TM-0093

ソフトウェア開発からみた第五世代計算機の役割

新言語普及グループ

February, 1985

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

ソフトウェア開発から見た第五世代計算機の役割

新言語普及グループ
斎藤　信男
石井　義興
岸田　孝一
玉井　哲雄
志村　順
大村　泰司
島田　伴彦
岡田　二郎
大野　尚郎
吉村鐵太郎

目　次

はじめに

このレポートは、新言語普及ワーキンググループの昭和58年度の活動に基づき、このグループのメンバーの見解をまとめたものである。第五世代の目標が明確になるにつれて、現在のソフトウェアに関連するいくつかの問題との関係を見きわめたいという気持ちが強くなる。このワーキンググループのメンバーは、日頃、実際のソフトウェア開発の第一線に立って努力している人達である。具体的な問題点とその解決の難しさをよく知っているからこそ、新しい情報処理技術を目指す第五世代計算機のプロジェクトに対する期待にも熱いものがあるわけである。

ここでは、現在のソフトウェアの生産や保守の体制と技術についての総括的見解、ソフトウェア開発方法論と開発環境の現状と将来、第五世代において１つの中心的な技術となる知識工学とソフトウェア開発との関係、第五世代の技術や新言語の普及と教育の四つのテーマについて、各メンバーのまとめを示した。これらのテーマについてはワーキンググループでも議論したが、このまとめはあくまでも担当したメンバーの個人的見解であることを断わっておく。諸氏の御批判、御意見をいただければ幸いである。

（斎藤　記）

第１章　第五世代におけるソフトウェアの生産性と保守

１－１　プログラムと外部環境との独立姓

ソフトウェア・エージー　石井義興

はじめに

プログラムはユーザ・アプリケーションを記述したものである。記述する内容と記述規則はプログラミング言語によって異なる。一旦開発されたプログラムはコンパイルされると固定化される。固定化されたプログラムは外部環境が変化すると正しく動作しなくなったり、答が正しくなくなったりすることがある。このようなプログラムは外部環境に対する独立性がないと定義する。現在多く用いられているCOBOL, FORTRAN, PL/I等はこの意味でプログラムの独立性は非常に低い。第５世代のコンピュータを用いてユーザ・アプリケーションを構築する際、どんな言語を用いて何を記述すればそのプログラムの外部環境からの独立性を高めることができるだろうか。

第５世代のアプリケーション・システムの開発言語は何を記述すべきで、何を記述すべきでないか、を考えることはプログラムの保守技術の上で非常に重要である。なぜなら意味が同一であっても、その記述が外部環境を意識して記述してあると、外部環境が変化すると、プログラムを変更しなければならなくなるからである。

以下のプログラムの独立性について論する。

- データ・フォーマットとプログラムの独立性
- データ構造をサーチするアルゴリズムとプログラムの独立性
- ターミナル仕様とプログラムの独立性
- ＴＰモニターとプログラムの独立性
- データの存在場所とプログラムの独立性

ここでは IBM 360, 370, 30XX, 308X およびそれとコンパチブルなマシン（富士通　Mシリーズ、日立　Mシリーズ）を用いているユーザを念頭において第３世代の問題点を明確にしたい。そしてそれらの問題を解決するために第４世代のシステム開発手段が持たなければならない機能について考えたい。

１．システム開発の生産性

システムをつくるためにはプログラムを組まねばならぬ。システムといっても次の２種類に大別され、それぞれを作るための手段と方法は大いに異なる。

a)　ユーザ・アプリケーシテョン・プログラム

b)　システム・プログラム

ここではa)ユーザ・アプリケーション・プログラム開発の生産性について論ずる。

ユーザ・アプリケーション・システムは大別して２種類ある。それらは次の通り。

ア）定型業務システム

イ）非定型業務システム

ここではア）定型業務システム開発の生産性について論ずる。

生産性の良し悪しを論ずる場合、いろいろな側面がある。

i）システムを開発するまでの生産性

ii）完成されたシステムが長持ちするかどうかの生産性（変化に対する適応性）

iii)システム保守時の保守の容易性に関する生産性

iv）完成したシステム自身のレスポンス・タイムなどスピードに関する生産性

これら４つの側面の中でiv）はシステムの効率と呼ばれ、システム・デザイン時に十分検討され、システム完成後もチューンナップされる。この問題はアプリケーション・デザインの問題と考え、ここでは論じないこととする。

ここでは作る時点での生産性(i）、システム保守時の生産性(ii, iii)について論ずる。

システム開発時の生産性を論ずる場合、①どの言語を用いて、何を記述すべきであるか、②その言語を用いてプログラムを開発する際の方法論（エディタなどの）が問題となる。ここでは①の問題について論じ、特にプログラムに記述してはならないものを可能なかぎり明確化したい。またコンパイラの害毒についても論じたい。

２．プログラム中に書いてはならないこと

我々がプログラムを組むということは何か特定の言語を用いて、その文法規則に従って書くことを意味する。したがって、その言語の能力が不十分であると記述すべきでないことも記述しなければならないこととなる。例えばCOBOL を例にとって考えると、「記述してはならないこと」を書かなければならないことが多すぎ、そのためにプログラムは「長持ち」しなくなる。COBOL 等のコンパイラ言語はこの意味で良くない。。何を書くと良くないかを列挙しよう。

２．１　プログラムとデータ・フォーマットの独立性

あるプログラムで外部データを入出力する際、データ・フォーマットを記述しなければならないが、記述する内容はそのプログラムで必要な項目を要求されるフォーマットで記述するのではなく、コンパイラ言語では外部データのフォーマットそのものを記述する。これは良くない。なぜかというと一般的にそのデータはそのプログラムだけで利用しているわけではなく、他のシステムでも使用する。そして、その都合からそのフォーマットが変わることがありえる。例えば新項目の追加がありレコード長が長くなったり、そのプログラムで使用していないデータ項目の長さやデータ・タイプが変ったりすることがありえる。もしその様なデータ・フォーマット自身の変化（成長）があると、今問題にしているプログラムはそれらの変化にまったく無関係であるにもかかわらず、そのプログラムを書き直しコンパイルし直さないと正しく稼働しなくなる。これはこのプログラムはその中で記述されているデータフォーマットに従属しているといい、そのプログラムが変化に対し

「長持ち」しないことを示している。すなわちプログラムとデータ・フォーマットの独立性を実現するためには、つまり、この問題を解決するためには、プログラム中で記述すべきデータ・フォーマットはそのプログラムで必要な項目のみを要求されるフォーマットで記述し、この情報はコンパイルされるべきではない。実行時にDBMSあるいはデータ管理がバインディングすべきである。こうすればこの問題は解決する。

２．２　プログラムとデータ・サーチ・ロジックの独立性

プログラムで外部データを入出力する場合、どのデータを選んで処理したいかは、プログラム毎に異なる。したがって、コンパイラ言語ではそのデータをリーチするためのアルゴリズムを手順的に記述しなければならない。また出力に際しても、出力したいデータをどこにストアしたいかをプログラム中で手順的に場所ぎめのためのロジックを記述しなければならない。この様なデータサーチ（あるいはストア）のためのアルゴリズムを手順的に言語で書くことは良くない。すなわち今必要とするデータをどうやって捜すか(HOW) を記述すべきではなく、そのプログラムで必要なデータが何であるか(WHAT)を記述すべきである。データ・サーチのためのアルゴリズム(HOW) はコンパイルされると固まり、これも良くない。なぜならデータストラクチュアは時代と共に変化（成長）するものであり、もし変化するとデータ・サーチのためのアルゴリズムを書き換え、つまりプログラムを書き直し、リコンパイルしなければならなくなるからである。そのプログラムで必要なデータはまったく同一であるのにデータ・ストラクチアュアが他の理由で変化しただけなのにプログラムを書き直さなければならなくなる。つまりプログラムが「長持ち」しないことになるからである。この様なデータ・サーチのためのアルゴリズム(HOW) はプログラム中で手順的に記述すべきではなく、何がほしいか(WHAT)を記述し、実行時にDBMS等がバインディングする方法を用いれば、この問題を解決することができる。

以上の 2.1, 2.2 はE.F.CODDのリレーショナルモデルの提案でもふれられた点である。

２．３　ターミナル仕様とプログラムの独立性

オンライン・リアルタイム・システムやＴＳＳターミナルを利用するシステムを作るためには、プログラム中にターミナルを意識したエリアの記述を書き、そのエリアにデータを移動させる手順を記述する必要がある。この場合、出力すべき画面を手順的に作り、出力する。あるいは入力された画面から必要な情報を各種エリアに移動させるという方法を用いている。手順的にデータを移動することは良くない。画面の定義が行なわれ、そこで使用されている項目名がはっきりすれば出力時に自動的にメモリ上のいろいろなエリアからそれらの項目を集め、出力すれば良いはずである。コンパイラ言語を用いるとこれが不可能であるのはコンパイラが項目名を実行時に引き渡す機能を持たないからである。

またターミナルへの入出力に際し、そのターミナルの物理的特性を意識した記述をすると、ターミナルが新型のものになった時、プログラム変更とリコンパイル作業を併うこととなり、これも好ましくない。したがってプログラム中には論理的なターミナル記述を行

うべきであって、物理的属性を強く意識した記述をすべきできない。この点はターミナルにかぎらず一般の入出力装置に関しても同様である。

2．4　ＴＰモニタとプログラムの独立性

オンライン・リアルタイム・システムを開発する場合は何かのＴＰモニタ・システム（例えばCICS，IMS/DC，AIM/DC，DCCMⅡなど）を利用しなければならない。現在利用可能なＴＰモニターを用いてユーザ・リアルタイム・システム（あるいはＴＳＳアプリケーション・システム）を開発するためにプログラムを書くと、そのプログラムはそのＴＰモニタの基でしか稼働しない。これはそのプログラムがＴＰモニタに従属していることを示している。メーカの都合等でＴＰモニターが新型のものになると、ユーザ・プログラムは全面的に書き換えになり、システムが「長持ち」しない一つの原因となる。これも好ましくない。この問題を解決するためには現存するコンパイラ言語を用いないで新しいシステム開発言語（第４世代言語とでも名付けよう）を用いなければならない。言語では画面を論理的に定義し、ＴＰモニタは単にその画面のやりとりを行なうだけのアクセス・メソードという位置付けにする。第４世代言語はコンパイル機能は最小限度にとどめ、実行時結合機能を十二分に採用する。こうするとユーザ・アプリケーション・プログラムはＴＰモニタと独立になり、ＴＰモニタが変わってもプログラム変更をせずにすむ。

2．5　データの存在場所とプログラムの独立性

プログラムに入出力したいデータが「どこにあるか」ということを意識して、それをプログラム中に記述するとそのプログラムはデータの存在場所に従属したものとなる。すなわち、分数処理環境において、例えば、東京のマシンの中にあるプログラムＡで「東京のデータを読め」、「大阪のデータを読め」と記述してあるとする。ある日名古屋にマシンが設置され、ネットワークが拡張し東京と大阪のデータの一部が名古屋に移ったとする。ネットワークが拡張後、プログラムＡをそのまま稼働させると名古屋のデータはぬけてしまうこととなる。したがってプログラムＡを名古屋のデータも読むように変更しなければならなくなる。業務そのものには何の変更もないのに、単にネットワークが拡張し、データの存在場所が変わっただけでプログラムが以前と異なる答を返すこととなる。これをさけるためにはプログラムＡで場所を意識した記述をせず、単に「どのデータがほしい」かのみを記述する。そして、実行時にどことどこを捜すかを分散処理機能およびDBMS側が判断するような方法をとれば良い。こうするとネットワークの拡張（成長）があってもプログラムは「長持ち」することとする。この場合もバインディングのタイミングは実行時であって、コンパイル時は良くない。コンパイルすることは変化に対する適応性をそこなうことになる。これらはコンパイラの害毒といえよう。

第２章　ソフトウェア開発方法論と開発環境

２－１　新言語普及W.G.　テクニカル・レポート
「第５世代におけるソフトウェアの開発方法論　－仕様とプログラムの問題－」

三菱総合研究所　　玉井哲雄

本ノートでは、まず現在のソフトウェア開発方法論の中から、いくつかの話題をとりあげ紹介する。そこでとくに共通する問題として、仕様記述および仕様とプログラムとの関係に注目する。ついで、新言語（ないし論理型プログラミング言語）によるソフトウェア開発に、これらの方法論がどのように適用されうるか、また新言語を核とするプログラミング環境が、これらの方法論をどう支援するかにつき考察する。

１．プロトタイピング

まず比較的最近の話題として、プロトタイピング技法をとりあげる。プロトタイピングとは、最終的に運用に供するソフトウェアを作成する前に、利用者の要求を明確化するため、あるいは設計の実現可能性を検証するために、実験的な、しかし実際に動く原型（プロトタイプ）を作るという方法をいう。(7)

もちろん、最終的なプログラムを作る前に実験的にプログラムを作って種々のチェックをすることは、古くから行われてきたことであるが、従来はその目的が、アルゴリズムの確認や効率の評価といった、開発者側の問題解決するものであったのに対し、最近いわれるプロトタイピングでは、利用者の要求を明確化することを主要な目的にしている点が異なる。

要求の定義技術は、ソフトウェア工学の重要なテーマの一つであり、ソフトウェアのライフサイクル論の主張も、要求仕様作成の重要さを強調することが大きな柱となっている。しかし、要求仕様をどのような言語を用いてどう記述すればよいかという問題は、技術的にも極めて難しく、定まった解答は見つけられていないといってよい。そのうえ、利用者の要求はもともと曖昧であり、しかも変わり易いのが普通である。そこで、きちんとした要求仕様の作成は、さらに困難なものとなっている。

プロトタイピングは、そのような完璧な仕様を求めることをやめ、適当な想定のもとにとにかく利用者が動かすことのできるプロトタイプをまず作り、それを使ってみることにより本当に欲しい機能を確かめ、また使い勝手のよいシステムの仕様を定めようとするものである。関連する考え方に、エンドユーザ開発がある。エンドユーザが簡単に使えるようなプログラミング言語やツールを提供し、開発はエンドユーザにやらせる。そのような言語は、たとえば非手続き的な性質をもつことなどにより、変更がしやすいという特徴を有するため、ユーザはあれこれ試しながら、自分の望む形にシステムを作っていく。この過程をプロトタイピングとみることもできる。また、そのようにしてエンドユーザが作ったシステムをプロトタイプと見なして、システム開発の専門家が、それに基づいて改めて

性能面等を考慮した、本格的なシステムを作るという方法もあるだろう。

別に関連する話題としては、実行可能な仕様という概念をあげることができよう。実際、プロトタイピングの目的が仕様を確定することにあり、そのために動くプロトタイプを作るのだとしたら、仕様として記述したものがそのまま実行可能であれば、大いに都合のいいことは明らかである。このような研究の例は、S.Gerhart 等のAFFIRM[4] , Goguen 等のOBJ [3] ,Balzer 等のGIST[1] 等、どちらかといえば理論的なものが多いが、これらをプロトタイピングと関係づけて論じる傾向も実際に見られる。

2．プログラム変換法

ここでいうプログラム変換法とは、BurstallとDarlington[2] により提唱されたもので、プログラムを定められた規則の適用により段階的に変換していく手法をいう。彼等のシステムは、再帰的等式によって関数を定義するような関数型プログラミング言語に対して適用され、変換の対象となる式に現われる関数を、その定義式の右辺の表現に置きかえるほどき(unfolding) と、その逆に部分式を対応する関数定義の右辺に照合させ、その関数とを用いた表現に置きかえるたたみ(folding) という二つの書きかえ規則を柱としている。

この変換システムへの入力として与える表現を仕様と見なし、出力をそれに対応するプログラムと見なせば、プログラム変換法は、与えられた仕様からプログラムを合成するプログラム合成技法の一つと見ることもできる。ただ、他のプログラム合成技法と異なる一つの特徴は、仕様とプログラムの差がそれほど決定的なものではないという点である。たとえば、プログラム合成に定理証明技法を援用する流儀では、仕様は通常、述語論理などの論理的表現で記述され、生成されるプログラムは何らかのプログラミング言語により表わされる。これに対し、プログラム変換では、仕様もプログラムも同じ言語で書かれる方が自然である。そこで仕様とは、実行不可能であるか、効率を考えると実用上実行できないに等しいプログラムで、しかしその意味が明解で簡潔なもの、といった位置づけになる。しかし実行ができないとか効率が悪いという評価は、対象とする機械を特定して意味をもつものであり、その意味で相対的な基準であるといえよう。

このプログラム変換の概念は、プログラム合成や最適化の手法として役立つだけでなく、むしろそれ以上にプログラミングの過程につき理論的な考察を行う道具として有用であることが、改めて認められてきている。たとえば、Scherlisと Scott[5] は、“推論プログラミング”を提唱しているが、それはプログラミングに関する論理を取扱う枠組の構築を目指すもので、プログラムの導出過程を形式化するシステムとしてのプログラム変換法は、そのための有力な武器となるはずである。

3．ソフトウェアの部品化

ソフトウェアの生産性の低さが問題とされ、それを一つの契機としてソフトウェア工学が提唱されてからすでに10年以上になるが、その間にソフトウェアの生産性は着実に向上したとはいえ、その上昇率は十分に大きいとはいえず、むしろ増大するソフトウェア開発

の需要の伸びに、追いつけないでいるというのが実態であろう。この問題に対する処方の一つとして、新たなソフトウェア開発に際し、既存のソフトウェアを再利用する、あるいはさらに進めて、ソフトウェアの部品をあらかじめ用意しておき、生産はそれらの部品から適当なものを選んで組み立てるという、工業製品の生産プロセスとの類推から生まれた手法が、かなり以前から有望視されてきた。

この部品化のアイデアは分りやすく、簡単に実現できそうにみえるが、実際に適用しようとすると、いくつかの難しい問題に直面する。その第一は、部品として、どのような単位でどんなものを用意すべきかという問題である。第二に、蓄積された部品を、どのようにして検索できるようにすればよいかも、難しい問題である。これには、部品の仕様をどう記述するかという問題と、部品を利用する際に、欲しい機能に合致する仕様をもった部品をどう見つけるかという問題とがある。第三に、ある問題に対するソフトウェアのために必要な部品が集められたとして、それらを組み合わせ、システムとして合成するのにどのような手段が用意できるか、という点も大きな問題である。

これらの障害にもかかわらず、ソフトウェアの部品化の試みは様々な形でなされてきており、成果をあげている例も多い。古くからあるものでは、数学ソフトウェアのサブルーチン・ライブラリやシステム・プログラム用などのマクロ・ライブラリが、部品集の例といえよう。事務処理の分野でも、典型的な処理パターンをライブラリ化し、問題に応じてパラメータを設定して利用するとい方法が、色々工夫されてきている。とくにCOBOL に埋めこむ形で実現されている例として、JASPOL（日本システムサイエンス）、PRECOBOL（日本電子計算）、HYPERCOBOL（富士通）などがある。

これら部品化を助けるようなプログラミング言語も、様々な形で出現してきている。Modula-2のモジュールや Ada のパッケージは、それぞれ明確なソフトウェア部品の概念を提供している。またSmalltalk に代表されるオブジェクト指向言語は、オブジェクトという部品の積み重ねでソフトウェアが構成できるという思想を実現したものとみることもできよう。

ソフトウェアの部品化がより理想的な形態にすすみ、前述のような問題点が解決されて、部品の検索や組み立てが自動化されるようになれば、これもソフトウェアの自動合成技術の一つ（しかもかなり強力なもの）とみなすことができる。しかしそのためには、必要とするソフトウェアの仕様を厳密に、かつ望むらくは簡潔に記述する方法があり、その仕様に対して適切な部品を検索し、それらを合成する手段が存在する必要がある。その実現は、そう容易なことではないように思われる。

4．仕様の問題

筋立てが三題噺めいてきたが、これまでに紹介した考え方の共通項として、仕様の問題が浮びあがる。改めてそれぞれの技法と仕様との関係をまとめてみよう。

(1) プロトタイピング

プロトタイピングは、エンド・ユーザが仕様を確定することを一つの大きな目的としている。また、実行可能な仕様という技術が、役に立ちうる。

(2) プログラム変換法

プログラム変換法は、仕様から具体的なプログラムへ変換する過程を形式化した手法とみなせる。ここでは、仕様とプログラムの差は、段階的なものである。

(3) ソフトウェアの部品化

部品を蓄積する際には、その部品の仕様を付け加える必要が一般にあろう。また、部品を利用してソフトウェアを開発する際、そのソフトウェアの仕様が、部品の検索や合成のために必要となる。

これらから言えることは、たとえば次のようであろう。

・仕様は重要である。

・しかし仕様の記述は、形式的な記述の難しさ、仕様の変わりやすさ、等色々な意味で、難しい作業である。

・仕様を簡単に実行できるようにするか、あるいはプログラムへの変換操作を追跡することなどにより、仕様を定める過程を反復の可能な柔軟なものにすることが望ましい。

5．新しいプログラミング言語との関係

仕様の問題を初めとし、これまでにあげたソフトウェア開発上の話題はすべて、さまざまな形で第5世代コンピュータにおける新言語と係りうる。その関係には、新言語によるソフトウェア開発に、ここに述べたような技法が適用されうるという面と、新言語やそれのもたらすソフトウェア環境が、これらの方法の実現を支援し、その問題点を解決するという面とがありうる。この両者は、必ずしも全く別のものと区別できるわけではない。

以下で、思いつくものをあげる。

(1) 論理型言語による仕様記述

たとえばPrologによるプログラムの記述も、とくにその論理的側面に注目すれば、従来のプログラム言語と比べ抽象のレベルが高く、仕様とみなすことも可能である。もちろん、同じPrologによる記述でも、抽象度の高い、仕様としての特徴を充分備えたものから、きわめて具体的なものまで大きな幅がありえよう。また論理的側面といっても、Prologの範囲はHorn節の集合ということになるが、より一般化した述語論理によれば、記述力も増し、読みやすさも増す。さらに高階の述語論理まで広げれば、より仕様記述の能力は向上する。

仕様記述の候補としては、関数型の言語も有力である。論理型言語との比較は、さらに今後の研究を俟つ部分が多い。しかしいずれにせよこれらの言語では、仕様とプログラムとの差異は抽象のレベルの違いであり、記法としては同じ形式をとりうることに、注目す

べきだろう。

(2) プロトタイピング・ツールとしての論理型言語

論理型言語で書いた抽象度の高いプログラムが仕様とみなせるとすれば、その仕様は動かすことができるのであるから、論理型言語をプロトタイピング用のツールとして考えてもよいといえよう。実際、Prologでは一般にプログラムを短く簡単に書ける。また Prologは本質的には解釈型の言語であるから、仕様を色々に変えて試してみるのにも便利である。これらの特徴は、プロトタイピング・ツールの重要な要件を満すものである。

(3) 変換法の適用

たとえばPrologにプログラム変換法を適用した場合、上の議論のように、変換前の仕様に相当するプログラムも一応動くのであるから、これは広い意味で最適化の一種とみなせる。変換法は、もともとは関数型言語に対し考えられたものであるが、Prologへの適用は、昨年の佐藤、玉木[6] の発表に続き、今年のロジック・プログラミング・コンファレンス '84 でも発表が多くみられた。

Prologで書けば仕様も動くと述べたが、純粋な論理的関係にのみ注目して仕様を書き、それをPrologで表現しても、プログラムとして走らせた時に意図通り動くとは限らない。これは、論理とプログラムとしての計算メカニズム（すなわちプログラミング言語のセマンティックス）との差といえようが、この間の橋渡しに、プログラム変換の方法が活かせるようにも思われる。もしそれが成功すれば、関数型言語への適用とは違った展開もみられよう。

(4) 部品化とオブジェクト指向言語

オブジェクト指向という考え方は、前にも述べたようにソフトウェアの部品化の概念になじむものといえよう。論理型言語にオブジェクト指向の概念を導入することは、ICOTを初めとして多くの努力がなされている研究分野である。実際、ESP やConcurrent Prolog に、そのような工夫がみられる。

(5) ソフトウェア知識ベース

ソフトウェアの部品化ができても、開発したいプログラムの仕様を与えて、それに適合する部品を検索し、それらからプログラムを合成する過程を機械化するのはかなり難しいのではないかという予想を、既に述べた。これは、部品のライブラリがあり、その外の世界にプログラムの仕様があり、その間に部品の検索と合成のしくみがあるというモデルを想定する限りは、困難が予想されるということである。むしろ、プログラムの仕様を記述するのに必要な問題分野の概念も部品化され、また検索や合成に関する知識も部品ベースの中に組みいれられたようなライブラリを考えるべきであろう。これはまさに、知識ベースの基本的な発想といえよう。すなわち、ソフトウェア部品と、その利用に関する知識を備

えたソフトウェア知識ベースが構築されるべきである。第５世代の研究が、このような知識ベース構築を可能にすることが期待される。

参 考 文 献

（１） Balzer,R., N.M.Goldman and D.S.Wile, "Operational Specificaion as the Basis for Rapid Prototyping," Softyware Engireering Notes, Vol.7, No5, 1982.

（２） Burstall.R,M. and J.Darlington , "A Transformation System for Developing Recursive Programs, " JACM, Vol.24(1977), pp.44-67.

（３） Goguen, J.A. and J.J.Tardo, " An Introduction to OBJ: a Language for Writing and Testing Algebraic Program Specifications, " in Proc. Specifications of Reliable Software Conf., IEEE Comput. Soc., pp.170-189, 1979.

（４） Musser,D.R., "Abstract Data Type Specification in the AFFIRM System," IEEE Trans. Software Eng., Vol.SE-6(1980), pp.24-32.

（５） Schrlis,W.L. anc D.S.Scott, "First Steps towards Inferential Programming," Information Processing '83, pp.199-212, 1983.

（６） 佐藤泰介，玉木久夫，"Prologに於るプログラム変換，" Proc. Logic Programming Conf. '83, 東京, 1983.

（７） 玉井哲雄，"ソフトウェア開発におけるプロトタイピング"，bit, Vol.15 (1983), No.13, pp.9-15.

2－2　第五世代におけるソフトウェア開発方法論
― プログラミング方法論と開発環境 ―

慶応義塾大学　　斎藤　信男

第５世代において、ソフトウェアの開発方法は従来のものとどう異なってくるだろうか。この問題について、特に、プログラミング方法論と開発環境の立場から考えてみる。

１．プログラミング方法論の新しい傾向

ソフトウェア工学のきっかけとなったE.W. Dijkstra の構造的プログラミング以来、プログラムをどのように記述し作成するかという方法論が多数提案されてきた。これらの流れは、その対象とすることにより、次の２つに分類できると考えられる。

(1) 記述要素による方法論

プログラムの記述は、何らかの方法や言語に従って行なう。従来から、手続き的言語が普及し使われてきたが、それを超越した新しい記述法や記述要素、言語がいくつか提案され、実際の処理系が開発されている。論理型、関数型、オブジェクト指向型、データフロー型、メッセージ指向型などがこれに属する。

(2) 作業形態による方法論

プログラミングとは、人間の知的作業である、その作業は、畢竟、何らかの創造をするものであるが、その形態はいろいろと異なってくる。創造的作業は、他の分野でもいろいろと行なわれている。どんな概念に基づき、何を組み立てているものかということが異なってくる。プログラミングカリキュラス、リテレートプログラミング、インファレンシャルプログラミングなどが、新しいものとして提案されている。

記述要素に関しては、個々のプログラミング言語論とともに多くの議論がなされているので、ここでは特に論じない。今後は、これらの方式の融合とより一般的な計算機構や記述形式の追及が行なわれるであろう。

以下には、作業形態についてその問題点を論ずる。

プログラミングカリキュラス(Programming Calculus)

プログラミング作業を、基本操作の繰り返しによる計算(calculus)として実現するのが、この方法である[1)]。これは代数や解析の様に演算規則を定め、その適用により式の値を求めたり式の変形をすることにより目的が達せられれば良い。このためには、何らかの枠組が必要であり、たとえば、E.W. Dijkstra のPredicate Transformer [2)]や、ある種のセマンティック関数などが考えられる。与えられた問題の仕様を、入力条件と出力条件の関係として先ず定義する。次に、その間を埋めるように、前向き又は後向きに、Predicate Transformer なりセマンティックス関数をだんだんに選んでゆき、最終段階としてそれらの

系列が与えられた仕様を満たす様にする。

この方法は、各計算ステップが正しさを保証しているので、出来上がったプログラムの正しさも保証されている。基本ステップの系列に対しては、等価性の定義も容易にでき、等価変換、等価性の証明なども容易になろう。

リテレートプログラミング(Literate Proguramming)

ソフトウェアの開発においては、そのドキュメントの記述が一つの重要な問題となる。計算機を利用した一般的なドキュメント処理系がいくつか実用化されているが、その中で最も精巧なものとされているTEX の開発者D. Knuthが、ドキュメント処理系をプログラム開発に応用したリテレートプログラミング[3]を提唱している。

プログラムの作成時には、アルゴリズムやその実現のためのデータ構造の設計など、いろいろのことを頭で考えながら作業をしている。勿論、始めからしっかりとした設計図ができていて、それらに関するドキュメントは別途作られるという場合もあるだろうが、個々の小さなモジュールやステップごとに考えている場合の方が多い。出来上ったプログラムだけ眺めていても、その背景となったアルゴリズムやデータ構造、設計上の判断などは必ずしも明確にはならない。そこで、リテレートプログラミングでは、それらのことをプログラムと共に同時に作成してゆく。これは１つの文学作品を作成してゆくことと同等と考える。プログラムは、任意の部分に分割され、それぞれに対してドキュメントが書かれる。プログラムも、ドキュメントの一部として清書体で表示される。一方、実際に動くプログラムには、モジュールの集合を１つのプログラムとして言語処理系がオブジェクトプログラムに変換すれば良い。この場合、処理系は機械的にプログラムを読むから、清書体である必要はなく、コードなども処理系が読めさえすれば良い。

Knuth の提唱したシステムは、PascalとTEX を組み合わせたWEB というシステムで、その機能は図 2-2-1に示す通りである。

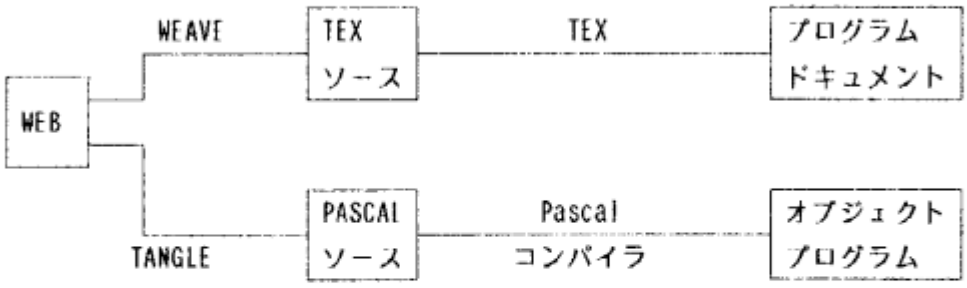

図 2-2-1 WEB システム

リテレートプログラミングシステムとして、言語C とnroff との組み合せ、日本語を取り入れたシステムなどが考えられる。

インファレンシャルプログラミング(Inferential Programming)

形式的な仕様から段階的にプログラムを導出(derive)することは、プログラミングカリキュラスでも用いていたように重要な概念である。この導出過程を、より自然に扱おうというのが、インファレンシャルプログラミング[4]である。対象の問題を規定するのが仕様記述であり、それを解くプログラムは定理と見做すと、プログラムの導出はその定理の証明となる。プログラマがプログラムを記述する時は、形式的な言語を使い、プログラムの導出も完全に形式的に行なわなければならない。計算機システムは、今のところ少し頭が堅いから、それに合わせてプログラム作業も必然的に堅くなる。これに対し、数学者が数学の定理の証明を行なう際には、必ずしも形式的には行なってはいない。数学的概念を使った直視や見通しなど非形式的作業を積み重ねて、最終的には形式の整った証明を作り上げてゆくらしい。

プログラム導出の過程でも、プログラマの直感、常識、見通しなどを十分に使えるようなツール類を用意し、その導出の過程自身を重要視し、その積み重ねが正しいプログラムの完成に到達できるようにするのが、インファレンシャルプログラミングの意図である。

2．ソフトウェア開発環境の現状とその将来方向

ソフトウェアの開発環境は、ソフトウェア工具から出発した1つのソフトウェア工学の分野をなしつつある。ソフトウェアの生産体制を近代化し、その生産性を高めるためには、確固とした概念に基づいたソフトウェア開発環境の支援が絶対的に必要である。

ソフトウェア開発環境を分類すると、次の様になる。

(1) 操作環境

与えられた計算機システムを管理し、その資源を多重化してユーザにリービスするのが操作環境である。これは、いわゆるオペレーティングシステムに対応する。この中には、プロセスやファイルの管理、テキスト編集系、ドキュメント処理系、言語処理系、リンカ、ローダなど、プログラムを作成するための基本的ソフトウェア工具が含まれているが、通常、これらは単独に使われる。この操作環境は、より高度な開発環境を構築する基礎となる。この例として、Unixシステム、高機能ワークステーションなどがある。

(2) 総合的開発環境

操作環境の上にあるいくつかのソフトウェア工具を統合化して、ソフトウェア開発の一連の作業をスムーズに実行することを支援するものを、総合的開発環境という。

操作環境上のいくつかの工具を使えば、同じような機能を提供できるが、個々の機能を単独に使用すると、各工具のコマンドを別々に覚えたり、使用するデータの形式が異なるなどの不便な点がある。総合的環境の場合、コマンド体系が統一化されること、データベースの共通化がはかれること、統一的なユーザインタフェースを実現できることなど多くの利点がもたらされる。環境の提供する機能間の結合も密接になり、ユーザの操作性も良くなる。

このような総合的開発環境の例として、Correll Program Synthesizer 5) Gardolf環境[6]、Mentorシステム[7]などがある。

(3) 知的開発環境

ソフトウェアの開発を、ある方法論に沿って行なうとか、ある管理技術の下で行なうためには、更に知的レベルの高い開発環境が必要になる。与えられた方法論に沿っているかどうかのチェック、その方法論に対する問い合わせに対する示唆、あるいはユーザのガイドなどを行なうためには、その方法論に関する専門知識を格納した知識ベースが中心に存在しなければならない。ユーザの作業は、むしろ設計の様な高度な判断力を有するものに集中し、具体的なコーディングなどの単純作業はシステムの自動化能力に任せる事が望ましい。この環境に定量的評価機能を持たせれば、作成したシステムの定量的評価も行なえるので、システムの最適化の機能を実現することもできる。

このような開発環境の実例は現在のところ見られないが、米国防省のSTARS プロジェクト[8]の目標は、この様な方向をねらっていると思われる。

以下には、上記の環境の具体的な例について、その特徴や問題点を述べる。

Unixオペレーティングシステム[9]

Unixは、スーパーミニコン、ミニコン、マイコンを対象としたTSS システムである。その機能や豊富なユーティリティは、Unixを使い易い操作環境として実現することに寄与している。その機能の特徴は、プロセス管理とファイルシステムである。ユーティリティとしては、各種言語プロセッサ(C, Pascal, Fortran77 など)、ドキュメント処理系、テキスト編集系、リンカ、ローダ、記号デバッグ系、ソフトウェア生成系(make機能)、電子メイル系などがあり、また、それらの結合を、パイプ機能を介して容易にできる。システムの大部分が、高級言語C で書かれているので、機能の拡張や追加が容易である。また、上位の開発環境を構築する事が効率よく行なえる。

プロセス管理やファイルシステムがより一般的に設計されているので、Ｕｎｉｘには性能上の問題がある。実時間システムに向かないとか、ファイルのアクセスが遅いなどの欠点があるが、それらの修正をしたバージョンも見られる。

高機能ワークステーションとネットワーク

高機能のパーソナルコンピュータを基礎としたパーソナルワークステーションは、すぐれたユーザインタフェースを実現する手段として、開発環境の中で利用できる。この場合、高機能のハードウェア資源(たとえば、レーザプリンタなど)の共用、データベースの共通化、ソフトウェア資源の共用、ユーザ間の情報の交換などを実現するためには、高機能ワークステーションを結合するネットワーク(たとえばEthernetの様なLAN)が必要である。これらの上に、良い分散処理系を実現することが、当面の課題であろう。この１っの

例として、カーネギメロン大学計算機科学科でSpice プロジェクト10) がある。

Gandalf 環境

これは、総合的開発環境の１つの例としてカーネギメロン大学で開発されたシステムである。その機能として、ソフトウェアの部品の版数を管理するシステムバージョン制御機能、個々のプログラムを構築する機能、およびプロジェクト管理機能があり、個々の３つの要素が統合化され、１つのユーザインタフェースの下で自由にこれらの機能を使用でき、作成したソフトウェアの部品は共通のデータベースに格納するシステムである。

システムバージョン制御では、ソフトウェアの構築の１つのモデルを想定し、そこに現われる部品を識別子をつけて管理し、任意の時点でのソフトウェアのバージョンを再構築できる機能を提供する。ソフトウェアの部品として、論理的な仕様を持つモジュール（module）を先ず考え、その実現方法に応じてインプリメンテーション（implementation）を設定する。１つのインプリメンテーションに対して、具体的なプログラム（あるいはC の関数群）を作成してゆくが、その作成過程では時間的に並んだリビジョン（revision）が複数個できる。１つの論理仕様に対してその実現方法は複数個考えてもよいので、インプリメンテーションを並列バージョン（paralell version）、時間的に並べられるリビジョンを直列バージョン（serial version）ともいう。ソフトウェアの実行形式は、各モジュールに対してそれを代表するリビジョンを指定して結合して作る。

個々のプログラムの構築は、構文向き編集系を中心に行なわれる。これは、ある特定のプログラミング言語に関する構文の情報を編集系が持っており、構文的に正しい文字列のみが入力編集できるようにしている。同様の編集系は、Cornell Program Synthesizer, Mentorなどでも実現している。Gardalf 環境では、この編集系の中からコンパイラを呼び出し、また、その結果を実行しテストすることが可能である。

プロジェクト管理機能としては、各モジュールに対するドキュメントの付加、データベースのロック機能、データベースのアクセス制御、変更の自動的なログ情報などがある。

知的ソフトウェア開発環境

ソフトウェアの開発は、１つのプロセスに沿って行なわれる。ソフトウェア工学の１つの話題は、ソフトウェアライフサイクルの管理であった。いま、図 2-2-2に示すような簡単なプロセスを考える。従来、このプロセスに沿って人間が作業を行なってきた。勿論、個々の作業、あるいはその集合に対して支援するツール類は使用していた。

これに対し、この人間の作業に於ける専門知識を共通の知識ベースに格納し、その支援によりこれらの作業の効率化、自動化を行なうことが将来の方向として考えられる。設計／仕様記述に対しては、プロトタイピングの実施による設計へのフィードバックができる。実装に対しては、知識ベースを利用したプログラムの自動生成が考えられる。テストに対しては、テストデータの自動生成が可能となる。

この様な知的ソフトウェア開発環境は、単純作業からの人間の解放を可能とし、より秀

れたソフトウェアの設計、コンピュータの利用の高度化等がもたらされよう。

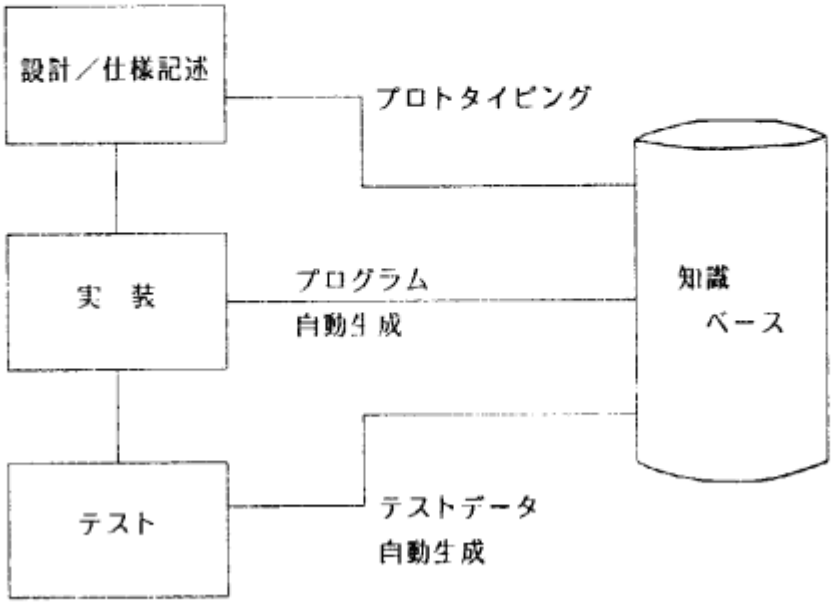

図 2-2-2　知的ソフトウェア開発環境

参考文献

[1] G.D. Berglard : A Guide Tour of Program Design Methodologies, IEEE Computer, Vol.14, No.10, Oct. 1981

[2] E.W. Dijkstra : A Discipline of Programming, Prentice-Hall, 1976

[3] D.Knuth : literate Programming, The Computer Journal, Vol.27, No.2, 1984

[4] W.L. Scherlis and D.S. Scott : First Step Towards Inferential Programming, CMU-CS-83-142, Dept. of Computer Science, Carnegie-Mellon University, Sep. 1983

[5] T. Teitelbaum and T. Reps : The Cornell Program Synthesizer : A Syntax-Directed Programming Environment, in Interactive Programming Environments,(ed. D. Barstow et al.), McGrew-Hill, 1984

[6] The Second Compendium of Gardelf Documentation, Dept. of Computer Science, Carnegie-Mellon University, May. 1982

[7] V. Donzean-Gonge, G. Huet, G. Kahn and B. Lang : Programming Environments Based on Structured Editors : The MENTOR Experience, in Interactive Programming Environments (ed. D. Banstow et al.), McGrow-Hill, 1984

[8] L.E. Druffel, S.T. Redwine and W.E. Riddle, The STARS Program : Overview and Rationale, IEEE Computer, Vol.16, No.11, Nov. 1983

[9] たとえばK. Christian : The UNIX Operating System, John Wiley & Sons, 1983

[10]J.E. Ball et al. : The Spice Project, Computer Science Research Review, Dept. of Computer Science,Carnegie-Mellon University, 1982

第3章　知識工学とソフトウェア開発

3－1　総　　論

構造計画研究所　志村　順

(1) ソフトウェア・エンジニアリングからのアプローチ

15年ほど前、ソフトウェア危機の克服を目標としてソフトウェア・エンジニアリングが誕生した。ソフトウェアの生産性、信頼性などの抜本的改革が目的である。以来、10数年に渡っていろいろな方法論、技法、ツールが提案、評価されてきた。'80 年代にはその成果がソフトウェア開発現場に導入され、効果が実証されつつある。

新言語普及グループのメンバーの所属する日本情報センター協会やソフトウェア産業振興協会のまわりでも、この課題に挑戦していくつかの技術開発を実施してきた。国家レベルで実施された大規模プロジェクトを列記すると次のようである。

(i) モジュール研究組合

時期：昭和48年～昭和51年

目的：事務処理、経営管理、設計計算、ＯＲ及び自動制御の５つの分野について、それぞれの分野でくり返し利用することが期待できる汎用的なモジュールを研究し試作した。

(ii)ソフトウェア生産技術開発

時期：昭和51年～昭和57年

目的：ソフトウェア生産を従来の手作業方式から自動生産方式に切り変えることにより、新しいソフトウェア開発生産のトータル・システムを確立する。すでに推進されてきているソフトウェアの中間製品、部品としての「ソフトウェア・モジュール」をデータベースとして活用しつつ、新たに開発する言語を用いて、顧客の需要に応じたプログラムを「モジュール」の編集・組立てにより自動的に作成することにより、プログラミングの生産性を向上させる。

(iii) ソフトウェア工学

時期：昭和52年～現在

目的：現在のソフトウェア・エンジニアリング領域の最先端技術を対象とした調査研究プロジェクトである。情報処理産業界だけでなく、学界などとも密接な協力体制をとってすすめている。58年度には、プログラム・シンセサイザの調査研究他７つのテーマを実施した。

(iv)　保守技術開発

時期：昭和56年～現在

目的：ソフトウェア開発専業者における保守関連売上げはすでに平均して約40％にも達しており、しかも保守対象ソフトウェアは年々累積され、この

比率は増加傾向にある。また、企業のＤＰ部門では総経費の50～70％が保守に費されており、近年特に保守の問題の重要性が認識されつつある。

しかし、これまで保守の問題解決の努力はほとんどなされておらず、研究レベルでもほとんど無視された状態に置かれている。

このような保守問題の現状および今後の重要性の増大に鑑み、ソフトウェア生産技術開発計画に続くプロジェクトとして“ソフトウェア保守技術開発計画”を開始した。

(2) 評　価

ソフトウェア・エンジニアリングの成果を知識工学的な視点から見てみよう。題材をソフトウェア生産技術開発プロジェクトからとる。開発の過程でプロジェクトの成果を次の２つに分類した。

(i) 開発支援型システム

ソフトウェア生産のプロセスの類似性に着目し、各生産工程での支援を目的とするシステムである。

(ii)ジェネレータ型システム

応用分野ごとのプロダクトの類似性に着目し、各分野ごとにプログラムを自動生成するシステムである。

これらの２つのシステムをエキスパートシステムとしての視点、すなわち知的インタフェース、知識ベース及び問題解決・推論の３つから見よう（表 3 -1-1)。

システム区分	知的インタフェース	知識ベース	問題解決・推論
開発支援型システム	ＪＣＬ エディタ	ツール	なし。 人の作業
ジェネレータ型システム	P.O.L	プログラム・パターン	なし。 生成するプログラムは P.O.L の記述内容から一意的に決まる

表 3 -1-1　人工知能的視点から見たシステム

ソフトウェア開発は知的作業であり、基本的には人がやるものである。システムにはその支援を期待するというのが開発支援型システムの主張である。問題解決・推論は完全に人の分担である。

ジェネレータ型システムにも問題解決・推論の機能はない。それでもプログラムが自動生成できる程度に応用分野を狭くしぼっている。

(3) 知識工学によるアプローチ

ソフトウェア開発の問題に知識工学の成果をとり込むとき、2つの方法が考えられる。一つは全く新しいパラダイムによって抜本的な解決を計ることである。そうはいっても過去に開発した膨大なソフトウェア遺産は無視できない。従って従来からのソフトウェア開発方法を踏襲し、その支援技術として知識工学を導入する。これが2つ目の方法である。本章では後者について、次の2つの具体的例について述べる。

(i) ソフトウェアの知識ベース

図 3. 1-1 のジェネレータ型システムに位置づけられるシステムの提案である。知識ベースを導入することによって、応用分野は事務計算全般が扱え、汎用的なシステムになっている。

(ii) 大規模システムにおける保守エキスパート・システム

特定のシステムを対象とし保守システムの提案である。システムの機能と保守のノウハウとを知識ベースとしてもつ。

3－2　ソフトウェアの知識ベース

日本電子計算株式会社　大村泰司、高田伴彦

1．はじめに

ソフトウェアの開発に対して今日さまざまな警告がなされている。これは、ソフトウェアが複雑化、大規模化しておりかつその信頼性が社会的にもますます要求されるようになったためである。これに対してソフトウェア工学的アプローチからソフトウェア生産にも種々の方法が提案されている。電子ファイルによる複写挿入等の容易性からフレンドリーなエディターは一般的である。また、ソフトウェアのライフサイクルに対応してソフトウェア開発環境、ツールが整備されてきた。

しかし、ソフトウェアの生産は本質的には知的な作業であり人間の創意工夫が重要なファクターである。ソフトウェア生産に関する知識とか技術などの人的要素がソフトウェアの生産には一番重大であるとの報告もされている。そこでソフトウェア生産に係るこれらのすぐれた知識、技術を別の人が利用して生産性を高めようというような要求がでてくる。

一般に、ソフトウェアの生産には次のような作業が必要であろう。

①ソフトウェア生産に係る知識をたくわえ、
②過去にあった事柄は記録し、
③それらを思い出してきて、
④新たな場面に適用していき、
⑤新しい方策を思いつく。

これらのうち近い将来は①～③の部分が知識ベース化され、④、⑤の部分は人間の仕事として残るであろう。最終的には④、⑤の部分もコンピュータにまかせ、人間はよりソフトの部分を担当してゆくことになるであろう。

2．知識ベースの内容と表現

人間の知識は必ずしも体系だっているとはいえない。そして人を説得したり人と議論する時には、その無体系な知識は必ずしも最良のものではない。知識ベースは他の人が利用して意味があるから、客観的で体系だったものに限定されるであろう。その意味で知識ベースを構築する前にその分野の体系づくりが必要であろう。同じことがソフトウェアの知識ベースにも言えて、ソフトウェア生産に係る知識、技術、経験、知恵を体系的に整理することから始まる。この作業については一般にはあまり公開されていないが、それぞれの会社の内部では相当体系化されているのではなかろうか。

ソフトウェア生産に係るノウハウには次の様なものが考えられる。

①ハードウェアに関する知識。
②プログラミング言語に関する知識。
③社内、部内の標準化などのルール。
④ソフトウェア部品とその利用方法。

⑤開発環境に関する知識。
⑥ツールに関する知識。
⑦対象分野の業務知識。
⑧開発方法論に関する知識。
⑨過去の類似システムに関する知識。
⑩情報の入手手段に関する知識。

これらは、近い将来知識ベースとして利用できるのではなかろうか。

知識工学としては、この他にソフトウェア生産に長く携わった人の経験をデータベース化したいものである。特に保守技術やエラー発生時の対応策などは貴重なものである。これらはあまり体系だっているとはいえず、本人も明確に意識していないものもあるが、利用できる形での知識ベース化が望まれる。

ソフトウェアの知識ベースにおける表現には次の様な点が要求されるであろう。

①知識間の関連をもつこと。
②独立的に記述できること。
③知的なアクセスに対して素早く対応ができること。
④推論、発見的な推定ができること。
⑤自己増殖の機能があること。
⑥知識間に遺伝性のあること。
⑦暗黙値があること。
⑧予測の機能があること。
⑨雑多な体系づけられていない知識も表現できること。
⑩あいまいさを含んだ事柄も表現できること。

知識ベースは広い範囲にわたるものになるであろうから、自然言語によるインタフェースが必要になってくるであろう。

3．知識ベースの構築と利用

知識ベースの構築には、知識ベースの管理者がいてその一括した管理のもとに構築されるものと、それぞれメンバー各自が知識ベースを更新していく方法とが考えられる。今までだと前者の方法のみがとられてきたが、知識ベース管理の自動化が進めば後者の方法も有効になる。すなわち、過去の知識との矛盾や重複のチェック、とり入れるべき価値などを判断し、分類し、体系づけて更新する機能である。知識ベースの育成にはこの機能がぜひ必要である。知識獲得の問題である。

ソフトウェア生産に関する知識ベースの構築では、ハードウェアやプログラミング言語に関する知識についてはメーカで構築していける。ソフトウェア生産における規則、標準化ルールのようなものは、ソフトウェア生産の管理セクションで行なっていく。業務分野の知識は、業務のエキスパートがそのノウハウを整理登録することができる。ソフトウェア部品はなどの登録については、ＳＥが適当であろう。実行時のエラーに対しても、従来

のエラー番号の表示のようなものではなくエラーの原因をアドバイスするものが考えられる。このための知識としては、ソフトウェア作成に携わったものが遭遇したエラーと原因等の経験が有効となる。

知識ベースの利用は、設計フェーズ、プログラミングフェーズ、テストフェーズ、保守フェーズに渡る。知識ベースから得た事柄は簡単に利用できることも必要である。例えばソフトウェア部品についての知識を知識ベースから得た時は、容易にとり込める形式になっていることである。知識ベースは推論機構をもっているから、明確な指示がなくても質問に応じた対応ができることも必要となるであろう。

4．一つの事例

ソフトウェア生産性向上のために現実的なアプローチがなされているものに、ソフトウェア部品の再利用がある。従来からの数値計算のサブルーチンや事務計算の汎用部品などは充分に利用されている。しかし事務計算の分野別の特定仕様を含むものについてはまだ実験段階である。

ここで筆者らが試作したソフトウェア部品を利用したプログラム合成のしくみを簡単に述べる。

このシステムではソフトウェア部品を一つのオブジェクトと考える。この中に部品に関することはすべて記述する。参照する他部品、他部品との関係、部品の属性、局所状態を表わす変数の宣言、パラメータを受けとる文型の定義、Target Program Code の生成手続きなどである。このソフトウェア部品を形式的に記述したものをクラス(class) と呼ぶ。これを構文解析し、中間コードに翻訳して部品データベースに登録する。

合成フェーズでは各部品内に定義された文型に従って部品を呼び出してパラメータを与える。この文型をメッセージ(message) と呼び合成フェーズにおいてユーザが入力すべき言語になる。この時部品には部分的にもパラメータをあたえることができる(partial parameterizing) 。メッセージを送られた部品はメモリー上に具体的な形をつくる。これをインスタンス(instance)と呼ぶ。インスタンスは局所変数と生成コードをもち、いつでも局所変数の値を変更できる。部品の中に他の部品の呼びだしが記述してある時は、その部品もインスタンスとして生成される。そしてこれらのインスタンスはシステム内で待ち状態になりいつでもメッセージを受けることができる。このことはシステム側からプログラム合成者に作業のコンサルテーションを可能にしている。クラスやインスタンスへの問い合わせは随時行うことができる。

適当と思われる時点でシステムコマンドをキーインすると、インスタンス間のリンクをたどって最終的なプログラムをつくり上げる。この時必要なパラメータが完全に与えられていないインスタンスがあればシステムはその警告を出してくる。

途中結果の保存と保守は、このインスタンスの局所状態に対して行なう。すなわちインスタンスの状態は合成途中ではメモリー上にあり、保存の時はそれをファイルに書きこむ。また再度の合成作業の時には、それをファイルからメモリー上にもってきて作業を続ける。

入力仕様に対してでもなく、最終合成結果プログラムに対してでもなく、この様にしてインストラクティブなプログラム合成を保守する。

この方法により次の様に事柄が実現され、有効性が確かめられた。

①ソフトウェア部品の利用をその特定分野の固有の言葉で記述する。

②部品の中に他部品の利用の制御をも記述しており、部品利用のノウハウが部品内に入っている。

③プログラム合成において最初から決まった順序で完全な仕様を与えていくのではなくて、不完全さを残しておける。また思いついた順に仕様を入力していける。

④③の状態をシステムは自動的に管理しておりプログラム合成者にアドバイスすることを可能にしている。

⑤プログラム合成の中断やプログラムの保守を自然な形で行なう。

3－3　「ソフトウェア保守ＥＸＰＥＲＴシステムについて」

東洋情報システム　岡田二郎

1．はじめに

総論で既に言及されているように現在のソフトウェア危機 -- ソフトウェア開発要求と開発能力の遊離、開発及び保守に伴う人件費コストの増大 — を抜本的に克服するためには、やはりソフトウェア生産保守の新しいパラダイムを確立することが本来的であろう。「新しい酒は新しい革袋に盛られるべき」である。新言語に対する期待もソフトウェア産業としてはこの点の占める割合いが大きいと考えられる。

しかしながら、近い将来理想的な新言語によるソフトウェア生産技術体系が確立されたとしても、これまでに開発されてきた従来型言語（旧言語というべきか）による膨大なソフトウェア遺産をすべて新言語で置換することは事実上不可能であり、又実際的ではない。

現在稼働しているソフトウェアはすべていわば生きているソフトウェアであり、常に変化し不断の改良を必要としている。現実の旧言語 -- 旧パラダイムによるソフトウェア・システムをサポートする手段が早急に求められている。現在でも、アセンブラ言語が特定領域で使用されているように、新言語が一般的に普及する時代においてもソフトウェア言語の重層的階層性(The Living bvain に於けるTriune conceptの如きもの）が存在するであろう。この見地から旧言語システムのサポートは過渡的なものという以上の考慮に値すると思われる。さらに、現在システムに対するサポート系を検討することは新言語によるそれのプロトタイピングの意味もある。

以上の考察から、ここではソフトウェア・ライフサイクルのコストのうち、ますますその比重を増大している保守に焦点を置き、知識工学的手法の適用を検討してみる。

2．ソフトウェア保守に於ける問題点とニーズの分析

「ソフトウェア保守問題の重要性に対する認識は近年ますます高まりつつある。ライフサイクル・コストの中で保守の占める比率が少なくとも5割を越え、ときには７～８割に達するという事実はすでに数年前から指摘され、その後いくつかの調査で裏付けられてきた。また、蓄積されたソフトウェア遺産の量的規模が増大するにつれて、その管理やさまざまのタイプの保守作業（修正、適応、完全化）の実施に、きわめて難かしい技術的問題が発生することも明らかになってきている。」（ソフトウェア保守技術開発計画全体構想概説書、昭和57年　共同システム株式会社）

ここでは、前掲の報告書を中心にソフトウェア保守に係わる事実と問題点の幾つかを列挙してみたい。

- 保守には多くの費用がかかる。しかも開発に比して割高である。アンケート調査によると、ソフトウェア会社と一般ユーザーを合わせた計18社の一年間に保守されたプログラムは全体の約35％であり、変更ステップ数は保守されたプログラムの約10％で

ある。これは全保有ステップの４％弱にすぎない。また、具体的な例としては、ある信販会社のシステム(COBOL言語で記述、約 100万step) の一部開発を含む保守的作業に常時20名の要員を必要としている。

- 現実の保守作業では全作業時間の約1/3 がドキュメント、プログラムを解読するのに使用されている。これはほとんどのドキュメントが開発用のものであり、保守のために書かれたものでないこと、ドキュメントの修正・追加が不断に行われるため、信頼性が次第に低くなることが掲げられる。従って、ソース・プログラムが唯一の信頼できる資料となっていることが大きく関係している。
- 保守支援ツールが不足している。また、存在していても、支援ツールとして効果的でも、標準化された汎用タイプでもないことが多い。
- 保守要員について、要員不足（量的）とは能力不足（質的）とさらに熱意不足（心理的）の問題がある。要員不足は保守の発生が予想しにくいことや、保守より開発に重点が置かれてることなどによる。能力不足も保守軽視からくる新人の登用や教育不足などに大きな原因がある。担当者の熱意不足にもこのことはいえる。また、一般に保守対象システムが大規模・複雑化するにつれて、保守要員の固定化が起りやすい。このことは保守担当者の向上意欲を阻害する最大の要因となっている。

以上のことから保守で最も望まれているニーズを掲げると、

- 顧客の要求を分析し、既存システムの変更箇所を特定させること。また、変更によって影響の及ぶ範囲を指定できること。
- 修正箇所に対する文書化が迅速にでき、かつ既存文書との整合性が満たされること。
- 完備したテストデータを生成できること。

がある。さらに、保守要員の省略化、ローテーションの促進の観点から、対象システムに対するＣＡＩ機能も強く望まれるものの一つである。

３．ソフトウェア保守ＥＸＰＥＲＴシステムの考え方

ソフトウェア保守におけるニーズに部分的に対応するものとして、知識工学を適用した保守支援システムの概念を考えてみたい。このシステムは大規模運用システムに於けるベテラン保守担当者をＥＸＰＥＲＴとみなし、その保守のknow howを知識ベース化することによって、該当システムに対する保守経験の浅い担当者にコンサルテーションを行なうことを目的とする。具体的には、システムレベルまでブレークダウンされた顧客の要請内容を解釈し、該当システムの変更箇所（プログラム・レベル）を固定すること。及び影響範囲と影響フローを指定できること。さらに特定されたプログラムに対して、保守の観点から重要な情報を対話形式で与えること等が掲げられる。また、保守のドキュメント作成を容易にするために、ドキュメントは端末を通してグラフィカルに作成されることが必要である。このためには、ＥＭＡＣＳ的エディター機能、マルチウィドウ・システム、ローカルエリア・ネットワーク機能を使用する端末が備えていることが望ましい。

このような目的を部分的に実現するために、まず初歩のプロトタイプとして、次のような保守情報コンサルテーション・システムが考えられないだろうか。（図 3-3-1参照）

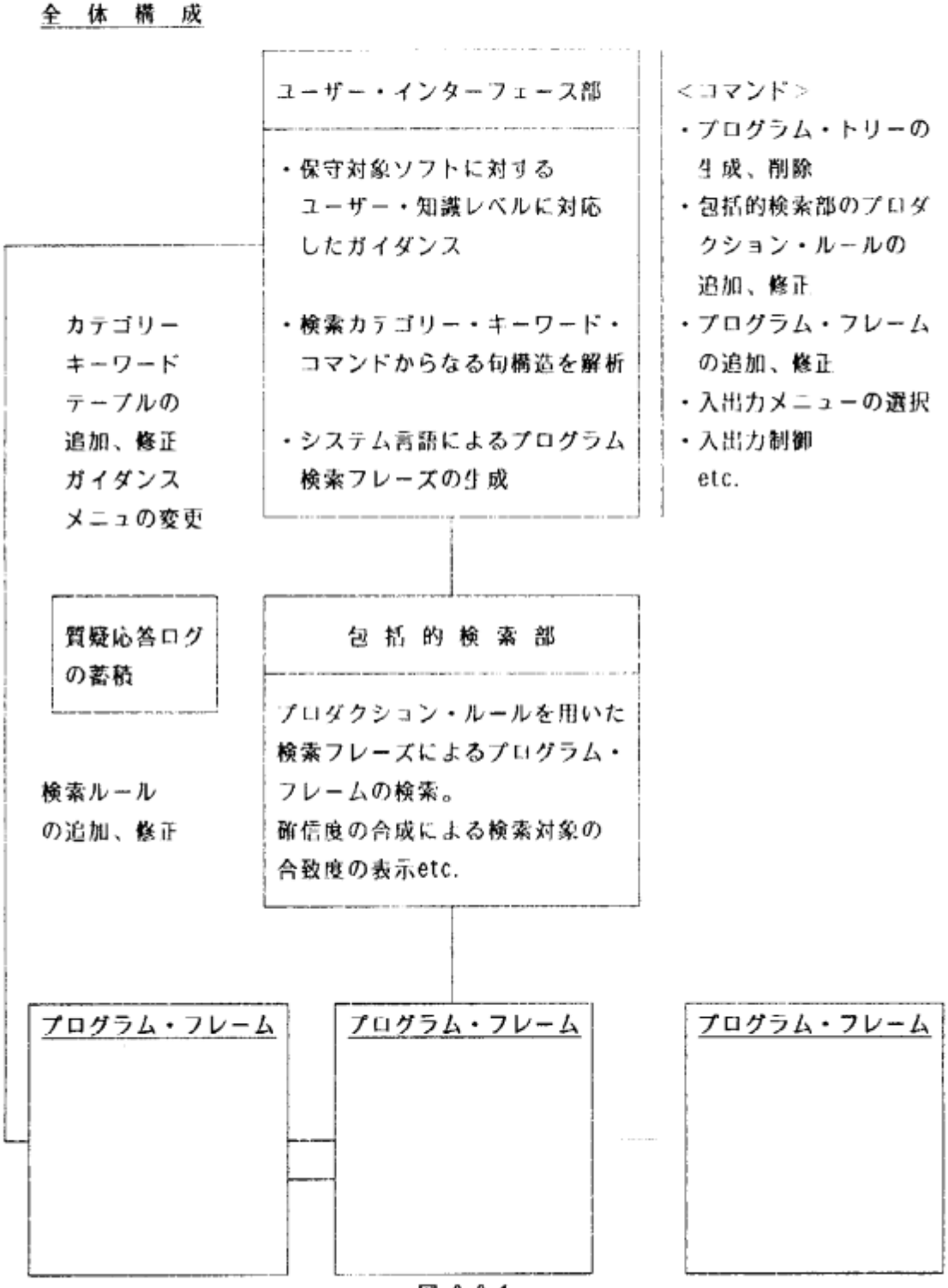

図 3-3-1

システムのユーザー・インターフェース部は、保守対象ソフトウェアに対するユーザーの知識レベルに対応したガイダンスとシステム言語による検索フレーズの生成を行う。このために、検索カテゴリーとキーワードの同義語テーブルを持ち、多様な入力を許すとともに簡単な構文解析と形態素解析を行なって、検索フレーズを生成する。このとき、質問応答のログを集積して、定期的に管理者が応答の実情に応じたカテゴリー、キーワードテーブル及びガイダンス・メニューの変更と検索ルールの追加、修正を行う。包括的検索部はプロダクション・ルールによるプログラム・フレームの検索を一次的に行う機能部であり、確信度を導入して対象の合致度の高い順序で表示する機能をもつ。この推論に基づいて、ユーザーは対話型で自分の望む対象のプログラム・フレームを検索し処理を進める。プログラム・フレームの内容を図 3-3-2に示す。

プログラム・フレームは一般記述、処理情報、入出力記述、特殊情報部など宣言的知識を記述する領域と、そのプログラムであることを同定したり、他のプログラムとの関連を判別するための手続き的知識をプロダクション・ルールとして格納する領域がある。包括的検索部で推論された結果に基づいてプログラム・フレームの指定がなされると、ルールに基づいて質問文を生成し、保守担当者に細部の確認を行う。また、この応答結果より他のプログラム・フレームとの関連情報等を提示する。宣言的知識のうち処理情報内容、入出力内容、関連情報はキーワードの集合、又は論理結合で表示する。この他に、プログラム仕様記述内容や変更仕様記述、アクセスログ情報をデータとしてそのまま保持するデータ部を持つ。

本システムは、保守担当者のＣＡＩ的性格をもち、個々の保守処理の経験をプロダクション・ルール又はカテゴリー、キーワード・テーブルや関連情報として積み上げることによって、次第にシステムの性能をあげてゆくという狙いをもっている。これは従来の保守支援システムのカバーしていない領域であり、知識工学の適した分野と考えられる。プログラムとファイル、レコード、データを結びつけるData Dictionary や、プログラム・レベルでの保守を支援するツール ── プログラム解析編集支援、データフロー解析支援、余波分析支援、保守用テストデータ生成支援等 ── のツールと合わせて使用することにより、より有用性を高めることができるであろう。

プログラム・フレーム　　図 3-3-2

1．一般記述部
- プログラム名
- 著　者　名
- 作成／最新変更日時
- 上位システム名
  下位ルーチン名
- プログラム・リンケージ
  等

2．属性情報記述部
処理情報記述部
- 処　理　内　容 ——→Key Word
- 処　理　形　態
  等

入出力記述部
- DD名
- ファイル名
- レコード名　等

同定・詳細関連記述部
検索対象が、本プログラムと同定されるに必要な、特徴的属性、及び他プログラムとの関連をproduction rule で表現する。

特殊情報記述部
重要なパラメータ、変数等の記述

3．データ部
プログラム仕様記述
変更仕様記述
アクセス・ログ情報
等

第４章　新言語の普及と教育

４－１　第五世代の新言語普及と教育

日本ビジネスオートメーション　大野侑郎

ソフトウェア業のなかの一企業の立場から、「第五世代の新言語普及と教育」について言及する。

第五世代プロジェクトの中期研究開発ステップにおいて予想されるソフトウェア企業の環境は次のとおりである。

(1) エキスパートシステムないしエキスパートシステムもどきのプロジェクトがエンドユーザからの委託で発足する。

(2) (1) に対する既存技術ないし自主技術の不足はこれを、海外技術の導入ないし産学協同プロジェクトで補う。

(3) 従って、利用言語はLispありSmalltalk ありまたUNIX上のC言語ありでマチマチである。向う３年間で、Prologないしその開発環境でのエキスパートシステム開発依頼は民間では多くを見込めない。

(4) それにつけても知識工学に向いた人材は不足しており、充分な教育環境が望まれるが、それを十二分に実現できる見通しは一企業では困難である。

それ故、次のような接点が「新世代コンピュータ技術開発機構」との間で実現できれば、個別企業内における新言語の普及ないし教育が図れる見通しにある。

(a) 第五世代プロジェクトへの参画／からの受注

これについては既に一部実現しているが、規模の拡大を計りたい。

(b) 第五世代プロジェクトと共同で、第五世代プロジェクトの一部を担当したい。候補プロジェクトの例は、たとえば次のとおりである。

(i) Prolog CAIシステム

Ada 教育のエキスパートシステムにならって、Prolog自習システムを制作する。特徴はユーザインタフェースにあり、ソフトウェアハウスの若年プログラマによる知的インタフェース設計へのフィードバック情報が入手できる。

(ii) 設計支援エキスパートシステム

エキスパートシステムを作成する上で、ソフトウェアハウスのほとんど唯一の固有プロブレムドメインは、ソフトウェア作成にある。設計についてはしかし、逐次プロセス間通信のようなしっかりしたフレームの上でエキスパートシステムを作成したい。特徴は問題領域の知識体系の整備にある。

(iii) 「音声・図形・画像応用システム」

たとえばアタリ社の「Britannica(Ver.15)」の知識ベース化のように、「広辞苑」や「大言海」を知識ベース化して

(イ) Enlightenment （自己啓発）

(ロ) Entertainment （遊　び）

を狙ったプロジェクトが想定できる。

(c) エンドユーザとの協同による「基本応用システム」

エンドユーザからの引会いによるエキスパートシステムの一端に次のような例がある。これらを公的資金により、支援する。

(i)　幼児向け教育システム

(ii)　電力プラント故障診断システム

(iii) 医療診断支援システム

(iv)　提案書作成支援システム

これらは、とくにソフトウェア企業がProlog開発環境を提供できないとき、既存海外技術の亜流で開発される見通しにある。

最後に、夢のある希望を述べさせてもらえば、日本文化の輸出に係わる次のような知識ベースシステムの開発にも参画したい。

(d) (i)　日本の品質管理手法紹介システム

(ii)　学術情報紹介システム

(iii) 華道・茶道・書道等紹介システム