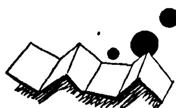
解　説



モジュラー・プログラミングのための支援環境†

湯浅太一†† 小島啓二††† 中島玲二††

1. はじめに

ソフトウェアの巨大化に伴い，信頼性の高いソフトウェアをいかに効率良く作成・維持するかという問題が深刻化している．この問題に対処するため，70年代のはじめに構造化プログラミング[2] (structured programming)や段階的詳細化法[15](stepwise refinement)などのプログラミング方法論が提唱された．これらをふまえて，抽象概念 (abstraction) の使用にもとづく問題分割[8] によってプログラム開発を進めるのがモジュラー・プログラミングである．モジュラー・プログラミングの方法は，プログラム開発における一般的指針として非常に有意義なものであるが，専用の言語・処理系および支援系など，その環境を整備することによってさらにその真価を発揮する．本論文では，モジュラー・プログラミングの効果的な支援についてさまざまな角度から論じる．

2. モジュラー・プログラミング

まず，モジュラー・プログラミングにおける諸概念を整理する．

プログラムの作成とは，仕様 (specification) と呼ばれるそのプログラムが果すべき機能の抽象的記述をいかにして実現 (implement) するかという問題を解決することであるとここでは定式化する．この問題が十分簡単なものであれば，これは直ちに解決される．しかし複雑な問題の場合には，いくつかの抽象概念 (抽象データ型や抽象関数など) を用いて解決をはかる．当然これらの抽象概念をいかに実現するかという問題が新たに生ずるが，これらの問題は必要に応じてさらにさまざまな抽象概念を用いながら解決される．抽象概念の使用によって問題は自然に分割され，問題全体は

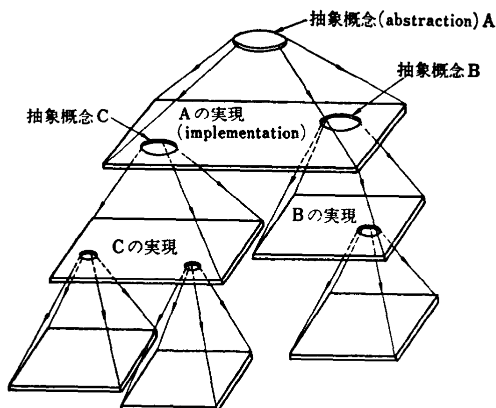


図-1 抽象概念の使用による問題分割

1つの階層構造を成す (図-1)．モジュラー・プログラミングによるプログラムは，この階層内における機能的にまとまった対象を記述するモジュールと呼ばれる単位から構成される．

高品質ソフトウェアをモジュラー・プログラミングによって効率良く作成・維持するためには各モジュールが以下の諸条件を満足することが必要である．

(i) 個々のモジュールの大きさがプログラマにとって扱いやすいものである．すなわち作成したり，読んで内容を理解したりするのに適当なサイズである．

(ii) モジュールが仕様 (モジュールの果すべき抽象的機能) 通り機能するか否かは，そのモジュールが使用される文脈には依存しない．

(iii) モジュールに対し，その仕様を満たすどのような具体的実現を与えても，そのモジュールを使用する他のモジュールの機能には影響しない．

モジュールが条件(ii)および(iii)を満たすならば，その信頼性は他のモジュールと独立に高めることが可能となり，条件(i)と合わせ，個々のモジュールの信頼性を向上させるのは比較的容易である．逆にこれらの条件を満たさないモジュールがプログラム中に存在すれば，そのプログラムの透明度は大きく損なわれ，

† Modular programming systems by Taiichi YUASA, (Research Institute for Mathematical Sciences, Kyoto University) Keiji KOJIMA (Central Research Laboratory, Hitachi Ltd.) and Reiji NAKAJIMA (Research Institute for Mathematical Sciences, Kyoto University).
†† 京都大学数理解析研究所
††† (株)日立製作所中央研究所

プログラム全体の信頼性を向上させるのは困難となる．したがって，条件(i)～(iii)が満たされてはじめて，高信頼ソフトウェアの作成・維持という目的に対してモジュラー・プログラミングがその効果を発揮するということができる．

通常のプログラミング言語(Fortran, Pascal など)を用いても，使用する言語機能などに制限を加えることによって，条件(i)～(iii)を満たすモジュールを構成するのは不可能ではない(種々の言語でモジュラー・プログラミングを行う際の議論が文献[1]にみられる)．また，通常の言語で記述されたプログラム単位間を結合することによって，モジュール構造を導入するための言語も提案されている（たとえば Intercol[14]）．

しかし，データの抽象化など，種々の抽象化を活用しつつモジュラー・プログラミングを行うことを想定すれば，それらの抽象化をサポートしている言語およびその処理系・支援系の上でプログラム開発を進める方がはるかに効率的で信頼度も高い．この観点からいくつかのモジュラー・プログラミング用言語が設計されている(CLU[7]，Mesa[9]，イオタ[11),17]，Ada[10),18]などがその例である)．以下では，モジュラー・プログラミングはこれらの専用言語を用いて行われると仮定する．

これまでモジュールの概念を，それに対する制約条件を与えることによって間接的に規定してきた．実際，前にあげた諸モジュラー・プログラミング用言語の間においても，モジュールの具体的な定義には大きな差異がみられる．そこでまずモジュールに対して，十分に一般性のある具体的なイメージを与えよう．すなわち，ここでいうモジュールとは，新たな抽象概念（抽象データ型，抽象関数など）の導入を行う部分と，それらの実現を与える部分とから成るものとする．抽象概念の導入部はインタフェースと呼ばれ，導入される抽象概念の名前とその基本的属性（抽象データ型であればその基本関数，抽象関数であればその定義域・値域など）がここで宣言される．一方実現部には，インタフェースで導入された抽象概念を実現するための手続き的な (procedural) プログラムが記述されている．図-2は仮想的なモジュラー・プログラミング言語によって記述されたモジュールの例である．あるモジュールAが別のモジュールBを参照しているとは，Aの中でBで導入した抽象概念を用いている場合をいう．モジュール全体は参照関係によって階層構造をつくる．

このようなモジュールの枠組は，前述のモジュラー・プログラミングの方法論を的確に反映しており，今後の議論の一般性を保証するものである．

次章では，モジュラー・プログラミング用言語の処理およびそれに必要な情報の管理について論じる．

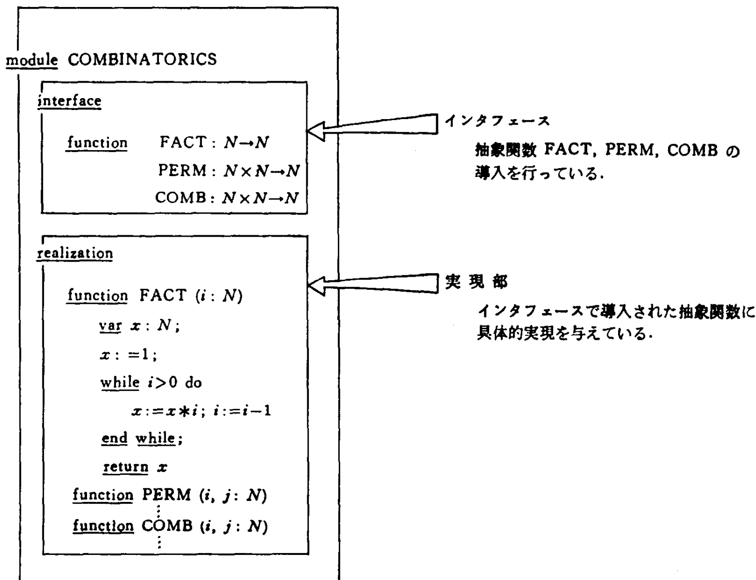


図-2　モジュールの例

3. モジュラー・プログラミングのための言語処理と情報管理

前章で議論したように，モジュラー・プログラミングの特長は個々のモジュールの信頼性を他のモジュールと独立して高めることを可能にする点にある．この達成には言語処理系および支援系（以下ではこれらをまとめてシステムとよぶ）もモジュールの独立性を活かすように設計されねばならない．たとえば言語処理系には構文，意味解析と目的コード生成をモジュールごとに行ういわゆる分割コンパイルが，また支援系にはプログラムのテストや変更がモジュール単位に行える機能がそれぞれ不可欠である．

システムがモジュールごとのプログラム開発を支援するといっても，個々のモジュールの意味解析やテストのためにはそのモジュールが参照する他のモジュールに関する情報も必要である．システムはこれらの情報をモジュール・データ・ベース（以下モジュール・ベースと称す）として管理しなければならない．各モジュールに関する情報はすべてそのモジュールのソース・テキストを解析することによって得られ，原理的にはソース・テキストのみをモジュール・ベースに蓄えておけば間に合う．しかし情報を必要とするたびにソース・テキストを解析するのは効率上およそ非現実的である．効率のよい支援のためには，同じ処理のくりかえしをせずに必要な情報をシステムが直ちに取り出せる形（たとえば内部形式のプログラムなど）でプログラムを保存する必要がある．

プログラム開発の途中では，モジュールは常に変更される可能性をもっており，モジュール・ベースの内容はきわめて流動的である．そこで問題になるのはモジュールの変更がモジュール・ベース内の情報に不整合を起す惧れがある点である．実現部の変更の影響はそのモジュールのみに限定され，不整合の検出，解消は比較的容易である．しかしインタフェースの変更はそれを参照するすべてのモジュールとの間に不整合を生ずる可能性があり，その影響はきわめて大きい．たとえばモジュールAがモジュールBで定義された関数 f を使って書かれているとしよう．Bのインタフェースの f の宣言が書きかえられた場合（f の引数の型や数が変更されたり f の宣言自体が除去されたりした場合），AとBの間に不整合を生ずることになる．不整合の検出と解消はプログラマの注意力のみにまかせられるべきではない．実際巨大なプログラムではこれらの不整合をシステムの支援なしに完全にチェックすることはむずかしい．また放置された不整合が累積されるとその解消は困難をきわめる．したがってシステムが不整合を検出しそのすみやかな解消を支援することが必要である．モジュラー・プログラミング言語は，モジュールのインタフェース情報を使ってこのような支援を可能にするが，十分な効果をあげるためには言語とシステムを一体として設計する必要がある．以下，さきにあげたモジュラー・プログラミング言語のうちAda を除く3つの言語について，それらの実際の処理支援系におけるモジュール・ベースの維持管理を不整合の検出解消の問題を中心にみていこう．Ada の支援に関しては，筆者が理解するかぎり基本的要請[19]が提示されているにすぎず具体的構想はいまだ発表されていない．

∘ CLU の場合[7]

CLU の処理系ではモジュール・ベースはライブラリとよばれる．ライブラリはいくつかの description unit (DU) から構成され各 DU には1つのモジュールのインタフェース情報，ソース・テキスト，および目的コードなどが蓄えられている．ライブラリ内では不整合を生ずる可能性のある変更はいっさい禁止される．すなわちDUの内容の変更は原則として許されない．すでにライブラリに登録されているモジュールを変更するには新しいDUを作成しなければならない．この結果，ライブラリ内には同一のモジュールの各版に対応して複数のDUが存在することになる．1つのモジュールのコンパイル時にはそれが参照する各モジュールについてどの版が実際に使われるのかをプログラマが指示する必要がある．

ライブラリにはその名前からもわかるようにすでにデバッグも終り相当安定したモジュールのみを登録することが前提となっている．これではシステムは未完成なモジュールを管理することを全く放棄しており，モジュールを完成に導く過程での支援にライブラリは役に立たない．先にものべるように，効果的なプログラム開発には不完全なモジュールこそモジュール・ベースに格納しておく必要がある．

∘ Mesa の場合

以下は Mesa で書かれた OS, Pilot[5] のプロジェクトで採用された方法である．ソース・テキストはモジュールごとにファイルに格納され，各モジュールの目的コードも目的ファイルとよばれるファイルに別々に格納される．これらのファイルがモジュール・ベース

を構成しており，各モジュールの目的ファイルに格納されたインタフェース情報を使って言語処理系は分割コンパイルを行う．モジュール・ベースの管理の主体はあくまでプログラマであるが，不整合の可能性が処理系によってある程度検出されるように以下の工夫がされている．

いまモジュールBがモジュールCを参照しているとする．Cのインタフェースが変更されるとBとCの間に不整合が生じる可能性がある．したがってプログラマはBの内容を調べ，必要があれば新しいCと整合するようにBを修正し，新しいCの目的ファイルを使ってBをコンパイルしなおすことが要求される．これら一連の処理を怠った場合はBの目的コードが古いCの目的ファイルを使って作成されたまま残ることになる．そこでCのどの版の目的ファイルを使ってBの目的ファイルが作成されたかという情報を保存することにより，処理系がその情報と現在のCの版を比較すれば不整合の可能性が検出される．(**図-3**参照．ここでBのコイパイル時に使われたCの版数C.1と新しいCの版数C.2が異なっている．)

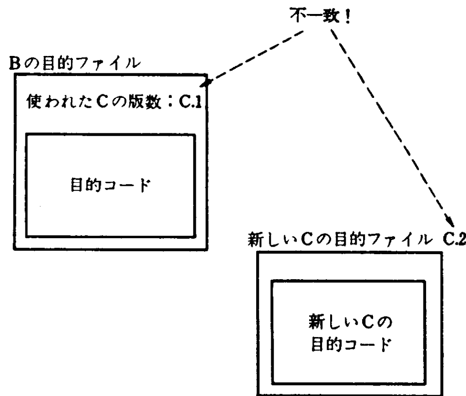


図-3 言語処理系による不整合の検出

この方法はきわめて単純であり，簡単な機能だけで実現できるが，次のような難点がある．第一の点として，モジュールの変更によって不整合が生じても直ちに検出されるわけではない．たとえば図-3において，BとCの間の不整合はコンパイルの必要性から処理系がBの目的ファイルの内容をチェックするときのみ検出される．こうして，不整合が累積することになりかねない．第2の点として，この方法は不整合の可能性のみを検出するもので，実際に不整合を生じているか否か，あるいはどのような不整合なのかなどすべてプログラマが調べなければならない．実際Cのインタフェース変更が必ずしもモジュール・ベース内に不整合を起すとは限らない．たとえばCに新しい関数を加えたり，全く使われていない関数を除去したりした場合は不整合は生じないが，そのような場合にもCのコンパイルがやりなおされることから不整合の可能性が検出されてしまい，プログラマに余計な負担をかけることになりかねない．次にのべるイオタ・システムは，モジュール・ベース内の不整合の検出，解消のための系統的な支援を行うことにより，このような困難を克服している．

° イオタ・システム[12]の場合

イオタ・システムはイオタ語によるプログラム開発と検証のための総合的な会話型プログラミング・システムである．各モジュールは入力されると直ちに構文，意味解析がほどこされ，木構造の内部コードに変更されモジュール・ベースに格納される．各サブ・システムはこれらの内部コードからすべての必要な情報を直接取り出し，次章で述べるようにモジュールの変更は内部コードに対して直接行われる．

内部コードがさまざまな会話的支援機能に共通に使われることから，内部コードの整合性が常に保たれることが不可欠であり，システム全体がこれを保証するように設計されている．特にインタフェースが変更されたときには，内部コードの整合性を保ちつつプログラマによるモジュールの修正を支援するための，一見単純だが簡明で効果的な方法が取り入れられている．たとえばモジュールMで定義された関数fの宣言が変更されると(**図-4**参照)，fを使っているAやBとの間に不整合が生じる．システムはまずfを使っているモジュールの有無を調べ，もしあれば変更前の宣言によるfをf%1と改名して保存し，同時にAやBにおけるfも自動的にf%1におきかえる(図-4 b参照)．f%1はAやBが新しいfに整合するように書き改められない限り保存される(図-4 c)，d)参照)．(もしA，Bが改められる前にfが変更されればfはf%2として保在される．(図-4 d)参照)) この結果，変更してきた過程と修正すべき個所が一目瞭然となり，複雑な変更がくりかえされても修正が容易となる．また変更をとり消して以前の状態を復元することが容易になるなどこの方法はきわめて融通性に富んでいる．さらに内部コードの間に常に整合性が保たれることから，後述の構文エディタをはじめとする内部コードの情報にもとづいた会話型支援機能の信頼性が保証される．

イオタ・システムではこの他にもモジュールの管理

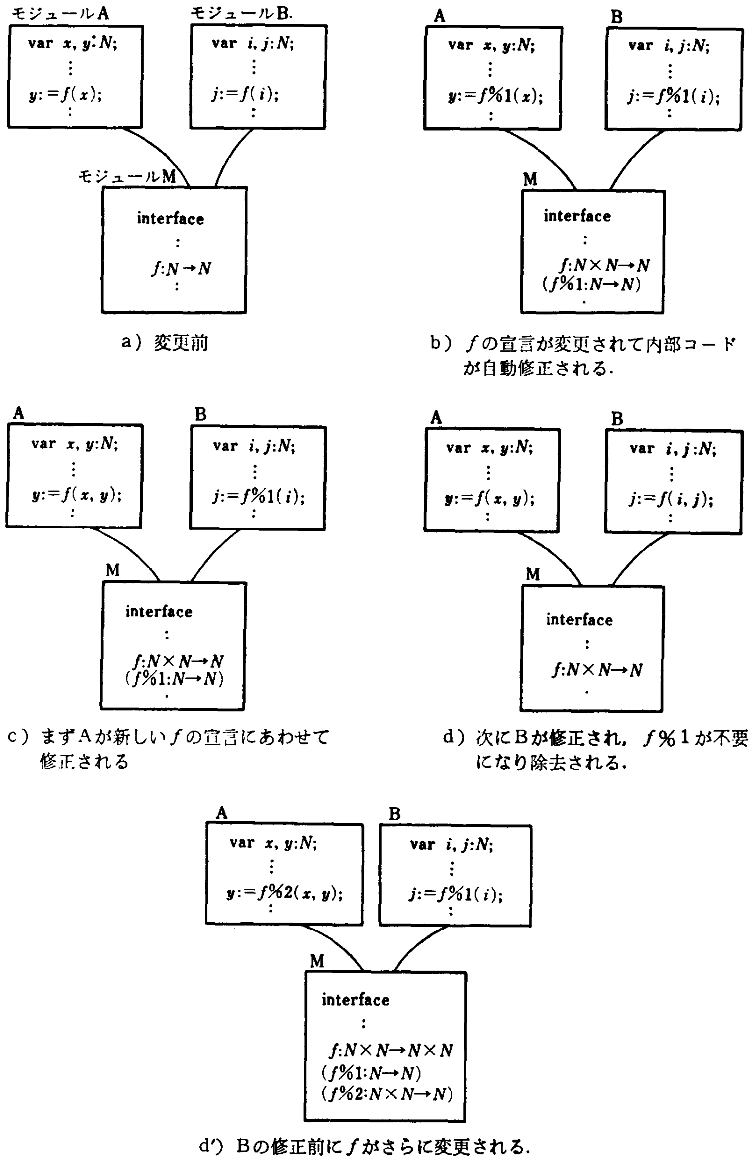


図-4 インタフェースの変更

に関してプログラマの労力を軽減するために，従来は分割コンパイルを行う処理系において大きな負担であった目的プログラムの管理などを完全に自動化するなど，さまざまな工夫がなされている．目的コードはモジュールごとに作成されるが，プログラマが1つのモジュールを指定することによってそのモジュールの実行に必要な他のモジュールの目的コードも次々と自動的に作成することができる．この際に，目的コードがまだ作成されていないモジュールおよび目的コードの再生成を必要とするモジュール（すなわち古い目的コード作成後変更のあったモジュール）のみが選択され処理される．

4. 対話的モジュラー・プログラミング

プログラミング・システムは，プログラミングに必要な情報をプログラマと交換しながら管理する情報シ

ステムであると考えられる．前節で論じたように，モジュラー・プログラミングを効果的に支援するためには，システムは多くの情報を管理する必要がある．仮に同じ言語をサポートするシステムであっても，それらのシステムの情報管理の巧拙によってプログラミングの効率が大きく異なることは論をまたない．

モジュラー・プログラミングを支援するプログラミング・システムには，高い対話性を有するものが適している．ここでいう『高い対話性』を有するシステムとは，次の条件を満たすシステムをいう．

（ⅰ） 単位時間当りのシステム－プログラマ間で行われる情報交換の頻度が高い．

（ⅱ） 一回に交換される情報の量が適当でかつその質が高い．すなわち情報の内容が適切で，その形式が理解しやすく与えやすい．

条件（ⅱ）は特に重要で，内容を理解するのに多大な時間と労力を要するメッセージを長々と羅列するようなシステムはきわめて対話性が低い．モジュラー・プログラミングにおいては，前章でも述べたように，プログラマは通常ある1つのモジュールの完成に精力を集中している．またモジュラー・プログラミングの利点を生かすために，プログラマは個々のモジュールを比較的小さくデザインしており，1つのモジュールに関する情報量は割合少ない．すなわちモジュラー・プログラミングにおけるシステム－プログラマ間の情報交換の形態は上にあげた条件に適合している．一方，モジュールの中では他のモジュールで導入された抽象概念を数多く用いているため，それらのモジュールの内容，特にインタフェースなどの情報を参照する必要が頻繁に生ずる．したがってモジュラー・プログラミングには条件（ⅰ）も満たすような高い対話性を備えたプログラミング・システムがきわめて有効である．

高い対話性を有するプログラミングシステムの例として，INTERLISP[13]などの Lisp 系プログラミング・システムがあげられる．Lisp の場合にも，モジュラー・プログラミング用言語と同様の理由によって，高い対話性を有するシステムが適しており，プログラム作成の効率化に大きな効果をあげている．前節で紹介したモジュラー・プログラミング・システムの中では，特にイオタ・システムが高い対話性を目標に設計されている．

抽象化を多く用いて記述されたプログラムは解析するのに比較的時間がかかり，モジュラ・プログラミング・システムに高い対話性を持たせるためには，さまざまな工夫が必要とされる．たとえばイオタ・システムでは，処理効率の良い対話的なモジュールの入力・修正機能[12]を導入することによって，この問題に対処している．

イオタ・システムのモジュール入力・修正サブ・システムは，ソース・テキストを編集する文字列エディタ，構文木を編集する構造エディタおよび二種類の構文解析部の有機的結合体となっている（図-5）．構文解析部Ⅰは，型のチェックなど他のモジュールのインタフェースを参照する必要があるような解析は行わず，構文解析部Ⅱがこれを行う．すなわち，構文解析部Ⅰを通ったテキストは，関数や変数の型チェックなど未解析部分も残っているが，一応木構造（これをスケルトンと呼ぶ）となっている．このスケルトンが構文解析部Ⅱに送られて未処理部分の解析が行われ，次々と完全な構文木へと変換される．この段階でエラーが検出された場合には，構造エディタを用いて構文木を部分的におきかえることによって修正を行う．修正された構文木が再び構文解析部Ⅱへ送られて解析されるが，構文解析部Ⅱは処理の必要な部分のみを解析するように設計されており，ここでは一切の処理の重複が避けられている．モジュールの修正の際も，モジュール・ベース内に内部コードとして格納されている構文木を，構造エディタに送って直接編集することによって，行われる解析を必要最小限にとどめることができる．このようにイオタ・システムでは，処理の重複を極力避けてシステムの応答速度を十分速くするのに成功している．

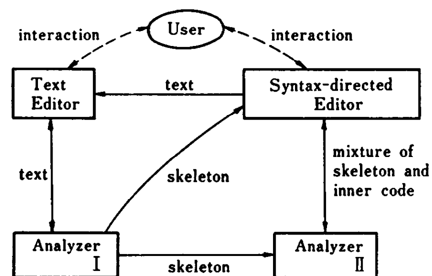


図-5 イオタ・システムのモジュール入力・修正サブ・システム

対話的プログラミング・システムにとって，各処理の応答速度と並んで重要な問題として，システムの統合性があげられる．統合性の高いシステムとは，各サブ・システム（エディタ，コンパイラ，デバッガなど）が有機的に結合されているものをいう．すなわち，各

サブ・システム間の移動の際に，必要な情報がプログラマの意図に沿って迅速かつ的確に伝達され，システム全体として制御と情報が自然にかつ円滑に流れることが要請される．INTERLISP やイオタ・システムなど，最初から言語の処理系および支援系全体が一括して設計されたインテグレーテッド・プログラミング・システムは統合性にもすぐれており，統合性の低いプログラミング・システムにくらべ，はるかに効率的なシステムとなっている．

5. ソフトウェアの共同開発とその支援

大規模ソフトウェアには，プロジェクト・チームによる共同作業が不可欠であり，モジュラー・プログラミングがこの目的に効果的であることは広く認められている．

このような状況の下では，1つのモジュールの変更がそのモジュールを直接あるいは間接に利用している他のプログラマに影響を与える可能性がある．したがって共同開発をスムースにすすめるためには，変更の影響をできる限り局所化することが重要である．そのための一手段として，イオタ・システムにおける仮想モジュールの概念を紹介しよう．いまモジュールAがモジュールMを参照しているとする．モジュールBでもMを参照したいが，BからのMに対する要請がわずかに異なっているとする（図-6 a）参照）．この状況はAとBの作成者が異なる場合に起る可能性が高い．Bの要求にみあったMの変更の影響をAに及ぼさないためには，Mのコピー M′ を別個のモジュールとして作成することが考えられる（図-6 b）参照）が，これはメモリの無駄であるうえに，せっかくのMとM′の関係が失われてしまうという欠点がある．ことにMとM′ には共通した部分が多く，さまざまなプログラム開発上の労力が重複することになる．このような場合に，イオタ・システムではM′をMの仮想モジュールとして生成できる．すなわち M′ の実体が存在するのではなく，システムはMとM′の相異に関する情報のみを記憶しておく（図-6 c）参照）．MとM′に共通な部分の変更はMに対してのみ行えばよい．また2つのモジュール間の関係が明確になるなど以後の維持管理にも有効である．

仮想モジュールの概念は広い応用性をもち，上の目的以外にも様々な応用が考えられる．たとえば Mesa のいわゆる context switch の機能もこれによって実現可能である．Mesa にはモジュールの配置，結合を

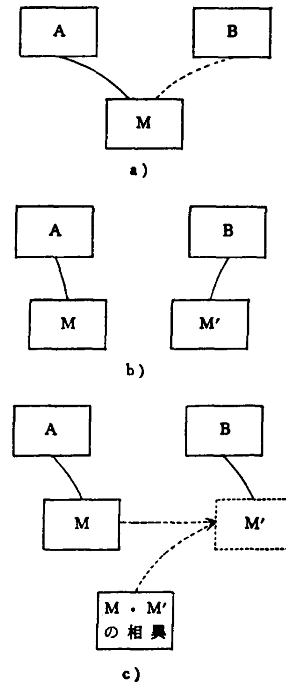


図-6

行うための C/Mesa[9] とよばれる言語（configuration language）があり，1つの抽象概念に対して複数の実現を用意しそれらを使い分けることが可能であるが，仮想モジュールを使えば同様の機能が特別な言語の導入なしに実現される．すなわち，実現部のみ異なる複数の仮想モジュールを作成し，他のモジュールからの参照関係を適宜切り換えるだけでよい．

円滑な共同開発のためには，各プログラマ間の的確な情報伝達に対する支援が欠かせない．たとえば，Intercol では，モジュールのインタフェースに変更が加えられた場合，影響を受けるモジュールの作成者に適当なメッセージを送る機能が導入されている．イオタ・システムでは，単なるメッセージの交換にとどまらない緊密な情報伝達を行うことによって，複数のモジュール・ベース上での共同開発を支援している．イオタ・システムでは各プログラマは独自のモジュール・ベースを用いる．モジュールをモジュール・ベース間で転送することにより，他のプログラマの作成したモジュールを利用できる．いまモジュール・ベースAで作成されたモジュールを他のモジュール・ベースBで使いたいとしよう（図-7 参照）．B のプログラマの作成するモジュールNがMを参照するときは，Nの

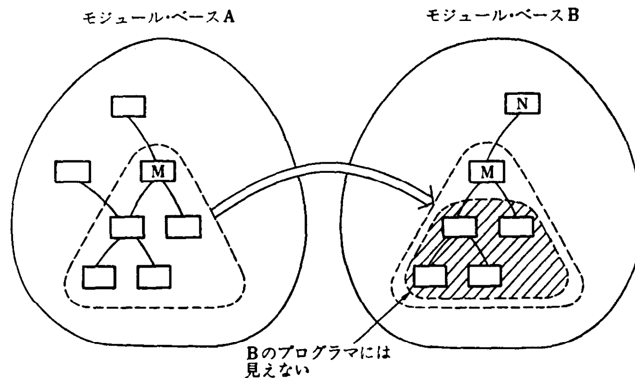


図-7 複数モジュール・ベースによるモジュールの共有

テストや実行のためにはMのみならずMが参照するすべてのモジュールもBに転送されねばならない．そこでシステムはM転送の要請に対し，必要なすべてのモジュールも同時に転送する．これらのモジュールはAとBに共有されることになるが，その管理は作成者，つまりAのプログラマにゆだねられ，B内での変更は禁じられる．Bへ転送されたモジュールのうちM以外はBの利用者に見える必要はなく，テストや実行などの場合を除いてはあたかもB内に存在しないかのように取り扱われる．これによってシステムは不必要な情報がBのプログラマに与えられることを回避する．Bに供給されているモジュールがA内で変更されると，システムはそれらをBへ再転送するようにうながす．再転送によってB内に不整合を生ずるか否かのチェックも自動的に行われ，必要であれば3章で述べたような方法によって不整合の解消が支援される．このようにして各プログラマのモジュール・ベース間の緊密な情報交換が支援されており，共同開発を効率よくすすめる環境が与えられている．

6. おわりに

ソフトウェアの作成と維持における諸問題が深刻化する中で，開発支援環境の重要性に対する認識はますます高まりをみせている．開発支援環境の整備は急務であるとはいえ，基礎となるべき方法論から遊離した対症療法的支援ツールの寄せ集めでは大きな成果は期待し難い．本論文ではモジュラー・プログラミングに対する支援を考える際に，常にその方法論の原点に立ち返って議論するよう努めた．基礎となる方法論の見直しなしに支援環境の本質的発展はありえないからである．

参 考 文 献

1) Dennis, J. B.: Modularity. Software engineering - an advanced course, Lecture Notes in Computer Science 30, Springer-Verlag (1975).
2) Dijkstra, E. W.: Notes on structured programming. Structured programming, A. P. I. C. Studies in Data Processing No. 8, Academic Press (1972).
3) Geschke, C. M., Morris Jr. J. H. and Satterthwaite, E. H.: Early experience with Mesa. Comm. ACM Vol. 20 No. 8 (1977).
4) Honda, M. and Nakajima, R.: Interactive theorem proving for hierarchically and modularly structured sets of very many axioms. International Joint Conference on Artificial Intelligence, Tokyo (1979).
5) Horsley, T. R. and Lynch, W. C.: Pilot: a software engineering case study. Proceedings 4th International Conference on Software Engineering (1979).
6) Lauer, H. C. and Satterthwaite, E. H.: The impact of Mesa on system design. Proceedings 4th International Conference on Software Engineering (1979).
7) Liskov, B., Russell, A., Bloom, T., Moss, E., Schaffert, C., Scheifler, B. and Snyder, A.: CLU reference manual. Lecture Notes in Computer Science 114, Springer-Verlag (1980).
8) Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C.: Abstraction mechanisms in CLU. Comm. ACM Vol. 20, No. 8 (1977).
9) Mitchell, J. G., Maybury, W. and Sweet, R. E.: Mesa Language Manual. Technical report CSL-79-3, Xerox Corporation (1979).
10) 中島玲二：Ada「批判」，情報処理，Vol. 22, No. 2 (1981).

11) Nakajima, R., Honda, M. and Nakahara, H.: Hierarchical program specification and verification - a many-sorted logical approach-. Acta Informatica 14 (1980).
12) Nakajima, R., Yuasa, T. and Kojima, K.: The iota programming system - a support system for hierarchical and modular programming-. Proceedings of IFIP 80, ed. S. H. Lavington, North-Holland Pub. Co. (1980).
13) Teitelman, W.: Interlisp reference manual. Xerox Palo Alto Research Center, Palo Alto (1978).
14) Tichy, W. F.: Software development control based on module interconnection. Proceedings 4th International Conference on Software Engineering (1979).
15) Wirth, N.: Program development by stepwise refinement. Comm. ACM Vol. 14, No. 4 (1971).
16) Yuasa, T. and Kojima, K.: The iota programming system manual (1982 (to appear)).
17) Yuasa, T. and Nakajima, R.: The iota language manual. (1982 (to appear)).
18) ADA reference manual. SIGPLAN Notices 6 (1979).
19) Requirements for Ada programming support environments "stoneman". Department of Defense (1980).

(昭和57年3月19日受付)