ソフトウェア工学　39－3
（1984. 11. 28）

データ抽象言語向けの構造化設計法

久野　靖
（東京工業大学理学部）

0．はじめに

最近、プログラム言語にデータ抽象ないしオブジェクト指向の概念が取り入れられるようになった。従来の考え方ではプログラムを手続きとしてとらえ、データ構造はプログラムの外にあるものと見るのに対して、これらはデータとそれに対する操作は一体化した不可分のものと考えるものである。これらの概念を取り入れることにより、プログラムをより局所性が高く、安全で、読みやすく保守しやすいものとすることができる。

既存のプログラム設計技法は、これらの概念を取り入れた言語に対してそのまま適用したのではその利点を十分に生かすことができない。そこで、この種の言語の特徴を生かした設計が行なえるような、新しいプログラム設計技法を提案する。具体的には、構造化設計法 1) を下敷きにしてそれをデータ抽象の考えが生かせるように改訂する。以下では、この新しい設計法を仮に改訂版構造化設計法(Reformed Structured Design, RSD) と呼ぶことにする。

以下、第１節でＲＳＤによる設計の考え方について述べ、第２節で例題を用いて具体的な設計過程について説明する。

1．ＲＳＤによる設計

本節では、まずＲＳＤによる設計の全体的な手順について述べ、続いてそこに表われる概念や図式等について順次説明する。

1.1 ＲＳＤの設計段階

ＲＳＤによる設計は概ね次のような段階を経て進められる。

(1) 対象物の規定
(2) データ流れ図を中心とした機能の規定
(3) 構造図への変換と階層構造の規定
(4) 対象物の実現選択とクラス界面の規定
(5) モジュール間の受け渡しの規定
(6) クラス内部の設計

すなわち、データ流れ図→構造図→受け渡しの規定、という構造化設計法の枠組みはＲＳＤでも同一である。ただし、データ抽象の概念を生かすために、設計の最初の段階として問題の解を得る上でどのような「もの」（対象物）が必要かを考え、規定しておく。これによって段階(2), (3)においてより抽象度の高い設計を行なうことができる。実際に各対象物をどのように実現するかは後の段階(4) まで遅らせることができる。また、この段階で抽象データ型のモジュール（本文ではクラスという言葉を統一的にこの意味で使うことにする）として実現する対象物についても、その外部仕様を規定する。それ以降はクラス内部の設計は独立に進めることができる。

1.2 対象物

ＲＳＤにおける対象物は、与えられた問題の解を得る上で必要となる「もの」であり、主体的に動作を行なう側面と受動的に操作を施される側面とを持つ。多くのシステムは現実世界に存在する物に関する情報を処理するので、ＲＳＤの対象物は現実世界の実体に対応することが多い。従って、例えば問題が自然言語で記述されている場合には、そこに現われる名詞が対象物の候補となる 4)。より形式的な記法が用いられている場合でもこれに相当する抽出が可能なはずである。

ただし、上述の候補のすべてを実際に対象物として扱うわけではなく、問題を解く上で必要がないか、または物理的制約等のため扱うことが困難であると判断したものは除外する。逆に現実には存在しないが仮想的に存在すると考えることでシステムの働きがより自然に記述できる、と判断したものを付け加えることもある。

ＲＳＤでは最終的に対象物を選択したあと、各対象物について、それが何を現わすか、どんな属性を持つか、どのような操作の対象となるかを記述する。

1.3 機能規定

システムが上記の「対象物空間」の上でどのように働くかを明記するのが機能規定の目的であり、その道具として文献 2) の階層化データ流れ図を一部拡張して用いる。その構成要素はプロセス（丸で表わす）、データフロー（矢線で表わす）、バッファ（水平線で表わす）の３つであり、それぞれデータを処理する部分、データの流れ、データのたまっている部分を意味する。

システム全体の規定方法としては、まずレベル０の図としてシステム全体を１つのプロセスとみなし、外界と行き来するデータを列挙したものを作成する。続いてレベル１の図としてシステムの内部をいくつかのプロセス、データフロー、バッファの集まりとして記述したものを作成する。さらにその各プロセスの内部をレベル 1.1, 1.2, ...の図として記述する。このような分割を繰り返して、処理が十分単純でこれ以上細かく分割する必要がない、と判断するところまで来たら、プロセス内部の処理を直接記述する。その記法としては擬似コードをはじめ様々な流儀があってよい。

1.4 構造図

ＲＳＤにおける構造図は文献 1) のものを一部拡張したもので、手続き（四角い箱で表わす）、対象物またはクラス（丸で表わす）、参照関係（矢線で表わす）から成る。ここで、「参照関係がある」というのは「手続きまたはクラス中の操作が他の手続きまたはクラス中の操作を呼び出す」ことを意味する。参照関係の図示は手続き間に限ることにした。

その理由は、クラスはその性質上多くの箇所から参照されるので、それをすべて描くことは図を不必要に複雑にさせてしまうからである。ただし、直接参照関係を図示しなくとも、参照する方を参照される方より上に描く、という原則は守ることにする。

機能規定から構造図への変換は構造化設計法におけると同様にして行なうが、ＲＳＤの場合にはクラスの内部は別に扱うことと、対象物が高レベルであることにより、複雑さはずっと小さくて済む。

1.5 階層構造

システムの構造に階層性を持たせることは、モジュール間の不要な絡みをなくし構造を分かりやすくする上で有効である。特にデータ抽象向き言語では階層構造がプログラムの上に反映されるので、階層設計のよしあしがプログラムの品質に影響しやすいと言える。

そこで、ＲＳＤでは構造図を作成した時点で手続きと対象物の間の階層分けを行なう。具体的には、構造図の上に階層の境界線を記入し、各階層に名前をつければよい。その際には、類似した機能や複雑さを持つものは同じ階層に入れること、および参照関係で上位にあるものは階層としても上にあることを原則とする。

階層分けを行なおうとしてうまく行かない場合には、それ以前の設計段階に誤りがある可能性が大きい。そのような場合にはその段階までさかのぼってやり直す必要がある。

1.6 受け渡し表

受け渡し表は手続きまたはクラス中の操作の呼び出しにおいて渡される情報を厳密に記述したもので、書き方としては 1）にあるものを手直ししたものである。すなわち、呼び出し元の手続きや操作ごとに別の表を用意し、呼び出される手続きや操作ごとに入力情報（呼び出し元から呼び出される側に渡す）、出力情報（その逆）を区別して記入する。

情報の各項目についてはその意味、型（データタイプ）、制御情報か否か、入力情報の場合に、その内容を呼び出された側で更新するか否かを明示するものとする。

1.7 対象物の実現

設計の最初の段階で選ばれた対象物が、すべて抽象データ型を定義するモジュール（クラス）によって実現されるとは限らない。具体的には次のような可能性がある。

(1) 基本型による実現。例えば数量、個数といった対象物は標準の整数型等によって実現できる。
(2) 複合型による実現。例えば～の並び、～の対、といった対象物は配列、レコード等の定義により実現できる。
(3) クラスによる実現。対象物を一般的な抽象データ型として定義する。
(4) 仮想的実現。概念的には対象物を考えるが、プログラム上には対応する実体がない場合がある。例えば「ページ番号を付けて打つ」という場合、「ページ」という概念だけあって、具体的な「ページ」という実体はない、ということもあり得る。

そこで、構造図が完成した段階で、対象物ごとに適切と思われる実現方法を選ぶ。また、クラスを定義する場合にはここでクラスの界面（操作名、各操作の働きと受け渡される情報）を規定する。受け渡される情報の記述には上記の受け渡し表を用いる。

1.8 クラス内部の設計

以上ではシステム全体に関する記述について述べて来たが、ＲＳＤではクラスも１つの閉じた世界であり、その内部についても同様に一連の図等を用いて記述できる。その過程は外部の設計とは独立に進められるが、外部の他のクラス等を参照することはあり得る。逆に、外部のクラスや手続きがクラスの内部事情によって影響されるようなことはあるべきではない。

1.9 実現段階以降

以上で(1)作成するモジュールのリスト、(2)各モジュールの外部仕様、および(3)各モジュールの機能記述が得られ、実現に取り掛かることができる。また、これらの設計文書は抽象度の高いものから具体的なものまで構造化されており、参照や変更の際の差し替えが行ないやすい。

作成されるプログラムについても、ＲＳＤを使用すればデータ抽象の考え方を生かしたものができやすく、したがって修正や拡張が行ないやすいと考えられる。

２．共通例題による設計例

８４年４月の設計技法シンポジウムで採用された例題 3）を使ってＲＳＤの実施例を示す。

2.1 問題に関する補足

問題文には明記されていない点が種々あるが、Unixオペレーティングシステム上で動かしてみることを想定して、ここでは次のような仮定を置く。

● システムの構成。受付係の手もとに端末があり、受付係は積荷票、出庫依頼の情報をここから対話的にキーインする。電話による注文は受付係が受けて端末に打ち込む。出庫指示書と電話による在庫なし連絡の指示は端末の横のプリンタに打ち出されるものとする。在庫不足リストについては、計算機内のファイルに累積されるものとする。
● システムの運用。朝、システムが起動される時に、在庫の状況をファイルから読み出す。夕方、終了時に在庫の状況をファイルに格納する。在庫不足リストを累積するファイルについては、次々に内容が追加される。これらのファイルの管理等はオペレータが適宜おこない、システムの範囲外とする。
● ファイルの割り当て。プログラムから見て、端末のキーボードは標準入力ファイルに、画面は標準エラーメッセージファイルに、プリンタは標準出力ファイルに、それぞれつながっているものとする。在庫を読み出すファイル、格納するファイル、在庫不足リストを累積するファイルの名前はそれぞれ 'stokc.in','stock.out','nostock.list'とする。

● 入出力の設計。問題文には入出力の書式、対話的操作の手順等が記されていない。これらの点について、本来はプロンプト、エラーメッセージ、書式設計等の詳細な仕様が必要であるが、問題を解いてみせるうえでさほど本質的ではないので、ここでは大まかな記述のみを示すことにする。

入力
(1) 受付係はトップレベルでＩ，Ｓ，Ｅのいずれかを打ち込むことで入庫処理、出庫処理、システム終了処理のいずれかを選択する。
(2) 入庫処理を選択した場合には、まずコンテナ番号、搬入日時を入力し、続いて品名、数量の組を繰り返し入力する［積荷票］。最後に終わりの印を打ち込むとそれまでに入力したデータについて処理を行ない、トップレベルの入力へ戻る。
(3) 出庫処理の場合には品名、数量、顧客名を入力し［出庫依頼］、処理後トップレベルへ戻る。
(4) 終了処理の場合には必要な後始末を行なってからシステムを終了する。

出力
(1) ［出庫指示］
　　注文番号、顧客名　（１行）
　　コンテナ番号、品名、数量、「空」(n行)
　（「空」は［空コンテナ搬出マーク］であり、そのコンテナが空である場合にだけ出力される。）
(2) ［在庫なし連絡］
　　顧客名、品名、数量
(3) ［在庫不足リスト］
　　顧客名、品名、数量

● 倉庫の検索アルゴリズム。簡単のため、「とりあえず見つかったものから出荷する」という、いちばん単純なやり方を採るが、ただしアルゴリズムの変更は倉庫モジュールを取り替えるだけでできるように設計する。

2.2 対象物の記述
　まず、問題文から対象物を記述することば（名詞）を抜き出したリストを作成する。

酒類販売会社、倉庫、コンテナ、酒、銘柄、倉庫係、積荷票、受付係、出庫指示、内蔵品、場所、コンテナ番号、搬入年月日時、内蔵品名、数量、出庫依頼、出庫依頼票、電話、在庫、依頼者、コンテナ数、在庫なし連絡、在庫不足リスト、送り先名、計算機プログラム、注文番号、空コンテナ搬出マーク

　これらについて、システムの扱う対象物として適当かどうかを検討する。

● 会社、倉庫係、受付係、計算機プログラムは外界との関係を記述するために出て来た言葉で、システムが自ら扱うわけではないので除外する。
● 場所も「・・・に移すことはない」ので、除外する。
● 電話はシステムが直接電話を掛けることはないので、除外する。
● 酒については、システムが個々の酒びんのレベルまで細かく扱うことはない、と判断されるので、除外する。
● 内蔵品、在庫はともに「現在注目している銘柄の酒の集合」の意味なので、独立した対象物とは考えない。
● コンテナ数は、「現在倉庫の中に入っているコンテナの個数」のことなので、独立した対象物とは考えない。

　残ったものについて、同じ対象物をいく通りかの名前で呼んでいるものを整理し、名前を適宜付け替えて次のリストを得る。

倉庫、コンテナ、銘柄（＝内蔵品名）、積荷票、出庫指示、コンテナ番号、日時（＝搬入年月日時）、数量、注文、出庫依頼（＝出庫依頼票）、顧客（＝依頼者、送り先名）、注文番号、在庫なし連絡、在庫不足リスト、空コンテナ搬出マーク

　これらのうちで、ファイル上のイメージに相当するものは次の通り。

　入力：積荷票、出庫依頼
　出力：出庫指示、在庫なし連絡、在庫不足リスト、
　　空コンテナ搬出マーク

以上の対象物については、既に仕様中に記述があるので、特に説明を必要としない。それ以外の対象物はシステムの中で自立した型の実体として存在するはずの物である。これらについて分かっている情報（何を表わすか、どんな属性・性質をもつか、どのような操作の対象になるか）を記述する。

● 倉庫 → コンテナのたまっている場所。コンテナを搬入すること、銘柄・数量を指定してコンテナの集まりを取り出すこと、倉庫の中味をファイルにセーブすること、中味をファイルからロードすることができる。コンテナの搬入は数個／日、取り出しの回数は数十件／日程度である。
● コンテナ → 酒を格納する入れ物。搬入日時、コンテナ番号を属性として持つ。銘柄を指定してその収蔵量をセットしたり読み出したりできる。また、空かどうかを調べることもできる。一つのコンテナには最大１０銘柄入ればよい。
● 顧客 → 取引先の名前。文字列で表わせる。
● 銘柄 → 酒の銘柄。文字列で表わせる。
● コンテナ番号 → コンテナの識別番号。文字列で表わせる。
● 日時 → 日付と時間。文字列で表わせる。
● 注文 → ＜注文番号、銘柄、数量＞の組。
● 注文番号 → 注文の一連番号。整数で表わせる。
● 数量 → 酒のビンの本数。整数で表わせる。
● コンテナの集まり → コンテナがｎ個集まったもの。コンテナを１個ずつ取り出したり追加できる。集まりが空集合かどうかを調べることができる。

　以上でもとの問題文の記述から取り出した対象物は列挙されたが、はじめに記した問題の扱いに関す

る仮定の記述についても同様の作業が必要である。ここではスペースの都合上、詳細については省略し、結果だけを示す。

● 入力ファイル → 端末から文字列を入力する。
● 出力ファイル → プリンタに文字列を出力する。
● メッセージファイル → 端末画面に文字列を出力する。
● 在庫入力ファイル → 在庫状況記録ファイルから文字列を読み出す。
● 在庫出力ファイル → 在庫状況記録ファイルに文字列を書き出す。
● 在庫不足ファイル → 在庫不足記録ファイルに文字列を書き出す。

これらがすべてファイル関係のものであることは、さきの仮定が問題文にシステムのファイルまわりの運用について記されていない点を補ったものであることを考えればうなずける。

以上に加えて、整数、文字列などを扱うことが当然必要であるが、これらを逐一列挙することは非現実的であるので、ＲＳＤでは基本的データ型については対象物として記述しないという方針を採る。

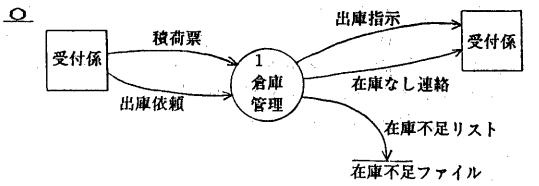
2.3 機能記述の作成

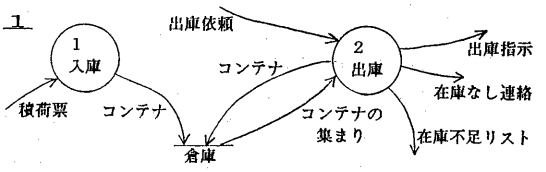
先の対象物記述をもとに作成した機能記述を図１に示す。まず、レベル０の図で、システムと外界とのやりとりを図示する（四角い箱はシステムの外にあるものを示す）。ここではシステムに「倉庫管理」という名前をつけてある。最初の仮定から、システムは受付係から積荷票と出庫依頼を受け取り、受付係に出庫指示と在庫なし連絡を渡す。また、在庫不足ファイルに在庫不足リストを出力する。

次に、レベル１の図で、システムの大域的な分割を示す。システムはコンテナが倉庫に入って来る際の「入庫」処理と注文に応じて出荷する「出庫」処理に分けられる。システムの外部とのやりとりのうち、「入庫」に関係するのは積荷票だけで、あとはすべて「出庫」が扱う。倉庫とのやりとりは、「入庫」はコンテナを格納する一方であるが、「出庫」は関係するコンテナをまとめて一旦取り出してから処理を行ない、空でないコンテナは再び倉庫に戻す、という方針にしたので、両方向の流れがある。

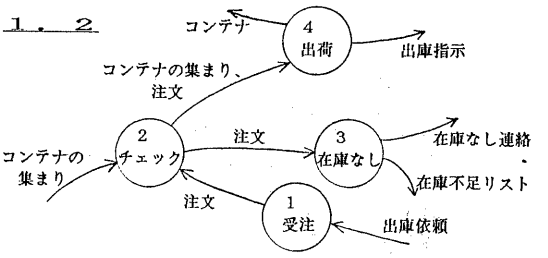
レベル１の記述に現われる処理のうち、「入庫」はすでにそれ自体十分単純であるので１．１項に擬似コードにより処理の内容を示してある。

「出庫」については、まだ十分単純とは言えないので、１．２の記述にあるようにさらにデータ流れ図による分解を行なう。すなわち、その中味はさらに出庫依頼を読み込んでシステム内部で扱う注文に変換する「受注」、注文に応じられる在庫があるかどうか調べる「チェック」、在庫が不足するばあいに在庫なし連絡と在庫不足リストを生成する「在庫なし」、在庫があった場合に出荷を行なう「出荷」の４つに分割する。これらの各処理は十分単純と考えられるので、１．２．１～１．２．４でこれらの内容を擬似コードで記述することで、全体の機能記述が完成したことになる。





１．１
コンテナ番号、日付を入力し、空コンテナを作る。
繰り返し積荷票の終わりまで
　銘柄、数量を入力し、コンテナにセットする。
コンテナを送り出す。



１．２．１
銘柄、数量を入力し、注文を生成する。

１．２．２
もし注文に応じられる在庫があるなら
　コンテナの集まりと注文を出荷処理に渡す。
そうでなければ
　注文を在庫なし処理に渡す。

１．２．３
在庫なし連絡として、顧客名、品名、数量を出力する。
在庫不足リストとして、顧客名、品名、数量を出力する。

１．２．４
出庫指示として、
　注文番号、顧客名を出力する。
　繰り返し注文数量に達しない間、
　　コンテナの集まりからコンテナを一つ取り出す。
　　コンテナ番号、品名を出力する。
　　もしコンテナ積載数量＜残り数量なら
　　　コンテナ積載数量を出力、積載数量を０にセットする。
　　そうでなければ
　　　残り数量を出力し、積載数量を残り数量だけへらす。
　　もしコンテナが空なら
　　　空コンテナ搬出マークを出力する。
　　そうでなければ
　　　コンテナを倉庫に戻す。
残ったコンテナを倉庫に戻す。

図１　階層化データ流れ図による機能規定

2.4 モジュール構造の規定

次の段階として、先に作成したデータ流れ図をもとに、作成する手続きとそれらの間の呼び出し関係を決定し、また対象物を含めたシステムの階層構造

を規定する。
　データ抽象を用いることで、データ流れ図の規模は比較的小さくて済み、従って手続きへの変換も簡単になることが多い。ここでもその例にもれず、データ流れ図の各基本処理に対応して１つずつ手続きを用意すれば済む。すなわち、

● 倉庫管理 → 入庫、出庫
● 出庫 → 受注、チェック、在庫なし、出荷

これらの他に、手続きおよびクラスからクラスへの参照も列挙する。

● 倉庫管理 → 入力・出力・メッセージ・在庫不足ファイル、倉庫、注文番号
● 入庫 → 入力・メッセージファイル、倉庫、コンテナ、銘柄、数量、コンテナ番号、日時
● 出庫 → なし
● 受注 → 入力・メッセージファイル、注文、銘柄、顧客、数量
● チェック → 倉庫、コンテナ、並び、注文、銘柄、数量
● 在庫なし → 出力・メッセージ・在庫不足ファイル、銘柄、顧客、数量
● 出荷 → 出力ファイル、倉庫、コンテナ、並び、銘柄、顧客、数量、コンテナ番号
● 倉庫 → コンテナ、並び、銘柄、数量、在庫入力ファイル
● コンテナ → 銘柄、数量

　これらすべてを図示することは図が込み入りすぎてかえって理解しにくくなるため、手続き間の参照のみを記入するが、関係は図示しなくても参照するものは参照されるものより必ず上に置く。また、階層構造は構造図の上に階層の境界と階層名を記入することで示す。図２に階層を記入した構造図を示す。
　ここではシステムの階層構造は次のようにして決定した。まず、システムの一番中核をなす、主として倉庫、コンテナを扱うモジュールをコンテナレベルの階層としてまとめた。この中にはクラス倉庫、コンテナ、コンテナの集まり、および手続き入庫、チェック、出庫が含まれる。
　次に、注文の処理を行なうためのクラス注文、手続き受注、在庫なしを注文レベルの階層としてまとめる。コンテナレベルの手続きチェック、出荷は注文も扱うので、注文レベルはコンテナレベルの下に位置することになる。
　手続き倉庫管理、出庫はこれまでに出て来た手続きの呼び出しを制御するのが主な役割なので、最上位のシステムレベルの階層としてまとめる。
　残ったモジュールはクラスばかりであり、これらは大きくファイル関係とそれ以外のものに分類できる。そこで前者は入出力レベル、後者は問題記述に現われる基本的な実体に対応するので問題むけ実体レベルという名前をつけた。これらの間には互いに参照関係がないのでどちらを上に描いてもよいが、ここでは入出力レベルを上に描いてある。

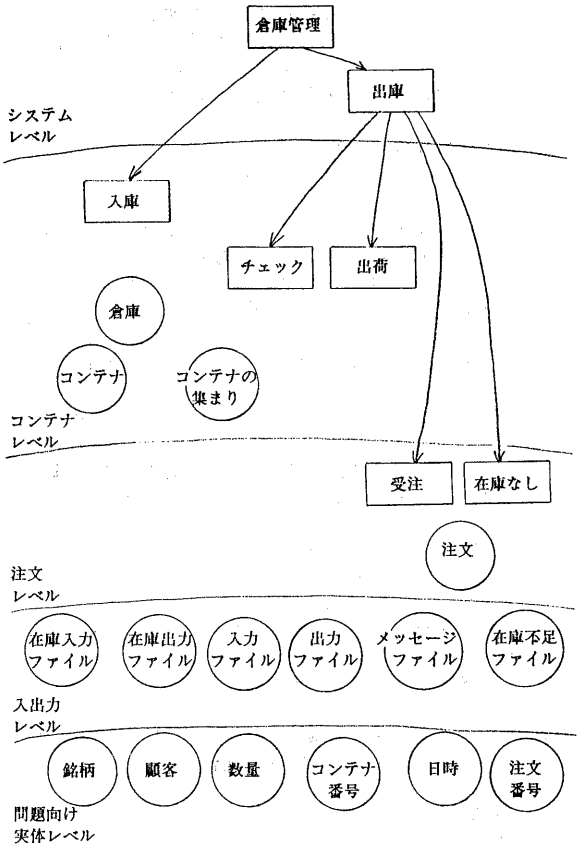


図２　階層構造を記入した構造図

2.5 実現の規定

　ここまでの段階では実現言語を特に意識せずに進めて来たが、ここではじめて具体的な言語を念頭に各モジュールの実現を規定する。ここでは実現言語としてＣＬＵ 7）を使用するが、Ada,Euclid等の抽象データ型をサポートする言語であれば結果にはどれでもあまり差はない。また、Fortran,Cobol 等の言語でも抽象データ機能を手で「シミュレート」することにより使用可能である。
　手続きモジュールについては、そのままＣＬＵの手続きとして実現するので、ＣＬＵの規則に合った名前をつけさえすれば済む。

```
倉庫管理→manage_stock
入庫→arrival
出庫→shipment
チェック→check_stock
出荷→do_ship
受注→read_order
在庫なし→no_stock
```

　一方、クラスとして扱われているものについては、抽象データ型としての実現、複合型（レコード、配列）による実現、言語の標準の基本型による実現な

どの中から適切なものを選択する必要がある。
まず、倉庫とコンテナについてはその内部構造が外から見えないように抽象データ型のモジュール（ＣＬＵではクラスタと呼ぶ）を作成する。

倉庫→クラスタ stock
コンテナ→クラスタ container

コンテナの集まりの実現には、ＣＬＵでは動的な配列が使用可能なので、これを直接使用する。注文は、成分として注文番号、銘柄、数量を持つレコード型を用いる。

コンテナの集まり→set = array[container]
注文→order = record[no:注文番号,
brand:銘柄, quant:数量]

入出力レベルのクラスはすべてＣＬＵの標準の流れ入出力ファイル型であるstreamを利用する。

在庫入力ファイル→stream
在庫出力ファイル→stream
入力ファイル→stream
出力ファイル→stream
メッセージファイル→stream
在庫不足ファイル→stream

問題向け実体はその性質に応じてＣＬＵの標準の文字列型(string)または整数型(int) を用いる。

銘柄→string
顧客→string
数量→int
コンテナ番号→string
日時→string
注文番号→int

次に、クラスタを作成するものについて、図３にその操作と呼び出し界面を規定する。表の部分は、左から操作名、入力情報、出力情報が記してある。
ＣＬＵには例外処理機構が備わっているので、倉庫から取り出そうとして在庫がなかったり、何らかの原因でファイルの読み書きが失敗した場合にはシグナルによって呼び出し側に通知する。これは例外処理機構を持たない言語の場合に論理値によって成功か失敗かを通知することに相当する。したがってここで渡される情報は制御情報であり、●マークがつけてある。
他に＊マークのついた項目があるが、これは入力として渡されたものに呼び出された側で変更を施すことを示している。
実は問題の指定の中にはコンテナの日付を利用する所はないので、操作get_dateは不要である。しかし、日付という情報を持っていながらそれを参照できないのでは抽象データ型として不合理なので、get_dateを入れてある。

cluster stock
load→ファイルから倉庫の内容を回復する
save→ファイルに倉庫の状態を退避する
fetch →倉庫から指定銘柄積載のコンテナを必要なだけ搬出する
store →倉庫にコンテナを搬入する

load	倉庫ファイル名：string	倉庫：stock ●not_possible：signal
save	倉庫ファイル名：string 倉庫：倉庫	●not_possible：signal
fetch	倉庫＊：stock 銘柄：string 数量：int	並び：set ●no_stock：signal
store	倉庫＊：stock コンテナ：container	

cluster container
create→空のコンテナを生成する
set_quant →指定銘柄の格納数量を指定数量にセットする
get_quant →指定銘柄の格納数量を読み出す
is_empty→コンテナが空かどうかを調べる
load→ファイルに格納してあった情報からコンテナを復元する
save→コンテナの情報をファイルに格納する
get_no→コンテナ番号を読み出す
get_date→搬入日時を読み出す

create	コンテナ番号：string 日時：string	コンテナ：container
set_quant	コンテナ＊：container 銘柄：銘柄 数量：整数	
get_quant	コンテナ：container 銘柄：string	数量：int
load	ファイル＊：stream	コンテナ：container ●not_possible：signal
save	ファイル＊：stream コンテナ：container	●not_possible：signal
get_no	コンテナ：container	コンテナ番号：string
get_date	コンテナ：container	日付：string

図３　クラス界面の規定

2.6 受け渡しの規定

ここまでくれば、手続き群に関する受け渡し表を作成して広域的なモジュール構造の設計を完成させることができる（図４）。
まず、手続きmanage_stockについては、streamとstock に対する前始末と後始末を行なうことと、指令に応じてarrival とshipmentを呼び出すことが仕事のすべてである。arrival ではstreamからデータを呼んでcontainer を作り、それをstock に格納する。shipmentはread_orderにより注文を読み、在庫の有無をcheck_stock により調べ、結果に応じてdo_ship かno_stockを呼ぶ。
read_orderはarrival と同様、streamからデータを読むが、生成される注文はレコード型として規定したので、それに対する操作は表には表われない。check_stock は倉庫から出荷を試みる。no_stockは

必要な記録をstreamに書く。do_ship は渡された集まりから１つずつcontainer を取り出しながら出庫指示を書き、空にならなかったコンテナは倉庫に返す。（remは配列から最下端の要素を取り去り、それを値として返す操作であり、配列が空の時はシグナルboundsを返す。）これらも注文データを扱うが、それらは表には記載されない。

このように、設計が正しく進められれば、機能記述とクラス界面を含めた対象物実現規定からほぼ機械的に受け渡しの規定を決めることができる。

2.7 クラス内部の設計

最後に、クラスタcontainer，stockの内部の設計が残っている。これらについては紙面の都合上、省略するが、基本的にはクラスを１つの世界としてこれまでに示したと同様のやり方で設計できる。少し異なるのは、クラスは外部から呼び出される操作を複数持つので、主プログラムが複数個ある形をとる点である。

ただし、元の問題の複雑さにもよるが、ほとんどの場合、クラスの内部はシステム全体に比べればずっと単純であり、図式等も簡単なもので済むことが多い。

また、前の段階で規定したクラス界面に一致していれば、クラスの内部実現はこの段階で自由に決めることができる。例えば、container の実現としてはレコードの配列を用いて銘柄ごとの数量を単純に表として保持し、stock の実現としては既に出て来た「コンテナの集まり」をそのまま使う、というのが最も単純な実現として考えられる。

2.9 プログラム作成

ここに述べた設計については、実際にＣＬＵ言語でコーディングし、Ｕｎｉｘ（バークレー版）上のＣＬＵ処理系により実行させてみてある。クラスタの内部設計は上述の「最も単純な」方法によった。プログラムのソース行数は約３００行、完成した設計をもとに直接エディタでコーディングするのに約４時間を要した。虫はすべて打ち間違い等に起因する単純なもので、修正に要した時間は３０分未満であった。

３．討論

最初に記したように、ＲＳＤは構造化設計法に抽象データあるいはオブジェクト指向の考え方を取り入れることにより、新しい言語にも適した設計技法を目指したものである。これをもととなった構造化設計法と比較した場合、新しい言語に適応したこと以外にも、次のような特徴がある。

(1) 設計手順の明確化。ＲＳＤではもとの構造化設計法よりずっと細かく設計段階が分かれているが、これはデータ抽象言語で書かれたプログラムが従来の言語よりも明確な枠組みを持つことに対応している。これにより、設計者は次に何を決めるべきかについて迷わずに、段階を追って設計を進めることができる。

manage_stock

→stream		
primary_input	—	入力ファイル：stream
primary_output	—	出力ファイル：stream
open	在庫不足ファイル名:string	在庫不足ファイル:stream ●not_possible：signal
getl	入力ファイル＊：stream	入力行：string ●end_of_file ：signal ●not_possible：signal
putl	出力ファイル＊：stream メッセージ：string	●not_possible：signal
→stock		
load	旧倉庫ファイル名：string	倉庫：stock ●not_possible：string
save	倉庫：stock 新倉庫ファイル名：string	●not_possible：string
→arrival	入力ファイル＊：stream メッセージファイル*:stream 倉庫＊：stock	●not_possible：signal
→shipment	入力ファイル＊：stream 出力ファイル＊：stream メッセージファイル*:stream 在庫不足ファイル＊：stream 倉庫＊：stock 注文番号：int	●not_possible：signal

arrival

→stock		
store	倉庫＊：stock コンテナ：container	
→stream		
getl	入力ファイル＊：stream	入力行：string ●end_of_file ：signal ●not_possible：signal
putl	出力ファイル＊：stream メッセージ：string	●end_of_file ：signal ●not_possible：signal
→container		
create	コンテナ番号：string 日時：string	コンテナ：container
set_quant	コンテナ＊：container 銘柄：string 数量：int	

shipment

→read_order	入力ファイル＊：stream 出力ファイル＊：stream 注文番号：int	注文：order ●not_possible：signal
→check_stock	倉庫：stock 注文：order	コンテナの集まり：set ●no_stock：signal
→no_stock	出力ファイル＊：stream 在庫不足ファイル＊：stream 注文：order	●not_possible：signal
→do_ship	出力ファイル＊：stream コンテナの集まり：set 倉庫＊：stock 注文：order	●not_possible：signal

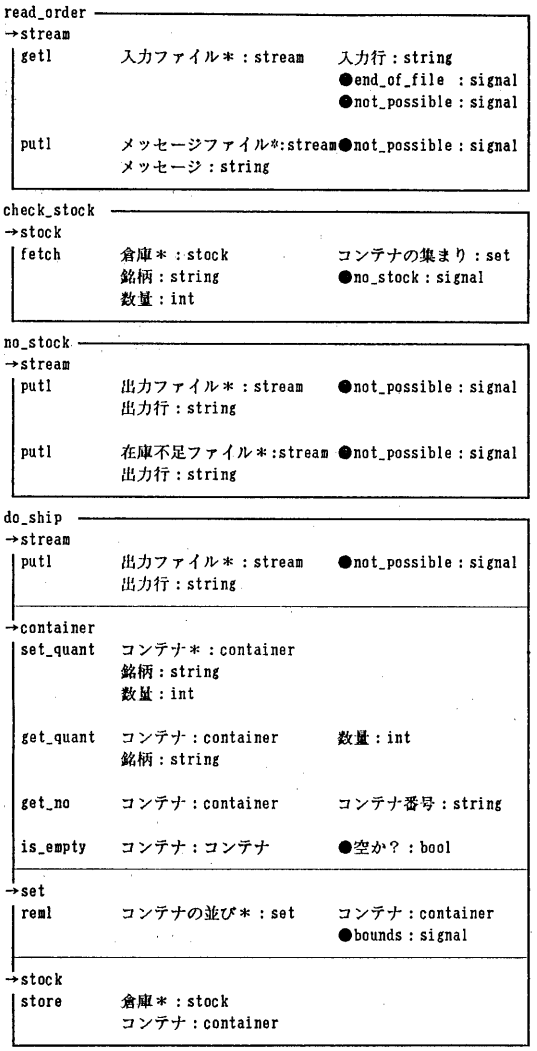


図４　受け渡し表

(2) データ主導の設計。ＲＳＤの、対象物を中心として設計を進める、という方針もデータ抽象あるいはオブジェクト指向の考え方から採られたものであるが、設計の各段階において設計者に先へ進むための手掛かりを与える上で有効であると思われる。

(3) 設計の分割。構造化設計法では設計は論理的には１つの大きな構造図として表わされていたが、ＲＳＤではクラスの内部を別に扱うことにより設計を分割し、一時に扱う問題の範囲を小さくとどめ、文書等の管理も行ないやすくできる。

ＲＳＤをワーニェ法 8)、ジャクソン法 9) などの従来のデータ主導設計法と比較してみることも興味深い。これらのやり方では、最初の段階でシステムの扱うすべてのデータの構造を細部まで規定するようになっている。そのようにした場合、後から下位のデータ構造を変更することは通常困難である。一方ＲＳＤでは対象物の内部構造はできるだけ後で決める方針を採っており、特にクラスの内部構造は後で取り替えても他の部分を変更する必要がない。例えば今回の例題で倉庫の内部で銘柄別のインデックスを別に用意して検索の効率を高めたとしても、他のモジュールはそのままでよい。

また、より新しいジャクソン開発法 10)では現実世界の実体に対応したエンティティに注目して設計を進める。ただし、そこでいうエンティティは後で順次実行プロセスに対応させられる手続き的なものであり、動的に生成されたり受け渡されたりするものではないので、そのデータ抽象機構としての働きは限られたものである。

ＲＳＤの実用性を測る試みとして学生（学部３年生）を対象に、ある程度の大きさのシステム（Unixのed相当のエディタ）を設計させる課題を出し、同時にＲＳＤについて100 分程度の説明を行なったところ、半数以上の学生がＲＳＤを利用して設計を行なった。ＲＳＤを使用した学生の主な意見は、手順や記述形式が明確なので設計を進める際に記法等で迷わなくて済む、他人が見ても分かりやすい設計ができる、ただし階層構造は何のために決めるのか分かりにくい、等であり全体的には好評であった。

今後の課題としては、図式等をより見やすいものとすること、設計手順をさらに明確にすること等のほか、マニュアルを整備し、より大規模な問題に適用することで実用性に関する裏付けを行なうことが考えられる。

参考文献

1) Stevense, W.P., Meyers, G.J., Constantine, L.L., Structured Design, IBM Syst. J., Vol. 13, No.2, pp.115-139, 1974.
2) DeMarco,T., Structured Analysis and System Specification, Prentice-Hall, 1978.
3) 山崎、共通問題によるプログラム設計技法解説、情報処理、Vol.25, No.9, p.934, 1984.
4) Booch, G., Software Engineering with Ada, Benjamin Gummings, 1983.
5) Meyers, G.J., Reliable Software through Composite Design, MASON/CHARTER, 1975.
高信頼性ソフトウェア－複合設計、久保・国友訳、近代科学社、1976.
6) Yourdon, E., Constantine, L.L., Structured Design, Prentice-Hall, 1979.
7) Liskov, B. et al., CLU Reference Manual, Lecture Notes in Computer Science 144, Springer-Verlag, 1981.
8) 鈴木、構造化プログラミングワーニェ・メソド、情報処理、Vol.25, No.9, pp.946-954.
9) Jackson, M.A.,Principles of Program Design, Academic Press, London, 1975.
10) Jackson,M.A.,System Development, Prentice-Hall, 1983.