Vol.2010-SE-168 No.4
Vol.2010-EMB-17 No.4
2010/6/1

不完全情報および思惟のあり方から考察するウォーターフォール・モデルの性質

巫　召鴻†

ソフトウェア開発方式としてのウォーターフォール・モデルの欠陥は 1980 年代前半には指摘され，今日ではソフトウェア開発に関連する共有認識ともなっている．1992 年に Jack W. Reeves は，論文"What Is Software Design?"で，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した．Reeves の提案は，彼の仮説の正しさを立証することを目的にしたのではなく，その仮説を承認した場合にソフトウェア開発における従来の問題を，それまでの説明よりもはるかに明確に説明することができるということを示そうとしたものである．筆者は Reeves の提案の核心部分が無視され続けている現状を見て，彼の仮説からの演繹を検証するだけでなく，彼の仮説そのものの正しさを，論証する必要があると考える．本稿では，部分的無知と思考の性質という二つの面からの検証により，Reeves の仮説の正しさの論証を試みる．

A nature of Waterfall model considered from incomplete information and property of thinking

Shaofung Wu†

The defect of the waterfall model as the software development method was first pointed out by the beginning of 1980's, and it has become shared common recognition about software development today. In 1992, Jack W. Reeves proposed a sharp insight about software design of software development in his paper "What Is Software Design?". The objective of Reeves' paper was not a proof of correctness of his proposition, but was to show that problems of software development until then were able to be explained much more clearly, when admitting the correctness of his hypothesis than when without his hypothesis. I believe that it is necessary not only to verify the deduction from his hypothesis but also to prove the correctness of his hypothesis, as I have seen the current situation in which the core part of his proposal of Reeves keeps being disregarded. In this paper, I try proof of the correctness of the hypothesis of Reeves by inspection from two aspects, one is partial ignorance and another is the property of thinking.

1．はじめに

ソフトウェアの規模が大きくなり，複雑化し続けている今日，ソフトウェアの開発における不確定要因の比重は大きくなり続けている[1][2]．ところが，企業などが情報システムのために費す予算を見ると，その 70%は保守運用に割かれ，開発には 30%が割かれるのみであり，その中の 80%を人件費が占めている[3]．さらにその少ない配分の開発予算の中で，50%を超える割合がシステムテストで費やされているという．

情報システム関連の予算において，オーバーヘッド部分ともいえる構成部分の割合が膨張し，開発予算が圧迫され，加えて，経済環境の悪化などにより全体の予算自身も縮小されると，高い品質のソフトウェア・システムの開発が困難になり，これを社会的な観点で見れば，情報システムの有効活用の機会損失であるともいえる．

テストフェーズを含むソフトウェアの開発全般が，ウォーターフォール・モデルを前提とするソフトウェアのライフサイクルのモデルに沿って行われており，それが正しく機能しているものと仮定すると，ソフトウェアのテスト工程に開発費用の半分以上を割かなければならないという状態は，かなり例外的なものであると考えなければならない．また，このような結果はソフトウェア開発方法論から説明できるものではなく，理論的な予測に反し，歓迎しかねる現実として到来した状況であるという[4]．

ソフトウェア開発におけるテスト工程で要する費用の予想外の増大という現象を，ソフトウェアのライフサイクルにおける保守費用の占める極度に大きな割合と合わせてみた場合，この現象の問題点と意味を踏み込んで検討するための切り口は，ソフトウェア開発，システム開発における工程の概念の不明瞭性に置くべきである．つまり立証されておらず，かつ疑問を呈せられていながら通用している前提の本質を理論的に解明できず，不明点を含んだ基礎の上に開発方法論が構成されている現状の検討，分析に求めるべきである．本稿では，このような観点から，ソフトウェア開発モデルを，実証的，経験的な議論にとどまらずに，その立脚点の妥当性の理論的な分析にまで踏みこんで検証する．また，仮説の論証を前提として，ソフトウェア開発における矛盾や問題が累積的に顕在化しがちなステージであるテスト工程の問題に関する初歩的な分析を試みる．

1.　ソフトウェア開発ステージの意味とプロセス・モデル

Barry W. Boehm は，ソフトウェア開発において，ウォーターフォール・モデルが採用される経済的な合理性を，次の二つの仮説に基づいて論考しているa.

† コーナンソフト
K onansoft co., ltd.

a Boeh m[6], pp.38-39

Vol.2010-SE-168 No.4
Vol.2010-EMB-17 No.4
2010/6/1

1．ソフトウェアの製造に成功するためには，ライフサイクルに含まれている各ステージを何らかの形で実施しなければならない．

2．ウォーターフォール・モデルと異なる順序で，これらのステージを実施することは，ウォーターフォール・モデルの順序での実施に比べ，経済性に劣った結果にならざるを得ない．

この前提仮説にはソフトウェア開発の性質に関する暗黙の認識が含まれているb. つまり，ソフトウェア開発は，所与の目的について，計画，要件定義から設計，製造などの経路を通り，情報が転化され，詳細化される過程であるという認識が含まれている．この場合，目的に関する情報の転化，詳細化の前提が特に重要な意味を持つのは，プロセス・モデルにおける設計から製造への移行部分であるが，ウォーターフォール・モデルに基づくソフトウェア開発におけるソフトウェアのテスト工程は，設計工程と製造工程に関する前提の制約を受ける．

ところが，ウォーターフォール・モデルに沿って進行するソフトウェア・プロジェクトのステージ間の移行が一方的な情報の転移の過程であり，新たな情報や意思決定が後の工程で発生し，それが前工程を規定する可能性はありえないとする立場については，有力な反論がある[7]．そして，それに関する議論は，高品質のソフトウェアを効率的に開発するには，どうすればよいかという方法論の奥深いところに踏み込むものである．

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．これは，「上流工程」における完全知識の仮説と表現できるものである．

b Boehm and others[5], p.2-1
"The Software Development Process is the formal process by which program objectives are transformed into program requirements, then into design specifications, implemented into codes, tested, and finally placed and maintained in operational status."

2. ハードウェア設計の検証とソフトウェアの「下流工程」からの手戻り

ハードウェアの設計でも，欠陥が完全に取り除かれる保証があるわけではない．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，それ設計工程の欠陥である．

ソフトウェア製品においても，上記のようなハードウェアの設計の欠陥で見られる現象に相当する不具合や事故が起こることは珍しくない．リリースされたソフトウェア・システムが正常に機能せず，たとえば銀行オンラインがダウンして，ATM が使用できなくなり，座席予約システムで二重予約が発生し，あるいは販売されているパッケージ・システムにバグがあり，そのためにデータが失われる事態が発生することなどが実際に起こってきた．しかし，このようなソフトウェアの事故の例においては，その欠陥の発生箇所，原因を，「上流工程」または「下流工程」のどこかのステージに特定することは，ハードウェアの場合ほど簡単ではない．

この種の不具合とは別に，ウォーターフォール・モデルに基づくソフトウェア開発においては，「下流工程」から「上流工程」への手戻りが発生して，開発プロジェクトが損害を蒙り，あるいはプロジェクトが存続できなくなることさえあるが，これもソフトウェア開発における重要な欠陥の事例として検討する必要がある．たとえば，通常のソフトウェア開発プロジェクトでは，データベースの設計で，項目，プライマリー・キー，テーブル間の相互関係，項目で NULL 値が許されるかなどの指定を「上流工程」段階で決定するが，「下流工程」にいたって，その構造に大きな変更を行わなければ開発を完成することができない事実が発覚する場合がある．同様のことは，処理やモジュールの分割方式，モジュール間のインターフェースなどをあらかじめ定め，プログラミング作業がかなり進行した後に，それらの構造では処理を完遂することが困難または不可能であることが発覚し，構造を手直しするようなことがある．このような場合，欠陥は設計に起因するものであると認識されることが多いが，「下流工程」の作業が「上流工程」の指示により，すべて一意的に定まるのでなければ，「下流工程」で

Vol.2010-SE-168 No.4
Vol.2010-EMB-17 No.4
2010/6/1

発生する上記のような手戻りの原因が,「下流工程」の進捗によって独自に生成されたものであるとする可能性を否定できない．たとえば,「下流工程」がプログラミング工程であるとして，プログラミングの実施における何らかの判断過程において，ある選択肢が採用されたために，ある種の手戻りが発生し，仮に別の選択肢を採用した場合には，そのような手戻り原因は発生しなかったというような場合もありうる．この場合，そのような欠陥（そのような現象を欠陥と認識すべきかどうかは問題でもある）は「上流工程」の対応を必要とするが, 発生は「下流工程」であるということになる.「下流工程」で，複数のプログラミングが並行的に進められている場合には，それぞれのプログラミングの相互の関連の中で, そのような手戻り原因が発生する可能性があり，その組合せの数は大きなものになる．その場合，ある種の手戻り原因が別の選択肢によって避けられたとしても，それが別の手戻り原因を作り出すことになったかもしれず,「上流工程」で規定すべきものと解されているものを含め，問題が完全に解決される時期は，プログラミングの完了からテストの完了までの，すべての検証が終わった時期であったという結果になる事例が一般的であるように見える．

ウォーターフォール・モデルでは，各ステージはソフトウェアのライフサイクルとして列記されている順序に，逐次的に完了していき，次工程に入ったときには前工程は終息するものと考える．しかし，Jack W. Reeves が指摘するように，ソフトウェアの開発工程は，いわゆる「上流工程」の「設計」ステージからコーディング，テストステージまでの全体が，エンジニアリングとしての設計工程に該当するとすればc,「上流工程」として認識される「設計工程」のステージを製造工程として認識されるプログラミングの段階が始まる前に閉鎖することは，理に反する．このような場合，プログラミング工程からテスト工程にいたる段階以降で，いわゆる「上流工程」で行うものとされる「設計」や「要件定義」というような作業が繰り返し実施されていたとしても，工程の実行順序が固定されているのであるから，統計的には，これらの作業の費用はテスト工程で発生したものと計数されることになる．Reeves の指摘より推察されるソフトウェア開発の構造の理解に基づけば，ソフトウェアの開発における費用配分の過半部分がテスト工程に割かれているという認識の内実は，このようなものであるとも解釈できる．

3. Reeves の仮説の論証

Jack W. Reeves の指摘によれば，ハードウェアの製造では，設計ステージは工程として独立しているが，ソフトウェアの開発においては，ハードウェアの設計工程に該当する作業工程は要件定義からテストまでのほぼ全体の工程であり，ソフトウェア開発において，ハードウェアの製作における製造工程に相当する部分はコンパイル・リンクである[7]．しかし，彼はこの仮説を経験的な認識として提案しているのであり，また，彼のソフトウェア開発理論に関する立場は，この仮説が正しいとしたときに，それがソフトウェア開発理論に対してどのような影響を与えるのであるかを検証しようというものである[7][8]d．これに対し筆者は，Reeves の仮説の本体を理論的に検討する必要があると思う．この論理的な検証の試みは，抽象的な議論ではあるが，決して空論ではない．また，説得力の強い Reeves の指摘が 20 年近くも前に提出されているのにもかかわらず，その提案の核心部分が無視され続けているような状況においては，理論的な議論が有効であると考える．ここでは，「上流工程」における知識の不完全性と，ソフトウェア開発における思考のあり方という二方向から，Reeves の仮説の論証を試みる．

4. 「上流工程」における知識の不完全性

プロジェクトで計画を策定する人員が支配される不確実性に関しては，1970 年代末における，社会主義計画経済の機能に関する研究を流用することにより，基本的な問題点を解き明かすことができる．Michael Ellman によれば，社会主義計画経済の基本的な問題は，計画経済を実施するための基礎になっている理論において，現実社会の重要な事実が無視されていることにあり，それは，部分的無知，不正確なデータ処理技術，そして複雑性である[12]．経済計画の策定時に中央当局が完全な知識を有していれば, 中央当局は完全な計画を作成し, 実施当局に計画を頒布することができるが，実際には，中央当局の利用できる知識は，部分的であり不完全である．そのような不完全性は，一方では情報収集と情報処理能力の不完全性，および情報収集の主体者の情報受入れ能力の不完全性（限界）によって発生するものであり，また他方では，人間が将来の事象に関して宿命的に負っている無知，不確実性に起因するものであるe．ところが，不完全情報という現実の状況が無視され，あるいは正確に認識されず，知識が完全であるという前提の下に計画が策定，実施されると，計画実施主体は現実と

c Reeves[7], “The overwhelming problem with software development is that everything is part of design process. Coding is design, testing and debugging are part of design, and what we typically call software design is still part of design.”

d Reeves[7], "This article assumes that final source code is the real software design and then examines some of the consequences of that assumption. I may not be able to prove that this point of view is correct, but I hope to show that it does explain some of the observed facts of the software industry, ...";
Reeves[8], I did not set out to prove that "source code is the design"; I will readily concede that what is a "design" is to some extent a matter of definition.

e Keynes[13], pp.213-14, “the expectation of life is only slightly uncertain. The sense in which I am using the term is that in which the prospect of a European war is uncertain, or the price of copper and the rate of interest twenty years hence, or the obsolescence of a new invention, or the position of private wealth holders in the social system in 1970. About these matters there is no scientific basis on which to form any calculable probability whatever. We simply do not know.”

Vol.2010-SE-168 No.4
Vol.2010-EMB-17 No.4
2010/6/1

計画の矛盾に適合できず，策定されている計画が束縛になり多大な浪費が発生する．ソフトウェア開発の進行についても，特にプロジェクトが大規模であればあるほど，「上流工程」で作成されたドキュメントが完全知識に基づくものであるとする前提があれば，計画経済の機能不全と同様の陥穽に陥る可能性は小さくない．

ハードウェア生産のエンジニアリングにおける設計工程でも，設計者は完全知識を与えられているわけではない．実際にハードウェアの設計において設計者が想定するのは，現実に発生しうる可能性のごく一部分でしかなく，想定する範囲内だけでの条件についての分析を行うことで，ハードウェア製造物の製造過程を詳細に定めることができる．ハードウェアの設計者は，ハードウェア製造物が将来的に出会う現象については不完全な知識しか持たない．しかし，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．

ソフトウェア開発において，ハードウェアの設計工程と同じ条件が成り立たないのは,「上流工程」における不完全知識の対象が，完成した製品にかかわる環境要因についてではなく,「下流工程」といわれるプログラミングやテストのステージの実施状況についてのものだからである．ウォーターフォール・モデルの下では,「下流工程」の局面で「上流工程」で決定する問題を判断することは,開発秩序を破壊するものであり,避けなければならない事態である．ところが，オブジェクト指向プログラムの多相(polymorphism)，後期バインディング(late binding)，あるいは例外ハンドリングなどのプログラミング･パラダイムの展開を見ると,プログラミング作業の時点に至っても,プログラム実行時の状況を完全には予測できないこと，つまり完全知識ではないことが想定されている[14]．この点から見ても，プログラミングを開始する前にすべての条件について完全情報を収集し，プログラミング作業の内実を，手戻りが発生しないほどの厳密さで確定することは現実的ではないし，それを求めることは浪費的であると考えられる．ウォーターフォール・モデルが連想させる，過重なドキュメントの作成と維持という問題は，このような視点から，かなりの程度で説明できる．

ソフトウェア開発においては，Reeves の指摘から演繹できるように,「下流工程」において「上流工程」では論点に上らなかった問題が確認され，それが「上流工程」で確定した設定のいずれかの変更に結びつくという現象は，例外的ではなく，日常的に，ある意味では不可避的に発生すると考えるべきである．この点の検証をさらに深めるために，言語と思考の性格という側面からの分析を試みたい．

5. 設計作業とそれを実体化する記述システム

設計工程と製造工程が分離しているエンジニアリングの工程体系では，設計作業は製造工程における思考や問題解決を総合的に記載したドキュメントを作成し，製造工程はこのドキュメントつまり設計書に基づいて，設計部門の要因とは隔絶された状態で製造作業を完成させる．設計部門の技術者は，情報収集や分析を行い，製造工程の作業を完全に分析し，設計書を作成する．伝統的な用語の意味に従えば，ソフトウェア開発における製造工程はコーディング，つまりプログラミングである．このとき，設計工程で作成される設計書の表現ツールである記述システムの性格を検討する必要がある．

ソフトウェア開発において，伝統的な意味での設計書は，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を集めたものである．しばしば，特別な記述方法や記述体系により，設計の効率が革命的に改善され，ソフトウェア開発の困難が基本的に消滅するというような提案が行われてきたが，設計書とされるドキュメントの記述システムが原理的に変更されたわけではない．また，プログラミング作業が設計工程には含まれないと考えるためには，プログラム言語は設計書を表現する記述システムには含まれない．

これらの設計ドキュメントはいずれもソフトウェア開発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したものであり，その表現形式は，人間の思考を反映する情報の記述システムである．この記述システムには自然言語も含まれる．

記述システムは情報伝達や描写のツールだが，人間の思考は記述システムに依存する.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．この事実は，ある種の記述システムを法則やマニュアルにしたがって埋めていった場合に，設計作業から技術要員の判断を除去することができはしないかという試行につながり，そのような提案が頻繁になされている．

記述システムの表現能力は，人間の思惟の範囲を規定し，また製造工程の作業のマニュアルになるので，設計書を記す記述システムの表現能力は設計の性格を規定する重要な問題である．前述のように，技術者がソフトウェア開発において使用する記述システムには，以下のようなものがある．

１．自然言語
２．書式
３．フローチャート

Vol.2010-SE-168 No.4
Vol.2010-EMB-17 No.4
2010/6/1

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

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，もっとも強力なものであり，すべての記述システムの基礎を成すものである．ソフトウェア開発によって製作されるソフトウェアは，最終的には電子計算機上で動作するプログラムに変換される．プログラムには，電子計算機の構造として設計された思考方法がハードウェアの次元で組み込まれており，人間の思惟の全領域を覆えるだけの広がりを持つものではない．プログラムの論理の根底には，数学や論理学の枠組みがある．論理学は人間の思考を実現する有力な枠組みでもあり，技術者は論理学に沿って大部分の思考を完遂することができるが，自然言語によって探索することが可能な思惟の領域には，論理学の枠組みを越えた部分がある．自然言語の多様性は逆にソフトウェア開発における記述システムとしての不完全性につながる．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だが，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，少なくとも開発作業の効率性の面での問題がある．

どのような方法論においても，自然言語だけで設計書のドキュメントを完成できるとは考えていない．ソフトウェア開発において，自然言語は必須の表現ツールであるが，設計書の本体にはそれ以外の記述システムの存在が想定されている．

プログラムを機械語まで還元し，処理と分岐の命令の集合である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考えられるが，筆者も，このような考えにとらわれ，1980 年代にはデシジョンテーブルでソフトウェアの設計書を作成した経験がある．しかし，結果は思わしくなく，その原因を長く究明することができなかった．現在では，プログラムの性質を，デシジョン・テーブルのような分岐表に置き換えることができるという発想が，誤っていたと考えている．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が，プログラムの性質を包括的に表現する，あるいは指示する記述システムとしては，不完全である． たとえば，すべてのプログラムは最終的には機械語に変換されるのであるが，機械語はアセンブラ言語のソース・コードに一対一に対応している．しかし，アセンブラー言語によるソフトウェア開発を有効な開発方法と考える人はほとんどいないし，その理由は費用効率だけの問題ではない．

フローチャート，グラフあるいは数式は，表現能力に限界があり．それだけでドキュメントを形成することはできない．他の記述システムの補助を必要とする．あるいは，他の記述システムの補助的な記述の役割を，効果的に果たすことができる．

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，しかも，さまざまに改良を加えられてきており，今後も変化する諸条件に適合して，改良が重ねられ，またいくつものバリエーションに展開していくことが期待される．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．つまり，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必然性を論証している研究は見当たらない．少なくとも，前述の Boehm のウォーターフォール・モデルに関する仮説が，それを論証していると受け取ることはできない．プログラミング言語に依拠する思考が，ソフトウェア開発におけるもっとも強力な思考であることを考慮すると，ソフトウェア開発の初期段階でも，プログラミングのイメージを描くことが可能になったときには，プログラム言語を離れて検討を行わなければならないという必然性はない．

ウォーターフォール・モデルを前提とする場合，ソフトウェア開発の「上流工程」では技術要員は不完全な知識しか持たず，ソフトウェア開発作業が終了するまでのすべての局面を予測することはできない．また，「上流工程」で使用することを想定されている記述システムは，「下流工程」で使用することを想定されるプログラミング言語に比較して，記述能力の面で不完全なものであり，そのような記述システムによって記述される情報によって，表現能力が豊かで，明瞭で，強力な記述システムであるプログラミング言語による思考および記述処理を確定しなければならないと考えることは，合理的でない．もし，「上流工程」によりプログラミング作業を完全に確定できるのならば，プログラミング言語という記述システムを使用してそれを記述するほうがはるかに容易で効率的である．実際に，多くのプロジェクトにおいて，プログラミング工程を詳細に規定するものとして作成されるはずのドキュメントが，プログラミングの終了後に，ソースコードから作成しなおされるということが，頻繁に行われているf.

f Reeves[7], “As just a small point, all programmers know that writing the software design documents after the code instead of before, produces much more accurate documents, The reason is now obvious. Only the final design, as reflected in code, is the only one refined during the build/test cycle. The probability of the initial design being

Vol.2010-SE-168 No.4
Vol.2010-EMB-17 No.4
2010/6/1

「上流工程」でプログラミング言語を使用して「仕様」を記述するのであれば，「下流工程」として，プログラミング工程が独立して存在する意味はない．これらの点から見て，ウォーターフォール・モデルのソフトウェア開発において，プログラミングの前に，プログラミングから隔絶した「設計」工程が存在し，プログラミングはその工程の終結後に，その工程で作成されたドキュメントの記述にしたがい，その範囲内で行わなければならないとすることは，方法論的な誤謬であると思える．

6. テスト工程の問題

Reeves の仮設を前提にした場合には，ウォーターフォール・モデルにおけるテスト工程も，誤って理解されていることになる．その点について，検討したい．

そもそも，ソフトウェアの品質はある単独の要因についてその程度を何らかの方法で測ることで，順位付けできるものではない．ソフトウェアの品質を決定する特性には複数のものがあり，それは互いに矛盾することもあるので，どの特性に焦点を当てるかにより，同じソフトウェアでも評価が異なる [5].

ソフトウェアの品質評価は，複数の特性を評価して，それらを総合するべきであるが，ウォーターフォール・モデルを前提とするテスト工程では，正確性，安全性，信頼性，効率性などの特性を重視して評価が行われる．これに対し，プログラムの可読性，移行性，一貫性などの特性は，レビューやインスペクションによって評価し，改善するという認識が一般的である[11]．しかし，ソフトウェア・インスペクションなどの工程のソフトウェアライフサイクルにおける構造的な位置づけや，その有効性については十分に研究されているとは言えず，未解明の点が多い．また，その効果に関する評価は実施者の主観または自己申告によるものであり，客観的に効果を比較することが困難である．

いずれにしろ，プログラミングが終了した後に製品としてのソフトウェアを出荷またはリリースするための品質を保証するための工程として，テスト工程があり，この工程ですべての問題が最終的に解決していなければならない．この工程における方法論の混乱は深刻な悪影響を及ぼすことになるが，その理論的な混乱を典型的に示す具体例から問題の分析に迫りたい．

7. 網羅率

網羅率とは，プログラムに存在する分岐によるすべての経路の数を分母とし，テストの実行で通過した経路数を分子として得られる割合である．この指標にはテスト工程の品質を判定するための限定的な材料を提供できる面もあるかもしれないが，テストの性質に関する重大な誤解を抱かせる原因になる危険性をはらんでいる．ソフトウェアの開発方式に関する基本理論が混乱している中で，テストに関する評価を行い，ソフトウェア製品の信頼性，安全性を保障することは，プロジェクト管理の他の管理作業にも共通することでもあるが，非常に困難である．これを客観的な計数値，つまりメトリックによって評価できれば，管理作業の負担が大幅に減殺され，結果としてプロジェクトに要する人的資源の程度を質的にも量的にも少なくすることができ，ソフトウェアの生産効率を増大させると予測できる．しかし，メトリックの限界を理解せずに使用することは，メトリックの有用性を相殺して余りある弊害をもたらす危険性をはらむ．ここで議論を全て展開することはできないので，留意点を挙げることにとどめる．

プログラムにおける分岐は，プログラムの各ブロックを構成させる根拠になる命令であるが，それ自体はプログラム言語の一部であり，プログラムにおけるブロックを構成させない種類の分岐も存在する．プログラムの分岐命令が正しく機能するかというテストであれば，これはコンパイラーのテストであるが，いわゆるテスト工程で実施しなければならないのはコンパイラーはなく，コンパイラーで作成されるプログラムのテストである．

総合的な意味で，適切に十分に実施されたテスト作業と，不適切に不十分にしか行われなかったテスト作業があるとき，両者の網羅率をそれぞれ測定できるとして，その値を比較したら，前者のほうが高い値を示すという推測は，正しいように思えるg．しかし，その値を評価の参考の材料として以上に使用することは危険である．

テストの検証体制が整い，テスト結果の提出物に網羅率が指定され，その意味が正しく理解されないままに網羅率の一定値が合否判定基準であることのみが共通的に認識された場合には，このメトリックの存在が，テストの実施者側に対しても評価側に対しても，誤ったテスト基準を形成させる強い原因になる可能性がある．部分的な実行が難しい複雑なプログラムにおいて，特にマクロやジェネレーターなどで生成されたソース部分がある場合には，網羅率を高くすることがテスト作業に過重な負担を与えることがある．このような場合には，網羅率を達成することによって得られる成果と,そのために支払う費用の大きさとをトレードオフで取捨選択しなければならない．しかし，総合的な理解能力が欠如したテスト実行者やテストの評価者が，評価基準としての取り扱いが容易なメトリック値を優先すると，メトリックの数値を高めるためだけの作業に大きな労力が割かれることになり，このような労力はテストの品質を向上させず，ソフトウェアの品質を向上させないので資源の浪費になる．

unchanged during this cycle is inversely related to the number of modules and number of programmers on a project. It rapidly becomes indistinguishable from zero.”

g ただし，実証的に証明されている研究があるかどうかについて，筆者は知らない．

Vol.2010-SE-168 No.4
Vol.2010-EMB-17 No.4
2010/6/1

8. すべての場合を包括するテストケース

筆者がテスト方法に関するもう 1 つの誤解と考えているものに,「包括的なテストケースを作成することにより, 完全なテストを行える」という着想がある. 極端な場合, 包括的なテストケースは, プログラミングとは無関係な, ソフトウェアの機能の客観的な観察から導き出されるとみなされる. これは, ソフトウェアの設計が, 最終的な記述システムであるプログラミング言語とは異なる記述システムによる思惟と表現によって作成され, プログラミング言語によるコーディングはそれらの思惟の結果をコンピューター上に実装する情報転化の技術過程であるという観点から演繹すると, 当然のように導かれる原理である.

前述のとおり, ソフトウェアにおけるプログラミングは, 非コンピューター的な記述システムによる人間の自然な思考をコンピューターが理解できる非人間的な記号に置き換える処理ではない. プログラミング言語以外の記述システムで思考し, 記述するソフトウェアは, 情報記述の面で不完全であり, そのような不完全な理解に基づいて作成するテストケースが, プログラミング言語による思考と記述に基づいて作成するテストケースよりも優れているという観点は, ハードウェアのエンジニアリングのアナロジーに起因する誤解である.

実際には, プログラミングを無視してテストケースを作成すると, 膨大な, 効果のないケースが作成され, テストの信頼性と効率は低いものになる. 信頼性を維持するために, すべての可能な場合に関するテストケースを機械的に作成しようとすると, テストケースの数が非常に大きなものになり, 使用に耐えないものになる. 厳密に考察すれば, 不完全な知識しか持たないテスト実施者あるいはテスト計画の策定者が, すべての起こりうるテストケースを完全に記述できるという発想は, 原理的な誤謬を含んでいる. すべての起こりうるテストケースが有限の件数に納まるのか否かも, 一概には確定できない. 仮に, 絶対的な能力の保持者がテスト計画に当たり, 有限件のテストケースですべてのテストケースを網羅することができたとしても, 件数がある程度を超えて大きくなった場合には, テストケースとして使用できない.

つまり, テストケースは, 超越的な操作によって作成するものではなく, プログラムの処理に応じて作成するものである. しかし, 仮にそのような作成方法を採ったとしても, テストケースの作成には, 判断と取捨選択が必要である. たとえば, ある処理Ｐが５つのブロックを直列に実行するものである場合で, 各ブロックを以下の記号で表すものとする.

$$P = \{P_1, P_2, P_3, P_4, P_5\}$$

このとき, 各ブロックを単体でテストすることができるとして, それぞれのブロックのテストケース数を Ti とし, つぎのように表されるとする.

$$Ti = T(Pi)$$

このとき, 各ブロックを単体でテストするテスト作業のケース数を Pu で表すと, 各ブロックのテストケースの和となる.

$$Pu = \Sigma T(Pi)$$

たとえば, 各ブロックのテストケース数が一律に 20 件であるとすれば,

$$Pu=100$$

である. しかし, 処理Ｐを全体でテストしようとすると, テストケース数 Pc は, 場合の数になり, 各ブロックのケース数の積になる.

$$Pc=\Pi T(Pi)$$

この数は, たとえば, 各ブロックのケース数が一律に 20 件だとすると, 2^5=320 万件となり, このようなケースをすべてテストすることは不可能である.

ところが, ブロックを単体で実行する条件を整えることが困難な場合には, モジュールを結合してから, テストを実施せざるを得ない場合が, 頻繁に出現する. このような場合には, すべてのケースをテストすることにより, プログラムや処理の完全な検証を行うという考え方は, 捨てなければならなくなる. 実際には, そのような環境下のテストにおいても, テスト担当者の取捨選択と, 処理やプログラムの性質, 使用形態などを個別に検討することにより, ごく限られた実行により, 全体の評価を下すことができるし, 下さなければならないのである.

9. テスト工程の分析

テスト工程の実施について, このような基準が暗黙裡にあるいは明示的に前提とされがちになり, それがテスト工程の円滑な実施を妨げ, 混乱や停滞が発生するのは, 珍しいことではない. テスト工程の実施について, このような数量的な基準が何ゆえに支配的な力を持ちうるのか, あるいは発揮しがちなのかについては, Reeve の仮説

とウォーターフォール・モデルの前提を比較検討することにより，説得力のある説明を得ることができる．ウォーターフォール・モデルの仮説に沿って考えると，テスト工程は設計工程と製造工程が完了した後に発生する，設計に基づく製造が完了した製品が一定の基準を満たして完成しているかどうかを確認する工程である．この目的を達成するためには，製造物が一定の基準を満たしているかどうかの数量的な規矩が与えられる必要があり，テスト工程に従事する要員には，マニュアル化されたテストの実施要綱が与えられ，静的に，客観的に製造物の合否が判断でき，それを公示することができなければならない．網羅率や包括的なテストケースのような，マニュアル化が容易に見える基準が提案されたときに，その基準が特別な意味を持ちがちになり，そのためにテスト工程の実施の対する弊害が発生する必然性は，この前提に起源を発するものである．

ところが，Reeves の仮説では，テスト工程はハードウェアのエンジニアリングの基準で言えば，設計工程に含まれる[7]．Reeves の仮説にしたがえば，ソフトウェア開発でハードウェア製作の製造工程に対応する工程はコンパイル・リンクであり，これはほとんど費用を発生させずに実施することができる．コンパイル・リンクの後のテスト工程では，品質の検証が静的に行われるのではなく，ソース・コードの変更が繰り返し行われ，機能の訂正や変更が継続する．このような工程を，エンジニアリングの観点で分析すると，設計工程に含まれるものになる．設計作業においては，客観的でマニュアル的な数量評価基準に従って，要員が判断をせずに規則的に作業を完遂できると期待する人はあまりいない．この前提を踏まえていれば，テスト工程の作業を無用心に単純労働化し，結果を数量的に把握しようとする動機は克服される．しかし，多くのプロジェクトにおいて，テスト工程に対する一面的な数量基準などが課せられがちになる．これは，テスト工程の性質に関するウォーターフォール・モデル的な工程観が，少なくとも暗黙的には，前提とされているからに他ならない．

10. 結論

大規模なソフトウェアの開発を組織的に行うためには，ウォーターフォール・モデルで採用されるものと類似の開発体制をとらなければ，プロジェクトの編成自体が困難である．ところが，プロジェクトの初期工程では，不完全な情報しか得ることができず，また，初期工程，あるいは「上流工程」で使用できる記述システムも，プログラミング言語に比して，不完全なものである．その結果，いわゆる「下流工程」において独自に，あるいは新たに発生する問題により，「上流工程」で決定した事項の変更が必要になる事態が，日常的に発生することが予測される．ウォーターフォール・モデルでは完全知識を前提にしているので，「上流工程」は「下流工程」の開始直前に終息するものとみなされ，「下流工程」は「上流工程」の決定事項を遵守しなければならないと考える．この結果，「下流工程」も「上流工程」も状況の変化に対応することができなくなり，プロジェクトの失敗や，資源の浪費が発生することになる．

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,　Co mputer 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 , Myr on L ipow. Gor don J. M acLeod and M ichael 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] 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

[8] Jack W. Reeves, “What Is Software Design: 13 Years Later”, 2005
http://www.developerdotstar.com/printable/mag/articles/reeves_13yearslater.html

[9]George Gamow “One two Three… Infinitive”, Dover Publication Inc., New York, 1947

[10]Michael D. Coe, “Breaking the Maya Code, revised”, Thames & Hudson, U.S. 1999

[11]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

[12]Michael Ellman, “The Fundamental Problem of Socialist Planning”, Oxford Economic Papers, vol.30, no.2, July 1978, pp.249-262

[13]J.M.Keynes, “The general theory of employment”, Quarterly Journal of Economics, vol.51, 1937

[14]Paul Kimmel, “Special Edition Using Borland C++ 5”, Que, 1996

[15]William Roetzheim, “Programming Windows with Borland C++ 4.5, ZD press, 1994