2009－SE－163 (31)
2009／3／19

システム開発の理論と実際

巫 少鴻
コーナンソフト

二十世紀に登場した新技術の中で，社会のあり方に革命をもたらしたものの１つにコンピューター関連技術があるが，この総合的な技術系が生み出す付加価値においては，ソフトウェア製品の占める割合が圧倒的に大きい．当然，ソフトウェアの開発を安定的に行う方法論が模索される．ソフトウェアは，自らの性格ゆえに，大きな潜在力と可能性を秘めている反面，一般の工業製品のように，品質を保証して大量生産することは困難である．その原因については，多くの考察が提案されてきたが，ソフトウェア開発に関する基本的な誤解が解決を妨げてきたと思える．本稿では，Barry W. Boehm と Jack W. Reeves の考察を比較しながら，本質に迫りたい．

Theory and Practice of Software System Development

Wu Shaofung
Konansoft

A computer-associated technology is certainly one of new technologies since twenty century, which brought revolution to the ideal way of the society. And in value added this general technology generates, a proportion of that of software product is fundamentally large. So methodologies have been inevitably sought to stably develop it. Though software has its peculiar wonderful nature of great possibility and potential, probably due to the same nature, it is difficult to mass produce software product of high quality like a general industrial commodity. A basic misunderstanding concerning nature of the software development seems to have prevented the solution about this issue though a lot of consideration has been proposed. In this paper, I want to approach essence of it while comparing consideration of Barry W. Boehm and that of Jack W. Reeves.

1. 初期の開発方法の展開と限界

コンピューターの関連技術が進歩し，コンピューターシステムが多方面における基盤として重要な役割を担うようになるにつれ，ソフトウェアシステムは，社会の必需品となったが，同時にハードウェアの製作よりもソフトウェア開発がコンピューター産業のボトルネックになった．そして，その安定した製作技法を与える有効な解が得られないので，ソフトウェア危機が意識されることになった．このために，現在でも，特に顧客の要求の可変度の大きな業務用のソフトウェア開発ないしシステム開発は，当初予算の超過，予定期日に対する完成日の遅れ，および開発結果の機能の当初計画に対する不満足という危険性あるいは不安定性の下にある[1]．

1940 年代の原型の登場後，1950 年代に第一世代メインフレームが実用化されたころ，高価稀少なハードウェア資産に適合させる必要から，ソフトウェア作成，そのほとんどを占めるプログラミングは，難解で専門的な作業だった．FORTRAN や COBOL などの「高級言語」が登場した 1950 年代に，個別プログラムではなく，継続的に稼動するソフトウェアシステムが軍事システムとして制作され，やがて大規模な民間企業の事務処理などにも採用されるに至り，その製作を安定的に行

[1] Gary Richardson and Blake Ives: Managing Systems Development, Computer March 2004, p.93

う方式が模索されはじめた．今日，ウォーターフォール・モデルと呼ばれている開発モデルは，このときの必要性に応じて発案，採用された方式を後に命名したものであるといえる．

初期のプログラミングは，そもそも習得が困難なものとみなされており，従事する人員も少数であったため，問題の解決は個別的に志向され，問題解決の経験を知識として体系化し，共有するような環境は存在していなかった．このような背景でのプログラミング，あるいはソフトウェア開発では，プログラムの作成が直接に行われたが，この状況が続くと，同じシステムの使用が定常化してプログラムが多様な要素を吸収し，規模が大きく成長するにしたがい，手に負えないソースコードが生み出され，プログラムのメンテナンスに困難が生じ，その費用あるいは手間は，予測される範囲を遥かに超えるものになる[2]．

そこで，ソフトウェア開発・保守が窮地に陥ることを防止しようと，何らかの措置が講じられることになる．たとえば「標準化」によって，プログラミング作業を規制し，あるいは開発の目的であるソフトウェアシステムに関する総合的な情報を集約する局を設け，開発チーム（プロジェクト）の進捗を統括管理する方式が採用された．このとき，開発の進行工程が分析され，概ね，要件定義，設計，プログラミング，テスト，および保守という工程が提案された．ソフトウェアのライフサイクルとも言われる，この工程の区分は，現在もほとんどの情報処理技術者や学者によって，自明のものとみなされている．ウォーターフォール・モデルは，この工程が列記している順序で，逐次的に発生することを前提として，各工程を完成させてから次工程に進むというプロセス・モデルであり，後に提案されることになる他のソフトウェア・プロセス・モデル(software process model)の基本となった．

プログラムの「標準化」は，共有化されない個々の技術者の問題解決法に共通性を保持させることを目的に，コーディング規約などの制約を設け，従わせようとするものである．構造化プログラミングというような，プログラミングの性質を，比較的，深く見通して適用される基準もあるが，機械的に強制された雛形がネガティブに動作し，生産物の質を劣化させる結果になることもある[3]．

また，ライフサイクルの工程を上記の順序で行えるという仮定に従い，大規模なプロジェクトを進行している場合に，その前提が崩れ，後の工程から前の工程に戻らなければならなくなった場合には，プロジェクトは非常に大きな損害を被ることになる．ウォーターフォール・モデルでは，各工程間の完全な独立性への疑問が留意されており，フィードバックの重要性が指摘されている．しかし，トップダウン方式と情報，管理，および意思決定の集中，一元化を特徴とし，その特徴のゆえに効果を期待されるこのモデルに，あとづけでフィードバックという機能を提案し，その効果を期待することは，原理的に無理があると思える．実際に，フィードバックという処理は，安定した手順として力を発揮するよりも，プロジェクトに参加している要員の個人的な倫理や発奮を喚起する役割を果たすものになる．これは，開発にかかわる未解明の問題から生じる負担を個々の人員に転嫁する働きであり，ソフトウェアシステムの社会的な利用の環境が整うに従い，作成されるシステムが複雑化し，巨大化すると，そこから発生する問題が，人間の能力の容量を超えるので，根本的な見直しを迫られるにことになった．

2. 批判，対案，その限界

ウォーターフォール・モデルが基本的な難点を抱えていることが経験的に認識されると，その克服が試行された．このときにも，ソース・コードを有効に管理できるようにするために，プログラミング資産の共有を可能にする努力が継続されたのはごく自然である．これは，ソースコードの標準化，プログラミング環境の整備，プログラム言語の改善などであるが，ここでは総称的にプログラミン

---

[2] Barry W. Boehm : A Spiral Model of Software Development and Enhancement, Computer May 1988, pp.61-63
William Roetzheim: Programming Windows with Borland C++ 4.5, p.23, Ziff-Davis Press (1994)

[3] たとえば，プログラムで使用する変数名の不合理で煩雑な命名規則，コーディングの体裁，特にカラム位置のインデントについての度を越えた制約，特定の命令や処理系の恣意的な禁止などがあり，さらには，プログラミングのもっとも自然な作業本体を規制するだけでなく禁止するといった極端な介入もみられる．

グ・パラダイムの改善と呼ぶことにする．他方，実際のソフトウェア製品を何らかの社会的な必要性から製作する場合に，その製作を束縛する開発期間，費用，人員の制約との関連で，ソフトウェア製作を予定どおりに完成させるための管理の視点からの試行があった．ここでは，この方向の努力をプロセス・モデルの改善と呼ぶ．両者は同じ目的を目指す努力であり，また後者は前者を通してのみ正常に機能するのだが，実際にはその相互の協力的な効果の発揮は難しく，漠然と別個の要因と認識され，実際の開発過程においては異なる視点として対立しがちである．本稿では，プロセス・モデル改善の努力の大筋を辿り，プログラミング・パラダイム改善の努力の視点との隔絶点を眺めながら，ソフトウェア開発に関する基本的な誤解と思えるものを考察したい．

1988 年に Barry W. Boehm は，"A Spiral Model of Software Development and Enhancement" という論文で，旧来のソフトウェア・プロセス・モデルを概観し，それらのモデルを特殊例として包摂する新しいプロセス・モデルとして「スパイラル・モデル」を提案した．Boehm は旧来のモデルを，code-and-fix モデル，stage-wise モデル，ウォーターフォール・モデル，evolutionary development モデル，および **transform** モデルと類別または例示した．最も古い code-and-fix モデルは，ソースコードを書いてから，要件，設計，テストおよび保守について考えるもので，コードにある程度の修正が行われると，コードの保守が非常に困難で工数を要するものになっていく事実から，コーディングの前に設計工程をおく必要性が確認され，また，作成された結果が顧客にまったく受け入れられない場合があるので，設計の前に要件定義の工程をおく必要性が確認されるとする．この点を考慮して案出された stage-wise モデルは，それらの工程を順次，完了していくモデルであり，それにフィードバックを加味したものがウォーターフォール・モデルである．Boehm は，フィードバックの留意にかかわらず，ウォーターフォール・モデルには基本的な問題があり，その過度のドキュメント重視と，ユーザー・インターフェースのサービス機能の貧弱さを指摘する．彼は他のモデルについての限界を指摘した後に，スパイラル・モデルを提案し，旧来のモデルの困難を解消すると主張しているが，根拠についての記述は必ずしも明瞭ではなく，論理的齟齬も見られる．彼はスパイラル・モデルを risk-driven アプローチであると定義し，旧来のプロセス・モデルを document-driven および code-driven であったと批判するのだが，リスクの評価方法については，特別な基準を提案せず，従来のあらゆる可能な方法を用いると書く．また，スパイラル・モデルはウォーターフォール・モデルの洗練によって導かれたものであり，ウォーターフォール・モデルを含む旧来のモデルが失敗する理由は，ステージの実施順序を誤っていたことにあるとする．

しかし，リスクを評価しながら予備調査を入念に行い，すべてのリスクが解消されてから，ソフトウェアの製造段階に入るという説明は，ウォーターフォール・モデルにおける要求仕様の確認工程を最大限拡張して，慎重に行うという方法に帰着しているように見え，その意味ではプロセスの移行は螺旋的ではなく，逐次的ではないか．また，リスク評価が確定すれば，状況に合わせて旧来のモデルのいずれかを選択し適用することになり，それゆえに新しい一般モデルが旧来のモデルを特殊例と包摂するとしているのだが，旧来のモデルによってもたらされるとする弊害，たとえば，ソースコードが複雑化し，再利用や改造に多大の費用が発生するといった問題や，ドキュメントを過大に重視しているために，膨大な意味のないドキュメントが作成され，それを維持していかなければならなくなるというような問題をどのように解決できるのかについては，触れていない．彼はスパイラル・モデルを適用した実際例を紹介しているが，それはソフトウェアシステム開発（または改善）を検討する顧客と請負元の両方の立場を兼備する視点からの検討例であり，ソフトウェアシステムが産業的に企業によって製造される必要性を考慮する場合の事例としては，現実味が薄いと思える．さらに，入念な予備調査を繰り返し行い，その中で新たなスパイラルが並行的に発生する可能性を述べ，柔軟なプロジェクトの発展を許容しながら，すべてのリスクが解消されてから，Creation prototype つまり製造工程に進むとしているが，これは要求仕様の確定であっても，後の工程から独立しては行えないのではないかという，ウォーターフォール・モデルに対する深い疑問

を素通りしているかに見える[4].

ウォーターフォール・モデルの開発方式に対する有力な批判として注意すべき，もう1つの方法論にアジャイル・ソフトウェア開発方式がある．これは，多様な観点を包含した複雑な方法論であり，単純にプログラミング・パラダイムの改善とプロセス・モデルの改善のいずれかに分類することはできない．しかし，その方式の核心部分として理解されている，プロジェクト管理に関する方法論，つまりプロジェクトの規模や期間を適度に細分し，職能を超えた要員間のコミュニケーションを促進し，レビューを頻繁に行い，知識を共有化し，プロジェクトが重厚長大になり，事態の変化に対処できなくなるために被るリスクを回避しようとする方法論は，後者の改善にあたると思える．Victor Skowronski は，アイザック・ニュートンやトマス・アキナスのような疑問の余地のない最高の能力をもつ人材が，この手法を適用されたグループに所属した場合，彼らは同僚や管理者の低い評価を受け，能力を発揮することはできないだろうと指摘するが，前述のようなプロジェクトのあり方にはSkowronski が指摘する弱点があることは，留意すべきである[5]．かなり乱暴な言い方になるが，プロセス・モデルの改善の提案は，ウォーターフォール・モデルに対する批判として説得力があるが，理論的，一般的にウォーターフォール・モデルに置き換わる方法論を提供するほどには強力でなく，代替的な方法論と確立されるまでにはなっていない．実際の開発プロジェクトの多くは，多かれ少なかれ，ウォーターフォール・モデルの特徴を強く残す方法で運営されている．その理由はなぜだろうか．

1950 代以降の経験で，ウォーターフォール・モデルと表現される開発方式が採用されたときに，この方式がソフトウェア・プロセス・モデルの可能な方式の一つでしかないとの認識があったのだろうか．ソフトウェアを組織的に開発するというはじめての経験の中で，いくつかの可能な方式から開発方式を選択したというのは，現実的にありそうもない．そうではなく，産業構造に関する当時の，そして今日でも保持されている，ある種の前提に基づき，最善の知識を総合してこのような開発形態が採用されたとする考えるほうが自然である．これがソフトウェア開発の1つの選択肢であるとする認識は，この方式に難点が指摘され，その難点を克服するための手当てが必要になり，追及されるにいたった後にしか発生しえないだろう．つまり，ウォーターフォール・モデルによるソフトウェア開発方式は，プロセス・モデルの提案が先にあったのではなく，ソフトウェア開発の実践とそこから発展した開発組織が初めにあったのであり，これに対し，この方式を批判して提案されたモデルや方式は，現実の開発方式への批判的な検討によって，新たに論理的に創案されたものであるといえよう．この点を踏まえれば，新たに提案されたモデルや方式が，実際にすでに採用されている方式に比して，具体性，一般性あるいはその両方を欠いているとしても無理のないことである．しかし，批判が対象の根本的な問題にまで迫り，その次元での新たな展望を示さなければ，批判から生み出される方法論が伝統的な方法論に置き換わることはできない．その意味での問題の解は，プロセス・モデルの改善とプログラミング・パラダイムの改善の関係をつきつめて理解を進めなければ得ることはできない．

3. プログラミング・パラダイムの改善とコンピューターシステム関連環境の発展

ウォーターフォール・モデルが，大きな批判を受けずに実施された1950年代から1970年代までの期間が，コンピューターの第一世代ともいうべき，メインフレーム・コンピューターを中心とするシステムの誕生と成熟の時期であるとすれば，1980年代以降のコンピューターを中心とする技術的な発達は，第一世代のコンピューター技術の基本通念を書き換えさせるほどの，急激で大掛かりなものであった．プログラミング・パラダイムの改善は，コンピューター関連技術の実用化の時期

---

[4] Daniel D. McCracken and Michael A. Jackson: Life Cycle Concept Considered Harmful, ACM SIGSOFT SOFTWARE ENGINEERING NOTES, Vol. 7, No 2, April 1982, p31
完全な要求仕様を事前に確定することは，原理的にみても，可能ではないと述べられている．

[5] Victor Skowronski : Do Agile Methods Marginalize Problem Solvers?, Computer October (2004), pp.118-120　pp.37-38

から，継続的に行われてきているが，それによって得られる成果は，コンピューター関連環境の急激な発展によって促進された面もあり，またそれによって発生するコンピューター関連環境におけるユーザー・インターフェースやプログラムなどの複雑化によって，概ね相殺されてきた観もある．それでも，1980年代から1990年代にかけて，オブジェクト指向として，有力なプログラミング・パラダイムの改善が提案され，この提案は多くの注目と期待を集めた．今日，オブジェクト指向は情報処理理論の一分野に取り込まれているが，登場した当初に提起されたその意味に関する考察は形骸化し，失われているように見える．ここでは，オブジェクト指向を代表する言語として注目されたC++言語に関する独自の考察を行っているJack W. Reevesの言葉を借りて，二つの方向の改善の接点にある，ソフトウェア開発の性質に関する先験的な誤解と思えるものに迫りたい．

4. ソフトウェアの設計とは何か

Boehmのスパイラル・モデルでは，ウォーターフォール・モデルで前提にされているプロセスあるいはステージの概念を承継し，プロセス・モデルの失敗の原因はプロセスの実行順序を誤っているからであるとした．彼がソフトウェア・プロセス・モデルという方向からソフトウェアシステム開発を再検討した1980年代に，オブジェクト指向プログラミングが提唱され，1980年代末には，ソフトウェアの世界に激震をもたらしていた．オブジェクト指向という言葉は，アメリカで注目され，魔法の言葉のようにさまざまな製品の広告コピーにも使用されたという[6]．1992年にC++ Journalという，現在の日本では入手が困難になっている雑誌に"What Is Software Design?"という論文を書いたJack W. Reevesは，この論文の中で，なぜ，突然このパラダイムが，これほどまでに注目されたのかを分析し，実際にはさまざまな要因が組み合わさって起こったことであろうとしながら，一般的に信じられている理由として自ら列挙したものとは異なる角度からの回答を示唆する．つまり，C++が支持されるようになったのは，これがソフトウェアの設計とプログラミングを同時に行うことを容易ならしめたからだと指摘する．ついで彼は，論文の基本テーマである視点を述べる[7]．

> After reviewing the software development life cycle as I understood it, I concluded that the only software documentation that actually seems to satisfy the criteria of an **engineering design is the source code listing**(私が理解している限りでのソフトウェアの開発ライフサイクルを検討した後に，私はエンジニアリングでの設計の基準を現実に満たすと思える唯一のドキュメントはソース・コード・リストであるという結論に達した．[翻訳筆者，以下同様]).

この主張は，Boehmが説き起こしているソフトウェア・プロセス・モデルの考え方と基本的な部分で対立する．Boehmは，初期のプログラミングの開発方法を支配していたプロセス・モデルをcode-and-fixモデルと呼び，このモデルによる開発では，作成されたプログラムが急速に修正困難な，手に負えない状態になるという事実から，コーディングの工程の前に設計の工程の必要性が理解されるようになったと書く．ウォーターフォール・モデルにおける過重なドキュメント重視の弊害は認めるにしろ，ソフトウェアの開発の前段階に，製造するソフトウェアに関する情報を整理して記述するドキュメントを作成することは，否定されるべきことではない．歴史的にもそのようなドキュメントがソフトウェアの製作によい効果をもたらしてきたことは，明らかである．伝統的に

---

[6] William Roetzheim: Programming Windows with Borland C++ 4.5, p.24, Ziff-Davis Press (1994)

[7] Jack W. Reeves: What Is Software Design?, first appeared in 'C++ Journal, Fall of 1992 issue', **internet site developer* p.1** http://www.developerdotstar.com/mag/articles/PDF/DevDotStar_Reeves_CodeAsDesign.pdf

このような工程を設計と呼んでいるのではないか.

しかし, Reeves の主張は単なる用語の意味へのこだわりではない. 彼の問題提起を,「設計」の定義は何かという問題に収斂させることは適切ではない. Reeves は, 現実に設計といわれ, 行われている工程が設計であるというような職能説的な観点で語っているのではなく, ソフトウェアシステム開発の方法論に要求されるある属性に沿って, 設計という概念を使用している. 彼は, engineering と craft という二つの生産形態を対立させ, ソフトウェア開発は現実には後者の形態で行われているが, 前者の形態で行わなければならないという前提の下に設計概念を導く. 前者は日本語の用語では「工学」と訳され, 後者は「技芸」,「工芸」,「手工業」などという用語が対応すると思われるが, ここでは, 機械的な単語の割り当てによる概念の誤った理解を避けるために, カナ書きで,「エンジニアリング」,「クラフト」と書くことにする.

ソフトウェア開発をエンジニアリングとして行うべきであるという彼の前提は, 彼が独自に主張しているものではない. ソフトウェアの開発技法を巡るソフトウェア工学の議論やプログラミング・パラダイムの議論は, ソフトウェア危機の認識を受けて発生しているものであり, Reeves の前提はすべての論者の共有する立脚点であるといえる. したがって,「エンジニアリングでの設計の基準を満たす」という彼の限定詞も共有されるべき前提であり, これを任意に取り除くことは許されない. 前に引用した彼の結論の文の直前に, 彼はエンジニアリングを前提としたときに, 設計がいかなる機能を持たなければならないのかについて, 次のように書いている.

> The final goal of any engineering activity is the some type of documentation. When design effort is complete, the design documentation is turned over to the manufacturing team. This is a completely different group with completely different skills from design team. If the design documents truly represent a complete design, the manufacturing team can proceed to build the product. In fact, they can proceed build lots of the product, all without any **further intervention of the designers.**(エンジニアリングの作業の最終目的は, 何らかのドキュメントである. 設計の作業が完成したときに, 設計書は製造部門に渡される. これは設計部門とはまったく異なる技術を有する, まったく別の人々の部門である. 設計書が真に完全な設計を表すものであれば, 製造部門は製品の製造に進むことができる. 実際に, 製造部門は, 設計者がまったく関与しないままに, 多くの製品を製造することができている.)

Reeves は, ソフトウェア工学以外の工学, たとえば, 機械工学や建築学などの工学になぞらえて, 他の工学において設計という工程が占めている役割と位置を整理し, 設計という概念を導き出している. そして, その設計の概念の基準で評価した場合に,「設計の基準を満たす唯一のドキュメントはソース・コード・リストである」と結論付ける.

前述のように, 私は, ソフトウェア産業の開発方式つまりプロセス・モデルが, ハードウェア産業になぞらえて構築されたのは, 1950 年代以降のコンピューターシステムの発展に応じた当然の帰結であったと考える. それは, Boehm のプロセス・モデルの概観を眺めても確認できる. しかし, ソフトウェア開発がハードウェアの開発と同じ基準によって効率化でき, 生産性を向上することができるのか否かという問題は, そもそも自明のものではない. 現在では, ソフトウェア開発は, 産業として企業によって担われている. だが, 営利企業が産業として担う生産物で, しかも産業の全体規模が相当程度大きなものであっても, すべての生産物を同じ原理で効率よく生産することができるわけではない. たとえば, 映画産業は日本でも, 外国でも, 相当程度の規模がある産業であり, 製品は営利企業により製作される. しかし, 映画という製品の生産性については, 一般の工業製品に適用できる工学原理は基本的に当てはまらない.

Reeves は前掲論文の中で, 何年か前にソフトウェア開発がエンジニアリング (engineering discipline) であるのか否かを問うセミナーに参加し, このときに, ソフトウェア産業界が全体としてハードウェアのエンジニアリングとの誤った対応付けをしているという考えを深めたと書いてい

る．彼の言う「誤った対応付け」を，ごく単純化して図式的に表現すると，以下のようになる．

ハードウェアのエンジニアリングに「設計」，「製造」と定義されている工程がある．ソフトウェア産業界では，ソフトウェアの開発工程にも同じような工程があり，プログラミングあるいはコーディングが「製造」工程に対応し，その前工程としてソフトウェアの「設計」工程が存在すると想定している．Reeves はこの対応付けは誤りであり，仮に，ソフトウェアの開発をハードウェアの開発になぞらえることができるとすれば，ハードウェア開発のエンジニアリングで設計工程に対応するものはプログラミングあるいはコーディングであり，製造工程にあたるものはソフトウェアの開発ではコンパイル・リンクであるとする．

この対応関係の誤解は，単なる工程の理解のズレとして看過できない．なぜならば，この誤解から，ソフトウェア開発における多数の誤った手法が導き出されると思えるし，逆に，ソフトウェア開発において，疑問をもたれながら正すことのできない多くの手法の誤りは，この対応づけの失敗から見直してみると，その意味を容易に解釈できるのである．その点の鮮明な理解を得るために，問題をソフトウェアの開発というテーマから，近代的な産業構造に由来する生産工程に関する先入観にまで遡り，Reeves の問題意識を再検討してみたい．

5. 分業，機械制大工業による労働生産性の飛躍的増大の経験

今日の産業社会は，18 世紀後半にイギリスを中心とするヨーロッパ諸国で起こった産業革命の経験にもとづく生産様式によって，構成されている．アダム・スミスは『国富論』の冒頭に近い部分で，小規模なピン工場における，分業による労働生産性の驚異的な増大について書いている．機械や分業に頼らずに職人が一人でピン作りを行った場合，職人は一日に一本のピンを作ることもできないだろうが，製造作業をいくつかの工程に分け，分業によってピンの製作に当たれば，一人当たりのピン製作量は数千倍になる．実際に，スミスが視察した小さな仕事場では，十八ほどの工程に細分されたピンの製造工程で，十人ほどの作業員がいるだけであり，何人かの作業員はいくつかの工程を兼務していた．ピンの製造作業は，あるものが針金を引き伸ばし，次のものがそれをまっすぐにし，次のものがこれを切断するというような工程に細分されていた．その作業場では，精出して働けば一日に 12 ポンド，本数に換算すると 48000 本のピンを製造することができる．一人当たりの労働生産性は 4800 本となり，職人が一人で作業する場合のそれの 4800 倍になる[8]．

スミスのピン工場の例は設計工程に触れてないが，当然のこと，ピンの原材料の選択，ピンの形状や大きさの指定，分業工程の分析・立案および，分業に従事する作業員の作業方法の指導などの作業が存在していたのであり，このような作業が設計工程に該当するものである．現代の大工業生産様式では，機械の導入による熟練労働の単純化と生産行動の濃密化などにより，飛躍的な労働生産性の増大を実現し，均質で良質の工業製品を大量に供給することを可能にした．それにより，消費者の生活水準が向上するとともに，企業は莫大な利潤を獲得することができた．Reeves がエンジニアリングとクラフトを対応させるときのエンジニアリングの形態による製造の意識の背後には，この近代的な工業生産を背景にした生産方式があることには疑問の余地がない．クラフトによる製造とは，大工業生産にたよらず，職人の熟練労働によって製造する方式といってもよいだろう．

つまり，エンジニアリングの製造形態によって実現されるものは，労働の生産性の飛躍的な増大によって保証される，均質で良質の大量の生産物の供給，それによる消費者への利便の提供，そして企業が得る巨大な利潤である．反面，大工業生産は，熟練労働を排除し，労働過程を単純化し，労働者を彼らの意思や人格を離れた，自立的な生産システムに包摂し，隷属させる結果を導く．

ソフトウェアの開発について，暗黙の前提として共有されているこの生産方式が適用されたのは，前述のようにソフトウェアが，映画や音楽などの娯楽製品ではなく，社会的な必需品となり，ソフトウェア産業が基幹産業の様相を帯びるようになった進展の結果であると思える．多くのソフトウェア開発企業が求めてきたのは，労働生産性の向上であり，それにもとづく利潤の獲得であった．

---

[8] アダム・スミス，大河内一男他訳: 国富論 I．中公文庫，1978 年，pp.10-13(Adam Smith, “Wealth of Nations”, 1776)

事実，効果の程は不明であるが，これまで労働生産性を向上させるために，多様なツールが考案，採用されてきた．そして，労働生産性を向上させるために，経験的に採用する手立ては，熟練労働の単純化，分業，労働者の生産工程への包摂といったものであった．これらの措置は，基本的に期待される効果を実現できなかったといえる．

大工業生産方式で労働の生産性を増大させる理由は，機械の導入を除けば，生産過程の分析による複雑な熟練労働の工程分化と単純労働化，それに伴う労働の集中があるが，ソフトウェア開発にも同じ手法が適用されてきた．ここで，プログラミングツールや言語の進歩は，工業化における機械の導入になぞらえることができるかもしれない．すぐれた設計思想に基づくプログラミング言語の登場は，確実にプログラムの生産性を高めたことは疑う余地がない．しかし，プログラミング過程の分析と，プログラミング作業の単純労働化は，それを実現するための管理の強化と共振して，ソフトウェア開発の現場に無用の混乱をもたらし，開発の作業を理不尽に抑制することによって，逆の効果をもたらしてきた観が否めない．

なぜ，これらの努力が実を結ばなかったのかについての解答は，Reeves の指摘の中に求めることができる．労働の単純化によって労働生産性を増大させることができるという逆説的な事実は，製品製造における作業の客観的な分析により，複雑な作業を単純な複数の製造工程へ分化し，工程全体を中央で管理することによって実現した．ソフトウェアの開発においてはどうだろう．Reeves の洞察に従えば，分業によって労働を単純化すべき製造工程に該当するものは，ソフトウェア開発ではコンパイル・リンクによるビルトである．この工程は，コンパイラーやリンカーが与えられている限り，ほとんど費用を生じさせないものであり，労働生産性を改善する余地はない．プログラミングやコーディングについて，同じ手法を適用してその生産性を高めようとする試みはどうだろうか．プログラミング作業を客観的に分析し，それをいくつかの単純な要素に分割し，複数の未熟練労働者による分業システムを作り上げることができるだろうか．プログラミング言語によるプログラム表現の可能性は非常に多様であり，プログラミング自体の質を低下させずに，それを少数の類型の中に整理して，プログラミングを単純化することは，ほとんどできそうもない．他方，プログラミング言語の改善や OS の機能強化により，確かにプログラミングの生産性は向上したが，それらの生産性増加は，ユーザーの要求の複雑化により，基本的に相殺されてきた．このように，ソフトウェア開発をエンジニアリングとして実施し，工業製品の製造工程になぞらえるとしても，産業革命以降，一応は成功した大工業的な生産方式による生産性の向上を同様の論理で期待することはできないのである．Reeves の洞察は，この事実を明らかにしたという点で，深い意味がある．

参考文献

[1] 有沢誠：岩波コンピュータサイエンス　ソフトウェア工学，岩波書店(1988)

[2] Brroks, Frederick P.: No Silver Bullet: Essence and Accident of Software Engineering., Computer, Vol. 20, No. 4(April 1987)

[3] Gary Richardson and Blake Ives: Managing Systems Development, Computer March 2004

[4] Barry W. Boehm : A Spiral Model of Software Development and Enhancement, Computer May 1988

[5] William Roetzheim: Programming Windows with Borland C++ 4.5, p.23,　Ziff-Davis Press (1994)

[6] Daniel D. McCracken and Michael A. Jackson: Life Cycle Concept Considered Harmful, ACM SIGSOFT SOFTWARE ENGINEERING NOTES, Vol. 7, No 2, April 1982

[7] Victor Skowronski : Do Agile Methods Marginalize Problem Solvers?, Computer October (2004)

[8] Paul Kimmel: Special Edition Using Borland C++5, p.36, Que Corporation (1996)

[9] Jack W. Reeves: What Is Software Design?, first appeared in 'C++ Journal, Fall of 1992 issue',internet site developer*
http://www.developerdotstar.com/mag/articles/PDF/DevDotStar_Reeves_CodeAsDesign.pdf

[10] アダム・スミス, 大河内一男他訳: 国富論Ⅰ. 中公文庫, 1978 年, (Adam Smith, "Wealth of Nations", 1776)