SEAMAIL

Newsletter from Software Engineers Association

Vol. 13, Number 11 March, 2003

目　次

ソフトウェア技術者協会

Software Engineers Asociation

ソフトウェア技術者協会 (SEA) は，ソフトウェアハウス，コンピュータメーカ，計算センタ，エンドユーザ，大学，研究所など，それぞれ異なった環境に置かれているソフトウェア技術者または研究者が，そうした社会組織の壁を越えて，各自の経験や技術を自由に交流しあうための「場」として，1985年 12月に設立されました．

その主な活動は，機関誌 SEAMAIL の発行，支部および研究分科会の運営，セミナー／ワークショップ／シンポジウムなどのイベントの開催，および内外の関係諸団体との交流です．発足当初約 200人にすぎなかった会員数もその後増加し，現在，北は北海道から南は沖縄まで，500 余名を越えるメンバーを擁するにいたりました．法人賛助会員も 24社を数えます．支部は，東京以外に，関西，横浜，名古屋，九州，広島，東北の各地区で設立されており，その他の地域でも設立準備をしています．分科会は，東京，関西，名古屋で，それぞれいくつかが活動しており，その他の支部でも，月例会やフォーラムが定期的に開催されています．

「現在のソフトウェア界における最大の課題は，技術移転の促進である」といわれています．これまでわが国には，そのための適切な社会的メカニズムが欠けていたように思われます．SEA は，そうした欠落を補うべく，これからますます活発な活動を展開して行きたいと考えています．いままで日本にはなかったこの新しいプロフェッショナル・ソサイエティの発展のために，ぜひとも，あなたのお力を貸してください．

ＳＥＡＭＡＩＬ Ｖｏｌ．１３，Ｎｏ．１１ ２００３年3月２５日発行 編集人 岸田孝一
発行人 ソフトウェア技術者協会（ＳＥＡ）
〒１６０-０００４ 東京都新宿区四谷３－１２ 丸正ビル５Ｆ
T: 03-3356-1077 F: 03-3356-1072 sea@sea.or.jp
印刷所 有限会社 錦正社 〒１３０-００１３ 東京都墨田区錦糸町４－３－１４
定価 １、０００円 （禁無断転載）

編集部から

☆

またしても船便状態に陥りそうになったので，今回は幹事会メンバーの方々にお願いして原稿を書いていただきました．

☆☆

教育論，技術論，書評をとりまぜて，なかなか読みごたえのある内容になったのではないでしょうか．

☆☆☆

次号もさらに何人かの幹事の方々からの寄稿が予定されているので，４月末までにはなんとか出せるのではと考えています．

☆☆☆☆

一般会員の方々もふるって原稿をお寄せください．

☆☆☆☆☆

教養としての情報学 序章

玉井哲雄

1 はじめに

東京大学では 1993 年から「情報処理」という授業を，文系理系を問わずすべての学部 1 年生の必修科目とした．その科目の内容は世間から「コンピュータ・リテラシー」を教えるものと見られている．それが科目の狙いの一部であることは確かだが，情報という対象を扱うのに固有な基本的原理や方法に目を向けさせるという大きな目的もある．大体，コンピュータ・リテラシーという言葉が何を指すもののかはっきりせず，あまり好きではないということは別に書いた [2].

さて，2003 年 4 月から高校の普通科目として「情報」が必修になる．その情報の教育を受けた高校生が，2006 年 4 月から大学に入ってくる．それに伴って「情報処理」の内容を変えなければならない．というより大学入学者がある程度は「コンピュータ・リテラシー」をすでに身に着けているものと仮定すれば，大学教養課程における情報教育の内容を一新できる．現在それに向けて，学内グループで議論を進めているところである．

文系理系と共通で必修の授業を行うことを前提として，まず教科書を作ってみたいと考えた．そこで次のようなメモを書いてみた．これは筆者のまったく個人的な構想である．

タイトルは仮に「教養としての情報学」とする．もっとよいタイトルを考えたいが，今のところ思いつかないので取りあえずこうしておく．

1. 趣旨

- リテラシー教育ではない.
- 文理共通に使える.
- 基本的な考え方を伝えたい.
- 最先端技術 (はやりの技術) も積極的に取り上げるが，それを通して基本概念，技術を教える．たとえば XML を取り上げるとすれば，XML についての解説を書くのではなく，これを題材として，データの蓄積と検索の問題，文書の交換，表示，標準化の問題などを一般的に論じる．すなわち，きちんとした枠組みの中に，流行の技術を位置づける.
- 情報の面白さと奥の深さを伝えたい.
- 「体系的」な本ではなく，面白く読めるものとする．たとえば例題を工夫して，ありきたりでない豊かなものとする.
- 座学的部分と実習部分とを組み合わせる.
- コンピュータを単なる道具と見るな，というメッセージを伝える.
- ページ数を多くする.
- 先行する駒場の参考例:
  - 「知の技法」シリーズ → 一般にも売れる

- Universe of English → 「花子の米国旅行」でもなく Shakespeare でもないことを狙いとしている.
  本屋に平積みされているマニュアル的な解説本でもなく Knuth でもない，というように読み替えればよいか.
- 野矢茂樹: 「論理トレーニング」 → 面白い演習

2. 読者対象: 駒場の 1・2 年生であるが，さらに一般に
   - 「情報」について考えてみたいが，マニュアルやその類いの機械的な文章には辟易している人.
   - 論理的な指向は好きだが，あまりの形式化は敬遠したい人.
3. 内容: 以下は単なる思いつきの羅列

(a) 「伝える」
communicate, 通信，伝達
メディア

(b) 「表現する」
書く，描く，記述，図示
言語，記号，符号，図式
修辞，文章作法

(c) 「考える」
推論，類推，演繹，帰納
論理

(d) 「決める」
選ぶ，判断，意思決定，評価
価値

(e) 「計算する」
数える，演算
計算可能性，計算の複雑さ

(f) 「測る」
計測，計量
統計

(g) 「探す」
探索

(h) 「解く」
求解
方程式，制約，束縛

(i) 「発見する」
パターン，法則

(j) 「理解する」
認識，認知
文章，意味，音声，図形

(k) 「変換する」
換える，翻訳，置き換え
コンパイラ

(l) 「編集する」
編む
テキスト，図，Web ページ，本

(m) 「覚える」
思い出す，記憶，蓄積
DB，記憶装置

(n) 「設計する」
design

(o) 「学習する」
知識獲得，知識適用

思いつくまま 15 の項目を並べてみたが，レベルがそろっているとはいえず，重複する部分もある．何より 15 という数は多すぎる．われわれのグループでは，とりあえず「表現」,「伝達」,「計算」,「検索」,「システム」,「社会」という 6 つの分野に絞って検討しようということになっている．

しかし，このままでは先に進まないので，部分的に草稿を書き始めてみることにした．内容的にもまだごく一部であり，練れてもいないが，SEA の会員諸兄姉にご批判をいただければありがたいと思い，未完成な原稿をお目にかけることにした次第である．以下は上の項目でいえば,「表現」に当たるところの一部である．

2 情報の表現

表現とは

「表現」という言葉は，感情表現，芸術表現などのように人が内面に持つ心理的・精神的なものを外面的な対象として表す行為を指す場合と，文字，音声，図などの何らかの記号によって外部的に表されたものを指す場合とがある．ここでは情報の表現を，現象，事象，事実，規則・法則，などを記号として表すこと，あるいは表したもの，としてとらえる．感情表現や芸術表現の手段は，言語という記号的なものだけでなく，表情，身振り，動作，音楽，絵画，彫刻，など多様である．ここでは記号的な表現に絞ることから，感情表現や芸術表現はとりあえずは除いて考える．ただ情報の表現と感情表現や芸術表現は関連があるので，今後それらに触れることもあるかもしれない．

手段を記号に限ると，表現の定義は情報そのものの定義とほぼ一体化してしまう．情報の定義としてたとえば吉田民人の定義を挙げてみよう．

> 「情報とは，最広義には物質 - エネルギーの時間的・空間的および質的・量的なパタン．最狭義には個体的・集合的な人間主体の意思決定を規定する，伝達された単用的・認知的な外シンボル集合<有意味な記号集合>.」[1]

かなり難解だが，最広義の方は宇宙線のパターンや DNA といった自然界の「情報」も含めて考え，最狭義の方は人と人との間で伝達 (communicate) される情報を対象としている．後者は要するに有意味な記号集合といっているわけで，いいかえれば情報は表現されて初めて情報になるのだといえよう．したがって表現の形や方法の考察は，情報を考える上でもっとも基本となるものである．

記号と符号

記号としてまず思い浮かぶのは文字である．文字の中でも数字 (アラビア数字) は今や世界共通に使われている．そこでまず電話番号や郵便番号などの数字による情報表現を考えてみよう．

その前に，記号としての文字の性質に注意しておく．漢字のような表意文字には当然意味が結びつけられているが，表音文字のアルファベットや「かな」にも，音という固有のものが結びつけられている．あるいは「?」のような文字にも「はてな」とか「疑問」という概念が文化的，習慣的に結びつけられている．この結びつけは文化によるので，たとえば日本では○は「よい」，×は「悪い」，△はその間という結びつけが小学校以来一般化しているが，欧文の文脈では必ずしもそのように解釈されないようである．なお，?や○や×や△を記号と呼んで，記号を文字の一部とする言い方もあるが，ここでは逆に記号の一部が文字であると考える．

数字にも数という概念が結びついている．しかし，電話番号や郵便番号で使われている数字は，数として扱われているわけではない．このように固有の意味と結びつけずに使われる記号を，ここでは符号 (code) と呼ぶことにする．モールス信号はトンとツーという長さで区別される 2 つの信号を用いるが，このトンとツーは固有の意味を持たないという点で，典型的な符号といえる．郵便番号や電話番号で使われる数字は，別の記号に置き換えてもかまわない．1 の替わりに?を使い，2 の替わり!を使うというように取り決めても，それほど大きな問題はない．その意味では数字を符号として用いているのである．もちろん，もともと記号すべてにそのような性質があり，?が疑問を表したり 1 が数の「1」を表したりすることに必然性はない．記号にはそのような意味で本来，中立性，代替可能性があり，特定の意味と結びつけられるのは社会の慣習によるのである．

符号化

電話番号は符号としての数字を，たとえば 10 桁並べたものとして表現される．このように固定した符号の集合（今の場合は 0 から 9 までの数字の集合）の要素の並び (符号列) で情報を表現することは，基本的な方法である．符号集合の大きさを n，符号列の長さを l とすると，n^l 個の対象を表現できる．このように情報の対象を一定の符号の組み合わせと結びつけることを，符号化 (coding) という．一般に，言葉は音素あるいは文字の並びで作られるという意味では同じである．ただ，言葉は単語というレベルでとらえても，文というレベルでとらえても，その記号列の長さが不定であるところが電話番号とは異なる．

識別

ここで，表現同士が識別されることと，識別された表現が決まった対象を指すという対応関係を持つことが重要である．このような対応関係を表現から対象への写像という．電話番号で言えば，10 桁の数字列から加入電話への写像が定義されているわけである．

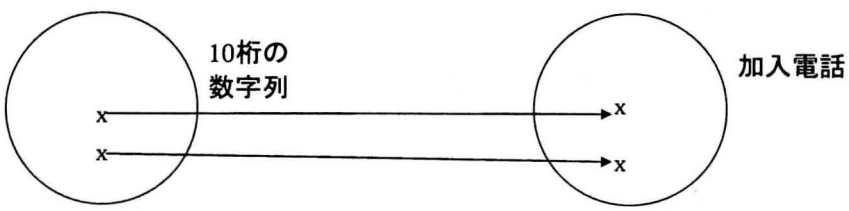


図 1: 表現から対象への写像

写像の基本的な性質として，写像元（定義域と呼ばれることもある）の 1 つの要素（今の場合電話番号の

10 桁の数字列）は，ただ 1 つの写像先（値域と呼ばれることもある）の要素（今の場合は加入電話）に対応している，ということが挙げられる．ソシュール流の言語学で言えば，写像元はシニフィアン（signifiant, 能記とか記号表現と和訳されることもある）に対応し，写像先はシニフィエ（signifié, 所記とか記号内容と和訳されることもある）に対応する．しかし，この両者を二面的に合わせもつものがシーニュ（signe, つまり記号）であるというのだから，われわれのこれまでの記号という言葉の使い方と異なり，ちょっとややこしい．

しかし，ソシュールのような小難しい言語学をもちださず，日常的な言葉の感覚で議論すると，同じ言葉が 2 つ以上の意味を指すことはよくある．これを同音異義語 (homonym) という．一方，違う言葉が同じ意味を指す場合もよくらい，これを同義語 (synonym) という[1]．

写像については同音異義語に対応するものは最初から排除されているが，同義語に相当するものは一般には許される．そのような事例をもたない写像，すなわち写像元の要素が異なれば写像先の要素も異なるものは，とくに単射と呼ばれる．電話番号のような人為的に定められた符号化の体系では，最初から同音異義語や同義語が存在しないように作られる．つまり，記号表現と記号内容の関係は単射な写像となる．

電話番号や郵便番号が数としての性質を持たないことは，それらの間でたとえば

- 四則演算
- 大小関係

が意味を持たないことからも納得できよう．電話番号同士を足したり，2 つの電話番号を比べてどちらが大きいといっても意味がない．

しかし，表現が識別できるためには，2 つの電話番号が同一か否かという判定は意味を持たなければならない．つまり 2 つの番号 a と b に対し，$a = b$ あるいは $a \neq b$ という論理式は意味を持つ．これが「識別できる」ということの基礎である．

少し面倒くさいことを言えば，符号の並びで表現された情報を識別することは，その要素である符号が識別できることに依っている．数字の場合は 10 個の文字の間で $a = b$（または $a \neq b$）の関係が判定できることを前提とする．これは自明なことではない．実際，郵便番号は手書きの数字を機械で読み取って，この判定をしているのである．たとえば  と は同じと判定するが， とは異なると判定するわけである．

その上で，2 つの記号列 $a_1, a_2, \ldots, a_n$ と $b_1, b_2, \ldots, b_m$ が等しいとは，両者の長さが等しく $(n = m)$，対応するそれぞれの記号が等しい場合 $(a_i = b_i, i = 1, \ldots, n)$ で，その場合に限る．

「近さ」の構造

電話番号や郵便番号に，等しいか等しくないかということ以上の構造はないだろうか．すぐ気づくと思うが，やはり構造はある．それは「近さ」という関係である．電話番号が近いもの同士は，それが指す加入電話の設置場所も互いに近いだろう．

ところで電話番号が近いとは，より正確にはどういう意味だろうか．10 桁の番号の内，9 桁までが一致していたら 8 桁一致しているものより近いだろうか．必ずしもそうではない．桁がより左にある方が近さの判定により大きな影響を及ぼす．そこで 2 つの電話番号の近さの基準として，両者の差の絶対値を取って，それが小さければ近い，大きければ遠いと判断することは考えられる．しかし，電話番号は数ではなく，その間に差とか大小関係は意味がないと言ったのではなかったか．

確かに電話番号の表現に数字の並びを用いたために，10 桁の 10 進数と同じ形となったのは偶然ではあった．0〜9 の代わりに a〜j を使ってもイ〜ヌを使ってもよかった．ただ，個々の記号の間にも何らかの近さ

[1] 同音異義語と対称に書くなら異音同義語そ記すべきだろう．ただし，同音とか異音とかいっても，音のみを念頭においているわけではなく，記号表現として同じか異なるかを問題としている．その意味ではホモニム，シノニムという言い方の方が，対称性もありよいかもしれない．

の性質が存在している必要はある．たとえばａとｄの間はａとｂの間よりも遠い，というような性質である．これはアルファベットやイロハの順序で対応させることができよう．ただ，ａとｄの間がａとｂとの間より3倍遠いかどうかは判らない．同様にある桁の1単位の違いは，その右の桁の1単位の違いより大きな違いであることは確かであるとしても，その差が10倍かというと，そうとはいえないだろう．

差や絶対値が定義されているのは数（自然数）の世界である．だから，電話番号の差の絶対値が近さを表すというとき，厳密に言えば次のような操作を行っていることになる．

1. 2つの10桁の数字の並び a と b を自然数の世界に写像する．写像を g と書き，a と b の g による写像先をそれぞれ $g(a), g(b)$ と書く．
2. 写像先の数の間の差の絶対値 $|g(a) - b(b)|$ で a と b との距離を定義する．
3. 10桁の数字の並びから加入電話への写像をｆと書く．$f(a)$ と $f(b)$ の設置されている場所の近さと a と b の距離が対応しているものとする．

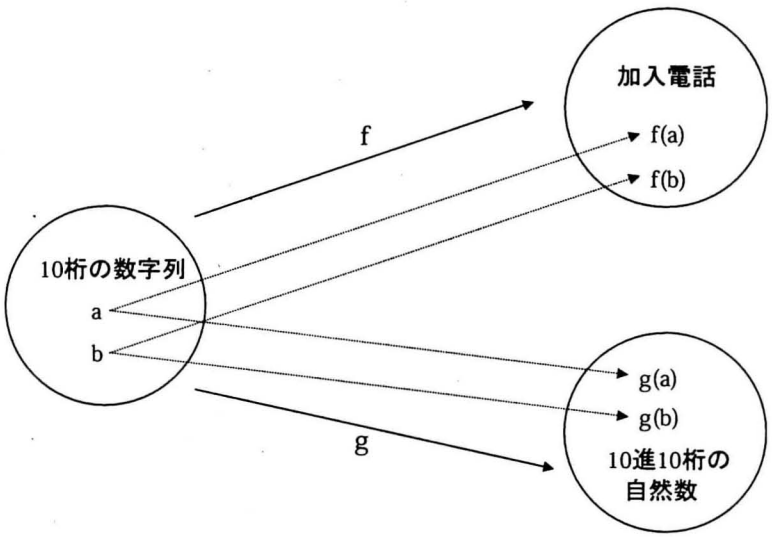


図 2: 2つの写像

階層構造

ここでちょっと待てという声が，聞こえてきそうである．電話番号を10桁の数字の並びとしているが，実は局番と局内番号に分かれているのではないか．郵便番号も，最初の3桁と後の4桁は扱いが違うのではないか．

確かに電話番号の内の局番は加入電話が設置されている地域という地理的な情報を反映しているかもしれないが，下4桁にはそのような対応はなさそうである．上位2桁は，01が北海道から秋田・岩手まで，02が東北の山形・宮城から西は長野まで，03は東京都区内とその周辺，04は都内の残りと，千葉，神奈川，埼玉，という具合に，日本列島の北から南西に地域を分割して決められているらしい．ただ，市外局番一覧表というものを見ると，その桁数は03という2桁のものから，01242のような5桁のものまでまちまちである．そして市外局番の次に，市内局番がある．

とにかく，10桁の数字の並び自身の中に，段階的な構造があるわけである．階層構造とは大きな概念を分割して次のレベルの概念を導き，それをさらに分割して次のレベルの概念を導くという段階的な構造を

いう．電話番号の場合は第 1 段階が市外局番，第 2 段階が市内局番，第 3 段階が局内番号という階層構造を持っている．この構造を明示するために，段階を表す部分列の間にハイフンや括弧を入れるという表記もよく用いられる． 電話番号では，この第 1 段階と第 2 段階の桁数がまちまちだが，これは歴史的な経緯によるものだろう．ただし，市外局番，市内局番というような地理的構造に対応する階層構造があるのは，通常の加入電話の場合である．携帯電話の場合は，仮にこのような階層構造があったとしても地域に対応するものではないだろうし，利用者はその構造を意識していない．間にハイフンなどを入れて区切った表示も，単に記憶の便宜に過ぎないかもしれない．

郵便番号では，第 1 段階が 3 桁，第 2 段階が 4 桁で，その区分は一律である．

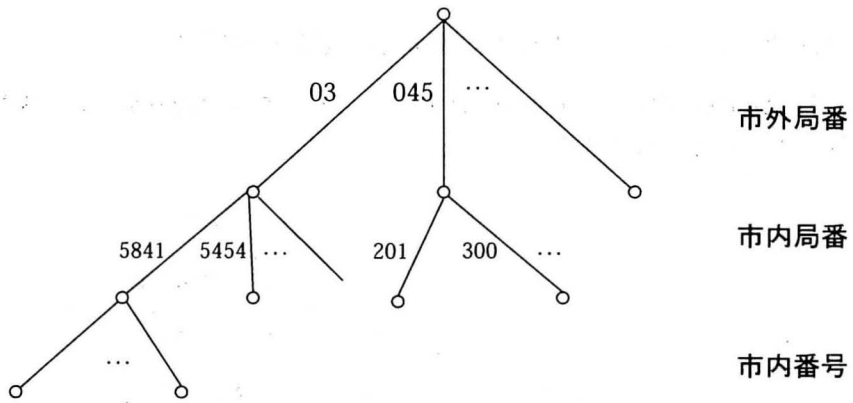


図 3: 電話番号の階層構造

このように部分列が段階を表していて，それを順に配置するという構造は珍しいものではない．たとえば年月日を表すのに，20010911 のような表現をとることがある．最初の 4 桁が年，次の 2 桁が月，次の 2 桁が日を表す．これもやはり左から右に段々小さな単位を置く階層構造になっている．この構造を明示するために，2001/09/11 のように区切りにスラッシュをやピリオドを置く記法もよく用いられる．ただし，ISO の標準は “YYYY-MM-DD” という形式である．つまり年 (4 桁，ただし下 2 桁による省略記法も可)，月 (必ず 2 桁)，日 (必ず 2 桁) と並べ，区切りはハイフンである．このハイフンは省略してよいことになっている．

階層構造の上位レベルを左に，下位レベルを右に置くことは，必然ではない．実際，ヨーロッパでは逆に日月年の順，つまり 11/09/2001 のように書く．面白いことに米国では日本流でもヨーロッパ流でもなく，月日年の順に並べる．つまり 09112001 となる．年月日によってどの流儀で表されているかユニークに決まるものと，そうでなくあいまいなものとがある．とくに米国流とヨーロッパ流の混同が問題だが，たとえば 1213 なら，これは米国式表記が使われていて 12 月 13 日を指していることは明らかである (年の 4 桁は共通なので省略)．しかし 1208 では米国式で 12 月 8 日なのか，ヨーロッパ式で 8 月 12 日なのかあいまいである．

なお，年を下 2 桁で表す表記もよく用いられる．これが 2000 年問題を引き起こしたことは，記憶に新しい．

日本流が大きな構造から小さな構造の順に表現するのに対し，欧米流では異なる順に並べる他の例に，住所表記がある．日本では，都道府県，市町村，丁目，番，号のように並べる．欧米だと，番地，通り，市，州などのように並べる．日本人は構造的な思考が弱いという指摘がよくなされるが，このように大きなものから小さなものへというトップダウンの記述は，階層構造を自然に反映してある意味では合理的だといえよう．

問題１　年月日の米国流とヨーロッパ流の表記で，どちらの表記によるものかがユニークに定まる月日とそうでないものとをすべて示せ．

問題２　図書分類表を調べ，それがどのような階層構造を表現しているか明らかにせよ．

数としての構造

これまで扱った数字列の表現では，識別と近さという構造を考えた．つまり等号と比較という，ごく基本的な演算が定義された集合としての記号列を対象とした．ここではもう少し複雑な数としての構造を考えよう．

すでに挙げた例の中で，年月日は部分的に数としての性質をもっていた．そもそも，われわれが慣れ親しんでいる数の 10 進表記が，情報の表現法の 1 種に他ならない．考えて見れば，数字の並びの位置で 10 進の桁を表すというのは，実に偉大な発明であった．つまり

$$a_1 a_2 \cdots a_n$$

という表現 (a_i は 0〜9) は数として

$$a_1 \times 10^{n-1} + a_2 \times 10^{n-2} + \cdots + a_n$$

を表しているのである．この表現の有用性はたとえば 568 と五百六十八や DLXVIII(ローマ数字) とを比べてみれば，明らかだろう．

年月日の表現の 20010911 は全体として 10 進数を表していないが，最初の 4 桁，次の 2 桁，最後の 2 桁は，それぞれ 10 進数を表している．ただし，月を表す 2 桁は 1 以上 12 以下，日を表す 2 桁は 1 以上 31 以下という制約がある．厳密に言えば，日の上限は月および年の値によって 31, 30, 29, 28 のいずれかに限定される．年の 4 桁は一般には 1 以上 9999 以下だが，日常的には 2000 ± 50 ぐらいの範囲で用いる．

これを月の 2 桁は 12 進数を表していて，12 を越えると年の最下位に桁上がり起こるとみてもよい．同様に日の 2 桁は 31 進数（または 30,29,28 進数）で，上限を超えると月の桁に桁上がりが起こる．

時分秒の表示も同じ構造をしている．たとえば 10 時 48 分 25 秒を 114825 と表現する．これは通常のデジタル時計の表示であり，区切り記号を入れるとすればコロン (:) である．面白いことに，この順序は米国でもヨーロッパでも変わらない．またこれは，コロンを省略できることも含めて，ISO の標準でもある．

最初の 2 桁は 24 進，次の 2 桁は 60 進，最後の 2 桁も 60 進である．ここでは年月日表示の日の桁のようなややこしさはない．この 6 桁をひとまとまりの数の表示とみなすこともできる．

まず左から 6 桁目と 4 桁目は 10 進で桁上がりが起こり，5 桁目と 3 桁目は 6 進で桁上がりが起こる．1 桁目と 2 桁目は 24 進数を 10 進表示しているので，2 桁目が 10 進で，1 桁目が 3 進というような言い方は正確ではない．60 が 10 の倍数なのに対し，24 は 10 の倍数ではないからである．しいて言えば，1 桁目は 3 進でよいが，2 桁目は 1 桁目が 0,1 の時 10 進，2 の時 5 進となる．

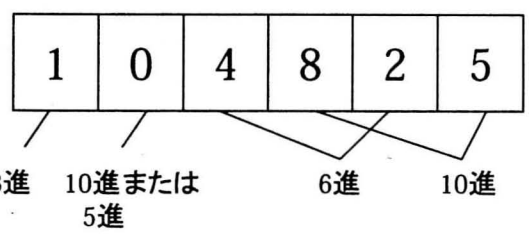


図 4: 時分秒の各桁の表現

このような桁上がり演算のもとで，この時分秒の値に対し加減算が定義できる．たとえば

$$104825 - 085336 = 015449$$

これは 10 時 48 分 25 秒から 8 時 53 分 36 秒を引くと，1 時間 54 分 49 秒になるという演算を表している．左から 3 桁目や 5 桁目の引き算は 6 進のため，上位桁から 6 を借りてきている．

ところで今の説明は，このような桁ごとの桁上がりや桁下がりの演算方法を定義したので時分秒のデータに加減算が定義できたような言い方だったが，これは正しくない．話は逆で，時分秒単位の時刻という値の間に加減算が定義できるが，それをこのような 0-9 の数字による 6 桁表現をした時に，その表現上で個々の桁ごとの桁上がり・桁下がりを含む演算に帰着できるということである．われわれは 10 進表現に慣れてしまっているので，小学校でやる 10 進の足し算や引き算の方法が，足し算や引き算の定義そのものと思い込んでしまうことがあるが，あれもたまたま 0-9 という数字を用いた 10 進表現を使う時に有効な方法に他ならないのである．

ところで，“104825-085336 = 015449” は「10 時 48 分 25 秒から 8 時 53 分 36 秒を引くと，1 時間 54 分 49 秒になるという演算を表している」と言った．ここで微妙な違いに気づいた読者もあるだろう．左辺の 104825 や 085336 は「時刻」を表しているのに対し，右辺の 015449 は同じ表現をしていながら「時間」という別の種類の情報を指している．時分秒の間で加減算ができると言ったが，正確に言えば，次のような演算が可能なのである．

時刻 − 時刻 = 時間
時刻 ± 時間 = 時刻
時間 ± 時間 = 時間

ここで引き算 $a - b$ に関して，b が a より大きい時はどうなるだろうか．まずうるさいことを言うと，a と b は通常の数でないから，「b が a より大きい」という意味を定義しておかなければならない．a の時間部分 (最初の 2 桁) を a_h, 分部分 (次の 2 桁) を a_m, 秒部分 (最後の 2 桁) を a_s と書くことにする．b についても同様．その時，次の規則で b が a より大きい ($b > a$) と定める．

1. $b_h > a_h$ なら $b > a$
2. $b_h = a_h$ の時，$b_m > a_m$ なら $b > a$
3. $b_h = a_h, b_m = a_m$ の時，$b_s > a_s$ なら $b > a$
4. それ以外はすべて $b > a$ でない．

ここで a_h, a_m, a_s などは数として扱っている．このように個々に大小関係のあるものの並びに関して，全体の大小関係を部分の大小関係を左から順位適用して決めるというやり方は，広く行われている．それに基づいて昇順（小さい方から大きい方へ）あるいは降順 (大きい方から小さい方へ) に並べたものを，辞書的順序という．

さてずいぶん手間をかけて $b > a$ を定義したが，実は a と b の時分秒表現をそのまま 10 進数とみなしてその数の大小関係で $b > a$ を決めたものと，結果的には同じである．つまらないことに精力を使ったと感じるだろうか，それとも厳密な議論をして気持ちがよいと思うだろうか．

回り道をしたが，$b > a$ の場合の $a - b$ の話である．たとえば今日の日の出が 054230, 日の入りが 175545 だとしよう．この差

$$054230 - 175545$$

を負の数を導入して -121315 と表してもよい．これを時刻 x に足せば，12 時間 13 分 15 秒前の時刻になる．つまり日の出は日の入りの 12 時間 13 分 15 秒前だったと読むのである．しかし，同じ時間を負の数は使わ

ずに，114645 と表してもよい．時刻 x の 12 時間 13 分 15 秒前は，x の 11 時間 48 分 45 秒後と等しいのである．そんな馬鹿なと思うかもしれないが，ここで扱っている時刻の範囲では，日の違いは無視される．要するに日（や月や年や曜日）の表示のない時計と同じである．このような計算を法 (modulo) 計算という．

空間の表現

時間の次は空間である．1 直線上の点を座標という数値で表すのも，一種の符号化といえるかもしれない．しかし，直線上の点は実数に対応している．たとえば線分の左端を 0，右端を 1 とし，その線分上の任意の点に対応する実数 $a(0 \leq a \leq 1)$ を考えた時，a の 10 進表記は一般に桁数が無限になる．そこで桁数を有限に固定して，0.31415 のように表現することが一般に行われる．この表現は，直線上の 1 点の表現というより，1×10^{-5} の幅を持った区間を表していると見ることができる．

一般に表現は有限の記号の有限な組合せで表されるから，本質的に有限である．もちろん記号で「∞」とか「無限」とか書いて無限を表すことはできる．数学の世界では，無限を有限の記号で表すためにさらに精緻な工夫を凝らしてきた．しかしここでは，有限桁の 0.31415 のような表現が空間の点 (の集合) の 1 つの符号化になっていることを指摘するのに留めておこう．この直線上の点を座標で表す方式は，デカルトによって，さらに 2 次元や 3 次元空間の点を表すのに拡張された．平面上の点を表すのに，直交する 2 つの直線，x と y を取り，そこへの射影をとることによって，2 つの数の組を用いるものである．図 5 の点 P は (2.6, 1.8) という数の対で表される．

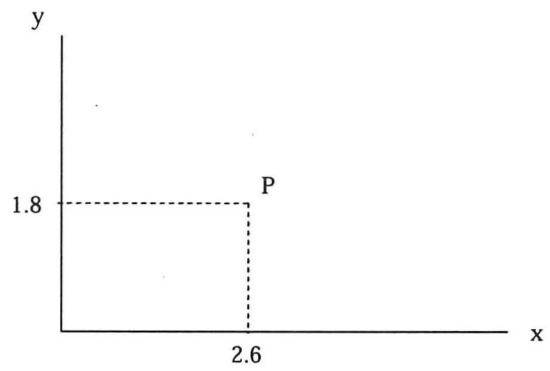


図 5: デカルト座標

このような表現は日常生活でもよく使われる．たとえば囲碁は，19 × 19 の格子の点に石を置いていくゲームである．その格子点は 4 五のように表される．横方向を左から右にアラビア数字で 1 から 19，縦方向を上から下に漢数字で一から十九として表す習慣である．ここで左右とか上下というのは，先手黒番側から盤を見た際の向きをいう．

9 × 9 盤を用いる将棋も同様であるが，面白いことに横方向は囲碁と逆で，右から左に 1 から 9 のアラビア数字をふる．縦は上から下に漢数字を当てることは同じである (図 6 参照)．新聞の将棋欄を見れば，「先手 7 六歩」というような表現で指し手を示しているのが判るだろう．

このように囲碁や将棋やチェスなどの盤面 (board) ゲームは，2 人で交互に指し手を繰り返すことで進行し，その手は盤上のマスないし格子点を表す座標と駒の種類で表現できる．そこで，これを計算機処理に向いたデータ表現にした場合も，データ量はきわめて少なくてすむ．たとえばコンピュータ碁で国際的に使われている SGF(Smart Go Format) では，

```
B[pd];W[cq];B[pq];W[po];B[dd];W[oq];B[or];W[op];B[nq]
```

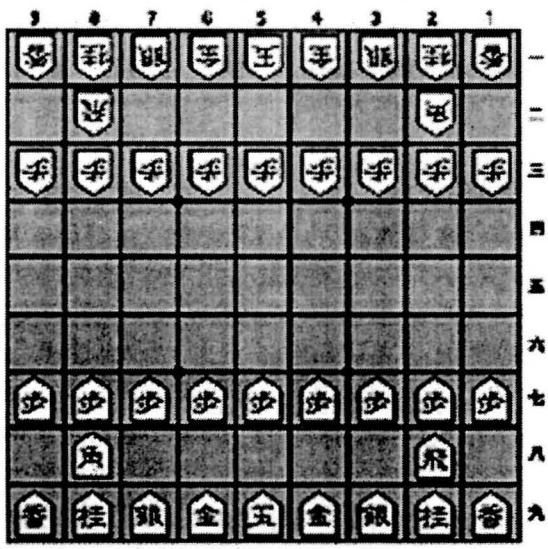


図 6: 将棋の棋譜

のように手の進行を表す．B と W はそれぞれ黒（先手），白（後手）を表し，[pd] などが格子点を表す．横座標も縦座標も a から s までのアルファベットで表し，横方向は左から右，縦方向は上から下に割り当てる．区切りにはセミコロンを用いる．SFG では他に，対局日，対局者名，手についてのコメント，勝敗の結果，など他のデータの記述方法が規定されているが，手の進行という棋譜にとっての本質的な部分は，このようにきわめて簡潔である．

将棋にもさまざまなコンピュータ用の棋譜形式がある．その 1 つ CSA という標準形式では，指し手は “+2726FU” のように表される．ここで＋は先手を意味し，後手なら－である．27 は座標を示し，横軸が 2 縦軸が 7，ただし横軸は右から左に進むことは従来の将棋の棋譜の慣習に従う．したがって 27 は，新聞の棋譜などでは 2 七と表されるものと同じである．次の 26 も同じく 2 六という位置を示す．ここでは駒の動きをあいまいさなく表現しており，先の 27 は指し手前，後の 26 は指し手後の位置を表す．最後の FU は「歩」という駒の種類を表す．だからこの表現は，「先手 2 六歩」に対応する．他に対局者，対局日などのさまざまな情報を記述でき，またある局面の盤面全体を表す表現もある．

このようなコンピュータ向きの表現は簡潔であるが，人間にとっての読みやすさは考慮されていない．しかしこのような形式で記述された棋譜データから，人間にとって判りやすい画像形式の表示を生成することは比較的簡単である．しかもコンピュータ上で表示する場合は，印刷物への表示と異なり，マウスをクリックすることで 1 手ずつ進行させたり，必要なら元に戻したり，任意の手数の局面に飛んだりすることなどが容易に実現できる．

名前

ここでしばらく数字から離れて，一般の文字による表現を考えてみよう．アルファベットやかな漢字による符号化の代表例は，人名，地名，商品名などの名前である．しかし名前に使われる文字は，電話番号における数字のように意味的に中立な記号として用いられるわけではなく，そのためこれを符号 (コード) 化とは呼ばないことが多い．しかし記号列の集合からそれが指し示すもの（人，地域，商品など）の集合への写像として名前が機能するという構造は同じであり，形式的には同じように扱うことができる．

符号化と呼びにくいもう1つの理由は，人の名前には同姓同名という現象があることである．地名や商品名でも同じことが言える．つまり記号表現と記号内容の関係は単射な写像とは限らない．実は筆者には，同姓同名 (漢字表記でもかな表記でも) で年齢も同じ，しかも同じ年に同じ大学に入学した人がいる．しかも現在，筆者と同じように大学勤務である．幸いにも職場と専門分野が違うので，さほど頻繁に混乱が起こるわけではないが，それでも過去に何度か混同から来るおかしな事件に遭遇している．

日本の名前は姓と名からなり，多くの国でもこのパターンが多い．姓は例外的な場合を除き，新たに作られることはない．むしろ，結婚に際し夫婦どちらかの姓を選んで新しい家族の姓とし，子供がそれを受け継ぐという現在の日本の制度を続ける限り，希少な姓は確率的に消滅していき，姓の数は単調に減るはずである．一方，名の方は子供が生まれるたびに新たに生成される．だから一般に，姓の種類の数の方が，名の種類の数より多そうである．

ところが日本の姓の数はなんと29万種類もあって，世界でも珍しいらしい．これは明治3年 (1870) 年に明治政府が平民に苗字を許し，さらに明治8年には必ず苗字をつけるように強制し，自分の苗字がはっきりしない場合は新しいものをつけてよいとしたため，爆発的に増えたもののようである．

日本で婚姻に際し改姓することになったのも明治の中頃からで，それまでは女性は結婚後も実家の姓を名乗っていたという．儒教の伝統がまだ生きている中国や韓国では，今でも女性は生家の姓を名乗る．それなのに中国の姓の数は少なく，3000という説がある．韓国の姓の数はさらに少なく，1985年の国勢調査で225という結果だったという．しかも，金，李，朴の3姓で人口の46%を占めるのだそうだ．日本のように爆発的に姓を創り出すといういい加減なことをしてこなかったからだろう．

日本の名前の付け方も，自由というかいい加減である．出生届の際に名前として使える漢字は，常用漢字だけでなく人名漢字というものが用意されていて量が増やされている（2003年現在で人名漢字は285文字）．その組み合わせ方も自由である．ただ，常用漢字と人名漢字合わせて2230文字と，カタカナ・ひらがなは使えるが，アルファベットやアラビア数字は許されない．また読み仮名を振ることになっているが，これと漢字との対応についても何の制限もない．ただ，読み仮名は戸籍には載らず，住民票にのみ登録されるようだ．

保険会社の明治生命が毎年，その年に生まれた赤ん坊に付けられた名前の人気ベスト10というのを発表しているが，それによると2002年の男の子は上位10は「駿，拓海，翔，蓮，翔太，颯太，健太，大輝，大樹，優」，女の子は「美咲，葵，七海，美羽，莉子，美優，萌，美月，愛，優花，凛」（同点のため11個）だという．これらのいくつかはどう読んだらいいか判らない．実際，同じ漢字表記にいくつかの読みがあるらしい．

あまりに自由なので，親は子供の命名に際し迷う．そこで字画などの姓名判断に頼り，わざわざ制約条件をつけ選ぶ範囲を狭めて命名するという，面白い現象がある．実際，本屋でその種の棚を見ると，字画による命名という類の本がおびただしく出ているの驚く．

ところがキリスト教，ユダヤ教，イスラム教の文化圏では，子供の名前は基本的に聖人の名前という限られた集合の中から選ぶことが多いようだ．たとえばフランスでは誕生日に結びつけられた聖人の名前から選ぶという習慣が強く，少し以前までは生まれた子供に対し定められた500ほどの聖人などの名前の中から名付けることが義務づけられていたという．筆者はある時，米国の学会が主催する会議のプログラム委員会に委員として出席して驚いたことがある．委員は全部で12名で筆者以外はすべて米国人．内2名が女性だったので米国人男性は9名いたことになるが，そのうち3人がDavidだった．アメリカ人は互いに姓でなく名で呼びあうので，紛らわしいことこの上ない．

内包と外延

つまり名前の付け方に2つの流儀があることになる．1つは日本流（と代表して呼んでしまう) で，名前が満たすべき制約条件を決めて，その条件を満たす範囲内で自由に生成する．もう1つはフランス流（とこれ

も勝手に代表させてしまう) で，定まったセットの中から選ぶ．付けることのできる名前の集合というものを想定したとき，日本流はその集合に入る要素がもつべき性質を定義するという意味で内包的 (intentional) 定義を与えていることになり，フランス流はその要素を具体的に並べあげるという意味で外延的 (extentional) 定義を与えていることになる．論理学では昔から，概念に対してその「内包」と「外延」という言葉を使う．たとえば「ジャイアンツの選手」というのが内包であり，「清原，上原，高橋 (由) ,...」などと並べあげるのが外延である．

命名規則

付けられる名前の集合についての内包的定義の記述は，命名規則 (naming rule) と呼ぶことができる．その典型は競走馬の命名規則である．日本の競走馬は，カタカナで 9 文字以内と定められている．さらにすでに登録されている名前と同一のものはもちろん，登録をすでに抹消されているものでも抹消後 5 年を経過していないものは付けられないそうだ．さらに有名な馬名（ＧＩ優勝馬や国際的に保護された馬名等）や馬名として不適当またはふさわしくないものも付けられないというあたりになると，規則としてはややあいまいになる．

生物には学名というものが付けられる．これはもちろん競走馬のように個体に付けられる名前ではなく，種に付けられるものである．学名の基となるはリンネ式の階層分類である．すべての生物が分類の上位から下位に向けて，界，門，綱，目，科，属，種という段階で分類される．たとえば人類は動物界脊椎動物門哺乳綱サル目ヒト上科ヒト属ホモ・サピエンス種である．学名はこの分類に基づいたラテン語表記で，まず属名を名詞で (頭文字は大文字)，次に種小名を形容詞またはその相当語 (小文字) で記述する．これが命名規則である．人名が姓と名からなるように，ここでも二名法が取られている．

プログラミング言語の識別子

コンピュータ・プログラムを記述するためのプログラミング言語は，典型的な人工言語である．プログラムを書く際には，プログラムや変数や関数に名前をつけなければならない．それらの名前を総称して識別子 (identifier) という．名前だから識別が重要なのは当然だが，とくに相手がコンピュータだから，ホモニムでも前後の文脈で判断するという融通が利かない．

たとえば C というプログラミング言語の識別子の命名規則は，次のようになっている．

> 1 文字以上の任意の長さの文字列．文字列に許される文字は，先頭は大文字か小文字のアルファベット，2 文字目以降は大文字か小文字のアルファベットか数字か_（下線）．

競走馬の名前のように長さ制限がないので，意味をよく表すように工夫した名前が付けられる．識別子にかなや漢字の使用を許すプログラミング言語もある．

しかし，古いプログラミング言語では長さの制限があった．たとえば今でも技術計算の分野ではよく使われる Fortran では，6 文字以内と規定されていた．しかも使える文字はアルファベットの大文字と数字だけで，小文字は使えなかった．Fortran77 までその制限があったが，Fortran90 と呼ばれる言語仕様では小文字の使用が認められ，長さの上限も 31 文字に広げられた．

長さの制限がなかったりゆるかったりすれば，名前がぶつからないようにするのは少し楽にはなるが，大きなプログラムを多人数で開発する際に，すべての識別子に重複が起こらないように管理するのは大変である．プログラマもそのようなことに余計な神経を使わないで，プログラミング作業を行いたいはずである．そこでほとんどのプログラミング言語で採用されている巧みな仕組みとして，名前の有効範囲 (scope) という概念がある．

プログラムは何らかの単位から構成される．この単位はプログラミング言語によって呼び方も大きさもまちまちであるが，ここではそれをモジュールと呼ぶことにしよう．変数をあるモジュールで宣言した時，その名前はそのモジュールの中だけで有効で，別のモジュールに同じ名前の変数があっても，それは別のものを指す，というのが名前の有効範囲の考え方である．モジュールの中にモジュールがあるという階層構造を許す場合が多いが，その時も，あるレベルのモジュールで宣言された名前の有効範囲は，そのモジュール自身かそれに含まれるモジュールの中（それがまたモジュールを含んでいればそれも含む）とする．これはある意味で，名前を文脈に応じて判断していることになり，あいまいさをなくしながらも，すべての名前を文字列として区別しなければならないという制約をはずすうまい工夫である．

構造的な表現

これまで扱ってきた電話番号のような符号列も人名も生物の学名も，対象となるものを指すための記号表現だった．それらをひっくるめて，改めて「名前」と呼んでもよい．しかし，ものを指す方法は名前によるばかりではない．そのものが持つ性質を並べ挙げて同定するというやり方もある．たとえば「イチロー」という名前である人を指す代わりに，「マリナーズという野球チームのレギュラー選手で，守備位置はライトで，打順は1番」というような具合である．

このように対象を定める性質の項目を属性という．属性には種別と値がある．イチローの例で言えば，「所属チーム」という属性種別の値が「マリナーズ」であり，「守備位置」という属性種別の値が「ライト」であり，「打順」という属性種別の値が「1番」である．このように属性の種別とその値の対の集まりという構造で情報を表現することは，きわめてよく行われる．

たとえば日頃，申込書の類を書かされることが多い．それらには名前，住所，電話番号，性別，年齢，メールアドレスなどの欄がある．欄は属性の種類に対応し，そこに書き込むものが属性の値である．最近は申込書という紙に記入する代わりに，インターネット上のWebのページから申し込む場合のように，指定された欄にキーボードから文字を打ち込んだり，メニューリストからマウスでクリックして属性の値を選ぶことも多くなった．

名前で指すとの別の方法として，属性と値の列挙があるとこの節の話を始めた．それなのに申込書の例では名前も属性となっているのはどういう訳だろうか．実際，名前で1つに定まらない場合，さらに社員番号とか学生証番号といった識別符号を属性に入れることもしばしばある．つまり，このように属性で対象を表現するのは，それを名前で指す代わりというわけでは必ずしもなく，むしろその対象のもつさまざまな情報をまとめて表現する手段と考えたほうがよい．

****

これから先，申込書のようなデータを集めたデータベースの話，それと等価な表現としてのn組や表の話，属性の値がまた構造を持った表現であるような階層構造，その表現法の1つとしてのXMLの話などを考えているが，長くなるばかりなので，SEAMAILの原稿としてはこの辺でやめにする．

参考文献

[1] 吉田民人. **情報と自己組織性の理論**. 東京大学出版会, 1990.

[2] 玉井哲雄. 国際的情報社会に立ち向かう. 浅野攝郎，他, 編, **東京大学は変わる–教養教育のチャレンジ**, pp. 100–115. 東京大学出版, 2000.

大学におけるソフトウェア工学教育の一例

落水 浩一郎

(北陸先端科学技術大学院大学)

ソフトウェア工学をどのように教えるのか、ソフトウェアを作る力をどのようにして身につけてもらうのか。まずは失敗談からはじめよう。

失敗談その１：

１５年ほど前、一年ほどかけてソフトウェア工学の歴史をまとめ、自分なりにもれなく体系化できたと思えた時点で講義してみた。結果は散々だった。要求定義技術はこのような動機で出現し、現在かくかくしかじかの流儀があり、それぞれの流儀はこのような特徴をもち、例えば、このような例題はこんな風に表現できる。先生「どうですおもしろいでしょう」、生徒１「経験がないのでよくわかりません」、生徒２「あたりまえのことを書いてあるみたいですね」、先生「いやいや、よく知らないけど実社会では大事なことらしいです。ここでは簡単な例題をもとに原理原則をしっかり理解してください」生徒「？？」

失敗談その２：

ソフトウェア工学は実践の学問であるという信念のもとに、とにかくバリバリものをつくるという講義もやってみた。要領を得た方法論の解説をやり、ツールの操作法等を例題をもとに示した上で、「さてこの大きな怪物を退治してみましょう」と始めた。生徒達は系統的にシステムを作る力がつくと信じて皆やる気をもって始めた。最初は、ハッピーだった。状態遷移図の状態をどのようにして認識するのか、継承関係をどのように整理したらよいのかというような基礎概念、基礎力に関する質問があいつぎそれに丁寧に答えてあげることで評判は上々であった。そのうちに学生は怪物にふりまわされはじめ、先生は「どのくらいできましたか」としかいわなくなり、学生一人一人の状況に応じたシステム構築に関する適切かつ系統的な助言をあたえられなくなった時点で講義ではなくなった。

失敗談その３：

UMLを丁寧に解説するという講義もやってみた。結果は、先生は話ながら飽きてくる。生徒は「本を読めばわかることを先生はしゃべっている」という反応であった。

失敗談のあとは成功談が続くべきだが特にない。自己満足した経験は数回ある。それは、いつも共通した時期におこる。新しい講義をはじめた２年目である。新しい話題や技術を自分が理解したその通りに講義でしゃべる。一年目はそれでもまだこなれていない。２年目はいつも学生が熱心に聞いてくれるしおもしろがってくれる。３年目になると要領がわかってきて流すところが出てくる。とたんに学生の目の輝きが薄くなる。４年目に再度新しい話題を補充して再チャレンジをこころみるが、すでにできあがった慣性は避けられない。

以上のような失敗の連続にも関わらず。今年から再び「ソフトウェア設計演習」という講義を新たに開講し実施した。結果をご紹介する前に私がJAISTで行ったことのある講義を中心に関連するいくつかの講義を紹介しよう。

「アルゴリズムとデータ構造」の講義はあまり失敗しない。情報系以外からきた人に対する接続講義でもあり、学生のレベルにあわせて十分注意しながら講義を進めている。この講義に関しては昨年からパワーポイントを利用するのをやめた。黒板に絵を書き、プログラムを書き、それを時間をかけて説明する。とくに、基本的な制御構造やデータ構造に関しては計算機のハードウェア動作との対応を中心に説明しつつ、だいたい計算機が理解できたかなというころ、アルゴリズムの効率（計算の複雑さ）と正しさを念頭におく説明に進めていく。

「プログラミング方法論」（講義経験はない）では関数型プログラミング、「ソフトウェア設計論」ではオブジェクト指向の基礎概念の徹底理解をはかっている。「ソフトウェア設計論」という講義はすでに１０年ほどやってきた。講義の初期バージョンはOMTであったが、数年前にUML、UP、JAVAに変えた。講義の目標はオブジェクト指向に関する基礎概念と原理および効用の理解、モデリング言語（UML）とプログラミング言語（Java）の習得、オブジェクト指向方法論の理解（UP）、いくつかのソフトウェアパターンの紹介である。週に2回計15回の講義で、上記内容を説明しつつ、午後週に１回演習を行う。

「ソフトウェア設計演習」の目的は、「ソフトウェア設計論」の単位を習得していることを前提として、ソフトウェア開発に関する最低限の経験をつみつつ成功体験をもたせることである。ツールはIIOSS(上流CASE）、JBuilder7（下流CASE）、CVS（版管理システム）、mailとWWW(情報交換、情報共有）を利用する。例題は、Hassan Gommaによる「Designing Concurrent, Distributed, And Real-Time Applications with UML」にある、Elevator Control System、 Banking System、Cruise Control and Monitoring System、Distributed Factory Automation System、Electronic Commerce Systemを使う。Hassan Gommaの上記例題は、問題記述、ユースケースモデルとユースケース記述、各ユースケースごとの分析クラスのコラボレーション図とイベント記述、統合コラボレーション図、制御オブジェクトのステート図、設計クラスのクラス図（の一部）とコラボレーション図またはシーケンス図および必要なものにはステート図、サブシステム設計、タスク設計、配置図、一部の例題にはユースケースに基づくαテストの方針などが丁寧に記載してある。方法論はCOMETと名付けられたUPの変形であり、ユースケースごとに実現統合を繰り返す。

また、チーム作業を訓練するため、数人で共同して上記例題プログラムを開発させる。役割分担、ベースライン管理、統合テスト、文書化などの良いマナーを身につけさせることが狙いである。

さて効能書きは良いとして実際はどうであっただろうか。２ヶ月間の演習を振り返ってみよう。

(1) まず、講義最初の日にチーム分けをおこなった。結果として5チームできた。そのうち1チームは日本語がわからない留学生5人からなるチームとなった。というわけで、講義の説明はすべて日本語と英語の二本立て、学生さんの発表資料はすべて英語で提出ということになった。さきゆきあぶなそうですね。

(2) 問題記述、ユースケースモデルとユースケース記述、各ユースケースごとの分析クラスのコラボレーション図とイベント記述、統合コラボレーション図の部分は「良い分析を見習う」という主旨で、教科書の内容を各自良く理解して自分のものにすることにした。すなわち、要求獲得の苦労は無し。また、分析クラス発見のノウハウも身につかず。10日後に発表会をおこなった。一応読んで理解してきてはいるが、まだ自分のものになっていないチームが大部分。１チームだけ仕様が現実的でない部分を自分たちなりに工夫してすべての文書に改良結果を反映させた発表をした。感激！

(3) 次の一週間は、第一次プロトタイピング計画立案と役割分担の決定。CVSアドミニを各チームで一人決めてベースライン制御方針の決定を指示。また、10数枚の各種UML図面群をIIOSSを利用して入力することにより操作法に習熟するように指示した。

(4) 一週間後に発表会。IIOSSの操作法の習熟だけが達成されている状態。ほかの事項は経験がないのでとにかくやってきたというレベルで、発表に迫力なし。しかしここでは理屈をこねないことにして、自分達で立案した計画をとにかく実施するように指示。また、情報公開のためのホームページを立ち上げる。次の課題はJavaとJbuilderを利用した第１次プロトタイプの実現。2週間後に発表会。

(5) 2週間後。すべてのチームがとにかく不十分な仕上がり。まずプログラムが動いていない。6-7人いたチーム構成が、あるチームは3-4人に減っている（脱走者出現）。各チームに共通した特徴は、一人が制御部分を担当しそれはほぼ完全な仕上がり、また別の一人はGUIを担当しそれも何

とか動きつつある。もちろん連動していない。チームの残りのメンバーはとにかく必死でついてきている状態（この状態は最後まで続いた）。ただし、留学生チームは例外、不十分なできにはちがいないがうまく役割分担し連携している。また、その中でも仕上がりが比較的良いチームが一つだけある。これはチームにスーパーマンがいたおかげ。そのかわり力のない人はほとんどやることがない状態。特に重要な問題点を発見。版管理が系統的になされていない。とにかくCVSに突っ込んだだけ。Karl Fogel著、でびあんぐる監訳「CVS バージョン管理システム」の2章と6章を読んでちゃんと工夫するように指示したのにこれはどういうことだ（陰の声：実はこのような状況をひきおこすことが最初からの狙いだったのです）。

ここで、ソフトウェア工学の知識を生かすように指示（喉が渇いているときの水はうまい）。まず、coplienの組織パターンを紹介し、アーキテクチャに基づく役割の再設計を指示、また、スキルのちがう人々でチームを組む時の作業割り当て法を助言。さらに、設計クラスを完全に洗いだしたあと、CVSプロジェクトを設計すること、また、リポジトリでベースライン制御ができるようにタグ名を決めること、 Jbuilderには各自リポジトリの全内容をインポートしcopy-modify-mergeの機能を活用することを指示。さらに、コンパイルが完了したもののみコミットするように注意を与えた。ここらへんは皆どうしてよいかわからず困っていたところらしくて、全チームが採用してくれた。以後チームの活動が協調した（ように思える）。この時点で、第2次プロトタイピングに入る予定を変更して一週間後に再発表を指示。なんとほとんどのチームが、(5)で述べてきた問題点をクリアしてきた。

(6) 演習開始から一ヶ月と少し経過した。途中に正月休みがはいったのであと3週間。次の一週間は第2次プロトタイピング。3チームは順調に進んでいる。残りの二つのチームのうち一つは脱落者が多く、量的にこなせない見通しなので規模を縮小した。もう一つのチームは人数がおおすぎてある人の遅れがが他の人の足を引っ張っている状態。進んでいるチームはおいといて、遅れているチームに対する支援を開始（この演習の目的は最後までついてきた全員に成功体験をもたせること）。

(7) 一週間後、第2次プロトタイピング終了時点でテスト計画の立案を指示。このへんはオブジェクト指向にピッタリした理論がないので落水は苦戦。

(8) いよいよ講義終了一週間まえになった。今日は最終発表会。第3次プロトタイプまで完了している予定である。各チームの発表を一日かけて順に聞いた。設計クラス以降、教科書にのっていない設計結果の発表を聞いたあと、センスのよいGUIをもつ動いているシステムのデモ、最終的に二人きりになってしまったチームをのぞいて小気味よく時間がすぎていく。3人ほどやったふりをしてまぎれこんでいる人がいたので不合格を通知。最初の年で、先生のほうがまだこなれていなかったがとにかく最後まできた。ソフトウェア開発に関する最低限の経験をつみつつ成功体験をもたせることに成功したかどうかはまだ不明であるが、これが大学の教室講義の限界かなと一人で思って自分をなぐさめる。最後に文書化についての指示を与える。

すべての文書をホームページにあげること。Javaプログラムにコメントをつけること。回帰テストのためにすべてのテストケースとテスト結果を残すこと。利用手引き書を書くこと。保守に備えていっさいがっさいをCVSに放り込むこと等々。１チームだけ一週間後の再発表を指示して講義は無事終了した（ように思える）。

さて、私としては結構頑張っているつもりですが、読者の皆さんのご感想はいかがでしょう。良く鍛えられた人材を世に送り出すために、ぜひ、皆さんのご助言をお願いいたします。

ところで、このような活動の支援環境を落水研では開発しています。例えば、

(1) 拡張可能な版管理システム

共同ソフトウェア開発をスムースに進めるには様々な合意を守らなければなりません。例え

ば、コーディング規則、リリースエンジニアリングの方法、フィーチャーフリーズの方法、テスト法、アクセス制御などがあります。これらをCVSの拡張コンポーネントとして利用するために、拡張機能を呼び出すタイミング、システムが反応すべきイベント、イベントハンドラーの呼び出し方、イベントが発生したときに実行すべき処理などを拡張可能なフレームワークをほぼ開発しました。CVSとアダプター、システムフレームワーク、拡張コンポーネントを利用して冒頭にのべた作業を自動的に支援する予定です(拡張コンポーネントの開発は今年の課題です)。

(2) SCORM標準に基づくLMSの開発

今後の開発環境は技術革新に対応し、スキルの違いを吸収するため、参加者の能力増幅の機能を持つ必要があると考えています。オブジェクト指向に関する基礎概念、システムの構造と利用法に関する説明などの知識コンポーネントを管理し、利用者の要求に応じて必要部分を提供するシステムを開発しています。SCORM標準に基づくLMSの開発がほぼ完了しました。中身はまだありません。

(3) 情報共有機構の開発

Gforgeの軒先をかり、電子メール群から討議内容を抽出するツールを張り付けたシステムの開発を進めます。後者は開発が一段落し運用に移行中です。

(4) 開発状況の不安定さの共有機構

ここ5年ほど開発を進めている、開発状況の不安定さを共有させ、問題発生時に関係者の頭をコツンとたたくというシステムの開発は、最後の詰めであるアクティブコーディネータの開発が難航しています。

来年度以降の講義では上記のツール群も利用して研究と実践の一致をはかりたいと考えています。また、現在、タイ、ミャンマーの研究者や学生さん達と協力して国際ソフトウェア共同開発の場を構築中です。すでに人の手配はおわりました。タイのチュラロンコン大学からは留学生が来ています。またミャンマーのUCSYからもこの4月には研究者が参加します。研究費から費用をだすので悲鳴をあげそうですがここが辛抱のしどころです。アジアのコア人材の育成とソフトウェア工学に関する研究成果のリアルなフィールドテストの場をもうけることが狙いです。

成果がでしだい、SEAMAILにてご紹介し諸賢のご批判をあおぎたいと思っています。

アスペクト指向プログラミング概観

野呂 昌満
(南山大学)

プログラミング技術がソフトウェア開発技術の主要な柱であるとの認識に立ち, アスペクト指向プログラミングについて解説する. アスペクトとは, ある特性により規定できる, ソフトウェアの一部と定義できる. アスペクト指向技術は, あるアスペクトに関連するコードが散在するとき, それらをひとまとめにして, 支配的分割(dominant decomposition)によって得られる構造と分離する事を可能にする技術である. さらに, 分離されたコード群はお互いに依存することなく変更可能になる. 代表的なアスペクト指向プログラミング言語を紹介しながら, アスペクト指向プログラミングについて説明し, アスペクト指向プログラミングの拡張との位置付けで MDSOC(Multi Dimentional Separate of Concern, コンサーンの多次元分離) を解説する.

1. はじめに

アスペクト指向技術[Elrad2001_1] はこれまでのソフトウェア開発技術では解決し辛かった問題に対処可能なものとして注目されている. アスペクトとは, ある特性により規定できる, ソフトウェアの一部分と定義できる. アスペクト指向技術は, あるアスペクトに関連するコードが散在するとき, それらをひとまとめにして, 支配的分割によって得られる構造と分離する事を可能にする技術である. さらに, 分離されたコード群はお互いに依存することなく変更可能になる.

近年, 過去のソフトウェア開発技術の集大成としてのオブジェクト指向技術は実用に供され, 生産性および信頼性の向上に寄与している. しかし, オブジェクト指向で設計・実現されたソフトウェアでは, 複数のオブジェクトにまたがる大域的な特性をオブジェクトとして実現することに限界がある. これを打破すべく, 領域依存言語(domain-specific languages), generative programming, generic programming, constraint languages, 自己反映計算とメタプログラミング, feature-oriented programming, views/viewpoints, asynchronous message brokering などの, POP(Post-Object Programming)技術が提案されている[Czarnecki2000]. これらのPOP技術が共通して目指すところは, 異なる特性を矛盾なく取り扱う事である. とりわけ, アスペクト指向技術はオブジェクト指向技術が過去そうであったように, 数多く存在するPOP技術の代表的なものの1つと考えられる.

本稿では, プログラミング技術がソフトウェア開発のもっとも重要な基盤技術の 1つであるとの認識に立ち, 既存の代表的なアスペクト指向プログラミング言語を概観する. 以下2節で, 既存のアスペクト指向プログラミング言語を紹介しながら, アスペクト指向プログラミングについて説明する. 3節では, MDSOC をアスペクト指向プログラミングの拡張と考えて解説する.

2. アスペクト指向プログラミング

プログラミング言語あるいはその計算モデルはソフトウェアの構造を規定するものである. この構造は支配的分割による構造またはコアコンサーン(Core Concern)アスペクトと呼ばれ, その構造がソフトウェアの複数のアスペクトすべてをいつも矛盾なく局所化して表現するために役立つとは限らない. 一方で, Parnas[Parnas72]が述べているように, システムをモジュールに分割するさいに, いくつかの基準がある. ある基準に従う分割が別の基準に従う分割と矛盾しないと言う保証はない. アスペクト指向プログラミングでは, 分割を規定するこれらの基準をコンサーン(concerns, 関心)と呼ぶ. 異なるコンサーンを別々に記述すればより良いモジュール分割が得られると言うのがアスペクト指向プログラミングの基本的な考え方である.

2.1 アスペクト

図 1[Elrad2001_1] にアスペクトとクラス構造が横断的に関連する例を示す(An aspect crosscuts classes.) この例は, 図エディタの内部形をオブジェクト指向設計したものである (UMLのクラス図で

書いてかる.) PointとLineがFigureElementとis-a関係にある型階層と FigureがFigureElementの集まりで構成されている事を示す, オブジェクト指向設計としては素直なものである.

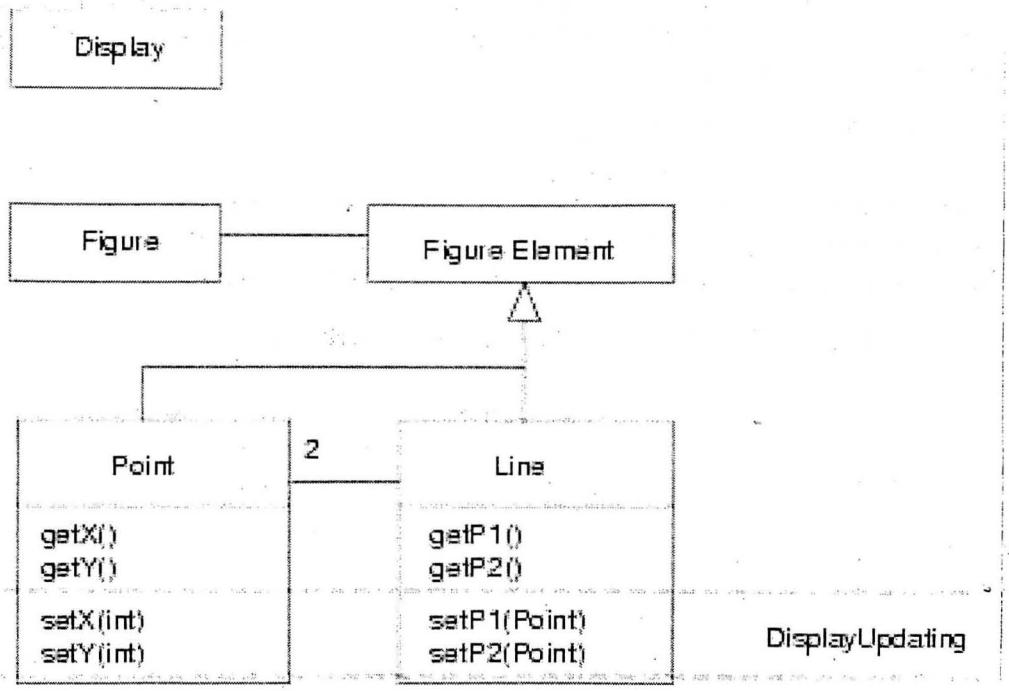


図1. アスペクトと支配的分割の横断的関係

図1の構造を前提として, スクリーンマネジャのコンサーンを考えてみる. 図やその一部分, すなわち点や線が移動すると, スクリーンマネジャは移動をつねに検知しなければならないとする. これを実現するためには, 図の移動に関連するメソッドはすべてスクリーンマネジャに通知するようにしなければならない.

図中のDisplayUpdatingで示したメソッド群がスクリーンマネジャのコンサーンを実現するものである. 図から直感的に理解できるように, スクリーンマネジャのコンサーンが規定するアスペクトがクラス構造と横断的に関連している.

2つのコンサーンが横断的であるとは, それらの実現コードが共通部分を持つ事である. クラス構造はデータコンサーンが規定するアスペクトが含意するものと考えられる. すなわち, この例はデータコンサーンとスクリーンマネジャのコンサーンが横断的に関連しているものである.

一般にプログラミング言語の計算モデルはコンサーンを表現している (このコンサーンをコアコンサーンと呼び, それに従ったモジュール分割を支配的分割と呼ぶ.) オブジェクト指向計算モデルはデータコンサーンを表現したものであり, クラス構造はデータコンサーンが規定するアスペクトを実現した構造である. 手続き指向計算モデルは制御コンサーンを表現し, 手続きの入れ子構造は制御コンサーンが規定するアスペクトを実現したものである. アスペクト指向プログラミング言語においては, コアコンサーンが規定するアスペクトと横断的なアスペクトを一級言語要素(firstclass language construct) として取り扱える. 以下では, "オブジェクト指向プログラミング言語が提供するクラス構造はオブジェクト指向計算モデルのアスペクトを実現したものである", "クラス構造はオブジェクト指向プログラミング言語のコアコンサーンアスペクトを実現したものである", 等と表現する.

2.2 合流点(Join Point), ポイントカット(Pointcut), アドバイス(Advice)

合流点, ポイントカット, アドバイスは AspectJ[Kiczales2001_1,Kiczales2001_2]の用語であるがアスペクト指向プログラミング言語を説明する上で有用なので以下の説明ではこれらを用いる. 合流点はコアコンサーンアスペクトと別のアスペクトの接点である. 合流点は言語要素の一部分である. ポイントカットは合流点の集合とそれらが保持する値を参照する手段である. アドバイスはアスペクトを

実現するコードの塊でポイントカットに設定し, 合流点に実行が到達したときに実行される.

コマンドパターンとステートパターン[Gamma95]を使ってアスペクトを分離して記述する例を用いてこれらの概念を説明する. 16進数を10進数に変換するプログラムの構造を図2に示す. それぞれのクラスの役割は以下のとおりである.

- hex: 16進数クラス. 値と文字列を保持.
- digit: 10進数クラス. 値と文字列を保持.
- scanner: 16進数の字句解析を行う. ステートパターンを使って実現.
- process: 走査時の行動をカプセル化. コマンドパターンを使って実現.
- calc: 16進数を走査, 字句解析し, 10進数に変化する処理をカプセル化.

コマンドパターンを使って実現. 動的な挙動は図にシーケンス図で示した.

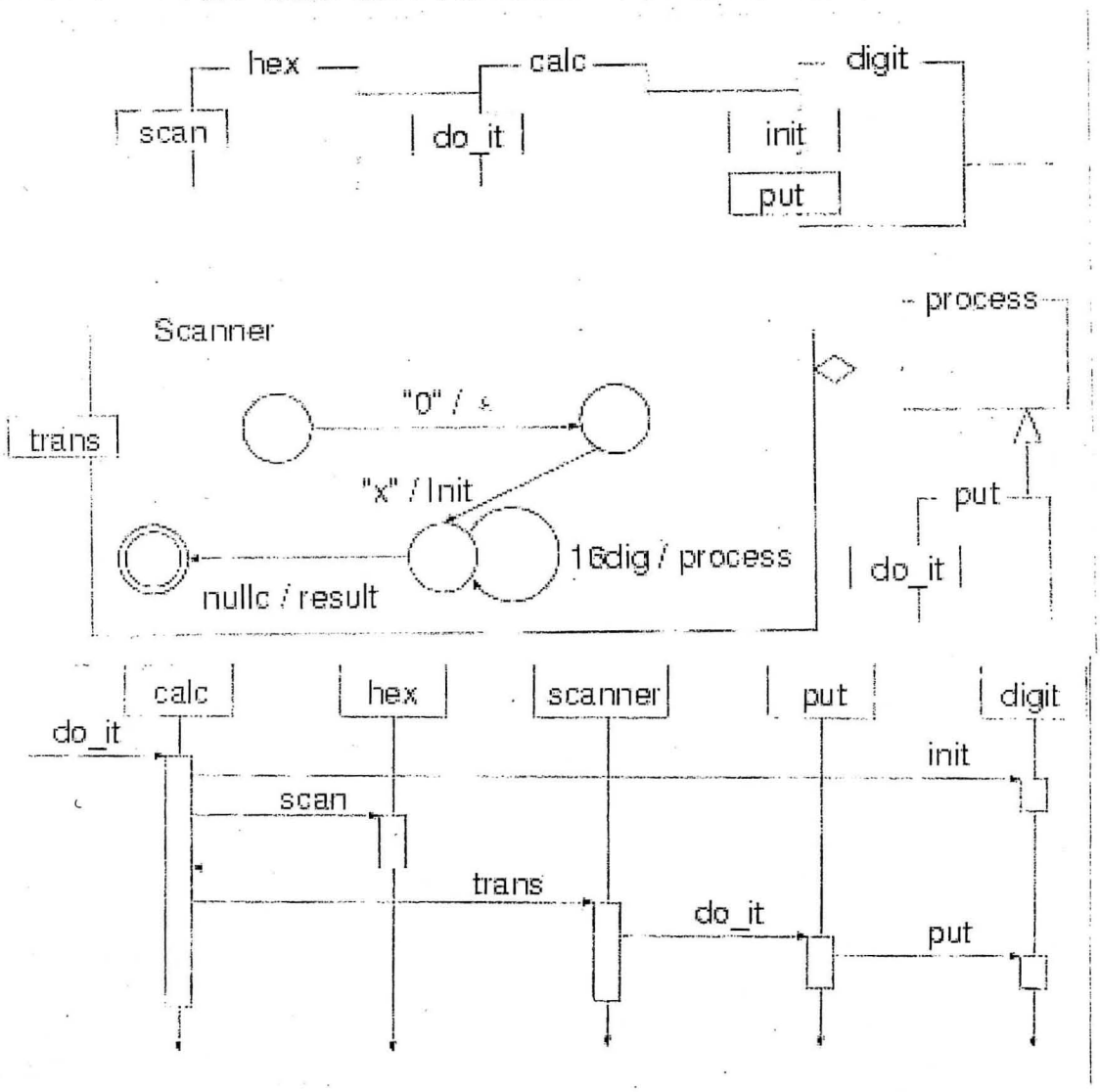


図2. デザインパターンを使ったアスペクトの分離

この例では, 字句解析とその意味処理という2つのコンサーンがある. 意味処理アスペクトを分離したものとしてこの例を眺めると, scannerがポイントカット, scannerのtransメソッド中の

```
p->doit(c);
```

が合流点, processがアドバイスに相当する.

2.3 アスペクト指向プログラミング言語

アスペクト指向プログラミング言語の実現方法は,

- あるプログラミング言語のライブラリとして実現する,
- 新しい言語を設計・実現する,

の2つに分類できる. ここでは, それぞれの代表的な例を挙げて説明する.

2.3.1 適応的(Adaptive)メソッド(Demeter/Javaライブラリ)

適応的メソッド[Lieberherr2001]はクラス構造の中に横断的に散在する処理を局所化してカプセル化するためのパターンでプロパゲーション(propagation)パターン[Lieberherr96] と呼ばれていたものの発展形である. DJライブラリ(Demeter/Javaライブラリ)は適応的メソッドを実現したJavaパッケージである.

図3[Lieberherr2001]にDJライブラリを使ってJavaで書かれた適応的メソッドを示す. この例はある会社の事務システムアプリケーションを記述したものである. この事務システムはオブジェクト指向設計・実現されており, 会社組織はクラス構造にモデル化されている. アスペクトとして特定のアプリケーション(この例の場合は, 給与計算コンサーン) を考える.

```
import edu.neu.ccs.demeter.dj.ClassGraph;
import edu.neu.ccs.demeter.dj.Visitor;

class Company.{
  static ClassGraph cg = new ClassGraph();
  Double sumSalaries(){
   String s = "from Company to Salary";
   Visitor v = new Visitor(){
    private double sum;
    public void start() { sum = 0.0; }
    public void befor(Salary host) {
     sum += host.getValue();
    }
    public Object getReturnValue() {
    return new Double(sum);
    }
   };
   return (Double)cg.traverse(this, s, v);
  }
  // ... rest of Company definition ...
}
```

図3. 適応的メソッド

図のプログラムでは, 組織はClassGraph型としてモデル化されており, その実体を変数cgで保持する. 適応的メソッドはsumSalariesであり, 給与計算コンサーンが規定するアスペクトを記述している. 適応的Visitor vが給与計算の論理を実現している. 式cg.traverse(this, s, v)は, 会社組織の構造をたどりながら, "from Company to Salary" が示すポリシーに従って, 給与の合計を計算する. メソッドtraverseでポリシーを参照するさいには, Javaの自己反映機能が使われており, Companyクラスの実体からhas-a関係にある実体をたどり, Salaryクラスの実体を見つける毎に, 給与計算を行う, と言う実現がなされている.

この例から分るように, 適応的メソッドであるsum-Salaries がアスペクトを実現している. ポイントカットは変数cg内に実現されている. 合流点はtraveseメソッドの中に定義されており, 会社組織のクラス構造をたどる前に, 適応的Visitorのstartメソッドが呼ばれ, 構造をたどっている最中にはbeforeメ

ソッドが給与の合計を計算するために必要回呼び出される．構造をたどり終わったあと，getReturnValueメソッドが起動されるように記述されている．アドバイスは適応的Visitorとして実現されており，上のように，合流点毎に異なるアドバイスが起動される．

2.3.2 AspectJ

AspectJ[Kiczales2001_1,Kiczales2001_2,AspectJ]は汎用的なアスペクト指向プログラミング言語であり，Javaの拡張として定義されている．合流点，ポイントカット，アドバイス等はAspectJで定義された用語である．これらの用語はAspectJが汎用的で広く認知された事からアスペクト指向プログラミング言語の説明のさいに広く用いられるようになってきた．

AspectJでは，アプリケーションはJavaで記述し，アスペクトの定義は構文的にはクラス定義と類似した AspectJ 独自の言語要素を用いて行う．アプリケーションとアスペクトの連係は合流点，ポイントカットおよびアドバイスに記述する．

ここでは，上で使った図エディタの例を使って説明をすすめる．図4[Kiczales2001_1]に詳細化した図エディタのクラス構造を示す．

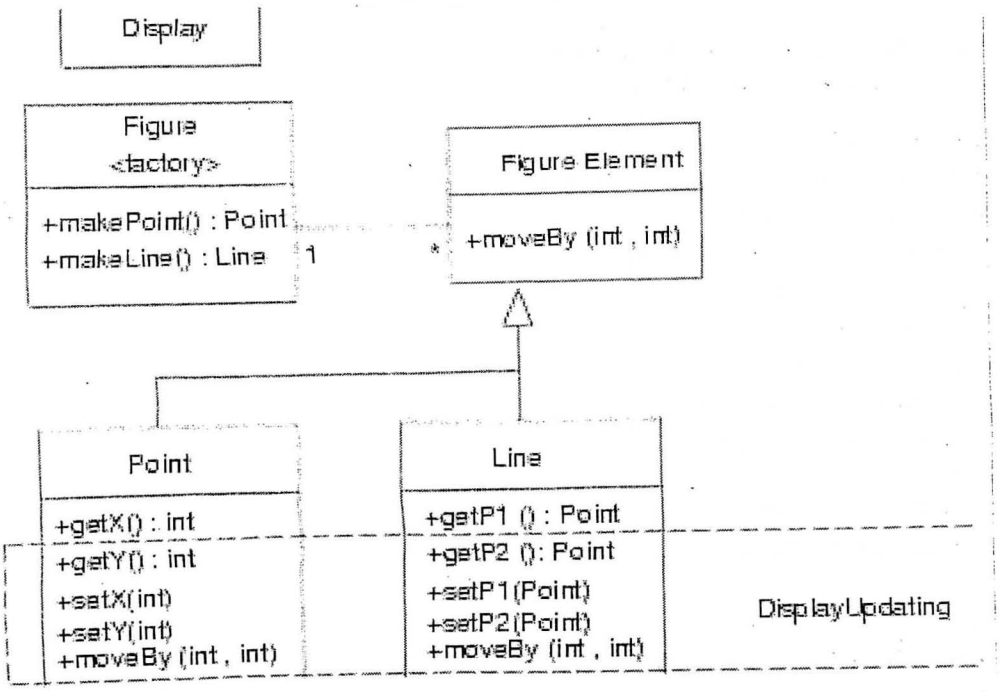


図4. 図エディタ内部形のクラス構造

図のようなクラス構造においてアスペクトDisplay-Updating を考える．合流点は

```
call(void Point.setX(int))
call(void Point.setY(int))
```

のように定義される．それぞれ，Pointクラスの実体のsetXまたはsetYメソッドを合流点として定義している．

ポイントカットは以下のように定義される．

```
pointcut move():
  call (void FigureElement.moveBy(int, int)) ||
  call (void Point.setX(int)) ||
  call (void Point.setY(int)) ||
  call (void Line.setP1(Point)) ||
```

```
call (void Line.setP2(Point));
```

これはポイントカットmoveが5つの合流点で構成されている事を示している.

この例のアスペクトは以下のように定義される.

```
aspect DisplayUpdating {
  pointcut move(): <上と同じ>
  after() : move() {
    Display.needsRepaint();
  }
}
```

これはポイントカット中の合流点が実行されたあとに, 表示画面を更新することを示している.

AspectJでは, アスペクトはウィーバ(Weaver)と呼ぶプリプロセッサで処理される. ウィーバは上で示したようなアスペクトの定義を読み込んで, 意味的に等価なJavaプログラムを生成する.

2.3.3 Composition Filters

合成フィルタ(composition filters)[Aksit2001,Bergmans2001]は横断的コンサーンを表現するためのプログラミング技術である. メッセージの受信時と返信時にメソッドの起動条件, 前処理, ならびに後処理が記述できる. 合成フィルタは入れ子にでき, 前処理は入れ子にした合成フィルタへのメッセージ送信としても記述できる.

図5[Bergmans2001]に合成フィルタの例を示す. 合成フィルタはアスペクトとアプリケーションをひとまとめにしてカプセル化するためのクラスである. 図中のinput filters, output filters, internalがアスペクトを記述している. implementation partはアプリケーションを実現するクラスである. 本来innerオブジェクトに送るべきメッセージを合成フィルタの実体であるClaimDocumentが受け取る. メッセージの送り先はinput filtersで指定され, この例の場合はinner に送られると同時に, internalで合成フィルタの入れ子として実現されているdoc にも送られる. メソッド実行後, この例では, output filtersに何も指定されいないので, innerの返答がそのまま送り手に返される.

この例からわかるように, ポイントカットはinput filtersおよびoutput filtersで記述され, アドバイスはinternalとして記述される.

ComposeJやThe Sina Language等の合成フィルタの実現の詳細については TRESE Webページ[TRESE2002]を参照されたい.

2.4 自己反映計算, メタオブジェクトプロトコルとアスペクト指向プログラミング言語

アスペクトの処理方式としては,

- ウィーバを使う生成系方式と,
- 自己反映的実現方式,

が考えられる. AspectJの実現が前者の典型的な例であり, 後者の例としては, DJライブラリが挙げられる. 前者はアスペクトを静的に取り扱うものであり, 後者は動的に取り扱うものである[Sullivan2001].

あるシステムが自己反映的であるとは, システムの自己表現(self representation)とその振る舞いが因果的結合 (causally connected) 関係にあることと定義できる. 自己表現はメタレベルで処理され, 振る舞いはベースレベルで処理される. 自己反映的プログラミング言語を考えてみると, 自己表現はプログラミング言語の意味を実現したものとなる. プログラミング言語の意味とその振る舞いが因果的結合関係にあるということは, 言語の意味を動的に変更できる事を指す[Maes87,Smith84]. すなわち, 自己反映的プログラミング言語ではメタレベルとベースレベルが同一の記述方法を持ち, ベースレベル

からメタレベルを変更する事で, 言語の意味を動的に変更できる. オブジェクト指向プログラミング言語の場合, メタレベルへのアクセスはメタオブジェクトプロトコルに従って行われる[Maes87].

自己反映的にアスペクトを取り扱う場合, ポイントカットをメタレベルの情報とする. ポイントカットはメタオブジェクトプロトコルに従って設定する. アドバイスはベースレベルで記述し, メタオブジェクトプロトコルに従って設定する.

自己反映的にアスペクトを取り扱う事の利点としては, コンパイル(Weaving, 織り合わせ)を省略できる, 動的にアスペクトを変更できるので柔軟な記述が可能になるなどの利点がある[Sullivan2001]. しかし, 型検査等の意味検査を動的に行う事になり実行効率に問題を残すかもしれない.

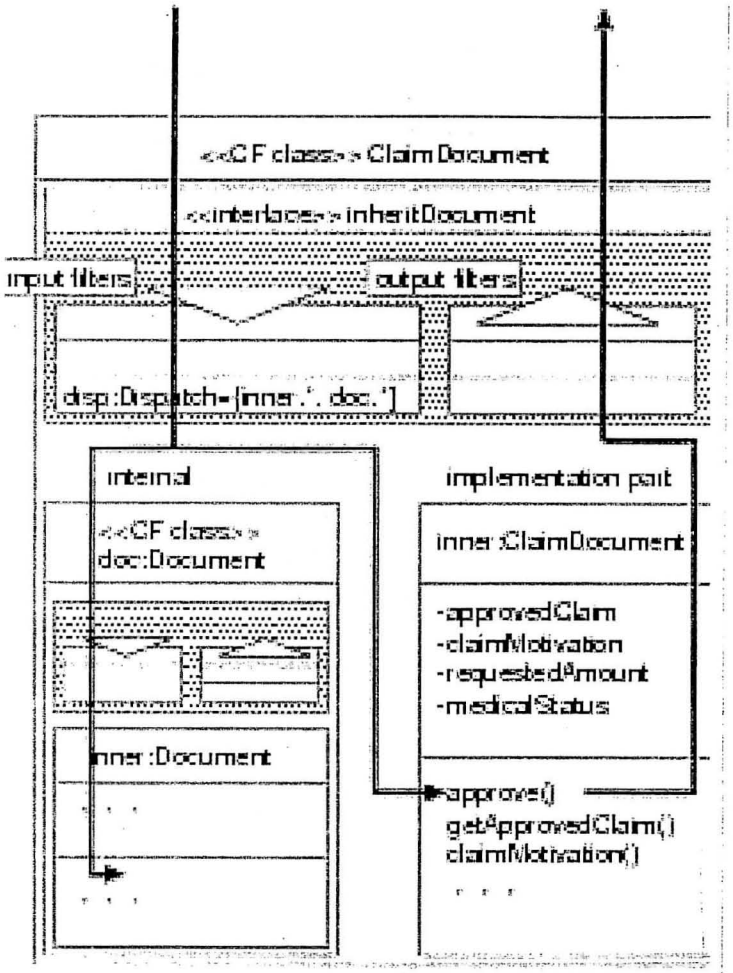


図5. 合成フィルタ

3. Multi Dimentional Separation of Concern

アスペクト指向プログラミング言語は, 計算モデルのアスペクトとして記述するアプリケーションから異なるコンサーンが規定するアスペクトを分離することを支援する. これをさらに一般化し, アプリケーションはアスペクトの集合であるとするのがMDSOC[Ossher2001_1,Tarr99]の考え方である.

Hyper/J[Ossher2001_1,Ossher2001_2], では, アスペクトをクラス木として記述し, 複数のアスペクトを合成してアプリケーションを作る. 合成の記述はHyper/J独自の構文を使って行う. 合成は, ウィーバによって処理されアプリケーションが生成される.

図6[Ossher2001_1] にある会社の組織図をオブジェクト指向でモデル化した例を示す. この構造の中には, 給与情報と人事情報が入り混じって記述されている. 各クラスの下線をひいたメソッドは給与

情報を取り扱うものである．これは給与・人事情報コンサーンとデータコンサーンが横断的に関連している例である．

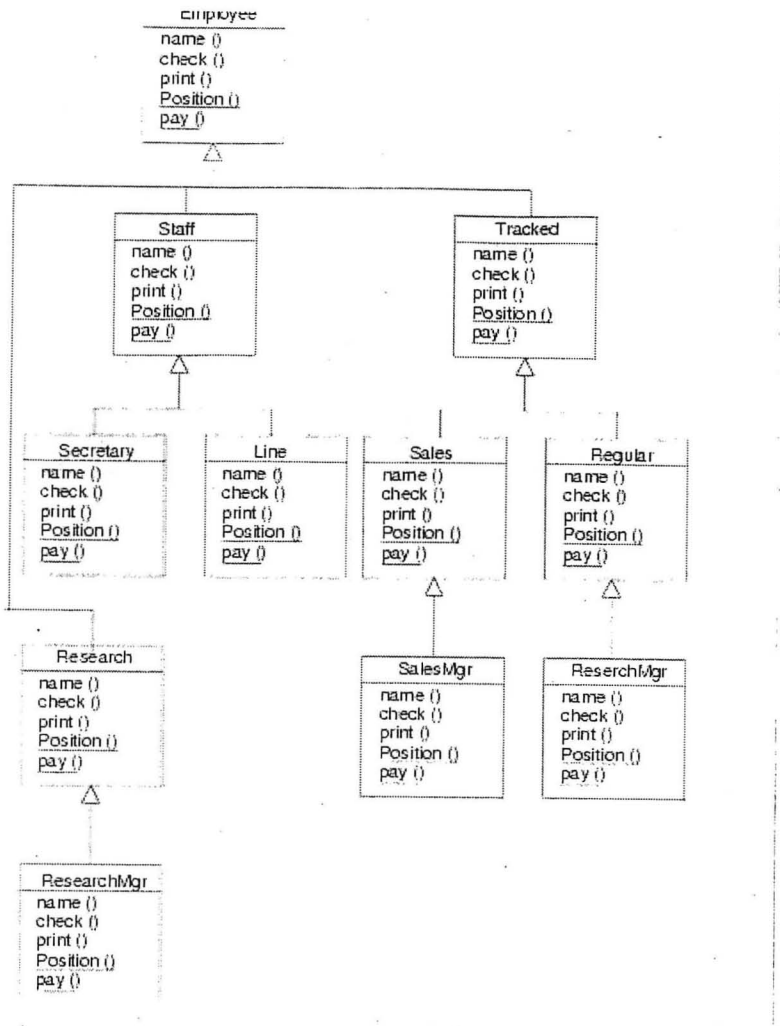


図6. ある会社の組織図

Hyper/Jでは，これらを図7[Ossher2001_1]のように分類することができる．コンサーン毎に次元を構成し，各次元の値はHypesliceを構成する．給与・人事情報コンサーンの給与情報の行で構成されるオブジェクト群を給与情報Hypersliceと呼び，もう一方の行は人事情報Hypersliceを表現している．これらの指定は以下のように行う．

```
package Presonel: Feature.Personnel
operation position: Feature.Payroll
operation pay: Deature.Payroll
```

HypermoduleはHypersliceの合成である．図7のように，分割したHypersliceの合成としてアプリケーションを記述すると，以下のようになる．

```
hypermodule PayrollPlusPersonnel
   hyperslices: Payrool, Personell;
   relationships: mergeByName;
end hypermodule
```

ここで，mergeByNameは合成にさいして，同じ名前のものは1つに統一する事を示す．

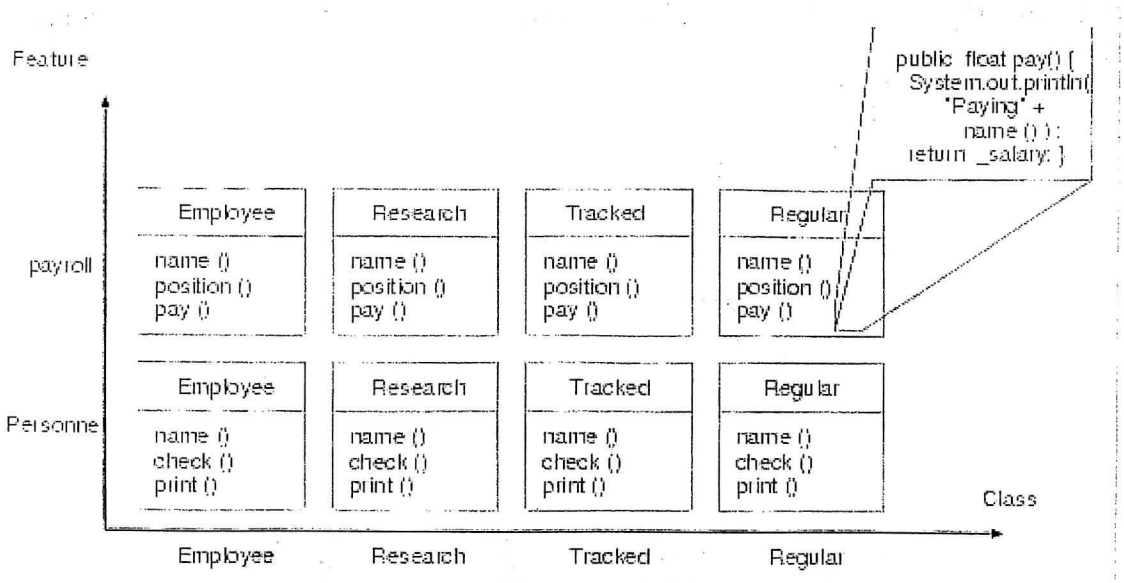


図7. Hyperspace}

4. おわりに

プログラミング技術がソフトウェア開発技術の主要な柱の1つであるとの認識に立ち, おもにプログラミング言語の視点からアスペクト指向開発技術について解説した. 上で用いた例からわかるように, 何をコンサーンと考え何をアスペクトにするかと言うことについては, 決まりや方法があるわけではない. "何をオブジェクトとすれば良い分析ができるか"と言う問の解が一意でないことと状況は同じであるか, より悪いと認識すべきであろう. 非機能特性を素直にモジュール化するためにアスペクト指向が有力な方法であることは, 広く認知されつつはあるが, 未だ実証された事実であるとは言いがたい. アスペクト指向技術がオブジェクト指向のあとの銀の弾丸になりうるかどうかは, "何をコンサーンと考え何をアスペクトにするか"との問に対する答を出すべく, 成功事例を積み重ねられるか否かにかかっている.

本稿では1節で触れたPOP技術のうち, アスペクト指向技術だけについて説明したが, その他の技術についても, いくつかは重要なものなので, 文献[Czarnecki2000]等で調査されることをお勧めする. また, 専用のプログラミング言語を使うのではなくパターンを用いてアスペクト指向実現を行う方法も提案されている[Constantinides2000].

参考文献

[Aksit2001] M. Aksit and L. Bergmans, "Guidelines For Identifying Obstacles When Composing Distributed Systems From Components", in M. Aksit, Ed., Software Architectures and Component Technology: The State of the Art in Research and Practice. Kluwer Academic Publishers, 2001.

[Aspect-Oriented Software Development] Aspect-Oriented Software Development, Web site; http://aosd.net./

[Bergmans2001] L. Bergmans and M. Aksit, "Composing Crosscutting Concerns Using Composition Filters", CACM, Vol. 44, pp. 51-57, Oct. 2001.

[Bollert99] K. Bollert, "On Weaving Aspects", in Proceedings of Aspect-Oriented Programming Workshop at European Conference on Object-Oriented Programming(ECOOP), Lisbon, Portugal, Jun. 1999.

[Czarnecki2000] K. Czarnecki and U. W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison Wesley, Boston, 2000.

[Constantinides2000] C. Constantinides, A. Bader, T. Elrad and M. Fayad "Designing an Aspect-Oriented Framework", Computing Surveys 32, 41, 2000.

[Elrad2001_1] T. Elrad, R. E. Filman, A. Bader, Eds., Special Issue on Aspect Oriented Programming, CACM, Vol. 44, 2001.

[Elrad2001_2] T. Elrad, R. E. Filman, A. Bader, Eds., "Aspect-Oriented Programming", CACM, Vol. 44, pp. 29-32, Oct. 2001.

[Elrad2001_3] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr and H. Ossher, "Discussing Aspects of AOP", CACM, Vol. 44, pp. 33-38, Oct. 2001.

[Fayad99] M. Fayad, D. Schmidt and R. Johnson, Eds. Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley, 1999.

[Gamma95] E. Gamma, R. Helm, R. Johnson J. Vissides, Design Patterns, Addison Wesley , 1995

[Harrison93] W. Harrison, and H. Ossher, "Subject-Oriented Programming(a critiqueof pure objects)", in Proceedings of the Comference on Object-Oriented Programming: Systems, Languages, and Applications(OOPSLA), Sep. 1993.

[Harrison2000] W. Harrison, H. Ossher, and P. Tarr, "Software Engineering Tools and Environments: A roadmap", in The Future of Software Engineering. A. Finkelstein, Ed., ACM, 2000.

[Kiczales97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J-M. Loingtier, J. Irwin,"Aspect-Oriented Programming", in Proceedings of the European Conference on Object-Oriented Programming(ECOOP), Springer-Verlag. Finland, 1997.

[Kiczales2001_1] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold,"An Overview of AspectJ", in Proceedings of the European Conference on Object-Oriented Programming(ECOOP), Springer-Verlag. Hungary, 2001.

[Kiczales2001_2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold "Getting Started With AspectJ", CACM, Vol. 44, pp. 59-65, Oct. 2001.

[Lieberherr96] K. J. Lieberherr, Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns, PWS Publishing Company, Boston, 1996; http://www.ccs.neu.edu/research/demeter.

[Lieberherr2001] K. Lieberherr, D. Orleans, and J. Ovlinger, "Aspect-Oriented Programming with Adaptive Methods", CACM, Vol. 44, pp. 39-41, Oct. 2001.

[Lippert2000] M. Lippert and C.V. Lopes "A Study on Exception Detection and Handling Using Aspect-Oriented Programming", in Proceedings of the 22nd International Conference on Software Engineering, Jun. 2000.

[Maes87] P. Maes, "Concepts and Experiments in Computational Reflection", in Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87), Orlando, FL, Oct. 1987.

[Ossher2001_1] H. Ossher and P. Tarr, "Multi-Dimensional Separation of Concerns and the Hyperspace approach", in Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2001.

[Ossher2001_2] H. Ossher and P. Tarr, "Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software", CACM, Vol. 44, pp. 43-50, Oct. 2001.

[Parnas72] D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules", CACM 15, 12, Dec. 1972.

[Smith84] B. Smith, "Reflection and Semantics in Lisp", in Conference Record of POPL '84: The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1984.

[Sullivan2001] G.T. Sullivan, "Aspect-Oriented Programming Using Reflection and Metaobject Protocols", CACM, Vol. 44, pp. 95-97, Oct. 2001.

[Tarr99] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, "N Degrees of Separation: Multi-Demensional Separation of Concerns", in Proceedings of the 21st International Conference on Software Engineering, May. 1999.

[AspectJ] The AspectJ Primer, http://aspectj.org/doc/primer/

[TRESE2002] TRESE, http://trese.cs.utwente.nl/composition_filters/

ゆうすふるユースケース

伊藤 昌夫

(ニルソフトウェア)

気がつくと，とてもたくさんのユースケースの本が出版されている．数えたことはないが両手では足りないのではと思う．私の日頃提唱している憧れ理論（人間は最も自分が不適な世界に憧れ，悲しいことにその苦手な世界を選択してしまう一別名カサブタ理論）によると，この現象もまた人の単純なるユースケースへの憧れかも知れぬ．私もまた少しだけ仲間入りをしたい．

ユースケースはその提唱者の定義に従えば，以下のとおりのものである．

> 「ユースケースは，システム中の，トランザクション列である．これにより，システムの個々のアクターに対して測定可能な価値を生み出す」

これだけなのである．システム境界を流れるイベントを意味ある括りでまとめたもの位である．しかし，見ていると人によって様々な思いとともに使われているようである．例えば，ユースケースを用いて，機能分割する場合などがそうである．結局は，どうしようが使う人の自由なのであるが，ここではちょっとヤセ我慢をしてみる．

まず何といっても，記法が良くない．どうもシステムのハコにユースケースを表す楕円を書くからいけないのである．定義を正とするならば，システムからたくさんの糸電話がでていて，他方には利用者がいるというのが良いように思う[1]．そうすると，間違ってもユースケースを使ってシステムを機能分割しようという気にはならなくなるであろう．

或いは，ユースケースの詳細をシーケンス図を使って書こうとも思わなくなるに違いない（そこではいきなり多種多様なオブジェクトが登場してしまう）．そんなハズはない．利用者にとって糸電話だけが唯一のモナド的口なのだから．

フルキヲタズネル．JSD（Jackson System Development）法に従うと，最初の実体／行為（action）段階において，行為と実体を見つける．

最初は，行為の定義である．

(1) 行為は，ある時点で発生する．
(2) 行為は，システム外の実世界で生じる．
(3) 行為は原子的であり，副行為には分割されない．

次に実体の定義である．

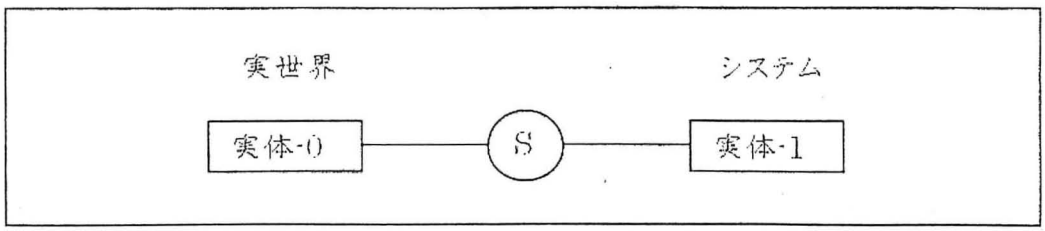
(1) 実体は，いつかは行為を行ったり，行為の影響を受ける．
(2) 実体は，実世界に存在しなくてはならない．システムの構成要素ではない．
(3) 実体は，個別のもので，かつ識別できなくてはならない．

先ず，興味深いのはJSDでは行為を優先する．行為の定義には，実体が含まれないことから明らかである．しかし，もちろん，実体を最初は考えないということではない．考えなくとも，そんなものは見れば分かる[2]．

さて，この行為は何かと考えると，実世界においては実体同士の関係ということができる（廣松渉氏のモノからコトへにあるとおり）．お金を貸すという行為は，貸すあなたからの見方である．当たり前であるが，借りる私からは，お金を借りることになる．これが関係性である．実体からは異なる見方になるが，行為としては明らかにひとつである．

さて，実体に関しては実体構造図を書く段階が続くのであるが，ここでは飛ばして，模型プロセス段階を見てみる．

図にすると次のようになる．



この図で，今例えば顧客という実体を考えたときに，-0と添え字がつけられたものは，実世界の「顧客」であり，-1と添え字がつけられたものは，システム内で対応するものである．システムの中とはいえ，必ず実世界に（対応する）実体が存在するので，定義(2)に反してはいない．Sはあるデータの流れを示している．

この図で示されるものがJSDでの模型（model）の意味だろうと思う．即ち，実体-1である．

ユースケースと上図は良い対応関係にある．実体-0をアクタとして，Sで示される部分がユースケースである．実体は行為に関連付けられている（実体の定義参照，それは今回飛ばした実体構造図によって図式化してある）．今実体Eが，行為 a-1, a-2, a-3 によって定義されているときに，Sは当然 a-1, a-2, a-3 をもれなく組み合わせて記述できることになる．

つまり，JSD視点に立つならば，行為→実体→模型プロセスという流れの中で，初めてユースケースが定まることになる．ユースケースを書くときに，悩むのだとしたら，これは一つの良いアドリアネの糸になる．

JSDは，広く普及はしなかったが（それは構造化分析／設計も同じ），実体の捉え方や今回見たような行為に対する考え方は，未だ凡百の手法の中で際立っている．しかし，残念なのは，JSDが決して手取り足取りの手法ではないことである．頭の良い人には，そうしようとする憧れがないからに違いない．

[1] 知るひとぞ知る(?) Catalysis ではその記法になる．
[2] そう私は日本にJSD を紹介された山崎利治さんから教えていただいたような記憶があります．もちろん，実体定義にある基準を満足しなくてはいけませんが．

情報システム開発にはエンジニアリングが必要

- IT systems have to be developed based on engineering processes -

松原 友夫
(松原コンサルティング)

昨年の秋に日本信頼性学会の主催で「みずほ銀行の事故から何を学ぶか ー 複雑化巨大化する社会システムと信頼性」というシンポジウムが開かれ，そこでの発表・討論の内容をめぐって，SEA-SPIN MLの上でも，活発なやりとりが行われました (SEAMAIL Vol.13, No.8, pp.24-31 参照).

このシンポジウムの最後のパネルで、司会の向殿政男先生(明治大学)が，パネリストに対して，「最も重要と思うことを１つ挙げて欲しい」と要望し，その後で，聴衆に同じ質問をされたのだそうです．そこでの松原さんの鋭い発言を聞いて，シンポジウムの幹事の先生から学会誌への原稿執筆依頼がありました．

以下に収録したのは，その依頼に応えて松原さんがお書きになったエッセイの改定版です．オリジナルは日本信頼性学会の機関誌「信頼性」(Vol.25, No.1, pp.14-21) に掲載されています (編集部)

1. 需要のないソフトウェアエンジニアリング

自動車を作るには自動車工学，ビルを建てるには建築工学は必須だ．しかし，ソフトウェアを作るのソフトウェアエンジニアリングが必要と思っている人は少ない．実際に，求人票に「ソフトウェアエンジニアリングの知識と実践経験が豊富」と書いた人がいる．それを見たリクルータは，開発者でなく研究者としての仕事を探すようにと助言した．ソフトウェア開発の仕事にありつくには，C++，Javaなどのプログラム言語のスキルと経験が豊富であることが重要な条件である．多くのソフトウェア会社は，若くて少々無理の利く体力があるコーダーを求めるが，ソフトウェアエンジニアリングの知識・経験には，ほとんど関心を示さない．

本屋のコンピュータ関係の本棚に並んでいるのは，ほとんどがハウ・ツーもので，かなり大きな本屋でも，ソフトウェアエンジニアリング関連の本を見つけるのは難しい．日本では，50万人のSEとプログラマがいると言われているのに，ソフトウェアエンジニアリング関連の本は，出版してもたかだか数千部しか売れない．売れないから出版されない．ソフトウェア構成管理関連の本を，ウェブで検索すると，原書は数十冊ヒットするが，翻訳されたものはごくわずかだ．見積もりモデルCOCOMOで知られるBarry Boehm のSoftware Economicsは欧米ではロングセラーの古典だが，訳本はない．

大学の情報学科では，むろんソフトウェアエンジニアリングを教えるが，企業側のリクルータは必ずしもそれを喜ばない．むしろ流行のプログラム言語を教えることを望む．情報学科の卒業生は，なぜかソフトウェア開発の仕事につきたがらない．大多数のソフトウェア会社は，開発にはエンジニアリングの知識が必要と考えていないので，文科系の学部から，あるいは商業高校からも人をかき集めてきた．ことほど左様に，我が国の情報産業では，ソフトウェアエンジニアリングが軽視されている．先進国はもちろん，インド，中国などの近隣諸国でさえ，大学教育と産業は直結している．大学での情報分野の教育が産業で活かされていないのは，我が国だけである．なにかが狂っている．

2. 日曜大工ではビルは建たない

我々は，ビルや交通システムや機械類に囲まれ，これらに依存して社会生活を送っている．もし頻繁に家が崩壊し，列車が衝突し，旅客機が墜落し，あるいはボイラーが爆発すれば，社会生活が脅かされる．長い産業の歴史の中で，我々は数多くのトラブルや災害から学び，危険を減らすための知識をエンジニアリングに蓄積し体系化してきた．製品やシステムは，エンジニアリングを学んだエンジニアによって設計・製造されるようにり，産業革命時代とは比較にならないほど，よいものを，安く速く強靱に作れるようになり，安全で効率的な社会生活を送れるようになった．

私はかつて機械工場とコンピュータ工場に在職したが，そこでは，すべてのプロセスは，工場工業規格の規定に従って行われる．製品に欠陥があると，原因追求のために実験を行い，結果に基づく対策をすぐに規格に反映させた．設計者は，設計図を書く前に規格の規定に沿って膨大な設計計算を行い，力学的な安全性を確かめる．作業者は，作業現場に手あかでよれよれになった規格書を拡げ，それを見ながら作業する．規格に従わなければ切削も溶接もメッキもできない．工業製品が信頼できるのは，エンジニアリングが支えているからだ．

ソフトウェア開発において，プログラムが書けるということは，単に文章が書ける，あるいは，鋸や金槌が使えるということに過ぎない．趣味の日曜大工で家を建てる人がいる．それが個人の家なら，たとえ倒壊したとしても，被害にあうのは彼の一家に止まるから，自らのリスクでそれをやるのもいいだろう．しかし，日曜大工ではマンションは建たないし，建ったとしても誰も入居したいと思わない．

3. 情報システムとシステムエンジニア

定義によれば，システムとは「ある相互作用又は相互依存性によって結合されたグループ又はサブシステムであって，多くの業務を実行するが，単一の単位として機能するもの（ANSI N45.2.10-1973）」である．システムのイメージは各人各様で，ある人はビジネスシステムを，ある人は，航空管制システムを，ある人は発電所を，ある人はオペレーティングシステム(OS)を思い浮かべるが，多くの要素が相互作用を行いながら共通の目的に奉仕すれば，それはシステムである．最近のシステムのほとんどは，何等かのかたちでソフトウェアを要素として含んでいる．

システムを設計する際には，設計者は契約やプロジェクトなどで明示された範囲はもちろん，通常その範囲を越えて，隣接するシステムや利害関係者への影響を考慮する．航空管制システムのような大規模なものなら，機械，電気，電子，ソフトウェア，アプリケーションなどの複数の専門分野にまたがる知識と経験に裏打ちされた高い能力が不可欠である．これから伸びる技術であるシステムLSI（システム・オン・チップ）の開発は，アプリケーションの種別ごとに，専門の異なる技術者からなる数人のチームが互いに協調しながら設計から製造までを一貫して行う．このため，各メンバーの専門的な知識・経験をオーガナイズする能力を持つシステムエンジニアが必須で，その養成は急務である．開発だけでなく，システムの事故原因解析も，システムエンジニアは不可欠である．

多次元的な視野を持ち，システム思考ができる優れたシステムエンジニア（日本の業界で使われるSEとは異なる）は，システムエンジニアリングの知識だけでは不十分で，重要なのはいくつかの異なる分野の経験である．従って，座学による即成教育では育成は困難である．昨今のリストラの流行で，企業が逸材に異なるキャリアーパスを準備する余裕がなくなっているので，システム設計のの能力を持つ人材は，今後ますます枯渇するだろう．

4. 情報システムは膨張し複雑化し劣化する

システムは，それがどんなに大規模で複雑なものでも，環境変化の要請で，ある日突然より大きなシステムにサブシステムとして組み込まれてしまうことがよくある．銀行や航空会社のシステム統合などは典型的な例である．このサブシステム結合のグルー（膠）の役割を果たすのが，ネットワークでありソフトウェアである．サブシステムを統合して膨張する巨大システムは，それ自体しばしば通常の人間の理解の範囲をこえる．

巨大化するだけでなく，システムは本質的に時間とともに変化する．M. M. Lehmanは，彼の古典的な名著，プログラムの進化[1]で，プログラムをS-タイプ，P-タイプ，E-タイプに分類し，その各々の進化の特性を説いた．S-タイプは作る前に仕様化が可能で仕様通り作れば済むもの，E-タイプは動作環境の中に埋込まれるもの（Eはエンベッデッドの略だが，ここでは機器への組込みを指しているのではなく，社会やビジネス環境への組込みを意味する）である．P-タイプは原理が与えられるがインプリメントに多数の選択肢がある例外的なもので，ここでは触れない．

E-タイププログラムは，本質的に常に進化（変化）する動作環境に埋込まれるので，仕様を決めた

瞬間から変更が求められ，変更をくり返すに従ってプログラム構造が劣化する．みずほ銀行システムのトラブルの遠因には，統合の母体である旧三行のシステムが，恐らく長期の手直しを経て巨大化し劣化していたことと，統合プロジェクトが旧三行間の密接な協調なしに行われたことが挙げられよう．

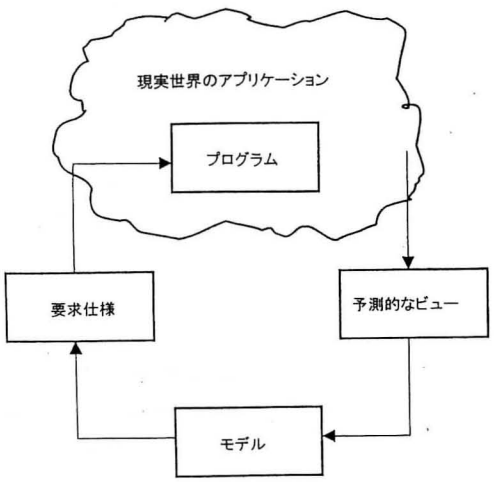


図-1a: E-タイプのプログラム

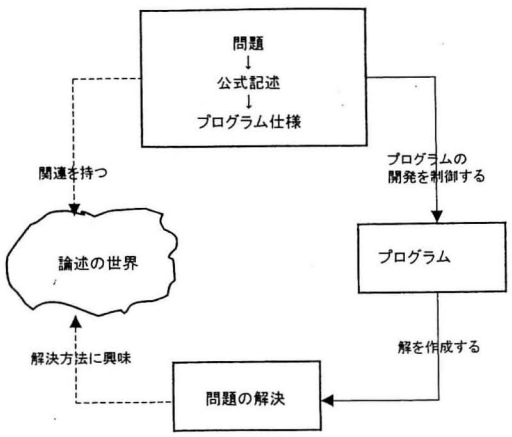


図-1b: S-タイプのプログラム

かつてはS-タイプの代表は組込みソフトウェアであった．しかし，携帯電話のように，エンドユーザの使用環境に直接かかわる機能の多くを分担する製品の組込みソフトウェアは，常に変化が求められる，もっとも扱いにくいE-タイプに変貌した．劣化を防ぐためにも，ソフトウェアエンジニアリングは，ソフトウェアアーキテクチャー，情報隠ぺいなど，さまざまな手法を用意している．

絶え間のない仕様変更のために，製品開発費に占めるソフトウェア開発コストの割合いが増大し，多くの製品でハードウェアコスト越えた．製造メーカは，ソフトウェアをうまくマネージできなければ，生産ビジネスを成功させるこが困難なことに気づきはじめた．ここ数年，ソフトウェアプロセスの改善に，危機感をもって積極的に取り組んでいるのは，製造メーカの組込みソフトウェアグループである．

5. クリティカルな情報システムの事故

みずほ銀行システムの事故で，世の人は，クリティカルな情報システムに一旦事故が起こった場合の影響の大きさをあらためて認識しただろう．Business Weekが報じるところによると，世界の先進国でほぼ毎月1件，新聞に報道される重大なトラブルが起こっているという[2]．ソフトウェアはディジタルな論理の記述であるから，些細な欠陥が重大な結果をもたらすことがある． 1990年1月に起きたニューヨークの電話網の9時間停止事故[3]の原因は，ハードウェアの部分故障を知らせる数個のフラグビットを，故障回復後にリセットしなかったために起こった．故障を知らせるエラーメッセージが周辺の電子交換機間で輻湊し，システムを止めた．また，1979年に米国で，原子炉の設計計算プログラムの安全係数の符号（１ビット）誤りが見つかり，同型の5基の原子炉が総点検のために停止命令を受けた．

訓練を受けていないコーダが起こした痛ましい事故がある．コンピュータで制御されたTherac-25放射線治療器が，1985年から1987年にわたって，ソフトウェアバグに起因する過剰被曝(1 5,000-20,000 rad，正常の治療レベルは200 rad，危険レベルは1,000 rad)により，数人の患者を死にいたらしめた．治療器は，ハードウェアの誤動作時に，メータに検知されないまま放射線を照射し続けた．原因の核心は単純なコードエラーで，不適切なソフトウェア設計とシステム構造設計が問題を隠した．問題のソフトウェアは，潜在バグを内在する前機種Therac-20からの流用であった．Therac-20ではハードウェアインターロックが装備されていたためにバグは顕在化しなかったが，Therac-25に流用した際，ソフトに同じ機能があるという理由でハードウェアのインターロック装置を廃止したため，バグが顕在化した．バグを作り込んだコーダはとうの昔に退職し，ドキュメントは存在しなかった．原因をマ

イクロスイッチの誤動作と思い込んだことから対策が遅れ，長期間患者の命を危険に曝し続けた．

事故の詳細な報告は，教訓を学ぶための最良の教材である．ワシントン大学のNancy Levesonは，Therac-25の詳細な事故調査報告[4]をまとめたが，その中で，「基本的な誤りは，ソフトウェアエンジニアリングの欠如と装置の安全をソフトウェアに依存したことである」と述べている．情報システムの事故事例と回避策についての名著Safeware[5]の冒頭で，Nancyは「７つのソフトウェア神話」というかたちで情報システムの事故からの教訓をまとめている．その一つ，

Myth 3. コンピュータは，それが置き換えた装置よりはるかに高い信頼性がある

はTherac事故からの教訓で，ソフトウェアには取りきれないバグを含んでいて，かえって信頼性が低い場合があることを，ソフトウェアの特性を知らないハードウェア出身のシステム設計者に警告したものである．

こうした情報システム事故の詳細報告が存在するのはほとんど欧米で，我が国では調査自体が行われないか，行われても報告書は滅多に公開されない．それは恐らく，日本の組織の根の深い隠ぺい体質と，事故調査における検察的態度が原因であろう．銀行システムのようなクリティカルな情報システムの事故は，事故から教訓を学び再発を防止するために，公けの事故原因調査委員会を設けて事故原因を詳細に調査し，調査報告を公開すべきである．

安全性に関わる情報システムの開発をより確実にするために，例えば英国では，形式的記述の適用に関する規格があり，強く推奨されているが，日本では形式的記述を使いこなす技術者は極めて少ない．

6. 情報システムにおけるフライトレコーダ

システム要素の相互作用は複雑で多様なので，万全を期したつもりでも完全とは言いきれない．事故から学び再発に役立てるために，航空機や鉄道システムには，障害が発生した瞬間に何が起こったかを詳細に記録する仕組みが組み込まれている．例えば，旅客機が墜落して不幸にして全員が死亡しても，フライトレコーダやボイスレコーダに残る記録から，かなりの程度事故原因が推定できる．これは，安全工学の成果である．

この仕組みは，複雑な情報システムでこそ必要なのに，あまり採用されていない．それどころか，権利を守るという理由でソフトウェア内部の振る舞いを秘匿しようとする傾向さえある．情報システムは，揮発性の信号を扱うことが多いので，事故に関わる可能性のある情報を記録し蓄積する機能は，あらかじめ組み込んでおくべきである．とくにクリティカルシステムでは，原因追跡可能な機能の組み込みを義務づける必要がある．それなしには，たとえ事故原因調査委員会を設置したとしても，原因の特定は極めて困難である．

7. エンジニアリングに基づく開発を

みずほ銀行の事故は，2002年4月1日前後，連日ジャーナリズムを騒がせた．大方の指摘は，トップのリーダーシップ不足，CIO不在，現場任せ，不毛の機能比較，拙劣なプロジェクト管理，などであった．たしかにこれらはもっとも重要だ．しかし，それと同じくらい重要なのは，みずほ銀行のシステム部門，及び統合プロジェクトチーム（と言えるようなものが三行間にあったかどうか）に，これほど責任が重い開発を，開発の基礎体力とも言うべきエンジニアリングプロセスなしでやったことである．むろん，これはチームの責任ではない．それが重要であることを理解せず，プロセス改善にコミットメントを与えなかったトップの責任である．

ソフトウェアが無形物であるところから，有形物のように物理原則に支配されない．ロジックが通りさえすれば，ルールを守らなくても動いてしまう．だが，動いてはいても，配線が蜘蛛の巣のようになっていて，ちょっといじればショートするような代物かも知れない．スパゲティー状のコード，コメントなしのコード，コードと一致しない設計文書，いいかげんな構成管理，洩れの多いテスト，

といった状態かもしれない．もしそうなら，こうした基礎的なプロセスを改善しない限り，トラブルは必ずくり返す．

幸い，世界中のどこかで，日常遭遇する問題領域の解決策が準備されている．それを知って応用する能力があるかないかが，状況を大きく左右する．もし，みずほ銀行の開発チームの責任者が，リスク管理を用いて統合のリスクを金額でトップに示していたら，判断は違っただろう．リスク被曝度(risk exposure)は，生起確率と，起こった場合のインパクト，すなわち推定損害額の積である．システム統合の場合，確率も推定損害額も極めて高いので，億単位の数字が示せたはずだ．恐らくトップは，リスクの程度を定性的にしか理解していなかったのではないか．だからこそ，4月1日の稼動開始にGoを出したのだろう．

プロジェクトが混乱しても，最低限やるべきことを怠ると混乱が増幅する．収束させるためには，プロジェクトマネージャは秩序を保つ部分を確保し，冷静に状況を数字で把握し，着実に作業イベントを消し込んでいかねばならない．プログラムを変更したら必ず回帰テストが必要だが，デグレードが発生していることから見る限り，それがおろそかになっていたのだろう．最低限の構成管理を実施し，だれがいつ何をやったか，ドキュメントやコードモジュールは，いつどういう状態であったかを追跡できるメカニズムを確立しておかねばならないし，テストは，高度なテスティング技術を使いこなせるテスト技術者が，十分時間をかけて実施せねばならないが，混乱の状況から見ると，それをやったとは思えない．こうしたエンジニアリングプロセスが確実に行われていなかったために，対策が後手にまわり，収束が長引いたのではないか．

みずほの事故では，プロジェクト開始の遅れが指摘されている．しかし，プロセスの側面から見ると，はたしてあらかじめしっかりした戦略と方針が示され，確度の高い見積りがなされ，それに基づいて現実的な計画が立てられていたかどうか．そもそも，見積もりプロセスが未熟であったために安易な日程がまかり通ったのではないか．

情報システムにはハードウェア集約的なシステムには見られない難しさがある．かつてのハードウェア集約的なシステムでやってきたプロダクトに注目したエンジニアリングに加え，プロセスに注目したエンジニアリングを適用しなければ，確実な情報システムの開発・保守はできない．ソフトウェアを改善するには，ソフトウェアエンジニアリングに基づくプロセスを，制度化して実施する必要がある．このために，ベストプラクティスを体系化したプロセス成熟度モデルに基づく，組織を挙げた成熟度レベルの向上が重視される．日本にも，ソフトウェアプロセス改善を熱心に推進しているJASPICやSPINのようないくつかのグループがあるが，なぜか銀行のシステム部門からの参加者はほとんどいない．

現実の大規模システムの開発チームは多重下請けの階層構造であり，下請け要員の多くはソフトウェアエンジニアリングの教育を受けていないコーダである．みずほ銀行の場合，銀行側の開発者でさえ，エンジニアリングとはほど遠い仕事をしていたのではなかろうか．エンジニアリングプロセスが未熟なのはみずほ銀行だけではない．多くの情報システムの開発・保守がこのような実態がこのようだとすれば，極めて憂慮すべきことである．

8. 危険なシステムには金と手間をかけよ

リスクの程度はシステム固有の条件と埋込まれた社会環境によって異なる．リスクは，システムが計画された段階で，障害が起こり得る確率と起こった場合の推定被害額からあらかじめランクづけできる．このランクに基づいて，システム障害が深刻な災害をもたらし得る原子炉，航空機，鉄道などは，安全を最優先にして設計するのが常識である．だが，情報システムではこの常識さえ確立していない．銀行システムも企業の一般会計システムも，同じレベルのスキルを持つ要員で，同じ見積り基準で設計・開発・保守されるのが普通である．

ソフトウェア集約的なシステムのリスクは，もっとも高いものにTherac-25のように人命に関わるもの，その次に銀行システムのように人命は損なわないが巨額な社会的経済的損失をもたらしうるものがあり，低いものには特定企業の損失にとどまる社内システム，個人が使うアプリケーションなどが

ある．クリティカルなシステムやソフトウェアの完全性に関連する国際規格を審議するISO/ IEC JTC1/SC7/WG9は，システムの要件段階で，要求すべき完全性を５つのレベルでランクづけする国際規格， ISO/IEC 15026: System and software integrity levelsを作成した[6]．この規格は，JIS X 0134: システム及びソフトウェアに課せられたリスク抑制の完全性水準，という名称でJIS化されている[7]．この規格の目的は，求められる完全性水準(SIL)に対して適切なエンジニアリングプロセス，及び厳格さの程度を選択することにある．こうした努力はまだ始まったばかりであり，この規格を土台にして，今後使いやすいガイドライン群を作成する計画がある．

高いIL水準のシステムに対して適用すべきプロセスには，独立テストチームによる厳格なテスト，Nバージョン開発，といった比較的簡単なものから，フォーマルな仕様記述，fault tolerantな基本構造の採用，クリーンルーム開発方法論などの高度な技術を用いるものまで数多い．恐らくこれらの厳格なプロセスを実施するには工数とコストが増加するが，課せられた責任の重いシステムには，それを許容する産業慣行を確立せねばならない．

9. おわりに

情報システムがソフトウェアエンジニアリングが希薄な状態で開発・保守されている状況を生んだ主な原因は，我が国固有の情報産業の構造にある．政府の情報システムの約80%は，大手の10企業グループが受注し，それを中小のソフトウェアハウスに4重5重の下請けに出している．元請けは，最下層でどこの誰が作業しているかを把握していない．

下請けの形態は，実質的に，成果に対してではなく作業時間に対して支払う人員派遣契約が多くを占める（契約は請負でも，実態は派遣であることが多い）．この契約の下では，エンジニアリングを学び，プロセスを改善し，品質や生産性を上げるインセンティブは何もない．それどころか，バグを作り込みそれを修正する作業はすべて支払いの対象になるから，トラブルがあれば儲かり，品質や生産性を上げると損をする．こうした契約慣行の下でクリティカルなものを含む情報システムが開発・保守されているのである．そもそも，高度な頭脳労働者が，肉体労働者なみに扱われていることに問題がある．

なぜこのような産業構造になってしまったのか？考えられる原因は，中小ソフトウェア企業の低リスク低マージン指向の経営と体力不足，元請け企業の開発に耐える発注仕様を記述できない未熟な調達プロセス，業界の契約慣行改善意識の欠如，，政府の数だけの質を問わないプログラマ育成策，それに，個人レベルではプロ意識の欠如などが挙げられよう．一方，アジアの開発途上国は，インドを手本にしてソフトウェアを主要な輸出産業に育てようとしている．すでに，インドのいくつかの有力企業は日本に進出して成功している．アジア諸国のソフトウェア技術者は，大学でエンジニアリング教育を受けてストレートにソフトウェア企業に入ってきたエリートである．いまのような産業構造の下で質と効率の悪い開発をしていると，気がついたらビジネスの多くをアジア勢に奪われることになりかねない．そうなってからでは遅い．

どうしたらこの事態を改善できるかは，稿を改めねばならないが，少なくともILの高いクリティカルシステムだけでも，しっかりしたエンジニアリングプロセスの下で，開発するのが当たり前になることを，情報システムと共存せざるをえない一市民として強く望みたい．

References

1. M.M. Lehman, L.A. Belady: Program Evolution, Academic Press, 1985.
2. Software Hell, Business Week, 1999. 12. 6 .
3. Vulnerability exposed in AT&T's 9-hour glitch, The Institute, 1990 Vol.13, No. 3.
4. Nancy G. Leveson, Clark Turner, An investigation of the Therac-25 accidents, IEEE Computer, 1993.
5. Nancy G. Leveson, Safeware, Addison Wesley, 1995.
6. ISO/IEC 15026: 1998 Information technology -- System and software integrity levels
7. JIS X 0134: 1999 システム及びソフトウェアに課せられたリスク抑制の完全性水準

＊＊わたしの本棚から＊＊

山崎利治

表題の「わたしの本棚から」というのは嘘です．狭いアパートに住んで本棚などなく，出入口の下駄箱の横に積んでいるだけです．昔そんな見出しの欄に，こんな本が面白いよと仲間に紹介記事を書いたことを思い出したからにすぎません．

荒木，張両先生による「プログラム仕様記述論」が上梓され，熊谷さんがseamailにその紹介をしてくださいます．それで，わたしも本の紹介記事が書きたくなったのです．両先生の本はまず表題が画期的です．短い頁の実務的感覚に溢れた教科書です．新宿の新しいほうの紀伊国屋で平積みになっているのをみて嬉しくなりました．というのはオブジェクト指向やUMLの本ばかりが目に付くからです．最近わたしはオブジェクト指向から脱却してどんな系でも並行プログラミングに移行すべきだと考えるようになりました．それは一般のオブジェクト指向の解説書が並行プログラムを陽には扱っていないからです．

ここで紹介したい本は二冊あります．いずれも主題はOOADでもUMLでもありません．その一冊は新刊ではありませんがつぎです．

はじめの本は

Jeff Magee and Jeff Kramer, *Concurrency – State Models and Java Programs.* John Wiley, 1999. pp. x+355.

並行プログラムについてきっちり学び，応答系 (reactive system) などの開発に役立てようという本です．応答系や実時間系 (real-time system) は並行プログラムとして実現されることが多く，並行プログラムの作成は難しいものと思われていました．難しいがきっちりと作る手立てがあるというわけです．想定している読者には学生の他，無論，職業的ソフトエア技術者が入っています．とくに，組込み系開発者に一読をお薦めしたい本です．また，オブジェクト指向分析設計やUMLなどのコンサルタントやインストラクターのみなさまもOOADやUMLから一旦離れて，並行プログラムの作成方法を振り返ってみるとそれらに対して新しく気づくことがあると思います．

本の狙い

まず，本の狙いは著者たちによればつぎのようです．

1. 並行プログラムにとって大切な原理を系統的に学ぶこと．
2. プログラム作成のための厳密な方法を考え，開発対象の記述法と，その模擬実行や検証を支援するプログラムの利用方法を学ぶこと．
3. 開発対象の Java による実現を考え，具体例によって説明すること．

それでこの目論見は十分に達成されています．わたしは大いに啓発されました．本の内容に触れる前に，FSP と Darwin について一言しておいたほうがいいようです．いずれも Imperial College(London) の Distributed Software Engineering Group で開発されてきたものです．

FSPによる並行プロセス

FSP(有限状態プロセス) はプロセス記述言語です．この本で並行プログラムを議論するための媒体は「プロセス」と呼ぶ抽象実体です．プロセスは実は並行プログラムの模型としてよく利用する札付き推移系 (LTS, Labeled Transition System) です．LTS は多重有向グラフで，頂点が状態を，辺が作用（action, 事象）を表すものです．そして，頂点の一つが初期状態になっています．初期状態から作用が生起するとそれが表す辺に沿って状態が推移すると考えます．FSP では，構文概念としての状態はありませんが，状態集合には「誤状態」(π) という特殊な状態が，作用集合には「観察不能作用（内部作用）」(τ) という作用があるとします．なお，FSP プロセスは抽象実体ではありますが，支援系を用意していて，プロセスの模擬実行や系のもつべき性質に対して検証ができるようになっています．

FSP の (抽象) 構文の一部はつぎのようです．

$$
\begin{aligned}
P &::= \mathbf{STOP} \mid \mathbf{ERROR} \mid (a \rightarrow P) \\
&\quad \mid (a_1 \rightarrow P_1 | a_2 \rightarrow P_2) \mid X \mid rec\ X = P \\
Q &::= P \mid (Q_1 \parallel Q_2) \mid Q/R \mid Q \setminus A \mid Q@A \mid Y \\
p &::= \mathbf{property}\ P\ =\ E \mid \mathbf{progress}\ A
\end{aligned}
$$

P は逐次プロセス，Q は並行プロセスを含む一般 FSP プロセス，p はあとで説明します．a, a_1, a_2 は作用です．

STOP は一状態でそれからの変化が観察できないプロセス，ERROR は誤状態から変化しないプロセスで特別の目的をもっています．

$(a \rightarrow P)$ は作用 a の生起のあと P となるプロセス，$(a_1 \rightarrow P_1 | a_2 \rightarrow P_2)$ は選択プロセスで a_1 の生起のあとは P_1,a_2 の生起のあとは P_2 となるプロセスです．LTS で考えれば和 LTS です．プロセス変数 X もプロセスです．$rec\ X = P$ は

プロセスの再帰的定義です．この定義は，じつは，$P = P1, \cdots, Pn = a \rightarrow P.$ のような書き方をします．

$(Q_1 \parallel Q_2)$ はプロセス Q_1 と Q_2 を同時に実行するという並行プロセスです．LTS では積になります．

Q/R はプロセスの作用の名前換え (relabeling) で，R は $\{new/old, \ldots\}$ の形をしていてプロセス Q の作用名のうち old を new に,... 置換えたプロセスを表します．$Q \setminus A$ はプロセスの名前を見えなくする制限 (restriction) で，A は $\{a_1, a_2, \ldots\}$ の形をしていてプロセス Q の作用 $a_1, a_2, \ldots$ を見えなくしたプロセスを表します．$Q@A$ はプロセス Q の作用のうち A に属する作用だけを見えるとしたプロセス，すなわち，A を界面とするプロセスを表します．

プロセス構成演算は上で述べた以外にも実際の課題が書きやすいように豊富に用意されています．FSP は結果として Hoare の CSP と Milner の CCS との混合になっています．プロセスは LTS でグラフ (推移図) として描けますから本のなかでは図を多用します．

つぎは HSP プロセスの一例です．Max 個の投票箱がある選挙の投票所に Max 人の投票者が来て投票する様子を記述したものです．

```
投票者 = (入場 → 投票 → 退場 → 投票者)@{ 入場, 退場 }.
const Max = 3;
range Int = 0..Max;
投票箱 (N = Max) = 投票箱 [N],
投票箱 [v : Int] = (when (v > 0)   入場 → 投票箱 [v − 1]
                  | when (v < Max) 退場 → 投票箱 [v + 1]),
|| 投票所 = (人 [1..Max] : 投票者 || 渋谷 : 投票箱 (1))
        /{ 人 [1..Max].入場/渋谷.入場, 人 [1..Max].退場/渋谷.退場 }.
```

上の FSP 構文を拡張して使っています．ふた付き作用，引数をもつプロセス，指数つきのプロセスなどです．(**when** b $a \rightarrow P$) は述語 b が真であるときに限り作用 a が生起できるというものです．$A = ([i : 0..2] \rightarrow o[i] \rightarrow A)$ は $A = (i[0] \rightarrow o[0] \rightarrow A \,|\, i[1] \rightarrow o[1] \rightarrow A \,|\, i[2] \rightarrow o[2] \rightarrow A)$ を意味します．これは $A(N = 2) = (i[j : 0..N] \rightarrow o[j] \rightarrow A)$ と書いても同じです．|| 投票所は並行合成プロセスであることを示す FSP の約束です．

方式記述言語 Darwin

巨大で複雑な系では論理的，機能的，算法的な構造の他に，実現上の構造を考えたいとする要求があります．成分 (component) や枠組み (framework)，あるいは，方式 (architecture) などというものがそれです．方式記述言語 (ADL) は「成分」(component) とそれらの間の相互作用，および，成分の合成を記述するための言語です．目的は巨大な系の開発のために必要な抽象（検証用）

と洗練の機構（成分合成，あるいは，成分分解）を求めたもので，成分の再利用，系の再構成などにも便宜を図るためのものです．成分とはプロセスやオブジェクトなど仕様単位，または，プログラム単位のことですが，他の成分との相互作用を明示することに特長があります．ここでいう相互作用とは共有作用や通信の送受や手続き呼び出しなどのことです．

Darwin は ADL の一つで 1991 年以来さまざまな版があります．Darwin は方式記述と成分記述 (C++) から実行可能コードを生成します．

一般に成分はプログラムの一部分なわけですが，その成分と他の成分との相互作用が問題になります．そこで成分をプロセスであるとして，その相互作用に関わる作用を参照する名前を成分の界面とします．この界面が成分が提供し (provide)，あるいは，要求する (require) 機能の窓口（端子）になります．成分による成分や系の構成は，プロセスの並行合成として行います．そこで成分の宣言を具体化し (instantiate), 界面同志を束縛結合する (bind) して辻褄をあわせます．

Darwin ではまたつぎのような図を用います．成分を矩形で表し，成分の界面をその辺に小さな丸で表します．要求側を白丸，提供側を黒く塗りつぶします．成分間で共有する作用があればそれらの小さな丸を線分で結びます．このとき名替えがあればそれを記します． つぎはさきの投票所の例を Darwin で記述したものです．

```
component 投票者 {
    require 入場; 退場; }
component 投票箱 (N = Max){
    provide 入場; 退場; }
component 投票所 {
    inst 人 [1..3] : 投票者;
    inst 渋谷 : 投票箱;
    bind 人 [1..3].入場 − −渋谷.入場;
    bind 人 [1..3].退場 − −渋谷.退場; }
```

本の内容

著者たちは放射線医療機器 Therac-25 のソフトウエアの誤りが人身事故を起こした例から話をはじめます．

まず，並行プログラムについて議論するために FSP の解説をします．系の動的な側面，つまり，挙動は推移図によって見やすくなりますが，系実現の指針になる静的な構造はプロセスではよく見えません．そこで，Darwin の構造図を借用します．つまり，構造図は FSP の並行合成, 制限, 名替えなどの静的演算の結果を視覚化します．ただ見やすいというだけではなく，構造図に

よって成分や方式の意味が明確になっています.

さて，FSP を用いた系の模擬実行を行う支援系 LTSA(札付き推移系分析系，Labeled Transition System Analyzer) が用意してあり，すべての例題についてその挙動を直観的に見ることができます (本にプログラムを載せた CD が添付してある).

成分，すなわち，並行プロセスは Java によって実現します．そこでプロセスはスレッドやモニターになります．モニターなどの Java のもつ並行プロセスのための機構もプロセスとして整理・解説します．そこで共有資源と相互排除，モニターと条件同期などが丁寧に解説されます.

つぎに,「竦み」(deadlock) を話題にします．有名な食事する賢人問題が例題になります．竦みは LTS でいえば，それから推移しない状態，つまり，STOP があり，そこへある作用によって推移して起こります．したがって，プロセスを LTS として, そのうえで STOP を探索すれば竦みの存在がわかります．さらに，竦みを起こす作用列もわかります．LTSA がこれを実行してくれます．これは，所謂，模型検査 (model checking) です.

そこでつぎに系の仕様記述として大切な安全性と活性を話題にします．安全性 (safety property) は竦みは起こらないなどの「悪いことは決して起こらない」性質です．この安全性をプログラムが決して悪い状態には達しない性質であると考え直します.

さて，安全性は合成的な性質です．つまり，もし部分系において安全性を保てば，この部分系を並行合成した系においてもその安全性を保ちます．そこで FSP では **property** $P = E$ を用意しています．これはつぎのような決定性プロセス P を定義するもので性質プロセスといいます．P は E で定義されたプロセスの状態に誤状態を追加し，すべての E の状態から E に出現する作用に対して推移が起こるようにしたプロセスです．ただし，E で定義した推移はそのまま，定義されていないものはすべて誤状態に推移するというものです．検討する系プロセスの安全性はこの性質プロセスと並行合成すれば確かめられるわけです.

活性 (liveness) は「良いことはやがて起こる」性質です．これも安全性と同様によい状態にやがて到達する性質であると考えます．ここでは進行性 (progress property) について議論します．a_i を作用として，**progress** $\{a_1, a_2, \ldots, a_n\}$ の形を用います．その意味は系の無限実行列には a_i のいずれかが無限回しばしば (infinitely often) 出現するというものです．つまり，公平性の要請です.

以上が本の 1 章から 7 章までの内容で並行プロセスについての基礎編です．つぎの第 8 章は並行系のひとつの設計法についてまとめて述べています.

模型に基づいた開発

系の開発における「模型」の役割をまず振り返ります．模型とは対象を抽象したもののことです，もっと技術的な術語としての「模型」もあり，事実，それと一致します．この本での模型は FSP によって記述されたもの，つまり，プロセスです．そこでは，関心を作用とその相互作用に集中させました．データ構造や作用によるデータの変化などは捨象しています．しかし，作用と相互作用を考えたことが系の挙動を明確にし，安全性や進行性の検討に繋がります．系に対する要求から模型を作成し，この模型によって系に要請する諸性質を検証し，正しい実現を図ることができるわけです．これが模型のもつ意義です．

設計は要員の経験，開発対象の種類などによってさまざまであるから「方法」としては述べないと断っているものの，立派な方法の提示になっています．オブジェクト指向や UML が全く顔をみせないというわけではありません．抽象，情報の隠蔽，合成単位などの考えはオブジェクト指向の原則であるといい，Java による実現のために UML のクラス図を使います．しかし，ここでは，Java 用語としてのオブジェクトはあるものの，はっきりプロセスといい，プロセスによってプログラムを作ろうとしています．つまり，クラス図は実現のためのものにすぎません．簡潔で明晰な設計の流儀は当世風だと思います．著者たちはこれを模型に基づいた開発 (model-based approach) と呼んでいます．

そのあらましはつぎのようです．

1. 系の主要な作用と相互作用（共有作用，プロセス間交信）の同定
2. 系の主要なプロセスの同定
3. 系の主要な安全性，進行性の同定
4. プロセス構造図を描く

本は車の巡航制御プログラムを例に FSP 模型を作成します．この模型は安全性と進行性が組み入れられています．したがって模擬実行を行い，模型検査ができます．これは LTSA が支援します．ここで模型検査について補足しておきます．「模型検査」とは，「時間論理式」Ｓと「時間枠」Ｍが与えられたとき，ＭがＳの「模型」になるかを判定することをいいます．説明のない用語の羅列で恐縮ですが，系の挙動が時間論理式で記述でき，また，系の抽象実現が時間枠で記述できます．また，模型とは論理式を真にするひとつの解釈構造である時間枠のことです．つまり，Ｓを仕様，Ｍを実現と考えることができ，ＭがＳの模型であることはＭがＳを満たす，すなわち，ＭはＳの正しい実現になっていることを示すわけです．そして，この判定問題が解けてプログラムできます．模型検査は演繹的ではない完全に自動的な正当性証明になっています．ＭがＳの模型ではないとき反例が具体的にえられることも利点です．通信プロトコルや順序回路の検証に実績があり，最近は事務系に

も適用が図られているようです.

つぎは実現ですが，構造図を描き「方式」を明確にして，Javaによるコードを書きます．それは7章までに学んだお決まりの作業ですが，具体的には，

1. スレッドとして実現を図る主要な能動的なプロセスを同定する (作用を発生するプロセスを能動的という)
2. モニターとして実現を図る (共有する) 受動的なプロセスを同定する
3. 環境との交信 (状態の表示など) は関連クラスとして実現を図る
4. クラス図を描く

というものです.

多くのOOAD(UML)では，系の模型はステートチャートか活動図 (ペトリ網とみなして) によって書き，性質（仕様）は協調図や作用系列図の相互作用図によって書いて模型検査を間接的に行うことになります．模型に基づいた開発では，propertyやprogressを明確に記述して，さらに，支援系によって模型検査を完全に自動的に行います．この利点は貴重です．また，なによりも模型検査によって開発に厳密さが加わっています.

なお，クラス図は実現段階でJavaプログラムのために描くにすぎません．目的はクラスの継承関係を示すことです．多重継承を許さないJavaのためインタフェースによる実現などを明示する意図もあります．クラス図がもつデータベース設計の比重は，開発対象が応答系であることから相対的に低くなっています（データベース設計にはクラス図よりも拡張ER図を用いたほうがいいと思います）.

進んだ話題

この本は第9章以降の4章で「進んだ話題」を取り上げます．プロセスが実行中に生成消滅するような場合 (9章)，プロセス間通信の様式，つまり，同期通信, 非同期通信, 逢引方式 (rendez-vous)(10章)，実時間の扱い (12章) などです．第11章は並行系のいくつかの方式について説明します．濾過送管網 (filter-pipeline) 方式，親方子方 (supervisor-worker) 方式，話し手聞き手 (announcer-listner) 方式などです．この辺は場当たり的でありますが示唆に富んでもいます.

付録に，FSPの早見表，具体構文，LTSによる形式的な意味記述，UMLクラス図がまとめてあります.

印象

本を瞥見した印象は，懐手して読める極めてわかりやすいというものです．支援系もよくできていて直観に訴えます．それに参考文献が豊富に挙げてあ

り，ちょっとわからなかったときや，本格的な勉強をはじめるのに便利です．最後になりましたがここで紹介した Magee+Kramer の本には付随した web 頁

http://www-dse.doc.ic.ac.uk/concurrency/

が設けられています．そこから教育（講義）用スライドや本にでてくる課題の Java のアプレットとして実行できるプログラムや LTSA の改訂版などが入手できるようになっていて，教育的配慮は万全です．

つぎの本は

つぎはまだ近刊といえると思います．

Leslie Lamport, *Specifying Systems – The TLA+ Language and Tools for Hardware and Software Engineers.* Addison - Wesley, 2003. pp. xvii+364.

本の著者である Leslie Lamport は LaTeX の作者として計算機関係者以外にも有名ですが，わたしたちには，並行プログラムの検証などで懐かしい名前です．実はわたしもいま LaTeX によってこれを書いています．

この本はハードウエア技術者とソフトウエア技術者が TLA+ によって簡潔で厳密な系の仕様記述方法と仕様分析方法を学ぶための教科書です．ハードウエア技術者というのは SpecC や SystemC の出現が物語るように， LSI や FPGA やアナログ装置を組み込んだ混成装置の仕様記述や設計の検証が問題になるからです．Lamport の論文はいずれも読みやすく教育的配慮の行き届いたものでしたが，これは，それをさらに徹底させ，老婆心の見本になっています．

TLA とは

TLA (Temporal Logic of Actions)「作用時間論理」は応答系の仕様記述とその性質を推論するために Lamport が考えた線形時間論理体系です．そして，論理体系に加えて仕様記述と系の抽象実現を単一言語によって行えるように構成したものです．すなわち，系がもつべき性質の記述とその抽象実現が同一の言語によって行え，さらに，その推論がその言語によってできるわけです．TLA は実務で重要な仕様の合成や洗練に便利なように時間論理を工夫修正したものです．

応答系に対してはその使用者と系との時間に沿った相互作用がとくに興味の対象になります．それが系の機能や論理的な性質を表しています．したがって，系の仕様は系の状態の時間的系列, つまり，挙動として記述すればいいことになります．

TLA 式は状態の無限列についての性質を述べるものです．つまり，TLA 式は状態列で評価して真偽が定めます．状態は (プログラム) 変数に値を割り当てる関数です．ここで，もう少し TLA 式をみてみましょう．式の構成要素は 4 種類あります．

1. 状態に全く依存しない**定数** c と**状態独立変数** v.
2. 単一状態で評価する**状態依存変数** x や定数から構成する**状態式**，真偽値をとる状態式を (状態) **述語** p，そうでない式を**状態関数** f などともいう．
3. 状態の組 (順序対) で評価する**作用式** A, 状態依存変数にプライムを付けた変数 x' を用意して 2 種類の変数 x, x' を用いて作用式を構成する．ここでも作用述語などという．
4. (無限) 状態列で評価する**時間式** (挙動述語)ϕ，この構成はあとで明示する．

作用は状態変化 (推移) を起こすもので，一般に事象とか行動とかといっているものです．作用式 A は状態 s を t へ推移させることを示すもので，式に現れるプライムなし変数は推移前，プライム付き変数は推移後を参照します．形式的に書けば，つぎのようになります．

$$\begin{aligned}
[c](s,t) &= c \quad (c \text{ は定数}) \\
[x](s,t) &= s(x) \quad (x \text{ は変数}) \\
[x'](s,t) &= t(x) \quad (x \text{ は変数}) \\
[f(e)](s,t) &= f([e](s,t)) \quad (f \text{ は関数}, e \text{ は式})
\end{aligned}$$

さて，TLA 式はつぎのように帰納的に定義します．

$$\phi, \psi \ ::= \ p \mid \text{可動}\ A \mid \Box[A]_f \mid \phi \wedge \psi \mid \neg\phi \mid \Box\phi \mid \exists m : \phi \mid \exists x : \phi$$

上で，p は状態術語，A は作用式，f は状態関数です．可動 A は作用 A が起こる可能性のあることを示す術語，つまり，[可動 A]s が真とは $[A](s,t)$ が真になる状態 t が存在することです．$\Box[A]_f \triangleq A \vee (f' = f)$ です．$\Box\phi$ は挙動のすべての状態で ϕ が成立することを意味して，$\Box$ は常に，いつもという時間演算子です．ただし，作用 A に対して $\Box A$ は A が状態述語に限り許します．$\Box$ の双対時間演算子やがて，いつかという，すなわち，挙動で ϕ を成立させる状態があることを意味する $\Diamond\phi \triangleq \neg\Box\phi$ も便宜上定義しておきます．

TLA 式は挙動，つまり，状態の無限列 $\sigma = s_0 s_1 s_2 \cdots$ に対して評価します．

$$
\begin{array}{rcl}
[p]\sigma & = & [p]s_0 \\
[\text{可動 } A]\sigma & = & [\text{可動 } A]s_0 \\
[\Box[A]_f]\sigma & \textit{iff} & \text{任意の } n \geq 0 \text{ に対し } [A](s_n, s_{n+1}) \text{ または } [f = f'](s_n, s_{n+1}) \\
[\Box\phi]\sigma & \textit{iff} & \text{任意の } n \geq 0 \text{ に対し } [A]s_n s_{n+1} \cdots
\end{array}
$$

$\phi \wedge \psi$, $\neg\phi$, $\exists m : \phi$, $\exists\, x : \phi$ については省略します．あとの二つは情報の隠蔽に用います．無論，通常通り，$\phi \vee \psi$, $\phi \Rightarrow \psi$, $\phi \Leftrightarrow \psi$ などを導入しておきます．

TLA は線形時間論理ですが，次にを意味する時間演算子 ○ や一般の作用 A に対する $\Box A$ を許していません．このように時間演算子の使い方を制限しているところが TLA の特徴です．その理由は，状態関数値を変えない複数個の状態を挙動に挿入，削除しても TLA 式の真偽が不変であるという性質，どもり不変性 (stuttering invariance) が仕様の洗練や合成のために必要で，上のように制限するとこの不変性がえられるからです．

仕様の書き方

仕様記述に TLA 式をどのように利用するかの詳細は本に書いてありますが，凡そはつぎのような具合です．

初期状態を状態述語「始」として定義します．つぎに状態を変える作用群をそれぞれ $A_1, A_2, \cdots$ と定義します．関心の対象である状態変数群をベクトルとして f と表します．作用群の選言を「次」とします．それが可動であれば必ず発生するという公平性を「活性」と定義して，その系の仕様「仕様」はつぎのように書けます．

$$\text{仕様} \triangleq \text{始} \wedge \quad \Box[\text{次}]_f \wedge \text{活性}\,[\text{次}]_f$$

定数や変数を宣言してそれらがもつ不変式を直接書くようにします．データの型をはじめから用意しておくことはしません（理由は本を見てください）．

つぎのプログラムをみてください．

```
const H ∈ {1..12}
 h := H;
do true → h := (h mod 12) + 1 od
```

これは，1 時，2 時と時を刻む時計プログラムです．do $b \rightarrow s$ od は述語 b が真である限り文 s を反復実行するというプログラム文です．この仕様を TLA+ ではつぎのような囲みを伴った図に書きます．

活性についても本を参照していただかねばなりませんが，弱公平性と強公平性をつぎのように定義しています．

module 時計

extends *Naturals*
CONSTANT H
VARIABLE h
型不変 $\triangleq H \in (1..12) \wedge h \in (1..12)$

始 $\triangleq h = H$
次 $\triangleq h' = (h \bmod 12) + 1$
時計 $\triangleq$ 始 $\wedge$ $\Box[$次$]_h$ $\wedge$ 活性$_h$(次)

THEOREM 時計 $\Rightarrow$ 型不変

A の弱公平性 作用 A が連続して可動となるなら A はやがて必ず生起する

A の強公平性 作用 A がとびとびでもいいが反復して可動となるなら A はやがて必ず生起する

これらは形式的には

$$WF_f(A) \triangleq \Diamond\Box \text{ 可動 } \langle A\rangle_f \Rightarrow \Box\Diamond\langle A\rangle_f$$
$$SF_f(A) \triangleq \Box\Diamond \text{ 可動 } \langle A\rangle_f \Rightarrow \Box\Diamond\langle A\rangle_f$$

と定義しています．なお，$\Diamond\langle A\rangle_f \equiv \neg\Box[\neg A]_f$ で，その意味は，挙動はやがて $A \wedge f' \neq f$ を満たすです．

TLA+は TLA を基礎として構成した仕様記述言語です．特徴はモジュール機構をもち，そこに集合その他の仕様記述上便利な数学的構造をモジュールとして用意し，書きやすさと読みやすさのための多くの構文の甘み付けを行っていることです．

本の読み方

時間論理などははじめてというひとは，とりあえず，第Ｉ部８３頁を読めということです．単純な数学の温習として命題論理，集合，述語論理についてまとめます．それから，簡単な例題の仕様を書きながら TLA+を覚えるようになっています．例題は１時２時と時間だけを表示する時計，非同期通信界面，先入先出緩衝域，高速記憶装置 (cache) をもった記憶装置などです．仕様記述の大部分は時間演算子とは関係なく，状態のもつ不変条件を安全性として書くことと，活性については公平性を書くだけです．

あとの第 II 部は活性と公平性，実時間，仕様の合成，データ構造の仕様化などを議論説明しています．仕様の合成などは TLA+利用の精髄です．

第 III 部は支援系の解説です．アスキー符号で書いた仕様を見やすくする TLATEX と模型検査系 TLC の使用法が丁寧に書いてあります．

第 IV 部は TLA+言語の仕様です．

この本もオブジェクト指向分析設計や UML との関係はありません．しかし，技術的な，すなわち，科学の基礎の上に立ったソフトウエア開発を目指すなら，技術者として看過できない重要な成果がここに示されていると思います．しばらく OOAD や UML を忘れて Magee+Kramer やこの Lamport で楽しんではいかがと思います．

書評「プログラム仕様記述論」

熊谷 章
(Kamakura Bears Institute)

生まれて初めて書評を書く．本のテーマがよいことと著者二人をよく知っているからである．本のタイトルは，「IT TEXT プログラム仕様記述論」で，著者は九州大学の荒木啓二郎と南山大学の張漢明である．この本は2002年11月にIT TEXTシリーズの一環としてオーム社から出版された．私と著者はソフトウェア技術者協会（SEAと呼ぶ）のフォーマルメソッド分科会（sigfmと呼ぶ）のメンバーである．sigfmのミーティングは，毎月開催され日本国内外のフォーマルメソッドの動きと内容を研究テーマとし活動している．

フォーマルメソッドへの期待は大きいが，日本語のよい本がないというのが課題であった．この難題を荒木と張の名コンビが見事に解決してくれた．フォーマルメソッドを学習する人のための待望の入門書である．彼らの努力に敬意を払いながら早速本を開いてみよう．

はしがき

「はしがき」にこの本の意図が書かれている．形式手法は約30年前から研究開発され続けており，欧米では実用システムへの応用事例が学会発表や出版物として公表されているという．そして，ソフトウェア開発の性質から考えて，論理的思考が最も重要だと説く．その性質とは，解決すべき問題対象を認識し，それに対する数理モデルを作り，それをベースに問題解決の手段を見つけ，それをコンピュータプログラムとして実現し，コンピュータ上で動作させ，そのシステムを保守改良するという一連の活動を指す．表現のあいまいさを排除し厳密な認識と議論を可能にするためには必然的に論理的表現である形式手法が必要になると主張する．

欧米で出版されている多くのフォーマルメソッド入門書では，集合論と論理学の基礎を最初に紹介しているが，この本では省略したらしい．その替わりに，プログラムの検証理論の入門解説を置いたという．その狙いは，形式仕様の基本となっている事前条件，事後条件，不変条件などをその由来と共に詳しく説明することにあった．これにより，1970年代のプログラム検証理論から現代のフォーマルメソッドまでをスムーズに理解できることを意図した，と力説している．

「目次」を眺めてみる．

ここで筆者のソフトウェアに関する経験と知識の程度を以下に紹介する．

1971年から1年間COBOLで流通業のアプリケーション開発，その後アセンブラでシステムプログラム開発に10年従事した．作ったものはTSSシステムとFORTRAN，BASIC，PASCALの処理系が主であった．このときに，コンパイラの技法，Fundamental Algorythm Vol.1－Vol.3，系統的プログラミング，Structured Programming，Multicsなどを読んだ．今でもKuthのFundamental Algorythm Vol.3とWirthのPASCAL入門を読んだ時の感激は忘れられない．この時代の課題は，プログラムの性能とバグの少なさであった．

その後，1980年頃からミニコンにC言語のUNIXを移植した．UNIXがProgrammers のWork Benchであることに感心し，ソフトウェア開発にコンピュータを有効に使うことを知った．ユーティリティからコマンドに至るまで同じインターフェースでかつそれらをフィルターという概念で任意に結合できる仕組みに感動した．UNIXはシステム全体が言語感覚に溢れていた．この時代の課題は，新ハード

ウェア開発，Proprietaryから国際標準への移行，対話型インターフェース，ソフトウェアの生産性向上であった．

次に手懸けたのがLISPとSmalltalkだった．1100SIPを購入しInterLispと Smalltalk-80に惚れ込んだ．専用マシン，ビットマップ，中間コード，バーチャルマシン，ポインティングデバイス，イメージ処理などの技術に心が騒いだ．同じようなマシンを自前で作り，その上にArtificial Intellgence などを作ろうと試みた．Lisp言語の持つ力強さとオブジェクト指向言語 Smalltalkの美しさに皆瞠目した．アセンブラとC言語は遠い昔のものに思え，プログラムの生産性は100倍向上したと感じた．この時代が1984年から10年間続いた．この時代の課題は，コンピュータの新しい分野での応用で，Artificial Intelligenceと自然言語処理がその具体例であった．ソフトウェアの開発では，文書ベースのWater Fall型からProtptype型やSpiral型のコンピュータを用いたTool群を使うCASEに変化した．

最近10年は分散コンピューティング時代である．最近はこれに並列コンピューティングが加った．コンピュータハードウェアも大きく変貌した．スーパーコンと汎用マシンが終焉し，IntelとSun MicrosystemのCPUがコンピュータアーキテクチャーを凌駕した．スーパーコンと汎用マシンはSMPのサーバマシンとPCクラスタに置き換えられ，ノートパソコンやPDAをインターネットで接続したMobile ComputingとPervasive Computingが台頭し始めた．更に，世界規模レベルでの協調としてGrid Computingの必要性が認識され，国内外に多くのGrid Computingに関するプロジェクトが起こされた．分散オブジェクト，コンポーネント，コミュニケーションプロトコル，並列プログラミング，Web Information Systems，Web Engineering， Design Patterns，UMLなどがこの時代のキーテクノノジである．この時代の課題は，並列分散コンピューティングのインプリメンテーション方法，短期間でのシステム開発，ヘテロジェニアスなコンピュータの統合，コンピュータで作成した社会システムの信頼性と安全性である．ソフトウェア開発では，ソフトウェアアーキテクチャ，ソフトウェアプロセス改善，プロジェクトマネージメント，UML，フォーマルメソッドがホットな話題になっている．

上述したような経験をしてきたソフトウェア技術者にとってこの本の構成は新鮮であった．このような本を読んだことがなかった．気になった点は，本のタイトルがシステム仕様記述でなくプログラム仕様記述であることだ．なぜなら，ここ30年の経験を概念的に言えば，最初の1970年代はプログラムが主テーマで，次の1980年代は新しいコンピュータとプログラム言語が主テーマで，1990年代はシステム全体とそのシステムの安全性と信頼性がキーテクノロジだと言えるからで，明らかに主題は「program in the small」から「program in the large」へと変化したからである．その点をこの本の著者らは心得ており，70年代と今を結びかつその間の橋渡しを狙っていると明言している．期待して読んでみよう．

第１章 プログラムの正しさ ープログラムの検証入門ー

この章ではプログラムの正しさがテーマである．プログラムの入力と出力をそれぞれ事前条件と事後条件としてプログラムの正当性を説明している．ここで用いた証明方法は，1960年代にNaurとFloydoが提案した帰納的表明で，流れ図プログラムを対象としている．数学の帰納法によって流れ図プログラムに対応した論理式が真であることを証明している．

流れ図プログラムの構成要素が，条件分岐，合流，代入，開始と終了の5つから構成されているプログラムを題材にして，それぞれの要素に対応する論理式を示しそれが真であることを示す．そして，最後に事前条件と事後条件を満足するようであればこの流れ図プログラムは部分的に正当であることが示されたとする．

使用されている例題は多項式aiの総和を求めるという単純なものだった．従って，流れ図を見ただけで成否が直ちに判る．普通，我々は流れ図やソースプログラムを見てそれらの正当性を見抜くことに慣れ親しんでいる．だから，判っている事柄を論理式に書き直し，それらのひとつひとつの真偽を確かめる必要があるかと訝しく思ってしまう．私の経験から推して，入力データの定義と出力データの定義を行い，そのプログラムで果たす機能を明確にし，そのアルゴリズムを決めればプログラムの仕様が決まったと考える．検証は，典型的なテストセットを用意し，それに対する合格基準を設定し，実際の検証テストを行えばよかった．従って，ここに単純な問が生まれた．なぜ，1970年代に上で述べた帰納的表明が実際のプログラム開発に使用されなかったか，という疑問だ．

この問に対する答えを私は二つ持っている．ひとつは，コンピュータのハードウェアの計算精度である．二進法のコンピュータでは実数を正確に表現できないため，実際の計算値には近似値を用いるしかない．その結果，ある式の計算結果が異なったコンピュータ間では異なるのが普通であった．これでは，異なったコンピュータ上で共通に使用できるプログラムを作成することができない．そこで採用された苦肉の策は，一番多く使用されているコンピュータの計算結果に合わせようということだった．実際，多くの実数処理の関数はその実行結果がIBMのそれに合うことが正しいとされた時期があった．このことから，プログラムの検証とは真偽ではなく，ある値に対する合致することだという認識があった．二つ目は，帰納的表明を我々は頭の中で行っており，その結果として流れ図プログラムや入出力データの定義ができるのだという思いである．アルゴリズムを模索し試行錯誤しているときは，頭の中で正に各々の段階で真偽を判定し，多くの選択肢の中から真らしいものを抽出しその解を展開している．従って，各々の段階の細かい事柄を論理的に書き出すことは多大な負荷となるため書き出さず，結果だけを式の形などで記述するに過ぎない．

私のような老兵でかつ独学でしかソフトウェアを作成しなかった者，つまり論理の飛躍が常識化してしまっている者にとって，論理式を用いてプログラムの正しさを証明するこの章は序章として十分な役割を果たしているようにみえた．

第２章 Floyd-Hoare論理

Whileプログラムを論理式で部分的正当性を証明している．部分的正当性の表明は，{P}S{Q}と書かれる．これは，「事象条件Pを満足するときに，プログラムSを実行すると，その実行後に事後条件Qを満足する」という意味であるという．このプログラムに関係する公理と推論規則を導入している．公理系から出発し，推論規則を適用して所期の論理式を導出することによって証明するのである．この証明できる論理式を，検証条件と呼ぶという．公理から検証条件を導出し，その検証条件が真であればそのプログラムは正しさが証明されたことになる，という主張は気持ちが良い．

例題として，２変数の値交換，階乗プログラム，最大公約数プログラム，掛算プログラムを挙げ，それぞれの部分的正当性を証明している．その証明例で判ることだが，証明図の作成を公理から出発し，推論規則を用いて正当性を証明する証明図を導出する，という手順ではない．これとは，まったく逆向きな手続きをとり，最後に公理に辿り着くようにしている．最後の公理は論理式であり，対象としているプログラム要素をまったく含まない一階述語論理で表現できる．つまり，プログラムの証明をこの純粋な論理式が真であることを示すことによって証明できることになる．

成る程と推論規則を適用しながら式を展開し，目標の論理式までを辿ってみる．次に示す例３：xとｙの掛算プログラムは，樹形図だと５階層になり説明図は１ページを要する．

```
{ x > 0 }
z:=y;
u:=x ? 1;
While u <> 0 Do
z:=z + y;
u:=u ? 1;
End
{ z = x * y }
```

論理の展開に納得はするのだがどうしてこんなに複雑になるのかと一章で述べた感想と同じになってしまう．上のプログラムは，日本語で表現すれば次のようになる．「ｘがゼロより大きければ，ｘにｙを掛けるということは，ｙにｙをｘ回足すことである」．この命題が真であるか否かを証明することが，フォーマルメソッドであると言ってよい．とすれば，それは最終的には論理語（これは，日本語でも記号でも良い）が持っている固有規則，つまり推論規則に依存していると考えられる．このように考えれば，どのような論理語が存在し，それらにはどのような推論規則があるかを理解することが必要条件だと言うことに気付く．それが記号論理というもので，コンピュータプログラムを対象としてその理論を打ち立てたのがFloyd-Hoare論理だと知る．

言説は真であるか偽であるかしかないが，果たしてある言説の真偽を見極めるためには言語に立脚した記号論理が必要である，ということがこの章から教わった．　　（以下次号）

ソフトウェア技術者協会
〒160-0004　東京都 新宿区 四谷 3 - 12 丸正ビル 5F
Tel: 03 - 3356 - 1077　Fax: 03 - 3356 - 1072
E-mail: sea@sea.or.jp
URL: http://www.iijnet.or.jp/sea