ICOT Technical Memorandum: TM-0082

TM-0082

Concurrent Prologによる オンライン在庫管理システムの記述

大木優，竹内彰一，宮崎敏彦，古川康一
(ICOT)
二村良彦
(日立製作所)

November, 1984

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Concurrent Prologによる オンライン在庫管理システムの記述

大木 優[+]、 二村良彦[++]、 竹内彰一[+]、 宮崎敏彦[+]、 古川康一[+]
[+]新世代コンピュータ技術開発機構、 [++]日立製作所 中央研究所

1．はじめに

本報告では、CP(Concurrent Prolog) [1] によるオンライン在庫管理システムのプログラミングについて述べる。プログラミングに際して2種類のシステム設計法を用い、2つのモデルを作った。そして、モデルに対応した2つのプログラムを作成した。

プログラムの作成過程について考えてみると、プログラムの作成は問題を分割することの連続と考えられる。また、プログラムの作成過程は次の4つの段階を経て行われていると考えられる。

(1) 要 求 定 義：
　解くべき問題を分析し、要求仕様を明確にする。
(2) シ ス テ ム 設 計：
　要求仕様に基づき、システムをいくつかのモジュール（サブルーチンやオブジェクトなど）に分割する。
(3) プログラムの設計：
　モジュールを詳細化する。
(4) コ ー ディ ン グ：
　プログラム言語への変換を行なう。

各段階では、色々な方法が提案されている。システム設計の方法の一つにJackson が提案している方法がある[2]。彼の提案は、「システム設計の際、現実をモデル化すること」である。これは、現実にあるものを単位としてモジュール化することを意味しており、オブジェクト指向プログラミングとよく一致している。一方、従来からは機能分割によるシステム設計法が広く使われていた。以降、Jackson が提案している方法を構成的方法と呼び、従来の機能分割による方法を機能的方法と呼ぶ。これらの違いは次の通りである。

(1) 構成的方法
　現実を忠実にモデル化する。
(2) 機能的方法
　機能分割を行いながら、機能単位にモデル化する。

本報告では、まず、機能的方法と構成的方法の2つのモデルを作り、CPでプログラム化することを試みる。次に、プログラム開発で使用したCPのプログラミング・テクニック技法について簡単に述べる。CPによるオブジェクト指向プログラミング技法については付録に示す。

なお、本報告で使用した例題は、「プログラム設計技法の実用化と発展」シンポジウムでの設計方法を解説するための共通例題である。本報告で述べるCPのプログラムは、著者の一人がそのシンポジウムでCP風に日本語で記述したものを基にしたものである[3]。また、この例題についてBasicで記述したプログラムが文献［4］にある。

2．在庫管理システムのモデル化

2．1 仕様

ここでは在庫管理システムの仕様について述べる。「プログラム設計技法の実用化と発展」シンポジウムの共通例題での説明にもあるように、この仕様にはあいまいな部分や不明確な部分があり、それらは設計者が適当に解釈するようになっている。図1はその仕様の中核を示している。この仕様を要約すると次のようになる。

(1) 倉庫係
　① コンテナを受け取り、これを倉庫に保管し、積荷票を受付係に手渡す。
　② 受付係の指示で内蔵品を出庫する。
(2) 受付係
　① 出庫依頼を受け、倉庫係に出庫指示書を出す。
　② 在庫がないか、数量が不足の場合には、在庫リストを記入する。

倉庫係と受付係との関連を図2で示す。

2．2 在庫管理システムの2つのモデル化

モデル化とは、システムをどのように構成するかを考えることであり、システムを構成要素に分割することである。分割された構成要素はプログラム・レベルのサブルーチンやオブジェクトに対応する。

ここではモデル化に際して、構成的方法と機能的方法の2つの方法を使用した。これらの方法は次に示す通りである。

(1) 構成的方法
　「現実にあるものを単位としてモデル化すること」である。機能はモデル化した後に考える。
(2) 機能的方法
　システム全体の機能を考えて、機能を分割しながら、機能単位にモデル化する。

Jackson は構成的方法を提案するに当って、機能的方法には次のような欠点があることを指摘している。

(1) 機能分割は難しい。

システムの全機能は複雑でつかみきれない。

(2) 頻繁なシステムの機能変更に追従することが難しい。

機能変更がシステムの構造をかえる可能性が高い。

一方、構成的方法にはこのような欠点がなく、次のような利点があることを指摘している。

(1) 現実をモデル化する。

> ……毎日数個のコンテナが搬入されてくる。
>
> その内容はびん詰めの酒で、1つのコンテナには10銘柄まで混載できる。扱い銘柄は約 200種ある。倉庫係はコンテナを受け取り、そのまま倉庫に保管し積荷票を受付係へ手渡す。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。……
>
> さて、受付係は毎日数10件の出庫依頼を受け、その都度、倉庫係へ出庫指示書を出すことになっている。……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する。

(a) 要求仕様

(a) 入力

積荷票　：コンテナ番号(5桁)
　　搬入年月日、日時
　　内蔵品名、数量（の繰返し）
出庫依頼　：品名、数量
　　送り先名

(b) 出力

出庫指示書：注文番号
　　送り先名
　　コンテナ番号（の繰返し）
　　品名、数量
　　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

(b) データ構造

図1　在庫管理システムの仕様

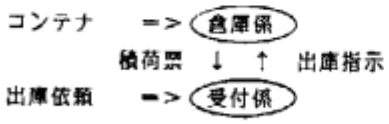


図2　倉庫係と受付係

モデル化が容易である。

(2) 機能的方法に比べてユーザからの仕様変更を柔軟に対処できる。

モデルが現実のシステムに近いため、変更箇所や変更内容および何を変更すべきかがわかりやすい。

前節の図2は、要求仕様をモデル化した図に相当するが、この図では仕様が不明確である。そこで図3のように、外部からメッセージを受け取る部分と倉庫を分けて考える。それぞれの構成要素では次の処理を行なう。

(1) 入庫受付係

① コンテナを受け取り、コンテナを倉庫に渡す。

(2) 出庫受付係

① 出庫依頼を受け、倉庫に出庫指示を出す。

(3) 倉　　庫

① コンテナ、積荷票を保管する。

② 出庫指示書を受け、内蔵品を出庫する。

③ 在庫がない場合、在庫不足として扱う。

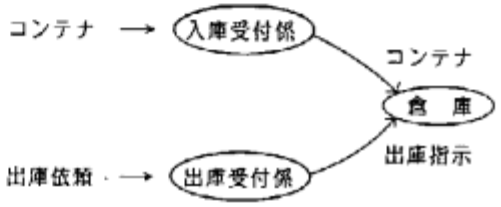


図3　モデル化

プログラムを書くためには、図3の倉庫をさらにモデル化する必要がある。倉庫をモデル化する前に、倉庫に関係するデータについて考えてみる。これらは、次の3つである。

(1) 内蔵品

倉庫に保管されている品目ごとの商品である。

(2) コンテナ

内蔵品を混載している。

(3) 積荷票

内蔵品とコンテナの対応を記録している。

これらをまとめると図4のような関係になる。

倉庫をモデル化するためには、先に述べた2つの方法を適用する。これらの方法の適用を次に示す。

(1) 機能的方法

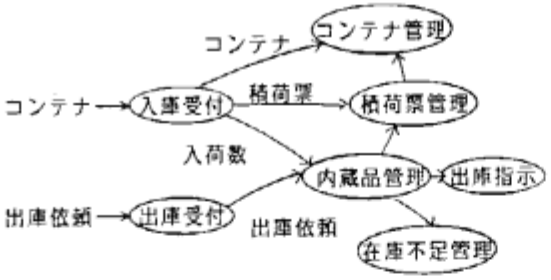
図4　コンテナ、積荷票、内蔵品の関係

倉庫の機能を機能分割し、分割された機能単位を構成要素とするモデルを考える。そうすると倉庫の構成要素は、コンテナや積荷票、内蔵品を管理する管理係となる。各管理係ではそれぞれの情報を管理したり、処理する。

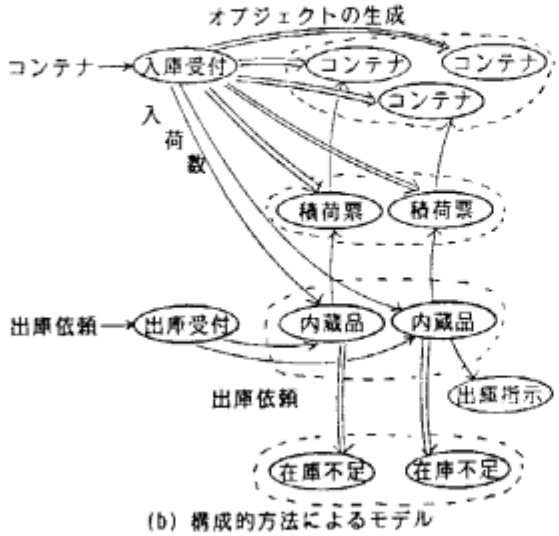
(2) 構成的方法

倉庫の中に存在する物を基本としてモデルを考える。そうすると倉庫は、コンテナや積荷票、内蔵品から構成されていると考えることができる。次に、それらの構成要素に機能を割当てる。割当てられる機能は、構成要素である情報自身の処理に対応する。

これらの方法によるモデルを図5のaとbに示す。機能的方法では、入庫受付係がコンテナを受け取ると、それぞれの管理係に必要なデータを送る。出庫受付係が出庫依頼を受け取ると、内蔵品管理係に出庫依頼を伝える。一方、構成的方法では、入庫受付係がコンテナを受け取ると、入庫したデータに従い、コンテナや積荷票そのものを生成する。そして内蔵品には入荷数を知らせる。また、出庫受付係が出庫依頼を受け取ると内蔵品に出庫依頼を伝える。図5のbでの点線は構成要素のグループを示す。



(a) 機能的方法によるモデル



(b) 構成的方法によるモデル

図5　機能的方法と構成的方法によるモデル

この2つの方法によるモデル化の難しさを比較すると、次の理由より機能的方法は構成的方法に比べて労力が多いと考えられる。

(1) 機能的方法

全体の機能を考え、機能分割する必要がある。その結果、情報を管理する1段レベルの高いものが必要となる。

(2) 構成的方法

現実にある物を基本としてモデルを考えればよい。機能についても同様に物を基本として考えればよいので、機能分割の作業は機能的方法よりも少ない。

しかし、この例題では全体の機能が大きくなかったので、モデル化の難しさに大きな差はなかった。

3．在庫管理システムのプログラム

CPによるオブジェクト指向プログラミング技法を使って、機能的方法と構成的方法で作成したモデルをプログラム化する。最後に、この2つの方法によるモデル化とプログラムを比較する。

3．1　機能的方法によるモデルのプログラム

在庫管理システムの機能的方法によるモデルの特徴は、情報を管理するための管理者がいることである。また、情報へのアクセスには、管理者を経由する必要があることである。ここで管理者は機能を分割した後のモジュールに対応する。機能的方法によるモデル化がプログラマなどに受け入れられやすい理由は、実際のオフィスでの業務分担のモデルに近いからであると思われる。しかし、実際のオフィスで、そのような業務分担を最初から考えることは難しい。このことは機能的方法でシステムを設計する時にも当てはまり、機能的方法による大規模なシステムのモデル化は難しいと考えられる。

このプログラムには、2章で述べた仕様以外に次の追加機能を持つ。

(1) 取り扱う銘柄や銘柄数をシステムの入力に従って動的に変更できる。

(2) [　]メッセージをオブジェクトが入力すると、そのオブジェクトは消滅する。

(3) 積荷票管理では、仕事がない間（メッセージが来ない間）に積荷票を並びかえる。これは、倉庫内のコンテナ数を少なくするために行なうものである。

ここで、オブジェクトとは入庫受付係や積荷票管理などである。

このプログラムのオブジェクトの構成を図6に示す。ここで示した構成は銘柄aとbを取り扱うシステムである。入庫受付係や出庫受付係、コンテナ管理以外のオブジェクトは実行時に作られる。これは追加機能(1) を実現するために、取扱う銘柄ごとにオブジェクトを動的に作り出すた

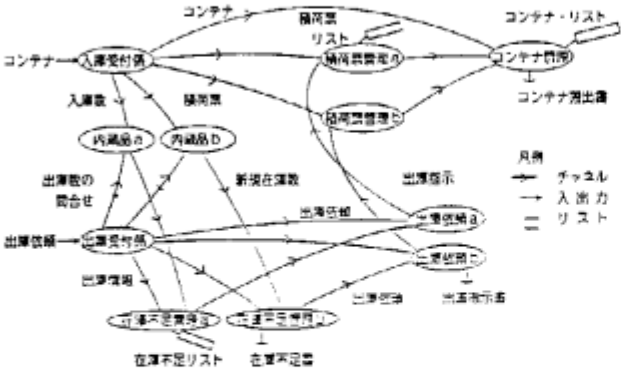


図6　機能的方法によるプログラムの
オブジェクトの構成

めである。コンテナや積荷票、在庫不足に関する情報はそれぞれの管理用オブジェクトで管理される。内蔵品は銘柄ごとに生成され、内蔵品の数を保持しているオブジェクトである。図中、←→で示されている線は、不完全メッセージによる双方向なデータの流れを示している。

次に、図6の各々のオブジェクトの処理概要を以下に示す。

(1) 入庫受付係

① コンテナが到着すると、

(i) 到着したコンテナに関する情報をコンテナ管理に送る。

(ii) コンテナに格納している銘柄ごとに、

(a) 入庫数量を銘柄に対応する内蔵品に知らせる。

(b) 積荷票管理に積荷票の内容を知らせる。

(2) 出庫受付係

① 出庫依頼要求を受け取ると、

(i) 内蔵品に出庫の問い合わせを行う。

(ii) 出庫が可能であれば、出庫依頼オブジェクトに出庫依頼を送る。

(iii) 出庫が不可能であれば、在庫不足管理に出庫依頼情報を送る。

(3) 内　蔵　品

① 入庫受付係より入庫数を受け取ると、

(i) 内蔵数を更新する。新内蔵数を現内蔵数＋入庫数とする。

(ii) 在庫不足管理に新内蔵数を知らせる。

(iii) 在庫不足管理により在庫不足分を差し引いた内蔵品の残数が戻されると、それを新内蔵数とする。

② 出庫受付係より出庫依頼を受け取ると、

(i) 内蔵数＞＝出庫依頼数ならば、

(a) 新内蔵数を内蔵数－出庫依頼数とする。

(b) 出庫受付係に”成功”を返す。

(ii) 出庫依頼数＞内蔵数ならば、出庫受付係に”失敗”を返す。

(4) 積荷票管理

① 入庫受付係より積荷票を受け取ると、それを積荷票リストに登録する。

② 出庫依頼オブジェクトより出庫指示を受け取ると、積荷票リストの最初の積荷票を取り出し、

(i) 出庫指示数＜積荷票の在庫数ならば、

(a) 積荷票の新在庫数を旧在庫数－出庫指示数とし、その積荷票を再度、積荷票リストに登録する。

(b) コンテナ管理に出庫数量などを知らせる。

(ii) 出庫指示数＝積荷票の在庫数ならば、コンテナ管理に出庫数量などを知らせる。

(iii) 出庫指示数＞積荷票の在庫数ならば、

(a) コンテナ管理に出庫数量などを知らせる。

(b) 新出庫指示数＝旧出庫指示数－積荷票の在庫数として、再度、出庫処理（②）を行なう。

③ 積荷票リストが変更され、メッセージが来ていないならば、積荷票リストの中の積荷票を在庫数の小さい順にソートする。

(5) 在庫不足管理

① 出庫受付係より出庫依頼情報を受け取ると、それを在庫不足リストに登録する。

② 内蔵品より在庫数を受け取ると、在庫不足リストの最初の出庫依頼情報を取り出し、

(i) 要求数（in出庫依頼情報）＜在庫数ならば、

(a) 出庫依頼オブジェクトに出庫依頼情報を送る。

(b) 在庫不足リストから出庫依頼情報を削除する。

(C) 新在庫数＝在庫数－要求数とし、再度、在庫不足処理を行なう。

(ii) 要求数＝在庫数ならば、

(a) 出庫依頼オブジェクトに出庫依頼情報を送る。

(b) 在庫残数を0とする。

(C) 在庫不足リストから出庫依頼情報を削除する。

(iv) 要求数＞在庫数ならば、在庫残数＝在庫数とする。

(6) 出庫依頼

① 出庫依頼を受け取ると、

(ⅰ) 積荷票管理に出庫指示を送る。
(ⅱ) 出庫指示書を出力する。

(7) コンテナ管理

① コンテナを受け取ると、コンテナ・リストに登録する。

② 出庫指示を受け取ると、コンテナ・リストにより該当コンテナを捜す。

(ⅰ) 商品を出庫後、コンテナ内数量が0になったならば、コンテナ撤出指示書を出す。

(ⅱ) 商品を出庫後、コンテナ内数量が0でないならば、再度、コンテナ・リストに登録する。

CPによるプログラムの一部を付録の図20に示す。

3．2 構成的方法によるモデルのプログラム

在庫管理システムの構成的方法によるモデルの特徴は、機能的方法によるモデルとは逆に、情報を管理するものがいないことである。情報自体がシステムの構成要素である。情報へのアクセスは情報そのものに対して行なう。ただし、ここでは情報そのものもオブジェクトであるので、アクセスはメッセージを通して行う。このモデルのわかりやすさは、倉庫には物が入っているという現実によく対応している点にある。

ここで述べるプログラムは、2章で述べた仕様以外の機能として前節の追加機能の(2)を含んでいる。このプログラムのオブジェクトの構成を図7に示す。図に示すように、コンテナや積荷票、在庫不足に関する情報そのものがオブジェクトになっている。これらは実行時に生成されたり、消滅される。

このプログラムの特徴の1つは、内蔵品から積荷票へのメッセージ送信方式がブロードキャスト方式であることである。ブロードキャスト方式のメッセージ送信とは、1対1の送信ではなく、複数のオブジェクトに一度にメッセージを送信する方式である。内蔵品が出庫依頼を受けた時、積荷票にブロードキャスト方式で出庫指示のメッセージを送信する。積荷票の中で、一番最初にメッセージを受けとった積荷票が商品を出庫する処理を行なう。これ以外の積荷票は、メッセージを読みとばす。このブロードキャスト方式のメッセージ送信は構成的方法のモデル化と大きな関係がある。構成的方法によるモデルでは、多くの場合、情報であるオブジェクト自体が主体的にその処理を行うからである。すなわち、誰がメッセージを処理するかは送信側で決めるのではなく、受信側で決めるのである。

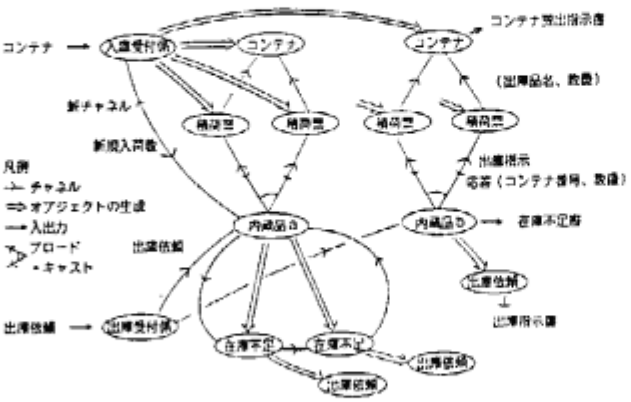


図7 構成的方法によるプログラムのオブジェクトの構成

一方、内蔵品から在庫不足へのメッセージ送信の方式はブロードキャスト方式ではない。在庫不足情報をチャネルのチェインでつなぎ、チェインに沿ってメッセージを流す方式である。ブロードキャスト方式にできない理由は、個々の在庫不足の不足数の状況が内蔵品にはわからないので、新しく入荷した数量によって在庫不足のどれが在庫不足を解消できたかがわからないためである。このように構成的方法でも、情報を管理する必要がある場合はある。しかし、同じ情報の管理でも、でき上がったモデルは機能的方法のものとは異なるだろう。在庫不足のモデルについては、図6と図7に示すような違いがある。図7では、在庫不足情報自体がオブジェクトの形で残っている。

次に、図7に各々のオブジェクトの処理概要を以下に示す。

(1) 入庫受付係

① コンテナが到着すると、

(ⅰ) コンテナを作る。

(ⅱ) コンテナに格納している銘柄ごとに、

(a) 内蔵品に入庫数量を知らせる。

(b) 積荷票を作る。

(2) 出庫受付係

① 出庫依頼が来ると、内蔵品に出庫依頼を知らせる。

(3) 内 蔵 品

① 入庫受付係より入庫数を受け取ると、

(ⅰ) 新在庫数を在庫数＋入庫数とし、在庫不足に新在庫数を知らせる。

(ⅱ) 在庫不足が解消されたために、出庫する必要が生じたならば、

(a) 積荷票に出庫指示を送る。

(b) 未出庫分があるならば、再度、積荷票に出庫指示を送る。

② 出庫受付係より出庫依頼を受け取ると、

(ⅰ) 出庫依頼数＞在庫数ならば、

(a) 出庫依頼情報を在庫不足として在庫不足のチェインの最後につける。

(b) 在庫不足書を出力する。

(ⅱ) 在庫数＞＝出庫依頼数ならば、

(a) 出庫依頼を生成する。
(b) 積荷票に出庫指示を送る。
(c) 未出庫分があるならば、再度、積荷票に出庫指示を送る。

(4) 積 荷 票
① 内蔵品より出庫指示を受け取ると、コンテナに搬出指示を送り、
(i) 出庫指示数＞＝積荷票の数量ならば、
(a) 未出庫数を出庫指示数－積荷票の数量とし、未出庫数を内蔵品に返す。
(b) 自分自身を消滅させる。
(ii) 積荷票の数量＞出庫指示数ならば、
(a) 未出庫数を0として内蔵品に返す。
(b) 積荷票の新数量を積荷票の数量－出庫指示数とする。

(5) 在 庫 不 足
① 在庫数を受け取ると、
(i) 在庫不足数＞在庫数ならば、
(a) 次の在庫不足へ在庫数を伝える。
(ii) 在庫数＞＝在庫不足数ならば、
(a) 出庫依頼を生成する。
(b) 新在庫数を在庫数－在庫不足数とし、次の在庫不足へ新在庫数を伝える。
(c) 自分自身を消滅させる。

(6) 出 庫 依 頼
① 出庫指示書を出力する。

(7) コ ン テ ナ
① 積荷票より搬出指示を受け取ると、コンテナの新数量をコンテナの数量－搬出数量とし、
(i) コンテナの新数量＝0ならば、自分自身を消滅させる。

CPによるプログラムと実行例を付録の図21と図22に示す。

3．3　2つの方法の比較

プログラム開発に関する機能的方法と構成的方法の比較は、プログラム開発全体にかかわる労力によって比較するべきであろう。しかし、ここでは、モデルの作成とプログラムの作成の労力に限って比較する。

モデルの作成の定性的比較は、2章で述べたように、大きな差がなかった。これは次の理由によるものと考えている。

(1) 例題として与えられた仕様の全体の機能が大きくなかった。
(2) システムの構成要素が容易に想像できた。

しかし、機能的方法は全体の機能を分割する作業が必要である。一方、構成的方法は物を単位として機能を考えるため、機能分割に関する作業は機能的方法に比べて少ない。システムの機能が大きくなれば、機能的方法では構成的方法よりモデル化が難しくなると思われる。

プログラムの作成の労力は、プログラムの規模によって定量的に比較できるだろう。ここでは、プログラムの規模の尺度としてステップ数を使う。2つの方法によるプログラム・ステップ数は次の通りである。

(1) 機能的方法によるプログラムのステップ数
：129ステップ

(2) 構成的方法によるプログラムのステップ数
：77ステップ

2つのプログラムとも、例題の仕様に関する部分のステップ数である。機能的方法のステップ数が多い理由は、機能的方法によるモデルが情報を管理しているためだと思われる。例えば、コンテナについて見てみると、機能的方法によるプログラムは図8となり、構成的方法によるプログラムは図9となる。これらのステップ数の比で2対1である。これは、コンテナ管理ではコンテナ・リストとしてコンテナを管理しているためにステップ数が増加したためである。コンテナ管理は、機能的方法によるモデルの構成要素の一つである。一方、図9のコンテナ自体は誰にも管理されていないため、その分のステップ数がない。コンテナは、構成的方法によるモデルの構成要素の一つである。機能的方法のように、情報を管理するとプログラムのステップ数が増える原因になる。機能的方法によるモデル化は、管理し

```
container([container(Container_number,Total)|C],Container_list) :-
        container(C7,[(Container_number,Total)|Container_list]).

container([out(Container_number,Item,Count)|C],Container_list) :-
        find_container(Container_number,Total,Container_list,Container_list_1),
        Total_1 := Total - Count & Total_1 =:= 0 |
        container(C7,Container_list_17).

container([out(Container_number,Item,Count)|C],Container_list) :-
        find_container(Container_number,Total,Container_list,Container_list_1),
        Total_1 := Total - Count & Total_1 > 0 |
        container(C7,[(Container_number,Total_1)|Container_list_1]).

container([],List) :-
        container_monitor(List).

find_container(Container_number,Total,
               [(Container_number,Total)|Container_list],Container_list).
find_container(Container_number,Total,
               [(C,T)|Container_list],[(C,T)|Result]) :-
        C \= Container_number |
        find_container(Container_number,Total,Container_list,Result).
```

図8　コンテナ管理の定義

```
container(Container_number,[total(Total)|C],_) :-
        container(Container_number,C7,Total7).

container(Container_number,[out(Item,Count)|C],Total) :-
        Total_1 := Total - Count & Total_1 =:= 0 | true.

container(Container_number,[out(Item,Count)|C],Total) :-
        Total_1 := Total - Count & Total_1 > 0 |
        container(Container_number,C7,Total_1).

container(Container_number,[],Total) :-
        container_monitor(Container_number,Total).
```

図9　コンテナの定義

なくてもよい情報まで管理してしまう傾向がある。

4．CPのいくつかのプログラミング・テクニック

在庫管理システムのプログラムを作成中に使用したCPのプログラミング・テクニックについて簡単に説明する。

(1) 返信用論理変数付きメッセージ送信

これは、送信するメッセージの中に受信側でinstantiate される論理変数を含むメッセージ送信である。図10に例を示す。図中、Ackが返信用論理変数である。送信側のオブジェクトでは、Ackの値が必要になる場所で waitする。waitはCPのprimitive predicate でwaitの引数がinstantiate していないならば、instantiate されるまで待つ。もし受信側オブジェクトでAckに値がinstanitate されていないならば、そこでAckがinstantiate するまで処理をwaitする。

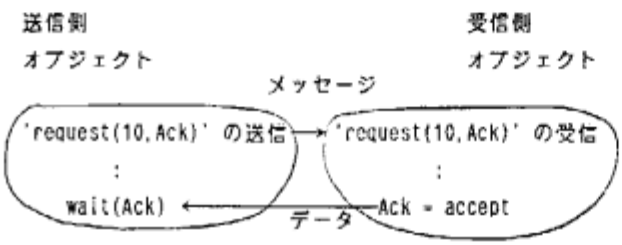


図10 返信用論理変数付きメッセージ送信の例

(2) 追伸用論理変数付きメッセージ送信

これは、送信するメッセージの中に送信側でメッセージ送信後にinstantiate する論理変数を含むメッセージ送信である。図11に例を示す。図中、Total が追伸用論理変数である。受信側のオブジェクトでは、その値が必要になる場所でwaitする。

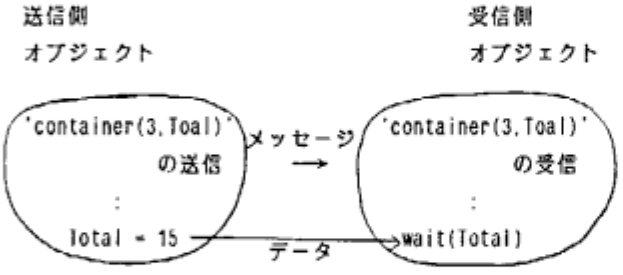


図11 追伸用論理変数付きメッセージ送信の例

(3) 局所的オブジェクト

ここで、局所的オブジェクトとは、そのオブジェクトへメッセージを送信できるオブジェクトが限られているオブジェクトのことを言う。図12の例では、在庫不足オブジェクトがこれに相当する。局所的オブジェクトを使う利点は、1つのオブジェクトをさらにモジュール分割することやオブジェクト内に持たなければならない局所的情報をオブジェクトとして保持することが可能となることである。

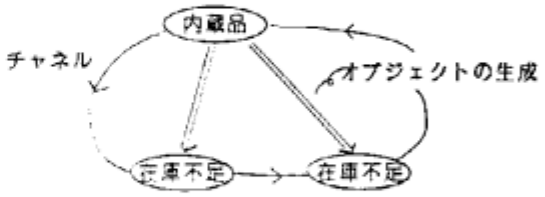


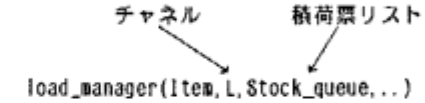
図12 局所的オブジェクトの例

(4) アイドル (idle) 処理

通常、オブジェクトはメッセージを受信しないと処理を行わない。アイドル処理は、これとは逆にメッセージが来ない時に行なう処理のことである。図13を使ってアイドル処理を説明する。ここで、論理変数Lはチャネルとして使われている。Lがinstantiate されていないならば（①）、積荷票リストをソートする（②）。そして、チャネルLがinstantiate されるまで待つ（③）。instantiate されると、 そのメッセージを処理する（④）。積荷票リストのソートはガード内で行なっているので、ソート処理中にチャネルLがinstantiate されると、instantiate されたメッセージで起動される別の候補節の実行も並列に始まる。どの節がコミットされるかは、プログラムと実行のスケジューリング次第である。

(5) オブジェクトの消滅

プログラムではオブジェクトが不必要になれば、それ




```
load_manager(Item,L,Stock_queue,..) :-
①    variable(L),
②    load_sort(Stock_queue,Sorted_stock_queue) |
③    wait(L) &
④    load_manager(Item,L,Sorted_stock_queue,..).
```

図 13 アイドル処理の例

```
merge([X|Xs],Ys,[X|Zs]):- merge(Ys,Xs?,Zs).
merge(Xs,[Y|Ys], [Y|Zs]):-merge(Ys?,Xs,Zs).
merge(Xs,[ ],Xs).
merge([ ],Ys,Ys).
```

図14 merge プログラム

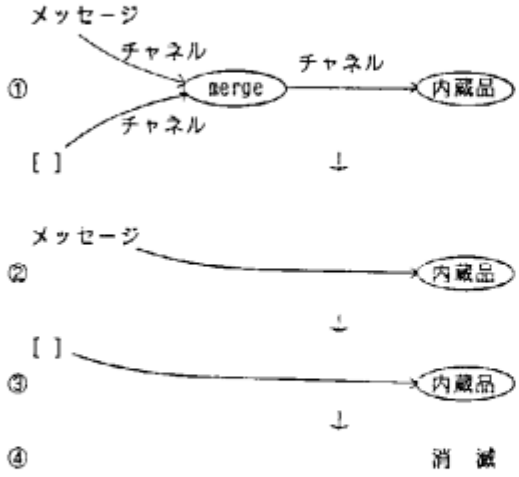


図15 オブジェクトの消滅

を消滅するのが望ましいであろう。ところが、CPによるオブジェクトはすべて並列に動いており、メッセージも並列に飛びまわっているので、オブジェクトを消滅する時期を判断することがむずかしい。これは、あるオブジェクトにとって、他のオブジェクトから二度とメッセージが来ないことを知ることが難しいためである。そこで、オブジェクトへのチャネルを閉じていくことによりメッセージの終了を検知することとした。これには、merge プログラムが重要な役割を果たす。merge プログラムは、２つのチャネルを１つにマージするものであるが、[ ] メッセージを受信するとmerge は消滅して、チャネルが直結する（図14参照）。オブジェクトを消滅させるメッセージとして[ ]メッセージを使うと、オブジェクトに[ ]メッセージが来たことを、そのオブジェクトへのチャネルがすべて閉じた信号として使うことができる。図15にその経過を示す。あるオブジェクトが別のオブジェクトを使わなくなったことを知らせるために、[ ]メッセージを使う。最初、２つのチャネルがオブジェクトにつながっていたが、片方から[ ]メッセージが来ると（①）、merge プロセスが消滅し直通となる（②）。さらに残りのチャネルからも[ ]メッセージが来ると（③）、[ ]メッセージを受信したオブジェクトは、そのメッセージを消滅メッセージだと判断し、消滅する（④）。

5．結 言

在庫管理システムの２つのモデルを作り、CPでそれらのプログラムを作成した。モデルの一つは、Jackson の提案に従った構成的方法によるものである。もう一つは、従来から行なわれている機能の分割に従った機能的方法によるものである。

Jackson の提案は、「現実にあるものを単位としてモデル化すること」であり、まさにオブジェクト指向そのものと言える。在庫管理システムでは、構成的方法はモデル化の労力を従来の機能的方法に比べて大きく軽減はしなかった。しかし、大規模なシステムでは、機能的方法のモデル化に比べてモデル化の労力を軽減するだろう、という感触を得た。また、プログラムの規模で見ると、構成的方法は機能的方法に比べて約３分の２のステップ数であった。これは、機能的方法にありがちな情報の冗長な管理が減少したためと思われる。

CPは、構成的方法に従って現実にあるものをオブジェクトとしてプログラミングする際、有効であった。CPでは、現実にあるもの、例えば、コンテナ自体をオブジェクトとしてプログラム化することが容易にできた。また、４章で示したように、プログラムを作る際いくつかの有効なプログラム・テクニックがあることもわかった。しかし、デバッグ・ツールやライブラリが、まだ不完全である。CPを真のユーザ言語にするためにはデバッグ・ツールやライブラリをさらに完備する必要があると考えている。

6．参考文献

[1] 竹内 ： 論理型並列プログラミング言語 ――Concurrent Prolog 、コンピュータソフトウェア、 1巻 2号 25－37

[2] Jackson M.A. ; Information Systems:Modelling, and Transformations, Proc.3rd International Conference on Software Engineering ,1978,IEEE

[3] 二村，藤田 ： プログラム設計法PAD/PAH とその効果、「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、1983年 4月

[4] 二村 ： プログラム設計法PAD/PAH 、情報処理学会誌、1984年11月号 1237-1246

[5] 竹内 ： Concurrent Prolog によるオブジェクト指向プログラミング、bit 、15巻12号 62－70

[6] 米沢 ： オブジェクト指向プログラミングについて、コンピュータソフトウェア、 1巻 1号 29－41

7．謝 辞

本報告に関して有益な討論をしていただいたICOT Working Group 2の委員の方々およびICOT第２研究室の方々に感謝致します。また、本報告に関して有益なコメントをしていただいた富士通国際情報社会科学研究所の田中二郎氏およびICOT第２研究室の近藤浩康氏に感謝致します。

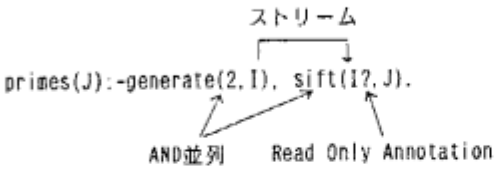


図16　CPのプログラム例

8．付　録

8．1　CPによるオブジェクト指向プログラミング

本報告で述べたプログラミングは、CPによるオブジェクト指向プログラミング技法を使ったものである。まず、CPの概要を述べた後、CPによるオブジェクト指向プログラミングについて述べる[5]。

8．1．1　CPの概要

CPは、共有論理変数によるメッセージ・パッシングを行なうストリーム AND並列型論理プログラミング言語である。CPの特徴をまとめると次の通りである。

(1) AND 並列　　：　複数ゴール（プロセス）を並列に解く。
(2) OR並列　　：　ゴールの候補節を並列に解く。
(3) 共有論理変数　：　プロセス間通信チャネルに使う。
(4) Read Only Annotation　：　プロセス間の同期を行なう。

図16にCPの節の例を示す。これは、エラトステネスのふるいのアルゴリズムに基づいた素数生成プログラムの一部である。generate述語で整数の列を論理変数Ｉに生成する。sift述語ではこの整数列を使って素数を計算する。"，"で結ばれたgenerate述語とsift述語は並列に処理される。論理変数Ｉはgenerate述語とsift述語の共有論理変数で、generate述語とsift述語間のチャネルとなっている。この論理変数Ｉを通して整数の列がストリームとして流れる。

図17　オブジェクト

```
内部状態
↓
counter([up | S],State) :-
    NewState is State+1,counter (S?,NewState).
```

図18　カウンタの定義

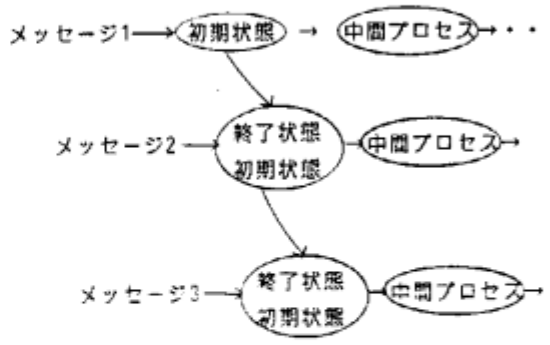


図19　CPによるオブジェクトの実行

sift述語の論理変数ＩにRead Only Annotationが付いている理由は、generate述語で整数の生成が遅れた場合、共有論理変数Ｉがinstantiate していないのにsift述語が先行して実行することを防ぐためである。

8．1．2　CPによるオブジェクト指向プログラミング

まず、オブジェクトを次のように定義する[6]。

オブジェクトー　実際の問題領域に登場するものが持つ機能や知識をモデル化した概念実体。

オブジェクトの特徴の一つは、メッセージを通してのみオブジェクトへの仕事の依頼が可能なことである（図17）。このようなオブジェクトをCPでは次のようにして実現する。

オブジェクト＝プロセス
内部状態　　＝プロセスの引数

CPによるオブジェクトの記述は図18のようになる。これは、オブジェクト"カウンタ"の定義の一部である。"ＵＰ"というメッセージを受信すると、オブジェクトは内部状態に１を加えて、その結果を新しい内部状態とする。これは、カウンタの定義であるので、実際には、次のようにゴールとして呼出して、使用する。

```
:-  instream(I) ,  counter(I?, State).
```

CPで実現されたオブジェクトの実行上の特徴の一つは、メッセージの処理をパイプライン的に並列に実行することができることである。それは、CPが AND並列にプロセスの実行が可能なためである。図19を使って、これを説明する。オブジェクトがメッセージを受けとったならば、メッセージを処理する中間プロセスと再帰的に自分自身を呼び出すプロセスの２つのプロセスを生成する。２つのプロセスは並列に実行できるため、中間プロセスはメッセージの処理を続けながら、色々な中間プロセスを生成する。再帰的に自分自身を呼び出したプロセスは先のメッセージの終了状態であり、また、これは次のメッセージの初期状態でもある。このプロセスは次のメッセージを入力し、そのメッセージの処理を行う。もし、その処理が前のメッセージの処

理結果を必要とするならば、それはRead Only Annotation
を使ってプロセスの同期をとる必要がある。このように、
CPによるオブジェクトは複数のメッセージの処理をパイプ
ライン的に並列に実行する。

8．2　プログラム

```
create_process([Item |Ci],Initdis,Dis] :-
        Initdis_1=[add([[Item:stock,Input_stock),
                (Item:load_manager,Input_load_manager),
                (Item:out_manager,Input_out_manager),
                (Item:stock_lack,Input_stock_lack)]]],
        stock(Item,Input_stock?,0,Dis_2),
        load_manager(Item,Input_load_manager?,[],0,Dis_3),
        stock_lack(Item,Input_stock_lack?,[],Dis_8),
        out_manager(Item,Input_out_manager?,Dis_5),
        merge5(Dis_27,Dis_37,Dis_47,Dis_57,Dis_67,Dis),
        merge(Initdis_17,Initdis_27,Initdis),
        create_process(Ci?,Initdis_2,Dis_6).

create_process([],[],[]).

reception_load_sheet([[sum(Container_number)|Load_sheet]|Ri],Channel) :-
        send(container,container(Container_number,Total),Channel,Channel_1),
        send_load_sheet(Ri,Container_number,Load_sheet,Total,0,Channel_1).

reception_load_sheet([],[]) .

send_load_sheet(Ri,Container_number,[(Item,Stock_count)| C],
                        Total,Container_amount,Channel) :-
        send(Item:stock,stock(Item,Stock_count),Channel,Channel_1),
        send(Item:load_manager,
                stock(Item,Container_number,Stock_count),
                                        Channel_1,Channel_2),
        Container_amount_1 := Container_amount + Stock_count,
        send_load_sheet(Ri,Container_number,C,
                Total,Container_amount_1?,Channel_2).

send_load_sheet(Ri,Container_number,C,
                        Total,Container_amount,Channel) :-
        C = [] | Total = Container_amount ,
        reception_load_sheet(Ri?,Channel).

stock(Item,[request(Item,Count,Result)|S],Stock_amount,Channel) :-
        Stock_amount_1 := Stock_amount - Count,
        Stock_amount_1 >= 0 ,Result = success |
        stock(Item,S?,Stock_amount_1?,Channel).

stock(Item,[request(Item,Count,Result)|S],Stock_amount,Channel) :-
        Stock_amount_1 := Stock_amount - Count,
        Stock_amount_1 < 0 ,Result = fail |
        stock(Item,S?,Stock_amount,Channel).

stock(Item,[stock(Item,Count)|S],
                        Stock_amount,Channel) :-
        Count > 0 |
        Stock_amount_1 := Stock_amount + Count,
        send(Item:stock_lack,enter(Stock_amount_1,Rest],Channel,Channel_1),
        stock(Item,S?,Rest?,Channel_1).

stock(Item,[],Stock_amount,[]) :-
        stock_moniter_3(Item,Stock_amount).

reception_out_request([(Item,Request,Destination)|C],Channel) :-
        send(Item:stock,request(Item,Request,Result),Channel,Channel_1),
        internal_1(Item,Request,Destination,Result?,Channel_1,Channel_2) &
        reception_out_request(C?,Channel_2).

reception_out_request([(Item,[])|C],Channel) :-
        send(Item:stock,[],Channel,Channel_1),
        send(Item:out_manager,[],Channel_1,Channel_2),
        send(Item:stock_lack,[],Channel_2,Channel_3),
        reception_out_request(C?,Channel_3).

reception_out_request([],[]).
```


図２０　機能的方法によるプログラム

```
souko_gakari([[num(Cont_num)|Tumini]|C],St_ch) :-
        container_total(Cont_num,Ch_cont,Total?),
        create_tumini(C,Cont_num,Tumini,St_ch,Ch_cont,0,Total).

souko_gakari([],_).

create_tumini(C,Cont_num,[(Item,Tu_cnt)|Tumini],St_ch,Ch_cont,Cont_amt,T):-
        send_stock(Item,stock(Item,Tu_cnt,Tu_ch),St_ch,St_ch_1),
        Cont_amt_1 := Cont_amt + Tu_cnt,
        merge(Ch_cont_07,Ch_cont_17,Ch_cont),
        tumini(Item,Tu_ch?,Tu_cnt,Cont_num,Ch_cont_0),
        create_tumini(C,Cont_num,Tumini,St_ch_17,Ch_cont_1,Cont_amt_17,T).

create_tumini(C,Cont_num,[],St_ch,[],Total,Total) :-
        souko_gakari(C?,St_ch).

tumini(Item,[req(Item,Count,Ack)|Tu_ch],Tu_cnt,Cont_num,Ch_cont):-
        cp_test_unify(Ack,ack(Cont_num,Rest),Res),
        tumini_result(Res?,Rest,Item,[req(Item,Count,Ack)|Tu_ch],
                      Tu_cnt,Cont_num,Ch_cont).

tumini_result(fail,_,Item,[req(Item,Count,Ack)|Tu_ch],
              Tu_cnt,Cont_num,Ch_cont):-
        tumini(Item,Tu_ch?,Tu_cnt,Cont_num,Ch_cont).

tumini_result(succ,Rest,Item,[req(Item,Count,Ack)|Tu_ch],Tu_cnt,Cont_num,
              [hansyutu(Item,Tu_cnt)|[]]) :-
        Count >= Tu_cnt, Rest := Count - Tu_cnt
        | true.

tumini_result(succ,0,Item,[req(Item,Count,Ack)|Tu_ch],Tu_cnt,Cont_num,
              [hansyutu(Item,Count)|Ch_cont]) :-
        Count < Tu_cnt,Tu_cnt_1 := Tu_cnt - Count
        | tumini(Item,Tu_ch?,Tu_cnt_1,Cont_num,Ch_cont).

container_total(Cont_num,C,Total) :-
        container(Cont_num,C?,Total).

container(Cont_num,[hansyutu(Item,Count)|C],Total) :-
        Total_1 := Total - Count, Total_1 =:= 0 |
        container_monitor(Cont_num).

container(Cont_num,[hansyutu(Item,Count)|C],Total) :-
        Total_1 := Total - Count & Total_1 > 0 |
        container(Cont_num,C?,Total_1).

uketuke_gakari([(Item,Request,Dest)|U],St_ch) :-
        send_stock(Item,req(Item,Request,Dest),St_ch,St_ch_1),
        uketuke_gakari(U?,St_ch_1?).

uketuke_gakari([],_).

stock(Item,[stock(Item,Tu_cnt,Tu_ch)|S1],S2,Tu_ch,
           [entry(Item,Stock_1)|Za_ch],Za_end_ch,Stock) :-
        Stock_1 := Stock + Tu_cnt |
        stock_zaiko(Item,S1,S2,Tu_ch,Za_ch,Za_end_ch?,Stock_1).

stock_zaiko(Item,S1,S2,[req(Item,Used,Ack)|Tu_ch],
            Za_ch,[entry(Item,Rest)|Za_end_ch],Stock_1) :-
        Used := Stock_1 - Rest, Used > 0 |
        ack_tumini(Ack?,Item,Tu_ch,Tu_ch_1) &
        stock(Item,S1?,S2,Tu_ch_1,Za_ch,Za_end_ch,Rest).

stock_zaiko(Item,S1,S2,Tu_ch,Za_ch,[entry(Item,Rest)|Za_end_ch],Stock_1) :-
        Used := Stock_1 - Rest, Used = 0 |
        stock(Item,S1?,S2,Tu_ch,Za_ch,Za_end_ch,Rest).

stock(Item,S1,[req(Item,Request,Dest)|S2],
           [req(Item,Request,Ack)|Tu_ch],Za_ch,Za_end_ch,Stock) :-
        Request =< Stock, Stock_1 := Stock - Request |
        syukko(Item,Request,Dest),
        ack_tumini(Ack?,Item,Tu_ch,Tu_ch_1) &
        stock(Item,S1,S2?,Tu_ch_1,Za_ch,Za_end_ch,Stock_1).

ack_tumini(ack(Cont_num,Rest),Item,Tu_ch,Tu_ch) :-
        Rest =:= 0 | true.

ack_tumini(ack(Cont_num,Rest),Item,[req(Item,Rest,Ack)|Tu_ch],Tu_ch_1) :-
        Rest =\= 0 |
        ack_tumini(Ack?,Item,Tu_ch,Tu_ch_1).

stock(Item,S1,[req(Item,Request,Dest)|S2],Tu_ch,Za_ch,Za_end_ch,Stock) :-
        Request > Stock |
        zaiko_fusoku(Item,Request,Dest,Za_end_ch?,Za_end_ch_1),
        zaiko_fusoku_monitor(Item,Request,Dest),
        stock(Item,S1,S2?,Tu_ch,Za_ch,Za_end_ch_1,Stock).

zaiko_fusoku(Item,Request,Dest,[entry(Item,Stock)|Za_ch],
                          [entry(Item,Stock)|Za_end_ch]) :-
        Request > Stock |
        zaiko_fusoku(Item,Request,Dest,Za_ch?,Za_end_ch).

zaiko_fusoku(Item,Request,Dest,[entry(Item,Stock)|Za_ch],
                          [entry(Item,Rest)|Za_ch]) :-
        Request =< Stock, Rest := Stock - Request |
        syukko(Item,Request,Dest).

syukko(Item,Request,Dest) :-
        syukko_monitor(Item,Request,Dest).
```

(つづく)

```
test :- Load_message=[  [num(1),(wine,10),(beer,20)],
                        [num(2),(whisky,30)],
                        [num(3),(wine,30),(whisky,10)],
                        [num(4),(wine,15),(beer,15),(whisky,10)]],
        Out_message=[(beer,15,junko),(wine,10,taro),(whisky,30,jiro),
                (beer,20,hanako),(whisky,10,nanako),(wine,35,kanae),
                (beer,20,satomi),(whisky,5,nobuko),(wine,35,seiko),
                (whisky,5,kumiko),(whisky,5,takamori)],
        souko_gakari(Load_message?,[S1_wine_1,S1_beer_1,S1_whisky_1]),
        uketuke_gakari(Out_message?,[S2_wine_1,S2_beer_1,S2_whisky_1]),
        stock(wine,S1_wine_1?,S2_wine_1?,_,Za_ch_wine,Za_ch_wine,0),
        stock(beer,S1_beer_1?,S2_beer_1?,_,Za_ch_beer,Za_ch_beer,0),
        stock(whisky,S1_whisky_1?,S2_whisky_1?,_,Za_ch_whisky,Za_ch_whisky,0).
```

図２１　構成的方法によるプログラム

```
| ?- ep test.

------ Zaiko Fusoku List ------        ** stock **
  Item =      beer , Request = 15 , Destination = junko

------ Zaiko Fusoku List ------        ** stock **
  Item =      whisky , Request = 30 , Destination = jiro

------ Shipment List of Goods ------        ** syukko **
  Item =      wine , Request = 10 , Destination = taro

------ Shipment List of Goods ------        ** syukko **
  Item =      beer , Request = 15 , Destination = junko

------ Zaiko Fusoku List ------        ** stock **
  Item =      whisky , Request = 10 , Destination = nanako

------ Shipment List of Goods ------        ** syukko **
  Item =      whisky , Request = 30 , Destination = jiro

------ Zaiko Fusoku List ------        ** stock **
  Item =      beer , Request = 20 , Destination = hanako

------ Zaiko Fusoku List ------        ** stock **
  Item =      beer , Request = 20 , Destination = satomi

------ Shipment List of Container ------        ** container **
  Container number =  2

------ Zaiko Fusoku List ------        ** stock **
  Item =      wine , Request = 35 , Destination = kanae

------ Zaiko Fusoku List ------        ** stock **
  Item =      wine , Request = 35 , Destination = seiko

------ Shipment List of Goods ------        ** syukko **
  Item =      whisky , Request = 10 , Destination = nanako

------ Shipment List of Goods ------        ** syukko **
  Item =      wine , Request = 35 , Destination = kanae

------ Shipment List of Goods ------        ** syukko **
  Item =      beer , Request = 20 , Destination = hanako

------ Shipment List of Container ------        ** container **
  Container number =  1

------ Shipment List of Container ------        ** container **
  Container number =  3

------ Shipment List of Goods ------        ** syukko **
  Item =      whisky , Request = 5 , Destination = nobuko

------ Shipment List of Goods ------        ** syukko **
  Item =      whisky , Request = 5 , Destination = kumiko

------ Zaiko Fusoku List ------        ** stock **
  Item =      whisky , Request = 5 , Destination = takamori
```

図２２　実行例