳労働の生産性
  - 机の前に何時間座っていたかでなく、全神経を集中して仕事に取り組んだ時間が重要
  - 肉体労働時間でなく、頭脳労働時間が重要
- 環境係数
  - 環境係数 = 割り込みなしの時間数 ／ 机の前に座っていた時間
    - 上限４０％ほどか？
    - 悪い環境 < 0.15
- IBMの調査
  - 最低限のオフィス環境
    - 一人当たり9㎡以上
    - 作業机2.7㎡以上
    - 騒音対策として、壁または1.8m以上の間仕切で空間を仕切る
- 理想のオフィス環境を得られなくても...
  - チームメンバーが自由にレイアウトする
  - 喫茶店を会議室にする
  - 喫茶店で本を読む
  - 会議室を作業場所にする
  - 自宅を作業場所にする

プロジェクト実施


プロジェクト実施
構造化ウォークスルー
MBWA (Managing by walking around)
(ヨードン)
IWM (Imagine the world movement)
（佐原）
缶詰作戦
(ドゥマルコ)
結束したチームを作るには
(ドゥマルコ)

構造化ウォークスルー

- 「ソフトウェアの構造化ウォークスルー」(Yourdon)近代科学社
- 基本的概念や意味的エラーの唯一の解決法
- 基本ルール
  - 作成者が主催する
  - エラー発見が目的なので、責任追及をしない
  - その場では修正しない
  - 具体的なデータを使い、トレースする
    - 机上デバッグの要領で
  - 評価をする管理者は参加させない
  - 短い時間で集中的に

MBWA (Managing by walking around)
(ヨードン)
IWM (Imagine the world movement)
(佐原)

- 歩き回って、事実を見る
  - 昼も夜もいろいろな時間帯に、オフィスや茶飲み場や喫煙場所など、ソフトウェア技術者がいそうな所はどこにでも表れる
  - こけたプロジェクトに働いている人達は、初めからこけるのが分かっている

缶詰作戦
(ドゥマルコ)

- 会社を離れた場所に会議室を借り、そこを作業場所にする
  - スキー場や海辺のホテルのシーズンオフ料金を利用する
- 何かの会議に参加させ、そのついでに２～３日仕事をさせる
- 成果物でチェックする

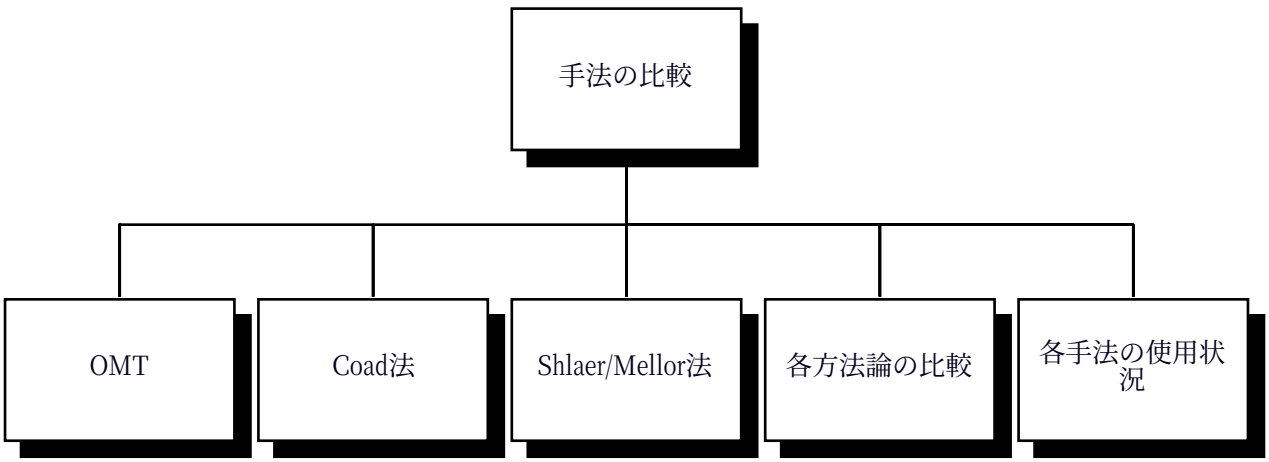
結束したチームを作るには(ドゥマルコ)

- 品質至上主義
  - 品質を追求した方が生産性は高くなる
- 満足感を与える打ち上げをたくさん用意する
  - 小さな成果があがるたびに、何か打ち上げを用意する
- エリート感覚を醸成する
  - 管理者にとって居心地がよければよいほど、チームの活力は弱まる
  - チームの構造はネットワーク
- チームに異分子を混ぜることを奨励する
- 成功チームを解散させないで保護する
- 戦術でなく戦略を与える
- チームメンバーの公募

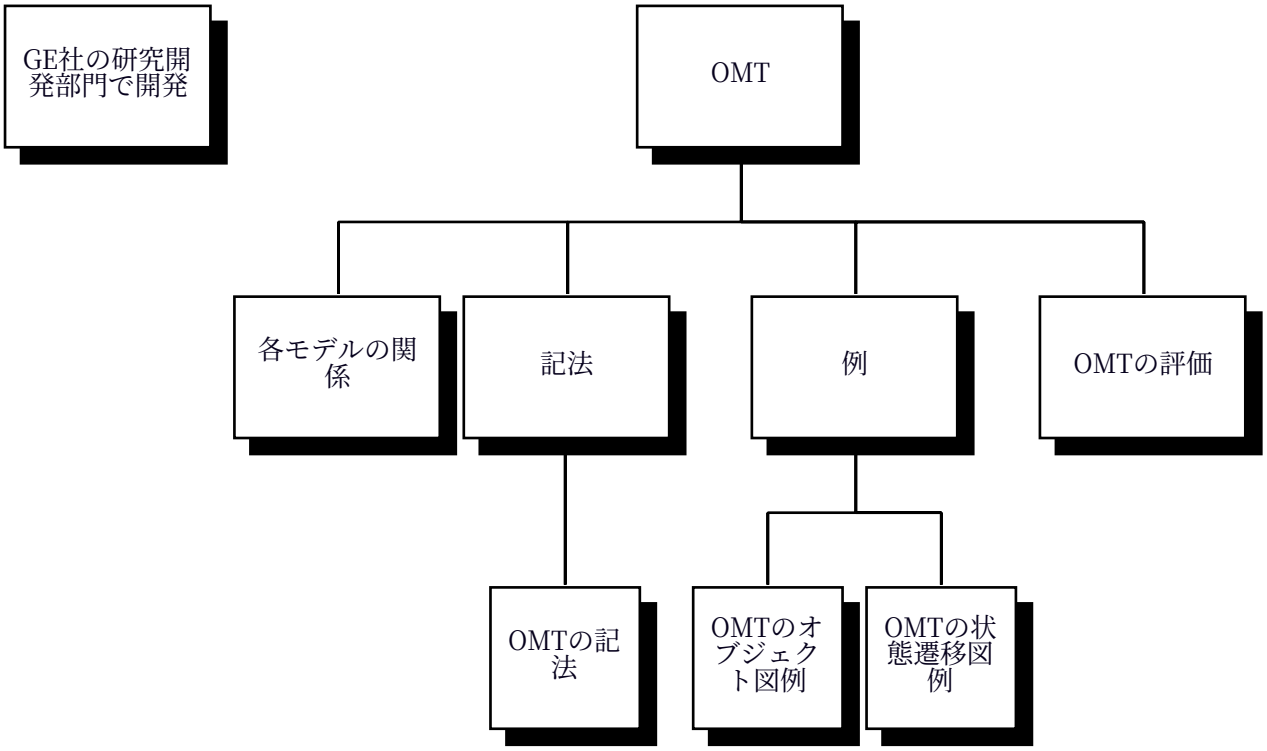
参考文献

- プロジェクト開発の人間的側面
  - ドゥマルコ、リスター．ピープルウェア．日立ソフトウェアエンジニアリング生産性研究会訳．日経BP、1989年
  - ワインバーグ．スーパーエンジニアへの道－技術リーダーシップの人間学．木村泉訳．共立出版、1991年
  - エドワード・ヨードン．ソフトウェア管理の落し穴:アメリカの事例に学ぶ．松原友夫訳．トッパン、1993年
  - アンドリュー S. グローブ．ワン・オン・ワン 快適人間関係を作るマネジメント手法．小林薫監訳．パーソナルメディア、1990年
- ソフトウェア工学全般
  - エドワード・ヨードン．ソフトウェア管理の落し穴:アメリカの事例に学ぶ．松原友夫訳．トッパン、1993年
  - T.G. Lewis．CASE: Computer-Aided Software Enginnering．Van Nostrand Reinhold、1991
  - B.W. Boehm．Software Engineering Economics. Prentice-Hall、1981
- COCOMO
  - B.W. Boehm．Software Engineering Economics. Prentice-Hall、1981
- ファンクションポイント
  - ドゥマルコ．品質と生産性を重視したソフトウェア開発プロジェクト技法 － 見積り・設計・テストの効果的な構造化．渡辺純一訳．近代科学社、1987.
- その他
  - Frederick P. Brooks Jr.．ソフトウェア開発の神話．山内訳．企画センター、1977年
  - フリードマン, ワインバーグ．ソフトウェア技術レビューハンドブック．岡田正志監訳．TBS出版会、1987年
  - アルヴィン D. クックス．ノモンハン．岩崎俊夫、吉本晋一郎訳．朝日新聞、1989年

手法の比較

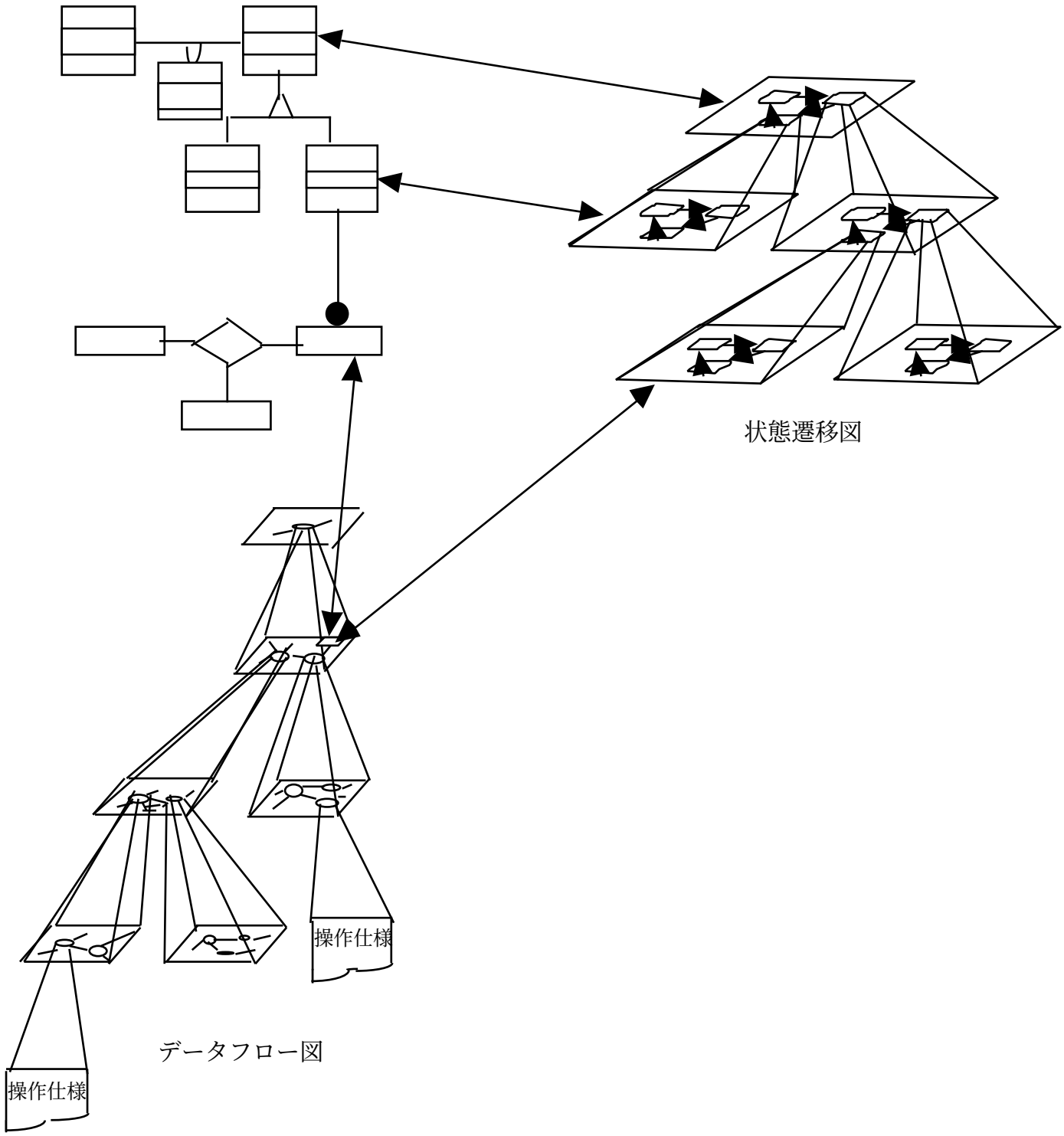


OMT


GE社の研究開発部門で開発
OMT
各モデルの関係
記法
例
OMTの評価
OMTの記法
OMTのオブジェクト図例
OMTの状態遷移図例

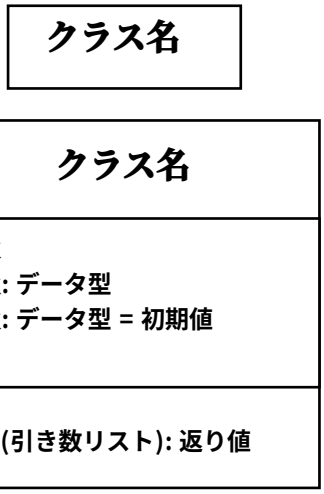
各モデルの関係

オブジェクト図

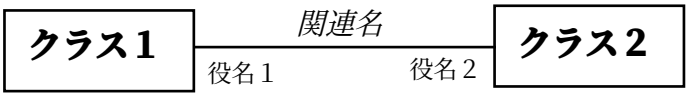


OMTの記法

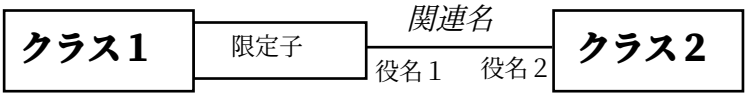
クラス



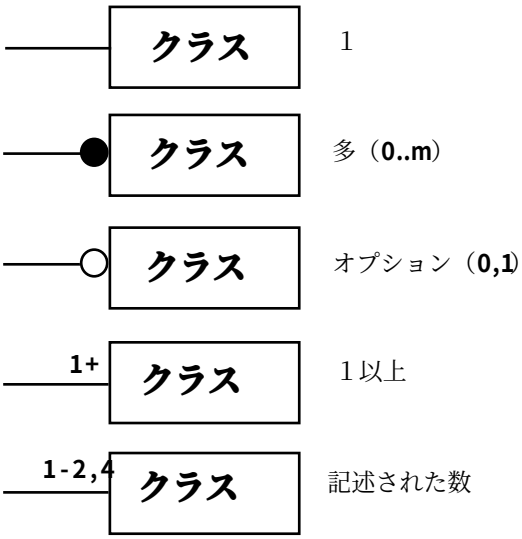
関連



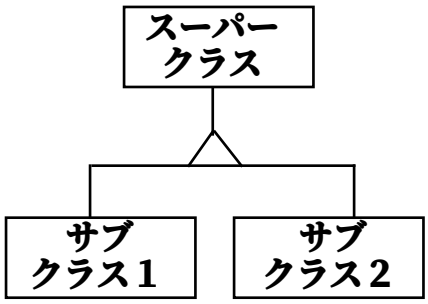
限定付き関連



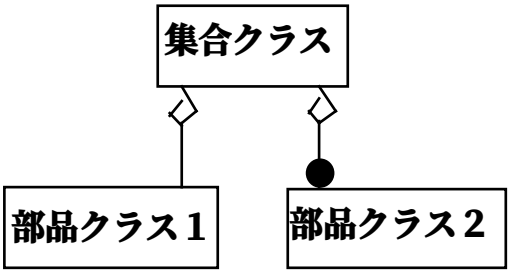
関連の多重度



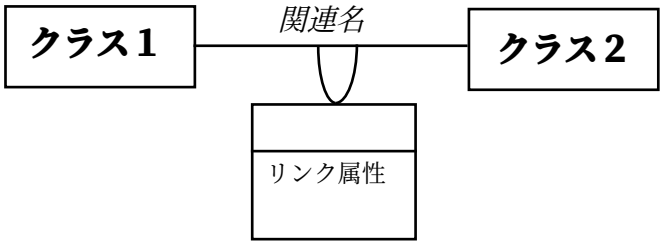
汎化（継承）



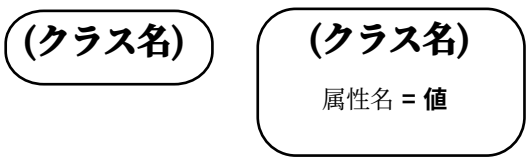
集約



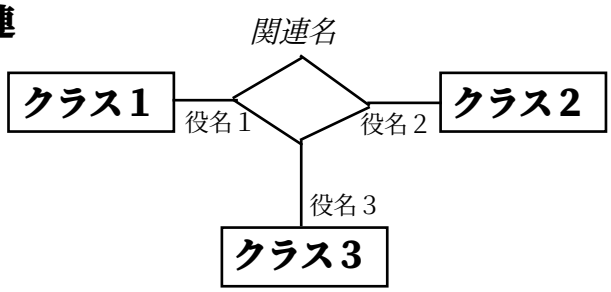
リンク属性



オブジェクト　インスタンス

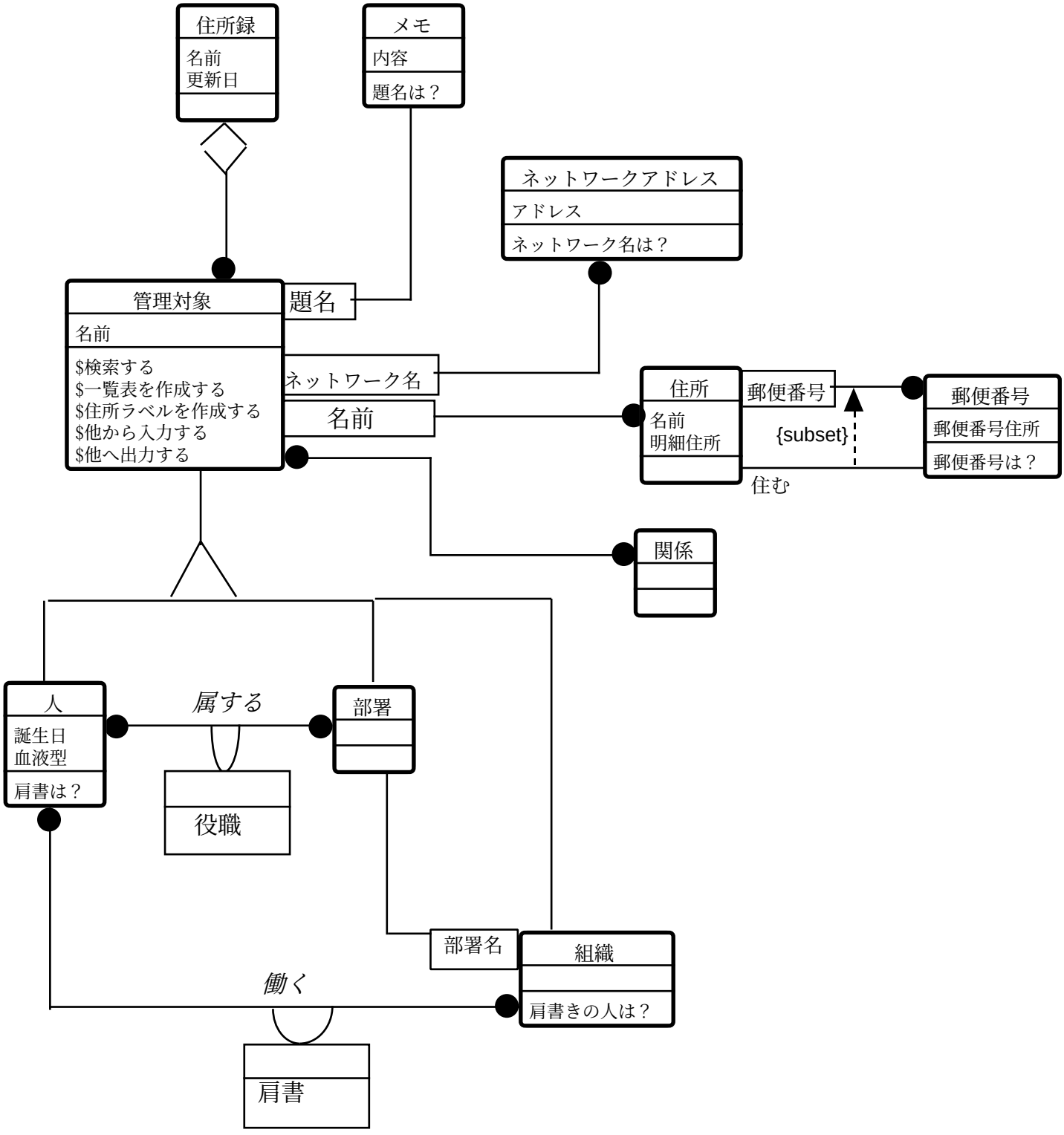


３項関連

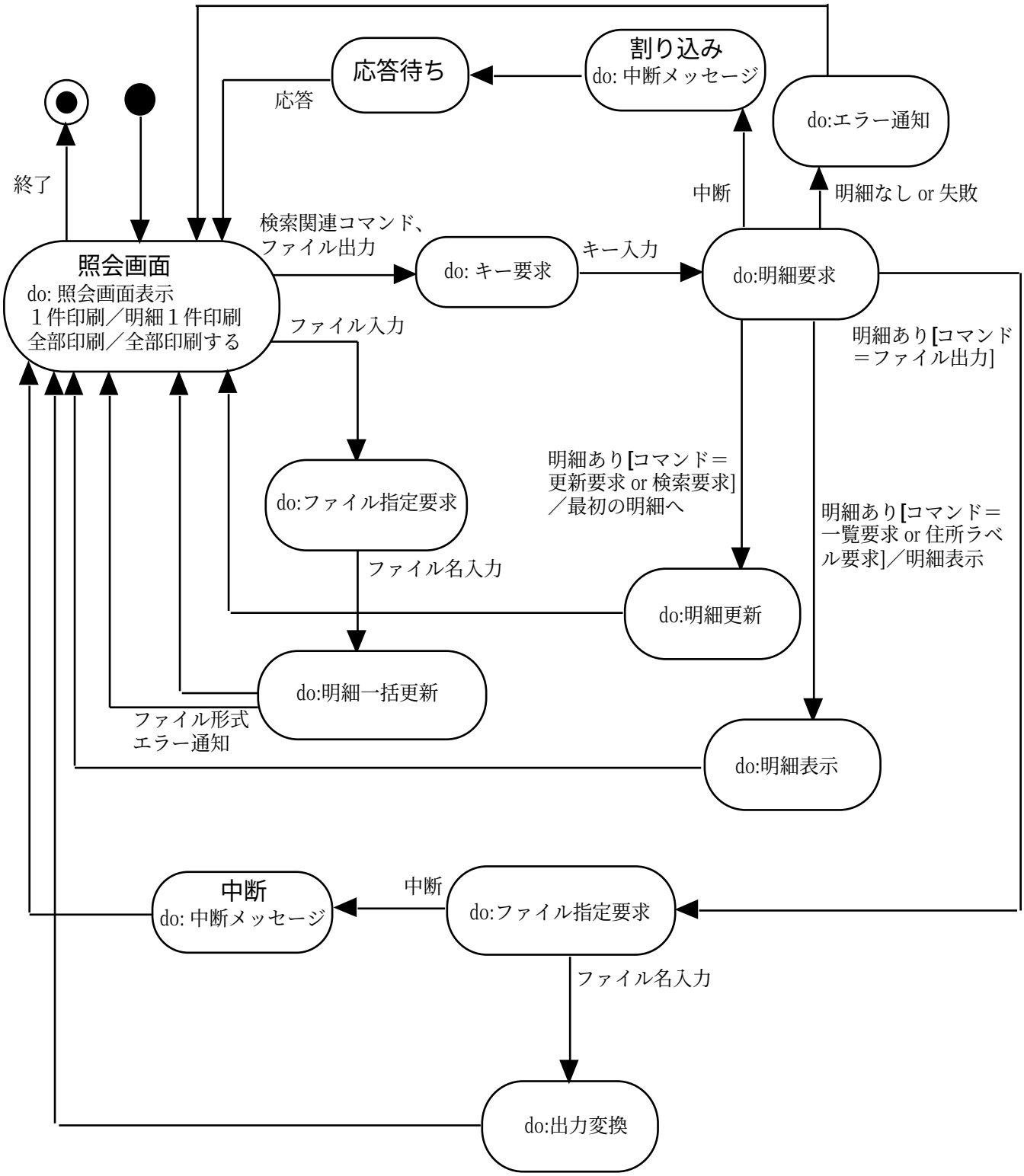


Source: Runmbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. Object-Oriented Modeling and Design. Prentice Hall, 1991

OMTのオブジェクト図例



OMTの状態遷移図例

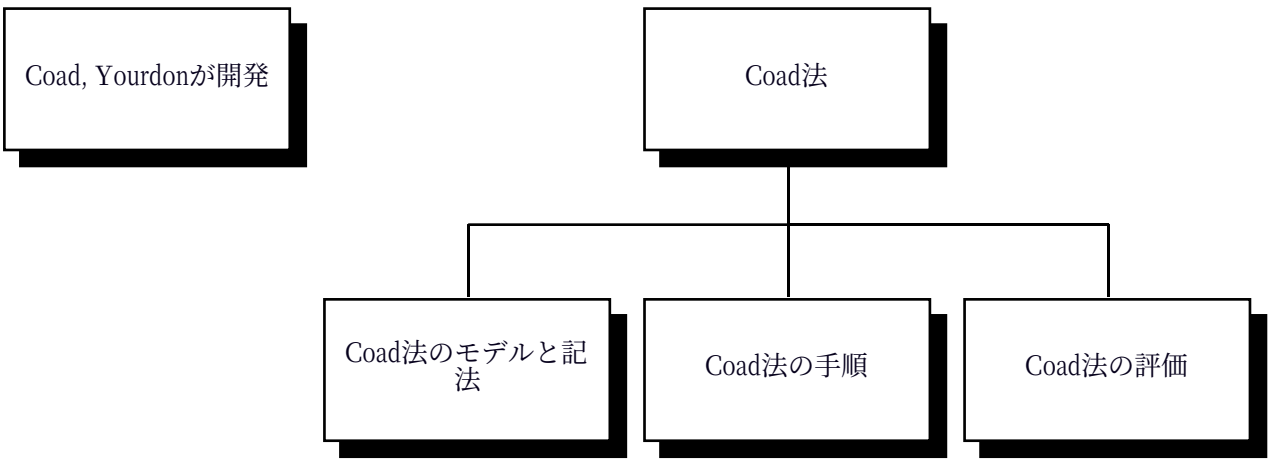


検索関連コマンド＝
　更新要求、検索要求、一覧要求、住所ラベル要求
応答＝
　明細あり、明細なし、失敗

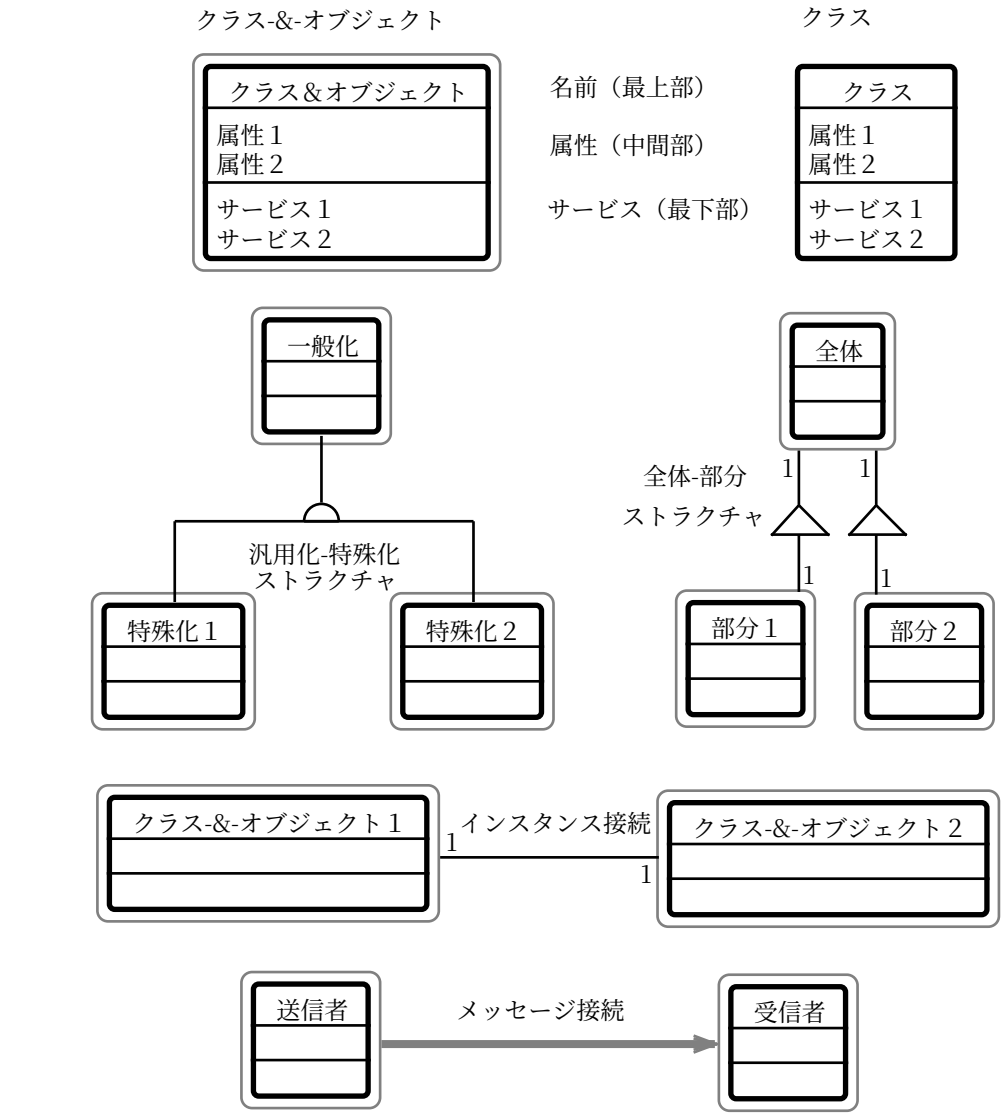
OMTの評価

- OMTの特徴は、分析から設計まで一貫したオブジェクトモデルで記述でき、分析手順と設計手順と実現手順とが、かなり詳細に分かりやすく用意されていることである。
  - オブジェクトモデルはBooch法よりは簡単で、Coad法より記述能力が高く、どちらの方法よりも関連に関しての記述力が高い。
- OMTで使用するモデルは言語独立性が高く、利用可能な分野が広い。
  - 実際に種々の言語で、いろいろな分野のアプリケーションを作成した経験が感じられる。
  - また、従来のSA/SD手法のE-R図をオブジェクト図に変えればOMTになるので、SA/SDからOMTに移る障害が少ない。
- 現時点で、もっとも完成されたOOA/OODの方法論である。
- あえて問題点を捜せば、
  - 既存のライブラリーとオブジェクト指向のライブラリーを混合した場合、Booch法やOOSDのようにそれを直接的に表すことができない
  - 既存のシステムを分析する場合にも、モジュールの呼び出し関係などを直接的に表せない。

Coad法



Coad法のモデルと記法



Source: Coad, Peter, Yourdon, Edward. Objet-Oriented Analysis. Yourdon Press, Prentice Hall, 1990

- **Coad法はオブジェクト図とも呼ぶべき図と、状態遷移図とフローチャートに近いサービスチャートでモデルを構築する。**
- **　状態遷移図とサービスチャートは、オブジェクトの内部のサービスの定義に使用する。**

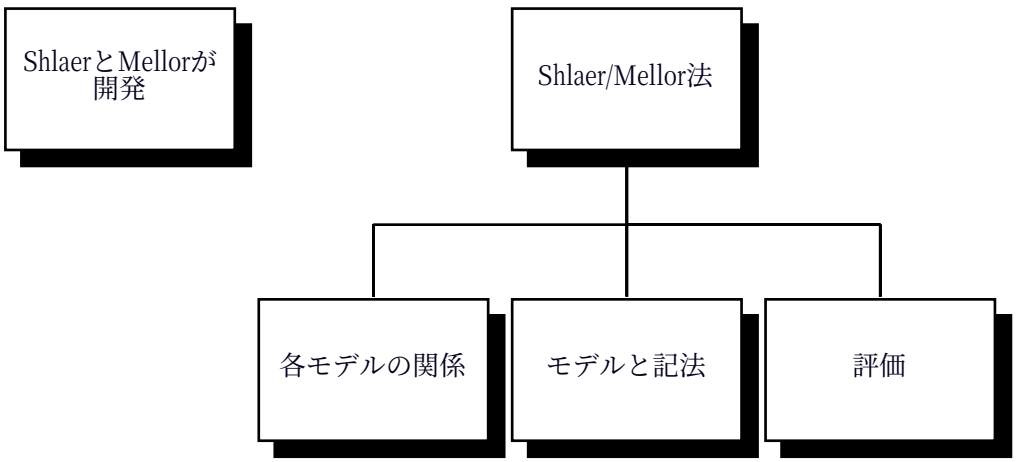
Coad法の手順

- Coad法の手順は大きく以下の５つに分かれ、それぞれかなり詳細に標準の手順が決められている。
  - オブジェクトの認識
  - 構造の認識
    - 一般化-特殊化構造、全体-部分構造などを認識する。そのためのノウハウも提供されている。
  - サブジェクトの認識
    - サブジェクトをいつ、どうやって選び、洗練するかの指針がある。
  - 属性の認識
    - 属性の抽出、属性の配置、インスタンス接続の認識、属性とインスタンス接続の必要性のチェック、属性の制約の記述などの指針がある。
  - サービスの認識
    - オブジェクトの状態の認識、状態遷移図の作成、サービスの抽出、メッセージ接続の認識、サービスの記述、サービスチャートの作成、ドキュメント化などの指針がある。

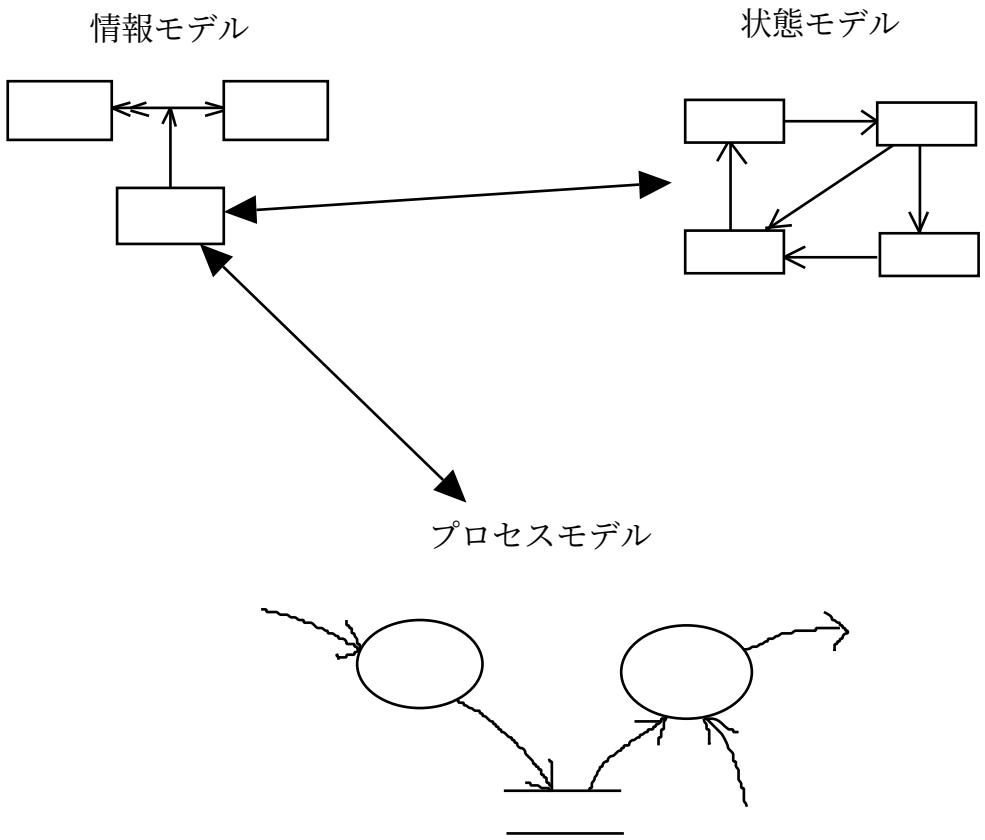
Coad法の評価

- Coad法では、オブジェクトの構造と挙動の両方の分析方法が示されている。
  - オブジェクトのかたまりとしてサブジェクトを置くことによって、概観の把握が容易になっている。
  - また、記法が簡単な割に有用性が高い。
- 一方、分析方法は割合しっかりしているが、挙動面の分析はまだ弱く、設計の方法論は詳細とは言えない。
  - Smalltalkなど、詳細設計にあまり留意する必要のない言語を使う場合有用である
  - C++, Object Pascalなどでは、詳細設計に適した記法と方法論で補強する必要があろう。

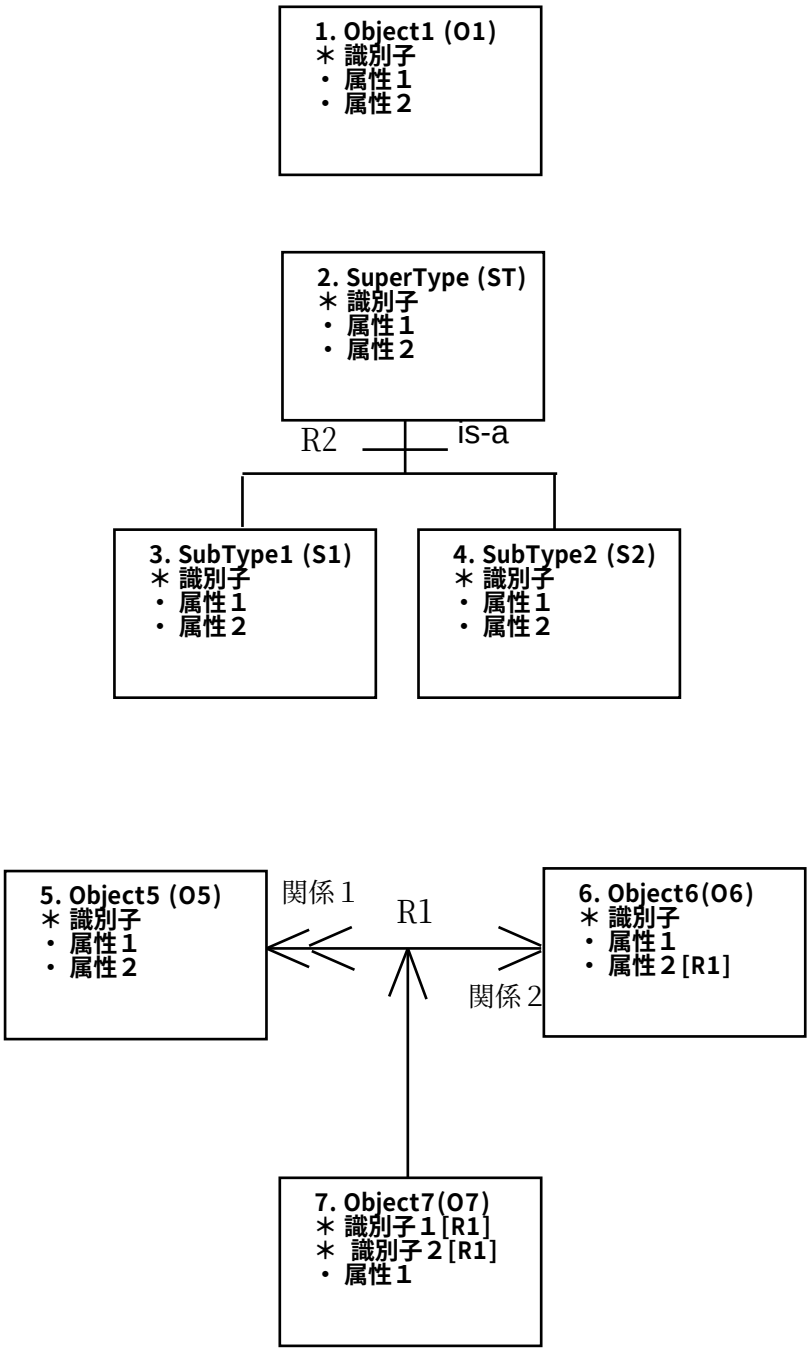
Shlaer/Mellor法



各モデルの関係



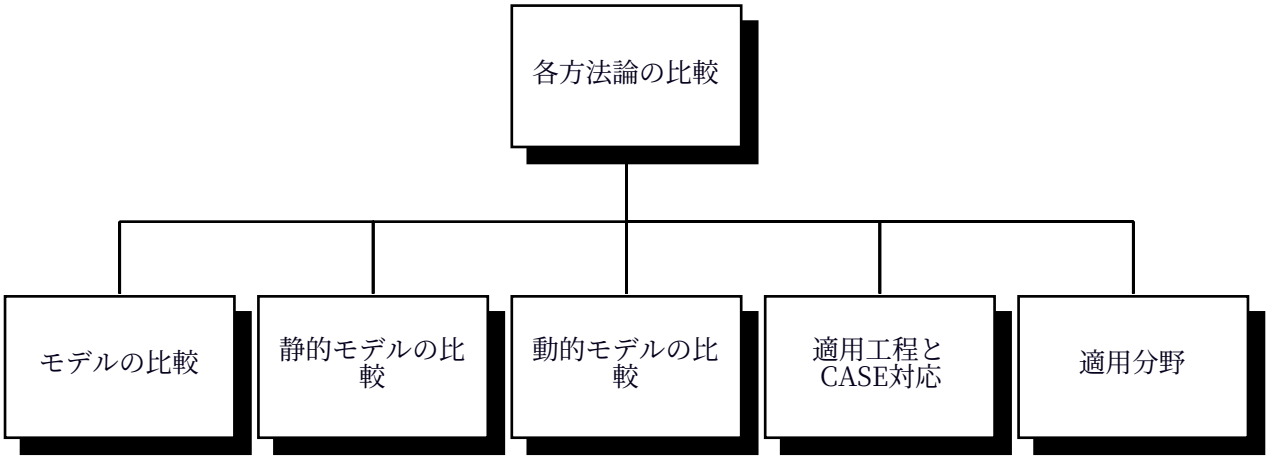
モデルと記法


1. Object1 (O1)
＊ 識別子
・ 属性１
・ 属性２
2. SuperType (ST)
＊ 識別子
・ 属性１
・ 属性２
R2
is-a
3. SubType1 (S1)
＊ 識別子
・ 属性１
・ 属性２
4. SubType2 (S2)
＊ 識別子
・ 属性１
・ 属性２
5. Object5 (O5)
＊ 識別子
・ 属性１
・ 属性２
関係１
R1
関係２
6. Object6(O6)
＊ 識別子
・ 属性１
・ 属性２[R1]
7. Object7(O7)
＊ 識別子１[R1]
＊ 識別子２[R1]
・ 属性１

評価

- Coad法とOMTの元になった手法であり、歴史的価値は大きい。
  - 状態モデルとプロセスモデルを、後から追加し実用的になった。
- OMTで整理された概念的混乱が残っている。
  - 実現工程の考え方が分析工程に影響する。
    - 正規化
    - 識別子
    - 関係を表す属性
- 分析と設計で異なる記法を使用するため、書かねばならない図の数が増える。
- 分析工程でも、Booch法ほどではないが、各モデル内のダイヤグラムの種類が多いため、複雑である。
- 言語独立で実用的。
- SA/SDのE-R図を情報モデルの記法に変換すればShlaer/Mellor法になるので、SA/SDからShlaer/Mellor法に移る障害が少ない。
- 手順が明示的には示されていないので、初心者には使いにくい。
- OMTに次ぐ完成度の手法である。

各方法論の比較



モデルの比較

モデル	OMT	OOA/OOD	OOSA	OOSE
	Rumbaugh	Coad/Yourdon	Shlaer/Mellor	Yacobson
静的	オブジェクトモデル(SDM)	クラス-オブジェクト図(SDM)	情報モデル(SDM)	ドメインオブジェクトモデル、分析モデル
動的	動的モデル(STD)	状態遷移図(STD)	状態モデル(STD)	状態遷移グラフ(STD)
機能	機能モデル(DFD)	サービスチャート	アクションフロー図(DFD)	
その他	イベントフロー図		オブジェクトコミュニケーションモデル、オブジェクトアクセスモデル	インタラクション図
特殊な設計モデル			OODLE	

静的モデルの比較

	OMT	OOA/OOD	OOSA	OOSE
	Rumbaugh	Coad/Yourdon	Shlaer/Mellor	Yacobson
オブジェクト	○	○	○	
クラス	○	○		○
抽象クラス		○	?	
クラスの種類		△		○
属性	○	○	○	○
関係	○	○	○	○
汎化／特殊化	○	○	○	○
集約	○	○		

動的モデルの比較

	OMT	OOA/OOD	OOSA	OOSE
	Rumbaugh	Coad/Yourdon	Shlaer/Mellor	Yacobson
状態	○	○	○	○
遷移	○	○	○	○
トリガー	○		○	○
動作	○		○	○
ガード	○			○
活動	○		○	
並行性	○		○	?
シナリオの強調	△		△	○

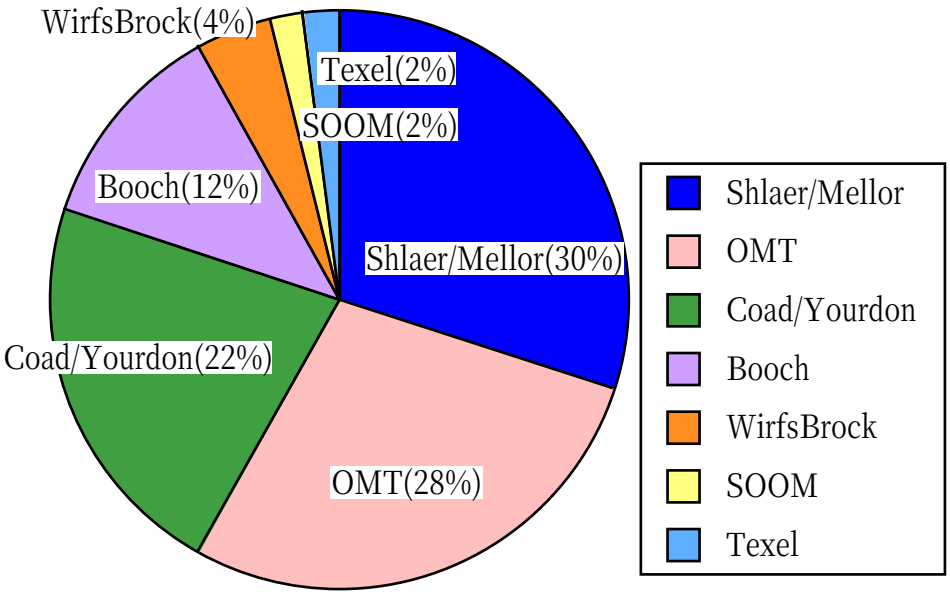
適用工程とCASE対応

活動	OMT	OOA/OOD	OOSA	OOSE
	Rumbaugh	Coad/Yourdon	Shlaer/Mellor	Yacobson
プロジェクト管理				△
分析	○	○	○	○
設計	○	○	○	○
実装	○	○		△
テスト				△
保守				△
ドキュメント作成	△	△	○	△
CASE対応	○(数種)	○(数種)	○(数種)	△(Objectory)
UI作成	?			△

適用分野

	OMT	OOA/OOD	OOSA	OOSE
	Rumbaug	Coad/Yourdon	Shlaer/Mellor	Yacobson
リアルタイムシステム	◎		◎	◎
ハードなリアルタイムシステム	○		◎	◎
制御系システム	○		○	○
CASEシステム	◎	○	○	○
情報システム	◎	○	○	○
事務処理システム	◎	○	○	○

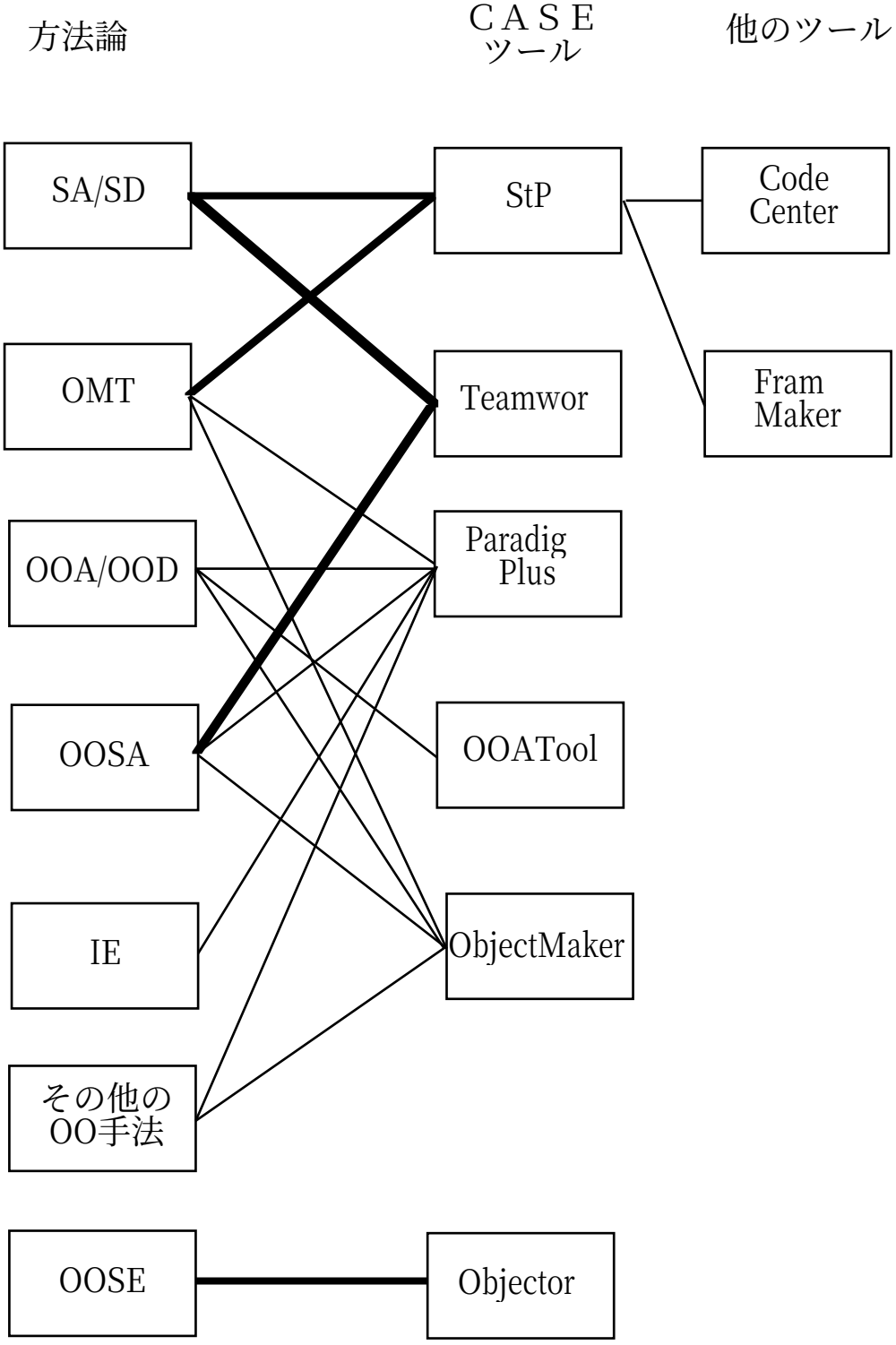
各手法の使用状況


WirfsBrock(4%)
Texel(2%)
SOOM(2%)
Booch(12%)
Shlaer/Mellor(30%)
Coad/Yourdon(22%)
OMT(28%)
Shlaer/Mellor
OMT
Coad/Yourdon
Booch
WirfsBrock
SOOM
Texel

方法論の選択

- どの工程に問題があるのか？
  - 分析
  - 設計
  - 実現
- 誰が担当するのか？
  - ソフトウェア技術者
  - 問題領域の専門家
- 開発環境は何か？
  - 分散ネットワーク＋ＷＳ
  - ＰＣ
  - メインフレーム
- どの言語を使用するのか？
  - Smalltalk
  - C++
  - Objective-C
  - CLOS
  - Eiffel
  - RDB
- 何年後を目指すのか？
  - １年後
  - ３年後
  - ５年後

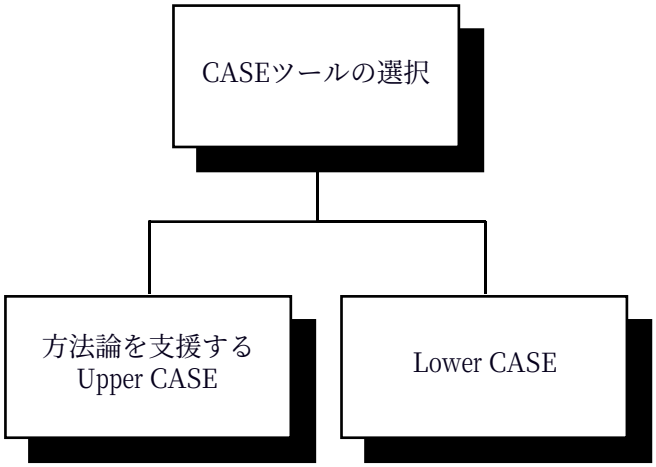
CASEツールとの関係



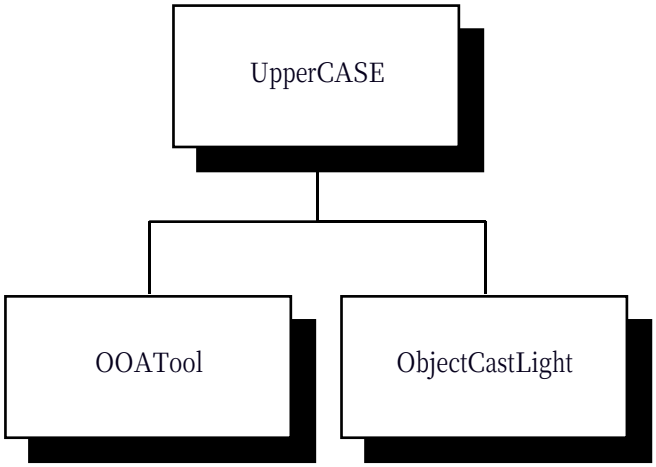
参考文献(OOA/OOD)

- J.Rumbaugh, M. Blaha , W.Premerlani, F.Eddy, and W.Lorensen. 羽生田訳. オブジェクト指向方法論：ＯＭＴ. トッパン, 1992
  - 現時点で最も完成されたOOA/OOD技法OMTの教科書
- J.Rumbaugh, M. Blaha , W.Premerlani, F.Eddy, and W.Lorensen. Solution Manual Object-Oriented Modeling and Design. Prentice Hall, 1991
  - 上の本の解答集
- 青木淳. オブジェクト指向システム分析設計入門. ソフト・リサーチ・センター, 1992
  - OOA/OOD/OOPの分かりやすい解説
- 青木淳著. 例題による!!オブジェクト指向分析設計テクニック. ソフト・リサーチ・センター, 1994
  - OOA/OOD/OOPの例題を豊富に使った解説
- 青木淳著. 訳. OMTによるオブジェクト指向システム分析設計実践口座. SRAセミナー資料, 1995年1月11日
  - 方法論の詳細な比較表
- Peter Coad, Edward Yourdon. 羽生田 監訳. オブジェクト指向分析（ＯＯＡ）［第２版］. トッパントッパン, 1993
  - OOA/OODの教科書
- Peter Coad, Edward Yourdon. Objet-Oriented Design. Prentice Hall International, 1991
  - OODの教科書
- Grady Booch. Object-Oriented Design With Applications. Benjamin/Cummings, 1990
  - AdaよりのOOD手法
- Sally Shlaer, Stephen J. Mellor. 本位田真一, 山口享 訳. オブジェクト指向システム分析　上流CASEのためのモデル化手法. 啓学出版, 1990
  - OMT, Coad法の元になった手法の教科書
- Sally Shlaer, Stephen J. Mellor. 本位田真一, 伊藤潔 監訳. 続・オブジェクト指向システム分析　オブジェクトライフサイクル. 啓学出版, 1992
  - 上の本の続編。状態モデルに焦点を当てている。
- Michael A. Jackson. 大野, 山崎 監訳. システム開発JSD法. 共立出版, 1989
  - JSD法の教科書

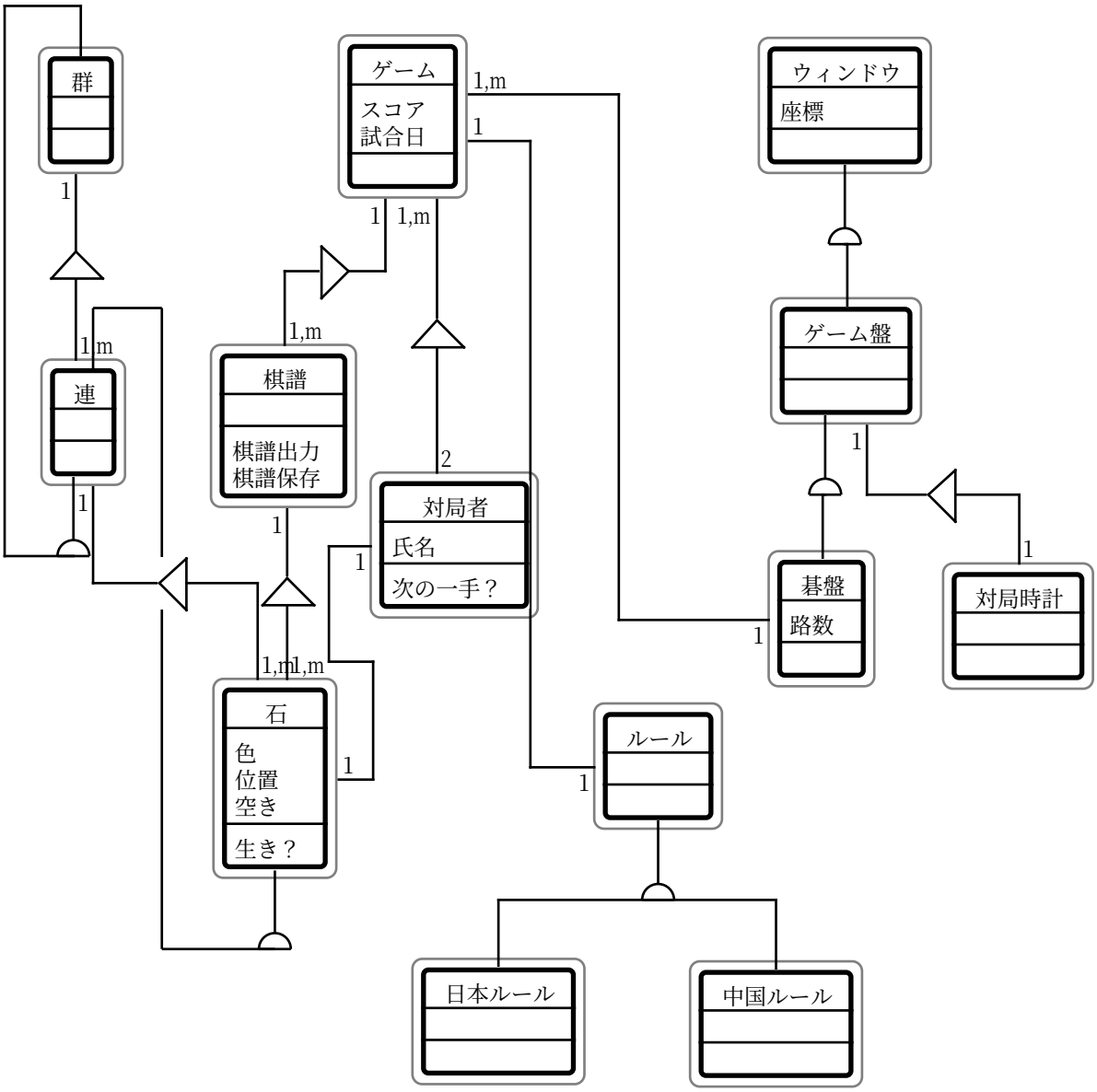
CASEツールの選択



方法論を支援するUpper CASE



OOATool

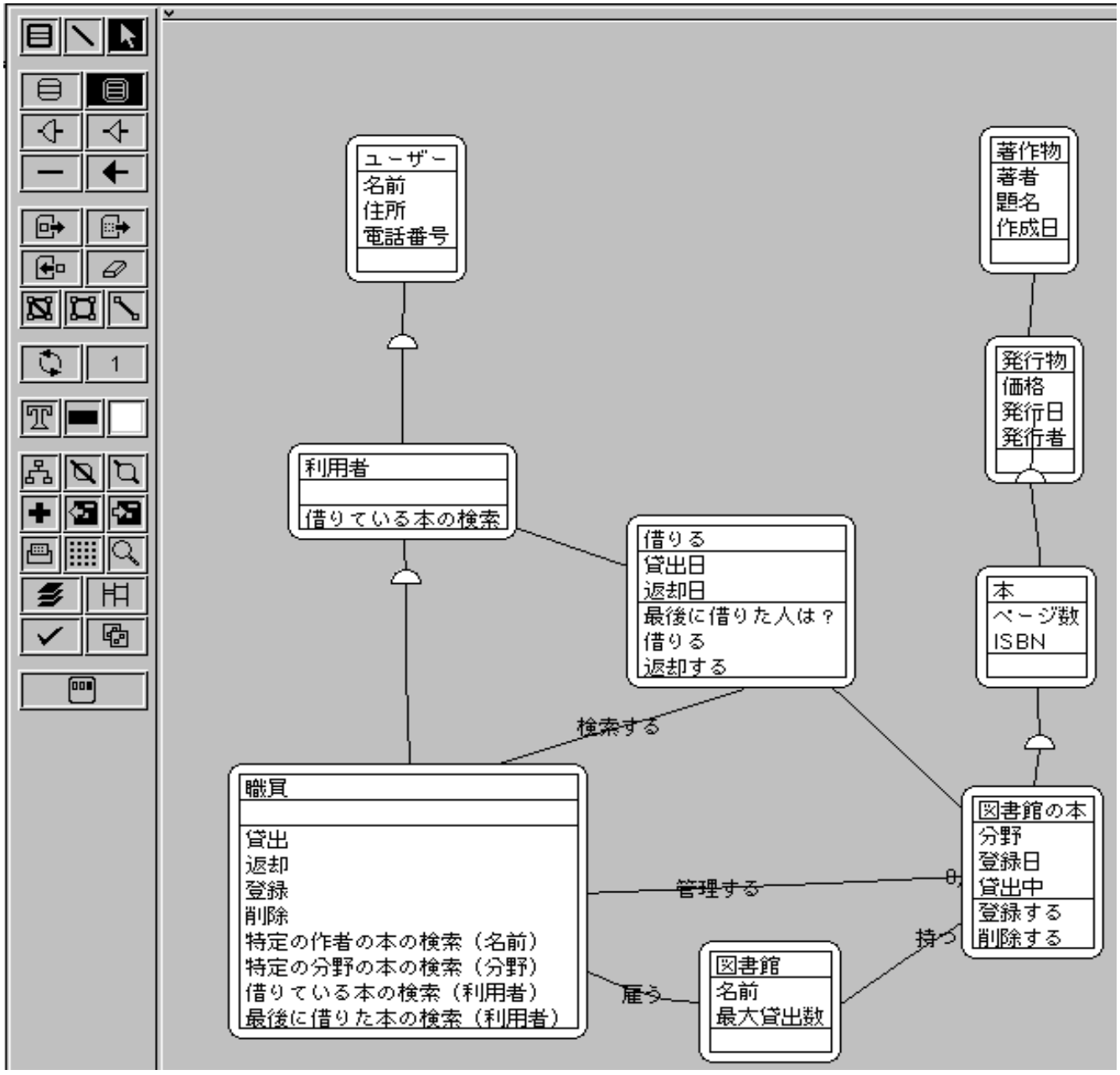


- **Object International社の、Coad法を支援するツールである。Macintosh上で稼働する。**
- **OOAToolの内容***

OOATool の内容*

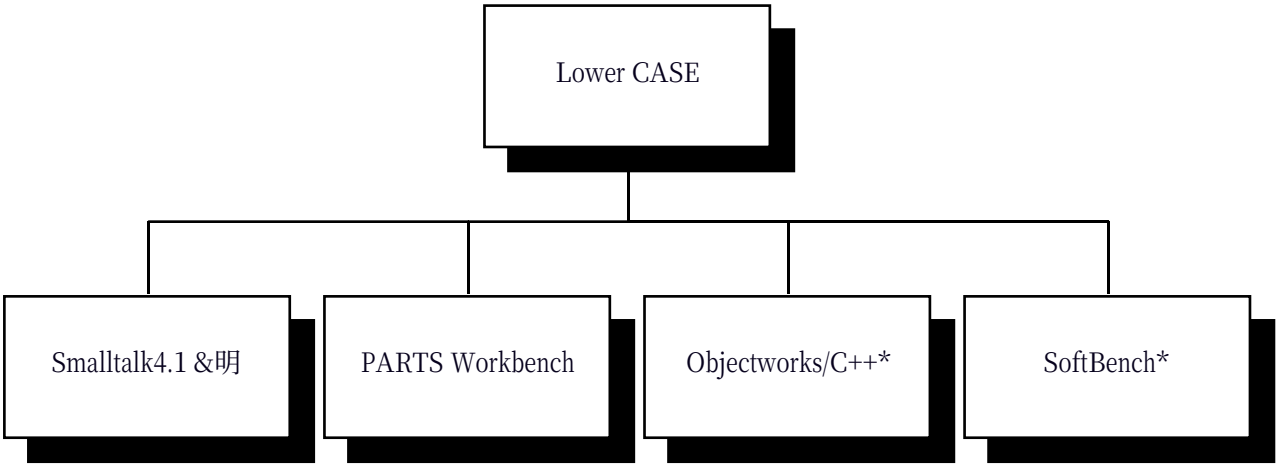
- Coad法を全面的に支援しているが、CASEツールとして付加された主な機能は、以下のとおり。
  - 評論機能
  - フィルター機能
  - ドキュメント支援機能
  - 戦略カード機能
    - Coad法の方法論を工程別に表示する。
- OOATool はCoad法全体を支援するツールであり、Smalltalk/Vのエディターや環境を流用して高機能なツールとなっている。
  - ただし、Smalltalk/Vの環境にはアクセスできない。
- ユーザーインタフェースは良い方であり、オブジェクト指向ユーザーインタフェースを採用している
  - Macintoshのアプリケーションとしては良くない

ObjectCastLight



- **富士ゼロックス情報システム社のCoad法を支援するツールである。**

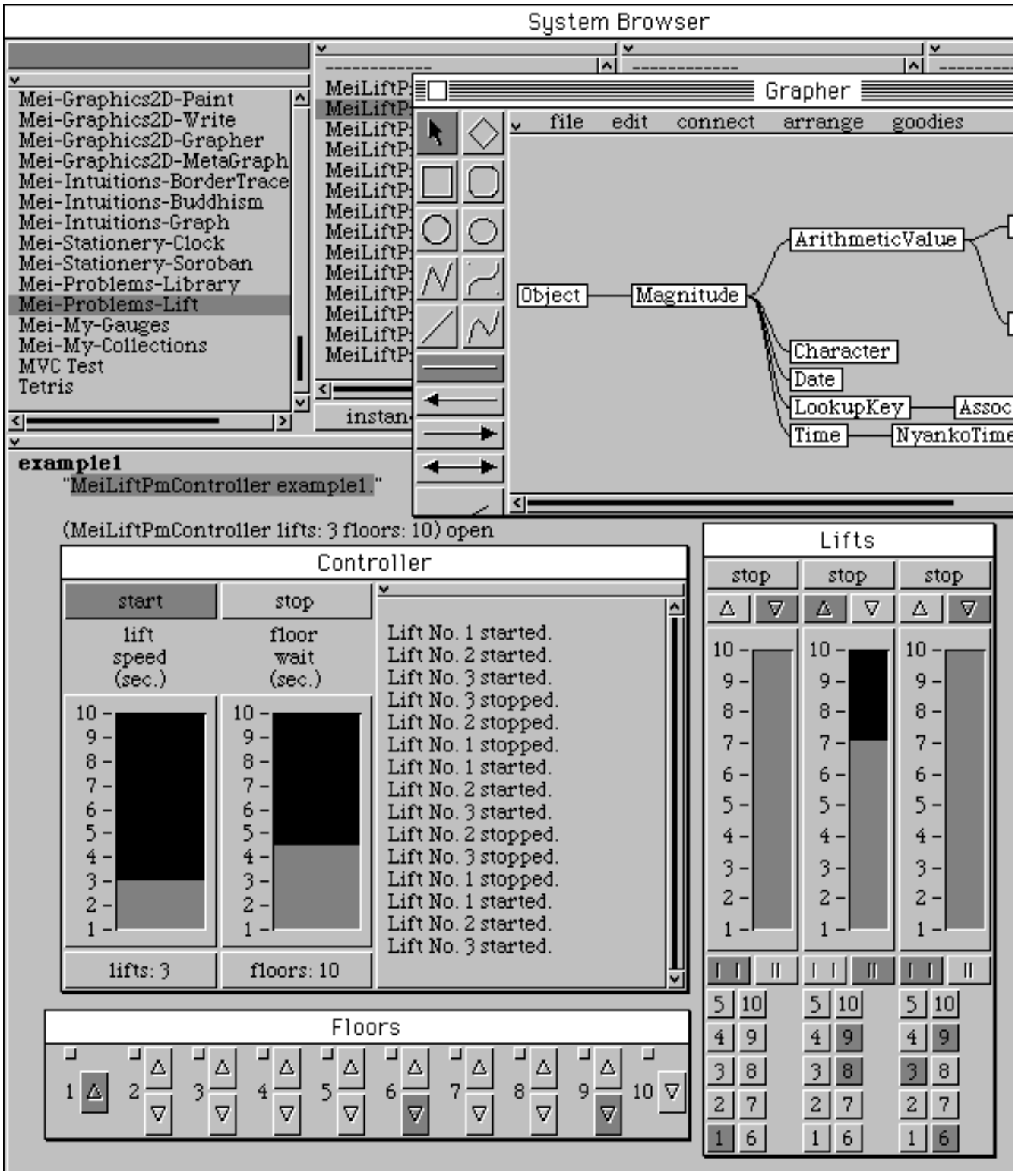
Lower CASE


Lower CASE
Smalltalk4.1 &明
PARTS Workbench
Objectworks/C++*
SoftBench*

Smalltalk4.1 &明

- ParcPlace Systems社のSmalltalk支援環境
- 明は、Objectworks/Smalltalk上に構築されたフリーのSmalltalk支援環境
  - 明は、本体のObjectworks/Smalltalkよりも多くのクラスとステップ数からなる巨大な支援環境と再利用可能なオブジェクトライブラリーの集合
  - ユーザーインタフェースは、Objectworks/Smalltalkオリジナルのものも使用できるが、普通はすべて明流に改造されている
- 画面例*
- Smalltalk4.0 & 明の内容*
- 全体として、Smalltalkの開発・保守支援環境として十分すぎるほどの機能を持っている
  - チームによる協同作業を支援する機能が弱い

画面例*


System Browser
Mei-Graphics2D-Paint
Mei-Graphics2D-Write
Mei-Graphics2D-Grapher
Mei-Graphics2D-MetaGraph
Mei-Intuitions-BorderTrace
Mei-Intuitions-Buddhism
Mei-Intuitions-Graph
Mei-Stationery-Clock
Mei-Stationery-Soroban
Mei-Problems-Library
Mei-Problems-Lift
Mei-My-Gauges
Mei-My-Collections
MVC Test
Tetris
Grapher
file edit connect arrange goodies
Object
Magnitude
ArithmeticValue
Character
Date
LookupKey
Time
example1
"MeiLiftPmController example1."
(MeiLiftPmController lifts: 3 floors: 10) open
Controller
start
stop
lift speed (sec.)
floor wait (sec.)
Lift No. 1 started.
Lift No. 2 started.
Lift No. 3 started.
Lift No. 3 stopped.
Lift No. 2 stopped.
Lift No. 1 stopped.
Lift No. 1 started.
Lift No. 2 started.
Lift No. 3 started.
Lift No. 2 stopped.
Lift No. 3 stopped.
Lift No. 1 stopped.
Lift No. 1 started.
Lift No. 2 started.
Lift No. 3 started.
lifts: 3
floors: 10
Lifts
stop
Floors

Smalltalk4.0 & 明の内容*

- 新ユーザーインタフェース
- アウトライン機能
- 各種の新ブラウザー
  - Blizzardブラウザー、クラス木ブラウザー、アウトラインブラウザー、アウトラインカテゴリーブラウザー、クラス階層ブラウザーなど、従来のSmalltalkより強力な多数のブラウザーがある。
- グラファ機能
  - 図エディターの基本機能を持っていて、この機能を利用してクラス木ブラウザーなどが実現されている。ノードやアークの整列機能も持っている。
- 移植性
- 動的コンパイル機能
- シンボリックデバッガー
- 変更管理機能
  - 各クラスおよびメソッド毎に変更履歴を管理できる。
- 効率測定機能
- 設計評価機能*
- ランタイムアプリケーション開発機能

設計評価機能*

- 特殊化-抽象化因子（HF）、組み立て-要素分解因子（RF）、多相化因子（PF）を計算して、ある程度の設計の評価ができる。
  - HFA = クラスAのスーパークラス数 / (クラスAのスーパークラス数 + クラスAのサブクラス数 + 1)
  - RFA = クラスAのトポロジカルソート順位 / 全クラス数
  - PFA = システム内でユニークなクラスAのメソッド数 / システム全体のメソッドの種類の数

PARTS Workbench

- DigiTalk社の部品構築ビジュアル言語＆環境
- Smalltalk/V, C++, Cなどでも部品構築できる
- RDBとのインタフェースあり

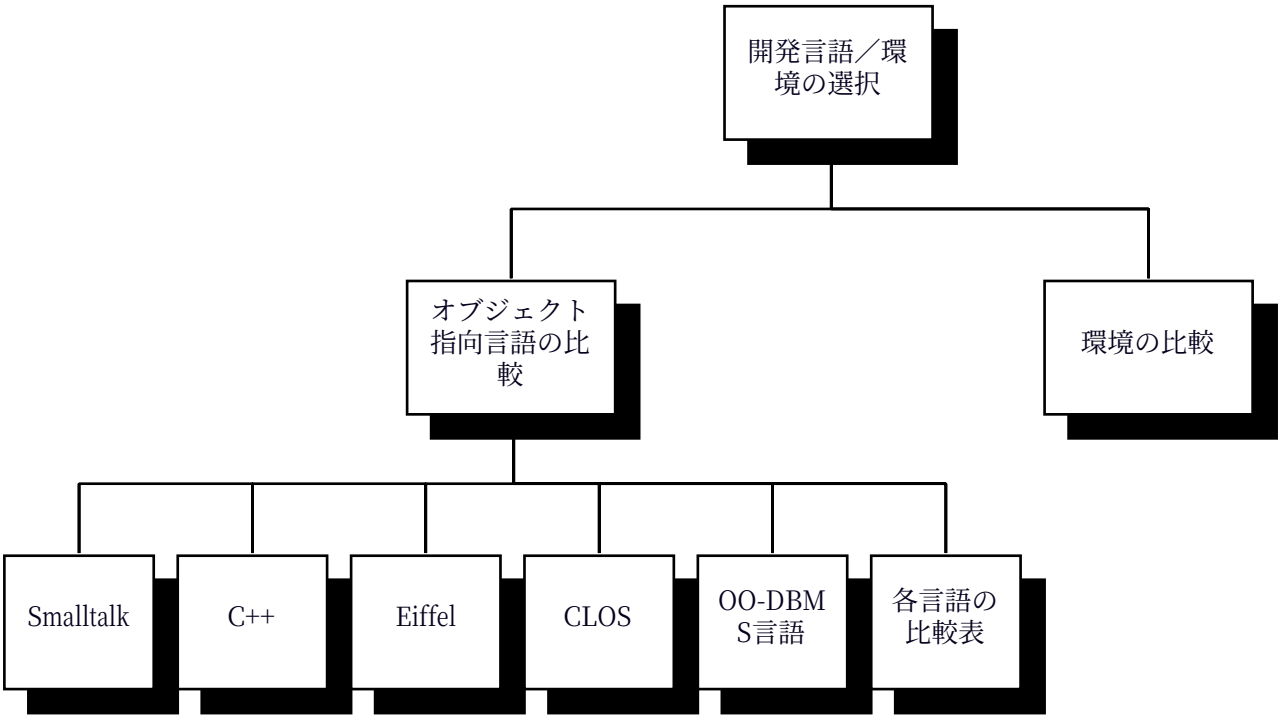
Objectworks/C++*

- Objectworks/C++はParcPlace Systems社のC++2.0支援環境である。基本的に、Objectworks/Smalltalk的な支援環境をC++に与えるものである。Sunワークステーション上で稼働する。
- ファイルシステムブラウザー機能
- ソースコードブラウザー機能
  - 継承構造・クラス・関数の検索とアクセス、コンパイル、などを実現している。
- ソースコードデバッグ機能
- 構成管理機能
- UNIXインターフェース機能
- 　全体として、C++の開発・保守支援環境として強力な機能を持っているが、チームによる協同作業を支援する機能が弱い。

SoftBench*

- HP社のSoftBenchは、HP-UX上で稼働し、Fortran, C, C++.の開発を支援する。
- ブラウザー機能
  - 継承構造・クラス・関数の検索とアクセス、コンパイルなどを実現している。
- 構文検査と自動修正機能
  - 言語規則と形式のガイドラインに反した構文を検査し、自動修正する。
- テンプレート生成機能
- プログラムデバッグ機能
- 版管理と構成管理機能
- 静的解析機能
  - クラス・メンバー関数・変数・関数の相互関係を解析し、照会することができる。
- カプセル化機能
  - Encapsulator機能と呼ばれ、Encapsulator記述言語を使用して、UNIXのコマンド行インタフェースを持つツールを、ツールのソースコードを変更することなく、SoftBenchに組み込むことができる。
- 全体として、大規模なシステムの開発を前提にし、UNIX上の既存ツールや他社製のツールを自らの環境内に取り込む機能を持った、Fortran, C, C++の開発環境である。
  - カプセル化機能を使って、OOSDなどOO-CASEツールを取り込む動きがある。

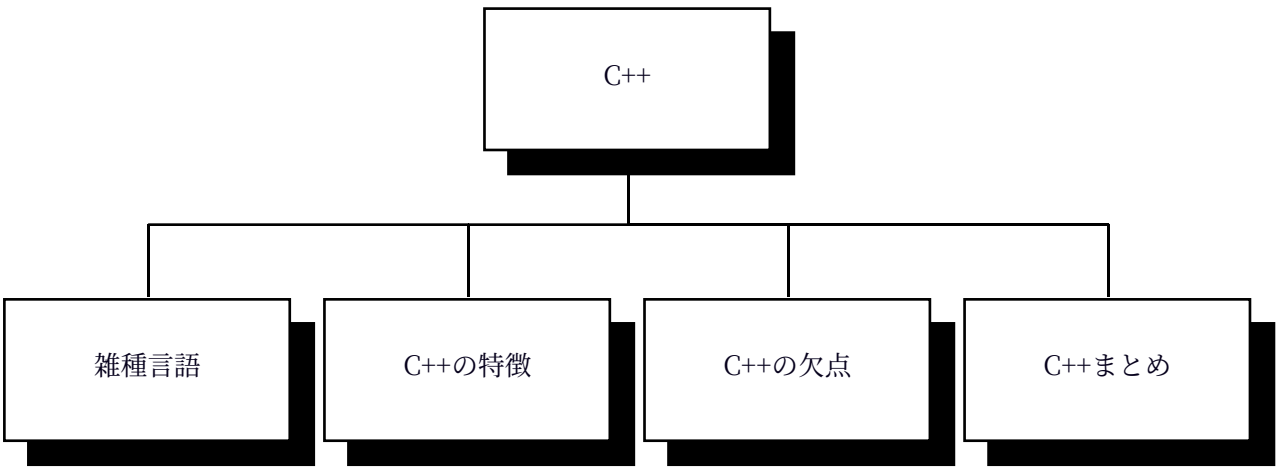
開発言語／環境の選択


開発言語／環境の選択
オブジェクト指向言語の比較
環境の比較
Smalltalk
C++
Eiffel
CLOS
OO-DBMS言語
各言語の比較表

Smalltalk

- 最初のポピュラーなOO言語
  - Xerox PARCで開発、今はPPS
  - Smalltalkの成功が、他の多くのOO言語を生んだ
- 言語だけでなく開発環境
  - 一人用環境として最高の部類に入る
  - 多人数のプロジェクトには弱い
  - 外部のソフトウェアやハードウェア機器とのインタフェースが弱い
- Smalltalkの特徴
  - インタープリターとクラスブラウザーを通して、Smalltalkの全ての能力を使うことができる
    - 最近ではインタープリターの後ろにインクリメンタルコンパイラーも控えている
  - 文法は単純
  - 変数と属性は型がなく、クラスも含めてすべてがオブジェクトである
  - クラスの追加・拡張・テスト・デバッグはすべて対話的に行うことができる
  - ガーベージコレクターがプログラマーをメモリー管理から解放する
- Smalltalkの利点
  - 高度に対話的な開発環境で、従来のコンパイラー言語の編集・コンパイル・リンクというサイクルによる遅れを避けることができる
  - プログラムの開発が早く済む
    - 理想的なインクリメンタル開発・デバッグ環境
  - 強力なクラスライブラリーにサブクラスを追加して開発できる
  - MVC機構によるユーザーインタフェース
    - 強力だが複雑
- まとめ
  - 実行時に修正可能なメタデータが使える、純粋なOOシステム
  - 自身に含む環境と密に統合化されているインクリメンタルコンパイラーで、インクリメンタルな手っとり早い開発とデバッグを理想的に支援する

C++



雑種言語

- AT&Tベル研究所で開発
- ある部分はオブジェクトだし、他の部分はオブジェクトでない
- Cの拡張なので、オブジェクト指向の機能だけでなく、Cの弱点の矯正も実現されている
  - サブルーチンのインライン展開、関数の多重定義、関数プロトタイプなどOOPと関係ない追加機能が多い
- Cの拡張が起源なので、多くのコンピュータ会社がサポートし、非専売言語と見なされている
  - フリーのコンパイラーがあり
    - FSFのg++
  - 主要な汎用のOO言語となりつつある

C++の特徴

- 強い型付け
- 動的束縛の実現方法は効率的
- 多重継承可能
- 可視性の制御が強力
  - public: どのクラスからもアクセス可能
  - protected: サブクラスからアクセス可能
  - private: アクセスはクラス内のみ
  - friend: 特定のクラスまたは関数からアクセス可能

C++の欠点

- 標準クラスライブラリーがない
  - 入出力・コルーチン・複素数演算のライブラリーはある
  - クラスライブラリーは複数の開発者が提供している
    - NIH, Interviews, ET++
    - 相互に互換性がない
- 継承と動的束縛は可能
  - だが、スーパークラスでvirtualと宣言しておかなければならない
    - このため、サブクラスで何を重複定義するかあらかじめ予知しておかなければならない
    - サブクラスを定義するたびにスーパークラスを修正しなければならない
      - 再利用可能なクラスライブラリーの構築が困難
        - ライブラリーのソースコードがないと悲劇的
- 宣言の構文はぶざまで、構文は複雑
- 演算子とメソッドの多重定義が可能
  - だが、引数の数と型が変えられるため、不要な複雑さを招く
- メモリーの割当方法が複雑
  - コンパイラーによる静的割り当て、スタックへの割り当て、ヒープへの実行時動的割り当て
    - プログラマーが異なるメモリータイプのオブジェクトの混同と、メモリーの破壊を避ける必要がある
- ひとつのクラスが、構成子とconversion関数を複数持つことができる
  - conversion関数は、代入と引数の型変換を行う
  - 混乱を招きやすい

C++まとめ

- 複雑で適応性のある言語で、以下の特徴を持つ
  - エラーの早期発見が可能
  - 実現のための種々の選択が可能
  - 設計の柔軟さと単純さを犠牲にした効率の良さ

Eiffel

- Bertrand Meyerが開発
- 特徴
  - 強い型付け
  - 多重継承
  - パラメータ化クラス（generics）
  - 自動メモリー管理
  - 表明
  - 質素なクラスライブラリー
    - リスト、木、２分木、スタック、キュー、ファイル、文字列、ハッシュ表
  - 移植性のためCへ翻訳される
  - ソフトウェア工学的に良い機能
    - カプセル化、アクセス制御、改名、スコープ
  - 商業的言語として、技術的には最高のOO言語
  - 属性と操作を抽象化し、featureとして同様に扱う
  - メモリー管理はコルーチンを通して行われる
    - 実行時にコルーチンの自動実行をOn/Off制御できる
  - 仮想メモリーを支援しないOSのために、自動ページングを行うためのコンパイラースイッチがある
  - 厳密なプログラミングモデルが支援される
    - 事前条件（precondition）
      - 操作の呼び出し側が満足すべき条件
    - 事後条件（postcondition）
      - 操作自身が達成すべき条件
    - 不変表明（invariant）
      - クラスがいつでも満足すべき条件
    - 条件と不変表明はクラス宣言の一部で、子供のクラス全部も従わなければならない
    - 例外
      - 条件と不変表明が犯されると、操作が失敗するか、例外ハンドラーが書かれていればそれが起動される
    - コンパイラーが幾つかのレベルでエラーをチェックできる
      - デバッグが終わったら、表明チェックを外すことができる

CLOS

- Common Lispのオブジェクト指向拡張
  - X3J13 ANSI標準ワーキンググループの成果
    - Flavors, Common Loopsなどを調査したが標準として採用せず
    - 成功が証明された言語機能を盛り込んだ新言語を設計
- 特徴
  - 弱い型付け
  - もともと雑種的だが、Common Lispの特性をうまく統合化し、純粋なOO言語のほとんどの利点も取り込んでいる
  - 事実上Lispの基本関数とオブジェクトの区別は付かない
  - インタープリターだがコンパイルもできる
  - デバッグ機能は実現により異なるが、一般的に非常に良い
  - 現在、標準ライブラリーは提供されていない
  - 強力な継承機能
    - 多重継承
    - 同じ名前の特性を継承することによる、結果の曖昧さを解決するルールがある
  - 汎化メソッドで動的束縛を実現している
    - 引数は陽に指定する
    - Smalltalkのselfにあたる特別な変数はない
    - 他の言語と異なり、多重多相性をサポートする
      - 一つ以上の引数によって、ある操作に対する特定のメソッドが決まる
        - あまり望ましくないが...
  - 実行時にアクセスしたり更新できるメタデータが豊富
    - 言語の標準（メタオブジェクトプロトコル）
    - 他の言語と異なり、Lisp系言語は実行時に新しい手続きを作ることができる
  - カプセル化は強制されない

OO-DBMS言語

- データベース管理とOO言語を結合
  - OOPLは表現力があるがデータの永続性がない
  - DBMSはデータの永続性があるが表現力がない
  - OO-DBMSは表現力とデータの永続性の両方を追及
    - OO-DBMSは大規模なデータを扱う大規模なプログラムを処理しなければならない
  - データベース照会のタイプ
    - 集合指向
      - RDBMSは大きなデータ集合への並列操作を意図する
    - 関連指向
      - OO言語はあるオブジェクトから他のオブジェクトを、ポインターを使って素早くたどるのが得意
      - RDBMSは結合（join）を使うが、ポインターに比べて数段遅い
    - OO-DBMSは両方のタイプの照会を効率よく実行できなければならない
  - システムが特定のオブジェクトの操作を効率良く実現するというのが、OO-DBMSの重要な特性である
    - RDBMSではひとつのオブジェクトの操作と、オブジェクト間の操作を効率良く実行できない
  - OO-DBMSの処理系
    - GemStone
      - Servio Logic社
      - GemStoneの言語はSmalltalkに似ている
        - CとPascalへのインタフェースもある
      - 単純な並行処理の仕掛けとサーバークライアントモデルと継承をサポートする
    - ONTOS
      - Ontologic社
      - SQLとC++による照会をオブジェクト指向でできる
    - Orion
      - MCCのプロトタイプ
      - Common Lispで実現
      - 文法はFlavorsとLOOPSに似ている
      - 多重継承、バージョン、並行性、IBM的ロックをサポート

各言語の比較表

	C++	Smalltalk	CLOS	Eiffel	Objective-C	Object Pascal
基本型とクラスの統合化	雑種	純粋	統合	統合	雑種	雑種
強い型付けによるチェック	Y	N	N	Y	Y	Y
可視性						
顧客からアクセスの制御	Y	Y	N	Y	Y	Y
サブクラスからのアクセスの制御	Y	N	N	Y	N	N
標準クラスライブラリー	N	Y	N	Y	Y	Y
パラメータ化クラス	将来	不要	不要	Y	N	N
多重継承	Y	N	Y	Y	N	N
クラス名のスコープ（パッケージ）	N	N	Y	N	N	N
メッセージモデル						
単一ターゲットオブジェクト	Y	Y	N	Y	Y	Y
複数の引数による動的束縛	N	N	Y	N	N	N
メソッド組み合わせ機能						
super概念	N	Y	Y	Y	Y	Y
before & afterメソッド	N	N	Y	N	N	N
表明と制約	N	N	N	Y	N	N
実行時のメタデータ	N	Y	Y	N	Y	N
ガーベージコレクション	N	Y	Y	Y	N	N
効率						
可能なとき静的束縛	Y	N	N	Y	Y	Y

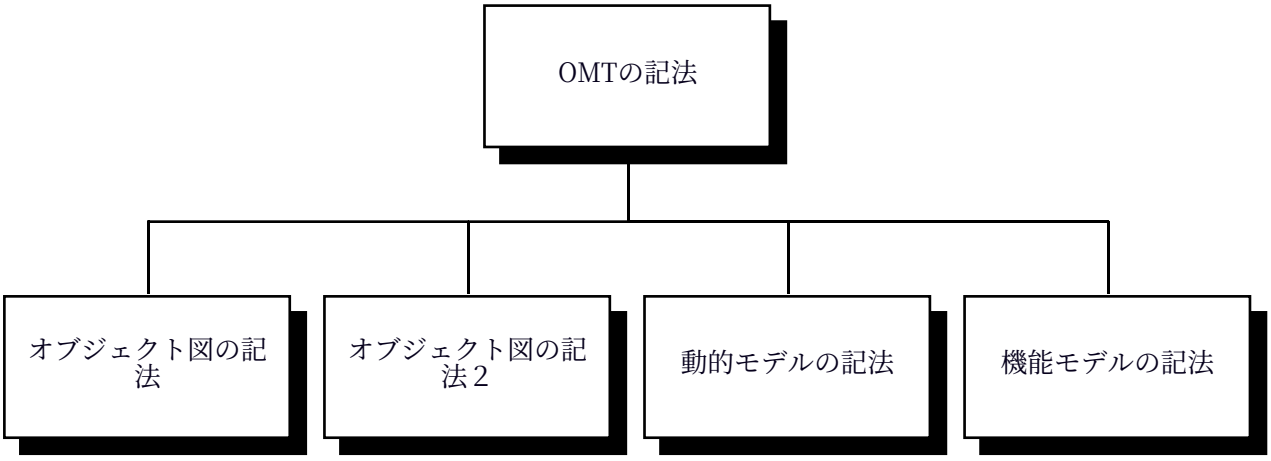
環境の比較

- 開発環境
  - PCTE
  - Smalltalk
  - UNIX
  - PC(MacOS, Windows)
- 実行環境
  - UNIX
  - PC(MacOS, Windows)
  - メインフレーム

参考文献

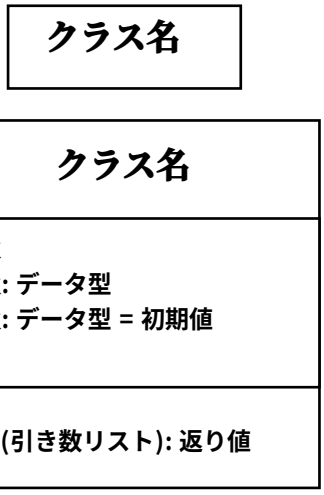
- [Acumen 91] Acumen Software. Widgets/V Mac Tutorial and Programming Guide. Acumen Software, 1991
- [Apple 93] Apple Computer. HyperCard Script Language Guide: The HyperTalk Language, Apple Computer, 1993
- [Conklin 87] Jeff Conklin. Hypertext: An Introduction and Survey. IEEE Computer, Vol. 20, No.9, pp.17-41, 1987
- [Digitalk 93] Digitalk Inc. Smalltalk/V Tutorial. Digitalk Inc., 1993
- [Digitalk 93] Digitalk Inc. Smalltalk/V Programming Reference. Digitalk Inc., 1993
- [Object 91] Object International, Inc.. OOATool for the Macintosh. Object International, Inc., 1991
- [萩谷 91] 萩谷昌己. 視覚的プログラミングと自動プログラミング. コンピュータソフトウェア, Vol.8, No.2, pp.27-39, 岩波書店, March 1991
- [富士ゼロックス 91] 富士ゼロックス情報システム. マルチクライアントプロジェクト「オブジェクト指向システム技術の実用に関するプロジェクト」成果報告書. 富士ゼロックス情報システム, 1991

OMTの記法

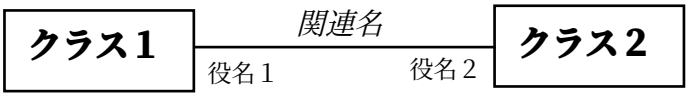


オブジェクト図の記法

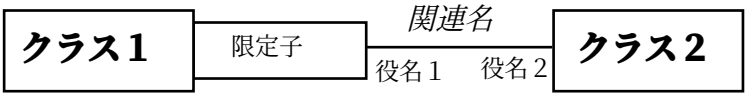
クラス



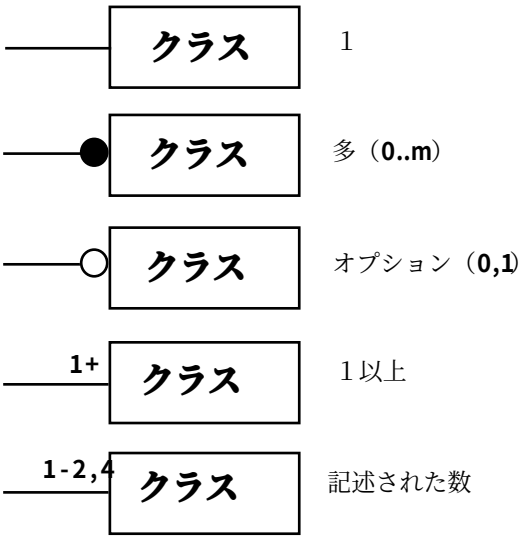
関連



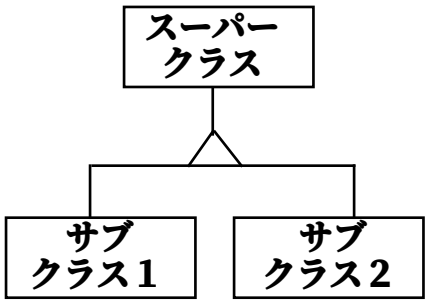
限定付き関連



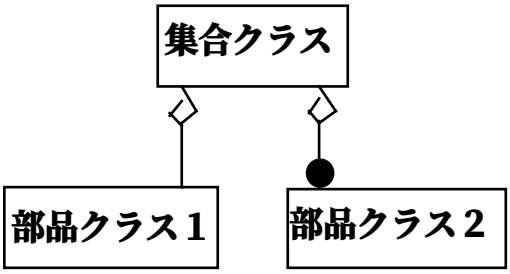
関連の多重度



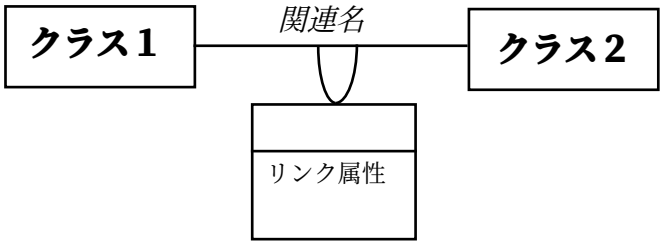
汎化（継承）



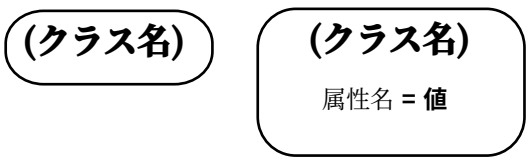
集約



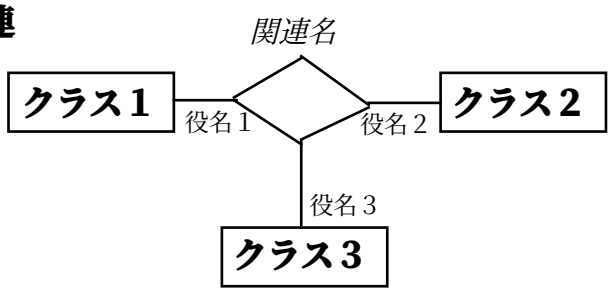
リンク属性



オブジェクト　インスタンス



３項関連

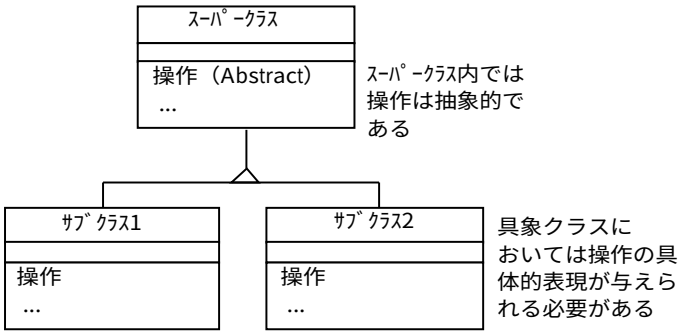


Source: Runmbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. Object-Oriented Modeling and Design. Prentice Hall, 1991

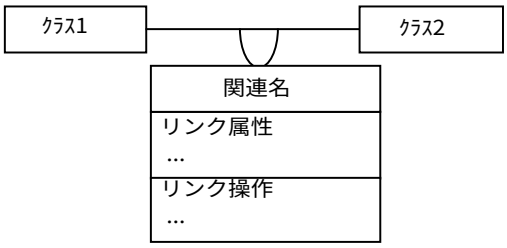
オブジェクト図の記法２

ｵﾌﾞｼﾞｪｸﾄﾓﾃﾞﾙ記法
応用概念

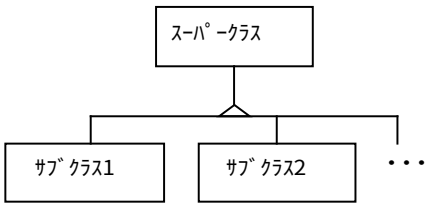
抽象操作：

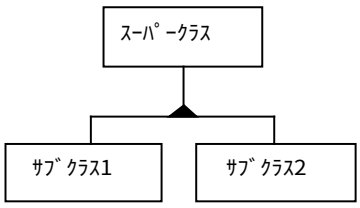


クラスとしての関連：

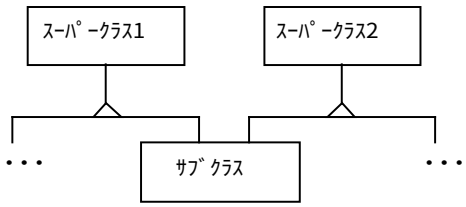


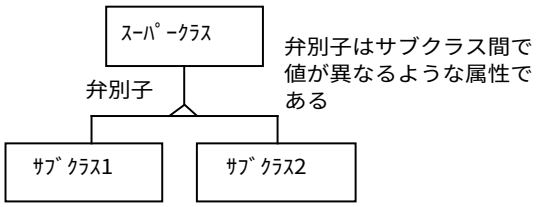
汎化の性質：





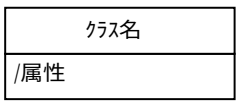
多重継承：



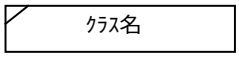


ｸﾗｽ属性とｸﾗｽ操作：

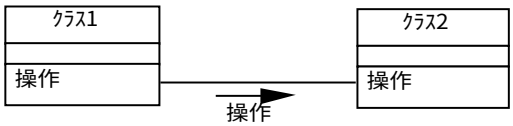
派生属性：



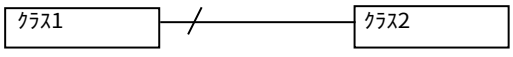
派生ｸﾗｽ：



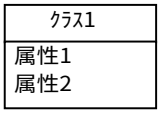
操作の伝搬：



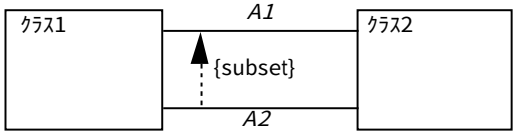
派生関連：



ｵﾌﾞｼﾞｪｸﾄに対する制約：



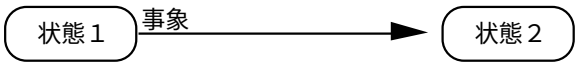
関連間の制約：



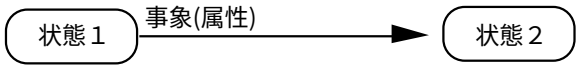
動的モデルの記法

動的モデル記法

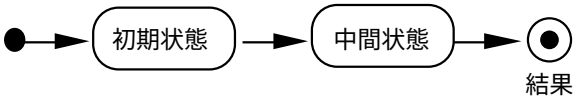
事象は状態間の遷移を起こす：



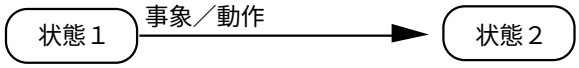
属性を伴う事象：



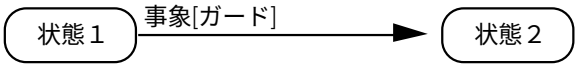
初期および終了状態：



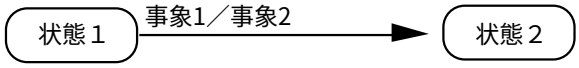
遷移上の動作：



ガード付き遷移：

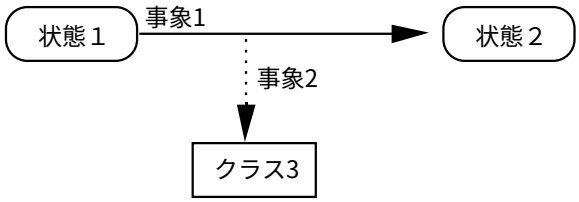


遷移上の出力事象：

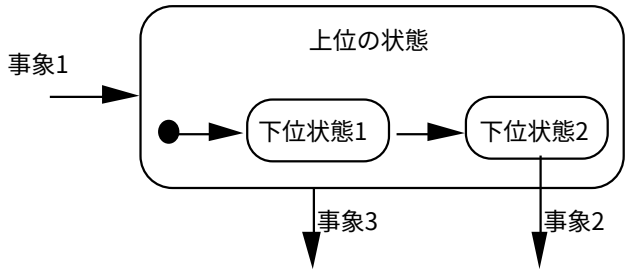


状態内の動作および活動：

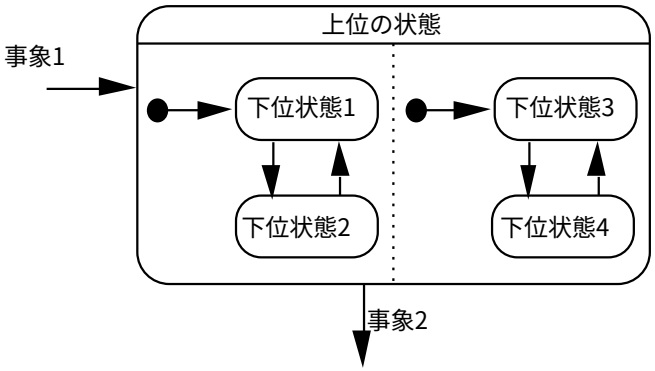
他のオブジェクトに事象を送る：



状態の一般化（入れ子構造）：

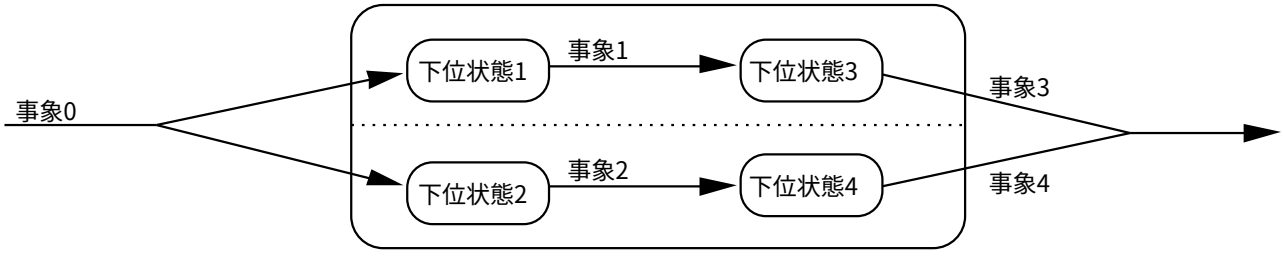


平行動作を表現する：



制御の分岐：

制御の同期：

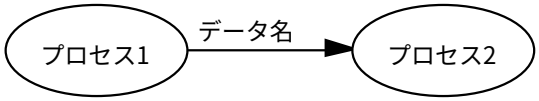


機能モデルの記法

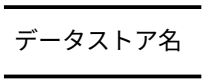
機能モデル記法

プロセス：

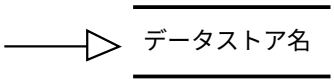
プロセス間のデータフロー：



データストアあるいはファイルオブジェクト：

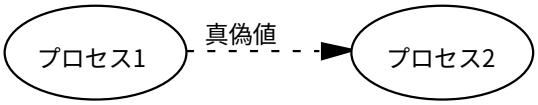


結果をデータストアに出力するデータフロー：

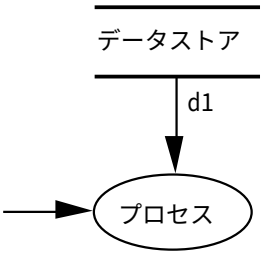


アクターオブジェクト（ソースあるいはデータ溜り）：

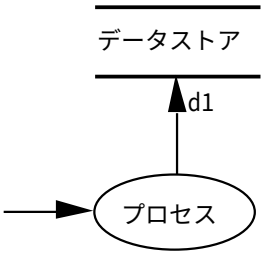
制御フロー：



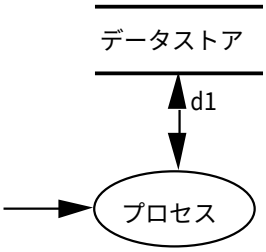
データストア値のアクセス：



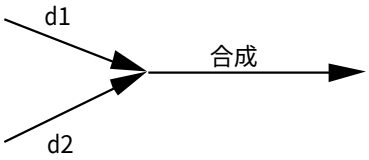
データストア値の更新：



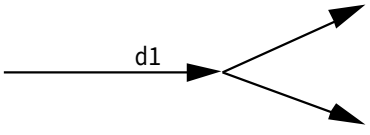
データストア値のアクセスと更新：



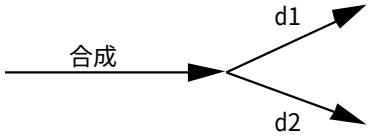
データ値の合成：



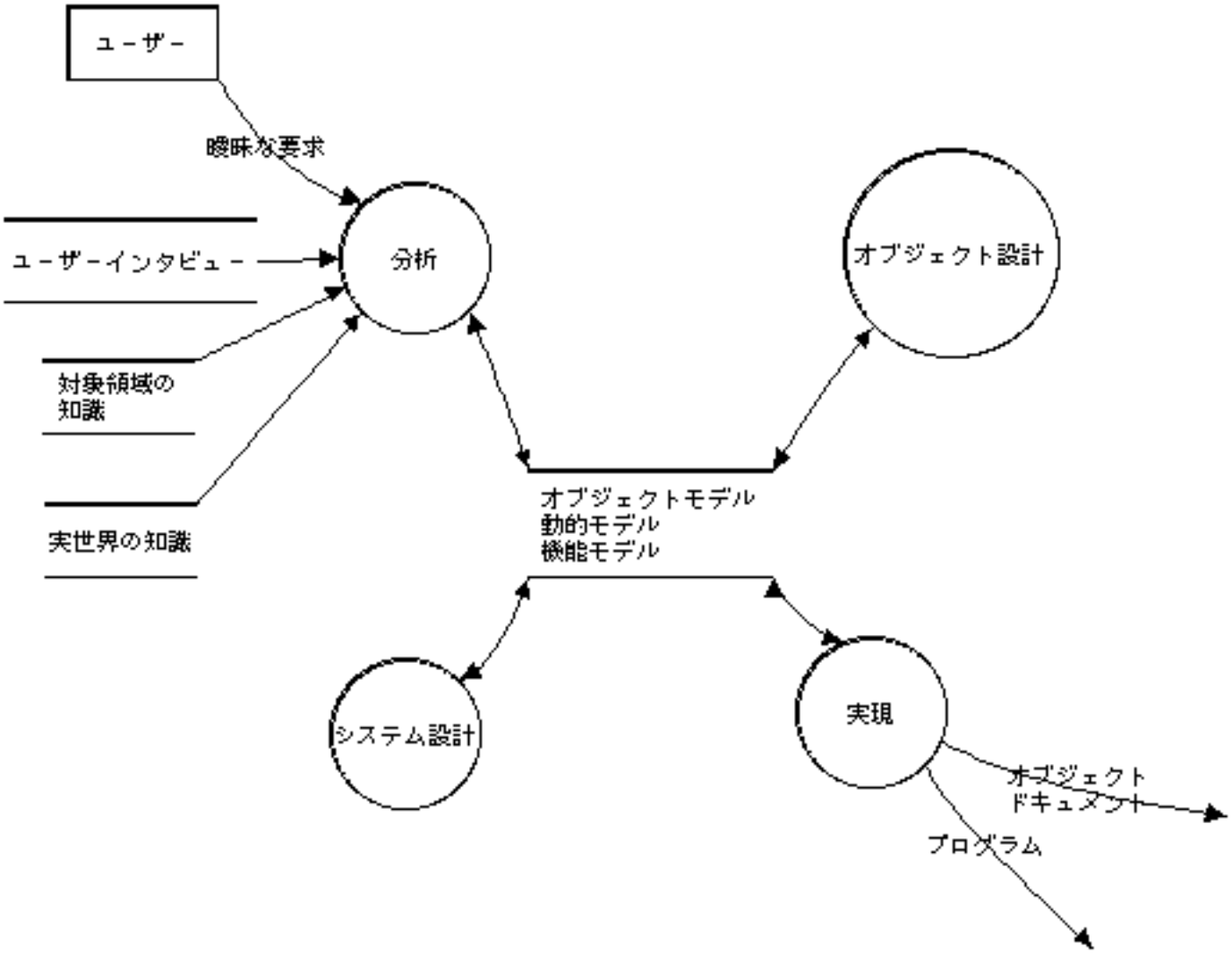
データ値の複製：



データ値の分割：



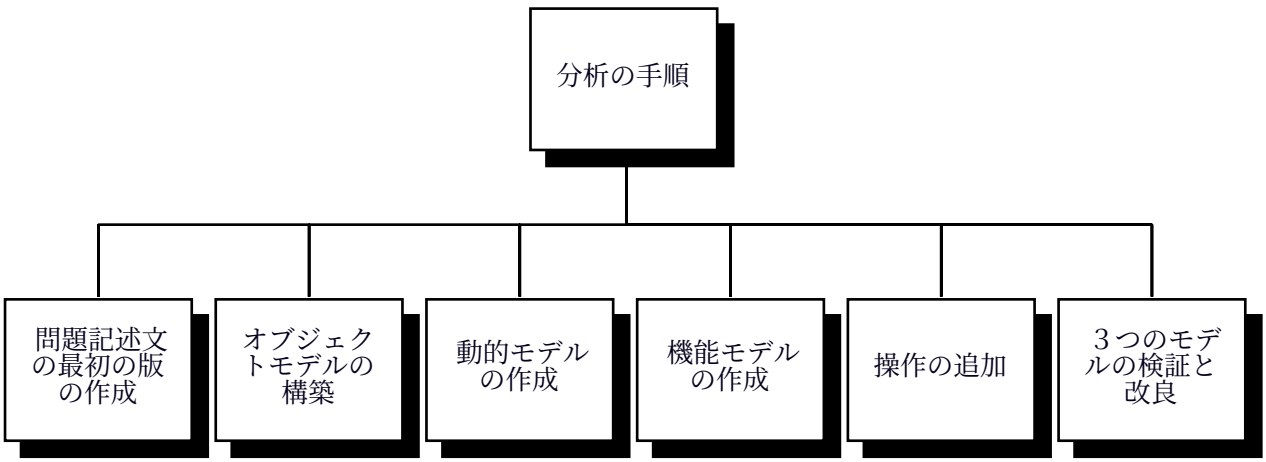
手順概観


ユーザー
曖昧な要求
ユーザーインタビュー
対象領域の
知識
実世界の知識
分析
オブジェクト設計
オブジェクトモデル
動的モデル
機能モデル
システム設計
実現
オブジェクト
ドキュメント
プログラム

分析の指針

- 分析の目的
  - 要求を明らかにする
  - ソフトウェアの要求者と開発者の間の基本的な同意点を与える
    - 開発者が要求システムを理解することも含む
  - 後の設計と実装の枠組となる
- システムが何をするべきかというモデルを作る
  - どうやって作るかは、敢えて書かない
- 分析は決まった順序で行うことはできない
  - 部分的なモデルから全体の完全で大きいモデルへ拡張していく
- 分析を機械的に行うことはできない
  - 専門的知識や実世界の経験を反映するため専門家と対話したり、概念の相違を明らかにしたり曖昧さを排除するために依頼者と対話する必要がある

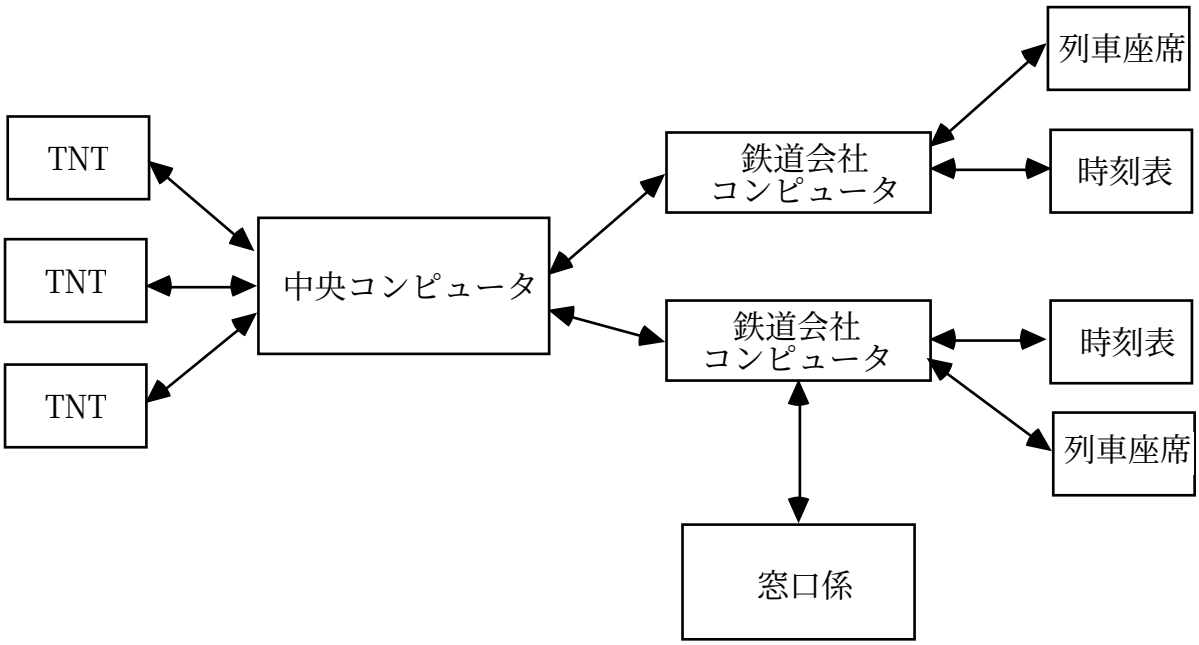
分析の手順


分析の手順
問題記述文の最初の版の作成
オブジェクトモデルの構築
動的モデルの作成
機能モデルの作成
操作の追加
３つのモデルの検証と改良

問題記述文の最初の版の作成

- HowでなくWhatを記述する
  - ユーザーマニュアルは良い問題記述文の一つ
- 何が必須で、何がオプションかを明記する
- システム内部の記述は避ける
  - 要求と見せかけた設計方針や実現方法に惑わされて、拡張性を制限してはいけない。例えば、
    - 言語の指定
    - アルゴリズムの指定
- 性能要求を記述する
- 問題記述文は、問題を理解するための出発点に過ぎない
- 例*

例*



コンピュータ化された鉄道情報ネットワークを支援するソフトウェアを設計せよ。このシステムには、ＪＲや私鉄を含む全鉄道の情報を供給する、鉄道情報協会の一般顧客用情報提供端末（ＴＮＴ = Train Network Terminal）や駅の窓口係が含まれる。

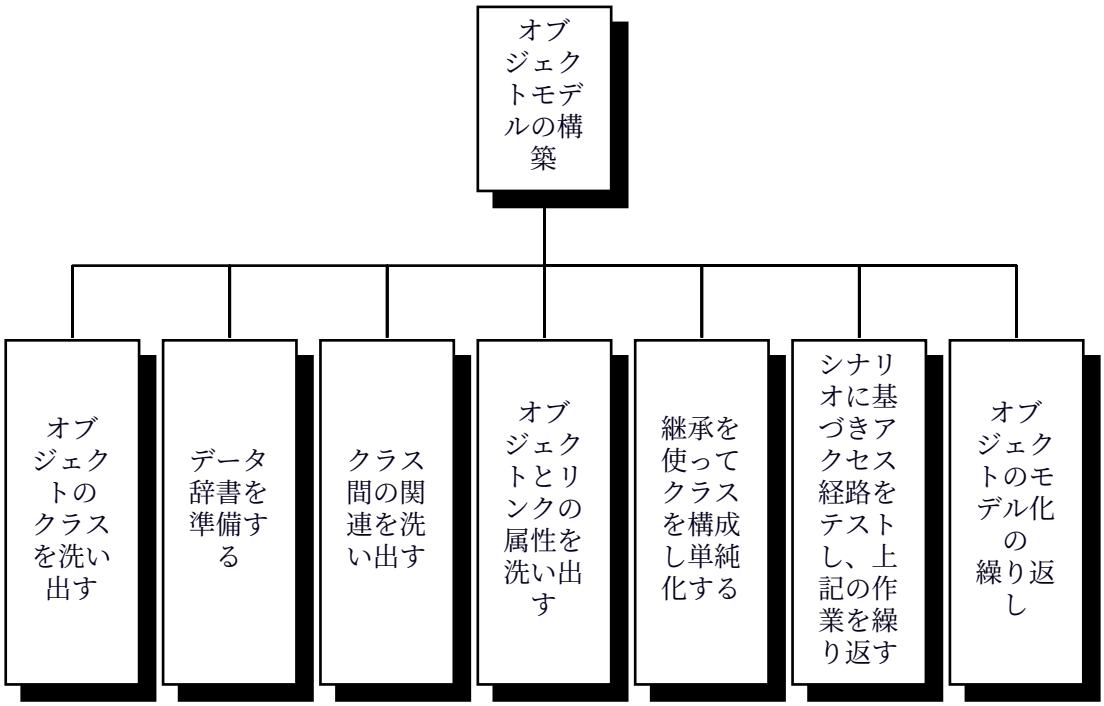
各鉄道はそれぞれのコンピュータによって、列車座席の管理を行い、時刻表や座席に対して発生するトランザクションを処理している。各鉄道は、自社のコンピュータに直接接続された窓口係用端末を所有している。窓口係は、座席の予約データなどのトランザクションデータを投入する。ＴＮＴは適切な鉄道にトランザクションを照会する中央コンピュータに接続されている。ＴＮＴは鉄道カードを受け入れ、ユーザーと対話し、トランザクションを実行するため中央コンピュータと通信し、情報を表示し印刷する。

このシステムは、適当な記録の保管設備とセキュリティ対策が必要である。また、同じ座席への並行アクセスを正確に扱わなければならない。

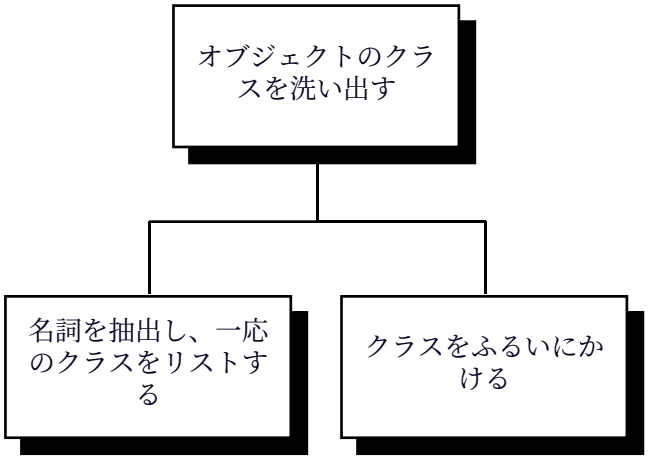
各鉄道は、自社のコンピュータにそれぞれのソフトウェアを用意しているものとする。そして、あなたがＴＮＴとネットワークのソフトウェアを設計するものとする。

共有されたシステムのコストは、鉄道カードを持っている顧客の数に応じて各鉄道に割り当てられる。

オブジェクトモデルの構築



オブジェクトの
クラスを洗い出す



名詞を抽出し、一応のクラスをリストする

- 物理的「もの」だけでなく概念もオブジェクトになる
- 対象領域の「もの」をオブジェクトとし、コンピュータで実現するための「もの」（例えば木構造とかサブルーチン）は対象としない
- 対象領域や一般常識から導かれるクラスもある
- 継承とか高レベルのクラスは、この段階では気にしない
- 例
  - 問題記述の名詞から抽出したTNTのクラス
    - 鉄道情報ネットワーク、ソフトウェア、システム、JR、私鉄、鉄道、情報、鉄道情報協会、TNT、駅、窓口係、鉄道コンピュータ、列車、座席、時刻表、トランザクション、窓口係用端末、予約データ、トランザクションデータ、中央コンピュータ、鉄道カード、ユーザー、記録保管設備、セキュリティ対策、アクセス、コスト、顧客
  - 問題領域の知識から抽出したクラス
    - 通信回線、切符、最短情報、列車情報、トランザクションログ、列車ダイヤ、運賃表、カード会社、口座

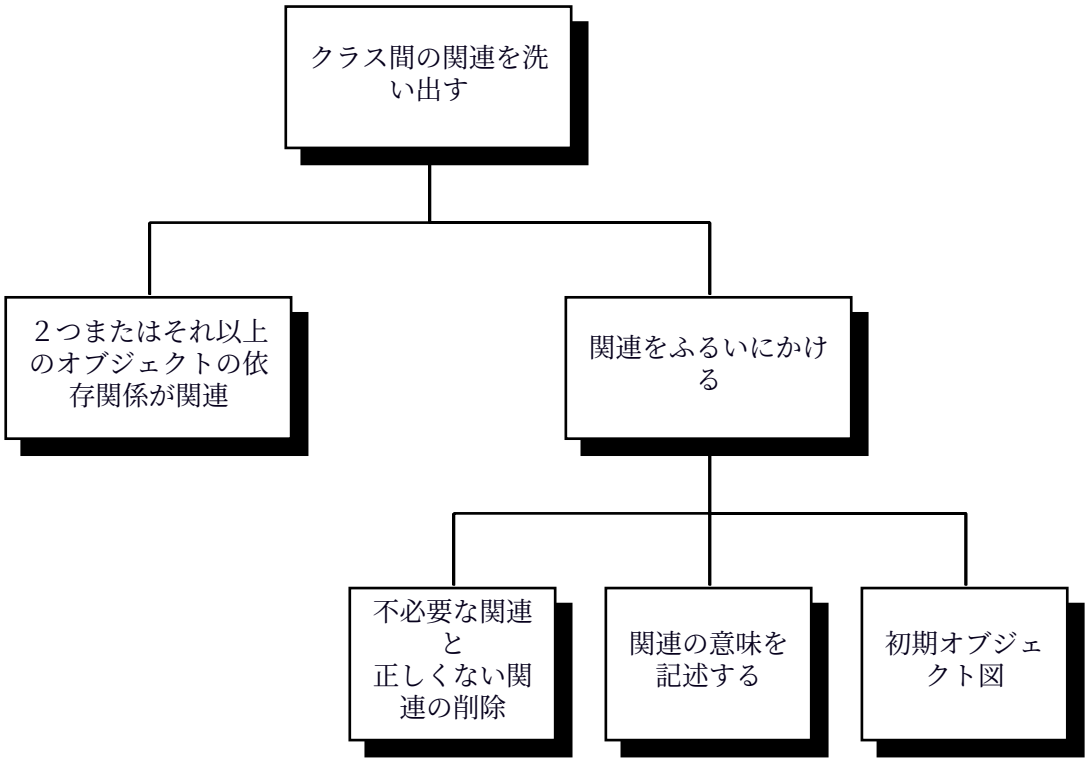
クラスをふるいにかける

- 冗長なクラス
  - より記述的な名前の方を使う
    - 例えば、TNTのモデルのユーザーと顧客では、顧客の方がより記述的
- 無関係のクラス
  - 問題の中でほとんど使われないクラスを削除する
    - コスト（処理の範囲外）
- 曖昧なクラス
  - クラスは明確でなければならない
  - 曖昧な領域や広すぎる範囲を占めるクラスを削除する
    - システム、セキュリティ対策、記録保管設備（トランザクションの一部）、鉄道情報ネットワーク、時刻表
- 属性
  - 名前・年齢・体重・住所など、主に個々のオブジェクトを説明するようなものは属性とする
    - 予約データ、情報、トランザクションデータ、JR（属性の値）、私鉄（属性の値）
  - ただし、その特性が独立した存在であることが重要ならクラスにしておくべきである
- 操作
- ロール
  - クラスの名前は、クラスの特性を直感的に捉えやすいものであるべきで、関係の中で果たす役割（ロール）の名前ではない
- 実現上の構造
  - 設計時にはクラスになるかも知れないが、今は不要
    - トランザクションログ、通信回線、アクセス、ソフトウェア
- 正しそうなクラスの例
  - 鉄道、鉄道情報協会、TNT、窓口係、鉄道コンピュータ、列車、座席、列車ダイヤ、運賃表、窓口係用端末、中央コンピュータ、鉄道カード、顧客、駅、切符、最短情報、列車情報、カード会社、口座

データ辞書を準備する

- クラス・属性(attribute)・関連(association)・操作(Operation)を含むデータ辞書を作成する
  - 単語は、単独で存在するといろいろな意味を持つので、単語とそれに対応するモデル上の意味をデータ辞書に定義する
    - クラスが表現する範囲
    - 利用や所属にあたっての仮定や制限
    - 関連、属性、操作
- 例
  - 鉄道カード
    - TNTを使った列車情報アクセスを認識するために顧客に配られるカードである。
    - 各カードは、鉄道コードとカードコードを保持し、クレジットカードと改良型イオカードの標準規格に基づいてコード化されている
    - 鉄道コードは、鉄道情報協会内の鉄道を一意に識別できるようにできている。
    - 各カードは一人の顧客によって所持される

クラス間の関連を洗い出す



２つまたはそれ以上のオブジェクトの依存関係が関連

- 動詞から捜す
  - 物理的場所・直接動作・対話・所有関係・ある条件を満たす関連がある
  - 関連と集約の区別に時間を使いすぎないこと
- 対象領域や一般常識から導かれる関連もある
- 例
  - 動詞句から探す
    - 鉄道はソフトウェアを用意する
    - ソフトウェアを設計する
    - システムは情報を供給する、システムにはTNTや窓口係が含まれる、システムは並行アクセスを扱う
    - 列車座席の管理を行う
    - 鉄道コンピュータは座席に対して発生するトランザクションを処理する
    - 鉄道は窓口係用端末を所有する、コストは鉄道に割り当てられる
    - 窓口係は、座席の予約データとトランザクションデータを投入する
    - TNTは鉄道カードを受け入れる、TNTは顧客と対話する、TNTは情報を表示する、TNTは情報を印刷する、TNTは中央コンピュータとトランザクションを通信しあう
  - 問題領域の知識
    - 鉄道情報協会はTNTを所有する
    - 鉄道カードはクレジット情報にアクセスする
    - 鉄道は窓口係を雇っている
    - 顧客は切符を買う
    - 列車は車からなり、車は座席を持つ
    - 列車は駅から駅へ移動する
    - 列車は列車ダイヤに従って運行する
    - 切符の値段は運賃表から決まる

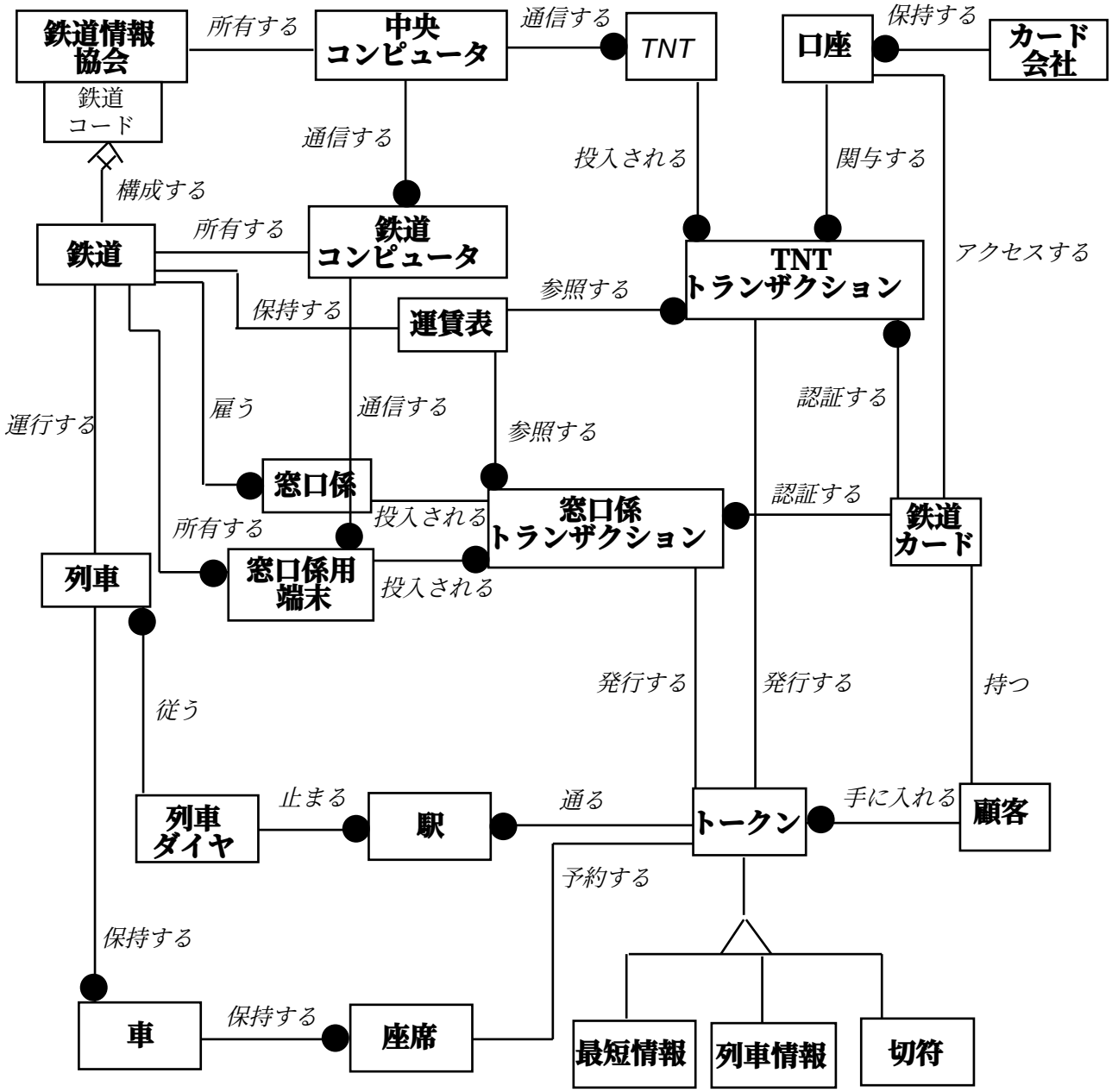
不必要な関連と正しくない関連の削除

- 削除されたオブジェクト間の関連
  - 鉄道はソフトウェアを用意する、ソフトウェアを設計する、システムにはTNTや窓口係が含まれる、システムは情報を供給する、コストは鉄道に割り当てられる
- 無関係な関連
- 実現上の関連
  - システムは並行アクセスを扱う
- 動作
  - 関連は、オブジェクト同士の構造的な特性を記述すべきで、一時的事象を記述すべきでない
    - TNTは鉄道カードを受け入れる（TNTと鉄道の間の静的・恒久的な関係でない）
    - TNTは顧客と対話する
    - TNTは情報を表示する
    - TNTは情報を印刷する
- ３項以上の関連
  - できるだけ２項関連か限定付き関連に代える
  - ４項以上の関連の例はほぼ無い
- 派生関連
  - 他の関連から導かれる関連は省略する
    - 例えば、「孫である」は「子である」から導かれる
    - 列車座席の管理を行う、は...
      - 列車は車を保持する
      - 車は座席を保持する
    - 鉄道情報協会はTNTを所有する、は...
      - 鉄道情報協会は中央コンピュータを所有する
      - 中央コンピュータはTNTと対話する
  - オブジェクトモデルのクラス・属性・関連は、できるだけ独立の情報を表すべきである

関連の意味を記述する

- 誤った名前で呼ばれる関連
  - その状況がどのように発生したとか、なぜ発生したとかを言うのでなく、それが何であるかと言うような名前を選ぶ
    - 列車座席の管理を行う
      - 鉄道は座席を保持する
- ロール名
  - 適当なところにロール名を割り当てる
  - 自明なら省略してもよい
- 限定付き関連
  - 限定付き関連を使うことで、オブジェクトを特定することができる
- 多重度
  - １対１の多重度を目指す
  - 限定子が使えないか調べる
  - 順序付け制限子{orderd}が使えないか調べる
  - ただし分析段階では、多重度の把握はほどほどにする
- 見逃された関連
  - 見逃した関連を見つける
    - トランザクションは窓口係用端末から投入される
    - トランザクションはTNTから投入される
    - トランザクションは鉄道カードによって認証される

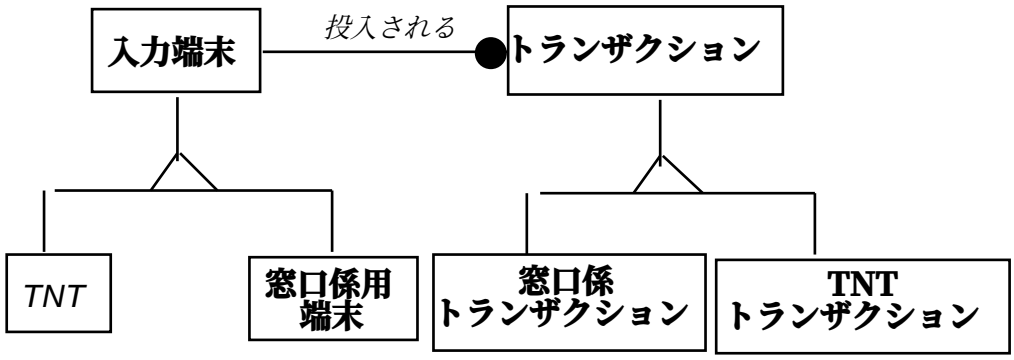
初期オブジェクト図


鉄道情報協会
所有する
中央コンピュータ
通信する
TNT
口座
保持する
カード会社
鉄道コード
構成する
通信する
投入される
関与する
鉄道
所有する
鉄道コンピュータ
TNT トランザクション
アクセスする
保持する
運賃表
参照する
運行する
雇う
通信する
参照する
認証する
窓口係
投入される
窓口係トランザクション
認証する
鉄道カード
所有する
列車
窓口係用端末
投入される
発行する
発行する
持つ
従う
列車ダイヤ
止まる
駅
通る
トークン
手に入れる
顧客
予約する
保持する
車
保持する
座席
最短情報
列車情報
切符

オブジェクトとリンクの属性を洗い出す

- 属性の洗い出し
  - 普通は、所有を示す句が続く名詞に対応する
    - 「ラベルの色」「ウィンドウの位置」など
  - 形容詞が列挙型の属性を表すこともある
    - 「青い」「無い」など
  - 問題の基本構造に影響を与えることは少ないので、分析段階では属性の洗い出しに深入りしすぎない
  - 実現のための属性は、この段階では不要
  - 派生属性は省略するか、目印を付けておく
    - 年齢は、生年月日と現在日付から派生する
    - 派生属性を操作として表さない
  - リンク属性はオブジェクトの属性と間違えやすい
    - 関連の属性は、１つのエンティティ属性でなく、２つのエンティティの間の関連の性質である
- 属性をふるいにかける
  - オブジェクト
    - オブジェクトにすべきものは属性から削除する
  - 限定子
    - 属性を限定子にできないか考える
  - 名前
    - 限定子にできないか検討する
  - 識別子
    - オブジェクトの識別子を属性としない
    - 対象分野の識別子（運転免許証番号・保険証番号など）は属性とする
  - リンク属性
    - リンクに依存する属性はリンク属性とし、オブジェクトの属性から削除する
  - 内部の値
    - 内部の状態を記述するもので、外から見えないとき、この段階では削除する
  - 詳細
    - ほとんどの操作に関係ない、マイナーな属性は省略する
  - 調和しない属性
    - 他の属性と調和しない属性は、クラスを二つに分けるべきかもしれないことを示す

継承を使ってクラスを構成し単純化する



- **共通の構造を継承を使って共有する**
- **継承は２方向あり得る**
  - 既存クラスの共通の機能を汎化しまとめる（ボトムアップ）
    - よく似た属性、関連、操作を持つクラスを調べる
    - 実世界の分類を使って汎化することができることもある
    - 対称性は、汎化する可能性のあるクラスを暗示していることがある
  - 既存クラスを修正し特化する（トップダウン）
    - 形容詞が付いた名詞句を選び出すとよい
      - 正選手、補欠選手、見習い選手など
    - 列挙型の場合分けが、しばしば特化の発見につながる
- **多重継承を使う**
  - 共有度を上げることができるが、複雑度も増すので本当に必要なときに限る
- **同じ名前の関連が同じ意味で何回も表れたら、関連するクラスを汎化できないか検討する**

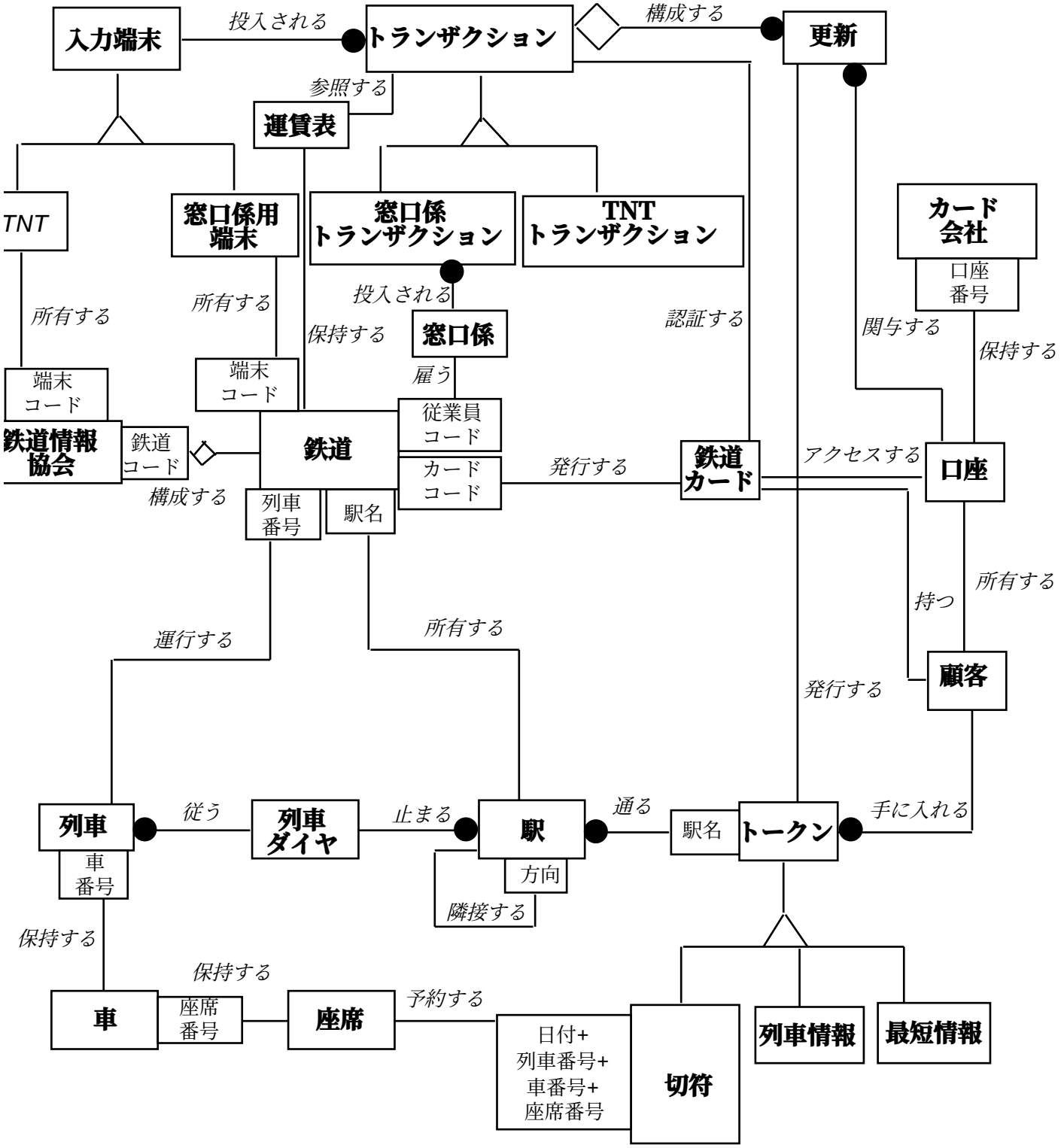
シナリオに基づきアクセス経路をテストし、上記の作業を繰り返す

- 各アクセス経路をたどってみて、各々が意味のある値を引き出すことができるか確認する
  - ユニークな値が期待されているところで、ユニークな値が得られるか？
  - 「多」の多重度が設定されているところで、必要ならユニークな値を取り出す方法が存在するか？
  - 有効な質問を考えてみて、答えがでないようなことがないか？
  - 実世界では単純な事柄が、複雑なアクセス経路で表現されるなら、何かが抜け落ちている可能性がある

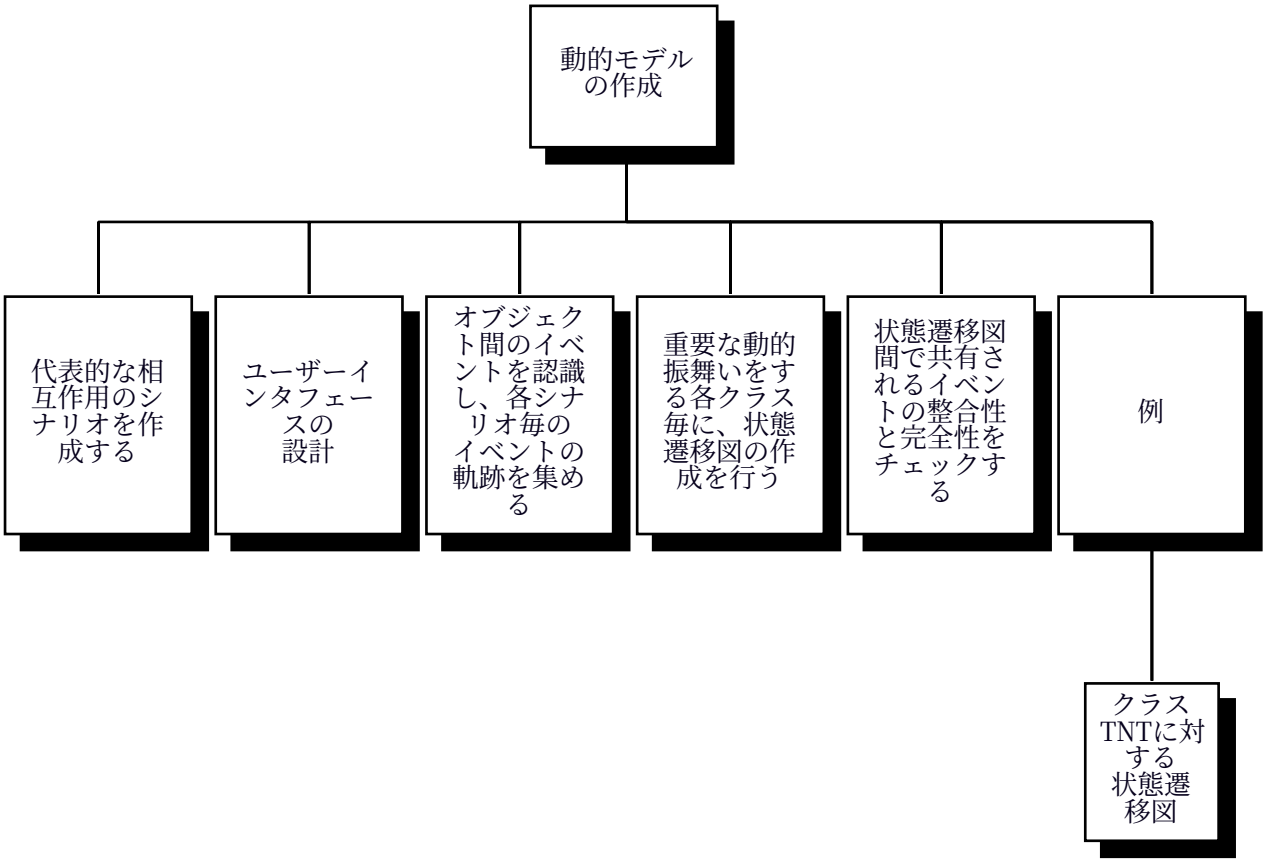
オブジェクトのモデル化の繰り返し

- 未発見のオブジェクトの兆候
  - 関連と汎化のなかの非対象性
    - 類似によるクラスの追加
  - クラス中の異種の属性と操作
    - クラスを分ける
  - きれいに汎化するのが難しい
    - １つのクラスが２つの役割を果たしていることが多いので、１つを上に持って行く
  - 対応する適当なクラスがない操作
    - 対応するクラスを作る
  - 同じ名前と目的の関連
    - これらを結び付ける、見逃していたスーパークラスを作って汎化する
  - 「ロール」がクラスの意味に大きな影響を与える
    - クラスとして独立させた方がよいことがある
    - 関連をクラスにする場合もある
- 不必要なクラスの兆候
  - 属性・操作・関連がないクラス
- 未発見の関連の兆候
  - 操作へのアクセス経路が無い
    - 関連を加える
- 不必要な関連の兆候
  - 関連に冗長な情報
    - 新しい情報を加えない関連を削除するか、派生関連にする
  - 関連をたどる操作がない
    - どの操作もその経路を使わないのなら、多分不要である
- 正しくない場所にある関連の兆候
  - ロール名があまりにも一般的すぎるか特殊すぎる
    - 関連をクラス階層の中で上げたり下げたりする
- 正しくない場所にある属性の兆候
  - オブジェクトをアクセスするのに、その属性の値が必要
    - 限定付き関連が使えるか検討する
- 何回か改訂した例*

何回か改訂した例*


入力端末
投入される
トランザクション
構成する
更新
参照する
運賃表
TNT
窓口係用端末
窓口係トランザクション
TNTトランザクション
カード会社
口座番号
所有する
所有する
投入される
保持する
窓口係
認証する
関与する
保持する
端末コード
端末コード
雇う
鉄道情報協会
鉄道コード
鉄道
従業員コード
カードコード
発行する
鉄道カード
アクセスする
口座
構成する
列車番号
駅名
所有する
持つ
運行する
所有する
発行する
顧客
列車
車番号
従う
列車ダイヤ
止まる
駅
方向
隣接する
通る
駅名
トークン
手に入れる
保持する
保持する
車
座席番号
座席
予約する
日付+列車番号+車番号+座席番号
切符
列車情報
最短情報

動的モデルの作成


動的モデルの作成
代表的な相互作用のシナリオを作成する
ユーザーインタフェースの設計
オブジェクトを認識し、各シナリオ毎のイベントの軌跡を集める
重要な振る舞いをする各クラス毎に、状態遷移図の作成を行う
状態遷移図さん有イベント整合性をチェックする
例
クラスTNTに対する状態遷移図

代表的な相互作用のシナリオを作成する

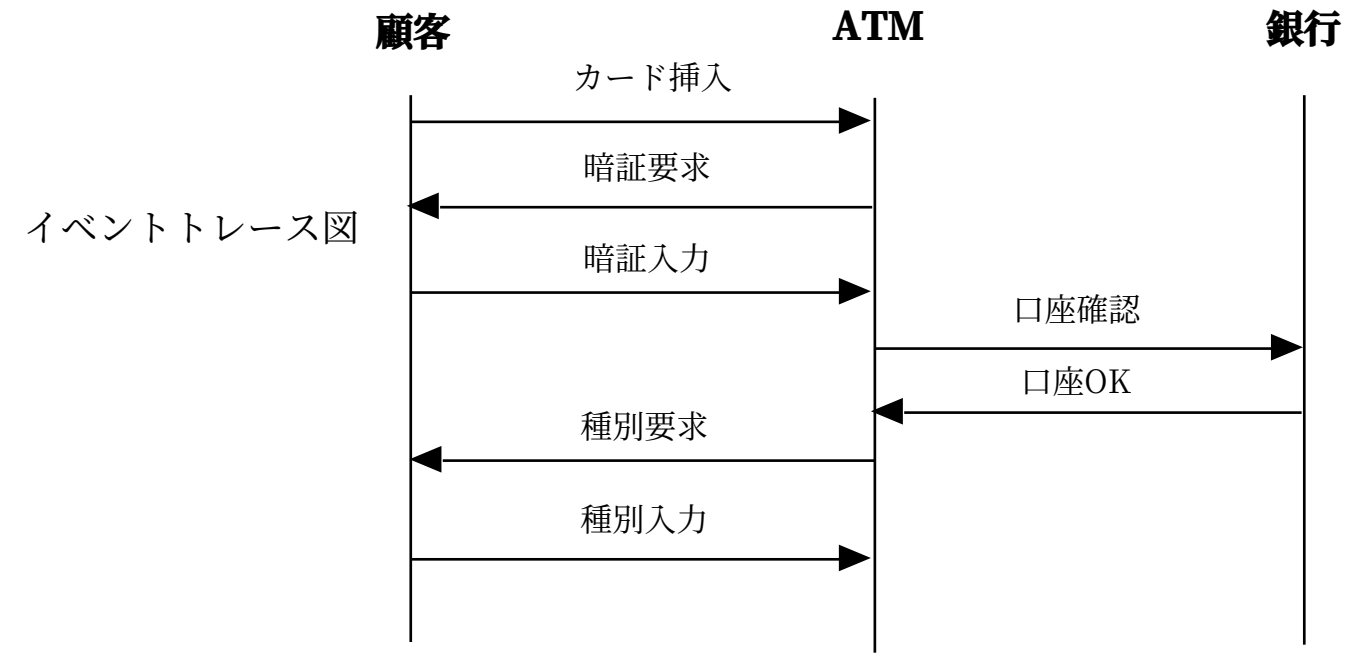
- シナリオはイベントの並び
  - 例
    - TNTが顧客に鉄道カードを挿入するように要求する。顧客が鉄道カードを挿入
    - TNTが鉄道カードを読み、カードIDを認識する
- シナリオ作成の順序
  - 正常な場合
  - 特殊な場合
  - エラーの場合
- 各イベントについて、アクターを認識する
- 出力形式は気にしない

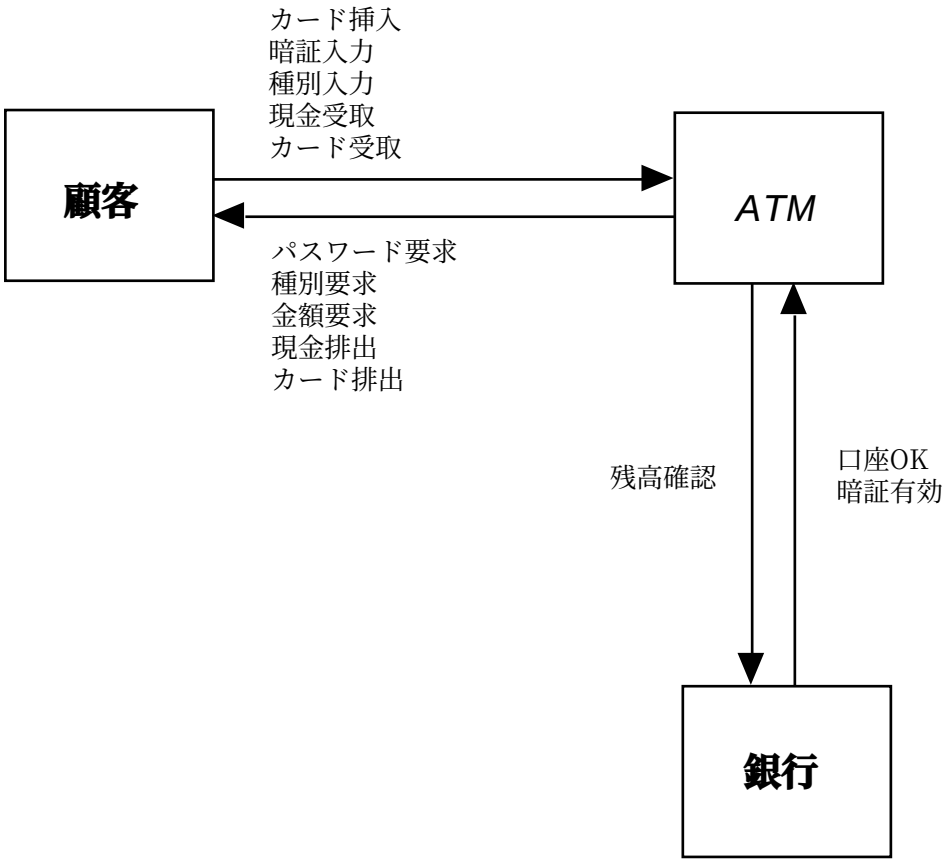
ユーザーインタフェースの設計

- アプリケーションのロジックとユーザーインタフェースを分離する
- ロジックの開発と並行して進められる
- 詳細は不要
- 必要ならプロトタイプ作成

オブジェクト間のイベントを認識し、各シナリオ毎のイベントの軌跡を集める

イベントトレース図





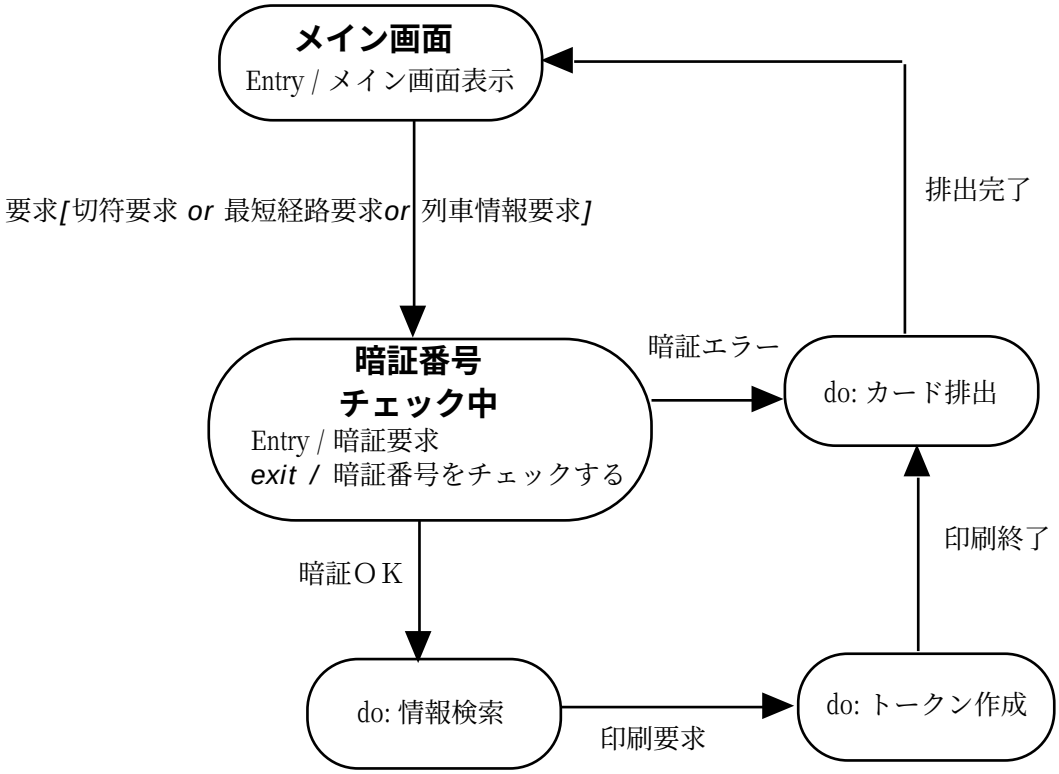
重要な動的振舞いをする各クラス毎に、状態遷移図の作成を行う

- イベントトレース図から開始する
- ２つのイベントの間が状態である
- 状態に名前を付けるのは重要だが、無理に付けなくてもよい
- ループを見つけ、無限ループしないか確認する
- 正常な場合が終わってから、境界ケースとエラーケースを追加する
- そのうち慣れてくれば、イベントトレース図なしでも、状態遷移図を書くことができる

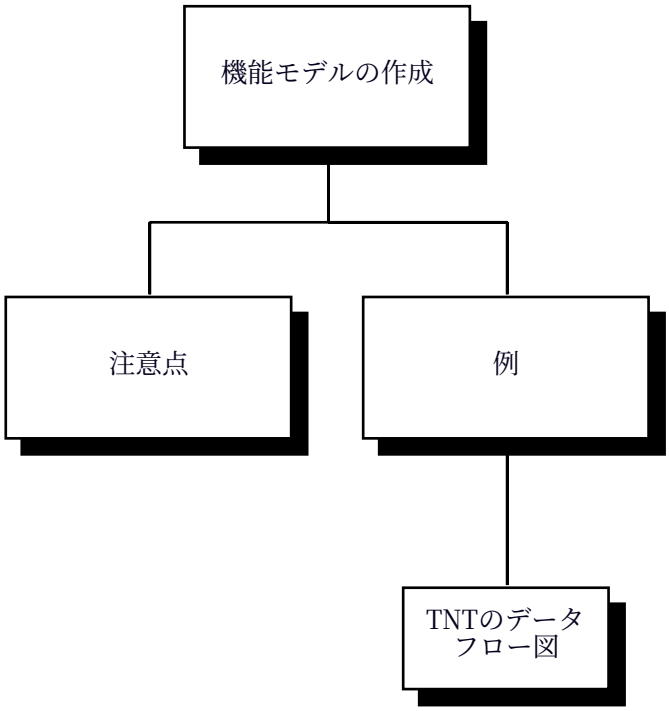
状態遷移図間で共有されるイベントの整合性と完全性をチェックする

- すべてのイベントには送信オブジェクトと受信オブジェクトがある
  - 時には同一のオブジェクト
- 遷移先、遷移元を持たない状態は疑ってかかる
  - 始点か終点になるべきだろう
- 入力イベントの引き起こす効果を順に追って、シナリオと一致するかどうか見る
- オブジェクトは並行に動く
  - 入力が悪いタイミングで起こることによる同期エラーに注意する
- 別々の状態遷移図でイベントが首尾一貫しているかどうか確かめる

クラスTNTに対する状態遷移図



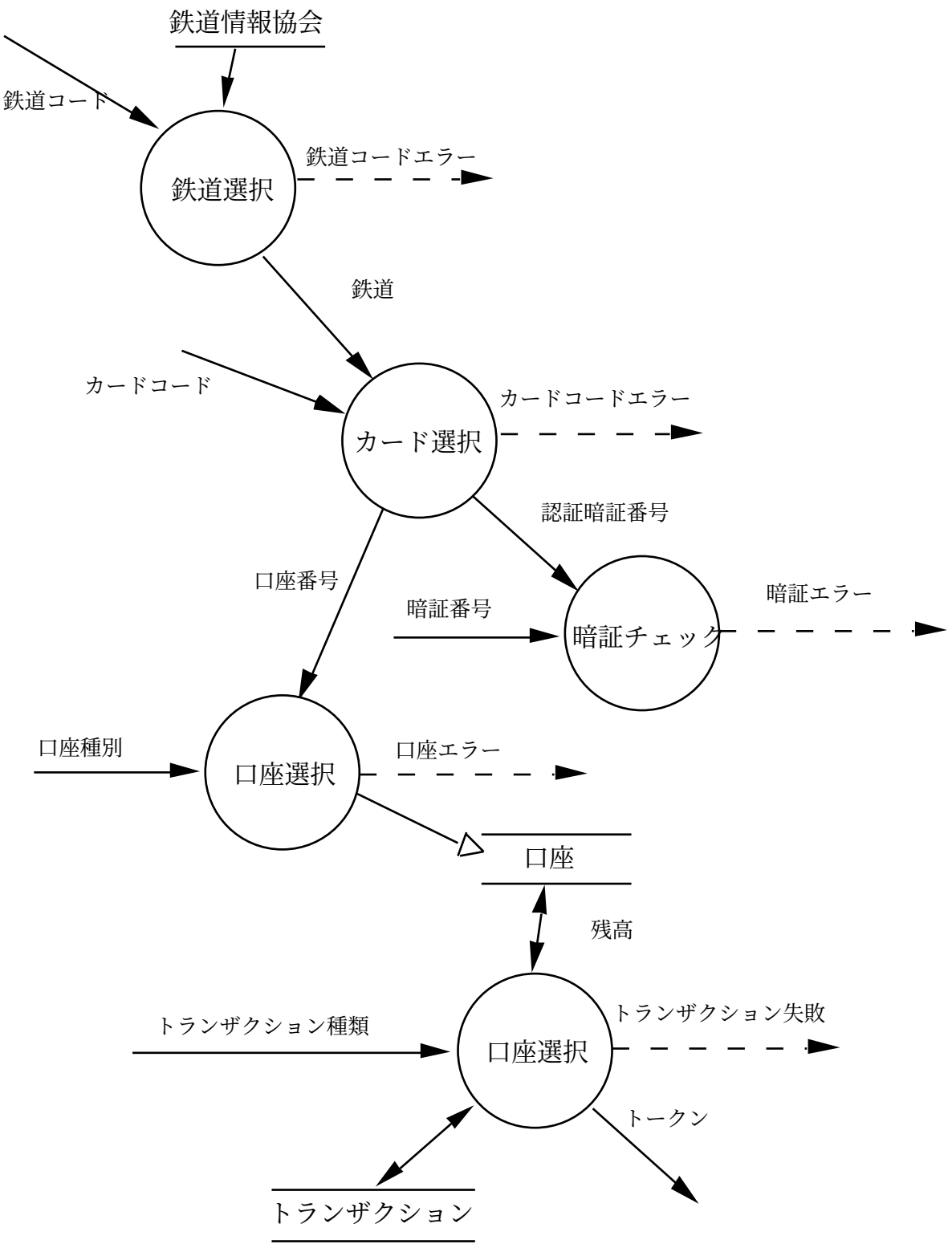
機能モデルの作成



注意点

- オブジェクトモデルと動的モデルを作ってから、機能モデルを作ると良い
- 入出力の値を洗い出す
  - 問題記述文から見つける
  - システムの境界を設定する
- 機能の依存関係を見るために、データフロー図を作成する
  - 階層的に作成する
  - トップレベル（概観図）はプロセスを１つにする
  - 出力から入力に向かって考える
- 各機能が何をするか記述する
  - HowでなくWhatを記述する
  - 宣言的記述の方が手続き的記述より良い
  - なるべく数学的・形式的記述にする
- 制約条件を見つける
  - 入出力関係とは別のオブジェクト間の関数的（値を規定しあう）関係
- 最適化の基準を作る
  - 最適化の要素は、空間と時間などのようにお互いに矛盾する
  - トレードオフの基準を明記する

TNTのデータフロー図



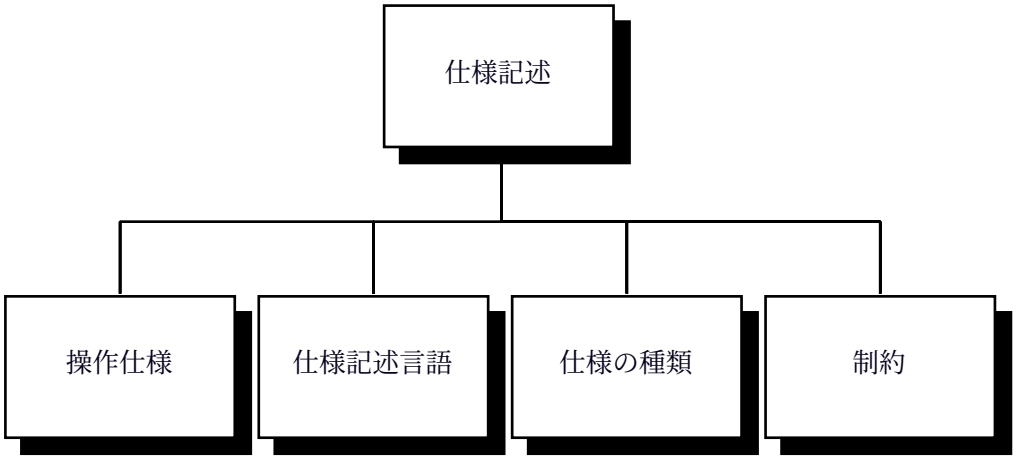
操作の追加

- オブジェクトモデルからの操作
  - 属性やリンクの値を読んだり書いたいるする操作は明示しなくてよい
  - 分析中はすべての属性がアクセスできると見なしてよい
  - "."記法を機能モデルと動的モデルで使うと便利
    - TNT.端末コード
  - 限定付きリンクは索引記法で表す
    - 銀行［銀行コード］.口座［口座番号］
- イベントからの操作
  - イベントが直接イベントとして実現されるかメソッドになるかは実現上の問題なので、分析中はオブジェクトモデルの操作としては表さない
- 動作と活動からの操作
  - オブジェクトモデルに操作として定義する
- 機能からの操作
  - オブジェクトモデルに操作として定義する
  - オブジェクトモデルを横断してアクセスする関数は除く
- 買物リスト操作
  - 直接に要求されていなくとも、実世界との対応から操作を付け加えたいことがある
  - 現在の問題解決のための必要性を越えて、オブジェクト定義の基盤を広げることになる

３つのモデルの検証と改良

- クラス・関連・属性・操作の整合性と完全性をチェックする。
  - 問題記述文と関連する対象領域の知識を３つのモデルと比較し、シナリオによってモデルをテストする
- 基本的なシナリオ以外に、エラー状態も含むより詳細なシナリオを作る。
  - このシナリオでさらに３つのモデルを検証する
- 分析が完成するまで、上記の作業を繰り返す

仕様記述



操作仕様

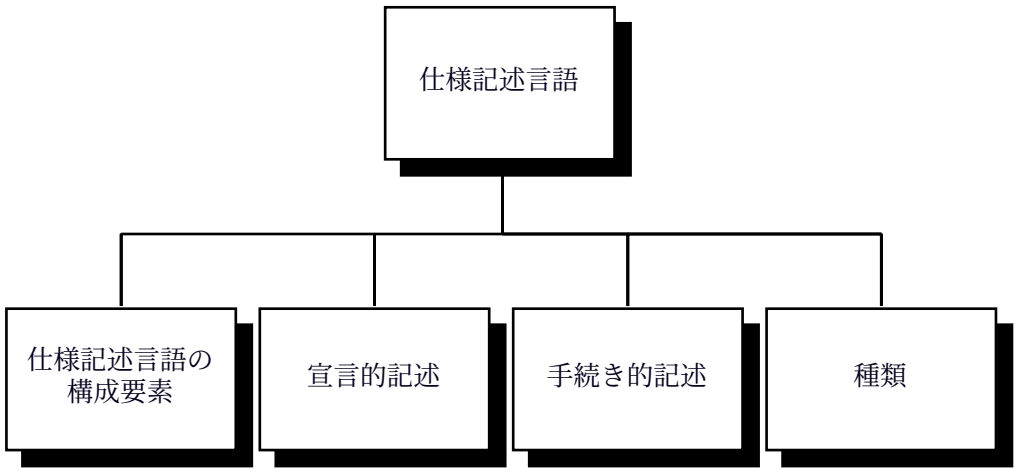
- SA/SDのプロセス仕様に相当する
- いつ（when）とかどうやって（how）を記述しない
- データの変換規則（計算規則）を記述
- 操作の記述はシグニチャと変換と制約（後述）を含む
  - シグニチャは操作のインタフェースを定義する
    - 引数（数、順番、型）
    - 返り値（数、順番、型）
  - 変換は操作の効果を定義する
    - 入力値の関数としての出力値
    - オペランドオブジェクト上の操作の副作用
- 操作の記述例*

操作の記述例*

- 関数
  - ビール注入
- 入力
  - 現在重量、規定重量、初期重量、誤差
- 出力
  - 注入終了
- 変換
  - Precondition
    - 0　＜　現在重量　＜　初期重量
    - 注入終了 = false
  - Postcondition
    - (1 - 誤差) ＊ 規定重量　＜　現在重量　＜　(1 + 誤差) ＊ 規定重量
    - 注入終了 = true
- 制約
  - 注入時間　＜　１秒

仕様記述言語

仕様記述言語は、まだはっきりとは定まっていない



仕様記述言語の構成要素

- 集合
  - ∈⊆⊇⊂⊃∪∩
- 論理
  - 命題論理
  - ∧∨¬⇒⇔
  - 述語論理
  - ∀∃
- 代数

VDM-SLの例

```
types
  生産地型, 消費地型 = token-set;
  地型 = 生産地型 | 消費地型

      inv 地 ≜ is-生産地型(地) ⇔ ¬is-消費地型(地);      // inv = 不変式

  輸送路型::生産:生産地型
            消費:消費地型;

                m
  量型=地型 → R

      inv 量 ≜  ∀n:地型・量(n) > 0
            ∧ Σ[生産地型](生,量) = Σ[消費地型](消,量);

                    m
  単価型, 割当量型 = 輸送路型 → R
```

OBJ3の例

```
obj STACK-OF-NAT is sorts Stack NeStack .
   subsorts NeStack < Stack .
   protecting NAT .

   op empty : -> Stack .
   op push : Nat Stack -> NeStack .
   op top_ : NeStack -> Nat .
   op pop_ : NeStack -> Stack .

   var X : Nat . var S : Stack .
   eq top push(X,S) = X .
   eq pop push(X,S) = S .
endo
```

宣言的記述

```
fun 出張費 (宿泊費, 交通費, 日当, 雑費) =
      合計(宿泊費) + 合計(交通費) + 合計(日当) + 合計(雑費);

fun 合計[] = 0
  ¦ 合計 (n::ns) = n + (合計 ns);  (* - 合計([1,2,3]); val it = 6 : int  *)

    val 宿泊費 : string -> int
    val 大都市 : string -> bool
    val 規定宿泊費 : string * string * bool -> int
    val 日当 : string -> int
    val 出発日日当 : string * Date.datetime -> int
    val 到着日日当 : string * Date.datetime -> int
    val 日当合計 : string * Date.datetime * Date.datetime -> int
fun 宿泊費(肩書) = if 肩書 = "主幹" then 9000
                   else if 肩書 = "主席" then 8500  else 8000;

fun 大都市(都市) = let val 政令指定都市 = ["東京","横浜","名古屋","京都","大阪", ...]
                       fun eq c = if 都市 = c then true else false
                   in  exists eq 政令指定都市 end;

fun 規定宿泊費(肩書, 都市, 領収書) = if not 領収書 then 規定.宿泊費(肩書) div 2
      else if 規定.大都市(都市) then 規定.宿泊費(肩書) + 1000
      else 規定.宿泊費(肩書);

fun 日当(肩書) =  if 肩書 = "主幹" then 3500
           else if 肩書 = "主席" then 3000 else 2500;

fun 出発日日当(肩書, 出発) = let val 出発時 = Time2Year(出発)
           in     if 出発時 > 0.5 then 日当(肩書) div 2 else 日当(肩書) end;

fun 到着日日当(肩書, 到着) =  let val 到着時 = Time2Year(到着)
           in     if 到着時 < 0.5 then 日当(肩書) div 2 else 日当(肩書) end;

fun 日当合計(肩書,出発日時,到着日時) = let val nissu = NumberOfDays(出発日時,到着日時)
                  val 出発 = nthtail(出発日時, 3)
                  val 到着 = nthtail(到着日時, 3)
                  fun 勘定する(日数, 出発, 到着) = case 日数 of
                    1 => 出発日日当(肩書, 出発)
                  ¦ 2 => 出発日日当(肩書,出発) + 到着日日当(肩書,到着)
                  ¦ n => 出発日日当(肩書,出発) + 到着日日当(肩書,到着) + 日当(肩書) * (n-2)
           in     勘定する(日数, 出発, 到着) end;

fun NumberOfDays(Datetime(y,m,d,h,mn,s), Datetime(y2,m2,d2,h2,mn2,s2)) =
  NetNumberOfDays(Datetime(y,m,d,24,0,0),Datetime(y2,m2,d2,h2,mn2,s2)) + 1


出張費([規定宿泊費("主幹","金沢",true) * 4,
      規定宿泊費("主幹","金沢",false)],
      [150,300,14600,1000,1700*4,850,1000,14600,300,150],
      [[日当合計("主幹",Datetime(1993,7,5,9,0,0),Datetime(1993,7,10,13,0,0))],
      []); should be 101250
```

手続き的記述

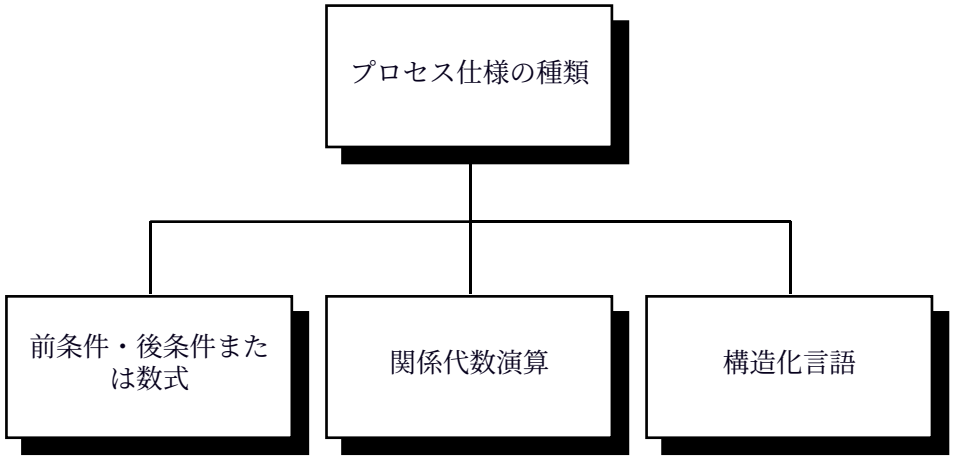
- 出張費 = 0;
  date = 出張開始日;
  while 出張開始日 <= date <= 出張終了日
  　出張費 = 出張費 + 宿泊費?（領収書, 役職, 都市）+ 日当?（ 出発時間, 到着時間, 役職）+ ...;
  　date = date + 1;
  end while
- ...
- 宿泊費?（領収書, 役職, 都市）;
  if 領収書 = なし then
  　　社員規程.宿泊費（役職） / 2
  else if 都市 = 10大都市 then
  　　社員規程.宿泊費（役職） + 1000
  else
  　　社員規程.宿泊費（役職）
  end if
- 日当?（ 出発時間, 到着時間, 役職）;
  if 出発時間 > 10:00 or 到着時間 < 14:00 then
  　　日当 = 社員規程.日当（役職） / 2
  else
  　　日当 = 社員規程.日当（役職）

種類

- VDM, RAISE
- Z
- OBJ3
- CSP
- 関数型言語
  - ML

仕様の種類

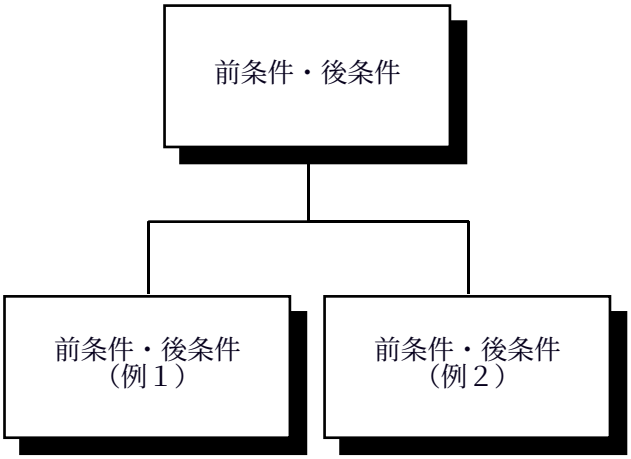
推奨される記述方法は以下のとおり（左の方がより厳密）



前条件・後条件または数式

集合論や数式の記号を使い、プロセスの処理前の状態と処理後の状態を記述する

最も推奨される記述法だが、訓練がいる



前条件・後条件（例１）

- 関数
  - ビール注入
- 入力
  - 現在重量、規定重量、初期重量、重量誤差
- 出力
  - 注入終了
- 変換
  - Precondition
    - ０　＜　現在重量　＜　初期重量
    - 注入終了　＝　false
  - Postcondition
    - (1 - 重量誤差) ＊ 規定重量　＜　現在重量　＜　(1 + 重量誤差) ＊ 規定重量
    - 注入終了　＝　true
  - 制約
    - 注入時間　＜　１秒

前条件・後条件（例２）

- 関数
  - 前のスピードに戻る
- 入力
  - 回転率、スロットル位置、回転率設定値、スピード制御フラッグ、回転率誤差
- 出力
  - スロットル制御、前のスピードに到達
- 変換
  - Precondition 1
    - スピード制御フラッグ　＝　「前のスピードに戻す」　かつ　スピード制御フラッグがセットされてからの時間　＜＝　0.5秒
  - Postcondition 1
    - スロットル制御　＝　スロットル位置
  - Precondition 2
    - スピード制御フラッグ　＝　「前のスピードに戻す」　かつ　スピード制御フラグがセットされてからの時間　＞　0.5秒
  - Postcondition 2
    - (1 - 回転率誤差) ＊ 回転率　＜　回転率設定値　＜　(1 + 回転率誤差) ＊ 回転率
    - 前のスピードに到達　＝　true
  - 制約2
    - 回転率の秒あたり増加率　＝　0.1 ＊（回転率設定値　－　回転率）

関係代数演算

- Coddの関係代数演算を流用する
- SQLなどの構文を流用する
- データ処理系には有効
- 抽出(Select)
  - select 住所録.all when 住所録.名前 ＝ "佐原"
- 射影(Project)
  - project (住所録.名前, 住所録.電話) when 住所録.名前 ＝ "ペレ"
- 結合(Join)
  - join (住所録.名前, 会社.所属) when 住所録.名前 ＝ 会社.社員名

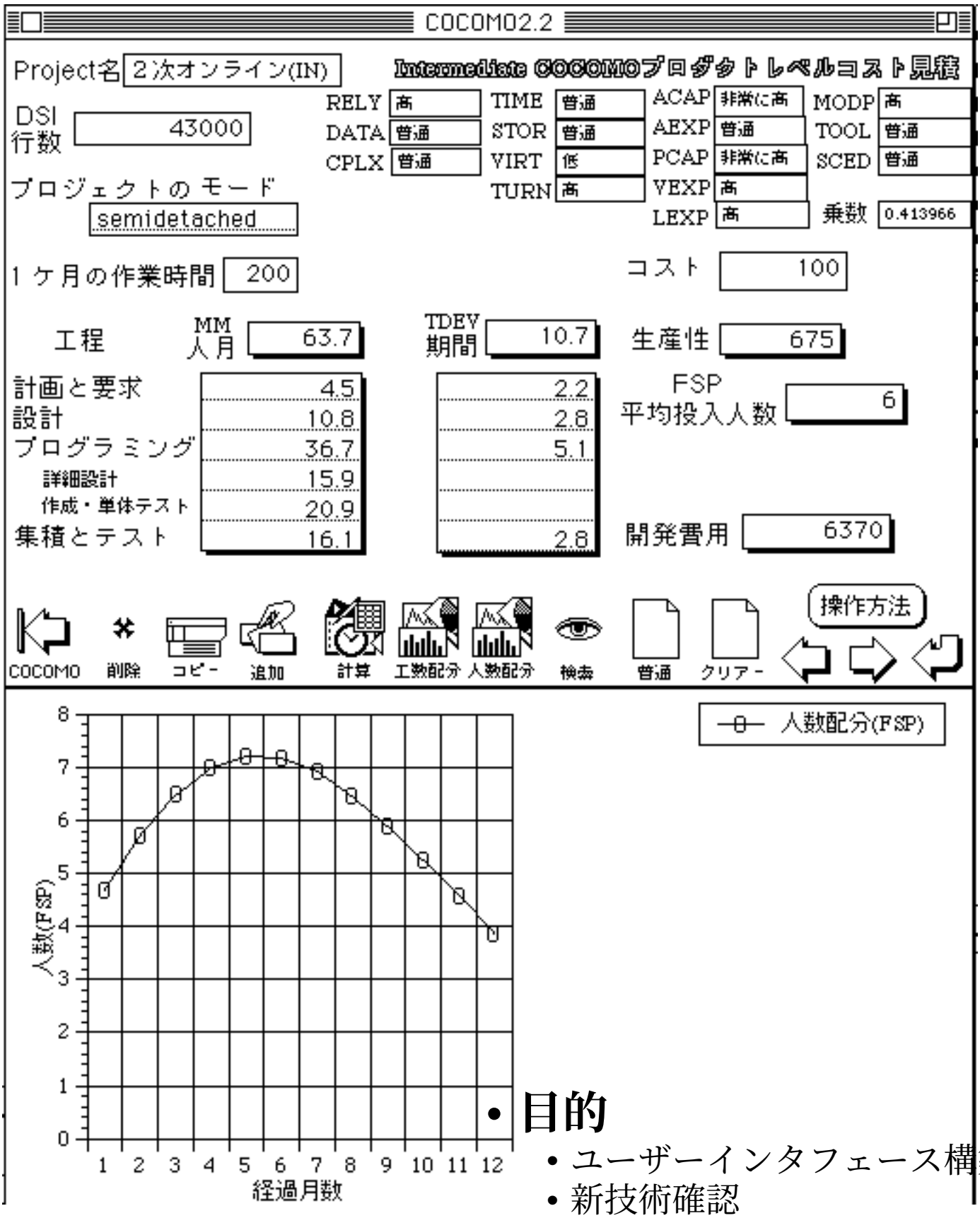
構造化言語

- 自然言語と構造化文を組み合わせて仕様を記述する
- 構造化文は高級言語から借用する
- 例
  - if 投入金額積算値　＞　総購入金額　then
    　投入金額有効処理
    else
    　投入金額不足処理
    end if
    case (総購入金額) of
    110:
    　　ランプ１１０を点灯
    120:
    　　ランプ１１０を点灯
    　　ランプ１２０を点灯
    150:
    　　ランプ１１０を点灯
    　　ランプ１２０を点灯
    　　ランプ１５０を点灯
    end case

制約

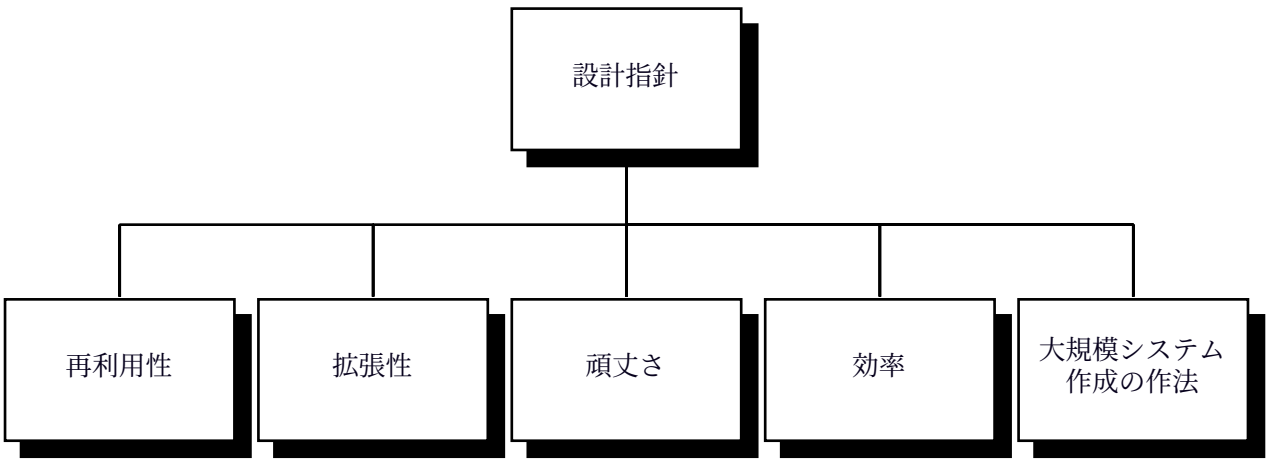
- 同じ時点の２つのオブジェクトの関係を示すか、異なる時点での同じオブジェクトの異なる値を示す
- 全体関数か部分関数として表される
  - 全体関数は、一つの値が他の値によって完全に記述される関数
  - 部分関数は、一つの値が他の値によって制限されるが、完全には記述されない関数
- オブジェクトモデルでは、オブジェクト間の依存関係を示す
- 動的モデルでは、異なるオブジェクトの状態またはイベントの関係を示す
- 機能モデルでは、操作の制約を示す
- 時間を越えて変わらない関係を不変式として書くとうまくいくことがある

プロトタイピング



- **目的**
  - ユーザーインタフェース構築
  - 新技術確認
- **ツール**
  - Smalltalk & 明
  - HyperTalk2.2

設計指針



再利用性

- 再利用性のための作法
  - 強度を高める
  - 小さくする
  - 一貫性を保つ
  - 方針（Policy）と実現（Implementation）を分ける
  - 均一にカバーする
    - 当面必要なものだけでなく、全ての組み合せに対するメソッドを書く
      - 例えば、リストの最後への挿入を書くのだったら、リストの頭への挿入も書くべき
  - できるだけ汎用にする
  - 大域情報を避ける
  - モードを避ける
- 継承の利用
  - ファクタリング
    - サブルーチンではなくファクタリングを使う
    - 共通コードをくくり出して、親クラスに書く
  - 委譲
  - 外部のコードをカプセル化する

拡張性

- クラスをカプセル化する
- データ構造を隠す
- 複数のリンクとメソッドをたどるのを避ける
  - 代わりに、隣のオブジェクトの操作を呼び出す
- オブジェクトの型による分岐を避ける
  - 代わりにメソッドを呼ぶ
- 公開操作と私的操作を分ける

頑丈さ

- エラーに備える
  - アプリケーションエラー
    - 分析段階で考慮する
  - 低いレベルのシステムエラー
    - 優雅に死ぬことを考える
      - 必要な情報を保存し、後で回復できるようにする
  - プログラムのバグに対する防御的プログラミングをする
- プログラムを走らせてから最適化する
  - ミクロの最適化に走らない
- 引数を検証する
- あらかじめ定義された限界を作らない
- デバッグと効率測定の仕掛けをプログラムに装備する

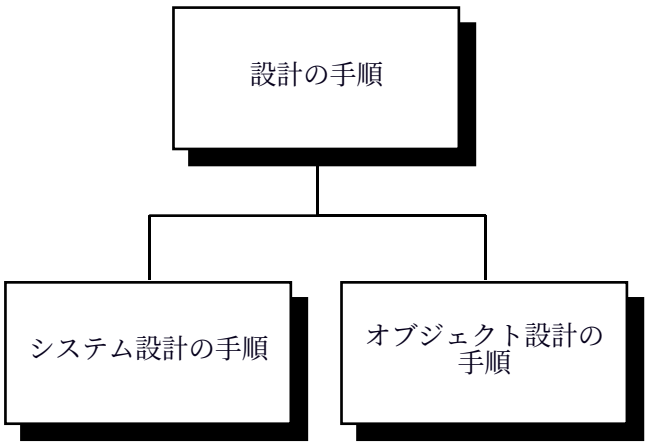
効率

- OO言語は遅いという評判がある
  - 初期のSmalltalkやLispベースの言語がインタープリタだったため
    - 動的束縛が遅い原因のひとつ
  - 現在は、コンパイラーやより速いインタープリタが提供され、選択の幅が広がっている
    - メソッドキャッシュとかハッシュ表の使用により、動的束縛のコストは一定に抑えられるようになった
    - 強い型付けを持つC++のような言語では、メソッド呼び出しのコストは一つの構造体の参照と同じで、手続き呼び出しとほとんど変わらない
    - 十分な情報が与えられれば、ほとんどのメソッド呼び出しは動的にならず、静的に行うことができる
      - プログラマーが重複定義を宣言する
        - C++のvirtual関数やCLOSの汎化関数の方式
      - 最適化パスを使う
        - プログラム全部を解析して多重定義がないメソッドを捜し、それらのメソッドを手続き呼び出しと同じように行う
          - Eiffelの方式
- 成熟したクラスライブラリーを持つOO言語では、非OO言語より速いこともある
  - OO言語のオーバーヘッドより、成熟したクラスのデータ構造やアルゴリズムの実現による効率向上の方が大きい
    - 例えばほとんどのプログラマはハッシュ表やバランス木を使おうとしないが、良いクラスライブラリーではこれらが用意されている

大規模システム作成の作法

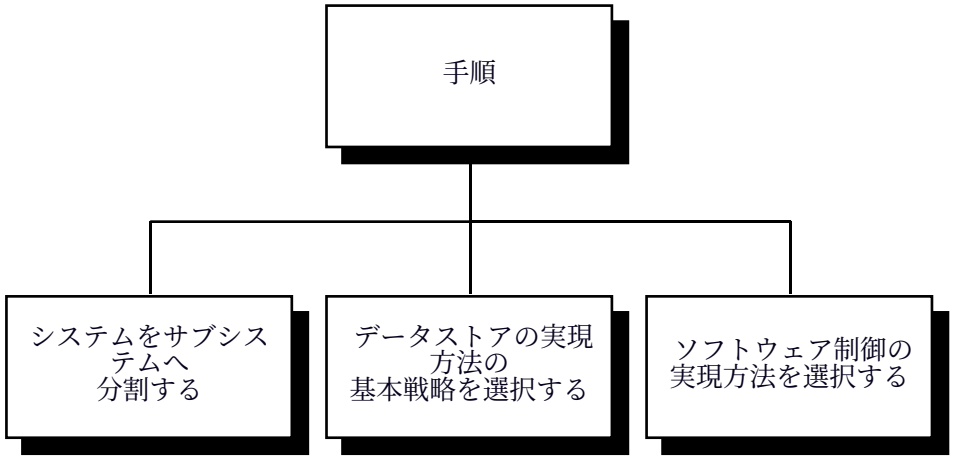
- はやまってプログラミングしない
- メソッドを理解しやすくする
  - メソッドを小さくし、強度を高める
- メソッドを読みやすくする
  - 変数名を意味のあるものにする
- オブジェクトモデルと同じ名前を使う
- 名前を注意深く選ぶ
  - 意味的に異なる操作に同じ名前を付けない
- プログラム標準を使う
- モジュールにまとめる
- クラスとメソッドのドキュメントを書く
- 仕様を公開する

設計の手順



システム設計の手順

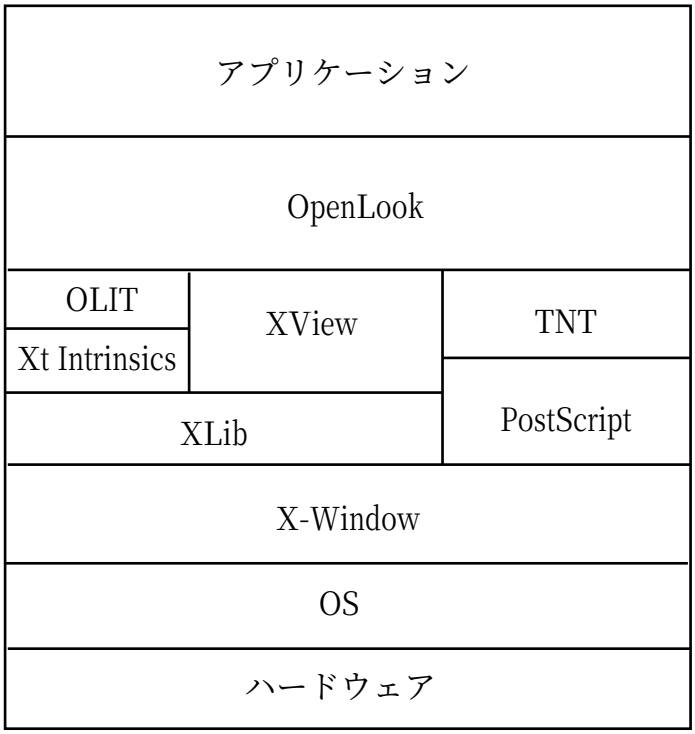
ここでは、システムの上位レベルの構造（システムアーキテクチャ）を決める。



システムをサブシステムへ分割する

- サブシステム
  - クラスと関連と操作とイベントと制約の集まりである
  - インタフェースが少なく、明確にされたサブシステムに分割する
  - ２０個のサブシステムは多分多すぎる
  - 再下層のサブシステムをモジュールと呼ぶ
- サブシステム間の関係は、顧客-供給者か友人-友人になる
  - 顧客-供給者関係
    - 顧客は供給者のインタフェースを知っているが、供給者は顧客のインタフェースを知る必要がない
  - 友人-友人関係
    - 互いに相手のインタフェースを知っている
  - できるだけ顧客-供給者関係に分解することが望ましい
- レイヤーとパーティション*

レイヤーとパーティション*



- **レイヤー**
  - クローズドアーキテクチャ
    - 各レイヤーは、すぐ下のレイヤーの言葉だけで構成されている
  - オープンアーキテクチャ
    - 一つのレイヤーが、下位のどのレイヤーでも使う可能性がある
    - 効率は良いが、頑丈さに欠ける
- **パーティション**
  - システムの垂直方向の分割

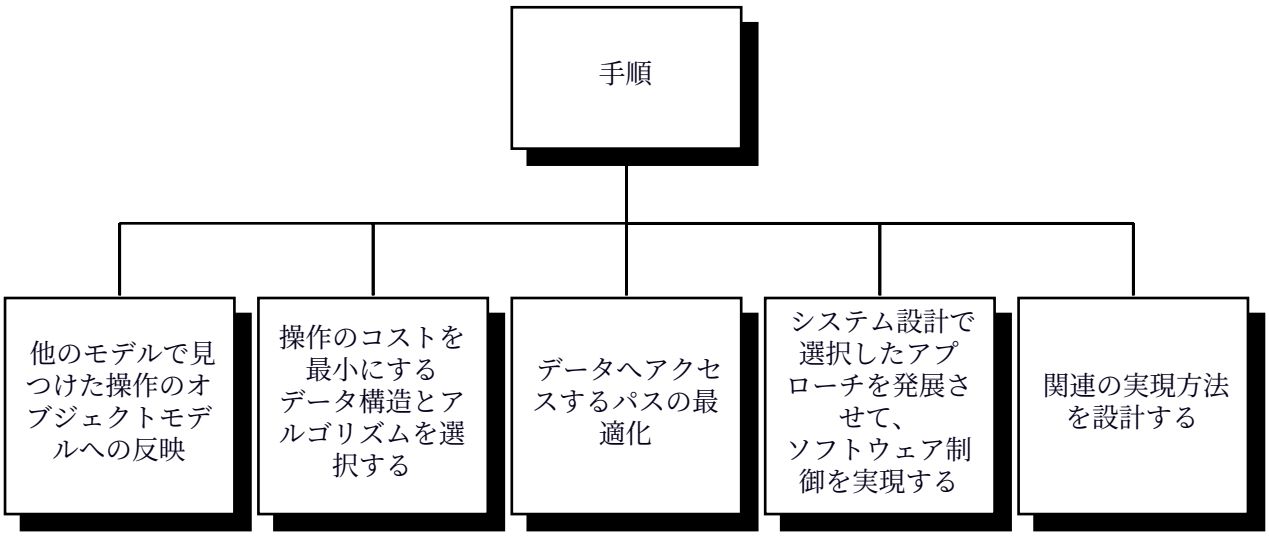
データストアの実現方法の基本戦略を選択する

- データストアの実現方法の基本戦略を選択する
  - データ構造か、ファイルか、データベースかなどを選択する
    - DBMSを使うべきデータ基準
      - 複数ユーザーが、細かいレベルまでアクセスする要求があるデータ
      - DBMSコマンドで効率的に管理できるデータ
      - 多くのハードとOSに持って行くデータ
      - １つ以上のアプリケーションからアクセスする必要のあるデータ
    - ファイルを使うべきデータの基準
      - 大量だがDBMSに入れにくいデータ（グラフィックビットマップデータなど）
      - 大量で密度の低いデータ（アーカイブファイル、ダンプなど）
      - データベースにまとめられる前の生データ
      - 一時的なデータ
    - データベースを利用することの利益
      - 基盤的機能
        - クラッシュからの回復、複数ユーザー間の共有、複数アプリケーション間の共有、データ分散、一貫性、拡張性、トランザクションのサポートなど多数の基盤的機能が提供されている
      - すべてのアプリケーションに対する共通のインタフェース
        - 各アプリケーションは、情報のうち必要な一部分をアクセスし、残りは無視する
      - 標準のアクセス言語
        - SQL言語は、ほとんどの商用関係データベースシステムでサポートされている
    - データベースを利用することの不利益
      - 効率が悪い
      - 高度なアプリケーションに対する機能が十分でない
      - プログラミング言語とのぶざまなインタフェース

ソフトウェア制御の実現方法を選択する

- プログラム内に状態を持つ
  - waitなどを使い、プログラムカウンターや手続き呼び出しのスタックや局所変数で「状態」を定義する
  - ほとんどの言語で実現できる
  - オブジェクト間の本質的な並行性を、制御の流れの連続にマップするため、分かりにくい
- 状態マシンを直接実現する

オブジェクト設計の手順



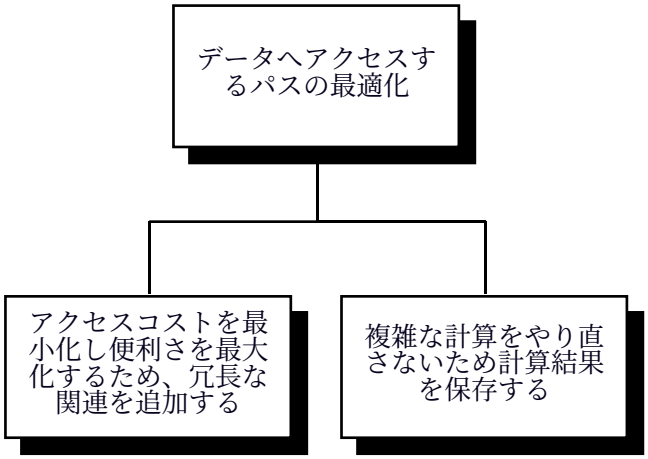
他のモデルで見つけた操作のオブジェクトモデルへの反映

- 分析段階のオブジェクトモデルは主要な操作以外記述していないので、その他の操作を見つける
  - 機能モデルのプロセスから操作を見つける
    - プロセスが入力データーフローから値を取り出していたら、入力データフローが目標のオブジェクト
    - プロセスが同じ型の入力フローと出力フローを持ち、出力フローが入力フローを更新したものであるとき、入出力フローが目標のオブジェクト
    - プロセスがいくつかの入力フローから１つの出力の値を組み立てているとき、出力クラスののクラス操作とする
    - プロセスが１つの入力か出力を、データストアーかアクター（Actor）との間に持っているとき、そのデータストアかアクターが目標のオブジェクト
  - 制御の実現方法に依存するが、動的モデルのイベントから操作を見つける
- 分析段階の論理モデルから、物理モデルの作成に移行する

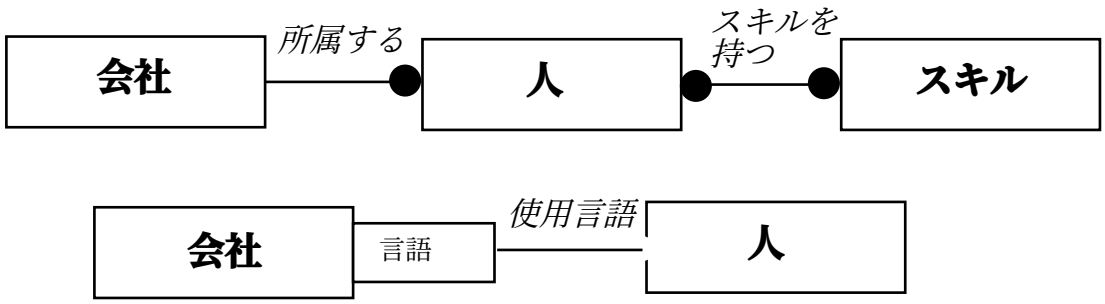
操作のコストを最小にするデータ構造とアルゴリズムを選択する

- 「データ構造とアルゴリズム」といった本から捜す
- 既存のアルゴリズムを書いたがなければ、専門家に聞く
- それでも分からなければ、考える
- アルゴリズムの選択にあたって考えること
  - 計算の複雑さ
    - 現実のアプリケーションでも事実上計算不可能なケースがあり得る
      - 組み合わせ問題が多い
        - 東京証券取引所のトラブル
      - データ構造を間違うと
        - 証券情報システム
  - 実現の容易性と理解しやすさ
    - 効率とか空間の面から特に問題にならない操作は、できるだけ単純に実現する
  - 柔軟性
    - 最適化したアルゴリズムは一般に読みにくく変更しづらい
    - まず最も単純なアルゴリズムで実現し、計測してから、必要な操作だけ最適化する
    - システム全体の効率に関係する操作は全体の２～３％
    - ミクロの効率化を図ると、全体的には効率化されないこともある
  - オブジェクトモデルの微調整
    - アルゴリズムの効率化のためのクラスを設定することがある
    - １回だけ計算し後はコピーする、など

データへアクセスするパスの最適化

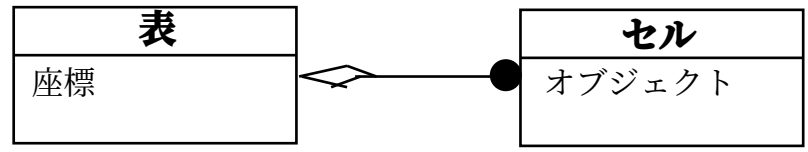


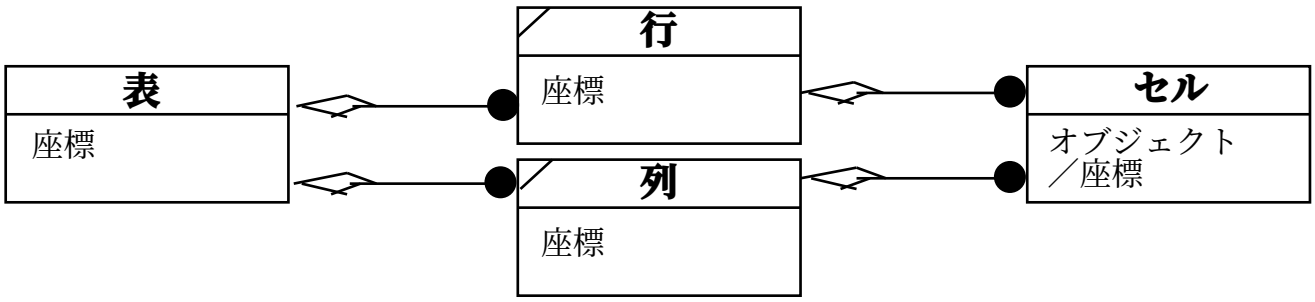
アクセスコストを最小化し便利さを最大化するため、冗長な関連を追加する

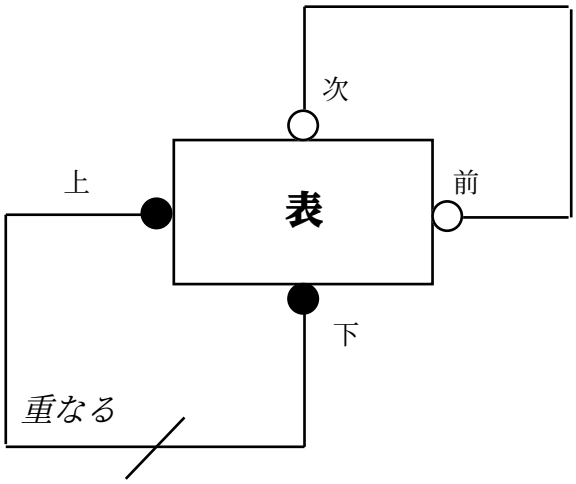


- **ある情報を得るためにどの関連をたどる必要があるか？**
- **双方向にたどらなければならない関連は何か？**
- **１方向だけたどれば良い関連は何か？**
  - １方向だけの関連は効率がよい
- **操作はどれくらい呼ばれるか？そのコストは？**
- **「多」の関連のパス沿いにどれくらい広がりがあるか？扇型の最後に位置するクラスのアクセス回数はどれくらいか？**
- **最後のクラスのオブジェクトがどれくらい「ヒット」するか？**
  - ほとんどのオブジェクトが「ヒット」しないのなら、単純ループは非効率

複雑な計算をやり直さないため計算結果を保存する

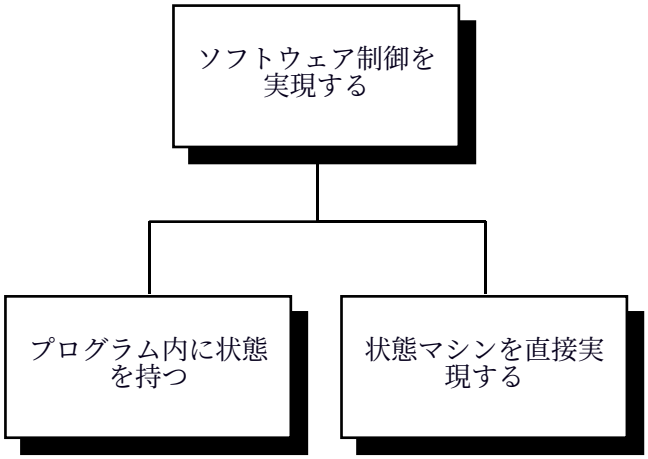






- **再計算しないため、新しいオブジェクトやクラスが必要なことがある**

システム設計で選択したアプローチを発展させて、ソフトウェア制御を実現する



プログラム内に状態を持つ

- 任意の有限状態マシンはプログラムとして実現できる
  - 安易な方法としてはgoto文を使う
  - プログラムの入れ子構造を使うのは難しいが、モジュール性を維持できる
    - 状態遷移図で主な制御のパスを見つける。初期状態から始め、正常なイベントをトレースする。このパスの状態名を順番に書く。これがプログラムの文の並びになる
    - 主なパスから分岐し後で合流するパスを認識する。これがプログラムの条件文になる
    - 主なパスから分岐し前に合流する後向きのパスを認識する。これがプログラムのループになる。互いに横切らない複数の後向きパスは、ループの入れ子になる。入れ子にならない後向きのパスは、全部のelse文に該当しない場合のgoto文になるが、滅多にない
    - 残りの状態と遷移は例外条件に対応する。言語で支援されるエラーサブルーチンや例外処理で実現されるか、状態フラグとして実現される。goto文を使って入れ子構造を抜け出る形でエラー処理を行ってもよいが、どうしても必要なとき以外避けるべきだろう

状態マシンを直接実現する

- 状態マシンエンジンと呼ぶべきクラスを利用するか作る
- UNIXのyaccやlexのような状態マシン生成ツールもある
- ユーザーインタフェースのプロトタイピングツールとして状態マシンを生成するものもある
- オブジェクト指向言語を使えば、状態マシンを作るのはさほど難しくない

関連の実現方法を設計する

- 関連の方向を分析する
  - 片方向関連
    - 多重度が１ならただのポインターで実現する
    - 多重度が多ならポインターの集合で実現する
    - {ordered}という制約があるなら、リストで実現する
    - 限定子は、辞書を使って実現する
  - 両方向関連
    - 片方向だけ実現し、逆方向の参照は検索を行う
      - 逆方向の参照が参照が極端に少ないとき
    - 片方向の関連を２つ作る
      - 片方の属性が変化したら、リンクの一貫性を保持するため、他方の属性も更新しなければならない
      - 更新より参照が多いとき
    - 関連を別個の関連オブジェクトとして実現する
      - 関連したオブジェクト同士の組の集合である
      - 限定子はオブジェクト同士と限定子の３つ組の集合である
      - 関連オブジェクトを辞書として実現してもよい
      - ほとんどのインスタンスが関連しないようなまばらな関連では、関連オブジェクトは領域を節約するのに有効
- リンク属性
  - １対１の場合、どちらかのオブジェクトの属性にしてしまう
  - １対多の場合、多側のオブジェクトの属性にしてしまう
  - 多対多の場合、各々のインスタンスがリンクとリンク属性を表すようなクラスを別個に作る

クラスライブラリー

- 汎化ライブラリー
  - 多くのOO言語は役に立つ汎化ライブラリーを提供する
    - Smalltalk
    - MacApp
    - Think C Class Library
    - NeXT
  - それ自身役に立つし、特定のニーズにはサブクラスを作って対応する
  - クラスライブラリーのおかげで多くの構成要素を何回も実現しなくて済む
- コンテナークラス
  - 最も役に立つのは、汎用データ構造を実現したクラスライブラリーである
    - 集合・動的配列・リスト・キュー・スタック・辞書・木・その他
- 「関連」クラス
  - 関連を実現するいろいろなクラスライブラリーも用意されるべき
- デバイス独立の抽象化インタフェースライブラリ
  - ストリーム・入力デバイス・ディスプレイデバイス・ファイルシステム
- コンカレントプロセスクラス
- ユーザーインタフェースクラス
  - ウィンドウシステム毎に
- 文字列の取り扱いクラス

開発プロジェクトにおける留意点

- スーパーマンの育成
  - 要求仕様記述の専門家養成
    - OOA
    - モデル化能力
    - 仕様記述言語の知識
      - 離散数学
  - 再利用可能クラスライブラリー作成の専門家養成
    - OOD
    - データ構造とアルゴリズムの専門知識
    - 仕様からプログラムへの段階的詳細化技術
    - 抽象化能力
    - オブジェクト指向言語の知識
- プロジェクトに先行したクラスライブラリー作り
  - 対象問題領域をOOAで仕様化
  - 対象問題領域をライブラリー化
- 優れた開発環境

参考文献(OOA/OOD)

- J.Rumbaugh, M. Blaha , W.Premerlani, F.Eddy, and W.Lorensen. 羽生田訳. オブジェクト指向方法論：ＯＭＴ. トッパン, 1992
  - 現時点で最も完成されたOOA/OOD技法OMTの教科書
- J.Rumbaugh, M. Blaha , W.Premerlani, F.Eddy, and W.Lorensen. Solution Manual Object-Oriented Modeling and Design. Prentice Hall, 1991
  - 上の本の解答集
- 青木淳. オブジェクト指向システム分析設計入門. ソフト・リサーチ・センター, 1992
  - OOA/OOD/OOPの分かりやすい解説
- Peter Coad, Edward Yourdon. 羽生田 監訳. オブジェクト指向分析（ＯＯＡ）［第２版］. トッパントッパン, 1993
  - OOA/OODの教科書
- Peter Coad, Edward Yourdon. Objet-Oriented Design. Prentice Hall International, 1991
  - OODの教科書
- Peter Coad, Jill Nicola. Object-Oriented Programming. Prentice Hall, 1993
- Grady Booch. Object-Oriented Design With Applications. Benjamin/Cummings, 1990
  - AdaよりのOOD手法
- Sally Shlaer, Stephen J. Mellor. 本位田真一, 山口享 訳. オブジェクト指向システム分析　上流CASEのためのモデル化手法. 啓学出版, 1990
  - OMT, Coad法の元になったOOSA手法の教科書
- Sally Shlaer, Stephen J. Mellor. 本位田真一, 伊藤潔 監訳. 続・オブジェクト指向システム分析　オブジェクトライフサイクル. 啓学出版, 1992
  - 上の本の続編。状態モデルに焦点を当てている。
- Ivar Jacobson.Object-Oriented Software Engineering A Use Case Driven Approach. Addison-Wesley, 1992
  - OOSE手法の教科書
- Michael A. Jackson. 大野, 山崎 監訳. システム開発JSD法. 共立出版, 1989
  - JSD法の教科書
- Edited by John McDermid著. 訳. Software Engineer's Reference Book. Butterworth Heinemann, 1991
  - ソフトウェア工学・科学について網羅したソフトウェア技術者向けのハンドブック

仕様記述

- 山崎利治著.ウィーン開発技法VDM. 日本ユニシス, 「技法」1993年2月号, 1993
- The RAISE Language Group著. The RAISE Specification Language. Prentice-Hall, 1992
- Cliff B. Jones著. Systematic Software Development using VDM Second Edition. Prentice-Hall, 1990
- B.ポター, J.シンクレア, D. テイル著. 田中武二 監訳訳. ソフトウェア仕様記述の先進技法-Z言語. トッパン, １９９３
- 中川 中著. 代数仕様記述言語OBJ3. SRA, 1993
- C.A.R. Hoare著. 吉田信博訳. ホーア CSPモデルの理論. 丸善, １９９２