ソフトウェア工学 58－5
（１９８８．２．４）

概念による設計法（ＤＭＣ）による在庫管理システムの記述

大林　正晴

㈱管理工学研究所

本稿では、オブジェクト指向的な設計方法論ＤＭＣによる在庫管理問題の解法および関数型言語ＭＬによる仕様記述の報告をする。筆者らは、問題に含まれる＜概念＞に着目し、プログラム構造を設計することを提唱している。今回は、共通問題を本手法で解いてみた。実際には、問題の分析、単語（概念）の整理、概念構造図作成、単語の割り付け、各概念の詳細な定義といった手順で設計を行なう。これらを在庫管理問題に即して説明する。記述実験の結果、ＤＭＣにより問題の構造を反映したプログラム構造を作りだすことができた。また、理解しやすい宣言的な仕様記述が得られた。

Declarative Specification of an Inventory Control System by the Design Method based on Concepts

Masaharu Ohbayashi
Kanrikogaku,Ltd.
2-2-2,Sotokanda,Chiyoda-ku,Tokyo 101,Japan

This paper presents how to solve the problem of an inventory control system and describes its specification by the declarative language ML. The "Design Methodology based on Concepts which handle the given problem have been proposed by us. We have applied the methodology to a bench mark problem.

First, we analyze the problem and find the instances which are denoted by words. Then we layout these words to create conceptual structure and assign the other words as value or type to those instances. Lastly, declarative specification of each instance is written by ML. According to this process flow, we explain the case of solving an inventory control system.

As a result, we could get the fine program structure reflecting to the problem domain structure. Also, we have known that declarative specification is easily understandable.

0．はじめに

概念による設計法（DMC）による問題解法は、与えられた問題の構造を素直にプログラム設計に反映させる1つの手法である。ここで取り上げる問題は在庫管理システムであるが、在庫管理の最適なモデル化を追及することが目的ではなく、在庫管理を1つの題材にしてどのようにプログラム構造を、あるいは、形式的な仕様記述を導くかを示すことがねらいである。また、宣言的な記述で、仕様を形式的に書く際の、記述のしやすさ、表現力、分りやすさなどを総合的に評価するための記述実験が目的である。

本稿では、理解を容易にするために核言語の予約語以外は、原則として漢字を使用した。現在試作中の支援系では、英数字のみしか使用できないので、実際には漢字を英文字列に置き換えて入力する必要がある。また、ここで述べるようなテキスト形式で仕様記述を入力するのではなく、構文誘導型エディタを介して行われる（意味的に同等な記述となる）。

なお、本研究の一部はIPAの委託による協同システム開発㈱のｿﾌﾄｳｪｱ環境統合化技術開発計画によるものである。

1．DMCによる設計手順

まず、プログラムの設計をどのような手順で進めるべきかについて簡単に説明する。カッコ内は、支援系を使った時の対応する手順である。

① まず、問題を具体的な<もの>に即して整理する。当然のことながら、最初から抽象的な概念を見つけるのは困難なので、初めは、具体的なものを登場させて分析し、モデルを作成する。問題の処理過程を検討しながら、入力データや処理アルゴリズムを確認する。（単語を分類、整理する）

② 分析・整理した結果、登場した<もの>を、いくつかの<もの>の集まりに分割する。（<もの>をインスタンスと呼び、それらをレイアウトする）

③ ②で定義した<もの>上で処理アルゴリズムが実現されることを確認する。

① <もの>間の相互関係を考えながら、各<もの>に必要な機能およびその機能の実行を指示するメッセージを決定する。（その名前をインスタンス定義表のｖａｌにセットする）

⑤ 各<もの>に固有のデータがあれば、データ型として定義する。（その名前をインスタンス定義表のｔｙｐｅにセットする）

⑥ 各機能の入力データ、出力データなどの詳細を決め、<もの>ごとに仕様を記述する。（構文誘導エディタでインスタンス定義表の記号群、宣言群の詳細を記述する）

⑦ 似た<もの>や本質的に同じ<もの>、一部だけ異なった<もの>などを整理・統合する。<もの>を抽象化して<概念>として部品化する。（記号群名、モジュール名を決定する）

2．問題の分析

2．1 在庫管理問題

つぎのように在庫管理問題の概略の要求仕様が自然言語により与えられている（下線は、注目した単語である）。

> ある酒類販売会社の倉庫では、毎日数個のコンテナが搬入されてくる。その内容はビン詰めの酒で、1つのコンテナには10銘柄（＝品名）まで混載できる。扱い銘柄（＝品名）は約200種類ある。倉庫係は、コンテナを受取りそのまま倉庫に保管し積荷票を受付係へ手渡す（＝積荷受理）。また受付係からの出庫指示によって内蔵品を出庫することになっている。内蔵品は別のコンテナに詰め替えたり、別の場所に保管することはない。空になったコナテナはすぐに搬出される。
>
> 積荷票： コンテナ番号（5桁）
> 搬入年月、日時
> 内蔵品名、数量（の繰り返し）
>
> さて受付係は毎日数10件の出庫依頼（＝注文）を受け、その都度倉庫係へ出庫指示書を出すことになっている。出庫依頼（＝注文）は出庫依頼票または電話（＝注文受理）によるものとし、1件の依頼（＝注文）では、1銘柄（＝品名）のみに限られている。在庫が無いか数量が不足の場合には、その旨依頼者に電話連絡し同時に在庫不足リストに記入（＝記録）する。そして当該品の積荷が必要量あった（＝在庫有り？）時点で、不足品の出庫指示をする。また空になる予定のコンテナを倉庫係に知らせることになっている。
>
> 出庫依頼（＝注文）： 品名、数量
> 送り先名
>
> 受付係の仕事（在庫なし連絡、出庫指示書作成および在庫不足リスト作成）のための計算機プログラムを作成せよ。
>
> 出庫指示書： 注文番号
> 送り先名
> コンテナ番号 （＝コンテナ札）
> 品名、数量 （の繰り返し）
> 空コンテナ搬出マーク
>
> 在庫不足リスト： 送り先名
> 品名、数量

本稿では、問題をより明確にするために、さらに、問題の補足説明をつぎのように想定する。

「受付係は、計算機の中に格納されている受付帳簿をもとに在庫管理を行う。積荷が搬入されたら、受付帳簿を更新し未出庫の調査を行う。その際、可能なものは出庫する。注文があったら、その品名の在庫があることを確認したあと、出庫指示書の作成や在庫の更新などの出庫準備をする。在庫がなければ、未出庫として在庫不足リストに記録しておく」。

2．2 制限事項

ここでの仕様記述では、つぎの点を仮定した。

・積荷票の内蔵品名と数量の繰り返しには、同一の銘柄の重複はないものとする。
・出庫依頼は、注文を受理した順に処理するが、在庫不足リストに登録されたものに関しては優先順位を設けず、出庫の順番は問わないものとする。
・内蔵品をどのコンテナから取りだすかは、自由であるとする。
・「数１０件の依頼」や「銘柄が約２００種類」などの数に関する情報は、考慮にいれない。

2．3 問題の整理

まず、問題を具体的な＜もの＞に即して整理するため、概念あるいはものとなりそうなものをこの中から拾い出していく。

たとえば、「酒類販売会社、倉庫、数個、コンテナ、搬入、内容、ビン詰、酒、銘柄、混載．．．」などといったぐあいに概念を挙げることができる。

この問題では、下線で示した単語が、最終的に選択され使用された概念やものなどである。

これらの単語を分類整理する前に、問題をより明確にするために、互いの関連を図１に示しておく。以下に述べる内容は、受付を最上位概念としてモデル化を行なう。

3．単語の整理

つぎに、これらの概念やものから、問題の構造を説明するために必要なより基本的な概念だけを選別する。その判断は、扱う問題により異なったり、いろいろなモデル化が可能であるため、一般的な選別基準を決めることはできない。けっきょく、設計者の判断に任されているわけである。しかし、従来の設計法でよく使われる機能中心の分割とは、やはり違ったものになる。基本的には、機能中心ではなく、機能とは直接関係ない＜概念＞を中心に考え、各機能はその＜概念＞に付随したものであるという見方をする。特殊な場合には１つの機能を１つの＜概念＞と考えることができる。

この問題では、

「受付、未出庫、倉庫、積荷、出庫準備、コンテナ、注文、在庫」

などが基本的な概念で、これらによって在庫管理というものが構成されると考える。これら以外には、より汎用的な概念として、

「品名、数量、時計」

といったものを登場させる。

型としては、まず、

「受付帳簿、積荷票、在庫不足リスト、倉庫、出庫指示書、コンテナ、コンテナ札、コンテナ番号、搬出マーク、在庫、注文、注文番号、送り先名、品名、数量、年月、日時」

などが考えられる。

残りの記号、

「注文受理、積荷受理、在庫有り？、調査、記録、出庫、指示書作成、更新、搬入、内容」

などを（関数を値とする）変数とする。

このように、概念やものを指示する単語を用途ごとに分類し、整理する。表１は、最終的に仕様記述が完了した時点での、単語表である。下線を引いた単語が前述の問題文の中に直接現れた単語であり、それ以外のものは、詳細な仕様記述の途中で必要に応じて登場させた単語である。

4．インスタンスのレイアウト

インスタンス（概念）として、採用したつぎの単語を２次元平面上にレイアウトする。

「受付、未出庫、倉庫、積荷、出庫準備、コンテナ、在庫、注文」

単語の配置の仕方は、原則として上位の概念を左上隅に置き、右下方向に行くに従って下位の概念となるように配置する。つまり、上位の概念を説明する際に使用する概念を下位に配置し、その参照の関係を結線して示す。概念は互に共有されることもあり、一般に、単純な木構造ではなく有向グラフになる。

この問題では、まず、全体を受付という概念で捉える。受付は、未出庫と倉庫と注文と言う概念だけを使って（それらの概念に所属する単語を使って）説明される。倉庫は、積荷、出庫準備、注文、在庫という概念を下位にもつと考える。さらに、積荷は、コンテナと在庫を下位の概念とし、出庫準備は、注文とコンテナを参照する。コンテナは、在庫という概念を使用して説明される。これらの関係を図示したものがつぎの図２である。

これらは、仕様記述の過程の試行錯誤を経て得られた結果の図であり、必ずしもこのような構造を最初に決定できるわけではない。

なお、グローバルな概念として品名、数量、時計を考えたが、これらは、レイアウトの右上側に配置して、他の任意の概念から参照できるようにしてある。

5．単語の割り付け

つぎに、分類整理した単語をレイアウト上の各インスタンスに割り付けていく。主に、型と（関数を値とする）変数の所属を決めることになる。このときには、目的の処理内容を念頭に置いて、やりたいことをどう実現するか考えながら行わなければならない。レイアウトした各インスタンスにそのやりたいことを機能分割して分担してもらうわけである。

この問題では、（最初の段階としては）つぎのような割り付けになろう。

・受付……受付帳簿　注文受理　積荷受理
・未出庫……在庫不足リスト　調査　記録

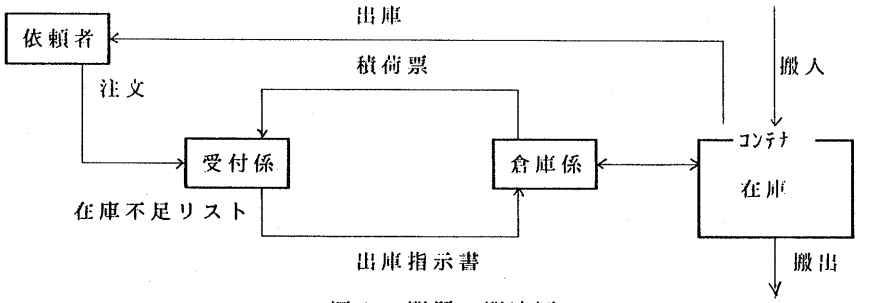

図１．問題の関連図

表１．在庫管理問題の単語表

word	global	instance	type	val
新倉庫	品名	受付	受付帳簿	注文受理
新在庫不足リスト	数量	未出庫	積荷票	積荷受理
出庫指示書リスト	時計	注文	在庫不足リスト	在庫有り？
在庫リスト		倉庫	倉庫	調査
コンテナリスト		積荷	品名	記録
新コンテナ		出庫準備	内蔵品リスト	出庫
新在庫リスト		コンテナ	出庫指示書	指示書作成
在庫リスト１		在庫	コンテナ	更新
残数量			コンテナ札	搬入
残数量１			コンテナ番号	新規
x			搬出マーク	登録
y			在庫	選択
z			注文	指示書
			注文番号	不足無
			送り先名	札作成
			日時	有り？
			数量	追加
			年月	追加０
				識別
				内容
				コンテナ箱

簿
票
庫
書
箱
札
受
内
出
留
内蔵
受注
空？
削除
帳簿

・組込の単語：bool int nil true false string

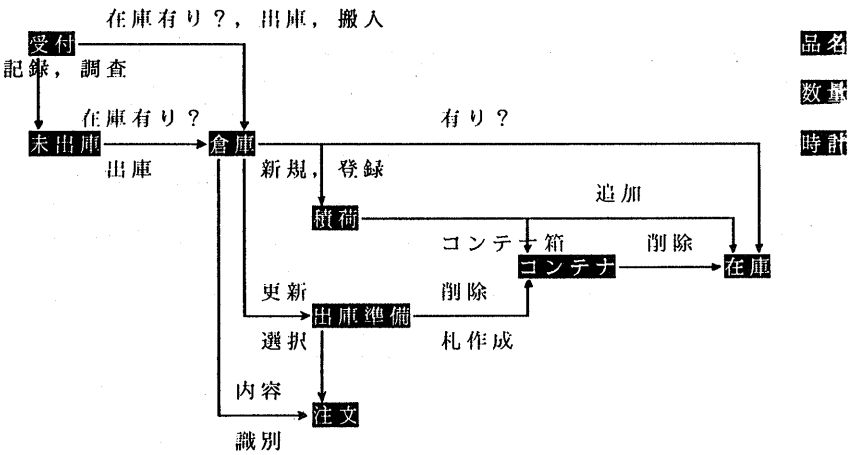

図２．在庫問題の概念構造図

・積荷……積荷票　新規　登録
・出庫準備……出庫指示書　更新
・コンテナ……ｺﾝﾃﾅ ｺﾝﾃﾅ札 ｺﾝﾃﾅ番号 搬出ﾏｰｸ
・在庫……在庫
・注文……注文　注文番号　送り先名　内容

6．インスタンスの定義

つぎに、各インスタンスの定義を核言語（標準ＭＬ）で記述する。記述形式は、できるだけ副作用を持たない宣言的な記述となるようにする。

前述の単語の割り付けで、すでに所属する主な単語が定まっているので、ここでは、まず、変数の型を決定し、記号群（signature）の定義を完成させる。つぎに、具体的な値の定義を行い宣言群（module）の詳細を記述する。

6．1 受付インスタンス

```
[記号群]
  signature 受付
  type 受付帳簿
①val 帳簿：倉庫 * 在庫不足ﾘｽﾄ → 受付帳簿
②val 注文受理：受付帳簿 * 注文 → 受付帳簿
  val 積荷受理：受付帳簿 * 積荷票 → 受付帳簿
  end
[宣言群]
  module 受付Mod
  instance 未出庫 倉庫 注文
③type 受付帳簿 = 簿 of（倉庫 * 在庫不足ﾘｽﾄ）
④val 帳簿 = 簿
⑤val 注文受理(簿(倉庫,在庫不足ﾘｽﾄ),注文) =
        if 在庫有り?(倉庫,注文)
        then 簿(出庫(倉庫,注文),在庫不足ﾘｽﾄ)
        else 簿(倉庫,記録(在庫不足ﾘｽﾄ,注文))
⑥val 積荷受理(簿(倉庫,在庫不足ﾘｽﾄ),積荷票)=
        簿(調査(搬入(倉庫,積荷票),在庫不足ﾘｽﾄ))
  end
```

① 帳簿（値変数）が倉庫（型）と在庫不足ﾘｽﾄ（型）を入力とし、受付帳簿（型）を出力とする関数型であることを宣言している。

② 注文受理（値変数）が受付帳簿（型）と注文（型）を入力とし、受付帳簿（型）を出力とする関数型であることを宣言している。

③ 受付帳簿（型構成子）に、簿（倉庫と在庫不足リストを引数とする値構成子）を束縛させる。これで、受付帳簿（型）のデータ（値）が、倉庫（型）と在庫不足リスト（型）の組からできていることがわかる。つまり、受付帳簿の内部表現が定義されたことになる。簿自体は、受付インスタンスの外部には見えない。

④ 帳簿（関数型の値変数）に、簿（値構成子）を束縛させる。これで、受付帳簿のデータ（値）の外部表現が定義されたことになる。つまり、倉庫と在庫不足リストを引数にして帳簿（関数型の値変数、つまり関数）を呼び出せば、受付帳簿（型）のデータ（値）が得られる。

⑤ 注文受理（関数型の値変数）に、右辺のif-then-else式を束縛させる。これは、関数定義の典型的な例である。仮引数には、

　簿(倉庫,在庫不足ﾘｽﾄ)

のようなパターン（変数体）も書ける。

この関数の処理内容はつぎの通りである。倉庫の注文の在庫があるか調べ、あれば、出庫した結果の倉庫と在庫不足リストで受付帳簿を作り値として返す。なければ、倉庫と在庫不足リストにその注文を記録したものとで受付帳簿を作り値として返す。

⑥ 積荷受理に、つぎの式を束縛させる。

　簿(調査(搬入(倉庫,積荷票),在庫不足ﾘｽﾄ))

つまり、倉庫と積荷票で搬入処理を行なった結果の値（倉庫型）と在庫不足リストで、出荷可能なものがあるか調査を行ない、その結果の値を新しい受付帳簿とする。

6．2 未出庫インスタンス

```
[記号群]
  signature 未出庫
  type 在庫不足ﾘｽﾄ
  val 不足無：在庫不足ﾘｽﾄ
①val 調査：倉庫 * 在庫不足ﾘｽﾄ
          → 倉庫 * 在庫不足ﾘｽﾄ
  val 記録：在庫不足ﾘｽﾄ * 注文 → 在庫不足ﾘｽﾄ
  end
[宣言群]
  module 未出庫Mod
  instance 倉庫　注文
②type 在庫不足ﾘｽﾄ is 注文 list
  val 不足無 = nil
③val 調査(倉庫,注文::在庫不足ﾘｽﾄ) =
      if 在庫有り?(倉庫,注文)
      then 調査(出庫(倉庫,注文),在庫不足ﾘｽﾄ)
      else let val (新倉庫,新在庫不足ﾘｽﾄ) =
                        調査(倉庫,在庫不足ﾘｽﾄ)
           in (新倉庫,記録(新在庫不足ﾘｽﾄ,注文))
           end
   |調査(倉庫,不足無) = (倉庫,不足無)
  val 記録(在庫不足ﾘｽﾄ,注文) = 注文::在庫不足ﾘｽﾄ
   |記録(不足無,注文)= [注文]
  end
```

① 調査の値は、倉庫（型）と在庫不足リスト（型）の組である。

② 在庫不足リストの内部表現の実体は、注文 listそのものであることを（透過）宣言している。

③ 在庫不足リストを順に調査し、在庫があれば、出庫処理をする。在庫がなければ、もう一度、在庫不足リス

トに記録しておく。

ここで、let-in-endは、局所的な値束縛宣言を行ないたい時に使用する。また、調査の定義は、引数のパターンによって場合分けされている。記述上は、｜で区切られている。

6．3 倉庫インスタンス

```
[記号群]
signature 倉庫
type 倉庫
val 倉庫：ｺﾝﾃﾅ list * 出庫指示書 list * 在庫 list
        → 倉庫
val 在庫有り?：倉庫 * 注文 → bool
val 出庫：倉庫 * 注文 → 倉庫
val 指示書作成：倉庫 * 注文 → 出庫指示書
val 搬入：倉庫 * 積荷票 → 倉庫
end
[宣言群]
 module 倉庫Mod
 instance 積荷 出庫準備 注文 在庫
 type 倉庫 = 庫 of (ｺﾝﾃﾅ list * 出庫指示書 list *
                    在庫 list)
 val 倉庫 = 庫
 val 在庫有り?(庫(_,_,在庫ﾘｽﾄ),注文) =
       let val(品名,数量) = 内容(注文)
       in 有り?(在庫ﾘｽﾄ,品名,数量)
       end
①val 出庫(倉庫 as
        庫(ｺﾝﾃﾅﾘｽﾄ,出庫指示書ﾘｽﾄ,在庫ﾘｽﾄ),注文) =
       let val(品名,数量) = 内容(注文)
       in 庫(更新(ｺﾝﾃﾅﾘｽﾄ,品名,数量),
           指示書作成(倉庫,注文)::出庫指示書ﾘｽﾄ,
           削除(在庫ﾘｽﾄ,品名,数量))
       end
②val 指示書作成(庫(ｺﾝﾃﾅﾘｽﾄ,_,在庫ﾘｽﾄ),注文) =
       let val(注文番号,送り先名) = 識別(注文)
           val(品名,数量) = 内容(注文)
       in 指示書(注文番号,
                 送り先名,選択(ｺﾝﾃﾅﾘｽﾄ,品名,数量))
       end
③val 搬入(庫(ｺﾝﾃﾅﾘｽﾄ,_,在庫ﾘｽﾄ),積荷票) =
       庫(新規(積荷票)::ｺﾝﾃﾅﾘｽﾄ,
                       登録(積荷票,在庫ﾘｽﾄ))
 end
```

① 出庫処理を行なう関数である。コンテナリストを更新し、指示書作成し、在庫リストから削除する。

② 出庫指示書を作成する関数である。選択関数でコンテナ搬出マーク処理を行なう。

③ 搬出処理を行なう関数である。積荷票をもとに、新規にコンテナをコンテナリストに登録する。また、在庫リストも更新する。

6．4 積荷インスタンス

```
[記号群]
 signature 積荷
 type 積荷票
 val 積荷表：ｺﾝﾃﾅ番号 * 年月 * 日時 * 在庫 list
          → 積荷表
 val 新規：積荷票 → ｺﾝﾃﾅ
 val 登録：積荷票 * 在庫 list → 在庫 list
 end
[宣言群]
 module 積荷Mod
 instance コンテナ 在庫
 type 積荷票 = 票 of
          (ｺﾝﾃﾅ番号 * 年月 * 日時 * 在庫 list)
 val 積荷表 = 票
①val 新規(票(ｺﾝﾃﾅ番号,年月,日時,内蔵品ﾘｽﾄ)) =
       箱(ｺﾝﾃﾅ番号,年月,日時,内蔵品ﾘｽﾄ)
②val 登録(票(_,_,_,内蔵品ﾘｽﾄ),在庫ﾘｽﾄ) =
       追加(在庫ﾘｽﾄ,内蔵品ﾘｽﾄ)
 end
```

① コンテナを新規に作る関数である。

② 在庫リストに内蔵品を登録する関数である。

6．5 出庫準備インスタンス

```
[記号群]
 signature 出庫準備
 type 出庫指示書
 val 指示書：注文番号 * 送り先名 * ｺﾝﾃﾅ札 list
          → 出庫指示書
 val 更新：ｺﾝﾃﾅ list * 品名 * 数量
          → ｺﾝﾃﾅ list
 val 選択：ｺﾝﾃﾅ list * 品名 * 数量
          → ｺﾝﾃﾅ札 list
 end
[宣言群]
 module 出庫準備Mod
 instance ｺﾝﾃﾅ 注文
 type 出庫指示書 = 書 of (注文番号 *
                          送り先名 * ｺﾝﾃﾅ札 list)
 val 指示書 = 書
①val 更新(ｺﾝﾃﾅﾘｽﾄ,品名,0)= ｺﾝﾃﾅﾘｽﾄ
   ｜更新(ｺﾝﾃﾅ::ｺﾝﾃﾅﾘｽﾄ,品名,数量) =
      let val (新ｺﾝﾃﾅ,残数量) =
                削除(ｺﾝﾃﾅ,品名,数量)
      in if 空?(新ｺﾝﾃﾅ)
         then 更新(ｺﾝﾃﾅﾘｽﾄ,品名,残数量)
         else 新ｺﾝﾃﾅ::更新(ｺﾝﾃﾅﾘｽﾄ,品名,残数量)
      end
```

```
②val 選択(ｺﾝﾃﾅﾘｽﾄ,品名,0) = nil
    |選択(ｺﾝﾃﾅ::ｺﾝﾃﾅﾘｽﾄ,品名,数量) =
      let val(ｺﾝﾃﾅ札 as 札(_,_,個数,_),残数量) =
                              札作成(ｺﾝﾃﾅ,品名,数量)
      in if 個数 = 0
         then 選択(ｺﾝﾃﾅﾘｽﾄ,品名,残数量)
         else ｺﾝﾃﾅ札::選択(ｺﾝﾃﾅﾘｽﾄ,品名,残数量)
      end
  end
```

①コンテナリストから、該当品を削除する関数である。
②コンテナリストから、該当品を削除した際の空コンテナに札を付ける関数である。

6．6 コンテナインスタンス

```
[記号群]
signature コンテナ
type ｺﾝﾃﾅ
type ｺﾝﾃﾅ札
type ｺﾝﾃﾅ番号
type 搬出ﾏｰｸ
val 箱 : ｺﾝﾃﾅ番号 * 年月 * 日時 * 在庫 list → ｺﾝﾃﾅ
val 札 : ｺﾝﾃﾅ番号 * 品名 * 数量 * 搬出ﾏｰｸ → ｺﾝﾃﾅ札
val 空? : ｺﾝﾃﾅ → bool
val 削除 : ｺﾝﾃﾅ * 品名 * 数量 → ｺﾝﾃﾅ * 数量
val 札作成 : ｺﾝﾃﾅ * 品名 * 数量 → ｺﾝﾃﾅ札 * 数量
end
[宣言群]
module コンテナMod
instance  在庫
type ｺﾝﾃﾅ = 箱 of (ｺﾝﾃﾅ番号 * 年月 *
                              日時 * 在庫 list)
type ｺﾝﾃﾅ札 = 札 of (ｺﾝﾃﾅ番号 * 品名 *
                              数量 * 搬出ﾏｰｸ)
type ｺﾝﾃﾅ番号 is int
type 搬出ﾏｰｸ = 出 | 留
val 空?(箱(_,_,_,nil)) = true
   |空?(箱(_,_,_,_)) = false
val 削除(箱(ｘ,ｙ,ｚ,在庫ﾘｽﾄ),品名,数量) =
      let val (新在庫ﾘｽﾄ,残数量) =
                在庫.削除(在庫ﾘｽﾄ,品名,数量)
      in (箱(ｘ,ｙ,ｚ,新在庫ﾘｽﾄ),残数量)
      end
val 札作成(箱(ｺﾝﾃﾅ番号,年月,日時,在庫ﾘｽﾄ),
                                    品名,数量) =
      let val (新在庫ﾘｽﾄ,残数量) =
                在庫.削除(在庫ﾘｽﾄ,品名,数量)
      in let val 搬出ﾏｰｸ = if 新在庫ﾘｽﾄ = nil
                          then 出
                          else 留
         in (札(ｺﾝﾃﾅ番号,品名,
                              数量 - 残数量,搬出ﾏｰｸ),残数量)
         end
      end
end
```

6．7 在庫インスタンス

```
[記号群]
signature 在庫
type 在庫
val 内蔵 : 品名 * 数量 → 在庫
val 有り? : 在庫 list * 品名 * 数量 → bool
val 追加 : 在庫 list * 在庫 list → 在庫 list
val 削除 : 在庫 list * 品名 * 数量
          → 在庫 list * 数量
end
[宣言群]
module 在庫Mod
type 在庫 = 内 of (品名 * 数量)
val 内蔵 = 内
val 有り?(内(品名,数量)::在庫ﾘｽﾄ,品名1,数量1) =
      if 品名 = 品名1
      then 数量 〉数量1
      else 有り?(在庫ﾘｽﾄ,品名1,数量1)
   |有り?(nil,_,_) = false
val 追加(在庫ﾘｽﾄ,内(品名,数量)::在庫ﾘｽﾄ1) =
      追加(追加O(在庫ﾘｽﾄ,品名,数量),在庫ﾘｽﾄ1)
   |追加(在庫ﾘｽﾄ,nil)= 在庫ﾘｽﾄ
val 追加O(内(品名,数量)::在庫ﾘｽﾄ,品名1,数量1) =
      if 品名 = 品名1
      then 内(品名,数量 + 数量1)::在庫ﾘｽﾄ
      else 内(品名,数量)::
                        追加O(在庫ﾘｽﾄ,品名1,数量1)
   |追加O(nil,品名,数量) = [内(品名,数量)]
val 削除(内(品名,数量)::在庫ﾘｽﾄ,品名1,数量1) =
      if 品名 = 品名1
      then let val 残数量 = 数量1 - 数量
           in if 残数量 〉= 0
              then (在庫ﾘｽﾄ,残数量)
              else (内(品名,- 残数量)::在庫ﾘｽﾄ,0)
           end
      else let val (在庫ﾘｽﾄ1,残数量1) =
                        削除(在庫ﾘｽﾄ,品名1,数量1)
           in (内(品名,数量)::在庫ﾘｽﾄ1,残数量1)
           end
   |削除(nil,_,数量) = (nil,数量)
end
```

6．8 注文インスタンス

```
[記号群]
signature 注文
```

```
type 注文
type 注文番号
type 送り先名
val 受注：注文番号 * 送り先名 * 品名 * 数量
        → 注文
val 識別：注文 → 注文番号 * 送り先名
val 内容：注文 → 品名 * 数量
end
```

[宣言群]

```
module 注文Mod
type 注文 = 受 of (注文番号 * 送り先名 *
                                    品名 * 数量)
type 注文番号 is int
type 送り先名 is string
val 受注 = 受
val 識別(受(注文番号,送り先名,_,_)) =
                                (注文番号,送り先名)
val 内容(受(_,_,品名,数量)) = (品名,数量)
end
```

7．記述結果の考察

この共通問題記述の目的は、形式的仕様に基づくソフトウェア開発の現実的なアプリケーションへの適用に関する実験を通して評価し、実用化のために克服しなければならない問題点を、明らかにすることであった。

ここでは、全体的なプログラムの構造、読み易さ、大きさ、記述能力などについて、つぎの４つの観点から分析した。

① 方法論

DMCのようなオブジェクト指向的な設計で得られたプログラム構造をＭＬのような関数型の特徴をもつ言語で、うまく記述できることが判明した。特に、単語の抽出やレイアウトなどを通して、問題の構造を反映したプログラム構造を作りだすことに成功している。このような手法は、十分一般性をもつものであり、他の問題記述にも適用できる。

② 記述性

この問題記述では、ＭＬの特徴である多様型を使用する場面がなかった。これは、問題がある程度限定された形で提示されており、より具体的な解を求めることが要請されていたからであろう。

また、今回の問題文ではエラー処理に関して具体的な要請が書かれていないので、ＭＬの例外処理機能を使用した記述は行わなかった。

以上のことから、核言語の記述性を十分例示することはできなかったが、基本的な要素だけでもかなりの記述能力があることが確かめられた。

③ 読み易さ

全体的に、副作用をもたない宣言的な記述となっているので、理解しやすい。記号群と宣言群が分離されているので、宣言部の詳細を知らなくても、使い方が記号群を見ればすぐに分る。

各関数の仕様記述は、最も大きいもので約１０行ぐらい、平均的には３～４行以内に収っている。このように非常に簡潔な記述になっているので読み易い。

④ 問題点

単語表による概念の分類は、大変有効であった。しかし、値変数（ｖａｌ）に相当する単語が比較的多くなるので、これらをもっと細分類するほうが、効果的ではないか。例えば、値構成子は別にする方がよいと思われる。

8．おわりに

現在、概念による設計法に従ったプログラム設計支援環境をワークステーション上に実現中である。特に、在庫管理問題の記述実験過程で得られた種々の知見が記述支援系の設計に反映されている。

今後は、仕様記述を実行させて、記述の正しさを調べることができる解析系や変換系を開発して行く予定である。

参考文献

〔1〕関根,大林,"新しいプログラム設計法---概念による設計法（DMC）"bit 1982年6月号 pp102-114.

〔2〕大林,"標準ＭＬを用いたプログラム設計支援環境"情報処理,ソフトウェア工学 51-1(1986.11.26)

〔3〕大林,"ＤＭＣ（概念による設計法）によるプログラム設計支援環境"情報サービス産業協会,ソフトウェア技術者協会,ｿﾌﾄｳｪｱｼﾝﾎﾟｼﾞｳﾑ87論文集 pp 181-190.

〔4〕松浦,中里,大林,"概念による設計法（DMC）に基づくプログラム開発環境の実現"情報処理,ソフトウェア工学 58-6(1988.2.4)

〔5〕R.Milner,"A proposal for standard ML"Conference Record of 1984 ACM Symposium on LISP and Functional Programming,ACM Aug.1984,pp 184-197.

〔6〕二村,他,"新しいプログラミング・パラダイムによる共通問題の設計"情報処理,Vol.26,No.5,pp.458-459.