WATERFALL Model 再考

ソフトウェア開発管理とソフトウェア開発方法論の融合について

北陸先端科学技術大学院大学　情報科学研究科
落水　浩一郎

本フォーラムの目的は、W.Royceが論文[1]で提起した大規模ソフトウェア開発管理に関する問題を、ソフトウェア工学40年の歴史（その後の技術発展やコンセプトの進化）をふまえつつ、再度検討することにより、今、明日、何をすべきかについて考えることであると理解する。まず、W.Royceの論文内容を著者なりに要約する。つぎに、1970年のW.Royceの論文を基準点として考え、彼が提起した問題に対して、その後どのような解決が試みられたかを整理する。最後に著者なりの問題提起を行い、フォーラムにおける議論のきっかけの一つにしたいと思う。

１．W. Royce による”Managing the Development of Large Software Systems(1970)”の紹介

彼の論文の主旨は、大規模ソフトウェアの開発管理をどのようにして実現するかという提案と、そのような開発手段を採用したときに起こる問題をどのように解決するかという2点である。まず、図１に彼が論文で導いた結論を示す（図１）。

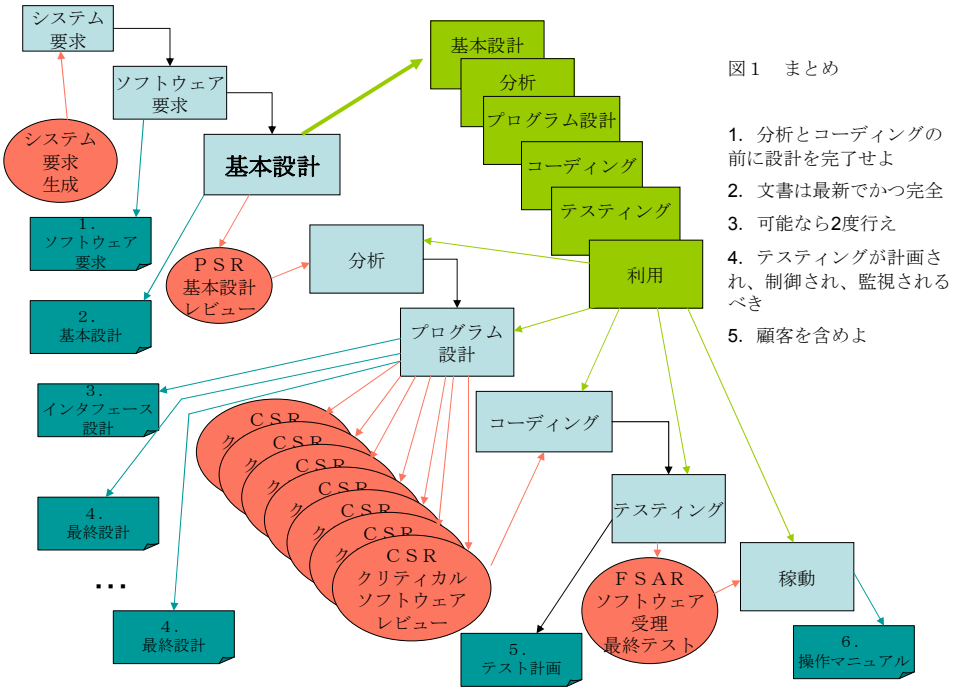


図１　まとめ

1. 分析とコーディングの前に設計を完了せよ
2. 文書は最新でかつ完全
3. 可能なら2度行え
4. テスティングが計画され、制御され、監視されるべき
5. 顧客を含めよ

図１にいたる、彼の思考の展開を論文の流れに沿って追ってみよう。

(1) プログラムの規模や複雑さにかかわらず、どのようなプログラムをつくるべきかとい

う「分析」と、それをどのように「コーディング」するかというステップは、プログラマにとって本質的なものである。また、最終プロダクトに直接貢献する創造的なステップとして、顧客はよろこんで報酬を出す。しかし、この簡単なモデルは、労力が十分に少なく、最終プロダクトが開発者によって稼動される場合のみ（すなわち内部利用の場合のみ）有効である（図２）。

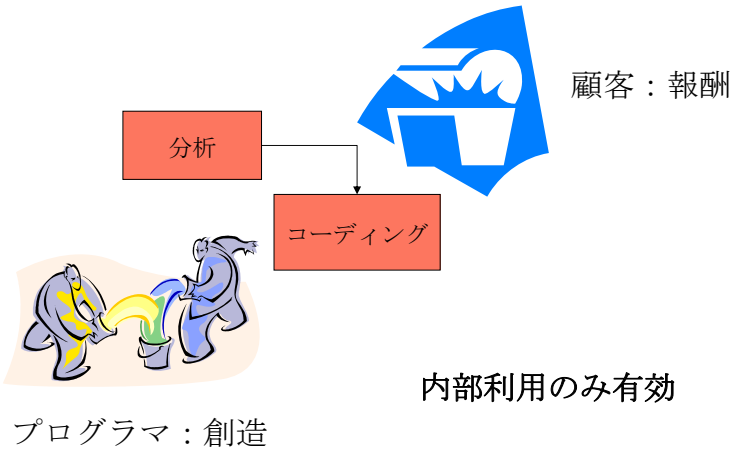


図２　内部利用のための小規模なプログラムを引き渡すための実現ステップ

(2) より規模の大きいシステムでは、この二つのステップのみではプロジェクトは失敗に終わるので、最終プロダクトに直接貢献しない多くの付加的な開発ステップが必要になる（訳注：最終プロダクトに直接に貢献しない付加的な開発ステップという表現の真意は不明である）。それらのステップはコスト増をまねく。また、顧客はあまり報酬を支払いたがらないし、開発担当者はそれらのステップを実行したがらない。管理の第一義の機能は、これらのコンセプトの必要性を双方のグループに納得させ、また、開発担当者には従うことを強制することである。それらのステップは、順に「システム要求定義」、「ソフトウェア要求定義」、「分析」「プログラム設計」、「コーディング」、「テスト」となる。「システム要求定義」、「ソフトウェア要求定義」、「プログラム設計」、「テスティング」の付加的ステップは、実行される手段が「分析」や「コーディング」とは異なるので、分離して処理される（図３　実行順序を示す矢印を除く部分）。

(3) ７つのステップに、「詳細化のために、順次実行する」、「一つ前のフェーズに後戻りする」という実行順序に関する情報を付加する（図３）。すなわち、「各ステップの進行につれて、設計内容はより詳細化される」、「一つ前のステップへの後戻りは起こるが、数ステップ前への後戻りはめったに起こらない」という仮定を置く。この考え方のよい

点は、設計が進行するにつれ、変更が管理可能な範囲に限定されることである。すなわち、予期せぬ困難が発生した時、どこに戻るかを指示している、安定した、移動するベースラインが近距離にある。

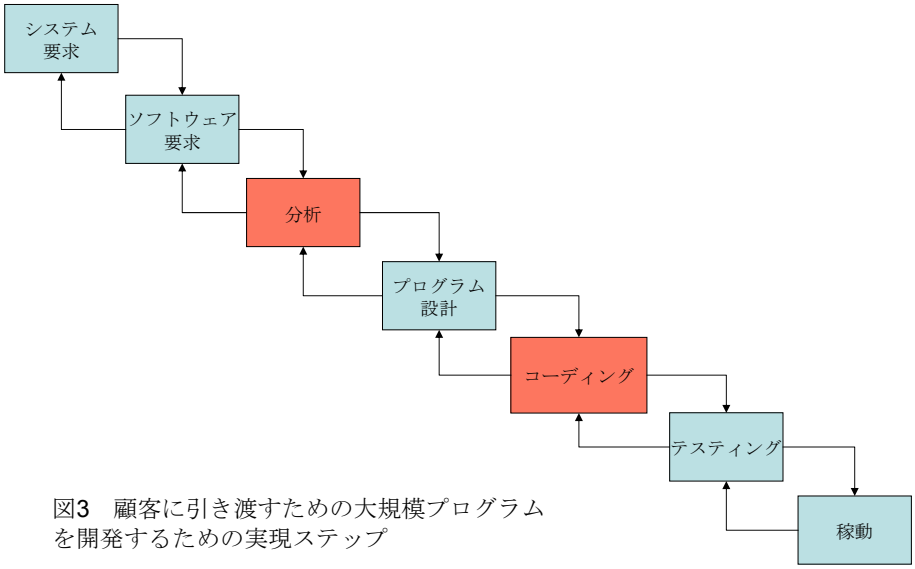


図3　顧客に引き渡すための大規模プログラムを開発するための実現ステップ

(4) 上記仮定をそのまま実現するのは危険であり、失敗をまねく（訳注：順次的なステップの実行、すなわち、ベースラインの移行が現実的な仮定ではないことを彼自身は認識していたことになる）。

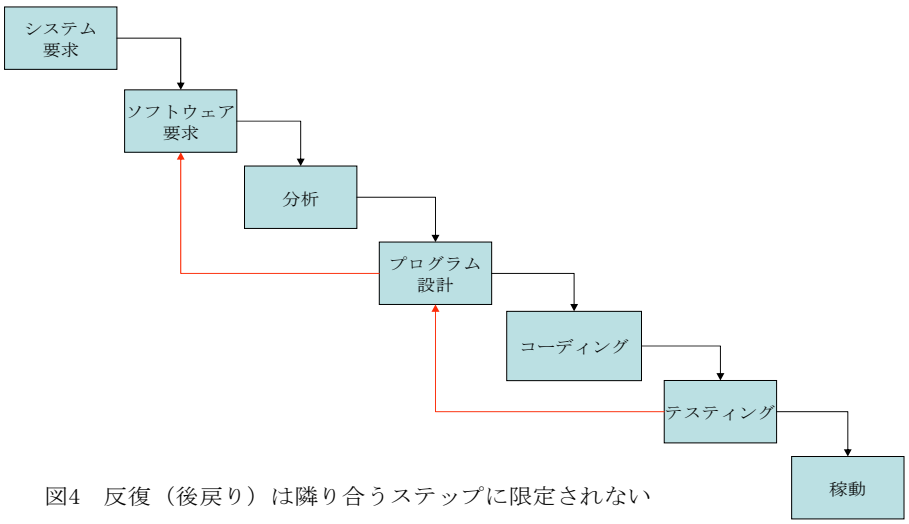


図4　反復（後戻り）は隣り合うステップに限定されない

何故なら、開発サイクルの最後にくるテスティングでは、タイミング、記憶容量、入出力転送などに関して、分析段階とは異なる現象が認知される。これらの現象を厳密に解析することはできない。さらに、これらの現象が様々な外部制約を充たすことに失敗した場合は、再設計が必要となる。必要とされる設計変更は破壊的なものであり、これはさらに、設計に根拠を与えている要求に違反することになる。この結果、「テスティング」、「プログラム設計」、「ソフトウェア要求」という大幅な手戻りを生じ、要求が変更される場合も、本質的な設計変更がある場合も、100%のオーバーランをコストやスケジュールに引き起こしてしまう（図４）。

(5) それにもかかわらずこれらのアプローチは基本的には健全なものであり、これらの開発上のリスクを除去するような以下の５つの特徴をつけ加えることで対応できる。

(5－1)「基本設計」フェーズの導入

「ソフトウェア要求定義」と「分析」の間に基本設計を挿入する（図 5）。

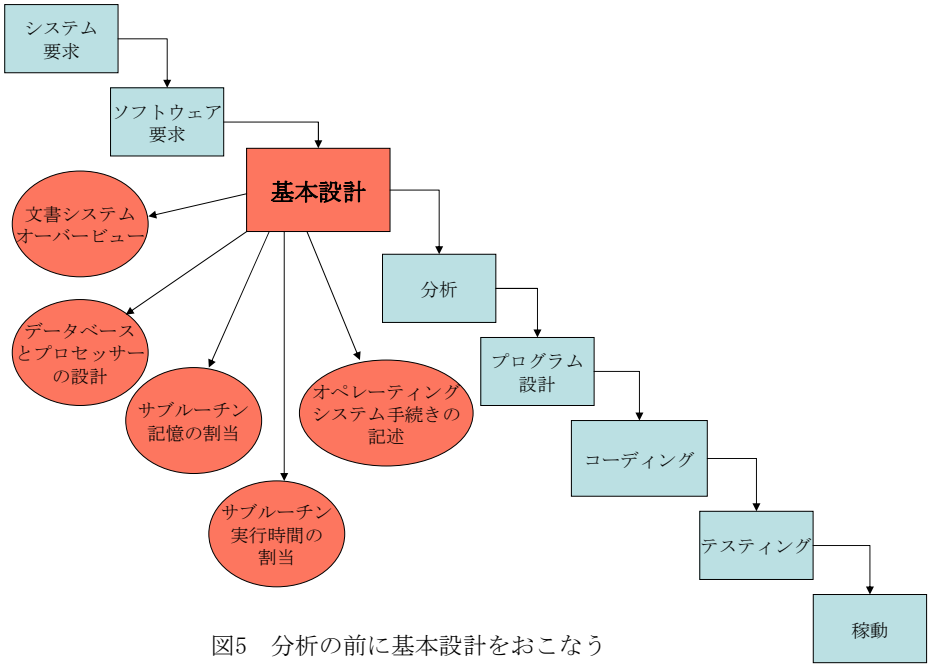


図5　分析の前に基本設計をおこなう

これは、「要求は存在するが分析結果が存在しないという相対的真空状態のなかで設計することをプログラム設計者に強制する」という批判を生み出しうる。この批判はあたっているが一つポイントをはずしている。基本設計を行うことにより、プログラム設計者が記憶容量、タイミング、データの不安定さによっては失敗しないことを保証できる。なぜなら、次の分析フェーズで、プログラム設計者は記憶容量、タイミング、稼動に関する制約を解析の課題に加えることができるからである。すなわち、タイミング、記憶容量、稼働に関する制約を考慮した「分析」を可能にする（訳注：これが納得のいくレベルで実施できる技術はまだ開発されていないと思う）。基本設計フェーズは以下の 3 点からなる。

・　基本設計プロセスを、分析者やプログラマとではなく、プログラム設計者と実施する
・　状況が悪いというリスク下でもデータ処理モードを設計し、定義し、割り当てる：　処理と機能を割当、データベースを設計し、データベース処理を定義し、実行時間を割当、オペレーティングシステムとのインタフェースと処理モードを定義し、入出力処理を記述し、基本的な稼動手順を定義する。
・　理解容易であり、情報に富み、かつ最新のオーバービュードキュメントを書く：すべての作業者はシステムについての基本的な理解を持つべきである。少なくとも一人は、オーバービュー文書を書き、システムについての深い理解を持つべきである。

(5－2) 設計結果を文書化する

ここでドキュメントをどの程度書くべきかという問題を提起する。「かなり書く必要がある」というのが著者の考えである。「かなり」とは、プログラマ、分析者、プログラム設計者が自身の工夫で書こうとするものより多いことを意味する。ソフトウェア開発を管理するための最初のルールは文書化要求を冷酷に課すことである。文書は受理できる水準にする必要がある。高水準の文書化なしにはソフトウェア開発管理は不可能である。5百万ドルのハードウェア装置には30ページの仕様書、5百万ドルのソフトウェアには1500ページの仕様書が必要である。何故そのように大量の文書が必要なのであろうか？

- 各設計者は、インタフェースをとるため、他の設計者とコミュニケーションする必要がある。また管理者、場合によっては顧客とコミュニケーションする必要がある。話し合いの記録だけでは決定事項を管理するのには不十分である。受理でき得る記述は設計者に明白な立場をとり、形ある完了の証拠を与えることを強制する。これにより90％シンドロームを回避できる。
- ソフトウェア開発の初期のフェーズでは、文書は仕様であり設計である。コーディングが始まるまでこれらの３つの言葉（文書、仕様、設計）は単一のものを示す。文書が悪ければ、設計も悪い。もし文書がまだ存在しなければ、設計もまだ存在しない。
- 良い文書の値打ちは、テスティングから稼動、再設計にいたる開発工程の下流で認識される。
  （ア）テスティングフェーズでは、良い文書は、管理者が担当者にプログラム中の誤りに集中することを可能にする。良い文書がない場合は、そのプログラムを理解している唯一の人間である誤りを犯した人によって、誤りは解析されることになる。
  （イ）稼動フェーズでは、良い文書はオペレータを稼動に専念させうる。良い文書がない場合には、それを書いた人が稼動させることになる。一般にプログラムを書いた人はそのオペレーションに関心がないので仕事を効果的に実施できない。
  （ウ）最初の稼動に引き続いて、システム改良の順番になったとき、良い文書は効果的な再設計、更新、改造などを可能にする。

図６に、各ステップのキーとなる文書化プランを示す。６つの文書が生成され、引渡し

時期に文書１、３、４、５、６が更新され最新化されることに注意して欲しい。

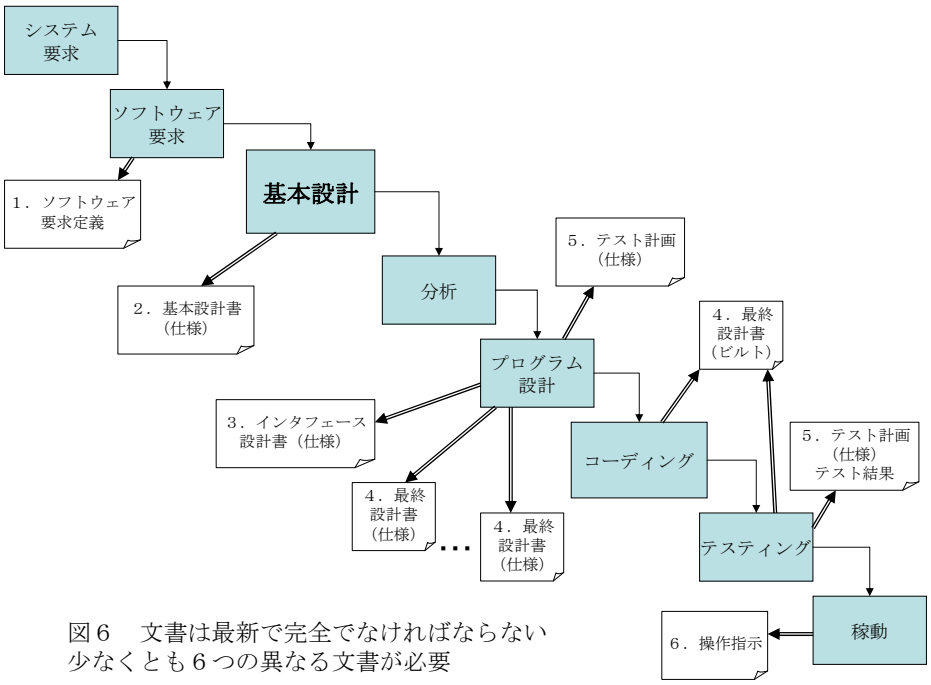


図６　文書は最新で完全でなければならない
少なくとも６つの異なる文書が必要

(5－3) 基本設計以下を２度行え

はじめてのシステムの場合、運用のために顧客に最終的に引き渡される版は第2版であるようにする。開発労力が30ヶ月の場合は10ヶ月、12ヶ月の場合は3ヶ月を開発のシミュレーションにあてる。設計における問題点を検知しモデル化せよ、代替案をモデル化せよ、不要な詳細を無視せよ、最終的には虫のないプログラムを達成せよ。この結果、タイミング、記憶容量その他の判断事項はある程度正確に理解できる。このように一度目は問題発見に力を注ぎ、2度目に解決策を組み込む（図7）。

(5－4) テストの計画と制御と監視

（マンパワーであれ、ＣＰＵ時間であれ、管理者の判断であれ）プロジェクト資源の最大の消費者はテストフェーズである。テストは、コストやスケジュールに関して、最大のリスクを持つフェーズである。それはスケジュールの最終点でバックアップする代替物がほとんど利用できないときに起こる。5－3までに述べた３つの推奨：分析やコーディングのまえに基本設計を行う；文書化を完全に行う；パイロットモデルを構築する；はすべて、テストフェーズに入る前に問題を発掘し解決することに狙いがあった。しかしながらこれらのことを行っていてもテストフェーズが残っており、そこでなされるべき重要なことがある。図８にテスティングに関する視点を示す。

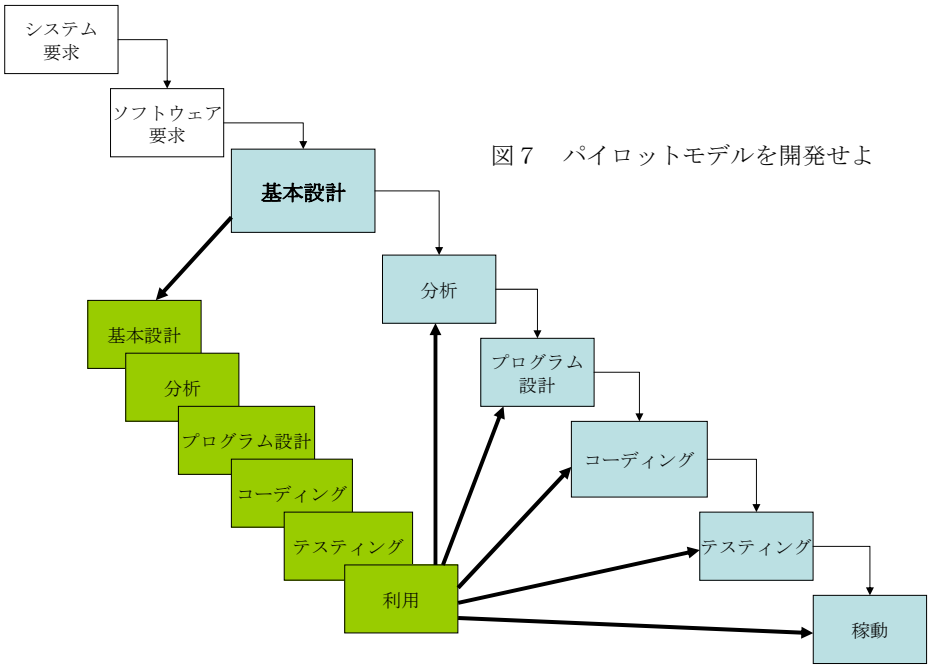


図７　パイロットモデルを開発せよ

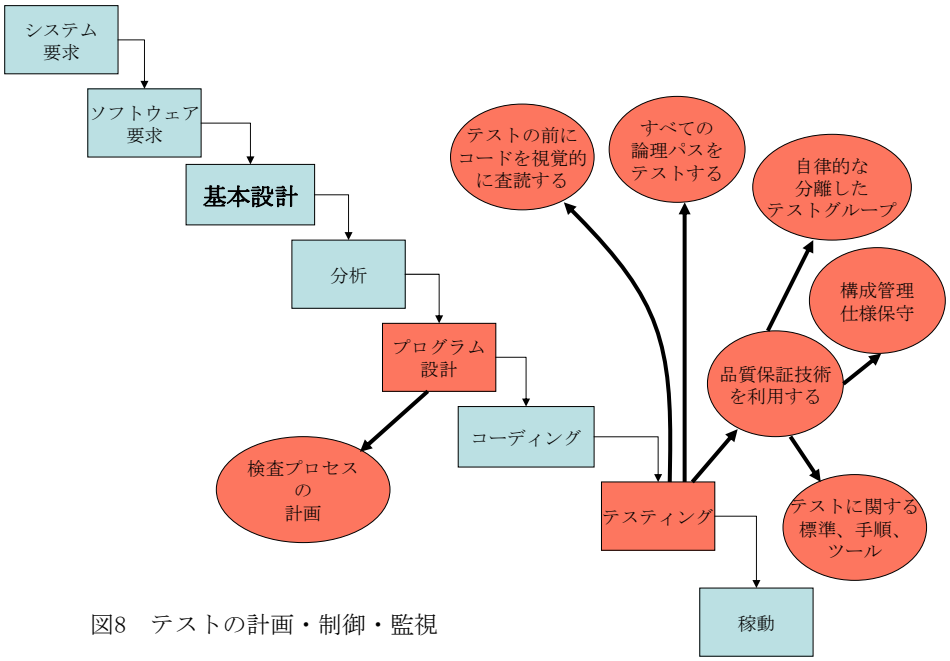


図8　テストの計画・制御・監視

テスト計画を立てるとき、以下の点を考慮することを奨める。

- テストプロセスの多くの部分は、テストの専門家によって最もうまく処理される。テスト専門家は必ずしもオリジナルな設計に寄与していなくてもよい。もし、設計者しかテストをできないことが問題になる場合は、文書化が不十分な兆候である。

- ほとんどの誤りは、視覚的な検査により容易に特定できる。すべての分析結果やソースコードは第3者の視覚的検査を受けるべきである。コストがかかりすぎるので、この目的で計算機を利用しないほうがよい。
- プログラム中のすべての論理パスをテストせよ。巨大で複雑なプログラムの場合、これは大変困難であり、ときには不可能な場合もある。しかし、カバレッジ100%の必要性を主張しておきたい。
- すべての単純な誤りを除去したのち（これがほとんどであるが）、点検のため、ソフトウェアをテストにまわせ。

（5－5）顧客を含めよ

最終引渡しの前、早い時点で顧客に責任を持たせるため、顧客を開発に公式にふくめることが重要である。「要求定義」と「稼動」の間で契約者に自由な裁量を与えることはトラブルを招く。要求定義の後にくる、顧客を含めるべき３箇所を図９に示す。

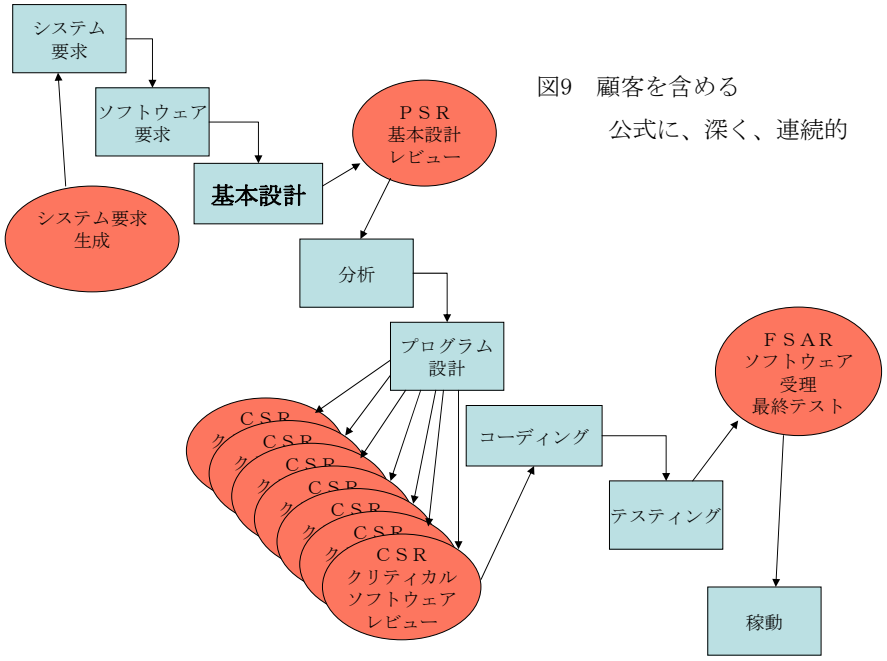
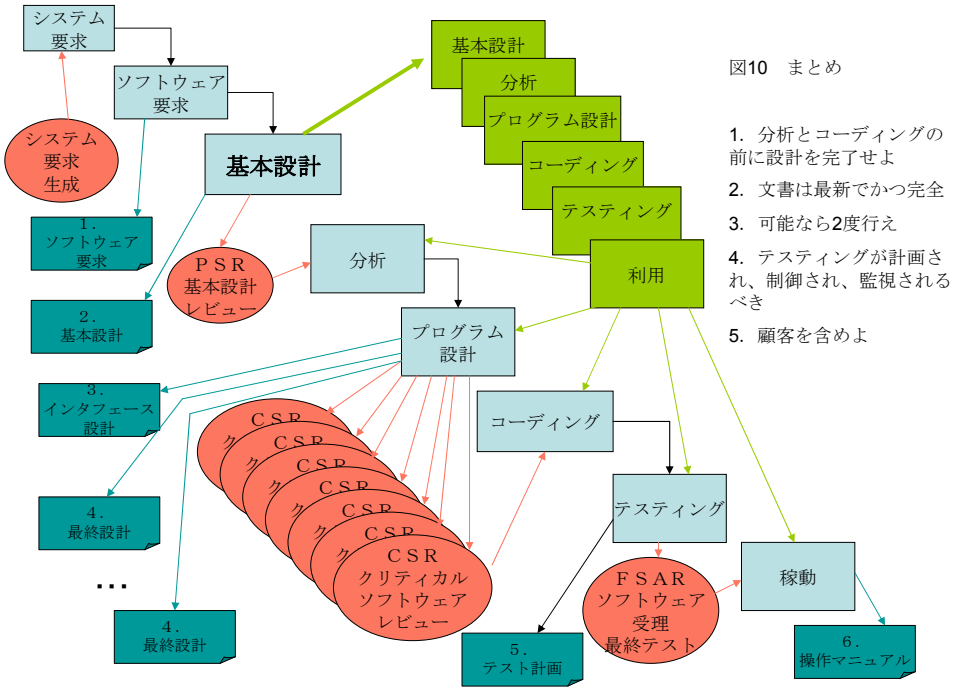


図9　顧客を含める
公式に、深く、連続的

（６）まとめ

大規模ソフトウェア開発の経験に基づいた以上述べてきた考察の結果として、冒頭（図１）に示したウオーターフォール・モデルを提案する（図10に再掲）。図10に、リスキーな開発プロセスを所望の製品を提供する開発プロセスに変換するのに必要な５つのステップをまとめ、それらを統合したものを示す。それぞれが幾分かのコスト増をまねくことを強調しておく。より簡単なプロセスを採用すれば余分なコストはかからないが、大規模なソフトウェア開発ではうまくいかない。



つまり、彼は1970年の時点で、あとで問題になるかなりのことを意識的（あるいは無意識に）予想し、それなりの対応策を提案していたことになる。彼の提案にはウォータフォール・モデルに限定されない、ソフトウェア開発が内蔵する様々な問題点が含まれている。彼が論文中で提案した解の是非も含めて、その後のソフトウェア工学の発展（試行錯誤？）の歴史をたどってみよう。

２．それ以降の技術発展

ソフトウェア開発方法論とプロジェクト管理は、ソフトウェア開発プロジェクトの成功にとって、車の両輪の関係にある。それらをまとめるものがソフトウェア開発パラダイムである。ソフトウェア開発方法論の研究では、ソフトウェアの望ましい「構造」を模索しつつ、それらの構造を作りこむ手段を検討する。追求されてきた「構造」の属性として、正しさの確認容易性、変更波及の局所化、作業分担の容易性、再利用容易性、進化容易性などがあげられる。一方、プロジェクト管理は、方法論を遂行するプロジェクトチームのコスト、スケジュール、品質などに関して、それらを計画する、見積もる、資源を割当てる、プロジェクト状況を監視し制御するための手法を追求する。

2.1　プログラミング方法論（1970年代前半）

構造化プログラミング（1972）、情報隠蔽原理（1972）、ＪＳＰ法（1975）により、全体の作業を、独立性を保証する作業単位へ分割する手段、作成中のプログラムが所望の動

作を行うことを確認する手段の提供などに力点を置いたプログラミング方法論が発展した。

2.2　設計方法論（1970年代後半）

複合設計（1978）、構造化設計（1979）などにより、変更の波及を局所化できるようなソフトウェア構造とその作りこみの手段が検討された。

2.3　要求定義（1970年代後半）

「問題を定義する手段なしに、解を構築する手段を追求しても意味がない」という反省のもとに、70年代の後半（1977）に、ウォータフォール・モデルの出発点となる要求定義フェーズの形式化が課題となり、機能要求や性能要求を記述し解析するための要求定義法に関する研究が始まった。要求定義法のみによっては、要求仕様を確定できないことがすぐに明らかになった。

2.4　定量的プロジェクト管理（1970年代後半から1980年代前半）

コスト見積り（ＣＯＣＯMO，1981）、危険要因の検出（MaCaveのＶ、1976，HalsteadのＥ、1977）、テスト時期の打ち切り（信頼性曲線、1982）などの定量的プロジェクト管理に関する試みが開始された。この段階で、ソフトウェア開発パラダイムとプロジェクト管理の関係、ソフトウェア開発パラダイムとソフトウェア開発方法論の関係が検討されはじめたが、ソフトウェア開発方法論とプロジェクト管理の関係が検討されなかったのは問題であると考える。

2.5　システム工学の導入（1980年代中期から後半）

この時期にウォータフォール・モデルの様々な改良が試みられた。文献[２]の内容を引用することにより以下に説明する。

１．ウォーター・フォールモデルの派生型はそのほとんどがシステム工学の原則から生み出されたものである。システム工学は複雑なシステムのコンポーネント設計、仕様化、実装の検証を主な目的としたものであり、元々はスペース・シャトルのような航空宇宙システムやNATOの指揮制御システムといった、大規模システム構築時に発生する問題を解決するために開発されたものである。上記のようなシステムは、あまりにも規模が大きいため、開発そのものを複数の企業に分散させねばならない。まず、ある企業が元請業者となり、システム全体の設計と統合を担当し、そして、元請業者のシステム・エンジニアが集まってシステム要求を分析し、その後、複数のサブシステムへと分割してシステム設計を行なう。

２．システム工学的アプローチは：システム要求を定義する；システム要求をサブシステムに配分する；詳細なコンポーネントを定義する；コンポーネントを検証する、サブシステムを検証する；といったＶ字型のプロセスになる。

３．システム工学アプローチは一見すると、大規模なソフトウェア・システムを開発するアプローチとして適切であるように思える。しかし、システム工学的アプローチが基としている前提は、ソフトウェア開発ではうまく適用できないものである。その前提とは、要求が安定していること、実装に先立ってインタフェースを適切に仕様化できることである。

４．ウォータフォール・アプローチによって、要求管理、アーキテクチャと設計、実装、テスト、運用、保守といった、プロジェクトの基本的なアクティビティが洗い出された。しかし、要求ドキュメントや設計ドキュメントを凍結できないというソフトウェア開発特有の性質のため、近代的なソフトウェア・プロセスにおける詳細なアクティビティや成果物は、システム工学プロジェクトのそれとは大きく異なっている。

2.6　反復型ウォータ・フォールモデル（1980年代前半）

これについても文献[２]の内容を以下に引用する。

W.Royce や B.Boehm といったソフトウェア開発管理の創始者達は、かなり初期から古典的なウォーターフォール・プロセスの問題に気付いていた。1980 年代に彼らは、ミニウォータフォール、あるいは Boehm スパイラルモデル(1988)といった、顧客とのやり取りを改善できる、ウォーター・フォールの改良版を考案している。

１．改良版では、開発サイクルを短い期間に分割し、それぞれの期間でウォーターフォール・プロセスを実行し、要求のレビュー、ドキュメントの更新、コードの開発を行う。ある反復での成果物は次の反復の入力になる。顧客レビューと仕様の更新といったプロセスを、連続しておこなうことによって、プロジェクトのリスク管理を可能にする。

２．古典的なウォータ・フォール管理アプローチにおけるいくつかのコンセプトは、適切に変更すれば、ソフトウェア・マネージャにとって有益なものとなる。中でも、プロジェクト計画と進捗管理の原則は常に重要である。特に、作業分割の構造、コストとスケジュールの変化、出来高といったものを理解することは有効である。アクティビティに基づいた計画や厳格なタスク分割は、プロジェクトの本当の性質を反映したものでないために、各アクティビティにかかった時間を評価してもプロジェクトの実状すら把握することはできない（著者注：この言明を是とするならば、付録 B「PMBOK の内容」における「計画プロセス群」、「プロジェクトタイム管理」における様々の技法はあまり有効でないという結論になる）。

３．問題を早期に洞察すればするほど、それを修正できる可能性がましていくというのが大規模システム構築における基本原則である。修正コストというものは、プロジェクトの進捗に従って劇的に増加していく。統合とテストを最終段階で行うウォータフォール・アプローチは、統合段階に到達しなければ実際の作業状況を認識できないという問題がある。

2.7　プロトタイピング（1980年代前半）

Royceの論文ではライフサイクルでの「内なる繰り返し」が推奨されているが、1980年代初頭（1985）におこったプロトタイピングは、利用者世界を含めてサイクルを回す必要があるとの認識に基づいている。この時代のプロトタイピングはどちらかというとイテレーティブイン（反復）開発であり、プロジェクト状況の学習にともなうプロセスの改善を目指すインクリメンタル開発ではなかったように思える（文献[３]の５．３節を引用して、両者の定義を付録Aで説明する）。後者の場合、固定的な標準化は学習の効果を無視することになり望ましくないように思える。しかし、プロジェクト進行中に頻繁にその進め方を変更することはある種の混乱を招く。プロセス改訂の頻度をタイミングが問題である。

2.8 実行可能仕様（1980年代中期）

リアルタイムデータフロー(Ward 1986)、JSD(M. Jackson,1983), PAISLey(P.Zave,1986), PROTnet(Bruno,1986), UI(Wasserman,1986)など上流工程をフォーマルに書く技術が発展した。これらは、後に上流工程の各種プロダクトを表現する言語の発展につながった。しかし、コーディングを労働集約型として蔑視する動きを助長した問題点もあるように思える。

2.9　フォーマルメソッド（1980年代中期）

上流の文書化の質が下流のプロダクトの質に影響を与えるというW.Royceの見解に対して、上流の文書を数学的記法で厳密に記述すれば、人間が解析困難であるさまざまなソフトウェア性質の検証、証明、自動化が可能になり、検査フェーズのコスト減や最終プロダクトの品質につながるという認識が、1980年代の中頃におこり、モデル検査につながっている

2.10 プロセスプログラミング（1980年代後半）

1987年、L.OsterWeilはソフトウェア開発プロセスを記述するプロセスプログラミング言語を開発することにより、プロセス改善の土台にできると考えたがあまり成功しなかった。

2.11　プロセス改善（1990年代初頭）

CMM(1989). 1990年代にはいっておこり、PMBOX[４]その他の体系化として実を結びつつある、ソフトウェアプロセスの改善に関するフレームワークの確立とベストプラクティスの整備は開発管理手法の発展をうながした。付録Bに、文献[４]における表３－４５に、その他の部分の記述内容を、主に、ツールと技法に注目して埋めこんだ表を示す。ところで私見であるが、CMMはレベル３以上に実質上の意味があると考える。すなわち、コスト、スケジュール、信頼性達成などの状況を、ファンクションポイント（FP）にもとづく様々なメジャー[５]（生産性に関する指標：FPあたりの費用、IT部門の生産性、開発速度、ユーザが利用する機能と開発された機能；　品質に関する指標：機能要求の規模、完備性、変更率、欠陥除去率、欠陥密度、テストケースの網羅率、文書量；　経済性に関する指標：

FP当りの費用、補修費率、ポートフォリオ資産価値；　保守性に関する指標：保守性、信頼性、要員配置、成長率、ポートフォリオ規模、バックファイア値、安定度）や、Caper Jonesが定義した意味でのベースラインやソフトウェアベンチマーク[６]などで（定性的に問題を把握し）、定量的に証拠だてることによりテクニカルプロジェクトマネージメントが可能になる。

2.12 CASEツール（1990年代初頭）

「上流工程で、早期に、コストのかかる誤りを除去する」などの目的のもとに、StP（１９８８）、PCTE（１９９３）などのCASEツールや統合環境が開発された。これについては興味深い意見がある[７]。文献[７]の内容は非常に興味深い、付録Cに文献[７]の要約引用を示すが、ぜひ[７]原本を一読されることを薦める。

真実５． 開発ツールの大げさな宣伝はたちの悪い伝染病みたいなものだ。ツールや技法の大部分は、生産性や信頼性を５－３５パーセント程度、改善するにすぎない。

真実６． 新しいツールや技法を学習すると、最初は生産性や品質が下がる。実際に効果がでるのは、ツールや技法が完全に身についてからだ（習熟曲線)。オブジェクト指向は表面的には３ヶ月、自由に使いこなすには３年かかる。ツールや技法の導入を効果的にするには、（１）効果が現実的である（２）気長に効果を待てる場合に限る。

真実７． ソフトウェア技術者は、ツールの話が大好きである。いくつものツールを買い、評価もしているが、開発で実際に使った人はほとんどいない。コンパイラ、デバッガ、エディタ、リンクローダ（無意識)、構成管理ツールで十分である。

2.13　アーキテクチャ中心開発

W.Royceの「基本設計フェーズ」の導入の意図は、1990年代中頃におこったアーキテクチャパターンにより加速されアーキテクチャを評価し、検証する技術が進んでいる。しかし、必ずしも十分な成果が得られているとはいえない。

2.14　オブジェクト指向

再利用のメリットについての議論はともかく、プログラムの構成単位（オブジェクト）に再利用性や変更容易性に関する機構を内蔵させたこと、要求仕様、設計仕様、プログラム、計算機内部の動作間の対応関係をシンプルにし、反復を容易にしたことは見逃せない。

2.15　UML

上流、中流における各種文書を記述するための言語を標準化しようとする努力は無視できない。

2.16　アジャイル

Royceの論文にある、「テストの計画と制御と監視」、「顧客を含めよ」などの問題提起に形を与えた。また、このアプローチはウォータフォールに対して正反対のアプローチである。[役に立たない要求文書や設計文書のことなど、すべて忘れてしまおう。こういったものが正しくなることはないのだ。ただひたすらにコードのビルドを行おう]。このようなラピッドプロトタイピングのアプローチには利点と欠点がある[２]。

- 利点：チームの生産性が向上する。顧客とのやりとりが建設的なものになる。
- 欠点：プロジェクトを収束させることが難しくなる。スケジュールや予算が見積もりにくくなる。大規模ソフトウェア開発には適さない。成果物がプロトタイプでしかない危険性がある。

３．まとめと課題

いわずもがなであるが、W.Royceの話がすべての原点であるなどというつもりはない。[２]によればウォータフォールモデルの欠点は以下の通りである。

- ウォーターフォール・プロセスが機能しない理由を理解しよう。ウォーターフォール・アプローチには、すべてのプロジェクトに適用可能な、重要な原則が含まれている。
  ①　開発におけるアクティビティを洗い出す。
  ②　アクティビティ毎に工数計画を立案する。
  ③　マイルストーンを用いてアクティビティの進捗を管理する。
- このような原則は、マネージメントがしっかりしているプロジェクトであれば、どのようなプロジェクトにも存在する要素である。このため、クリティカル・パス分析や細密マイルストーン(inchstone)管理といった、成熟した古典的プロジェクト管理テクニックが採用できると考えるのは自然なことである。
- このアプローチの問題として、**まず、ソフトウェア開発プロジェクトは建築プロジェクトほど簡単に計画、評価できない**と言う点をあげることができる。橋の塗装が半分完了したというのは、誰の目にも明らかだが、コードの半分がいつ完了したのかを知ることは難しい。また、橋の各部分を塗装する時間は簡単に見積もることができるが、コーディングやデバッグにかかる時間など、最終的なコードの規模が判らなければ誰にも判らない。
- 精密な進捗見積りと完了時間を要求することで、こういった不確実性に取り組もうとするマネージャもいる。しかし、これはマネージャに対して嘘をつくことを開発者に強要することになる。
- **２つ目の問題は、あるアクティビティが完了しないと次のアクティビティが始まらないという、アクティビィティの直列化によってひきおこされる。**
- アクティブティが完了するということは、成果物が完全なものとなったことを意

味する。橋の塗装の場合は簡単にわかるだろうが、要求文書が完成したかどうかなど、確信を持って答えられるはずもない。

- ソフトウェア開発のアクティビティを直列化しようとした場合、たいていのプロジェクトは遅かれ早かれ失敗することだろう。興味深いことに規律が厳しいプロジェクトほど早い段階で失敗する傾向にある。要求定義の完了や設計仕様文書の完成といった、初期のマイルストーンに到達する前に失敗してしまう。ドキュメントがレビューされる度に新たな問題が持ち上がり、疑念が生じ、不整合が発見され、誰にも答えられない質問がでてくる。それに対して、マネージメントは、品質を力説し、規格に固執することで、規律を守ろうとする。彼等の要求が満足されることはなく、最後には多くの資金を投じた役に立たない成果物だけがのこることになる。
- 早い段階での失敗よりも巷でよく見かけるのが、遅い段階での失敗である。この場合、プロジェクトは終了間際までうまく進んでいるように見受けられる。ドキュメントは完成し、レビューも済み、前半のマイルストーンも完了したかのように見える。しかし、現実的には、日程だけが消化されており、内容は基準に満たないものであることにみな感づいているはずである。

物事の起源の一つを振り返り、それを基準点にして、これまでの研究、技術開発の世界を体系的に把握し、それをふまえて今やるべきことを考えることは大事な作業である。いくつかの課題を整理してみる。

1. 方法論については、変更や検査・検証、再利用が容易な構造をどのようにしてソフトウェア構造につくりこんでいくかと理論と技術の開発に主眼がおかれてきたと思う。
2. プロジェクト管理については、コスト見積り、スケジュール管理、品質保証などの各種メトリックスとそれを活用するためのベストプラクティスが整備されてきた。
3. 双方の研究が最近では乖離しているように思える。
4. 温故知新。W.Royceの初期の考察から学べる最大のポイントは，ソフトウェア開発方法論とプロジェクト管理技術を融合してとらえるという姿勢であろう。
5. 融合の手段についてはいくつかの視点がある。
   (ア)いくつかの問題を早期に洞察すればするほど、それを修正できる可能性がましていくというのが大規模システム構築における基本原則である(マレー・カンター)。
   (イ)プロジェクト開始時には、向かっている目標がわからない。インクリメンタルな開発は、一度開発サイクル全体を経験させ、新しく発見した知識をすぐ使えるようにする。すべてを知っていると思っていても、プロジェクトには突発事項が待ち構えている。インクリメントによって、そのような突発事項に早く気付くことができる。突発事項は何かということに早く気付くと、作業方法を変更する機会

が得られる。各インクリメント境界は、行っていることすべてを改善する機会である（アリスター・コーバーン）。

(ウ)アセスメントとは、ソフトウェアプロジェクトが設置され維持される方法を検討するための、公式的かつ構造化された手段である。定性的データに基づくことが多い。ベンチマークやベースラインと相補的に利用する。定性的データによって原因を理解し、定量的データでそれを正当化する（ケーパージョーンズ）

これらの指摘が示唆していることは以下の点であると思う。

（1） ソフトウェア開発プロジェクトの特徴は不安定である

（2） しかもその不安定さは、プロジェクト依存である

（3） プロジェクト状況を把握にするにはプロジェクトに関する「学習」が必須である

ソフトウェア開発方法論とプロジェクト管理をどのように融合できるのであろうか。その一つの視点は「開発管理とはプロジェクト状況の学習である」というとらえ方ではないだろうか。ソフトウェア開発方法論は、変更容易性、検査容易性、再利用容易性などの、ソフトウェアが持つべき望ましい性質をつくりこむ技術である。ところで、ドメインの特性、規模、新規開発か経験済みか、それに割り当てることのできる資源はどのくらいか、担当者の技術レベルはなど、ソフトウェア開発活動を特徴づけるパラメータがある。ソフトウェア開発が進行するにつれて、これらの内、見積もりや計画段階で読みきれなかったことがらが顕在化してくる（状況）、プロジェクト管理の一つの重要な活動として、これらの状況を知る（学習）ことがあげられる（プロジェクト監視）。 問題を定性的に把握しそれを定量的に証拠だてる。 開発手段とそれを実行するための諸元は、プロジェクト毎にことなるので 、開発手段とプロジェクト管理の技術をうまく結合するには、特定の開発手段（例えば、インクリメンタル開発）やその実行環境に依存する議論が必要なのではなかろうか。その接点として、プロジェクトの状況を学習する手段の検討が挙げられる。学習に必要なデータとその利用法（フィードバック法）はひとつの接着剤候補であるように思える。

フィードバックをかけるタイミングはいつかという問題がある。次期プロジェクトで学習の成果をフィードバックするというのがごく自然な発想であるが、私見では、状況はプロジェクト毎に異なり、効果がないことになる。インクリメンタル開発では、「一回のインクリメント毎にプロセスを改善せよ」と指針がアリスター・コーバーンによって与えられている。確かに同じプロジェクト内でのフィードバックは学習の効果を期待できる。しかし、物事の進め方（標準）を頻繁に変更することは混乱をまねきやすい。さじ加減を検討する必要がある。

４．おわりに

ソフトウェア工学の成果が少しでも世の中の状況を改善しうるためには、問題点を産学

協同で真正面から取り組む必要がある。できることのみをやっていたのでは、我々の問題はいつまでたっても解決しない。ソフトウェア開発管理に関する要素技術はかなり成熟してきているように思える。それらを実践し、さらなる改良を積み重ねる努力が必要である。

文献

[1] W.W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques", Proc. of IEEE WESCON, pp.1-9, 1970 (Proc. of 9th ICSE, pp328-338, 1987 に再掲)

[2] マレー・カンター著、村上雅章訳、「ソフトウェア開発管理の要」、ピアソン・エデュケーション、2002.

[3] アリスター・コーバーン著、長瀬嘉秀、今野睦監訳、「アジャイルプロジェクト管理」、ピアソン・エデゥケーション、2002.

[4]「プロジェクトマネージメント知識体系ガイド第３版（PMBOK ガイド)」、Project Management Institute, Inc. 2004.

[5] デービッド・ガーマス、デービッド・ヘロン著、小泉、中村、向井訳、児玉監修、「ファンクションポイントの計測と分析」、ピアソン・エデュケーション、2002.

[6] Caper Jones, "Software Assessments, Benchmarks, and Best Practices", Addison-Wesley Information Technology Series, 2000.

[7] ロバート・L・グラス著、山浦恒央訳、「ソフトウェア開発５５の真実と１０のウソ」、日経 BP 社、2004.

付録 A　インクリメントとイテレーション（文献[3]より引用）

インクリメントは開発プロセスを修正または改善できるようにする。インクリメンタルな開発のメリットは以下の通りである。

1. 教育：プロジェクト開始時には、向かっている目標がわからない。インクリメンタルな開発は、一度開発サイクル全体を経験させ、新しく発見した知識をすぐ使えるようにする。
2. 無知の無は何かを発見する：すべてを知っていると思っていても、プロジェクトには突発事項が待ち構えている。インクリメントによって、そのような突発事項に早く気付くことができる。
3. 作業方法を修正する：突発事項は何かということに早く気付くと、作業方法を変更する機会が得られる。各インクリメント境界は、行っていることすべてを改善する機会である。

イテレーションは、システムの品質を修正または改善できるようにする。イテレーティブな開発は、システムの各部分を改訂し、改善する時間をどのように取るかという、やり直しのスケジュールを立てる戦略である。イテレーティブな開発はインクリメンタルな開

発を前提にしてはいないが、同時に行うととても効果的である。「イテレーティブな開発」という言葉自体は、納品物を改訂する量とタイミングをしめしていない。要求を改訂するか、設計だけを改訂するか、改訂を別のインクリメントとして扱うか、インクリメント内に組み込むかは、選択の余地が大きい。以下はイテレーティブな開発のメリットである。

1. リスク削減
2. 品質向上
3. コミュニケーション改善
4. ユーザが使用できるものを納品していることの保証

Ｖ－Wステージングは、検証Ｖ（要求・設計・テスト・出荷）－学習－検証Ｖ（要求・設計・テスト）（全体としてW）からなり、インクリメンタルな開発とイテレーティブな、スパイラル、プロトタイプ戦略を含む一般的な戦略である。Ｖ－Wステージングは、プロジェクトマネージャに以下の作業を行う方法を提供する。

- スパイラル開発の螺旋をまっすぐにし、アクティビティがプロジェクトスケジュールに対して一直線になるようにする。
- 恒常的進捗という考えを犠牲にせずに、プロトタイプとイテレーションを用意する。
- 「フェーズ納品物」ではなく、納品された機能に基づいて、効果的な進捗表示を導き出す。
- フェーズ納品物ではなく、納品されたインクリメントで下請を管理する。
- 禅を学習せずに「ゲシュタルトラウンドトリップ（システム全体を経験すると、プロジェクトが進む方法に関して「そうか」という感情が得られる)」の価値を得る。

プロジェクトインクリメント

- 一回目のインクリメントの目的
  - ソフトウェアのアーキテクチャを確立する
  - プロジェクトスポンサーとユーザにフィードバックを届ける
  - プログラミングと設計の良いパターンを開発する
  - おばけを発見し、無知の知を学習する
- 二回目のインクリメント
  - 一回目のインクリメントで発見された主な間違いを修正する
  - ソフトウェアアーキテクチャを確立、または調整する
  - 役立つプロセスとチームを作成する
  - よい開発習慣を決める
  - ユーザに多くの機能を納品する
- Ｎ回目のインクリメントまでに、成功の習慣とパターンを得ている。注意すべきリスクが二つある。眠ってしまうこと、および拡張しすぎてしまうこと。

付録 B　PMBOX　第 3 版の内容

プロジェクト管理プロセス群 / 知識エリアのプロセス	**立上げプロセス群:**新しいプロジェクトやプロジェクトフェーズを開始するための公式な認可を支援する	**計画プロセス群**：目標を定め、それを洗練し、プロジェクトが取り組むべき目標とスコープを達成するために必要な一連の活動を計画する	**実行プロセス群**:プロジェクト管理計画書を実行するために、人やその他の資源を統合する	**監視コントロールプロセス群**：潜在的な問題をタイムリーに識別できるように、プロジェクトの実行を監視。必要に応じて是正処置を講じられるように、プロジェクトの実行を制御	**終結プロセス群**：プロダクト、サービス、所産を正式に受け入れ、プロジェクト（またはフェーズ）を公式に終了する
プロジェクト管理統合：プロジェクト管理の様々な要素を統合するプロセス	**プロジェクト憲章作成**(プロジェクト選定手法。プロジェクト管理（PM）方法論。PM 情報システム（IS)、専門家の判断) **プロジェクトスコープ記述書（暫定）**高レベルの記述	**プロジェクト管理計画書作成**（PM 方法論。PMIS（構成管理、変更管理)。専門家の判断）プロジェクト管理計画書は、プロジェクトの実行、監視コントロールおよび終結の方法を規定する。その内容は、プロジェクトの適用分野と複雑さによって異なる。それは、統合変更管理プロセスを通じて更新され、改訂が行われる。PM 計画書作成プロセスは、すべての補助の計画書の定義、統合、調整等を行い、プロジェクト管理計画書とするために必要な活動からなる。	**プロジェクト実行の指揮・管理**（PM 方法論。PMIS）プロジェクトスコープ記述書に規定された作業を達成するために、計画したプロジェクトアクティビティの実行を指揮し、プロジェクト内に存在するさまざまな技術上と組織上のインタフェースの管理をおこなう	**プロジェクト作業の監視コントロール**（PM 方法論。PMIS。プロジェクトのパーフォマンスの測定と予測を行うアーンド・バリュー法。専門家の判断)。**統合変更管理**　承認・却下、承認済み変更のベースラインへの組込み	**プロジェクト終結**　事務終了手順。契約終了手順
プロジェクトスコープ管理：プロジェクトの成功に必要な作業を過不足なく含めることを確実にするために必要なプロセス		**スコープ計画**（専門家の判断、テンプレート・書式・標準）**スコープ定義**（プロダクト分析（システム分析/工学、価値工学価値分析、機能分析)。代替案識別(ブレーンストーミング、水平思考)。専門家の判断。ステークホルダー分析）**WBS 作成**（テンプレート。要素分解）		**スコープ検証**（検査）**スコープコントロール**（変更管理システム。差異分析。再計画。構成管理システム）	
プロジェクトタイム管理：プロジェクトを所定の時期に完了させるために必要なプロセス		**アクティビティ定義**（要素分解。テンプレート。直近の作業は下位レベルまで詳細に、遠い先は上位レベルだけ WBS を定めるローリング・ウェーブ計画法;専門家の判断;計画構成要素)。**アクティビティ順序設定**（プレシデンスダイアグラム法、アロー・ダイアグラム法、スケジュールネットワークテンプレート、依存関係の決定（強制、任意、外部依存関係）　**アクティビティ資源見積り**（専門家の判断。代替案分析。公開見積りのデータ。プロジェクト管理ソフトウェア。ボトムアップ見積り）　**アクティビティ所要期間見積り**（専門家の判断、類推見積り、係数見積り、三点見積もり、予備設定分析)、**スケジュール作成**（スケジュールネットワーク分析、クリティカルパス法、スケジュール短縮、what-if シナリオ分析、資源平準化、クリティカルチェーン法、プロジェクト管理ソフトウェア、カレンダーの適用、リードとラグの調整、スケジュールモデル）		**スケジュールコントロール**（進捗報告、プロジェクトスケジュールを変更する際の手続きを規定したスケジュール変更管理システム。スケジュールの差異に対して是正処置が必要か否かを決定するパーフォーマンス測定（スケジュール差異、スケジュール効率指数)。プロジェクト管理ソフトウェア。目標スケジュール日付を予測開始日や実開始日及び実終了日と比較する差異分析。バー・チャートによるスケジュール対比）	
プロジェクトコスト管理：プロジェクトを承認された予算内で完了させるために必要なプロセス		**コスト見積り**（類推見積り、資源単価、ボトムアップ見積り、係数見積り、プロジェクト管理ソフトウェア、ベンダー入札の分析、予備設定分析、品質コスト)、**コストの予算化**（コスト集約、予備設定分析、係数見積り、限度金による資金調達）		**コストコントロール**（コスト変更管理システム。アーンドバリュー法によるパフォーマンス測定分析、予測。プロジェクトのパフォーマン	

				スレビュー。プロジェクト管理ソフトウェア。差異管理)	
プロジェクト品質管理：プロジェクトが所定の目標を満たすことを確実にするためのプロセス		**品質計画**（費用便宜分析によるトレードオフスタディ。ベンチマークによる比較改善。実験計画法による開発中のプロダクトやプロセスの特定の変数に影響を与える要因の識別。品質コスト。品質計画ツール)	**品質保証**（品質計画のツールと技。品質監査。プロセス分析。品質管理のツールと技法)	**品質管理**（特性要因図、管理図、フローチャート化、ヒストグラム、パレート図、ラン・チャート、散布図、統計的サンプリング、検査、欠陥修正レビュー)	
プロジェクト人的資源管理プロジェクトチームを組織化し管理するためのプロセス		**人的資源計画**（チームメンバーの役割と責任を階層型、マトリックス型、テキスト型のいずれかの書式で記述する組織図と職位記述図。積極的相互交信、昼食会議、非公式の会議、取引上の会議などの人的資源のネットワーキング。要員、チーム、単位組織などの行動様式に関する情報を提供する組織論)	**プロジェクトチーム編成**（先行任命、交渉、調達、バーチャル・チーム）**プロジェクトチーム育成**（一般的な管理スキル、トレーニング、チーム形成活動、行動規範、コロケーション、表彰と報酬)	**プロジェクトチーム管理**（観察と会話、プロジェクトのパフォーマンスの評価、コンフリクト管理、課題ログ)	
プロジェクトコミュニケーション管理プロジェクト情報の生成・収集・配布・保管・廃棄をタイムリーかつ適切に行うために必要なプロセス		**コミュニケーション計画**（コミュニケーションに対する要求事項の分析によってプロジェクトステークホルダーの情報に関するニーズを収集する。情報の緊急度、技術の可用性、予想されるプロジェクト要員配置、プロジェクト期間、プロジェクト環境などのコミュニケーション技術)	**情報配布**（コミュニケーションスキル、情報収集システム、情報配布手法、教訓プロセス)	**実績報告**（情報提示ツール、パフォーマンス情報の収集・編集、状況レビュー会議、タイム報告システム、コスト報告システム)	
プロジェクトリスク管理プロジェクトのリスク管理を行うプロセス		**リスク管理計画**（計画会議と分析)、**リスク識別**（文書レビュー。情報収集技法（ブレーンストーミング、デルファイ法、インタビュー、根本原因の識別、強み、弱み、好機。脅威の視点から分析する SWOT 分析)。チェックリスト分析。前提条件の妥当性分析。図解の技法（特性要因図、システムやプロセスのフローチャート、インフルエンス・ダイアグラム)) **定性的リスク分析**（リスク発生確率・影響度査定、発生確率・影響度マトリクス、リスクデータ品質査定、リスク区分、リスク緊急度査定)、**定量的リスク分析**（データ収集・表現技法。定量的リスク分析とモデル化の技法（感度分析、期待金額価値分析、デシジョン・ツリー分析)) **リスク対応計画**（マイナスのリスク（脅威）に対する戦略（回避、転嫁、軽減)。プラスのリスク（好機）に対する戦略（活用、共有、強化)。脅威・好機両面戦略。発生時対応戦略)		**リスクの監視コントロール**（定期的に実施するリスク再査定、識別したリスクとその根本原因に対するリスク対応策の有効性やリスク管理プロセスの有効性を調査し文書かするリスク監査。アーンドバリュー分析による差異・傾向分析。技術的実績の測定。プロジェクトの任意の時点で残存している予備が十分であるかどうかを、コンティンジェンシー予備の残存量を残存リスク量と比較する予備設定分析、状況確認会議)	
プロジェクト調達管理：プロダクト、サービス、所産の購入または取得のプロセス		**購入・取得計画**（内外製分析。専門家の判断。契約タイプ（定額契約または一括請負契約、実費償還契約（コストプラスフィーまたはコストプラスパーセンテージ,コストプラス固定フィー、コストプラスインセンティブフィー)、タイム・アンド・マテリアル契約))、**契約計画**（標準書式、専門家の判断)	**納入者回答依頼**（入札説明会、入札公告、適格納入者リストの作成)、**納入者選定**（重み付け法、独自見積り、スクリーニングシステム、契約交渉、納入者点数評価システム、専門家の判断,プロポーザル評価法)	**契約管理**（契約変更管理システム、購入者主催のパフォーマンスレビュー、検査および監査、実績報告、支払いシステム、クレーム管理、記録管理システム、情報技術)	**契約終結**（調達監査、記録管理システム)

マトリックスの格子点には、関連するプロセスと、そのプロセスを実行する際に利用できるツールと技法を記す。

付録 C　ロバート・L・グラス著、山浦恒央訳、「ソフトウェア開発５５の真実と１０のウソ」より要約引用

プロジェクト管理

人員

真実１. ソフトウェアの開発で最も重要なことは、プログラマが使うツールや技法でなく、プログラマ自身の質である。SEI-PCMM を調べよ。

真実２. プログラマ個人を分析した研究によると、最も優秀なプログラマは最悪に比べ、２８倍優れている。給与が能力を反映していないとすると、優秀なプログラマは、最高の掘り出し物と言える。

真実３. 遅れているプロジェクトに人を追加すると、教育に時間をとられるため、もっと遅れる。

真実４. 作業する環境は、プログラムの生産性や品質にきわめて大きく影響する。狭い場所に詰め込むな。

ツールと技法

真実５. 開発ツールの大げさな宣伝はたちの悪い伝染病みたいなものだ。ツールや技法の大部分は、生産性や信頼性を５－３５パーセント程度、改善するにすぎない。

真実６. 新しいツールや技法を学習すると、最初は生産性や品質が下がる。実際に効果がでるのは、ツールや技法が完全に身についてからだ（習熟曲線）。オブジェクト指向は表面的には３ヶ月、自由に使いこなすには３年かかる。ツールや技法の導入を効果的なにするには、（１）効果が現実的である（２）気長に効果を待てる場合に限る。

真実７. ソフトウェア技術者は、ツールの話が大好きである。いくつものツールを買い、評価もしているが、開発で実際に使った人はほとんどいない。コンパイラ、デバッガ、エディタ、リンクローダ（無意識）、構成管理ツールで十分である。

見積り

真実８. プロジェクトが大失敗する原因は二つある。一つは「見積りミス」だ。もう一つは「仕様未凍結（真実２３）」だ。

真実９. ソフトウェア開発の見積りは、プロジェクト開始時に実施する場合が非常に多い。これだと要求定義が固まる前に見積ることになり、どんな問題がどこにあるかを理解する以前に予測するので意味がない。従って見積り時期として適切でない。

真実 10. 見積りは、上層部かマーケティング部門が実施する場合がほとんどだ。結局、適切な人が見積もっていない（政治的見積りと本質的見積り）。

真実 11. プロジェクトが進むに従って、見積りを調整することはまずない。従って、不適切な時期に不適切な人が実施した見積りが修正されることはまずない。

真実 12. 見積り精度がいい加減だと、実際のプロジェクトが見積もり通りに進まなくてもまったく気にならないはずだが、現実にはみな気にする。

真実 13. 管理者とプログラマの間には、大きな断絶がある。ある研究によると、重大な見積りミスを起こし、管理者は大失敗プロジェクトと考えているのに（コスト 419%増、スケジュール 193%増、規模 130%増、ファームウェア 800%増)、開発担当の技術者はこれまで従事した中で、最も成功したプロジェクト（仕様を充たし、リリース後のバグが０）と考えているものがあったらしい。

真実 14. 実現可能性の分析の結果はいつも YES である（ICSE87 ワインバーグの講演）

再利用

真実 15. 小規模な再利用（例えば、ライブラリやサブルーチン）は、50 年前に始まり、すでに解決済みの問題である。

真実 16. 大規模な再利用（コンポーネント単位）は、誰もがその重要性を認識し、なくてはならないと感じているが、今なお未解決である。多種類にわたる特定目的のシステムで使えるように、コンポーネントの機能を一般化するのはほぼ不可能。

真実 17. 大規模な再利用は、類似システム間でうまくいく可能性が高い。応用分野の類似性に依存するため、大規模流用の適用範囲は狭くなる。

真実 18. 再利用には二つの「３の法則」がある。（１）再利用可能なコンポーネントを作るのは、単一のプログラムで使うモジュールを開発する場合に比べて 3 倍難しい。（２）再利用可能なコンポーネントは、ライブラリに取り込む前に、３つの異なるプログラムでテストする必要がある。

真実 19. プログラムを再利用する場合、流用母体の変更は、バグの原因になる。２０－２５％も変更する必要があるなら、最初から作ったほうが、効率が上がるし、品質も良い。ベンダーが作ったパッケージのソフトウェアシステムを改造するのはまず、うまくいかない。

真実 20. デザインパターン方式の再利用（設計方式の再利用）は、ソースコードの再利用における問題を解決する一手段である。

複雑性

真実 21. 対象となる問題の複雑度が２５％増加するたびに、ソフトウェアによる解法の複雑性は１００％上昇する。改善しなければならない数字ではない。こうなるのが普通である。

真実 22. ソフトウェア開発の開発は、８０％が知的な作業である。かなりの部分が創造的な仕事であり、事務作業はほとんどない。

ライフサイクル

要求仕様

真実 23. プロジェクトが途中打ち切りになる二つの原因のうち、一つは仕様を確定できないことだ。

真実 24. 仕様不良の修正コストは製品出荷後は最も高い（100 倍　by Boehm and Basilli）が、開発の初期であれば最も安い。

真実 25. 仕様の抜けは、仕様関係の不良の中で修正が最も難しい。最もしつこいバグ、すなわち、テストをかいくぐり製品に潜り込むのはロジック抜けの不良だ。仕様の漏れがロジック抜けにつながる。

設計

真実 26. 仕様定義フェーズから設計フェーズに移るとき、膨大な数の「派生仕様(derived requirements：仕様を具体的な設計方式にブレイクダウンする場合、設計方式に対する要求仕様)」が生じる。これは問題解決プロセスが複雑なために発生するもので、この設計仕様の量は、元の仕様の５０倍になることもある。

真実 27. ソフトウェア開発において、ベストの解法が一つしかないことはまずありえない。

真実 28. ソフトウェアの設計は、複雑で反復が必要なプロセスである。従って、最初に考えついた設計方式が間違っている可能性は高く、最適解ではない。

コーディング

真実 29. 問題を「基本モジュール」レベルにまで詳細化できた時点で、設計フェーズからコーディングフェーズへ移る。コーディングをする人と設計者が同一人物でない場合、「基本モジュール」の認識にズレが生じトラブルのもととなる。設計とコーディングを分けるのは良くない。

真実 30. COBOL は非常に悪い言語だ。しかし、他のビジネスデータ処理用言語はもっとひどい。

不良除去

真実 31. ソフトウェア開発のライフサイクルで、不良除去に最も時間がかかる。

テスト

真実 32. 十分テストをしたとプログラマが自信を持つソフトウェアでも、全パスの５５－６０％程度しか網羅していない。パス・カバレッジ・アナライザのような自動化ツールを使うと、網羅率が８５－９０％に上がる。しかし、１００％のパスを網羅するのは不可能である。

真実 33. １００％のテスト網羅が可能でも、完全テストとはいえない。バグの約３５％は、パスの抜けが原因であり、４０％はパスの特定の組み合わせを実行したときに起きる。このバグは、パスを１００％カバーしても検出できない。

真実 34. ツールを利用しないと不良除去はうまくいかない。デバッガはみんな使うが、カバレッジアナライザはほとんど使わない。組込みソフトウェアのテストは、環境シミュレータがないと不可能である。

真実 35. テストの自動化は非常に難しい。テストプロセスの中には、自動化できるし、自動化しなければならないものもあるが、大部分は自動化が不可能な作業である。

真実 36. プログラマがソースコードの中に組み込むデバッグ用の命令語（条件コンパイルできることが望ましい）は、テストツールを補ってあまりある。

レビューとインスペクション

真実 37. インスペクションを厳しく実施すると、プログラム実行前に９０％ものバグをたたきだせる。

真実 38. 厳密なインスペクションは大きな効果を上げるが､テストのかわりにはならない。

真実 39. 出荷後レビュー（遡及的レビュー[retrospectives]と呼ぶ場合もある）は、顧客満足度の測定およびプロセス改善の両方の観点から重要だが、実施している企業はほとんどない。

真実 40. ピアレビューには、技術的問題と社会学的問題がある。レビュー相手のことを考慮しないと悲惨な結果になる。集中力こそ、レビューに不可欠な厳密性を生む。

保守

真実 41. 保守には、ソフトウェアコストの４０－８０％（平均 60%）がかかる。従ってソフトウェアライフサイクルの最重要フェーズである。

真実 42. 保守の６０％は機能拡張であり、バグ修正は１７％にすぎない。このため、ソフトウェアの保守は不良修正ではなく、古いソフトウェアに新しい機能を追加する作業である。

真実 43. 出荷後レビュー（遡及的レビュー[retrospectives]と呼ぶ場合もある）は、顧客満足度の測定およびプロセス改善の両方の観点から重要だが、実施している企業はほとんどない。

真実 44. ソフトウェアの開発作業と保守作業は大部分は共通している。例外は、保守の場合、「既存プログラムの理解」が新たに加わることであり、保守作業の約３０％が必要となる。したがって保守は開発より難しい。

真実 45. 優れたソフトウェアエンジニアリングに沿ってプログラムを開発すると、構造がよくなるため、保守は減らずかえって増える。

品質

品質

真実 46. 品質とは属性（移植性、信頼性、効率、利用容易性、検証性、理解容易性）の集合である。

真実 47. ソフトウェアの品質とは、ユーザを満足させることではない。仕様を満足させることでもなければ、コストとスケジュールを満足させることでも、信頼性でもない。ユーザの満足度＝仕様の実現度+スケジュール通りの出荷+適正なコスト+高品質の製品。最初の３つは品質に関係ない。

信頼性

真実 48. 誰もが共通に作りこむ種類のバグが存在する。

真実 49. バグは固まって存在する。

真実 50. 不良除去に唯一無二のベストな方法はない。

真実 51. プログラマ中の残存バグは簡単には摘出できない。従って不良除去では、重大なバグを最小に抑えたり、なくすことを目標とすべきである。

効率

真実 52. プログラムの処理効率には、良いコーディングよりも、良い設計が大きく影響する。

真実 53. 高級言語は、適切な最適化コンパイラがあれば、アセンブリー言語で記述した場合の９０％の効率で処理できる。最新の複雑なハードウェアを使えば、アセンブリープログラムより速くなる。

真実 54. プログラムの大きさと処理時間には、トレードオフがある。片方をよくすると、他方が悪くなる。

研究

真実 55. 大部分の研究者は、技法を「分析」するのではなく、「擁護」する。その結果、(1) 研究対象の技法は、実際には、自分が信じるほどの効果はなく、(2) 技法の本当の価値を検証する評価研究が足りない。

プロジェクト管理

管理

ウソ１． 計測できないものは管理できない。

ウソ２． ソフトウェア製品の品質は管理できる。

人員

ウソ３． プログラミングからエゴを取ることは可能だし、そうでなければならない。

ツールと技法

ウソ４． ツールや技法はどんな状況でも適用できる。

ウソ５． ソフトウェアには、もっと開発方法論が必要である。

見積り

ウソ６． コストやスケジュールを予測する場合、まずソースコードの行数を見積もる。

ライフサイクル

テスト

ウソ７． ランダムテストにより、テストを最適化できる。

レビュー

ウソ８． 多くの目にさらせばバグはとれる。

保守

ウソ９． 将来の保守に要するコストや、いつ現在のソフトウェアを入れ替えるのかの決定は、過去のコストのデータを見ればわかる。

教育

ウソ 10. プログラムをどう書くかを見せれば、プログラムの方法を教えられる。(実際は書く前に読む)