オブジェクト指向設計

- 仕様記述は
  どうするのか？
- 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？
- データ構造とアルゴリズ
- 実行効率はどのあたりで考えるのか？
- 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？

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

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

データ構造とアルゴリズ

- 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？

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の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 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
0664408533370012842864126543944950507739545600000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000
```

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

実行効率はど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

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係する操作は全体の２～３％
      - 残りの９７～８％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