総　論

1. 自動プログラミング

1.1 ソフトウェア工学からみた自動プログラミング†

大野　豊††　阿草清滋††

1. はじめに

自動プログラミングという言葉を素直に解釈すれば，自動的に，すなわち人手によらずに機械によってプログラミングが行われることである．このプログラミングという言葉は，狭義にはプログラム・コードの作成とそのデバッグ活動を意味し，広義にはプログラムに対する要求発生後，目的プログラムを獲得するまでの活動全体を意味する．

ソフトウェア技術が計算機技術の発達とともにその歴史を歩んでいることから，当然，自動プログラミングの意味も変遷してきている．計算機が珍しく，それを使うこと自体が問題であった計算機出現の当初は，プログラムは機械語で書かれた．番地の計算が難しく，しかも既存のプログラムを使うには番地の再計算が面倒であることなどから，アセンブリ言語が開発された．これで機械語でのプログラミングに比べてかなり楽にプログラミングが可能となった．さらに，コンパイラ言語が出現し，より高度なプログラムの開発が可能となった．

それら言語の発達は機械語のプログラムを得るまでの過程の多くを自動化している点で，自動プログラミングと呼ぶこともできよう．しかし，今日，FORTRAN のようなコンパイラ言語を自動プログラミングツールと呼ぶ人はいない．目的とするプログラムが複雑になるにつれ，単にプログラミング言語だけの問題では対処できなくなったためである．

自動プログラミングはソフトウェア工学の見地からは次のように捉えることができる．すなわち，ソフトウェア工学の研究成果により明らかにされてきたプログラム開発の方法論を仕様記述からプログラムにいたるまでの変換規則として定式化し，演繹・推論・帰納などの問題解決手法として知られるアルゴリズムを利用したプログラム生成系で実現しようとするものである．したがって，ソフトウェア工学的な自動プログラミングを論ずるには，自動プログラミングを可能とするほどに定式化されたソフトウェア開発手法とはどのようなものであるかをまず明らかにしなければならない．

しかし，これまでのソフトウェア工学の研究では，ソフトウェアの開発過程は必ずしもその目的に十分なほどに定式化されたものではなかった．これは，ソフトウェアの定義が狭義にはプログラムを指すが，広義にはマニュアルや運転・管理の方法なども含んでおり，ソフトウェア工学は広義のソフトウェアの生産性と品質の向上を目指したものであるからである．ソフトウェア工学の目指すところは，予定された期間と予算で信頼性の高いソフトウェアを生産する手法・技術を確立することである．このため，単にプログラミングの過程の効率化のみでなく，よりシステム的な捉え方でのアプローチが主流であったといえる．

ソフトウェアの開発が人手を主としているかぎり，特に大規模のソフトウェアの品質と開発の生産性には限界があり，開発の各過程での自動化を可能なかぎりすすめることは重要な研究課題である．この意味でソフトウェア開発過程の自動化は強く求められており，自動プログラミングもその一工程の自動化として研究されなければならない．そこで，これまでのソフトウェア工学の研究によって明らかにされてきた仕様化技法，設計技法，プログラミング技法などをより形式化し，またプロトタイピング技術や再利用技術などの新しい技術を応用することによって，計算機によるソフトウェア開発の自動化を可能とする方向の研究との整合性をもった自動プログラミングの研究がすすめられる必要がある．

そのために，ソフトウェアを作りあげる過程（ソフトウェア・プロセス）全体を分析し，品質（性能・信

† Automatic Programming in View of Software Engineering by Yutaka OHNO and Kiyoshi AGUSA (Department of Information Science, Kyoto University).

†† 京都大学工学部

頼性）のよいソフトウェアを効率よく生産する方法，技術を確立することから始めなければならない．したがって，ソフトウェア工学からみた自動プログラミングはソフトウェア・プロセスの観点から議論すべきである．

人工知能では問題解決機構の解明とその計算機による実現を目指しているといえるが，この意味では人工知能からみた自動プログラミングは1つの例題に過ぎない．このため，ある特定の目的プログラムを得るための自動プログラミングシステムを研究することが多い．ソフトウェア工学研究では必ずしも目的システムを意識できないので，ソフトウェア開発の個々の過程の要素技術を確立するという点でボトムアップ的な，しかも地道な研究と考えられる．

本稿では，このような観点からソフトウェア・プロセスとその自動化，およびソフトウェアのパラダイムに対する考え方を示し，問題の定式化を重視し，プロトタイピング（仕様の直接実行）とプログラム変換により自動化を図る技法，また，部品化と再利用により，経験・知識の蓄積を図るとともに自動化の手掛かりを得ようとする研究について概観する．

2. ソフトウェア・プロセスとその自動化

自動プログラミングとしては，日常用語（非形式言語）である日本語や英語で表現された要求記述を入力し，目的とするプログラムを自動的に得ることが1つの理想である．このため，一般名詞からデータ型，参照関係から変数，動詞や形容詞から演算子，接続詞や文の構造から制御構造を導くなどして，非形式言語をもとにプログラミングを行う試みもあるが，この非形式言語からプログラムへの変換に対しては，自然言語理解の問題もさることながら，目的システムの環境や背景に関する多大な理解と知識が必要で，自動変換はなおほど遠い[1]．

先に述べたように，コンパイラ言語によるプログラミングを今日では自動プログラミングとは呼ばないが，人間に分かりやすく書きやすい言語によって書かれたプログラムがコンパイラによって変換されて，機械語のプログラムが生成されることは，アセンブラがソースプログラムから機械語への1：1の置き換えにすぎないことを考えると，まさしく自動プログラミングといえる．

コンパイラは機械語より抽象度の高い言語から機械語へ自動的に変換する．ここでの抽象度とは，機械の動作あるいはその意味概念の世界から人間の日常生活の概念の世界に少しでも近づく程度を表すもので，したがって，人間にとってその言語が使いやすく理解しやすくなるという結果になり，言語の高級化ともいわれる．計算機言語の研究開発はこの線に沿った自動化の研究といってよい．

しかし，ソフトウェア工学の研究が始められてから，プログラムの構造化にはじまり，プログラムの構造とともにその作り方，開発の手順が問題にされるようになり，ソフトウェアライフサイクルの概念が打ち出された．言語開発にもこのことが反映され，また開発過程の上流にさかのぼって抽象度がより高い（人間の日常概念により近い高級化）言語が開発されるようになった．これらは仕様記述言語，要求記述言語などであるが，それとともに過程の各段階の方法論を支援／自動化するツールも開発されて，ソフトウェア開発の自動化のための環境が整備されてきた．

また一方，限られた応用範囲に対しては，プログラムの機能と構造が限定されるため，その応用範囲に向いた豊富な言語要素を用意することにより言語をより高級化しやすく，自動化の程度をかなり高くすることが可能となる．これを核にしてその適用範囲の拡張を行うことも，自動化をすすめる1つのアプローチである．

ソフトウェア開発は利用者の要求を計算機が理解できるように変換することであり，言語の重要性は大きい．その言語をどうすべきかの観点からも，プログラミングを自動化するためには，プログラミングという過程そのものを明確に理解する必要がある．システム工学の流れをくむソフトウェア工学からの自動プログラミングへのアプローチとして，このソフトウェア・プロセスに関する研究が最近さかんに行われている．

従来のウォータフォール型のライフサイクル・モデルは，システム工学においては 1950 年代から一般的であったモデルをそのままソフトウェア・プロセスに踏襲したものであるが，開発に適用するには現実的にいくつかの問題点があることが知られている．

新しく提案されているモデルには，スパイラル・モデル[6]のようにウォータフォール・モデルを拡張したものもあるが，現在大勢を占めているのはラピッド・プロトタイピングの流れをくむ連続的開発モデルである．このモデルでは問題指向の過程と機械指向の過程，すなわち，対象問題に関する知識が重要な役割を果たし問題の分析に重点がおかれる過程と，計算機の

構造・仕組みに関する知識が重要な役割を果たす機械指向の過程とに分離するが，それぞれの過程では仕様やプログラムが連続的に詳細化されていくものと考える．

ソフトウェア・プロセスの難しさは利用者のまわりの「実世界」と計算機の中に作られるモデルである「抽象世界」の間の対応を取ることである．これらの間には

抽象化 (abstraction)：
応用システム概念(A)→仕様 (SW)

具体化 (reification)：
仕様 (SW)→仕様 (SH)→製品 (の実現) (p)

の2つのプロセスがあるが[10]，ここでは具体化のプロセスの自動化を自動プログラミングと考える．ここで仕様 (SW) は「何を」するかを定義したものであり，仕様 (SH) は「いかに」するかを定義したものである．

具体化プロセスは抽象化プロセスによって得られた最初の形式的仕様から出発し，ほかの言語体系によるモデルに変換する一連の設計・実現ステップと考えられる．仕様 (SW) はソフトウェア・プロセスの中で最初の形式的な記述である．これをどのような形にするかは自動化システムの方式設計をきめる重要な課題である．AからSWの導出と同様，SWからSHへの変換にもなお大きな意味的ギャップがあるため，この変換を自動化するためには，SWにある程度制約を与えなければならない．われわれの研究室で開発しているシステムでは，この第一仕様を与えられた問題の扱うデータに関する仕様としている．このシステムは，データの構造と関数の構造との対応関係を利用して仕様 (SW) から関数型の仕様記述 (SH) を導き，さらにこれを手続き型プログラム(P)に変換する．前半のステップでは与えられた SW をもとに仕様部品のデータベースを自動検索するという手法をとっている[2]．

人間の思考の形態は「発散」過程と「収束」過程からなるといわれる．与えられた問題を解決するために，いろいろな文献に当たったり，ほかの開発者と議論したり，ときにはブラブラと歩き回ったりして，広くその問題に関する情報を集める過程である．ブラウジング過程ともいえるこの過程は，人間の創造的な活動の多くが現実に存在する知識の組み合わせ・変形と類推により行われるとすれば，関連する事項をより多く得るために重要である．この過程では図形を用いることが多く，また，各種の情報をランダムともいえるほど多方面にわたりアクセスするため，その計算機支援はかなり難しい．収束過程は発散過程で得られた知識を組み上げて1つにまとめる作業である．ブラウジング作業に比べて部屋にこもってじっくりと考える過程であり，計算機支援しやすい過程といえる．

ソフトウェア・プロセスの各過程もそれぞれ発散過程と収束過程からなると考えられるが，ソフトウェア開発全体をみると，その初期過程である要求分析の過程は発散過程がより重要な過程であり，プログラミング過程は洗い出された要素を組み上げることが重要という意味から収束過程といえる．利用者のいる実世界から計算機処理する対象を切り出す過程である要求分析過程と，切り出された結果を入力され，計算機内という抽象世界での変換作業であるプログラミング過程では，計算機内に閉じているという点で計算機支援・自動化がしやすい．

この意味で問題領域の定義が簡単な問題については，自動プログラミングが容易となり，ソフトウェア工学の分野でソフトウェア自動合成としてよく研究開発されているものはこの種のものが多い．問題領域の定義ができると，そこで必要とされる概念が定まり言語が用意できるからである．

ソフトウェア工学の立場からの自動プログラミングはソフトウェア開発全体の自動化を目指しているわけであるが，その中で近年ソフトウェア工学での研究で話題となっているものの1つはプロトタイピングに関するものであり，これは問題指向過程の支援ツールであり，もう1つの話題である再利用技術は再利用可能なものを数えあげることができるという点で機械指向過程の支援ツールといえる．

3. ソフトウェア開発のパラダイム

ソフトウェア開発は「実世界」と「抽象世界」を結び付けることであるが，言い替えると計算の方式と人間の思考過程の隔たりを埋めるものがソフトウェア開発である．一般にこの間の距離が大きいほどソフトウェアの開発が難しくなる．計算機の抽象度を上げることでこの問題に対処しようとする立場から，人間の思考過程に近い計算モデルが研究されている．計算モデルとは，計算機によって翻訳・解釈され実行されることを前提としてはいるが，問題解法を論理的に記述するための体系である．この計算モデルに基づいて，パラダイムと呼ばれるソフトウェアの表現法と構成法を定めることができる．しかし，現在研究が進んでいる

のは関数型，論理型，オブジェクト指向など，プログラミング・レベルにおけるパラダイムである．

これに対して，ソフトウェア・プロセス全体をどのように進めていくかを議論するソフトウェア開発過程全体のモデル化，およびそれに対応するソフトウェア開発パラダイムが考えられる[24]．古典的な開発パラダイムは周知のとおりウォータフォール型のライフサイクルモデルに基づくものである．これはプロジェクト管理を指向したものといえる[17]．新しく提案されているモデルのうち，以下に列挙するものはいくつものフェーズ分けを排した連続的開発モデルである．

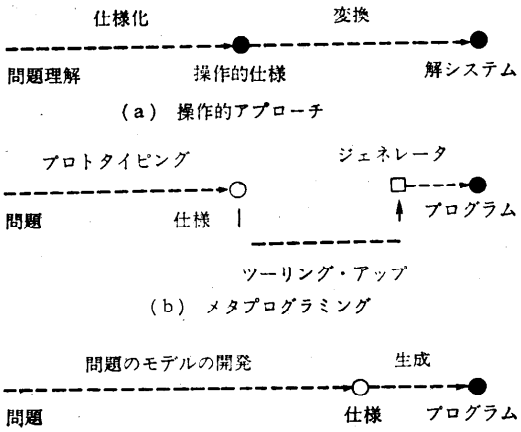
なお，仕様の与え方という観点からは，完全な形式的入出力仕様をもとにする構成的定理証明技法，および不完全仕様記述である例題による仕様から帰納的にプログラムを合成する技法などもこの開発パラダイムに含まれ，ソフトウェア工学的アプローチにおいてもその応用の可能性を追求する必要はあるが，これらの技法については人工知能からみた自動プログラミングに譲る．

操作的アプローチ

問題指向の過程と機械指向の過程を分離した連続的開発モデルの代表的なものは操作的 (operational) アプローチ[22]と呼ばれるものである．問題指向の開発過程はすなわちプロトタイピングの過程であって，その結果は直接実行可能な仕様（操作的仕様）で記述される．機械指向の過程は操作的仕様を出発点とするプログラム変換の過程である（図-1(a)参照）．このアプローチで用いられる言語は広領域 (wide-spectrum) 言語と呼ばれるもので，操作的仕様も変換されたあとの仕様も，同じ枠組みの言語で記述される．PAISLey[23]や Gise/Glitter[8]は操作的仕様記述言語およびその変換システムの例である．

メタプログラミング

操作的アプローチと並ぶ新しいソフトウェア開発パラダイムの1つに，メタプログラミング[11]がある．これも2つの開発過程を含むモデルで，まず最初はやはりラピッド・プロトタイピングによる問題指向の過程である．ここで，プログラムの抽象的形式，すなわち計算モデルが決定される．抽象世界と実世界のモデルの隔たりの大きさがソフトウェア開発の難しさを決めるため，このモデルの選択がメタプログラミングにおいても重要である．次に機械指向の過程がこのパラダイムに特徴的なところで，目的とするプログラムをプロトタイプから直接変換するのではなく，目的プログラムを生成するプログラム（アプリケーション・ジェネレータ）を書くことによって開発をすすめる．したがって，この過程はジェネレータを作成するツーリング・アップの過程と呼ばれる（図-1(b) 参照）．



(c) 非手続き的仕様記述からのプログラム自動生成

図-1 ソフトウェア・プロセス・モデル

非手続き的仕様記述からのプログラム自動生成

MODEL システム[15]のように，厳密かつ詳細な非手続き的仕様記述から完全に自動的にプログラムを生成する方法においては，開発過程は問題の詳細な仕様を開発するという単一の過程しかなく，プログラムはその副産物として自動的に得られるという考え方である[20]（図-1(c)参照）．仕様記述には要素データ間の関係や出力データを生成する関数が等式を用いて記述される．この記述はデータフローグラフに変換され，ノードの直線化を施すことによって手続き型プログラムを生成する．このようなアプローチでは，完全な仕様記述を得るまでの工数を減らすための計算機支援のあり方が問題で，また，そもそも完全な仕様記述が可能かが問題になろう．

ソフトウェア開発環境

ソフトウェア・プロセスの観点からみれば，あるソフトウェア・プロセスのパラダイムは特定の支援環境を必要とする．一方あるソフトウェア環境は若干の特定のパラダイムを支援するように作られ，ほかのパラダイムについては必ずしも支援できない[18]．最近研究が着手されつつあるプロセス・プログラミングの観点からすると，ソフトウェア開発環境のツールはソフトウェア・プロセス・プログラムのオペレータと考えられ，個々のソフトウェアをインスタンスとする型（クラス）の階層によりその構成が決められる[13]．

これらのソフトウェア開発環境としては従来型ライフサイクルモデルに基づくプロジェクト管理環境とプログラミング環境に大別される．

前者は管理データベースを核とするものである．これはソフトウェア開発における種々の仕様や文書の変更がどこに起こったかを捉え，あらかじめ定義された関係に基づいて必要に応じて他の文書の変更などなんらかの動作をおこす機構を含む．しばしばアクティブ・データベースと呼ばれる[14]．

後者は構文指向，言語指向エディタなど高レベルエディタに関するものが中心であり，中にはプログラマの専門家知識を用意して単にエディタとしてでなく，プログラマ助手として利用しようとするものもある．現在の研究の方向は，それらのエディタを生成するいわゆるツール作成ツールあるいはシンセサイザ・ジェネレータの方向に進んでいる[16]．

4. プロトタイプと変換技術

プロトタイピング技術は，ユーザ要求の確定，目的システムのスケルトンの作成，目的システムの技術的問題点の把握などがその利用の目的として考えられる．その性格によって「使い捨て型プロトタイプ」と「発展型プロトタイプ」がある．プロトタイプは目的システムの結合関係やインタフェースを必ずしもうまくモデル化することができないので，「発展型プロトタイプ」であってもプロトタイプとして作られたものがそのままの形で最終システムに取り入れられるものではない．プロトタイピングは，実用に際してはコストの議論が必要ではあるが，今まで動かすことのできなかった仕様記述を直接動かすことができるという意味で，プログラミングの自動化であるといえる．

高度な会話型情報システムの開発では，実世界における利用者の要求が不明確なまま十分理解されておらず，できあがったシステムの利用の経験によって変更される可能性がある．このようなシステムではユーザの要求の変化により反復的 (iterative) 開発が必要とされる．これは抽象化過程におけるモデル化ステップの繰り返しが行われるわけで，ライフサイクルに従ってフェーズ分けされた手順を踏む大規模ソフトウェアにおける開発手法を建築的方法と称するならば，彫刻的方法とも呼んでもよい．反復的開発は，プロトタイプが早くできて変更容易な場合にのみ適用可能な経験蓄積的手法を必要とする．もしそうでなければ，要求の確定のためのラピッド・プロトタイピングを利用し，その確定された要求をもとにフェーズ分けされた開発手法（建築的）が用いられるであろう[5]．

問題指向の過程と機械指向の過程を分離した連続的開発モデルにおいては，一般に前半部はプロトタイピングの過程，後半部はプログラム変換の過程と考えられる．

仕様の直接実行（プロトタイピング）

仕様の直接実行は，ラピッド・プロトタイピングのひとつの形態といえる．仕様が直接実行可能であるとは，仕様記述が縮約，項書き換え，演繹などによるエバリュエーションが可能であることを言う．現在，ラピッド・プロトタイピング・システムとして提供されているものの多くはシステム全体の設計や，内部処理機能の設計を目的とするのではなく，ユーザ・インタフェースの設計を目的としている[21]．言い換えると，プロトタイプ技術の必要性は実世界のユーザに抽象世界で行われようとしていることを知らせることであり，一方，設計の側からは，それは目的システムの性格によるユーザとのインタラクションに関するトランザクション設計を，目的システムの内部構造設計から切り離すことができるという利点がある．

しかし，仕様記述の立場から考えると，問題の定式化のすべてがユーザ・インタフェースであるわけではなく，目的システムの行う処理機能が問題の定式化にとって本質的な概念を含んでいることも多い．操作的仕様と称されるものにはそのような処理に関する記述も含まれる．計算機応用の拡大により，操作的仕様の作成，実行には問題分野のさまざまな知識が必要となった．この問題分野の知識を駆使して操作的仕様に基づくプロトタイピング作業を支援するのが知識によるソフトウェア・アシスタントと呼ばれるもので[9]，いわゆるエキスパート・システムで実現しうるという立場で研究，開発をすすめているプロジェクトもある．

プログラム変換

ソフトウェアの具体化プロセスはある言語体系で与えられた仕様を出発点として，他の言語体系によるモデルに変換する一連の設計ステップと捉えることができる．一般に仕様は高レベルな言語体系で与えられ，変換のゴールである言語体系は低レベルである．ここでのレベルとは現在の計算機の計算機構にどれだけ近く，また，人間の思考過程にどれだけ近いかで定義される．

プログラム変換もソフトウェア具体化プロセスで利用される技術であるが，必ずしも言語体系が入力と出

力で異なるわけではない．プロトタイプとして得られたシステムを変換して実用的なソフトウェアにしようとする試みの手法であるプログラム変換もまた知識に基づく過程である．この過程は問題指向の仕様記述を機械指向の記述に変換するものといえる．簡単にいえば，効率の向上，空間的な最適化を行い実用的な目的システムを得る一過程である．すなわち，仕様として記述されたセマンティックスを保存して，仕様の書き換えを行う．これは一種のコンパイラとも考えられるが，構文に従って再帰的にプログラムを分解すればよいコンパイラと異なり，入力仕様のセマンティックスの解釈に多義性があり，また，プログラム変換のゴールとしてのプログラムに多くの可能性がある．そのため，プログラム言語の構文や意味に関する知識をもとに変換規則を選択するプロダクションルールを構成することになり，変換そのものは自動化できても，ルールを選ぶためのインタラクティブなフロントエンドを必要とすることが多い[3]．

プログラム変換を基礎としたソフトウェア開発においては，変換戦略（知識）はいうまでもなく，変換される初期仕様の記述の仕方も問題になる．初期仕様によっては，変換規則を適用できなかったり，適当な変換戦略を発見できないことがあるからである．

われわれの研究室では，初期仕様をプランと呼ぶ述語で記述することにより，少ない変換規則を用いて実用的なプログラムに変換する研究を行っている．また，仕様からプログラムへの変換を考えたとき，仕様記述とプログラムでは抽象レベルが異なるために，汎用的な変換規則だけでは不十分であり，抽象レベルを埋めるための知識や問題分野に応じた変換規則が必要となる．われわれは，変換過程を抽象レベルに応じて分け，各過程で異なった変換戦略を適用することによって，このレベルの差を埋めようとしている[7]．

5. 再利用技術と知識の利用

特に高度に複雑化したシステムを構築するには，信頼性の評価された部品を利用し，しかも確立された手法でそれらを組み合わせてシステムを作りあげることができれば，信頼性の予測と制御が可能となり，品質の高いシステムの効率のよい実現が期待できる．さらにシステムとしての価値を考えると，単に個々の部品の機能を使い切るだけでなく，それよりもさらにシナジィ効果としての価値，すなわち部品の共生による高度な機能の創出の効果が期待されなくてはならない．

一般に，ソフトウェア工学で対象とする問題はある程度の規模と複雑さをもったシステムであるが，このようなソフトウェアシステムにおいても信頼性の評価されたソフトウェア部品の利用は全体の信頼性の制御には必要不可欠のものである．また，自動プログラミングという観点からもきわめて実際的効果の大きいアプローチである．

再利用技術はさまざまな観点から捉えられるが，大きくわけると仕様やプログラムの一部分（部品やブロック）を再利用するものと，アプリケーション・ジェネレータや変換システムにおける規則やパターンを再利用するものとの２つがある[4]．したがって，たとえば操作的アプローチの前半過程において，操作的仕様を部品化することも可能であるし，後半過程の変換規則も再利用技術の対象となる．メタプログラミングも再利用の観点からすれば，ジェネレータに含まれるプログラム生成パターンの再利用と捉えられる．古典的パラダイムにおいては，プログラム仕様やプログラム・コードを部品として利用するといった組立ブロックの再利用が中心になろう．もっとも，部品化と再利用という言葉の意味からすると，この組立ブロックの再利用の方がよりふさわしいと思われ，それに関する研究，開発は国内外で多く行われている．

部品化の問題としてソフトウェアの柔軟性があげられる．ソフトウェアがハードウェアと異なる点はその可変性にある．部品として用意されたものも，時として，不要な部分を削除したり，一部，新たに手を加えたりしがちである．部品の組み合わせに際しては，自動的に不要部分の削除などの最適化処理が必要とされる．

部品は１つの知識と考えられ，部品データベースは知識データベースといえる．また部品の組み合わせ方に関する知識はメタ知識であり，これらの扱いには人工知能の分野での研究成果に期待される部分も少なくない．しかし，部品の組み合わせというのは，計算機の命令語を組み合わせること，高級言語の文を組み合わせることと本質的な差異はなく，この意味で部品化と再利用はこれまでのソフトウェア開発とほかならない．

部品化・再利用に関しては，オブジェクト指向のプログラミングパラダイムが特に注目される．このパラダイムで用いるクラスという概念は，実世界の対象を抽象化する能力にすぐれているため，ソフトウェア開発の発散過程とのギャップが少なく，またインヘリタ

ンス（継承）という機能を用いることにより，ほかの部品と整合をとるための修正も行いやすいという利点がある．さらに，クラスはフレーム型の知識表現と解釈することもできるため，部品データベースを知識データベースとみなして知識処理を行うことも容易であるからである[19]．

6. おわりに

人工知能の分野での知識の利用とソフトウェア開発の効率化とはきわめて関連が深いものと考えられる．しかし部品化・再利用にみられるように，知識の利用もソフトウェアにほかならず，人工知能の研究で知識の処理手法が確立されたとしても，ソフトウェアの開発・研究が不要になるのではなく，より高度なソフトウェアの開発を可能とする足掛かりができたに過ぎない．たとえば，かつてコンパイラ言語は自動プログラミングと呼ばれたが，その出現によってわれわれはプログラミングから解放されたわけでなく，それまでより複雑なソフトウェアシステムの開発が可能となったに過ぎない．

ソフトウェア工学の目指している自動プログラミングとはこのように，より高度ソフトウェアシステムの開発手段の獲得であるともいえよう．したがって，今後ますますその研究開発はすすめなければならないが，そのためにはソフトウェアの基礎的研究も強力にすすめる必要があると考える．

一般に革新的な開発アプローチについては，技術的問題もさることながら，組織的な問題，すなわちそのアプローチを推進する体制を保守的な組織を説得して確立することの方が難しいという指摘が多い[12]．また，開発されたソフトウェアの革新的技術を実際面に活用していく技術移転はここ数年来議論されているが，なかなか進展をみていない．しかし，有効な技術が使いやすいよく構成されたツールとして供給されていれば必ず実用されていくものといえよう．

謝辞 本稿を草するにあたり，京都大学工学部情報工学教室の鰺坂恒夫助手にご協力いただいた．ここに厚く謝意を表する．

参考文献

1) Abbott, R. J.: Program Design by Informal English Descriptions, Comm. ACM, Vol. 26, No. 11, pp. 882-901 (1983).
2) 鰺坂恒夫，阿草清滋，大野 豊：関数スキーマベースを用いたソフトウェア設計自動化，情報処理学会論文誌，26巻2号，pp. 304-311 (1985).
3) Balzer, R.: A 15 Years Perspective on Automatic Programming, IEEE Trans. Softw. Eng., Vol. SE-11, No. 11, pp. 1257-1268 (1985).
4) Biggerstaff, T. J. and Perlis, A. J.: Special Issue on Software Reusability: Foreword, IEEE Trans. Softw. Eng., Vol. SE-10, No. 5, pp. 474-477 (1984).
5) Blum, B. I.: Iterative Development of Information Systems: A Case Study, Software-Practice and Experience, Vol. 16, No. 6, pp. 503-515 (1986).
6) Boehm, B. W.: A Spiral Model of Software Development and Enhancement, Proc. Int'l Workshop on the Softw. Process and Softw. Environments, ACM SIGSOFT Softw. Eng. Notes, Vol. 11, No. 4, pp. 4-5 (1986).
7) 星野 寛，江指正洋，阿草清滋，大野 豊：論理型言語を用いたソフトウェア開発，第1回ソフトウェア科学会ソフトウェア研究会（関西地区），SW-87-1-3, pp. 17-24 (1987).
8) Fickas, S. F.: Automating the Transformational Development of Software, IEEE Trans. Softw. Eng., Vol. SE-11, No. 11, pp. 1268-1277 (1985).
9) Frenkel, K. A.: Toward Automating the Software-Development Cycle, Comm. ACM, Vol. 28, No. 6, pp. 578-589 (1985).
10) Lehman, M. M.: A Further Model of Coherent Programming Processes, Proc. Software Process Workshop, IEEE Comp. Soc. Press, pp. 27-34 (1984).
11) Levy, L. S.: A Metaprogramming Method and Its Economic Justification, IEEE Trans. Softw. Eng., Vol. SE-12, No. 2, pp. 272-277 (1986).
12) Mathis, R. F.: The Last 10 Percent, IEEE Trans. on Soft. Eng., Vol. SE-12, No. 6, pp. 705-712 (1986).
13) Osterweil, L.: Software Processes Are Software Too, Proc. of 9th Int'l Conf. Softw. Eng., pp. 2-13 (1987).
14) Penedo, M. H.: Prototyping a Project Master Data Base for Software Engineering Environments, Proc. of the ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. on Practical Software Development Environments, ACM SIGPLAN Notices, Vol. 22, No. 1, pp. 1-11 (1987).
15) Prywes, N. S. and Pnueli, A.: Compilation of Nonprocedural Specifications into Computer Programs, IEEE Trans. Softw. Eng., Vol. SE-9, No. 3, pp. 267-279 (1983).
16) Reps, T. and Teitelbaum, T.: The Synthesizer Generator, Proc. of the ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. on Practical Software Development Environments, ACM

Softw. Eng. Notes, Vol. 9, No. 3, pp. 42-48 (1984).
17) Riddle, W. E.: Report on the Software Process Workshop, ACM SIGSOFT Softw. Eng. Notes, Vol. 9, No. 2, pp. 13-20 (1984).
18) Riddle, W. E. and Williams, L. G.: Software Environments Workshop Report, ACM SIGSOFT Softw. Eng. Notes, Vol. 11, No. 1, pp. 73-102 (1986).
19) 垂水浩幸，岡村和男，阿草清滋，大野 豊：クラス再利用のためのオブジェクトモデル，コンピュータソフトウェア，Vol. 3, No. 3, pp. 61-70 (1986).
20) Tseng, J. S., Szymanski, B., Shi, Y. and Prywes, N. S.: Real-Time Software Life Cycle with the Model System, IEEE Trans. Softw. Eng., Vol. SE-12, No. 2, pp. 358-373 (1986).
21) Wasserman, A. I., Pircher, P. A., Shewmake D. T. and Kersten, M. L.: Developing Interactive Information Systems with the User Software Engineering Methodology: IEEE Trans. on Soft. Eng., Vol. SE-12, No. 2, pp. 326-345 (1986).
22) Zave, P.: The Operational versus the Conventional Approach to Software Development, Comm. ACM, Vol. 27, No. 2, pp. 104-118 (1984).
23) Zave, P.: An Overview of the PAISLey Project—1984, ACM SIGSOFT Softw. Eng. Notes, Vol. 9, No. 4, pp. 12-19 (1984).
24) 大野 豊：ソフトウェア工学の背景と展望，情報処理，Vol. 28, No. 7, pp. 845-852 (1987).

（昭和 62 年 9 月 7 日受付）