Vol.2011-SE-171 No.21
2011/3/15

Jack W. Reeves の仮説の理論検証と仮説によって得られる新たな展望

巫　召鴻†

1992 年に Jack W. Reeves は，ソフトウェア開発におけるソフトウェアの設計に関する鋭い洞察を示す仮説を提案したが，その原理の導く結果はソフトウェア開発の産業界に重大な影響を及ぼすので．Reeves の提案の検討を徹底的に推し進めようとする研究が成り立ちにくかった．本稿では，彼の仮説の理論的な検証により，仮説の正しさの論証を試み，それを前提とした場合のソフトウェア開発方法の展望について考察する．

The theory inspection of the hypothesis of Jack Reeves and the new prospects provided by the hypothesis

SHAO-FUNG WU†

In 1992, Jack W. Reeves suggested a hypothesis to show the sharp insight about the design of the software in the software development. However, it has been hard to have studies which has examined his suggestion fundamentally, because an impact of this theory was so serious on IT industry. In this paper I try proof of the correctness of his hypothesis by the theoretical inspection of it and consider the prospects of the software development method of the case assuming it.

1. はじめに*

筆者は 1980 年以降，情報処理産業の末席を汚し，いろいろなソフトウェア・システム開発プロジェクトに参加してきた．その経験において，プロジェクトの成功例も失敗例も見てきたのであるが, 1990 年代以降, 深刻な失敗例が目に付くようになった．深刻とは，プロジェクトの失敗の原因がプロジェクトのメンバーや管理者に理解されず，失敗が克服されないままにプロジェクトが停止し，消滅してしまうという現象が発生するようになったということである．その原因は一つではない．たとえば，ソフトウェア開発の産業構造とそれに起因する契約形態にも問題がある．企業や官公庁の大規模な業務システムの発注・受注は，大手ゼネコンを最上位にピラミッド化されてきた建設業界の産業構造のあり方の影響を受け, 大規模な情報処理企業･資本の傘下の企業が元受けし，そこから幾重もの下請け･孫受け･ひ孫受け等を通し，人材の提供企業により，現場のプログラマーが集められプロジェクトが成立する．この状況では，プログラマーやシステムエンジニアがプロジェクトに参加して能力を発揮する機会が不安定になり，ソフトウェア開発に関する方法論の理論的な検討以前に，ソフトウェアに関するもっとも基本的な技能が現場で共有されず，実務を通して質の高い技術者が育成される機会がない．失敗については，損益計算の角度で検討されても，開発技術や方法論の角度からの検討を行う余地は失われる．失敗が技術革新を生み出す可能性は皆無に近い．

20 世紀後期以降を鳥瞰すると，ソフトウェア開発の方法論は，他のいかなる技術系も比較できないほどの速さで発展することを要求されてきた．歴史に類を見ないコンピューター・ハードウェア技術の急速な発展を追跡する必要性が，それを強制してきたからである[18]. 日本の産業構造や契約形態では, この急速な展開に対処できない．したがって，この問題は，私的な企業努力だけによって自助的に克服することは難しい．たとえば行政における健全な情報処理産業の育成のための指導や協力が必要である．市場に任せていても，健全な産業の育成は得られない．そのような政策を策定する前提として，健全な情報処理産業がどのようなものであるかを計るための理論が共有されていなければならない．

筆者はこの役割をソフトウェア工学という学問に期待するが，研究は不十分である．日本における議論はさまざまな主張を受容して併記する傾向にある．これでは，その中に正しい指摘が一つの立場として含まれていたとしても, ソフトウェア･システム開発の問題点を矯正する指導的な力にはなりえない．意識変革が現場にもたらされるための明快な論証と合意が必要である．そのための一助になることを願って，本稿を提出する．

2. 日本におけるソフトウェア開発の発展と現況

日本のソフトウェア・システム開発は，少なくとも 1960 年代にさかのぼる．1970 年代までのコンピューター・システムは，第一世代のメインフレームを中核に構成されていたものであり，高価なハードウェアを導入できるのは，官公庁や大企業に限られていた．ハードウェア・メーカーは自社のハードウェアの販売を確保するために，OS などを差別化し，ソフトウェア開発の環境のかなりの部分はハードウェア・メー

* † コーナンソフト
Konansoft

カーに依存していた．この開発環境において，コンピューター・システムを導入できる大規模な企業は，自社のマンパワーの一部をシステム要員としてシステム開発に割き，ソフトウェア開発の主要部分を自社内でまかなうことができた．現在まで，いかなる意味でもソフトウェア開発の決定的な方法論は確立されていないゆえ，その製作は常に試行錯誤を伴ったが，新技術としてのソフトウェア・システムの導入は大きな効果を期待させたので，強い予算の制約を受けずに開発に取り組むことができた．とは言え，この開発を支えていたものは，プロジェクト管理者の能力および努力[20]と，ソフトウェア技術要員の超過労働[19]であった．

コンピューターの世代が進み，メーカーから独立した OS（たとえば Unix）が実用化され，1990 年代からは Windows を OS として搭載した PC が普及して，ソフトウェア・システム開発はハードウェア・メーカーからから独立して行われるようになった．その結果，ソフトウェア・システム開発には，ハードウェアの知識とも業務知識とも異なる，独自の方法論が必要になり，ソフトウェア・システムの開発技術者の要件は，システムを導入する企業の要求の枠内に納まらないものになった．これと並行して，ソフトウェア・システムの開発環境は，安定性を失った．日本では，第一世代の時期に，ソフトウェア開発に日本語を使用することの模索など，方法論において一定の主導権あるいは独自性を確保しようとする動きもあったが，主にアメリカ合衆国から発信されるダイナミックで豊富なパラダイム展開についていけず，実を結ばなかった1．

開発方法論が安定しない状況下では，ソフトウェア開発の見積もりが困難なものになり，独自企業によるソフトウェア・システムの開発受注は，予測できない危険を伴うものになった．開発方法論の決定打が登場しないという事情は米国でも同様だったが，同国ではさまざまな試行を繰り返して方法論を科学的，工学的に追及する努力の余地があり，多様な提案が試され，あるいは発展し，あるいは淘汰された．日本の方法論の展開は，これに比べてかなりの程度で遅れており，米国の理論展開の紹介や後追いの水準を大きく出ることがない．信頼できる見積りを得られない競争原理の下では，ソフトウェア開発を請け負う企業が安定的に製造を担う環境は生まれない．あえてそれに挑戦する企業のプロジェクトの正否は偶然的な要因に支配され，リスクを管理できずに淘汰されていくことになる．その結果，日本では，ソフトウェア開発を担う企業は，最も根本的な部分で，開発物の提供ではなく，人材の提供に業務を絞り，リスクを回避することになった．これが競争原理の合理的な効果である．

このようにして，日本のソフトウェア・システム開発企業の歪んだ産業構造が固定化されてきたが，前述のように，このような産業構造では質の高い開発技術者を産業の中で育成することができず，日本のソフトウェア・システム開発産業の貧困という悪循環に陥ることになるといえる．これを克服するためには，理論面におけるソフトウェア・システム開発方法論の解明が重要である．

3. 設計工程の先入観の効果

筆者の経験から言えば，ソフトウェア開発プロジェクトの失敗の直接的な原因の大半は，プログラミングの軽視である．たとえば，無理な計画やその他の原因による作業工程の遅延のため，スケジュール上は総合テスト工程に至っていながら，プログラミングや単体テストの実施が不十分であるという事例がある．多くの例において，スケジュールの管理者はプロジェクトの進捗を正確に把握しておらず，進捗の遅れが認識される時期は，システムを目に見える形で起動するテスト工程にずれこむ．特に，スケジュールの管理者にプログラムに関する知識が不足している場合には，このような事態が頻繁に発生する．

そういう事態になったときには，プロジェクトには過剰とも思えるマンパワーが投入されるが，そこではプログラムの作成や読み直しを行わず，仕様書あるいは設計書とされるドキュメントの整備や読み直しに基本的な労力を割こうとする．そのために，それまでに行ってきたプログラミングやテストの作業が帳消しにされ，傷口が，つまり損害が広がる．筆者は，単体テストの検証が依拠する対象は，仕様書ではなくソースコードでなければならないと考える2．あるいは，システム構築が巨大なレガシーシステムからの継承を前提としているのにも関わらず，レガシーシステムのソースプログラムを読むことができない事例があった．これは，マンパワーの不足と，信頼できるソースプログラムが管理保存されてこなかった事情との二つの側面からの困難であった．ところが，このような初期環境が明白であったにもかかわらず，膨大で不要な基本設計などのドキュメントが当初予算のかなりの部分を割いて作成され，システム開発の直接作業の必要な予備作業を欠いたまま，技術者たちがとつぜん開発作業に放り込まれた．このような状況下でプロジェクトが行き詰まるのは，むしろ当然である．これらの失敗例の共通点を見ると，プログラミングの工程の手前に，設計工程が存在しなければならないという強い先入観が観察される．これが事態を悪化させ，好転を妨げていた．設計工程の存在の先入観は，プログラミングの検証をドキュメントの検証に置き換えさせる効果を持つ．道に迷ったときに元いた地点に戻るというのは，問題解決の鉄則であるから，プロジェクトが行き詰ったときに，設計に戻ろうとするの

1第一世代のメインフレームにおいて，日本は通産省などの政府主導型でコンピューターハードウェアの開発技術の獲得に力を入れており，また富士通や NEC などの企業が米国製のメインフレーム機に対抗しうる独自のハードウェアの生産技術を有していた．ソフトウェアの開発においても独自性を獲得しようという意欲は，このような環境から生まれたものといえる．しかし，第二世代以降，半導体を大量生産する高い水準の技術を保持し続けても，ハードウェア生産技術の優位性はソフトウェア生産の優位性を保障しなくなった．日本の情報処理技術を主導する人々は，ソフトウェアの独自の価値を正確に認識できないという遅れから回復することができなかったといえる．

2 これは，Agile 方式論の展開により，支持されている視点である[8][17].

は，セオリーとして正しいように見える．しかし，これはウォーターフォール･モデルを前提にしてはじめて成立するセオリーである．

他方，ソフトウェア・プロジェクトの発注･受注契約の構造には，設計工程が存在することにより私的利益を得る人たちや企業の存在を生み出す事実にも注意しなければならない[19]．どのような種類のものであれ，利害関係や既得権益は強力である．学問としての理論がこの要因を克服して方法論を改善するためには，ソフトウェア・システム開発の設計工程の工学的な性質に関する考察を展開し，それをソフトウェア開発に関わるすべての人々の共有認識にするべきであると筆者は考える．この目的を達成するために，1992 年に Jack W. Reeves が提案したソフトウェア開発の設計工程に関する洞察，つまり Reeves の仮説を理論的に検証することが有効である．

4. Reeves の仮説

1992 年に Jack W. Reeves は，論文” What Is Software Design?”において， ソフトウェア開発における設計工程についての重要な仮説を提案した．その仮説を整理すると以下のようになる．

定義．エンジニアリング（工学）の条件を満たす設計工程で作成される設計書は，後続工程を完全に定義し，後続工程は設計書に基づいて，設計部門の関与がないままに作業を完遂できなければならない
仮説．ソフトウェア開発において，エンジニアリングの設計書の条件を満たすドキュメントは，ソースコードだけである
系１．ソフトウェア開発のライフサイクルにおいて，エンジニアリングの条件を満たす設計工程にはプログラミングが含まれる．
系２．通常，プログラミングは製造工程に擬せられているが，実際にエンジニアリングの製造工程に対応する工程はコンパイル・リンクである．

彼は，ソフトウェア開発におけるドキュメントの製作や作業の計画など，プログラミングの前工程の必要性を否定したのではないし，前工程までに作成されるドキュメントを不要としたわけでもない. 1992 年の” What Is Software Design?”においても, 2005 年の“What Is Software Design: 13 Years Later”においても, 彼はソフトウェア開発プロジェクトの編成や進行について，何らかの変更を行う必要があるとか，あるいはどのように変更すべきであるとか言うような提案を全く行っていない．ただし，そのような提案は，彼の仮説あるいは原理が承認されれば，その応用として発展してくるものであろうことは，容易に予測できる．その仮説は，ソフトウェア開発の方法論のあり方に大きな影響を及ぼす，重要な仮説であり，Reeves は，その仮説を証明できないかもしれないが，その仮説を採用することで，ソフトウェア技術者が経験的に認識している多くの現象を，合理的に説明することができると書いている[10].

5. 「上流工程」完全知識の要件

周知のとおりウォーターフォール・モデルでは，要件定義の工程の次工程に設計工程がおかれ，プログラミング作業は設計工程の次工程の製造工程に含まれると認識される．Reeves の仮説で示した定義で，ウォーターフォール・モデルの認識を見ていくと, ウォーターフォール･モデルにおける｢上流工程｣完全知識の仮説が導き出される．

つまり，ウォーターフォール・モデルが，安定的に正常に機能するためには，「上流工程」で作成される成果物により，「下流工程」が独立して目的とする成果物を製作することができ，さらに「下流工程」で独自に発生した問題が｢上流工程｣に影響を与えるという可能性が否定されなければならない.「下流工程」で発生した，あるいは発覚した欠陥が,「上流工程」における対応を必要とする場合，つまりステージの手戻りが発生した場合には，そのような欠陥は「上流工程」で除去することが可能であったにもかかわらず，何らかの過誤により,「上流工程」でのチェックを通り抜けて｢下流工程｣に継承され, そのために損害が発生したものでなければならない. このような条件を満足させるためには,「上流工程」の技術者が，次工程以降の作業において発生する現象に関する完全知識を持たなければならない．そうでなければ,「上流工程｣で想定していない事象が起こり,「下流工程｣は「上流工程」で定めた作業計画作業計画には全く記述のなかった現象に独自に対応し，想定外の現象が｢上流工程｣で定めた計画に矛盾する場合には,｢上流工程」のドキュメントにさかのぼって作業を変更しなければならなくなる．このような現象が起こらないためには,「上流工程」が完全知識を持っていなければならない．これが｢上流工程｣完全知識の要件である．

6. 設計記述システムの完全性の要件

ところで,｢上流工程｣完全知識の要件の成否とは別に,｢上流工程｣の技術者は，得られた情報を総合分析して, ドキュメントを作成し, 次工程に渡さなければならない. ソフトウェア開発におけるウォーターフォール・モデルを前提とすれば，設計工程とされる工程の次工程は製造工程に対応付けられるプログラミング工程である．設計部門は製造部門にドキュメントを渡して，次工程以降はそのドキュメントにしたがって工程の作業を完成させることができなければならない．そのドキュメントは何らかの記述システムによって表現されているものだが，その記述システムは，設計者の思惟を十分に表現できるものでなければならず，設計部門の担当者と製造部門の担当者との間で共有できるものでなければならない．つまり，記述システムの表現能力が十分

なものである必要がある．これが設計記述システムの完全性の要件である．

ソフトウェア・ライフサイクルにおけるウォーターフォール・モデルが安定的に成り立つためには，上記の二つの要件，「上流工程」完全知識の要件と設計記述システム完全性の要件が満足される必要があることは自明であるといってよい．このとき，ウォーターフォール・モデルが安定的に成り立つという条件は，Reeves の仮説が正しいという想定と相反する条件であるといえる．この点を一般化して，上記の二つの要件の両方が成り立つ可能性が論証されれば，ソフトウェア・ライフサイクルにおいて，ウォーターフォール・モデルが安定的に実施できることが論証され，Reeves の仮説が成り立たないことも論証されることになり，上記の二つの要件の一方または両方が満足されないことが論証されれば，Reeves の仮説の正しさが論証されたと考えても，大きな危険性はないだろう．

7. ハードウェア製作における設計の二要件の成立

ところで, 前述の Reeves の仮説における定義の部分は論理演繹によって導かれたものではなく，ハードウェア製作の工学原理から得られたものである，つまり，ハードウェアの製作においては，この定義を満たす設計工程が成立し，多くの製品はこの工程分離を経て製造されている．つまり，ハードウェアのプロセスモデルにおいては，設計工程における完全知識の要件と記述システムの完全性の要件が成立しているのである．そこで，その要件はどのようにして成立しうるのか，またソフトウェア開発においては，ハードウェア開発の類推において，上記二要件が成立しうるのか，あるいはどのような相違により，成立できないのかが問題になる．

ハードウェアの設計工程が二要件を満たしているということは，設計における欠陥が完全に取り除かれる保証があるという意味ではない．同時に，ハードウェアの設計にあたる技術者は，厳しい検証作業を繰り返し行い，設計を完全なものに近づけるために努力する．たとえば，建物の耐震性について，一定の衝撃に対してまでの安全性を確保するためには，どのような建築方法を実施するべきかを，事前に検討し，その結果に基づいて設計を完成する．そのために，計算による推測を行い，あるいは模型やコンピューター上でのシミュレーション実験などを行う．ハードウェアの設計において，設計工程の信頼性の検証は設計工程内で完了するものであり，製造工程に設計工程の正確性を検証すべき要因は含まれない．製造工程について検証されるべきものは, 設計工程で指定した方式を正しく反映して製造工程が行われているか否かである．たとえば，ある程度の衝撃まで耐えうると設計工程が計算し，予測したハードウェアが，実際の使用例において条件が満たされていないことが発覚したとしても，製造工程が設計書の指定を満足していれば，それは設計工程の欠陥である．

ここで，ハードウェアのエンジニアリング製作における設計工程の完全知識の要件のいう，完全知識の意味を再確認しなければならない．一般論として言えば，考察が科学に基づくものである限り，人間は将来の事象について，常に不完全知識しか持たないというべきである[14][15]．ハードウェアの設計では，製品が使用に供されるに当たり，どのような条件下に置かれるか，どのような現象が使用に当たって発生するかを予測し，予測にあわせて設計を行う．そのときに，たとえば数学のモデルが設計のツールとして使用される．しかし，数学のモデルはハードウェアが使用される実際の環境と同じではないことは言うまでもない[21]．このような意味で言えば，ハードウェアの設計において，設計の技術者が利用できる知識は，必然的に不完全な知識である．

このように, ハードウェアのエンジニアリングにおいて Reeves の定義が成立するための条件としての完全知識の要件の知識の対象は，製品が将来どのように使用されるかについての完全性ではない．しかし，同時に設計者が作成するドキュメントは，ハードウェアの製作の諸条件を網羅的，包括的に規定するものでなければならず，その規定の根拠は，不完全な知識を総合した上で，設計者が最善と判断する意思決定にある．つまり，ハードウェアの設計者は，設計書のドキュメントが製造工程に渡された後の生産過程については，完全情報に近いものをもつ立場にいると考えられる．これが，ハードウェア製作の工学原理における「上流工程」完全知識の要件である．

また，ハードウェア製作の設計書が作成され，たとえば指図書のようなドキュメントが作成され，製造部門の工程が設計部門の介入なしで作業を完成させることができるのならば，設計工程で作成されたドキュメントの記述システムが完全であることになり，記述システムの完全性の要件が満たされていることになる．この要件の成立には，一定の留意点がある．設計工程で指定する製造工程の作業に，何らかの技術的熟練などが要求される場合，設計工程の要員は製造工程の熟練技術を理解していることを求められるが，同程度の技術水準を会得していることを要求されはしない．あるいは，製造工程の熟練技術が設計要員にとってブラックスボックスであることも可能であり，設計要員は製造工程の熟練技術が出力する成果物の属性について理解して，そのような熟練技術が利用可能であることが確認されていれば，設計工程の完全知識の要件は満たされるといえる．設計要員と製造要員の間に，それらの熟練技術を含めた製造工程のすべての，有限の生産要素についての共有理解が存在し，それらを記述システム(たとえば記号)によって表現することができれば，記述システムの完全性要件も満たされるということになるのである．

8. ソフトウェア設計における完全知識の要件

次に，ソフトウェア開発における設計の二要件の成立について考察する．まず，Reeves の仮説における系２において，つまり，ソフトウェア開発で製造工程に該当す

るプロセスはコンパイル・リンクであると仮定し，ソースコードがハードウェア製作における設計書に該当すると考えた場合には，設計二要件が成立していることは明らかである．

そこで，ソフトウェア開発におけるプログラミングが製造工程であり，プログラミングを規定する何らかのドキュメントが設計書であると仮定した場合，設計の二要件が成立するか否かが問題になる．第一に完全知識の要件について検討したい．

ソフトウェア開発のライフサイクルで，プログラミング工程の前にエンジニアリングとしての設計工程があり，その工程で作成されたドキュメントでプログラミングの作業を確定する場合に，設計工程には完全知識の要件が成立しうるのだろうか．

ソフトウェア開発の最終的な目的は，電子計算機上で起動するプログラムまたはその集合であるシステムであり，プログラムはソースコードのあり方を捨象して機能を抽象化できるように見える．プログラムをそのような抽象機能に置き換えることができれば，抽象機能を組み立てることで，全体像としてのシステムを概念上で構想し，その概念をドキュメントで記述することにより，ソフトウェアの設計とすることができるように見える．システムとして作成される場合に，特に重要なのは，データの構造や，データの入出力，あるいは表示形式の定義である．これらの関係を概念化し，プログラムの機能も概念化すれば，実際のプログラムの作成については製造工程たるプログラミング部門の要員が専念する（インプリメンテーション）ことにしたら，「上流工程」の介入しない製造工程を実現できることになる．しかし，問題はそれほど単純ではない．

第一にプログラミングの自由度とプログラムの複雑性の関係を考える必要がある．プログラムを機能として抽象化するにしても，実際のプログラムを単一機能だけで作成することは不可能または無意味であり，逆にユーザー要求の複雑化に伴いプログラムは必然的に複雑化する[16]．また，複雑なプログラムの作成方法は，画一的ではなく，ほとんど無限大の方法がある．つまり，プログラム作成の自由度は大きい．このように作成されるプログラム群は，非常に大量の情報を含むものであるから，システムを構成する全プログラムに関する情報を，一つの部門，たとえば設計部門で完全に集約することは困難であり，敢えてそれを実現しようとすると，プログラムの自由度に制約を設ける必要が出てくる．

またハードウェア製品に比較して，ソフトウェアは変更するのが容易で，歴史的にも，プログラムは諸条件に合わせて頻繁に変更され，そうすることによりソフトウェアの価値が発揮されてきた[8][18]．

大規模なシステムにおいては，多数のプログラムの作成を複数の人員あるいはチームにより分担して行わざるを得ないが，その場合にはプログラムモジュールの機能分担とモジュール間のインターフェースの決定が，プログラミングに対する深刻な制限となる．ところが，プログラミングを実施する前に，これらの要素をすべてあらかじめ決定するには，考慮しなければならない要素の数が多すぎる．敢えて，事前の確定にこだわっているとプロジェクトの進行に重大な遅れを生じさせることになるので，経験をつんだ技術者であれば，そのような危険を回避して，解決できない問題については後続の工程において解決していく方法を選択する．つまり，実務において完全知識の要件の不成立が認識されているのである．複数のグループでモジュールが別個に開発されているとき，モジュール間のインターフェースの決定形式も多様であるが，その形式により，モジュールを作成するグループの立場がかなりの影響を受ける．インターフェースのあり方には，あるグループの開発作業を有利にすることもあり不利にすることもある．「上流工程」で予測したインターフェースの形式，要件が実際のモジュール化では欠落事項を含むことは，珍しくないことだが，インターフェースが完全に定義されているという前提が共有されている場合には，欠落した情報の処理をどのように消化するべきかが問題になる．「上流工程」の完全知識の前提のための中立的なインターフェースの調整機能が働かないと，この欠落を補うための損害の配分が公平に行われず，力関係や駆け引きの巧拙により，損害を負担しないグループと，大きな損害を負担させられるグループが生じることになる．

そもそもプログラム工程の前に「設計」工程を設けて，情報の整理を行う目的は，ソフトウェアの高い品質を維持するためであるが[5]，ソフトウェアの品質は複数の相矛盾する属性によって，多面的に測られるものである [7]．そして，ソフトウェアの品質においては，ソースコードの可読性，理解しやすさ，メンテナンスしやすさなどの属性が，重要な役割を占める．ところで，読みやすい，分かりやすいプログラムが書かれるかどうかは，根本的にはプログラマーの能力に依存するものである．コーディングに関する何らかの規則を当てはめたり，特殊なジェネレーターを使用して自動的にコーディングを行い，この角度から見た高品質のプログラムを作成しようとする努力が行われてきたが，これによって決定的に問題が解決することはなかった．

つまり，「設計」工程における完全知識の要件を満たすためには，知識を完全に収集する能力を保証するのではなく，逆に，プログラミングの機能を「完全知識」の範囲内に規制しなければならなくなる．これは，プログラミングのそもそもの価値，ソフトウェアの最も重要な価値を犠牲にして，ウォーターフォール・モデルのライフサイクルを維持しようという，本末転倒な試みに発展することになる．したがって，プログラミングの基本的な性質から見て，プログラミングの前工程に設計工程の存在を仮定し，その工程の技術者が「上流工程」完全知識を獲得できるという条件は自己撞着に行き着くものであり，この要件はみたされない．

9. ソフトウェア設計における記述システム完全性の要件

次に，記述システムの完全性要件の成立について検討したい．前節で論じた分析と

記述システム完全性要件の問題は，同じ分析対象を別の角度，別のアプローチから検討する問題である．ここでは，記述システムと思惟の関連を軸として，ソフトウェア開発のウォーターフォール・モデルにおける記述システムの完全性要件について検討したい．

10. 言語，言語の記述および思考の性格について

ウォーターフォール・モデルを前提とするソフトウェア開発では，「上流工程」で作成されるドキュメントの一部または全部を設計書と解釈し，そのドキュメントに基づいて「下流工程」であるコーディング，プログラミングを行うと想定する．設計書の内訳は，自然言語による説明，モジュール分割を反映したプログラムやモジュールの一覧表，フローチャート，ハイポチャートあるいはデシジョン・テーブルのようなプログラムの詳細な設計書，ファイルレイアウトや画面レイアウトのような定義書などからなる．

ところで，これらのドキュメントはいずれもソフトウェア開発に従事する技術者の分析や情報収集，整理などの思考過程の結果を表現し，格納したものである．つまり，これらのドキュメントのそれぞれの表現形式は，人間の思考を反映する情報の記述システムであるといってよい．

記述システムは一定の情報を記録し伝達するものであるが，他方において人間の思考は記述システムに依存する．言語も記述システムも存在しない絶対抽象の思惟を展開しそれを具体的な記述システムで表現することもあるかもしれないが3，ソフトウェア開発の工程においては，技術者の思惟の前提に記述システムがあり，記述システムにもとづいて思惟を展開していく例がほとんどであるといってよい．ソフトウェア開発は一定の予算と工期に束縛された事業であり，思考方式も一定の枠組みに当てはめてから進める必要がある．たとえば，プログラミングについて，最善の解説書よりも最悪のサンプルコードのほうが役に立つというようなことが言われる．

さて，記述システムがソフトウェア開発の技術者の思考の基礎になっているとすると，さまざまな記述システムの表現能力が問題になる．それについて考察を進めるために，いくつかの記述システムを例示し，それぞれの方式の特性について検討する．この例示は包括的とはいえないが，実際にソフトウェア開発のドキュメントとして使用されている記述形式の大半を包含している．

1．自然言語
2．書式
3．フローチャート
4．デシジョン・テーブル
5．グラフ
6．数式
7．プログラム言語

言うまでもなく，自然言語は人間が有している表現形式，記述形式で，表現能力の包括性において，最も優れたものである．自然言語によって処理することが可能な領域は人間の思惟によって創造されるすべての領域に近いものである．一方，自然言語は冗長や曖昧さを許容する傾向が大きく，同一の表現について複数の異なる解釈の並立を許す場合がある．このような曖昧さを除去するためには，多量の自然言語を使用するか，自然言語の表現方法に対して相当の前提規約を設定する必要がある．しかし，前提規約を付したり，多量の自然言語を使用して，曖昧さを除去しようとすると，理解しやすさが失われ，意味を伝えにくくなる．

ソフトウェア開発における情報記述において，自然言語は必須であり，特に他の記述形式では表現できない部分については，自然言語で表現しなければならない．しかし，すべてのドキュメントを自然言語だけで表現することは，開発作業の効率性の面での問題があると思える．

ドキュメントを記述するための書式を作成して，各欄に定められた事項を書き込むことによって，必要な情報を記述させるように導き，記述者の思考に有効な枠をはめることができる．このような書式は，各種の申請書やビジネス文書などで使用されているが，ソフトウェア開発でも有用であり，ドキュメントとして広く使用されている．たとえば，データベースのテーブル設計において，各項目の名称，属性，長さなどを網羅的に記述することによって，効率よく，明瞭に情報記述を行える．帳票の印刷形式などについても，この方式は強い力を発する．しかし，書式が効力を持つのは書式の作成時に，特性を完全に認識されている情報に関する選択や記述であり，この方式だけでソフトウェア開発に要求されるすべての情報を記述できると考えることはできない．

フローチャートは，プログラムのアルゴリズムなどを効果的に記述することができるので，単純なプログラムの仕様書としては有効である．記述能力に限界があるので，これだけでプログラミングに替えることはできないが，プログラムの動作のおおよその構造を記述する表現能力は高い．

3 Michael D. Coe は，いかなる言語とも結びつかず，思惟を直接に表現する記述システムが存在しうるのだろうかと問いかけ，そのような可能性を肯定的に考えた例を紹介した．イギリスの言語学者 Geoffrey Sampson は，記述システムには semasiographic なものと glottographic なものがあり，前者は特定の言語には結びつかないと考えているという．しかし，Coe は実際には，semasiographic 的な記述方式の記述システムの起源や発展に果たした役割は皆無に近いほど小さいと指摘する．Coe[12], pp18-19

プログラムを分岐と処理の論理要素の集合と考える立場からみると，デシジョン・テーブルは完全なプログラムの記述形式であるように見える．しかし，プログラムの性質をこのように単純化して捕らえることは危険である．実際に，デシジョン・テーブルは，限られた条件下における処理を明瞭に網羅的に表現することができ，効果的に使用できる領域も小さくない．しかし，条件が複雑な場合には，テーブル表現が困難になり，表現できたとしても理解が難しいものになる．また，分岐構造の中の長い処理の詳細な記述を行うには適当ではない．

グラフは，概要などを感覚的に把握するために利用できるが，作業の具体的な指示として使用できる独自の能力は小さい．

数式は，ある面で自然言語の曖昧さを解消する強力な表現力を持つ．しかし，表現能力の多様性や豊富さという面では強力とはいえない．

これらのさまざまな記述システムに比較すると，プログラム言語は他の記述システムにはない多様で強力な能力を有し，改良を加えられている．プログラム言語は，自然言語，書式，数式など，他の記述システムの特性を多く取り入れて，それらの能力の主要な要素を具備している．ソフトウェア開発において，いずれの部門に属するにしろ，開発要員がもっとも完全，明瞭かつコンパクトに情報を記述することができる記述システムはプログラミング言語である．したがって，ソフトウェア開発における思考の展開が，もっとも多く依拠することになる記述システムも，プログラミング言語であるといえる．

ソフトウェア開発において，プログラミング言語以外の記述システムに基づく思考と，それによるドキュメントの作成が必要な局面があることは当然だが，そのような記述ドキュメントがプログラミング言語によるドキュメントの作成の前にすべて完了していて，プログラミング言語以外の記述システムで作成されているドキュメントがプログラミング言語によるドキュメントの作成工程を完全に規定しなければならないと考えるためには，プログラム言語の記述能力とそれ以外の記述システムの記述能力に差がありすぎるという問題がある．ソフトウェア開発において，完全性を体現しているとみることの可能な記述システムはプログラム言語であり，自然言語を除けば，プログラミング言語以外の設計用の記述システムは，プログラミング言語よりも表現能力も思惟の補助能力も劣っているのである．

11. 設計作業とプログラミング作業の等質性または連続性

作業の等質性または連続性の問題を考察するために，ハードウェア製作のエンジニアリングにおける設計工程と製造工程の性質に戻らなければならない．ハードウェア製作のエンジニアリングにおける設計工程と次工程である製造工程には，明らかに作業の性質の変化がある．ソフトウェア開発のプログラミングとそれ以前の工程の作業にはそのような異質性があるだろうか．その点を確かめるために，工程で作成される成果物を比較する必要がある．ハードウェア製作の設計工程で作成される成果物は，製造工程を規定するドキュメントである．設計工程では，さまざまな所与の条件を総合し，ハードウェアをドキュメントにおいてモデル化し，製作の方法論を定義する．次工程の製造工程では，前工程で定められた方法論にしたがい，前工程で確立されたモデルとしてのハードウェアを実際の生産物として体現させていく（インプリメンテーション）．この工程の間には作業の質の変化，作業の流れの切断がある．

ところが，ソフトウェア開発におけるプログラミング工程と前工程の間の作業には，このような切断は認められない．ソフトウェアのライフサイクルの考え方によれば，ソフトウェア開発プロセスは要件定義，設計，製造（プログラミング），テスト及び運用と分類され，概ねこの順序に生産活動が推移するのだが，要件定義からプログラミングまでの作業は，視点の相違や対象の規模の相違はあるとしても，作業の性質は同じである．

さらに，前節で明らかにしたとおり，自然言語を別にすれば，プログラミング工程の前工程で設計作業として使用するものと想定される記述言語の表現能力は，プログラミング言語の表現能力よりも貧弱である．人間の思惟が記述言語にもとづいて実施されている場合，貧弱な表現能力は，人間の思惟の展開を貧弱な水準で実現する．貧弱な思惟の奥に不明瞭な問題があれば，人間にはその問題自体が認識されず，未解決のままに思惟が完結することになる．ソフトウェア開発に照らして考えれば，未解決のまま温存された問題点は，より表現力の豊富なプログラミング言語に基づく分析を実施するときに顕在化し，人間がその問題を認識したときになって，ようやく解決の方途が検討されることになる．このように，ソフトウェア開発のステージの進展においては，いわゆる設計工程から製造工程の移行において，質の変化が見られず，作業の等質性と連続性が保たれているのである．

加えて，ソフトウェア開発のステージにおいては，プログラミング言語という記述システムによる思惟の完結の後の作業においても，作業の等質性と連続性が切断されるとはいえない．つまり，Reeves の仮説の系２を承認し，ハードウェア製作の製造工程に対応するソフトウェア開発のステージがコンパイル・リンクである考えても，ソフトウェアが非常に簡単に変更できるという性質により，ハードウェア製作とは異なり，製造工程の完結は生産物の固定化には結びつかない．ソフトウェアの開発作業は，製造工程としてのコンパイル・リンクの後のテスト工程に継続され，ソースコードの見直しと修正が繰り返され，製造工程であるコンパイル・リンクも繰り返し実施される．このように，テスト工程の作業と設計工程の作業の間にも，作業の同質性と連続性が維持されているのである4.

4 Reeves[9], “The overwhelming problem with software development is that everything is part of design process.”

ハードウェアの製作過程とソフトウェア開発のライフサイクルのステージには，すでにその実行順序において根本的な相違がある．

さて，作業または労働の質に変化がないままに，いわゆる設計工程からプログラミング工程，さらにテスト工程へと工程が進むとすると，各工程で主に取り扱う記述システムが問題になる．前節で説明したように，前工程において主に取り扱う記述システムは自然言語を除けば，すべてプログラミング言語よりも表現能力が劣っているのだから，同質の工程間における次工程の作業を規定するための記述システムとしては不十分であることが自明である．つまり，ソフトウェア開発でウォーターフォール・モデルを前提にした場合には，設計工程には記述システムの完全性の要件が満たされていないのである．

12. ソフトウェア設計の二要件の不成立と Reeves の仮説の論証

以上を整理すれば，ソフトウェア開発において，ウォーターフォール・モデル，あるいはそれに類似するプロセス・モデルにおける設計工程の認識を前提にした場合，つまりプログラミングを製造工程に擬し，プログラミングの何らかの前工程に設計工程があるとした場合には，設計工程における完全知識の要件と記述システムの完全性の要件が成立しないことが確認される．このようにして，Reeves の仮説は経験的に妥当であると認識されるだけでなく，演繹的な検証を通して，真実性を推定することができるのである．

13. Reeves の仮説の価値と応用

Reeves の仮説は，システム開発に携わる多くの技術者の経験に照らせば，特別な着想ではなく，受け入れやすい議論であり，これが広く承認されない主要な理由は，ソフトウェア産業界の拒絶にある5．では，Reeves の仮説を受け入れた場合，ソフトウェア産業界に何がもたらされるのか，何が得られるのだろうかという疑問が生じる．ソフトウェア開発における設計とコーディングの問題，あるいはドキュメントとソースコードの問題は，ソフトウェア開発の固有の問題で，繰り返し議論されてきた．ソフトウェア開発におけるプログラミングの重要性は，多数の技術者や研究者により，以前から認識され，指摘されてきた．実際のソフトウェア開発においても，プログラミングの重要性を理解する技術者や責任者の下では，プログラミングを重視する開発が行われ，成果を遂げてきているのである．

これらの事実を見た場合，Reeves の仮説の承認がどのような一般的な価値を有するのかという，疑問がわくかもしれない．実際に，Reeves はプログラミングを重視した上での，ソフトウェア開発の具体的な方法論を提供していない．

Reeves の仮説はソフトウェア開発の原理的な問題点についての理論的な仮説であり，彼は仮説または原理を確認した後の応用論にまで進んでいない． Reeves の経歴を見ると，彼は情報科学の理論研究を専ら行っているものではなく，経験をつんだソフトウェア開発の技術者である．とするのならば，彼は実際の業務において，ソフトウェア開発の実践を行ってきたのであり，さまざまな制約をそれぞれに克服しながらプロジェクトの目的を達成する努力を行ってきていたと容易に推察できる．彼は工学的な，エンジニアリングとしてのソフトウェアの開発をめざすという課題に取り組む中で[10]，ソフトウェア産業界が陥っている重要な誤謬に気づいたので，論文でそれを指摘したのである．工学的，つまりエンジニアリングとしてのソフトウェア開発の方法論を追及するときに，ある誤った仮説があり，その仮説をほとんどすべての関係者が暗黙裡にあるいは明示的に前提としており，そのために開発方法論の誤謬が克服できない場合，誤謬の根元にある仮説を訂正することに大きな意義がある．また，ある仮説がほとんどの関係者にとっての暗黙の通説あるいは意識された通説である場合，その仮説に逆らう提案は権威により，あるいは集団により否定・排除され，その仮説の誤りを指摘して，共有化させることは容易ではない．その場合には，原理の応用の前に原理の真偽に関する議論が必要となり，その確認がなければ応用に進むことができない．それゆえに，Reeves の仮説の価値は，ハードウェアのエンジニアリングのアナロジーにより，想定されていたソフトウェア開発における設計工程あるいはソースコードを規定する設計書についての誤解をあらため，エンジニアリングとしての設計書の要件を具備するドキュメントはソースコードだけだという命題を指摘したこと自体に存するのである．実際には，現在においても Reeves の仮説の意味は日米の情報処理技術者や情報科学の研究家に十分に理解されているとはいいがたい．Reeves の仮説は，ソフトウェア開発においてソースコードとドキュメントのどちらが重要なのかというような価値判断ではない．それはハードウェアのエンジニアリングとソフトウェアのエンジニアリングとの対応関係の理解についての，誤解の訂正であり，事実認識の問題である．たとえば，地動説はそれ自体で価値があり，セー法則を否定する有効需要の原理がそれ自体で意味があるように，Reeves の仮説の真実性を確認することは，応用の前提の問題であり，それ自体が意味を持つのである．

14. さまざまな方法論の提案と Reeves の仮説との関係

ソフトウェア開発で，効率よく高品質の製品を生産することは，産業界の共通目的であり，そのためのさまざまな方法論が提案されてきた．その中には，たとえば形式手法，要求工学，ＵＭＬなどのように，部分的に Reeves の仮説と相反するかのように見えるものも存在する．それらの方法論を信奉する人々が主張したり宣伝したりする

5 Reeves[9], “The reason has more to do with the refusal of the software industry to accept code as design than any real engineering difference.”

議論では，Reeves の仮説あるいは原理から見て，成立し得ないと思える命題が提案されることもある．それらの方法論に関する検討において，Reeves の仮説との関係を検討し，方法論の評価検討と Reeves の仮説の評価検討を相互に実施することは，もちろん，有意義なことである．しかし，Reeves の仮説と要求工学のような方法論の検討は，次元の異なる議論であり，それを必ず直接的に対峙させなければならないと考えることは問題を見違えている．

Reeves の仮説は精製された命題であり，真であるか偽であるかの判定を最終的に下せるものと考えることもできる．しかし，たとえば，ＵＭＬのような記述形式が，あるいは開発方法論が，有用であるか否かという問題を，真偽のいずれかに決することは適当ではない．

ある開発方法論がある開発グループで採用されれば，その方法論に基づくプロジェクトが成功したり失敗したりする．プロジェクトが成功しても，それはその方法論が正しいのだということの論証にはならない．逆に，プロジェクトが失敗しても，その方法論の誤りを論証するものではない．プロジェクトが成功した場合，その方法論が採用されなければプロジェクトは失敗していたのかどうかが問題であり，プロジェクトの成功と方法論の採用との厳密な因果関係の存在が検証されなければならない．プロジェクトが失敗した場合の検証についても同様である．また，ある開発グループが何らかの理由である方法論を採用し，その方法論に習熟しながら開発能力を向上させていった場合，そのグループにとってその開発方法論は非常に重要な基盤となる．しかし，一般的にその方法論がどのような価値を持つのかという評価を，そのグループの視点から下すことは誤りであり，そのグループの成功例にはひとつの事例としてだけの重みを与えるべきである．

特定の開発方法論の採用とプロジェクトの成否の因果関係は，自然科学のような実験によって確かめることはできないし，産業界の動向を推定する統計資料からそれを読み取ることも困難である．方法論の採用から，方法論の一般的な価値が抽出され，客観的な理解が得られるには，数回のプロジェクトの実施例では得られない，長期の観察と経験の蓄積が必要であり，経験の蓄積の中では，提案された方法論自体も変化してもとの提案とは完全には一致しないものになる．

これら方法論などの提案は，ソフトウェア開発の技術の進歩のために貢献する重要な試行である．しかし，これらの手法の有用性の評価と Reeves の仮説の検証との関係はあくまでも間接的であることに注意する必要がある．

15. Reeves の原理の応用例の提案

Barry W. Boehm は，Software Engineering という概念を「コンピューター・プログラムなどを使用してコンピューター設備の能力を人間にとって有用にするための科学及び数学の応用である」と定義する6．Reeves の仮説が原理的に成立することが確認された場合には，エンジニアリングとしてのソフトウェア開発の目的から，応用により，コンピューターの能力を人間にとって有用にする成果を求めなければならない．本稿では，Reeves の原理の論証を試みているが，論証の完了を前提として原理の応用について，検討を進めたい．

Reeeves の仮説の応用によってコンピューター設備の能力を人間に有用にできるのは，どのような分野においてだろうか．筆者は，ソフトウェア開発方法論における新たな展望に，その本領があると考える．

16. フォード・システムの否定または放棄

18 世紀から 19 世紀にかけての産業革命は，人類社会の生産力を劇的に発展させたが，20 世紀初頭にベルトコンベアーの流れ作業による大量生産方式が案出，採用され，工業生産方式の基本となった．この方式を一般化して，劇的な効果を印象付けたフォード社の名にちなみ，ここでは流れ作業による大量生産方式をフォード・システムと呼ぶことにする．

Reeves の仮説を前提にした場合，フォード･システム的な大量生産がソフトウェア開発では成立し得ないことは明らかである[1]．すなわち，大量生産方式では，労働者の熟練技能を生産ラインに組み込み，熟練労働を単純労働に置き換えて，生産性を向上させ，同一規格，等品質の製品を大量生産するのであるが，これはハードウェア製作における製造工程の流れ作業化に基づいている．しかし，ソフトウェア開発では，製造工程に該当するプロセスはコンパイル・リンクの過程であり，このプロセスを効率面で合理化する余地はほとんど残っていない．

フォード・システムはエンジニアリングの当然の帰結ではないが，産業革命以降の人類社会の生産方式の劇的な変化を踏まえ，工学的な生産と大量生産の概念のオーバーラップが一般的に浸透し，ソフトウェア開発方法論においても，フォード・システム的な形態あるいはそれがもたらした効果が実現されることが，エンジニアリングとしてのソフトウェア開発の実現であるとする先入観が広く形成されている．したがって，ソフトウェア開発においても，理論的に，あるいは実際の開発プロジェクト内での運用形態において，フォード・システムから類推される開発方法論や作業方式が採用される動機が強く存在している7.

6 Boehm[6], “Software engineering is the application of science and mathematics by which the capabilities of computer are made useful to man via computer programs, procedure, and associated documentation.”, p.16

7 ソフトウェア開発にフォード・システム的な大量生産方式の効果を期待するのは，大量生産方式が生産効率を上げ，かつ企業に大きな利潤をもたらしたことによるものであるが，フォード・システム的な開発体制が大量生産の効果を実現しなくとも，「設計」を担当する企業や人員は，この体制の存在による利益を享受することができる．しかし，これは生産効率を増大させることによる利益が，生産に参与する人々や社会全体

Reeves の仮説を承認するのならば，このような方式は，ソフトウェア開発のプロセスモデルに関する誤った前提仮説に基づくものであり，否定されなければならない．ただし，ソフトウェア開発に関する多くの方法論が，明示的に否定しない限りは，フォード・システムを少なくとも暗黙の前提にしているとしても，それぞれに方法論の提案において，Reeves の仮説の採用がどのように方法論に影響を及ぼし，どのように方法論が否定あるいは修正されるべきであるかを検証することは，個別に行わなければならない重要な作業である．しかし，この個別検証は本稿のスコープを超えており，別の機会において論じたい．

17. 管理労働の異質性

Reeves は，ソフトウェア開発においては，すべての作業が設計作業の一部分であると指摘するが，これはソフトウェア開発に一種類の作業あるいは工程のみが存在しうるという意味ではない．基準となっている工程の概念がハードウェア製作のプロセスモデルを基準に分類されたものであり，そのような分類ではソフトウェア開発の工程に関する表現･描写能力が不足しているので，すべての工程に一つの概念を割り当てざるを得なくなるのである．

ところで，ソフトウェア開発のプロジェクトが成功するか失敗するかの重要な部分が，プロジェクト管理が有効に行われたかどうかに依存することが，よく知られている．ソフトウェア開発のプロジェクトが複数のメンバーで構成されると，管理の必要性が生じるが，これはソフトウェア開発に限るものではない．管理という職責は，水平的な種々の業種の活動に対し，垂直的に独自の方法論を持つ作業であり，目的を定め，そのための活動を促し，構成員間の矛盾を調整し統制するという性質を持つ．純理論的には，管理労働を行うときに，業種固有の知識や経験は必須要件ではない．もちろん，ソフトウェア開発のプロジェクトの管理に従事する場合，管理者にソフトウェア開発に関する豊富な知識や経験があれば，よい効果を成し遂げるために有利であることは当然である．

いずれにしろ，ソフトウェア開発のプロジェクトにおいて，管理作業はプログラミングを中心に形成される作業に対する異質性を保っているといえる．この点を踏まえ，Reeves の仮説を承認した場合に，ソフトウェア開発のプロジェクトの構造や活動形態がどのようなものになるのかを，いくつかのモデルを挙げながら検討してみたい．

18. ワンマン管理者型

ソフトウェア開発の目的は，ソフトウェア・システムを製作することであり，その成果は製作されたソフトウェア・システムの効用によって理解することができる．ソフトウェア・システムがどのような役割を果たすことを期待するのかは，ユーザー部門の要件定義の工程に属するものと解されているが，要件定義はコンピューター・システムの環境によって大幅に制限されるものであり，事前の完全な要件定義は基本的に不可能であり，ソフトウェアの製作の進行とともに，要件が変更され，あるいは形成されていくという方式が最も妥当な方式であるといえる．

ユーザー側に，あるいはユーザー側に非常に近い人員により，ソフトウェアの開発チームとの間に不断の緊張感が保たれ，要求が発信され続け，その要求に基づいて不断にソフトウェア・システムの要件がすりあわされて改善していく場合に，結果的に有用性の高いシステムが実現する可能性が高いと思える[20]．管理者がユーザー側のニーズに対する深い知識や責任感をもち，管理者側の要求が常にプロジェクトをリードしているようなプロジェクトの形態をワンマン管理者型と呼ぶことにする．この型は，現実のソフトウェア開発のプロジェクトで多々見られるものであり，相当の成果も遂げてきた．有能なワンマン管理者はソフトウェア開発に関する専門知識を持たなくとも，ユーザーのニーズからプロジェクトを推進させることができ，また実際のソフトウェア開発に当たっては，ソフトウェアの開発技術者の技術を尊重する．さらに，開発技術者との意見交換において，管理者自身も，独自の視点から，ソフトウェア開発の技術に対する知識を理解するようになる場合が多い．

この型は，管理者の意欲，使命感，能力および経験に依存し，条件が整っている場合には成果も期待できるが，管理者の質が悪い場合には，貧弱な成果がもたらされ，あるいは大きな損失に発展することになる．

19. 管弦楽団型

多数の構成員によって共同である成果をもたらすようなチーム編成の典型的な例として，管弦楽団の構成が例示できる．管弦楽団の構成員は，それぞれに独自の技術能力を習得しており，各人の能力を総合して，目的物を実現するという意味で，ソフトウェア開発プロジェクトの管理，進行の過程の分析，あるいはプロジェクト・チームの編成方法の参考になる．

管弦楽団の活動により実現するものは，音楽の演奏であり，音楽は作曲家の創意によって基本的な部分が完成されている．また，作曲家の作品に対する解釈については，指揮者の見解，理解が大きな比重を占める．管弦楽団の演奏者，あるいは歌手は，これらの制約の範囲で，各人の技術能力を発揮することになるが，実際の演奏における独自の問題解決の可能性は限定的である．

ソフトウェア開発においても，作曲家ないし指揮者に当たるユーザーまたは管理者の創意により，作品の製作の枠が定められ，メンバーの技術能力を発現する過程にお

にもたらされるという効果ではなく，ソフトウェア開発の成果の分配を，特定の部門や企業が，生産に対する貢献度よりも大きな割合で得ることができるという効果である．

いても，ユーザーまたは管理者との意見交換が行われ，目的のソフトウェアの仕様が修正され，あるいは発展していけば，プロジェクトが失敗する危険性をある程度回避して，よい成果を実現することができるのではないかと期待される．

20. 演劇劇団型

演劇や，映画やテレビ番組のドラマ制作は，プロデューサー，監督，俳優，効果などさまざまな役割を分担したチームによって担われる．このチームの構成員は，それぞれに製作される成果物について，よく理解し，それぞれの役割分担を受け持って作品が実体化される．ソフトウェア開発において，構成員の役割分担とチームへの参加の動機付け，貢献の総合には，演劇劇団型のチーム構造が参考になると考えられる．

管弦楽団型のプロジェクト形態に比べると，演奏者が作曲者や指揮者に対して実現される自由度よりも，演劇や映画における台本・シナリオに対する演技者の自由度のほうが大きいように見える．ここには，ソフトウェア開発プロジェクトにおいて，当初の製作対象の構想や予測に対するプログラミングの自由度の大きさという角度での共通性を認めることができ，ソフトウェア開発方法論の性格に関する参考点があるのではないかと期待される．

21. スポーツチーム型

野球やサッカーなど，複数の構成員でチームが組まれるスポーツ競技では，監督の意向によりチームの特色（チームカラーなど）が規制されるが，競技の実施時点での判断の多くは競技者に委ねられる．競技進行における監督の影響力は間接的である．このようなチーム構成による協業の事例は，ソフトウェア開発プロジェクトの運営管理のあり方についての参考点を提供するものではないかと期待される．

22. 職人集団型

江戸時代の日本には士農工商の身分制度が存在したが，「工」の身分に該当するのが職人だった．たとえば，建築などについても，近代的な建築工法が導入される以前の職人の建築では，大工や左官屋などの職人が，それぞれに技能を磨き，建築物の完成は職人の個人技の習熟度に大きく依存していた．これらの職人を統合し，プロジェクトの目的遂行のために仕事を纏め上げる役割を担ったのが，棟梁である．ソフトウェア開発においても，それぞれの構成員にはプログラミングなどの技能の習熟が要求され，技能の習熟から組み合わされる製作物のあり方の可能性は非常に多様である．これらの構成員の作業を統合し，全体を構成させるための管理者には，技術的な問題に対する理解に基づく，柔軟な状況の判断が要求される．職人集団型のプロジェクト形式は，ソフトウェア開発を効果的に実現するチーム構成方法論として，多くのヒントを与えるものではないかと期待される．

23. 最後に

ソフトウェア開発プロジェクトをどのように構成し，各人の技能をどのように総合するかは，未解決の問題であり，今後の広範で活発な議論と研究により，理解の深化が実現していくことが強く期待される．ここでも，いくつかの可能性を検討したが，フォード・システムの放棄による喪失感は非常に大きなものである．これを克服して新たな展望を見出すためには，なお，多くの時間と労力必要とされるだろう．ここで重要なのは，単に多くの方法論の提案が現れ，試されることではなく，ソフトウェア開発における設計作業の特性の分析により，開発方法論の成り立ちに一定の枠がはめられ，新たな提案がその枠組みによる具体的な方向性を与えられることである．

もとより，筆者の能力はこれらの問題の完全な解決を展望するにはあまりにも限定的であり，焦燥感すら覚えるものであるが，問題解決の小さな助けになることを希望して，さらに研究を深めたいと思っている．

参考文献

[1] 巫召鴻,「システム開発の理論と実際」, 社団法人情報処理学会　研究報告, 2009-SE-163(31), 2009 年 3 月 19 日
[2] Gary Richardson and Blake Ives: Managing Systems Development,　Computer March 2004,
[3] 下垣典弘,「実践段階に入ったインフォーメーション・オンデマンド」, Information On Demand Conference Japan 2009 基調講演 2，　2009 年 3 月 6 日
http://www-06.ibm.com/itsolutions/jp/solutions/leveraginginformation/events/iodc2009/
[4] 居駒幹夫他 2 名,「シンプルかつ現実的なモデルベース・テストツールの提案」, 社団法人情報処理学会　研究報告，2009-SE-163(38)，2009 年 3 月 19 日
[5] Barry W. Boehm, John R. Brown, Hans Kaspar, Myron Lipow. Gordon J. MacLeod and Michael J.Merritt “Characteristics of Software Quality”, North Holland, New York, 1978
[6] Barry W. Boehm “Software Engineering Economics”, Prentice-Hall, Englewood Cliffs, N.J., 1981
[7] Barry W. Boehm : A Spiral Model of Software Development and Enhancement, Computer May 1988
[8] B, Boehm, “A View of 20-th and 21-stCentuary Software Engineering”, Proc. 28th int’l Conf. Software Eng., ACM Press, 2006, pp.12-29
[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] Jack W. Reeves “Letters to the Editor”, 1992,internet site developer*
[11] Jack W. Reeves, “What Is Software Design: 13 Years Later”, 2005

http://www.developerdotstar.com/printable/mag/articles/reeves_13yearslater.html
[12] Michael D. Coe, “Breaking the Maya Code, revised”, Thames & Hudson, U.S. 1999
[13] 森崎修司,「ソフトウェアインスペクションの動向」,『情報処理』2009 年 5 月号, 情報処理学会, pp.377-384
[14] Michael Ellman, “The Fundamental Problem of Socialist Planning”, Oxford Economic Papers, vol.30, no.2, July 1978, pp.249-262
[15] J.M.Keynes, “The general theory of employment”, Quarterly Journal of Economics, vol.51, 1937
[16] Paul Kimmel, “Special Edition Using Borland C++ 5”, Que, 1996
[17] Ted Faison, “Borland C++ 3.1 Object-Oriented Programming Second Edition”. SAMS, 1992
[18] Frederick P. Brookss, "No Silver Bullet - Essence and Accidents of Software Engineering", IEEE Computer 20, 4 (April 1987), pp. 10-19.
[19] Nevile Homes, “The Agile values and ethical values”, Computer, July 2009, p.100 pp.98-99.
[20] 原敏恭, 『阿修羅になったヤス』, 日本の民事裁判を考える会, 1999 年 8 月
[21] Anthony Hall, “Seven Myths of Formal Methods”, IEEE Software, vol. 5, no. 7, September 1990, pp.11-19

Jack W. Reeves の仮説の理論検証と仮説によって得られる新たな展望の正誤表

3 ページ右 2 行目

誤：合理的に説明することができると書いている[10]

正：合理的に説明することができると書いている[9]

3 ページ右 2 行目

誤：作業計画作業計画

正：作業計画

5 ページ右段 16－18 行目

誤：そもそもプログラム工程の前に「設計」工程を設けて，情報の整理を行う目的は，ソフトウェアの高い品質を維持するためであるが[5]，ソフトウェアの品質は複数の相矛盾する属性によって，多面的に測られるものである [7].

正：そもそもプログラム工程の前に「設計」工程を設けて，情報の整理を行う目的は，ソフトウェアの高い品質を維持するためであるが[7]，ソフトウェアの品質は複数の相矛盾する属性によって，多面的に測られるものである [5].

5 ページ右段下から 4 行目

誤：自己撞着

正：自家撞着