視　点

モジュール化

横浜国立大学大学院国際社会科学研究科教授　　鳥居　昭夫

最近、「モジュール化」という言葉を良く聞くようになった。この言葉は、最近の生産マネジメント・産業構造の動向を論じるには避けて通れないキーワードとなっている。ところが、「モジュール化」という言葉は、コンピュータのプログラミングに携わったことのある人にとっては、長く聞きなれた言葉である。筆者は、現在生産マネジメントにおいて議論されている「モジュール化」は、プログラミングにおける「オブジェクト指向」に、多分とても近い概念ではないかと思っている。前者のいくつかの重要な論点にも、後者の例を考えることによって理解しやすくなる点が多い。今回は、経営学におけるモジュール化を、プログラミングのモジュール化に沿った形で紹介し、稿をすすめたい。

私事で恐縮だが、筆者は産業組織論において実証分析を専門としてきたので、計量的処理に長く携わってきた。特に計量分析を始めたころは、当時としてみれば大量のデータを扱わなければならなかったことと、通常の統計分析パッケージには組み込まれていない処理を行う必要があったため、多くのプログラムを書かざるを得なかった。そのため幾つものプログラム言語を試したが、その中で特に印象が強かった言語はModula-2とforthである。このうちModula-2はPascal系の言語で、「構造化言語」と呼ばれた言語の一つである。経営学で「モジュール化」という言葉を始めて聞いたときにも、筆者はまずこのModula-2を思い浮かべた。ここでは、これらのプログラム言語を簡単に紹介することから始めるが、筆者はあくまでユーザーとして多くの言語を使っただけであり、プログラミング論を研究した経験は無いため、以下の解説には多く誤解が含まれているかもしれないということを、最初にお断りしておきたい。

1　Modula-2

プログラムというのは、「どのデータとどのデータを使ってこういう計算をして、ここに覚えておけ」とか、「この条件が満たされていればああしろ、そうでなければこうしろ」等の「命令」を組み合わせて一定の処理を行う仕様を書き下したものである。初期に発達した、FORTRANやBASICなどでプログラムを書くと、基本的には計算処理を行う手順にしたがってこれらの命令をずっと繋げたものができあがる。ところで、一連の処理を行う時には、かならず繰り返し行う手続きがある。たとえば統計処理のプログラムにおいては、頻繁に逆行列を計算しなければならない。逆行列を計算する対象となる行列が異なるだけで、基本的な作業は同一であり、「ある行を対角要素で割り算する」などの命令の並び方は変わらない。そこで、この作業をまとめて一つの独立なユニットとし、このユニットを繰り返し使えばプログラミング作業はずっと楽になる。作業が楽になるだけでなく、プログラムの流れが分かりやすくなる。たとえば、「ある行列を得た後、第１行第１列の

要素で第１行を割って、次に……」と書いてあるより、「ある行列を得た後、逆行列をとって」と書いてあるほうがはるかに分かりやすいだろう。このように、プログラムを整理するとプログラミング作業が効率化できるだけではなく、処理を行う手続きの構造がずっと見やすくなる。このプログラムの整理は「構造化」と呼ばれている。Modula-2は一つ一つのユニットの関係を整理してまとめたモジュールという単位を作ることを、プログラマーに要求することによって、構造化を行いやすくすることに徹底した言語であった。

ところで、ある行列の逆行列を計算する場合には、通常もとの行列の他に別の行列を用意し、この二つの行列に対して一連の処理をすることによって逆行列を作り出してゆく。このとき当然、結果として逆行列が算出できさえすれば、作業の過程で計算されるこれらの行列や変数は必要ない。必要ないが、それぞれ名前も割り当てられるし、コンピュータの中でメモリーなど一定の資源も消費している。それらの作業用の変数の存在を意識しないで他の部分をプログラムした場合、様々な干渉を起こしてしまい「バグ」と呼ばれるプログラム・ミスの原因となる。たとえば、同じ名前を持った作業用の行列を、以前に使ったことを忘れて変数として使ってしまうと、意図しないメモリーを操作してしまうことになり、プログラムのほかの部分の動作を妨害してしまうかもしれない。このように見せなくとも良い部分は、なるべく外からはアクセスできない方が、プログラミングにおいて余計な神経を使わなくとも良いので楽である。Modula-2は、モジュールのうちで外に見せなくとも良い部分と、外に見せインターフェイスに使う部分とを、明確に区別するように要求した言語であった。

筆者は、プログラム言語のユーザーとして非常に怠惰なプログラマーであった。「怠惰な」という意味は、プログラムを書く時、再利用することをほとんど考えないし、あとでプログラムを見直すことも考えないと言うことである。プログラムの一部を再び用いたり、あとで処理の仕様を微調整したりということをあらかじめ考えて、変数の役割や処理の手続きをプログラム内に書き込んでいけば、分かりやすいプログラムになり、後にプログラムを解読することも容易になり作業を続けやすいし、プログラムのある一部分を再利用するという作業も容易である。この準備作業をドキュメンテーションという。しかし、このドキュメンテーションにはかなりの労力を費やさなければならないということは想像に難くないだろう。そのため、とにかく何とか早くプログラムを完成させてデータの処理をしたいという時には、わざわざ手間のかかるドキュメンテーションを、あるかどうかも分からない再利用のことを考えて措置しておくなどとはしないものである。そうなると、自分の書いたプログラムも１週間も経てば、下手をすると１日経てばなにを意図していたのか分からなくなることが多く、書き直しを余儀なくされることも多かった。Modula-2で驚いたのは、作成に数ケ月も掛かった膨大なプログラムを書いた時、途中で他に用事が入って1ケ月作業が中断した後にも、特に問題なく連続して作業を続けられたことである。もちろん怠惰なプログラマーとしては、意識してドキュメンテーションすることなどしなかった。仕様として構造化を要求されているので、自然と分かりやすいプログラムになっていたのである。

しかし、結果としてModula-2はあまり使わなかった。とにかく早く、動きさえすればよいプログラムを完成させたいという時には、面倒な指定を多く要求されるModula-2のような構造化言語を用いず、FORTRAN型の言語で、全体の構成を忘れないうちに一気に書いてしまうのが一番早

い。同様に、モンテ・カルロ実験など何千回何万回とシミュレーションを繰り返すためにできるだけ速く動かしたい、ないしは、完成したプログラムはできるだけコンパクトに小さくまとめたい、等という特定の効率化を狙ったプログラムはFORTRAN等で書くのがよい。筆者が学部生の時にデータ処理の授業で、フーリエ解析をできるだけ短くという課題が出されたことがあった。皆、はりきって様々な「裏技」を駆使した。たしか、もっとも短いプログラムはパンチカード５枚（80カラム×５）だったと思う。このようなまねは構造化言語にはできないだろう。

繰り返し使うモジュールにおいても、Modula-2のように外から見えないように設定した変数は絶対にアクセスできないようにせず、適当に外から操作する「裏技」を可能にした方が、効率良いプログラムを書くことが出来る。たとえば、逆行列を計算する時に副産物として算出されメモリーの片隅に残っている数字を、後になってからちょっと見に行って使うと、速いプログラムが書けるかもしれない。もちろん、こういう風に書かれたプログラムは、まず１週間経てばたとえ自らが書いたプログラムであっても解読不能となってしまう。逆に言うと、Modula-2等の構造化言語は、明瞭さ作業のしやすさでは優れていても、プログラムの速度などの効率という面においては劣っていたのである。もっとも、最近はコンピュータの速度と容量が劇的に改善されて、メモリーという資源の制約もほとんど気にしなくて良くなったため、効率を考えてプログラムする必要はほとんど無くなっている。

構造化言語によって、命令の手順は構造化されたことにより、仕様さえしっかり作れば複数の人で一つのプログラムを組むことが可能になった。しかし、それぞれのモジュールが扱うデータはモジュールの外にあるので、データにそれぞれプログラムを組む人が意図しない変更がプログラムの他の部分によって施され、相互に干渉が起こってしまう可能性は残っている。統計処理などの計算を行うプログラムの場合、目的は一連の計算処理の結果だけを得ることなので、干渉による問題は少ない。しかし、多くのプログラムは、単にデータの処理を行うだけでなく、一定の機能を期待されている。データを通じた干渉が起きると、この機能が満たされなくなる可能性がある。そこで、モジュールにデータと一定の機能を実現するためのデータに対する処理を合わせて入れてしまい、外から使用できるのはモジュールの機能だけにしてしまうという発想が生まれる。これが「オブジェクト指向」である。構造化言語の逆行列計算モジュールは、行列を渡せば逆行列を返してくれる。オブジェクトは「さっき問い合わせたデータの逆行列は存在するかい？」あるいは「あの行列の逆行列にその行列をかけた時、対角要素の和はどうなるの？」という問いに応えてくれるかもしれない。オブジェクト指向言語はこのように、機能を果たすのに不必要で、見せなくともよいものは徹底して見せないのである。

ここまでに述べたことを、以下にまとめておく。

①構造化され、よりオブジェクト指向の性格の強い言語ほど、プログラムの構造が明瞭になっており、それぞれのモジュールは部品として独立性が高く、単独のプログラミング（生産）が容易である。

②しかし、モジュールを構成するためには、追加的なコストがかかる。

③構造化され、よりオブジェクト指向の性格を持つ言語ほど、モジュールの中身は隠蔽される。

④モジュールによる構造化と、プログラムの効率性の間にはトレード・オフの関係がある。

2 「モジュール化」

経営学におけるモジュール化については、『モジュール化　新しい産業アーキテクチャの本質』（青木・安藤［2002］）という優れた研究書・解説書が出ているので、詳細はそちらを参照されたい。その本の中に収録されているボールドウィン・クラークの論文には、モジュール化は「それぞれに独立に設計可能で、かつ全体として統一的に機能するより小さなサブシステムによって複雑な製品や業務プロセスを構築すること」と定義されている。プログラミングをそのまま生産活動と読み替えれば、両者におけるモジュールの意味はほとんど等しい。モジュールの典型は、世界のメイン・フレーム市場を席巻したIBM System/360や、パソコン市場で後発だったにもかかわらず、もはや並ぶもののない同じIBMのPC/ATのモジュールであると考えられている。それぞれは、一定のデザイン・ルールの下で、能力も信頼性もコスト競争力も急激に進歩した製品に進化した。

ここで、先ほどプログラミングにおけるモジュール化から導いた４つの性質を、経営学の文脈にあてはめて説明してみよう。まず①はモジュールの基本的性質を示している。それぞれのモジュールの機能を明示的に設定した、全体の構成のデザインはアーキテクチャと呼ばれている。モジュール化を志向した優れたアーキテクチャの下では、それぞれのモジュールは高い独立性を持っているので、インターフェイスと呼ばれる設定された窓口をとおすもの以外の部分で他のモジュールとのコーディネーションをすることなしに、自律的に改善することができる。すなわち、局所的な最適化をはかることが容易となる。この改善は、競争の力を借りることによってさらに促進することができる。

ただ、プログラミングの場合には設定さえきちんとすればモジュール間の干渉を遮断できるが、生産過程に読み替えた場合、一つの製品の内で発生する物理的な干渉を完全に阻止するのはすこぶる難しい。したがって、整合性がありしかも干渉が少ないモジュールを構成するためにはコストがかかる。すなわち、②にあるように、モジュール化は初期にそれなりの投資を必要とする。

また、モジュールを構成するにあたっては、生産組織が扱う情報は「目に見える情報」と「隠された情報」とに分けられる。「目に見える情報」とは、ボールドウィン・クラーク論文では、システム設計の初期段階で確立され広く公開される「明示的なデザイン・ルール」として示され、アーキテクチャと、モジュール間の相互の情報交換の仕様を定めた「インターフェイス」、さらにモジュールのデザイン・ルールへの適合性を評価する「標準」からなると説明されている。一方、「隠された情報」は「そのモジュールを超えて他の設計に影響を及ぼさない意思決定」である。あるモジュールに携わるものは、他のモジュールの中の「隠された情報」を気に懸ける必要がない。当然、モジュールの中に情報が隠されれば隠されるほど、事後的なコーディネーションが容易となる。③と同様、いかに情報を秘匿するかがモジュール化の成果を生み出すためのポイントとなる。

最後に、モジュール化にはそれなりのコストが伴うと考えられている。このコストは、②の初期設計コストだけではない。一つのモジュールの中では「隠された情報」デザイン・パラメータをそのモジュールの中で最適化する。この最適化は、あくまで局所的な最適化であり、システム全体の最適化とは異なるかもしれない。局所的な最大化の積集合はあくまで条件付の最大化の解（セカンド・ベスト）にすぎず、ファースト・ベストとは異なるだろう。プログラミングにおける④と同様

に、システム全体の最適化を目的とするのであれば、モジュールにまたがった資源を用いた方がファースト・ベスト解に近く、より効率的となるかもしれない。

ただ、現在は多くの産業・分野においてモジュール化による品質の向上と効率化への効果が着目されている。恐らくここ当分の間、モジュール化は産業の競争力を決定するキー・ファクターであると考えられるだろう。

3　FORTH

以上のように、プログラミングのモジュールと生産マネジメントのモジュールとは非常に類似した関係がある。どちらもデジタル化したコンピュータから派生した概念だから、当然といえば当然なのであるが。

ここで、また少しプログラム言語に戻らせていただきたい。この稿の最初の部分でModula-2とあわせてFORTHという言語について言及した。FORTHはその他のプログラム言語と異なり非常にユニークで効率的な言語であった。紙面の関係で詳細は省くが、プログラムも日本語に近い形で記述できる。たとえば、

ある 数字 と 同じ数字 をつくり　この二つを 掛ける これを 自乗 と名付ける

3 の 自乗 と 5 を 足す と いくつ？

のような形でプログラムが書けるのである。このような形でどんどん新しい命令を追加することができる。この命令をWORDという。プログラムとは、WORDを次々に組み合わせて新しいWORDを作っていく作業に他ならない。さらに、FORTHの処理系は非常にコンパクトであるという特徴を持つ。そのためか、いまだに制御系の分野では根強いファンが実用に供しているそうである。筆者も、FORTHの処理系を他の言語と組み合わせてプログラムを単純にするなど、一時とても重宝して用いていた。

このFORTHという言語は、他の構造化言語とは大きく異なっているのであるが、一つ一つのWORDは独立したモジュールとなっている。FORTHの特徴の一つは、非常にフレキシブルなシステムを構成することができることである。WORDを積み木のように組み合わせて新しい部品を作り、その部品を組み合わせて次の部品を作り、という作業を、システムのかなり基本的なところから組むことができるからである。したがって、アイデア次第でどんどん新しい機能を持つ部品を創造することができる。

プログラムのモジュールはこのように、独立したモジュールを組み合わせて、自由に展開できる可能性を生むものである。考えてみれば、プログラミングの作業自体において、プログラマーが、「X＋2を計算して記憶場所Yにおけ」というプログラムを書く時、「IOモジュールがキーボードの入力を走査していて、こうやってXのキーを押すと、割り込みがかかって……、CPUの中でANDとXORがこう組み合わさって、……」などと考えることはない。高級と呼ばれるプログラム言語であればあるほど、多層をなす何段階ものモジュールの処理の上に立った作業を行うことになる。このおかげで、作業の中身にある「隠された情報」に煩わされることなく、抽象的なレベルでプログラムを組むことができる。言い換えれば、モジュール化を進めていくほど、創造の可能性が高まる。

現在まで、生産マネジメントのモジュールについての議論においては、モジュール生産の効率性や調整の容易さ、モジュールによる製品の能力・品質の向上という側面が強調されてきた。その中で『モジュール化』に収録された青木論文では、シュンペーターの「古い結合を創造的に破壊して、新結合を実現する」モジュールによる進化の可能

性を示している。今後の「モジュール化」が貢献すべき方向を示唆するものと思われる。

最後に、前回でも紹介したカウフマン博士の*At Home in the Universe*では、モジュール化の全く異なった側面を議論しているので、簡単に紹介しておきたい。カウフマン博士はモジュール化に漏れて残ってしまったモジュール間の干渉に着目する。当然これらの干渉は、モジュール間の調整においてコンフリクト（衝突）を発生させる。システムがモジュール構造を持っていない場合には、部分（部品）間の関係が複雑であればあるほど、システムの適合性は、パラメータの変化に対して一様でなく凹凸の多い関係を持つようになる。したがって、なんらかの最適化を行っても選ばれた解は、局所的最大化の解に過ぎないかもしれない。このように複雑なシステムほど、局所的最大化の罠にはまりやすく、大局的最大化を実現しづらい。しかし、カウフマン博士は、モジュール間のコンフリクトがあることによって、かえってシステム全体が局所的最大化の罠に落ち込んでしまうことなく、システムを罠から抜け出させ、大局的最大化を実現させることが可能になると論じている。各モジュールでは、おのおのの状態を最適にしようと試みる。そのままほっておかれれば局所的最大化に終わってしまう。しかし、隣接するモジュールとのコンフリクトがあるため、この局所的最大化の状態に安住していることが出来ない。そのため、さらに最適な解を求めて変化してゆけるというのである。

文献

青木昌彦・安藤晴彦編著［2002］、『モジュール化　新しい産業アーキテクチャの本質』、東洋経済新報社。

Stuart Kauffman［1995］*At Home in the Universe*, Oxford University Press.