オブジェクト指向ソフトウェア開発方法論への一考察

—新しいソフトウェアの再利用研究に向け—

A study on Object-Oriented System development method
Towards the NEW software Re-Use model and method

春木　良且

0.はじめに（問題提起）

米国Xerox社の開発による言語処理系Smalltalk-80によって実現されたオブジェクト指向技術が生み出されてから、約四半世紀が経過した。その間オブジェクト指向型データベースや、ウィンドウシステムなどを含めたPC上の開発環境などが実現され、またいわゆるオブジェクト指向分析、設計などの開発手法も洗練されてきており、今やオブジェクト指向技術自体、特殊な先端技術ではなくなってきている。筆者が最初に入門書[1]を書いた時点とは、おおげさではなく隔世の感がある。

そうした状況において考えるべきは、当たり前のように取り入られ行われている一連のオブジェクト指向開発方法論とその有効性についてである。言うまでもなく「モノ作り」のための学問である工学（エンジニアリング）はプラクティカルな領域であり、最終的な評価はその研究が有効性を持つか否かである。どんな新規技術であっても、それが「モノ作り」において何らかの有効性を持たねば、工学上は価値を持たない。そのため、工学系の研究においては、実証実験とその評価が重視される。

確かにオブジェクト指向技術は、ソフトウェア開発において大きな有効性を持っている[2]。しかし[3]などで述べたように、有効性を持っていたのはオブジェクト指向技術そのものであり、それにもと

づいて実現されたシステムは、端的に言えばオブジェクト指向技術の持つ様々な利点を享受しているがゆえに有効なのである。その意味で言えば、オブジェクト指向型のシステムを実現するための手段として提起されている、いわゆるオブジェクト指向開発手法は、オブジェクト指向技術自体とは別に考察せねばならない。手段の有効性と結果（創作物）の有効性は別なのである。

特に工学の研究者として意識せねばならないのは、全てに有効な技術は存在しないということである。ソフトウェア工学の世界では、「銀の弾は存在しない（No silver bullet）」というフレッド・ブルックスの有名な言葉で示されるように[4]、ソフトウェア開発の現場が抱えている様々な問題を解決するための決め手となる技術は、少なくとも万能技術としては存在しない。本論文では、開発方法論の善し悪しではなく、オブジェクト指向技術を含め、その利点と限界を明らかにしていくことを目的とする。私見では、多くの開発方法論においては、それらの有効性に関する議論が欠如しているように感じることが多い。開発方法論が、基本的には開発現場から生まれてきており、さらにCASE（Computer Aided Software Engineering）などの開発支援ツールと共に商業ベースで展開してきたことも、そうした傾向の背後にあると言えるであろう。

本論文では、ソフトウェア開発手法、特にオブジェクト指向開発手法には何を期待すべきか、すなわち方法論の効果について考察する。特に問題は、一般的に喧伝されているように、手法に従うことで全ての開発者がその効果を実現し、享受できるのかという点にある。もしその通りだとすれば、優れた開発手法があれば、個々の技術者に開発のためのスキルは不要となってしまうはずであり、それこそ技術者自体が不要となってしまうということでもある。

1.「開発手法・方法論」と「開発パラダイム」

ソフトウェア開発手法とは、「開発ライフサイクル全体を通して、最適なシステムを作成する方法を体系づけたもの」と定義される[5]。但しここで言うライフサイクルは、特にオブジェクト指向手法においてはライフサイクルモデル自体を否定する傾向があり[6]、必ずしもウォーターフォールモデルのみを意味しているのではない。

表1には、オブジェクト指向技術が登場する以前の、主なソフトウェア設計技法を示す[2]。やはりオブジェクト指向技術の登場以前は、1970年代における構造化設計に関するものが中心的であり、それが様々に応用されているのがわかる。ここで示したように、いわゆる設計技法はそれだけが独立して構成されているのではなく、かならずその技法の裏づけとなる開発のための基本的な考え方が存在している。例えば代表的なものとしては、構造化設計に対する、構造化技法、あるいは構造化定理がある。

そうした一連の技法の背景にある、開発のための基本的な考え方を、ここでは「開発パラダイム」と呼び、前述の開発手法の定義を借りて、「最適なシステムの源泉を明らかにしたもの」と定義する。言うまでもなくパラダイム（paradigm）とは、科学技術史家のT.クーンによる言葉であり、科学と科学の歴史についての見方、科学論のことを意味する。[7]では、パラダイムを「一般に認められた科学的業績で、一時期の間、専門家に対して問い方や答え方のモデルを与えるもの」と定義している。

そうした意味から言えば、明らかに構造化定理は構造化設計を実践する際の基本的なモデルであり、同様にオブジェクト指向技術は、オブジェクト指向分析、設計等手法に対する基本モデルであると言えよう。つまり開発技法とは、ある特定の開発パラダイムに基づいたシステムを実現するための、いわば実現手段ということになる。

表1　ソフトウェア設計技法年表（オブジェクト指向以前）

1971	モジュール化技法（Parnas）、段階的詳細化法（Wirth） ワーニエ法（Warnier）、トップダウンプログラミング（Mils）
1972	構造化プログラミング（Dijkstra）
1973	複合設計（Myers）
1974	構造化設計（Constantine）、抽象データプログラミング法（Liskov）
1975	ジャクソン法（Jackson）
1977	システム階層分割（Ross）
1979	トップダウン設計法（Yourdon）、形式的仕様記述（Guttag）
1980	統一的設計方法論（紫合）
1981	機能的設計法（花田）、SP-FLOW（臼井）
1982	PAM（二村）
1983	データフロー設計法（Jackson）
1985	抽象データプログラミング向き設計法（久野）

こうしたモデルとしてのパラダイムと実現手段としての手法の関係は、しばしば経営学の文脈などでも用いられる、「戦術」と「戦略」に例えられることがある。戦略や戦術などの語は、言うまでもなく兵法用語、軍事用語であるが、経営学やソフトウェア工学、あるいは政界など、どちらかと言えばプラクティカルな分野における理論的な側面と実践的な側面を表すために用いられることが多い。

戦略（STRATEGY）とは、一般に「戦いに勝つための長期的な計略」と定義される[8]。いわば大局的な方針で、目的・目標・ゴール・夢・ロマン・方針、などと言った言葉で言い表されたりもする[9]。それに対して戦術（TACTICS）とは、戦略を達成するためのより局地的な個々の手段・方法・やり方などを意味する。戦略を「What」、戦術を「How」とすると、より把握がし易いであろう。システム開発においては、開発手法が戦術であり、その背後にある開発パラダイムがいわば戦略にあたる。

さて戦略と戦術に関連する指針として、しばしば「戦略なき戦術は

無意味、戦術なき戦略は空想」という指摘がなされる。これは、実現手段を持たない理論、あるいは理論を持たない手段の欠点を指摘するものであるが、その意味で言えば、システム開発手法はそれのみでは考察ができないということになる。但し同様な指針として「戦術は戦略に従属する」という指摘もある。これは、「戦略は戦術を支配し戦術は戦略に支配される」ということでもあり、手法がパラダイムに基づいて実現されるという指摘は、表1にも明らかである。しかし開発現場では往々にして、実現手段である戦術、即ち開発手法に主眼が移りがちであり、実際にオブジェクト指向開発においては、手法の優劣が長く議論されてきたという事実も否定できない。プラクティカルな側面を持つ領域では、実現手段自体が強調されるのはある意味しかたがないとは思われるが、問題はパラダイムを手法がどのように実現しているか、すなわち戦略と戦術の関係なのである。

開発手法に関しては、しばしば「手法はレシピなり」といった指摘がなされることがある。「レシピ（recipe)」とは、処方、あるいは秘訣という意味であるが、最近は料理の調理法という意味として使われることが多い。要するに、それに従えば「誰も」が「優れた」料理を作れるのが「レシピ」であり、開発手法は、それに従えば「誰も」が「優れた」ソフトを作れる、いわば「ソフトウェアのレシピ」であるという意味である。

レシピとは料理の専門家ではない素人が、ある一定レベルの料理を作り上げるための、いわば「失敗しない」料理を作るための手段であるという点に注目したい。レシピは、料理を均質化することにより、完成品の一定レベルを保証する。言い換えれば、決して失敗しないが飛びぬけた成功もしないという側面を持っている。レシピがあっても、料理の専門家は存在するのである。

その意味で言えば、ソフトウェアの開発手法に従えば、スキルの低い技術者の場合でも、ある一定のレベルの品質が保証されるため、確かにソフトウェアの品質を均質化する効果があるように思える。

一般的に見て技術の効果には、２つの方向性がある。一つは、その技術を採用することにより、旧来不可能であったより高度なことを実現するといった方向であり、いわゆる先端技術と呼ばれているものはそれに該当する。もう一つとしては、その技術により多くの開発者が平均的に優れた品質を実現することができるという方向である。前者を先端技術、あるいは高度化技術と呼ぶならば、後者はいわば平均化技術と言うべき位置付けである。一連のソフトウェア開発手法は、スキルの低い技術者に一定レベルを保証する効果があるという意味で、後者の平均化技術として機能していると考えられる。

2.「パラダイム」としてのオブジェクト指向

以降にはオブジェクト指向技術に関して、特に開発パラダイムとしての側面に関してまとめる。元来オブジェクト指向技術はその技術的な起源がいくつかあり、明確に定義を与えたりその意義を明らかにしたりするのが難しい。私見では、そういった点もオブジェクト指向技術が手法中心に議論されている一つの理由でもあると思われる。

一般には、①ソフトウェア工学、②DB（データベース）、③人工知能の3つの分野が、オブジェクト指向技術の起源とされている[2]。②の領域では、リレーショナルモデルの限界を補うものとしてオブジェクトモデルが位置付けられており、③では知識表現の一つとしてオブジェクト指向を捉えることが出来る。ここでは、システムの開発パラダイムとして、①の文脈で捉えることにする。前述のように、言語処理系Smalltalk-80の登場により現実化された、ソフトウェア工学としてのオブジェクト指向は、端的に言えば、「優れたソフトウェアを記述するための技術概念を体系的に統合化したもの」と定義することができる。そのため、C++やObjective-C、そしてJava

など、Smalltalk-80以降に新たに設計されたプログラミング言語は、こうした意味でのオブジェクト指向を言語仕様の中に取り込んでいる。

ここで言う、「優れたソフトウェア」とは、言うまでもなく、生産性、信頼性、そして機能性という工業製品に必要とされる品質を各々極大化したものである[3]。一般には、機能性、信頼性、使用性、効率性、保守性、移植性の、いわゆる「ベームの品質特性」がその指標とされる[10]が、[3]でも述べたように、ベームの品質特性には、生産性、経済性に当たる概念がなく、コストが考慮されておらず、他にも使用性や効率性は機能性に包含されるとも考えられる。

それらを記述するための「技術概念」とは、変数の保護、情報隠蔽、カプセル化、モジュール化など、ソフトウェア工学、特に構造化技術の文脈で徹底的に明らかにされているものである。その意味で言えば、オブジェクト指向技術自体は、決して先端技術あるいは新規技術ではなく、いわば旧来の技術の集大成といった側面を持っている。

2. 1 モジュール間インタフェースとオブジェクト間インタフェース

ここではその技術概念の例として、ソフトウェア記述においては最も重要な概念の一つであるモジュールとその記述に関して取り上げる。

モジュールとは元来「計測の単位」を意味するが、そこからソフトウェア工学では、特定の処理を一つの機能要素として独立させて扱うことができる要素を意味する。宇宙工学ではモジュールが、「母船から分離して作動できる宇宙船の一部」という意味で使われていることでわかるように、元来どちらかといえばハードウェア寄りの概念であるが、ソフトウェア工学では、特に大規模ソフトウェアの開発における分散開発を行うために、プログラム部品というニュアンスで導入されたものであると言えよう。特にいわゆる高級言

語にあたるプログラミング言語には、必ずモジュールに当たる概念が含まれているが、関数やサブルーチンなど、呼称は様々である。但しここで言うモジュールは、必ずしもサブルーチンや関数のみならず、それらを複数纏めた、いわばより上流寄りの論理単位という側面もある。

モジュールという概念において重要なのは、プログラムの機能単位がコンパイルやアセンブル等の処理単位であり、さらには開発者の作業や思考の単位でもあるということである。モジュールとその設計原則などが洗練されたのは、構造化技法と呼ばれる一連の開発技術においてであったが、それが開発現場において実効性を持っていたのは、プログラムの単位がそのまま開発の工程管理に用いることができるという、プラクティカルな側面があるのは見逃せない点である。そのため、システムの開発における進捗の管理には、ウォーターフォール型の開発モデルとともに、モジュール概念も大きく貢献をしていると言える。

こうした本来のモジュールの目的からすれば、モジュールは「自己完結的」でなければならないという設計原則が出てくる。特定のモジュールの設計やその仕様変更などが、他のモジュールやシステム全体に影響が及ばないように、即ちモジュール自体が局所性を持つ必要がある。

以上を前提に、モジュールの設計の善し悪しを判断する指標として、特に構造化技術において明確にされたものに、モジュール強度とモジュール結合度の２つがある。前者はモジュールの内部構造に関するものであり、後者はモジュール相互の関係に関するものである。前述のようなモジュールに対する自己完結性を実現するには、前者を強くそして後者を出来る限り弱いものとする必要がある。その意味で言えば、モジュール強度とモジュール結合度は、同じ概念を観点を変えて表現したものである。

ここでは、オブジェクト指向技術との関わりを考えるため、特に

モジュール結合度について考察する。モジュール結合度とは、端的に言えばモジュール相互で交換する情報の形式や量によるもので、その尺度としては、最も結合度の高い（即ち独立性が低く、設計として劣っている）「内部結合」から、「共通結合」、「外部結合」、「制御結合」、「スタンプ結合」、そして「データ結合」までが区別されている。この尺度から、例えば大域変数の弊害といった構造化技術で最も重要と言える問題意識が導き出されることになる。しかしながら、実際に開発現場においては、こうした設計原則に則り開発を行うのは、かなり困難である。端的に言えば、かなりスキルのあるエンジニアでなければ、なかなかこうした指標を特に実装のレベルで実現するのは難しい。これは、前述の様にモジュールが設計の論理単位であるのみならず、実装単位でもあるという点に起因する。設計工程で作られるシステムの論理モデルは、必ずしも実装モデルと一致するわけではないためである。

そのため構造化技術においては、モジュール間のインタフェースという観点から、優れた設計原則が明らかにされている。通常開発者は、このインタフェースに関する原則を意識して設計や実装を行うことが多い。それらインタフェース原則をまとめると、以下の4つになる。

① 最少モジュール間インタフェース
② 最小モジュール間インタフェース
③ 明示的モジュール間インタフェース
④ モジュール内部情報の隠蔽

①は「特定のモジュールが情報交換するモジュールは、出来る限り少ないものが望ましい」という意味で、インタフェースの数的な原則である。②は、「モジュール相互が交換する情報の量は、出来る限り少ないものが望ましい」という、量的な原則である。ここから前述の優れた設計である「データ結合」が実現できる。③は、モジュール間にインタフェースが存在することが、特にレキシカルに

(表記として) 明示化されていることを意味する。④は、モジュール相互で、各モジュール内の変数や制御構造などの実装情報が、インタフェースレベルに表れずに、それらを参照できない構造を持っていることを意味する。この原則から、「制御結合」が排除されることになる。

これら一連の設計原則に則った、独立性の高いモジュール間のインタフェースを、構造化技術においては「(モジュール間の) 正常参照 (Normal Reference)」と呼び、構造化技術における重要な表記法である「構造化チャート (Structured Chart)」を用いて図1の上に示す。尚、構造化チャートはモジュールの階層構造とモジュール間インタフェースを記述するもので、モジュールを上下の階層構造で捉えるため、階層構造図(Hierarchical Structured Chart)とも呼ばれる。

ここで示す図は、上位モジュールAが下位のモジュールBを参照している構造を示すが、モジュールAの中から出てモジュールBの枠に触れている「矢印 (→)」は、Aからのレキシカルな参照を意味する。参照に沿った丸と矢印の結合は、インタフェース上で交換されるデータを示す。「白丸+矢印(○→)」を「データ結合」と呼び、モジュール間で処理の対象となるデータそのものを示す。また「黒丸+矢印(●→)」を「フラグ結合(旗幟結合)」と呼び、他のモジュールを制御するためのデータ、すなわち制御関係があることを意味する。

逆にこうした設計原則から外れる悪いモジュール設計を「病的参照 (Pathological Reference)」と呼び、それを図2に示す。図2の①は、XがYの内部値Qを直接読み込むものであり、②はYがZの内部変数Rにデータを書き込むもの、そして③はZが制御をXに委ねる構造のものである。これらは、インタフェースを取る各モジュール相互の独立性を落とすことになり、構造化設計では排除されねばならないとされる。しかしここで指摘する例は、ある意味教科書的なも

のであり[11]、実際のシステム開発においては、より複雑なモジュール構造となり、その優劣を検証することは難しくなりがちである。そのため、設計原則を実現するには、開発者にそれ相応のスキルを必要とするのは否定できない。

非オブジェクト指向システムにおいてモジュールは関数などの機能要素であるが、オブジェクトはさらにその発想を進化させ、関数やルーチンなどの機能要素と内部変数を一体化し、外部への情報隠蔽機能を与えたものである。すなわちオブジェクト自体がモジュールであり、関数はオブジェクトのメソッドとしてモジュールの要素と捉えられることになる。オブジェクト指向型プログラミング言語におけるクラスがそのオブジェクトにあたるが、特にC++のクラスがC言語の構造体（struct）のメンバにスコープを与えて、拡張させたものと位置付けられている点でそれは明らかである。

オブジェクトは、①変数とその変数への参照関数（アクセスメソッド）を（論理的に）一体化させ、さらに②オブジェクトの内部の情報を外部から不可視にすることで、変数の参照や内部関数の呼び出しを外部に遮断したしたものと定義される。そのため、オブジェクトの内部関数の呼び出しは、オブジェクトの持つ唯一の外部インタフェースであるメッセージ通信に限定される。

図2の下側に、モジュールとしてのオブジェクト間におけるインタフェース、すなわちメッセージ通信の模式図を示す。前述のようにオブジェクト自体に外部への情報隠蔽機能があるため、オブジェクト相互では各相手オブジェクトの変数や内部関数は操作できない。すなわち構造化設計におけるモジュール設計原則のうちの情報隠蔽機能は、オブジェクトのメッセージ通信においては、本来的に実現されているものと考えられる。

さらにオブジェクト指向におけるメッセージ通信は、メッセージ式という形式で明示的に記述され、その意味においては、モジュール間の「明示的インタフェース」が実現される。さらにまたメッセ

ージ式のシンタックスは、「相手オブジェクト　メッセージ」という形式(注1)で、受信オブジェクトを指定する。この記述をメッセージ「通信」と呼ぶ点から明らかなように、シャノンの通信モデルを例として引用するまでも無く、そこには送信オブジェクトと受信オブジェクトの2つの主体によるコミュニケーションが前提とされている。これはインタフェースを取るオブジェクトが「最少」の数であると言う意味で、モジュールの「最少インタフェース」である。さらにまた、メッセージはオブジェクトのメソッド（内部関数）の呼び出しではなく、端的に言えばメソッドの呼び出しのトリガであって、実際に呼び出されるメソッドへのフェッチは受信オブジェクト自身にまかされる。そのためメッセージを送る側は、具体的な処理を意識せずに、ポリモフィックにメッセージを扱うことが出来る。これはオブジェクト間のインタフェースに含まれる情報の抽象度の高さと言う意味で、モジュール間の「最小インタフェース」と解釈することが出来る。

注1：例えばC++では、メッセージ式の記述は「クラス変数.メンバ関数の呼び出し」という形式を取る。これはC言語にいう構造体のメンバへのアクセスと同じ文法に則ったものであるが、変数がC++のクラスである場合、より抽象度の高いメッセージ式として解釈することができる。

ここで述べたオブジェクト間インタフェースの例で見るように、オブジェクト指向技術においては、構造化設計技術において明らかにされた設計原則が、明示的に技術要素として包含されている。前述のように、構造化技術に基づく様々な設計原理を実際のシステム開発において実現するには、それほど容易なことではなく、開発者のスキルに大きく左右される。しかしこれらの設計原理が明示的な技術要素として包含されているオブジェクト指向技術に元づいてシ

ステム設計を行うことで、逆に設計原理を意識せずに実現することが可能となる。オブジェクト指向秘術には、開発者のスキルの違いを埋めるという側面があるということがここからも明らかである。

以上から、オブジェクト指向技術に関する仮説として、設計結果の品質を高めている源泉は、オブジェクト指向の実現手段である開発手法ではなく、パラダイムとしてのオブジェクト指向技術自身にあると思わざるを得ないのではなかろうか。すなわち「手法はレシピなり」ではなく、オブジェクト指向技術自身がある種レシピ的な側面を持っているのである。以降には、そのパラダイムとしてのオブジェクト指向の持つ効果や意義について考察をし、その仮説について検証する。

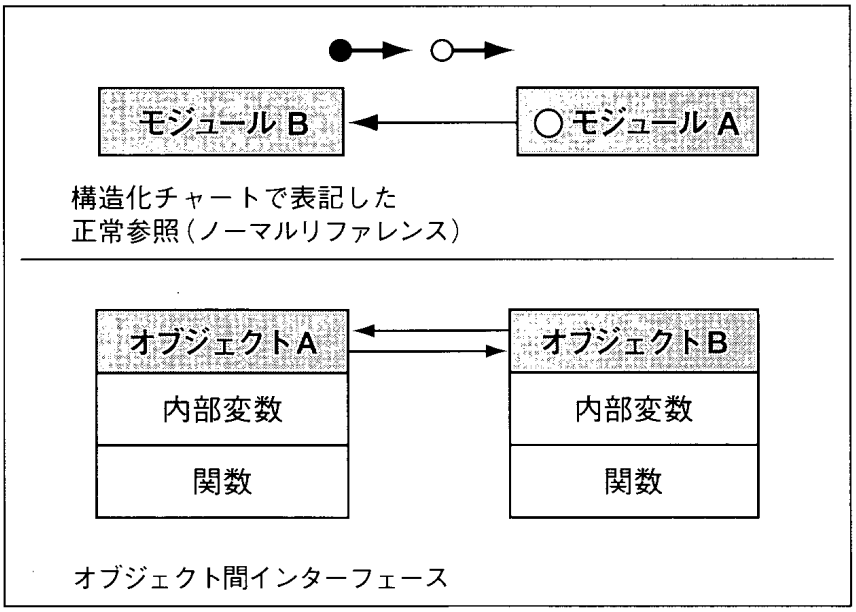

図1　モジュール間正常参照とオブジェクト間インタフェース

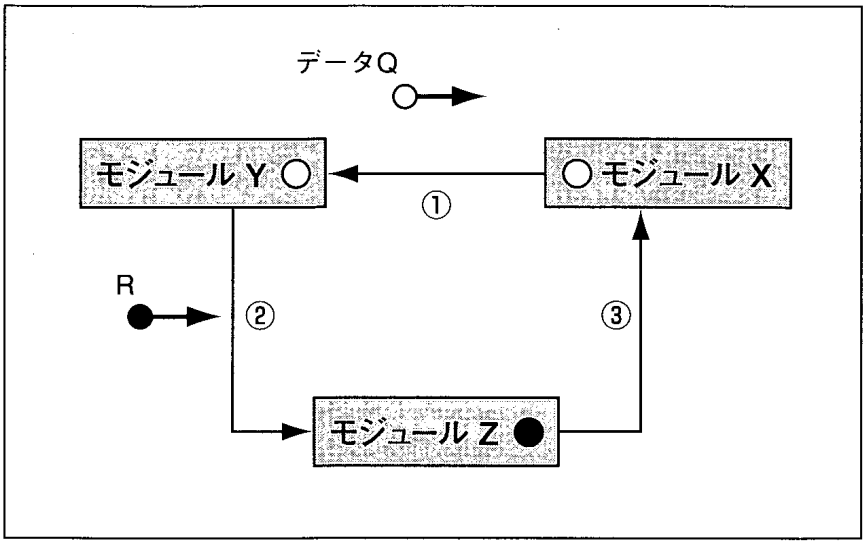

図２　モジュール間病的参照

2．2　オブジェクト指向技術の意義

[2]で述べたように、オブジェクト指向技術は「Object-Oriented technology」の訳であるが、他に関連する技術概念として「Object-Based」や「Class-Based」といったものもあり、原語では明確に区別されるが、それらに該当する訳語は存在せず、日本語では全てが「オブジェクト指向」とされてしまう。私見では、ここにオブジェクト指向技術が日本において誤解されている大きな理由があるように思えてならない。

[12]などでは「算体」という訳を充てられているオブジェクトは、前述の様に「情報隠蔽機能を持った変数と関数の集合体」であり、それは自律的な情報処理機能を持った計算要素（モジュール）である。そのオブジェクトをある種のデータ型として定義することができ、さらに階層化することによって既存の型（オブジェクト）定義をインポートすることができる、いわゆるクラス階層などの技術要素も、普通オブジェクト指向の中で考察されることが多い。特にクラス構造を洗練させて行く作業は、オブジェクト指向手法では中心的な作業として扱われている[13]。しかし[3]で述べたように、クラ

スの階層化は基本的に再利用を前提とした生産性効果を意図したものであり、モジュールとしてのオブジェクトとは若干意義が異なっている。Object-OrientedとObject-Basedが区別されている通り、オブジェクトによるモジュール化とクラスの階層化は別個に考察せねばならないだろう。

前述のように、オブジェクト指向技術は、ソフトウェア工学的に見れば、構造化技術の集大成という側面がある。その意味で言えば、オブジェクト指向技術にはソフトウェア工学的な新規性があるわけではない。結論から言えば、オブジェクト指向の新規性は、「オブジェクト」そのものを定義することにある。

ウォーターフォール型の開発モデルを前提とすると、まず上流工程のシステム開発者は、ユーザとのコミュニケーションなどをもとに要求仕様を作成する。この工程をシステム分析（Analysis）と呼ぶが、ここではBPM（Business Process Modeling）(注2)などの手法が用いられる。ここで明らかになった対象となる問題空間及び問題解決行為をもとに、さらに下位の設計（Design）工程では、それらをモデル化して定式化して設計仕様を作成する。さらにその仕様をもとに下流工程では、作成されたモデルをプログラミング言語記述に変換していく。すなわち各工程の成果として、仕様やプログラムなどが作成されることになる。

表2に工程とモデルをまとめるが、システム分析では、現実の世界である対象領域や要求を、特に情報やその流れという観点からモデル化するものであり、さらにシステム設計ではそのモデルを特にシステムによる問題解決という観点でモデル化をする。そしてプログラミングでは、プログラミング言語の持つモデルに基づいて、実際の言語記述に変換していくことになる。

工程ごとに作成されるそれらの仕様は、作業の結果として作成されたモデルの記述、表現であり、各工程の作業は、いわばモデルの変換を行っていくことである。モデルの作成作業であるシステム分

析工程も、捉え方によっては、現実世界に存在している業務を、定式化して言語や図式などを用いて表現することであり、その意味ではモデルの変換作業と考えることもできる。

表2　工程とモデル

工程	モデルの変換	作業の観点
システム分析	対象領域・要求→要求仕様	業務の実態・情報の流れ
システム設計	→設計仕様	システムによる実現
製造(プログラミング)	→プログラミング言語	言語モデル

注2：BPMとはビジネス・プロセス・モデル（Business Process Model）あるいはモデリングのことで、対象となる業務を特に作業や情報の「流れ」という観点から記述する。例えばBPMを明示的に包含する手法のDATARUNでは、①ビジネス・プロセスの理解、②基本データ発生源（Primary Data Generator）の発見、③システム化対象範囲の決定、④解決すべき問題点の発見、を目的に作業が行われるとされる[14]。

しかしながら、注意したいのは各工程で作られるモデルは、各々作業の観点が異なった、いわば原理が違うモデルであるということである。旧来の（非オブジェクト指向型の）システム開発においては、最終的に実装として記述されるのは手続き型のプログラムモデルである。ここで言うプログラムモデルとは、プログラムを記述するための基本的な原理のことであり、例えばAlgol系の手続き型プログラミング言語では、現在のノイマン型コンピュータ上で稼動する、プログラム内蔵方式、メモリとファイルの二重記憶システム、単一のCPUによる逐次処理などと言った、コンピュータのハードウェア構造を反映したアルゴリズム記述となる。しかしシステム化の

対象となる現実の業務は、組織構造を元に帳票や伝票など様々な形態の情報が流れているものであり、そこには逐次的なアルゴリズムやファイルなどは明示的には存在していない。そのため、こうしたモデルの変換作業である工程が多くなるだけ、作られるモデルの乖離は大きなものとなり、システムが本来の期待される効果を上げることができなくなる。そうした問題意識から、オブジェクト指向技術を捉えることができる。

前述の様に、オブジェクトは「変数と関数の集合体」であるが、この定義はプログラムモデルとしてのものである。より上流のレベルでは、オブジェクトとはその名前の通り、「モノ」そのものである。現実の業務においては、帳票や伝票はモノ、すなわち他のモノとは区別できる存在である。オブジェクト指向は、上流工程のモデルに含まれるこの「モノ」を、下流工程で「変数と関数の集合体」として実装まで展開することを意図するものである。

変数と関数の集合体とは、自律的な情報処理機能を意味するため、しばしば「モノが自分の行うことを知っている」といったメタファーで、オブジェクト指向モデルが説明されることがある。例えば、図3に示すのは、在庫管理業務をモノの観点でオブジェクトモデル化したもので、[6]で取り上げられているものを[15]から引用する。ここでは後述するオブジェクトモデルの表記法であるUML (Uniformed Modeling Language) が用いられているが、例えば「商品」というモノ（オブジェクト）には「入庫依頼をする」という内部手続きが含まれている。「商品が自己の入庫を依頼する」という自律型のモデルであり、在庫管理業務に含まれる様々なオブジェクト群が各々自律的に処理を行うという分散型の情報処理システムが記述されることになる。こうした側面を、「自然なモデル」と呼ぶことがある[6]。確かに現実世界に存在しているモノに着目するほうが、アルゴリズム的に把握するよりもわかりやすいとは言えるだろう。

このようにオブジェクト指向においては、モノとしてシステムを記述し実装するといった、旧来の手続き型とは視点の異なったモデル記述がなされるが、その最大の利点は対象領域や問題が可視化することにある。図3に示す在庫管理のモデルの図式表現は、確かにデータ構造や処理の流れで見るより、またテキストで仕様記述をするよりも、明らかに理解度が高いと言えるであろう。例えばこれを工程に即して考えれば、上流から下流まで、ユーザから設計者、プログラマまで、開発に関わる全ての人間がモノという観点のモデルを共有することが出来ると言われている[6]。すなわちモデルの変換の少なさがその利点とされていると解釈できるが、実際のシステム開発においては、ことはそれほど単純なものではない。しかしそうした問題意識は、近年のシステム環境の変化や分散化においては重要なものとなっている。例えば公的な資格ともなっているシステムアドミニストレータが、システム開発にユーザ側として関与するといった開発形態も、そういった問題意識の表れと考えることもできるであろう。

前述のように、対象領域には含まれていないプログラミングモデルを作り出すための一連の作業がシステム開発ということになるが、それらの作業はある種高度な知的作業であり、開発者のスキルの差が表面化するのは否定できない。オブジェクト指向技術は、そこに「モノ＝オブジェクト」というモデルを与えることにより、仕様を可視化させるという機能がある。「モノが自分の出来ることを知っている」という、非常にわかりやすいメタファーを使って、高度なスキルを要するシステム開発作業を設計者に自然に実現させるという、ある意味巧妙な開発技術であると言える。要するに、オブジェクト指向技術のこれらの側面は、高度な仕様を作り出すためのものと言うよりも、開発者のスキルの違いを埋めようとするものである。

その意味から言えばオブジェクト指向技術は、技術者のスキルの

差を埋めるという意味で明らかに平均化技術であり、そこに設計結果の品質を高める源泉があると言える。前述の様に、開発手法自体が平均化技術の側面を持っているが、ことオブジェクト指向に関しては、パラダイム自体がそういう側面を持っていることに注意をしたい。すなわちオブジェクト指向技術は「飛びぬけた」仕様を作成するためのものではなく、「平均的な」あるいは「失敗の無い」仕様を作成するもので、どちらかと言えば余り高いスキルを持たない技術者にとって有効な技術であると結論付けることが出来る。以降には、その平均化技術としてのオブジェクト指向が持つ限界について、開発事例、経験などをもとに検証を行うことにする。

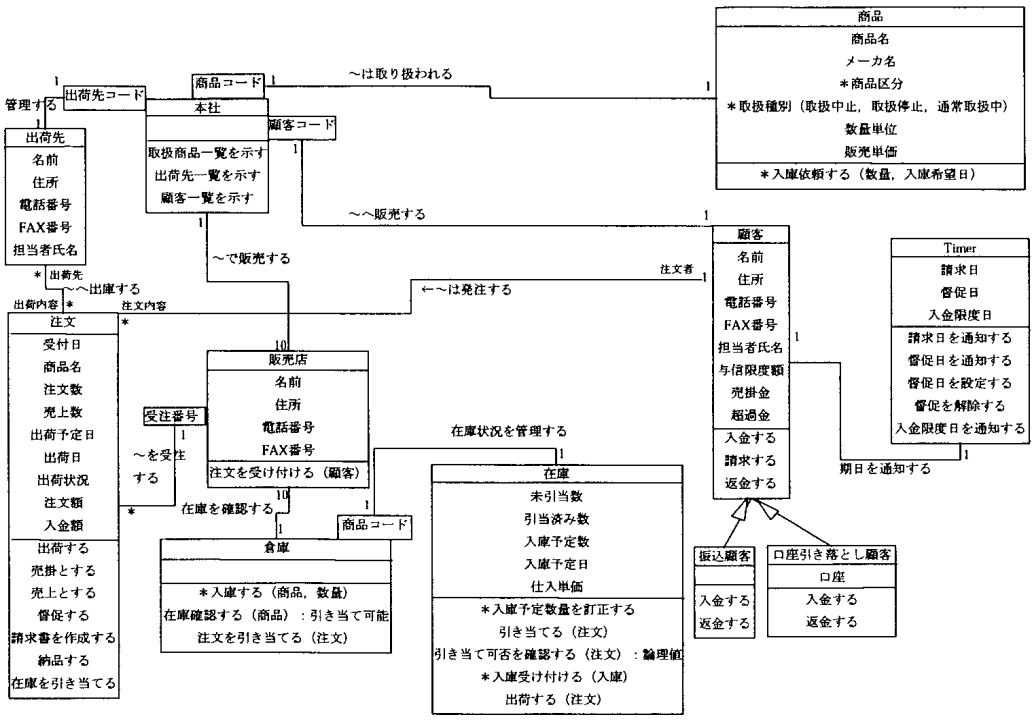

図３　在庫管理問題のオブジェクトモデル例

3．問題の構造とシステム開発技術

3．1　モデルの誤り

前述の「銀の弾は存在しない（No silver bullet）」という言葉を挙げるまでもなく、技術には必ず限界が存在する。その限界を補完するものとしてまた新規技術が生まれていき、それがひいては現代の産業資本主義の原動力となっている[16]。その意味から、技術の限界を認識することは決してネガティブなことではなく、むしろ成功事例よりも失敗事例によって多くの知見が得られることが多いのは否定できない。

オブジェクト指向技術に関しても多くの開発事例が蓄積されてきており、また筆者もいくつかの開発経験をすることができた。その中で、その一連のオブジェクト指向技術が額面通り機能していないシステム、すなわち失敗事例も散見できるようになってきた。しかしながら、オブジェクト指向技術の限界に関する議論はあまり見られない。失敗事例は技術的な限界というよりは、モデル化の間違いという形で、開発方法論の問題として吸収されてしまうことが多々ある[6]。

図３に示した在庫管理システムなどは、オブジェクト指向に関するいわゆる共通問題として、さまざまな解説で使われる問題である。ビデオの貸し出しや酒屋や米屋の在庫など、対象を変えてさまざまに取り上げられている[15]。その意味では、これは成功事例と言っていいだろう。

前述の様に、オブジェクト指向システムにおいては、「モノが自分のやることを知っている」というメタファーによって、自律的な情報処理要素を作り、分散型システムを構築していくことになる。しかし全てのシステムが分散型アーキテクチャで実現されているわけではなく、「モノが自分のやることを知っている」メタファーによるモデルが不自然なものは確かに存在する。

ここでは分散型システムの代表として、LAN（Local Area

Network・局所通信網）そのものについて考えてみる。LANとは言うまでもなく、IEEEの定義では「多くの独立した装置が、適度なデータ伝送速度の物理的伝送路を通じて、適当な距離内で直接通信可能なデータ通信システム」のことである。「適度な速度」や「適当な距離」の範囲が明確にはされていないので、いわゆるWAN（Wide Area Network・広域通信網）との区別は管理主体の存在という側面で捉えることが多い。LAN自体、元来が分散システムであるため、オブジェクト指向によるモデル化が有効な典型領域のように思える。

通常LANシステムは、いわゆる「クライアント・サーバモデル」によって構成される。クライアント・サーバモデルとは、LANなどの分散型システムの構成法の一つであり、システムを、処理の中核を実現する「サーバ」と、そのサーバが提供するサービスを利用する「クライアント」に分けて構成する。ホストコンピュータを中心としたホストシステムなど集中型システムとの最大の違いは、システムの機能が提供されるトリガが、ユーザの要求によるか否かといった点にある。クライアント・サーバモデルでは、その名前の通りネットワークユーザであるクライアント（client・顧客）が、サーバ（server・召使）に機能を要求することが無ければ、サーバ側は何も処理を行わない。そのモデルの模式図を図4に示すが、ここで示すようにクライアント側が主体になって機能を呼び出すことになる。クライアントとサーバ間のコミュニケーションは、抽象度の高いメッセージ通信であり、サーバ側の挙動はクライアント側には隠蔽されている、すなわちこのクライアント・サーバモデルは、優れたオブジェクト指向モデルであると言える。

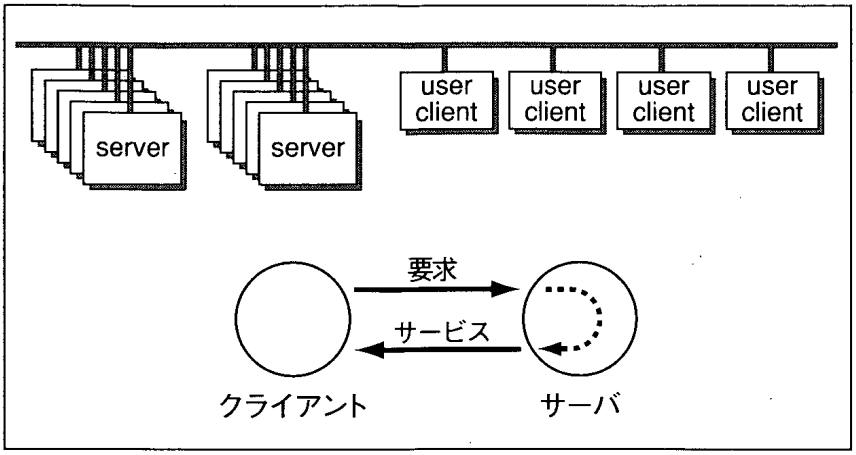

図4　クライアント・サーバモデルの模式図

FTPやWWWなど、Internetを支える様々なアプリケーションは、このサーバ・クライアントモデルを前提としたものであり、情報技術の世界では広く認知されているモデルである。しかしLANの構成方式としては、さらに「ピア・ツー・ピアモデル（Peer to Peer)」も存在する。これは、端的に言えば全てのネットワーク端末を対等なものと考えて接続する方式であり、近年サーバを使わないインターネット上のファイル交換ソフトGnutellaやNapstarなどで採用されている。

多くの端末から構成される分散型ネットワークシステムを、モノに着目しモデル化するとなると、おそらくはこのピア・ツー・ピアモデルになるであろう。すなわちこれは、「自然なモデル」である。逆にクライアント・サーバモデルは、前述のようにLANの機能を元にオブジェクト化したものであり、本来問題空間に存在していないモノがオブジェクト化されている。すなわち問題空間が操作されており、その意味においては「自然なモデル」ではない。しかしピア・ツー・ピアモデルには、各端末にクライアント機能及びサーバ機能が同居するため、処理速度やパフォーマンスの低下などが起こりがちであり、さらにユーザの増加によって管理が困難になるとい

う問題を含んでいることも指摘されている[17]。これは、各端末がモデルとしてモジュール性を欠いているという点に起因する。その意味から言えば、ピア・ツー・ピアモデルはオブジェクト指向モデルとしては、クライアント・サーバモデルに劣る設計だと言わざるを得ない(注3)。

注3：ここでの設計が劣るという指摘は、あくまでもモジュールとしてのオブジェクトモデルという観点で見た場合の評価であり、ネットワーク技術としての観点とは別である。クライアント・サーバモデルには、導入や運用にまつわるコストの問題などの欠点も指摘される[17]。

ここで指摘したいのは、クライアント・サーバモデルとピア・ツー・ピアモデルは、本質的にモデル構造自体が違っているという点である。前述の様にクライアント・サーバモデルは、機能を元にしたモデルであるが、それはモノとしての各端末をモデル化したピア・ツー・ピアモデルに対して、クラス構造や階層構造の単純な洗練を行っても、クライアント・サーバモデルが創り出されるのでは無いということを意味する。

以上から、モノによるオブジェクト指向モデルが有効なものとそうでは無いものには、モデル化の誤りという工程的な問題ではなく、問題の構造自体に違いがあるということが明らかになる。

3．2　2つの問題類型

システムの開発とは、開発者が設計問題に対して何らかの解を与えるものであるが、さらにここで注意したいのは、特に開発の対象が情報システムの場合、その情報システム自体が特定の問題に対して何らかの解を与えるものであるということである。往々にしてこれらが混同されている例があるが、ここでは便宜上前者を「設計問

題の解決」、後者を「対象問題の解決」として区別し、それを図5に示す。開発者は「対象問題の解決」手段を明らかにすることにより「設計問題の解決」を行っているということである。

例えば在庫管理システムの場合、開発されるシステムは在庫管理という対象問題に対して解を与える。図3はそのシステムを設計するという設計問題に対して、オブジェクトという観点で一つの解を与えたものである。同様にLANシステムでは、コンピュータの端末同士でコミュニケーションを行うという対象問題に対して、例えばクライアント・サーバという設計の解を与えたものということになる。端的に言えば、システム開発者が設計問題の解決をする際にモノを元にしたモデル、すなわちオブジェクトを用いるものがオブジェクト指向技術ということになる。

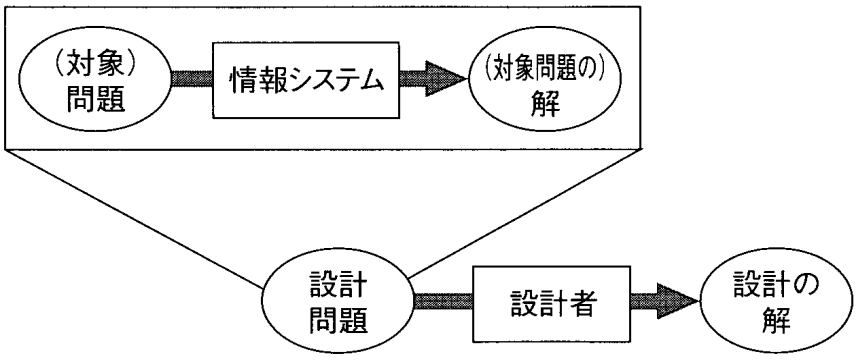

図5「設計問題の解決」と「対象問題の解決」

こうした2つの問題解決という観点で見ると、設計問題には①モノの記述で問題への解を与えることができるレベルのものと、②モノの記述だけでは不十分であり、さらに明示的に問題解決自体のモデルが必要なレベルのものの、2つの類型があることが明らかになってくる。ここでは仮に前者①を「タイプ1」の問題、後者②を「タイプ2」の問題と分類する。

「タイプ１」の問題は、前述のいわゆる共通問題のように、問題空間（対象問題）に問題解決が明示的に包含されているとみなすことができるものである。「商品が自己の入庫を依頼する」といったような、「モノが自分の行うことを知っている」というメタファーは、「入庫」という問題解決が「商品」というモノに包含されている。こうしたタイプの問題の場合、対象問題を表現することで、設計問題を解くことができる。その意味では、確かにモノをベースとして設計するオブジェクト指向技術は有効である。

「タイプ２」の問題とは、このLANのモデル例などのように、モノによる単なる問題空間の記述だけでは、仕様に何らかの不具合や欠点が生じてしまうため、優れた仕様を得るためにはさらにモノのみではなく、「対象問題の解決のモデル」を記述しなければならないものである。

例えば前述の「クライアント・サーバモデル」は、1970年代のイーサネットを中心としたネットワークシステムの開発を起源として作られたモデルだが、LANという対象問題に対して、機能を元にしたクライアント、サーバという非対称なモデルによって、設計の解を記述する。これは、開発者の極めて高度な創造的行為の産物である。[6,15]などでは、そのレベルでのクラスの作成を「導出」あるいは「発見」と表現しているが、むしろそれは「創発」とも言うべき創造作業であって、決して単純な定型的作業の結果ではない。

このタイプの問題は、要するに「モノが自分のやることを知っている」というメタファーが不自然なものであり、問題の持つ特徴としては、以下のようなものが指摘できる。

●問題空間が巨大な場合。

これはいわゆる「大規模問題」であり、問題中に全体情報が存在せず、そのため全体機能を司るオブジェクトを創り出すことが不可能なものである。例えばInternetなどはまさにその典型例であるが、

その場合各オブジェクト自身に全体に対する調整的な機能を与えることは、設計上不可能である。Internetで言えば、各接続ノード、接続サイトが、巨大なInternet上における「自分の位置を知っている」という、メタファーとしても非常に不自然な設計になってしまうであろう。

●含まれる要素が受動的なもの。

各オブジェクトが、単純な情報のみを保持する、いわゆる「コンテナ」型のデータである場合、それらに対する操作は、外部から与えられると考えるのが自然である。コンテナ型データは、本来抽象的なデータ操作のみが与えられているものであり、それのみで十分機能を果たすものである。アプリケーション上の実データにおいても、機能の実態がそれらの抽象データに集約されてしまう場合、設計としてモノ以外にそれらに対する操作を行うオブジェクトを「創造」せざるを得ないであろう。

●要素に対する操作が複合的で、様々な観点、側面を持つもの。

例えばエンジニアリング系システムにおいて、設計の対象に対してさらに評価や診断を行うなど、各オブジェクトに対して、様々なアプリケーション操作が行われる場合などがそれに該当する。この場合、個々のオブジェクトに対してアプリケーション機能を持たせると、オブジェクト自体が大きなシステムになってしまい、モジュール性が疎外されることになる。さらにそれらに対して単独のオブジェクトとして扱ったり組織化して扱うなど、様々な観点で操作される場合、その観点の違いによって、様々なオブジェクトが現れてしまうなど、モノによるモデル化がし難い場合なども考えられる。

ここに挙げた特徴が全てでは無いが、タイプ２の問題には、こうした特徴が見られるものが含まれる。これらに対しては、単純なモ

ノによるモデル化を行ったとしても、モジュール性を疎外したり、要求する仕様が実現できないなど、オブジェクト指向技術が本来目指すべき効果が上がらないことが往々にしてある。

結局成功事例として上げられているものは、タイプ１の範疇の問題にしか過ぎない。但しその成功事例の評価に対しては、問題の類型としての側面に対して自覚的であるようには思えない。端的に言えば、そこにオブジェクト指向手法の限界があるように思われる。以降には、こうした問題類型に対するモデルの違いに関して考察する。

３．３　問題の記述と問題の解決

元来システムの仕様を作る作業は、その本質はモデルの変換とは言え、開発者の創造的な作業である。特にソフトウェアは労働集約的な側面を持っているため、往々にして各開発者のスキルの違いが仕様に反映することになる。しかし工業製品としてソフトウェアを捉えた場合、こうした開発者の個性により仕様が相違するのは致命的である。そのためソフトウェア技術は、オブジェクト指向も含め、いかに仕様を平均化するか、すなわち職人芸を排除し工業製品化していくかがその目標として含まれている。前述の様に、構造化技術では「優れた」ソフトウェアの特徴を明確化することにより、開発者が自覚的に仕様の精度を上げることを意図したものである。

オブジェクト指向手法では、様々な表記法が導入されているが、結論的に言えばこの表記法が仕様の平均化に寄与している。現在標準表記法とされているUML(Unified Modeling Language)では、システムの構造や振る舞いなどを表す9～10種類のダイアグラムが、必要に応じ組み合わせて使われる。UMLは1980年代後半から提唱されてきたOMT法、Booch法、OOSE法など様々なオブジェクト指向手法を、標準的な設計手法として統合したもので、1997年に米Rational Software など11社がオブジェクト指向技術に関する標準化

団体OMG（Object Management Group）に提案し、1997年11月に承認されている。前述の様に、現在のオブジェクト指向手法と呼ばれているものは、このUMLで定められているダイアグラムで記述することがその主要な内容である。実際OMGでの標準化過程においては、Rational Softwareから方法論の統一が提案されたが、最終的には表記法のみ統一した形で標準化されている [18]。すなわちUMLはあくまでも表記法であって、方法論そのものではないことに注意しなければならない。

前述の様に、仕様の作成は創造的な行為であるが、そこに標準化された表記法を与えることによって、創造作業を図の作成といういわば定型的な作業によって実現するという効果がある。例えば図6に示すのは、ダイアグラム中で設計の最も根幹となる静的構造図（クラス図）の例である。クラス図は、クラスの内部構造とクラス間の関係を記述する。図6では、クラスAとクラスB各々の内部構造とその関係が記述されている。例えばクラスBはその内部構造として、2つの内部変数（v1, v2）と、2つの内部メソッド（f1( ), f2( )）を持つことが示されている。さらにこのモデルでは、特にクラス相互の関係として以下のような設計上の制約条件を持つことが表現されている。

- Aは必ずBに属する
- Aは1つのBにしか属せない
- Bには0からNまでのAが属する

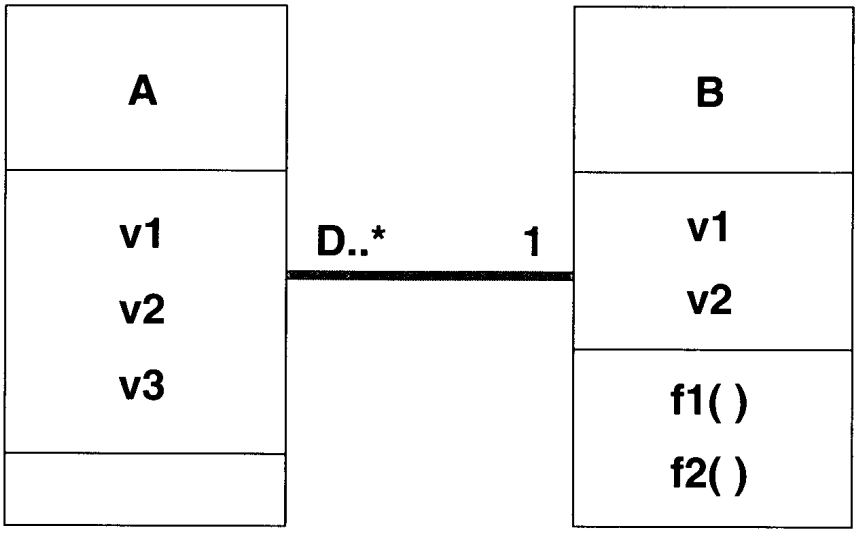

図6　UMLによるクラス図

この例で示すように、ダイアグラムに用意されている様々な表記を用いることによって、少なくとも設計者は内部変数とメソッドという2つの側面からモデルを記述することができるし、さらにデータ構造（すなわちオブジェクト）相互の様々な制約の見落としを回避することができるという効果があるのは容易に理解が出来る。

こうした問題空間に含まれる実体を、変数と操作関数という観点で記述するのは、システムの開発者として極めて重要且つ基本的な作業であり、さらにそれらをモデル化したデータ構造相互に含まれる制約の検証をすることも、特にデータベース設計作業においては重要な作業である。それらがこの表記法に明示的に包含されているということは、UMLにおいては特定の表記法に従って記述することが、すなわち設計作業であると想定していると考えることが出来る。

「モデリング（モデル化）」とは、広く設計作業において使われている概念であり、エンジニアリングの領域ではほぼ無定義で使われている。端的に言えば、特定の対象が持つ現象や機能を理解するために、抽象化して表現する作業を一般にそう称している[2]。「抽象化」が中心的な作業であるため、しばしば微分方程式のような抽

象度の高い表現形式が用いられ、表現したもの自体（この場合は微分方程式自体）をモデルと呼ぶ。ソフトウェアエンジニアリング、特にオブジェクト指向手法の領域では、対象システムを特定して文書化する作業が広くモデリングと言われている[6,15]。すなわち、UMLを例とすればそこに含まれている各種のダイアグラム自体がモデルにあたるということになる。

ここで注意したいのは、モデリング作業には、①「抽象化」と②「表現」という２つの知的作業が含まれているということである。前者は開発者の知的作業であり、それは開発者の能力や知識が反映する。少なくとも、技術者の誰でもが機能の観点からLANを抽象化してクライアント･サーバモデルを作り上げることが出来るようには思えない。後者は、実際にその知的作業の結果を現実化していく作業であり、例えばダイアグラムなどがその手段である。これは我々の「知識」と「言語」という関係にも対比することができるが、その例えで考えるまでもなく、抽象化が適切に行われていなければ、表現することはできない。設計において重要な役割を果たすのはあくまでも開発者による抽象化作業であって、表現そのものではない。

以上の前提で前述した問題の類型について考えてみると、タイプ１の問題はモノ自体が明示的であって、開発者の誰もがスキルの有無に関わらず同じモノを把握することはできるため、問題空間の記述をそのまま設計作業とみなすことができる。すなわち表現作業のみによって、モデル化が行える問題である。

問題は、タイプ２の場合である。この分類の設計問題においては、前述のように問題空間には明示的に解を導き出すための要素が含まれていないため、設計者が解となるモデルを創造しなければならない。即ち表現作業のみではモデル化が行えないものである。

UMLを含め、オブジェクト指向手法では開発者の創造行為に対して明示的に言及をしていない。クラスの洗練や発見といった言葉で表されている作業がそれに当たると思われるが、その内容自体も

極めてあいまいである。クラスの抽出に関しては、例えばUMLのユースケース記述中の名詞や名詞句に着目してオブジェクトの候補とするとか[19]、概念モデルに現れたアクタを候補とする[20]といった観点でクラスの発見が行われるとされている。またそれらクラスの洗練に関しては、ユースケースをもとに作成したシーケンス図(Sequence Diagram) によってシナリオを描き検討するとか[21]、一つのクラスに抽象度の高い言葉を与え、3つを超える場合は、クラスの分割を検討する[22]といった提案がなされている。

しかしこれらは比較的主観性が強く、その意味では開発者依存であるということは否定できない。特にこれら手法では、作業がクラス単位で考察されているが、問題は単独のクラス設計を行うことではなく、複数のクラスを構築することによって、最終的に全体システムを設計することにある。即ち個々のクラスではなく、クラス構造を作り上げねばならない。例えばクライアント·サーバモデルにおいては、クライアントのみでは意味が無いのである。

またこれら一連のクラスの抽出、洗練作業では、基本的に各ダイアグラムで詳細化していくことによりなされるとされている。すなわちクラスの洗練作業はクラス構造の変形として捉えられている。その意味で言えば、ある設計とそれに対して洗練作業を行った結果の設計には、ある種の抽象関係があるはずである。特に洗練作業が詳細化をベースにしたものでもあるとするならば、洗練前後の設計自体もクラス階層によって記述できねばならないということになる。

ここで再度LANの例で考えると、ネットワークで結ばれた端末をクラス図で記述し、それらを他のダイアグラムで洗練していくことで、果たしてクライアント・サーバモデルが作られるであろうか。

このことに関しては別稿で検証するが、筆者にはピア・ツー・ピアとクライアント・サーバの距離がとてつもなく遠く思える。どう設計を洗練させたとしても、両者が同じ設計の結果として生まれて

くるようには思えない。モデル構造に関して類似性があるようにも思えないし、ピア·ツー·ピアとクライアント・サーバを同じレベルでモデル化することは無理があるように思える。つまりクライアント・サーバモデルを作り出すのは、クラスの洗練作業などではなく、設計者がその問題に対して解を創造した結果なのである。

以上から明らかなように、問題の記述と問題に対する解を与える作業は本来全く別個であって、問題空間が記述できたとしてもそれがそのままその問題に対する設計問題の解であるということではない。但しタイプ１のような問題においては、例外的に問題の記述が設計問題の解となるということであり、極言すれば、たまたま設計が上手く行ったタイプ１に属する問題が、オブジェクト指向手法の成功事例として報告されていると考えられる。タイプ２の問題に対してモノベースで記述をしたとしても、適切な設計という解が与えられないのは、表現手法の問題ではなく、モデル自体の問題なのである。

４．有効な開発手法とは

４．１　手法と開発者の問題

以上のように、設計者にとって問題を記述することと問題を解決することは、明らかに違う作業である。しかしUMLの役割は仕様の正しい記述を保証することであって、仕様の創造自体は開発者の能力に依存する。タイプ１の問題に対しては、仕様の記述と仕様の創造を同じ行為とみなすことが可能であったため、仕様の記述手段を与えることが有効であった。しかしタイプ２の問題に対しては、仕様の記述と仕様の創造は別であるため、記述手段の他に設計を現実化していくための手法を与える必要があるであろう。

タイプ２の問題に対して開発手法が有効性を持つためには、設計者の創造的な作業がどうしても介在せざるを得ないため、まず手法自体が「開発者のスキルの違い」を前提とするものでなければなら

ない。ここで言うスキルとは、その手法がベースとするパラダイムに基づいたモデル化能力を意味する。またモデル化能力は、前述した意味での「表現」ではなく「抽象化」という側面を意味する。

前述の様に、表記法を中心とした開発手法は、各開発者の個性が顕在化しないように仕様を平均化させる。しかし「戦術は戦略に従属する」という言葉を待つまでも無く、実現手段である開発手法は、その前提となる開発パラダイムとは切り離すことができない。すなわち開発者のモデル化能力を設計に反映させるために、開発者に対して、少なくとも以下の2点が提供されねばならない。

- 開発者のスキルアップを吸収できる
- 「解」を強要しない

前者は、仕様の平均化ではなくより高度な仕様を作成する余地を保証する。オブジェクト指向パラダイム自体が平均化技術であるがゆえ、ことさら手法においては平均化を意識する必要は無いと思われる。また後者は、オブジェクト指向設計においては絶対的な解は存在しないということを意味する。同じ対象問題に対する設計の解は唯一ではなく、また同じ対象に対して異なったシステムを構築するならば、クラス構造が異なっているはずである。すなわち絶対的なモデル構造は存在しないという前提で考えられねばならない。

4．2　問題解決能力の問題

では、最終的なシステムの成否を左右する開発者のモデル化能力、特に創造的な問題解決能力を開発作業に反映させるには、開発手法中に何が必要であろうか。前述の様にUMLでは、開発時の成果物である分析、設計モデルの表記法のみが標準化されているため、開発工程自体をユースケースベースのものとしていくというRUP（Rational Unified Process・ラショナル統一プロセス）や、より柔軟な開発プロセスを指向するXP（eXtreme Programming・エクストリームプログラミング）などが、UMLをベースに提起されている。

また方法論においても、DATARUNのようにオブジェクト指向技術に留まらず、「構造化分析・構造化設計」「データベース理論」「データ中心アプローチ」の側面を併せ持った柔軟なものもあり、各々が注目に値する。

端的に言えば、工程や作業、あるいは表記法などは、それぞれ一長一短があり、それぞれが最終的に「優れた」ソフトウェアを作るという目的で一致しているとするならば、それらがどういうものでもかまわないはずである。繰り返しになるが、本稿で指摘したい問題点は、タイプ2の問題に対してアプローチするために、開発者の創造行為、すなわち問題解決行為を、システム開発にどう吸収し現実化していくかという点に集約される。端的に言えば、どうすればLANにおけるクライアント・サーバモデルのような創造的なモデルを、工業製品として作り上げていくことができるのであろうか。

工業技術における創造性は、芸術や人文の分野とは異なり、必ずある制約の元で発揮される。工業製品には必ず仕様があり、その仕様の実現を目標としなければならない。また特に情報システムの開発においては、前述の様にモデルの変換作業がその実体である。その意味で言えば、工業技術においては無から何かを作り上げると言った純粋な創造行為は存在せず、多かれ少なかれ創造行為は、いわゆる類似設計（Pattern Design）であることがその特徴である。

ここで言う類似設計とは、設計者が新たな要求に対する設計を行う場合において全く白紙の状態から新たな設計解を創り出すのではなく、過去に実現された設計事例などを利用し、それに対する一部の修正によって設計解を導き出す設計法を言う[23]。一般に類似設計的なアプローチは、過去に実現された設計事例が新たな設計に強く影響を及ぼすため、制約的にはたらくことが多いと言われている。しかし、過去の設計事例は、設計情報の再利用という観点だけでなく、新たなアイデアの糧としても有用である[24]。よって、開発者の創造性を類似設計によって吸収するアプローチは、有効なものと

思われる。

旧来情報システム分野における類似設計は、プログラムや設計の蓄積を元に、特に再利用という手法によって行われていた。再利用部品をライブラリと呼び、関数やモジュールを単位とするものや、オブジェクト指向システムにおいてはクラスを単位とするものがある。さらに単独のクラスのみならず、クラスの構造や設計知識を蓄積したデザインパターンやフレームワークなどが提唱されている。特に上流工程においては、デザインパターンやフレームワークなどの蓄積に対して、再利用度を高めることによって、類似設計を実現するアプローチが行われている。

デザインパターンとは、種々の状況における設計上の一般的な問題解決に適用できるように，オブジェクトやクラス間の通信を記述したのものと定義される[25]。またフレームワークとは、ある特定のソフトウェアを対象にした再利用可能な設計成果物を構成するクラスの集合である[26]。フレームワークは対象領域に特化したものがほとんどであり、パターンよりも抽象度は低く、いくつかのパターンから構成されていることが多いと言われている。これらを利用することによって、開発者は既に解が与えられている設計問題を流用して他の設計問題に対する解を導き出すことができるであろう。このように、特にデザインパターンにおける問題解決とは、分析や設計の結果としてのモデルを対象としている。その意味で言えば、デザインパターンを用いた類似設計は、特に設計の洗練作業において有効であろうと思われる。

ここで再度開発者の創造性について考えてみると、前述の様に設計者の創造作業は設計の洗練ではないため、必ずしもデザインパターンの蓄積が、タイプ２の問題解決に対して効果を持つようには思えない。つまり設計の流用というレベルの類似設計では、クラスの洗練においてしか効果を発揮しないのである。必要なのは、現実世界を含めたより幅広い人間の問題解決行為のモデルに基づいた類似

設計である。前述のクライアント・サーバモデルは、その名前の通り、現実世界における人間の職務分担による問題解決をモデル化したものであるし、ここで挙げたデザインパターンという概念自体、元々建築の分野で提起されていた設計手法である[27]。すなわちデザインパターンというソフトウェアの設計問題に対する類似設計を実現するためのモデル自体が、建築の分野からの流用なのである。

以上で明らかなように、特に情報技術においては、設計者の創造は現実世界における様々な問題解決行為を流用して行われている。よって、こうしたさまざまな問題解決のモデル化とその蓄積、再利用による類似設計が有効ということになる。特に知識工学の観点からは、知識ベースシステムの構築に関して、具体的に実現されている設計事例からの帰納的な知識の抽出が提案されており[28]、またデータベース中の過去の設計事例から、与えられた設計要求に類似した事例を検索するといった知識処理に関する先行研究もある[19]。

以上をもとに、「知識」そのもののみではなく「知識に基づいた解決法」、すなわち「問題解決行為」に特化したモデルを蓄積し、その再利用自体をシステムの開発と捉える手法が有効であると考える。本稿では、そうした開発者の問題解決のモデル化と再利用を目標とした「問題解決指向開発」を提案し、今後の研究の方向性とするものである。

5. おわりに

筆者がソフトウェアの再利用を問題意識として研究に取り掛かってから、10年以上が経過してしまった。その間前述の様にさまざまな技術的な変化が起こり、オブジェクト指向技術自体も一般的な技術概念となってきた。しかし端的に言ってオブジェクト指向手法が一般に喧伝されているような大きな成果を上げているようには、決して思えない。IT不況と言われる経済環境においても、相変わらずシステム部門のバックログは大きく、特に対象となる産業自体に競

争力、活力を失っている金融系のシステム部門においても、何ら問題は解決していない。それがひいては、産業の活性力を失わせている要因の一つでもあろう。

筆者には、オブジェクト指向技術への多くのアプローチが、プラクティカルな側面、即ちどのような品質効果を上げているのか、そしてその源泉がどこにあるのか、自己言及を見失っているように思える。実際様々なオブジェクト指向手法が提起されたが、それらは現実のシステム開発に適用するというよりは、その手法自体の実現に重きをおいた、いわば机上のものだったとの感がある。何よりもUMLが表記法に留まっている、と言うよりも留まらざるを得なかったのは示唆的である。例えば現在の情報技術において、最も重要なデータ表現技術であるXMLが、メタ言語であって対象のセマンティックスあるいはモデルに立ち入らないということと同じような意味合いを持っているものと思える。その意味で、対象のモデルに言及していくことが、今後の大きな研究課題であると考える。ここで提案した「問題解決指向開発」にそうした問題意識を取り込んで行きたい。

本論文は、2001年度 第11回ソフトウェア開発環境展における講演を元にしたものである。執筆にあたり、研究の直接の切っ掛けを与えて頂いた株式会社PFUの加藤貞行氏、及び様々な形で研究の支援をして頂いている株式会社ウェッブアイの森川勇次、戸本ひで美両氏に、心から感謝いたします。また、東京大学先端科学技術研究センターの堀浩一教授には、折りに触れアドバイス頂いています。併せて感謝いたします。

参考文献

[1] オブジェクト指向への招待，春木良且，啓学出版（近代科学社），1989
[2] オブジェクト指向実用講座，春木良且，インプレス，1995
[3] オブジェクト指向による再利用，春木良且，松本正雄他編「ソフトウェア

のモデル化と再利用」，共立出版，1996
[4] 人月の神話「ソフトウェア開発の神話」，フレッデリックPブルックスJr.，滝沢他訳，アジソン・ウェスレイ・パブリッシャーズ・ジャパン
[5] ソフトウェア開発ライフサイクル（Software Development Life Cycle），小口達夫他，アイテック，2001
[6] オブジェクトの組織化と進化に関する研究,中谷多哉子他,IPA 独創的情報技術育成事業フィージビリティスタディ報告書,1996
[7] 科学革命の構造，T・クーン，みすず書房，1971
[8] コトラーの戦略的マーケティング，フィリップ・コトラー，木村達也訳，ダイアモンド社，2000
[9] 脳力開発講座，中野俊一，明聖アカデミー 脳力開発研究会，http://www.meisei-a.co.jp/brainpower/koza04.html
[10]ソフトウェア品質評価ガイドブック，東基衛編著，日本規格協会，1994
[11]ソフトウェアの複合／構造化設計，G. J.マイヤーズ，國友義久他訳，1979
[12]ある新種の算術生命体—ナミーバ，竹内郁雄他，情報処理学会第42回プログラミングシンポジウム報告集，2001年
[13]オブジェクト指向システム分析設計入門，青木淳，SRC，1993
[14]C/Sデータベース設計入門，ダニエル・パスコット，落水浩一郎訳，日経BP社，1996
[15]オブジェクト指向98シンポジウムモデリングワークショップ資料（http://www.graco.c.u-tokyo.ac.jp/~tina/oo98/），情報処理学会ソフトウェア工学研究会，1998
[16]情報技術とグローバリゼーション，春木良且，国際交流研究（フェリス女学院大学国際交流学部紀要）No.2，2000
[17] 3層クライアント／サーバ・システム構築技法，喜 英行他，SRC，1996
[18]UMLによる統一ソフトウェア開発プロセス,ジェームズ・ランボー他著,日本ラショナルソフトウエア訳,翔泳社，2000
[19]UML入門ソフトウェア開発の”新・共通言語”をマスターしよう Part2 UMLを用いたシステム開発，奥村洋，DB Magazine2001年6月号
[20]オブジェクト・モデリングのソフトウェア・プロセス，山田正樹，http://www.metabolics.co.jp/OOTechnology/OOPA/，2000
[21]ユースケースマップ，R.J.A.ブ—ア他，佐藤啓太他訳，トッパン，1999
[22]かんたんUML，オージス総研，翔泳社，1999
[23]事例ベースアプローチによる設計支援　設計支援システムSUPPORTにおける事例の再利用（事例ベース推論と類推），仲谷善雄他，人工知能アブストラクト No.075-005，人工知能学会，1990

[24]プロセス情報に注目した設計開発支援，田浦俊春，吉川弘之監修「技術知の本質」，東大出版会，1997
[25]オブジェクト指向における再利用のためのデザインパターン，エリックガンマ, 本位田真一他訳，ソフトバンクパブリッシング，1999
[26]パターンとフレームワーク: オブジェクト指向ソフトウェア開発技術, R.E.Johnson他，共立出版, 1999
[27]A Pattern Language，C. Alexander et al.，Oxford University Press，1977
[28]帰納的学習と演鐸的説明づけによる分類型知識の獲得支援，辻野克彦他，研究報告「情報学基礎」Vol12，情報処理学会，1991
[29]Expert Systems for Structural Design，M. L. Maher， Journal of Computing in Civil Engineering, Vol.1，No.4，1987,