Vol.2011-IS-117 No.8
2011/9/6

情報システムのソフトウェア開発工程再考 ープログラミング工程は，設計か製造かー

小高文博† 佐藤建吉††

現代社会を支える情報システムは，巨大化し階層化された複雑なシステムであるが，複雑化した仕様によるシステム完全理解の困難さによる実装の不完全のため，未来に障害となりうる要因を宿命的に抱えているシステムとも言える．
情報システムの核となるソフトウェア開発プロセスの信頼性向上や生産性向上を目的とした言語や開発方法・技法が生まれたが決定打はない．これは，プログラミングの開始時点ではまだ設計工程が完了形ではないにも関わらず，製造工程として，プログラミングを行うためである．
プログラミングを設計工程としてとらえ，詳細設計とプログラムコードを同時に表せるような設計記法の可能性について考察する．

Rethinking information systems software development process
-Programming process, or design or manufacture?-

Fumihiro Odaka† Kenkichi Sato††

Information systems that support modern society are a complex system or layered to massive.
And because of the incomplete implementation of a complete understanding of the difficulty and complexity of the system by design, said system also having fatally factors that could be an obstacle to the future.
How decisive was born languages and development techniques aimed at improving the reliability and productivity of software development process is not the core of information systems.
This is in the form at the start of the programming process is complete yet, despite the lack of design, manufacturing as a process, which is for programming.
We see the program as a design process. Then, consider the possibility of rules that are designed to represent the detailed design and code programs simultaneously.

1. はじめに

1.1 情報システムとは

情報システムは，「世の中の様々な問題を認識し解決するために必要な情報を，保存，管理，変換，応用，流通するための仕組みであり，その仕組みの構築にあたりコンピュータを使用したものである．」と定義できる．現在では，この情報システムは，巨大化し階層化された複雑なシステムであり，複雑化した仕様によるシステムの完全理解の困難さによる実装の不完全性のため，未来に障害となりうる要因を宿命的に抱えているシステムとも言える．社会的に重要なものでありながら多くのリスクを潜在的に含んでいるシステムである．

情報システムの中核をなすのはソフトウェアであり，システムを構築する主な作業は，ソフトウェア開発である．また，ソフトウェアを構成しコンピュータを動作させるものとしてプログラム言語がある．プログラム言語は，機械語(machine language)からアセンブラ言語(assembly language)が誕生しフォートラン(FORTRAN: FORmula TRANslation)，コボル(COBOL: COmmon Business Oriented Language)などの高級言語へと発展してきた．同時に，プログラムの生産性の効率化や正確性のためにサブルーチン化やプログラミングの構造化が生まれた．

言語や開発手法の発展は，複雑化した仕様を分解し単純化し，実装への正確性を与えソフトウェア開発の発展に寄与してきたと思えたが，実務を通してみると開発の困難は，未だ解決されていない．開発手法の考案は，職人的な方法でのアプローチを強める方向に働き，組織的でも論理的でもなく，まして工学的なアプローチもとられていなかったと考える．“複雑化した仕様によるシステムの完全理解の困難さによる実装の不完全性”の改善には，あまり寄与できなかった．

改善のためには，ソフトウェア開発を工学として捉えるソフトウェア工学としてのアプローチが必要である．IEEE は“ソフトウェア工学”を「ソフトウェアの開発，運用，保守の系統的な方法」と定義している．これは、ソフトウェア開発は，系統的な工学の原則を確立したうえで行われなければならないことを意味している．

しかし，不完全な人間が，本来は完全であるべきソフトウェアを開発することができるのだろうか．この問題の解決には，技術とそれを作業遂行の上で生かすプロセスから始まって,プロジェクトを含む組織の運営の問題やソフトウェア技術者個人の「人間としてのあり方」の問題まで，幅広い領域をカバーしなければこの問題を解決できないだろう．ソフトウェア開発を設計・製造・試験・運用などの工程に分けてプロセ

† 千葉大学/NTT データ先端技術株式会社
Chiba University/NTT DATA INTELLILINK CORPORATION
†† 千葉大学
Chiba University

スを管理するプロジェクトマネジメントも重要になってくる．

1.2 ソフトウェア開発の問題点

ソフトウェア開発の問題を組織の運営の問題や人間性の問題といってすませることができるのだろうか．本稿では，ソフトウェア開発を歴史的に捉えると伴に工学的に問題の所在を考察する．

特にプログラミングの変遷を自身の職務から省察すると，ソフトウェア開発では，プログラミングというプロセスが重要であるが,プログラミングは,下流工程であり，プログラミングを製造，すなわち「誰でもできる単純作業」だと捉えてしまうため，システムの中核であるソフトウェアを最終的に「製造」するプログラミングが軽く見なされ効果的なソフトウェア開発が行なわれていないと考える．

この原因をプログラミングの歴史的背景を知ることから始め，プログラミングを「製造」と考えることが，ソフトウェア開発をいたずらに不合理している点を示し，「詳細設計」として捉えることにより，従来のソフトウェア開発のプロセスの考え方の問題点を明らかにし，効率的なソフトウェア開発の在り方を示す．

また，詳細設計としてのプログラミングとコードを製造するためのプログラミングを並行かつ同時に表現するために必要な条件について示し，その実現可能性について述べる．

2. ソフトウェア開発再考

2.1 プログラミングの歴史

2.1.1 黎明期

1950 年から IBM や UNISYS の前身の会社が，多くのコンピュータを世に送り出しているが，用いられたプログラムは，機械語やアセンブラ言語で作成されていた．機械語やアセンブラ言語でのプログラムは，その機種しか動作しないことを意味した．この時代は約 10 年続いた．このため，当時のプログラマは，コンピュータのハードウェアを理解し操作し，そのための機械語を理解しアセンブラ言語でプログラムし，そして結果を検証するためにメモリのダンプを取るという広範な専門的能力が要求された.コンピュータに対するハードウェア的な知識のほか数学的な知識も必要とされた．これを自動車の運転に例えるなら，メーカや車種が違う車では，その都度運転方法が異なるということを意味した．エンジンの違いにより始動方法が異なったり，ブレーキのかけ方が違ったりすることであり，運転方法を習得するためには，車の構造の理解や操作方法の習得に多くの労力が必要となる．これでは，車の運転が非常に特殊な技術となってしまう．当時のコンピュータは，これに類していたのであった．

作成されたプログラムは，あるメーカのある機種のある処理を行うためのものであり，他への流用や改造が難しかったので能率性や効率性が低かった．同じ目的の処理を違うコンピュータで行うためには，最初から作り直さなければならなかった．従って，この時代のプログラムは作成者以外にとってはブラックボックスであり，プログラミングは,「黒魔術」と呼ばれるほど，プログラマの個人的な技術と才能にかかっていた．

2.1.2 高級言語の誕生

「黒魔術の暗黒時代」を打破したのが，1954～7 年にかけて IBM のジョン・バッカス（John W. Backus，1924 年-）らによって開発された FORTRAN の登場であった[1]．これは，すべての命令を仮想的なコードで書き，機械語に翻訳するものであり，当時の人が自動プログラミングと呼ばれていたものを発展させ完成した．これにより，プログラムの生産性が著しく向上した．バッカスらが書いた論文中には，例として 47 個の FORTRAN の命令文から 1000 個の機械語の命令が生成されたとある．FORTRAN はコンパイル言語であるが，コンパイラのもう一つの効用は，抽象化である．FORTRAN は IBM の 704 用に開発されたものであったが，他メーカのコンピュータでも使えたのである．こうして抽象化されたプログラムは，特定のコンピュータに依存せず有効性を保ちながら使用することが可能となった．FORTRAN プログラムを覚えると，応用が効きプログラミングが魔術から技術へと変わった．ここから，技術教育としてのプログラミング教育が開始されたとも言える．

FORTRAN は科学技術計算に主に用いられたが，その当時，COBOL，LIPS(LISt Processing)などのコンパイラプログラミング言語が作られた．COBOL は事務処理用のプログラミング言語であり LIPS は再帰処理に特徴を持つプログラミング言語であり，のちの構造化プログラミングにつながる．これらのプログラミング言語が同一時代に開発されたのは興味深い．オブジェクト指向は現在のプログラミング環境の主流をなすものであるが，この考え方が生まれた背景には，コンピュータの性能向上と社会的要求により大規模なソフトウェアが作成されるようになり複雑化していったことが挙げられる．「ソフトウェア危機の到来」である．当初，この危機からの脱却のためにソフトウェアの再利用，部品化が行われるようになった．このような流れの中で，プログラムを構成するコードについては手続きや関数といった構成単位にブラックボックスとすることで再利用性を向上し部品化を推進する仕組みが提唱され,「構造化プログラミング」[2]としてエドガー・ダイクストラ(Edsger Wybe Dijkstra，1930 年-2002 年) らによってまとめられた．

コードのブラックボックス化は進んだが，データについては主記憶上に置かれている基本データ型の値の集まりであるという低レベルの抽象化しかできなかった．コードはそれ自身で意味的なまとまりを持つがデータはそれを処理するコードと組み合わせないと十分に意味が表現できないという性質のためである．そこでデータを構造化し，ブラックボックス化するために考え出されたのが，データ形式の定義とそれを処理する手続きや関数をまとめて一個の構成単位とするという考え方で「モジュール」

と呼ばれる概念である．

2.1.3 オブジェクト指向の誕生

しかし，定義とプログラム内の実体が一対一に対応する手続きや関数とは異なり，データはその形式の定義に対して値となる実体（インスタンスと呼ばれる）が複数存在し，各々様々な寿命を持つため，そのような複数の実体をうまく管理する枠組みも必要であることがわかってきた．そこで単なるモジュールではなく，それらの実体を整理して管理する仕組みとしてのクラスとその継承まで考慮して生まれたのがオブジェクトという概念である．

2.2 ソフトウェア開発の問題点

しかし，構造化プログラミング，オブジェクト指向プログラミングを行っても，ソフトウェア開発には困難が伴う．ソフトウェア開発の難しさは，対象が目に見えないところにあり，ビルの建設や自動車の製造のように何か部品をかき集めて組み立てるというより，ひとつの長編小説を分担して書くような作業に実態としては近いと考える．それも大規模なソフトウェアになると大長編小説となり，しかも大勢のメンバで分担して書くため，全体像をとらえることは難しくどんなものを作ればいいのかさえはっきりわからない点にある．また，進捗について開発を行っている当のメンバでさえ今どのくらい出来上がっているかの認識がまちまちだったりする．大規模なソフトウェア開発の問題は，プログラムの全体像を開発者や開発チームのメンバでも内容や進捗が把握できないという形で発生する．

対策としてソフトウェア開発プロセスを改善しようと，ウォーターフォール[3]という段階的な開発モデルや，スパイラル開発型のラピッド・プロトタイピング[4]を適用することなどのプロセス改善の手法が導入されている．これで問題は解決されたのだろうか？答えは，“No”である．プロセスの改善では，問題の改善とはなっていない．こうして，ソフトウェア開発が本質的に抱える課題とは何だろうという疑問が生まれる．

ソフトウェア以外の工業製品の開発においては，各プロセスの活動を経てなされる最終アウトプットは，何らかのドキュメントとなっている．例えば，自動車の開発においても，設計作業が完了すると設計書・設計図が製造工程に引き渡され，製造工程では設計図が実際に製作可能な設計となっていることを確認し製造が開始される．ここでは，前工程である設計者からの影響を受けることなく，その製品（自動車）を大量生産できるようになっている．しかし，ソフトウェアの設計ドキュメントは，プログラミングの開始時点ではまだ完成形ではない．このため，ソフトウェアの設計書は，プログラミング前に記述するのではなく，プログラミング後に記述した方が正確であると揶揄される．このことは著者ばかりでなく，プログラマであれば誰でも経験することである．理由は明らかで，コードに反映された設計のみが，唯一の最終設計であるからである．

3. ソフトウェア開発は工学か

3.1 技術とはなにか

工学は，技術を生み出すための系統的な方法であるので「ソフトウェア開発は工学か？」という議論の前に「技術」について述べる必要があるだろう．筆者らは，技術を「経験を通して獲得した対象行動への優れた適応力」[5]と定義しており，技術を「労働手段の体系」[6]とする相川春喜や岡邦雄のように生産労働に関する作業や活動に限定してはとらえない．また「人間実践における客観的法則性の意識的適用」[7]や「人間の実践的な生産における客観的な規則による形成の判断的過程である」[8]とする武谷三男や三枝博音のような考えは，プログラミングの作業自体を視しているだけで，著者らの悩み(完成された最終アウトプットは何か)としての疑問には答えられない．つまり，「技術」を道具や機械そのものの存在や機能を指すのではなく，優れた最終アウトプット（ソフトウェア）を作り上げる有効なプロセスを遂行する能力を「技術」としてとらえることにより，悩みや疑問に答えることになる．このとき，技術としてのソフトウェア開発は，「各々の開発プロセスを的確に効率的に実行する適応力」と表現できる．こうした意味では，ソフトウェア開発は，前述のように「技術」とはなっていないのである．

3.2 プログラミングは設計工程

ウォーターフォールでもプロトタイピングでもプログラミングを行う前には，上位の設計を完了させ，凍結しなければならない．そして，設計に基づき忠実にプログラミングされ，テスティングは，テスト実施者によりプログラミング時の過ちを除去するためだけに行なわれる．このプロセスを確立し効率的に実施するためにプログラム言語が考案され，言語に適した開発手法が考えられてきたのである．ソフトウェア開発は，ソフトウェア開発の工程をプロセスとして細分化しプロセス毎に実施する内容を定義する方向に進んできたが，プログラミングは，最終製品であるソフトウェアを作り上げるという意味においてソフトウェア開発のプロセスの中核であると言える．

しかし，中核プロセスであるプログラミングを行なうときに「対象行動への優れた適応力」を発揮せず「その場で思いついた改造」や「再設計」という行為を止めれば，ソフトウェア開発は確立した技術となり工学分野として成熟すると信じられてきた．設計プロセスでは，その場で思いついたことを行なう試作や再設計が受け入れられるが，プログラミングでは受け入れられなかったことを意味する．しかし，現実的には，ソフトウェアの仕様書という設計書を上位の技術者が最初から完璧な形で設計できるとは考えておらず，誰も期待していないといった現実がある．これは，どこに問題があるのだろうか？プログラミングを製造プロセスとして定義してきた歴史そのものに齟齬があるのではないか．これには，プログラミングは，設計プロセスであり，製造プロセスでないと考えることが妥当であろう．プログラミングを設計プロセスと考え

るならば,「試作」や「再設計」を行なうことは当然であると考えることができる．結論として,プログラミングは,「設計書に基づきコードを書くという製造プロセスではなく，ソフトウェア開発の詳細設計工程のプロセス」と定義すると矛盾がなくなる．

4. プログラミング言語は鳥の目を持てるか

4.1 プログラミングのアウトプット

スティーブ・マコネルの「Code Complete」[9]によれば，命名規約は，“型定義はすべて大文字とする”,“ローカル変数は大文字と小文字の混成とする”,のほか”複雑な式はテーブル参照に置き換える”,“配列の次元を下げる”,“配列の添え字を long にする”等が,プログラミングを行うための有用なノウハウや守るべき規則とされている．しかし，プログラムに必要なことは，言語の文法を理解し効率的なコードが書けるようにすることではない.なぜなら,プログラミングを詳細設計プロセスと考えた場合，プログラミングのノウハウや規則はもちろん必要であるが，それだけでは十分ではないからである．前述したように，技術とは最終的な製品がどのようなものになるのかということではなく，どのようにプログラミングというプロセスを実践するのかということである．

前述したように，どのような工学設計プロセスでも，その目標は何らかのドキュメント作成になる.プログラミングのプロセスの目的は,ソースコードの作成ではなく，システム全体を鳥瞰できる詳細設計書としてのドキュメントが必要となる．このドキュメントは，重要なものは，補助的情報としてコードに直接組み込まれることはないが，目的や考慮すべき点についての重要な情報を記述することである．これは，後でモデルを変更する場合に必要となるものである．また，設計自身から直接抽出することが難しい側面を図表により表現し，ドキュメント化しておく必要もある．

このことは，ソースコード作成と同時に，一番良いのは，ソースコードの中に詳細設計書，補助的情報を埋め込むことを意味する．

4.2 設計コンセプトを実現できるプログラミング言語

しかし，詳細設計とプログラムコードの作成の双方に適した記法は，存在していないので，詳細設計や補助的情報に相当する部分は，プログラミング言語で記載されることになる．ここに矛盾が生じる．ビルや機械の設計図面をすべて文字で表すようなことをソフトウェア開発では行なっているのである．

「プログラミングは設計である」が理解されたとしても，その詳細設計時（ここでは，プログラミング時）には，設計記法によって書かれた結果をプログラミング言語に変換しなければならない．現在の進んだといわれる開発環境やモデリングを使用したとしても，選択したプログラミング言語とうまく対応付けられていない設計記法から変換を行うような場合には，プログラマは要求レベルにまで遡り，上位レベルの設計をやり直し，それに従ってプログラミングを行うこともしばしばである．これはソフトウェア開発の現実であり，解決には，すべての設計レベルに適切で共通な統一記法が必要である．換言すれば，上位レベルの設計コンセプトを実現できるプログラミング言語が必要とされる．しかし現時点では，詳細設計とプログラムコードを同時に表せるような設計記法は，存在していないので，詳細設計は何らかのプログラミング言語でコード化されなければならない．

この問題に対しては，視覚的に設計，プログラミングといったプロセスをレイアーとして重ねることができる表記法により解決できる糸口があると考えている．

5. まとめ

情報システムは発展し社会の基盤となっているが，巨大化し階層化された複雑なシステムである。情報システム構築の中核をなすソフトウェア開発においては，なお課題が残されている．しかし、プログラミング言語、ソフトウェア開発の歴史の変遷をたどることによりソフトウェア開発プロセスの問題点を示し，この問題の解決については，プログラミングを設計プロセスとして定義し、詳細設計とプログラムコードを同時に表せるような設計記法を実現するためのプログラム言語として実装されなければならない機能について一つの方向を示したが，今後は、実際に言語の実装設計について報告したい．

参考文献

1) E．W．ダイクストラ，C．A．R．ホーア，O．-J．ダール，野下浩平訳，川合慧訳，武市正人訳:『構造化プログラミング』，サイエンス社，1975年
2)パターソン＆ヘネシー，成田光彰訳：『コンピュータの構成と設計』，日経BP社，1996年
3) Jr.,フレデリック・P.ブルックス，滝沢徹，富沢昇，牧野祐子訳：『人月の神話―狼人間を撃つ銀の弾はない』，ピアソン・エデュケーション，2002 年
4) スティーブ・マコネル，日立インフォメーションアカデミー訳:『ラピッドデベロップメント―効率的な開発を目指して』，アスキー，1998 年
5) 佐藤建吉:「技術」についての一考察（技術の定義とその意味），日本機械学会・技術と社会部門，技術と社会の関連を巡って：過去から未来を訪ねる講演会論文集，2008 年
6) 佐野正博：技術の歴史的発達過程と法則性，東京農工大学一般教育部紀要，第 25 巻 1988 年
7) 武谷三男:『弁証法の諸問題・武谷三男著作集』第 1 巻，勁草書房，1968 年
8) 三枝博音:『三枝博音著作集』第 8 巻，中央公論社，1972 年
9) スティーブ・マコネル，クイープ訳:『Code Complete 第 2 版〈上・下〉―完全なプログラミングを目指して』，日経 BP ソフトプレス，2005 年