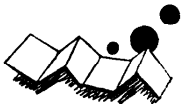
解　説



● 並列オブジェクト指向言語 ABCL による在庫管理システムの記述†

柴山悦哉†† 松田裕幸††* 米澤明憲††

1. はじめに

近年，オブジェクト指向という考え方が各方面で注目を集めている．特に，高機能ワークステーション上のウィンドウシステムの実現に際しては，何らかの形でオブジェクト指向の思想が関与している．たとえば，Xerox 1100 や Symbolics リスプマシンのウィンドウシステムは，それぞれ Smalltalk[2),3)], Flavors[9)] と呼ばれるオブジェクト指向言語で記述されているし，Sun Workstation では，C言語をオブジェクト指向風に用いてウィンドウシステムを記述している[7)]．ウィンドウシステム以外にも，オペレーティングシステム[8)]，自然言語処理[5),6)]，シミュレーション[13)]等の分野で，オブジェクト指向の考え方が有効に活用されている．

本稿は，並列オブジェクト指向という新しいプログラミングパラダイムを，共通問題[10)]の解法を通して解説することを主たる目的とする．ただし，プログラムを記述する言語としては，並列オブジェクト指向言語 ABCL[4),12)]を用いる．また，本稿の末部では，オフィスオートメーションにおける電子書式をオブジェクトと捉えて設計・実現した場合の利点などに触れ，さらにオブジェクト指向プログラミングによるプロトタイピングについても簡単に議論する．

2. オブジェクト指向

2.1 オブジェクトとは

オブジェクト指向の考え方の真髄は，与えられた問題の中に現われる物やデータをオブジェクト (object) という単一の枠組を用いて記述することにある．たとえば共通問題には，倉庫やコンテナのような物体，倉庫係や受付係のような人間，さらには出庫指示書や在庫不足品リストのような帳簿類等が出現するが，これらはすべてオブジェクトとして自然かつ統一的に捉えることができる．

以下，本稿で述べるオブジェクトとは次の性質を満たすようなものである．

a．固有の内部状態を持ち，他のオブジェクトと独立（並列）に計算を行う能力を持つ．

b．各オブジェクトは，自分自身の内部状態を見たり変更したりすることはできるが，自分以外のオブジェクトの内部状態を直接的に見たり変更したりすることはできない．

c．各オブジェクトは任意のオブジェクトに対してメッセージを送る機能と任意のオブジェクトからメッセージを受け取る機能を持つ．メッセージを受け取ったオブジェクトはそのメッセージの内容にしたがって定められた手続きを実行する．ただし，複数のメッセージを一つのオブジェクトが受け取った場合，対応する手続きは逐次的に処理される．

オブジェクト間の相互作用（通信・同期など）はすべてメッセージを介して局所的に（メッセージをやりとりするオブジェクト間のみで）行われる．共有メモリ (shared memory) やグローバルクロック (global clock) の存在は一切仮定しない．

たとえば，あるオブジェクトが別のオブジェクトの状態を知る必要が生じたなら，相手のオブジェクトにメッセージという形で質問を発する．メッセージを受け取ったオブジェクトはこの質問に対する返事を返す．また，あるオブジェクトが別のオブジェクトの内部状態を変更する必要が生じたら，相手のオブジェクトにメッセージという形で依頼をする．この場合，メッセージを受け取ったオブジェクトは返事を返す必要はない．

---

† A Description of an Inventory Control System Based on an Object Oriented Concurrent Programming Methodology by Etsuya SHIBAYAMA, Hiroyuki MATSUDA and Akinori YONEZAWA (Department of Information Sciences, Tokyo Institute of Technology).

†† 東京工業大学理学部情報科学科
* 現在　日本電気技術情報システム開発(株)

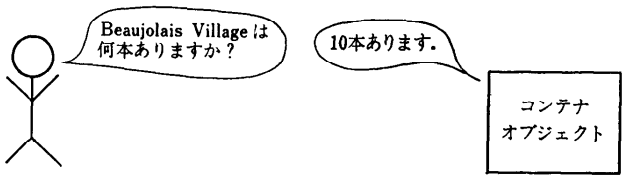


図-1 コンテナオブジェクトへの質問

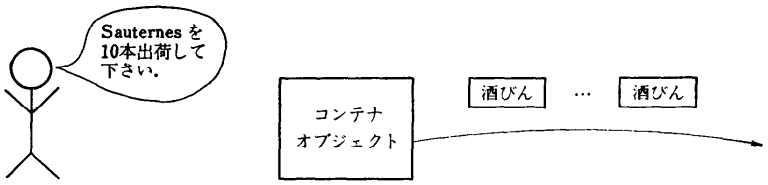


図-2 コンテナオブジェクトへの依頼

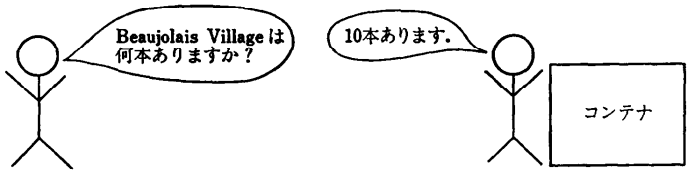


図-3 コンテナ係への質問

例として，共通問題[10]に現われるコンテナをオブジェクトとして捉えてみよう．ここでは簡単のため，コンテナオブジェクトは次のような能力のみを持つものとする．

a．「どんな銘柄の酒が何本あるか？」という質問に答えられる（図-1）．

b．「ある銘柄の酒を出庫してくれ」という依頼に答えられる（図-2）．

図-1 の質問に対しコンテナオブジェクトは，自分の内部に存在する Beaujolais Village の本数を数え上げ，その結果を質問者に伝える．また 図-2 の依頼に対しては，自分の内部にある Sauternes を 10 本出荷する．なお，この時出荷される酒びんも一般にはオブジェクトとして実現される．

ところで，コンテナが能動的に振る舞うという点について不自然な印象を感じる読者もおられるであろう．実際，共通問題の世界を忠実にモデル化するとコンテナは単なる箱であり，倉庫係や運送係に扱われてはじめて意味を持つ受動的存在となるはずである．

本来受動的なコンテナが能動的に動き回るという不自然さを解消するためには，各コンテナにそれを管理するコンテナ係が一名ずつ付いていると考えるとよい（図-3）．コンテナは単なる箱であるがコンテナ係は能動的な人間である．コンテナが他人からの質問に答えたり自分で酒びんを出荷したりするのは不自然であるが，コンテナの管理人がこれらの仕事をするのであれば何の不自然さもないであろう．なお，コンテナとコンテナ係の間には次の条件を満たす関係があるものとする．

a．コンテナの中身を直接見たり触ったりできるのは専任のコンテナ係だけである．

b．コンテナ係以外の人（倉庫係など）が，コンテナの中身の状態を知ったり変更したりしたい時には，専任のコンテナ係に質問または依頼をする．

オブジェクトとは，単なる物やデータのモデル化ではなく，物やデータと専任の管理人を結合した複合体のモデル化と考えるとよいであろう．

2.2 データ抽象，モジュラリティ，知識の分散

2.1 で述べたように，各オブジェクトの内部状態やメッセージを受け取った後に起動される手続きの詳細は他のオブジェクトから見ることができない．したがって，オブジェクト指向の考え方は必然的にデータ抽象（data abstraction）と手続き抽象（procedural abstraction）の概念を包含する[11]．

今仮に，共通問題に登場する積荷票をオブジェクトとして捉えてみよう（ここでは比喩的に，積荷票オブジェクト＝積荷票＋積荷票係と考えて話を進める）．積荷票係は，「新しく搬入されたコンテナに，何がどれだけ入っていますか？」という受付係からの質問に答える能力を持つものとする．この時，受付係が知らなければならないのは，積荷票係に対する質問の形式（メッセージプロトコル）だけであり，積荷票の書式は専任の積荷票係だけが知っていればよい．仮に，ある日突然に積荷票の書式が変更されたとしても，迷惑を被るのは積荷票係だけであり，受付係はこの変更の影響を一切受けない．したがって，この例から分かるように，オブジェクトという概念を導入することによ

り，モジュラリティが高く修正に強いプログラムの作成が可能となるのである．

さらに複雑な場合として，酒びんを搬入する会社ごとに積荷票の書式が異なっている場合を考えてみよう．積荷票を単なるデータとして扱う場合には受付係が積荷票の書式をすべて知る必要がある．しかし，積荷票がオブジェクトとして実現される場合，各積荷票の書式は専任の積荷票係だけが知っていればよく，受付係の持つべき知識は大幅に削減される．したがって，オブジェクトという概念を導入することにより知識の自然な分散化を図ることが可能となる．

3. 共通問題の解法

この章では並列オブジェクト指向の考え方による共通問題の解を与える．一般に，オブジェクト指向型プログラムの設計は次の二つの過程に還元される．

（1） 何をオブジェクトにするか決定し，それらのオブジェクト間の関係（どのオブジェクトからどのオブジェクトにどんなメッセージを送るべきか）を解析する．

（2） 各オブジェクトの内部状態と手続きの設計を行う．

（1）はオブジェクトの外部設計であり，（2）はオブジェクトの内部設計に相当する．このうち本当に重要なのは外部設計の方である．各オブジェクトが十分に小さく分割されていれば，その内部設計はどんなやり方を用いてもそんなにむつかしいものではない．

3.1 受付係オブジェクトの外部設計

ここでは，受付係及び受付係と直接的に係わり合いを持つオブジェクトを列挙し，さらに受付係オブジェクトが受理すべきメッセージの種類を考察する．

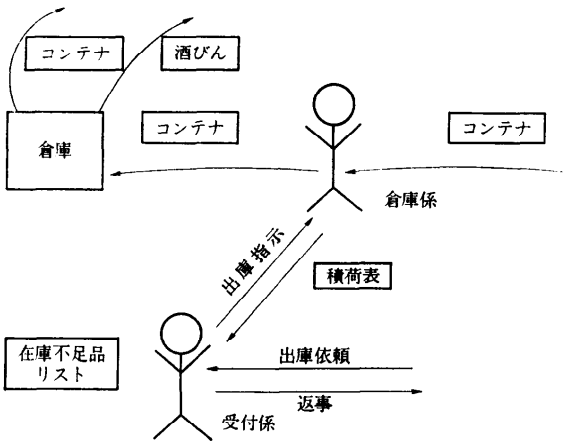


図-4 共通問題仕様の図式化（その1）

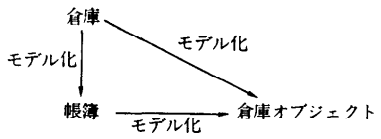


図-5 帳簿と倉庫オブジェクト

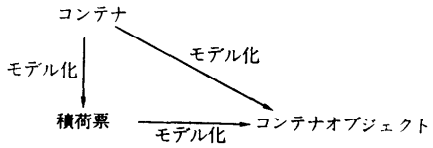


図-6 積荷票とコンテナオブジェクト

まず，オブジェクトを列挙するという問題から解いていこう．仕様の文章を図式化すると 図-4 のようになる．この図に出現する物をすべてオブジェクトとするのがこの問題に対する一つの解である．ただし今回は，次の点を考慮して 図-4 を変形したものを最終的な解とする．

1. 2章で述べたようにオブジェクトは物やデータに専任の管理人をつけた複合体のモデルである．したがって，倉庫と倉庫係は合わせて一つの倉庫オブジェクトとした方が自然である．

2. 図-4 には現われていないが，受付係が仕事を行うためには，現在の倉庫の状態を知る必要がある．現実の世界では倉庫のモデルとして，ある種の帳簿を受付係が管理することになるが，オブジェクトの世界では倉庫オブジェクトが倉庫の自然なモデルとなる．

そこで，倉庫に関して受付係が知っているはずの情報を表現するために，倉庫オブジェクトを一つ用意する．倉庫オブジェクトは，現実の倉庫のモデル化であると同時に，帳簿のモデル化でもある（**図-5**）．

3. 積荷票は本来コンテナを伝票としてモデル化したものである．オブジェクトの世界ではコンテナオブジェクトがコンテナの自然なモデルを与える．そこで，積荷票の代わりにコンテナオブジェクトを用いる．コンテナオブジェクトはコンテナのモデル化であると同時に，積荷票のモデル化でもある（**図-6**）．

上の1，2，3より，図-4 は **図-7** のように変形される．図-7 で，受付係の右隣に描かれた倉庫オブジェクトは仮想的な（いわゆる帳簿上

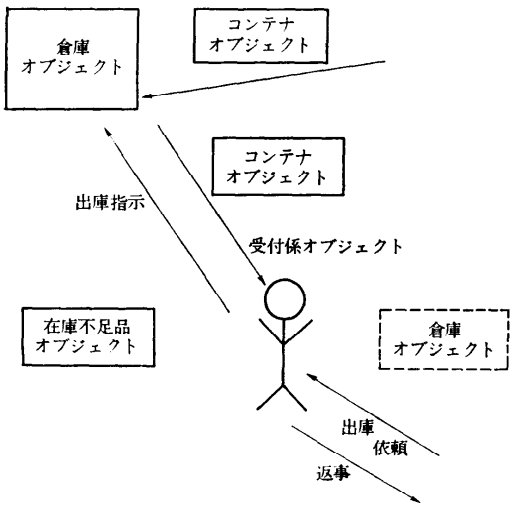


図-7 共通問題仕様の図式化（その2）

の）倉庫を表現し，もう一つの倉庫オブジェクトが現実に存在する倉庫を表現する．

ところで，倉庫オブジェクトが二つあると，受付係はそれぞれに対し出庫依頼等のメッセージを送る必要があり，これは面倒である．二つの倉庫オブジェクトは同じ論理構造を持っているはずであるから，同一視して一つにしてしまう（**図-8**）．

結局，共通問題の世界には 図-8 に出現する次の四種類のオブジェクトが存在することになる．

1. 受付係オブジェクト
2. 倉庫オブジェクト
3. 在庫不足品オブジェクト
4. コンテナオブジェクト

そして，受付係が受理するメッセージは次の二種類

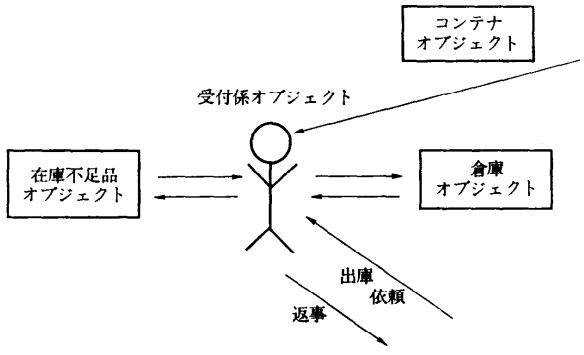


図-8 共通問題仕様の図式化（その3）

```
[object オブジェクト名
  (state : 内部変数宣言)
  (=> [メッセージパターン] 手続き記述)
  (=> [メッセージパターン] 手続き記述)
    ⋮
  ⋮ ]
```

図-9 オブジェクト定義の構文

```
[object 受付係
  (=> [搬入 : *コンテナ] ...)
  (=> [出庫依頼 : *銘柄 *本数 *送り先 *依頼者] ...)]
```

図-10 受付係オブジェクトのトップレベル定義

である．

1. コンテナ搬入メッセージ
2. 出庫依頼メッセージ

3.2 受付係オブジェクトの内部設計

ここでは，われわれが Vax11 及び Sun Workstation (OS は共に Unix 4.2 BSD) の Franz Lisp 上で開発中の並列オブジェクト指向型言語ABCL[4),12)]による受付係オブジェクトの内部記述を与える．

ABCL では，各オブジェクトの内部は Lisp によって記述される．ただし，ここで Lisp を用いているのは，たまたま Franz Lisp 上で ABCL を実現しているためである．オブジェクトの内部記述には，逐次的なプログラミング言語にメッセージ送信の機能を付け加えたものならば何を用いてもかまわない．

ABCL によるオブジェクト定義の構文は，一般的には **図-9** のようになる．ここで内部変数とは，オブジェクトの内部状態を保持するための Lisp の変数で，他のオブジェクトからは直接参照することができない．各オブジェクトは，あるメッセージパターンとマッチするメッセージを受け取ると，対応する手続きを実行する．なお，この手続きは，メッセージ送信等の若干のプリミティブを付け加えた Lisp のプログラムとして記述される．

受付係オブジェクトのトップレベルの記述は **図-10** のようになる．受付係の仕事は，倉庫と在庫不足品リストのみを見比べて行われ，固有の内部状態（あるいは内部データ）を必要としない．そこで，内部変数宣言は省略した．また，メッセージパターンとしては，コンテナ搬入用と出庫依頼用の二つが用意されている．メッセージパターン中で ':' または '?' で終わる文字列はタグと呼ばれ，字面が自分と等しい物としかマッチしない．図-10 では '搬入 :' と '出庫依頼 :' がタグである．また，'*' で始ま

る文字列はパターン変数と呼ばれ任意の Lisp 式とマッチする．したがって，'＊銘柄' は 'Chablis-Grand-Cru' とも 'Bordeaux-Rouge' ともマッチする．あるメッセージパターン中に現われたパターン変数が続く手続き中にも出現した時は，マッチした Lisp 式を値として持つ Lisp の変数として扱われる．したがって，受付係オブジェクトが［出庫依頼：越乃寒梅　10　山崎酒店　山崎酒店］というメッセージを受け取ると，［出庫依頼：＊銘柄 ＊本数 ＊送り先 ＊依頼者］に続く手続きが ＊銘柄＝越乃寒梅，＊本数＝10，＊送り先＝山崎酒店，＊依頼者＝山崎酒店という変数束縛のもとで実行される．

受付係オブジェクトの仕事を遂行する際には，倉庫オブジェクトと在庫不足品オブジェクトの状態を参照したり更新したりする必要が生じる．これらの作業は，倉庫オブジェクトや在庫不足品オブジェクトに対してのメッセージ送信によって実現される．ABCL では，メッセージ送信のために次の三つのプリミティブが用意されている[12]．

（1）［オブジェクト <- ［メッセージ］］

（2）［オブジェクト <== ［メッセージ］］

（3）［変数 :- ［オブジェクト <- ［メッセージ］］］

（1），（2），（3）の型のメッセージ及びメッセージ送信式をそれぞれ past 型，now 型，future 型と呼ぶ．past 型のメッセージを送ったオブジェクトは，相手のオブジェクトと非同期的に自分の仕事を続ける．一方 now 型のメッセージを送ったオブジェクトは相手のオブジェクトからの返事が届くまで待ってから仕事の続きを行う．すなわち，now 型のメッセージは通信だけでなく同期のプリミティブとしても使える．future 型のメッセージを送信したオブジェクトは，相手のオブジェクトからの返事を待たずに自分の仕事を続け，相手からの返事が返ってきた段階でその結果を左辺の変数に代入する．もし，返事が到着する前にこの変数を参照する必要が生じたなら，メッセージを送ったオブジェクトは返事待ちの状態となる．

```
[object 受付係
  (=> [ 搬入: ⁎コンテナ ]
    [ 倉庫 <= [ 入れて下さい: ⁎コンテナ ]]
    [ 在庫不足品リスト <= [ リセット:]]
    (case-loop [ 在庫不足品リスト <== [ 次は? ]]
      (is (⁎銘柄 ⁎本数 ⁎送り先)
        (case [ ⁎コンテナ <== [ 何本ありますか? ⁎銘柄 ]]
          (is ⁎在庫数 s.t. (< 0 ⁎在庫数)
            [ ⁎コンテナ <=
                [ 予約して下さい: ⁎銘柄 (min ⁎在庫数 ⁎本数) ⁎送り先]]
            [ 在庫不足品リスト <= [ 減らして下さい: (min ⁎在庫数 ⁎本数)]]
            (if (= ⁎本数 ⁎在庫数) then
              [ 倉庫 <= [ 予約品を出庫して下さい: ⁎銘柄 ]]))))))
  (=> [ 出庫依頼: ⁎銘柄 ⁎本数 ⁎送り先 ⁎依頼者 ]
    [ 倉庫 <= [ リセット: ]]
    (case-loop [ 倉庫 <== [ 次は? ]]
      (is ⁎コンテナ s.t. (and ⁎コンテナ (< 0 ⁎本数))
        (case [ ⁎コンテナ <== [ 何本ありますか? ⁎銘柄 ]]
          (is ⁎在庫数 s.t. (< 0 ⁎在庫数)
            [ ⁎コンテナ <=
                [ 予約して下さい: ⁎銘柄 (min ⁎在庫数 ⁎本数) ⁎送り先]]
            [ ⁎本数 := (- ⁎本数 (min ⁎在庫数 ⁎本数))])))))
    (if (= ⁎本数 0) then
      [ ⁎依頼者 <= [ ただちに送ります: ]]
      [ 倉庫 <= [ 予約品を出庫して下さい: ⁎銘柄]]
    else
      [ ⁎依頼者 <= [ 在庫がありません: ]]
      [ 在庫不足品リスト <= [ 入れて下さい: ⁎銘柄 ⁎本数 ⁎送り先 ]]))]
```

図-11　受付係オブジェクト

図-11 が ABCL による受付係オブジェクトの一記述例である．ただし，倉庫・在庫不足品リスト・コンテナの各オブジェクトは次のようなトップレベルの構造を持っていると仮定する．

```
[object 倉庫
  (state: コンテナリスト)
  (=> [入れて下さい: ＊コンテナ] ...)
  (=> [リセット: ] ...)
  (=> [次は?] ...)
  (=> [予約品を出庫して下さい: ＊銘柄] ...)]
[object 在庫不足品リスト
  (state: 不足品リスト)
  (=> [入れて下さい: ＊銘柄 ＊本数 ＊送り先] ...)
  (=> [リセット: ] ...)
  (=> [次は?] ...)
  (=> [減らして下さい: ＊減少数] ...)]
[object コンテナ
  (state: 酒びんリスト)
```

(=> [何本ありますか？ *銘柄] ...)
(=> [予約して下さい：*銘柄 *本数 *送り先] ...)
(=> [予約品を出庫して下さい：*銘柄] ...)]

倉庫オブジェクトのメッセージパターンのうち最初の三つは，このオブジェクトにストリームとしての機能を持たせるためのものである．[入れて下さい：*コンテナ] とマッチするメッセージを倉庫オブジェクトが受け取ると，'*コンテナ' とマッチしたコンテナが倉庫の中に入れられる．また，[リセット：] の後で何回か続けて [次は？] を受け取ると，各 [次は？] に対する返事として，内蔵しているコンテナを一つずつ順番に返す．ただし，内蔵するコンテナの個数以上の [次は？] を受け取ると，nil が返される．それ以外に倉庫オブジェクトは，「指定された銘柄の酒ですでに予約されているものを出庫する」ためのメッセージを受け取ることができる．在庫不足品リストオブジェクトも倉庫オブジェクト同様に，ストリームとしての機能を支援する三つのメッセージパターンを持つ．それ以外に，「[次は？] によって最も最近アクセスした不足品の本数を減らす」ためのメッセージを受けつける．また，コンテナオブジェクトは「コンテナ内の指定された銘柄の酒びんのうち予約されていないものの本数を聞く」，「指定された銘柄の酒びんを指定された本数だけ予約するよう依頼する」，「指定された銘柄の酒びんのうち予約されているものを出庫するよう依頼する」ためのメッセージパターンを持つ．

図-11 で，':=' は代入を表わす．また，case は次のような引数を持つパターンマッチ付の条件分岐関数である．

```
(case 式
  (is パターン s.t. 条件式
  式 ...)
  ⋮
  (is パターン s.t. 条件式　式 ...)
  (otherwise 式 ...))
```

すなわち，case の第一引数の値（第一引数が Lisp 式ならその評価値，now 型のメッセージなら返ってきた値）とマッチするパターンで，しかも 's.t.' の次の条件式が真になるものがあれば続く式の列が，もしなければ 'otherwise' 以下が，パターンマッチによって作り出された変数束縛環境のもとで評価される．なお，'otherwise' 以下と 's.t. 条件式' は省略することができる．'otherwise' 以下が省略されていて，かつ第一引数とマッチするパターンが存在しなければ，こ

```
[object 倉庫
 (state: コンテナリスト pointer)
 (=> [ 入れて下さい: *コンテナ ] (EnterList コンテナリスト *コンテナ))
 (=> [ リセット: ] [ pointer := コンテナリト ])
 (=> [ 次は? ]
   (case pointer
     (is (*最初 . *残り) !*最初 [ pointer := *残り ])
     (otherwise !nil)))
 (=> [ 予約品を出庫して下さい: *銘柄 ]
    [コンテナリスト <= [ 予約品を出庫して下さい: *銘柄 ]])
 (=> [ 搬出: *コンテナ ]
    (RemoveList *コンテナ コンテナリスト))]

[object 在庫不足品リスト
 (state: 不足品リスト pointer last-returned)
 (=> [ 入れて下さい: *銘柄 *本数 *送り先]
   (EnterList 不足品リスト (list *銘柄 *本数 *送り先)))
 (=> [ リセット: ] [ pointer := 不足品リスト ])
 (=> [ 次は? ]
   (case pointer
     (is (*最初 . *残り)
       !*最初 [ last-returned := *最初 ] [ pointer := *残り ])
     (otherwise !nil)))
 (=> [ 減らして下さい: *減少数 ]
    case last-returned
     (is (*銘柄 *本数 *送り先) s.t. (< *減少数 *本数)
        (SubstituteList (list *銘柄 (- *本数 *減少数) *送り先)
                        last-returned 不足リスト))
     (otherwise (RemoveList last-returned 不足品リスト)))]

[object Createびん
 (=> [ new: *銘柄 ]
  ![object びん
     (state: 送り先)
     (=> [ 銘柄? ] !*銘柄)
     (=> [ 予約して下さい: *送り先 ] [ 送り先 := *送り先 ])
     (=> [ 出庫して下さい: ] [ 送り先 <= [ 酒びん: びん ]])])]
```

(つづく)

```
[object Createコンテナ
  (=> [new: #酒びんリスト #ID ]
   ; #酒びんリスト = ((銘柄 未予約びんリスト 予約済びんリスト) ... )
   ![object コンテナ
      (state: [ 酒びんリスト := #酒びんリスト ])
      (=> [ あなたのIDは? ] !#ID)
      (=> [ 何本ありますか? #銘柄 ]
        (case (assoc #銘柄 酒びんリスト)
          (is nil !0)
          (is (# #未予約びんリスト #) !(length #未予約びんリスト))))
      (=> [ 予約して下さい: #銘柄 #本数 #送り先 ]
        (temporary: 予約するびんのリスト)
        (case (assoc #銘柄 酒びんリスト)
          (is (# #未予約びんのリスト #予約済びんリスト)
             [ 予約するびんのリスト := (FirstN #本数 #未予約びんリスト)]
             [ 予約するびんのリスト <= [ 予約して下さい: #送り先 ]]
             [ #予約済びんリスト :=
                 (append #予約済びんリスト 予約するびんのリスト)]
             [ #未予約びんリスト := (ButFirstN #本数 #未予約びんリスト)]
             (SubstituteList (list #銘柄 #未予約びんリスト #予約済びんリスト)
                             (assoc #銘柄 酒びんリスト) 酒びんリスト))))
      (=> [ 予約品を出庫して下さい: #銘柄 ]
        (case (assoc #銘柄 酒びんリスト)
          (is (# nil #予約済びんリスト)
             [ #予約済びんリスト <= [ 出庫して下さい: ]]
             (RemoveList (assoc #銘柄 酒びんリスト) 酒びんリスト))
          (is (# #未予約びんリスト #予約済びんリスト)
             [ #予約済びんリスト <= [ 出庫して下さい: ]]
             (SubstituteList (list #銘柄 #未予約びんリスト nil)
                             (assoc #銘柄 酒びんリスト) 酒びんリスト))))
        (case 酒びんリスト
          (is nil [ 倉庫 <= [ 搬出: コンテナ ]]))))])]
```

図-12 倉庫，在庫不足品，びん，コンテナの各オブジェクト

の case 式は何も実行せずに終了する．'s.t. 条件式' を省略した節(is パターン 式 ...)は(is パターン s.t. t 式 ...) と同等である．ただし，'t' は常に真の値を持つ Lisp の式である．パターンの中には，メッセージパターン同様にタグとパターン変数が書ける．また，case-loop は case と似た関数で，第一引数の値とマッチするパターンが存在する間繰り返し実行される．

図-12 は倉庫，在庫不足品リスト，コンテナ，ビンの各オブジェクトの定義である．なお，コンテナとビンに関しては，それぞれを生成するオブジェクトの定義を掲げた．'!' は，メッセージを送ってきたオブジェクトに返事を返すためのプリミティブである．'!' の次に Lisp 式がくれば，その評価値が返され，オブジェクトの定義式がくれば，この定義を満たすオブジェクトが新しく作られて返される．内部変数宣言部に代入式がきた場合，左辺の変数が内部変数として宣言され右辺の値が初期値となる．また，ただの変数名がきた場合，この変数が内部変数として宣言され，その初期値は，nil になる．past 型のメッセージの左辺にオブジェクトでなくオブジェクトのリストが現われた時には，このリストの要素の各オブジェクトに右辺のメッセージが同時に送られる (multi-cast)．EnterList, RemoveList, SubstituteList, assoc, append, FirstN, ButFirstN 等は Lisp の関数である．EnterList は，第一引数のリストの最後に第二引数を挿入する．RemoveList は，第二引数のリストから第一引数と (eq で比べて) 等しいものを取り除く．SubstituteList は，第三引数のリストの要素の中に第二引数と (eq で比べて) 等しいものがあれば，第一引数の値と置き換える．これらは，副作用によってリストの値を書き換える．また assoc は，第一引数が key，第二引数が ((key 1...) (key 2...) ... (key n...)) の時に，key＝key i なる i が存在すれば (key ...) を返し，さもなくば nil を返す．append は二つのリストを結合して新しいリストを作りだす．FirstN は第二引数のリストの要素を，第一引数の値の個数だけ先頭から取り出してリストを作る関数である．ButFirstN は第二引数のリストの要素を，第一引数の値の個数だけ先頭から取り去った残りのリストを返す．

4. 議 論

以上で，ABCLによる共通問題の解答を終る．引き続き，共通問題の解答だけでは示せなかった並列オブジェクト指向の利点に関する議論を行う．

4.1 書類 VS オブジェクト

書類というものをオブジェクトとして捉えると，単なる紙切れを超える概念が生まれる．筆者らの経験から言っても，現実世界の書類には不便なものが多い．書式が面倒であったり，おびただしい数の印鑑を必要としたりしていつも苦労をさせられる．もしこれらの書類がすべてオブジェクトであったなら，書式等に関して何も知らない人でも，依頼のためのメッセージをこれら書類オブジェクトに送るだけで，すべての手続きが完了する．これは，各書類に有能な秘書がついている状況に相当し，われわれの負担は大きく削減されるであろう．

計算機上でも，書類をCOBOLのプログラムのような形でモデル化するかわりにオブジェクトを用いてモデル化するといくつかの利点がある．たとえば，各書類の書式，書き方，制約条件，さらにその書類にまつわる慣例等はオブジェクトが知っていればよく，書類の利用者は細かいことを何も知らなくても利用できる．さらに，この書式，書き方，制約条件，慣例等に変化が生じた時も，利用者には最低限の負担しかかからないはずである．

オブジェクトを使うと，書類・帳簿以外に絵，図形，一般的なデータ等を統一的に扱うことができる．近年OAの分野でも，オブジェクト指向の考え方が注目されている．

4.2 プロトタイピング

近年，人工知能分野におけるプログラミングのように，簡単なプロトタイプを作ることから始めないことには，問題の解析すらできない場合が増えてきた．こういう場合，まず仕様があって，それからプログラムの設計を行うという考え方では問題を解決できない．オブジェクト指向の考え方はむしろこういう分野でこそ生かされる．すなわち，オブジェクト指向は単なるプログラム設計のためのパラダイムではなく，問題の解析と仕様の設計を含めたトータルなプログラム開発のための道具なのである．

今回の共通問題は，プログラム設計法の優劣を競うためのものである．そのため仕様の完成度が非常に高い．プログラム開発の道具としてのオブジェクト指向の威力を発揮するためには，もう少し完成度の低い仕様を与えられた方がよかったものと思われる．もっとも，共通問題の仕様は，受付係が全情報を管理する中央集権型のものであったが，3章の解答は情報がコンテナや倉庫に分散する地方分権型のものになっている．したがって3章では，いつの間にか仕様を変更していたことになる．オブジェクト指向の考え方によらない仕様をもとに，オブジェクト指向の考え方によるプログラム設計を行うと，必然的に仕様の修正を伴うのかもしれない．

4.3 Lisp について

ABCLでは，オブジェクトの内部記述にLispを用いている．Lispは(1)柔軟なデータ構造と(2)プログラムとデータが同じ形式であるという特徴を持ち，プロトタイピングにむいた言語である．

たとえば，共通問題の仕様には，銘柄数やコンテナ番号のケタ数が細かく記されているが，Lispを使うとそんなことはまったく気にせずにプログラムが組める．仮に，コンテナ番号が1000ケタあっても何ら不都合は生じない．プロトタイピング時には，あまり細かいことは気にしたくないものである．

Lispのもう一つの利点は，読み込みルーチンを簡略化できる点にある．入力の解析部をLispを使って書くと，たいていは'(read)'の六文字で済んでしまう．この場合，入力としてはS式しか使えないが，S式はデータ構造として十分に一般的であり，Emacsエディタ等の入力支援環境も活用できるので，さほど不便さを感じない．プロトタイピング用としては，これで十分であろう．

4.4 並列性について

ABCLは並列・分散処理を一つの特徴にしている言語であるが，今回はその能力を発揮する場がなかった．今仮に，受付係のほかに電話係が何人かおり，それぞれが電話で外部から注文を受けて受付係に知らせるように仕様が変更されたとする．この仕様の変更に対しては，次の電話オブジェクトを必要数作るだけでよい．このような状況下でこそABCLの並列性が生かされるのである．

```
[object 電話
  (=> [出庫依頼：＊銘柄 ＊本数 ＊送り先 ＊依頼者]
      [受付係 <=
        [出庫依頼：＊銘柄 ＊本数 ＊送り先 ＊依頼者]
      ])]
```

謝辞 ABCL の設計を進めるにあたって，三ツ井欽一，金田重治，大澤一郎の三氏をはじめとする東京工業大学情報科学科米澤研究室の方々との討論が大変参考になった．また金田重治氏には，ABCL 支援環境の設計・実現に際して多大な貢献をして頂いた[1]．これらの方々に深く感謝の意を表します．

参 考 文 献

1) 金田重治：GAE：A Graphic Display Oriented Programming Environment System, 東京工業大学情報科学科修士論文 (1985).
2) Krasner, G.: Smalltalk-80—Bits of History, Words of Advice, Addison Wesley (1983).
3) Goldberg, A. and Robson, D.: Smalltalk-80—The Language and Its Implementation, Addison Wesley (1983).
4) 松田裕幸，米澤明憲：ABCL ユーザズマニュアル，内部資料 (1984).
5) 三ツ井欽一：An Object Oriented Approach for Natural Language Comprehension, 東京工業大学情報科学科修士論文 (1985).
6) 大澤一郎，米澤明憲：オブジェクト指向方式による対話理解システム，コンピュータソフトウェア，Vol. 2, No. 1, pp. 11-28 (1985).
7) Programmer's Reference Manual for Sun Windows—the Sun Windows System (release 1.1), Sun Microsystems Inc. (1984).
8) 田胡和哉，益田隆司：オペレーティング・システムの構造記述に関する一試み，情報処理学会論文誌，Vol. 25, No. 4, pp. 524-534 (1984).
9) Weinreb, D. and Moon, D.: Lisp Machine Manual (fourth edition), Symbolics Inc. (1981).
10) 山崎利治：設計方法解説のための共通例題，プログラム設計技法の実用化と発展シンポジウム，情報処理学会 (1984).
11) 米澤明憲：オブジェクト指向型プログラミングについて，コンピュータソフトウェア，Vol. 1, No. 1, pp. 29-41 (1984).
12) Yonezawa, A., Matsuda, H. and Shibayama, E.: An Object Oriented Approach for Concurrent Programming, Res. Rep. on Inf. Sci., Tokyo Institute of Technology, No. C-63 (1984).
13) Yonezawa, A., Matsuda, H. and Shibayama, E.: Discrete Event Simulation Based on an Object Oriented Parallel Computation Model, Res. Rep. on Inf. Sci., Tokyo Institute of Technology, No. C-64 (1984).

(昭和 60 年 2 月 15 日受付)