

オブジェクト指向 システム分析 / 設計 Q & A



- 佐原伸
 - E-Mail
 - sahara@sra.co.jp
 - URL
 - http://www.sra.co.jp/people/sahara
- S R A オブジェクト指向グループ
 - URL
 - http://www.sra.co.jp/

目次

- オブジェクト指向の考え方
- オブジェクト指向方法論
- オブジェクト指向分析
- オブジェクト指向設計
- オブジェクト指向プログラミング
- オブジェクト指向プロジェクトの管理
- オブジェクト指向の教育

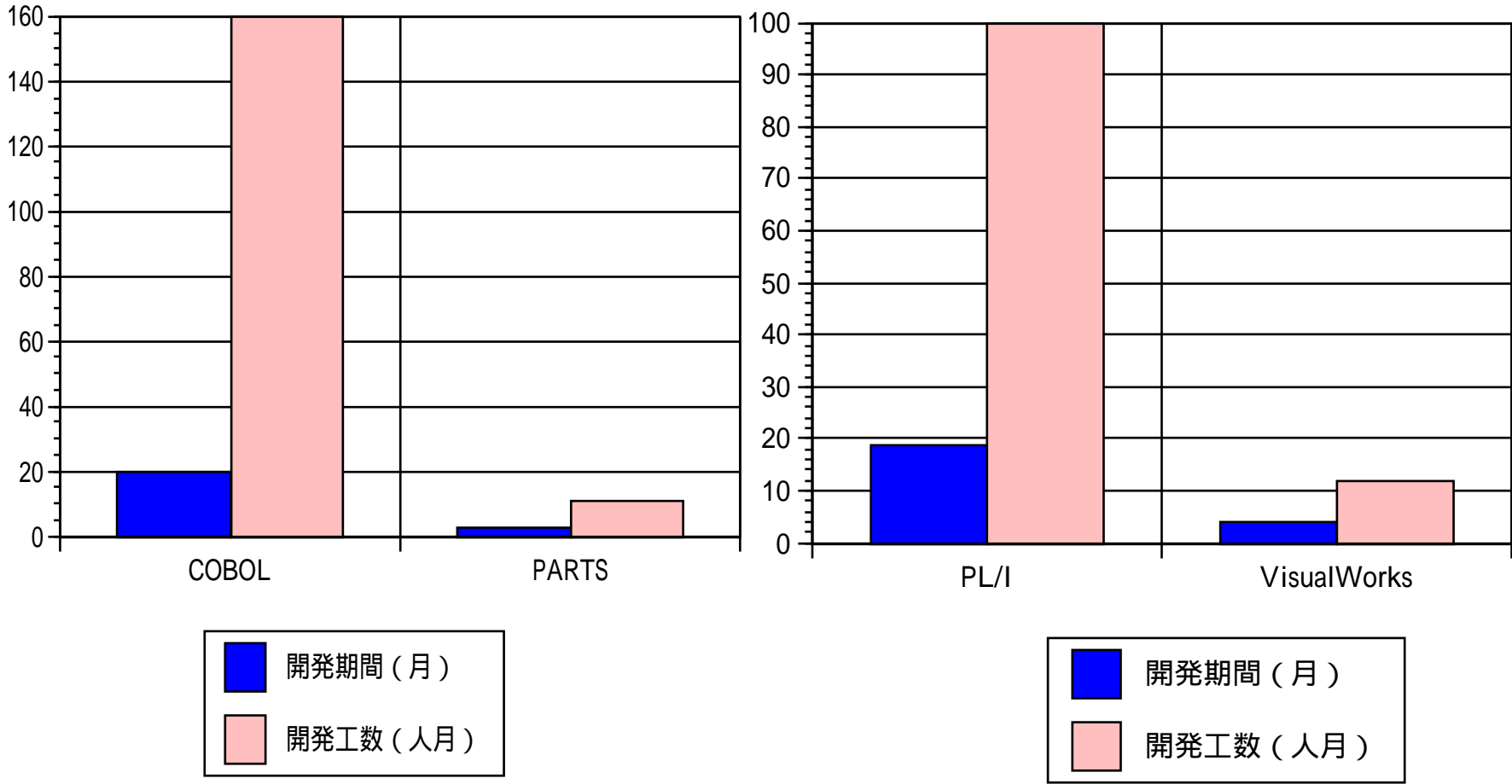
オブジェクト指向の考え方

- なぜオブジェクト指向なのか？
- オブジェクト指向は、実際に使われているのですか？
- オブジェクト指向の基本概念
- オブジェクト指向の位置付け

なぜオブジェクト指向なのですか？

- 経済的理由
- 哲学的理由
- ソフトウェア工学的理由

経済的理由



哲学的理由

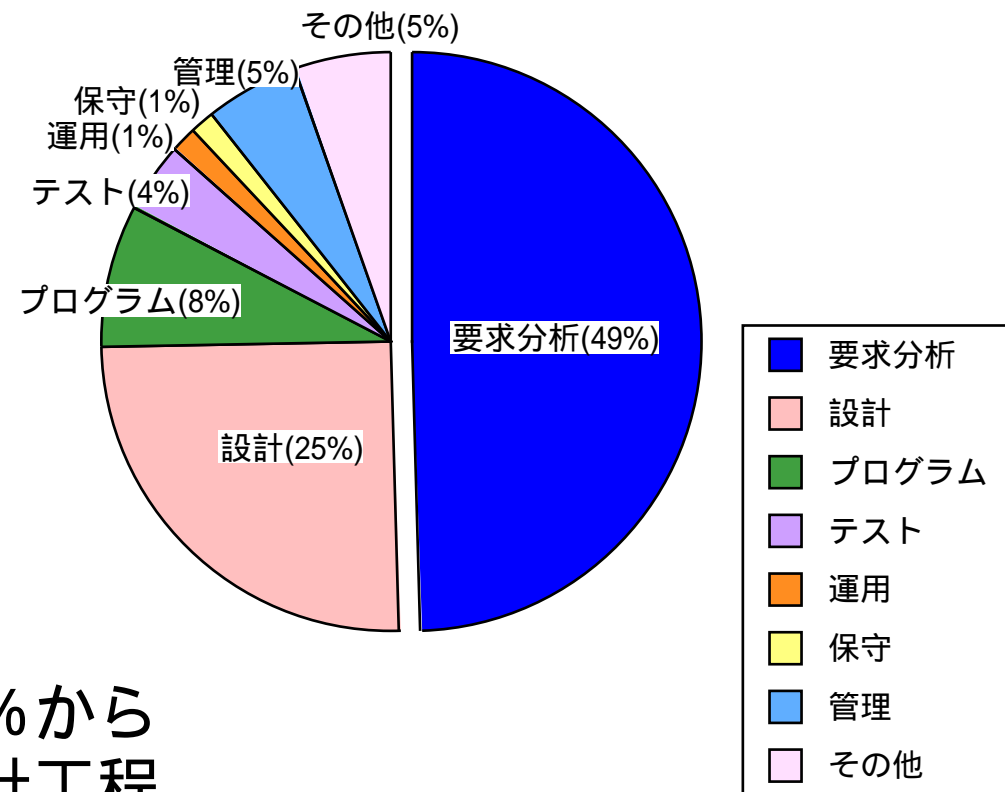
哲学的理由

	Instanc e	クラ ス	関 連	汎 化	集 約	属 性	操 作	Modul e	状態遷 移	DF D
1 抽象化	-	-	-	-	-	-	-	-	-	-
a 手続きの抽象化							X			
b データの抽象化	X	X				X	X			
2 カプセル化	X	X				X	X			
3 継承				X		X	X			
4 組み合わせ (関連)			X							
5 メッセージ										
6 組織化の方法	-	-	-	-	-	-	-	-	-	-
a 「もの」と属性	X	X				X				
b 全体と部分					X			X		X
c	X	X		X		X	X			
7 階層								X		X
8 ふるまいの分類	-	-	-	-	-	-	-	-	-	-
a 直接原因による分類							X		X	X
b 時間推移による分類							X		X	
c 機能類似による分類							X		X	

ソフトウェア工学的理由

- 現状
 - 上流工程のミス
 - ユーザー要求の誤解
- ソフトウェア工学からの要求
 - 外的品質要因
 - モジュール性の原則
 - 再利用へのアプローチ

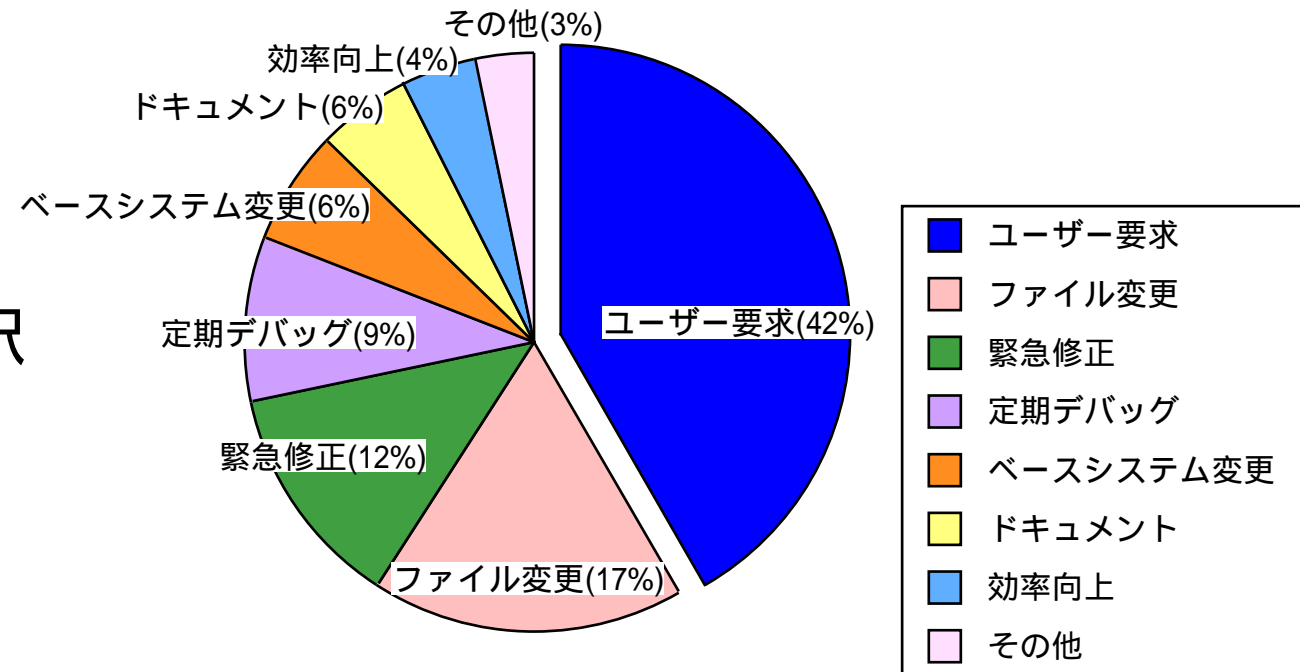
上流工程のミス



- 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

ユーザー要求の誤解

• 保守作業の内訳



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

外的品質要因

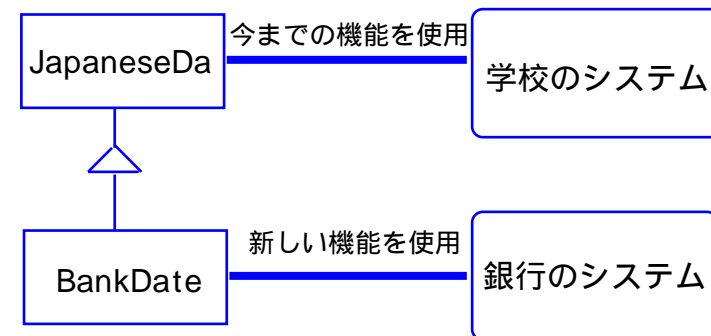
- 正確さ
 - 要求された通りに仕事を行う能力
- 頑丈さ
 - 異常な状態においても機能する能力
- 拡張性
 - 仕様の変更に容易に適応できる能力
- 再利用性
 - 新しい応用にどの程度再利用できるかを示すもの
- 互換性
 - ソフトウェア相互の組み合わせやすさ

モジュール性の原則

- 言語としてのモジュール単位
- 少ないインタフェース
- 小さいインタフェース
- 明示的なインタフェース
- 情報隠蔽
- 開放 / 閉鎖の両立

- 以下を同時に満たさなければならない

- モジュールが拡張可能である（開放）
- モジュールが他のモジュールから使用できる（閉鎖）



再利用への アプローチ

- 再利用へのアプローチ
 - ソフトウェア工学の目標の一つ
 - 繰り返されるプログラミング
 - 単純なアプローチ
 - 再利用可能なモジュール構造の要件

繰り返されるプログラミング

- なぜハードウェアのように再利用できないのか？
- 表の検索
 - いつも、同じような検索プログラムを書いていないか？
- 再利用は、今までの技術では難しい
 - 表検索のパターンはほとんど同じ
 - 整列していない配列、整列済みの配列、整列していない連結リスト、整列済みの連結リスト、順次ファイル、二分木、ハッシュ表
 - 微妙な違い
 - 表の要素の型、最初の位置、次の位置への進み方、データのばらつきの違い

単純なアプローチ

- ソースコードの再利用
 - UNIX, Lisp, Smalltalkの成功
 - ただし、情報隠蔽ができない
- 人材の再利用
 - 経験を生かすことができるが...
- 設計の再利用
 - 設計のパターン化
 - ただし、ソフトウェアの進化の過程で、設計とコードの一致を維持するのが難しい

再利用可能なモジュール構造の要件

- 型の変化に対応できる
- データ構造とアルゴリズムの変化に対応できる
- 関連した操作がまとまって定義されている
- 顧客モジュールが実現方法を知ることなく操作を要求することができる
 - 今までの技術ではうまく解決できない
 - 後述する動的束縛で解決できる
- 実現方法の間の共通部分がうまくまとめられている
 - サブグループ間の共通性をうまく記述できるか
 - 表検索の場合では、順次形式の表がひとつのサブグループになる
 - 順次配列、連結リスト、順次ファイルの共通点をうまくまとめられるか？

オブジェクト指向は、実際に使われているのですか？

- オブジェクト指向
システム分析 / 設計に適した分野
- カールスバーグ・
ビール
(デンマーク)
- Siemens-Nixdorf Informationssysteme
(オーストリア)
- 某ソフトウェア会社@日本
- MacApp
(アップル社の調査)

オブジェクト指向 システム分析 / 設計に適した分野

- マルチメディア
 - グラフィック
 - Adobe Photoshop、CADソフトウェア
 - 動画
 - Apps Micro TV
 - 地図情報システム
 - DODの空母搭載基地情報システム
- ビジネス
 - 銀行
 - 在庫管理
- リアルタイム制御システム
 - 航空管制システム、衛星制御システム、中国の鉄道システム（マカオ国連大学）
- CASE
 - Excelerator II、ObjectCastLight、Mei

カールスバーグ・ ビール (デンマーク)

- ビールの在庫管理システム
 - 内容
 - 人
 - スケジュール
 - 成果
 - ユーザーから見た成果
 - 開発者から見た成果

内容

- メインフレームとPC
 - 従来のメインフレームのシステムでは拡張不可能
- DB2
 - ローカルとホスト (DB2/MVS) 連動
 - 50リレーション、700属性

人

- プロジェクトマネージャー 1名
- システム・プログラマ 1名
 - OS/2, DB/2の広範囲の知識
 - OS/2上のCプログラミングの知識
- アプリケーションプログラマ 2名
 - メインフレームのプログラマとしての高いスキル
 - PC 上での開発の経験なし
- コンサルタント 1名 (200時間)

スケジュール

- 開発ツール評価
 - 1993年8月～10月
- PARTS Wprkbenchで開発
 - 1993年11月～1994年1月
- オブジェクト指向技術とツールの研修も含む
 - PARTS Workbench 4日間セミナー、Smalltalk/V 4日間セミナー、CASEツール3日間セミナー
- ビジネスモデル作成
 - 30リレーション修正、20リレーション新規作成
- データベース設計
- Smalltalk/Vクラスの生成
 - 50クラス、データアクセス = 1500メソッド、ビジネスロジック = 500メソッド
- PARTS Workbenchによる組み立て
 - 20GUI部品、SQL文 < 10

ユーザーから見た成果

- 50 %の経費を節約
- 1週間7日、1日24時間稼働
- 製品をFIFOの法則によって出荷できる
 - リアルタイムに在庫が更新されるため、いつでも在庫の状況が分かる
 - リアルタイムのトランザクションによって、より良い流通のプランを立てられるようになった
 - すべてのトランザクションは発生と同時に受付られるようになった
 - それに伴ない、作業の重複が無くなり、時間が節約された
 - より良い出荷のコントロールができるようになった
 - 製品が無くなれば、すぐに、追いかけることが可能
- 倉庫、本社ともに、大量の伝票処理をするようなことは無くなった
 - エンドユーザは、このアプリケーションが使い易く理解し易いと感じていて、ほとんど研修の必要もない

開発者から見た成果

- ミッションクリティカルシステム
 - あらかじめ用意されたコンポーネントをビジュアルに組み合わせるという概念は、大きな効果をもたらした
- 生産性
 - Smalltalk/V を使って、アプリケーションのビジネス・ロジックを組むことが非常に簡単であることが分かった
 - 4 日間のトレーニングを受けただけで、Smalltalk/V の生産性が良いことが分かった
- 再利用性
 - 最初のプロジェクトにもかかわらず、コードを簡単に再利用できることも分かった
 - CASE モデルやリレーショナル・データベースのような、オブジェクト指向技術でないものをラッピングする場合の使い方も、非常に使いやすかった
 - アプリケーションは、標準的なコンポーネントを組み合わせで作られた
- 保守性
 - アプリケーションのコードを変更せずに、CASE ツールのデータベースを変更できた

Siemens - Nixdorf Informationssysteme (オーストリア)

- 銀行業務アプリケーション
 - 内容
 - 人
 - スケジュール
 - 成果

内容

- SNI 社はオーストリアのウィーンに本社を持つシステムインテグレータ
 - コンピュータメーカー（ホスト、PC、UNIX サーバ、キャッシュマシン、通帳記入機など）でもある。SNI 社の市場占有率は、ドイツとオーストリアの銀行業界の約 45% を占めている。SNI 社の取引先の銀行から、バラバラな業務形態を統合するよう要望がだされた。
- 開発した金融業向けソリューションシステム（FINIS）は銀行支店業務に必要な機能をすべて盛り込んだシステム
 - 窓口や店内業務サービス、顧客と会計管理、コンサルティング、効率的な営業活動と業務運営活動、貸付と証券、管理情報、これら基本機能を個別の銀行向けに拡張できるようなアプリケーション構造を持つ
- アーキテクチャ
 - ほとんどの部分は Window 版 Smalltalk / V で開発
 - 銀行端末のための下位層 API の開発は、必要に応じて C や C++ を使った
 - 分析と設計のチームは Objectory を使用
 - データベースには SQL Server

人

- 25 人の社員

- プログラマ17 人

- プロジェクトのスタート当初から C または C++ に詳しかった。ある程度 Smalltalk の経験を積んだものもいた。Smalltalk/V の教育はオンザジョブで行った。

- (注) 当時はコンポーネントウェアが無かったため、GUI作成にかなりの労力がかかった

- オブジェクト指向に興味を持っていたプログラマはすぐに慣れた

- そうでないプログラマは慣れるまで3 ~ 6 カ月かかった

- 開発チームは4つのグループ

- 銀行業務のオブジェクトモデルとデータベースの開発グループ
 - 分析と設計のグループ
 - FINIS のアプリケーション・フレーム・ツール開発グループ
 - FINIS の基本構成要素の開発グループ

スケジュール

- Smalltalk / V を使って 3 年間で開発
 - 全体完成には 3 0 0 人月
- 中核アプリケーションを 2 0 カ月で開発
 - 基本構成要素 (データベース、LAN のインタフェース、銀行専用プリンタ・インタフェースのクラスなど)
 - FINIS アプリケーション・フレーム・ツール (GUI とアプリケーション生成のツール)
 - FINIS アプリケーション自身 (ユーザインタフェース部分を含む)
 - 800 のクラス
 - 20,000 のメソッド

成果

- ユーザーから見た成果
 - 銀行が自分のニーズに合わせてアプリケーションを拡張できる
- 開発者から見た成果
 - すっきりした設計
 - 再利用性を実現
 - 柔軟で拡張性のある簡単なプログラミング環境
 - プロトタイプによるユーザー要求の早期確認
 - Cで作った類似システムは、
 - 開発が長期化
 - 保守が大問題

某ソフトウェア会社@日本

- CADソフトウェア
 - 内容
 - PC用CADソフトウェア
 - 人
 - スケジュール
 - 成果

人

- プロジェクトリーダー 1 人
- プログラマー 5 人
 - C の経験者 3 人
 - COBOL の経験者 2 人
- 電子メールによる技術サポート
 - PARTS Workbench および Smalltalk / V 経験者 2 人
 - PARTS Workbench および Smalltalk / V の経験は半年
 - 1 人はソフトウェア工学・オブジェクト指向分析 / 設計の専門家
 - C、Objective-C、Eiffel、Perl、COBOL の経験
 - もう一人はコンピュータ科学専攻のマスター卒業の新人
 - 仕様記述言語 Z の経験 2 年、C の経験 4 年
 - 2 人あわせて 0.1 人月くらいの負荷

スケジュール

- 1994年8月 開発ツール検討
- 1994年9月 PARTS Workbenchで開発開始
 - PARTS Workbenchセミナー 2日間
 - 1994年10月 Smalltalk/Vセミナー 2日間
- 1994年12月 入力画面350面作成

成果

- ユーザーから見た成果
 - 開発が早く、安い
 - 保守性
 - 再利用性
- 開発者から見た成果
 - 急速にオブジェクト指向技術をマスター
 - 当初、Smalltalk/Vプログラマー 1 人養成のはずが、全員がほぼマスター

MacApp (アップル社の調査)

- 開発期間が 1 / 3 から 1 / 4 に短縮
- 完全で一貫性のあるユーザーインタフェース
- ソースコード量が減った
- 洗練されたエラー処理

オブジェクト指向の基本概念

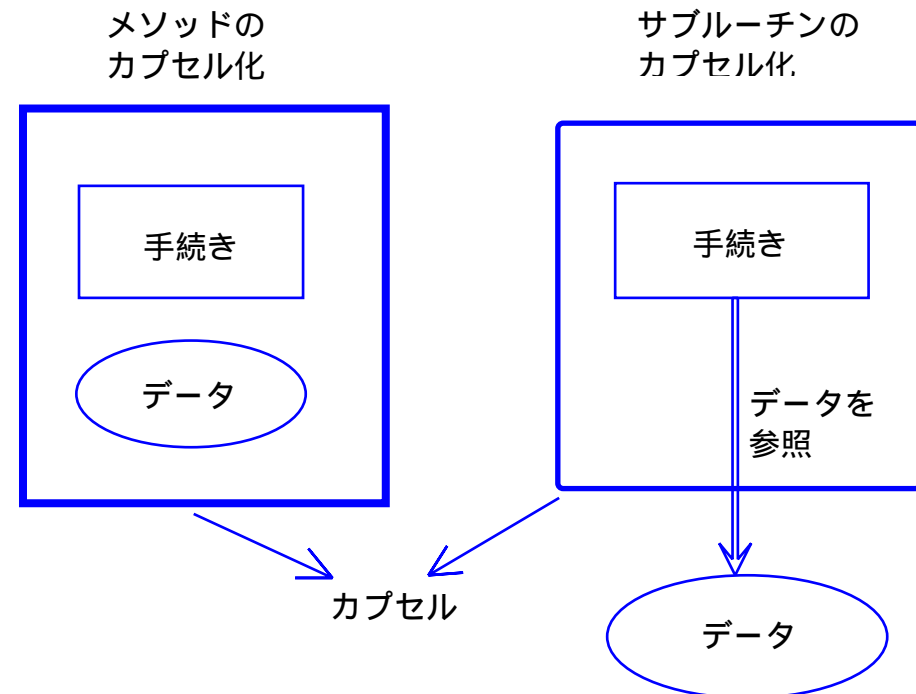
- クラスとインスタンス
- カプセル化
- メッセージ
- 多相と動的束縛
- 継承
- 自動メモリ管理

クラスとインスタンス

- なぜオブジェクトをクラスとインスタンスに分けるのですか？
 - 単なる都合
 - クラスにインスタンス共通の性質を記述するため
 - インスタンスベースのオブジェクト指向言語もある
- メタクラスとは何ですか？
 - クラスの性質を記述するためのクラス
 - メタクラスから見ればクラスはインスタンス

カプセル化

- カプセル化しない方が効率がよいのではないですか？
 - システム全体の効率に関する操作は全体の2～3%
 - 分析 / 設計段階からミクロの効率化を考えると、マクロ的に見れば非効率なシステムになる
- カプセル化はサブルーチンで十分ではありませんか？
 - サブルーチンは手続きのカプセル化だけ
 - データのカプセル化が十分でない
 - データを不必要に外部に晒してしまう



メッセージ

- オブジェクトにメッセージを送るのは、サブルーチンにパラメータを送るようなものですか？
 - 全く異なる概念
 - サブルーチン
 - 手続きにデータを送る
 - 相手がどんなデータ構造を扱えるか知っている必要がある
 - メソッド
 - データ（オブジェクト）に手続きを送る
 - 相手がどんな構造のデータか知る必要がない
 - 相手がどんなサービスを提供しているかは知らなければならない

多相と動的束縛

- 多くのメッセージに同じ名前を付けて混乱しませんか？
 - 異なる機能に同じ名前を付けると混乱する
 - 多相を使う方が概念が単純化する
 - 分析 / 設計 / 実現いずれ工程でも
- 動的束縛と静的束縛の違いは何ですか？
 - コンパイル時にメソッドが確定するのが「静的」
 - 実行時にメソッドが確定するのが「動的」
- 動的束縛の利点は何ですか？

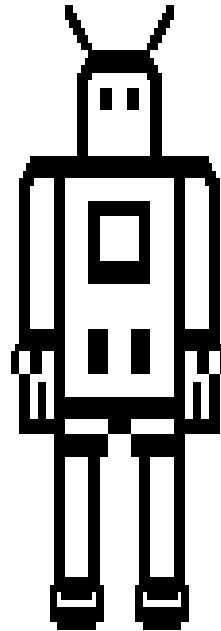
動的束縛の利点は何ですか？

```
if data = Text then  
  displayText(data)  
else if data = Picture then  
  displayPicture(data)  
end if
```



```
if data = Text then  
  displayText(data)  
else if data = Picture then  
  displayPicture(data)  
else if data = Image then  
  displayImage(data)  
end if
```

data display



Home



継承

- 継承を使った差分プログラミングと、サブルーチンを使った共通化の効果は同じくらいではありませんか？
 - 「再利用可能なモジュール構造の要件」が異なってくる
 - 型の変化に対応できる
 - データ構造とアルゴリズムの変化に対応できる
 - 関連した操作がまとまって定義されている
 - 顧客モジュールが実現方法を知ることなく操作を要求することができる
 - 実現方法の間の共通部分がうまくまとめられている

自動メモリ管理

- 自動メモリー管理をすると効率が悪くなりませんか？
 - 世代別ごみ集め（ generation scavenging ）法では、スーパープログラマー以外がメモリー管理するより速い
 - プログラマーがプログラミング時にメモリーの取得・解放をするのは、非常に難しい
 - 結果として、実行時エラーの嵐
 - 例
 - Windows, Windows NT
 - Mac OS
 - C, C++で構築されたほとんどすべてのシステム

オブジェクト指向の位置付け

- オブジェクト指向言語とOODBMSの違いは何ですか？
- 構造化分析 / 設計と
どちらが良いのですか？
- データ中心分析と
どちらが良いですか？
- IEと
どちらが良いですか？
- オブジェクト指向の位置づけ

オブジェクト指向言語とOODBMSの違いは何ですか？

- データの永続性
- オブジェクト指向言語
 - データの永続性の実現を目指す
 - OODBMSとの融合
- OODBMS
 - 照会言語の機能強化と標準化

構造化分析 / 設計と どちらが良いのですか？

- S A / S D O O A / O O D
 - O O A / O O D = リアルタイム S A / S D + 継承
+ 方法論改良
 - 方法論改良 = ER図 オブジェクト図
 - リアルタイム S A / S D = S A / S D + 動的モデル
 - 動的モデル = 状態遷移図
 - S A / S D = 情報 (データ) モデル + 機能モデル

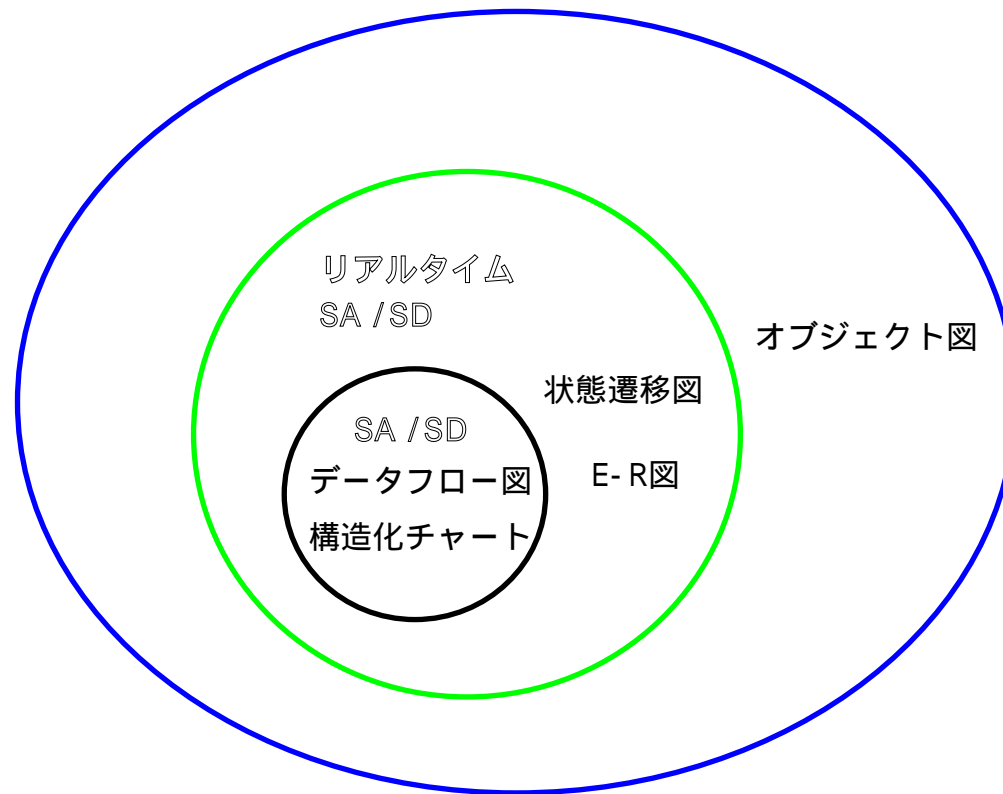
データ中心分析と どちらが良いですか？

- DOA OOA / OOD
 - OOA / OOD = DOA リアルタイム SA / S
D + 方法論改良
 - 方法論改良 = ER図 オブジェクト図

IEと どちらが良いですか？

- IE OOA / OOD
 - OOA / OOD = IE リアルタイムSA / SD
+ 方法論改良
 - 方法論改良 = ER図 オブジェクト図

オブジェクト指向の位置づけ



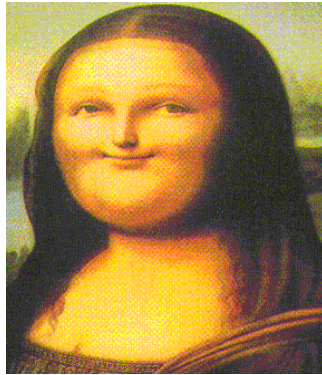
オブジェクト指向 方法論

- 幻想と現実
- 方法論の選択
- 上流CASEの選択

幻想と現実

- 「上流から下流まで一貫した自動CASE」ができれば、ソフトウェア開発は不要になるのでしょうか？
 - CASEの幻想
- コンポーネントウェアなどを使えば、分析 / 設計はあまり必要ないのではないですか？

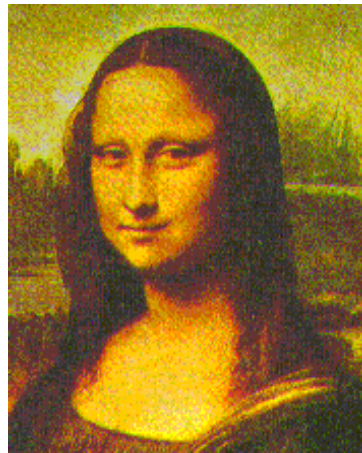
要求モデル



設計仕様



CASEの幻想



ユーザー要求

- 「ソフトウェア工学で分かったところを」というのが当面の目標



プログラム

コンポーネントウェアなどを使えば、 分析 / 設計はあまり必要ないのではない ですか？

- 何を作るかが分からなければ、ユーザー要求にあった物はできない
 - 何を作るのかを明確にするのが「分析」
- 正しい設計をしないと、保守性と再利用性が下がる
 - どうやって作るかを明確にするのが「設計」

方法論の選択

- どの方法論を選択すれば良いか？
 - モデルの比較
 - 静的モデルの比較
 - 動的モデルの比較
 - 適用工程とCASE対応
 - 適用分野
 - 各手法の使用状況

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				

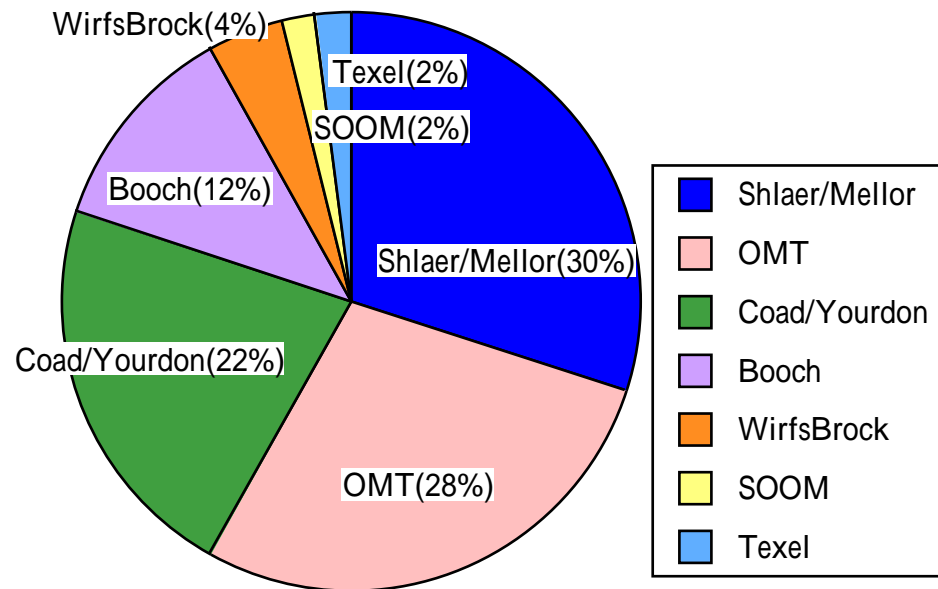
適用工程とCASE対応

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
	Rumbaugh	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



上流CASEの選択

- どのCASEを使えばよいか？
- CASEを選定する基準は？

どのCASEを使えばよいか？

どのCASEを使えばよいか？

ツール名	記法	開発販売元	OS	日本語 対応
Object Cast	OOA/OOD	FXIS	Mac, UNIX, Windows	
Objectory	OOSE	Objectory Corp.	Windows, UNIX	×
ObjectTeam, Paradigm Plus/Cadre	OMT, OOSA	Cadre Technologies	UNIX	
ObjectTool	OOA/OOD	Object International	Mac, Windows, OS/2, UNIX	
Paradigm Plus	OMT, OOSA, OOA/OOD,	Protosoft	Windows, Windows NT,	×
Rational Rose	Booch, OMT	Rational, オージス総研	Windows, OS/2, UNIX	
StP/OMT	OMT, OOSD	IDE, ニチメンデータシステム	UNIX	
Turbo CASE	Booch	Struct Soft	Mac	×

CASEを選定する基準は？

• オブジェクト指向CASEの持つべき機能

- 図エディター
- ユーザーインタフェース
- 仕様記述言語
- エラーチェック
- ハイパーテキスト
- 版管理
- 構成管理
- 手法のガイド
- ブラウザー
- データ辞書
- 共有

仕様記述言語 (RAISE, Z, OBJ, VDM)

集合

論理

命題論理

述語論理

代数

value

```
floor : Real  Nat  /* floor(1.3)  1 */
sq_root : Real  Real /* sq_root(25.0)
5.0 */
fl_sq : (Real  Real)  Real  Nat
/* fl_sq(29.0)  5 */
```

axiom

[floor]

```
r : Real,  i : Nat • floor(r) as i
```

```
post i  r < i+1
```

[sq_root]

```
n : Nat,  s : Real • sq_root(n) as s
```

```
post s 2 = n  s  0.0
```

[fl_sq]

```
fl_sq  floor ° sq_root
```

オブジェクト指向 分析

- 手順
- 分析は概要だけにして、
設計に進みたいのだが？
- 分析は必ずデータ（オブジェクト）中心でやらなければならないのか？
- ドキュメント化

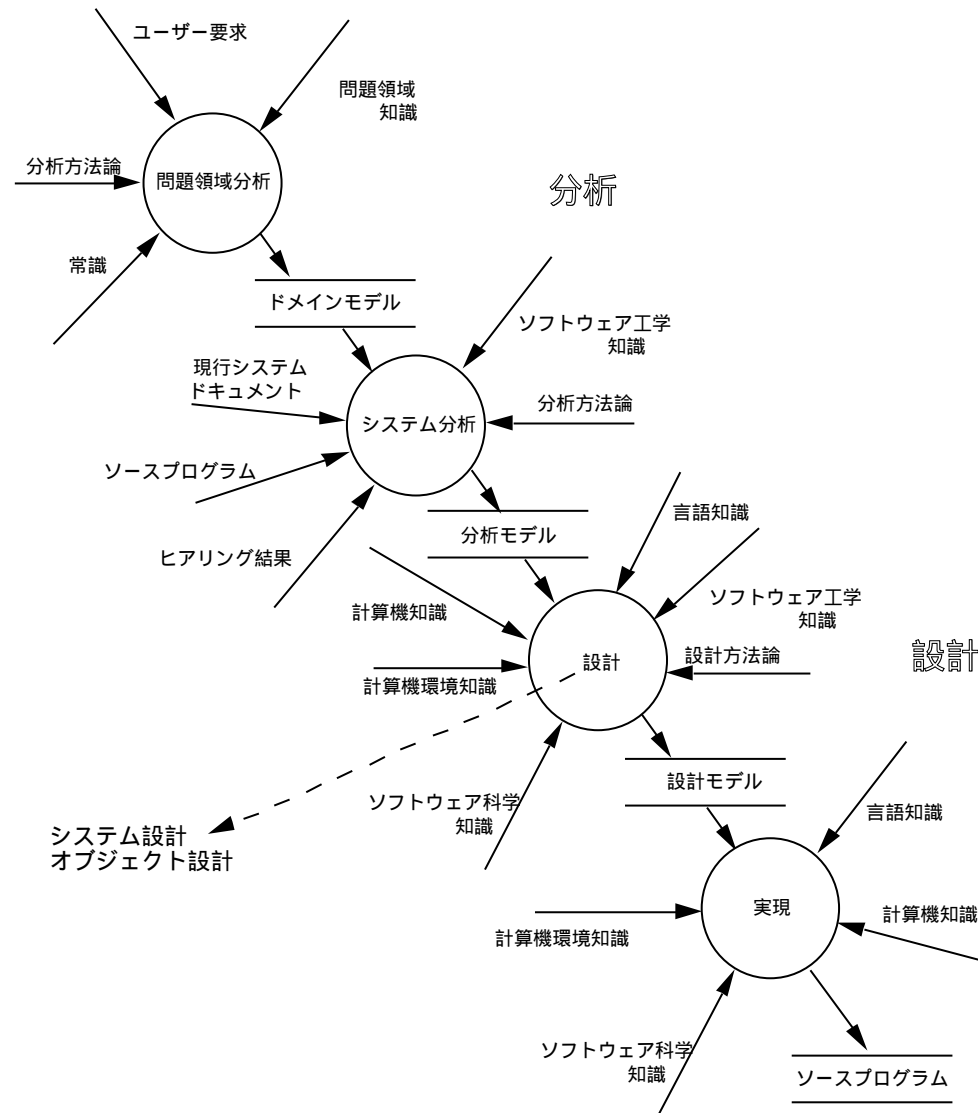
手順

- 手順は？
- 分析や設計局面の
終了条件が見えない
- 誰が（あるいは、どれが）
正しいオブジェクトと検証するのか？
- 手順は絶対か

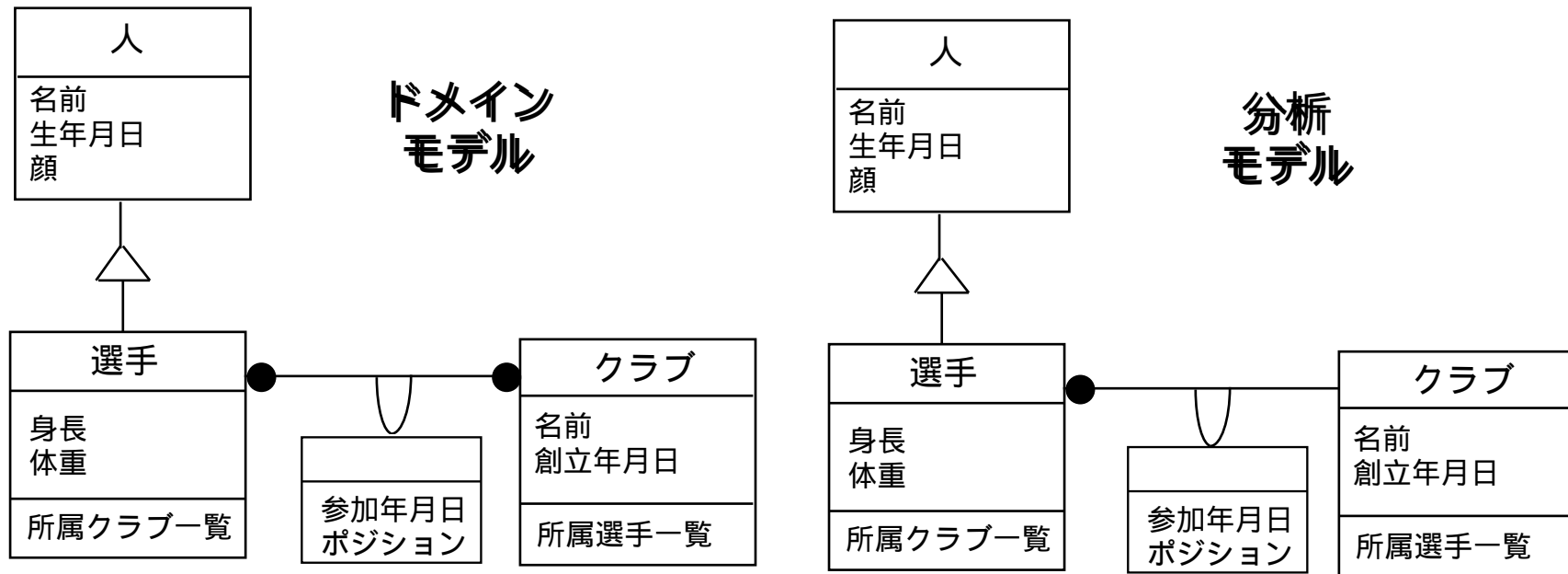
手順は？

- 分析 / 設計 / 実現の手順（概観）
- 分析の手順（イメージ）
- 設計 / 実現の手順（イメージ）
- 実現（部品配置）
- 実現（部品接続）
- 実現（実行）
- 分析手順詳細

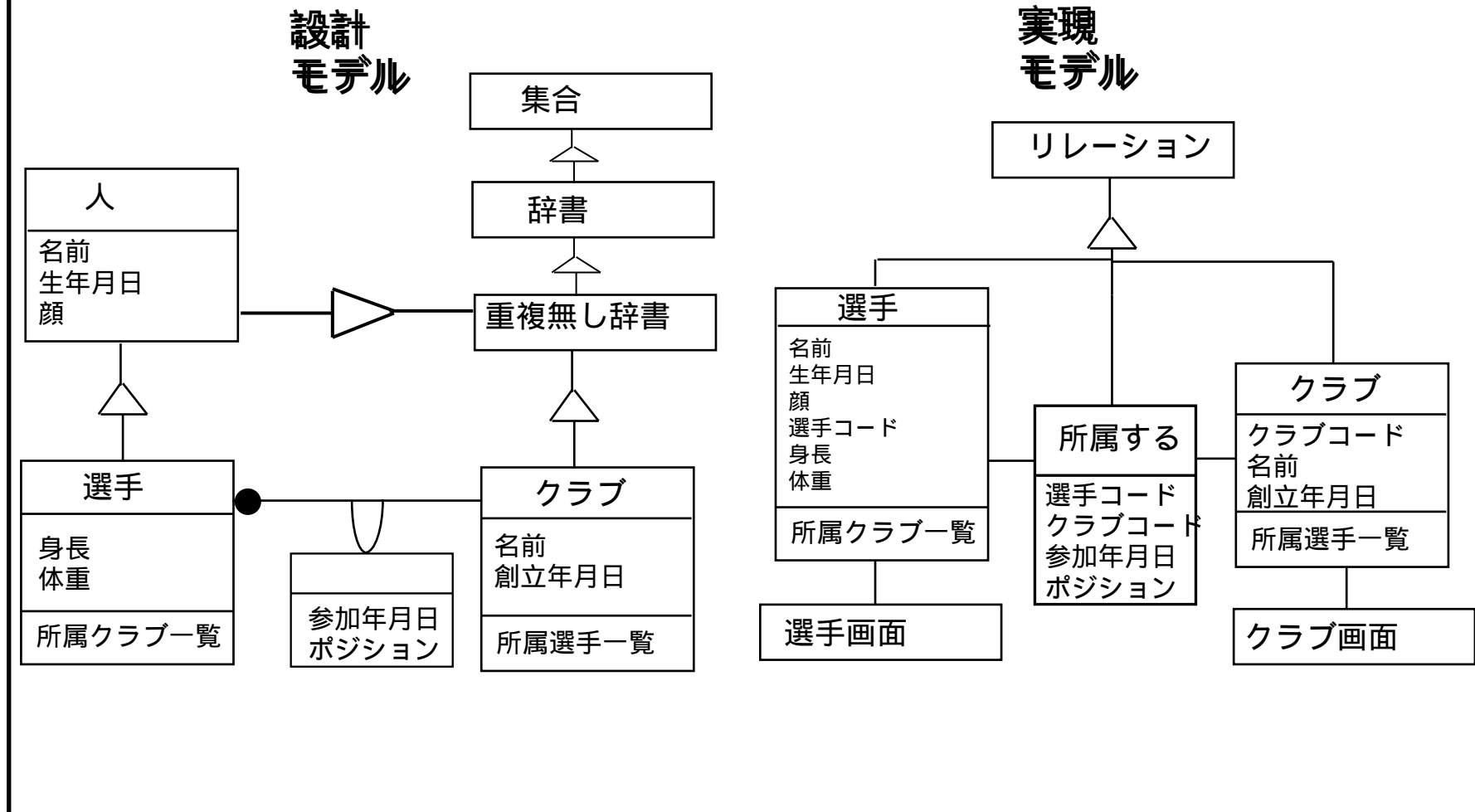
分析 / 設計 / 実現の手順 (概観)



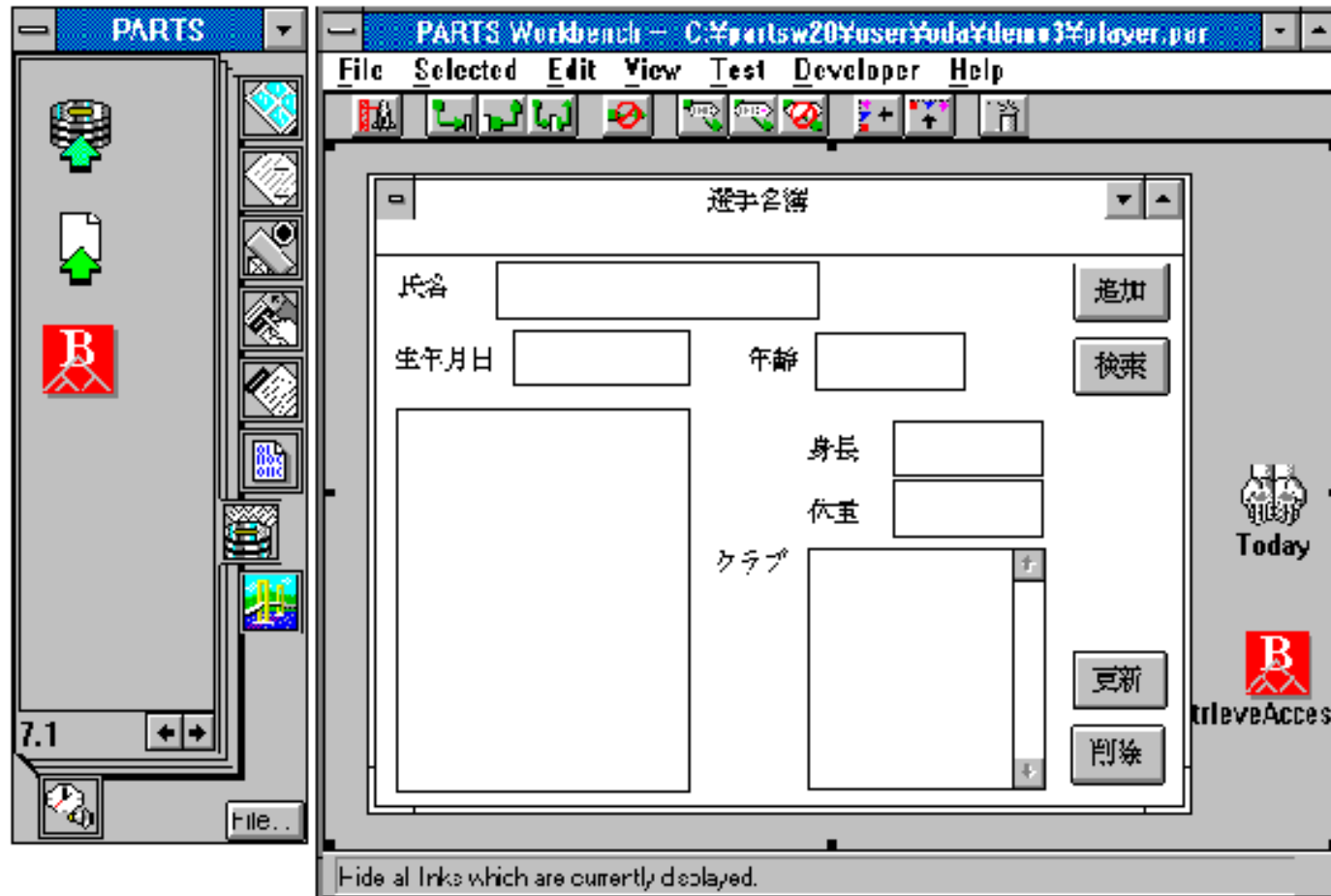
分析の手順（イメージ）



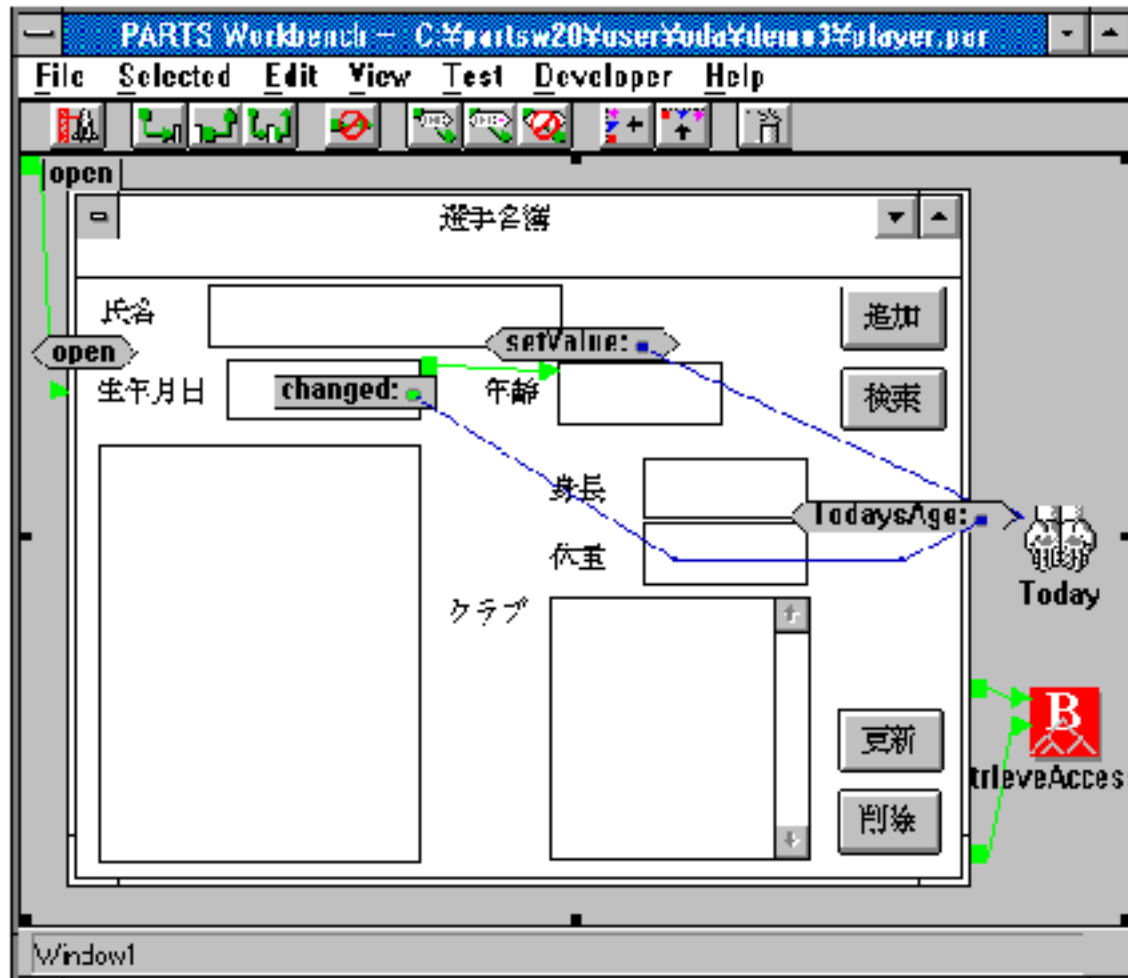
設計 / 実現の手順 (イメージ)



実現（部品配置）




実現（部品接続）



実現（実行）

選手名簿

氏名	佐真伸	追加		
生年月日	1951/09/29	年齢	42	検索
	身長	177.0		
	体重	99.0		
クラブ	東京教育大学 読売クラブ FCサンデー 小金井3K NFC	更新		
		削除		

分析手順詳細

- 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

問題記述文の最初の版の作成

- 問題記述文を最初に作るのでは、従来のドキュメントと変わらないのではないか？
 - 最終的にはユーザーズマニュアルに持っていくための最初の版
 - すでにユーザーズマニュアルがある場合はそれを使用する

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、鉄道情報協会的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
- データ辞書を準備する
- クラス間の関連を洗い出す
- オブジェクトとリンクの属性を洗い出す
- 継承を使ってクラスを構成し単純化する
- 何をもって最適とみなすのか？
- 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
- オブジェクトのモデル化の繰り返し
- 関係のあるクラスをグループ化してモジュールにする

オブジェクトのクラスを洗い出す

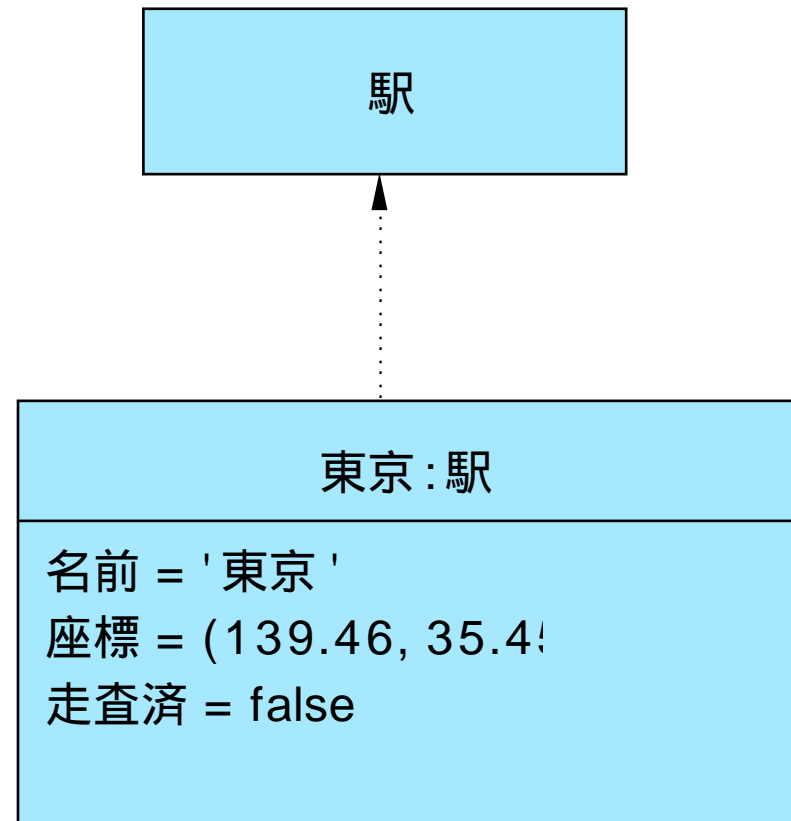
- クラスはどう見つければよいのか？
- インスタンス図を作る
- クラスの吟味方法は？
- 複合オブジェクトが必要かどうか検討する
- インスタンスの多重度を検討する
- 何をもって見つけたクラスが最適とみなすのか？

クラスはどう見つければよいのか？

- 既存のモデルから盗んでくる
 - OO・SA/SD・DB・数学・形式的仕様記述の教科書、過去のプロジェクト、対象問題領域の教科書やマニュアル
- 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
- 対象領域や一般常識から導かれるクラスもある
 - 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
- 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

インスタンス図を作る

- 何をクラスとし、何をインスタンスとするか整理する
- インスタンス図は必要なときだけ作ればよい



クラスの吟味方法は？

- 冗長なクラス

- より記述的な名前の方を使う
 - ユーザーと顧客では、顧客の方がより記述的

- 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

- 属性

- 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

- 操作

- ロール

- クラスの名前は、関係の中で果たす役割（ロール）の名前ではない

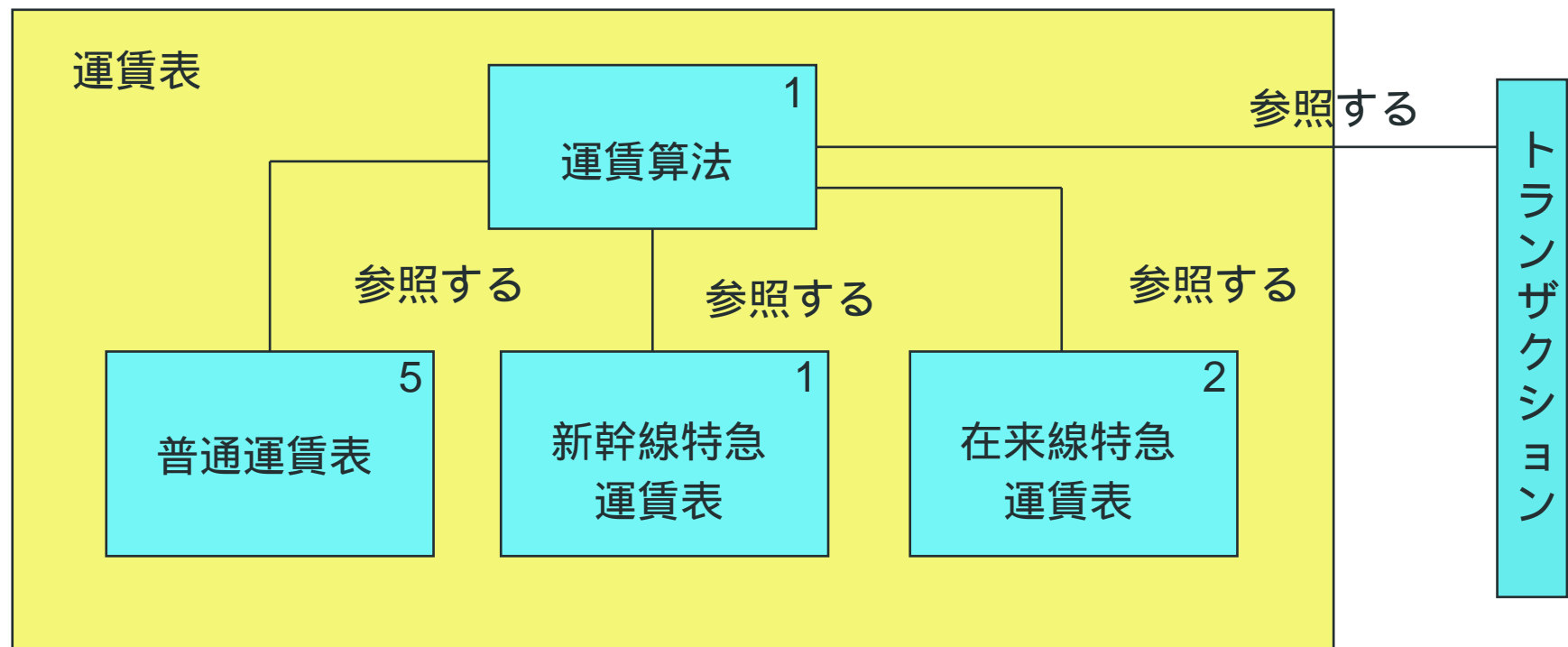
- 実現上の構造

- 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

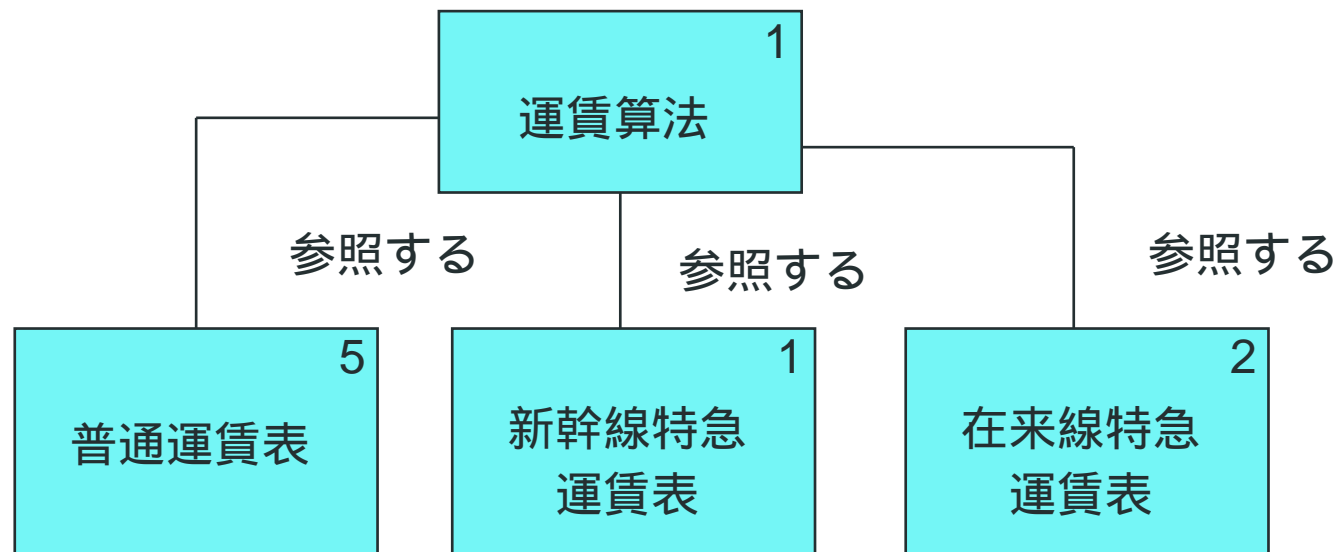
複合オブジェクトが必要かどうか検討する

- クラスの入れ子



インスタンスの多重度を検討する

- singletonクラス



何をもって見つけたクラスが最適とみなすのか？

- 最適なモデルは普通見つからない
- モデルの良くない点を、以下のソフトウェア工学の原則に従って修正していく
 - 外的品質要因
 - モジュール性の原則
 - 再利用可能なモジュール構造の要件

データ辞書を準備する

- **どんなデータをデータ辞書に入れるのですか？**
 - クラス・属性・関連・操作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
- **リポジトリと要求辞書というのがありますが、データ辞書はどこが違うのですか？**
 - 意味は同じ
 - データ辞書はSA/SDのころに発明された言葉
 - 要求辞書はリアルタイムSA/SDで発明された言葉
 - リポジトリはIEで発明された言葉
- **なぜデータ辞書を作らなければならないのですか？**
 - 要求に関わる言葉の意味の定義のため
 - エラーチェックのため
 - 検索のため

クラス間の関連を洗い出す

- 関連はどう見つければよいのか？
- 関連の吟味方法は？
- 何をもって最適とみなすのか？

関連はどう見つければよいのか？

- 動詞から探す
 - 物理的场所・直接動作・対話・所有関係・ある条件を満たす関連がある
- 対象領域や一般常識から導かれる関連もある
 - 顧客は切符を買う
 - 列車は駅から駅へ移動する

関連の吟味方法は？

- 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
 - 実現上の関連
 - 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）
 - 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
- 関連の意味を吟味する
 - 誤った名前と呼ばれる関連
 - 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はなく、それが何であるかと言うような名前を選ぶ
 - 多重度
 - 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
 - 見逃された関連

何をもって最適とみなすのか？

- クラスの場合と同じソフトウェア工学の原則
 - 外的品質要因
 - モジュール性の原則
 - 再利用可能なモジュール構造の要件

オブジェクトとリンクの属性を洗い出す

- 属性はどう見つけるのか？
- 属性の吟味方法は？
- 何をもって最適とみなすのか？
 - クラスの場合と同じソフトウェア工学の原則

属性はどう見つけるのか？

- 既存のモデルから盗んでくる
- 形容詞や名詞に対応する
 - 「ラベルの色」「ウィンドウの位置」など
 - 「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

属性の吟味方法は？

- 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
- 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
- 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

- 継承構造はどう見つけるか
- 継承構造の吟味方法は？

継承構造はどう見つけるか

- 既存のモデルから盗んでくる
- 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
- 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

継承構造の吟味方法は？

- 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
- 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つを上を持って行く
- 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

何をもって最適とみなすのか？

- クラスの場合と同じソフトウェア工学の原則
- オブジェクト指向らしさ
 - 部品の割合
 - 抽象クラスの割合
 - 多相メッセージの割合
- 青木の指標
- ChidamberとKemererの指標

青木の指標

- 階層内のクラス的位置 (HF)
- 全クラス中の、クラスの半順序位置 (RF)
- ポリモルフィズム量 (PF)

階層内のクラス的位置 (HF)

- クラスの継承階層でどのあたりに位置するかを示す指標
- 0 の場合クラス階層の一番上、1 の場合クラス階層の一番下になる
- 0 に近いほど、クラスの抽象度が高く、再利用性が高いと言える
- $HF_A = \frac{\text{クラスAのスーパークラス数}}{\text{クラスAのスーパークラス数} + \text{クラスAのサブクラス数} + 1}$

全クラス中の、クラスの半順序位置 (RF)

- あるクラスの相互参照関係（半順序）でどのあたりに位置するかの指標
- 他のクラスを参照する割合が大きいクラスほど後ろに来る
- 0に近いほど、クラスの独立性が高く、部品として使われている可能性が高い
- $RF_A = \frac{\text{クラスAの相互参照トポロジカルソート順位}}{\text{全クラス数}}$

ポリモルフィズム量 (PF)

- あるクラスのメッセージが、どの程度他のクラスのメッセージと重なっているかを示す指標
- 1の場合全メッセージが重なっていることを示し、0の場合まったく重なっていないことを示す
- 1に近いほど、ポリモルフィズムが使われていて、理解しやすく保守性が良いことを示す
- 0に近いほどユニークな部品で、抽象的である可能性が高い
 - もっとも0に近くても、ポリモルフィズムをうまく使っていないだけで、ユニークな部品でもなく、抽象性もない場合もあり得る
- $PF_A = \frac{\text{クラスAで定義されているメッセージで他のクラスでも定義されているメッセージ数}}{\text{クラスAに定義されているメッセージ数}}$

ChidamberとKemererの指標

- クラスの複雑性
(WMC = Weighted Methods per Class)
- 継承木の深さ
(DIT = Depth of Inheritance Tree)
- 子クラスの数
(NOC = Number of Children)
- オブジェクト間の結合度
(CBO = Coupling Between Objects)
- メッセージの応答数
(RFC = Responce For a Class)
- ソッドの強度の欠落度
(LCOM = Lack of Cohesion in Methods)

クラスの複雑性

(WMC = Weighted Methods per Class)

- メソッド毎に重み付けをした複雑度の合計
- WMCの高いクラスほど、複雑で開発や保守のコストがかかる

$$WMC = \sum_{i=1}^n C_i$$

ここで、 C_i は*i*番目のメソッドの静的な複雑度である
簡易的に計算するときは、 $C_i = 1$ とする
この場合、WMCはクラス毎のメソッド数になる

継承木の深さ

(DIT = Depth of Inheritance Tree)

- あるクラスが継承木の何レベル目にあるかを示す
- クラス階層（継承木）内の深い位置にあるクラスほど、継承するメソッドの数が多くなる傾向にあり、より複雑になりやすい
- DIT スーパークラスの数

子クラスの数 (NOC = Number of Children)

- あるクラスの直接のサブクラスの数である
- 子クラスの多いクラスは、それだけ他のクラスへの影響力が強く、再利用性が高いといえる
- NOC 直接のサブクラス数

オブジェクト間の結合度 (CBO = Coupling Between Objects)

- 継承以外の方法で結合されているクラス数を計測
- この種の連結はオブジェクトのモジュール性を下げ、再利用を阻害する
- CBO 継承を除く、結合している他のクラスの数

メッセージの応答数 (RFC = Responce For a Class)

- 与えられたメッセージに回答するために、クラスが起動しなければならないメソッドの数
- このサイズが大きいほど複雑なオブジェクトだと言える
- RFC クラスが起動しなければならないメソッドの数

ソッドの強度の欠落度

(LCOM = Lack of Cohesion in Methods)

- クラス内のメソッド間のインスタンス変数の共通部分の集合の内、互いに素な集合の数
- この数が少ないほど、メソッドの強度が高い
- 強度が高い方が、保守性や理解しやすさの点で望ましい
- COM n 個の集合 $\{I_1\}, \dots, \{I_n\}$ の共通部分で形成する互いに素な集合の数
 - ここで、 $\{I_1\}, \dots, \{I_n\}$ はあるクラス中の i 番目のメソッド M_i のインスタンス変数の集合。

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

- シナリオとはどんなものですか？
- どうやってオブジェクトモデルをテストするのですか？
- シナリオはテストケースに流用できるのですか？
- シナリオを書くツールはあるのですか？

シナリオとはどんなものですか？

• シナリオとは

- 複数のシナリオを統合したものがアプリケーション
 - コンポーネントウェアの世界あるいはオブジェクト指向の世界では、複数のオブジェクトが通信しながら仕事を行っている
- あるひとまとまりの仕事を記述したものがシナリオ
 - アプリケーションあるいはシステムの仕事の単位は利用者の目からは「シナリオ」あるいは「協調作業単位(UseCase)」として見ることができる
- コンポーネントウェアアプリケーションの設計に際しては、いかに「協調作業単位」を切り出して部品間のリンクとして実現するかを考えなければならない
- テストケースにもなる

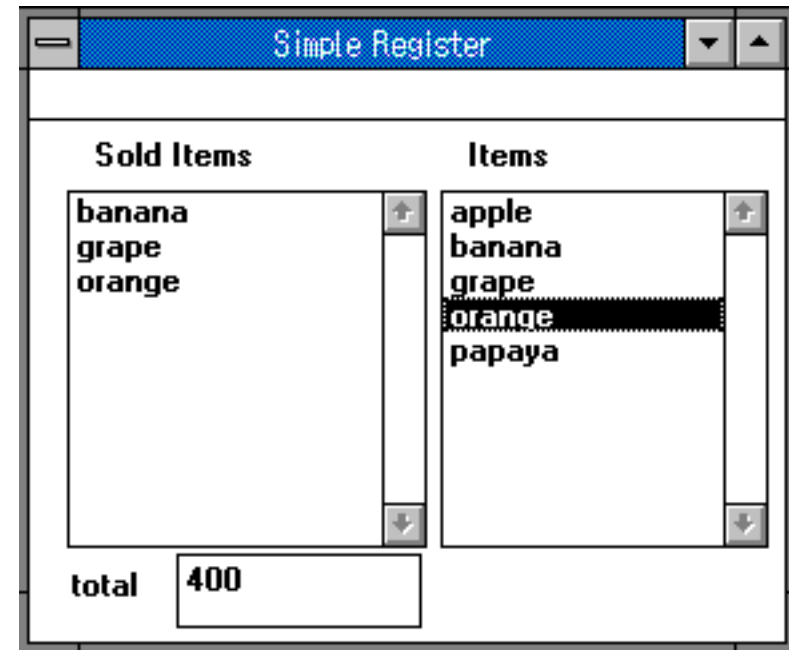
• シナリオの内容

- 事前条件
 - どのような条件がなりたっているときに、このシナリオは実行できるか
- 開始条件
 - 実際の仕事の開始のきっかけ
- 実行内容、関連部品
 - 具体的作業内容と参照、変更、生成する部品
- 事後条件
 - シナリオが定常状態に落ちついた時点で、成立していることが期待されている状態
 - 例：「預金すると、口座の残高が預金額だけ増えている」

• 例

例

- シナリオ 1
 - アプリケーション開始時に商品リストを初期化する
 - 事前条件
 - アプリケーション開始時のみ
 - 作業開始
 - アプリケーションの起動
 - 作業内容と関連部品
 - ウィンドウを開く
 - 価格データベースから商品名を引き出し商品リストとして表示
 - 事後条件
 - 商品リストが表示されている
 - シナリオ 2
 - 商品リストがダブルクリックされると、売り上げリストと売上合計を更新する
 - 事前条件
 - 商品リストが表示されている
 - 作業開始
 - 商品リスト上の項目がダブルクリックされた
 - 作業内容と関連部品
 - 選ばれた項目名を売り上げリストに追加
 - 選ばれた項目名に対応する価格を価格データベースから取り出して、合計金額欄に追加
 - 事後条件
 - 選ばれた項目が売り上げリストの最後に追加されている
 - 売り上げリストの金額合計が正しく計算されている



どうやってオブジェクトモデルをテストするのですか？

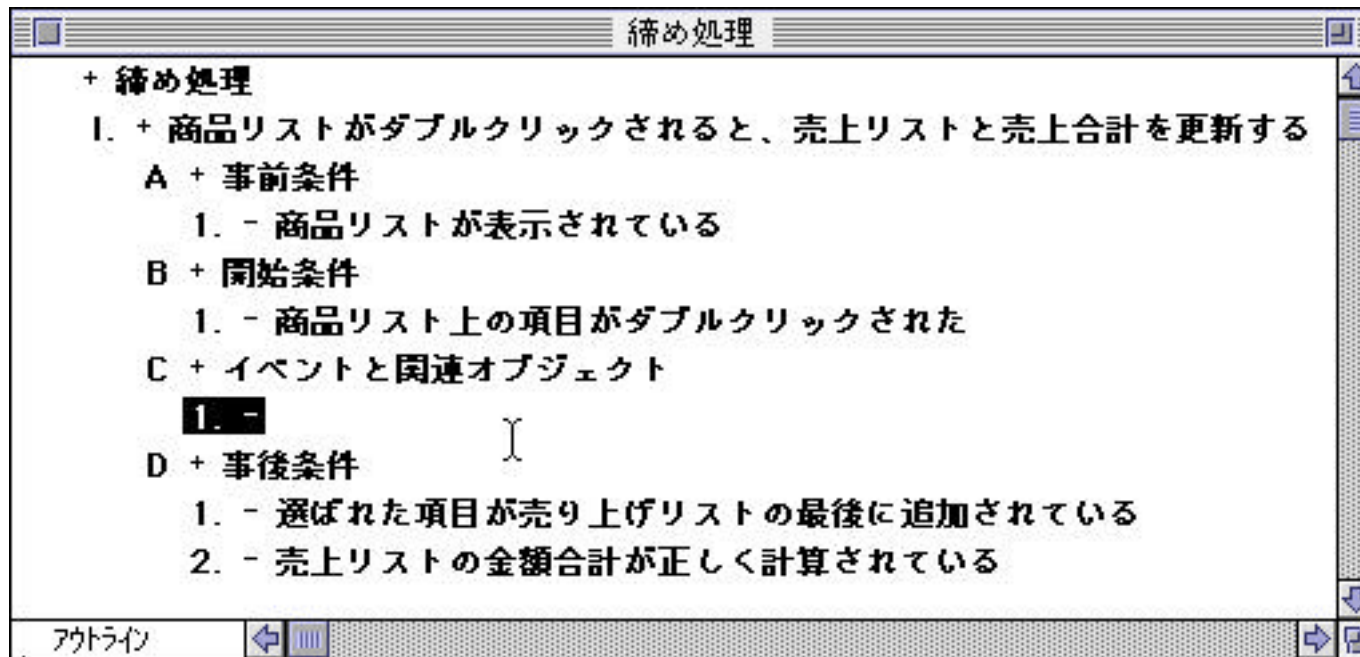
- 各アクセス経路をたどってみて、各々が意味のある値を引き出すことができるか確認する
 - ユニークな値が期待されているところで、ユニークな値が得られるか？
 - 「多」の多重度が設定されているところで、必要ならユニークな値を取り出す方法が存在するか？
 - 有効な質問を考えてみて、答えがでないようなことはないか？
 - 実世界では単純な事柄が、複雑なアクセス経路で表現されるなら、何かが抜け落ちている可能性がある
- ウォークスルー法を使ってテストする
 - 意味的エラーの60%以上を発見できる

シナリオはテストケースに 流用できるのですか？

- 「流用できる」というよりテストケースそのもの
 - この段階でテストに使用し、分析モデル全体ができたらまたテストし、設計モデルができたらさらにテストし、単体テストや総合テストにも使える

シナリオを書くツールはあるのですか？

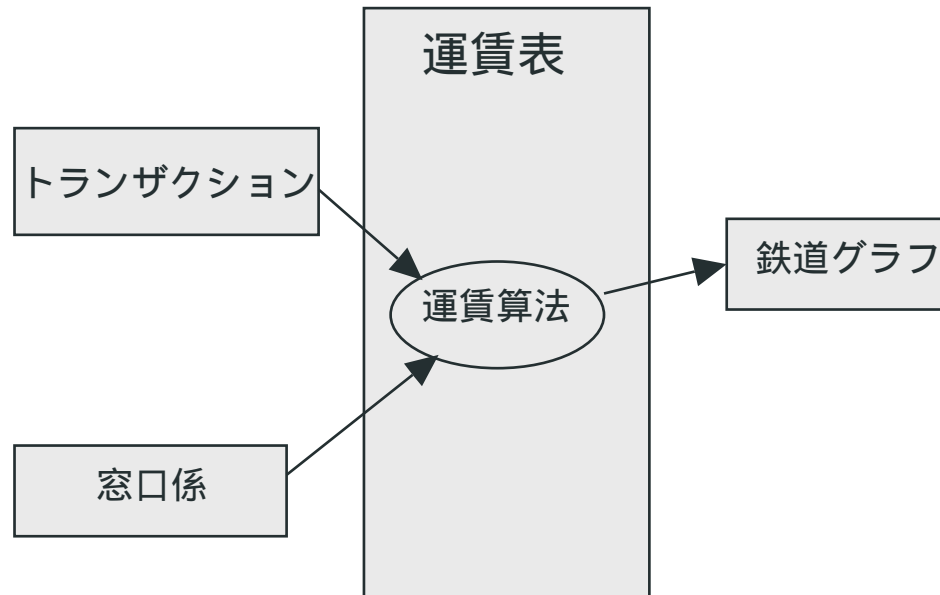
- アイデアプロセッサ
 - Inspiration, Acta, More, Emacs
- OO-CASEツール
 - Use Case
 - Objectory, StP/OMT



動的モデルの作成

- 手順
 - Use Caseを作成する
 - 代表的な相互作用のシナリオを作成する
 - オブジェクト間のイベントを認識し、各シナリオ毎のシナリオ図を作成する
 - 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
 - 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
 - ユーザーインタフェースの設計
- どういう時に状態遷移図を入れ子にするのですか？

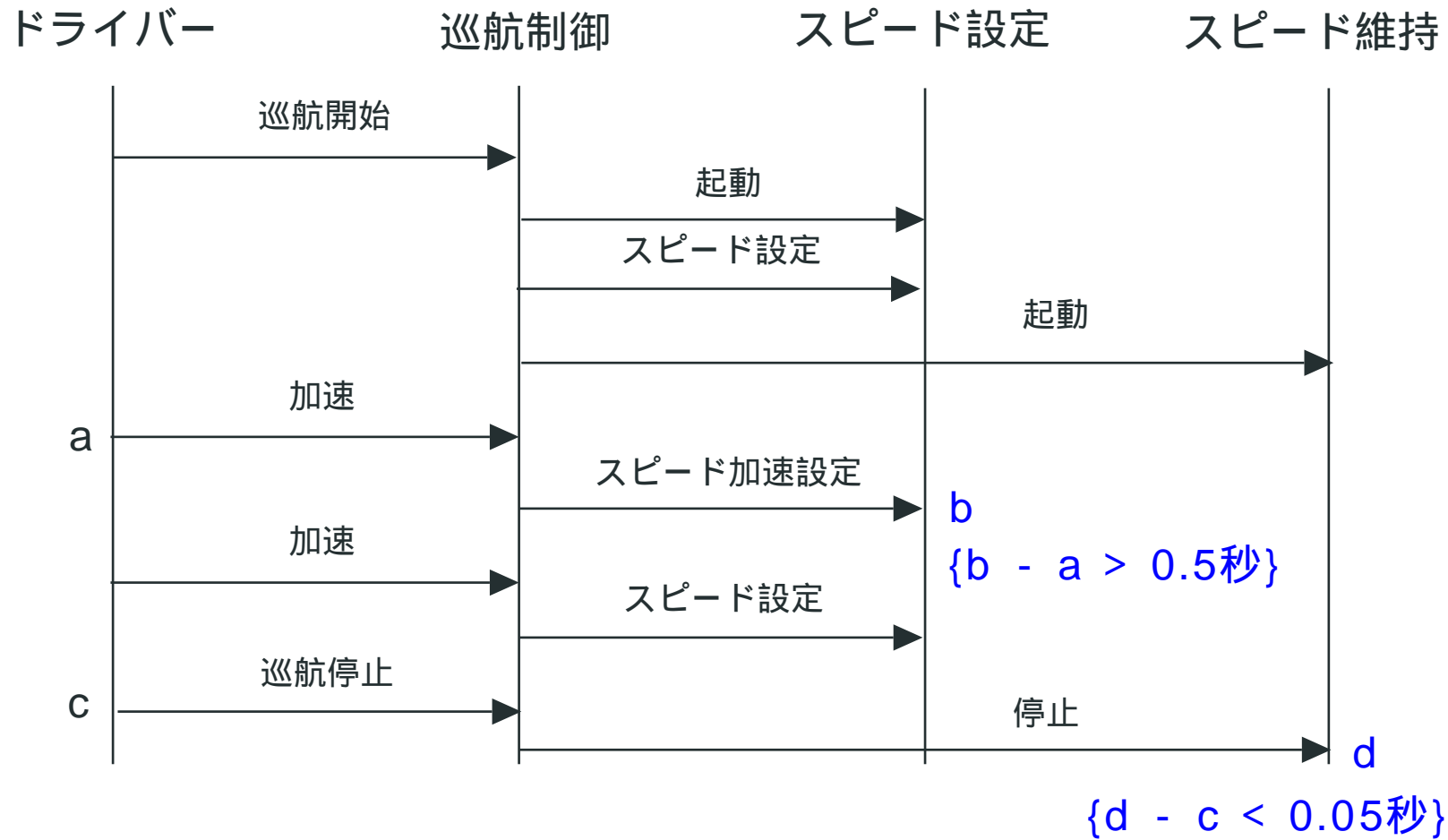
Use Caseを作成する



代表的な相互作用のシナリオを作成する

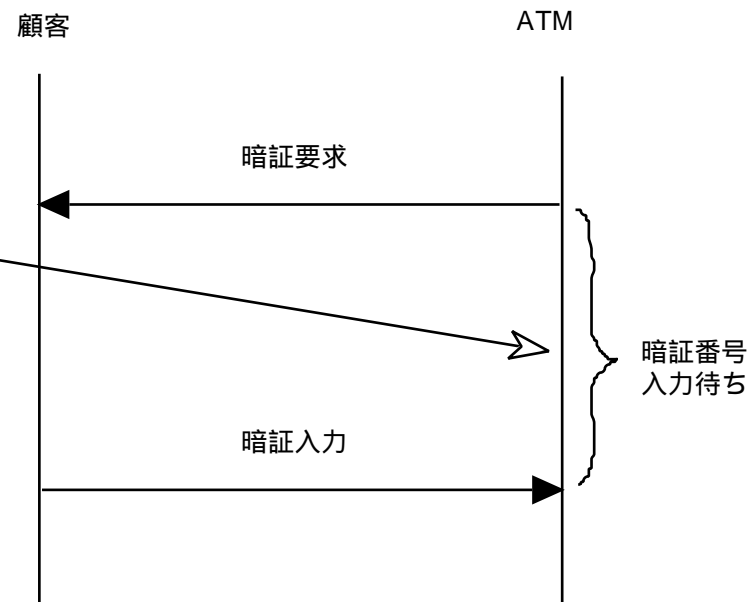
- シナリオはUse Caseのインスタンス
- シナリオ作成の順序
 - 正常な場合
 - 特殊な場合
 - エラーの場合
- 各イベントについて、アクターを認識する

オブジェクト間のイベントを認識し、 各シナリオ毎のシナリオ図を作成する



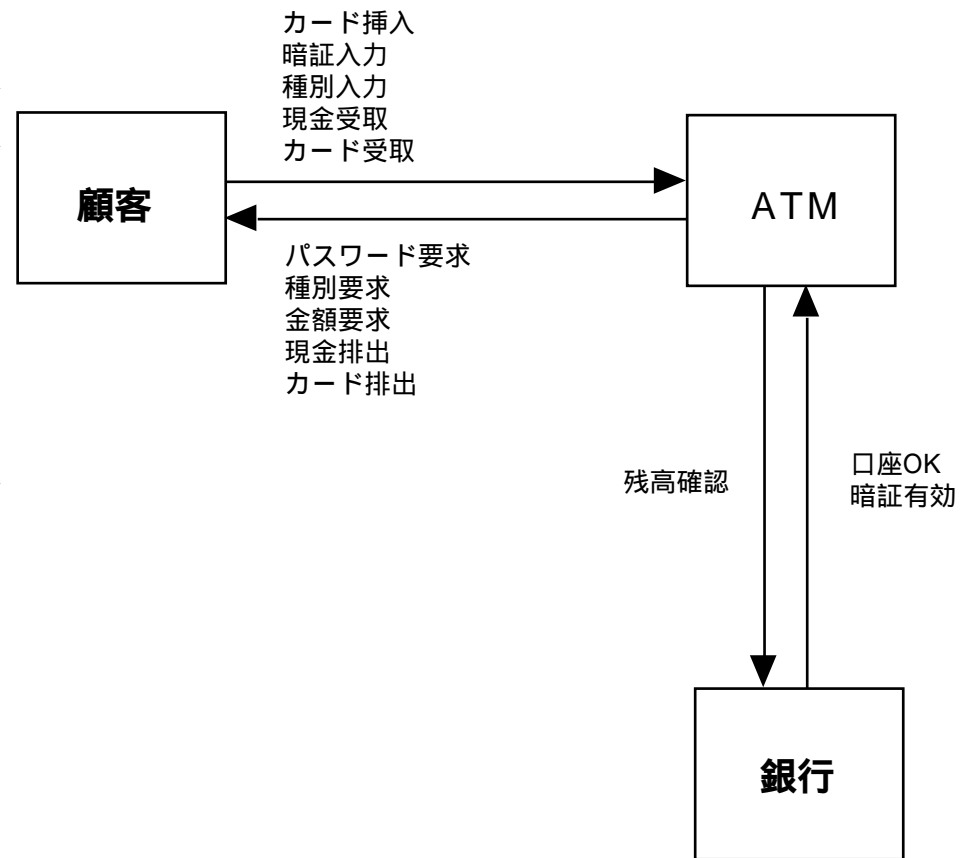
重要な動的振舞いをする各クラス毎に、状態遷移図の作成を行う

- 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



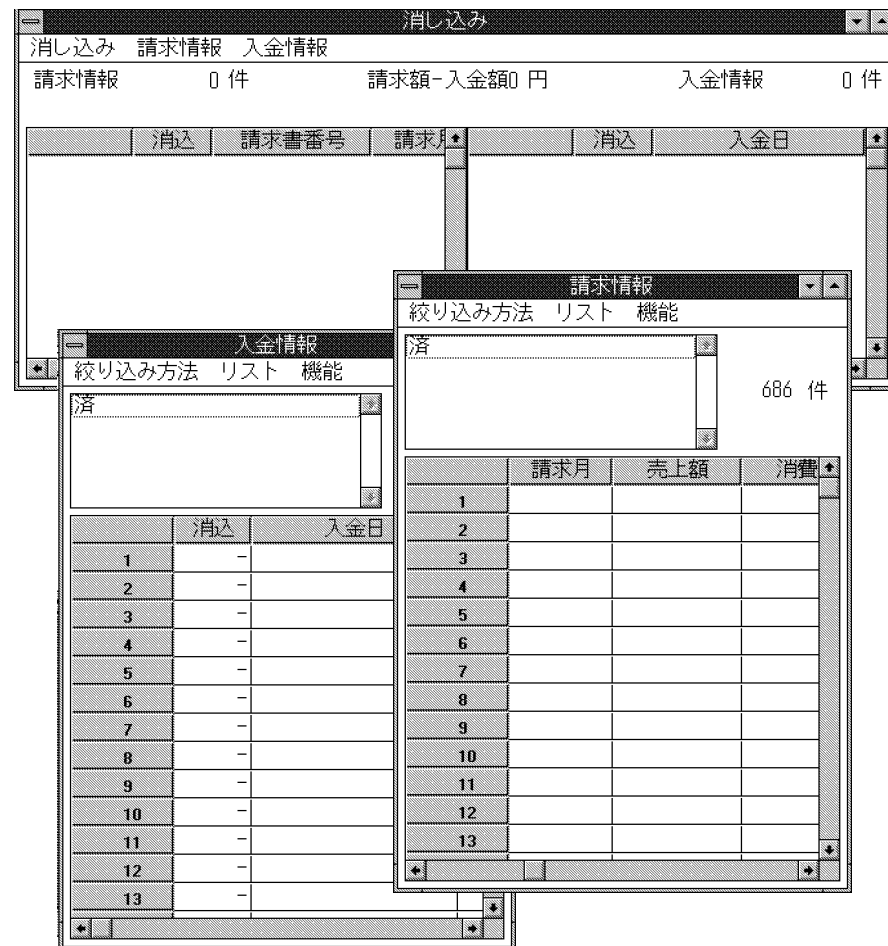
状態遷移図間で共有されるイベントの整合性と完全性をチェックする

- 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
- 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
- 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



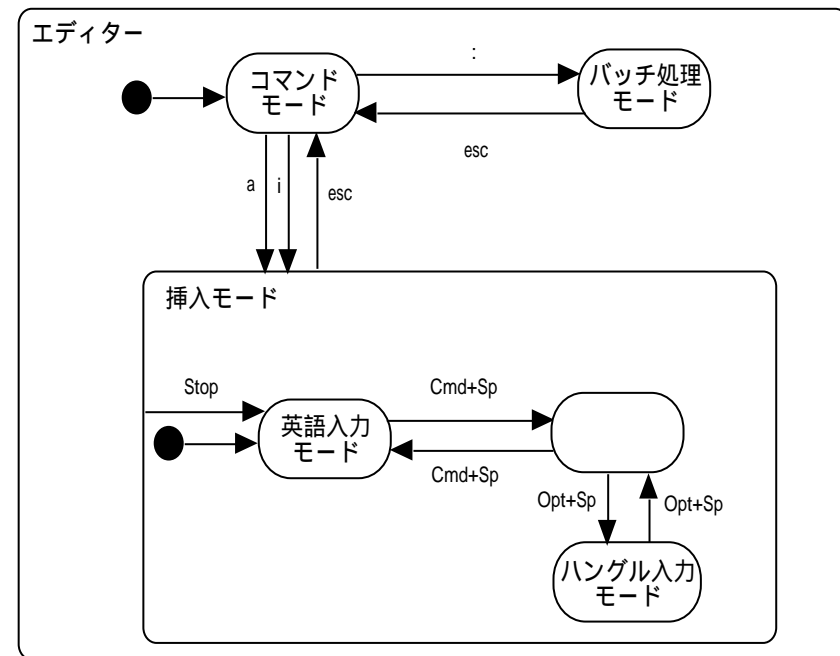
ユーザーインターフェースの設計

- アプリケーションのロジックとユーザーインター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



どういう時に状態遷移図を入れ子にするのですか？

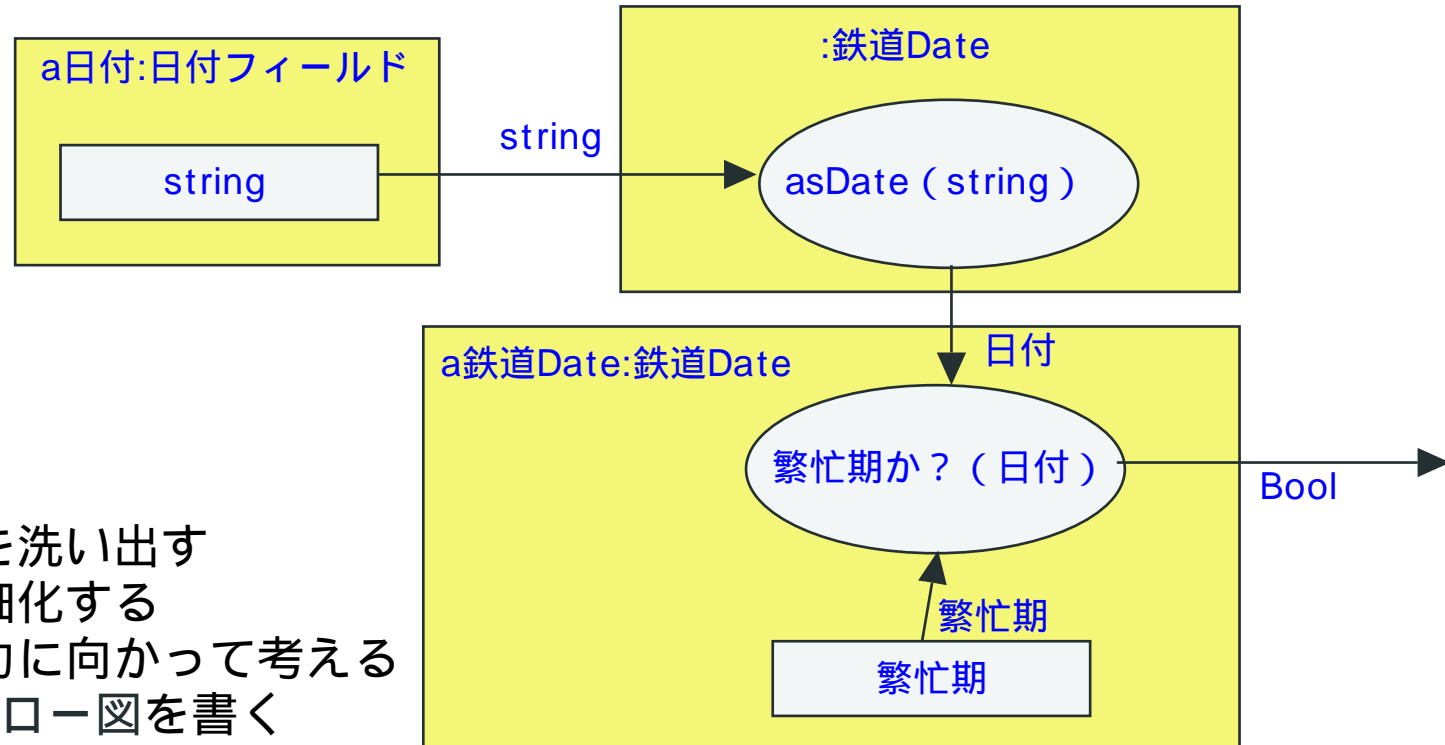
- いくつかの状態に同じ遷移が発生するなら、入れ子の状態を使う
- サブクラスの状態遷移図は、スーパークラスの状態遷移図からできるだけ独立にする
 - サブクラスの状態遷移図は、サブクラスのための属性に集中する



機能モデルの作成

- 手順
- どうやって入出力を洗い出すのですか？
 - 従来と同じ方法
 - 伝票や帳票や画面を調べる
 - 対象問題領域の教科書やマニュアルを読む
- 各機能の記述はどうするのですか？
 - 仕様記述言語を使って、
厳密に宣言的に記述する 1
 - 仕様記述言語を使って、
厳密に宣言的に記述する 2
 - 事前条件・事後条件を使う

手順



- 入出力の値を洗い出す
- 段階的に詳細化する
- 出力から入力に向かって考える
- OOデータフロー図を書く
- 各機能が何をするか記述する
- 制約条件を見つける
- 最適化の基準を作る

仕様記述言語を使って、 厳密に宣言的に記述する 1

- value
 - floor : Real Nat /* floor(1.3) 1 */
sq_root : Real Real /* sq_root(25.0) 5.0 */
fl_sq : (Real Real) Real Nat /* fl_sq(29.0) 5 */
- axiom
- [floor]
- r : Real, i : Nat • floor(r) as i
- post i r < i+1
- [sq_root]
- n : Nat, s : Real • sq_root(n) as s
- post s $s^2 = n$ s > 0.0
- [fl_sq]
- fl_sq floor ° sq_root

仕様記述言語を使って、 厳密に宣言的に記述する 2

- value

- ダイエットする: (現体重, 目標体重, 誤差, 月毎減量値, 減量期間) Bool
- 月毎減量値は適正である: (現体重, 月毎減量値, 1 カ月前の体重) Bool
- 総減量値は適正である: (現体重, 目標体重, 月毎減量値, 減量期間) Bool

- axiom

- [ダイエットする]

- $d : (\text{現体重}, \text{目標体重}, \text{誤差}, \text{月毎減量値}, \text{減量期間}), \quad b : \text{Bool} \cdot \text{ダイエットする}(d) \text{ as } b$
- $\text{post } b = (1 - \text{誤差}) * \text{目標体重} < \text{現体重} \quad \text{現体重} < (1 + \text{誤差}) * \text{目標体重}$
- $\text{pre } (0 < \text{目標体重} \quad \text{目標体重} < \text{現体重}) \quad \text{総減量値は適正である} \quad \text{月毎減量値は適正である}$

- [月毎減量値は適正である]

- 月毎減量値は適正である $\text{abs}(1 \text{ カ月前の体重} - \text{現体重}) \quad \text{月毎減量値}$

- [総減量値は適正である]

- 総減量値は適正である $\text{abs}(\text{現体重} - \text{目標体重}) \quad \text{月毎減量値} * \text{減量期間}$

事前条件・事後条件を使う

- Fusionを使った例

操作: 距離探索 (from: 駅, to: 駅)
責任: from 駅から to 駅までの営業キロ数を返す
入力: from = 出発駅, to = 到着駅
返回值: from 駅から to 駅までの営業キロ数
変更オブジェクト: 鉄道グラフ
事前条件: 探索済駅集合 = , 探索駅集合 = 駅-set
事後条件:
 (return = 距離 (from, to) to 探索済駅集合)
 (探索済駅集合 = 駅-set 探索済駅集合 to =)
 from 駅-set =

分析や設計局面の 終了条件が見えない

- シナリオに基づいたウォークスルーでエラーがなくなったら
 - 作成者が召集
 - 具体的なデータに基づいてテストする
 - その場で直さない
 - 誰の責任か問わない
 - マネージャーを入れない
 - 1時間半程度で終わる
- 再利用性・保守性分析をどこまでやるかは、プロジェクト方針次第
 - 一般には品質をできるだけ追求した方が、生産性が上がる
 - DeMarcoの法則（ピープルウェア）

誰が（あるいは、どれが） 正しいオブジェクトと検証するのか？

- シナリオに基づいたウォークスルーで検証する
- 正しいモデルはいくつもある
- 間違いを徐々に排除していく
 - 主要目標
 - 保守性
 - 再利用性
 - 拡張性
 - 評価項目
 - 小さいモジュール
 - 単純なインタフェース
 - 少なく・小さく・明示的なインタフェース
 - 情報隠蔽
 - 解放 / 閉鎖の両立

手順は絶対か

- 出張日数の計算はどうするか？

- 今日の12時に出張にでかけ明日の11時に帰ってきた場合、普通は出張日数を2日と計算する会社が多い。この仕様はどう書けばよいか？

- 日付計算で、春分の日と秋分の日の計算はどうするか？

- ほとんどの会社では、毎年春分の日と秋分の日の表を変更して計算するカレンダールーチンを使っている
- もちろん、正式の決定は年末に発表されるのだが、実用的な意味では、今後100年以上はあらかじめ春分の日と秋分の日を計算しておくことができる
- 従って、毎年年末に表を修正するという仕様は、保守性を考えていない分析ということになる
- この仕様はどう書いたらよいか？

- 手を抜いてはいけない

- 「モデルを作るのが大変だから」というのは駄目

- エラーの検出効率を最大にするのが目的

- この目的を逸脱しないなら、標準手順を変えてよい場合もある
- 最初のうちは定石通りに

- オブジェクトモデルと動的モデルと機能モデルのあいだは何回もフィードバックする

分析は概要だけにして、 設計に進みたいのだが？

要求記述文

- 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
- システムが何をすべきかというモデルを作る

問題領域を対象
要求事項
アプリケーションの内容
仮定
性能要求

設計 & 実現

汎用的アプローチ
アルゴリズム
データ構造
アーキテクチャ
最適化

分析は必ずデータ（オブジェクト） 中心でやらなければならないのか？

- データが重要で余り変化しないシステム（事務処理など）ではデータ（オブジェクト）中心
 - オブジェクトモデルを先に作る
 - データ（オブジェクト）の洗い出しが最初のステップ
- イベントが重要で、重要なデータがあまり無いシステムでは、振る舞い（イベント）中心
 - 動的モデルを先に作る
 - シナリオ作りが最初のステップ

ドキュメント化

- どの局面で「何」を実施し、
どういう成果物を残すのかが見えない
- 成果物を作成するための具体的な作業項目が見えない
 - Booch法・Coad法などでは曖昧だが、OMT法でははっきりしている

どの局面で「何」を実施し、 どういう成果物を残すのかが見えない

- Booch法などでは曖昧だが、OMT法でははっきりしている
- 成果物
 - OOモデル
 - オブジェクト図
 - 操作の仕様を含む
 - 状態遷移図
 - シナリオ図
 - OOデータフロー図
 - 操作の仕様を含む
 - オブジェクト・メッセージ図
 - シナリオ
 - Use Case
 - データ辞書
 - なぜそうしたかの記述
 - プログラム
 - プロジェクトの履歴

オブジェクト指向設計

- 設計手順
- サブシステムへ分割する指針は何か？
- データストアの実現方法の基本戦略を選択するための指針は何ですか？
- データ構造とアルゴリズムなど適当でよいのでは？
- 設計からそのままプログラミングへ移行できるのか
- 宣言的仕様を手続き的アルゴリズムへ書き換えるのはどうするのか？
- 宣言的仕様から手続き的アルゴリズムへの変換例
- 形式仕様記述法を知らなくても...
- 形式仕様記述法もアルゴリズムも知らなければ...
- 仕様記述は
どうするのか？
- 実行効率はこのあたりで考えるのか？
- データへアクセスするパスの最適化はどうやるのか？
- クラス構造を吟味して継承を
増やすのは分析でもやったが？
- 関連はどうやって実現するのか？
- オブジェクトの属性の表現方法を定めるのは、いつやるのか？
- オブジェクト指向で設計すると
効率が悪いのでは？
- どのようなクラスライブラリーがあるのか？
- プロトタイピングは有効か？
- 設計上の決定の
ドキュメント化は必要か？

設計手順

- システム設計

- サブシステムへ分割する
 - レイヤーとパーティション
- データストアの実現方法の基本戦略を選択する
 - DBかファイルかデータ構造か？
- 共通リソースの扱いを決定する
 - 守護オブジェクトの使用

- オブジェクト設計

- 操作を実現するためのアルゴリズムの設計
 - 宣言的仕様を手続き的アルゴリズムへ書き換える
 - 操作のコストを最小にするアルゴリズムを選択する
 - アルゴリズムにあったデータ構造を選択する
- データへアクセスするパスの最適化
- クラス構造を吟味して継承を増やす
- 関連の実現方法を設計する

サブシステムへ分割する指針は何か？

- サブシステム間のインタフェースが少なく、個々のサブシステムの強度が高くなるように分割する
- 保守性や再利用性を考えて、できるだけ顧客-供給者関係に分解することが望ましい
- サブシステムへの分割は、縦方向（パーティション）と横方向（レイヤー）の2種類が考えられる

データストアの実現方法の基本戦略 を選択するための指針は何ですか？

- DBMSを使うべきデータ
 - 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
 - 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
- メモリ上のデータ
 - 速いアクセスが要求されるデータ
 - 比較的少量のデータ
 - 各クラスがメモリ上のデータに直接アクセスするのではなく、アクセスするためのメソッドを使う

データ構造とアルゴリズムなど適当で よいのでは？

- アルゴリズムの計算量
- NP完全問題
- 解けそうにない問題をどう見つけ解決するのか？
- 自分ではアルゴリズムは分からないが、どうするか？

アルゴリズムの計算量

	計算量	問題例
決定不能	解けない	2つのプログラムの同値性判定
非常に難しい	2^n 乗	Presburger論理式の真偽判定
難しいかどうか不明	不明	ナップザック問題など
かなり難しい	n の <i>i</i> 乗	探索・整列など
やさしい	$\log n$	木探索・ハッシュ検索など

? (fact 361)

```
1437923258884890654832362511499863354754907538644755876127282765299227795534389618856841908
0031411960714137944348905859683839682333043216077138088370565578796691924861827097800358990
2110057945010733305079262777172275041226808677528136885057526541812043502150623466302643442
6736326270927646433025577722695595343233942204301825548143785112222186834487969871267194205
6095333064139357106351972007214733787338269803085351043174203653673779887217565513450041291
0616505061544962655811028242414284066270545855623101563752892899924857388316647687165212001
5362189137337137682618614562954409007743375894907714439917299937133680728459000034496420337
0664408533370012842864126543944950507739545600000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

NP完全問題

- 現実の問題にかなり含まれている
 - 危ない言葉
 - 組み合わせ
 - 東京証券取引所システムのトラブル
 - 最適解
 - 時間割作成システム
 - 例
 - 巡回セールスマン問題
 - ナップザック問題

解けそうにない問題をどう見つけ解決するのか？

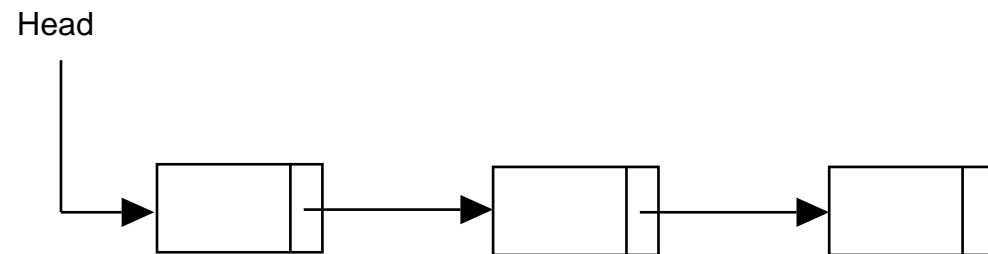
- 実用的には解けることもある
 - 分枝限定法
 - バックトラック
 - 枝刈り
 - チェス
 - 国のチャンピオンクラス
 - 将棋
 - アマチュア 2 段くらい
 - 囲碁
 - アマチュア 5 級くらい
- 動的計画法
 - 部分的な解を表で持つ
 - 表に要素を 1 個付け加え、最適解を計算し直す
- 近似アルゴリズム
 - 最善の解をあきらめて、近似値を求める
 - 巡回セールスマン問題
 - 999都市くらいまで実用的な時間で解ける
 - しらみつぶしだと、30都市でも大変

自分ではアルゴリズムは分からない が、どうするか？

- 自分でやる
 - 教科書を集める
 - 各データ構造の特徴を覚える
 - リスト
 - 木
 - ハッシュ
- 人にやってもらう
 - データ構造とアルゴリズムをクラスライブラリーとして実現しておいてもらう
- 分からなければ、専門家に聞く

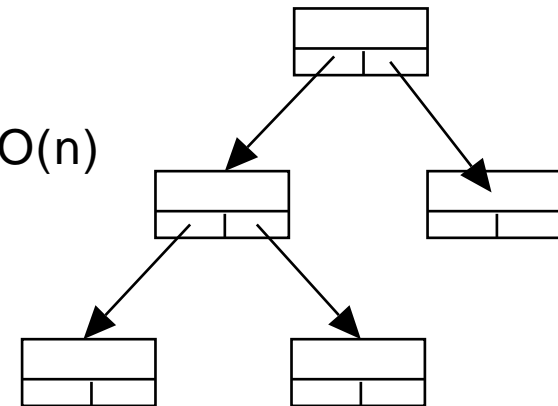
リスト

- 性質
 - 個々の参照は早くない
 - 挿入や削除が早い
- 種類
 - 双方向リスト
 - 環状リスト

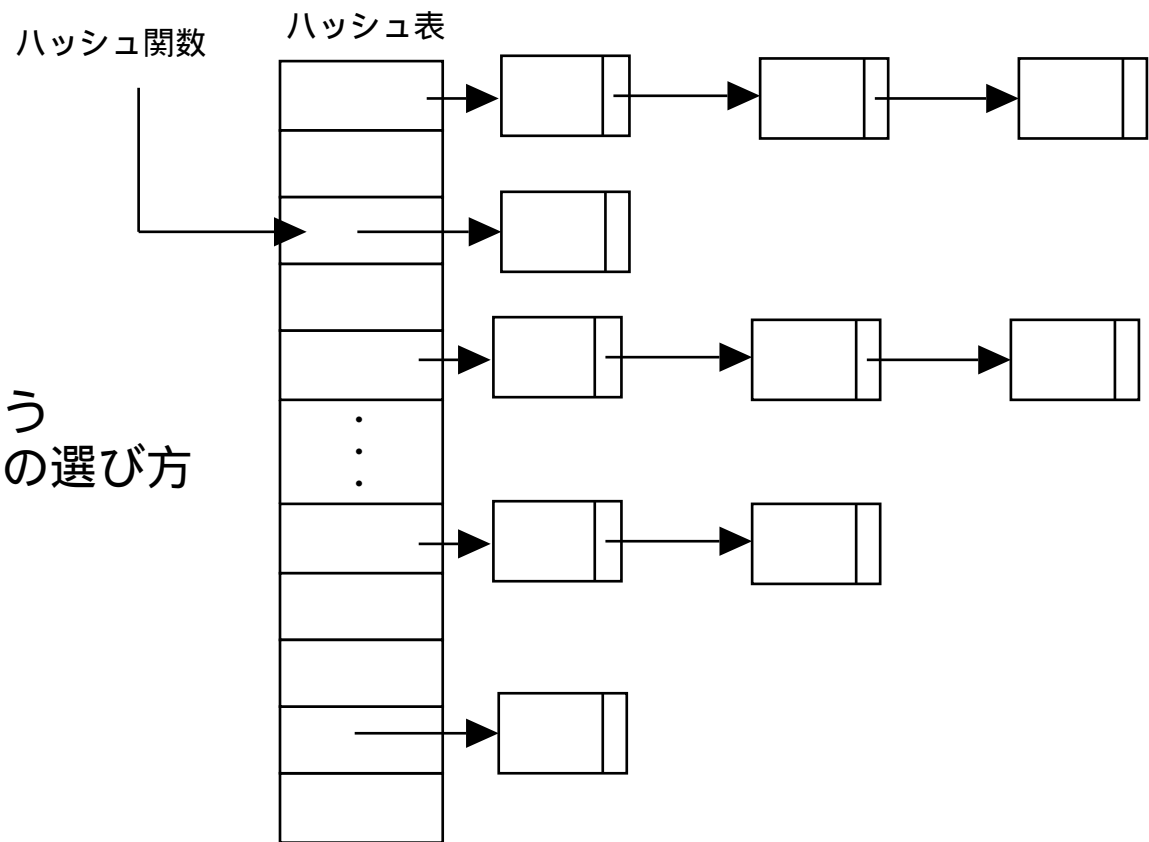


木

- 性質
 - 階層的なものごとの構造を表現するのに便利
 - 挿入や削除や検索が早い
 - 挿入しつつソートするのに便利
- 種類
 - 2分木
 - もっとも利用頻度の高い木
 - 最善および平均の場合 $O(\log n)$ 、最悪の場合 $O(n)$
 - 最悪の場合、リストと同じ
 - AVL木
 - 最悪の場合でも $O(\log n)$ の計算量
 - B木
 - 最悪の場合でも $O(\log n)$ の計算量
 - 2次記憶のデータの格納に向いている
 - m分木なので、 $m=200$ とすると100万件のデータを、最悪でも4回の読み出しで探索できる
 - 2次記憶は、少量のデータでもある程度まとまった量のデータでも、アクセス回数と同じならば読み書きの時間はあまり変わらない



ハッシュ

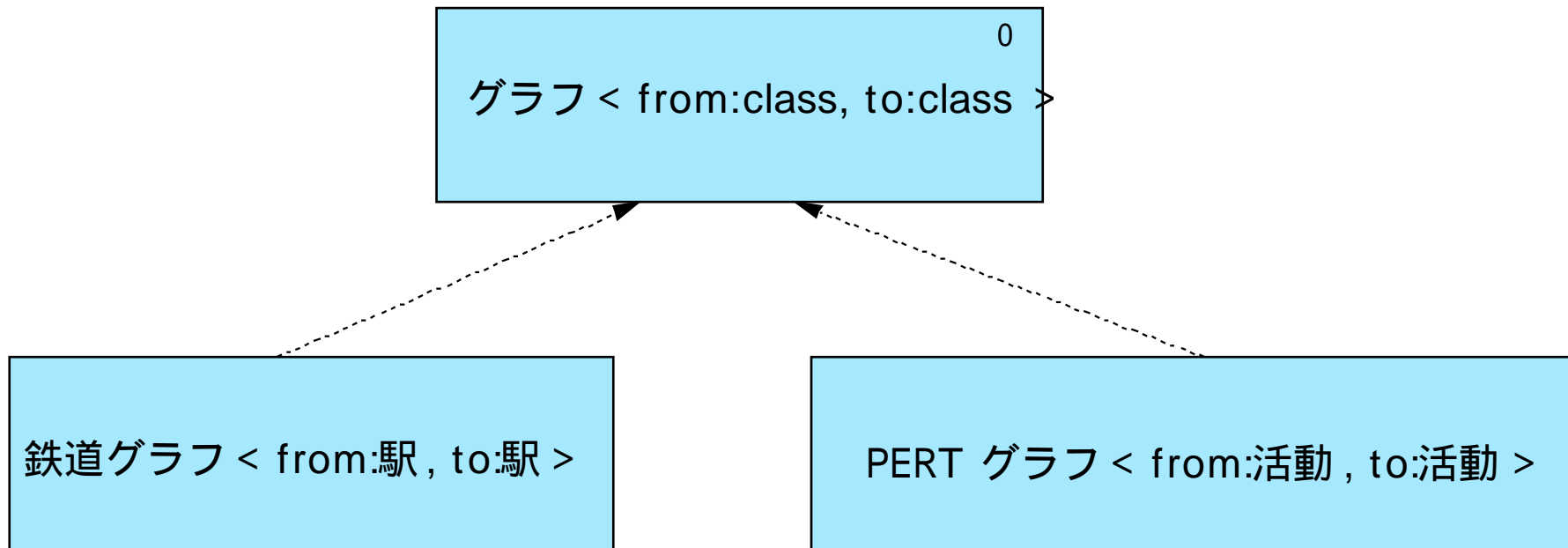


- 性質
 - $O(1)$ の計算量
 - メモリーをやや喰う
 - ハッシュ関数の選び方で異なる

設計からそのままプログラミングへ 移行できるのか

- 実装言語を意識した詳細設計が必要
 - パラメータ化クラス
 - スコープの設定
 - 疑似コード
 - パラメータの型・初期値の確定

パラメータ化クラス



スコープの設定

- クラス全体から見える\$
- public+
- protected#
- private -

会員

+ 入会年月日:Date
+ 会員番号:会員コード

+ \$ 入会する
+ \$ 会費未納者は？
+ 退会する
+ 会費納入する

駅

+名前:String
座標:Point
- 走査済:Bool = false
/ + \$ 駅数

+ \$ create
+ \$ 駅数？
+ 距離？ (to:駅)
- 走査済 (mark:Bool)

宣言的仕様を手続き的アルゴリズムへ 書き換えるのはどうするのか？

- 形式仕様記述の教科書の中に、宣言的仕様から手続き的仕様への変換公式が多数あるので、それらを使って段階的に変換する
 - 「グリース 著、筧 訳。プログラミングの科学。培風館、1991」
 - 「ポター 他著、田中 監訳。ソフトウェア仕様記述 先進技法-Z言語。トッパン、1993」
- 実用サイズのシステムでは、形式仕様記述言語支援ツールの助けを借りる
 - RAISE Tool
 - <http://dream.dai.ed.ac.uk/raise/>
 - VDM-SL Toolbox
 - <http://www.ifad.dk/>
- 「データ構造とアルゴリズム」といった本から探す
- 対象分野のアルゴリズムを書いた本から探す

宣言的仕様から手続き的アルゴリズム ムへの変換例

- 宣言的仕様
- 変換の準備
- 変換公式の適用
- 繰り返しへの変換
- 2分探索への変換

宣言的仕様

value

floor : Real → Nat /* floor(1.3) = 1 */

sq_root : Real → Real /* sq_root(25.0) = 5.0 */

fl_sq : (Real → Real) → Real → Nat /* fl_sq(29.0) = 5 */

axiom

[floor]

r : Real, i : Nat • floor(r) as i

post i ≤ r < i+1

[sq_root]

n : Nat, s : Real • sq_root(n) as s

post s² = n ∧ s ≥ 0.0

[fl_sq]

fl_sq = floor ∘ sq_root

変換の準備

/ floorの定義から */*

```
fl_sq(n)      n : Nat,  i : Int • fl_sq(n) as i
  post i      sq_root(n) < i+1
```

/ sq_rootの定義から */*

```
fl_sq(n)      n : Nat,  i : Int • fl_sq(n) as i
  post i2    n < (i+1)2  i ≥ 0  /* i2 はiの2乗 */
```

/ i+1 を qで置き換える */*

```
fl_sq(n)      n : Nat,  i, q : Int • fl_sq(n) as i
  post i2    n < q2  i ≥ 0  i+1 = q
```

/ P i² n < q² i ≥ 0とすると */*

```
fl_sq(n)      n : Nat,  i, q : Int • fl_sq(n) as i
  post P      i+1 = q
```

変換公式の適用

/* 詳細化の公式sequential compositionを適用して変形すると */
fl_sq(n) post P ; post P i+1 = q pre P

「段階的詳細化の公式sequential composition」を適用してこのように変形したのは、fl_sq(n) を解くには解の範囲を絞り込むために反復計算が必要であり、その反復を終了する条件と、反復の間変化しない不変条件を見つけようという「意図」がある。

もちろん、このような「意図」は対象問題領域の知識（今の場合、平方根に関する数学の知識や数値計算の知識）なしには発生し得ない。

分析モデルの宣言的仕様を、機械的に設計モデルの手続的仕様に変換できない所以である。

繰り返しへの変換

/* Pが真になるには、明らかに $i < q$ でなければならず、 $q - i$ を可変部として、その値を減らしていき、 $i + 1 = q$ となったときの i が答えである。不変条件をPとし、繰り返しの条件は $i + 1 = q$ の否定を持ってくればよい。ここで i_0, q_0 は、それぞれ i, q の初期値である。 */

fl_sq(n)

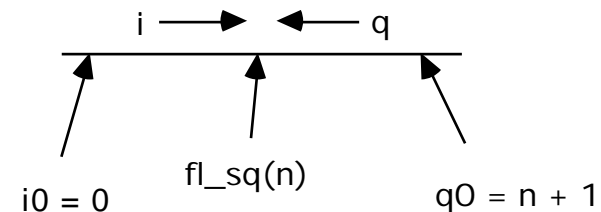
post P ;

/* 不変条件 P */

while $i + 1 \leq q$ do

post $q - i < q_0 - i_0$ pre $i + 1 \leq q$

end



この変形で、while文の条件に「 $i + 1 = q$ 」の否定「 $i + 1 \leq q$ 」を持ってくるのも、不変条件を決めたときからの「意図」である。

事後条件「 $(i \geq 2 \wedge n < q \leq 2) \wedge (i = 0) \wedge (i + 1 = q)$ 」の項を一つ取り除いて不変条件を作り、取り除いた項「 $i + 1 = q$ 」の否定「 $i + 1 \leq q$ 」を反復の条件にするのは、「段階的詳細化」の定石のひとつである

2分探索への変換

/* 最初の「post P;」は、「 $i := 0; q := n + 1$ 」すなわち $i_0 = 0, q_0 = n + 1$ とすれば真になり条件を満たす。また、区間 $q - i$ を反復のたびに1ずつ減らしていくのも一つの方法だが、ここでは区間 $q - i$ を半減していく方法を採用する。要するに2分探索法である。 */

```
fl_sq(n)
```

```
  i := 0;
```

```
  q := n + 1;
```

```
  /* 不変条件 P */
```

```
  while i + 1 < q do
```

```
    d := (i + q) / 2;
```

```
    if d <= n then
```

```
      i := d
```

```
    else
```

```
      q := d;
```

```
  end
```

形式仕様記述法を知らなくても...

- もちろん、アルゴリズムについて知識があり、 n 変数非線形関数（平方根を求めるのもこの範疇）の解を求めるのにニュートン法が使えることや、2分探索法を知っていれば、宣言的仕様から最終段階に直接変換することができる
- 「形式的仕様記述」の教科書を勉強する暇がない方でも、「データ構造とアルゴリズム」の勉強をある程度やっていれば、既存のアルゴリズムで解決できる問題かその周辺の問題ならば、手続的仕様とする事ができるわけである

形式仕様記述法もアルゴリズムも 知らなければ...

- 「形式的仕様記述」と「データ構造とアルゴリズム」のいずれも勉強していなければ、分析モデルの宣言的仕様を手続的仕様に変換するのは、困難になる
- この場合、そもそも信頼性のある設計をすることはほぼ不可能になり、設計上の決定は「偶然」が支配することになる

仕様記述は どうするのか？

- 自然言語では？
- 仕様記述言語を使うと？
- 疑似コードを使うと？

自然言語では？

- 「**購買金額が一定額以上の顧客の報告書を作成する**」という、操作「**選択する**」の仕様があるとする
 - 仕様は簡潔であるが、明確とは言いがたい
 - 例えば、購買金額が負であることは想定しなくてよいはずだから、購買金額は自然数なのだが、そのことはどこにも書いていない
 - 従って、プログラムを作るときに、負の場合を排除することを忘れてしまう可能性が高い
 - また、一定額以上の購買金額の指定方法が書いていないため、プログラマーがプログラムで「一定額」を定数で持ってしまう可能性もある
 - もちろん、これではまずく、「一定額」はパラメータとしたい
 - さらに、「一定額」以上かどうかを判定する部分と、「選択する」操作の仕様本体とは分離しておいた方が、あとあと、顧客の集合から抽出する条件を変更するときにより便利なのだが、その保証も得られない

仕様記述言語を使うと？

type

```
購買金額 = Nat,  
顧客データ,  
顧客レコード = 購買金額 × 顧客データ,  
顧客データベース = 顧客レコード-set,  
報告書_データ,  
報告書_レコード = 購買金額 × 報告書_データ,  
報告書 = 報告書_レコード-set
```

variable

```
database : 顧客データベース,  
report : 報告書
```

value

```
対象購買金額か? : 購買金額 × 購買金額 Bool,  
編集する : 顧客データ 報告書_データ, /* 関数本体はまだ未定義 */  
選択する : 購買金額 read database write report Unit
```

axiom

```
対象購買金額か? ( 購買金額, 一定額 )  
  購買金額 一定額,  
  選択する ( 一定額 )  
  report := {};  
  for ( amount, data ) in database ·  
    対象購買金額か? ( amount, 一定額 ) do  
      report := report  {(amount, 編集する ( data ))}  
  end
```

疑似コードを使うと？

- 自然言語よりは明確に仕様を記述することができる
- 仕様記述言語より覚えることが少なくて済む
- しかし、構文的な欠陥を見つけることは難しい
- 「構造化文」の文法を設計する開発コストがかかる

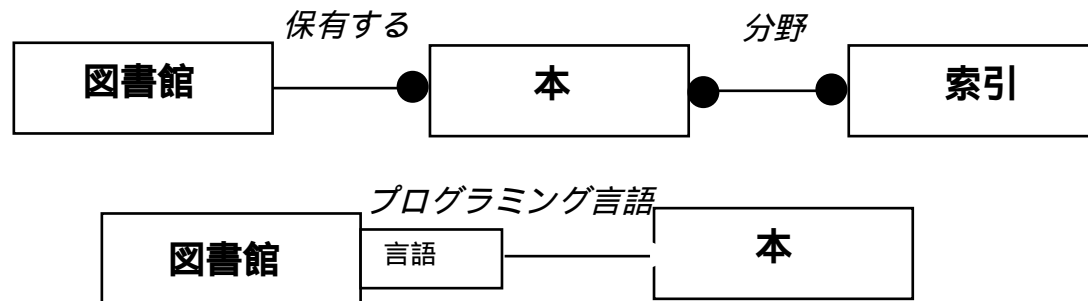
実行効率はそのあたりで考えるのか？

- 基本的には「オブジェクト設計」工程で考える
- 最適化したアルゴリズムは読みにくく変更しづらいので、まず最も単純なアルゴリズムで実現し、計測してから、必要な部分だけ最適化する
- システム全体の効率に関わる操作は、全体の2～3%であることがKnuthらの研究によって明らかになっている
- 従って、より低レベルの言語で実現するとか、反復文中の文を反復文の外に移動するなどの、ミクロの効率化をシステム全体にわたって考えるのはコスト的に非常に無駄になる
- 効率化を図る上で、最大の効果があるのは適切なデータ構造とアルゴリズムの選択である
 - この選択の仕方いかんで、数10倍から数百倍あるいは数万倍の効率の差が出る可能性がある

データへアクセスするパスの最適化 はどうやるのか？

- 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
- 効率化のため
計算順序を変える
- 複雑な計算をやり直さないため計算結果を保存する

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



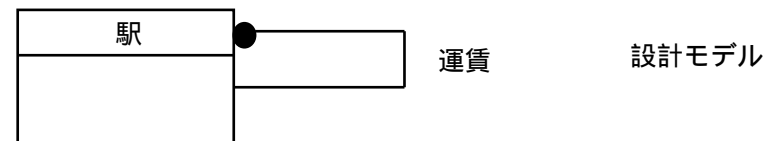
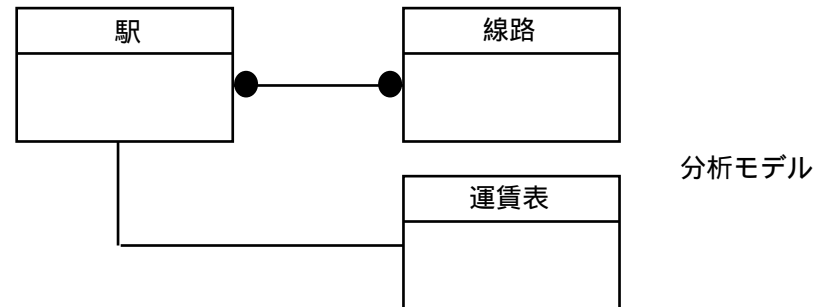
- ある情報を得るためにどの関連をたどる必要があるか？
- 双方向にたどらなければならない関連は何か？
- 1方向だけたどれば良い関連は何か？
 - 1方向だけの関連は効率がよい
- 操作はどれくらい呼ばれるか？そのコストは？
- 最後のクラスのオブジェクトがどれくらい「ヒット」するか？
 - ほとんどのオブジェクトが「ヒット」しないのなら、単純ループは非効率

効率化のため 計算順序を変える

- 計算対象の集合をなるべく早く小さくする
 - 例えば、プログラムに言及した小説を探す場合を考えよう
 - プログラムという索引を持つ本が1000冊、小説は10万冊あるとし、両方の索引を持つ本が50冊だとする
 - このとき、まず、索引として「プログラム」を持つ本の集合を抽出し、次に「小説」を抽出すれば、的中率は20分の1で済む
 - 一方、最初に「小説」を抽出し、次に「プログラム」を抽出すると、的中率は2000分の1になってしまう。

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
- 派生クラス・派生属性・派生関連に計算結果を保存する



派生クラス・派生属性・派生関連に 計算結果を保存する

- 元の属性が変化したとき、派生属性も更新しなければならない
- 更新がいつ必要か認識する方法
 - 陽に更新を指定
 - 設計者がいつ更新するか決定する
 - 定期的な再計算
 - 属性の変化が連続して起こるとき、まとめて定期的に派生属性を再計算することもある
 - 能動的な値
 - その値に依存して変化する他の値があるとき、能動的な値と言う
 - プログラミングシステムによっては、この依存性を設定できる場合がある
 - すなわち、元の値が変化すると、導出される値も自動的に更新される

クラス構造を吟味して継承を増やすのは分析でもやったが？

- 分析でやったのは問題領域の共通性を継承構造に反映しただけ
- ここでは、データ構造やアルゴリズムの共通性を反映して継承構造を作る

関連はどうやって実現するのか？

- 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
 - 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
 - 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

オブジェクトの属性の表現方法を決めるのは、いつやるのか？

- 「オブジェクト設計」工程で行う
- この段階では、まだ言語に依存した属性の決定はしなくてよい
 - 例えば、何らかの「状態」を表す属性を考えたとき、その属性の型を「状態」と定義しておけばよく、状態は整数で表せるから「整数」型にするといった決定をする必要はない

オブジェクト指向で設計すると 効率が悪いのでは？

- サブルーチン呼び出しとメソッド呼び出しの差は最大50%
 - アルゴリズムの差の方がはるかに大きい
 - 10 ~ 1000 倍の差がつく
 - 良いアルゴリズムの効率よいクラス・ライブラリーを揃えれば、かえって速くなる
- ミクロの効率化とマクロの効率化
 - 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
 - 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
 - システム全体の効率に関する操作は全体の2 ~ 3%
 - 残りの97 ~ 98%の部分を最適化しても無駄

どのようなクラスライブラリーがあるのか？

- コンテナークラス
 - 配列・リスト・集合・辞書・木・ハッシュ表・スタック・待ち行列・行列・グラフなど
- 大きさのあるクラス
 - 数・日付・時間・座標・関連 (Association) など
- GUIクラス
 - 各種ウィンドウ・フィールド・ボタン・メニュー・グラフ・表など
- OS インタフェース
 - ディレクトリー・ファイル・イベント・プロセスなど
- 問題領域別クラス
 - 有価証券・統計・有限状態マシンなど

プロトタイピングは有効か？

- 分析段階のプロトタイピングは仕様のユーザーの要求確認のため
- 設計段階のプロトタイピングは、技術的問題点の明確化のため
 - 実現可能性の調査
 - 効率の実験
 - 未体験技術の確認
- コンポーネントウェアやGUI構築ツールなどを使うと早くできる
 - SmalltalkやCLOSなどをベースにしたものが良い

設計上の決定の ドキュメント化は必要か？

- 設計上の決定はその場でドキュメントにすべきである
 - そうしないと、後で忘れてたり混乱したりする
 - どこに書くか？
 - ハイパーテキストシステム上
- 視点が異なるので、分析ドキュメントと設計ドキュメントは別にする
- 構成管理ツールや版管理ツールが必須

オブジェクト指向プログラミング

- 設計から実現へはどう持っていくのですか？
- 既存クラス・ライブラリーが膨大すぎて、どこから手を付けてよいのか分からないのですが？
- 設計モデルのイベントを、イベントとして実現するか操作にするかの指針は何ですか？
- プログラミング上の注意点は何ですか？
- オブジェクト指向言語で実現するのは遅くありませんか？
- どのオブジェクト指向言語が良いのですか？
- コンポーネントウェアを使うと、プログラムが早く作れるのでしょうか？

設計から実現へはどう持っていくのですか？

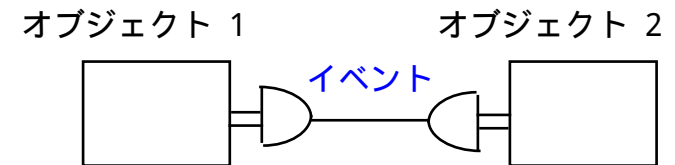
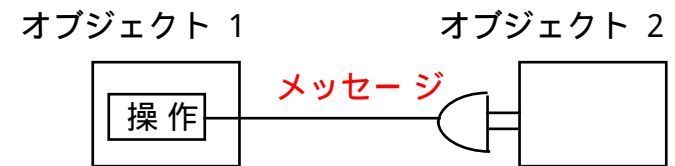
- 設計段階のオブジェクト図は言語独立なので、クラス階層をプログラミング言語にあわせて修正する
 - ここでは、独自のクラス階層を作り出すより、既存クラス・ライブラリーを調査し、既存のクラスのサブクラスとして実現することが多い。
- 属性の型をプログラミング言語に最適なものにする
 - すべての属性について、型あるいは所属クラスを決めなければならない。
- 動的モデルのイベントの実現方法を決める
 - イベントとして実現するか操作にするか決める。

既存クラス・ライブラリーが膨大すぎて、どこから手を付けてよいのか分からないのですが？

- 大きく3つのクラス群に分けて考える
 - 対象問題領域のモデルを記述しているクラス群
 - コンテナ・クラス（Smalltalkの場合Collectionクラス）のサブクラスとなる場合が多い
 - モデルの見え方（ビュー）を記述しているクラス群
 - 最近では、GUIに関わるクラスを設計する必要はあまり無く、コンポーネントウェアやGUI構築ツールを使えば、クラスをあまり意識せずにGUIを構築できることも多い
 - アプリケーションの構造を記述するクラス群
 - コンポーネントウェアなどでは、アプリケーション・フレームワーク・クラスまったく意識せずにアプリケーションを構築できるようになってきた

設計モデルのイベントを、イベントとして実現するか操作にするかの指針は何ですか？

- 操作は、**メッセージ**という形で2つのオブジェクトを接続するため、効率は良いものの、2つのオブジェクトは比較的密に結合してしまう
- **イベント**は、2つのオブジェクトとは独立の概念なので、効率は悪いものの、付け替えが比較的容易であり、2つのオブジェクトの結合は緩やかになり、独立性は高まる



プログラミング上の注意点は何ですか？

- 再利用性
- 拡張性
- 頑丈さ
- 大規模プログラミングの作法
- Smalltalkのプログラミングの注意点

再利用性

- 再利用性のための作法
 - メソッドを小さくする
 - メソッドの強度を高める
 - 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、全ての組み合わ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
 - メソッドを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
 - 共通コードをくくり出して、親クラスに書く

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

```
normalize
```

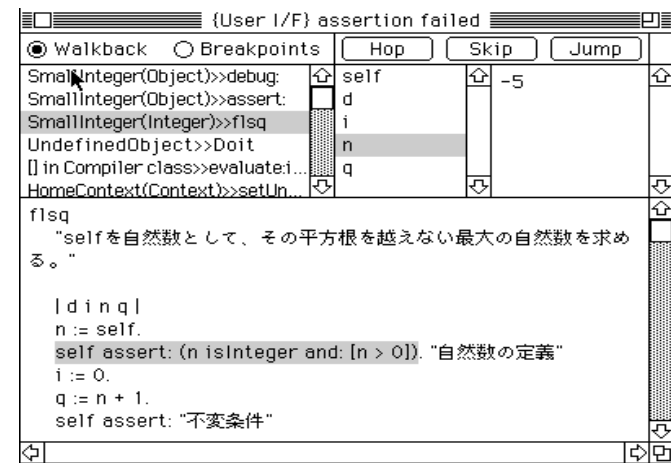
```
"Private -- Queueを正規化する。すなわち、先頭リストheadsが空なら、後尾リストtailの逆転リストを先頭リストに入れ、後尾リストを空にする。"
```

```
    !rev!  
(self heads isNil or: [self heads isEmpty])  
    ifTrue: [self tails isNil ifTrue: [^self].  
            rev := self tails reverse.  
            self tails: nil.  
            ^self heads: rev; yourself]  
    ifFalse: [^self]
```

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
- 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

```
self assert: "不変条件"  
(((i raisedToInteger: 2) <= n) and:  
[(n < (q raisedToInteger: 2)) and: [i >= 0]]).  
"raisedToInteger:Nは、整数のN乗"
```



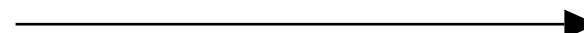
大規模プログラミングの作法

- 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

Smalltalkのプログラミングの注意点

- 既存のクラスのメッセージを調査し、新たに作る必要があるクラス(MeiTree)のメッセージと重なる部分が多いクラスをスーパークラス(Collection)とする
- 抽象クラスであるスーパークラスの抽象メッセージ(add:, do:, remove:ifAbsent:)を実現する
- do:メッセージを実現する
- printOn:メッセージを実現する
- あるクラスを構築する時、そのクラスのインスタンスの空値を決定する
- 新たに作ったクラスのメッセージだけでなく、スーパークラスのメッセージもテストする

do:メッセージを実現する

- collect:, reject:などが同時に実現される
- オブジェクト自身が繰り返し方を知っている
 - inorder: aBlock
 - "木を間順に走査し、ノードの要素でaBlockを評価する"
 - "do: inorder:"
 - ! node !
 - node := self root. 
 - node isEmpty ifTrue: [^self].
 - node isLeaf ifTrue:
 - [aBlock value: node element]
 - ifFalse:
 - [node leftTree inorder: aBlock.
 - aBlock value: node element.
 - node rightTree inorder: aBlock]

強度が低下するので、
インスタンス変数root
を直接呼ばない

```
!t s!  
t := MeiBinaryTree new.  
t addAll: #('5' '1' '2' '3' '4').  
s := ''.  
t do: [:each ! s := s , each , ' '].  
s '1 2 3 4 5'
```

printOn:メッセージを実現する

- 評価した結果を表示するメッセージ

- printOn: aStream level: anInteger
-
- | node |
- node := self root.
- (node isNil or: [node isEmpty]) ifTrue: [
 - aStream nextBytePut: 13.
 - anInteger timesRepeat: [aStream nextPutAll: '.'].
 - nil printOn: aStream]
- ifFalse: [
 - aStream nextBytePut: 13.
 - anInteger timesRepeat: [aStream nextPutAll: '.'].
 - node element printOn: aStream.
 - node leftTree printOn: aStream level: anInteger + 1.
 - node rightTree printOn: aStream level: anInteger + 1]

```
50 ==> 'nina'  
.49 ==> 'bun'  
..nil  
..nil  
.103 ==> 'shin'  
..54 ==> 'anna'  
...nil  
...nil  
..nil
```

あるクラスを構築する時、そのクラスのインスタンスの空値を決定する

- 番兵オブジェクトを決めると良い
 - nilを持つクラスUndefinedObjectで、作ったクラスのメッセージに応えると、情報隠蔽に問題が起こる
 - MeiBinaryTree>>isEmpty
 - "ノードの要素がnilであれば空とする"
 -
 - ^self root element isNil

新たに作ったクラスのメッセージだけでなく、スーパークラスのメッセージもテストする

- addAll:, asArray, asSortedCollection, asString, collect:, detect:, includes:, inject:into:, reject:, removeAll, sizeなど
 - addAll:とcollect:とasArrayのテスト例
 - !t c!
 - t := MeiBinaryTree new.
 - t addAll: #(5 1 2 3 4).
 - c := t collect: [:each ! each raisedTo: 2].
 - c asArray (1.0 4.0 9.0 16.0 25.0)

オブジェクト指向言語で実現するのは遅くありませんか？

- 初期のSmalltalkやLispベースの言語がインタプリタだったため
 - 現在はインクリメンタル・コンパイラ
 - 動的束縛が遅い原因のひとつ
 - 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
 - 現在では高々50%遅いだけ
 - 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
- 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

どのオブジェクト指向言語が良いのですか？

	C++	Smalltalk	CLOS	Eiffel	Objective-C
強い型付けによるチェック	Y	N	N	Y	Y
可視性					
顧客からアクセスの制御	Y	Y	N	Y	Y
サブクラスからのアクセスの制御	Y	N	N	Y	N
パラメータ化クラス	Y	不要	不要	Y	N
表明と制約	N	N	N	Y	N
実行時のメタデータ	N	Y	Y	N	Y
ガーベージコレクション	N	Y	Y	Y	N
効率					
可能なとき静的束縛	Y	常に動的束縛	常に動的束縛	Y	Y
標準クラスライブラリー	N	Y	N	Y	Y
多重継承	Y	N	Y	Y	N

コンポーネントウェアを使うと、プログラムが早く作れるのでしょうか？

- ファンクションポイント法
 - 1FP当たりのソースコード行数

言語	コード行数/1FP
アセンブラ	320
C	150
COBOL	106
FORTRAN	106
Pascal	91
PL/I	80
Ada	71
Prolog	64
APL	32
Smalltalk	21
Visual Smalltalk	10
スプレッドシート	6

オブジェクト指向プロジェクトの管理

- ソフトウェア・プロジェクト管理に関する定石
- ピープルウェアに関する定石
- プロジェクト管理に関する定石

ソフトウェア・プロジェクト管理に関する定石

- OOでソフトウェアの生産性は上がるのですか？
- オブジェクト指向を使えばプロジェクトはうまくいくのか？
- オブジェクト指向技術をなかなか覚えられないのだが
- コンポーネントウェアとか良いツールがあるのだから、分析や設計は大概にしてプログラムを作っては駄目か？
- オブジェクト指向を使ってプロジェクトの遅れを取り戻せるか？
- プロダクト

OOでソフトウェアの生産性は上がるのですか？

- ソフトウェアの生産性は低い
(宮本)
 - 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

ソフトウェア生産性とコストの例

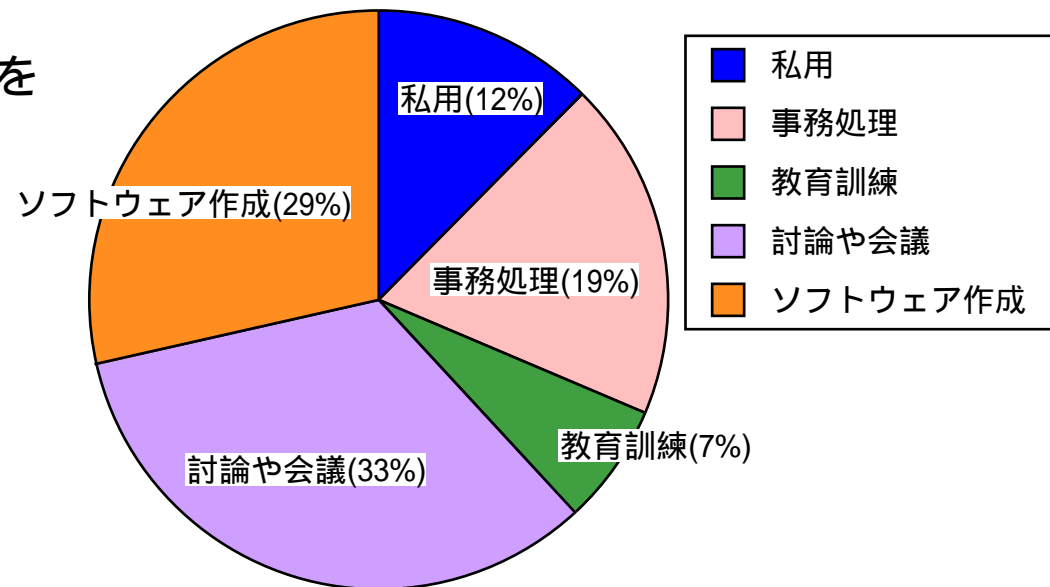
- a. 生産性の例 (従来)
- システム・プログラミングの場合：10～15行/日
 - 事務処理プログラミングの場合：30～50行/日
 - 1人月当たりの平均プログラミング生産性：200～1000行/人月
- b. ソフトウェア・コストの例 (従来)
- アメリカの例
 - 1953年 約\$4.00/命令
 - 1975年 約\$8.00/命令
 - 1980年 約\$5.00～\$25.00/命令
 - 日本の例
 - 1980年 ¥1000～¥5000/命令
 - 特殊な例 (アメリカの指令制御システム)
 - 開発コスト=\$75/命令
 - 保守コスト=\$4000/命令
- c. 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

オブジェクト指向を使えばプロジェクトはうまくいくのか？

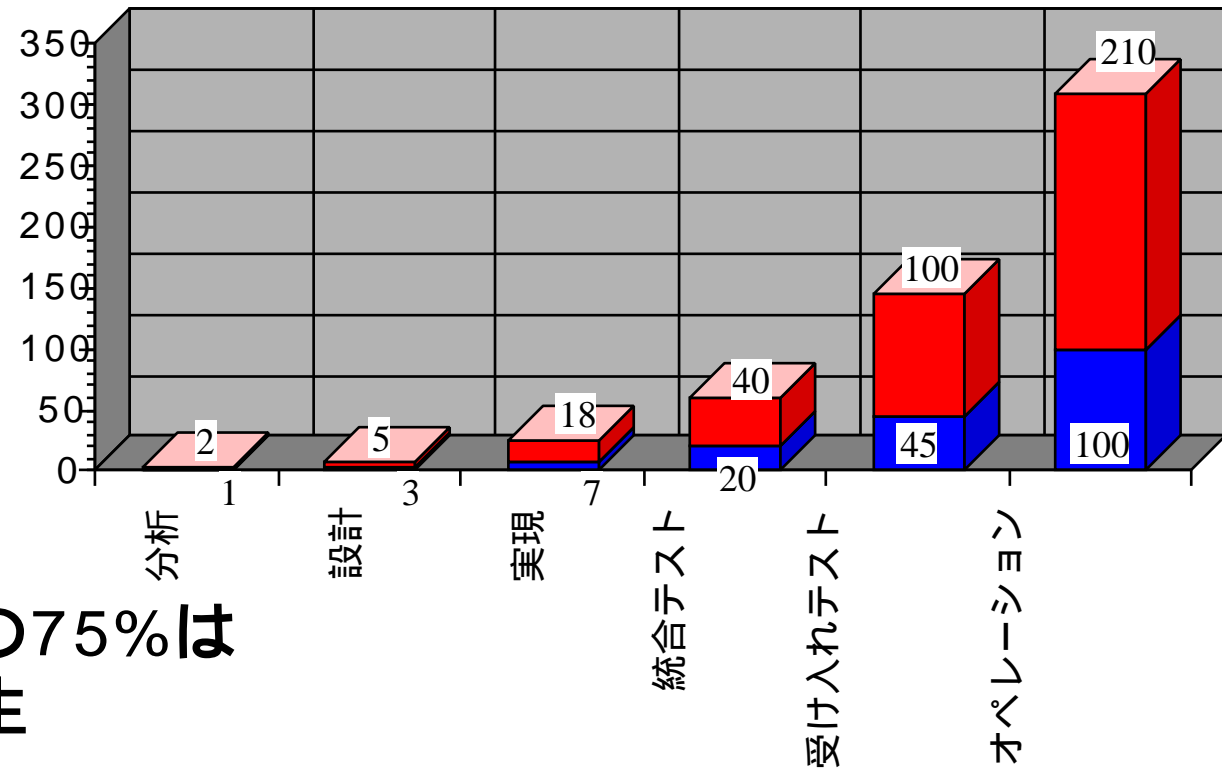
- オブジェクト指向で解決できるのは技術的問題点だけ
- 失敗の原因は、単なる技術的問題ではない
 - 失敗したプロジェクトの調査(DeMarco/Lister)
 - 「政治的要因」による失敗と言われていた
 - 実際には...
 - 意思疎通の問題
 - 要員の問題
 - 管理者や顧客への幻滅感
 - 意欲の欠如
 - 高い退職率

オブジェクト指向技術をなかなか覚えられないのだが

- 英語の勉強にどれくらいの時間を掛けているか？
 - 中学3年で280時間 = 12日間
- OOの勉強にどれくらいの時間を掛けているか？



コンポーネントウェアとか良いツールがあるのだから、分析や設計は大概にしてプログラムを作っては駄目か？



- 重大なエラーの75%は上流工程で発生

■ 最悪の場合

■ 最善の場合

オブジェクト指向を使ってプロジェクトの遅れを取り戻せるか？

• 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
- オブジェクト指向言語のコーディングに大量に人を投入すれば何とかなるのでは？
 - 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

• それではどうすればよいのか？

- 出来る技術者を少数投入する
 - 4倍から1000倍の差
- オフィス環境を良くする
 - ピープルウェアの節参照
- 開発環境を良くする
 - SmalltalkやCLOS系の環境が圧倒的に優れている

プロダクト

- うまくいったから製品化したい
 - 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
- 出来たプログラムが遅くて使い物にならない
 - 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
 - 焦点を絞って効率化すれば10倍くらいはすぐ速くなる
 - ツールの20%が、時間の80%を使う

ピープルウェアに関する定石

- どのようなチーム構成とすればよいか？
- プロジェクトがうまくいかないのだが？

どのようなチーム構成とすればよいか？

• 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・ライブラリアン・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済む
- 責任は明白になる

• 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）
- 専門家がいなかったため、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

プロジェクトがうまくい かないのだが？

- 開発プロジェクトの人的側面
- 失敗する管理方法
- ピープルウェア的管理(ドゥマルコ)
- チームの触媒
- 規則は破るためにある(ドゥマルコ)
- 品質第一(ドゥマルコ)
- 目標設定者による生産性の違い(ドゥマルコ)
- オフィス環境と生産性(ドゥマルコ)

開発プロジェクトの人的側面

- **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
 - OO開発に向けた人々

失敗する管理方法

- **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

- **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**

- OOには、質の高い仕事が不可欠

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
 - OOは間違いの修正の連続
- 開発標準や作業規定の無理強いは悪である
 - OOは直線的な開発ではないため、開発標準や作業規定で全部をカバーできない
 - 便利なツールを使った結果が標準になるように工夫する

• 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
 - もっとも安いクラスライブラリーの宝庫
 - ソフトウェア代が異常に高い
 - OOは物まねの連続

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
 - OOは複雑なソフトウェアを比較的簡単に作れるが、反面複雑な欠陥も簡単に作り込める

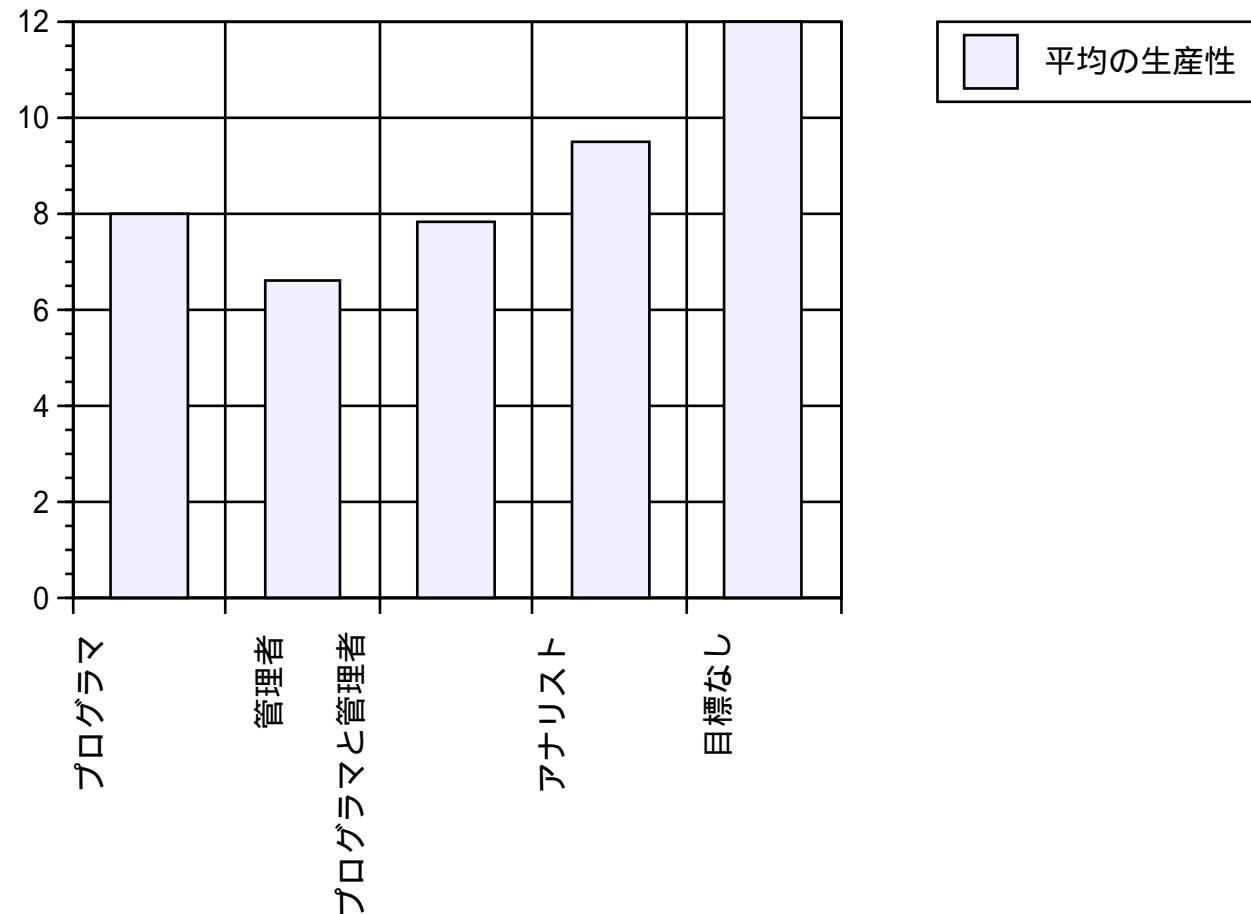
規則は破るためにある(ドゥマルコ)

- 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
 - すでに数億円の受注に結びついている
 - Mac OBJ
 - Smalltalkアルゴリズムライブラリー
 - Smalltalk + 形式的仕様記述言語
- 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

品質第一(ドゥマルコ)

- 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
 - 仕様記述言語と段階的詳細化による証明

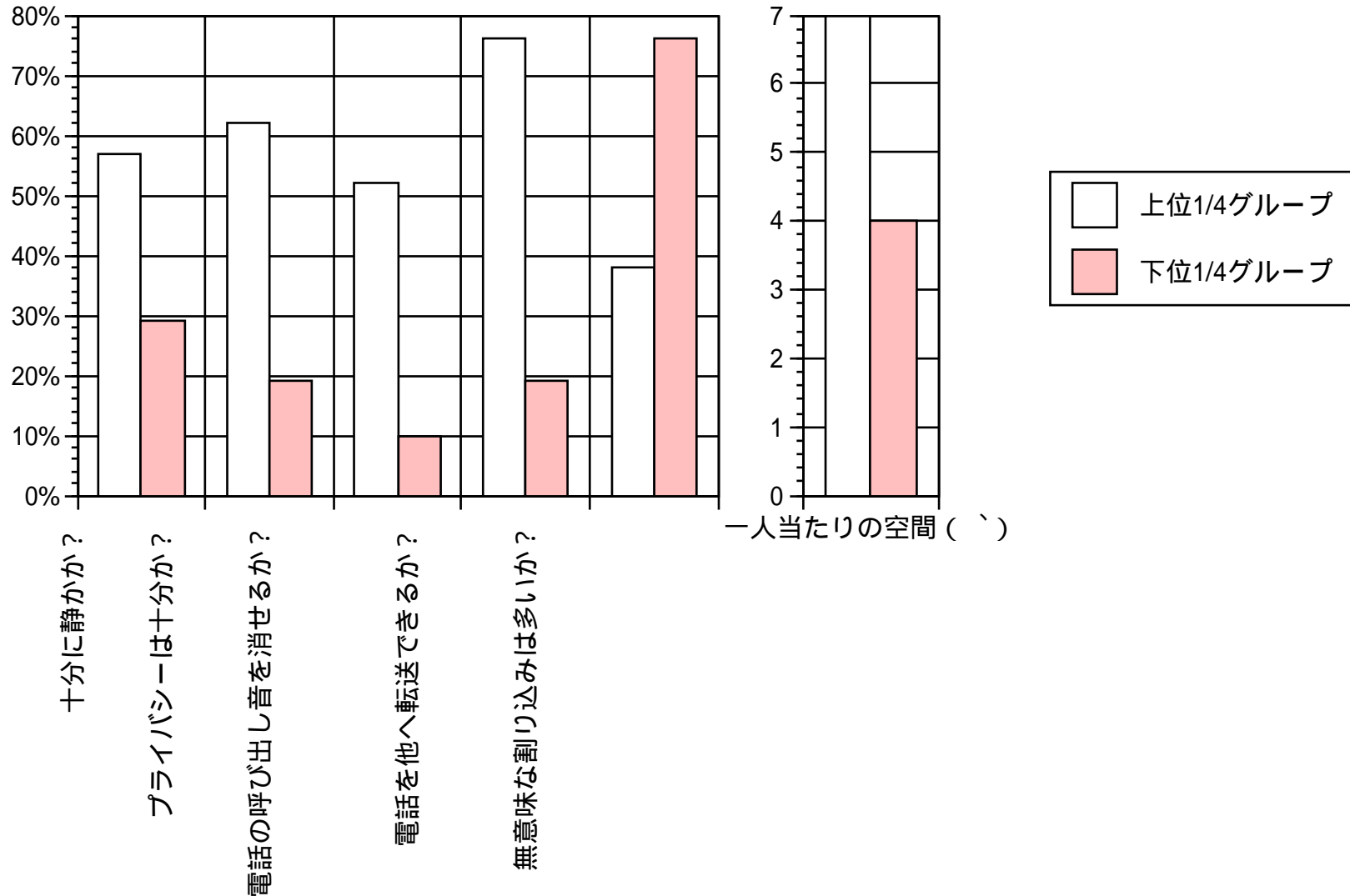
目標設定者による生産性の違い(ドゥマルコ)



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
- 悪い環境の兆候
- オフィス環境の改善

オフィス環境の重要性



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秒以内を取れ**
 - 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はなく、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9mm²以上
 - 作業机2.7mm²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...

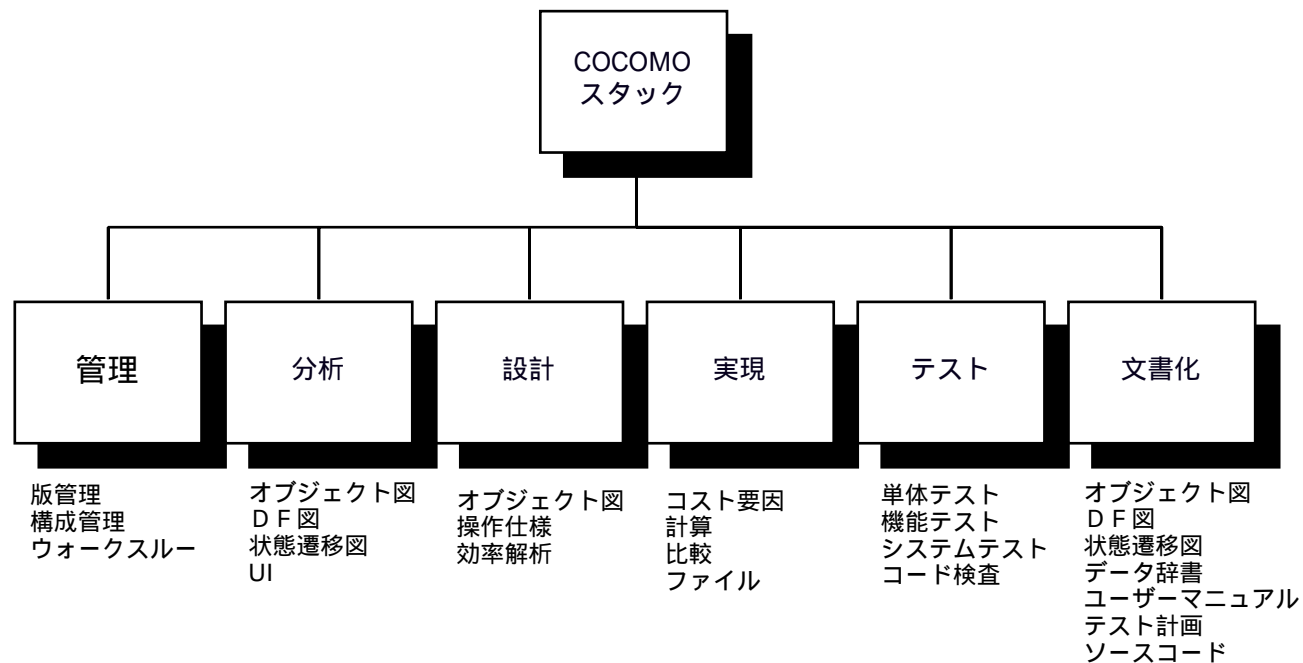
- チームメンバーが自由にレイアウトする
- 喫茶店を会議室にする
- 喫茶店で本を読む
- 会議室を作業場所にする
- 自宅を作業場所にする

プロジェクト管理に 関する定石

- 見積もりはどのようにするのですか？
 - ワーク・ブレイクダウン・ストラクチャ
 - ファンクションポイント
 - COCOMO
 - 責任マトリクス
 - スケジュール管理

ワーク・ブレイクダウン・ストラクチャ

- プロダクトWBS
- 活動WBS



ファンクションポイント

- FPワークシート
- PCワークシート
- プログラミング言語との対応

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
照会	10	3	30	2	4	8	2	6	12	50
									To	216

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	2.07		

プログラミング言語との対応

言語	コード行数/1FP
アセンブラ	320
C	150
COBOL	106
FORTRAN	106
Pascal	91
PL/I	80
Ada	71
Prolog	64
APL	32
Smalltalk	21
スプレッドシート	6

COCOMO

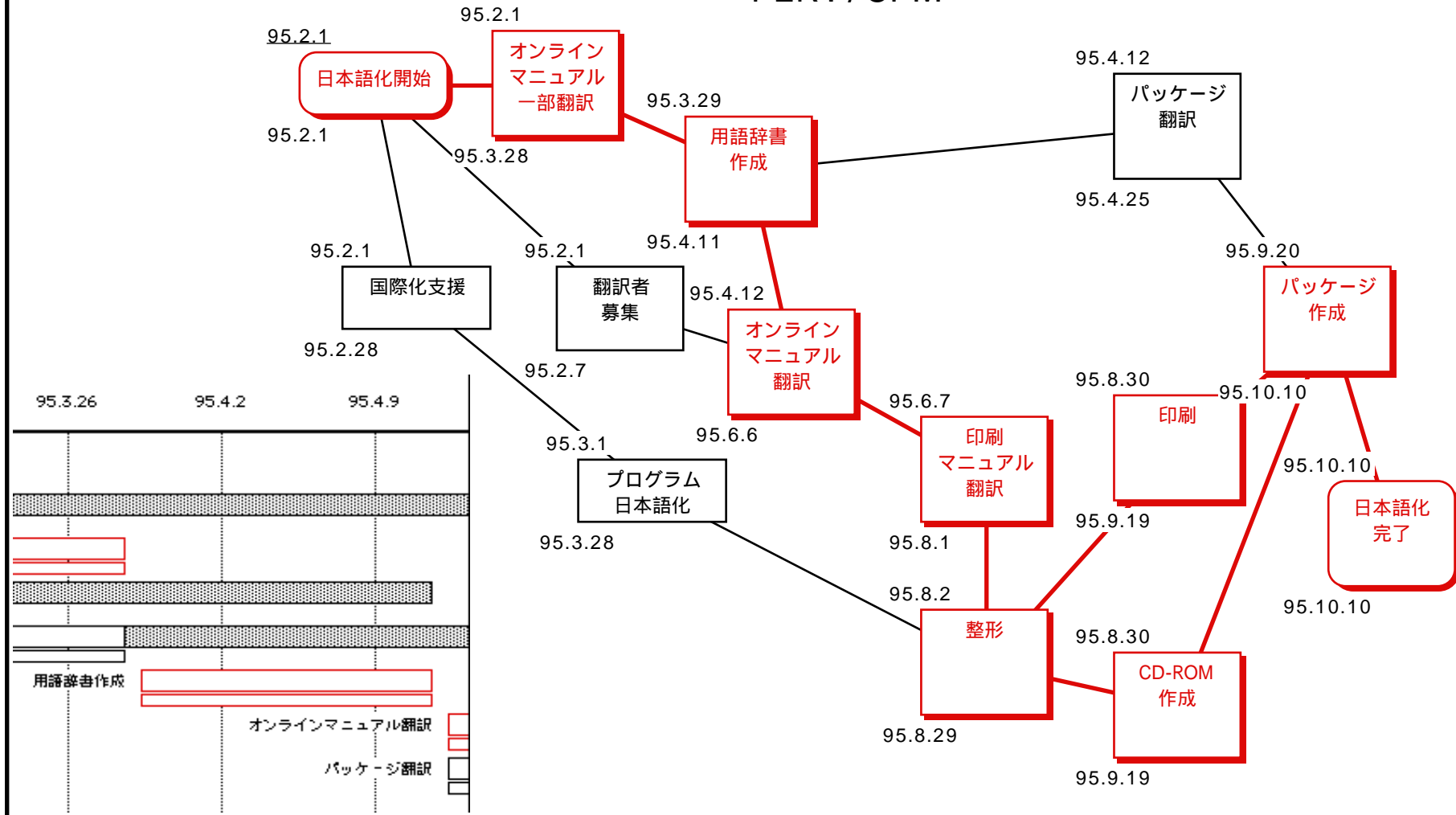
COCOMO									
Project名 <input type="text" value="Micro Comm."/>		Intermediate COCOMOプロジェクトレベルコスト見積							
DSI 行数 <input type="text" value="10000"/>	RELY	<input type="text" value="普通"/>	TIME	<input type="text" value="高"/>	ACAP	<input type="text" value="高"/>	MODP	<input type="text" value="高"/>	<input type="radio"/> 非常に低 <input checked="" type="radio"/> 普通 <input type="radio"/> 高 <input type="radio"/> 非常に高
	DATA	<input type="text" value="低"/>	STOR	<input type="text" value="高"/>	AEXP	<input type="text" value="普通"/>	TOOL	<input type="text" value="低"/>	
	CPLX	<input type="text" value="非常に高"/>	VIRT	<input type="text" value="普通"/>	PCAP	<input type="text" value="高"/>	SCED	<input type="text" value="普通"/>	
プロジェクトのモード <input type="text" value="Embedded"/>		TURN	<input type="text" value="普通"/>	VEXP	<input type="text" value="低"/>	乗数 <input type="text" value="1.170911"/>			
1ヶ月の作業時間 <input type="text" value="152"/>		コスト <input type="text" value="100"/>		生産性 <input type="text" value="192.3"/>		<input type="radio"/> 低 <input checked="" type="radio"/> 普通 <input type="radio"/> 高 <input type="radio"/> 非常に高			
工程	MM 人月	<input type="text" value="52"/>	TDEV 期間	<input type="text" value="8.9"/>	FSP 平均投入人数 <input type="text" value="5.8"/>				
計画と要求		<input type="text" value="4.2"/>		<input type="text" value="2.5"/>	開発費用 <input type="text" value="5200"/>				
設計		<input type="text" value="9.4"/>		<input type="text" value="2.9"/>					
プログラミング		<input type="text" value="29.5"/>		<input type="text" value="3.9"/>					
詳細設計		<input type="text" value="14"/>		<input type="text" value="2.2"/>					
作成・単体テスト		<input type="text" value="15.5"/>							
集積とテスト		<input type="text" value="13.1"/>							
<div style="display: flex; justify-content: space-between; align-items: center;"> COCOMO 削除 コピー 追加 計算 検索 普通 クリアー 操作方法 </div>									

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				
D F 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				

スケジュール管理

• PERT/CPM



オブジェクト指向技術者の育成はどうするか？

- オブジェクト指向プログラミング教育
- オブジェクト指向設計教育
- オブジェクト指向分析教育

オブジェクト指向プログラミング教育

- 初級

- ソフトウェア科学の基礎
 - 離散数学
 - 集合・述語論理
 - データ構造とアルゴリズム初級3日
 - テスト技法1日
- コンポーネントウェアによるプログラミング
 - 入門セミナー 2日
 - Smalltalkのサブセット教育 2日
 - OO設計の初歩も含む

- 中級

- Smalltalk初級
 - 入門セミナー 4日
- OOD2日
- ソフトウェア科学の基礎
 - 離散数学
 - 集合・述語論理・代数
 - データ構造とアルゴリズム中級3日
 - コンパイラー入門3日

オブジェクト指向設計教育

- ソフトウェア工学概論
 - 仕様記述言語
 - Z, VDMなど
 - プロジェクト管理
- ソフトウェア科学中級
 - 離散数学
 - 集合・述語論理・代数
 - データ構造とアルゴリズム上級
 - コンパイラー中級
- OOA / OOD3日
- Smalltalk中級

オブジェクト指向分析教育

- ソフトウェア科学上級
 - 離散数学
 - 集合・述語論理・代数
- 形式的仕様記述方法論
 - RAISE, VDMの方法論
 - 正当性評価
- OOA / OOD 演習 3日