博　士　論　文

設計レビューを重視したソフトウェア開発方式に関する研究

２００２年１月

大筆　豊

目次

第１章　まえがき ・・・・・・・・・・・・・・・・・・・・・・・・・ 1
　１．１　本論文の背景・・・・・・・・・・・・・・・・・・・・・・ 1
　１．２　ソフトウェアの特性・・・・・・・・・・・・・・・・・・・ 3
　１．３　ソフトウェア開発の本質的な難しさ・・・・・・・・・・・・ 9
　１．４　ソフトウェア工学の成果とその目指すところ・・・・・・・・ 18
　１．５　本論文の構成・・・・・・・・・・・・・・・・・・・・・・ 20

第２章 ソフトウェア工学の集大成としての統合ＣＡＳＥ環境：ＩＭＡＰ　23
　２．１　ソフトウェア開発工業化の概念・・・・・・・・・・・・・・ 23
　２．２　ＩＭＡＰの特徴と機能・・・・・・・・・・・・・・・・・・ 28
　２．３　ＩＭＡＰの波及効果・・・・・・・・・・・・・・・・・・・ 34

第３章　品質問題解決のための設計レビュー制度 ・・・・・・・・・・・ 36
　３．１　全社運動の背景・・・・・・・・・・・・・・・・・・・・・ 36
　　３．１．１　ロスジョブの撲滅・・・・・・・・・・・・・・・・・ 36
　　３．１．２　ロスジョブ分析・・・・・・・・・・・・・・・・・・ 37
　３．２　品質システムの概要・・・・・・・・・・・・・・・・・・・ 37
　　３．２．１　設計レビュー制度の概要・・・・・・・・・・・・・・ 39
　　３．２．２　設計レビューの実施責任者・・・・・・・・・・・・・ 39
　　３．２．３　設計レビューの実施内容・・・・・・・・・・・・・・ 42
　３．３　段階的な設計レビューの徹底実施・・・・・・・・・・・・・ 42
　　３．３．１　ＤＲ－０の実施・・・・・・・・・・・・・・・・・・ 44
　　３．３．２　ＤＲ－Ｘの実施・・・・・・・・・・・・・・・・・・ 45
　　３．３．３　ＤＲ－Ｆの実施・・・・・・・・・・・・・・・・・・ 45
　３．４　設計レビュー徹底実施の効果・・・・・・・・・・・・・・・ 46
　３．５　設計レビュー制度の考察・・・・・・・・・・・・・・・・・ 50

第４章　現状の開発手法と管理手法の効果と問題点・・・・・・・・・・・ 52
　４．１　ソフトウェア開発をめぐる３つの時間・・・・・・・・・・・ 53
　４．２　現状の手法の問題点・・・・・・・・・・・・・・・・・・・ 58
　４．３　統合ＣＡＳＥ環境の効果とその問題点・・・・・・・・・・・ 63
　　４．３．１　統合ＣＡＳＥ環境ＩＭＡＰの問題点・・・・・・・・ 63
　　４．３．２　統合ＣＡＳＥ環境ＩＭＡＰ開発の効果・・・・・・・ 67
　４．４　組織的な開発管理手法の効果とその問題点・・・・・・・・・ 67
　　４．４．１　設計レビュー制度の効果・・・・・・・・・・・・・・ 68
　　４．４．２　問題点とその解決策・・・・・・・・・・・・・・・・ 69

第５章　設計レビューを重視したソフトウェア開発方式の考察・・・・・　72
５．１　設計者を主役とする既存の開発手法・・・・・・・・・・・・・　74
５．２　既存の手法についての考察　・・・・・・・・　・・・・・・・　78
５．３　設計レビューを重視した開発方式の提案・・・・・・・・・・　80
５．３．１　提案するモデルの前提条件・・・・・・・・・・・　82
５．３．２　モデルの全体概要・・・・・・・・・・・・・・・・　83
５．３．３　設計レビュー重視の開発・・・・・・・・・・・・・　97
５．４　提案するモデルの特徴　・・・・・・・・・・・・・・・・・・・101
５．５　提案するモデルの実用上の鍵・・・・・・・・・・・・・・・・108

第６章　むすび　・・・・・・・・・・・・・・・・・・・・・・・・・・・　114

参考文献・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・　117
謝辞・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・　120
研究業績一覧表・・・・・・・・・・・・・・・・・・・・・・・・・・・・　121

第１章　まえがき

１．１　本論文の背景

コンピュータが誕生してからほぼ半世紀が経つ．この間，主として半導体技術の急速な進歩により，その高性能化・小型化・低価格化が進み，現代生活の隅々まで応用されるようになった．科学技術計算用，事務処理用，産業機器制御用，通信用など，全分野を含む産業界はもとより，家庭電器製品を通して一般家庭，また各種社会システムにまで応用されるようになった．さらにインターネットの急速な普及や高度情報社会の進展に伴い，コンピュータは社会システムの中枢に入り込み，社会生活全般に多くの影響を与えるようになっている．

西暦年号を下２桁で表現していたため生じた西暦２０００年問題は，幸いにも大きな被害がなくて済んだが，コンピュータがいかに社会に浸透し，それが正しく動作しなかった場合の影響の大きさを改めて認識させるものであった．また，コンピュータと通信技術の融合によるインターネットの急速な普及により，情報化社会の恩恵が一般の市民レベルまで浸透しつつある．しかし，コンピュータウィルスやハッカーの攻撃といった犯罪行為の影響が，市民生活の安全まで脅かすことになってきた．これら悪意の犯罪行為については本研究の範囲外であるが，コンピュータがいかに社会生活に大きな影響を持つようになったかの，一つの証左である．

コンピュータ応用の進展に伴い，コンピュータを動かすためのソフトウェ

アの需要が増大し，コンピュータ応用が高度になるにつれ，ソフトウェアが複雑化しその規模が大きくなってきた．また，コンピュータの社会的な影響が大きくなっているので，品質に対する要求も厳しいものになっている．さらに，ソフトウェアが高度化し大規模化するにつれ開発者は専門化し，利用者との契約で，あるいはパッケージのように商品として売ることを目的として開発するようになった．因みに，主としてソフトウェア開発を専門とする我が国の情報サービス産業は，1998 年度売上は９兆８０００億円でGDP(Gross Domestic Product)の約２％ ，就業者数は５３万人となっており，産業としても重要な位置付けを占めるようになっている[1].

大規模ソフトウェアを一人の技術者が一定の期間に開発できないので，ソフトウェア開発は，顧客，管理者，技術者など多数の開発関与者により共同開発をせざるを得ない．これらの状況に対処すべく，ソフトウェア開発を体系的に行うことを目指した「ソフトウェア工学」という言葉が１９６８年のＮＡＴＯ会議以来使われるようになった．

ソフトウェア工学の研究分野は，開発手法や管理手法を含む多岐の項目にわたるが，その究極の目的は，ＱＣＤ（品質：Quality，コスト：Cost，納期：Delivery Time）の向上である．すなわち，利用者に満足してもらえるように，高品質のソフトウェアを，低価格で，短期間に開発することがソフトウェア工学の目的である．

ソフトウェア工学の課題としては，開発プロセス，開発手法，品質管理を含む開発管理手法など多岐にわたるが，１９８０年代から１９９０年代にかけ，その集大成ともいえるＣＡＳＥ（Computer Aided Software

Engineering）環境が開発され，広く活用されるようになった．上流ＣＡＳＥと下流ＣＡＳＥ，さらには開発管理ツールを統合させる，統合ＣＡＳＥ環境がソフトウェア開発の究極の救世主であるかのように考えられ，国家レベルや企業レベルで数多く開発されたが，現在ではあまり活用されなくなっている．

本研究では，何故統合ＣＡＳＥ環境がソフトウェア開発の究極の解となりえなかったか，事例を踏まえその本質的な原因を考察する．

また実際の企業において全社的に組織的に取り組んだ開発管理事例を取り上げ，ソフトウェア開発管理のあり方を考察する．

それらの考察を基にソフトウェア開発についての問題点を解決するための開発手法と管理手法を含む新しい開発モデルを提案し，その効果について考察する．

１．２　ソフトウェアの特性

ここでは，ソフトウェアの持つ本質的な性質について，他の工業製品や人間の創造物との対比で考察する．本節では，ソフトウェアの特性のうち，次の項目について議論する．

（１）　コンピュータという人工物に依存している．

（２）　他の製品と比較して論理的な複雑さが格段に大きい．

（３）　ソフトウェアの価値の基準が明確にできない．

（4） ソフトウェアはどの分野とも結びつき応用範囲が広い.

（i）人工物としてのコンピュータに依存する

ソフトウェアと他の工業製品との本質的な違いは,ソフトウェアはコンピュータという人間の作り出した人工物に依存していることである.他の製品は,材料の持つ重さ,硬さ,電気的性質といった物理的あるいは化学的な性質を受け継ぎ,物理法則や化学法則に従う.すなわち,その基礎に物理学や化学といった確固たる理論体系を持っている.

それに比較して,ソフトウェアは命令語の体系といったコンピュータ・アーキテクチャを基に作られ,いわば人間の作った約束事をベースにして組み立てられる.他の工業の場合も,製品を構成する部品や材料に対してＩＳＯ（International Standardization Organization）やＪＩＳ（Japan Industrial Standard）で標準化が図られ人間の約束事に従うが,物質そのものの持つ物理的あるいは化学的な性質は直接製品に現れてくる.

コンピュータ・アーキテクチャはメーカ毎に決められ,また時代の進展に従い発展しており,それまで開発されたソフトウェアが役に立たないということが頻繁に起こってきた.もちろん,コンピュータを構成する部品は,主に電気的性質という物理的な特性を持っており,実行速度といった面で影響するが,ソフトウェアで重要な論理面とは別である.

ＦＯＲＴＲＡＮやＣＯＢＯＬなどの高級言語や標準的なＯＳ（Operating System）の開発により,コンピュータ・アーキテクチャに依存しないソフトウェア開発が可能になったかのように見えるが,コンピュータ・アーキテ

クチャの微妙な違いによりソフトウェアの互換性がないといったことが起こってきた．長い時間で見るとコンピュータ・アーキテクチャや言語，ＯＳも進化しており，ソフトウェアに対する確固たる理論的な基盤が作れないという本質的な宿命を持っている．現在主流となっているIntel社のコンピュータ・アーキテクチャ，Ｃ言語等の言語，Microsoft社のＯＳもいずれは他に取って代わられることは明らかである．

ソフトウェア開発は，ほぼ半世紀前，コンピュータ誕生と同時に始り，またソフトウェアの体系的な開発を行うことを目指したソフトウェア工学がスタートしてからも約３０年経っている([2]，[3])．しかし，いまだに確立した開発手法や管理手法がなく試行錯誤をしているのは，他の工業のように確固たる理論的基盤に基づかないソフトウェアの持つ本質的な性質による．あえていえば，ソフトウェア開発についての開発手法や管理手法はコンピュータ技術の進化や応用分野の広がりなどにつれていつまでも進化する性質を持っているといえる．

ソフトウェア開発は，例えば小説など人間の知性に基づいた創作活動と対比できる．すなわち，自然言語と人工言語の違いはあるが，言語で記述する創作物と言う点で類似した面がある．源氏物語が書かれて１０００年以上たち，その間多くの小説が創作されているが，いまだに小説を創作するための「小説工学」がない，あるいはありえないのは明らかであろう．ソフトウェアと小説の違いは，小説はそれが読まれて人間の感性に訴えることが目的であるのに対し，ソフトウェアはコンピュータ上で稼動し，社会生活や生産活動に具体的に有用な効果を出すことを目的としていることである．実社会に

影響を与えるがゆえに,ソフトウェア開発を工学にしたいという願望があるだけかもしれない.

（ii）論理的な複雑さが他の工業製品に比べて格段に大きい

ソフトウェアはコンピュータを動かすことを目的として開発される.ソフトウェアを使ってコンピュータを動かすことにより,情報を処理したり他の機械を制御したりし外界に影響するが,ソフトウェアそのものは論理的な働きを記述するだけである.論理的な複雑性を他の工業製品と比較することは難しいが,論理的にもっとも複雑な構造をしている機械としてのコンピュータチップと比較する.

最近では論理素子の開発にはハードウェア記述言語で設計されるが,トロン仕様のチップ（TX1）を設計するのに約３０００ステップが必要であった[4].ハードウェア記述言語の記述能力はFORTRANやC言語などの高級言語とほぼ同程度であるので,論理的に最も複雑な機械であるコンピュータの複雑さは，ソフトウェアと比較して数千ステップ程度のものである．最新のコンピュータチップでも論理的な複雑度はその１０倍もないと推測できるので，高級言語で記述して高々数万ステップくらいである．最大のソフトウェアシステムは高級言語で１億ステップを超すものがあり[2]，その論理的な複雑さは桁違いである．

もちろん，コンピュータチップの設計には，電気的な制約やタイミングといった別の要素を考える必要があり，必ずしも単純には比較できないが，ソフトウェアは人類の作り出す最も複雑な論理的創造物である．

（iii）ソフトウェアの価値の基準が明確にできない

ソフトウェアの価値を数値化するなら，開発者と利用者が異なり，それが商品として扱われる以上，貨幣価値すなわち価格で表すことが妥当である．また他の製品でも同様であるが，企業活動として開発される以上，価格が開発に掛かった原価以上でなければ開発されない．パッケージソフトの場合，コピーの費用はほぼ無視できるので，どれだけ開発費用が掛かろうとも数多く売れれば利益が出るので，多く売れればより価値があるとみなせる．書籍やレコードなどの著作物と同じである．

受託契約で開発する場合，開発に要した技術者の工数，すなわち人件費が原価の大部分を占めるので，それ以上の価格がつかないと開発されない．工数が価値の基準になると優秀な技術者が短い期間で開発した場合，優秀でない技術者が長い期間をかけ開発した場合より価値が少ないことになりかねない．もちろん，現実には優秀な技術者の単価が高くなるという市場原理で妥当なところに落ち着こうが，ソフトウェアの価値を開発の工数というプロセスで評価することになる．

他の価値決めの基準として開発成果のプロダクトとしての価値を見ればいいのであるが，開発されたソフトウェアの量や質で測ることになる．開発された量，例えば開発ステップ数で測ることが行われているが，何度も見直すことにより洗練されたソフトウェアは一般的にステップ数が少なくなる現実があり，矛盾が起こる．開発されたソフトウェアの質，すなわち要求された機能が満足され，かつ信頼性などの品質が良いものといっても，その価

値あるいは価格をどう表すかは明確な基準がない．ソフトウェアの価値を決める明確な規準がいのが，ソフトウェアの本質的な性質である．

（iv）ソフトウェアはどの分野にでも結びつき応用範囲が広い

いま，仮にソフトウェア以外の工業製品開発を，ソフトウェアの反語としてハードウェアと呼ぶことにする．ハードウェア製品には，コンピュータ関連機器，家電製品，自動車，船舶，航空機，建設機器，医療機器など多岐にわたっている．ソフトウェアの場合，ソフトウェア技術者と一括して呼ばれるが，例えば自動車の技術者をハードウェア技術者と呼ぶことはない．それより，自動車のエンジン技術者，照明技術者，ミッション関連技術者といったようにより細分化がなされている．コンピュータ誕生の当初は，ソフトウェアは科学技術計算用だけに使われていたが，今日ではあらゆる分野へと応用が広がっている．

ソフトウェアによりコンピュータの応用が広がったわけである．すなわち，ソフトウェアはどんな分野にも適用できるという本質的な性質を持っているわけである．今のところ，ソフトウェア技術者という包括した名前で呼ばれているが，いずれ成熟すると，ソフトウェア技術者も分野毎に特化した専門技術者となろう．しかし，ソフトウェアそのものが現実社会に直接影響すると言うより，他の具体的な応用と結びついてはじめて意味を持つ．どんな応用分野とも結びつくという本質的な性質は変らない．

１．３　ソフトウェア開発の本質的な難しさ

ソフトウェア開発は，他の製造業と比較すると１．２で述べたような特性があるため以下に述べる根源的な難しさがある．

（１）ソフトウェア開発は，他の工業と対比する時，設計段階までであり製造段階がない．

（２）究極には技術者個人の創造性に依存する．

（３）契約関係で開発する場合でも，発注者が何を開発すべきか分からない場合が多い．

（４）人間は誤りを犯しやすい．

（５）定量化が難しい．

（ｉ）ソフトウェア開発は設計段階だけであり，製造段階がない

ソフトウェア開発は，他の工業と比較する時，広義の設計段階だけであり，製造段階がない．あえていうなら，ソフトウェアのコピーが製造段階に対応する．図１．１にその概念を示す．図１．１では，対比のため工業製品，ソフトウェア，および音楽レコードを例にしている．また，簡単のため，設計段階と製造段階だけがあるとしている．広義の設計段階には工業製品の場合，企画，設計，試作，テスト等が含まれ，量産を目的とするなら量産試作も含まれる．ソフトウェア開発では，企画，要求定義，設計，プログラミング，テストが含まれる．音楽レコード製作の場合，企画，作詞，作曲，吹き込みなどレコード原盤が出来るまでとする．

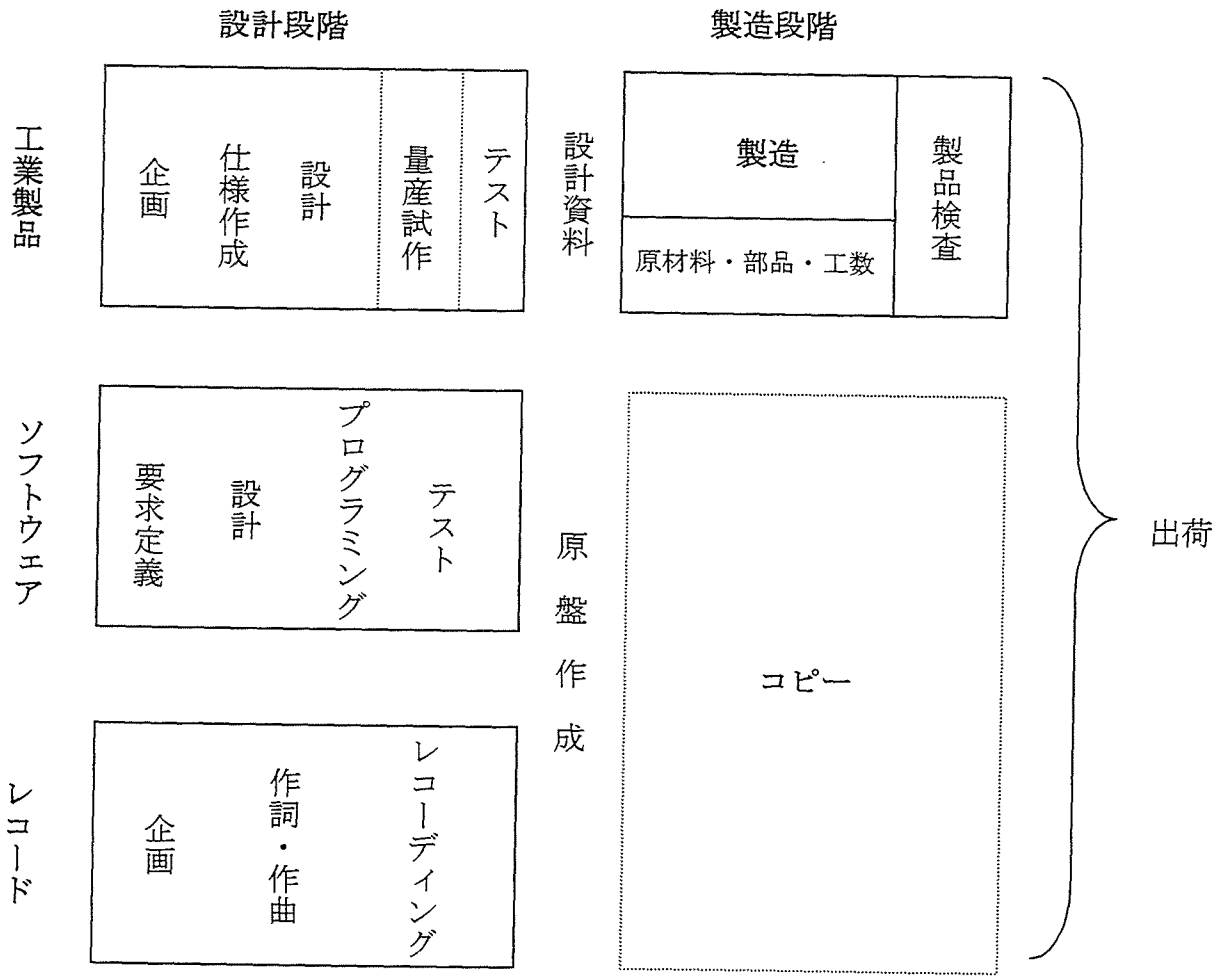


図１．１　ソフトウェア開発は設計段階（工業製品との対比）

設計が終わると，工業製品の場合，設計に基づき原材料，部品，および工数が投入され製造段階に移る．量産製品なら，製品の数に比例した原材料，部品，工数が必要である．特に量産ものでは，コストの多くが製造段階にかかり，コスト削減や生産性向上を考える時，努力の多くが製造段階に注力されていた．製造された製品は，製造が正しく行われたかどうかを調べるため検査される．不良率などの指標は，主に製造段階で発生する不良を計測するためのものであった．日本の製造業の強みは，主に教育水準の高さによる優秀な技能者がいるため可能となった製造段階の生産性の高さ，品質管理の強みであったと言える．ソフトウェア開発やレコード製作の場合原盤が出来ると，製造と言えるのはそれをコピーすることである．もちろん，コピーする数だけの CD(Compact Disk)などの媒体は必要であるが，全体の開発費用から見ると無視できるくらいに小さい．

ソフトウェア開発では，しばしばプログラミング段階をプログラム製造と呼び，工業製品の製造段階であるかのように誤解されてきたが全くの誤りである．確かに過去，ほぼ一意的な変換が可能にまで詳細な設計をし，それをプログラムに変換させるといった作業もあったが，たとえ一意的な変換が可能であり，作業は補助的で創造的でなかったとしても，設計段階の作業である．工業製品の設計の場合でも，概略設計に基づき，CAD（Computer Aided Design）装置へ入力する作業があり，CAD オペレータの補助的な作業と考えられているが，これもあくまで設計段階の作業である．手書きの設計書をタイピストに清書させる作業は設計の補助作業であるが，製造の作業でない

のと同じである．設計段階の単純な補助作業は，いずれツールなどに取って代わられ自動化される．

（ii）ソフトウェア開発は創造的な業務である

前項で議論したようにソフトウェア開発は設計段階の業務であり，そのため知的な創造性が必要とされる．工業製品の製造段階でも定型的で単純な作業は製造ロボットに置き換えられ，今後製造現場で必要とされる人材は，高度な技能を有する者，あるいは多機能工として柔軟性に富む者である．

ソフトウェア開発でも，ほとんど一意的な変換が可能になるまで記述された詳細設計からプログラムへ変換するだけ，あるいは厳密な帳票の仕様を決められた段階からその帳票を出力するほぼ定型的な COBOL プログラムだけを作成するといった単純な作業をするものを，プログラマあるいはコーダと呼んでいた．しかし，定型的な単純作業は，それを行うソフトウェアツールに取って代わられる傾向にある．

ソフトウェア開発は知的な創造性を必要とされる業務である．もちろん，業務の性質により，専門性が異なる．顧客の要求を聞き出しソフトウェアシステムの骨格であるソフトウェア・アーキテクチャを作り出すシステムエンジニアやプロジェクトチームを纏め上げるプロジェクトマネージャは，顧客などとの人間的な折衝というより高度な能力が問われるし，ソフトウェアシステム設計を行う設計者は，使用するハードウェアなどの制約のもとで最適な解を見いだす能力が問われる．プログラミングやテストを担当する者も，最善のプログラミングやテストを行うためには，自己の知識や経験を生かし

た高度の専門性を必要とし，知的・創造的な能力を問われる．いずれにしても，ソフトウェア技術者個人個人が能力を高めるとともに，その創造的な能力を生かせるような開発手法や開発管理手法が求められる．

１９９０年頃，Cusumano が，当時世界を席巻した日本の製造業の強さに関心を示し，ソフトウェア開発においても日本が世界を席巻するのではないかとの考えから，日本のソフトウェア工場を調査したことがある[5]．日本のソフトウェア開発の基本方式は，製造業と同じくウォーターフォールモデル（図１．２参照）に基づく，工程ごとの分業方式であった．しかし，確かに日本の企業は生産性や品質は高いがソフトウェアで大きな利益をあげている事例はない．ソフトウェアで莫大な利益を出しているのは Microsoft 社であるという現実から，改めて Microsoft 社の開発手法を調査した．Microsoft 社のソフトウェア開発は，多くの技術者の共同開発でありながら，数多くの少人数よりなるチームで開発する方式である．これにより，多人数での共同開発ではあるが，個人の独創性や少人数のチームによる機動性を生かせるようになっている[6]．これについては第５章で詳細に考察する．ソフトウェア開発で成功するには，技術者の創造的な業務を支援するような開発手法や管理手法が必須の条件である．

（iii）ソフトウェアの発注者は明確な仕様を提示できない

パッケージソフトや大量生産される製品の組込みソフトの場合，開発すべき仕様は開発者が決める．開発コストを上回る売上があれば利益が出るし，

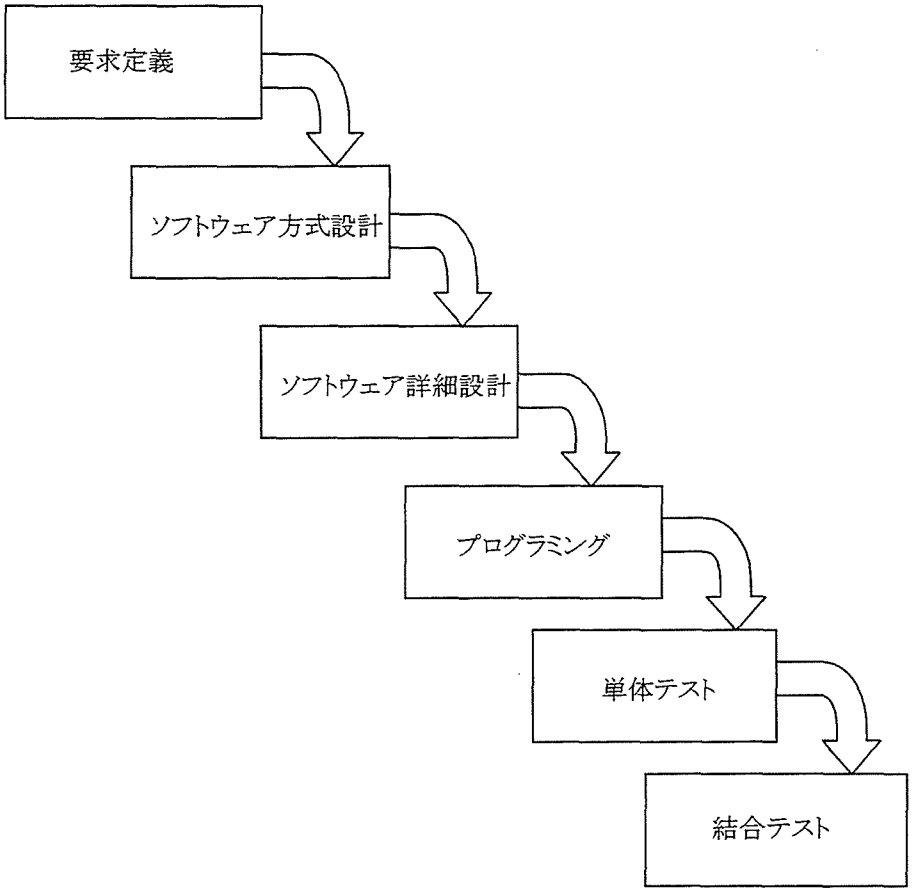


図1．2 ソフトウェア開発のウォーターフォールモデル

売上が開発コストに満たない場合，事業として失敗するだけである．

受託契約でソフトウェアを開発する場合，発注側から仕様が提示され，価格・納期など付帯条件が明確になってはじめて契約されるのが契約の基本である．しかし，特にソフトウェア開発に関する我が国の商習慣では，仕様など契約条件が曖昧なまま開発に着手する事が多い．発注側はソフトウェアで実現したい業務知識を良く知っているが情報技術についての知識が不足しているし，開発側はその逆で情報技術には熟知しているが発注側の業務知識が不足し，発注者が本当に何を求めているのか明確に出来ない．例えば，開発段階の終わりごろになって，発注側にも分かる具体的な帳票や画面が出てはじめて仕様上の細かい指摘をし，しばしば，システム全体の方式にまで影響する仕様変更の要求が起こる．仕様変更がソフトウェア開発のコストを上げ，納期を遅らせる最大の要因であることはよく知られている．

契約関係が明確な欧米では，開発者あるいは第三者の専門家が，まず開発すべき仕様を明確にすると言う段階で契約を行う．仕様が明確になった段階で，その仕様に基づき改めて開発についての契約をすることになる．

しかし，ソフトウェア・ライフサイクルを何度も回し段階的に仕様を明確にするためのスパイラルモデルやプロトタイピングモデルの提案は欧米から出ており[7]，開発に先立ち仕様明確に確定できないのはソフトウェア開発の本質的な難しさであることを示している．

（iv）人間は誤りを犯しやすい

ハードウェアの場合，部品や材料の経年疲労による問題などがあるが，ソ

フトウェアの場合，開発はすべて設計段階であり，そこで生じる殆んど全ての誤りは，人間的な要因で生じる（[8]）．誤りの主な原因として，組織的要因によるものと個人的要因によるものとに分類できる．組織的要因は一言で言えば組織成熟度の低さであり[9]，検査不足，情報伝達の不備，情報共有の不備などに現れる．個人的な要因は，個人の性格や行動様式，意欲や機嫌といった人間的な要因と，経験不足，知識不足，理解力不足といった技術力不足による要因がある．結果としてコスト高，納期遅れ，品質問題による客からのクレームとして現れ，短期的には直接経営に打撃を与え，信用失墜により長期的にも経営に影響する．

解決策はいろいろ考えられるが，人間は誤りやすいということを是認して，その上でいかに予防するか，あるいは混入した誤りを摘出すべきかを考えざるをえない[10]．

（ｖ）定量化が困難である

ソフトエア開発では，その開発過程，すなわちプロセスと，開発結果としてのソフトウェア製品，すなわちプロダクトとの両面で計測される．プロセス測定の目的は主に管理のため，プロダクトの測定の目的は主に品質確保のためである．

例えば，ＩＳＯ（International Standardization Organization）ではソフトウェアの品質特性を標準化し決めているが，それをいかに計測するかについては明確な指針がない．表１．１はＩＳＯ標準に従った品質特性表の例であるが，品質特性あるいはその細目である副特性を計測するための尺度を

表１．１　品質特性表

品質特性 Quality Characteristics	品質副特性 Quality Sub-Characteristics	計測尺度 Metrics
機能性	合目的性	ユーザ改良要求率
	正確性	操作説明書と実動作の合致度
	セキュリティ	暗号化率
	互換性	制御文字コード標準化率
	接続性	データ形式の合致度
信頼性	無欠陥性	平均故障発生間隔・残存誤り密度・条件分数
	誤り許容性	誤入力・誤操作検出率
	可用性	稼働率・平均復旧時間
使用性	理解性	デモストレーション装備率
	習得容易性	習熟時間・ヘルプ機能装備率
	操作性	収束作業時間・タッチ回数
	対話性	状態進捗表示率・ガイド機能装備率
効率性	実行効率	レスポンス時間・トランザクション処理件数
	資源効率性	主記憶使用量・ファイル使用量
保守性	拡張性	モジュール結合度・強度
	変更容易性	平均修正時間・パラメータ化率
	テスト容易性	原因分析支援機能装備率（ログ等）
移植性	導入容易性	導入作業時間
	再利用性	プログラム変更率
	ハードウェア独立性	適用可能機種率・ハード独立関数使用率
	ソフトウェア独立性	適用可能ＯＳ率・ソフト独立関数使用率

示している．ＩＳＯで決められた品質特性を計測するのは，現実の開発現場では難しいものが多い．

開発規模を表すのに，従来より開発ステップ数が用いられていたが，開発規模の尺度として問題が多いことが知られている．ステップ数の代替尺度として，開発すべき機能量で表すファンクションポイント法が早くから提案されており[11]，その改良や普及のため多くの研究や努力がなされているが，この手法を使うのには習熟する必要があること，ファンクションポイント値に経験的・主観的な要素があることなどがあり，いまだに定着していないのが現状である．

１．４　ソフトウェア工学の成果とその目指すところ

ソフトウェア工学は，複数の開発者でソフトウェアを体系的に開発することを目指している．研究課題として大別すると，ソフトウェア開発手法および開発管理手法に分かれ，多岐の項目がある．表１．２はCarnegie Mellon大学のＳＥＩ（Software Engineering Institute：ソフトウェア工学研究所）がまとめた学部学生のソフトウェア工学のカリキュラムである[12]．それぞれの項目は，今までのソフトウェア工学の研究分野を表しており，大学の教育で教えるに足りるほどに成熟し成果を上げてきた．

教育の目的は，専門的な技術者となるのに必要な知識や技術を教えることであるが，技術者が働く環境，すなわち職場環境や開発支援環境も重要な要素である．

表１．２　ソフトウェア工学のカリキュラム
（ＣＭＵ／ＳＥＩ［9］から）

１．ソフトウェア工学プロセス（Software Engineering Process）
２．ソフトウェアの進化（Software Evolution）
３．ソフトウェア生成（Software Generation）
４．ソフトウェア保守（Software Maintenance）
５．技術コミュニケーション（Technical Communication）
６．ソフトウェア構成管理（Software Configuration Management）
７．ソフトウェア品質（Software Quality Issues）
８．ソフトウェア品質保証（Software Quality Assurance）
９．ソフトウェアプロジェクトの組織と管理（Software Organizational and Management Issues）
１０．ソフトウェアプロジェクトの経済学（Software Project Economics）
１１．ソフトウェアの運用（Software Operational Issues）
１２．要求分析（Requirements Analysis）
１３．仕様（Specification）
１４．システム設計（System Design）
１５．ソフトウェア設計（Software Design）
１６．ソフトウェア実現（Software Implementation）
１７．ソフトウェアテスト（Software Test）
１８．システム統合（System Integration）
１９．組込み型実時間システム（Embedded Real-Time System）
２０．ヒューマンインタフェース（Human Interfaces）
２１．職業としてのソフトウェア技術者（Professionalism）

ＩＳＯ９０００シリーズやCarnegie Mellon 大学ＳＥＩが提唱しているＣＭＭ（Capability Maturity Model：組織成熟度モデル）[9]は，品質の高いソフトウェアを開発するには,その前提として開発組織がきちんとした仕事の手順を規定化し，その手順に従い仕事をすることが必要である，また，開発組織が自己学習により成熟度を成長させるべきであるとの仮説で規定化され実施されている．すなわち，品質の高いソフトウェアは，成熟した組織で常に改善されるプロセスにより生み出されるとしている．

開発支援環境に関しては，技術者がその能力を十分発揮し，生産性を上げるのに十分な設備環境を与え，開発業務の支援をするべきである．

１．５　本論文の構成

本論文の構成は，図１．３に示すようになっている．第１章では，ソフトウェアの持つ特性およびソフトウェア開発の難しさについて議論し,ソフトウェア開発を体系的に行うためのソフトウェア工学について述べた．

第２章および第３章では実際の企業において,ソフトウェア工学の成果を実践した事例について述べる．第２章では，ソフトウェア工学の成果を集大成しソフトウェア開発の工業化を目指した統合ＣＡＳＥ（Computer Aided Software Engineering）環境について述べる．第３章では，ソフトウェア品質の向上を目指し，全社的な設計レビュー制度の実践例について述べる．

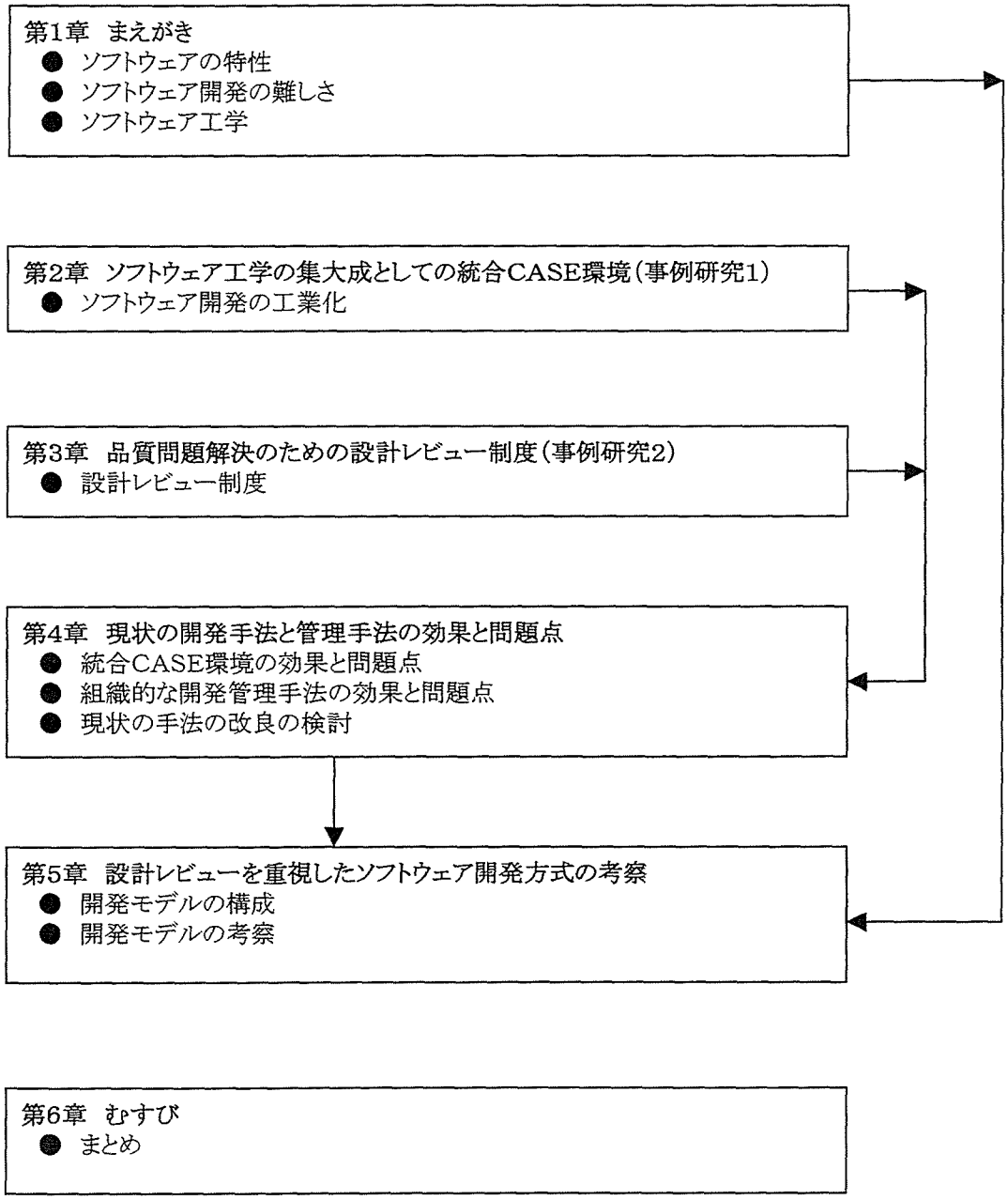

第1章 まえがき
● ソフトウェアの特性
● ソフトウェア開発の難しさ
● ソフトウェア工学
第2章 ソフトウェア工学の集大成としての統合CASE環境(事例研究1)
● ソフトウェア開発の工業化
第3章 品質問題解決のための設計レビュー制度(事例研究2)
● 設計レビュー制度
第4章 現状の開発手法と管理手法の効果と問題点
● 統合CASE環境の効果と問題点
● 組織的な開発管理手法の効果と問題点
● 現状の手法の改良の検討
第5章 設計レビューを重視したソフトウェア開発方式の考察
● 開発モデルの構成
● 開発モデルの考察
第6章 むすび
● まとめ

図1．3 本論文の構成

第４章では，第２章および第３章で述べた実践例を，第１章で述べたソフトウェアの特性や開発の困難さについて対比しながら議論し，問題解決の糸口について触れる．

第５章では，第４章の議論を踏まえ，設計レビューを重視した新しいソフトウェア開発手法および開発管理手法の提案を行い，その効果について議論する．

第６章で，本論文のまとめを行う．

第２章　　ソフトウェア工学の集大成としての統合ＣＡＳＥ環境：ＩＭＡＰ

ＩＭＡＰ（Integrated Software Management and Production Support System）は，ソフトウェア工学の成果を統合ＣＡＳＥ（Computer Aided Software Engineering）環境として実現し，ソフトウェア開発の工業化を目指したものである．

２．１　ソフトウェア開発工業化の概念

ＩＭＡＰでは他の工業生産との対比で，ソフトウェア開発の工業化とは開発過程を工程に分割し，各工程の作業基準を規定し，品質基準を満たした製品を指定された期間内に一定の工数で開発できるようにすることであるとした．

この工業化を実現するためには，

（１）　管理手法の確立

（２）　作業手順の明確化と分業化

（３）　ソフトウェアの部品化・再利用

（４）　開発環境の確立

（５）　技術者育成のための教育体系の確立

が必須の条件である.

（ｉ）管理手法の確立

ソフトウェア開発は，第１章で述べたように，他の工業と対比させると全てが設計段階に位置し，知的生産活動である．それ故，開発過程においても成果物においても，その詳細な意味内容を客観的に見ることは困難である．このため,管理者は技術者の報告を自己の経験と照らして憶測するしかなく,開発の進捗状況・問題点の把握に苦慮しているのが実情である．ソフトウェアの内容を詳細に理解できなくても,ソフトウェア開発を直接担当している者以外の第三者が客観的に現状が把握でき適切な行動を取れる管理手法が必要である.管理手法の確立のためには客観的な実績データの収集が原点となる．基礎となるデータの収集は，正確性，客観性，即時性などの理由で，できる限り技術者の手を煩わせることなく自動的に収集できるようにすべきであり，そのための制度・機構を整備することが必要である．収集されたデータは，工程管理，品質管理，成果物管理などの管理目的に合わせてモデル化し，統計処理により所要の管理資料に変換する.

こうすることにより抽象的･知的成果物であるソフトウェアの管理は数値化され，客観化し，図表などを使用することによって可視化が可能となる．さらに管理者と技術者は,数値化された指標の性質や傾向を理解することにより指標の見方や使い方の合意が得られ,効果的な管理法が確立することに

なる.

（ii）作業手順の明確化と分業化

ソフトウェア開発の各工程の作業手順をソフトウェア工場として規定化し，各工程で使用する帳票・用語・記述方法を標準化する．各工程への入力データ,出力としての成果物を明確にし,その成果物の品質基準を規定する.

工程間の受け渡しは,前工程で品質を保証した成果物を入力として次工程へ引き渡す（図２．１参照）．分業化とは，工程ごとの分業化と機能ごとの分業化という二通りの意味がある.前者は開発の各工程を担当する専門家により，要求仕様から次々に加工され，製品として完成する事を意味する．後者は直接開発に携わらないものも含めて品質の評価,管理データの収集や分析，成果物の管理，後述する標準部品の開発など，間接的に開発を支援する専門家による組織化を意味する.

（iii）部品化・再利用の促進

ハードウェアと同様，ソフトウェアにおいても，品質の保証された標準部品の存在が工業化の前提となる.

同じ分野の類似ソフトウェアの開発において,既存のソフトウェアの流用,あるいは繰り返し使用される部分を“切り出し”，再利用することは良く行われている．特に同一組織においては，既存ソフトウェアの流用や部品の切り出しによる再利用は６０％以上が可能であり,生産性や品質の向上に貢献している．しかし切り出された部品を，他の分野あるいは他の組織の

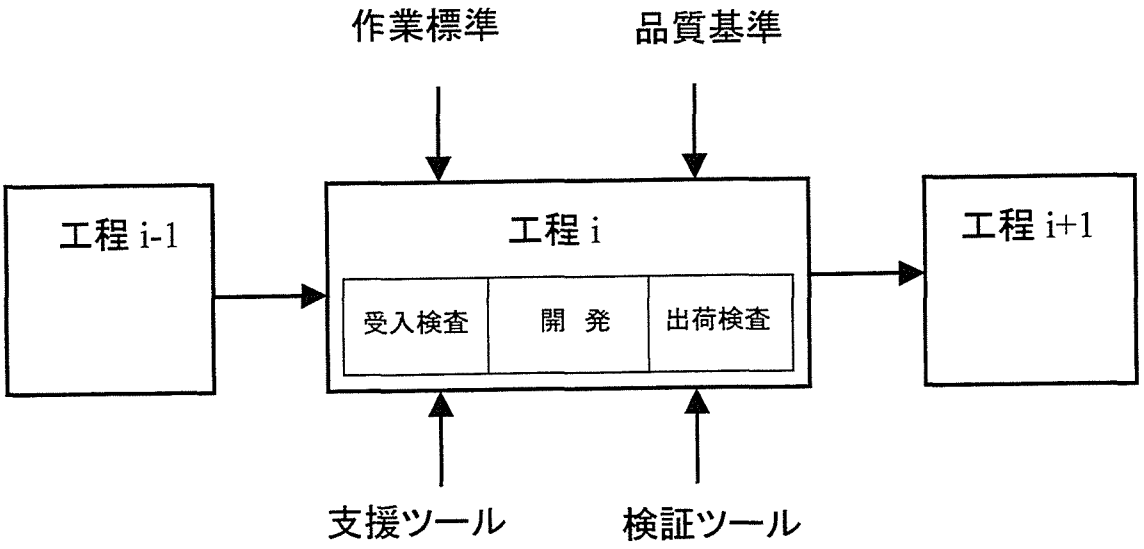


図2. 1 IMAPにおける工程間の受け渡し

ソフトウェア開発に再利用しようとすると，切り出された部品の理解や修正・カスタマイズに手間取り，新規に開発する以上の工数がかかることがあり，トラブル時に対応しきれない例も発生している．

ソフトウェア開発の工業化における標準部品とは，既存ソフトウェアからの切り出しではなく，再利用することを前提とした部品を，標準部品規定に基づき事前に作成し，品質を保証し，正式な手続きを踏んで認知された部品のことである．この標準部品規定には，部品仕様の記述法，インタフェース仕様，品質基準と検査方法を整備し，標準部品を認知する正式の機関の位置付け，権限，構成メンバーの規定も含まれる．

このようにして作成された標準部品の再利用を促進するためには，部品の蓄積・検索・活用が容易にできる支援ツールが必要である．設計してから仕様に合った部品を検索するのではなく，標準部品に合わせた設計法を確立する必要がある．

品質の保証された標準部品を使用することはソフトウェア開発の生産性向上のみならず，ソフトウェアの品質向上にも不可欠である．

（iv）　開発環境の確立

ソフトウェア製品が工芸作品から工業製品化するためには，ソフトウェア工場全体の設備計画や組織体制に裏打ちされた開発環境の確立が必要である．設計者の個性によりソフトウェア製品設計，開発設備，支援ツールが異なっていたのではソフトウェア工場は確立できない．組織として分業体制を確立し，工場全体としてのソフトウェア開発管理システムを確立し，各部門

共通の汎用設備と，部門別，分野別あるいは製品別専用設備を明確にし，ソフトウェア工場全体としての設備の有機的結合を図ることが必要である．

このようにしてソフトウェア製品の流れと管理が一体化し，工場として品質保証した製品を開発できる環境整備が必要である．

（v）技術者育成のための教育体制の確立

ソフトウェア開発は技術者の能力に依存する要素が大きく，ソフトウェア開発の工業化にとっても技術的な教育・訓練だけでなく，組織として活動するのに必要な技術者の“しつけ”は重要な課題である．

教育の主なねらいは職人から企業人への変革であり，そのために

（1） 作業基準・標準化規定の徹底

（2） 全体システムの理解

（3） 最新技術の習得

が重要となる．技術者のレベルアップ・士気向上こそがソフトウェア開発工業化の基礎である．そのためには，教育体系を整備し，組織的・計画的に継続して行う必要がある．

2．2　IMAPの特徴と機能

IMAPは前節で述べたソフトウェア開発の工業化を目指した統合CASE環境である．本節ではIMAPの基本構成，組織化，設備環境について述べる．

（ｉ）　ＩＭＡＰの構成

ＩＭＡＰはソフトウェア開発の各工程を支援するするツールを統合化し，ソフトエア開発および開発管理を一貫して支援する統合ＣＡＳＥ環境である．ＩＭＡＰではソフトウェア開発の工業化という観点から，従来システムの構成を見直し，

（１）ライフサイクルの各工程を支援するシステム

（２）共通支援ツール群

（３）開発支援ツール群

（４）管理システム

の４つのブロックよりなる構成をとった（図２．２参照）．

（１）のライフサイクル支援ツール群は，機種ごとの違いにはとらわれない汎用的なツール群よりなる．（２）の共通支援ツールは，標準部品の再利用を支援するツール，マイクロコンピュータのソフトウェアに見られるような，機種ごと，ＯＳ（Operating System）ごとの違いを吸収するツール群として分類し，今後の技術移転の容易化を図った．また，ソフトウェア開発という観点から，ネットワーク，データベースなどの（３）の開発支援ツール群をＩＭＡＰの基盤と位置付けた．ソフトウェア開発の工業化という観点から，開発支援と対等の立場で（４）の管理システムを位置付け，各ツールの開発を行う．

ライフサイクルの工程として位置付けている，“５０ＳＭ開発”は単一の

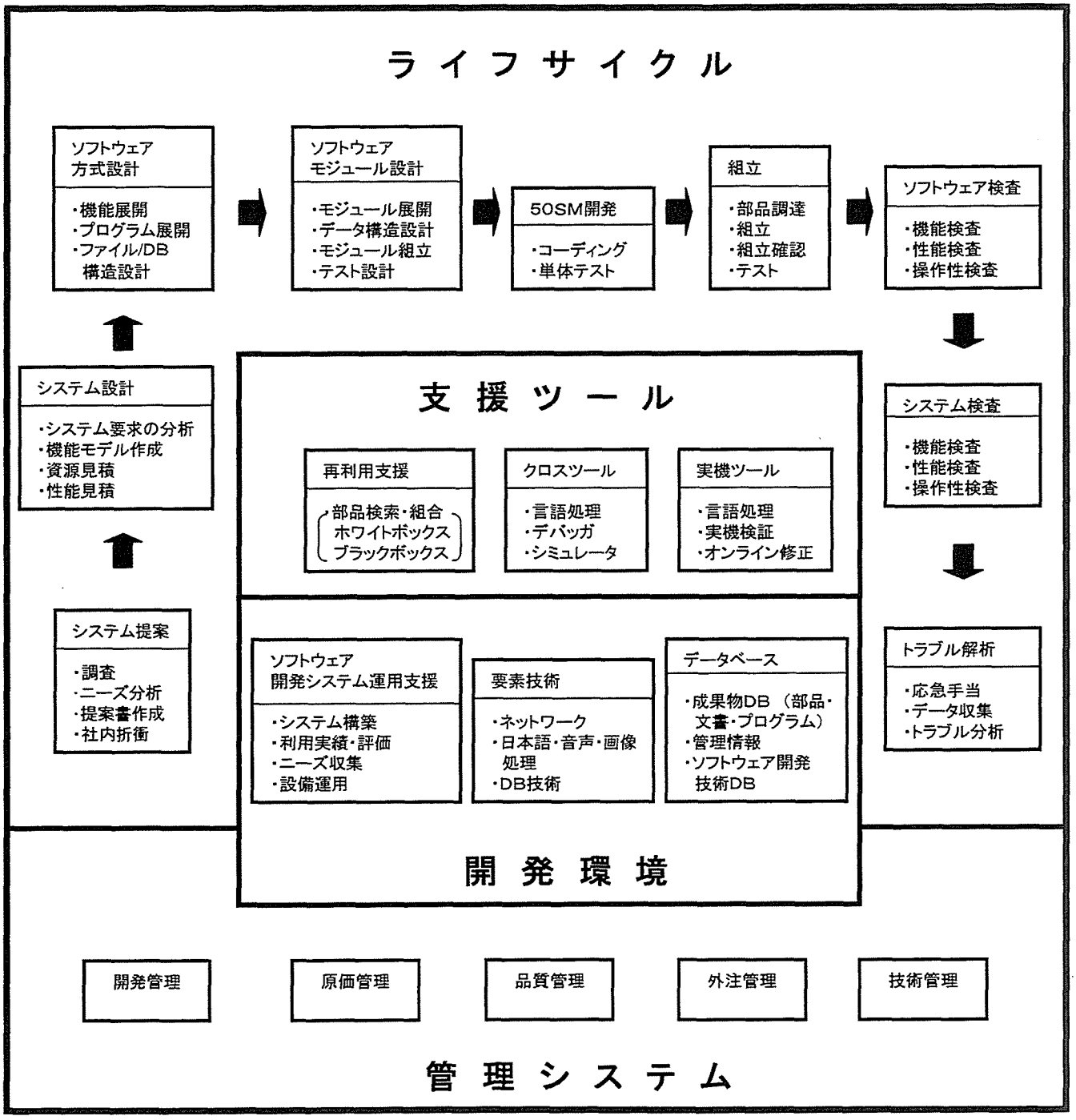


図2. 2 IMAPソフトエア開発支援体系図

モジュールの詳細設計およびコーディング・デバッグを行う工程である．５０ＳＭ（Fifty Steps / Module）は，１モジュール５０ステップ以内とすることを意味し，それを強調するため明示的に表した．５０ステップ以内のモジュールなら，モジュールの開発の容易性，品質保証，および理解性のいずれの観点からも有効であるとの経験を積極的に生かそうとしたものである．

“トラブル解析”は通常“保守”といわれる工程である．これまで保守という言葉で，“不具合の修正”と共に“機能の拡張”の意味でも使われてきた．ＩＭＡＰでは保守を不具合の修正だけに限定し，機能の拡張は，既存のソフトウェアを利用した再開発であると考え，ライフサイクルの各工程を最初から始めれば良いと考える．

図２．３は，ＩＭＡＰの機能モデルを示している．特徴として３つのＤＢ（Data Base：データベース）があり，それを中心にして，ライフサイクルの各工程を成果物が流れる仕組みである．

標準部品ＤＢには標準ソフトウェア部品（仕様書，プログラム，検査成績書，使用状況を含む）が蓄積・管理され，部品検索システムにより設計支援システムにおける再利用の促進が図られる．

成果物管理ＤＢは，開発完了した成果物を管理し，納入システムの修正・変更に速やかに対処するためだけでなく開発中の成果物も管理し，工程間の受渡し，ツール間のインタフェースを標準化させる機能も持っている．

管理ＤＢには，技術者に意識させることなく，バックグラウンドで実績データを自動収集するシステムと，収集されたデータを自動的に加工するシステムを連動し，各種管理データを蓄積させる．

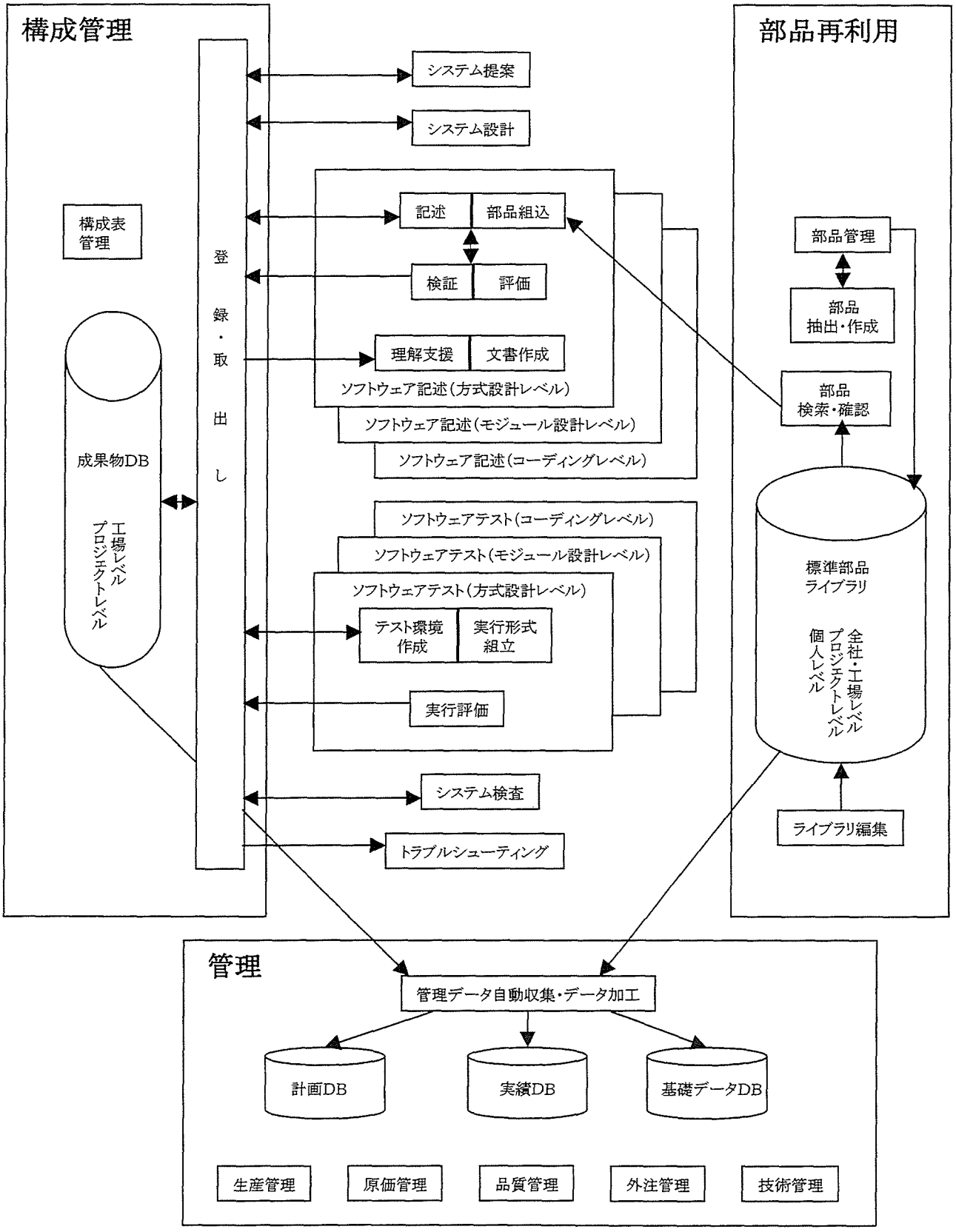


図 2.3　IMAP機能モデル

ソフトウェアライフサイクルを支援するツールの主なものとして

（1）要求定義を行うツール

（2）設計書からプログラムへの変換を支援するツール

（3）レビュー・検査を設計段階で行うための支援ツール

（4） 設計支援ツールと連動した部品検索ツール

（5） ソフトウェアの品質を設計段階で評価するツール

があり，ソフトウェアの一貫開発を目指した各種ツールの整備を行った．

（ii）組織化

組織は，開発すべきソフトウェアの規模，応用分野により変化するのが普通である．ここでは，ＩＭＡＰとしての基本的な考え方を述べる．

ソフトウェアの分業化を促進するために，ソフトウェア工場全体を見る開発管理部，ソフトウェア開発を行う開発部，品質管理を行う品質管理部のほかに，ソフトウェア部品を専門に作成する部品開発部を独立させることを考えている．

前節で述べたように，責任範囲の明確化，作業規定の定着化のため，組織の独立化は必要である．組織の分化は，組織の硬直化・官僚化の恐れを内在するが，ソフトウェアの開発はあまりに柔軟に行われ過ぎているので，工業化という観点で組織を独立させることが必要である．

（iii）設備環境

このＩＭＡＰの実現には，大型コンピュータをセンターホストマシンとし

て用い，工場全体としての開発管理システムや各種管理資料の作成，完了したソフトウェアの成果物の管理を行う．各部門ではサーバマシンをローカルホストマシンとし，ファイルサーバ，プリントサーバなどの機能を持たせる．技術者には，各種ＥＷＳ（Engineering Work Station）が提供される．このＥＷＳでは，ソフトウェアライフサイクルの“ソフトウェア方式設計”から“ソフトウェア検査”まで連続して行えるようになっている．センターホストマシン，ローカルホストマシン，およびＥＷＳは，ＬＡＮ（Local Area Network）で接続され，しかも資源の共有が可能な密結合となっている．この結合のことをＩＭＡＰでは，“汎用環境”と呼んでいる．

汎用環境で開発されたソフトウェアは，製品に依存した専用機やマイクロコンピュータ組込製品などを搭載したテスト環境（専用環境）で，ソフトウェアの検査を行えるようになっている．

ソフトウェア開発環境を汎用環境と専用環境に整理したことから，汎用環境によりソフトウェア開発の標準化，資源の共有化が可能になり，専用環境により実ジョブの変化にダイナミックに対応して，品質向上および検査の効率化を可能にする．

２．３　ＩＭＡＰの波及効果

ＩＭＡＰの開発と並行して，全社技術管理部門を中心に，ソフトウェア生産性向上のための運動を展開した．この運動では，ＩＭＡＰの概念に基づき

（1）社内のソフトウェア開発規定の全面的な見直し

（2）ソフトウェア開発の標準化を図るための品質管理体系図の作成

（3）各部門でのカスタマイズ

（4）技術者の設備装備率の向上

（5）諸施策普及のための教育活動

を展開した.

統合ＣＡＳＥ環境としてのＩＭＡＰそのものは,一部のツールが全社的に使われたものの，システム全体の全面的な活用はなされるに至っていない.これについては第４章で議論する.

この運動の延長として，その後社内では，ソフトウェア生産性向上を目指した，ＣＡＳＥ普及の社内運動，ソフトウェア品質向上のための運動，オブジェクト指向技術普及のための運動を継続して行っている.間接的ではあるが，ソフトウェア生産性と品質向上に関する，社内意識改革がＩＭＡＰの最大の効果である.

第３章　品質問題解決のための設計レビュー制度

本章では,品質改善の取り組みの一つとして行った全社レベルでの設計レビュー制度の徹底実施とその効果について述べる.設計レビュー制度の当初の適用対象は，ソフトウェア開発であったが，現在では全事業をその適用対象としている.

品質問題の解決手段として，ＩＳＯ９０００シリーズの適用や，ＩＳＯ１２２０７（ソフトウェア・ライフサイクル・プロセス）に基づくＳＬＣＰ[12]の適用による,開発状況の可視化,開発プロセスの標準化が注目されていた.さらに，開発プロセスを評価するCMM（Capability Maturity Model）が注目されている[9]．これらの状況に鑑み，ＩＳＯ９０００認証取得が経営上必須の条件となった時点では,いつでもその取得ができるようなレベルになることを当面の目標とし，段階的に品質向上の施策を行うことになった．この施策の各段階で，具体的な効果が得られることを目指した.

３．１　全社運動の背景

３．１．１　ロスジョブの撲滅

ロスジョブとは,開発費用が売上金額を上回るようなソフトウェア開発プロジェクトのことである．ロスジョブによる損失合計が，全社の利益にも大きく影響を及ぼす実情があった．特に，規模の大きい大型プロジェクトでの

ロスジョブが発生したときには，金額的にも大きく，会社の存亡にも関わる事態を起こす懸念があった．ロスジョブをいかに少なくするかが，経営の観点からも火急の課題となっていた．

３．１．２　ロスジョブ分析

ロス金額の大きい物件についてはそれまでも，その原因を分析していた．ロスジョブ分析によると，ロスの大きな要因は表３．１のようになっている．この分析を受け，上流工程の重要性が認識されていたので，まず，プロジェクト開始前の設計レビューを徹底して行うことに決まった．

３．２　品質システムの概要

社内には，すでに以前から制定された品質システムおよび品質管理規定がある．これらの規定は制定されてから年月が経っていること，事業構造の変化などから，部門ごとの事業が多種・多岐にわたり，必ずしも一律な適用が出来ないことなどから，実施については各部門に任され，部門毎の実施状況にはばらつきがあるのが実情であった．そのため社内品質システムの改善にあたり，ＩＳＯ９０００を基本として改訂し，合わせて各種管理規定を整備することになった．

表3.1 ロスジョブの要因

ロスジョブの要因	関連度
1. プロジェクト受注時のレビューの不足 ・見積り精度が悪い ・仕様確定の遅れ	36%
2. プロジェクト開始時の開発計画の不備 ・新規技術・業務の対応 ・開発計画の不備	25%
3. プロジェクト運営の不備 ・コミュニケーション不足 ・管理不足 ・設計不備	19%
4. 製品出荷時の審査の不徹底	20%

３．２．１　設計レビュー制度の概要

社内の標準的な開発プロセスは，ウォータフォールモデルに基づき，引合い，受注・プロジェクト計画，設計・開発に関わる各工程，出荷の各プロセスから成る．それぞれのプロセスの間では，設計レビューを実施する事になっている（図３．１参照）．

各段階で行われる，DR-X，DR－０，および DR－１～６，および DR－F については，３．２．３において述べる．

品質改善の具体施策の第一段として，全社設計レビュー制度として徹底実施することになった．部門ごとのばらつきをなくすために全社レベルでの実施手順を決め，それぞれの部門に特有のチェック項目を追加するようにした．

３．２．２　設計レビューの実施責任者

設計レビューの実施は原則として各部門で行う．プロジェクト毎の重要度を，受注金額，業務・業種や必要な技術の習熟度などで決め，重要度の高い A ランクと指定されたものについては，全社の品質管理部門も設計レビューに参加する．設計レビューの実施状況については，全社レベルでフォローし，定期的に経営陣に報告する事になっている．開発するソフトウェアの重要度に従い，実施責任者のランクを表３．２のように決めた．

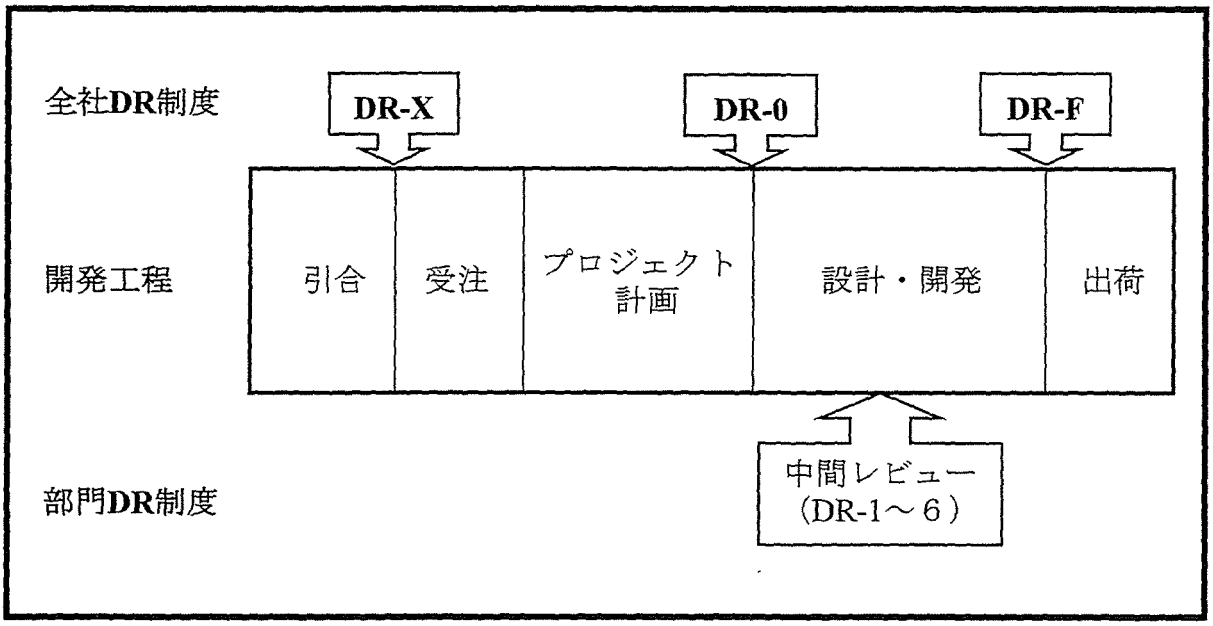


図3．1　設計レビュー制度の概念図

表３．２　　種別・ランク毎の設計レビュー実施責任者

	DR-X	DR-０	DR-１～６	DR-F	
				品質審査	出荷承認
Aランク	事業部長 (本社部門： 技師長 または 品質管理部長)	事業部長 (本社部門： 技師長 または 品質管理部長)	技術部長	事業部長 (本社部門： 技師長 または 品質管理部長)	営業部長
Bランク	部長	部長	技術部長 または 技術課長	部長	営業部長

注：①　Aランク物件の設計レビューには本社部門が立ち合う
　　②　部門毎に責任者を決定する

３．２．３　設計レビューの実施内容

設計レビューの種別は，プロジェクト開始前のＤＲ－０から始まり，設計・開発の各工程間のＤＲ－１～６，出荷を判定するＤＲ－Ｆがある．後述するように，一部の部門で行われていた，引合いから受注を決める段階でのＤＲ－Ｘと称するレビューも本制度に取り組まれた．表３．３に，各設計レビューでの主な実施項目を記述する．

設計・開発の各工程間で行う設計レビューＤＲ－１～６は

ＤＲ－１：ハードウェアを含むシステム設計が終わった時点

ＤＲ－２：ソフトウェアシステム設計が終わった時点

ＤＲ－３：ソフトウェアモジュール設計が終わった時点

ＤＲ－４:モジュールのプログラミングおよび単体テストが終わった時点

ＤＲ－５：ソフトウェアシステムの結合試験が終わった時点

ＤＲ－６：システム全体の総合テストが終わった時点

に行うことになっている．

３．３　段階的な設計レビューの徹底実施

本制度を，ＩＳＯ９０００認証取得の準備段階と位置付ける[14]．ＩＳＯ９０００取得のためには，社内品質システムの整備とともに，その実施を証拠付けるための文書によるエビデンスが必要になる．部門ごとに，さまざまな取り組みがなされている現状からみて，最初からＩＳＯ９０００取得を目

表３．３　デザインレビューの実施内容

		DR−X	DR−０	DR−F
実施方法	実施時期	プロジェクトの引合い時	プロジェクトの開始前 （受注後すみやかに実施）	製品の出荷前
	実施者	・営業責任者 ・技術責任者 ・プロジェクト責任者 ・判定責任者	・技術責任者 ・プロジェクト責任者 ・営業担当 ・判定責任者	・技術責任者 ・プロジェクト責任者 ・営業担当 ・判定責任者
	レビュー対象	・RFP(Request For Proposal) ・リスク判定票 ・DR−Xチェックリスト	・プロジェクト計画書 ・DR−０チェックリスト	・成果物および品質記録 ・DR−Fチェックリスト
	実施結果	・受注是非 ・リスクランク ・フォロー責任者を決定する.	プロジェクト開始の是非を ・合格 ・条件付き合格 ・不合格 として判定する.	製品の品質を ・合格 ・不合格 で判定する. 出荷の可否判定は別途出荷承認が必要である.
実施のポイント		①RFPに基づき ・契約範囲 ・顧客と当社の役割分担 ・社内技術力の有無 などを評価し、実現可能性を判断する. ②・実現可能性 ・リスクの影響度 ・納期遅延 などを考慮し、リスクランクを判定する.	①・顧客と合意した内容と合意に達していない部分を明確化 ②・詳細スケジュール ・中間レビュー計画 ・品質計画 を設定 ③・仕様確定度 ・見積制度 ・対価確定度 ・必要技術者参入度 ・リソース確定度 ・工程確定度 ・客先責任体制 などがチェックの対象	①製品毎の検査指標 ・検査密度 ・エラー発生密度 に照らして判定する ②品質審査（DR−F） に合格しないものは、原則として出荷出来ない.

指すのは現場に混乱を与えるなど現実的でないと判断し,段階的に実施することになった.ただし,実施の各段階で具体的な効果が出ることを目指した.

３．３．１　ＤＲ－０の実施

ロスジョブの分析などを考慮し,早い段階でリスクを避けることを目指し,まず,ＤＲ－０（開発に着手して良いかどうかを判定する設計レビュー）を全社的に実施することになった.施策推進のためのキャッチフレーズとして,「ロスジョブの撲滅」のように,過大とも見える目標を掲げたが,結果的には言い過ぎでもなくなっている.

全社レベルでの実施状況のフォローを徹底して行った結果,ほぼ全ての物件に対して実施されるようになった.しかし,後述のように１年後にはロス金額の合計が思ったほど減少しないなど,必ずしもその効果が経営上の利点として直接つながるものではなかった.

その大きな理由の一つは,ＤＲ－０では遅過ぎることである.ＤＲ－０の段階でリスクがあることが判明しても,その時点では既に顧客との契約がなされており,受注を取りやめることはできないからである.もちろん,ＤＲ－０による指摘によりプロジェクト計画の見直しがなされるなど,プロジェクト実施方法についての改善はなされるが,本質的なリスクそのものは避けることは出来ない.

ＤＲ－０では合格,条件付合格,不合格の判定がなされ,条件付合格あるいは不合格の場,指摘された問題を是正した後,再度ＤＲを受けることにな

る．問題点の是正がなされないかぎり開発の着手は出来ない．

３．３．２　ＤＲ－Ｘの実施

ＤＲ－０ではリスクを避けるには遅過ぎるとの認識から，次の段階としてＤＲ－Ｘ（リスクを考慮し，受注してよいかどうかを判定するための設計レビュー）の全社的な実施を始めた．一部の部門では，部門規定として既に実施していたが，全社的な制度として取り上げることとなったわけである．

ＤＲ－Ｘでは，Ｘ，Ａ，Ｂ，Ｃのリスクランクが付けられ，最も危険性の高いＸと審査判定された物件は原則として受注しないことになる．毎半期，ＤＲ－Ｘの決定で始めから受注を断念するものが数件あり，また顧客と再度契約条件をすり合わせても，折り合いがつかず断念せざるをえないものが数件出ている．もちろん，リスクが高くても戦略的に受注するものもあるが，ＤＲ－Ｘを行うことにより，営業部門と技術部門のコミュニケーションが今までより良くなり，またリスクのあるものについてあらかじめ経営陣に報告できるなどのメリットが出るようになった．それまでは，よほど大規模物件でなければ，受注時に経営陣の判断を求めることはなく，全て部門長の判断に委ねられていた．ロスが出た場合，その結果が経営陣に報告されるだけであった．

３．３．３　ＤＲ－Ｆの実施

ＤＲ－０およびＤＲ－Ｘの徹底実施が全社レベルで可能となったので，次の段階として，ＤＲ－Ｆ（完成したソフトウェアを出荷しても良いかどうかを判定する設計レビュー）の徹底実施をすることになった．それまで，設計レビューの対象は主として，システム・ソフトウェア開発であったが，この時点から全社の全てのビジネスに適用することとなった．もちろん，システム・ソフトウェア開発用の制度をそのままシステム運用や用品販売など，性質の異なる事業に適用できないので，各部門の特性に合ったようにカスタマイズを行っている．

ＤＲ－Ｆで不合格となった場合は，原則として出荷出来ない．顧客の了解をとり，営業部門や上級管理者の判断で暫定出荷をする場合があるが，会社としてリスクを予め明確に顧客に知らせることができるようになったのがメリットである．出荷後にトラブルが出た場合でも，それまでたびたびあった，顧客との感情的な対立関係が少なくなっている．

３．４　設計レビュー徹底実施の効果

設計レビュー制度を実施するとともに，定期的なロスジョブ分析を行い，その効果を測定した．３年間の分析によると，ロスジョブによる損失が明らかに減少しており，設計レビューの徹底による効果が経営レベルで現れていることが確認できた（図３．２参照）．

（ｉ）２年間のロスジョブの増加

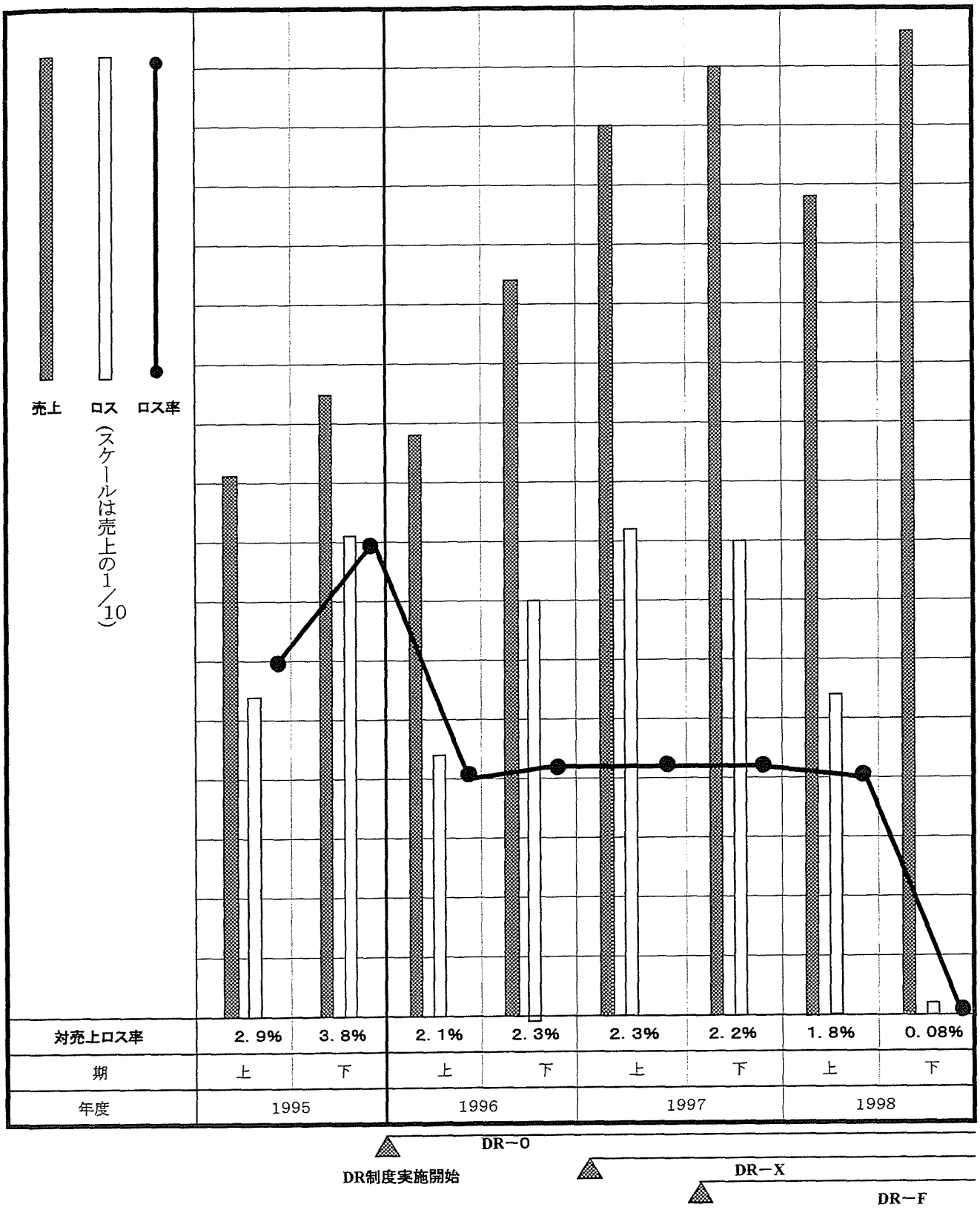


図3.2　設計レビュー制度実施後の発生ロスの推移

この設計レビュー制度の徹底実施を始めてから２年間くらいはロスジョブによる損失の金額が増加した．その大きな理由は，この制度を実施する前から開発が始まっていた大型のプロジェクトにロスジョブが発生したことである．すなわち，ＤＲ－０を実施していない物件も，終了時点でその半期の数字として計上される事になっている．大型物件によるロスは金額も大きくなり，他の多くの小さいプロジェクトでわずかづつ利益を積み上げても，大きなプロジェクトにロスジョブが起こると，それらの効果を根底からひっくり返してしまう．ＤＲ－０を徹底しても，ロスジョブが減らず，経営的には役に立っていないとの批判も出た．設計レビューを行っていない物件を除いた分については効果が出ているのだが，半期ごと経営の結果には直接表れてこないからである．

（ii）売上総額に対するロスジョブ比率は確実に減少

ソフトウェア開発だけでなく設計レビュー制度の対象業務を拡大するにつれ，ロス金額も増加している．しかし，設計レビュー制度が実施される前の総売上に対するロスジョブの比率は３％程度あったが，実施後は２．２％程度に減少している．この時点になると，全ての物件についてＤＲ－０が完全に実施されており，さらにほぼ全ての物件にＤＲ－Ｘも実施していたので，積み残し物件のロスジョブがほとんど無く，設計レビューの実施による効果が経営上の数字としてもはっきりと現れてきたのである．

（iii）３年後にはロス金額が１／１０以下に激減

ＤＲ－０の実施を始めて３年後には，ロスジョブの総額が，当初に比べて１０分の１以下に激減した．諸般の事情によりシステム・ソフトウェア関係の売上が減少したにもかかわらず，全社の利益総額が増加した．

もちろん，すべてが設計レビュー実施による効果とは言えないのも確かである．近年，ダウンサイジングの進行する中，大型物件が減少の傾向にあり，特にこの時期の大型物件が少なかったことから大きなロスジョブが生まれなかった．また，統計的なばらつきから，この年たまたま幾つかの要因が都合の良い方に変動したこともある．しかし，設計レビューの徹底による効果が最大の要因であることは明白であり，設計レビューの徹底による経営への貢献がはっきりしている．

（ⅳ）設計レビュー制度の実施による透明性の増大

設計レビュー制度を徹底実施する前では，プロジェクト完了時になってはじめて，その結果が経営陣に上がるだけで，プロジェクト開始時点あるいは開発過程で，どれだけリスクのあるジョブがあるかは報告されていなかった．設計レビュー制度が完全に実施されて以来，ＤＲ－Ｘ，ＤＲ－０，およびＤＲ－Ｆの報告に加えて，各部門内で行われている開発段階の設計レビュー（ＤＲ－１からＤＲ－６まで）の結果が集計され，報告されるようになった．この結果，危険性のあるプロジェクトの状況が定期的に把握できるようになり，経営レベルあるいは管理レベルでの素早い対応が取れるようになっている．

経営陣への報告だけでなく，技術部門と営業部門が情報の共有ができるよ

うになり,その結果として,顧客への素早い状況報告が可能になったことも,顧客満足度の観点からみて大きな効果である.

３．５　設計レビュー制度についての考察

設計レビューの徹底実施を始めて３年後には,その効果ははっきりと出てきた.この制度をはじめる３年前,全ての設計レビューを徹底して行うなどと言うと,多くの部門から,拒否反応があったことは確かである.しかし,ＤＲ－０に始まり,ＤＲ－Ｘ,ＤＲ－Ｆを段階的に,しかし徹底して実施してきた経緯があり,また,その効果がはっきりと実感でき,会社全体の利益にまで貢献していることが明白になった.

設計レビュー制度の実施と並行して,ＩＳＯ９０００認証取得を視野に入れた社内品質システム・品質規定の整備を行ってきた.全社システムの整備とともに各部門へのカスタマイズも完了し,設計レビュー以外の項目も順次実施してきた.

本章で記述したのは,決して新しい管理手法ではなく,実施して当たり前のことである.また,顧客満足度の観点から見ても,まだまだ,顧客にあまり迷惑をかけなくなったレベルに到達したくらいである.しかし,ここで述べた施策を実施することで確実な効果を上げることができたので,本設計レビュー制度を着実に一歩づつ前進させ,真に顧客の満足を得られる状態に到達できると考えている.

また，経営・技術・営業の間のコミュニケーションが良くなると共に，設計レビュー制度を通じて顧客との前向きで具体的･建設的な対話が出来るようになってきたのも大きなメリットである．ＣＭＭで目指している組織成熟度も徐々に，しかし着実に向上してきていると考えられる．

因みに，２００１年度に全社のすべての事業に対してＩＳＯ９０００の認証取得に挑戦し，６月に取得を果たしている．

第４章　現状の開発手法と管理手法の効果と問題点

１．２で述べたように，ソフトウェアはコンピュータという人工物に依存しており，

・ 自然法則に基づく絶対的な理論基盤がない

・ 論理的な複雑さが他の工業製品に比べて格段に高い

などの性質がある．このため，ソフトウェア開発には１．３で述べた本質的な難しさがある．

ソフトウェア開発の問題を解決するためにソフトウェア工学という研究分野が生まれたが，研究対象のほぼ全てが，人間の創造物あるいは人間の活動であることに特徴がある．土木工学や電気工学といった既存の工学では，物質や材料といった自然物が研究対象であり，数学や物理学，化学の助けを借り，定式化できる部分が多い．ソフトウェア工学の場合，ソフトウェア開発に関係するあらゆる事項が研究対象になるが，それらが人間の創造物，あるいはその活動が対象であるが故に，定式化・定量化できる部分が少ない．

定式化・定量化するには，管理工学と同様，多くの事例を集め統計的に扱い，プロダクトやプロセス，あるいは使用する開発手法なり管理手法の妥当性を検証することになる．また，人間の考え方や行動を対象とするため，社会学や経済学，経営学，心理学といった社会科学が扱うような対象も扱うことになる．ソフトウェア工学は，ソフトウェア開発に関わるあらゆる事柄を対象にする，いわば，研究対象の百貨店とも言うべきものである．

本章では，まず今までのソフトウェア開発，そしてソフトウェア工学の流

れを概観し，その本質的な性質について議論し，それを基本にして第２章および第３章で述べた事例について考察する．

４．１ ソフトウェア開発をめぐる３つの時間

約５０年前に誕生して以来，コンピュータの進歩は著しい．ソフトウェア開発を含むコンピュータを取り巻く世界には同時に次の３つの時間が流れていると言える．

（１）非常に早く流れる時間．
（２）着実に流れる時間．
（３）進歩が遅く，退歩すら感じられる時間．

（ｉ）非常に早く流れる時間

非常に早く流れる時間は，コンピュータ関連のハードウェアの進歩，すなわち，コンピュータ本体の進歩，あるいはその周辺機器や周辺技術の進歩を表す．この進歩の多くは半導体技術の急速な進歩に支えられているが，半導体技術の進歩は，シミュレーションなどを通じコンピュータに支えられ，補完的な関係にある．

コンピュータの速度は，当初のミリ秒（ms）のオーダから今ではナノ秒（ns）のオーダへ進歩しているし，主記憶容量においても，当初の高々数１００バイトから今日ではパーソナルコンピュータ（パソコン）でも数１０

０Mバイトの容量を持つようになってきており，１００万倍を超える進歩をしている．周辺機器でも同じような進歩を示しており，いまだにこの進歩のスピードは止まっていない．人類の歴史が始まってこれほど進歩の早い人工物はない．

ハードウェアの進歩につれて，ソフトウェアの開発量も増加の一歩をたどっている．Microsoft 社のＯＳ（Operating System）は５０００万ステップを越すといわれているが，それが今日では，パソコンの普及もあり，１億コピー以上出回わっている．家庭電器を始めとし，今日ほとんどの工業製品にコンピュータが組み込まれており，そこにソフトウェアが入っているので，実際に稼動しているソフトウェアの絶対量を測定するなら，その絶対量や増加のスピードは想像を絶するものとなろう．

（ii）着実に進歩し流れる時間

この５０年，ソフトウェア開発の基盤（インフラストラクチャ）も初期の段階に比べ，徐々にではあるが着実に進歩を遂げている．紙テープベースから始まったソフトウェア開発が，カードベースになり，エディタを通して磁気ディスクに自動的に格納されるようになった．

かつて，コンピュータを象徴するものと言えば計算機室に汎用コンピュータと磁気テープ装置が並んでいる風景であったが，コンピュータ本体や周辺装置の小型化・高性能化が進み，ダウンサイジングの流れから，汎用コンピュータはサーバとしての役を受け持つだけになり，主役はEWS(Engineering Work Station)へ，そしてパソコンへと移ってきている．

ソフトウェア開発の思想，手法，ツールなども着実に進展している．コンピュータ言語で言えば，機械語，アセンブラから始まり，ＦＯＲＴＲＡＮやＣＯＢＯＬ等の高級言語への移行により，ソフトウェア開発の生産性は１０倍位向上したといわれている．コンピュータ言語は着実に進化を遂げており，PL／１，Ada，Pascal やＣ言語の開発に進み，今日では，オブジェクト指向を取り入れた C++や Java が主流言語となりつつある．

開発プロセスの考え方も，上流工程の重要性の認識から，工程を明確に分割するウォーターフォールモデルの考え方が定着した．逆に，過度に工程を分割すると弊害が起こり，形骸化した建前だけになると言う，ウォーターフォールモデルへの反省から，スパイラルモデルやプロトタイピングモデルの提案がなされている[7].

大きなソフトウェアシステムを開発するのに，サブシステムに分割し，さらにモジュールまで分割する階層的分割法，すなわち「分割統治」を行おうとする構造化分析・構造化設計という考え方が定着した．

インタフェース条件だけを明確にし内部の詳細を見せないようにする情報隠蔽の考え方から，抽象データ型に発展し[30]，その延長として，継承の概念も取り入れたオブジェクト指向技術が普及の段階にある．

当初，エディタやデバッガといった，プログラミングレベルのツールが開発支援の中心であったが，分析や設計といった上流工程の支援もできるようになり，CASE（Computer Aided Software Engineering）ツールによる開発支援が中心になりつつある．

ソフトウェア開発の目標をソフトウェアをなるべく開発しないこととす

るなら，その目標に向けて着実な研究開発がなされてきた．その一つの重要な課題として，ソフトウェアの部品化・再利用あるいはコンポーネントウェアがあり，この課題は常に中心課題であり，着実に進展している．しかし，ソフトウェアの自動作成という究極の夢がある限り，いつまでたっても完全には解決されることのないテーマとして残るものと思われる．

（iii）進歩が遅く，退歩すら感じられる時間

コンピュータの急速な進歩，コンピュータ関連のインフラストラクチャの着実な進歩に比較し，ほとんど進歩していないのではないかと思われる時間がある．それは個人あるいは組織，すなわち人間に関わる時間である．

人間あるいは組織は保守的で，例えばそれまでに獲得した手法を捨ててまで新しい手法を習得しようとしない．また，例え新しい手法を習得しても，それを維持するのに努力が必要で，それを行っても目に見える大きな効果がなければ，あるいは強い強制力が働かなければ，楽なほう，すなわち以前使っていた手法に戻ってしまう．

ミニコンピュータ用構造化アセンブラを開発した時には（[15]-[18]），それまでアセンブラしか使えなかったので，アセンブラでソフトウェア開発していたのに比べて，生産性向上や品質向上にはるかに効果があった．その効果を実感できたから，その後１０年間，C 言語に移行するまで，鉄鋼プラントや発電プラントといった，ミニコンピュータによる制御分野の全てのソフトウェア開発に使われた．これは効果がはっきりと認識されたからである．

しかし，HIPO[19]や SADT[20]といった新しい設計手法や要求分析手法

を普及させるための教育・普及活動を行った時には，一時的には定着したように見えた[21]．しかし，それらの手法を使うと手間がかかり文書量が増えるなどの理由で，徐々に敬遠され，それぞれの部門や技術者がもともと使っていた，我流の手軽な手法による分析や設計に戻ってしまい，１０年後にはほとんど誰も使わなくなってしまっていた．社内で標準的な手法を使うことの効果は，会社全体の標準化という面では大きいが，それらを使用する部門や個人には間接的であり，直接的な効果を実感できない．そういう場合には，規定などでいくら強制しても一時的なことになり，元に戻ってしまう．

第３章で述べた設計レビュー制度の事例でも，それより１０年以上前，社内の制度として設計レビューを行うことが定められていたのが，時間が経つに従い，いつのまにか忘れられ，殆んど行われていなかった．設計レビューを実施することの必要性や効果を認識していても，どうしても楽なほうに行ってしまう．社内運動として新たに強制力を持たせて，段階的に適用し，その過程で実際に効果があることが，組織的にも，技術者個人にも実感できたので徹底できたのである．

分析や設計を標準化された手法で行うこと，レビューを工程間のつなぎとして行うことの重要性は誰しも認識しており，何も新しいことでもない．しかし，同じ状態を保つにも努力が必要で，その努力を維持できない場合どうしても，元の楽なほうに戻ってしまう．

組織や人間のもつ本質に関わることについては，時間は前に進むより，退歩すらしている面がある．

1975 年，Brooks がその著書「ソフトウェア開発の神話」でソフトウェア

開発の現状の問題点を指摘した[24]．20 年後その続編「人月の神話」を出版するにあたり再度調査したが，ソフトウェア開発に関する問題点は殆んど変わっておらず，内容は殆んど変える必要はなかった[25]．このことは，いかに，ソフトウェア開発に関わる人間的な面での進歩が遅いかを示している．

４．２　現状の手法の問題点

現在，多くのソフトウェア開発の現場で実際に行われている手法，あるいは提案されている手法には，次のような問題がある．その多くは，４．１節（iii）で述べた，組織や人間が楽なほうに走るとの認識から，それをできる限り避けるように，管理的側面を徐々に強くさせている傾向がある．

（１）開発手法や管理手法がいまだに未熟である．

（２）普遍性を求め過ぎる傾向が強い．

（３）巨大ソフトウェアシステムの新規開発を意図した手法が多い．

（４）過大なドキュメント作成を求める傾向がある．

（５）開発者と発注者が協力的関係でなく，敵対関係にある．

（６）技術者の独創性を生かしていない．

（i）開発手法や管理手法がいまだに未熟である

ソフトウェア開発が始まって半世紀の歴史があり，多くの手法が提案されてきたが，いまだに，固定・定着した手法はないといえる．例えば，設計記

述法一つ取り上げても，フローチャートが生まれてから後，NS チャート，TFF[22]のような構造化チャートなど，さまざまな記述法が提案され使われてきたが，いまだ新しい記述法が生まれる可能性がある．オブジェクト指向設計の記述法にしても，提唱者ごとの多くの記述法があったが，最近になり主要な提唱者が一つにまとまり，UML[23]として統一された．しかし，まだまだ進展する可能性がある．

このソフトウェアに関する手法の未熟さは，主としてコンピュータの進歩があまりに速く，ソフトウェア技術が追随できなかったことによる．逆にいえば，ソフトウェア技術はいつまでも未熟で，常に発展する可能性があるとも言える．

（ii）普遍性を求めすぎる傾向がある

第1章でも議論したように，ソフトウェアはどんな分野にも応用される可能性があり，そのため提案されてきた手法は，どの応用分野にも適用されるような普遍性を求める傾向が強かった．どの分野にも適用可能な普遍的・標準的な手法はもちろん追求すべきであるが，応用分野ごと，あるいは開発規模に適した手法もあってしかるべきである．しかし，今後ソフトウェア開発は分野ごとに専門化する可能性もあり，ゲームプログラムの開発や携帯電話のソフトウェアといった分野ごとに適した開発手法が出てくる可能性がある．

（iii）巨大ソフトウェアシステムの新規開発を意識した手法が多い

多人数で行うソフトウェア開発を,いかに体系的に行うかがソフトウェア工学誕生のきっかけであったので,このことは当然の帰結かもしれない.しかし,小規模ソフトウェア開発の場合にも大規模開発と同じような開発手法が求められ,いわば出刃包丁で小魚を料理するようなことを行っている.

また,ほとんどの開発プロセスモデルは,新規ソフトウェアを開発する場合だけを考えている.しかし,現実には既に開発され実際に運用されているソフトウェアに機能を追加したり,改良を加えるというソフトウェア開発が多い.既に,稼動しているシステムをいかに改良し拡張するかについての具体的な手法は今のところない.

(iv) 過大なドキュメントの作成を求める

ソフトウェア開発を組織的に行うには,開発関係者間の情報共有のため,あるいは開発完了後の保守のためにドキュメントの作成は必要である.しかし,開発途中のワーキングドキュメントと,保守のために残すべきドキュメントの区別を明確にしていない場合が多い.保守のためにと称し,大量のドキュメントを残すが,ほとんど使われもせず,結局保守に役立つのは,プログラムリストであることが多い.必要十分なドキュメントとは何か,まだ明確にされていない.

また,ISO9000 認定の条件では,各開発プロセスが本当に品質システムで規定された作業を,確実に行ったかの証拠まで残すことが求められている.良いプロダクト(製品)を作るためには,整然としたプロセス(開発過程)が前提であるのは当然であるが,第三者への証拠作りのために多くの精力を

費やさざるを得なくなり，目的と手段を取り違えているような現実がある．

（v）開発者と発注者が協力的関係でなく，敵対的関係にある

契約に基づいて，ソフトウェアを開発する場合，そのソフトウェアによって，どんな効果を上げたいかを言えるのは発注者である．受注者（開発者）は，契約された機能・性能を持つソフトウェアを開発するに足りる専門知識・技術を持つが，そのソフトウェアでどんな効果を望むかについては，発注者から知らされる．しかし，契約時点で発注者が全ての仕様を明確にできることは現実的に望めない．

今日でも，ほとんどの開発で使われている，ライフサイクルモデルあるいはその派生モデルに従う場合，開発が終了した時点で初めて発注側がソフトウェアの出来栄えを確認できるので，問題が起こる場合が多い．これらの問題を避けるため，スパイラルモデルやプロトタイピングモデルが提唱されているが，手法としていまだに成熟していない．

発注者と受注者が，契約を通じて敵対的関係にあるといえる．本当に役に立つソフトウェアを開発するには，発注者と受注者が協力して開発するという開発モデルが必要である．

（vi）技術者の創造性を生かしていない

ソフトウェア開発は全て人間が行う．ソフトウェア工学で対象とするソフトウェア開発は複数の技術者の共同作業で行うので，必ず共同作業を円滑に行うための調整作業，すなわち管理が必要になる．管理の目的は，チームの

共同作業を効率化し，与えられたリソースでQCD(Quality, Cost, Delivery)面で最大の利得を得ることである．しかし，管理行動として必ずしもチームの調整役としてだけ動くのと違い，強制力を持つことになる．

作業の仕方を細かく規則で縛られ，管理者から見張られ，強制されるような環境下で，ソフトウェア開発といった創造性を問われる仕事をすることは難しい．

工程ごとの分業で開発する場合，個々の技術者には，開発すべきソフトウェアの全体像が見えず，設計仕様に基づき部分的なプログラミングをするというだけでは創造する喜びがない．最近はハードウェア製品の製造工場の現場でも，ベルトコンベアに並んで決まったネジを締めるだけといった部分的作業でなく，全製品の組立を一人の技能者が行うワークショップ制が主流になり，そのほうがかえって効率が良いと言われている．また，作業者が自分で製品を作り上げたという誇りと喜びを持つことができ，製品に責任を持てるし，士気も上がる．

現状のソフトウェア開発の手法や環境は必ずしも，創造的な業務を遂行するのに適しているとは言えない．

本節で述べた問題点を，全て解決できるような開発手法や管理手法があるとは考えていない．しかし，どんな手法でも，これらの問題点を十分認識し，その手法でどの部分が解決され，どういう種類のソフトウェア開発に適しているかを明確にする必要がある．

本章の後半では，４．１節および４．２節での議論に照らして，第２章お

よび第３章で記述した事例について考察する．

４．３ 統合ＣＡＳＥ環境の効果とその問題点

第２章で記述した IMAP を研究開発した同時期に，国家レベルでシグマシステムの開発が行われた[26]．シグマシステムも，ＩＭＡＰと同じように，ネットワークで結合された EWS(Engineering Work Station) 上に，統合 CASE(Computer Aided Software Engineering)環境を実現し，ソフトウェア開発の工業化を目指したものである．

同様に，コンピュータメーカのほとんどで，同じようなシステム開発がなされており，統合 CASE 環境がソフトウェア開発の究極の解であるかのような期待があった．欧州でも，Esprit 計画として，イギリス，フランス，ドイツ等の国家連合の研究開発がなされていた．

幾つかの技術的な成果，あるいはソフトウェア開発の重要性を知らしめ，組織的な取組みをさせるきっかけとなるという波及効果はあったものの，残念ながら，ほとんど全ての統合ＣＡＳＥ環境は当初の目的を達せずに終わっている．

４．３．１ 統合ＣＡＳＥ環境 IMAP の問題点

以降では，IMAP による統合 CASE 環境が何故成功しなかったかについては，以下の項目について分析する．

（1）ソフトウェアツール開発の技術が未熟であった.

（2）ハードウェアの進歩を過大視し過ぎていた.

（3）手法の普及をツールに頼り過ぎていた.

（4）ソフトウェア開発は工業化になじまない.

（i）ソフトウェアツール開発の技術が未熟であった

IMAP 開発を計画していた当時，エディタやデバッガ，コンパイラ等のプログラミングレベルの支援ツール（下流 CASE）に加えて，要求分析や設計を支援する上流工程の CASE の開発が行われ，実際にソフトウェア開発にも活用されていた[21]. 上流 CASE を下流 CASE に結合すれば，設計からプログラミングまでを連続的な支援が可能になるという期待があった.また，リポジトリ(ソフトウェア開発で作成される各種のドキュメントや管理情報を統合的に格納・管理するデータベース）の概念が一般化し，開発ドキュメントを管理情報と同時に格納すれば,開発と管理を統合化できるという期待もあった.

しかし，上流 CASE で扱う情報は，ソフトウェア開発の方式設計レベルの情報で，抽象度の高いものである. その情報から，プログラミングにつなぐには，上流 CASE の結果を参照しながら，下流 CASE で設計し直す必要がある. もちろん，上流 CASE の結果を下流 CASE へ自動変換ができればよいのであり，その研究開発も行われていたが，情報の抽象度の差異が大きく，自動変換できるまでに至っていない. せいぜい，上流 CASE の結果か

らプログラムの枠組み（スケルトン）を出せるくらいで，内部のプログラムロジックは技術者が改めて作成し入力する必要がある．

リポジトリについても，個人レベルあるいは小さなプロジェクトレベルで適用できるレベルのものは開発されたが，部門全体あるいは工場全体の成果物を格納管理できる規模のリポジトリは，ドキュメント量が莫大になり，データベースの技術がそれを開発するまでには成熟していなかったため，結局開発できなかった．

上流 CASE と下流 CASE 間の自動変換，リポジトリの研究開発は現在では行われていない．

（ii）ハードウェアの進歩を過大視し過ぎていた

当時は EWS が実用化され始めた時で，汎用機の持つ機能を TSS（Time Sharing System）端末を通して分割して使うのではなく，技術者個人が汎用機に匹敵するすべての機能が使えるとの期待感が強かった．また，マルチウィンドウ機能を活用し，多くのドキュメントを画面に出して，画面だけで設計やプログラミングができるようになるとの期待があった．

しかし，画面の解像度の点からもマルチウィンドウで多くのドキュメントを出すとかえって見難くなる．その上，処理量があまりに多くなりスピードが追いつかなくなるなどの問題があった．その後のダウンサイジングにつながる，EWS の登場で，過大な期待を描いていたのである．

（iii）手法の普及をツールに頼り過ぎていた

要求定義や設計手法の普及を目指し，社内教育などを行っており，一部使われてはいたが，手作業で設計文書を書くことについての抵抗があり，全ての業務に完全に適用されていたわけではなかった．上流 CASE を導入することで，手法の普及が促進するとの期待があった．

しかし，ソフトウェアツールに限らず，ツールは人間が手で行っていることを支援し，効率を上げるものである．ツールの支援があれば，それまで使っていない手法が普及するようになるとの期待は幻想に過ぎなかった．

例えツールがあっても，その手法を自由に使いこなす訓練を受け，使いこなすだけの能力がなければ，何も役にも立たない．ツールがなければ手間がかかりすぎ，とても使えないような手法についても，ツールの支援があれば実際のソフトウェア開発の現場で使えるようになるとの期待があった．しかし，その場合でも，手法の意味を理解し，その手法を用いて分析なり設計ができるように習熟していることが前提である．

（iv）ソフトウェア開発は工業化になじまない

第 1 章でも議論したが，ソフトウェア開発は全て，設計段階の業務であり，製造段階で見られるような，工程別の分業には向いていない．もちろん，ネットワークの専門家，データベースの専門家，方式設計の専門家，プロジェクト管理の専門家などが専門性を生かし協力して業務を遂行することはあっても，工程別の分業でなくそれぞれの専門性を生かした協業であり，工程別の分業という開発方式は，技術者の創造性を生かせない．

IMAP はそれまでのソフトウェア工学で確立した技術を，EWS という当時ではエポックメーキングな新技術のもとで集大成し，統合 CASE 環境としてまとめたものであったが，一部のツールが使われただけであった．

４．３．２　統合ＣＡＳＥ環境ＩＭＡＰ開発の波及効果

統合ＣＡＳＥ環境としてのＩＭＡＰは，所期の目的のようには，活用されなかったが，ＩＭＡＰの研究開発をベースに，ソフトウェア開発の問題に対して全社的な取組みを行おうという気運が高まり，以下のような波及効果をもたらしている．

（１）全社品質管理体系の整備と部門へのカスタマイズと実施．
（２）オブジェクト指向技術研究開発グループの組織化と普及活動．
（３）ＣＡＳＥ活用委員会の設置と普及活動．
（４）ソフトウェア品質向上の推進．

４．４　組織的な開発管理手法の効果とその問題点

第３章で述べた設計レビュー制度は，社内の品質問題がきっかけとなった．品質問題のためロスジョブ（第３章参照）が多く発生していた．特に大型プロジェクトではロス金額が大きく，ロス合計が会社の総利益に匹敵するまで

大きくなっていた.

品質問題の解決のため社内の品質システムを全面的に見直し，ＩＳＯ９０００認証取得を目指すことが検討されたが,その頃既に取得した会社の状況を調べると，

（１）認証取得に足りる品質システムの再構築に多くの労力が取られる

（２）そのシステムを社員に徹底するには，多くの教育が必要

（３）品質システムに準拠している証拠として，膨大なドキュメント作成が必要

（４）取得した後，定期的に審査があり，維持するのが大変

（５）社員は業務命令として従うが，積極的ではない

等の問題があることが分かった．直接 ISO９０００認定取得に挑戦するとソフトウェア開発の現場が混乱し,効果が出るより弊害が出る恐れがあった.そこで，ＩＳＯ９０００取得を視野に入れ，段階的に，かつ各過程で具体的な効果を得られるようにとの決定をした.ロスジョブの分析の結果を踏まえ，ＤＲ－０（開発直前の設計レビュー）から段階的に徹底して行うことになったのである.

４．４．１　設計レビュー制度の効果

第３章でも述べたが，ＤＲ－０から段階的に実施したことにより，定期的

な設計レビューを行う風土が,組織や個人に定着したことが最大の効果である．風土が定着していたので，以降ＤＲ－Ｘ（引合い時，受注前の設計レビュー），ＤＲ－Ｆ（出荷前，開発終了時の設計レビュー）を段階的に増やしていっても，抵抗感がなくスムーズに実施できた．具体的な効果が実感できたことが，この制度が定着した最大の理由である．組織や技術者個人にメリットがなければ，定着はおぼつかなかったと思われる．

開発中の設計レビュー（ＤＲ－１～６）は，各部門に任されており，その実施状況の報告を受けるだけであったが,各部門ともプロジェクトの規模や重要度に従い，実施するＤＲの種別を決めて実施していた．

ＤＲ－Ｘはもともと，一つの部門が自主的に実施していたもので，それまでの制度にはなかったが，経営的な観点から，全社規定に取り入れられたものである．

設計レビュー制度を継続実施することにより CMM（Capability Maturity Model）でいう組織成熟度が確実に上がった．設計レビュー制度の実施と並行してＩＳＯ９００取得に向け，社内品質システムの整備を行っており２００１年，全社的な取得の決定がなされたが，ほとんど混乱もなく，６月に認定取得を果たしている．設計レビュー制度の実施により組織成熟度が確実に上がっていたことが，認証取得ができた最大の要因と考えられる．

４．４．２　問題点とその解決策

そもそも，なぜ品質問題が起こったかといえば，２つの原因がある．第１の原因は，それまでにも品質規定があったにもかかわらず，それが日々の業務を行っているうちに，いつのまにか守られなくなっていた，いわば，組織成熟度が落ちていたことである．第２の原因は，技術者個人の技術的な向上心が，いつのまにか薄れてきたことである．

第１の原因に対しては，設計レビュー制度の徹底実施により向上してきたと考えられる．第２の原因に対処すべく，技術者の自主的な技術力向上を図る２つの施策を実施している．一方ははスペシャリスト制度であり，他方は公的資格取得策である．

スペシャリスト制度は，情報処理技術者上級資格あるいは相当の資格を持ち，かつ部門から推薦を受けた技術者を，社内審査委員会での面接試験で認定するものである．すなわち，スペシャリストは，客観的に認められた知識や技術力（資格）を持ち，実際にその力を仕事に生かせている（部門推薦）者で，かつ他の技術者の模範となる人物（面接試験）を認定する．スペシャリストに認定された者は，社内に広報するとともに，年間定額の技術活動に対する資金が与えられる．ゆくゆくは昇格制度に連動し，管理職への昇進の条件となる．資格の取得が必ずしも技術力の証明にはならないので，具体的に仕事に生かせていることが条件である．

資格取得についてはそれまでも技術力向上の一環として推進していたが，取得者が年々減少する傾向にあった．社内制度として資格取得を改めて推進するため，そのキックオフを兼ね，全社一斉模擬試験を行った．受験対象は部長以下全員とし，事業部長以上も受験できるとした．結果としては事業部

長などの対象者以外も積極的に受験をし，当初予定していた受験者数を上回っていた．設計レビュー制度の定着などで，社員の意識が変ってきたものと思われる．

第５章　設計レビューを重視したソフトウェア開発方式の考察

現状のソフトウェア開発手法および管理手法には，４．２節で述べたような以下の問題点がある．

（１）開発手法や管理手法がいまだに未熟である．

（２）普遍性を求め過ぎる傾向がある．

（３）巨大ソフトウェアシステムの新規開発を意図した手法が多い．

（４）過大なドキュメント作成を求める傾向がある．

（５）開発者と発注者が協力的関係でなく，敵対的関係にある．

（６）技術者の独創性を生かしていない．

本章ではこれらの問題を解決するためのソフトウェア開発手法および管理手法について考察する．

今までの手法で特に問題になるのは，ソフトウェア開発の原点を直視していなかったことである．大規模開発と言う側面を強調し過ぎるあまり，開発全体の構造や組織運営に注力し，ソフトウェアとは何かの原点を見失ってきたのである．

ソフトウェアと言うのはプログラム一行一行にいたるまで，技術者の創造性から生まれる設計そのものであり，それ故プログラムリストが究極のドキ

ュメントである．今まで，プログラマ（設計者）がその創造性のある仕事をする環境を整備することより，他の工業との類推から，管理者，分析者，設計者，プログラマ（コーダ），テスタといった分業化を進めてきた．おのおののプログラマ（設計者）は，全体の構造が見えない歯車として働くような仕組みを作ってきたと言える．

いわば全体主義国家の行政・軍事・警察機構は緻密に作られているが，国民一人一人が見えないようなものである．分業体制の中でも管理者，分析者や設計者が上位におり，プログラマ（コーダ）が軽視される風潮があった．本来主役であるはずのプログラマの創造性を軽視した本末転倒のことが行われてきた．

本章では，プログラマ（設計者）が開発の主役であるような開発モデルの幾つかを検証する．それらのモデルの利点を生かし，かつ，改善するような新しい開発手法を提案し，それを支える開発管理手法を提案する．

管理者はあくまで開発の主役であるプログラマを支える縁の下の力持ちに徹し，プログラマが働きやすい環境を整備する役割を担うものとして位置付ける．野球やサッカーの試合において主役はあくまで選手であって，監督やコーチに例え命令する権限があっても試合が始まれば単なる黒子であるのと同様である．テレビタレントが主役であり，彼らを管理する興行会社はあくまで黒子である．プログラマこそがソフトウェア開発の主役であるべきである．

５．１　設計者を主役とする既存の開発手法

今まで設計者が主役となるような開発手法および管理手法が，幾つか提案され実施されている．ここでは，その代表的な３例を挙げる．

（１）チーフプログラマ制

（２）Microsoft 社のフィーチャチーム制

（３）エキストリームプログラミング

（ｉ）チーフプログラマ制

チーフプログラマ制は Mills の提案し実践した手法で[27]，外科手術チームとの類推から考え出された．外科手術は，数人のチームで行われるが，実際手術を行うのは執刀医一人だけである．他のスタッフはすべて心電図をチェックしたり，メスやピンセットを渡したり，執刀医の汗を拭いたりして，執刀医の手術をサポートするだけである．

チーフプログラマ制のプログラミングチームもチーフプログラマと数人のスタッフで構成されるが，設計やプログラミングの全権限と責任を持つのはチーフプログラマだけである．他のスタッフは，ライブラリアン，バックアッププログラマ，テスタなどとして支援作業を行う．技術的な情報はすべてチーフプログラマに集まり，すべての決定はチーフプログラマが行い，開発が進められる．この手法の成否は，チーフプログラマとして優秀な技術者が得られるか否かにかかっている．

かつて構造化アセンブラの開発[15]で，ほぼ新人ばかりの４人とでチームを組み，チーフプログラマ制に類似の手法をとったことがある．全体の方式設計の後，各人の分担範囲や設計方法，記述方法などを決めた後，それぞれに設計・プログラミング・テストは任せた．自からの担当部分はもちろんのこと，全員の設計・プログラムにもすべて目を通し，不具合点の指摘やテストの方法を指示するとともに，ユーザマニュアルの作成も担当した．

この手法は，チーフプログラマに掛かる負担が非常に大きいので，開発規模がそれほど大きくなく短期間の開発，すなわち高々１０万ステップ，開発期間６ヶ月以内のソフトウェア開発に適用できると思われる．

（ii）Microsoft 社のフィーチャチーム制

Microsoft 社が Office パッケージの開発で用いている，個々のプログラマの創造性を生かすソフトウェア開発手法である[6]．Office パッケージには，Word，Excel，PowerPoint 等が含まれるが，ソフトウェア規模は既に巨大になっており，大勢の開発要員により定期的なバージョンアップで，機能の追加や改良が行われている．

この手法では，プログラマとテスタのペアを単位としてチームを組ませ，幾つかのプログラマとテスタのペアを集めてフィーチャチームとしている．フィーチャ（Feature）とは，例えばタスクバーの機能あるいはプルダウンメニューの機能のように，一連のまとまった働きをする機能のことであり，次に開発すべきフィーチャが決まると，その開発をフィーチャチームあるいはプログラマとテスタのペアに任せる．

各フィーチャチームには管理者がいるが，必ずしもプログラマではなく，開発すべきフィーチャの外部仕様を Visual Basic 等で記述し，どのフィーチャをどのペアに任せるかを決めるだけである．もちろん，大きなフィーチャの開発は複数のペアに分担させることもあるが，開発は原則としてペアが責任を持つ．

プログラマが与えられたフィーチャの実現方法を考え，設計やプログラミングをするのと並行して，相棒のテスタがテスト方法を考え，テストプログラムを作成し，出来上がったプログラムをテストする．プログラマとテスタは常に協力し，そのフィーチャを完成させることになる．

開発すべきフィーチャが完成し，単独で動くことが確認されると，テストラボ（テストの設備が整ったテスト部門）に持ち込まれ，Word なら Word の全体のプログラムに組み込まれ，機能が正しく動くか，システムダウンを起こさないか，などの負荷テストを夕方から朝にかけて一晩中コンピュータで行う．ペアは翌朝テスト結果を受け取り，次の開発を行うことになる．

この手法では，まず品質の保証された全体プログラムがあり，新しいフィーチャを組み込んで，一晩の負荷テストをしてきちんと動けば，新しい全体プログラムが完成しているという考え方をとっている．どのペアあるいはどのフィーチャチームがどんな順に新しいフィーチャを組み込もうが，全体のソフトウェアは常に完成の状態にあり，いつでも出荷ができる状態にある．機能が徐々に増加するあるいは改善されるだけである．あるフィーチャが予定していたバージョンアップの出荷に間に合わない場合でも，そのフィーチャは次のバージョンの開発に延期されるだけである．

大きなシステム開発では，フィーチャチームの数が増えることになる．全体の開発期間が長期間にわたるものでも，数週間ごとの期間（フェーズ）に区切り，各フィーチャに優先度をつけ，優先度に従いどのフェーズで開発するかを決め，どのフィーチャチームやペアに開発させるかを決める．

この手法では，プログラマとテスタのペアは，指定されたフィーチャの開発責任を負わされるが，開発の方法についてはすべて任され，技術者として全精力を注ぎ込める．

分割された工程だけを担当する手法では，自分が開発ソフトウェア全体ののどの部分を担当しているのかすら分からず，ただ与えられた作業を行っているだけであるのに比べると，開発に誇りと喜びを感じることができる．

この手法はパッケージ開発のように自社で仕様を決められる開発に向いた手法と言える．

（iii）エキストリームプログラミング

Ken Beck らによって提唱されている開発手法で（[28]，[29]参照），１２個のプラクティスと呼ばれる指針を与えている．

指針の一つに「ペアプログラミング」が提唱されており，プログラマ２人のペアで，小規模な開発（ここでは「機能」と呼ぶ）を短期間に開発する．開発された「機能」はシステム全体に結合される．

各「機能」をプログラムで実装する前にまず，その「機能」をテストするプログラムを開発する．プログラムを単純にするため，何度も書き換える（リ

ファクタリングする）ことを前提としているが，書き換えるたびに始めに作成したテストプログラムで正しさを確認することで品質の確保を行っている．

まず，動くプログラムを完成させる．そして重複しているなど，少しでも冗長な部分があれば重複を除き単純化する．但し，リファクタリングしたものが正しく動くことは最初に作成したテストプログラムで確認することになり，リファクタリングでプログラムの仕様が変ることは許されない．

５．２　既存の手法についての考察

５．１節で述べた手法で共通しているのは，技術者は開発すべき仕様が与えられると，その開発はすべてその技術者の裁量に任されることである．５．１節（ｉ）のチーフプログラマ制は，チーフプログラマだけに権限と責任があり，サポートスタッフは補助的な作業をするだけであるので，少し異なるが，（ii）フィチャーチーム制および（iii）エキストリームプログラミングでは，

（１）大規模なソフトウェア開発を，小さな機能の集まりに分割すること

（２）それぞれの小機能の開発を２人のプログラマのペアに任せ，短期間に開発すること

（３）小機能が完成するたびにシステム全体に組み込み，その時点ではシステムが完全に動くことを保証すること

（4） システム全体は徐々に成長し，最終的には最初に計画した機能全体を含むシステムになること

に共通点がある．(2) により，例え小機能であっても，その機能の開発を完全に任され，プログラマが本来持つ創造性を発揮できる．ウオーターフォールモデルで推奨されているような，他人が分析・設計したプログラムモジュールをただコーディング・デバッグするだけ，また他人が作ったテスト仕様に基づき，他人の作ったプログラムのテストをするだけといった退屈な職務をなくすことができ，プログラマが本来持っている創造性を発揮できるようになる．

しかし，これらの方式には幾つかの欠点あるいは限界がある．主なものとして次のようなものがある．

（1）システム全体を小機能の集合だけで実現できるソフトウェアには限界がある．システム全体の枠組み（アーキテクチャ）を予めきちんと決めておかなければならないものが多い．例えば，ボトムアップに小機能をまず動かすのに，簡単なファイルシステムを採用したが，全体の性能が出ず，データベースシステムを採用せざるを得なくなれば，システム全体の作り直しにもなりかねない．システム全体のアーキテクチャをきちんとしてから機能分割する必要があるものも多いが，これが明確でない．

（2）管理者の関わりが不明確である．多数の人間が同じ目的に向かって共同作業を行う時，必ず全体の調整を行う管理者が必要なのは明らかであ

るが，３．１節で述べた２つの手法では管理者がどう関与すべきかが明確ではない．

（３）契約に基づきソフトウェアを開発する場合，完成したソフトウェア全体の品質が，本当に発注側(顧客)の要求に合致するものか明確でない．小機能が完成するたびに全体システムに組み込まれテストされるので，システムが暴走したりせず確かに動くことの保証はある．しかし，ボトムアップに小機能を積み上げて本当に顧客が満足できるシステムとなり得るのかとの危惧は残る．

５．３　設計レビューを重視した開発方式の提案

５．２で参照した開発モデルは，今までの管理中心に向きがちであった開発モデルに比較してはるかに現実的で，かつプログラマの創造性を生かせる開発モデルである．チーフプログラマ制は，中小規模のソフトウェア開発にしか適用ができないが，フィーチャチーム制やエキストリームプログラミングは，大きなシステム開発にも適用できる．しかし

（１）システムのアーキテクチャをいつ考えるのか，

（２）管理者の関与をどうすべきなのか，

（３）出来上がったシステムの品質（顧客満足度）をどうとらえるのか

といった幾つかの疑問点がある．

ここでは，これらの問題を解決するため，図５．１に示すような開発・管理モデルを提案する．このモデルは，開発者としてのプログラマと管理者，

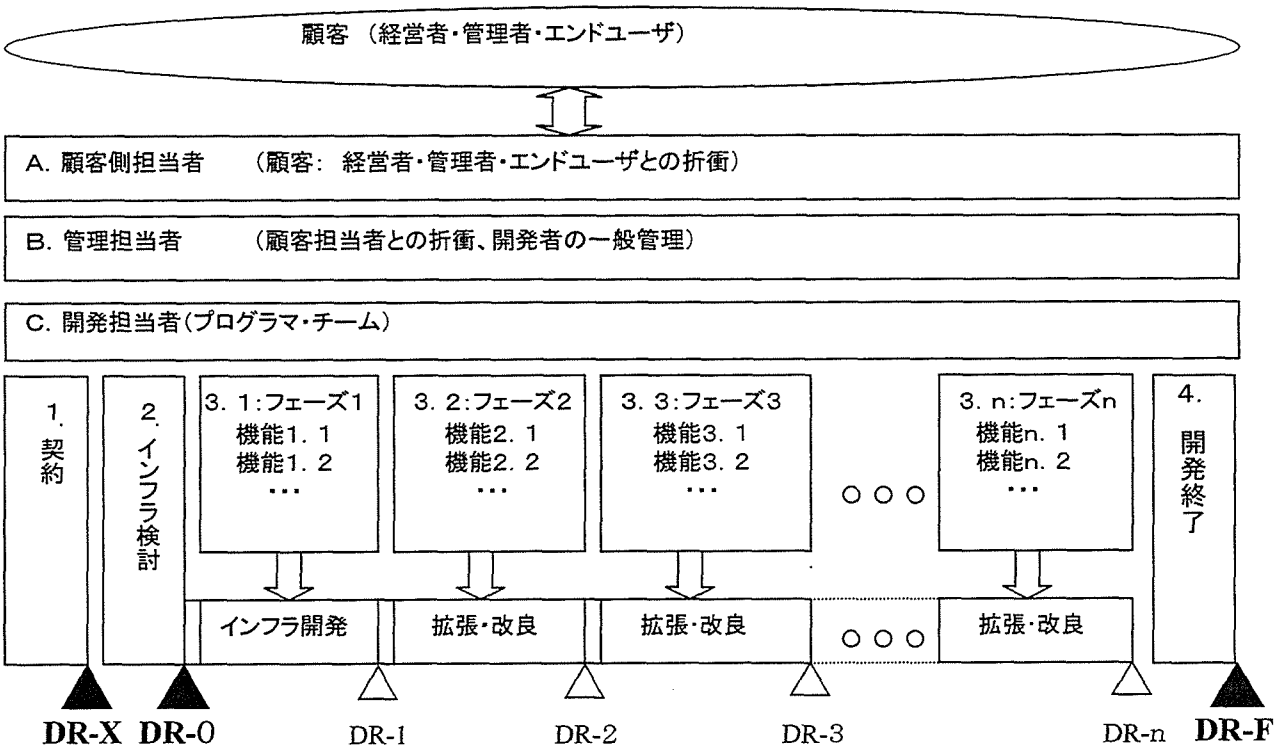


図５．１　設計レビューを重視したソフトウェア開発方式の概要

および開発委託する発注者の関連を明確にするものである.

５．３．１　提案するモデルの前提条件

提案するモデルが対象とするソフトウェアおよび前提条件は以下の通りである.

（１） 開発に関わる人数は数人から高々数十人である.
（２） 対象分野のソフトウェア開発の経験がある.
（３） 新しい開発手法やツールなどの新技術を使用していない.

関与者が何百人にもなる巨大プロジェクト，未知の分野のシステム開発，新しい技術が必要となるシステム開発は研究開発的な要素が強く，本来開発プロジェクトとは別に前もって研究・開発すべき課題である.

実プロジェクトに上記（１）～（３）の要素を含むものは過去の経験から品質，コスト，納期のいずれの面からも問題を起こした例が多い.

我が国のソフトウェア開発の現場では経験の全く無い技術を使い，あるいは技術の無い担当者をプロジェクトに入れそこで教育・訓練させるといった例が多いが，とても専門家集団による知的業務とはいえない．実務で鍛えられることは当然として，基本的な技術の習得や新技術の習得を顧客との契約で実行する開発プロジェクトで行うことは論外である.

５．３．２　モデルの全体概要

（ｉ）　開発の関与者

本モデルでは開発の関与者は表５．１に示すように，顧客側担当者，管理担当者，開発担当者の３種類がいる．もちろん，開発プロジェクトの規模により同じ種類に複数の担当者がいることもあるし，他の開発と兼任してもよい．モデルを精緻にすると多くの役割に細分できるが，原則はこの３種類でよい．

このモデルの特徴は顧客側担当者が開発プロジェクトのメンバーとして組み込まれていることである．もちろん直接開発に携わらないが，開発の各フェーズで開発すべき機能の詳細な仕様を決め，優先度付けを行い，開発するソフトウェアシステムに責任を負う．

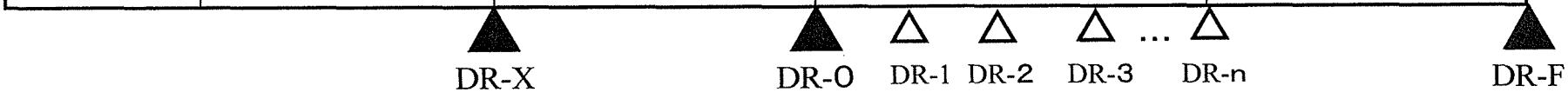
Ａ．顧客側担当者：

プロジェクトの最初から最後まで，発注者側の立場で参画する．１．３節（iii）で述べたように，契約段階で開発すべきソフトウェアの詳細な仕様を決めることは現実として望めず，そのため４．２節（ｖ）で述べたように，開発者と発注者が敵対関係になる傾向があった．顧客側担当者をプロジェクトメンバーとすることにより協力関係を保つようにするのが目的である．

顧客側担当者の最大の役割は，顧客側のシステム関与者(経営者，管理者，エンドユーザ)と開発プロジェクトとの密接なコミュニケーションを

表５．１　開発関与者の主な役割

	契約段階	インフラ検討段階	開発段階	開発終了段階
顧客側担当者	●概略仕様の作成 ・システムの目的 ・システムの範囲 ・システム概略仕様 ●契約折衝 ●顧客側関与者との折衝	●ｼｽﾃﾑｱｰｷﾃｸﾁｬ検討参加 ●アプリケーション機能の開発優先度付け	●開発順序の決定 ・詳細機能の洗出し ・優先度付け ●各フェーズでの機能完了確認 ●各フェーズ完了時の設計レビューへの参加 ●顧客側関与者への説明	●開発完了の確認 ・開発完了範囲の認定 ・機能・性能・品質の確認 ●当初予定機能との差分確認と対応策の策定
管理担当者	●契約折衝 ・開発範囲 ・開発期間 ・価格 ●開発プロジェクトチーム編成の準備	●プロジェクト計画作成 ・プロジェクトチーム編成 ・フェーズ開発計画策定 ・基本スケジュール ・品質計画	●一般管理 ・スケジュール管理 ・コスト管理 ・品質管理 ●開発担当者の開発機能決定 ●開発担当者への支援	●開発完了認定への立会い ●未開発機能の取扱い折衝
開発担当者（プログラマチーム）	●再利用性の検討 ・市販パッケージ調査 ・類似システム開発調査	●システムアーキテクチャ決定 ・ネットワーク仕様 ・共通ライブラリの検討 ●開発フェーズの定義と開発機能の洗出し・優先度付け	●コア部分の改良・拡張 ●各フェーズでの ・開発機能の洗出し ・機能を実現するプログラム開発 ●各フェーズの設計レビューの実施	●開発完了レビューの実施



とり，要求を吸収することである．

顧客側の関与者との折衝により，利用者側の立場から開発すべき機能の洗い出し，開発の優先度付けを行う．また，開発された機能の確認を設計レビューに出ることにより行うと共に，顧客側のシステム関与者に，開発の各フェーズが終了するたびに実際にシステムを動かして説明し，顧客関与者からの要望を聞き出す．

後述するように，ここで提案する開発方式では開発のどのフェーズでも，完成したシステムとして動くことが可能で，ウオーターフォールモデルのように，システム全体が完成してはじめて稼動するということはない．常に部分的ではあるが完成したシステムとして動くことが出来るので，開発の途中でもデモストレーションが可能である．

発注者側に適切な人材がいないとき，コンサルタントなど開発者側とは完全に立場の違う代理人を任命してもよい．

B．開発管理担当者：

開発に関わる管理全般を行う．開発範囲の決定，価格交渉など契約に関わる交渉はAの顧客側担当者と行う．開発管理に関わる事項として，コスト管理，スケジュール管理，品質管理，プロジェクトチームの立上げなどの一般管理を行うが，技術内容については開発担当者に任せる．

C．開発担当者：

開発するシステムの技術に関わる全ての責任を負う．開発すべきシステ

ムの詳細な仕様に関しては，顧客側担当者と協議して決める．

開発担当者は，プログラマチームのリーダのもとで，インフラストラクチャ開発担当者チームとアプリケーション機能開発担当者チームに分類される（図５．２参照）．インフラストラクチャ開発担当者は使用するOS（Operating System）やデータベース管理システム，ネットワークなどの決定を行うと共にシステムの骨組みを決めるアーキテクチャを決定し，システムのコア（後述）の部分を開発する．システムコアは，アプリケーション機能が決められたインタフェース条件で取り付けられるもので，システムコアを中心に，開発システムは段々と成長していく（図５．３参照）．

アプリケーション機能開発担当者は，技術者２人のペアを組みとし，各フェーズごとに優先度に従って割り当てられるアプリケーション機能を開発する．

このモデルの特徴は，システムを発注する側の担当者を，最初から最後まで開発チームの中に入れ，共同責任をとるようにしたことである．１．２節述べたように，顧客側（発注側）は，開発すべきソフトウェアの詳細な仕様については，発注時点でも分からないことが多い．開発当初には仕様が確定しないという性質を持っている以上，仕様の確定の責任を開発段階になってからも発注者側にも持たせるためである．

今まで発注者側は発注するという優位な立場から要求を出すだけで、その実現の責任はなかった．要求仕様の中には

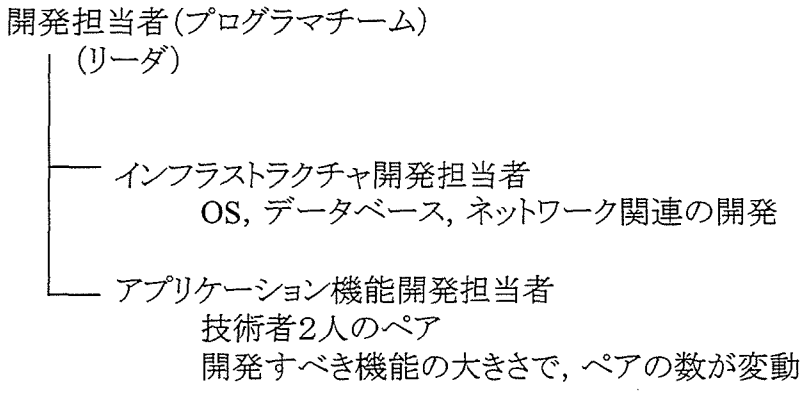


図5．2　開発担当者の役割分担

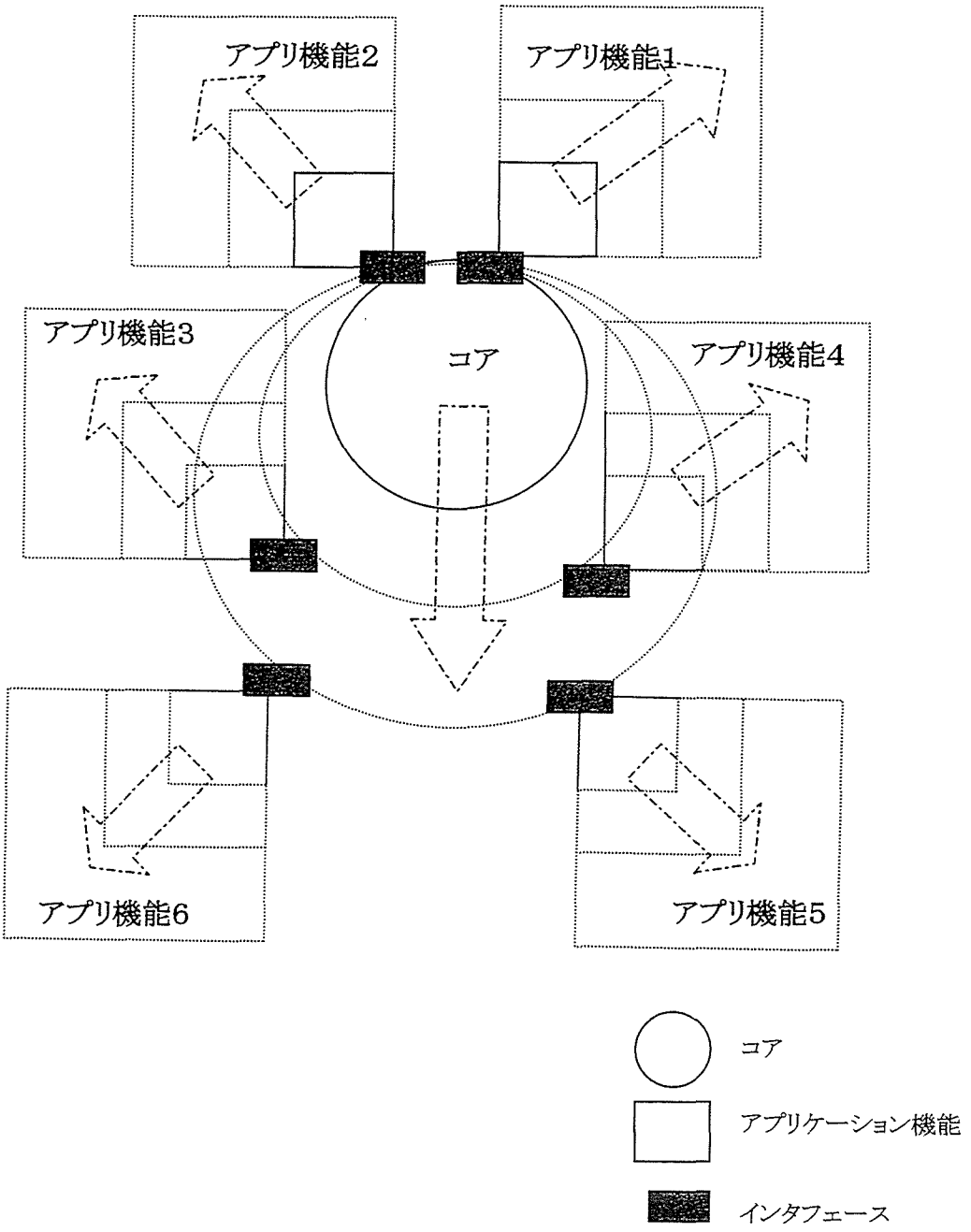


図5．3 フェーズごとのシステムの成長（概念図）

・ ないよりあったほうがいい

・ 技術的な裏づけもないのにコンピュータなら何でもできるといった願望から思いつきで言う

などの例が多かった．このモデルでは，発注側がプロジェクトの一員として入っているので，必然性のない要求，実現性のない要求，あるいは方式の変更にまで関わるような要求を，システム開発の途中で出してくることは防げる．この開発方式の問題として，顧客側担当者が発注側と開発側の間で板ばさみになり，精神的にも労力的にも過大な負担を担う恐れがあることである．また，この役割を果たすには，権限と実力が必要になり，顧客側にそのような人材が期待できるかという問題もある．専門のコンサルタントがいれば良いが，今の日本ではソフトウェア開発のコンサルタントが育っていないという現状もある．

（ii） 開発の過程

図５．１に示すように開発を大きく契約段階，インフラストラクチャ検討段階，開発段階，および開発終了段階の４段階に分類する．各段階の終了時点では，設計レビューを徹底して行うことになる．

契約，インフラ検討，開発終了の後の設計レビューは第３章で述べたDR－X，DR－０およびDR－Fに相当する．

開発段階の各フェーズの終了時点で行うDR－１，DR－２，･･･，DR－ｎは第３章で述べたものとは違っている．第３章で述べたDR－１～ｎは，

開発の各工程の終了時点であるが，このモデルでは，どのフェーズ終了時点でも部分的ではあるがアプリケーション機能が動き，完全でなくともシステムとして完成した状態になっている．

第1段階（契約段階）：

開発すべきシステムの目的，システムの範囲，概略の仕様などを明確にし，必要最低限の契約文書を作成する．大切なのは，開発費用，開発期間を明確にすることで，詳細な機能などについては開発期間を通じて協議することになる．

この段階で重要なのは，概略の仕様から開発に必要な費用と開発期間をいかに見積もるかということである．現在，開発ステップ数（SLOC：Source Line Of Code）やファンクションポイント(FP：Function Point)法など，開発規模を見積もるためのいろいろな技術が提唱されているが，開発の初期には，精度の高い見積を行うことが困難で，かつ，見積りを行う者の主観的な要素が入ることが避けられない．

今まで，建前としての詳細なソフトウェア契約文書を長時間かけて作成していたが，開発途中で仕様が変ることが多く，完成されたシステムは，要求定義段階で作成された仕様からかけ離れたものになる例が多い．契約時，仕様に対して価格付けすることが建前としてあるが，結局，発注側が予算化した金額をベースにして，価格の交渉をしているにすぎない．

始めから，金額と納期を決め，その制約内で開発できるところまで開発する方が現実的である．このモデルでは，現実的な対応を行うことを仮定して

いる.

契約の折衝は,顧客側担当者と管理担当者で行うが,開発担当者は過去に開発したシステムに類似のものがないか,市販のパッケージで使えるものはないか等を検討し,なるべく新規開発する部分を避けるべく技術的な検討を行う.新規開発する部分が多ければ多いほど,プロジェクトは失敗する確率が高いのは良く知られている事実である.

第2段階（インフラ検討段階）:

①管理担当者は,契約に基づき開発計画を作成し,アプリケーション機能開発担当者を含めたプロジェクトチームを編成する.開発費用,開発期間は契約で決められるので,それを基礎にプロジェクトへの投入人数および開発フェーズの数を決定する.開発フェーズとは開発期間を,ある単位期間で分割することである.例えば,開発期間が3ヶ月あるとし,1フェーズを2週間と決めると,このプロジェクトでは6つのフェーズで開発することになる.（図5．4参照）.

②開発担当者は,契約段階で決められた概略仕様に基づき,システムとしてのアーキテクチャを分析・設計し,データベース,ネットワークなどのインフラを新たに開発すべきかどうかを検討する.

システムアーキテクチャとして,データベースやネットワーク,アプリケーション機能から共通に使われる共通ライブラリなどのシステムの基盤となる部分（以降コアと言う）と,開発すべきアプリケーション機能の洗い出しを行う.

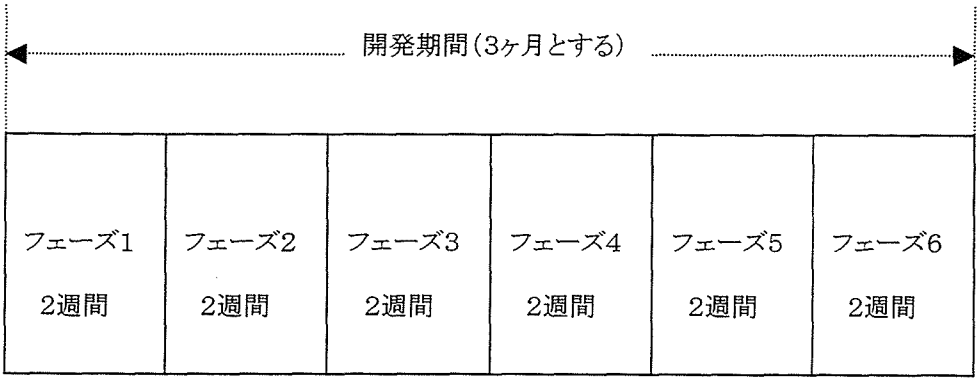


図5．4　開発期間のフェーズへの分割

ここでは，過去の類似システムの活用（再利用）や市販のパッケージソフトウェアが活用できるかどうかを検討し，なるべく新規開発を避けることが重要な観点である．開発するにしても最小限にとどめる．このインフラ検討段階では，開発担当者のうち，インフラストラクチャ開発担当者が行う．

③開発すべきソフトウェアの機能を，その時点でわかる範囲で，インフラストラクチャ機能とアプリケーション機能の両方とも，ペアプログラマで１フェーズ内に開発できる範囲にまで，図５．４のように独立した機能に分割していく．

④分割された各アプリケーション機能を実現するのに必要な，インフラストラクチャ機能との対応付けを行う．図５．５のアプリケーション機能の後ろに例えば，「：{I-A-1．1．2}」と書かれているのは，アプリケーション機能A-１．1．２を実現するのに必要な，インフラストラクチャ機能の集合である．

⑤それぞれのアプリケーション機能を，顧客側担当者と協議し，優先度をつけ，その順位に従い，各フェーズに割り付ける．こうして優先度の高いアプリケーション機能から順に開発されることになる．

⑥アプリケーション機能と④で対応付けられたインフラストラクチャ機能から，どのインフラストラクチャ機能をどのフェーズまでに開発すべきかを決定する．表５．２にどのフェーズにどの機能を開発すべきかを決定した表の例を示す．

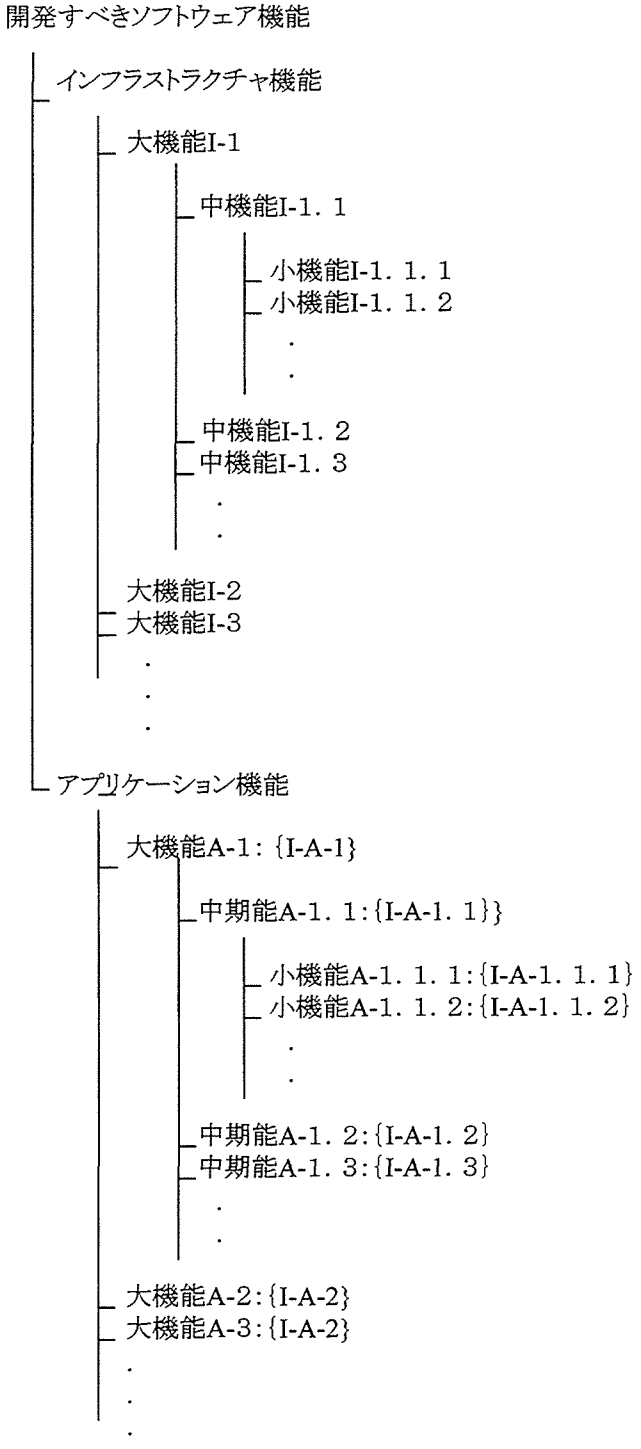


図5．5 開発すべき機能の分割

表5．2 各フェーズで開発すべき機能

	フェーズ1	フェーズ2	・・・	フェーズn
アプリケーション機能	機能A-1.1 機能A-1.2 機能A-1.3 ・・・	機能A-2.1 機能A-2.2 機能A-2.3 ・・・	・・・	機能A-n.1 機能A-n.2 機能A-n.3 ・・・
インフラストラクチャ機能	機能I-1.1 機能I-1.2 機能I-1.3 ・・・	機能I-2.1 機能I-2.2 機能I-2.3 ・・・	・・・	機能I-n.1 機能I-n.2 機能I-n.3 ・・・

この段階で，どの機能を開発すべきかを一応決められるが，必ずしも，この通りに開発されるとは限らない．各フェーズの間に設計レビュー(DR-1～n)を行い，それまでの開発の状況を調べ，開発すべき機能と優先度の見直しを行う．

第３段階（開発段階）：

管理担当者は，第２段階で決められた開発の枠組みに従い，各フェーズの開始時点で，そのフェーズで開発すべき機能を，どのプログラマペアに開発させるかを決定する．開発途中では，スケジュール管理，コスト管理，品質管理といった一般管理を行い，各フェーズの終了時点では，そのフェーズで所期の開発が行われたかどうかを検討する設計レビュー（DR-1～n）に参加する．

もし，開発が予定通り行われていない時には，次のフェーズ以降でどの機能を開発すべきか見直し，改めて優先順位を付け，表５．２の各フェーズでの開発機能の内容の更新を行う．

アプリケーション機能開発担当者は，プログラマペアに対してアサインされた開発機能の実現を行う．開発すべき機能を検討し，プログラムとして実現するためのモジュール構造設計を行い，モジュール内の詳細設計，プログラミング，デバッグ，テストを行う．そして，開発されたアプリケーション機能をコアと結合して，目的とする機能が実現されているか，コアと結合したことにより他の部分に悪影響を及ぼすことはないかを確認する．

目的とするソフトウェアは，コア部分，アプリケーション部分とも，各フェーズが終了するごとに徐々に成長していく．そして，各フェーズの完了時には、全体の機能は実現していない部分的なものではあるが，常に完成したシステムとして動作することが保障される．

インフラストラクチャ開発担当者は,各フェーズでアサインされたインフラストラクチャ機能の開発を行い,それまでに完成しているコア部分に組み込み，コア部分を成長させていく．また，各フェーズで開発されるアプリケーション機能を組み込み，どの時期をとっても，システム全体が完成状態にあることを保障させる．

第４段階（開発の終了）：

最後の開発フェーズが終わった時が開発の終了になる．その時点で，システム全体として，実運用に耐えるかどうかを認定する設計レビュー（DR-F）を実施する．

システムはそれまでに組み込まれた機能で完成とする．当初システムに組み込む予定の機能が開発されずに残っていても,開発は優先度に従い行うので、実運用にさほど影響が少ないはずである．どうしても必要とするなら，別の計画として改めて契約し開発計画を立てることになる．最終設計レビュー（DR-F）はシステム開発関与者全員が参加することになるが，必要なら顧客側のシステム関与者（エンドユーザなど）の参加を求める．

５．３．３　設計レビュー重視の開発

ここで，提案するモデルの最大の特徴は，設計レビューを重視することである．設計レビューには

（1）プロジェクト全体の公式の設計レビュー

（2）ペアプログラマで行う非公式の設計レビュー

の２種類がある．

公式の設計レビューは，表５．３に示すように，契約段階，インフラ検討段階，および開発完了段階の終了時に行う DR-X，DR-０，DR-F と開発段階の各フェーズの終了時に行う DR-１～ｎがある．第３章で述べたように，ソフトウェア開発で失敗する多くの事例では，上流工程できちんとした方針や計画を立てなかったことによる．その意味でも，本モデルでは DR-X，DR-0 を重視している．また，システム完成の認定を甘くすることで，問題の残っているシステムを出荷し，運用させていたことにより，システム運用時に大きな混乱を引き起こしていた．DR-F を厳密に実施することにより，システムの運用後に起こる混乱を避けることは言うまでもない．

DR-1～ｎは開発の各フェーズで行う設計レビューであるが，ここで重要なのは，DR-１～ｎのどの時点でも，たとえ全機能の開発が終了していない部分的なシステムであっても，完成したシステムとして動作することを確認することである．

非公式な設計レビューとして，ペアプログラマ間で行う，相互チェックを徹底することが重要である．ペアプログラマに与えられるのは，各フェーズでアサインされる機能であるが，その開発は全てペアプログラマに任されている．与えられた機能から，詳細な仕様の作成，モジュール構造の作成や

表５．３　公式設計レビューの概要

	DR-X （契約段階の終了）	DR-０ （インフラ検討の終了）	DR-１～ｎ （各フェーズの終了時点）	DR-F （開発終了）
設計レビュー参加者	●顧客側担当者 ●顧客側システム関与者 ●管理担当者 ●開発担当者（リーダ） ●営業担当者	●顧客側担当者 ●開発担当者 ●管理担当者	●顧客側担当者 ●開発担当者 ●管理担当者	●顧客側担当者 ●顧客側システム関与者 ●管理担当者 ●開発担当者（リーダ） ●営業担当者
設計レビューの内容	●システム概略仕様 ・システム目的 ・システム範囲 ・システム概略機能 ・システム目標品質 ●システム開発の実現性 ・過去の開発事例 ・実現可能性 ・開発規模 ・開発機能優先度 ●契約事項 ・開発費用 ・開発期間	●開発計画 ・フェーズ分割 ・開発スケジュール ・品質計画 ・人員計画 ●開発機能 ・開発機能一覧 （コア部分 アプリケーション部分） ・開発優先度 ・使用パッケージ （市販,再利用） ●開発機能担当者アサイン	●フェーズ開発終了の確認 ・開発完了機能の確認 ・品質・性能確認 ・（部分）完成システムの確認 ・未完了機能の確認 ●顧客側関与者の要求の確認 ●次フェーズ計画 ・未完了機能の扱い ・優先度の付け直し ●開発機能の担当者へのアサイン	●システム完成の確認 ・開発完了機能 ・未開発機能 ・追加開発機能 ●品質確認 ・機能・性能・操作性 ●未開発機能の取扱い ●システム運用上の留意点
設計レビューの結果による意思決定	●契約（開発開始決定） ・開発に伴うリスク取扱い ・顧客側参加程度・責任範囲 ●契約に至らない場合の取扱い ・再検討 ・契約不成立	●開発実施の決定 ●開発実施不可 ・再検討項目 ・契約破棄	●追加・拡張機能 ●未開発機能の取扱い ●プロジェクト続行・中断	●開発完了宣言 ●未開発機能の取扱い ●拡張・改良の指摘

モジュールの設計，プログラミング，デバッグ，単体検査，結合検査といった，ソフトウェア開発の全ての工程は，ペアプログラマに任されている．与えられた機能の実現を単に分担するのではなく，全て相互にチェックを行い，全てを一人で開発するのと同じように責任が与えられる．一人のプログラマが全てこれを行うと，個人的な思い違いや思い込みにより是正できないことは良く知られていることであるが，相互にチェックすることでそれらを防ぐことが可能になる．公式の設計レビューや多くの関係者で行うコードインスペクションでは，コードの一行一行にまで亘る詳細なチェックが不可能であり，あるいは可能としても非常に能率の悪いことになるが，ペアプログラマの相互チェックではそれが可能になる．

開発段階で作成される詳細設計などの設計資料はほとんどがワーキングドキュメントして公式には残らないが，プログラムリストは，プロジェクトの最終成果物として残される．プログラムの作成方法として常識である

- 分かりやすいアルゴリズムを使っているか
- 変数などの名前付けは適切か
- コメントは適切につけられているか
- 冗長なプログラムになっていないか
- コーディング規則に準拠して書かれているか

など，保守時，他人が後で読んでも読みやすくなっているかどうかは，この相互チェックをいかに綿密に行ったかどうかに関わっている．公式の設計レビューでは，巨視的な観点からのチェックを行うが，最終的に製品として稼動するプログラミングレベルでの品質を確保するのはこの相互チェックに

関わってくる.

５．４　　提案するモデルの特徴

ここで提案する開発モデルは次のような特徴を持っている.

（１）開発するソフトウェアシステムは進化的に成長する.

（２）開発のどの時点でも完成したシステムとして動作し，顧客からの反応や新たな要求を吸収できる.

（３）技術者の能力を十分に発揮させる.

本節ではこれらについて考察する.

（ｉ）開発するソフトウェアシステムは進化的に成長する

提案する方式では，ソフトウェアシステムは進化的に成長し，どの時点でも完成したシステムとして動作することになる．図５．６にその概念図を示す.開発するシステムはコア機能部分とアプリケーション機能部分よりなる.

簡単のためフェーズ１ではコア部分の（Ｉ）で示された部分だけを開発するとする．コア（Ｉ）の部分は，フェーズ２で開発されるアプリケーション機能の開発に必要な部分だけであり,フェーズ１の終了時点までに開発を完了する.

フェーズ２ではアプリ機能１とアプリ機能２の（Ⅱ）の部分が開発され，フェーズ２の終了までにコア（Ｉ）と結合される．この状態で，システムはアプリ機能１（Ⅱ）とアプリ機能２（Ⅱ）だけの部分的な機能を持っている

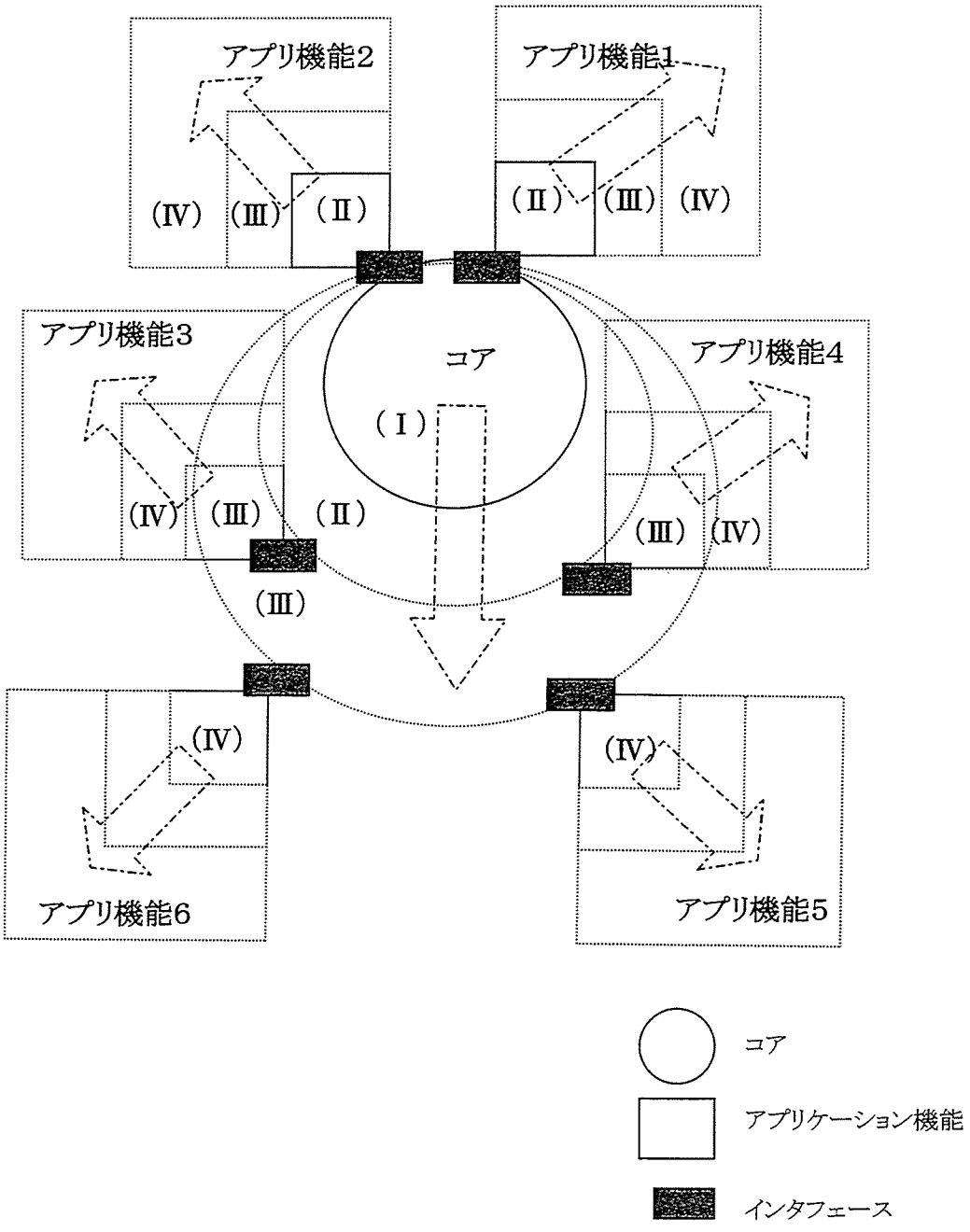


図5．6 フェーズごとのシステムの成長（概念図：機能の開発順序）

が,完成したシステムとして動作する.フェーズ２で開発されるコア部分は,フェーズ３で開発されるアプリケーション機能部分に必要なものである.同様にして,フェーズが進むにつれてシステムは部分的な機能を持つものとして，完成したシステムとして徐々に成長する.

例えば，フェーズ２で完成させる予定の機能が完成できなかったとき，DR-2で，この機能をいかに扱うかが決定される．次回のフェーズ３で完成させることにするか,この機能の実現が当初想定していたより難しいことから，優先度を下げもっと後のフェーズで完成させるか，あるいは，その機能の開発を断念するかなどを決断する．もちろん，機能の実現が想定していたよりやさしかった場合，そのフェーズに予定外の機能を実現させても良い.

システムが徐々に成長する様子を図５．７で示す．図５．７では，図５．６での例えば「アプリ機能 1」の記述を「A1」と記述している．そして，各フェーズの太線で輪郭した部分が,そのフェーズまでに完成した部分である．図５．７でインタフェースとしているのは，コア部分やアプリ部分が成長していっても,コア部分とそれぞれのアプリ部分のインタフェースは変らないことを示している.インタフェースがシステムの成長とともに変るなら,システムが成長するたびに,アプリ部分の全面的な書き換えを必要とする可能性があるので，原則としてインタフェースは固定する．もちろん，システムの成長とともにインタフェースの変更をせざるを得ない場合もあるが,そのときは，該当するアプリ全体の見直しを覚悟する必要がある.

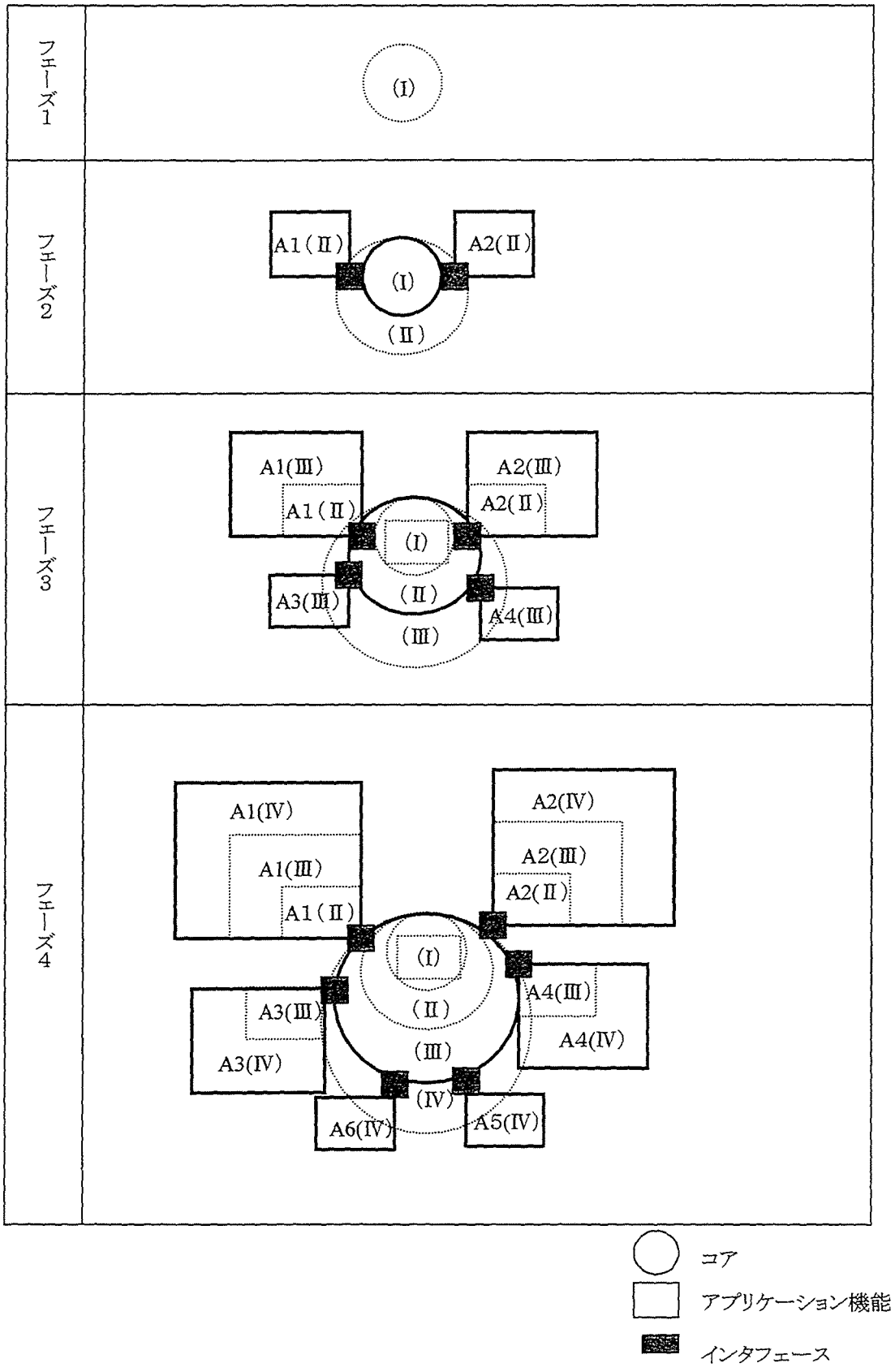


図5．7 システムの成長の遷移

（ii）開発のどの時点でも完成したシステムとして動作し，顧客からの要求を吸収できる

開発のどの時点でも、たとえ盛り込まれた機能は部分的であっても完成した状態にある．顧客側担当者の重要な（最大の）役割は，頻繁に自社に戻り顧客側システム関与者（経営者，システム関係者，エンドユーザなど）にシステムを実際動かしてデモストレーションを含む説明を行い，顧客側の反応や要求を吸収することである．今まで，例えばエンドユーザに開発途中の説明をする場合，開発者が行わざるを得なく，会社間の折衝を行う必要があるなどオーバヘッドが多かった．開発プロジェクトメンバーとして顧客側担当者を入れるメリットは，顧客側との密度の濃いコミュニケーションが図れることである．

フェーズ終了ごとの設計レビュー（DR-1～n）で，開発すべき機能の見直しや優先度の見直しが行われるので，顧客側の要求を必ずしも全て取り入れることは可能ではないが，柔軟に計画の変更が行えることになる．もちろん，顧客の要望により，新たな要求を取り入れるとき，当初開発を予定していた機能の優先度が下げられるものも出てくる．

ウオーターフォールモデルで開発する場合，顧客に実際にシステムを動作させて説明できるのは総合試験が終わり，システムが完成した後であった．顧客は実際に動作して始めてシステムがどのように動くのか分かるので，その時点で顧客からの変更の要求があると，たとえ表面上は些細な変更でも，システムの方式全体に関わる大幅な変更をせざるを得ないものがあり，問題が多かった．

この弊害を防ぐため，プロトタイピングモデルが提唱されているが，多くの場合，人間とシステムとのインタフェース（マンマシンインタフェース）を確認する「使い捨て型プロトタイピングモデル」である．すなわち，システム仕様をなるべく早期に決定するのが目的で，プロトタイピングで仕様が確定するとそのプロトタイプは捨てられ，改めて開発に着手するものである．

仕様を確認しつつかつ継続した開発につなげる「進化型プロトタイピングモデル」の提唱はあるものの，具体的な手法の提案は見あたらない．本モデルは，進化型プロトタイピング開発の具体的な提案としても位置付けることができる．

また，例え部分的であっても，システムが実際稼動することから，顧客側での部分的なシステムの運用を開始し，システム運用の習熟度を高めることも可能になり，スムースなシステム移行を図ることが可能になる．

（iii）技術者の能力を十分発揮できる

ウォーターフォールモデルで開発する場合，開発に携わる技術者は自分に割り当てられた工程を行うだけであった．分析や設計，プログラミング，デバッグ，テストなどを行っても，部分的でシステム全体のどの位置付けをやっているのかが分からない場合が多い．特に，人が作成したプログラムのデバッグやテストだけを与えられたものは，自らの創造性を発揮すると言うより，与えられた業務を受身で義務としてこなすだけという状況にあった．最近製造工場の作業者でも，製造ラインで決められた部分のネジを締めるだけ

といった単純な仕事をすることより,ワークショップ制で始めから終わりまで製品全体の組立を行う方式が作業者の意欲を高め,かえって生産性が高くなるとして注目されている.

開発期間が長期にわたる大規模プロジェクトでは,例えシステム分析を行い要求仕様をまとめても,それが実現され実際にどう動くかを確認するまで時間がかかり，技術者の知的な緊張を維持することが難しかった.

提案するモデルでは，与えられた機能に対しそのフェーズ内に，ソフトウェア開発全般の仕事をすることが求められる.目標が身近にあるため緊張を維持でき，また，分析･設計・プログラミング・テストといった，ソフトウェア開発に必要な全ての作業を任されるので,与えられた機能開発に責任を持てる.

目標が身近にあること,全ての仕事を任されその仕事に責任と権限を与えられることから，技術者の創造性と能力を十分に発揮できるようになる.与えられた業務に愛着を持ち，誇りを持てるようにすることこそ，技術者の士気を高める最大の動機付けである.

豊臣秀吉が，地震により壊れた城壁を修理するのに，修理する範囲を一定の区間ごとに区切り，その区間を少人数のチームで修理させることにより，修理期間を大幅に短縮させたと言う逸話がある．このモデルは，技術者の知的な緊張を持続させ士気を高める意味では，この逸話と同じ意味を持っている.

５．５　　提案するモデルの実用上の鍵

５．３節で提案した開発モデルは，５．４節で議論したように，ウォーターフォールモデルに代表される今までの開発モデルに比較して多くの利点を持っている．しかし，新しい技術が実際に適用されるにはいろいろな障壁が立ちはだかる．特に，４．１節（iii）で論じたように，組織や個人の習慣といった人間的な側面の障壁が多い．本節では，本モデルの実用化について想定される以下のような事項，

（１）契約や商習慣

（２）技術者の能力

（３）組織の成熟度

（４）適用分野

（５）テスト技術

について議論する．

（ｉ）契約や商習慣：

ほとんどのソフトウェア開発で，契約段階では詳細な仕様を決まらないと言う現実を踏まえ，本モデルでは契約段階では，概略仕様に基づき開発期間と開発費用（契約金額）だけを決めることにしている．実現する機能は，運用上の優先度により決め，フェーズが進むに連れて見直し，徐々に開発すべき機能が明確になり，最終の開発フェーズが終了した時点で開発完了としている．

これを可能にするため，開発プロジェクトに顧客側担当者を入れ，顧客側のシステム関与者とのコミュニケーションを密にとれるようにしている．このような契約方式を取れるかどうかが実用化されるかどうかの鍵である．

顧客側担当者には，相当の技術力と折衝能力が要求される．中小企業ではそれを実行できる人材を求めることが困難で，第三者としてのコンサルタントに代行させるしかない．日本では今までコンサルタントがあまり活躍できるような風土が育たなかったが，IT コーディネータ制度[31]が２００１年度から発足したので，今後はこういった人材が活用できると期待できる．

きちんとした仕様に基づき，そのシステムの価値を見積もるという本来あるべき姿の契約を行っておらず，結果としてかかった工数により費用が支払われている現実を認めるなら，このモデルの妥当性を認められ，実施することが可能になると考える．

（ii）技術者の能力：

本モデルで技術的に難しいのは，図５．１のインフラ検討段階である．この段階でシステムのアーキテクチャが決まり，アプリケーション機能を追加するシステムの基盤としてのシステムコアが決まるので，この段階でほぼシステムの骨格が決まる．このアーキテクチャ設計を行う技術者の能力がシステム開発の成功の鍵を握ると言える．今のところシステムアーキテクチャを決めるのは開発経験の豊な技術者に限られているが，システムアーキテクチャを決めることの出来る技術者の育成を考える必要がある．

開発段階の各フェーズで与えられた機能を実現する技術者も，その機能を

実現する責任を果たす能力が求められる．このため，分析・設計・プログラミング・テストの全てを自ら行う必要があり，自らの能力開発を持続して行う必要がある．

情報処理技術者試験やベンダーが行う資格を取得するなどして，プロフェッショナルとしての客観的な評価を得ると共に，受身でなく能動的な技術者として育つ必要がある．組織的にも教育カリキュラムを考えることが必須である．

（iii）組織成熟度：

ソフトウェア開発を行う組織風土が生産性や品質に影響するとの認識から，CMM(Capability Maturity Model：組織成熟度モデル)[9]が提唱されている．図５．７に示すように組織の成熟度をレベル１からレベル５までに分類しており，ISO9000 認証取得[14]できるのはレベル３に相当する．最上位レベル５では，ソフトウェア開発プロジェクトの成功や失敗の経験が常に組織全体に反映され，いわば学習する風土が達成された状態である．

提案するモデルでは，（ii）でも述べたように図５．１のインフラ検討段階でのアーキテクチャ決定が，システム開発の成功や失敗に大きく影響する．組織として過去の事例を蓄積し，インフラ検討段階で類似プロジェクトからの教訓を再利用できるかどうかにかかっていると言える．組織成熟度が少なくともレベル３以上なければ，アーキテクチャを決める技術者の個人能力に依存することになる．逆にいうと，このモデルによる開発を

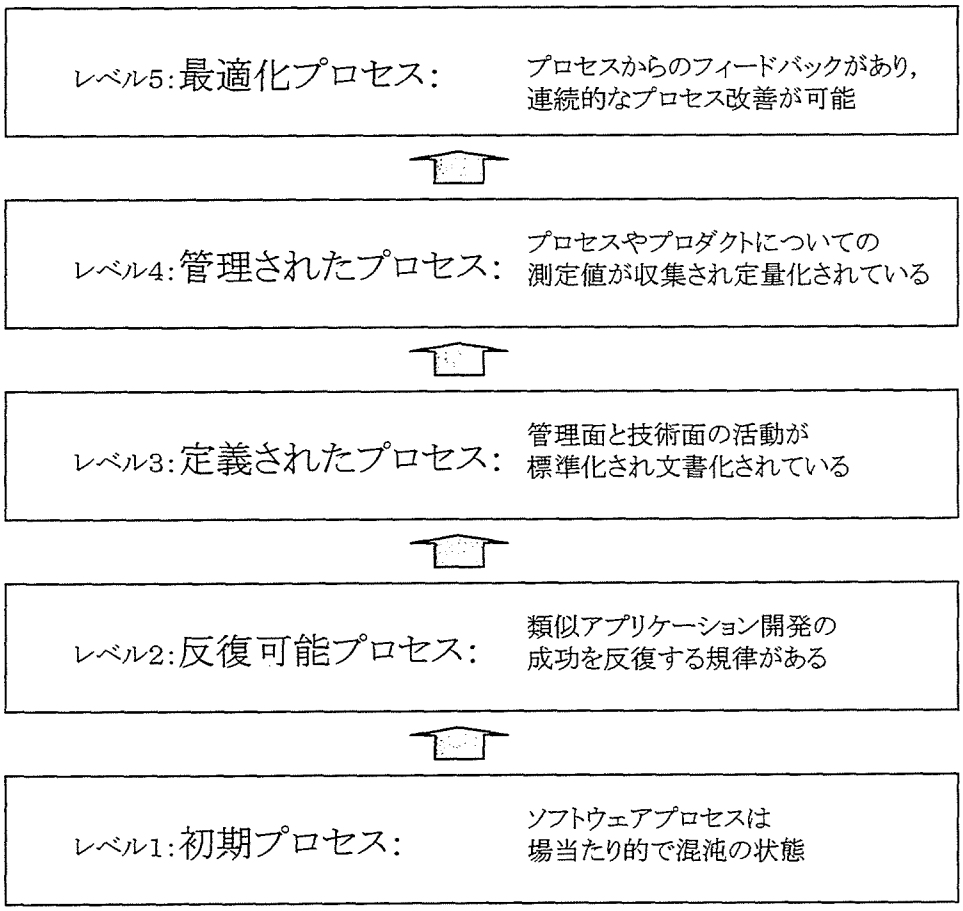


図5．8　CMMによるプロセス成熟度の枠組み

継続していくと組織成熟度が段々と上がっていくことになる.

（iv）適用分野

本モデルが適用できる分野として，例えばデータベースが中心にあり，その周りにデータの入力，加工，出力するようなアプリケーションが繋がるようなデータが中心となるシステムがある.オブジェクト指向分析が可能なシステムなら，個々のオブジェクトを単位機能ととらえることにより，適用が可能である.マイコン組込システムのように制御が中心になるシステムについての適用性は今のところ定かでないが,オブジェクト指向的な分析をするなど，適用の仕方を工夫すると可能と想定できる．今後，適用事例を増やすことにより，適用のノウハウや限界を見定めることが必要である.

（v）テスト技術

提案するモデルではどの時点でもシステムは完成した状態にあり,稼動できる．その状態にフェーズが進むに連れて，コア機能部分とアプリケーション機能部分が追加される.機能が追加されるたびにシステムが完成していることを保障する必要がある．そのため，それまでのテストプログラムとテストデータを全て繰り返す仕組み，すなわち再帰テスト（Regression Test）を機能が追加されるたびに行う必要がある.言い換えるとコア機能やアプリケーション機能が追加され,システムとして成長するたびに再帰テストを行うためのテストプログラムも成長させる必要がある.

このようなテストプログラムを成長させる機能は実現可能であるが[32],

実際にその機構を作り上げるのは大変な労力を要する.開発プロジェクトごとにその機構を作り上げると,テストプログラムのほうが多くの工数を要するかもしれない.例えば,組織的に類似分野ごとのテスト機構を作り,その労力を軽減する工夫が必要である.これを行うためにも(iii)で述べたように組織成熟度を上げ,具体的な開発事例から段々とテスト機構を改良するような組織風土が必要である.

第６章　むすび

本論文では，第１章でソフトウェアの本質的な性質を論じ，ソフトウェアの需要の増大と共に大規模化し複雑化してきたソフトウェア開発を，体系的に行うためのソフトウェア工学の発展を見てきた．そして第２章でソフトウェア工学の集大成としての統合 CASE 環境の開発事例について触れたが，所期の目的を果たしたとはいえなかった．第３章で，ソフトウェア開発に対する取組みの成功事例として，全社的に行った設計レビュー制度に触れ，第４章で２つの事例研究の失敗・成功の原因を議論した．

ソフトウェア工学の生まれたきっかけは，大規模で複雑化したソフトウェア開発を体系的に行うことであるので当然の帰結かも知れないが，今までのソフトウェア開発モデルでは，普遍性を追い求める過ぎ，また管理主導的になりプログラマの独創性を阻害する方向に走りがちであったことを指摘した．

第５章でそれを改善するための開発モデルを第３章で述べた成功事例を取り込むことで設計レビューを重視した開発方式として提案した．

本モデルの特徴は，顧客側の担当者を開発プロジェクトのメンバーとし，顧客側のシステム関与者の要求を取り入れることである．また，開発期間を短期間のフェーズとして区切り，開発のどの時点でも，例え部分的な機能しか実現してないにしても完成したシステムとして動くようにすることである．このことにより，開発各フェーズが終了したときには，顧客側のシステム関与者にシステムを実際に動かして説明できるようになる．

また開発を,顧客側の要望の高い優先度の高い機能から実現するようにし,各フェーズごとに開発の機能や優先度を変えることが出来るようになっている.

このモデルはまだ一部で試しているだけで,大規模なソフトウェア開発に適用しておらず,その有効性を実証する必要はある.今後,大学での学生の共同開発実験や企業での実務への適用を通じて,このモデルの有効性を実証していく所存である.

ソフトウェア開発は,初期にはすべて技術者の裁量に任されていたのが,開発するソフトウェアが高度化し,開発規模が大きくなるにつれ,管理的な側面が増大してきた.特に,大勢の技術者で長期間にわたり開発するようになってからは,QCD(品質,コスト,納期)の3要素を確保すべく,他のハードウェア製品の製造工業と同じように,極度に管理中心になり,技術者の創造性を損なうようになった.

本論文で提案しているモデルは,極端に行き過ぎた管理中心から,技術者の創造性をできるだけ出すようなモデルである.極端に,技術者の自由度を許すと,ソフトウェア開発の黎明期のような混沌の世界に戻る可能性もある.「自由」と「管理」を両極端とする時,その間のいずれかの点に最適解があるはずである.提案する手法が必ずしも最適なものと言えないかもしれない.長い年月をかけ,実務に適用し試行錯誤を繰り返し洗練することが必要になろう.

近年,インターネットの普及により,インターネットやイントラネットのWeb上で動くソフトウェアを,XMLといった言語で開発する比重が大きく

なっている．ソフトウェア開発の持つ本質は変らないが，新しい環境に適合すべく，さらに新しいモデルが求められるかもしれない．環境の変化に対応せざるを得ないのも，ソフトウェア開発のもつ宿命かも知れない．しかし，Web ソフトウェア開発は少人数で短期間で行うものが多く，提案するモデルはウォーターフォールモデルなど既存の開発モデルよりはるかに適合すると考えられる．

参考文献

[1] 情報サービス産業協会（編）:「情報サービス産業白書２０００」，コンピュータ・エージ社，東京（2000）.

[2] J.J.マーキニアク，片山卓也他（監訳）:「ソフトウェア工学大事典」，東京，朝倉書店，(1998).

[3] R.S.プレスマン， 飯塚悦功（監訳）:「実践ソフトウェア工学」，日科技連出版社，(2000).

[4] 宮田操：“ハードウェア記述言語によるトロン仕様チップＴＸ１の記述”，東芝社内レポート（1992）.

[5] M. A. Cusumano : “*Japan's Software Factories*”, Oxford University Press, New York (1991).

[6] M. A. Cusumano and R. W. Selby : “*Microsoft Secrets*”, Free Press, New York (1995).

[7] W. Boehm : “A Spiral Model of Software Development and Enhancement”, *IEEE Computer Magazine*, pp .61-72(May 1988).

[8] 山田茂，高橋宗雄，:「ソフトウェアマネジメントモデル入門」，共立出版，東京（1993）.

[9] M. Paulk, et al. : “*Capability Maturity Model for Software*”, Software Engineering Institute, Carnegie Mellon University, Pittsburgh（1993）.

[９] 江崎和博：“ソフトウェア設計過程における品質の定量的評価と予測に関する研究”，鳥取大学博士論文（2001）.

[10] 山田茂:「ソフトウェア信頼性モデル」，日科技連出版社，東京（1994）.

[11] A. J. Albrecht: "Measuring Application Development Productivity", *Proc. IBM Application Development Symp.*, pp.83-92 (1979).

[12] G. A. Ford : " *1990 SEI Report Undergraduate Software Engineering Education* " ,CMU/SEI-90-TR, Carnegie Mellon University, Pittsburgh(1990).

[13] 大野豊（監修）:「システム開発取引の共通フレーム SLCP-JCF94」, 通産資料調査会，東京（1994）.

[14] 日本品質システム審査登録認定協会（編）:「ＩＳＯ9000 の認定とその仕組み」，東京（1994）.

[15] 大筆豊，石井暁他："ミニコン用システム記述言語も作成"，情報処理学会第１６回全国大会予稿集，pp.341-342（1975）.

[16] 大筆豊，石田和夫他："TOSBAC - ４０用システム記述言語 PL／４０（制御用)"，電子通信学会全国大会予稿集，p.1281（1977）.

[17] 大筆豊，市瀬敦司他："TOSBAC - ４０用システム記述言語 PL／４０の開発"，情報処理学会第１８回全国大会予稿集，pp.761-762（1977）.

[18] 大筆豊："設計言語から記述言語へ，その一例 PL／４０"，情報処理学会第１８回全国大会予稿集，pp.763-764（1977）.

[19] H. Katzan,Jr. : " *System Design and Documentation: An Introduction to HIPO Method*", Van Nostrand, New York(1976).

[20] D.Ross and K.E.Shoman : "Structured Analysis for Requirement Analyasis " , *IEEE Trans. on Software Engineering*,Vol.SE,No.3,pp.16-34（Jan. 1977）.

[21] 中村英夫，大筆豊："SADT サポートツールについて"，情報処理学会

第２１回全国大会予稿集，p. 3C-9（1977）.

[22] 古谷克二，松村一夫，大筆豊："部品化指向の設計･コーディング技術５０SM"，情報処理学会第３１回全国大会予稿集，pp.351-352（1985）.

[23] M.ファウラー，羽生田栄一（監訳）:「UMLモデリングのエッセンス」，星雲社，東京（1998）.

[24] F.P.ブルックス Jr.，山内正寛弥（訳）:「ソフトウェア開発の神秘」，共立出版，東京（1977）.

[25] F.P.ブルックス Jr.，滝沢徹也（訳）:「人月の神話」，ピアソン，東京（1997）.

[26] Σシステム：http://www.pro.or.jp/~fuji/mybooks/okite/okite.9.1.html.

[27] H.D.Mills："*Chief Programmer Team: Principle and Procedures*"，IBM Federal System Division,Technical Report（1971）.

[28] http://objectclub.esm.co.jp/eXtreamProgramming/xp-faq.html.

[29] K.Beck and M.Fowler："*Planning Extreme Programming*"，Addison-Wesley，Boston（2001）.

[30] B.H.Liskov and S.Zills："Programming with Abstract Data Type"，Proc. Symp. *Very High Level Language, SIGPLAN Notices* 9，pp.50-59（1974）.

[31] http://www.itc.or.jp/

[32] C.J.マイヤーズ，松尾正信訳:「ソフトウェアテストの技法」，近代社，東京（1975）.

謝辞

本研究を遂行するに当たり，多大のご指導・ご鞭撻を戴きました鳥取大学工学部社会開発システム学科山田茂教授，河合一教授，および知能情報工学科池原悟教授に，深く感謝します．特に山田茂教授には，指導教官になって戴くきっかけとなった不思議なご縁から始まり，鳥取大学大学院工学研究科博士課程に在学中，一方ならないご指導を賜り感謝の言葉もありません．また，得能貢一助教授をはじめ，山田研究室の皆さんにも暖かいご支援を賜り感謝します．

本研究は，（株）東芝システムソフトウェア技術研究所での研究開発，東芝情報システム（株）での業務が基礎になっており，多くの方々との共同研究の成果ともいえます．これまで御一緒に仕事をさせて戴いた多くの方々に感謝します．特に，３０年近く同僚としてソフトウェア工学関連の研究開発を共にしてきた松村一夫氏に感謝します．

また，本研究を進めるに当たり，力強い励ましとご指導を賜った鳥取環境大学副学長都倉信樹教授に深く感謝します．

最後に，本研究を進めるに当たり日頃から心の支えとなり，また多くの支援をしてくれた妻智子に感謝します．

研究業績一覧表

主論文

[1] Yutaka Ohfude, Kazuo Aida, and Souji Okamoto : "A Stepwise Refinement Tool SPISE", Proceedings of the Sixth International Conference on Software Engineering (ICSE '82), pp.36-38, October 1982.

[2] Yutaka Ohfude : "Expected Change in Software Development using Engineering Workstation" ,Proceedings of the Fall Joint Computer Conference (FJCC '88), pp.14-16, October 1988.

[3] 大筆豊："ソフトウェアコストの見積り技術"，情報処理（情報処理学会誌）,Vol.33, No.8, pp.906 - 911, １９９２年８月.

[4] 大筆豊, 小松一雄, 清水豊, 宝蔵高啓："ＤＲの徹底による経営への貢献"，第１９回ソフトウェア生産における品質管理シンポジウム発表報文集, pp.87-94, １９９９年11月.

[5] Yutaka Ohfude, Kazuo Komatsu, and Yutaka Shimizu : "Through-going Design Reviews Reduce Non-Profitable Project", Proceedings of the Second World Congress for Software Quality (2WCSQ), Yokohama, Japan, pp. 357-362, September 2000.

[6] Yutaka Ohfude : "How to Operate Y2K Problem in a Software House" ,Proceedings of the Fifth China-Japan International Symposium on Industrial Management (ISIM2000), Beijing, China, pp.424-429, October 2000.

[7] Yutaka Ohfude and Shigeru Yamada : "Software Development and Management Methods Considering Programmer's Creativity" ,Proceedings of the Sixth China-Japan International Conference on Industrial Management (ICIM2002), Xi'an, China, September 2002(掲載予定).

参考論文

[1] 高橋生宗，大筆豊："ソフトウェア生産工業化システムＩＭＡＰ"，東芝レビュー，Vol.41, No. pp.2-5, 1986年8月.

[2] 大筆豊，本位田真一，斉藤悦生："オブジェクト指向ソフトウェア開発の現状と展望"，東芝レビュー, Vol.48,No.1,pp.4-8, 1993年1月.

[3] 大筆豊："産業界から学会へのメッセージ，独創的業務の定量化への挑戦"，品質，Vol.23,No.4,pp.169-170，1993年10月.

[4] 大筆豊，津田淳一郎，松村一夫："ＣＡＳＥ技術と当社の取組み"，東芝レビュー，Vol.49, No.9, pp.644-647, 1994年9月.

[5] 松村一夫，津田淳一郎，大筆豊："ソフトウェア品質技術と当社の取組み"，東芝レビュー，Vol.51, No.2, pp.32-34,1996年2月.

著書

[1] 大筆豊：「知的コンピュータ辞典」，産業調査会，平成元年8月
：「ソフトウェア開発技術」, pp.456-464.

[2] 大筆豊：「ソフトウェア品質管理ガイドブック」，日本規格協会，平成2年10月，
：第8章「ソフトウェア工学」pp.131-156,
：第14章「レビューと工程別品質管理」pp.283-304.

[3] 大筆豊：「ソフトウェア工学大事典」，朝倉書店, 1998年12月
：「ソフトウェア開発における総合品質管理」pp.609-620.

研究発表

[1] 大筆豊，石井暁他："ミニコン用システム記述言語も作成"，情報処理学会第１６回全国大会予稿集，pp.341-342，1975年.

[2] 大筆豊，石田和夫他："TOSBAC－４０用システム記述言語PL／４０（制御用)"，電子通信学会全国大会予稿集，p.1281，1977年.

[3] 大筆豊，市瀬敦司他："TOSBAC－４０用システム記述言語PL／４０の開発"，情報処理学会第１８回全国大会予稿集，pp.761-762，1977年.

[4] 大筆豊："設計言語から記述言語へ，その一例PL／４０"，情報処理学会第１８回全国大会予稿集，pp.763-764，1977年.

[5] 中村英夫，大筆豊："SADTサポートツールについて"，情報処理学会第２１回全国大会予稿集，p.3C-9，1977年.

[6] 古谷克二，松村一夫，大筆豊："部品化指向の設計･コーディング技術５０SM"，情報処理学会第３１回全国大会予稿集，pp.351-352，1985年.

[7] 大筆豊，都倉信樹，福山峻一："鳥取環境大学における情報処理教育・グループ研究の取り組みについて"，情報処理学会第６３回全国大会，講演余講習（4），pp.165－166，2001年9月.

END