エンタプライズシステム開発の生産性向上に寄与するモデルベース開発ツールの研究

石川　貞裕

電気通信大学　大学院　情報システム学研究科

博士(工学)の学位申請論文

2014 年 3 月

電気通信大学　大学院　情報システム学研究科

博士(工学)の学位申請論文

博士論文審査委員会

主査　田野　俊一　　教授

委員　小池　英樹　　教授

委員　末廣　尚士　　教授

委員　大須賀　昭彦　教授

委員　栗原　聡　　教授

A model-based development tool for productivity enhancement of enterprise system development

Sadahiro Ishikawa

Abstract

Enterprise systems that support enterprise business processes are indispensable tools for companies today. They embody the ideas of businesses and are a necessity for their continued existence.

Many enterprise systems have been developed over the past 40 years in Japan. System architectures, processing modes, and programming languages became popular at the time at which they were developed. Continued and long-term use of enterprise systems has been a conventional way of maintaining productivity and cost performance.

Development tools are sought after that contribute to productivity and improve the quality of system development of coexisting new and old systems of various architectures.

This research describes (with results of actual field use) a proposed enterprise system development tool with multi-architecture support from batch processing, widely used in the 70's, to on-line processing of the host concentration type, client servers, Web developing, and today's service-oriented architecture (SOA) . The proposed tool is based on model driven engineering (MDE).

Proposed is a style of modeling systems with finalized specifications. It is not a tool for evaluating the productivity of the programmer, but rather the user directly with the aim of improving productivity and quality.

This thesis is comprised of several chapters. First, Chapter 1 describes the background of enterprise systems and their development.

Multi-architecture support is indispensable to enterprise systems developed today, and any development tool that contributes to improving their productivity and the quality is assumed to have the flexibility to respond to various architectures. A development tool is proposed that fulfils this requirement and the results of actual field use are described.

It proposes the development tool that fills these preconditions, and it tries to use in an actual field, and this thesis describes the result.

Three problems hung when it proposed the development tool. One is a necessity for the multi architecture. The second is achievement of the development tool corresponding to the development process where the model base development and the iterative development were combined. The third point presumes the cost of the software development project, and is establishment of the technique for estimating the scale to measure the development progress accurately.

Three issues were encountered when proposing the tool. First, the necessity for multi–architecture support. Second, the complexity of creating a tool that supports processes where model-based and iterative development are combined. Third, the establishment of a technique for estimating the scale of a project so cost and progress can be accurately measured.

Chapter 2 analyzes the three issues posed in Chapter 1.

Chapter 3 describes a new technique for multi-architecture support. The technique is able to manage all aspects of system architecture and the processing mode from which the simple, strong three-layer (P- layer, F- layer, and D-layer) application structure is now often seen.

Chapter 4 describes the development of the proposed tool's capacity for managing the combination of model-based and iterative development. It explains the functions required to integrate two conflicting development processes in a balanced way. The proposed tool is not capable of coding all processing, so the outside these limits must be

manually coded. The proposed tool's method of integrating this automatic and manual coding is described.

Chapter 5 describes the proposed tool's highly accurate scale estimation technique. As enterprise system development (especially on a large scale) is a difficult process, an accurate estimate of the scale of the system is an important factor in project success.

Chapter 6 describes the specific method of operation and the features of the proposed tool. These are comprised of the automatic code generation function of the program, which is expected to be used during the development process, and, where the model-based development is combined with iterative development proposed in Chapter 4 and it according to the three-layer application structure to achieve the multi-architecture support proposed in Chapter 3. Moreover, the functions of other similar development tools available at the same time are compared, and the practicality of this tool's is described. Additionally, the technique for accurately estimating system scale, introduced in Chapter 5, is described.

Chapter 7 envisages that 10 years after the full-scale rollout, the tool is used in the system development of more than 300 enterprise systems in large companies. It shows that iterative development is still effective in continued development and effects on productivity, quality, and utilization of multi-architecture support functions.

Chapter 8 summarizes the results of this research.

One of the factors that has been thought about for ten years or more in the field is the integration of multiple architectures. Most thought that it was an indication of a technology's adoption when new software featuring that technology can be clearly, simply, and easily created.

Moreover, the software development process didn't attach enough importance to the automatic generation of code from the model. Iterative development in this approach is for balance with manual coding.

Agile and model-based system development is assumed to become more popular than normal methods in the future, as they make the best use of the NoSQL database extension. The proposed development tool enables such development.

エンタプライズシステム開発の生産性向上に寄与するモデルベース開発ツールの研究

石川　貞裕

概要

企業のビジネスプロセスを支えるエンタプライズシステムは，今日，企業活動に深く密接に結びつき，事業アイデアを具体化する主要ツールであり，また事業継続にも必要不可欠な存在となっている．日本の主要企業では，システムの導入を開始してから 40 年を超え，開発した年代に合わせ当時流行した様々なシステムアーキテクチャ，処理形態，プログラミング言語が採用され，旧来のものでも生産性やコストパフォーマンスの観点から継続利用が合理的なものは，今日まで引き続き利用されている．

システム開発の生産性と品質向上に寄与する開発ツールは，それら様々なアーキテクチャが混在利用されている現状を是としたものが求められている．

本論文は，70 年代では主流であったバッチ処理やホスト集中型のオンライン処理からクライアント・サーバ，Web，今日の SOA(Service Oriented Architecture)までをカバーしたマルチアーキテクチャに対応可能なエンタプライズシステム向けの開発ツールの提案とその適用成果について述べるシステム開発型論文である．

本ツールは MDE(Model Driven Engineering)に基づいたモデルベース開発ツールである．モデルベース開発ツールは，プログラマの生産性に注目したツールではなく，ユーザと確認した仕様であるモデルを生産性と品質の向上に直接作用させる開発スタイルである．

本論文は以下のように構成されている．

まず第 1 章で本論の目的を述べる．今日開発されるエンタプライズシステムはマルチ

アーキテクチャへの対応が必須であり，開発ツールはそれらに柔軟に対応することを前提にソフトウェアの生産性と品質向上に寄与することが求められている．本論文では，長年フィールドでエンタプライズシステムの開発に携わってきた経験から，これら前提条件を満たす開発ツールを提案し，実際のフィールドで十分結果を残せるか試し，その成果を述べるものである．

開発ツールを提案するにあたり，3 つの課題を掲げた．一つはマルチアーキテクチャ対応の必要性である．2 つ目はモデルベース開発と反復型開発を組合せた開発プロセスに対応する開発ツールの実現についてである．3 点目がソフトウェア開発プロジェクトのコストを推定し，開発進捗を正確に計測するための規模を見積る手法についてである．

第2章では第1章で掲げた3つの課題の分析を行う．

第3章では，一つ目の課題であるマルチアーキテクチャへ対応できる新たな手法を提案する．P層，F層，D層と命名した単純だが強力な3層のアプリケーション構造が，今日までよくみられるシステムアーキテクチャ，処理形態全てに対応できることを明らかにする．

第4章では2つ目の課題であるモデルベース開発と反復型開発を組合せた開発プロセスに対応する開発ツールの実現について述べる．相反する特徴を持つ2つの開発プロセスをバランス良く取り入れるためにモデルベース開発ツールで実現すべき機能要件を明らかにした．あらゆる処理をモデルで示すのではなく，モデルで記述した方が効率的な部分と手でコーディングした方が明確な部分を示し，モデルからの自動生成と手コーディングの両方を反復的に実行できるツールの実現方法を示す．

第 5 章では第 3 の課題として取り上げたソフトウェアの規模見積手法について述べる．本ツールの利用時に効果を発揮する，精度の良いソフトウェア規模の見積手法を提案する．エンタプライズシステム開発は特に大規模な場合，今日においても困難なプロジェクトであることが知られており，精度の良いソフトウェア規模見積は，プロジェクトを成功に導く重要ファクタであることが知られている．

第6章では，第3章で提案したマルチアーキテクチャを実現する3層のアプリケーション構造に従い，第4章で提案したモデルベース開発と反復型開発を組み合わせた開発プロセスとそれと同期して実現するプログラムの自動生成を備えたモデルベース開発ツールと，第5章で示した精度の良いソフトウェア規模の見積手法を実現するツールについて具体的操作方法と特長を述べる．また同時期に出現した他のツールとの機能比較を行い，本ツー

ルの実用性の高さを示す.

第 7 章では，本格的な適用が開始されて既に 10 年がたち，大企業のシステムを中心に 300 以上のシステム開発で利用されている事例を分析する．継続的な開発に効果のある反復型開発機能の適用状況，生産性と品質への効果，マルチアーキテクチャ機能の活用状況を示す.

第 8 章では本研究の成果を総括する.

10 年以上にも渡ってフィールドで継続的に利用されてきた要因の一つは，当初よりマルチアーキテクチャへの対応を考え，実現するソフトウェア構造が明確でシンプルであり，新たに登場する技術を吸収しやすい構造であったことと考える．また，ソフトウェア開発プロセスは，モデルから一方的にソフトウェアを自動生成することにだけに執着せず，適所にとどめ，手作業コーディングとのバランスを求め，反復型開発スタイルを是として取組んできたことにあると考える.

今後は Agile 型の開発スタイルも主流となり，一方では形式手法を取り入れたモデルベース開発も試行され，NoSQL データベースを活かしたアプリケーション開発も広がると想定されるが，本ツールはこれらにも十分対応していけると考えている.

目次

第１章　序論

1．1　本研究の目的

エンタプライズシステムとは，企業のビジネスプロセスを支えるシステムの総称である．販売，在庫，生産など企業活動に直接かかわるもの，財務，調達，人事，給与など組織内部を支えるものがある．また情報を蓄積し，活用しやすい形態で提供することで，企業戦略の立案支援やナレッジ提供などを行う情報分析・提供型システムなども含まれる．金融，電力，水道，ガス，鉄道，航空，宇宙，防衛，公共サービスなどの一部システムは企業・組織内に留まらず，社会全体を支える重要な役割を担っている．ATM などに代表されるオンラインサービスや，Web サービスなど IT システム自身が企業の顔としてネット上より直接エンドユーザや企業にサービスを提供するシステムも近年多くみられるようになってきた．

今日，エンタプライズシステムは，企業内に閉じて利用されているものから，複数企業間，社会全体，さらには国境を跨いでサービスを展開するものまで，機能・サービスを提供する範囲，種類は広がりをみせている．今後も企業活動の施策を具現化するツールとしてその適用範囲を拡大していくものと考えられる．

エンタプライズシステムは開発が活発化した 80 年代から今日まで，長い開発期間，高い開発・保守費，リリース後品質の不安定感，期待した機能・効果が発揮されないなど，開発とその保守にまつわる困難さが度々社会問題としても取り上げられてきた．

初期において，その原因は IT 人材不足と片付けられることが多かった．今日においてもシステム構築は困難な作業であり，同様に人材不足も叫ばれるが，単に開発パワーの不足のみを問題視しているのではなく，企業戦略から連なる IT 化の企画，非機能面を含め利害関係者間の合意に基づいた要件定義，安定して継続的にサービス提供するところまでを範囲として議論されており，ステークホルダとなるサービスを提供する企業の経営者，サービスを提供する部門，IT 部門，開発ベンダ含め関係者全体の問題と捉えられている．今日，IT システムは多くの企業において，経営と直接深く結びついた基盤であると理解されているからである．

欲する機能を満足できる品質，適切なコストで構築するためには，構築を依頼するユーザ企業と請負うベンダ間でしっかりとしたパートナーシップを構築した上で，リスクを共有して進めなければ，成功は望めないと語られるまでになった．つまりリスクは自ら負うのか相手に任せるのか決め，相手に頼るのであればそれ相応の見返りは要求されることを双方合意で取り決めるということである[1]．このように今日ではシステム開発で発生する問題を一方的に開発者のみの責とすることはないが，開発時の困難さは以前と比して楽になったとは言い難い．

その主な要因として，以前に比べれば開発技術，運用技術ともに進歩しているものの，それらが成熟する以上のスピードで，次々と新たな高性能機器，技術が出現し，企業はそれらをビジネスアイデアの具現化，効率向上の武器としてどん欲に取り入れているからである．今や大企業の多くは，これらによる営業活動の活性化，売上げ増やコスト削減，単位時間当たりの効率向上などの定量的効果，社員間のコミュニケーション活性化など定性的効果を計測し，企業の IT 戦略との整合性も意識している[2]．

また，特に日本においては，創業何十年という老舗企業が多く存続しており，業務プロセスの其処此処に独自性を残していることから，お仕着せの COTS (commercial off-the-shelf，既製品)を利用せず，スクラッチ開発を選ぶ比率がいまだに高い．特に大企業において，特注仕様を盛り込んだオーダーメイド型のスクラッチ開発が多くある．

近年，全く新規にシステムを開発するケースは少数であり，既存システムをリニューアルするシステム開発が多い．リニューアルの過程で新技術や新機能を追加し，機能強化が行われる．このシステムは単に新たな技術のみを追求した形態とはならず，80 年代から今日まで変遷をたどってきたエンタプライズシステムの構成技術のいくつかを組合せて形作られるのが一般的である，場合によっては 80 年代あるいはそれ以前から使い続けてきたプログラムをリニューアルして再利用することもまた一般的である．

加えて，そのライフサイクルの長さも大きな特長である．企業の基幹を支えるシステムのライフサイクルは 15 年に迫ろうとしており[3]，その間に発生する大小の機能強化，障害対応はもちろん，サポート期限切れなどから生ずるハードウェアやミドルウェアの入替え，バージョンアップなど，適宜適切なメンテナンスを要す．

エンタプライズシステムのソフトウェア開発における生産性と品質の向上，維持する直接的な仕掛けとして，これまで 80 年代に CASE(Computer Aided Software Engineering)

ツールが，90 年代に第四世代言語(4GL：4th Generation Language)とその開発環境が，2000 年代にはMDA(Model Driven Architecture)が登場してきた.

CASE ツールはメインフレームでの開発が盛んだった 90 年代前半までは花形だったが，4GL 登場により生まれた，まずは作って動かして試す Try & Error 型を推す反復型開発推進派からは敬遠され，また技術的にはクライアント・サーバ，RDBMS(Relational Database Management System)，Web といった技術のキャッチアップに遅れ，次第に衰退した.

4GL が提供してきた開発環境は，今日，特別な言語のための特別な開発環境ではなくなり，ほとんどのプログラミング言語向けに提供されている．プログラミング工程のための高生産環境と捉えられている.

MDA は当初，多くの有力ベンダ，先進ユーザ企業が仕様策定・推進に参加したこともあって華々しく登場し，UML(Unified Modeling Language)普及の論拠としても扱われた．しかし，概念が複雑で，リリースされたツールも初期において未熟だったため，フィールドでの成果が得られず，2000 年代中旬を過ぎるとほとんどの大手ツールベンダが撤退した[4].

2000 年代後半に，組込みソフトウェア開発でモデルベース開発が MDE(Model Driven Engineering)あるいは MBD(Model Based Development)と呼ばれ盛んになり，フィールドでも効果を発揮しはじめた．これに刺激を受ける形で近年エンタプライズシステムのソフトウェア開発でも MDA のリバイバルとして再度注目を集め出した.

組込みソフトウェア開発で MDE は，モデルを初期の MDA のような UML 記述のみに囚われることなく，DSL(Domain Specific Language)や数学的なモデルなども含み，形式的に仕様記述するものと広く捉え直した．MDA では大きく扱われなかったモデルレベルでの自動検証に品質向上効果を見出していた[5].

本論文は，今日においても開発に困難が伴うエンタプライズシステムのスクラッチ開発において，様々な処理形態，システムアーキテクチャをカバーし，これまでの CASE ツールや MDA の開発，利用経験をもとにモデルベース開発と反復型開発の両立を実現した MDE 指向のモデルベース開発ツールを提案し，このツールをシステム開発に適用することで生産性・品質向上と保守性，さらにソフトウェア規模見積精度向上に貢献することを述べる.

1．2　本研究で扱うエンタプライズシステムとソフトウェアの定義

今日，企業活動を支援するエンタプライズシステムは多様であり，本システム開発型論文で扱うエンタプライズシステムの範囲を明確にする．本論文ではエンタプライズシステムを構成する要素のうち，特にアプリケーションソフトウェアの生産性向上を狙ったツールについて述べるため，まずアプリケーションソフトウェアの位置づけについて明らかにする．

エンタプライズシステムのアプリケーションソフトウェアとその他のソフトウェアは以下のように分類する[46].

	分　類	具　体　例
1	エンタプライズシステムの アプリケーション ソフトウェア	業務システムや情報システムのソフトウェア，重要インフラのソフトウェアのうち，パッケージを除いた部分
2	組み込みシステムの ソフトウェア	自動車や家電等のハードウェアに組み込まれているソフトウェア
3	その他	・パッケージソフトウェア ・SaaS やクラウド技術を構成するソフトウェア

図 1-1　ソフトウェアの分類

Figure 1-1　Classification of software.

エンタプライズシステムは，主にアプリケーションソフトウェアとパッケージソフトウェア，そしてハードウェアで構成されるが，本論文で扱うアプリケーションソフトウェアは図 1-1 の１であり，OS やミドルウェアなどパッケージソフトウェアは３に分類される．

アプリケーションソフトウェアの開発は，主にそのシステムを必要とする企業自身，IT ハードウェアベンダ，情報サービス業に従事する人員によって開発されるが，今日におい

ても多くの人員を要し，また依然活発に行われている．

例として国内の情報サービス業のデータを見ると 2012 年度では，情報サービス業全体の売上高は 13.9 兆円であるが，そのうち受託開発ソフトウェア業が 6.9 兆円，49.6%を占める．受託開発ソフトウェアのほとんどがエンタプライズシステムのアプリケーションソフトウェアである．他の組込みソフトウェア業(0.2 兆円，1.3%)，パッケージソフトウェア業(0.4 兆円，2.6%)に比して巨大な産業であることがわかる[47]．

情報サービス業で開発，制作に関わる従事人員は 46 万人であり[47]，そのうち受託ソフトウェア開発に関わる者は 23 万人程度と想定されるが，これにシステムを必要とする企業自身の開発人員，IT ハードウェアベンダのソフトウェア開発者が加わる．如何に多くの人員が関わる産業であるか伺える．

次に，エンタプライズシステムのアプリケーションソフトウェアのうち，どのような特徴を持ったアプリケーションを対象にするかを明確にする．

本研究で開発したツールが対象とするエンタプライズシステムのアプリケーションソフトウェアは，リレーショナルデータベースとのインタラクションを中核処理とするアプリケーションであり，そのソフトウェアの生産性向上を狙う．

リレーショナルデータベースとのやり取りを行なうアプリケーションをターゲットする大きな理由は，アプリケーションソフトウェアの実に 70%～86%がリレーショナルデータベースを採用したシステム構成を取っているからである[14][15]．

1．3　本ツール開発に取り組んだ背景

本研究の中核となるモデルベース開発ツールに取り組もうと考えたのは，1997 年から 2001 年ごろに得たフィールドでの経験による．当時のエンタプライズシステムの技術トレンドとしては，中小規模システム向けの 2 層型のクライアント・サーバを経て，大規模システム向けの分散 OLTP を中核とする 3 層型のクライアント・サーバや CORBA を利用した分散オブジェクト技術が実用期に入り，それらを使ったオンラインシステムの開発が盛んになりつつある時代であった．加えて，数年後には Java 言語が開発言語の主力になると誰もが信じていた時代でもあった．

当時，こういった技術トレンドの波は，メインフレームを中心とした開発との比較で，ダウンサイジング（メインフレームより安価で小型のシステムを連携させてシステム構築），

オープンシステム（特定ベンダの技術にロックインされない，開かれた仕様に基づいた製品の組合せでシステムを構築）などと称されていた．オープンシステムは，90 年代前半は中小システムでの採用が主であった．しかし，97 年頃は従来メインフレームで構築していたようなシステムの開発にも新しい技術でチャレンジするプロジェクトが増えてきた．

これら技術を採用し，エンタプライズシステムを構築しなければならない我々IT ベンダとしては，個々の技術者はこれらの技術を早く吸収し，組織としては信頼性の高いシステムを効率よく構築する技術の確立が求められた．

メインフレームが主力の時代にメインフレーム，ミニコン（サーバ），ワークステーションを連携させた分散処理型のシステム開発にミドルウェアの設計担当として長く参画していた経験を生かし，97 年頃から 2001 年頃までは分散 OLTP や分散オブジェクトを採用するプロジェクトに多く関わり，開発プロセスを整備し，1999 年からは何度か開発ツールを自作した．

2002 年からは組織全体で利用する開発プロセスと開発ツールを整備する立場となり，本研究で述べるツールを開発するに至った．またこれ以降，本ツールを組織に展開する役割も担ってきた．

組織全体で利用する開発プロセスと開発ツールを整備するにあたって，私が課題としてあげたのは，大きく以下の 3 点であった．

- 技術者のスキル
- 保守性
- 技術トレンドの変化

以下，これら課題をどのように対処しようとしたかを述べる．

(1) 技術者のスキル

IT 技術は，他産業に比べればその変化は速く激しいと言われているが，そこで働く大方の技術者は，過去の成功体験に固執し，変化を嫌う点は，他の産業と変わらない．

これら技術者を新しい技術に早くなじませるにはショック療法が必要だが，エンタプライズシステムの開発は，数十名から多ければ千人を超える人員を動員し，また同じメンバ構成で継続作業する例は珍しく，プロジェクトが終了する前から人員はどんどん流動し，同じ知識，同じスキル，一定レベル以上の人員を常に揃えることは至難である．よってプ

ロジェクトマネージャは，絶えず人員の補給と最適配置，継続的な技術力の維持・レベルアップを考えるのが重要な仕事となっている．

このような状況では，現有技術から大きくかけ離れた技術の習得や，そもそも習得に何ヶ月も要する技術の導入は，習得コストの面からも学習時間の面からも実現が困難である．

私が組織に導入する開発プロセスと開発ツールは，従来技術しか知らない，あるいは経験のない技術者にも早期に着手可能なものにしようと考えた．

(2) 保守性

比較的短期間に多くのプロジェクトに関わることが出来たのは，新技術の採用で問題が発生するプロジェクトを渡り歩いたからである．問題の解決には，新技術のメリットと共にリスクを認識し，要求レベルとの折り合いを見つけることが重要であるが，多くのプロジェクトに関わる中で度々苦戦したのは，新技術の良否よりも調査対象の実像を捉えることであった．エンタプライズシステムのソースコードは巨大であり，ソースコードだけを前にして問題点を特定することは不可能であるため，システムの外的症状，動作ログ，仕様書，そして問題が潜んでいそうなプログラムを開発した技術者の証言からプロファイリングしていく．

この中で最も手こずるのは，ソースと同期していない仕様書や，あやふやな証言をする技術者よりも，モジュール化されず，あちこちのメモリ領域に悪さをするプログラム構造と，まともに動作ログが収集されていない点にあった．

組織に展開する開発ツールは，あちらこちらに勝手なグローバル変数を定義させず，機能単位にきっちりモジュール化させ，また開発者に意識させずとも十分な動作ログをパフォーマンスに極力影響を与えずに自動で収集する機能を実現して，問題箇所の特定を容易にしたいと考えた．これは同時にプログラムの保守性向上にも貢献すると考えた．前述したようにエンタプライズシステムのアプリケーションはライフタイムが長く，長期の保守を必要とする．最新の言語，最新の機能を網羅したプログラムが良いプログラムであるとは限らず，平均的な技術をもった開発者であれば誰でも保守が容易なプログラムが良いプログラムと考えた．

(3) 技術トレンドの変化

エンタプライズシステムの開発に関わるようになってからも，システムを構成するあら

ゆる部位で IT 技術は変化してきた．C 言語，UNIX，LAN，リレーショナルデータベース，光メディア，ワークステーション，オブジェクト指向言語，分散 OLTP，4GL，ルータ，SMP サーバ，PC，Internet 技術，オブジェクト指向設計，Web，分散オブジェクト技術など，数え上げればきりがないが，早い変遷に惑わされて主流とならない技術に乗った結果として，数年でプログラムやプラットフォームが負債化したり，またいずれ廃ることが明らかな技術に長期に拘ると，メンテナンス頻度が落ち，結果的にシステムの維持費を高価なものにしてしまう．

このため，この開発ツールに採用する技術を選定するにあたり，長期間主流でありつづけるであろう技術のみを採用することとした．

このように，技術スキルの高い一部の技術者のみにしか使いこなせない技術ではなく，多くの技術者が使いこなせることを念頭に，保守性の良いプログラムを自動生成して，ログ出力などデバッグや保守時に強力な武器となる仕組みを備え，流行の技術を必要以上に追わない，極力長く使い続けることが可能な開発ツールを作りたいと考えた．

1．4　本研究の概要

本論文において提案したいポイントは以下の 3 点である．

1 点目は，長期間主流でありつづけるであろう技術に注目し，エンタプライズシステムで求められる処理方式，システム構成などの要件を整理する．そして保守性の良さを念頭に前記要件を満たすアプリケーションの構造を提案する．これについて概要を（1）で述べる．

2 点目は，ユーザと開発者がターゲットとなるシステムを描き，議論するために必要十分で，多くの開発者が習得可能なモデルを採用してモデルベース開発を実現することと，モデルベースを主としながらも反復型開発の良さを取り入れた開発プロセスを体系化する．またこの開発プロセスに基づき，モデルからどの部分のソフトウェアソースコードを生成し，具体的にどのようなツールを操作して行うかも明らかにすることである．これについての概要を（2）で述べる．

そして 3 点目は，モデルベース開発を行うことでプログラム自動生成による生産性向上と，自動生成以外の手作業でプログラミングを行う部分にも強い制約をもたらしたことか

ら，ソフトウェア規模見積の精度を向上させることが可能になった．その概要について(3)で述べる．

(1)　マルチアーキテクチャに対応出来るアプリケーション構造の提案

今日のエンタプライズシステムに求められる要件は，1．1 で述べたものをまとめると以下のように整理される．

① 新たな技術，機器を受け入れやすい構造・機能を持つ
② スクラッチ開発時において生産性向上，品質確保に寄与する機能，構造を提供する
③ 従来型の処理形態，システム構成も実現可能である
④ ひとつのシステムで複数の処理形態，システム構成を混在実現できる
⑤ 長期に渡る利用に耐えるよう，特定技術に極端に深く依存せず，必要以上に流行に流されていない基本構造となっている

このような要件が生まれた背景には特に日本においてエンタプライズシステムがたどってきた発展の歴史があるからである．

エンタプライズシステムは日本において 50 年代後半から一部大手金融機関や製造業で採用されはじめ，官庁や大企業の多くでは 70 年代に入って導入がすすんだ．ここまでは当時高価な資源であったコンピュータの性能を最大限に活かすため，同様の処理を複数件一括して処理する「バッチ処理」による利用が一般的だったが，70 年代後半には通信回線との結びつきが進み，人とシステムがインタラクティブに情報をやり取りすることで業務処理を進めるいわゆる「オンラインシステム(ホスト集中型オンライン処理)」の導入が促進される．

当時はバッチ処理であれ，オンライン処理であれ，センタ内に設置されたコンピュータいわゆるメインフレームが集中処理する形態が主流で，年々高性能高信頼の機器が開発され大規模な処理に利用されていった．一方で小型化と低価格化も進み，80 年代には中小企業を含む多くの企業で一気に導入が進んだ．

80 年代まではメインフレームを中核とするシステム構成が一般的であったが，80 年代後半より興るダウンサイジングの波により，UNIX サーバ，RDBMS，LAN，PC クライアント，PC サーバ，4GL，OLTP(Online Transaction Processing)モニタや CORBA に代表される分散オブジェクトなどの製品，技術が次第にシステム構成の中核を占め「クライアント・サーバシステム(クライアント・サーバ型オンライン処理)」と呼ばれる処理形

態が一般化した.

90 年代後半からはInternet とその周辺技術が, エンタプライズシステムにもネットワーク技術とネットワークを利用した応用技術として導入されるようになってきた. 特に世界に広がる Internet 上に情報のハイパーリンクの網を作り出す Web 技術と, 当時はネットワーク言語とまで称され, プラットフォーム間のポータビリティに優れた Java 言語が日本においても 2000 年に入ると爆発的に採用されるようになり, 今日までシステム構成の重要要素となっている. これら技術を利用したシステムは「Web システム(Web 型オンライン処理)」と呼ばれ, 今日において最も多くみられる処理形態である.

2000 年代後半から今日までは SOA(Service Oriented Architecture)と仮想化, クラウド, スマートデバイスが特に重要なシステム構成要素となっている.

エンタプライズシステムは, 以上述べてきた 80 年代あるいはそれ以前に開発されたシステムの機能から, 今日の SOA やクラウド上に構築したシステムまでが同時並行的に存在し, 連携し構成されている点が特徴である. 今後新規に開発されるシステムにおいても 80 年代の技術が使われることは珍しいことではない.

エンタプライズシステムでも特に基幹系と称される企業の基盤を支えるシステムの寿命は, 途中で補修や一部入替えなどを施しながら存続し, 平均で 15 年近くに及んでいる[3]. これはサーバなどのハードウェア, OS などの 1 世代など商用ミドルウェア[6]より遥かに長寿命である.

エンタプライズシステムは, 新たに生まれた技術を業務処理に活かす創意工夫から, 各世代で特徴的な処理方式やシステム構成がある(これらを総称してアーキテクチャと呼ぶ). 例えばこれまで述べたバッチ処理, ホスト集中処理型のオンライン処理, クライアント・サーバ型のオンライン処理, Web 型のオンライン処理, SOA などである.

システム開発ツールは, 新たな技術がある程度市場に受け入れられ, 効率的な実装アーキテクチャが明らかになって来た時期に, この新たな技術を素早く高品質にシステムに取り入れるためのツールとして登場する. そのため一般にシステム開発ツールは特定技術との結びつきが強く, 最適化され, その技術が主流のうちは大きな効果を発揮する.

しかし, 次に出現する技術のアーキテクチャが以前とは大きく異なると, 従前のツールでは対応が難しくなる. 例えばCASE ツールはGUI(Graphical User Interface)やRDBMS などクライアント・サーバで主流の技術への対応に手間取り, 4GL は Web への対応が遅

れた. Web 開発におけるフレームワークは毎年のように優れた新たなフレームワークが登場するが，技術革新の激しさは逆に長期利用を前提とする企業の基幹システム構築には採用しにくい.

提供するツールは，時代の技術の変遷に必要以上に流されることなく，長期に渡って利用可能で，効果を発揮し続けるものが求められていると考えた.

大規模なエンタプライズシステムは，ひとつのシステム内に複数の処理形態・システム構成（これをマルチアーキテクチャと呼ぶ）を内在させることも珍しくない．大規模なシステムはひとつのシステムでありながらさまざまな要求を満たす必要があり，たとえばオンライン処理において一般エンドユーザ向けには Web でサービスするが，専任オペレータには RIA(Rich Internet Application)で構築した画面で提供する，また大量データ処理は夜間のバッチで処理するなど，マルチアーキテクチャで構成することは一般的である．開発ツールはマルチアーキテクチャへの対応が必須条件と考えた.

以上，これまでのエンタプライズシステムの開発の歩みを踏まえ，本論文では，まず 2．1 で長期間主流でありつづけるであろう技術に注目し，今日利用されている処理形態，システムアーキテクチャ，プログラミング言語などを明らかにし，3．1 でそれらの技術を使って実際に構築されるシステム構成を定義する．続いて 3．2，3．3 で，3．1 て定義したシステム構成で動くアプリケーションソフトウェアの構成を，保守性の良さを念頭に提案する.

(2)　モデルベース開発と反復型開発を実現する開発ツールの実現

MDE は，一般的なプログラミング言語よりも，ユーザと設計者との仕様相互理解に優れ，対象ドメインの構造，動作をより直接的に表現可能な仕様記述言語（この仕様記述言語をモデルと呼ぶ）を開発に利用する．このモデルを主体に仕様検証して品質を担保し，またこのモデルからソースコードの生成，テストシナリオの生成などに利用することで，IT システム開発の生産性，品質向上とユーザ，ベンダ間の理解齟齬解消を目的とする技術である.

MDE は，エンタプライズ系のシステム開発で OMG(Object Management Group)が 2001 年に MDA を提唱して一躍脚光を浴びた．時には厳しい批判も受けているが[4]，今も MDE 実現のためのリファレンス的な役割を果たしている.

近年，産業界における MDE 適用状況が報告された[7][8]．適用に向けた努力はエンタプライズ系，組込み系の両分野に渡って継続的に行われている．

MDE の生産性や品質向上に対する一定の効果は産業界においても認識されてきているが，一方では適用先ドメインに適合するモデル選定の難しさや，モデラーの育成，開発環境に対する学習コストの吸収，組織マネジメントの欠落や不完全なツールによるラウンドトリップ・エンジニアリングの適用失敗など，抱える問題が未だ多いことも指摘されている．また，開発コストが 1/10 になるといった革新的な成果は，限られた条件下でしか報告されていない．

ここでは一つの製品，プロジェクト，あるいは一企業内の開発に閉じて採用された例がほとんどであり，特にエンタプライズ系では大規模なシステム開発に対して長期間，組織を挙げて採用された例は見られなかった．

1980 年代初期より，当時急増していたエンタプライズシステムの開発要求に応えるため，多くの CASE ツールが提案，開発され，多数の企業で採用された．

それまではプログラミング言語自身の進化と，コーディング，デバッグを行なう環境が発展してきたが，開発量が爆発的に増えてきた 80 年代に入り，開発プロジェクトに関わる人数が増え，また保守という作業も意識されるようになってきた．多人数が関わる作業には標準化が必要であり，この時期，開発プロセスとドキュメンテーションの標準化が各々のプロジェクト，あるいは大手のベンダ，一部の IT 導入先進企業で取組まれるようになった．

標準化，形式化したドキュメントには，当然ながらプログラムソースを記述するために必要な情報が多く含まれるため，じきにドキュメントからプログラムを生成するという発想が生まれる．これが CASE ツールである．

しかし大半の企業では CASE ツールの適用が限定的な範囲に終わり，また多くの COTS の CASE ツールは，自身で実行基盤をも提供する，あるいは実行基盤を狭く限定していたことに起因して 1980 年代後半から隆盛する実行基盤，開発基盤の大きな変革となったダウンサイジング（メインフレームなど大型コンピュータの一極集中処理から，安価な小型コンピュータを複数台連携させる分散処理へのパラダイムシフトを指し，Visual Basic に代表される 4GL とそれが実現する GUI，RDBMS，LAN を中核技術とした）の流れに素早く追従出来ず衰退した．そのためエンタプライズシステムにおいて MDE への期待は

CASE の衰退を教訓に語られている[5].

我々もメインフレーム上でのソフトウェア開発が主流であった 1980 年よりエンタプライズ系ソフトウェア開発向けの CASE ツールを自社開発し, 広く社内外で適用していた[9]. 1980 年代後半のダウンサイジング時に流行したクライアント・サーバ・アーキテクチャ, 1990 年代後半から出現する CORBA や J2EE, .NET, SOA などコンポーネント指向, 分散オブジェクト指向アーキテクチャ, Web 技術などめまぐるしく新技術が登場する中, 独自に技術を進化させ追従してきた [10].

特に CORBA に端を発するコンポーネント指向, 分散オブジェクト指向へ対応するため大幅に構造を見直し, アプリケーション・フレームワークとモデルから生成されるプログラムコードより構成する形にリニューアルを果たした. モデルを原本とし, 生成されるコードは必ず 3 層構造にマッピングされることを特長とするモデルベース開発ツールを開発した[11]. これにより (1) で述べたメインフレーム時代からあるバッチ処理からクライアント・サーバ, Web オンライン, SOA など複数の処理形態・システム構成 (マルチアーキテクチャ) への対応が可能となった. 加えて反復型の開発も可能なように進化させてきた.

私が提案する開発ツールは, MDE を指向しているが, モデルで全てを表現し, モデルで全てを制御しようとは考えず, モデルで扱った方が効率良く, ソフトウェア構造をシンプルに維持し, 開発・保守プロジェクトを統制することに都合が良い部分に限定している. 残された部分はモデルで表現しても実装言語で書いても作業効率上大差なく, モデル化する効果は薄いと判断し, あえてモデル化の対象から外している.

本論文では, まず 2. 2 でモデルを主としながらも反復型開発を行えるツールの必要性を述べる.

続いて 4. 1 で, 多くの技術者が受け入れやすい仕様記述をモデルとして採用することを述べ, そのモデルで記述してプログラムを自動生成する範囲と手作業でコーディングする範囲を明確にし, モデルベース開発と反復型開発を両立させる開発プロセスを体系化する. 4. 2 では 4. 1 で定義したモデルから自動生成されるソフトウェア部位を明確化する.

(3)　モデルベース開発がもたらすソフトウェア見積の精度向上

80 年代では IT 技術者不足が主要因と考えられていたシステム開発作業の困難さは，今日，IT システムが社会や我々の生活に密着した重要インフラとして浸透し，システム障害によるサービス停止や機能障害が社会的影響を与える存在となったこともあり，開発作業のみならず，企画・要件定義，保守・運用までのシステムライフサイクル全般を通して，開発を請負う者のみならず，発注者，サービスを提供する組織，そしてその組織の長まであらゆる関係者が権限と責任，作業を役割分担して負うものという考えが浸透してきた[12].

しかしながら，この関係者の中でも直接利害が相反する間柄でもある，システム開発を依頼する発注者と，受託開発を行う受注者の関係は長年課題を抱えてきた．意図に沿った機能を備え，妥当な品質を持つシステムを妥当なコストで開発するには発注者と受注者間のパートナーシップが構築されている必要があるが，発注者から受注者にはコスト構造の不透明さが，受注者から発注者に対しては要件定義の不透明さが，それぞれパートナーシップ確立の大きな阻害要因であると指摘されている[13].

ソフトウェアを新規に開発する場合，そのコストに最も大きなインパクトのあるパラメタはソフトウェアの開発量・規模であり，発注者から受注者に投げかけられるコスト構造の不透明さは，ソフトウェアの規模を合意可能な方法，表現で明確化することで，不透明さの解消に寄与できる.

ソフトウェアの規模は，従来からファンクションポイント(FP : Function Point)か，ソースコードのステップ数(LOC : Lines of Code)を単位として利用することが広く認知されている.

システム開発の初期段階では，システム化する外部仕様から算出可能で，計測ルールも明確なため客観性が高い FP がよく利用される．比して LOC を初期の見積に利用するには様々な条件を考慮しなければならない．例えば開発者の能力，嗜好，プロジェクト個別の標準といった人的・組織的要因，プログラミング言語，付帯のライブラリや開発環境など技術的要因に大きくサイズが左右される．このため LOC での見積は過去の類例に頼ることになるため客観性に欠け，長くパートナーシップが構築されている発注者受注者間であれば利用に問題ないが，新たにパートナーシップを確立するもの同士では誤解を生みやすい.

一方，FP は計測を自動化することが難しく，トレーニングを受けた人手に頼るため，

特に大規模なソフトウェア開発を行う場合は，計測コストがかかることと，プロジェクト後半においては，成果物(特にプログラムソースコード)との直接的マッピングが困難なケースもあり，開発プロジェクト後半では利用価値が半減してしまう．プロジェクト後半では現在でも LOC で成果物量を計測し，プロジェクトの進捗と品質評価指標に利用されるケースが圧倒的に多い．

このように現在のシステム開発においては，FP と LOC を適切に使い分けて開発プロジェクトを運営していく必要があるが，LOC は前述したようにプロジェクトで採用される人員，標準，技術など様々な要素が絡んでいるため，FP から適切に LOC に変換することは一般に困難であることが知られている．

本論文では，2．3 で FP と LOC の各々の利点と問題点を再整理するとともに 2 つの指標を組合せて利用することで，プロジェクトマネジメントへ寄与できることを述べる．

5．1 で FP の計測ルールからその算出構造を明確にし，5．2 において本研究で提案したモデルベース開発ツールの採用によってもたらされる統制が，LOC が持つプロジェクト固有の人員，プロジェクト標準，技術などの差異を小さくするため FP から算術的に LOC を精度よく導出可能ではないかとの仮説を立てる．5．3 では実データを利用した換算実験を行い，課題に対しての対応策をまとめる．

第 6 章では，これまで述べた(1)から(3)までの研究成果を実現したツールの具体的操作を示し，第 7 章では，これまで適用してきたシステム開発におけるツールの実績と成果を，さまざまなシステム構成に対する適応性，長期に渡る反復型開発の実践，生産性，品質の観点から述べる．

1．5　本研究論文の構成

第 1 章では，ここまで述べてきたように，本研究で扱うエンタプライズシステム開発の課題とその社会的背景，私が考えるエンタプライズシステム開発の生産性向上を達成する上での問題点と取組み方針，そして解決する手段を簡単に述べた．

第 2 章では，以下第 3 章から第 5 章で扱う課題を詳細に示す．

第 3 章では，長期利用されるエンタプライズシステムの特徴を考慮し，今日採用される

処理形態，システムアーキテクチャ，プログラミング言語の組合せで示す８つのシステム構成を定義する．次に多くの技術者が理解しやすいＰ層，Ｆ層，Ｄ層と命名した単純な3層のソフトウェア構造を提案し，この3層に分けたソフトウェア構造で８つのシステム構成を実現する方法を述べる．

第4章では，多くの技術者が習得しやすく対峙するユーザにも理解しやすい仕様記述のレベルと，目的とするソフトウェア構造を自動生成することが可能な仕様記述のレベルを調整して仕様を記述するモデルを選定し，モデルベース開発ツールで実現すべき機能範囲を明らかにする．あらゆる処理をモデルで示すのではなく，モデルで記述した方が効率的な部分と手でコーディングした方が明確な部分を示し，モデルからの自動生成と手コーディングを反復的に繰り返せる開発プロセスを体系化し，体系化したプロセスに沿って利用されるツールの実現方法を示す．

フィールドで長期，多数の実績を得るには，ツール開発と同時に開発プロセスを整備し，開発部隊だけでなく支援組織も整備し，教育を行い，利用に関するノウハウを蓄積・共有していく必要がある．第5章ではこれら付帯的に整備すべき方法論，ツール，組織の中から開発対象となるソフトウェアの規模見積手法について述べる．

第6章では，これまで述べた第3章から第5章までの研究成果を実現したツールの具体的操作を示す．

第7章では，これまでフィールドで10年以上，300システム以上に適用した実績を，さまざまなシステム構成に対する適応性，長期に渡る反復型開発の実践，生産性，品質の観点から述べる．

第8章は，結言とし，まとめと今後の課題について述べる．

第2章　問題点の分析

この章では，1．2 で述べた本論文の 3 つの主題である（1）マルチアーキテクチャに対応出来るアプリケーション構造の提案，(2) モデルベース開発と反復型開発を実現する開発ツールの実現，(3) モデルベース開発がもたらすソフトウェア見積の精度向上の 3 つのテーマについて，それぞれ現状の問題点を分析する．

2．1　マルチアーキテクチャで構成されるエンタプライズシステム

はじめに，現在どのようなエンタプライズシステムが，スクラッチ開発の対象となっているかを示す．

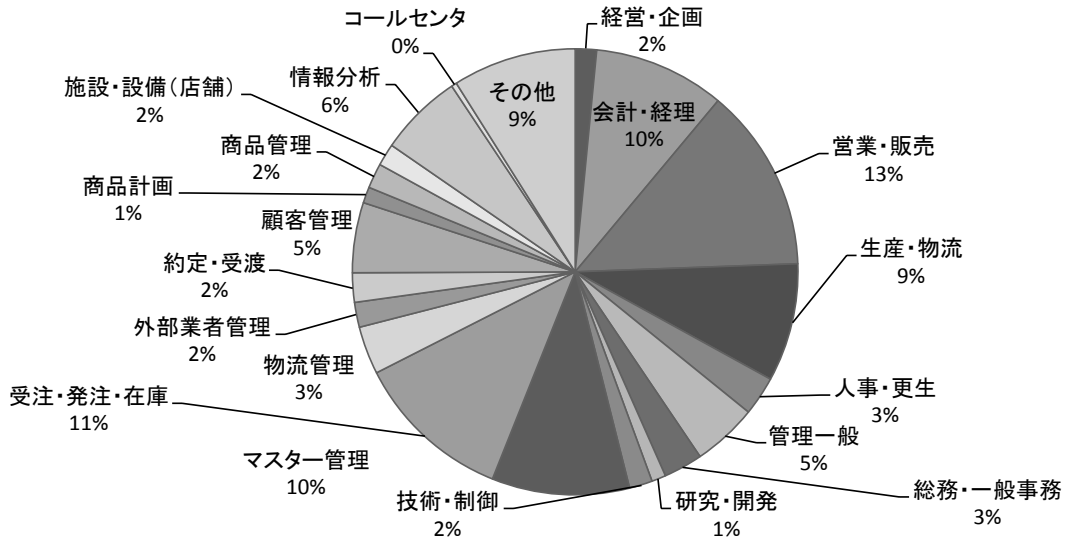


図 2-1　業務種別（JUAS データ）

Figure 2-1　Kind of business system specialty (Data of JUAS providing).

図 2-1 はユーザ企業が自社利用のために開発したエンタプライズシステムの業務種別を表しており，製造業，情報通信業，電気・ガス・水道業，金融・保険業を中心にここ 10 年で開発された開発予算が 500 万円以上の 918 プロジェクトのデータからなっている[48].

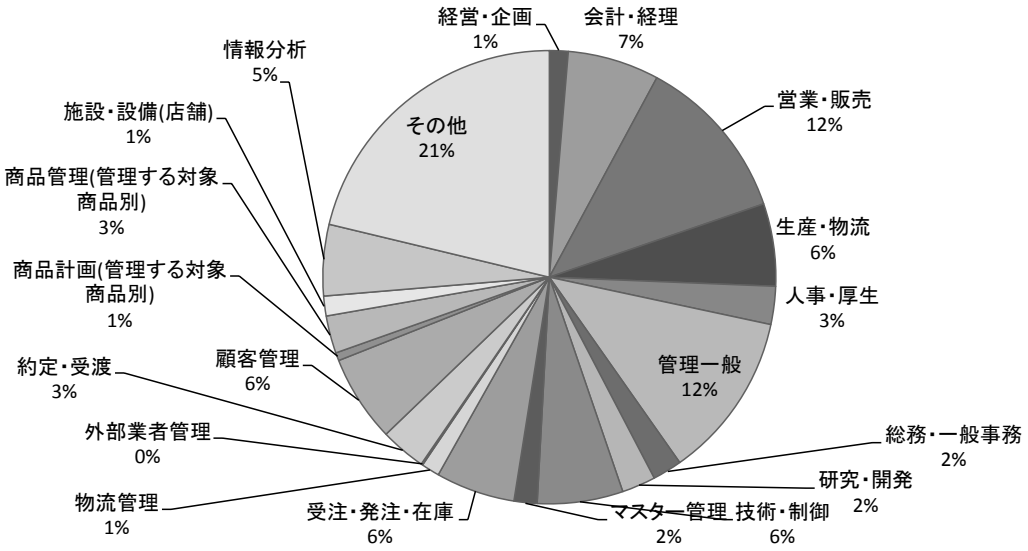


図 2-2　業務種別（IPA/SEC データ）

Figure 2-2　Kind of business system specialty (Data of IPA/SEC providing).

図 2-2 は主に IT ハードウェアベンダ・IT サービス企業から収集した，ここ 10 年で開発された 2,834 プロジェクトのデータからなっている[49]．エンタプライズシステムではないソフトウェア開発も少数含まれている模様だが, 90%弱のデータが製造業, 情報通信業, 卸売・小売業，金融・保険業，公務などからの受託により開発したプロジェクトのものである.

両データで 5%を超えているのは会計・経理，営業・販売，生産・物流，管理一般，技術・制御，受注・発注・在庫，顧客管理，情報分析といった業務であり，エンタプライズシステムとしてはどれも一般的で，データベースへの情報蓄積と，蓄積されたデータの分析・加工により業務を支援するものが大半を占めている.

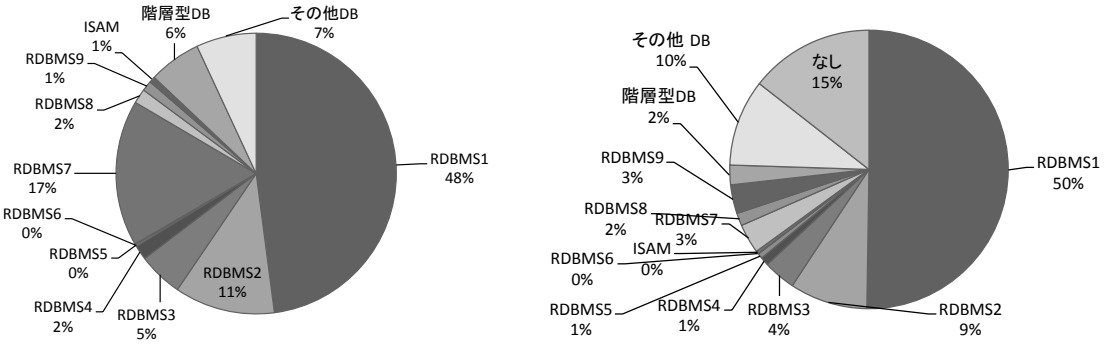


図 2-3　データベースの利用状況（JUAS データ，IPA/SEC データ）

Figure 2-3　Database management system adopted with system (Data of JUAS and IPA/SEC providing).

図2-3は, 図2-1,2-2のプロジェクトで採用したデータベースの状況を示している. JUASのデータでは86%[14]が, エンタプライズシステム以外のデータが若干含まれるIPA/SECのデータでも70%強[15]のプロジェクトがリレーショナルデータベースを利用している.

日本においては諸外国に比し，大企業や国の機関などで利用される基幹システムの構築にあたり，商用パッケージやSaaSなどCOTSの利用よりもスクラッチ開発が選択される傾向にある[8]．これは国の機関をはじめ日本の多くの企業が創業何十年という老舗が多く，百年を優に超す企業も珍しくない点と関連がある．

長年の工夫によって培われた仕事のやり方を組織の強みや独自性と捉え，IT化においてもお仕着せのビジネスプロセスや機能を全ては受け入れず，独自のスタイルを貫いている部分がある．このため重要な基幹システムの開発にあたってはスクラッチ開発を選ぶ企業が多く，また一度開発されたシステムは長年に渡って利用される点も大きな特長である．

日本企業のIT導入は大企業でみれば既に40年を超えている企業が多く，もともとはメインフレーム上のアセンブラやCOBOL言語で書かれたバッチ処理からスタートし， ITテクノロジの変遷に沿ってメインフレーム集中型のオンライン処理やVisual Basicなど4GLで書かれたクライアント・サーバシステム，近年作られたWebシステムなど新たに出現する技術が次々採用された．新しく登場した技術は旧システム，旧技術を完全にリプレースする場合もあるが，旧システムに追加される形で採用されるケースも多い．

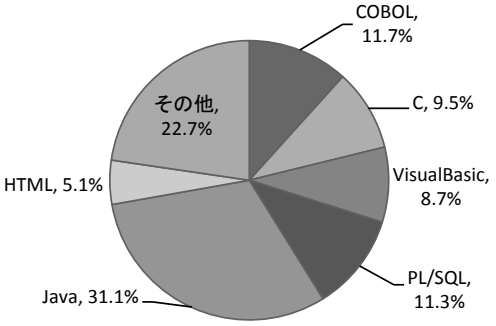


図 2-4　プログラミング言語の利用状況

Figure 2-4　Programming languages used by development.

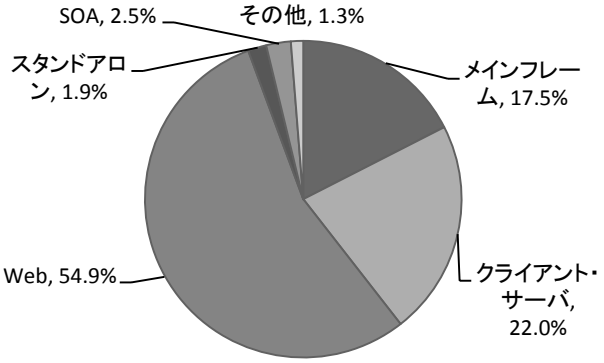


図 2-5　システムアーキテクチャの利用状況

Figure 2-5　System's architecture adopted with system.

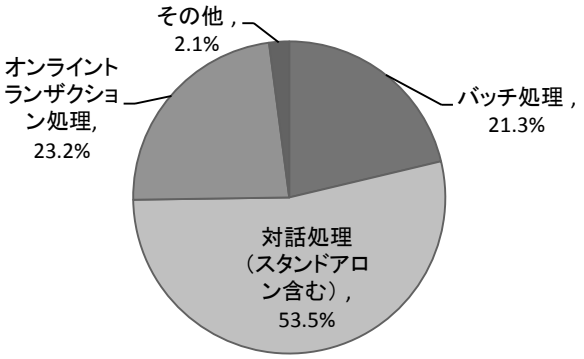


図 2-6　処理形態の適用状況

Figure 2-6　Processing type adopted with system.

図 2-4，2-5 は JUAS のデータで 2012 年までに完成したシステムの言語，システムアーキテクチャ[14]を，図 2-6 は IPA/SEC のデータで 2011 年までに完成したシステムの処理形態[15]を示している．

これをみると近年でも，言語では COBOL から Java まで，システムアーキテクチャではメインフレームから SOA まで旧来の技術から新しい技術まで幅広く利用されていることが伺える．

また，1 つのシステム内で複数のプログラミング言語と，バッチ処理，オンライン処理といった処理形態，スタンドアロン，クライアント・サーバ，Web といったシステムアーキテクチャが混在していることも珍しくない[14]．

さらに企業全体では，開発された時期の影響を受け，異なった技術の組合せで出来た複数のシステムが共存して成り立っている．このようにひとつのシステムに複数の技術，あるいは複数技術が併存するシステム群の構成を本論文ではマルチアーキテクチャと呼ぶ．

新たに高効率の開発ツールが現れようとも既存機能のリワークを含めて新しいアーキテクチャに一気に移行することはコスト，開発期間から考えて現実的な選択とならず，既存リソースをうまく再利用しながら，新たな機能を追加するアプローチを取る企業が多い．

開発ツールに求められる機能とは，スクラッチ開発において，特にリレーショナルデータベース利用時の生産性を高め，複数の言語をサポートし，既存システムや既存資産の再利用を意識し，多様な処理形態・システムアーキテクチャ，つまりマルチアーキテクチャを実現可能な柔軟性が求められる．

2．2　モデルを主とする反復型開発ツールの必要性

開発生産性に強く影響する要因は，事例の蓄積による努力により明らかにされている[16]．

図 2-7 は，右に正の方向の影響度が高い要因を，左に負の方向に影響度が高い要因を並べてある．

これによれば再利用(Reuse of deliverables)が良くも悪くも生産性に最も影響を与える．

次に続くのが開発者の経験に係る部分であり，特にプロジェクトマネージャに代表され

る管理者の経験(Management experience)の影響度が大きい．続いて要求の確定度合い(Clear and understandable requirement)，開発プロセス(Methods or process)，ツール，言語と続く．

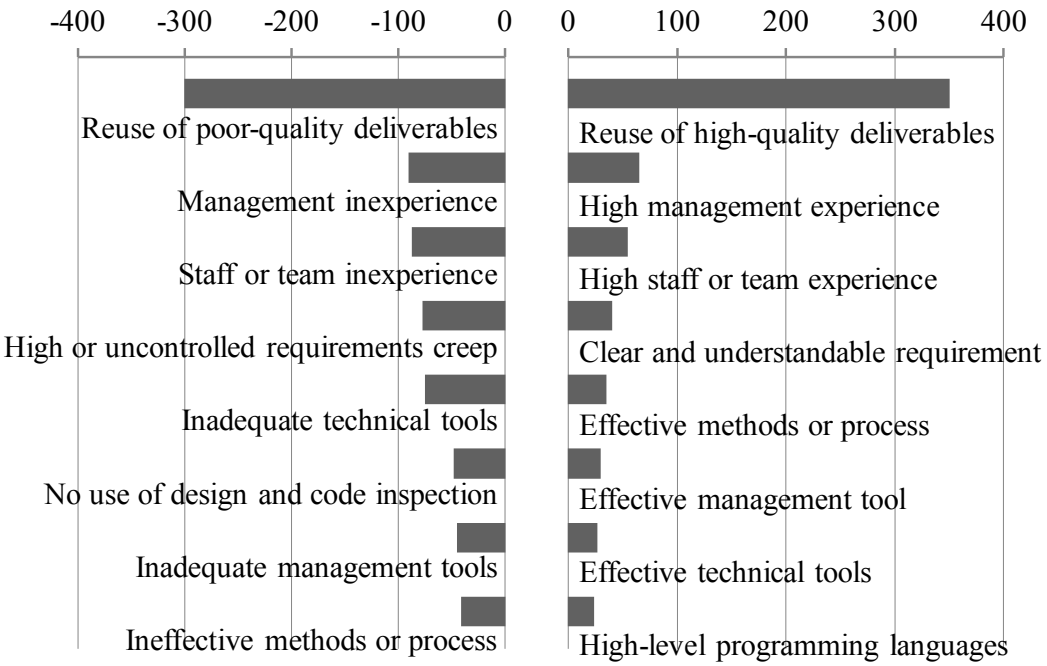


図 2-7　開発生産性に影響を与える主な要因

Figure 2-7　The main factors affecting development productivity.

最も影響度の大きい再利用には 1) ERP をはじめとする業務パッケージや商用ライブラリなどの COTS の採用，2) SPL(Software Product Line)の実現などに代表される既存資産の整理と再構成による自社の共通機能のドキュメント，ライブラリやフレームワークの整備, 3) 同様の機能を有す他サイトの成果物流用などがまず考えられるが, 4) モデルベース開発を適用し，既存のモデル流用やコード自動生成による高品質なコードの入手も再利用に加えることができる．

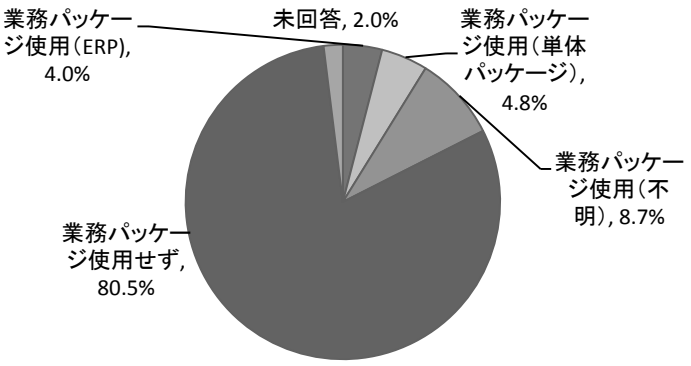


図 2-8　COTS(商用パッケージ)の採用状況

Figure 2-8　Adoption rate of COTS.

日本においても 1)のパッケージの採用は決して珍しいことではないが，スクラッチ開発の例が多いことが証明しているようにそれら単独では多くの要求を満たしていない[17].

2)の SPL はエンタプライズ系の開発においても古くから類似の取組が繰り返し取り行われてきたが，組織として技術力と経営が安定した IT 部門を抱える企業でなければ継続は困難である．そのため日本においては，IT ベンダが提供するドメイン色の薄い共通機能的なライブラリやフレームワークを採用する例が多く，古くから一般化している．一方でドメイン特化の機能を SPL 化する試みは現在もあまり成果を収めていない.

3)は，同業種に直接の競合相手がひしめく日本では実現が非常に困難である.

4)のモデルベース開発を採用するケースにおいて，一度書いたモデルを継続的に成長させることで生産性を向上させる，つまりコード自動生成とモデリングを交互に繰り返してシステム開発を行う反復型開発は 2)の SPL 適用で課題としたドメイン特化の機能を再利用する手段として位置づけられる.

以上の分析にもとづき，モデルベース開発ツールに反復型開発機能の実現が重要であると考えた．これにより開発生産性に最大の影響を及ぼす再利用の適用範囲を拡大できる.

2. 3　ソフトウェア規模見積の問題

2.3.1　ソフトウェア開発プロジェクトの大きな課題

大規模なソフトウェア開発を含むシステム開発プロジェクトにおいては，常にソフトウェア開発作業の見積が大きな課題である．第 1 章で述べたように新規にソフトウェア開発を行う場合，開発作業量に最もインパクトを与えるパラメタはソフトウェアの開発規模であり，一般に FP と LOC がそれらの予測に利用される．

図 2-9[18]に示すように，FP，LOC 共にシステム構想(Concept)を練る段階やプロジェクト計画(Planning)のようなプロジェクトの初期段階では利用できない．新規ソフトウェアの開発は，プロジェクト当初から大きなリスクを抱えていると言える．そのため，近年では発注者となるユーザ企業と開発を請負うベンダ間で，段階的に見積を詳細化し，合意しながら進めることが一般的になりつつある．

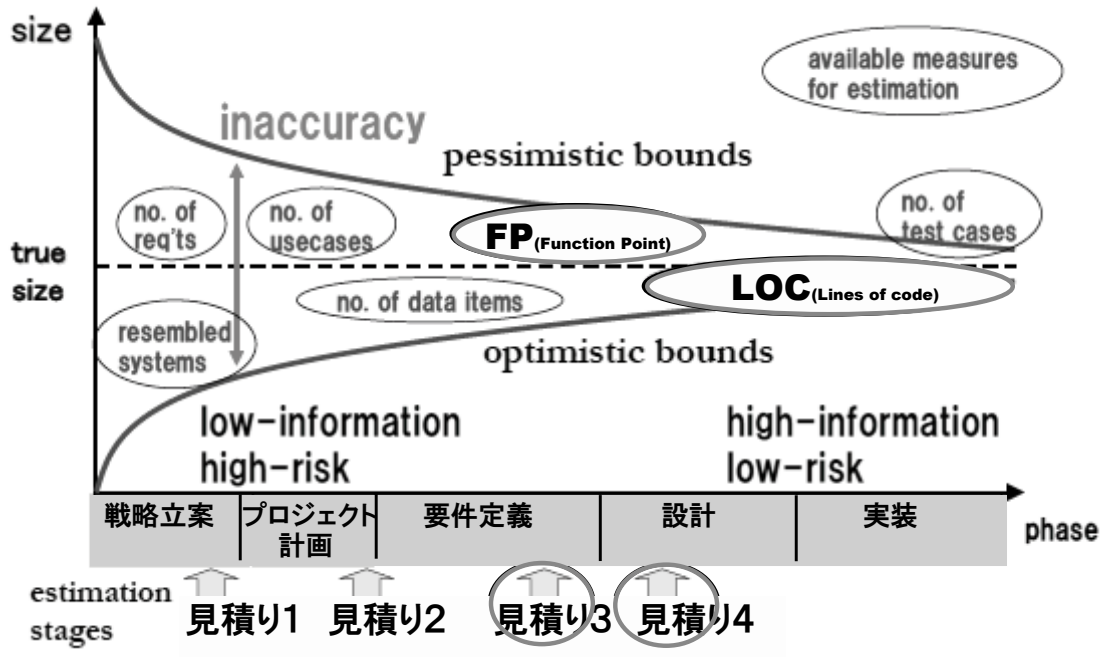


図 2-9　見積の段階とリスク

Figure 2-9　Software size estimation stages and related risks.

Drawn based on Barry Boehm : “Software Engineering Economics”, Prentice-Hall

FP の計測が可能となるのは，早くとも要件定義プロセス(Requirements Definition Process)の後半(図 2-9 の 3rd)からで，この時期は簡易手法を利用する必要がある[19]．エンタプライズシステムで最も利用される FP 計測手法である IFPUG 法を利用する場合は，外部仕様を明確化する設計プロセス(Design Process) まで(図 2-9 の 4th)待つ必要がある．

LOC が計測可能となるのは実装プロセス(Implementation Process)終了時であり，それ以前に LOC を得る場合は何らかの換算手段が必要となる．

2.3.2 LOC と FP それぞれの特徴

段階的見積の導入によりリスクは徐々に低減可能となるが，ソフトウェア開発の最終成果物はソフトウェアであり，最終的には規模を表現するにあたり FP か LOC，あるいは双方を選択することになる．

図 2-10 で 2 つの指標を比較する．

FP はソフトウェアの設計段階から利用可能で客観性のある指標であるのに対し，LOC はソースコードから直接計測可能で計測コストが安い．また古くから使われているため，パートナーシップの確立しているユーザ，ベンダ間は共通認識があり，特に日本では基軸通貨的位置づけにある．

FP はエンタプライズ系アプリケーションの計測向けに ISO/IEC 20926 として IFPAG 法が規格化されており，計測の専門家を養成することで客観的にソフトウェア規模を計測できる．

IFPUG 法はシステムの外部仕様を利用してソフトウェアの規模を算出する．LOC がソースコードから自動計測が可能なことに比すると，人手が多くかかり，計測にコストを要す．もちろん FP を自動計測可能なように外部仕様を形式的に記述するという仕掛けは実現可能だろうが，外部仕様書としての実用性も必要とされ，加えて外部仕様を実現する手段は近年，従来の HTML 中心の Web 画面や PC クライアントに加え，RIA，スマートフォン，Web インタフェースなど多様性があり，広く活用される形式を提案するのは，それ自体が大きなチャレンジになるのではないかと思われる．

	FP (Function Point)	LOC (Lines of code)
単位	FP（ポイント）	ソースコード行数 （物理または論理）
入力情報	外部仕様書と データエンティティ 設計書	ソースコード
計測コスト	高価	安価
計測難度	専門性あり （訓練された専門家が必要）	容易
客観性 （ソフトウェア 規模として）	高い (ISO/IEC 20926 IFPUG)	低い

図 2-10　2 つの方法の比較（FP と LOC）

Figure 2-10　Comparison of two methods.

LOC が客観性に欠け, またユーザに対して提供する機能量を示すメジャーとして不適切なのは以下のような要因があるからである[20].

(a)　プログラム作成者の能力差，嗜好により行数に差がでる.

(b)　同じ機能を実装する場合においてもプログラミング言語ごとに要する行数が異なる. 言語間で比較すると同等機能を実装して行数の少ない言語のほうが，一般に生産性は高いというパラドックスを抱えている.

(c)　(b)の派生要因となるが，ビジュアル言語など直接 LOC では計測困難な部位を持つ言語が存在する.

(d)　同じく(b)の派生要因であるが, ライブラリやフレームワークを利用することで大きく行数は変わる.

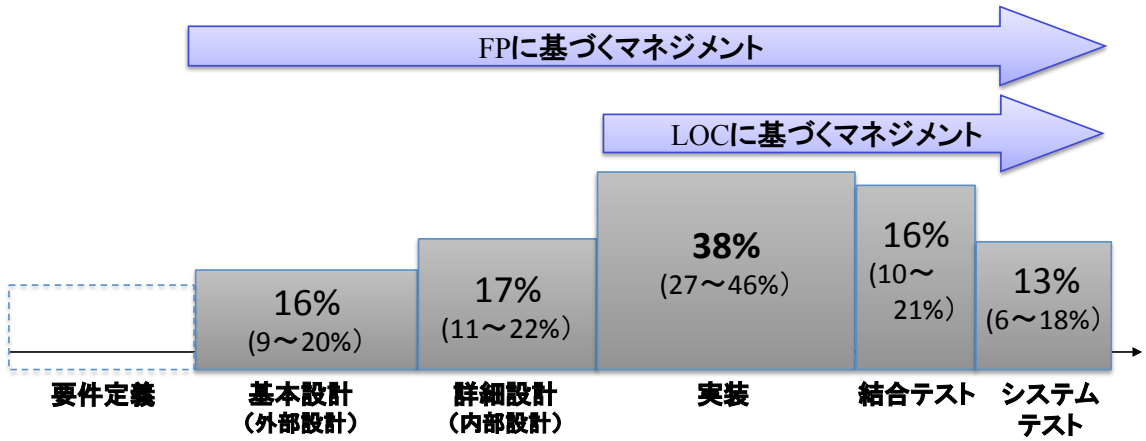


図 2-11　ソフトウェア開発における各工程の工数比

Figure 2-11　Phase-based actual effort ratio (Software Development).

以上のような問題があるものの, 現在でも LOC が用いられる大きな理由は, ソフトウェア開発において非常に大きな工数を要する実装プロセス(Implementation Process, およそ全体開発工数の 27%~46%を占める[21])及び以降のテストプロセスにおいて, 計測が誰にでも容易で, プログラムの開発進捗と品質を管理する基本単位として有用だからである.

2.3.3　2 つの指標を利用したプロジェクトマネジメント手法

ここまで述べた通り, FP と LOC は一長一短がある. 仮に 2 つの指標を使い分け可能と考えればどうだろうか. FP は設計の初期段階から使え, 客観性が高いため, ユーザ, ベンダ間の合意を得るのに向いている. 一方 LOC はソースコードとして成果物が見えてくると効果を発揮する. プロジェクトの前半を FP で, 後半を LOC 中心でプロジェクトをマネジメント出来れば, プロジェクトの進捗の見通しは非常に良いものとなる.

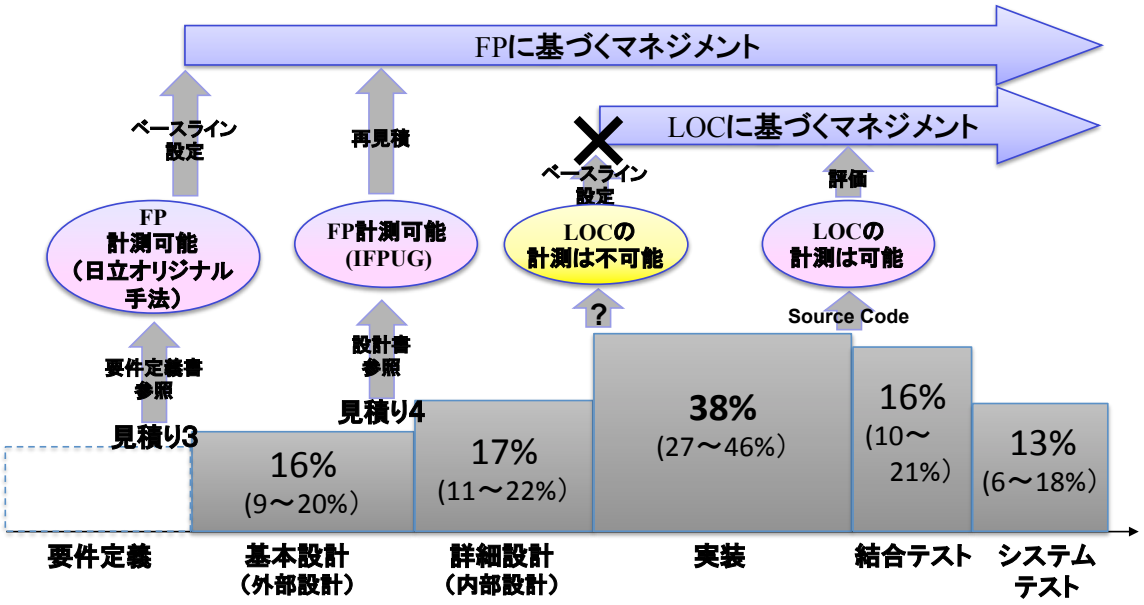


図 2-12　LOC 適用のタイミング

Figure 2-12　Timing of LOC turning on.

FP は計測に人手によるコストがかかるので，プロジェクト後半で LOC に乗り換え可能なことは管理コスト面でもメリットがある．

LOC は，実装プロセスの最初ではまだ計測不能(Can't measure LOC)であり，実装プロセス終盤にならないと計測できない．しかし，実装プロセスは最大の工数を要すプロセスであり，ソースコードを生産する工程であることから，LOC の見積値をもってプロジェクトの進捗と品質を管理するのが理想である．

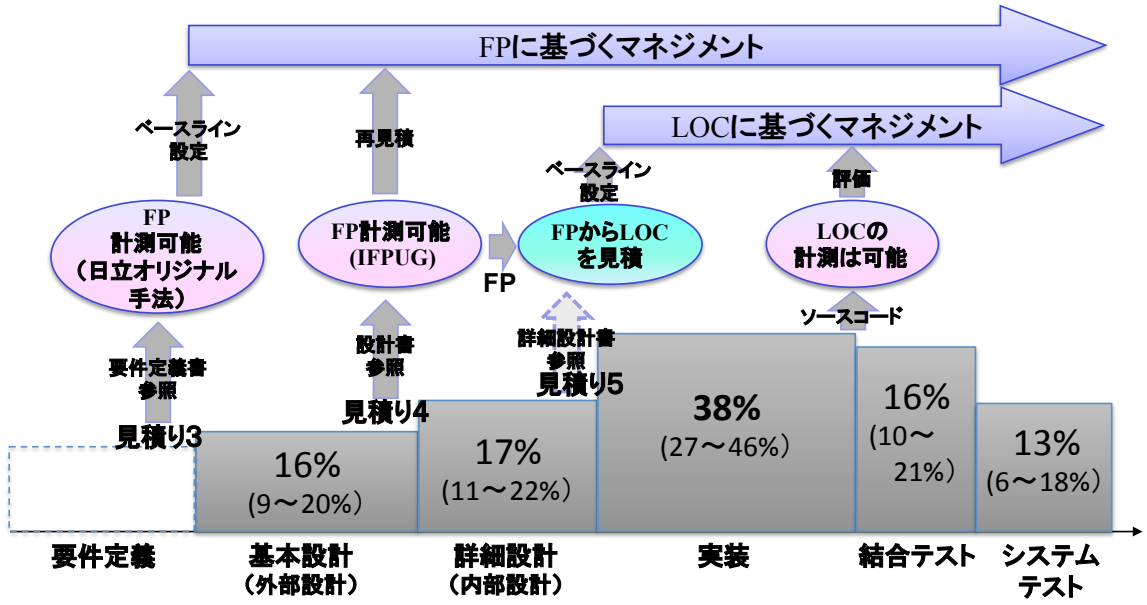


図 2-13　FP から LOC への変換

Figure 2-13　Conversion from FP to LOC.

そこで，図 2-13 に示すように，基本設計(外部設計：External design)で計測された FP を利用し，詳細設計(内部設計：Internal design)を参考に補足を加え(Reference to design documents)，LOC が算出出来れば良い.

2.3.4　精度良く LOC を導出する上での問題点

FP から LOC への変換は米 SPR(Software Productivity Research)が公開している換算表を利用すれば容易に算出できるが，この値は以下のようなアーキテクチャを内包する大規模システム開発ではそのまま利用できない.

(a)　一つの機能を実現するのに複数のプログラミング言語が利用される.

(b)　2.3.2 に述べたように，変換先となる LOC は客観性に欠ける尺度である.

(c)　近年においては人とのインタフェースとなる機器や CPU，ネットワークの性能が向上し，メモリ容量の制約緩和などからこれまで FP(IFPAG 法)で想定していた外部要因の制約を超えるプログラムが容易に作成可能で，また実用的でもある.

しかし，前述したようにLOCはプロジェクトの後半で必ず利用される指標である．プロジェクトの初期段階ではFPによる規模計測を行うが，ソースコードを管理対象とする工程からは，現物から直接計測可能なLOCがマネジメント上優れていると考え，FPからLOCへの変換精度向上に取り組んだ．

2．4　本研究で解決すべき課題

この章では，1．2で述べた本論文の3つの主題である（1）マルチアーキテクチャに対応出来るアプリケーション構造の提案，(2) モデルベース開発と反復型開発を実現する開発ツールの実現，(3) モデルベース開発がもたらすソフトウェア見積の精度向上の3つのテーマについて，それぞれに対応する形で現状の問題点を分析した．以降の章でこれらの問題解決に取組むための本研究が取組む課題を明確にする．

2．1では，現在開発されているエンタプライズシステムが，マルチアーキテクチャで構成されていることをデータで示した．特に大規模エンタプライズシステムの開発に利用されるツールを提案するには，マルチアーキテクチャへの対応が必須である．

そこで第3章ではマルチアーキテクチャへの対応として，長期利用されるエンタプライズシステムの特徴を考慮し，フィールドで多く採用される処理形態，システムアーキテクチャ，プログラミング言語の組合せで示す8つの基本的なシステム構成を定義する．これら8つのシステム構成への対応可能な開発ツールがマルチアーキテクチャへの実現である．

次に多くの技術者が理解しやすいP層，F層，D層と命名した単純な3層のソフトウェア構造を提案し，この3層に分けたソフトウェア構造で8つのシステム構成を実現する方法を述べる．

2．2では，生産性に最も寄与する施策が再利用であることから，再利用を促進するモデルベース開発と，再利用資産を継続的に成長させる反復型開発の組合せが有効であると主張した．

そこで第4章で多くの技術者が習得しやすく対峙するユーザにも理解しやすい仕様記述のレベルと，目的とするソフトウェア構造を自動生成することが可能な仕様記述のレベルを調整して仕様を記述するモデルを選定し，次にモデルベース開発と反復型開発が共存で

きる開発プロセスとツールの関係を明確にして体系化し，さらにツールが生成すべきソフトウェア機能について明らかにする.

2. 3 では，特に大規模なエンタプライズシステム開発の大きな課題として，ソフトウェア規模の見積問題があり，それぞれ特長と適用上の課題がある FP と LOC を組合せて利用することがソフトウェア開発プロジェクトのマネジメントに寄与することを述べた.

そこで第 5 章では，私の提案するモデルベース開発ツールが，ソフトウェア規模の見積精度向上に寄与するとの仮説を掲げ，実際に変換精度の向上に取組んだ結果について報告する.

第 6 章では, これまで述べた(1)から(3)までの研究成果を実現したツールの具体的操作を示し，第 7 章では，これまで適用してきたシステム開発におけるツールの実績と成果を，さまざまなシステム構成に対する適応性，長期に渡る反復型開発の実践，生産性，品質の観点から述べる.

第 3 章　マルチアーキテクチャへ対応可能なソフトウェア構造の提案

2. 1 で述べたように現在のエンタプライズシステムはマルチアーキテクチャで構成されており，開発ツールは，このマルチアーキテクチャに対応できなければ特に大規模な基幹システムの開発には採用されない.

この章では，マルチアーキテクチャに対応するツールを実現する為に，現在も採用され続けているアーキテクチャを全てサポートできるアプリケーションのソフトウェア構造を提案する.

3. 1 では，2. 1 で示した 3 つの処理形態，5 つのシステムアーキテクチャを直交する概念と捉え，これがマルチアーキテクチャのパターン全てと定義する．それぞれについて如何なるケースで採用される技術であるかを明確化し，代表的な実装方法を述べる．また実装時に複数の技術的選択肢がある部分については，それぞれの技術を吟味し，どの技術を本ツールが採択するか明確にする.

3. 2 では，アプリケーションソフトウェアの構造として，私が提唱する 3 層 3 種のソフトウェア・コンポーネントである P 層，F 層，D 層を定義し，境界を明確に示す.

3. 3 では，3 つの処理形態，5 つのシステムアーキテクチャの直交点において，P 層，F 層，D 層の各ソフトウェア・コンポーネントがどのように配置されて，機能するかを述べ，マルチアーキテクチャのパターン全てに対応出来ることを示す.

3. 1　エンタプライズシステムでカバーすべきアーキテクチャ

現在，エンタプライズシステムで一般的にみられる処理形態，システムアーキテクチャは 2. 1 で示したように，処理形態においては 1）バッチ処理，2）対話処理，3）トランザクション処理の 3 つに，システムアーキテクチャは a）メインフレーム，b）クライアント・サーバ，c）Web，d）スタンドアロン，e）SOA となっている.

SOA を除けば，処理形態，システムアーキテクチャ共に，2000 年代当初も現在も大きく顔ぶれに差はない．メインフレームはここ 10 年で大きく利用形態が広がってきているので，後で扱いを明らかにする．またデータベースについては，当時も今日も主流である

RDBMS の利用を前提にすすめる.

マルチアーキテクチャをサポートする場合，ここに示された処理形態，システムアーキテクチャをカバー可能なアプリケーション構造を提案する必要がある.

この章では，3 つの処理形態，5 つのシステムアーキテクチャを 3×5 の直交する概念と捉え，それぞれについて如何なる利用場面で採用される技術であるかを明確化し，代表的な実装方法を述べる．また実装時に複数の技術的選択肢がある部分については，それぞれの技術を吟味し，どの技術を本ツールが採択するか明確にする.

3.1.1　処理形態

(1) バッチ処理

バッチ処理は，基本処理形態として 1 台のサーバ（スタンドアロン）上で実行される.

応用形態として同一のプログラムを，入力データを分散配置することで複数プロセス，あるいは複数サーバ上で分散並列実行させることもあるが，プログラムとしては 1 本である.

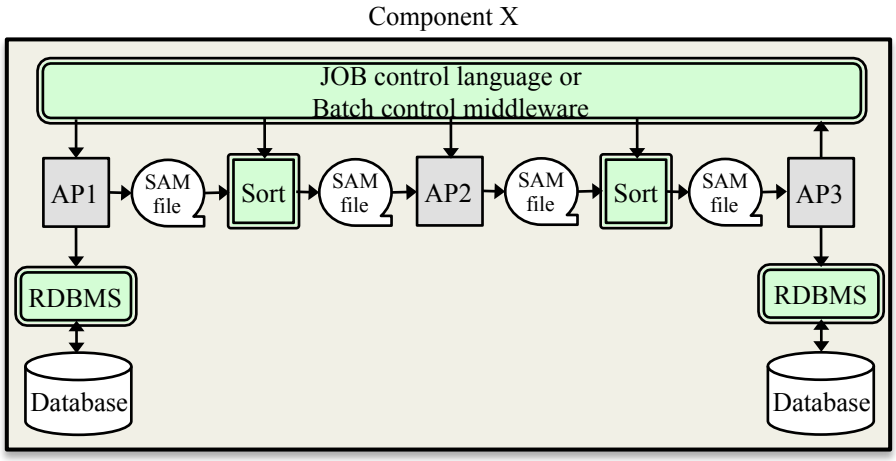


図 3-1　バッチ処理の基本形態

Figure 3-1　Basic model of the batch processing .

また生産性，保守性など品質向上を目的に，直列実行される一連の処理の流れを適切にプログラム分割することで, プログラム各々のロジックをシンプルにして見通しを良くし，実行時は複数のプログラムをジョブ制御言語等で連結させて実行させる形態も一般的であ

る（図 3-1）.

複数のバッチ処理を同時に実行させる場合には CPU やメモリ，ディスク容量などリソース割当ての最適化のためにリソース管理とジョブスケジュール機能を持つミドルウェアを導入する例も多い.

バッチ処理は最も古い処理形態であり，メインフレーム時代からの技術蓄積も多く，本ツールでもそれらノウハウの大部分を踏襲している[22]．バッチ処理は単位時間あたりの処理件数を最大化する技術であるため，データ格納媒体は入力も出力も極力シーケンシャルファイルとなるようにデザインする．シーケンシャルファイルは一般に外部媒体への記録方式としては単位時間当たりの読込みデータ量も書込みデータ量も最大である.

バッチ処理ではこのシーケンシャルファイルの性能を最大限に活かすため，アプリケーション・プログラムが処理しやすい順序にソート処理を適宜挟んでシーケンシャルファイルを作り次のプログラムに送り，結果をまたその次のプログラムが処理しやすいようにソートを挟む．これを繰り返す形でバッチ処理全体を構成していくことがスループットを最大化するポイントである．性能を最大化するために RDB をはじめとするランダムアクセスを要するデータへのアクセスは必要最低限に留めるのもノウハウである.

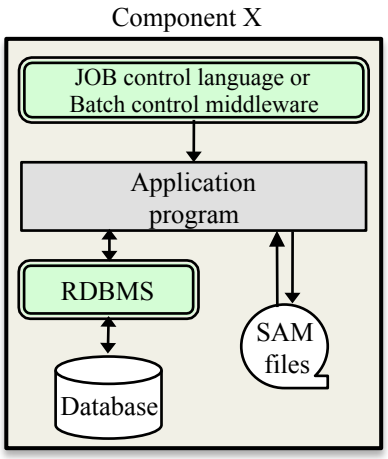


図 3-2　非同期実行向けアプリケーションテンプレート（基本構成 A）

Figure 3-2　Application template for asynchronous execution.
(Basic configuration A)

従来技術から加えるべきアプリケーション実装上のポイントは RDBMS へのアクセス機能の充実である.

近年，スループットの最大化を狙うよりは，非同期処理の手段としてバッチ処理を利用

するケースも多い. 時間のかかる処理で画面の前のユーザを長い時間待たせるのではなく，バックグラウンドに回し，複雑で時間のかかる処理をこなす．このような処理は対話処理やオンライン処理と同等，あるいはそれ以上に複雑なデータベースアクセスを含む．またこの処理を複数件のトランザクション分，一度に処理するケース（基本構成 A）も多い．

対話処理やトランザクション処理から直接バッチ処理を非同期呼び出しする手もあるが，データの引き渡しや起動ログの取得，多重度の制御などを行うため，MessageQueue やファイル転送製品を使ってデータを渡す延長で起動させるケースが多くみられる．データベースにデータとキューを貯めて自動起動させる部品を自作するケースも多くみられる．

（2）対話処理とトランザクション処理

文献[15]では，2）対話処理と 3）トランザクション処理の差は，アプリ実装時にトランザクション制御の有無で分類されているが，マルチユーザで利用するシステムは，ユーザ間の処理が混信しないようにトランザクション制御を実装することが常識的であるため，ここでは OLTP モニタ等トランザクション制御ミドルの有無によるアプリケーションソフトウェア実装の差異として捉える．

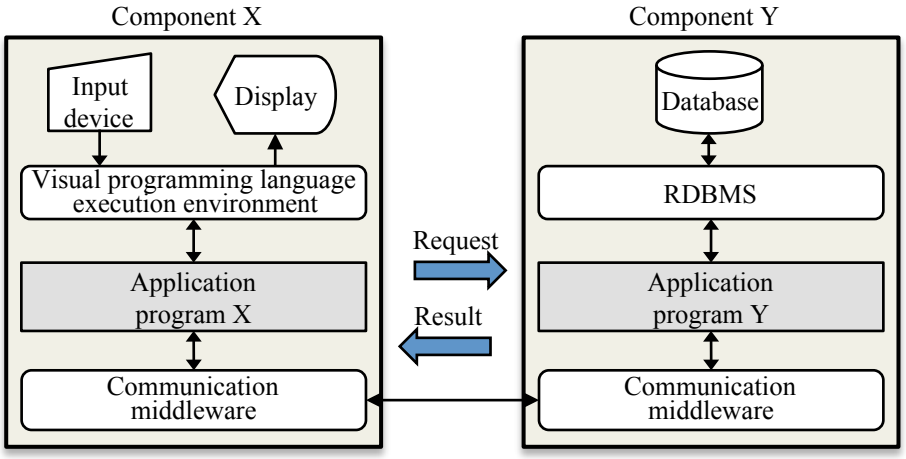


図 3-3　分散アプリケーション処理の基本モデル

Figure 3-3　Basic model of distributed application processing.

対話処理やトランザクション処理を実装する場合は，少なくとも 2 つ以上の機器に股がって実装されるアプリケーション間で処理分担を行う必要がある．課題は，機器間のアプリケーションの機能分担，トランザクション連鎖の範囲，通信プロトコルの 3 点に整理

できる.

以下，利用者サイドに近い機器を X(component X)，データベースが接続されている機器を Y (component Y)として実装ケースを考える.

3.1.2 機器間の機能分担

2 つの機器間で実装するアプリケーション機能の役割分担のバリエーションを述べる. パターンとしては，フロントヘビーな役割分担として（a‐1）X に機能が偏重する分散アプリケーション，バックエンドヘビーな役割分担として（a‐2）Y に機能が偏重する分散アプリケーションと，その中間のケースとして（a‐3）X と Y で役割分担する分散アプリケーションの 3 つのパターンが想定される[23].

（a‐1）X に機能が偏重する分散アプリケーション

X に機能が偏重するアプリケーションの典型的な例は，クライアント・サーバ型データベースを利用した通称 2 層型クライアント・サーバと呼ばれるシステム構成である.

X と Y に流れるデータは SQL(データベース操作言語)とその結果となる.

データベースに問合せが発生した場合だけ X から Y へ処理依頼が飛ぶ. それ以外は X 内で処理が完結する. しかし，エンタプライズシステムでは，データベースへの問合せは多く発生する. LAN による接続が一般的になり，また LAN の性能を十分活かせるまで機器の性能(主に CPU，メモリ，バス等の性能と通信ソフトウェア)が向上して実現が可能となった処理形態である.

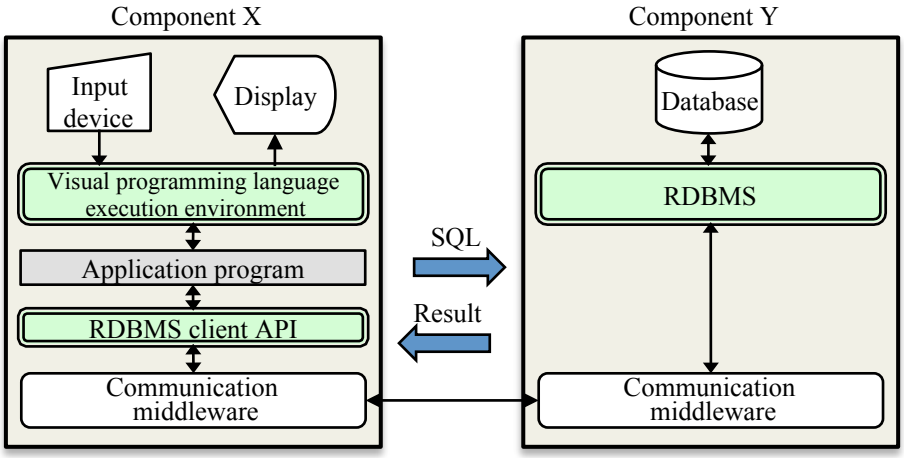


図 3-4　X に機能が偏重する分散アプリケーション（基本構成 B）

Figure 3-4　Distributed application that function overemphasizes to X. (Basic configuration B)

90 年代前半では，パフォーマンスの観点から一台の Y でまかなえる X の台数は数十台が限界であった．しかし，サーバの性能向上と LAN の性能も同期して向上し，90 年代後半には 100 台を超える X を接続可能となった[24].

また複数サーバを LAN で高速につないでデータベースへのアクセス負荷が分散可能なほどに性能が向上した．データベース処理をフロントとバックエンドで分担して行うクライアント・サーバ型のデータベース，またバックエンドのサーバを複数台にしてアクセスをパラレル化することにより，大量データアクセスの負荷分散や，相互アクティブ型の HA (High Availability)構成による信頼性向上などが可能になってきた. このような構成もこのころより一般に利用されるようになった.

このように（a - 1）のパターンは現在,

- 2 層型クライアント・サーバパターン（基本構成 B）
- 負荷分散を意識したデータベースサーバのクライアント・サーバ構成

として利用されている.

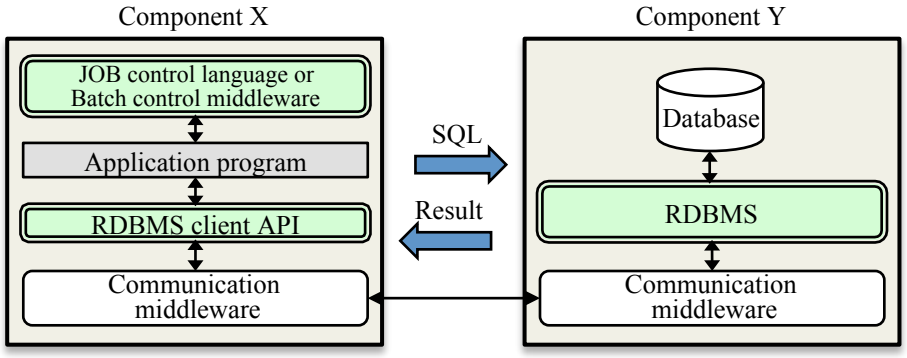


図 3-5　クライアント・サーバ型データベースを利用したバッチ処理（基本構成 C）

Figure 3-5　Batch processing using client server type data base.
(Basic configuration C)

2 層型クライアント・サーバはバッチ処理でも一般に利用されている（基本構成 C）.

（a - 2）Y に機能が偏重する分散アプリケーション

Y に機能が偏重する分散アプリケーションは X 側に特徴がある．典型的な例は過去であれば端末エミュレータや X クライアント，近年であれば Web ブラウザが該当する．また画面のドットイメージを送るという意味では前者 2 つとは方式が異なるもののシンクライアントや VDI(Virtual Display Infrastructure)もここに類型できる.

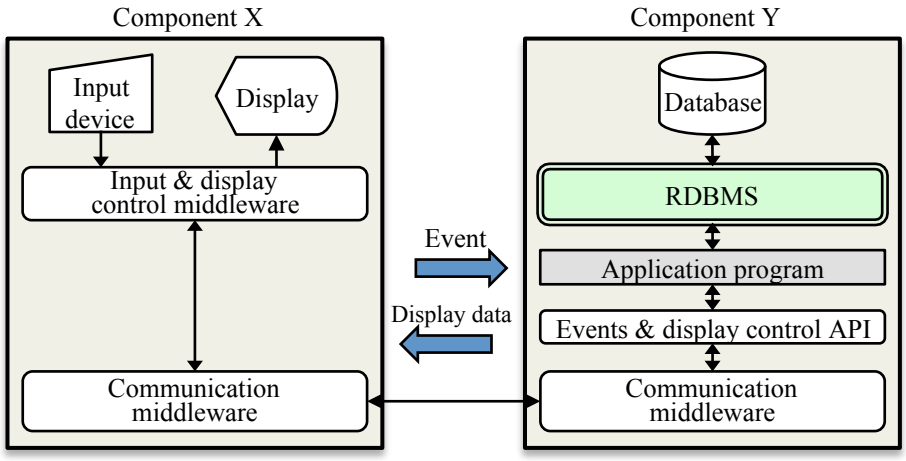


図 3-6　Y に機能が偏重する分散アプリケーション

Figure 3-6　Distributed application that function overemphasizes to Y.

端末エミュレータも Web ブラウザもプログラマブルな切り口を持ち，画面との入出力

処理の延長上にカスタムな処理を仕込めるが，Y との通信手段はイベント通知と画面データの受信に限られる．

Web ブラウザ，シンクライアント，VDI などは今現在広く利用されている技術だが，端末エミュエータも使われ続けている．端末エミュエータを利用したメインフレームを中核としたシステム構成（基本構成 D）を示す．メインフレームによる集中処理の場合には，（a-3）にみられる OLTP モニタがコンポーネント Y（メインフレーム）に実装され，アプリケーションの制御（起動）を行う．

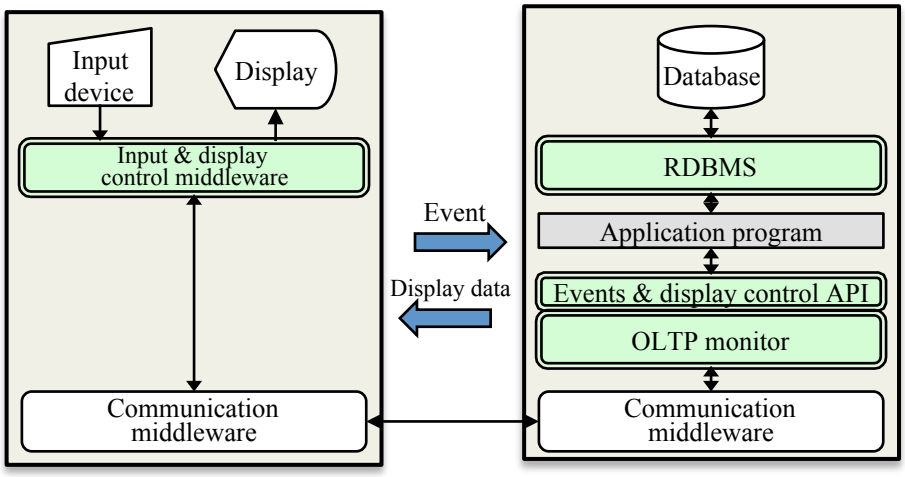


図 3-7　過去によく見られたメインフレームのオンライン処理構成（基本構成 D）

Figure 3-7　On-line processing architecture of mainframe seen well in the past. (Basic configuration D)

（a-2）に分類されたこれらの技術は，利用者サイドの実装を軽くする処理形態として，セキュリティの確保やメンテナンス性の向上などの利点から，今後も姿，形は変化していくだろうが，広く利用され続けると考える．

（a-3）X と Y で役割分担する分散アプリケーション

X と Y で役割分担してアプリケーションを実装するケースとしては，X に Visual な開発環境を持つ言語を利用し，Y に OTLP モニタ等のトランザクションヘビーな処理に耐えうる環境を持つ場合が典型例である．

X と Y で役割分担する上でもっとも重要なのは，X から Y へ渡すリクエストの粒度をど

う決めるかということである.

少なくとも 2000 年代の前半までは機器間をつなぐ公衆ネットワークのパフォーマンスが低く，大規模システムにおいて，（a-1）のようにデータベースへのアクセスの度にリクエストを渡す方式や，（a-2）のように，画面を操作する度にXとY間で通信が発生する方式ではリッチなデータを送り合うと処理が集中するサーバも心配だが，それ以前にネットワークがパンクした.

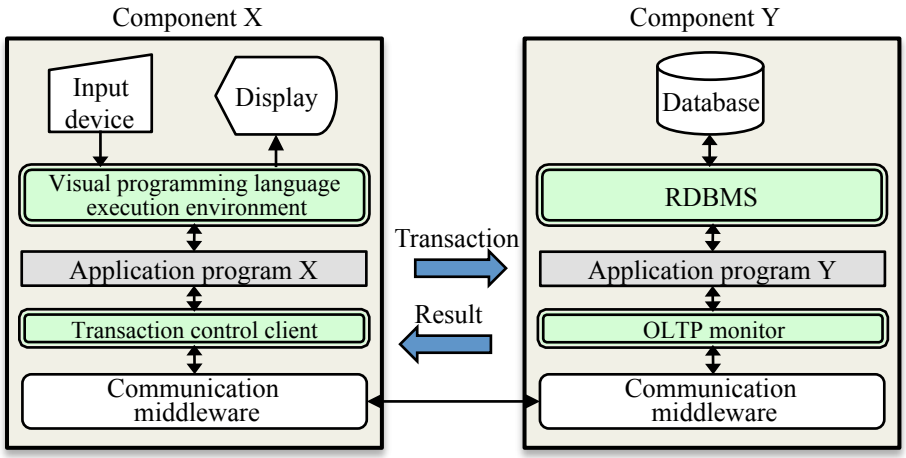


図 3-8　X と Y で機能分担する分散アプリケーション

Figure 3-8　Distributed application to which role is allotted by X and Y.

しかし，90 年代後半になるとルーティング技術，続いてスイッチング技術が発展し，トラフィックを分散させて全体のスループットを向上させることが可能となり，2000 年代に入ると公衆網でも高速な LAN 間接続が次第に安価に提供されるようになり，加えて LAN 自身の性能も向上してきた.

（a-2）の典型的な構成例となるシンクライアントでは，今日企業内で数万台を接続するケースはざらにあり，負荷の集中する Y 側でもデータベースを利用するアプリケーションでなければ負荷を分散させる手立ては多い.

今日（a-1）のモデルで大規模なシステムを作る例はないが，Y 側に巨大なサーバを導入することがコスト的に可能であれば，数千台の X を接続可能な大規模システムを，性能要件を満たすよう構築することは技術的に十分可能である.

（a-3）のモデルを利用する価値は，現在主に大規模システム向けであり，コスト的に

許容可能なサイズの機器Yで大量のトランザクションを処理する必要性から生じている.

当然ながらXとYとの通信回数, 通信データ量共に少ない方がコストパフォーマンスの観点からもレスポンスの観点からも優れている.

（a-Mix）3つのモデルの組合せ

これまで紹介した3つのモデルは，それぞれ単独でもシステムとして成立するが，組合せも可能である.

（a-1）は先にも述べたように，いまや典型的なデータベースのスケールアウト手段でもあるので如何なるモデルとも組合せ可能である．たとえば,

（a-1）＋（a-2）は，フロントをWebブラウザと仮定すると旧来のCGI，PerlやPHP，JSPを使った2層Webモデルが典型例である（基本構成E）.

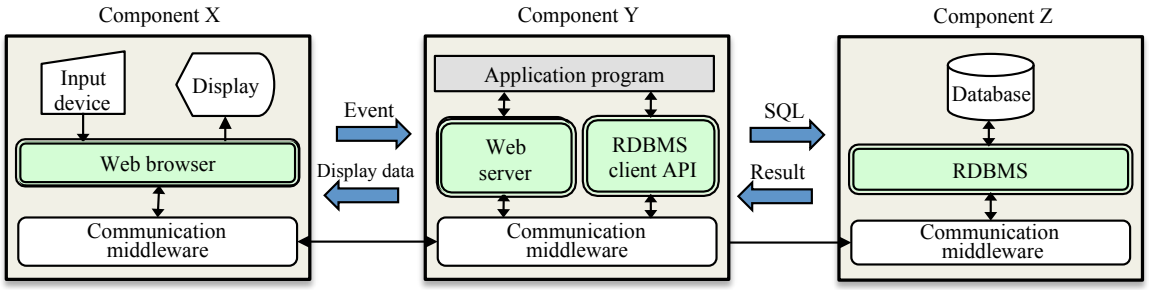


図3-9　小規模なWebシステム構成（基本構成E）

Figure 3-9　Small-scale Web system architecture.
(Basic configuration E)

（a-1）＋（a-3）は，典型的な3層クライアント・サーバモデルである（基本構成F）.

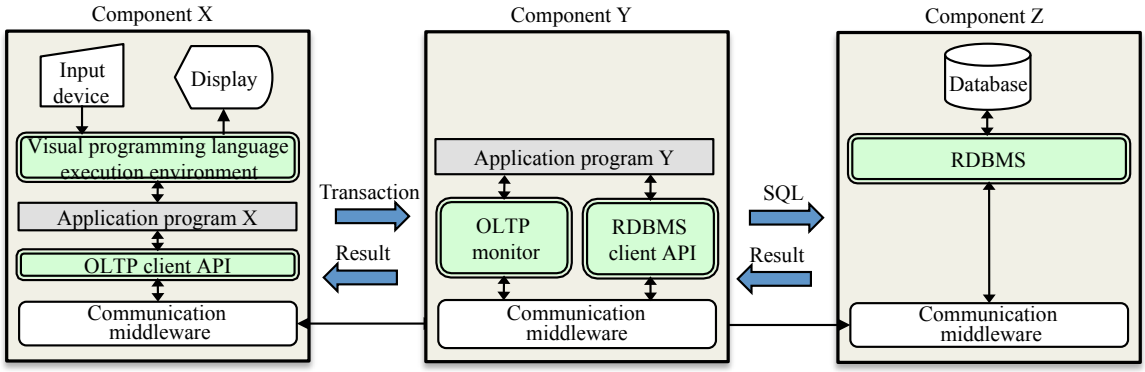


図 3-10　3 層クライアント・サーバシステム構成（基本構成 F）

Figure 3-10　Three layer client server system architecture.

(Basic configuration F)

（a-2）＋（a-3）は，端末エミュレータと OLTP を使った 3 層クライアント・サーバのオンラインシステムが該当する（基本構成 G）が，近年は少数派と言える．

（a-1）＋（a-2）＋（a-3）とすると，フロントを Web ブラウザとみれば，典型的な Web3 層モデルとなる（基本構成 H）．

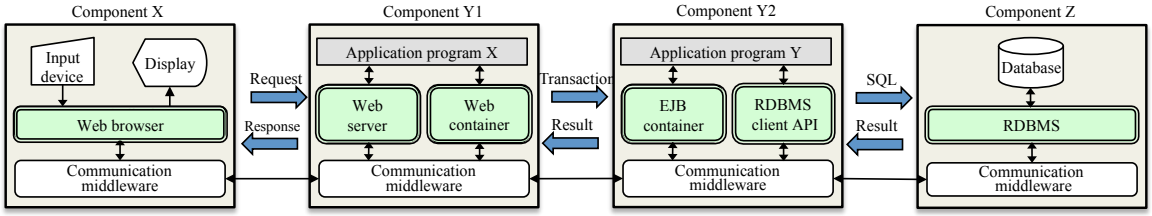


図 3-11　3 層 Web システム構成（基本構成 H）

Figure 3-11　Three layer Web system architecture.

(Basic configuration H)

3.1.3　トランザクション連携の方法と範囲

2 つの機器は，連携してトランザクション処理を完成させる必要があるが，どの範囲まで 1 つのトランザクションとして制御するのかを決める必要がある．複数の機器を股がってトランザクションを完成させるには分散トランザクションを利用することが一般的に知

られているが，分散トランザクションは機器をまたがって排他を取るため，機器間で排他制御の通信が煩雑になり，また排他時間も長くなることからパフォーマンスを大きく損なうリスクを持つ．また競合やデッドロック発生時は複数機器間にまたがって影響が伝搬するためペナルティも大きく，大規模システムを構築する場合においては，あらゆる場面で使える唯一解ではない．

分散トランザクションはプロトコルが標準化されており，異機種，異ミドルウェア間でも接続が可能である．エンタプライズシステムにおいて分散トランザクションはデータベースアクセスと同期してこそ存在意義があるが，実際には利用するデータベース製品によって，それらが管理するデータ領域，インデックス，バッファなどのリソース制御方式が異なり，複数のデータベースにまたがって適用しているシステムは決して多くない．

トランザクションとしてACID (Atomicity, Consistency, Isolation, Durabilityの略でトランザクションの保障すべき性質を指す)を成立させる方法と範囲の分類として（b-1）クライアント・サーバ型のデータベースを利用する方法と，（b-2）分散トランザクション機能を利用するタイプ，（b-3）分散トランザクションを利用せずACID属性を犠牲にして業務上支障がない方法でアプリケーションにて排他処理を実装する方式が存在する．

（b-1）クライアント・サーバ型データベースを利用する方法

トランザクションを保障する方法として，データベースの機能に依存するモデルである．

クライアントとサーバに分かれたデータベース内においては分散トランザクション技術を実現する製品もある．

トランザクションの開始をアプリケーション内で宣言し，コミット制御もアプリケーションがデータベースのAPIを利用して行なう．

トランザクション処理を実行する上で，データベース上で起こりうる大きな問題はロックの長期化による並列実行性の阻害と，デッドロックである．いずれも完全に解決された課題ではないが，データベースベンダの努力とフィールドでの使いこなしノウハウの積み上げにより，データベースが単一ベンダから提供され，Yが1台あるいは数台程度で構成されるケースにおいては，実用的なモデルといえる．一方，近年のWebサービスのように数百～数万のデータベースサーバを平行稼動させるモデルにおいては，ACIDにこだわった運用は不可能であり，代替手段と同時にアプリケーション，運用上で回避策を打つといった，実用的な実装方法を選んでいる．

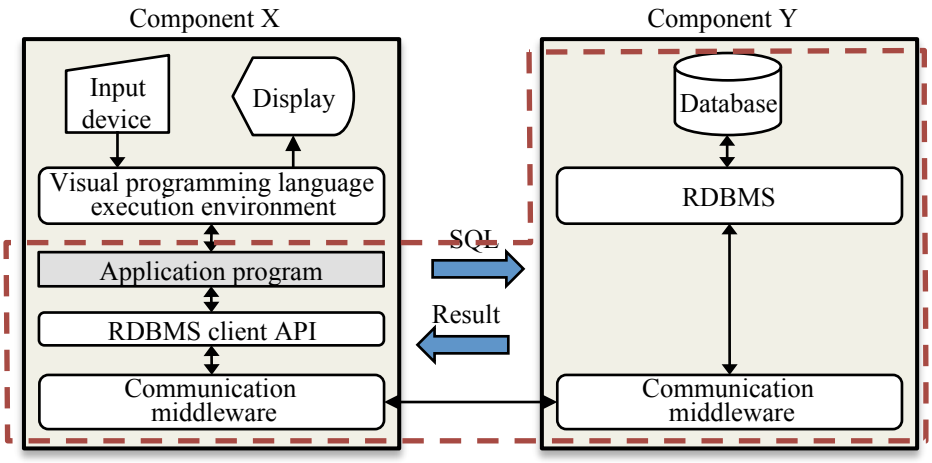


図 3-12　クライアント・サーバ型データベースを利用する分散アプリケーション

Figure 3-12　Distributed application that uses client server type database.

また，前述したようにデータベースベンダの努力とフィールドでの使いこなしノウハウの積み上げにより性能を維持しているのであり，X のアプリケーションから意識して長期間資源を占有すれば，たちどころに並列実行性は阻害されうるし，機器 X と Y 間の通信環境が粗悪であれば，特定の X からのリクエストが遅いことで全体のパフォーマンスを低下させる危険性もある．

リソースへの排他が長時間に及ぶ可能性がある場合は，（b-3）のアプリケーションでの排他処理との併用が望ましい．

（b-2）分散トランザクション機能を全面的に利用する方法

分散トランザクションは，機器やソフトウェア・コンポーネントを跨ってトランザクションを保障する機構である．

図 3-13 にあるように，X 上のアプリケーションから Y 上で動くアプリケーション，データベースまでを包含してトランザクションとして成立させる．

アプリケーションからみれば，トランザクションを実装する上では理想的な機能といえる．しかし，現実場面では，あらゆるトランザクション処理をこの機構で実装すると，前述したように，長期間資源を占有し，特にデータベースのデータに長期に排他をかけられると，並列実行性の低下や，最悪デッドロックの多発といった問題にもつながる．

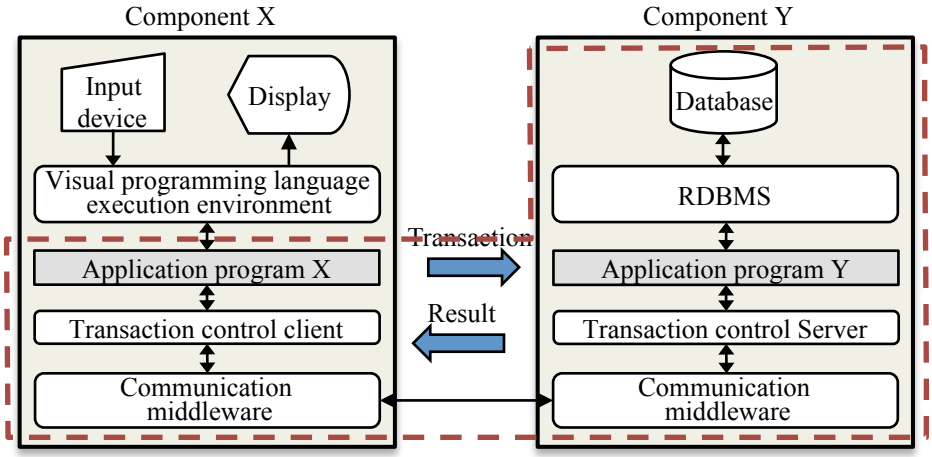


図 3-13　分散トランザクション機能を利用する分散アプリケーション

Figure 3-13　Distributed application that uses distributed transaction function.

（b-1）でも述べたが，機器 X と Y 間の通信環境が粗悪であれば，特定の X からのリクエストが遅いことで全体のパフォーマンスを低下させる危険性もある．

したがって，制約は（b-1）と同様で，X と Y 間の通信環境が良好で，また X と Y の数が数十台程度に限られる場合に限定して利用すべきである．

（b-3）機器間は分散トランザクションを利用せずアプリケーションで排他処理を実装するモデル

データベース資源の排他を含むトランザクションとして排他をかける範囲は Y の中に留めることで，並列実行性を向上させる実装モデルである．

OLTP モニタ等ミドルウェアの機構で排他をかける範囲は Y 内に留める方法を取るため，X から Y に送信する 1 通のトランザクションで完了しなければならず，意味のある範囲でアプリケーション X と Y の間で処理分担を行なう必要がある．

概念的には，Y で一度に実行される単位は，途中人間や外部システムとのインタラクションが入らない範囲で，しかも業務からみて意味ある一塊を単位とする．時間で言えば長くても数秒程度で処理が完了するものである．

それ以上の時間を要するものは，バッチなど非同期型の処理に回す．また途中人間とのインタラクションが発生し，処理完了までに長時間を要す可能性があるものは，アプリケーションで排他処理を実装する．この方法については 3.2.3 で述べる．

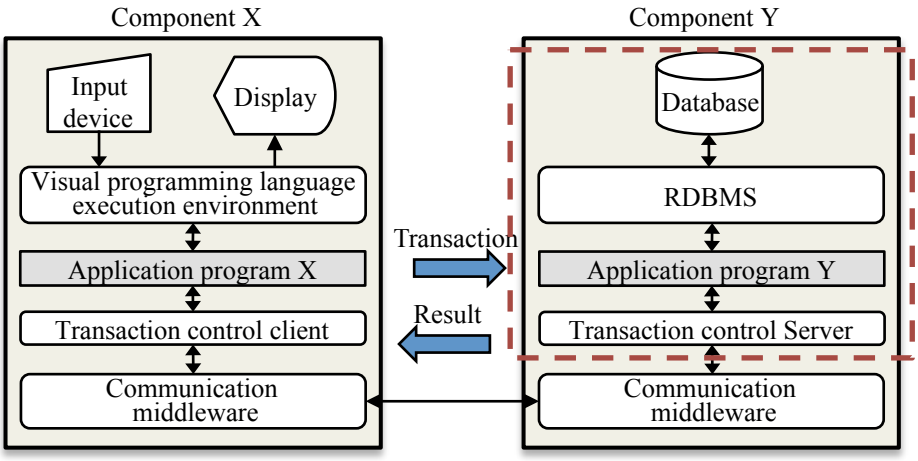


図 3-14　長時間の排他制御を自身で実装する分散アプリケーション

Figure 3-14　Distributed application that oneself implements exclusive control for a long time.

過去に大規模なエンタプライズシステムの開発において（b-1）や（b-2）の処理モデルを採用し，多くの苦労を重ねてきた経験から，私はこれまで（b-3）の手法を選択するケースが多い．また Internet 上で大量のサーバを並列分散して動作させて大規模な Web サービスを展開する場合には，（b-3）の処理モデルを選択することが今日一般的となっている[25].

3.1.4　通信プロトコル

今日，機器を股がって処理を行う場合，CORBA の ORB(Object Request Broker)がもたらした分散オブジェクト技術を採用することは一般的となった．以降，J2EE の RMI(Remote Method Invocation)，.NET の COM+や SOA で採用される SOAP など，ORB に類する技術が 90 年代後半から 2000 年代にかけて次々に登場したが，それら登場前の技術についても検討しておく必要がある.

分散オブジェクト呼出し以前のリアルタイム型通信方式は（c-1）RPC 型，（c-2）メッセージ転送型，（c-3）Peer to Peer の対話型に大きく分類出来る.

（c-1）RPC 型

RPC(Remote Procedure Call)は UNIX のネットワーク機能，分散コンピューティング

機能強化の中で発展してきた機能で，その後 OLTP モニタ製品で標準的に搭載されるようになる．

X 側のアプリケーションからは，Y 側のアプリケーションをモジュール呼出しの要領でアクセスすることが可能となる．プログラミングモデルとしては自然な形態である．

このモデルが後に分散オブジェクト呼出しに拡張されていく．

特長は同期呼出しである点であり，そのため相手が受信可能な状態で待機している必要がある．

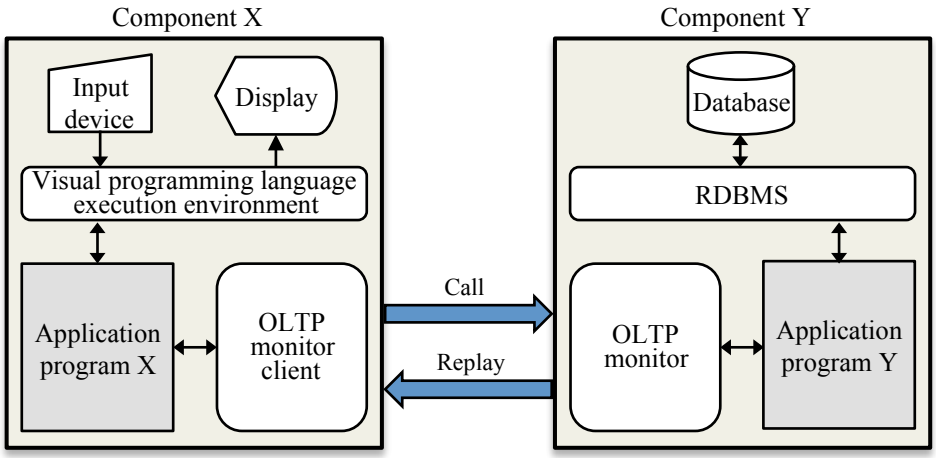


図 3-15　RPC を利用した分散アプリケーション

Figure 3-15　Distributed application using RPC.

通信が失敗した場合は機器 X 側で検知して再度実行するか決める必要がある．送信した Call メッセージをミドルでキューイングすることも考えられるが一般的ではなく，操作者，あるいは機器 X のアプリケーション A で再送する．

今日においても，オブジェクト指向言語以外の言語ではもっとも良く利用される連携手段である．

（c-2）メッセージ転送型

メッセージ転送型は，通信伝送系のアプリケーションでよく見られる連携方式である．基本は一方向通信であり，相手に向かって一方的にデータを送り込む．そのため同期通信型の RPC と比して一方向に対してではあるが高スループットを実現しやすい．

問合せ応答型のアプリケーションを実現したい場合には，送り手と受け手で別々の通信

路を確保して，送信と受信で通信路を使い分ける．特に上位アプリケーションで同期的な規約を設けなければ全二重通信としても成立する．

メッセージ転送を保証するために，ディスク等の外部媒体にキューを設ける方式も以前より良く知られた構成である．

問合せ応答型のアプリケーションからみると送信，受信別々の通信路を意識して実装を行なう必要がある．特に通信相手が多いと Y 側のアプリケーションは実装，管理が面倒である．用意する通信リソースも多くなる．

一方で非同期にデータをやり取りしたい場合は，RPC のように相手の待機を求めないため柔軟性がある．キューに溜めておけば，受信側のアプリケーションが必要とするタイミングで取得できる．この通信処理の延長上でバッチ処理を起動させるといった使い方も多くみられる．

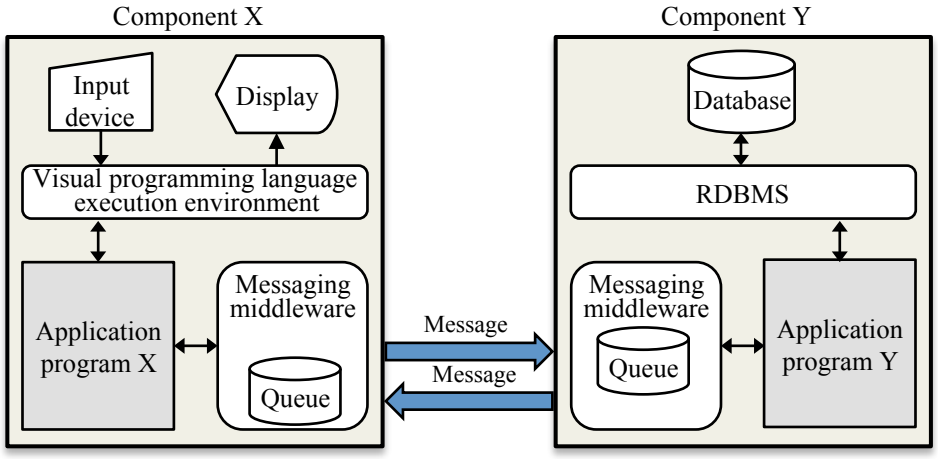


図 3-16　メッセージ転送を利用した分散アプリケーション

Figure 3-16　Distributed application using message transfer protocol.

（c－3）Peer to Peer 対話型

以前は，コンピュータにおいても通信においても，一方が親あるいは一次局となり，他方が子，あるいは二次局となって，親あるいは一次局が主となってコントールすることで通信する手段が主流を占めていた．

ここでいう Peer to Peer 対話型は，通信相手間の関係が対等で，いずれからも通信を開始することが可能であり，いずれが用意した通信路上であろうと送信権を握ることが出来る通信手段をいう．

コンピュータネットワークの世界では OSI 参照モデルにも採用された IBM の APPC(Advanced Program to Program Communication)が有名である[45].

APPC は半二重通信が基本であり，図 3-17 の CD(Change Direction)で送信権を譲渡するまで送信する権利を握る仕組みである.

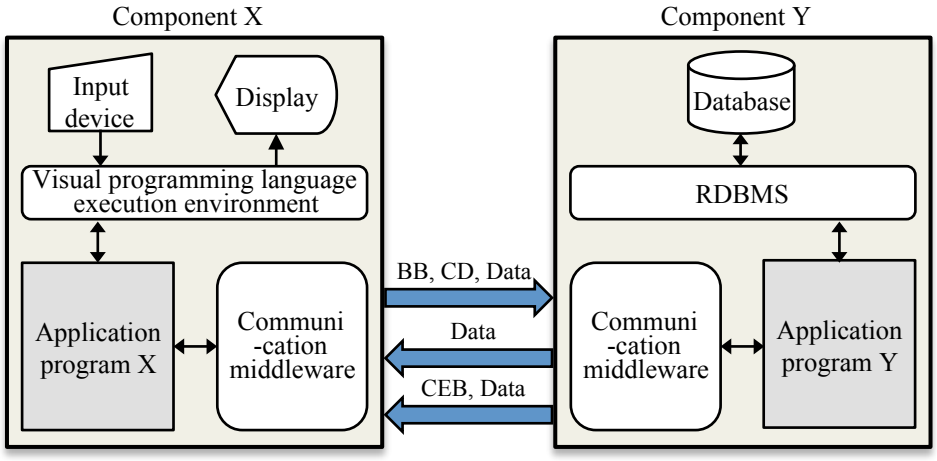


図 3-17　対話型プロトコルを利用した分散アプリケーション

Figure 3-17　Distributed application using peer to peer conversation protocol.

メッセージ転送型と同様に通信路を 2 本用意すれば全二重通信も実現可能であり，プログラマブルにデータの送受信を開始できることから，2 つの機器間で自由度の高いデータのやり取りを実現できる.

しかし，エンタプライズアプリケーションを実装するプログラマに通信規約に基づいたプログラムを記述させるのは難しく，また大規模プロジェクトでは混乱のもとでしかない. 従って，ほとんどのプロジェクトでは通信シーケンスを固定的に規定し，この通信基盤上に通信処理を支援するソフトウェア・ミドルレイヤを構築して利用する．実際 APPC 上での利用を想定した通信ミドルの COTS も販売されていた.

これらミドルウェアで実現する通信プロトコルには，もちろん前に述べた RPC 型やメッセージ転送型が含まれる．ファイル転送すら開発が可能である．実際筆者も開発を経験したことがある.

今日において通信プロトコルは，オブジェクト指向言語環境では，ORB の延長上にある技術が利用され，それ以外の言語では RPC が主流であるが，いずれも問合せ応答型の

同期通信であり，同類の技術と言える．

その他としてメッセージ転送型プロトコルは，RPC 型では実現しにくい，非同期で連携することでメリットが大きな処理方式．例えば転送先でバッチ処理を行うであるとか，大量のデータ転送を特定の通信者間で間欠的に行うなどのケースで有用な技術である．

しかし日本においては，非同期処理を実現するにあたり，本来大量データのバッチ的な転送に向いているファイル転送プロトコルを採用するケースが多い．特に近年は機器，ネットワーク共に性能向上が著しく，ニアリアルのニーズでもファイル転送で間に合ってしまう．

システムの主役は今後も RPC 型のプロトコルであろうが，非同期型の通信手段も意識しておく必要がある．

3.1.5 システムアーキテクチャ

システムアーキテクチャで扱われるａ）メインフレーム，ｂ）クライアント・サーバ，ｃ）Web，ｄ）スタンドアロン，ｅ）SOA は，メインフレームを除けば，特定の OS や特定のハードウェアに拘束されないが，ａ）メインフレームだけは，特定のハードウェア，あるいは OS を指す．メインフレームの扱いを明確にしておく必要がある．

（1）メインフレームの定義

システムアーキテクチャのａ）メインフレームは，過去においては集中処理の代名詞であり，メインフレーム上にアプリケーションを集中実装していた．物理的なクライアント機器は専用端末であり，端末はノンインテリジェントターミナルと呼ばれ，メインフレームから送られるデータの表示とユーザからの入力をメインフレームに転送する機能に特化されたものだった（図 3-7，基本構成 D）．今なお 20 年以上動いているシステムではこの構成のシステムも現存するが，新たにこのような実装を選択することは稀であり，システムの再構築時は真っ先に見直しの対象となる．現在ではメインフレームをクライアント・サーバのサーバ，Web システムの Web サーバ，AP サーバ，バッチシステムのサーバ，いずれもの DB サーバとして利用することが一般的である．またこれらをメインフレームの論理分割機能で区分けした仮想計算機上で各々実装することもよく見られる．いまやアプリケーションから見ればメインフレームと他のサーバ機器とで実装上の差異は非常に少

ない.

しかし，文献[14][15]では依然として，新規アプリケーション開発に一定割合でメインフレーム系データベースの利用が報告されているため，ここではａ）メインフレームを旧来の使い方をしているものとして分類する.

3.1.6　マルチアーキテクチャを実現するために満たすべき要件

3. 1 では，ここまで文献[14][15]が示す処理形態の 1）バッチ処理，2）対話処理，3）トランザクション処理を軸に，システムアーキテクチャのａ）メインフレーム，ｂ）クライアント・サーバ，ｃ）Web，ｄ）スタンドアロンを絡めて，機器間の処理分担，トランザクション連携の範囲，通信プロトコルについて検討してきた.

登場したシステム構成をまとめると図 3-18 のようになるが，マルチアーキテクチャを標榜する上では，最低でもこれらのシステム構成を実現できなければならない.

	バッチ処理	対話処理	オンライン処理
メインフレーム	基本構成Ａ	-*2	基本構成Ｄ 基本構成Ｇ
クライアント．・サーバ	基本構成Ｃ	基本構成Ｂ	基本構成Ｆ 基本構成Ｇ
Web	-*1	基本構成Ｅ	基本構成Ｈ
スタンドアロン	基本構成Ａ	＊基本構成Ｂ	-*3
SOA	-*4	-*4	-*4

図 3-18　システム構成の組合せ

Figure 3-18　Combination of system architecture.

埋まらない部分は，組合せとして成立しないと判断した．以下に理由を示す.

＊1　非同期起動の処理をバッチとすれば，Web や SOA でもバッチ処理を埋めることは可能だが，そのような処理はメインフレームやクライアント・サーバ，スタンドアロンで

も実現可能であり，あえて埋めることはしなかった.

＊2　メインフレームで対話処理を実現したものも過去にあったが，今日選択肢としては，考えにくいとして対象外とした.

＊3　スタンドアロンでOLTPモニタを搭載することは無意味である.

＊4　SOAでの実装形態は，7．3のマルチアーキテクチャ実現の実績で説明する.

マルチアーキテクチャを実現するために図3-18で示した基本構成AからHの8つのシステム構成に加えて，以下の技術要素を実装要件に加える.

トランザクション処理の実現方式を以下のように整理する.

- バッチ処理，対話処理については，データベースのトランザクション機能を利用して実現する.
- オンライン処理については，OLTP モニタあるいは相当の機能を持つミドルウェアのサーバサイドでのみ連携させることにする.
- 途中に利用者のオペレーションが介在する必要があるなど，長期間に渡るトランザクションの保証機構については別途対応策を講じる.

通信プロトコルについては，クライアント・サーバ型のデータベースにおける機器間の通信を除くと，いずれのケースでもRPCあるいはORBの後継技術に連なるリモートオブジェクト呼出しといった問合せ応答型，同期型の通信プロトコルを採用する.

3.1.7　プログラミング言語の選択

本ツールは外販を意識していなかったため，サポートするプログラミング言語は，企業の IT 戦略に基づいて決定した．多くの言語をサポートすることはそれだけ開発費並びに保守費がかさむからである.

ツール提供のスタートとなった2002年当時ではプログラマの質，量からも，またユーザ企業からのニーズを聞いてもCOBOLのサポートは必須であった．またJavaは勢いがあり，当然サポートは必須であった.

一方でサポートの打ち切りが見えていたVisual Basicと言語仕様からエンタプライズシステムのソフトウェア開発としては適用リスクが大きいC/C++はサポート対象外とした.

第3の言語としてVisual Basicの後継と目されたC#やVisual Basic .NETなどをサポートする.NET 環境を選択した.

今日利用されている言語は 2．1 の図 2-4 に示したように，ミドルウェアベンダが固定化されてしまう PL/SQL を除けば，C と Visual Basic が予想以上に生き残っており，システムライフサイクルの長期化を裏付けている．一方．NET は思った以上に伸びていない．

言語の選択は，以前はトレンドや最新の機能を求めて決まるケースが多かったが，近年はシステムライフサイクルの長期化と共に企業の IT 戦略に基づいて決まることが多い．こういった場合，既存資産の多さや技術者の手当のしやすさなどが評価の決め手となることもある．

3.1.8　データベースについて

サポートするデータベースも企業の IT 戦略に基づいて決定した．多くのデータベース製品をサポートすることはそれだけ開発費並びに保守費がかさむからである．

図 2-3 [26]，[27]をみるとシステム開発においてRDBMS が大凡 70～86%利用され，メインフレーム系のデータベースが 5～10%，その他が 10%程度である．90 年代後半は現在よりメインフレームでの開発が多かったと想定されるが，当時でも RDBMS が圧倒的主流であった．

RDBMS は SQL によって操作されるが，残念ながら各製品の SQL の互換性は低いレベルに留まっている．サポートされる機能の大小程度であれば多少のやりくりで使えるが，対応するデータタイプ，その定義名称というベーシックなレベルから異なっており，サポートする関数，関数名，機能も微妙に違う．独自の機能拡張も熱心であり，本ツールがサポートする製品を明確に決めざるを得ない．

また近年になって急激に NoSQL 型データベースの利用が伸びているが，今回の検討対象からは外して話を進める．

3. 2　多様なアーキテクチャへ対応するアプリケーション構造の提案

多様なアーキテクチャに対応するためには，P 層(Presentation layer)，F 層(Function layer)，D 層(Data access layer)とアプリケーションを役割で分類した 3 層，3 種のソフトウェア・コンポーネントで形成されるアーキテクチャの実現が必要と考えた.

本章では，なぜ 3 層に分割する必要があり，またそれぞれが果たす役割は何かについて述べる.

3 層には以下のような役割を与えている.

プレゼンテーション層（P層）	**人間や他システムなど外界との変換を行う. オンライン処理では，F層が実現する以外の機能全て，例えば画面の表示，入力項目の選択や単純なチェック，サニタイジング，画面遷移などを実装する.**
ファンクション層（F層）	**主にP層を介し，ユーザからのビジネス要求を処理する. Façadeとして複数のD層や他のF層を統合し，ビジネスロジックを実装する.**
データアクセス層（D層）	**主にデータソースをカプセル化する. 内包するリレーショナルデータベースについてのアクセスは，必要とするデータがテーブルの部分集合，あるいは複数テーブルのJoinを必要とする場合は，内部で演算して結果を提供する.**

図 3-19　3 層の基本的な役割

Figure 3-19　Basic role of three-layer.

P 層は，人間や他システムなど自システムと外部とのやり取り全てに責任を持つ層であり，例えば画面の表示，画面遷移，外部イベントのハンドリングなど UI(User Interface)に関わる部分全般，外部入力データの検証などが主な役割となる．ビジネスルールの実行などは他の F 層などに依頼する.

F 層は，P 層の依頼を受けて，ビジネスロジックを扱う層である．ビジネスロジックの実現に必要な D 層を都度呼出して自身にビジネスロジックを実装しても良いし，ビジネスロジックを実装した機能を呼出し，統合してもよい．F 層はこのようにいくつかの機能を統合する入り口(Façade)の役割を持つ.

D 層は今日 DAO(Data Access Object)と呼ばれるものであり，データソースをカプセル化して F 層などに提供する．代表的なデータソースは RDBMS であり，提供開始当初より Join 処理を含む SQL を内蔵している点が大きな特徴である.

また 3 層には明確にアクセス階層の順序を設けた.

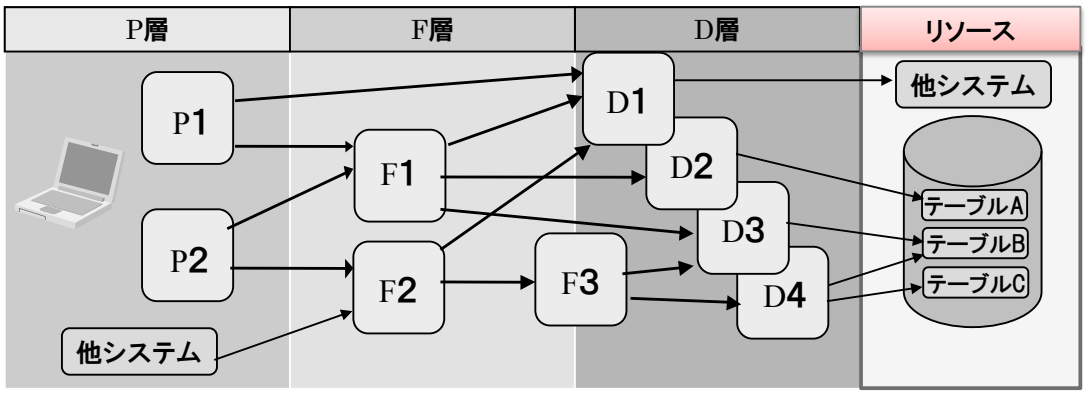


図 3-20　3 層間のアクセス順序

Figure 3-20　Three-layers access order.

図 3-20 に示すようにアクセス順序は P 層(P layer)，F 層(F layer)，D 層(D layer)の順のみを許し，逆は許さない．単純な機能の実現の場合は，P 層から D 層を直接アクセスすることで不要に F 層を作らず済むようにした（図 3-20 の P1→D1）．

また F 層は複数 F 層で階層を形成することも可能とする．これによりビジネスルールをコンポーネント化した F 層を作り，それを複数の F 層や P 層から共用することで再利用性を向上させる（図 3-20 の F3）．

ソフトウェアの 3 層構造は OOSE(Object-Oriented Software Engineering)の分析モデル[28]などにもみられる古典的な考え方であり，OOSE 登場以前でもメインフレーム，ミニコン，UNIX サーバなどを階層化して使った大規模なシステムでは実際のシステム開発で実践された実績ある分割方法である[24].

以下，図 3-19 で示した P,F,D 各層の役割分担を，簡単な処理例を使って具体的に述べる.

3.2.1　P 層と F 層の境界

3．1 の結果を見ると機器 X,Y をまたがってアプリケーションを配置せざるを得ない構成は，基本構成 F と H であり，一般に 3 層クライアント・サーバシステム，3 層 Web システムと称される．この場合の 3 層あるいは 3 階層は，物理装置の役割を表しており，3

層クライアント・サーバシステムであれば，1 層 : クライアントマシン，2 層 OLTP サーバ，3 層 : DB サーバを指し，3 層 Web システムであれば，1 層 : Web ブラウザを持つクライアントマシン，2 層 : Web/AP サーバ，3 層 : DB サーバとなる．構成する要素が増えると 4 層，5 層などと細分化して説明されることもあった[29].

本章で扱う P 層，F 層はソフトウェア・コンポーネントの区分であり，上記の 1 層，2 層など物理装置の区分とは異なるが，基本構成 F は，P 層 F 層と 1 層 2 層の切れ目がたまたま同じであるため，3.2.1 では基本構成 F を例として，P 層と F 層間の機能分担について述べる.

基本構成 F では, 利用者が操作する機器 X とデータベースが配置される機器 Y の間で, トランザクション制御を行うミドルウェアを挟んで連携している.

図 3-21 で示すように，基本構成 F では，機器 X 上に配置されるアプリケーション X は P 層であり，機器 Y に配置されるアプリケーション Y は F 層あるいは D 層ということになる.

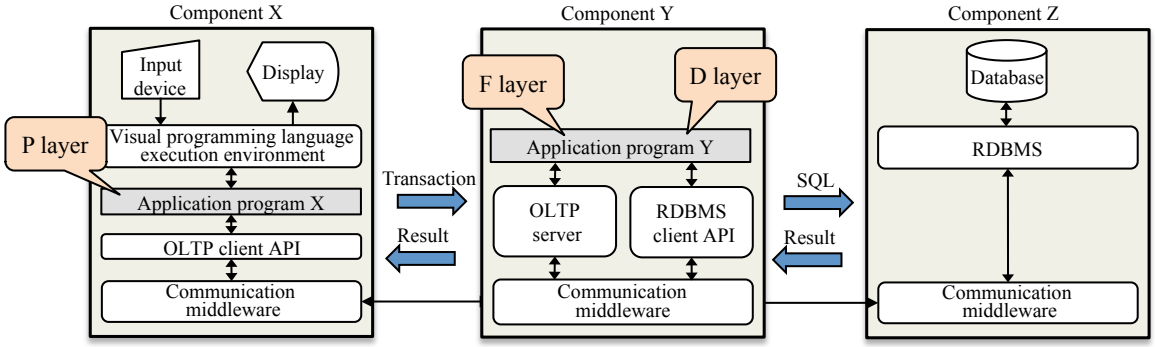


図 3-21　P 層，F 層，D 層の配置（基本構成 F）

Figure 3-21　Three layer client server system architecture.
(Basic configuration F)

ここで，主に P 層と F 層間の役割分担を，画面の入力から出力までの流れをもとに図 3-22 に示した.

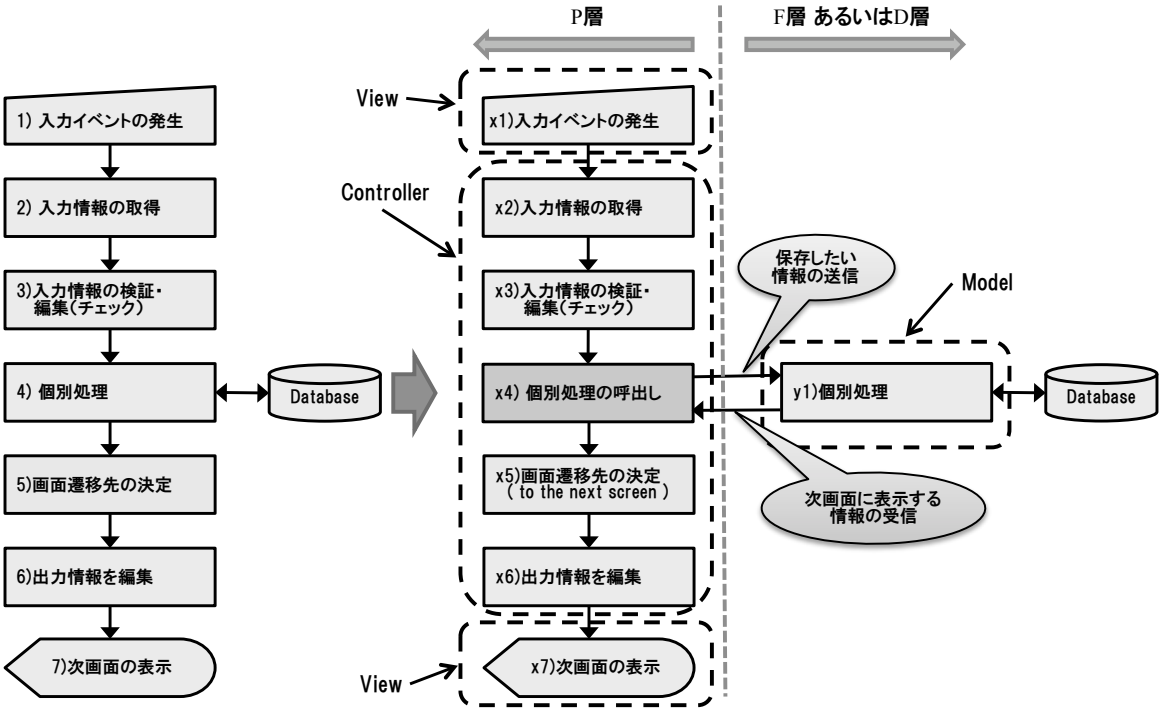


図 3-22　アプリケーション X と Y の処理分割例

Figure 3-22　Example of roles in application X and Y.

図 3-22 は，画面からの入力イベントが通知されてから結果を画面出力するまでの簡単なフローをベース（図左）に，そのフローを MVC (Model-View-Controller) パターン[30][31]に基づいて，P 層と F,D 層に分割したもの（図右）である.

大まかな流れは，キーボード入力やマウスのクリックなどをトリガーとして 1）入力イベントが発生すると，2）入力情報を取得し，次に 3）入力情報の検証・編集（チェック）を行い，4）個別処理を実行する．次に 5）画面遷移先を決定して，6）出力情報を編集(して，7）画面を表示する.

右の流れは，4）の処理のみを F,D 層に移動した例である.

MVC パターンに照らし合せると，ｘ2）～ｘ6）までが Controller で，ｘ1）ｘ7）が View，ｙ1）が Model となる.

ここでは Controller をユーザに近い P 層に配置した．View は利用者に近い P 層に配置することが自然であるし, Model はデータベースに近い F,D 層に配置することが自然だが, Controller は F,D 層に配置する選択肢も考えられる．しかし，私は Controller を P 層に配置することがより合理的だとして，図 3-22 の形態を基本形とした.

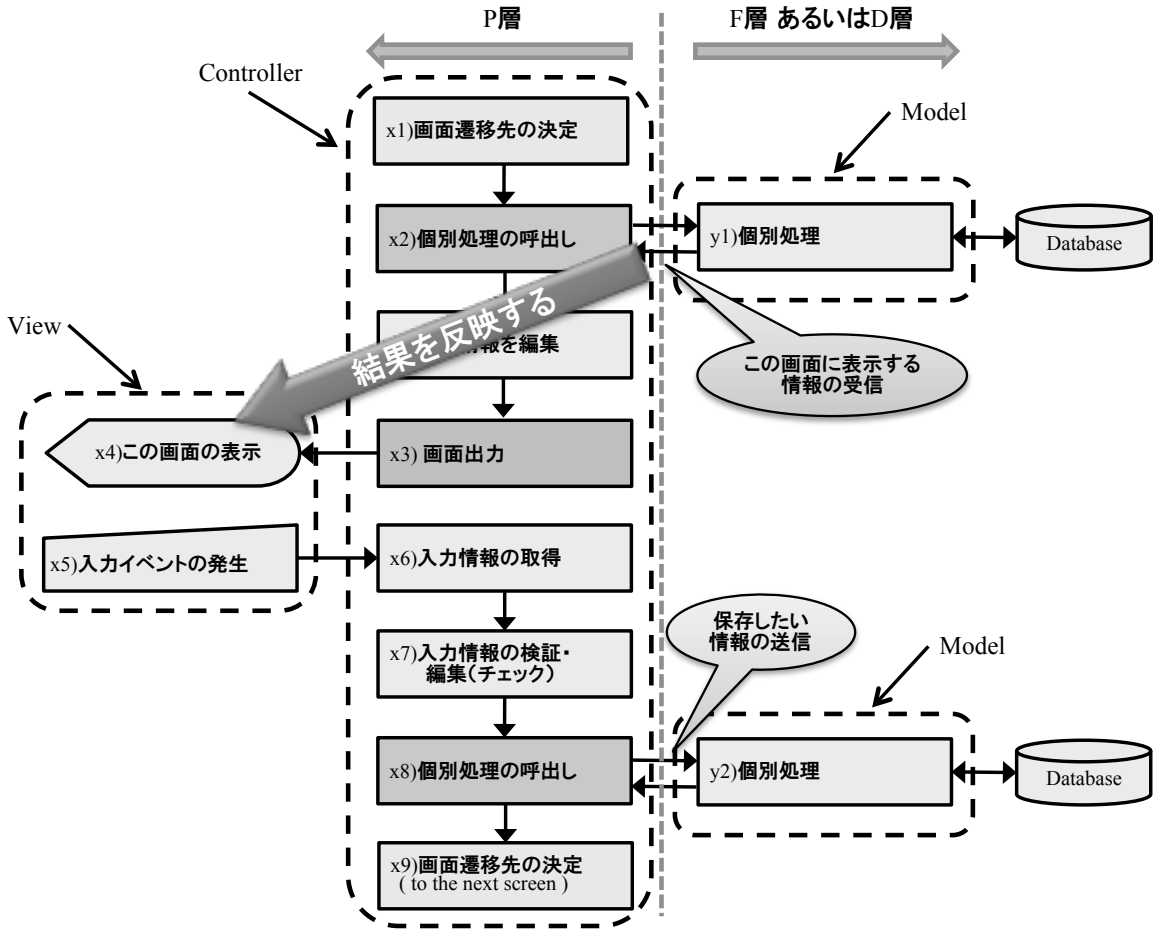


図 3-23　MVC パターンによる実装例

Figure 3-23　The implementation by the MVC pattern.

図 3-23 は図 3-22 の右の図を，画面遷移を起点に書き直したものである．ここでｘ2）個別処理の呼出しを加えて図 3-22 では無かったｙ1）個別処理を新たに呼出している．これはｘ4）画面に表示する情報を F 層から取得している．

図 3-22 では，F 層の個別処理は一度の呼出しで,データベースに永続させたい情報を送り，応答で次の画面に表示する情報を取得しているが，図 3-23 では画面を表示する前にｘ2）＋y1)で画面に表示する情報を取得し，データベースに永続化させる部分は，ｘ8）＋ｙ2）に分離実装する．このようにすることで，F 層側の実装は画面や画面遷移の実装に影響を全く受けないことになる．

MVC パターンはもともと Smalltalk のグラフィカルユーザインタフェースの実装時に提案されたもので，Model 処理と View 処理を分割して実装し，見た目の仕様(View)が変

更されてもModelを書き換える必要を無くそうとした.Model部分の再利用性を高める試みである．しかしMVCモデルでは基本的にViewはModelに従属しており， Modelの一つの見え方を具体化したものがViewという関係に縛られてしまっており，エンタプライズシステムのアプリケーションでは，このような厳密な縛りは，アプリケーション実現の柔軟性を阻害する可能性もある.

たとえばModelの写像であるViewを画面に表示しただけでは，要件を満たせないケースなどである．P層とF層に分割する際に，「画面表示とF層の呼び出しを1：1に対応させる必要はない（P層とF層の分割ルール）」と前述したとおり，必要な情報をF層に依頼して取得し，どのように見せるかはP層で独立して実装した方が，遥かに自由度が高く，またF層は当然のこと，P層も保守性が良く再利用性も向上する.

以上のことを示したパターンは PAC(Presentation-Abstraction-Control)と呼ばれる[32].

図3-24にその実装例を示す．Presentationを実装する部分の処理系は近年のHTML5やRIAなどの利用を想定して，検証や編集の処理も織り込んでみた.

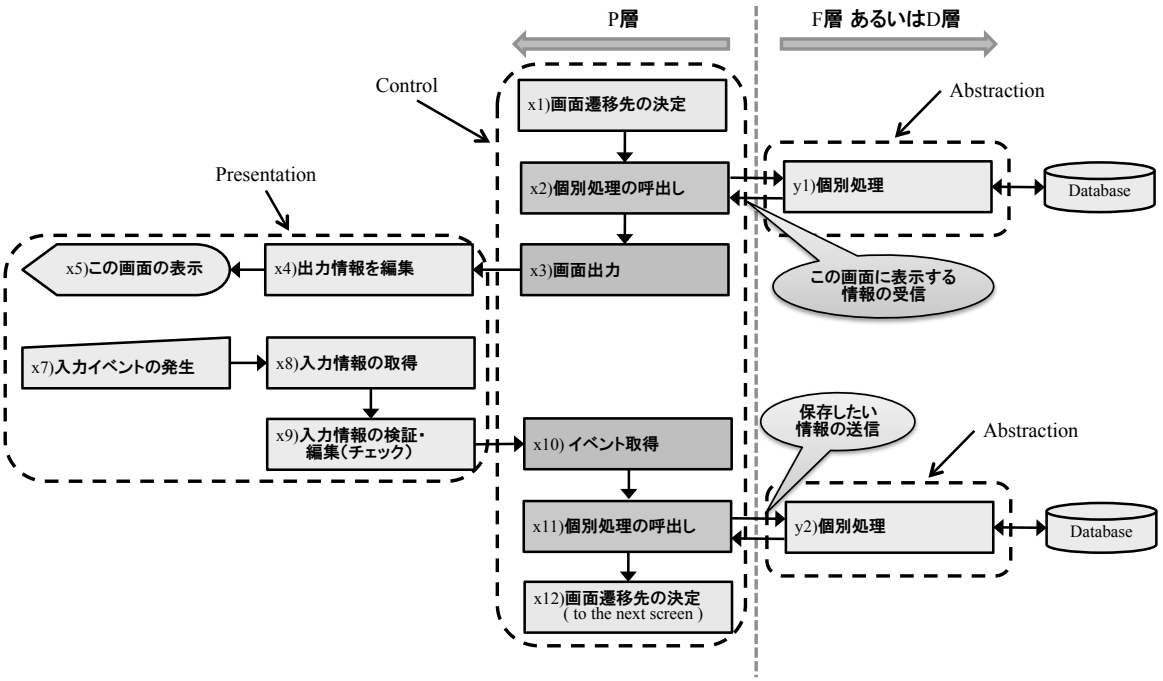


図3-24　PACパターンによる実装例

Figure 3-24　The implementation by the PAC pattern.

PAC パターンは, Model と View のような強い関連性はない. Control を仲立ちとして, Presentation と Abstraction が独立して存在し，Control から必要な時に呼出される構造である．Presentation は必ずしも特定の Abstraction に縛られず，一方の Abstraction も特定の Presentation を必要としない.

これは SOA 型のアーキテクチャを示しており，ユーザインタフェースを司る Presentation は，ユーザが望む方法，形で実装することが可能であり，Abstraction は権限のあるものであれば，相手がどのような形で利用者と対峙していようと無関係に必要とされた情報を提供するだけである.

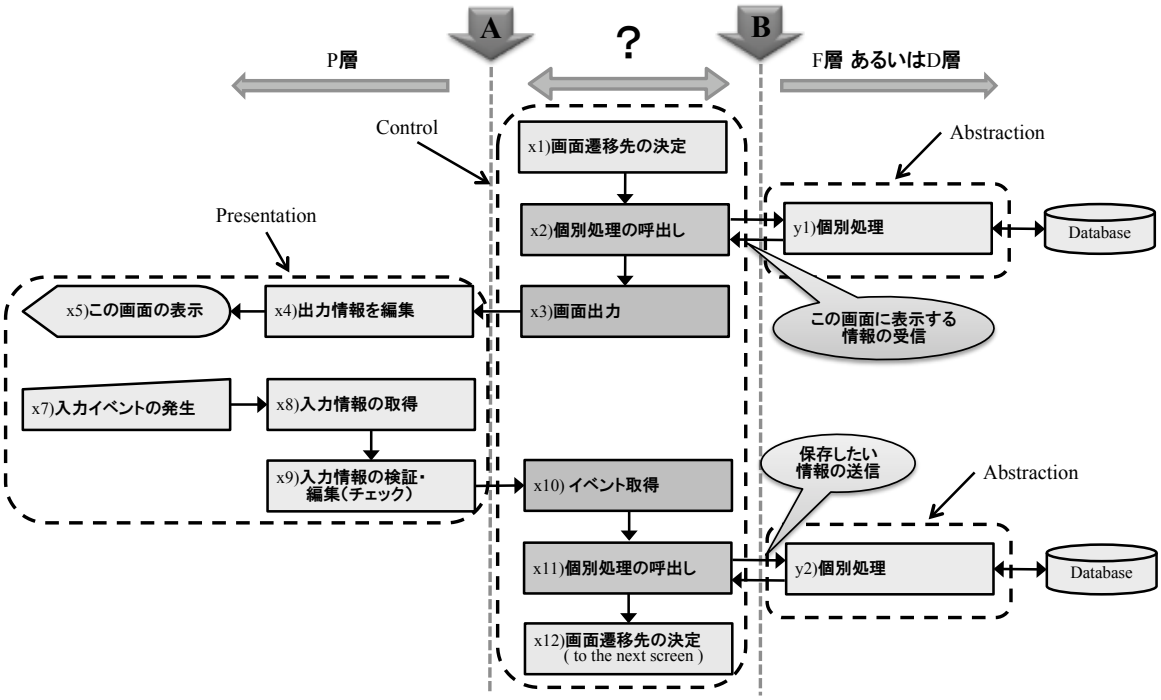


図 3-25　層分割を行う 2 つのポイント（P 層，F 層）

Figure 3-25　Two points where layer is divided. (P layer and F layer)

図 3-22 から図 3-24 までは説明の都合上，図の真ん中に配置される Control 部（MVC パターンでは Controller）を P 層に配置してきた（図 3-25 の B ポイント）が，技術的には Control 部を F 層に配置する（図 3-25 の A ポイント）ことも可能であり，またフィールドで動いているシステムでこのように配置したシステムも実際に存在する.

本ツールでサポートするシステム構成では，いずれを標準とするか決定する必要があるため図 3-26 で比較を行った.

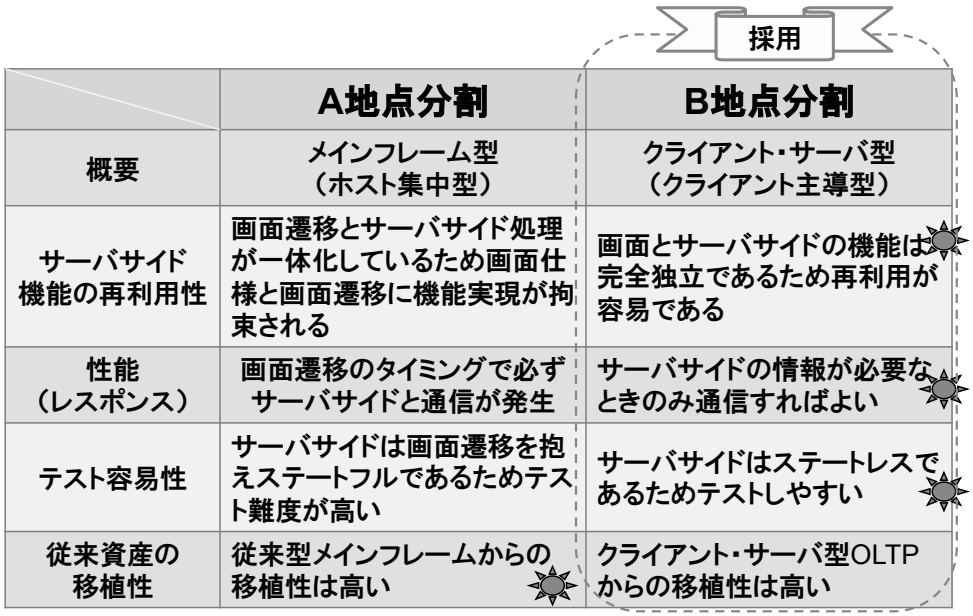


図 3-26　層分割ポイントの比較（P 層，F 層）

Figure 3-26　Comparison of two points where layer is divided. (P layer and F layer)

実際に比較を行った 1997～2001 年当時は，主に性能とテスト容易性から B 地点での分割を推奨し，また本ツール開発においても採用した．今日においては，当時よりもネットワークを含めハードウェア，ミドルウェアのめざましい性能向上があり，テスト時に多くのリソースを費やすことも許されるかもしれない．性能，信頼性に非常にシビアなシステムでなければ A 地点での分割が合理的なケースも想定されるが，サーバサイドの機能（F 層）の再利用性を向上させるのであれば，B 地点での分割を採用すべきである．

P 層と F 層の分割ルールを以下のようにまとめた．

Controller を P 層に持つ理由（P 層と F 層の分割ルール）

- F,D 層では状態を持つデータは純粋に業務ロジックに基づくものに限られるためほとんどのケースにおいてデータベースに永続化される．よって F,D 層のプログラム内ではステートフルな実装を行なう必要性がほとんどない．サーバサイドに実装されるプログラムはステートレスのほうが，はるかにテストが容易である．
  （長期間のトランザクションの扱いについては 3.2.3 で述べる）
- 画面に関連する処理は基本的に P 層内に綴じるため，画面に関連する仕様変更は，F,D 層に影響を及ぼさない．
- 画面表示と F,D 層の呼び出しを 1 : 1 に対応させる必要はなく，データベースを必要とする場面に任意に呼び出しが可能である．これにより画面の一部データのみを取得する場面でも F,D 層を呼び出すことが可能であるし，必要が無ければ画面をいくら遷移させてもまったく F,D 層を呼び出さずに済ませることも可能である．

一言でいえば，この分割ルールに従うことで P 層と F 層の間をより疎結合に出来ると判断した．この判断はツール内のみならず，後にパッケージとの連携や SOA に対応する場面でも良い方向に作用する．

3.2.2 D 層について

D 層は今日，DAO(Data Access Object)と称されるものとほぼ同等の役割を担う．

DAO は J2EE のパターンとして登場した[33]と言われているが，データソースのアクセスを楽にするライブラリだと単純に捉える．OOSE のエンティティクラスや Java のエンタプライズ向け実装セットである J2EE（Java 2 Platform, Enterprise Edition 現在は JEE と呼ばれる）のエンティティ Beans，O-R マッパーなどは，オブジェクト指向に沿ったデータリソースの隠蔽手段なのだろうが働きに大きな違いはない．また，さらに古くからある Microsoft 社の DAO(Data Access Objects)もおおくくりでは似たような概念に基づ

く実装の一形態である. 加えて古くはメインフレームでの開発が主流であった時代でさえ, データソースへのアクセスをカプセル化するアイデアは多くあり, 実際に適用されてきた.

DAO を層として実装する理由は以下である.

データアクセスをカプセル化するメリット（D 層の存在理由）

- データアクセスに関連する一連の処理を共通化し，D 層内に隠蔽化することで，F 層開発者の生産性とデータアクセスプログラムの品質を一定に保つ.
- データアクセスに関わる処理を一元化することで，トラブルの多いデータアクセス周りの問題を一元的に解決可能とする.
- DBMS のリプレースなど仮にデータソースの管理対象が変わっても，F 層への影響を最小限に保つ.

DAO はパターンであり，リファレンスとなる実装形態があるわけではないので，一概に比較は出来ないが，私が考える D 層は，例外無くデータソースへのアクセスをラッピングすることとしている.

特に 2000 年代前半のころに登場した DAO は，RDBMS のラッピングに関して単純なデータの新規登録，更新，削除，主キーによる検索だけをサポートして DAO と称しており，SQL が苦手なプログラマからは歓迎されたであろうが，とてもデータアクセスを隠蔽したとは言い難かった.

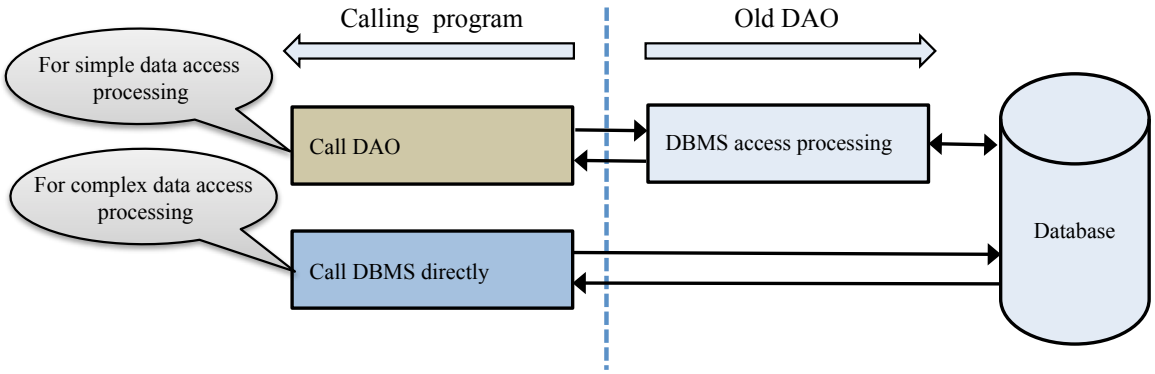


図 3-27　古い DAO が提供する機能

Figure 3-27　Function that old DAO provides.

私が提案するD層はJoinを含む複雑な検索を当初よりサポートした点に特徴がある.

また，オブジェクト指向言語で実装する場合は，F層から勝手に同じ外部リソースへアクセス出来ないように可能な限りガードもかけている.

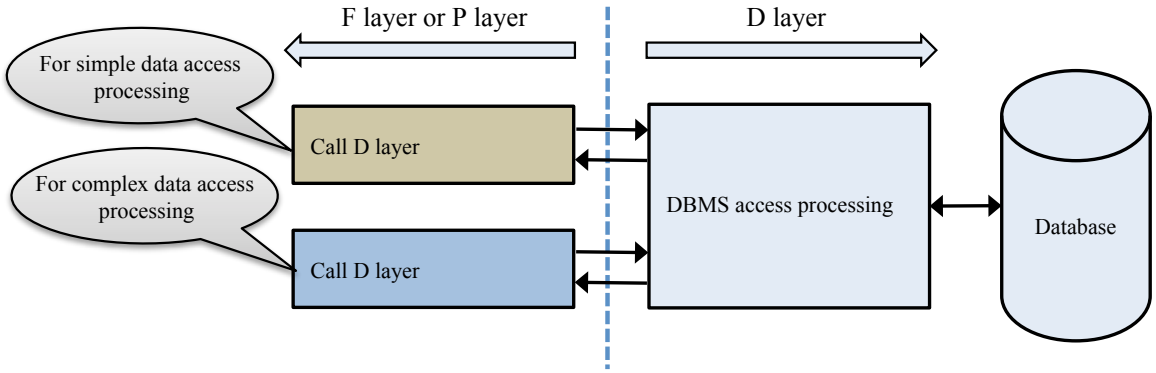


図 3-28　D 層が提供する機能

Figure 3-28　Function that D-layer provides.

今日 DAO を名乗るフレームワークでは，複雑な SQL は，SQL ライクな独自 DSL での記述でサポートするものが増えてきているが，本ツールは 2002 年の提供当初から開発ツール上の UI(User Intrface)で SQL をデザインするスタイルを確立してきた.

3.2.3　長期間トランザクションのサポート方法

3.1.4 で述べたように，RDBMS 等が提供するクライアント・サーバ型の分散データベース機能や OLTP モニタ等の分散トランザクション機能は，利用範囲を限定した．理由は既に述べたように，これらはいかなるケースにおいても満足なパフォーマンスを得られる万能な機能ではなく，利用シーンを考慮して限定的に利用する必要がある．配慮に欠けた利用方法では著しく並列実効性を毀損し，デッドロックや障害発生時はペナルティが非常に大きい.

このような経緯から，途中に利用者のオペレーションが介在するような長期間トランザクションを実現するためには，別の方法を提供する必要がある.

長期間トランザクションの問題も，クライアント・サーバや Web など分散処理が台頭

してきて生まれた課題ではなく，メインフレーム全盛のころから取組んできた課題である．DBMS や OLTP モニタなどミドルウェアの機能をあてにしないため，アプリケーションでトランザクション処理を実現する必要がある．長期間トランザクションを実現する上での最大の課題は，特定リソースを長期間（あるいは長時間）占有するための排他機能をどのように実現するかである．

排他処理は，大きく悲観的排他と楽観的排他が存在する．

悲観的排他は，最初にリソースに対して操作権を獲得したものが，他者が更新不可能なように独占的に排他をかける方法である．予め独占的な排他をかける用心深さを形容して悲観的(Pessimistic)と呼ばれる．メリットは自身が操作したいリソースは処理の開始当初から占有しているので，処理エラーや機器の障害，デッドロックが発生しない限り，必ず最後までトランザクションを完結出来る．日本では古くから好まれている排他方法で，メインフレームが中心のシステムでは良く見られた方法である．欠点は不用意に多用してしまうと多くのリソース，あるいは長時間リソースを独占して並列実効性を阻害してしまう．今日では一つのリソースに更新処理が集中することが明らかな場合において限定して利用すべきである．またバッチ処理や非同期処理などは基本的に悲観的排他を利用すべきだが，これらのケースは RDBMS が提供する排他機能でカバーできる．

一方が楽観的排他と呼ばれる方法である．楽観的排他は，誰かがリソースに対して操作を開始しても，別の人が同じリソースの操作を開始することを阻害しない．つまり最初にリソースを操作した人であってもリソースを独占せずに，他人にも操作の開始を許してしまう．この鷹揚さを称して楽観的(Optimistic)と称す．

楽観的排他の場合，トランザクションを成功裏に完結できるのは，最初に操作者を開始した人とは限らず，最初にリソース変更を実行した人となる．この不公平感が日本では嫌われたのか，長い間利用する人は少なかったようである．

楽観的排他は，多くの場合単一のリソースを競合して排他を掛け合うケースは存外少ないという前提に成り立つ排他方式である．まさに楽観的なケースで利用する．つまり大多数のトランザクションが競合すること無く成功するケースに利用が限られる．

これまでエンタプライズシステムで発生するリソースの長期占有やデッドロックは，純粋にビジネスルール（アプリケーション）に起因した例は少なく，ほとんどが不味いデータベース設計や，ミドルウェアの実装の都合によるデータ格納領域，データベースのイン

デックス，メモリ管理，通信セッションなどの占有や奪い合いに起因している．

エンタプライズシステムのアプリケーション･プログラムにおいて排他が競合するのは，ほとんどがデータベース上に既に格納されているレコードの変更（書き換え）処理に限定できる．しかもそのビジネスユースケースはあまり例がない．

以上のような理由から，本ツールが提供するアプリケーション構造では，データベース上の特定レコードを対象とする楽観的排他のみをサポートすることとした．

本ツールが採用した楽観的排他の実現方式は，今日ではVersion Numberパターン[34]と呼ばれる方式である．データベースの特定のカラムにバージョン番号を埋め込んでおき，データを取得(select)した時のバージョン番号を保持する．実際に当該レコードを更新(update)する場合はバージョン番号が前回と同一であることを確認して更新データと一緒にバージョン番号も更新する．仮に更新時にバージョン番号が異なっていた場合は，既に他者が当該レコードを更新したことになるため今回の更新処理は失敗となる．同様の処理を再度実施したい場合は前回取得した情報を破棄し，取得からやり直す．

3．3　3層アプリケーション構造によるマルチアーキテクチャの実現

ここでは，3．2で提案した3層3種のソフトウェア・コンポーネントが3．1で示した各種のシステム構成上で動作するアプリケーション・プログラムとして実現可能であることを示す．

3．1で述べたようにプログラミング言語，RDBMSやミドルウェアの選定については，あらゆる組合せを検討するのではなく，企業戦略上優先度の高いものをサポートできれば良いとする．

エンタプライズシステムではRDBMSに限らず，バッチはバッチ制御ミドルウェア，クライアント・サーバならばOLTPモニタ，Webならば APサーバなどミドルウェアが充実しているため，これらの利用を前提にアプリケーション・フレームワークやライブラリを用意し，意識したコード生成する．

F層やD層は，当初よりCORBAやJ2EEを意識して設計したためリモート呼出しを前提とした実装を行っている．代表的な例としては3．2で述べたようにコンポーネントの独立性を高く，また高多重での実行，テストの容易性向上を狙いF層，D層はステート

レスでの実装を強いる.

サーバサイドで状態を保持したい場合には D 層を使ってデータベースにデータとして保持するか，フレームワークが提供するデータキャッシュ機能を利用する.

(1) スタンドアロン，バッチ処理の実現方法（基本構成 A,C）

バッチ処理は今日でも大量のトランザクションを効率よく処理する場合に利用される処理方式である．一括で処理するため，途中で利用者が介在することはできない．また他の処理形態と比べ排他期間が長くなるため，競合を運用で調整する必要がある.

スタンドアロン構成におけるバッチ処理を 3 層ソフトウェア・コンポーネントで実現するための構成を示す（図 3-29）.

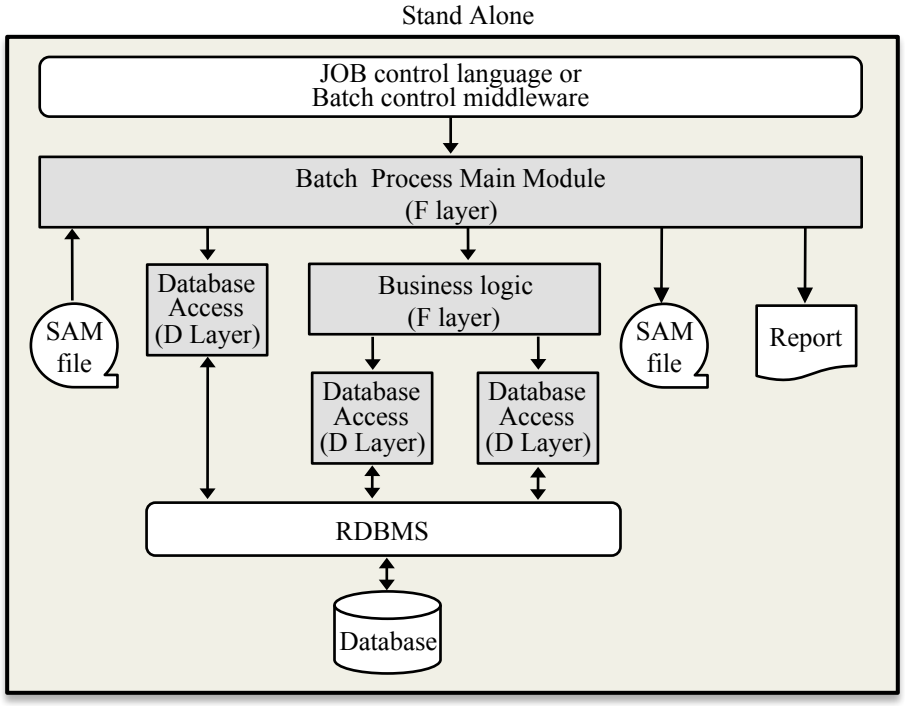


図 3-29　バッチ処理（基本構成 A）

Figure 3-29　Butch processing. (Basic configuration A)

スタンドアロンとなる対象の機器は，メインフレームでも商用 UNIX でも Linux, Windows でもよく，また PC でも良い. 言語は現在でも COBOL を選択する例が多いが，近年は Java を選択するケースも増えてきている.

従来型のバッチ処理は，前述したとおり，既に確立されたフレームワークが存在するため，ここでは非同期実行用の処理フレームワークを示す.

図3-29のアプリケーション・プログラムは, コマンド起動やShell系のスクリプト言語, バッチ制御ミドルウェアなどからの起動を想定している.

図中D層と業務ロジック(Business logic)を抱えたF層は，3．2で述べてきたD層, F層そのものだが, バッチ処理メインモジュール(Butch process main module)機能を持つF層は従来のF層の機能に加えて，トランザクションを一括処理するためのテンプレートを提供する.

バッチ処理の場合，トランザクションは性能を考慮しシーケンシャルファイルで提供されるケースが多い（図中では，左のSAM file）.

ここから一件ずつトランザクションレコードを取り出し，処理を行ない，結果を左のシーケンシャルファイル, あるいはリストなどへ書き出す. 一般に左のシーケンシャルファイルの最終レコードまで繰返し実行される. バッチ処理メインモジュール機能を持つF層は，この流れを制御する.

バッチ処理においてはRDBMSに対するアクセスもシーケンシャルreadになるように設計することで処理を高速化できるケースがある．RDBMSがシーケンシャルreadで高速化オプションを持つ場合にはD層でも高速化オプションに対応する機能（たとえば一括readを行なってバッファリングする）を実装しておく必要がある.

Shell 系のスクリプト言語，バッチ制御ミドルウェアなどで実行制御を行なうケースでは，当該プログラム1本だけを実行させるのではなく，処理が関連するプログラムを順次実行させる．巨大なものでは，関連するプログラムが数千本に及ぶケースもある.

図3-29はスタンドアロン構成であったが，バリエーションとして図3-30にDBMSのクライアント・サーバ接続機能を利用したクライアント・サーバ構成によるバッチ処理を示す. 構成AからCへの変更のためにアプリケーション・プログラム上で特段の変更作業は発生しないが，これはDBMSの機能により実現される.

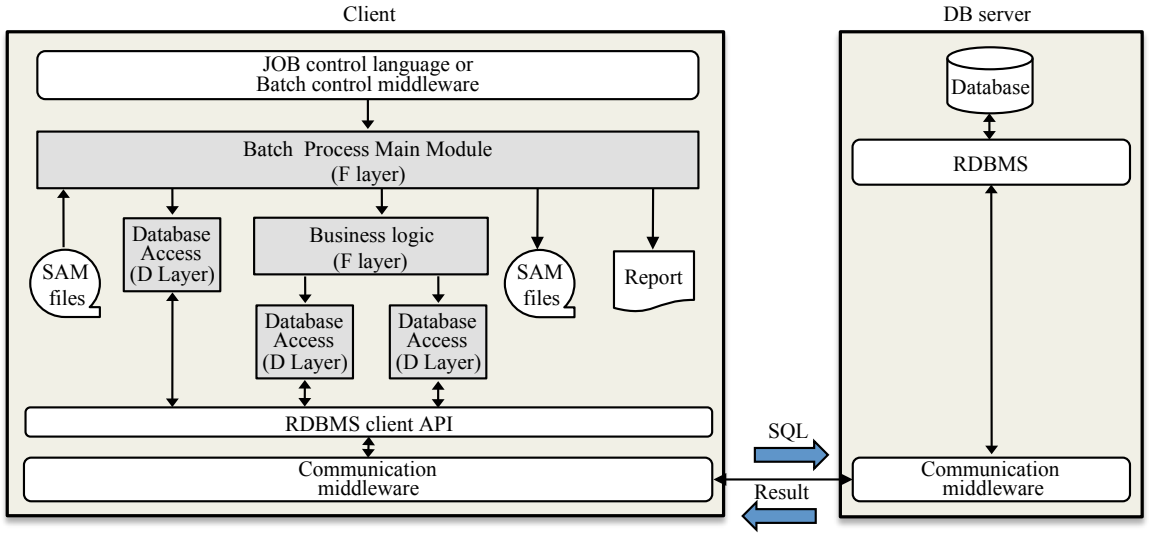


図 3-30　バッチ処理（基本構成 C）

Figure 3-30　Butch processing. (Basic configuration C)

(2) 対話処理型クライアント・サーバシステムの実現構成

クライアント・サーバシステムは，旧来のメインフレーム中心のシステムと比べると主従逆転が起こったシステム構成である．クライアント・サーバシステムは，クライアントが必要とするリソースを持つサーバとの間で動的に関係を構築して処理を行なう．クライアントが主導権を持つ分散型のシステムである．

図 3-31 はクライアント・サーバの対話処理構成を 3 層ソフトウェア・コンポーネントで実現している．クライアント・サーバの対話処理構成は，比較的少人数で利用される小型のシステム開発時に適用されるシステム構成である．今日，サーバ能力もネットワークの能力も巨大になってきているため，この構成で数万の利用者をかかえることも実現可能だが，次に紹介するオンライントランザクション処理型と比べるとより巨大なサーバを用意する必要があり，現実的でない．

対話処理は OLTP モニタ相当の機能を用いないオンライン処理と定義されていることから，RDBMS が提供するクライアント・サーバ連携機能を使って実現するため，3 層ともクライアント側に実装される．

UI(User Interface)は，以前は 4GL などと呼ばれていた.NET 言語などビジュアルな画面開発をサポートする開発環境を持つ言語を使って実現される．そのため，3 層ともに同一言語で実装されるのが一般的であり，実現難易度は高くない．この点においても比較的

小規模なシステムの開発に向いているといえる.

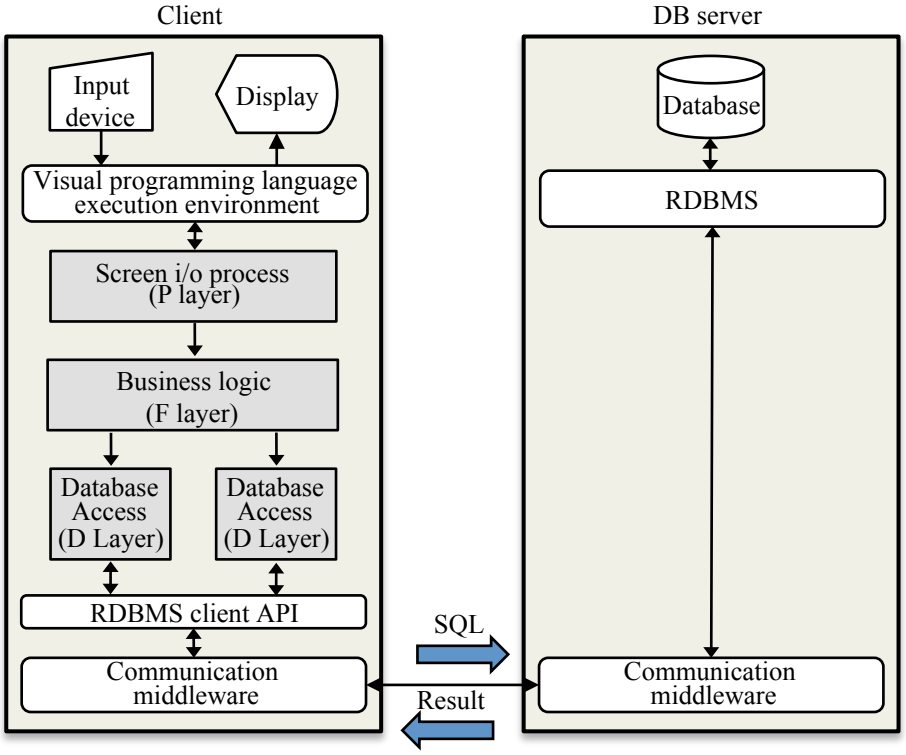


図 3-31　対話処理型クライアント・サーバシステム構成（基本構成 B）

Figure 3-31　Client server system configuration of conversation processing type. (Basic configuration B)

スキルの低いプログラマの立場から見れば，3 層に分けてプログラムを書くよりも，画面からのイベントの延長ごとにプログラムを書くほうが理解しやすく，また速く完成に近づく．しかし，プロジェクトからみれば，そのようなスタイルの開発を許してしまうと，プログラムの記述様式が個人の力量，趣味が反映されてばらつき，品質が揺らぎやすい．当然保守性は悪く，再利用の実現はまったく不可能になってしまう．

対話処理型クライアント・サーバ構成で 3 層コンポーネントを適用する場合は，簡単に作れるという誘惑を断ち切る強い理性を求められる．

(3) オンライントランザクション型クライアント・サーバ構成

図 3-32 はオンライントランザクション処理型クライアント・サーバ構成である．

オンライントランザクション処理型は OLTP モニタと呼ぶミドルウェアの利用を前提とした構成である.

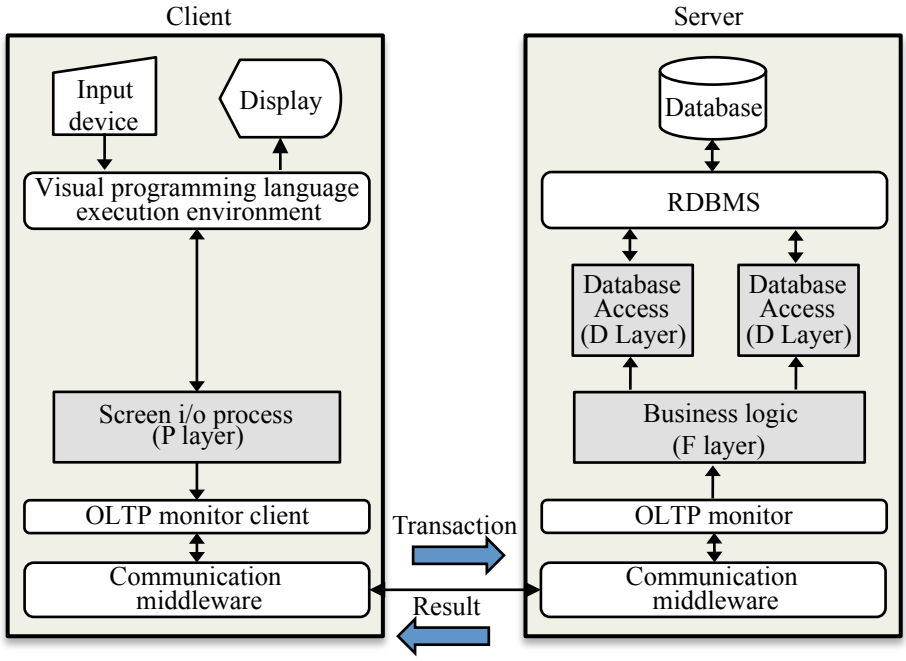


図 3-32　オンライントランザクション処理型クライアント・サーバシステム構成
(基本構成 F)

Figure 3-32　Client server system configuration of online transaction processing type.
(Basic configuration F)

OLTP モニタは多数のクライアントからの要求をいったんキューに溜め，リソースの空き状況を見ながら優先順位に基づいてトランザクションを流す．サーバが持つリソースを破綻無く最大限有効活用するための仕掛けである．このため比較的大規模なシステムの開発に向いている.

P 層と F 層の間に物理的な境界が出来るため，P 層と F 層は別個に作られる．3．2 に示した明確な分割ルールに基づいて開発する．OLTP モニタのサポート範囲により利用可能な言語は変わってくるが，たとえば P 層は.NET 言語，F 層，D 層は COBOL という組合せはポピュラーである．P 層と F 層の機器をまたがる通信機能は，OLTP モニタの機能を使って実現する.

図 3-33 は, 図 3-32 のバリエーションであり, Web システムでも良く見られるようにサー

バサイドを AP サーバ，DB サーバに分割した 3 層クライアント・サーバ構成である．これもアプリケーションに対して変更インパクトは発生しない．RDBMS の機能で実現される．

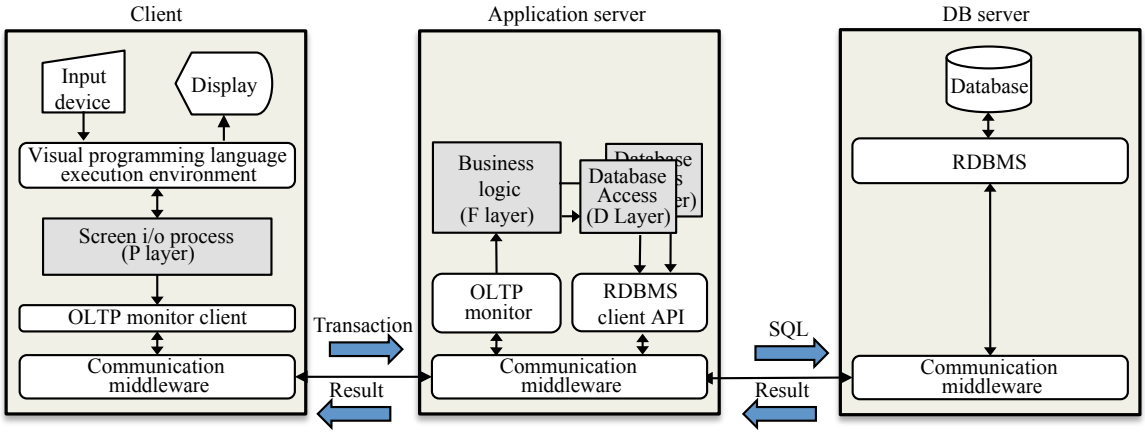


図 3-33　オンライントランザクション処理型クライアント・サーバシステム構成
（基本構成 F`）

Figure 3-33　Client server system configuration of online transaction processing type.
(Basic configuration F`)

3 層ソフトウェア・コンポーネントを適用する場合，対話処理でもオンライントランザクション処理でも分割ルールは同じであるからソフトウェア開発の難度は本来同じである．

しかし，現実にはオンライントランザクション処理を適用するとなれば，採用される言語の数が増え，またサーバサイドのプログラムのテスト・デバッグ技術も必要となる．

本ツールはこのような障壁を下げる働きを持つ必要がある．

(4) 対話処理型 Web システムの実現構成

Web システムは今日主流のシステム実現方式である．クライアントに置いた Web ブラウザからネットワーク上にあるあらゆるリソースを(権限さえあれば)自由に利用できる．利用者からすればエンタプライズシステムもそのひとつに過ぎない．クライアント・サーバよりもさらに主従は逆転し，クライアントからネットワークの世界につながる自由度を手に入れられる．

Web システムとクライアント・サーバシステムの大きな違いはフロントにある．Web

システムはWebブラウザを置く機器がクライアントとなり,P層を抱える部分はWebサーバ上にある．これがクライアント・サーバシステムとの大きな違いとなる．逆にこの差を除くと，考え方はクライアント・サーバシステムと大差ない.

図 3-34 に対話処理型の Web システム構成を示す．フロントに Web ブラウザを置く機器をクライアントとして置く．ここにはアプリケーションはない．実際には Web サーバから動的にダウンロードされる JavaScript などの仕掛けでローカルリソースに直接アクセスできないなどの制約の中で任意の処理を実現できる.

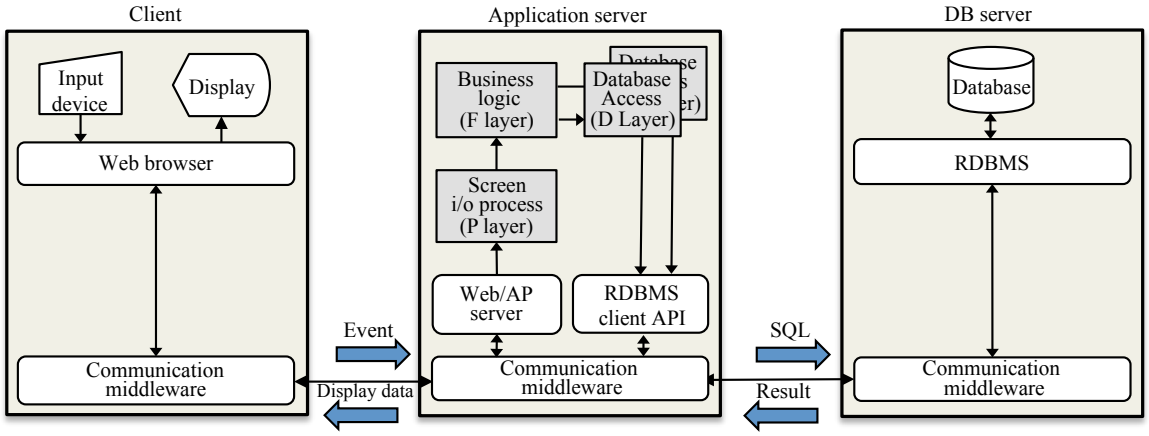


図 3-34　対話処理型の Web システム構成（基本構成 E）

Figure 3-34　Web system configuration of conversation processing type.
(Basic configuration E)

対話処理型の場合は，3 層のソフトウェア・コンポーネントは Web サーバ上に全て実装される．3 層が混載される点はクライアント・サーバと類似しているが，Web システムの場合はフレームワークの利用が進んでおり,何らかの役割分担で実装されるケースが多い.

ただし，本論文の 3．2 で示した明確な分担ルールで実装するには，プロジェクトマネジメントの力量が問われる.

3 層のソフトウェア・コンポーネントについては，F 層，D 層は Java のサーバサイドプログラムの基本である JavaBean として実装する．P 層は採用する Web フレームワークに依存して実装方法が決まる.

(5) オンライントランザクション処理型 Web システムの実現構成

図 3-35 にオンライントランザクション処理型 Web システムの構成を示す.

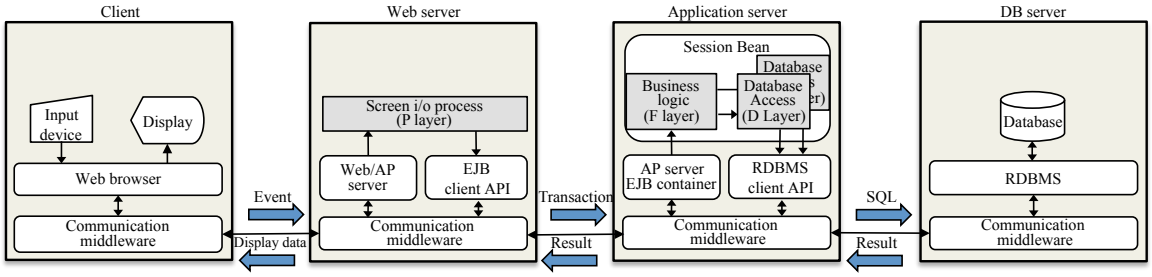


図 3-35　オンライントランザクション処理型 Web システム構成（基本構成 H）

Figure 3-35　Web system configuration of online transaction processing type. (Basic configuration H)

クライアント・サーバの OLTP モニタと同等の機能は，AP サーバ（EJB コンテナ）とサーバサイドのプログラムを実装する仕掛けである EJB(Enterprise Java Beans)で実現される.

しかし，2000 年代当初の EJB は重い実装を強いて，さらにパフォーマンスが出ず，大規模システム向けをうたいながら，実際には大規模システムでは遅くて使えなかった. EJB2.0 以降性能は改善されたが，重い実装は敬遠の的となり，OSS(Open Source Software)を中心に代替策が多く提案されるようになる．EJB3.0 が出るころには，完全に OSS や独自実装が主流となり，今日 EJB は選択肢の一つでしかない.

OLTP モニタが提供していたトランザクションキューやリソース割り当て，優先度制御などは，AP サーバ（EJB コンテナ）と EJB の組合せにおいて EJB のサービス単位に実現できるよう各社の製品で提供されているが, Web/AP サーバでも単独で HTTP セッション単位に流量制御が可能な製品があり，JavaVM のチューニング（メモリ割当とスレッド数）と組合せればサーバリソース量を意識した流量制御を実現できる.

以上のように EJB をサポートせずとも対話処理型の実装で高トラフィックへの対応は可能と判断していたが，パフォーマンスが出るようになった EJB2.0 の時期に一時サポートしていた. 本ツールが EJB をサポートする方式は, F 層にステートレスセッション Bean のアダプタをかぶせる形で実装する．D 層までもエンティティ Bean に割り当てると重い実装となってしまうため, D 層は F 層のサブルーチン的な位置づけで取り入れる形とした. じきにリモートインタフェースを持つ重い実装のエンティティ Bean はなくなったため,

本ツールの実装方針は正しかったようである.

3. 4　まとめ

第3章では, マルチアーキテクチャの実現を検討するために, まずマルチアーキテクチャとして扱う範囲のシステム構成群を定義し, 次に, 上位に載るアプリケーションソフトウェアをどのように分割して配置すべきかを示した.

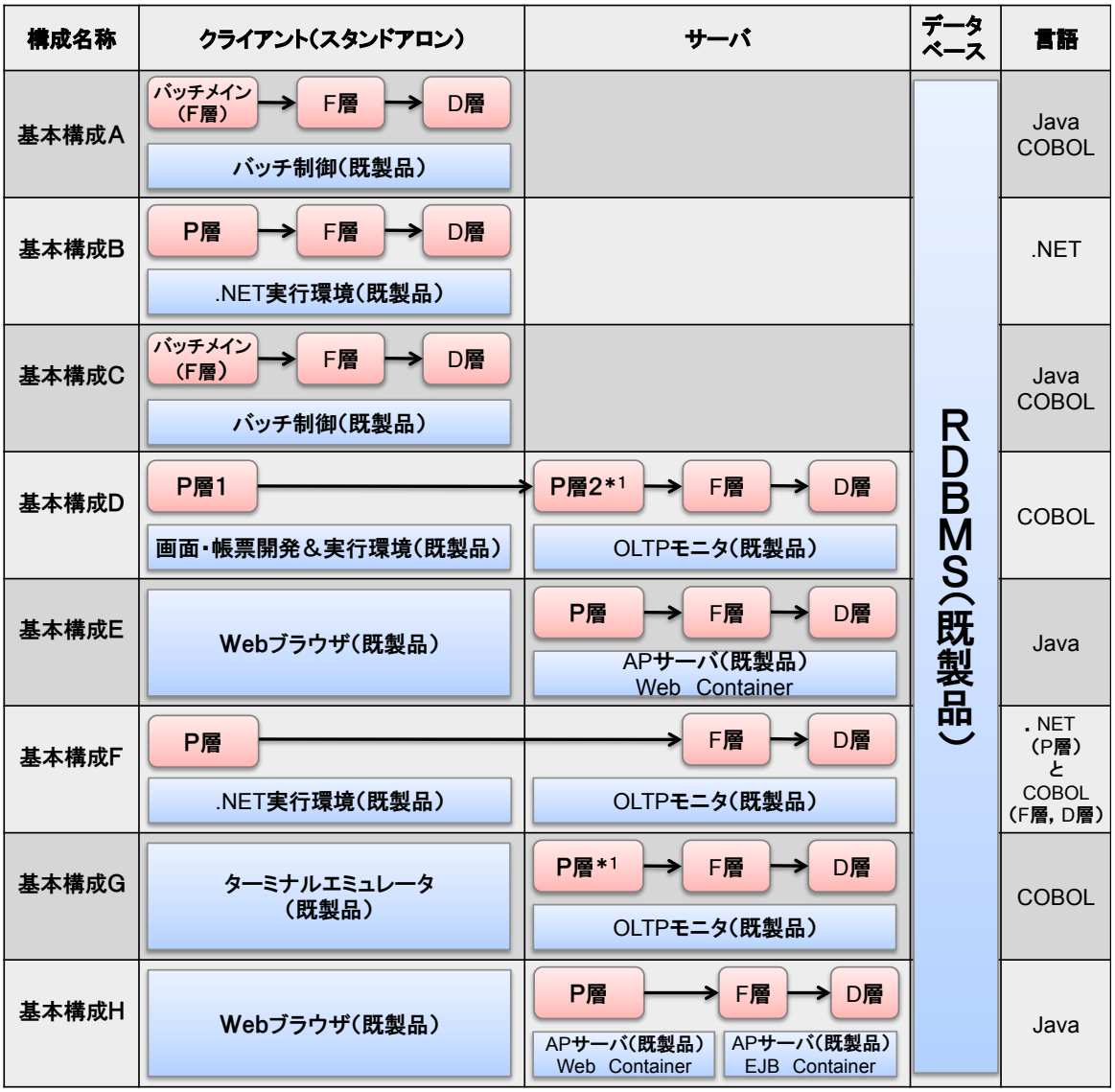


図 3-36　マルチアーキテクチャの基本となる８つのシステム構成

Figure 3-36　Eight system configurations that are basic of multi architecture.

＊１：COBOL のオンライントランザクション処理では，本ツールが提供する画面遷移フレームワークを利用する．また基本構成 D では P 層が１と２に分かれているが，これは図 3-26 の A 地点分割を実現している．

3.1 では, マルチアーキテクチャで扱う基本的なシステム構成を 8 つのパターンとして, に定義し，3．2 ではアプリケーションを P 層，F 層，D 層という 3 つのソフトウェア・コンポーネントに分割して実現することとし，3．3 で 3 つのコンポーネントを 8 つのシステム構成にどのように分散配置するかを定義した．以上を一覧にまとめたのが図 3-36 である．

エンタプライズシステムは，プログラミング言語や OS などに加え，処理形態のバッチ処理，対話処理，トランザクション処理や，システムアーキテクチャのメインフレーム(ホスト集中型のオンライン)，クライアント・サーバ，Web，スタンドアロン，SOA など，これらエンタプライズシステムを構築する上で繰り返し現れる共通的な仕掛け，機能の実現をサポートするミドルウェアと称するソフトウェアが存在する．多くの COTS (既製品)，そして近年はオープンソースも存在し流通しており，ほとんどのケースで何らかのミドルウェアが採用される．

P 層，F 層，D 層という 3 つのソフトウェア・コンポーネントをこの 8 つのシステム構成上に実装する場合，これらミドルウェアの仕様，制約を受ける．具体的には実装方法，実行方法，API(Application Programming Interface)，コマンドなどを意識して P 層，F 層，D 層を実装しなければならない．

本研究で提案するツールは，これら実装上の仕様，制約を吸収し，開発者が P 層，F 層，D 層のアプリケーション機能の実現に集中出来る環境を作り上げる必要がある．

具体的にはツールがプログラムを自動生成する時に，これら実装上の仕様，制約を吸収する指示を行い，前提となるミドルウェア上で動作可能なソースコードを生成するという機能が必要である．如何に自動でコードを生成するかを，ツールの全体像を示した第 6 章の 6．2 (2) F 層の設計から生成までの流れで示す．

2．4 で掲げた本論文の課題である，マルチアーキテクチャに対応するツールを実現するために，この第 3 章では，マルチアーキテクチャ自身を定義し，次にそのマルチアーキテ

クチャ上で動作するアプリケーションソフトウェアの構造を定義した.

次章では，P 層，F 層，D 層と命名したアプリケーション・コンポーネントをいかなる開発プロセスで，またどの部分をプログラムの自動生成で開発するかについて述べる.

第 4 章　モデルベース開発，反復型開発の両立を目指す開発プロセスの体系化と自動生成されるソフトウェアの構造

本章では，モデルベース開発と反復型開発が共存できる開発プロセスとツールの関係を明確にして体系化し，さらにツールが生成すべきソフトウェア機能について明らかにする．

前章では，　P 層，F 層，D 層の 3 層からなるソフトウェア・コンポーネントを提案し，マルチアーキテクチャに対応可能であることが明確化できた．しかし，これらコンポーネントを効率よく，高品質に開発できなければ，エンタプライズシステムの開発には適用できない．

この章では，これらソフトウェア・コンポーネントをどのような手順，プロセスで開発を進めるのか開発プロセスを提案し，次に開発プロセスで作成されたモデル（仕様記述）からどのような部位をソフトウェアとして自動生成するのかを明らかにする．

モデルベース開発と反復型開発というトップダウン指向の開発プロセスとボトプアップ的取組みという相反する側面を持つプロセス同士をどのように融合するのか，またモデルベース開発を行うことで何が自動で生成され生産性と品質向上に寄与するのかを明らかにする．

4．1　モデルベース開発と反復型開発を両立させる開発プロセス

ここでは，本ツールを利用したモデルベース開発で想定している開発プロセスについて述べる．この開発プロセスの中で P,F,D の 3 層からなるソフトウェア・コンポーネントを設計し，開発ツールを利用してソフトウェアを自動生成する．

4.1.1　モデルで記述する範囲

はじめにモデルで記述する範囲と，開発ツールで開発を進める部分を明確にするために開発プロセスを定義し，モデリングで進める部分と開発ツールで生成を進める部分を明確にする．

開発プロセスは，ISO/IEC12207:2008[35]と，日本における強化版である共通フレーム

2013[36] に基づいて定義する.

図 4-1 は，ISO/IEC12207:2008 からソフトウェア実装プロセスの 4 つのサブプロセスを抜き出し，図式化したものである．主にソフトウェアの設計から実装までに関連するものが並んでいる．この 4 つのプロセスの上位にはシステムレベルの開発プロセス，また上流には，「超上流」[37]と呼ばれる企画プロセス，要件定義プロセスがある.

4 つのサブプロセスは，上位から下位に設計情報を受け継ぐ形で役割分担されているが，規格や共通フレームでは具体的な成果物を定義していないし推奨もしていないので，自身の開発プロセスを設計する者が独自に定義して良い．また，どこかの作業を省略したり，新たな作業を追加することも独自に行って良い．このように自身の組織，プロジェクトの意思に沿うようカスタマイズする作業をテーラリングと呼ぶ,

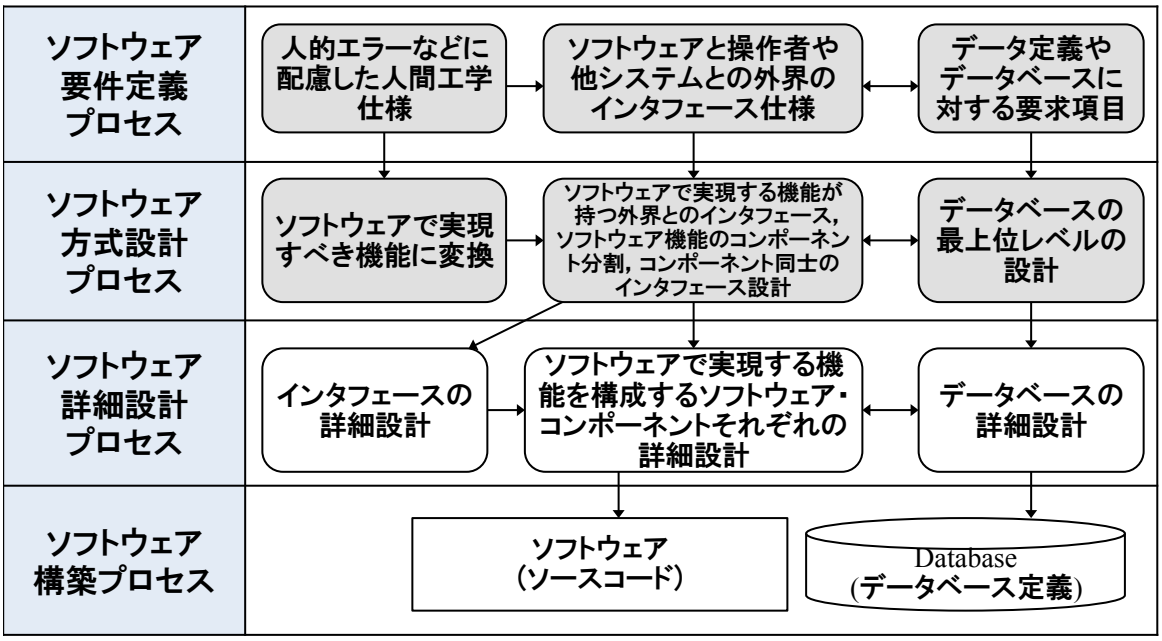


図 4-1　ソフトウェア開発プロセス

Figure 4-1　Software development process.

最上位のプロセスはソフトウェア要件定義プロセス(Software Requirement Analysis Process)であり，人的エラーなどに配慮した人間工学仕様(Human-factors engendering specification)や，ソフトウェアと操作者や他システムとの外界のインタフェース仕様(Interfaces external to the software item)，データ定義やデータベースに対する要求項目(Data definition and database requirements)を検討する.

具体的な成果物としては，ビジネスプロセス図（業務フロー），ユースケース，画面・帳票デザイン，外部システムとの概略インタフェース図，システム機能一覧，データ項目辞書，概略 E-R 図などが想定される．

次の階層がソフトウェア方式設計プロセス(Software Architecture Design Process)で，ここでは要件をソフトウェアで実現すべき機能に変換(Transform the requirements for the software items)し，ソフトウェアで実現する機能が持つ外界とのインタフェースを設計し，加えてソフトウェア機能をコンポーネントに分割し，コンポーネント同士のインタフェースも設計(Design for the interfaces external to the software item and between the software components)する．またデータベースの最上位レベルの設計(Top-level design for the database)を行う．P,F,D の 3 層からなるソフトウェア・コンポーネントに分割するタイミングもここになる．

具体的な成果物としては，詳細なユースケース（あるいは詳細なシステム機能一覧），ソフトウェア・コンポーネント図（分析クラス図），画面設計図，外部インタフェース仕様書，E-R 図などが想定される．

3 層目がソフトウェア詳細設計プロセス(Software Detailed Design Process)で，インタフェースの詳細設計(detailed design for the interface)，ソフトウェアで実現する機能を構成するソフトウェア・コンポーネントそれぞれの詳細設計(Detailed design for each software component of the software item)，データベースの詳細設計(Detailed design for the database)を行う．

具体的な成果物としては，各ソフトウェア・コンポーネントの詳細仕様書，各ソフトウェア・コンポーネントの詳細インタフェース仕様書，詳細画面仕様書，詳細 E-R 図などが考えらえる．

最下位層がソフトウェア構築プロセス(Software Construction Process)で，ソフトウェアとデータベースの実装を行う．

下位層に下がるほど，設計情報は具体化され，プログラムコード化しやすくなるが，逆に作成コストは嵩む．したがって一般的には初期の開発コストを削減するには最上位のソフトウェア要件定義プロセスで作成した設計情報をツールの入力とするのが理想である．

しかし，このプロセスの設計情報からデータベース定義やソースコードを生成すると，データベース構造は人や他システムとのインタラクションを記したユースケースから得られる情報に頼って自動生成することになるため，パフォーマンスや拡張性を考慮したテー

ブル構造を得られない可能性がある. また, テーブルを所望の形に直接人手で改変すると, 以降はモデルベース開発に戻ることが難しくなる.

同様にソフトウェア・コンポーネントへの機能の割振り（構造設計，クラス設計）も自動化に頼るため，実現したい仕様を完全にプログラムコードへ自動生成できるなら問題は少ないが，生成されたプログラムへ人手でコード追加が必要であるとか，生成されたコード部分へも改変が必要な場合は，コードの理解に時間を要すばかりか，その後モデルベース開発を継続することが難しくなる．このプロセスではP,F,Dの3層からなるソフトウェア・コンポーネントに分割するには無理がある.

一方で，最もソースコードに近い情報量を備えるソフトウェア詳細設計プロセス(Software Detailed Design Process)で作成した設計情報を入力とする場合は，必要な情報を網羅しているためラウンドトリップ型の開発には最も向いている．しかし，このレベルのモデルを作成，維持することは最もコスト高であり，生産性向上には寄与しない.

そこで私はソフトウェア要件定義プロセスの設計情報に中間のソフトウェア方式設計プロセス(Software Architecture Design Process)レベルの情報であるデータベース構造（ER 図）とソフトウェア・コンポーネント構造（クラス図とシーケンス図）を加えモデル化の対象とした（図 4-2）.

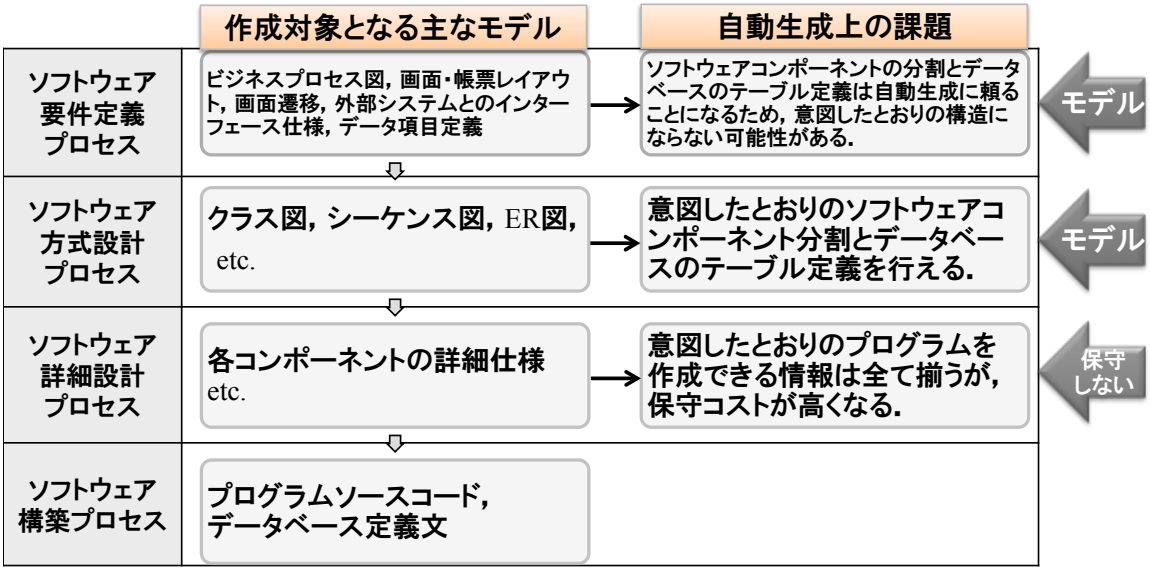


図 4-2　モデルで記述する範囲

Figure 4-2　Range described by models.

ソフトウェア要件定義プロセスにおいては，ソフトウェアで実現するドメイン特化機能

と外部とのインタフェース(Interfaces external to the software items)を表すビジネスプロセス図，画面・帳票レイアウト，画面遷移図，外部システムとのインタフェース仕様と，データ項目の名称，型，仕様を整理したデータ項目定義(Data definition)をモデルとする（図 4-2）．これらは日本において業界標準的な位置づけで良く知られている表記法[38]と類似のもの[39]を採用している．これらの採用によりユーザ，ベンダ双方ともモデルベース開発導入のための新たな学習コストを最小限に抑えられる．繰り返しとなるが，これらに加えソフトウェア・コンポーネント間の構造(Between the software components に該当)を明示的に示すため，ソフトウェア方式設計プロセスの設計情報であるクラス図とシーケンス図，それに ER 図(Design for the database)を想定する．

このように UML の採用は必要部分にとどめ，大半がこれまでエンタプライズシステム開発において日本で利用されてきた書式をモデル記述に採用した．UML だけに固執せず適所に使う方針は海外においても同様の傾向を見せている[40]．

ソフトウェア詳細設計プロセスで設計されると想定される設計情報は全てをモデル化の対象とはせず，大半を開発ツール上で設計する．これらは自動生成が効果的な部分である．他は直接手作業でコードへ反映する．具体的に自動生成対象とする部分は 4．2 で述べる．

4.1.2　モデルベース開発と反復型開発の両立

モデルベース開発と反復型開発は，適用時にアプローチとして対立する部分がある．

モデルベース開発はモデルを主とし，極力ソフトウェアをモデル記述で開発しようとする試みであり，機能追加や保守フェーズにおいてもモデルを修正することで機能実現を図るトップダウン型のアプローチである．

一方，反復型開発は，アジャイルソフトウェア開発宣言憲章[41]にみられるようにドキュメンテーションよりもソフトウェアを動かすことに注力する傾向にあるため，ラウンドトリップ機能をサポートした開発ツールを採用する例も多い．これらは主に UML とソースコードのいずれかに変更があった場合に，他方にその修正を自動反映するものである．これらのツールは多少の制約があるものの，UML もコードも自由にプログラマが変更可能である．開発初期には UML ベースで開発を進めるが，開発の後半や追加開発，保守においてはコードを主として修正し，UML への反映は，ドキュメント保守自動化の意味合いが強い．つまりボトムアップ的な利用が一般的である．

本研究が達成したい目標は，モデルを原本として，モデルの再利用により生産性と品質を向上させることを主眼とするモデルベース開発の実現を狙っている．しかし，一方であらゆる処理をモデルで記述することは効率を削ぐこともよく知られた課題である[42].

モデリングで進めた方が効率が良い作業と，手作業によるコーディングで進めた方が効率が良い作業を明確に分離し，行きつ戻りつ反復的に行った方が効率的である．そのためにはモデルベース開発を主としながらも，開発効率を極力削がないよう部分的にソース中心のボトムアップ的な反復型開発の柔軟性を取り入れる必要がある．

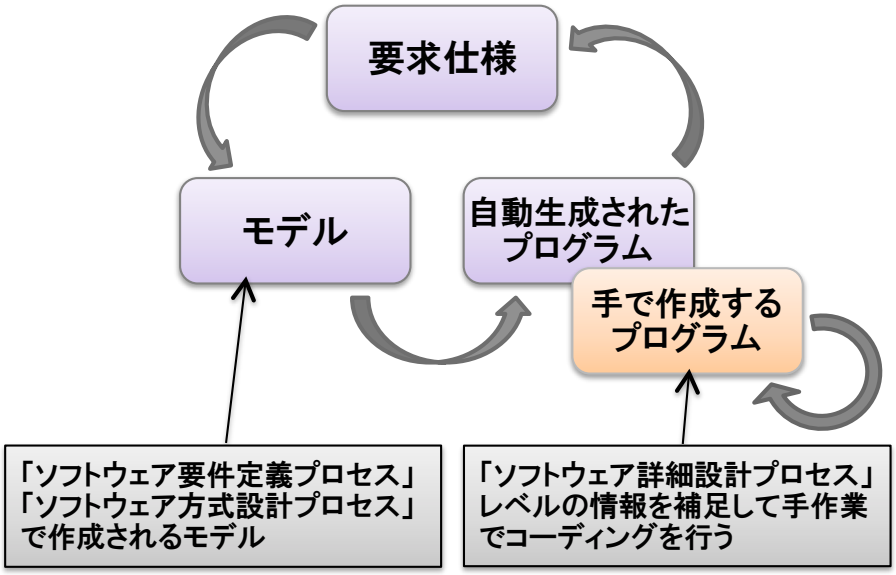


図 4-3　新しい開発プロセスの外観

Figure 4-3　General view of new development process.

図 4-3 は，新しい開発プロセスの外観を示している．モデルベースの開発は，要求仕様の作成，モデリング，モデルからプログラムの自動生成の順で開発サイクルを回す．新たな機能の追加や仕様変更は要求仕様の定義からスタートして同様に開発プロセスを回す．

ここでモデル化の対象となるのは，4.1.1，図 4-2 で示したようにソフトウェア方式設計プロセスまでの範囲になる．それ以上の詳細化はもう一つの反復プロセスである手でのプログラム作成で実現される．

図 4-4 はモデルとコードの関係を中心とした開発の流れを定式化した．開発の流れをモデルと，実行環境となるフレームワーク，ライブラリ，モデルから生成されるコード，手で追加するコードの関係で示している．

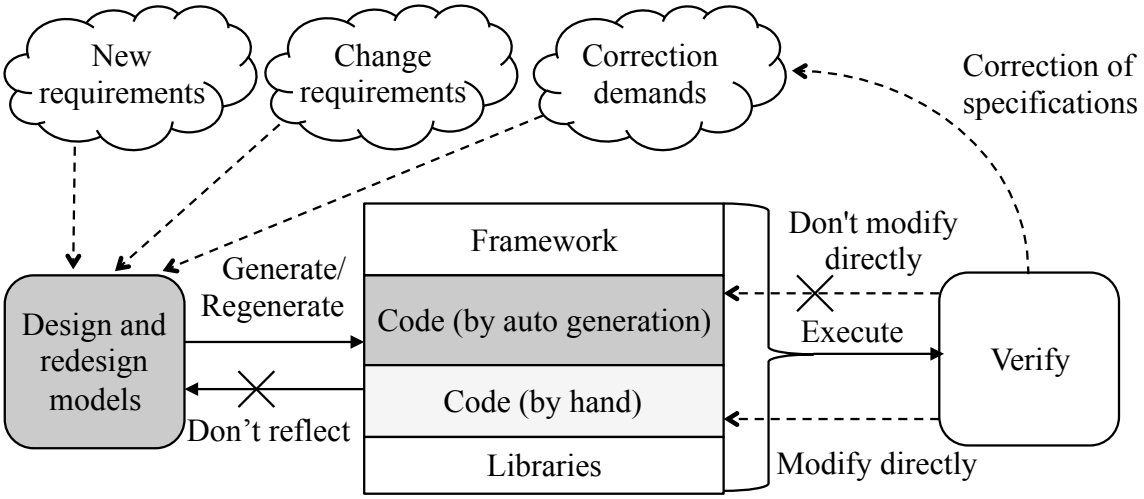


図 4-4　モデルとコードを中心とした開発の流れ

Figure 4-4　Flow of development with a focus on the models and code.

自動生成されるコード(Code (By auto generation))はソフトウェア要件定義プロセスとソフトウェア方式設計プロセスで設計された部分とソフトウェア詳細設計プロセスで設計された一部が具現化される．手作業でコード化した部分(Code (By hand))はソフトウェア詳細設計プロセスで設計された残りの部分となる．

フレームワーク(Framework)には，エンタプライズシステムに必須の制御機能をあらかじめ実装する．言語毎に実装方法は異なるが，例えばJavaであればアプリケーションサーバ，COBOLであればOLTPモニタなどミドルウェアを前提に，詳細なログ出力やソフトウェア・コンポーネント間でのユーザIDやデータベースのセッション情報引渡しなど不足機能部分を本ツールが提供するフレームワークが補足する．

ライブラリ(Libraries)は，エンタプライズシステムでよく利用される日付，数値編集処理など予め用意し，モデルから生成されたコードや手で追加されたコードから呼び出して利用する．

一般的なラウンドトリップ機能を備えたツールとの大きな違いは，手で追加されたコード(Code (By hand))をモデルに反映しない点にある．なぜならば手で追加されたコードは4.1.1で述べたソフトウェア詳細設計プロセスで追加された仕様に限定されるからである．モデル(Models)はほとんどがその上位にあるソフトウェア要件定義プロセス，ソフトウェア方式設計プロセスの情報を表現しており，自動生成されたコード(Code (By auto

generation))とのみ同期(Generation/Regeneration)をとる.

手で追加したコードはモデルに捕捉されない(Don't reflect)ため，ソフトウェア詳細設計レベルの仕様追加やバグを含む修正作業は大部分がソースに対して直接行える(Modify directory)のでプログラマの開発作業を阻害しない. ソフトウェア詳細設計レベルの設計書やモデルは，ソースとほとんど情報量が変わらないため，保守フェーズではJavadocなどに代表される保守ドキュメント生成ツールの利用が一般化しており，複雑なロジックを整理する条件表や状態遷移図などのみが手で維持する対象となる．このことを考慮すれば，この方針は合理的といえる．またモデルから生成されたコードに対する直接の修正もモデルには反映させないし，作業自体を禁止とした(Don't modify directly).

禁止とするのは，モデルがソフトウェア要件定義やソフトウェア方式設計レベルの仕様を表現しているからであり，仮に生成されたコードに修正が必要な場合，それはシステム要求レベルでの仕様の追加，変更やミス，洩れである可能性があり，単にコードの修正では済まず，ユーザや他システムとの整合性をチェックする必要性が出てくる．加えてソフトウェア要件定義レベルのモデルとプログラムコードとは１：１に対応付くとは限らず，コードの修正も問題とする１か所の修正が，他のコードへも影響を及ぼす可能性があることはよく知られている．仕様レベルの調整を要する修正をコードに対し各プログラマが併行して無作為に行うと，モデルレベルで仕様の不整合をもたらす危険性も出てくる.

そのため，モデルから生成されるコード部分の修正は，モデルの修正によってのみ反映させるトップダウン型のアプローチを採用した.

モデル上で機能に修正が加えられた場合は，モデルから生成されたコードは修正が反映されるが，手作業で作成していたコードは以前記述していたものがそのまま再現される. よって再現されたコードをモデルの修正意図に沿うよう見直さなければならない.

それを支援するため本ツールでは，モデルから生成されたコードと手作業で加えたコードがプログラマから区別可能なように，コード生成と同時に明示的にコメントが生成される．手作業でコード追加が可能な部分を早期に特定出来る.

加えて各言語が持つモジュール化機能（例えばターゲット言語がCOBOLであっても外部サブルーチンとして）を利用して，自動生成されたコードは極力完成されたモジュールとして生成する．そのためプログラマは必ずしもプログラム全体を読む必要はなく，モデルや仕様書からモジュール構造を想定し，該当モジュールの機能を記したコメントを読んでいけば，早期に全体を把握できる.

4. 2　ソフトウェア自動生成機能

4.2.1　生成されるソフトウェアが守るポリシー

既にソフトウェアの基本構造は，第3章3. 2で詳しく述べた.

図3-19，3-20で示したように，3層ソフトウェア・コンポーネントのポリシーと基本構造を持つ.

加えて，以下のようなソフトウェア実装上のルールを設けた.

- ツールが提供するフレームワークやライブラリ内部を除いてコンポーネント内でのグローバル変数利用は禁止し，ソフトウェア・コンポーネント間で機能分担し，インタフェースを通して連携することで所望の処理を実現する.
- データリソースへのアクセスはD層に限定する.
- 実装にオブジェクト指向言語を利用する場合，フレームワーク自身の実装以外で継承を使わせないという厳しい原則も採用した.

理由は，無秩序な継承は可読性と保守性を損なうからである．既存のクラスに機能拡張を施す場合でも既存モデルを修正するか，新たな機能をモデリングして新しいソフトウェア・コンポーネントとして実現することを推奨している.

上記の厳格な実装ポリシーにより，生成される各ソフトウェア・コンポーネントは役割がより明確になる.また手で追加されるコードもこの各コンポーネントに内包されるため，これらの原則に縛られる.

多人数で仕様の追加，変更や，コーディングの追加，変更を繰り返す反復型の開発スタイルにおいて，不用意なコードの毀損やバグの混入を低減させる効果がある.

4.2.2　自動生成の全体像

第3章で定義した3層のプログラム構造は，プログラム自動生成もP層，F層，D層の

3 層構造を強く意識して行う.

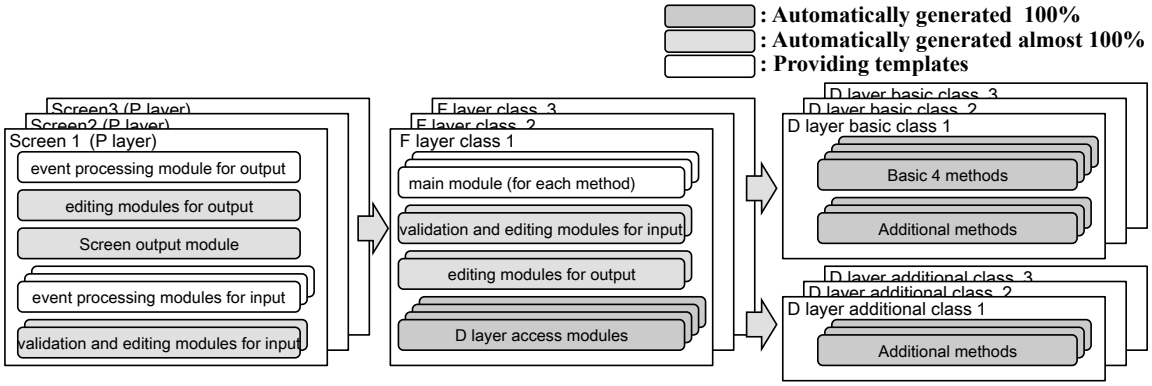


図 4-5　自動生成されるプログラムの全体像

Figure4-5　Whole image of program generated automatically.

4. 1 で述べたように，全ての処理をモデルから自動生成するのではなく，ソフトウェア方式設計プロセスまでに設計された部分は自動生成で，ソフトウェア詳細設計プロセスで設計した部分は手コーディングでとしたため，図 4-5 に示すように，プログラムの全ての部分が自動生成されるわけではない．ただし，D 層は 100%の自動生成を行う．その理由は 4.2.4，4.2.5 で述べる.

4.2.3　ソフトウェア・コンポーネント基本処理部分の生成

私がコード自動生成で最も重要視したのは，3. 2 で定義した 3 種類のソフトウェア・コンポーネントの基本構造を生成することである.

第一はコンポーネントの顔にあたる部分，例えば P 層ならばイベントハンドリング，F 層，D 層であれば他コンポーネントから呼び出されるメインモジュール部分である.

これらの部分は，図 4-5 ではテンプレート(Template)と表記したとおり，内部処理はサンプルコードを生成するだけで，必要な処理は開発者が手で実装する必要があるが，自クラス・メソッドのインタフェースは顔であり要である.

図 4-6 は F 層の例だが b)の部分が該当する．ここは外部に公開する自クラス・メソッドのインタフェースと他の内部モジュール(図 4-6 の c)，d)，e)),あるいは外部モジュールとして提供される f)などの呼出しコードが自動生成される．外部インタフェースはソフト

ウェア要件定義プロセスを経てソフトウェア方式設計プロセスで具体的な名前と型を持つ変数群にブレークダウンされる.

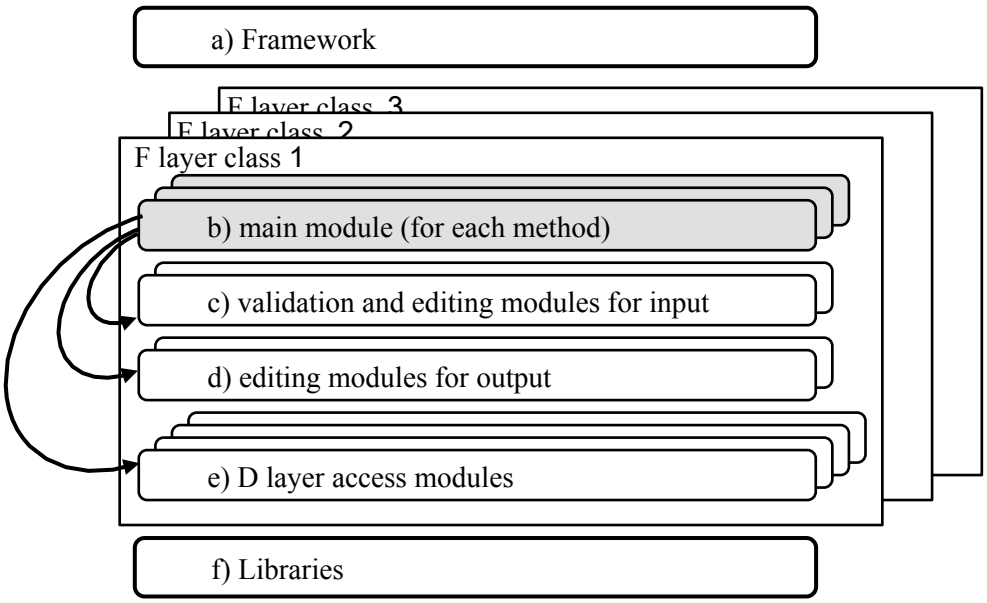


図 4-6　ソフトウェア・コンポーネントの構造 (F 層)

Figure 4-6　Internal structure of the software component. (Layer F)

第二は，入力データのチェック，編集処理(validation and editing modules for input)と出力データの編集処理(editing modules for output)であり，図 4-6 では c)と d)にあたる. データベースや外部インタフェース情報などから収集したデータ項目の名称，型，仕様を整理したデータ項目定義(Data definition)を利用して生成する．仕様とは主に入力形式，出力形式，内部形式，値域をいう.

第三に他クラスの呼出し部分の生成であり, 図 4-6 では e)が該当する. これはソフトウェア方式設計プロセスで設計されるクラス図，シーケンス図から P，F，D 層の各ソフトウェア・コンポーネント間の呼出し関係を解析し生成する.

最後が制御機能である．標準的なエラー処理，詳細なログの取得，データベースや ID 等の受け渡しなどを自動で生成する．地味ながらこれらの自動生成による提供がトラブル時の強い味方となり，保守性の向上をもたらし，長期の保守を可能とする.

制御機能の実装は，COBOL 言語の場合は極力自動生成で b)に実現し，オブジェクト指向言語の場合は主に a)のフレームワークにより実現する.

またSOAPなどの外部連携用のアダプタやテスト用のスタブ，ドライバ，テストスクリプトの生成も行う．以上によりアプリケーション開発者は機能実装に集中出来るようになる．

4.2.4　アプリケーション機能部分の自動生成方針

次に機能部分の生成であるが，モデルベース開発においては，可能な限りコード自動生成を行い，コーディングの量を減らすことで生産性向上を狙うのが基本的な考え方である．しかし現実にはコード自動生成率を限りなく 100%に近づけようとする，あるいは望む形のコードに近づけようとすると，モデリング作業の負担が増し，モデリングとコーディングの実質的な作業内容，作業量に差がなくなってしまう．所謂モデルベースのジレンマであるが，私はソフトウェア開発プロジェクトを進めるうえで組織として効果的な部分からモデルベースによる自動生成を導入することとした．

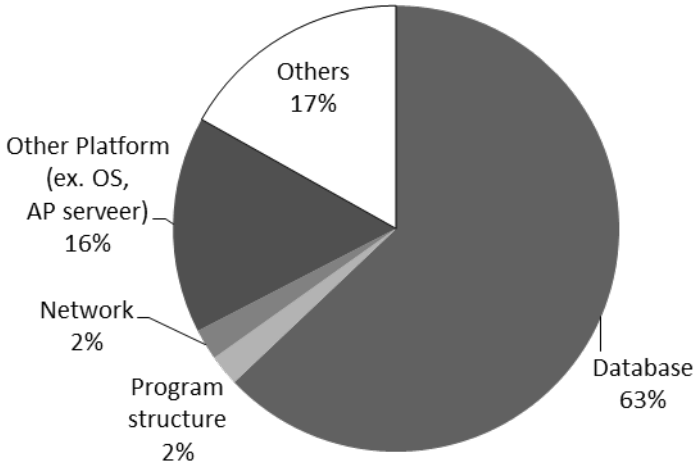


図 4-7　専門家チームの対応理由

Figure 4-7　The reason that expert teams were called

図 4-7 は我々の組織が収集しているデータで，プロジェクトでは自力解決に至らなかった技術的問題を表している（特定製品の機能に関する問合せは除いている）が，データベースに関する問題が圧倒的に多いことが分かる．エンタプライズ系のシステムは主にデータベースとのインタラクションによって成り立っており，高機能化が進むデータベースをう

まく使いこなせるかどうかがシステム開発における技術的なポイントとなっている．そのためD層の自動生成が最重要であることが分かる．

4.2.5　D層の自動生成

D層が一般的なDAO(Data Access Object)と異なる点は，3．2で述べたように，D層がデータアクセスに関する処理を全て内包する点にある．

一般に流通している低機能なDAOは単純なデータアクセス処理をDAOのフレームワークが提供する．しかし複雑なSQLは外部からSQL文を挿入する，あるいはDAOの管理外で実行させる例が多い．一方，D層はモデルベースの開発環境下で設計したSQLのみを扱い，SQLの外部挿入や管理外での実行を許さない．つまりD層を介さずにデータベースをはじめとしたデータリソースへのアクセスは出来ない構造となっている．

これはD層にリソースへのアクセス処理を集中させることで，データベースの利用方法に起因するソフトウェア開発上の問題が発生した際，処理の流れを可視化し，問題個所を素早く特定可能とする．加えてD層の開発状況をモデル上で集中的に管理することで開発の進捗も可視化可能となり，プロジェクト運営上大きなメリットを生む．

SQLコードを書くための設計は最終的にソフトウェア詳細設計プロセスで行われる．本ツールでは，基本的に自動生成されるコードはソフトウェア要件定義プロセスとソフトウェア方式設計プロセスで設計された部分であるが，SQLは例外であり，詳細設計情報から自動生成する．

D層はほぼ図4-6に示すF層と同様の構造を持つ(唯一e)は存在しない)が，自動生成されたSQLはb)のメインモジュール内に配置される．D層は特に独自の機能追加が必要なければ，クラスとして完成されているため，手でコードを追加する必要なく利用できる．

4.2.6　F層の自動生成

F層は主に複数のD層アクセスをまとめるFaçade(入口，玄関のような役割)，としての役割を担う．F層で生成されるコードは4.2.3に述べた通りだが，ビジネスルールを手で実装する前提に生成されている．ほとんどのケースでは図4-3のb)に直接コーディングする例が多いが，内部モジュールとして手で記述することも，外部モジュールを予め作成し，

他のF層を呼出すようモデルとして定義して生成することも可能である.

外部モジュールを呼出す情報はクラス図とシーケンス図で得られるが，モデルで記述するメリットは, この外部モジュールが他のF層コンポーネントと共用可能となる点にある.

F層には外部から提供されたライブラリの呼び出しや他システムの呼び出しなども統合する.

4.2.7 P層の自動生成

P層については，Java版においてStrutsをベースにした画面レイアウト実装の自由度が高い自動生成機能を標準提供している.

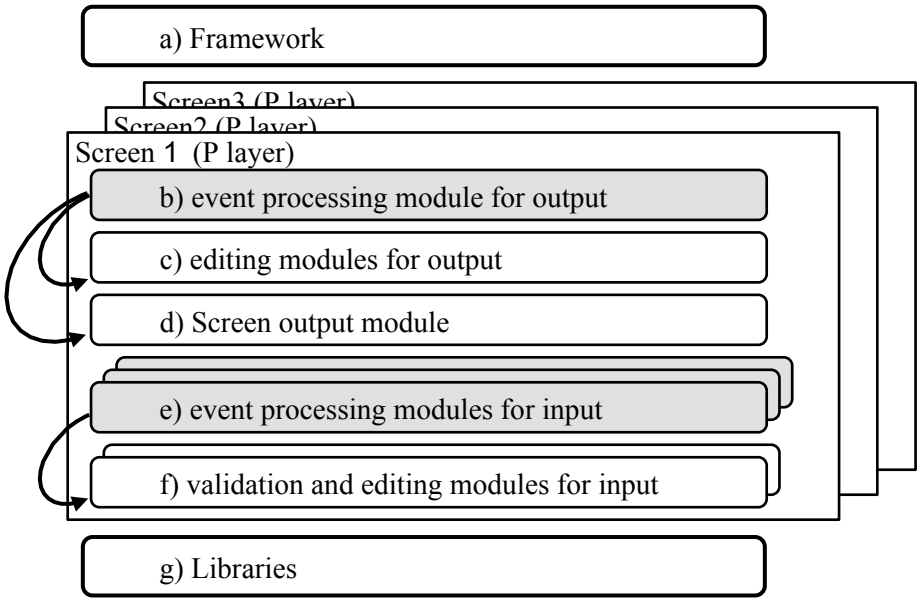


図 4-8　ソフトウェア・コンポーネントの構造 (P層)

Figure 4-8　Internal structure of the software component. (Layer P)

構造はF層と異なり，メインモジュール相当の部分は，外部へ情報を表示する出力イベント処理(event processing module for output)と外部から入力される情報を処理する入力イベント処理群(event processing modules for input)が並ぶ（図 4-8）.

出力イベント処理の延長上で出力データの編集処理(editing modules for output)と画面出力モジュール(Screen output module)が呼出される．入力イベント処理の延長上で入力

データのチェック，編集処理(validation and editing modules for input)が呼出される．

画面遷移は Struts に任せるため図上は a)となる．b)は a)の遷移情報に基づいて呼出される．e)は外部からのイベント（ブラウザからの入力）により呼出される．

P 層は近年なら，例えば RIA(Rich Internet Application)やスマートフォンなどの採用例も多く，技術が多様で変化も激しく，要求レベルもさまざまであるため，1 ツールで要求に応えるには限界がある．そのため F 層のアクセスインタフェースを SOAP に見せるなど他システムやデバイスとの接続性を確保するオープン化戦略をとった．COTS として提供される多くの画面作成のためのサードベンダ製品を利用できる．

4．3　まとめ

この章では，4．1 ではじめにモデルベースで開発する範囲を一般的な開発プロセスの標準モデルから導きだし，そのプロセスの中でモデルベースによる反復型開発と手作業による反復型開発の境界を明確に示すことで，モデルベース開発と反復型開発の共存を可能とした．開発ツールが生成する部分はモデルベースの反復型開発を行い，手作業でコーディングする部分は手作業で反復型開発を行う．

4．2 ではモデルベースの開発ツールが生成するソフトウェアの機能について，ツールが一貫して生成する基本部分と，P,F,D 層の各々で特徴ある生成部分を明確化した．

本章で提案した開発プロセスを前提とし，本章で提案したソフトウェアの自動生成を実現するモデルベース開発ツールについては第 6 章で述べる．

第5章　モデルベース開発ツールを前提としたソフトウェア規模評価方法の提案

大規模なソフトウェア開発を伴うシステム開発においては，ソフトウェア開発の見積精度を如何に向上させるかが課題であることを 2．3 で述べた.

FP と LOC いう尺度が以前から用いられているが, FP は客観性が高く, 比較的プロジェクトの前半から利用可能である点は優れているが，適用に工数がかかる点と，ソースコードとの対応が必ずしも容易ではないためプロジェクト後半で利用しにくい.

一方でLOCはソースコードから直接計測するためプロジェクト後半で利用しやすいが, 経済的な規模を表現する方法としては客観性に乏しく，加えてプロジェクト前半では利用できない.

以上のことから，お互いの長所を活かす形でプロジェクトの前半を FP で，後半を LOC でソフトウェアの規模管理が出来れば良いが, 従来, 新規ソフトウェア開発においては FP から LOC への変換は必ずしも精度が高くないという問題があった.

これまで述べてきた本研究の成果であるモデルベース開発ツールは，ソフトウェアの大半をモデルからの自動生成を使って得るため, LOC の最大の難点である客観性の低さを解消できる可能性があり，変換精度の向上に取組んだ.

5．1　FP の構成要素

FP はエンタプライズ系アプリケーションの計測向けに ISO/IEC 20926 として IFPAG 法[44]が規格化されており，計測の専門家を養成することで客観的にソフトウェア規模を計測できる.

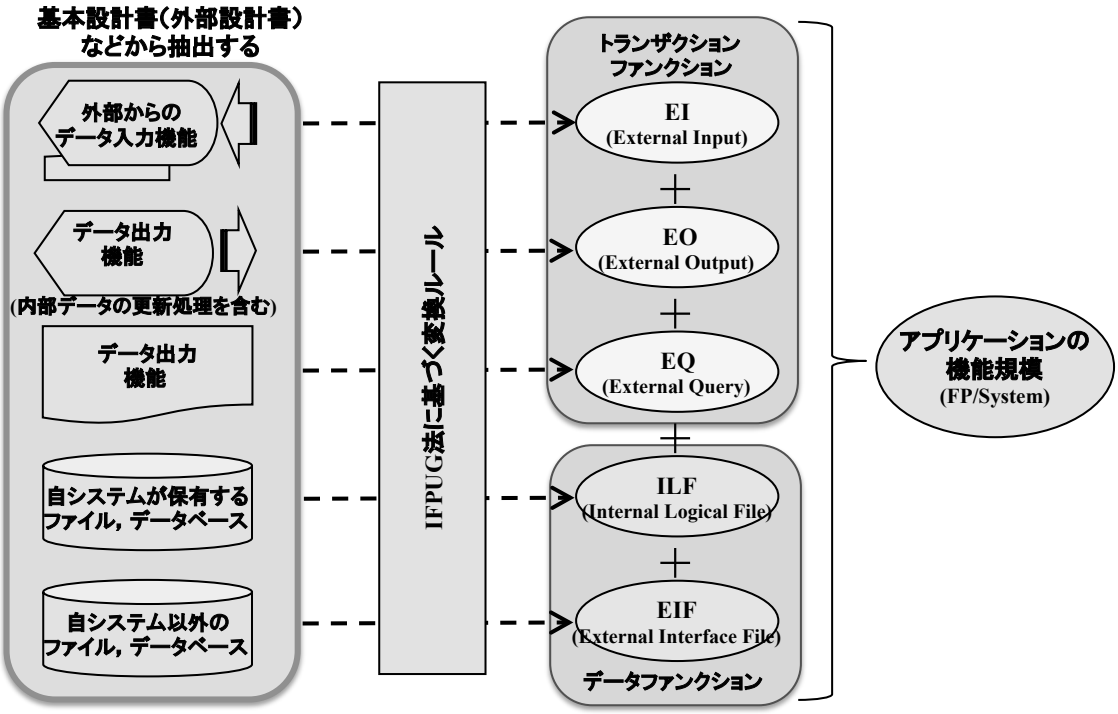


図 5-1　IFPUG 法の構成要素

Figure 5-1　Component of IFPUG method.

IFPUG 法は，トランザクションファンクションとデータファンクションから構成される.

トランザクションファンクションは，人や外部システムとの接点に注目し，対象となるシステムとのやりとりを洗い出す．外部入力処理(External Data Input Function)から算出される EI(External Input)，外部出力処理(Data Output Function to External)から算出される EO(External Output)，外部照会処理(Data Query Function to External)から算出される EQ(External Query)の 3 種類が定義されている.

外部出力 EO と外部照会 EQ は同じくシステムの外部に対して情報を提供することが主務だが，EO は外部からの要求に呼応する処理も同時に実行し，システム内部のデータに対して変更を加えている(Maintenance Internal Data).

一方，データファンクションは，当該システムに関連する DB やファイルに注目する．DB やファイルの数え方は，物理的なテーブル数やファイル数（レコードの種類）と論理的なファイル数の両面で捉える必要がある.

論理ファイルを，システムが内包し維持する DB，ファイル(with Maintenance)である

ILF(Internal Logical File，内部論理ファイル)と，参照のみで維持対象ではない DB，ファイル(don't Maintain it in This System)である EIF(External Interface File，外部インタフェースファイル)の 2 種類に分類して扱う．

FP は，基本的にトランザクションファンクションの総数とデータファンクションの総数の和で構成される．最終的にはシステム特性を数値化した 14 の項目に対する評価係数を掛け合わせることで，最終的なファンクションポイント(調整済ファンクションポイントと呼ばれる)を算出するが，システム特性の扱いについては，未だ議論が多く，本論文では調整前ファンクションポイントを FP として扱う．

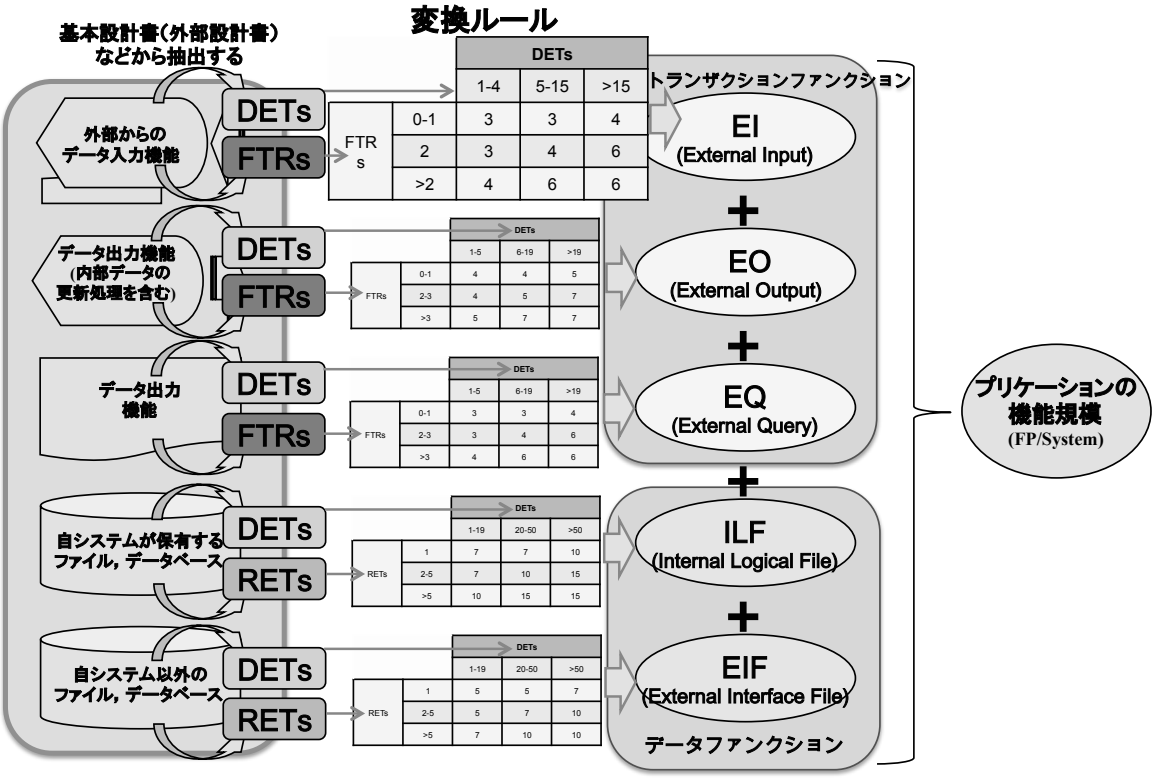


図 5-2　FP 算出の流れ

Figure 5-2　Flow of FP calculation.

もう少し細かく算出過程を述べる．

トランザクションファンクションの算出は，EI あるいは EO，EQ として認識した処理(Recognizes it to EI, EO or EQ) から，FTR(File Type References)と DET(Data Element

Type)を抽出する必要がある.

FTR は ILF と EIF から構成されるが，ILF，EIF とはデータファンクションで説明した DB やファイルの数である．ここではカウント対象とした処理，具体的にはひとつの画面やひとつの帳票, ひとつの外部インタフェースといった単位で, その処理に関連する ILF, EIF をカウントする.

DET はデータ項目のことであり，これもカウント対象とした処理，具体的にはひとつの画面やひとつの帳票，ひとつの外部インタフェースといった単位で，その処理に関連する DET をカウントする.

カウントした FTR(ILF,EIF)と DET を元に，対象とした処理が EI，EO，EQ のどれに該当するか確認して，処理の複雑さとして重みを付けた変換表から係数を求める．この係数がトランザクション FP である．IFPUG 法では 1 つの処理で 3~7 の範囲のトランザクション FP が得られる.

この作業を関連する処理全て，具体的には画面や帳票，他システムとのインタフェース全てに対して行なう.

データファンクションは，ひとつの ILF，あるいは EIF ごとに RET(Record Element Type)と DET をカウントする．RET はレコードの種類をカウントするものであり，ひとつの ILF あるいは EIF を構成する物理レコードの種類を数える.

データファンクションにおいて DET は，RET に含まれるデータ項目の種類をカウントする.

カウントした RET と DET を元に，計測対象としたファイルが ILF または EIF に該当するか確認して，変換表から係数を求める．この係数がデータ FP である．IFPUG 法では 1 つの論理ファイルで 5~15 の範囲のデータ FP が得られる.

この作業を関連する DB やファイル全てに対して行なう.

以上の説明から，FP は，以下のシステム構成要素から導き出されることが分かる.

- トランザクションの種類（外部入力，外部出力，外部照会）
- DB，ファイルの種類（内部論理ファイル，外部インタフェースファイル）
- データ項目数
- 論理ファイル数

- 物理レコードの種類
- 物理画面数，物理帳票数，外部インタフェース数

FP から LOC へ変換を実施するうえで，ここで示したパラメータ以外の要素を使うことは想定されない．つまりこれらのパラメタを使って変換を行うことが前提条件となる．

5．2　モデルベース開発ツールを前提とした解法のアプローチ

5.2.1　モデルベース開発ツールを前提とした LOC の見積と FP への変換

本研究が提案するモデルベース開発ツールは，設計情報をモデルとして表現し，フレームワークやライブラリ，プログラム自動生成技術を用いて，モデルとソースコード間を反復的に繰り返して開発することで，設計や実装工程の省力化，高品質化を目的としている[43]．

モデルベース開発を行なうと実現したい機能の大部分はモデルからの自動生成でカバーされ，人手によってソースコードを記述すべき部分は限定的になる．

また，人手によって実現すべき部分についてもコーディングする位置を明確に特定し，手コーディングの方法にも制約を設けることで個人差によるコーディング上のばらつきが小さくなると期待される．またオブジェクト指向言語の継承，アスペクト指向といった LOC 計測上の課題は，これらの利用をフレームワーク側のみに制限してしまうことで隠ぺいする．

一時隆盛を極めたビジュアル言語は現在，大規模システム開発に限ればユーザインタフェースの改善にのみ利用するのが一般的となり，局所的な利用にとどまっている．

以上のようにモデルベース開発ツールの利用を前提とすれば，ソースコードがどの部分にどのように展開されるか想定しやすくなるため LOC の計測は見通しやすいと期待される．

加えて，どのモデルからどの部分のソースコードが生成されるかも明らかであることから，仮に FP とモデルの関係が明確化出来れば，FP で計測した値がどの部分のソースコードと密接に関係が深いかも明確になると考えられる．

つまりはFPからLOCへの変換精度も高めることが可能と考えられる.

5.2.2 モデルベース開発ツールが生成するソースコード

図5-3にJavaのWebシステムをターゲットとしたP,F,D層各コンポーネントに生成する内部機能構造を示す.

D層はツールによって全ての機能が自動生成される．P層，F層も多くの機能が自動生成されるが，P層のイベント処理(event processing modules for input)，F層のメイン処理(main module)らは追加コーディング前提のテンプレートのみを生成する．このイベント処理やメイン処理にコードを追加して，画面出力(screen output modules)，検証や編集(validation and editing modules)，D層アクセス(D layer access modules)などの処理を呼び出し，目的の機能を完成させる.

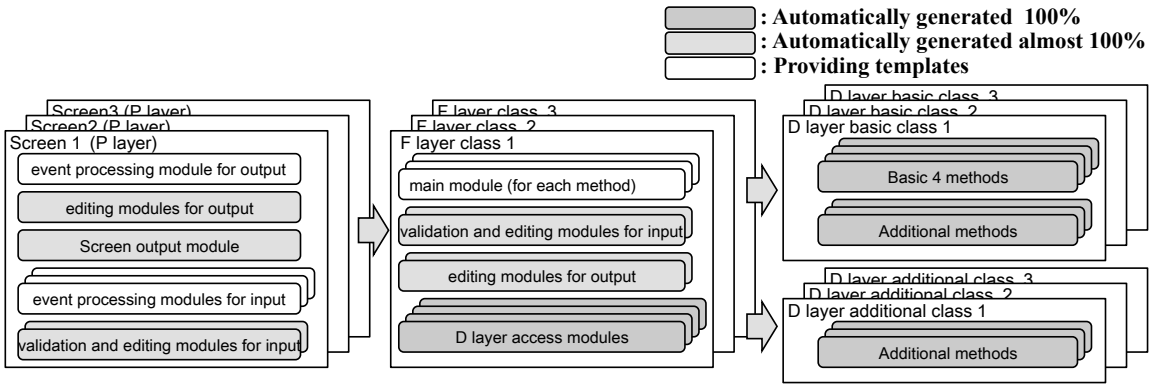


図 5-3　自動生成されるプログラムの構造

Figure 5-3　Automatically generated program structure.

このようにプログラム構造がコンポーネント化と自動生成によって明確かつ固定化されており，故意にプログラマが書き変えない限り，プログラムの基本構造が大きく変わってしまうことはない．よってFPからLOCへの変換精度は高いと予想される.

5.2.3 FPからLOCへの変換方法

FPからLOCへの変換は，コンポーネント構造を意識して行う．コンポーネントごとに

LOC を見積もることが出来れば，単にソースコードの全体量だけでなく，コンポーネント各々の完成量が明確になり，マネジメントが容易になる．

私が当初想定した FP から LOC への変換イメージは図 5-4 に示す様に，トランザクションファンクションが P 層，F 層の LOC 算出に利用され，データファンクションが D 層に対応するとした．

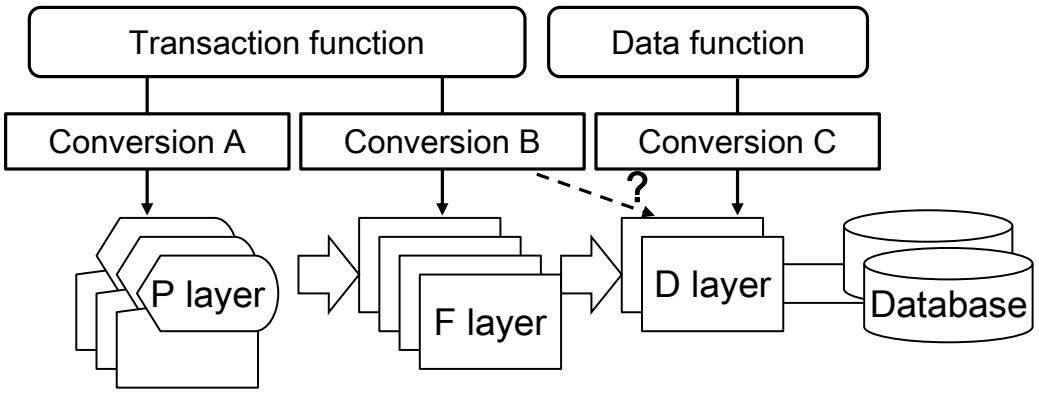


図 5-4　FP からコンポーネント別に LOC を算出する流れ

Figure 5-4　Flow to calculate LOC of every component from FP.

しかし，D 層は実際にはリレーショナルデータベースをカプセル化して実装するケースが圧倒的に多く，一般的なファイルであれば F 層で実現するデータの選択抽出，マージ，ソートといった機能を SQL で容易に実現できる．またそのような実装が今日コスト的に妥当であり，保守性も高い．つまり D 層はトランザクションファンクションの一部も実現していることを考慮しておく必要がある．

5.2.4　D 層の位置づけ

D 層の開発は，詳細設計プロセス(Detailed design process)で実質終了している．人手によるコーディングを前提としている P 層や F 層と異なり，モデルベースの開発環境内でコンポーネントとして機能するレベルまで完成している．検索や照会処理を SQL 定義ツール(SQL definition editor)で設計している時点で機能確認まで行えば，以降の開発プロセスでコーディング作業や個別に機能デバッグを実施する必要がない．モデルベース開発ツールが生産性と品質向上に寄与する一端を示している．

同時にデータファンクションに関しては LOC への変換精度向上に取組む必要性が薄いことも示している.

5. 3　提案方式の課題と対応

実際のプロジェクトデータを用い，残された P 層，F 層について FP から LOC への変換を試みた. ここではF層コンポーネントへの変換時に発生した問題と解決方法を述べる.

5. 1 に述べたとおり，トランザクションファンクションには EI,EO,EQ の 3 種類あり，それぞれのトランザクションが含む DET と FTR による重み付けで FP を決定する.

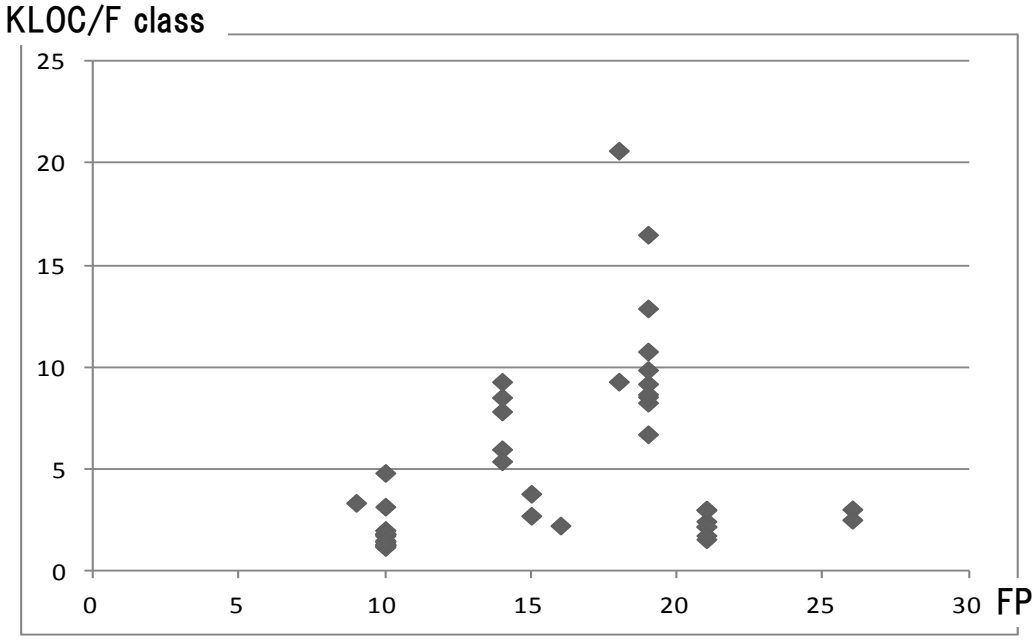


図 5-5　FP と LOC の相関

Figure 5-5　Correlation between FP and LOC.

図 5-5 は実際のプロジェクトデータの F 層をクラス単位に FP と LOC でプロットしたものだが，想定よりもかなり相関が低く，単純に FP から LOC への換算ではマネジメントに利用できるほどの精度がないことが判明した.

IFPAG では，1 つのトランザクションファンクションで想定する FTR と DET の数は各々せいぜい 3 と 16 程度であるが，この図 5-5 のシステムでは，FTR で平均 4 を超え，DET は 40 を超える. 最大はそれぞれ 14 と 128 であり，IFPAG 法の想定を大きく上回っ

てしまう.

そこで2つのアプローチをとって解決策を探った.

一つは実測LOCを基準としてFPより相関の高いパラメタを探すこと. もう一つはソースコードを調査して想定していない実装やデッドコードが存在しないかチェックすることである.

一つ目のアプローチである相関の高いパラメタの調査であるが, トランザクションファンクションの場合, トランザクションの種類(EI,EO,EQ)とFP算出に利用されるFTRとDETが候補となる. これら以外のパラメタは新たな調査作業が発生するため, 今回のケースでは採用できない.

以下にEIについて2つのパラメタとLOCとの相関を示す.

Regression analysis	Explanatory variable	Correlation	Coefficient
Simple linear regression	**DET**	0.62	8.11
Simple linear regression	**FTR**	0.20	
Simple linear regression	**DET×FTR**	0.49	
Multiple regression	**DET**	0.66	8.16
	FTR		-8.00

図5-6　EIタイプの相関分析

Figure 5-6　Regression analysis for the EI type.

図5-6は図5-5で示したデータ群のうち, EIタイプのトランザクションファンクションを中核とするクラス群のFTR, DETあるいは両方とLOCとの相関を調査したものである. DET単体, あるいはDETとFTRとの重回帰に高い相関性が見られた. FPはもともとDETとFTRから算出されるので, 今回も重回帰の値は納得出来るものだが, DET単体でも十分な相関が得られた. 重回帰ではFTRの係数がマイナスとなっており実感と合わない.

以上より, このコンポーネント群ではDETをパラメタ候補とした.

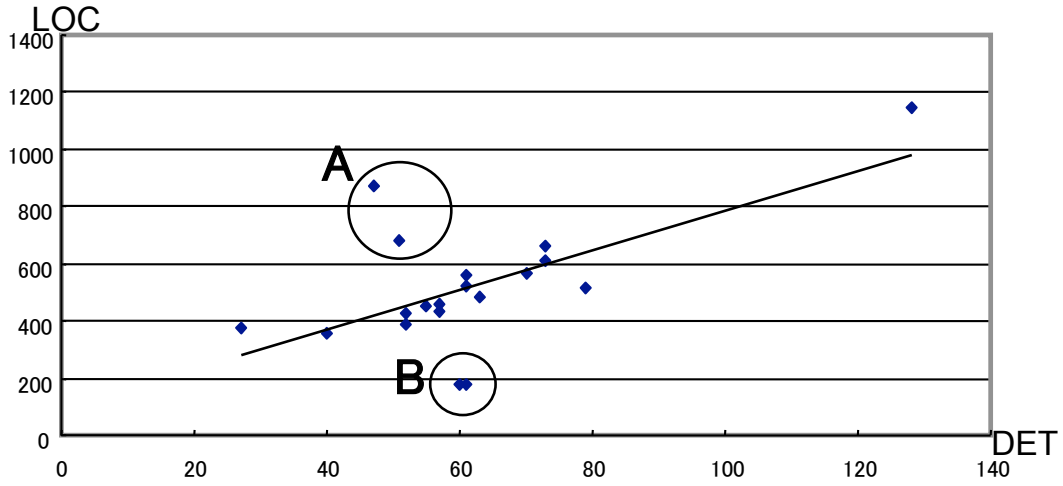


図 5-7　EI タイプの DET と LOC の相関

Figure 5-7　Correlation between EI type DET and LOC.

図 5-7 は実測値と図 5-6 の DET の単回帰分析の結果をグラフにプロットしたものである．回帰式は以下となる．

$$y = 8.11x + 17.33$$

図 5-7 には A，B として大きな外れ値を示した．これらを改善できれば，変換精度はおおいに向上する．そこで A,B に含まれる 4 本のコンポーネントを調査した．

A に含まれる 2 本のうち 1 本は，複雑なビジネスロジックを含むものであった．FP はこの点万能ではなく，計測時に設計情報から注意深くこのような処理を別途カウントする必要がある．残りの 1 本はデッドコードやクローンコードが多く発見された．加えてクラス内のメソッド構成も見通しの悪い構造となっていた．このプロジェクトは開発標準がきっちり決められていたが，それに照らすと違反したプログラムということが出来た．このように可視化することで潜在不良を抽出できるという知見を得た．

B の 2 本は，いずれも固有の処理ロジックはなく，単純に D 層コンポーネントをアクセスする Façade(入口，玄関のような役割)であった．

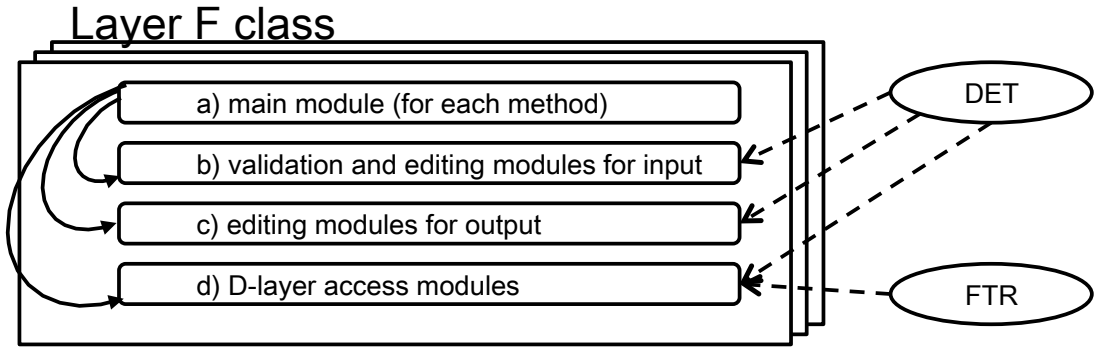


図 5-8　F 層のクラス構造

Figure 5-8　F-layer class structure.

図 5-8 は F 層のクラスを 1 つのメソッドを中心に描いたものである．実際には複数のメソッドで 1 つの F 層クラスを構成する例が多いが, 各メソッドの構造は同様である. メソッド内の処理は大きく，a. メイン処理, b. 入力データの検証, 編集, c. 出力データの編集, d. D 層のアクセス処理から構成される．このうち, DET の数に依存して LOC が増す処理が b.c.d.といえるのに対し，FTR に依存して増えると断言できるのは d.のみであり，この差が解析結果に表れたといえる.

以上は EI タイプについて述べたが，同様の結果は EQ，EO タイプについてもいえる. EO，EQ に関しては，検索結果を D 層内で Join 処理を行うメソッドから得ることにすれば，ますます FTR の数の影響を受けにくくなる．実際，このような実装方法は本モデルベース開発ツールに限らず一般的である.

このことから，EI と，EQ，EO タイプとの間の DET の係数に着目する.

Transaction type	Coefficient
EI	8.11
EO	2.65
EQ	2.58

図 5-9　DET を変数とした場合の各タイプの係数

Figure 5-9　Coefficient of each transaction type.

EO ,EQ がほぼ同等であるのに比べ EI の係数が突出して大きいのでは，FP から LOC へ単純な変換が出来ないため，理由を明確にする必要がある．

EO,EQ はいずれもデータの出力機能を指しており，FTR から目的とする DET を抽出することが主たる処理である．違いは導出データを生成するか否か，ILF を維持管理するか否かの差である．今回のモデルベース開発ツールでは，FTR から DET を抽出する処理はリレーショナルデータベースを内包する D 層コンポーネントで実現されるため, D 層内で既に選択抽出，複数テーブルのマージ(Join)，ソートなどの処理が施されている．このため EO,EQ の処理は単純化され EI との間で大きく係数の差が生じた.

これまでの分析は F 層に限定したものであったが，同じくトランザクションファンクションから LOC を道出しようとする P 層にも同じことが言える. P 層は F 層が Façade(入口，玄関のような役割)として FTR に依存する処理を隠ぺいするため，FTR の影響はなく DET の数にのみ依存する.

我々が経験した 10 ほどのシステムのデータで P 層，F 層に関し同様の分析を行ったところ，今回の分析事例と同様の結果を得ることができた．複雑なビジネスロジックや，設計ミスによるクローンコード, デッドコードの生成を除外できれば変換誤差 10%前後という見通しの良い値を得ることができる.

なお図 5-9 に示す係数は，残念ながら本研究で規定したアプリケーション構造，フレームワークの実装上の特長，採用するプログラミング言語，コーディング規約等に左右されるため一般解ではない．しかし傾向は表現できていると考える.

5. 4 まとめ

ソフトウェア開発プロジェクト, 特に新規にソフトウェアを開発するケースにおいては, ソフトウェアの規模がコストや進捗を代替的に表現可能とする重要な尺度である．モデルベースによる開発スタイルを採用した場合，ソフトウェアは構造が統制され，コーディングに制約を設けやすくなる.

モデルベース開発ツールのこの制約下では， FP 計測時に取得する係数をうまく利用することで LOC への変換精度が高くなることを示すことができた.

この技術を利用すればプログラムコンポーネント単位で規模を推定できる．これによりプロジェクトの進捗が把握しやすくなり，外れ値の解析による不良検出にも利用できるな

ど，マネジメントへも大きく寄与することを示した.

本論文における本章が達成すべき目標は，本研究が提案するモデルベース開発ツールが，ソフトウェア規模の見積精度向上に寄与するとの仮説を裏付けることであったが，達成できたと考える.

第６章　モデルベース開発ツールの実現

本章では，第３章から第５章まで述べてきた研究成果を実現したツールについて述べる．特にＰ層，Ｆ層，Ｄ層と３層独立で進める開発スタイルへの対応，マルチアーキテクチャへの対応，反復型開発の実現など特長的な機能部分の詳細を述べる．また，同時期に出現したMDAツールやORマッパーとの比較も行う．

6．1　モデルベース開発ツールの全体像

本開発ツールは，4．1で述べたとおりソフトウェア要件定義およびソフトウェア方式設計で作成されるモデルを入力情報とする．

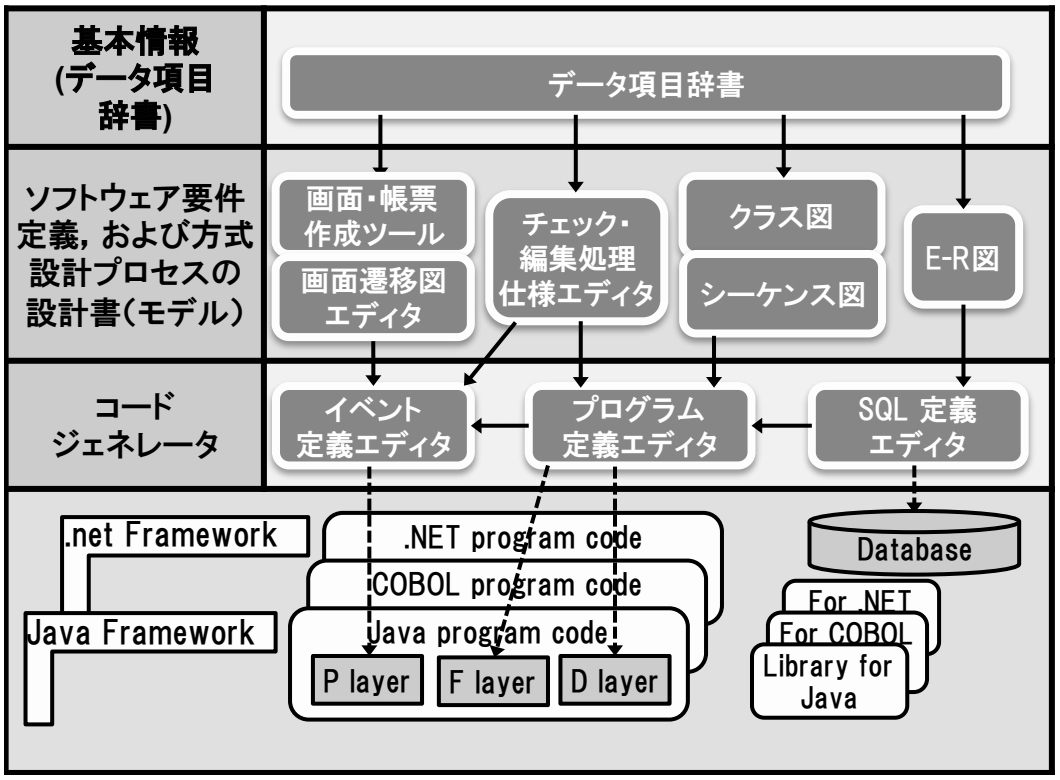


図 6-1　ツールの全体構成

Figure 6-1　　Composition of the tools.

具体的には図 6-1 に示すように画面・帳票作成ツール，画面遷移図エディタ，データ項

目のチェック・編集仕様エディタ，クラス図，シーケンス図，ER 図などを入力情報とする．データ項目辞書が各モデル間の整合性を調整する役割を果たす．ソフトウェア詳細設計プロセスの作業時にコードジェネレータ群を使って設計作業とコーディング作業の一部を自動化する．

2.1 に掲げたように，本開発ツールはバッチ処理から SOA やスマートフォン連携まで，複数のアーキテクチャを組合せて構成される現在の基幹システムの開発に対応可能である必要がある．具体的には第 3 章で示したソフトウェア構造，第 4 章で示したモデルベースの反復型開発プロセスをサポートし，コード自動生成ポリシーに従ってプログラムコードを生成出来る必要がある．

プログラムコードの生成は，P 層定義エディタ＆ジェネレータ，プログラム定義エディタ＆ジェネレータ，SQL 定義エディタの 3 種類のコードジェネレータによって実現される．以下，これらジェネレータ群を中心にプログラム自動生成の流れを以下に述べる．

6. 2　プログラム自動生成までの流れ

プログラムを自動生成するまでの開発プロセスの流れはツール間の関連に沿って，凡そ P 層開発の流れ，F 層開発の流れ，D 層開発の流れの大きく 3 つのプロセスから構成されている（図 6-2）．データ項目との整合性さえ確保されれば，3 つの開発プロセスは自動生成を行う直前まで個々独立に作業可能だが，実際には，F 層は P 層から呼び出される時のタイミングと必要とされるインタフェースを知っている必要があり，D 層に含まれるデータベース設計（E-R 図）は，P 層に含まれる画面や帳票に並んでいるデータ項目とそのライフサイクルを意識しなければ設計出来ない．さらに D 層は，F 層から要求されるデータ項目とその抽出条件を知る必要がある．このようにお互い必要とされる相手の要求を知る必要がある．

しかし，作業自体はデータ項目を共有している以外，仕掛けによる縛りがないため，横の調整が進めば並行に進めることが可能である．つまり多人数による分業を意識したツール構成と開発プロセスとなっている．

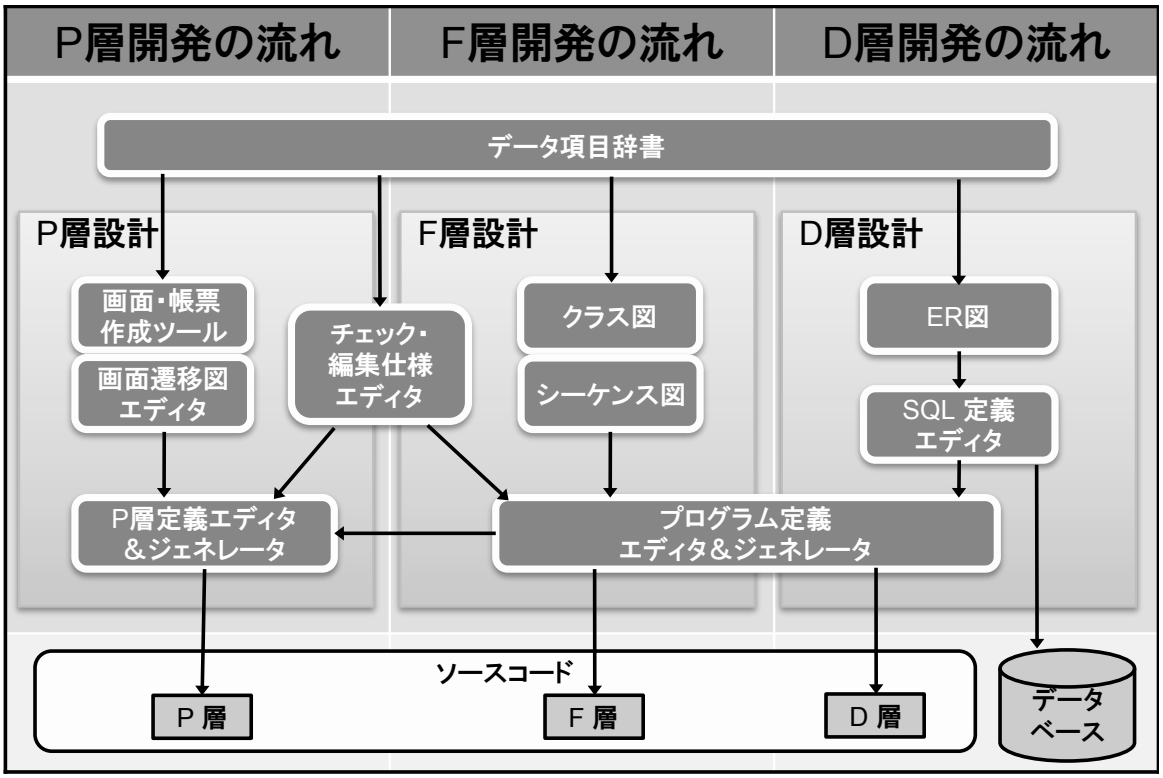


図 6-2　各層の開発の流れ

Figure 6-2　Development process of each layer.

次に，P 層開発の流れ，F 層開発の流れ，D 層開発の流れを，ツールを中心に述べる．

(1)　P 層の設計から生成までの流れ

P 層は 4.2.7 で述べたようにオープン化戦略を取っており，次々と登場するさまざまな端末，Viewer ソフトウェアなどに対応するために，サードベンダツールとの連携も可能としているが，本ツールではJava の場合，現状で最も要望の多いWeb ブラウザと，Web 環境上で表現力の高い画面を開発可能な RIA(Adobe Flex)をサポートしている．COBOL は当社の COBOL 言語パッケージ製品の開発環境を，. NET はサードベンダ製品の利用を推奨している．

以下はJava の Web システム向けの自動生成機能について説明する．

P 層の設計から自動生成に至る流れは，図 6-3 に示すように画面遷移図と画面・帳票作成ツールを起点とする．画面遷移図の情報を P 層定義エディタ＆ジェネレータに読み込ませることで，P 層プログラム群を生成するシート群が用意される．

P 層定義エディタは画面遷移図の情報から Control 部（図 3-24 参照）となる P 層定義

シートに画面遷移の流れが移送され，その流れに沿って画面出力，画面イベントが並び，それぞれの入出力インタフェースとなる ActionForm（の名称）が自動展開される．次に手作業で ActionForm 定義書上の入出力項目としてデータ項目を貼り付けていく．画面出力定義書, 画面イベント定義書には, ここで呼び出す F 層, あるいは D 層のクラス・メソッド名とインタフェースを手で記述する．また画面描画(JSP)は，画面・帳票作成ツールの情報（HTML）と該当する ActionForm 定義書から自動生成する．

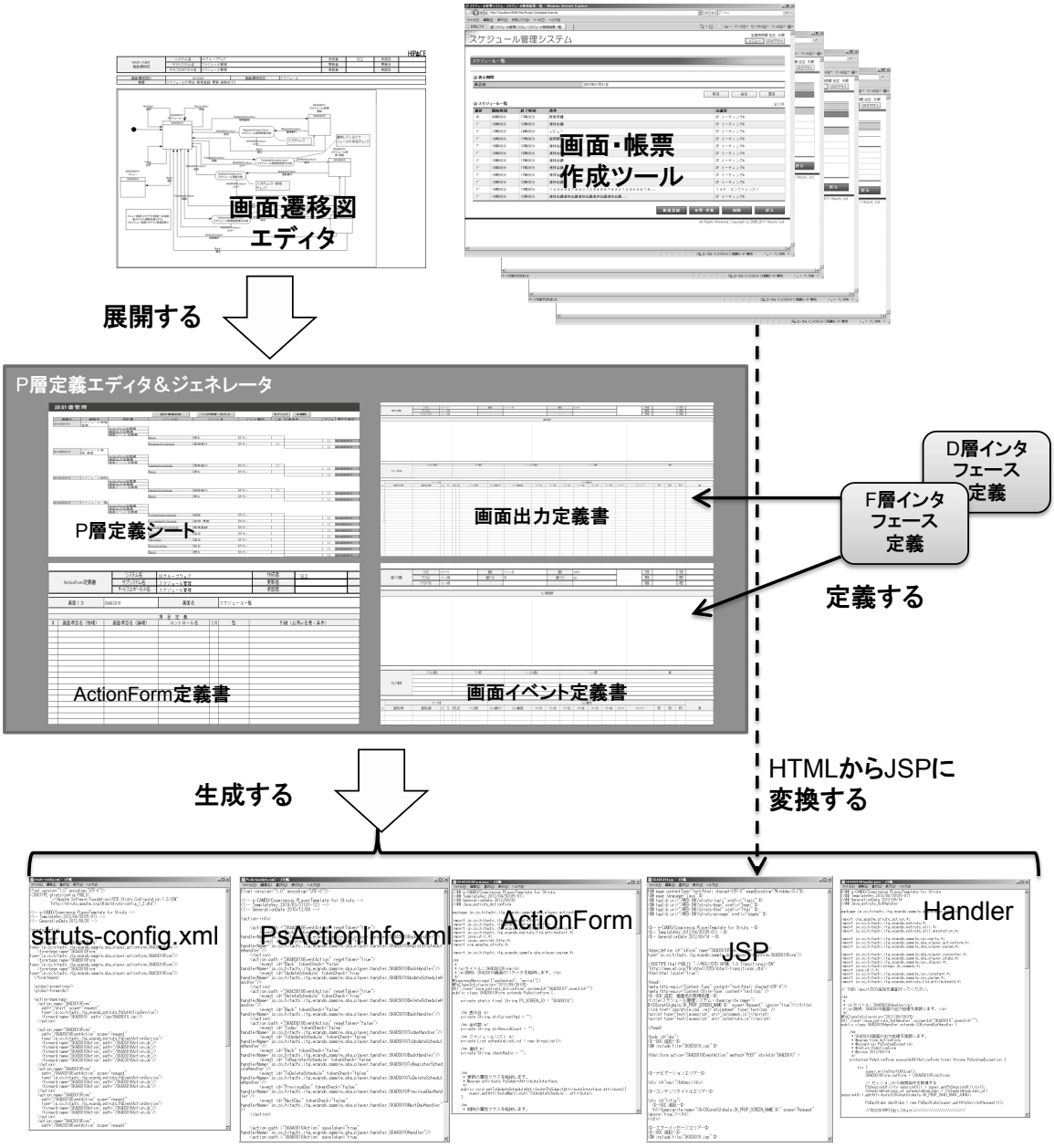


図 6-3　P 層開発の概要

Figure 6-3　Outline of P-layer development.

自動生成されるファイルは以下のようになる.

・PsActionInfo.xml：　P 層を構成する環境（ファイル）の情報

・Struts-config.xml：　Struts のコンフィグファイル
主に画面遷移と呼びだされる処理の関係が記述される.

・ActionForm：　画面との入出力を行うインタフェース

・JSP：　画面プログラム

・Handler：　P 層内の個別処理を記述するプログラム
画面とのデータの入出力，チェック・編集処理，F 層，D 層の呼び出しなどを記述する.
（各処理の呼び出し部分は自動生成される）

本ツールで生成するプログラム構造はPAC パターンに基づいているため(図 3-24 参照)，画面処理部分(Presentation)と，呼び出される F 層，D 層(Abstraction)は独立していることから，おのおのを結びつけるコードは，P 層全体を自動生成したあとに手で Handler 内にコーディングする．特定の処理同士が結びついていないため手でのコーディング量は確かに多くなるが，好きな処理を好きなタイミングで，必要ならば何度でも，複数の処理でも呼び出すことが可能である.

後ほど説明するチェック・編集処理は，入力データの場合は ActionForm 定義書に，出力データの場合は画面出力定義書に付記する.

(2)　F 層の設計から生成までの流れ

F 層はクラス図, シーケンス図をプログラム定義エディタ＆ジェネレータに読み込ませ, F 層を自動生成する流れである（図 6-4）.

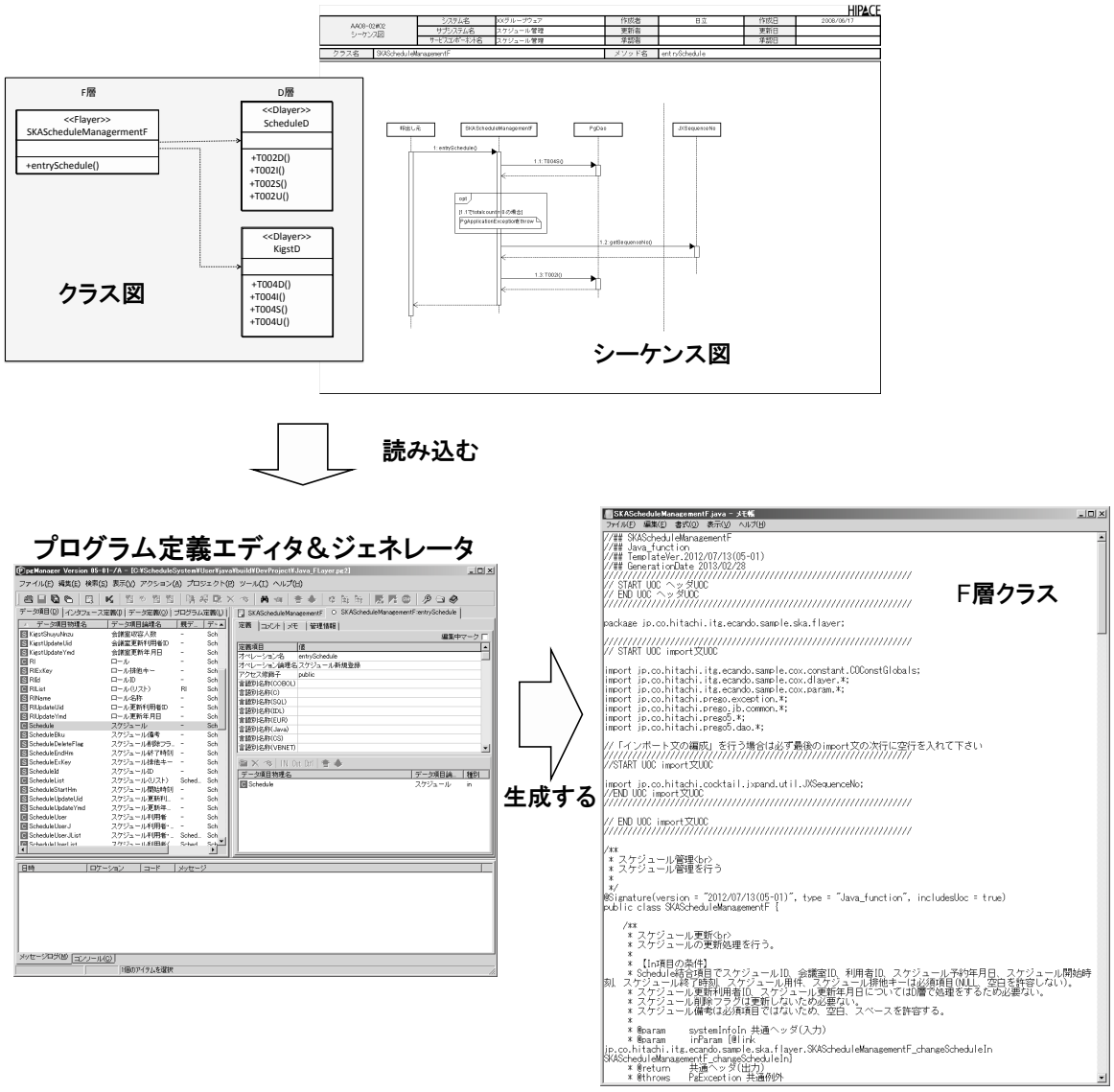


図 6-4　F 層開発の概要

Figure 6-4　Outline of F-layer development.

クラス図，シーケンス図は一般的な UML ツールを使用しているため，クラス情報を XMI 形式(XMI:XML Metadata Interchange)で抽出し, プログラム定義エディタ＆ジェネレータで編集可能な内部定義ファイルの形式に変換する.

次にこの内部定義ファイルに変換されたクラス情報に対し，生成ターゲットとなるプログラミング言語，層（F 層，D 層），バッチ処理，対話処理，オンライントランザクション処理といった処理形態，Web，クライアント・サーバと行ったシステムアーキテクチャの実装依存の情報をプログラム定義エディタ＆ジェネレータ上で定義してプログラムを自動生成する.

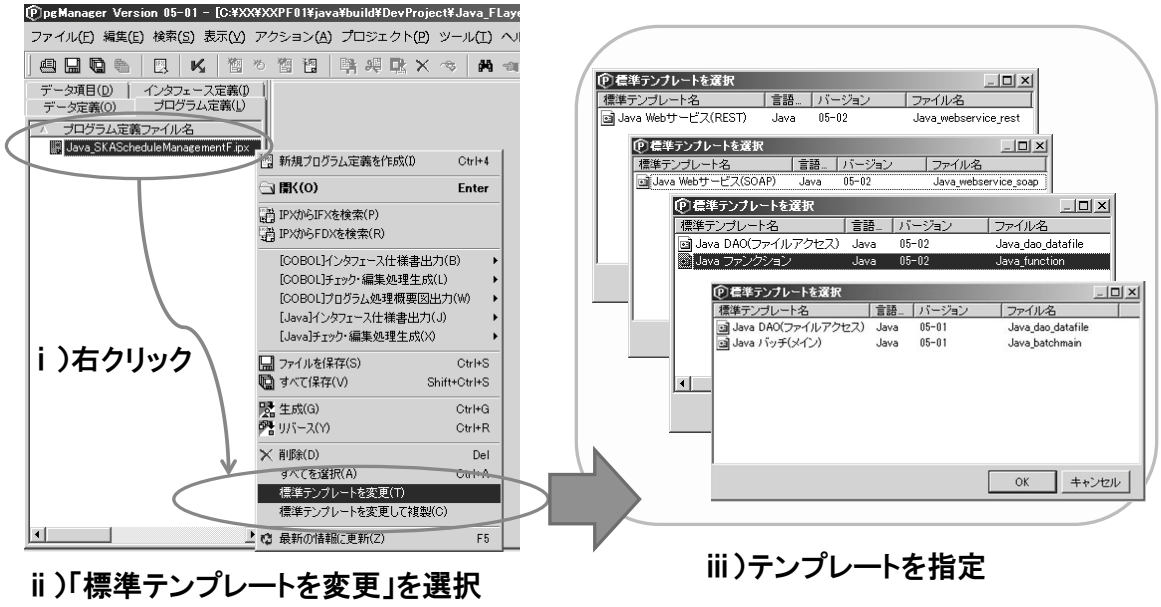


図 6-5　F 層生成の詳細１（プログラム定義エディタ＆ジェネレータ）

Figure 6-5　Details of F-layer generation No.1.（Program definition editor & generator）

プログラム定義エディタ＆ジェネレータでは，自動生成する言語ごとに層，処理形態，システムアーキテクチャを選択し，ターゲットとなるアーキテクチャにあったソースコードを生成する.

図 6-5 は，UML ツールから変換された初期状態の「標準テンプレート」に対して，ターゲットとなるアーキテクチャに合致する Java プログラムの形態を選択している場面である. 図 6-5 の右に並んでいるように, バッチ処理向けの F 層, D 層, 対話処理向けの F 層, D 層，SOAP 接続向けの F 層のアダプタ，REST 接続向けの F 層のアダプタが並んでいる.

図 6-6 の上部に並んで見えているように COBOL や．NET においても同様に，この操作から選択する.

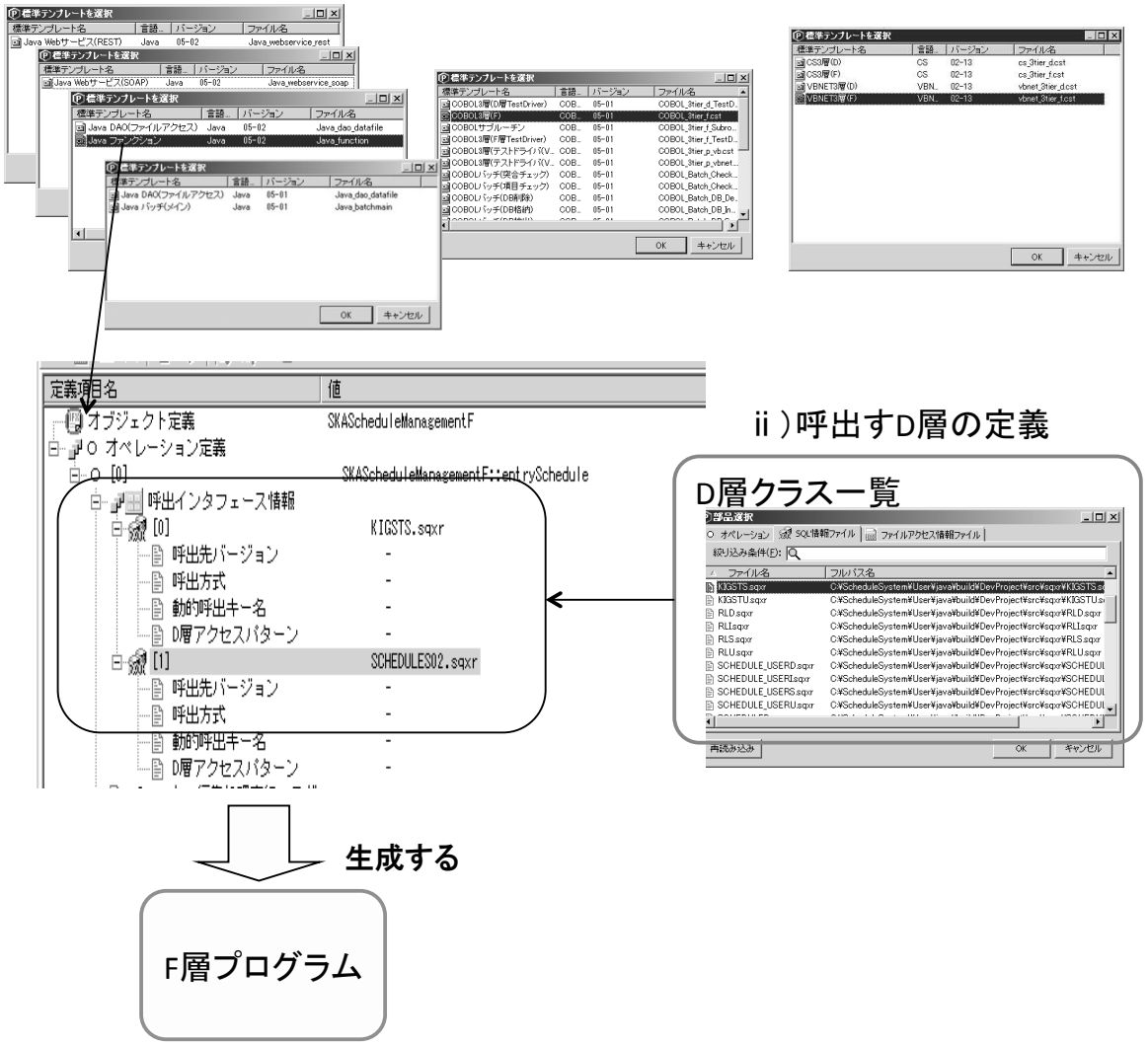


図 6-6　F 層生成の詳細２（プログラム定義エディタ&ジェネレータ）

Figure 6-6　Details of F-layer generation No.2.（Program definition editor & generator）

図 6-6 では F 層から呼び出す D 層を定義している様子も示している. クラス間の関係はこれまで述べたようにクラス図とシーケンス図から移送することにしていたが，フィールドで多くのフィードバックを得て，インタフェース部分を含め詳細にクラスの情報を UML で書くのは非常に効率の悪い作業であり，本ツール上でインタフェースとなるデータ項目を記述することも可能とした. 加えてのクラスやメソッド自身の追加，メソッド内で呼び出す他クラスの指定も，このツール内で実施可能である.

プログラム定義エディタ&ジェネレータは, クラス自身の情報（クラス名, メソッド名,

インタフェース，言語，層(F 層,D 層)，処理形態，システムアーキテクチャ）に加え，呼び出し関係にあるクラスの情報も管理しており，双方のインタフェースを自動生成する．定義されたクラス，メソッド間の呼出し関係に忠実に両クラスのインタフェースを生成するため，インタフェース不整合によるバグは理論的には発生しない．

なお，プログラム自動生成時は，上記の必要情報が全て揃っている必要があり，不完全な状態では生成は出来ない．以上がソースコード生成までの簡単な流れである．

このプログラム定義エディタ＆ジェネレータで先ほど述べた SOAP などアダプタの生成を行えることに加え，テスト用のスタブ，ドライバなどもオプションとして生成可能としている．

詳細は 6．3 で述べるが，反復型開発を支援する再生成機能もこの画面で指定する．人手によって後にコーディングされた部分は自動退避し，自動生成されたコードが毀損してないかチェックし、再生成後に該当場所へ復元する．復元先が仕様変更などで無くなった場合などはアラームを出す．

新規開発時は日々発生する仕様変更や追加，漏れなどが発生するたび繰り返し利用する機能であり，また保守時でも同様である．

(3)　D 層の設計から生成までの流れ

D 層は最終的には F 層と同様にプログラム定義エディタ＆ジェネレータから生成されるが，DB 設計と SQL 設計を，プログラム定義と分離して行う点に特長がある．

まず，ER 図でデータベースを設計する必要がある．次にデータベース設計情報を SQL 定義エディタに読み込ませ，SQL を設計する．

設計された SQL はプログラム定義エディタに取り込み, D 層として生成する (図 6-7)．

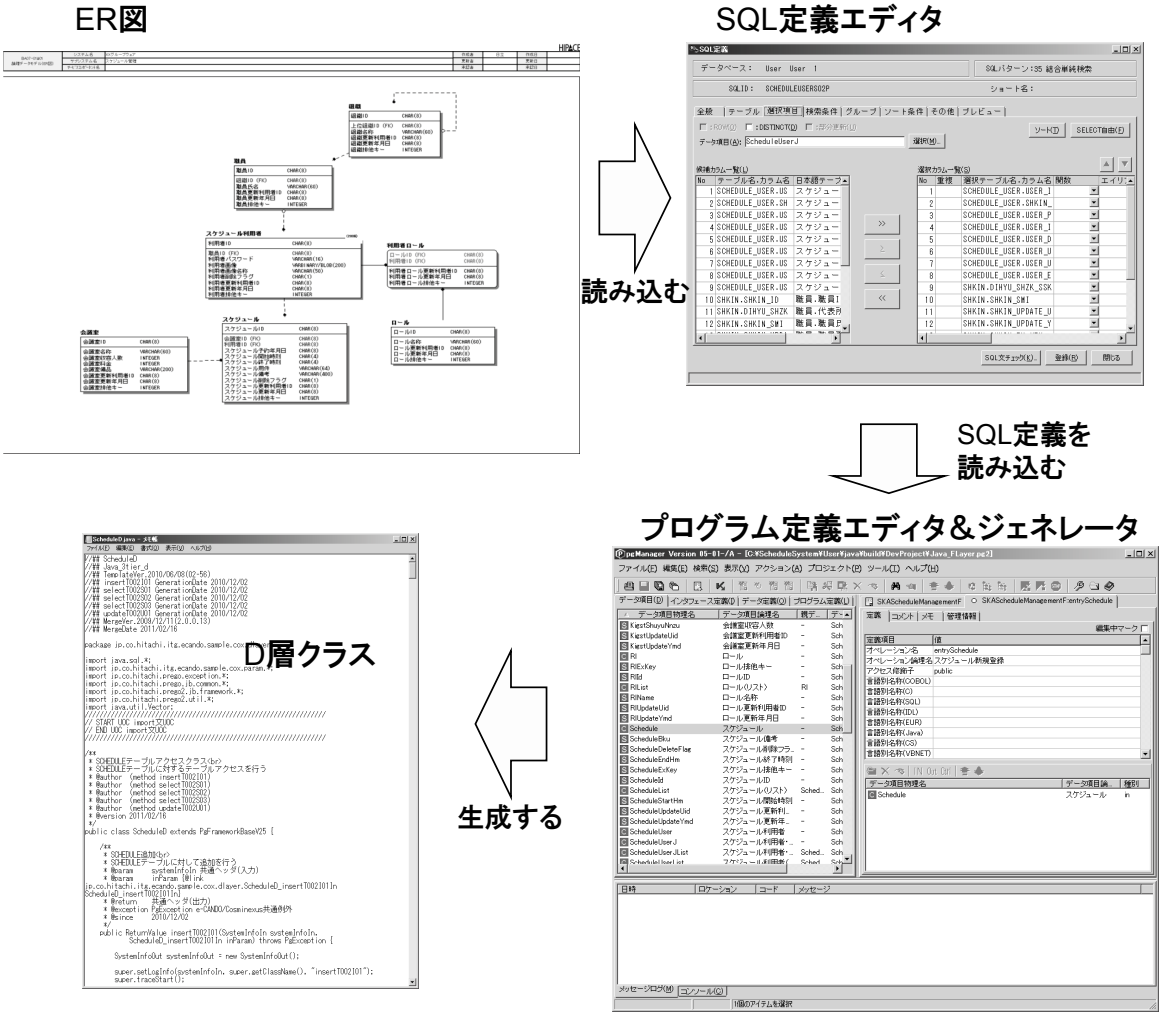


図 6-7　D 層開発の概要

Figure 6-7　Outline of D-layer development.

ここで，D 層クラスの情報は，F 層生成で作成したクラス図，シーケンス図から取り込

んだ情報を利用しても良いが，クラス図でインタフェース部分を含め詳細に書くのは非常に効率の悪い作業であり，F 層と同様にクラスやメソッドの追加，インタフェースの作成をプログラム定義エディタ＆ジェネレータ内で実施しても良い.

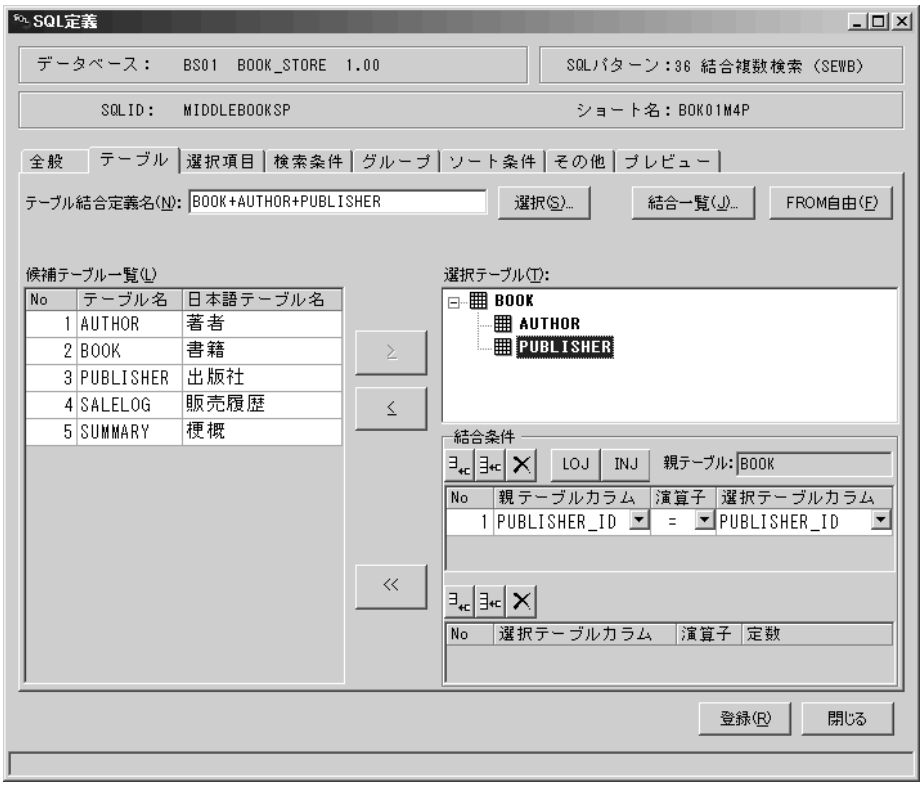


図 6-8　Join を定義している例（SQL 定義エディタ）

Figure 6-8　Example that defines an SQL join. (SQL definition editor)

図 6-8 は SQL 定義エディタで SQL の Join を定義している例である.

4.2.4 で述べたように D 層は SQL を完全に内包する．したがって SQL の定義は図 6-8 のツールに全て定義し格納される．このツールからデータベース定義と，プログラム定義エディタ＆ジェネレータで D 層を生成する時に必要な SQL 情報が生成される.

図 4-7 に示したようにプロジェクト内で解決できない技術課題はデータベースに集中する．このツールに SQL 定義を集中的に格納しておくことで，SQL スキルの低い開発者には SQL の書き間違いを減少させ，一方で無駄に凝った SQL の実装を未然に防止する検証機の役割を果たす.

特定の SQL 実行時に不具合が発生した場合，このツールで問題の SQL を抽出し，似た

ような書き方をしている他の SQL 文を一気に直すということも可能である．優れたマネジメントツールとなる．

SQL 定義エディタは SQL の開発をアシストする．単純なデータの追加，プライマリキーによる参照，更新，削除は特段の定義の必要なく，ER 図から取得したテーブル情報を使って自動生成される．

また典型的な Join はパターンが用意されており，続いて関連するテーブル，取得するデータ項目（属性），検索条件の順にツールの画面の流れに沿って定義していけば確実に動作する SQL が自動で生成出来る（図 6-9）．

ツールとしては，この高品質に SQL を生成できるパターンによる方法で極力 SQL の開発を行うことを推奨するが，パターンでカバーされていない複雑な SQL も自由記述機能で作成が可能である．

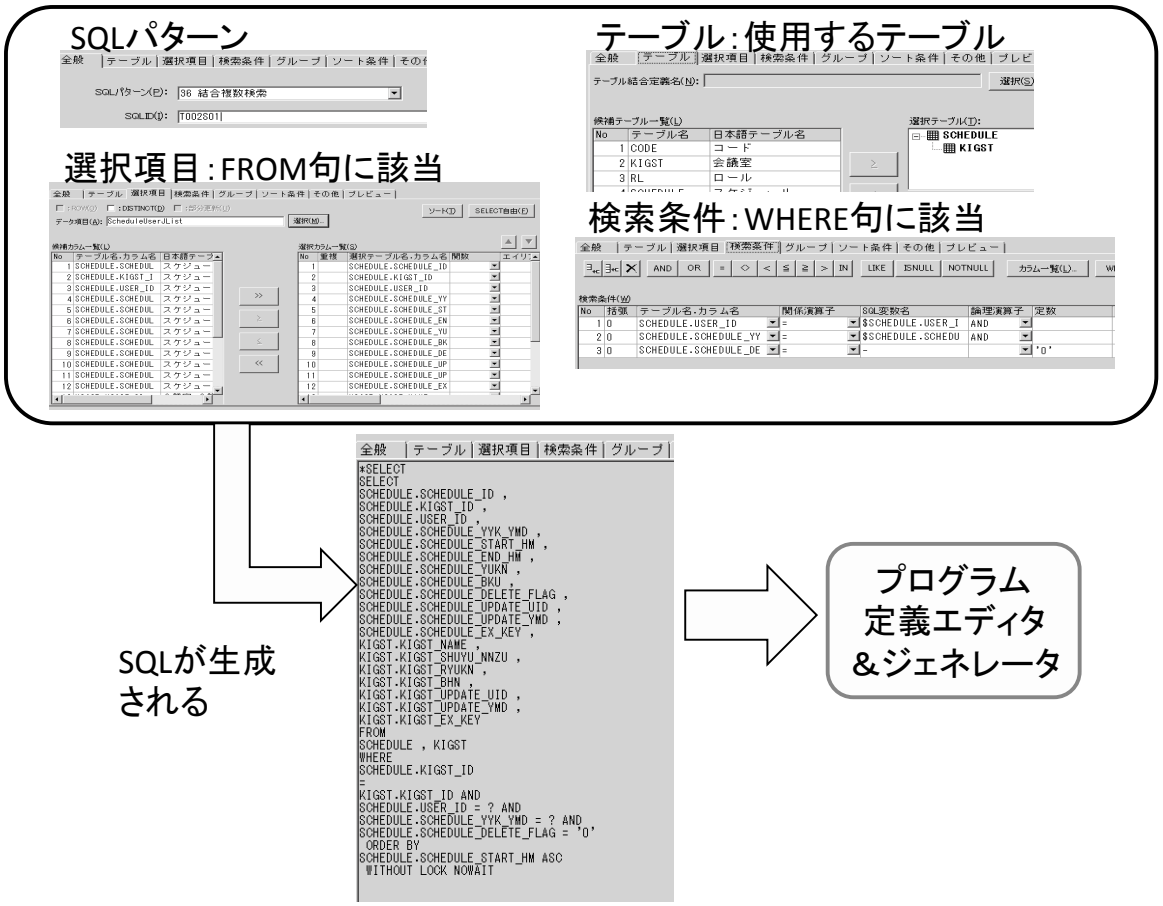


図 6-9　SQL 生成の詳細

Figure 6-9　Details of SQL generation.

(4) チェック・編集処理の定義と生成

チェック・編集処理とは，データ項目に依存する処理のことで，チェック処理の例であれば入力された日付が休日かどうかの判定などがあげられる．編集処理の例であれば和暦から西暦への変換などがあげられる．本ツールではこのようなチェック・編集処理を自動生成する機能を提供する．

エンタプライズシステムで多用されるチェック・編集処理は事前にチェック・編集仕様エディタに組み込まれている．もちろん任意の処理を組み込むことも可能で，その場合は事前にチェック・編集仕様エディタで該当の処理と，呼び出す時のドメイン名称を指定しておく．

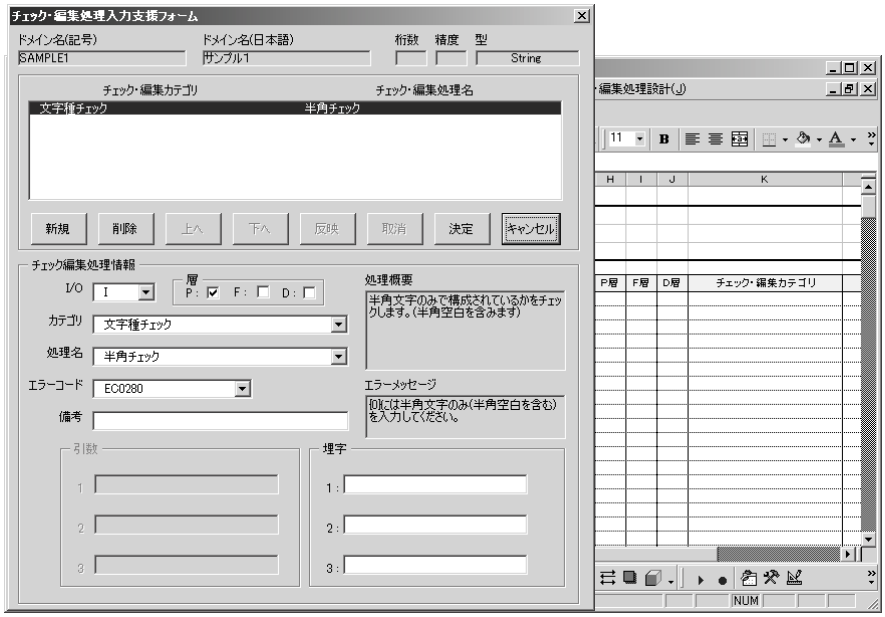


図 6-10　チェック・編集仕様エディタ

Figure 6-10　Specification editor for validation & edit.

図 6-10 は文字列のチェック処理を定義している．入出力区分，どの層（P 層，F 層，D 層）で利用する処理なのか，チェックする文字の種類，エラー時の出力メッセージなどを指定して，ドメイン名というパターン名称を付ける．

図 6-11 は，P 層でチェック・編集処理を生成する時の流れを示している．

画面入力されたデータに対してチェック・編集処理を実行したい場合は，図 6-10 で示

したチェック・編集仕様エディタ上に所望の処理が登録されていることを確認し，次に画面からの入力データに対してチェック・編集処理を実装するのであれば，入力データのActionForm定義書の該当のデータ項目にドメイン名称を指定する．画面に対する出力データに対してチェック・編集処理を実装するのであれば，画面出力定義書の該当のデータ項目にドメイン名称を指定する．

#	ドメイン名(物理)	ドメイン名(論理)	型	桁数	小数	I/O	P層	F層	D層	チェック・編集カテゴリ	チェック・編集処理名
1	YMD	年月日	String								
2						0	○			日時文字列編集	日時フォーマット変換
3	YMD_Y	年	String								
4						1	○			文字種チェック	半角数字チェック
5						1	○			文字列長・桁数チェック	以下チェック(文字列のバイト数)
6						1	○			パターンチェック	日時チェック(入力フォーマットを指定)
7	YMD_M	月	String								
8						1	○			文字種チェック	半角数字チェック
9						1	○			文字列長・桁数チェック	以下チェック(文字列のバイト数)
10						1	○			パターンチェック	日時チェック(入力フォーマットを指定)
11						1	○			文字列編集	文字列長合わせ(左端の0追加)
12						0	○			文字列編集	左端の0削除
13	YMD_D	日	String								
14						1	○			文字種チェック	半角数字チェック
15						1	○			文字列長・桁数チェック	以下チェック(文字列のバイト数)
16						1	○			文字列編集	文字列長合わせ(左端の0追加)
17						0	○			文字列編集	左端の0削除
18	HM	時刻	String								
19						0	○			日時文字列編集	日時フォーマット変換

	パラメータ仕様							チェック/編集仕様	
#	画面項目名(物理)	画面項目名(論理)	I/O	型	反復	必須	ドメイン名(物理)	チェック・編集カテゴリ	チェック・編集処理名
1	strCurrentYmd	表示日	0	String			YMD		
2								日時文字列編集	日時フォーマット変換
3	strRecordCount	全件数	0	String					
4	scheduleList	スケジュールリスト	0	結合項目					
5	strScheduleStartHM	開始時刻	0	String			HM		
6								日時文字列編集	日時フォーマット変換
7	strScheduleEndHM	終了時刻	0	String			HM		
8								日時文字列編集	日時フォーマット変換

図 6-11　チェック・編集処理の生成（P 層の例）

Figure 6-11　Generation of validation and editing process. (example of P-layer)

チェック・編集処理はP 層，F 層，D 層いずれにも生成が可能であり，F,D 層であればプログラム定義エディタ上でオペレーション定義（メソッド定義）にあるインタフェースのデータ項目名（変数）にドメイン名をくくりつける．

この機能は，チェック，編集処理に限らず，特定のデータ項目に処理をくくりつけることが可能であるため，簡易 BRMS としても利用可能である．

6．3　反復型開発における再生成の流れ

本ツールの大きな特長は，モデルベース開発で反復型開発を実現するためのプログラム再生成機能にある．以下はF 層を例として再生成の流れを示す．

プログラム定義エディタ＆ジェネレータで定義されたF層のプログラム情報は，本ツール独自形式の定義ファイルとして格納される．F層プログラムを生成する際は，この定義ファイルに対してプログラムコードへの変換を指示する形になる（図6-12）．

変換指示は，図6-5に示したように，プログラミング言語，層，処理形態，システムアーキテクチャを意識して行われる．

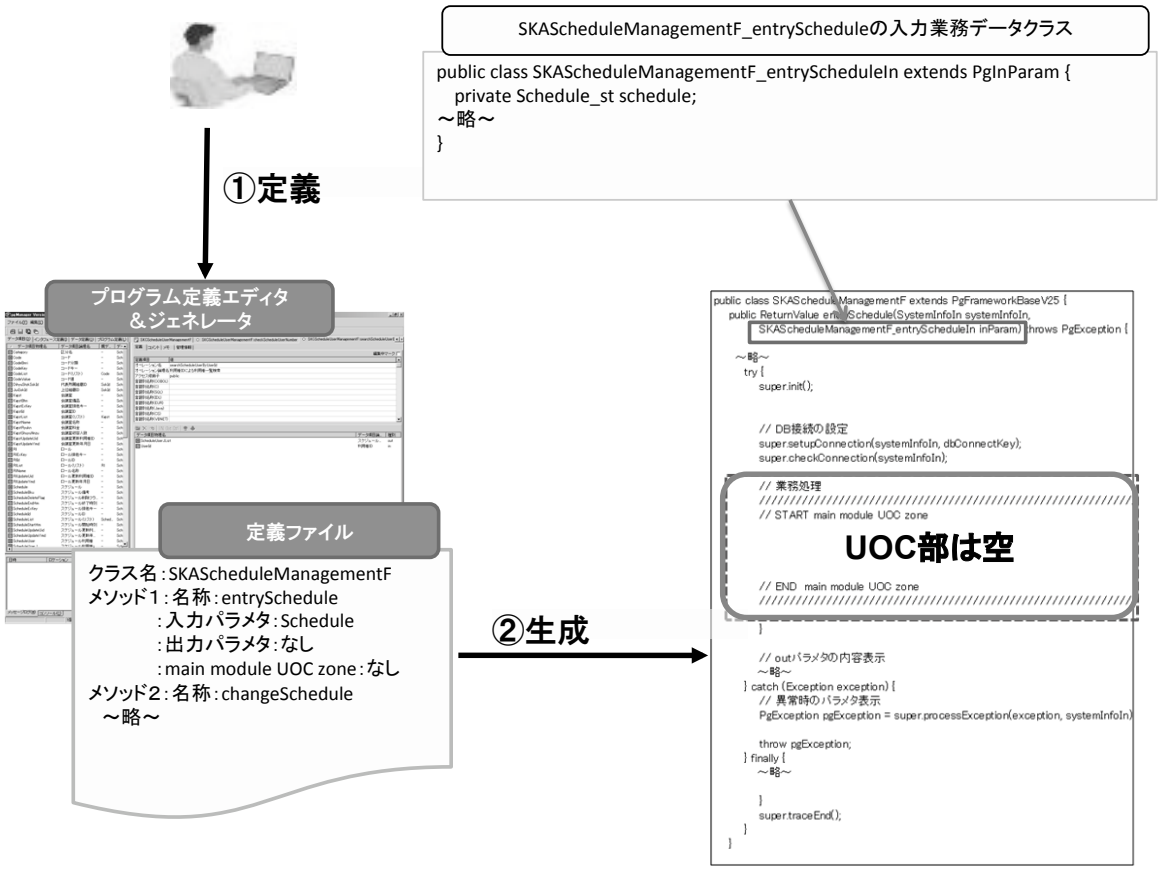


図6-12　F層の初回生成

Figure 6-12　Generation first time of F-layer.

プログラム生成処理は，大規模な開発を意識しており，特定のクラスを指定して行うことも可能であるし，定義済みのクラス全てを一括して行うことも可能である．

生成されたプログラムに対して，不足する処理部分を開発者が手で追加コーディングを行う．本ツールでは手で追加したコードをUOC(User Own Code)と呼ぶ．

手で追加したコードは，リバース処理を実行すると定義ファイルにUOC部分だけ選択

して格納される（図 6-13）．リバース処理も任意のクラス単位で実効することも可能であるし，一括で行うことも可能である．

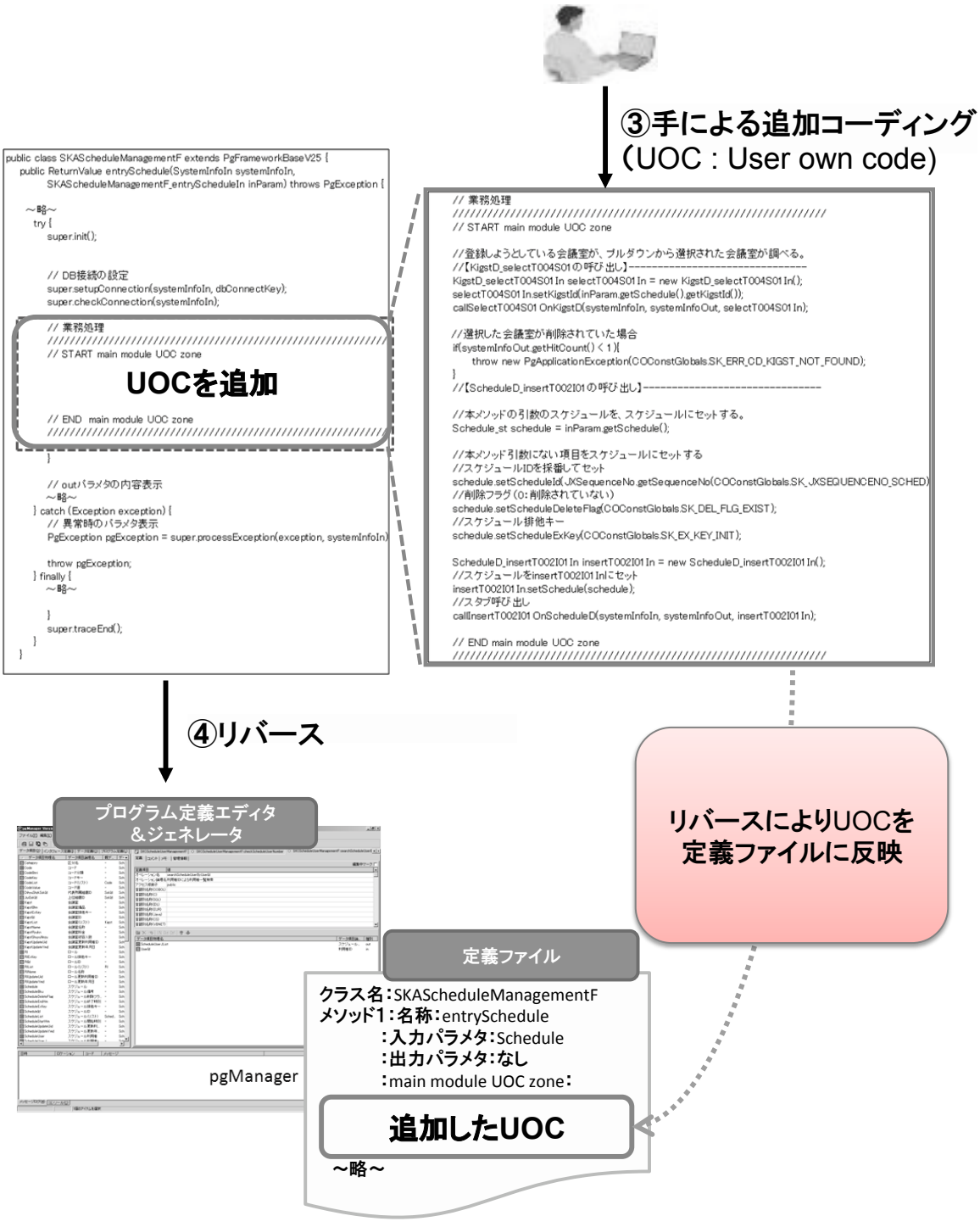


図 6-13　F 層への手コーディングとリバース処理

Figure 6-13　Hand coding and reverse processing to F-layer.

図 6-14 にモデルを追加，修正を行った場合の再生成時の処理の流れを示す．モデルの変更部分が自動生成されたソースコードに反映され，一方では UOC 部分が復元される．

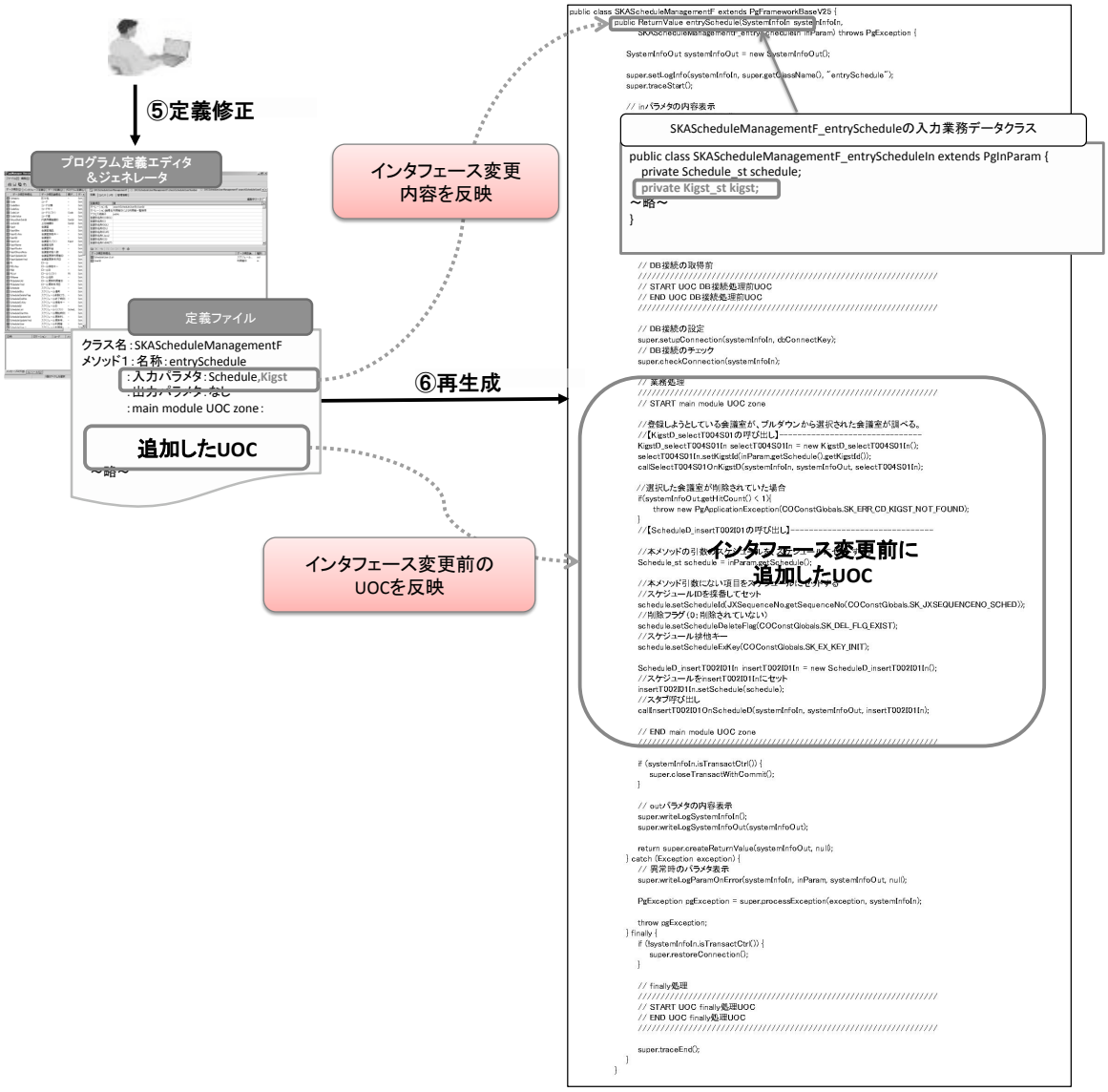


図 6-14　モデルの修正と F 層の再生成

Figure 6-14　The model's correction and re-generation of F-layer.

再生成時は，図 6-13 で示したリバース処理が前提となる．仮にリバース処理が行われていない場合は，再生成処理は整合性がとれていないとしてワーニングメッセージを出力し，勝手には実行されない．

プログラム生成・再生性処理は本ツールの中核機能であり，仕様の変更，保守など，モデルへ仕様の追加・変更が発生する度に繰り返し利用される．

また，自動生成するコード以外の特定処理（プロジェクトが個別に用意したソースコードや外部処理の呼び出しなど）を，生成・再生成するソースコードの特定の箇所に同時に生成することももちろん可能である．このような機能を活用することで生産性を大きく向上させたプロジェクトも多い．

6．4　他ツールとの機能比較

(1)　MDA ツールとの比較

他のツールとの機能比較を行う．本ツールの試作は 1999 年から行い，本格的にフィールドへ投入したのは 2002 年であるので，同時期に商用製品として登場した OptimalJ を比較対象に選定する．

比較対象とした OptimalJ は，MDA を忠実に具現化しようとした製品で，当時としては MDA ツールとして最も先進的な機能を備えていた[50]．本研究では 2003 年から 2004 年にかけて比較を行った．この間 OptimalJ は大きく機能強化がなされた．比較評価は 2004 年当時の最新バージョン 3.1 に基づいて行っている．

両ツールの共通点は，クラス図を起点にしているところである．OptimalJ は，サービスモデルとクラスモデルという 2 種類のクラスを定義し，本研究で開発したツールは F 層と D 層と特色付けした同じく 2 種類のクラスを定義する．

サービスモデルと F 層はほぼ役割が似通っており，基本的には Façade(入口，玄関のような役割)としての役割を受け持つ．それぞれ手コーディングで機能を追加する点も同じである．

一方クラスモデルと D 層は大きく性格が異なる．両者はデータの永続化（RDBMS のアクセス）を受け持つ点は同じだが，実装方法は全く異なる．OptimalJ のクラスモデルは極力クラス図に忠実に実装するよう工夫されている．一つの基本形は EntityBean クラスに一つのリレーショナルデータベースのテーブルが割り当てられ，そのクラスが持つ基本機能（永続化オブジェクトの生成，変更，オブジェクト ID による検索，削除）が自動でメソッド生成される．クラス間に集約（コンポジション），継承（汎化）の関係があれば，クラス間でその関係に沿ったアクセスメソッドが生成される．ただし，UML 上は正しく

ともコードを生成出来ないパターンが当時は存在した.

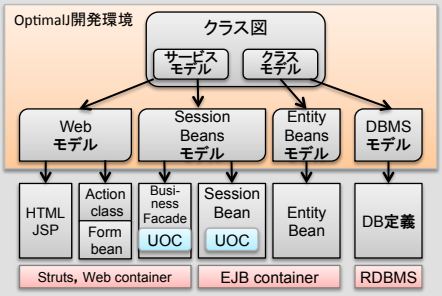
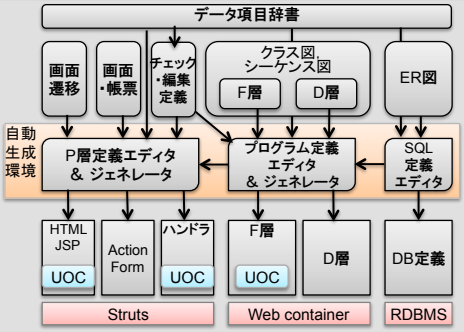
比較項目	OptimalJ(2003～2004)	本ツール
自動生成の流れ		
処理の概要	・クラス図から全てのコードを生成する. クラス図は大きくサービスモデルとクラスモデルで構成される. ・MDAのPSM(Platform Specific Model)にあたるアプリケーションモデルに展開し, その後ターゲットとなるコードを生成する. ・アプリケーションモデルは, Webモデル, Session Beansモデル, Entity Beansモデル, DBMSモデルがある. ・ソースコードはStrutsとEJB2.0に忠実に沿って実装される. ・画面レイアウトは画面エディタで編集する.	・画面遷移, 画面レイアウト, チェック・編集処理, クラス図, シーケンス図, ER図からコードを生成する. クラスは大きくF層, D層で構成される. ・自動生成環境に上記データを取り込んで, コードを生成する. ・コードジェネレータは, P層定義エディタ&ジェネレータ, プログラム定義エディタ&ジェネレータ, SQL定義エディタからから構成される. ・ソースコードはJavaの場合P層はStrutsに沿って, F層とD層はJavaプログラムに生成される. EJB化する場合はEJBアダプタを生成する.
自動生成の範囲	・ビジネスロジック以外の, 画面, 画面遷移, データベースアクセス処理, データベース定義を生成する.	・画面上の細かな処理, P層の個別処理(ビジネスロジック), F層の個別処理(ビジネスロジック)以外の, 画面, 画面遷移, データベースアクセス処理, データベース定義を生成する.
優位点	・インプットとなるのはクラス図のみ. 生産性が高い. ・モデルへの変換はパターンで行う. パターンを加工すれば特定処理を加えることも可能. ・一つのツールで実現可能.	・画面遷移, 画面レイアウトは自由に定義可能, 画面内の処理も自由度高く記述できるため, 所望の画面, 画面遷移を作成可能. ・データベースアクセスはSQL定義エディタで自由度高く生成可能のため所望のデータアクセスを実現可能. ・P層, F層, D層ごとに必要機能だけを利用することも可能.
劣位点	・画面遷移, 画面内の処理は外部から定義したものではないため, 定型的なものしか生成できない. ・データベースアクセスも個々のテーブルへの単純なアクセスロジックとクラス図に定義されたクラス間の関連に基づいたアクセスロジックしか生成出来ないため, 個別にSQLを作成するか, Entity Beanの機能を使って必要データを収集して加工するプログラムを手でコーディングする.	・OptimalJに比べ画面遷移, 画面レイアウト, ER図など多くのインプットを要す. ・多くのツールを利用しなければならない.

図 6-15　MDA ツールとの比較

Figure 6-15　Comparison with MDA tool.

一方 D 層は，一つのクラスに一つの RDB のテーブルが割り当てられ，そのクラスが持つ基本機能 (テーブルに対するレコードの追加, 更新, プライマリキーによる検索, 削除) を自動で生成するところまでは同じだが，それ以上の複雑な検索処理は，図 6-8 に示した SQL 定義エディタ上で設計する.

クラス図でオブジェクト同士の関係を記述するより SQL は遙かに複雑なデータアクセスを記述可能であるため，結局 OptimalJ で複雑なデータアクセスを実現するには，BusinessFacade 上に手コーディングで SQL を含む Java コードを追加するか，同じく手コーディングで必要なデータを抱える EntityBean クラスから必要となるデータを

BusinessFacade 上に引き出して SQL 相当の編集処理を自力で実装する必要がある. いずれも相当困難な作業になることが予想される．いずれは EntityBean クラスに SQL を含む独自のメソッドを生成する機能がサポートされたであろうが，当時は存在しなかった.

データベースアクセスに関しては D 層の方が圧倒的に実用性で勝っていた.

次に画面の実装であるが，OptimalJ は SVC パターンに忠実に作られており，サービスモデルあるいはクラスモデルからアプリケーションモデルで変換した Web モデルを MVC パターンの Model として捉え，その Model にアクセスする画面を自動生成する．サービスクラスは F 層同様に空であるため，そのままでは動作しないが，クラスモデルから生成された画面は，クラスモデルが既に基本機能（テーブルに対するレコードの追加，更新，プライマリキーによる検索，削除）やクラス間の関係に沿ったアクセスメソッドを実装しているため，それらメソッドを駆動するボタンまでが自動で実装され，テーブルに対する基本的な入出力を行うアプリケーションであれば実行可能なレベルまで完成する．ボタンの名前（ラベル）や画面内のレイアウトなどは画面エディタで編集可能である.

しかし，画面や画面遷移は型どおりのものしか生成出来ないため，所望のものを得ようと画面に手で大幅に修正を加えたり，外部のソフトウェア部品を導入したり，画面遷移を変更するとなると，手作業での開発と難易度は変わらなくなってしまう.

一方，P 層は，ほぼ Struts の構成クラスを意識した情報を定義することでプログラムを生成する．OptimalJ に比べ遙かに手間がかかるが，アプリケーション開発の自由度は Struts 上に手作業でプログラムを作成する場合と遜色ない.

マルチアーキテクチャ対応という観点では，.NET 等オブジェクト指向言語への対応は将来的に可能な構造と推定された（当時は未対応）が，COBOL への対応は困難であり，本ツールより適用範囲は狭い.

2. 1 に述べたように日本のエンタプライズシステム，特に基幹系のシステムはスクラッチ開発が好まれるが，画面や帳票にこだわりがある点もスクラッチ開発を選ぶ大きな要因となっている. 少なくとも 2004 年時点では OptimalJ で日本のエンタプライズシステム，特に基幹系のシステムを開発することは困難であった.

なお，OptimalJ は 2008 年に実質的な開発を終了している.

(2)　OR マッピングツールとの比較

次に D 層と他の OR マッピングツールとを比較する．比較するのはいずれも 2001 年ころから開発がスタートした Hibernate[51]と iBatis（MyBatis）[52]である．

3 つのツール共にオブジェクト内にリレーショナルデータベースをカプセル化し，オブジェクトとして呼び出せるようにするという目的は同一だが，実現ポリシーが全く異なる．

Hibernate は，オブジェクトに対して，それに見合うリレーショナルデータベースのテーブルを自動生成することで目的を達成しようとする．その対極にあるのが D 層でデータベースのテーブル構造を基とし，それにアクセスするオブジェクトを生成する．2 つとは全く異なる発想で開発されたのが iBatis（MyBatis）であり，テーブルに対してではなく SQL に対してオブジェクトを生成する．

	比較項目	Hibernate	iBatis（MyBatis）	本ツール
#1	マッピング方向 （オブジェクト ⇔リレーショナル）	ORマッピング （DB定義自動生成）	ORマッピング （DB定義は自動生成しない）	ROマッピング （クラス自動生成も可能）
#2	SQLの生成	HQL（SQLライクなクエリ言語）でサポート	未サポート	専用UIでサポート
#3	SQLの編集	自動生成されるSQLは不可能 （ネイティブSQLは可能）	可能	専用UIでサポート
#4	インターフェース （操作性）	コマンドライン	マッパーXMLに記述	専用UI
#5	ネイティブSQLの記述	可能	可能	可能
#6	対応プログラム言語	Java，．NET	Java，．NET	Java，COBOL， ．NET（C#）
#7	複数RDBMS製品への対応	可能	可能	可能
#8	特長機能	キャッシュ， レイジーロード， 属性（データ項目）の チェック処理	ダイナミックSQL， レイジーロード	複数データー括読込み， 属性（データ項目）の チェック・編集処理， DBアクセス解析支援， インタフェーストレース， SQLトレース

図 6-16　OR マッピングツールとの比較

Figure 6-16　Comparison with OR mapping tool.

図 6-16 に各ツールの比較を示すが，どのツールも日々機能強化されている状態にあり，詳細機能の比較は参考に過ぎない．例えば Hibernate は，以前はオブジェクトからテーブ

ルを生成する機能しかなかったが，フィールドのフィードバックを受けて，リレーショナルデータベースのテーブル定義情報からオブジェクトを生成する機能（RO マッピング）もサポートするようになっている．また Hibernate や iBatis（MyBatis）はオープンソースということもあり，機能強化を支援する他のオープンソースが出現する期待もある反面，利用時に問題が発生しても自己責任での解決が原則というリスクもある．

本ツールで D 層開発に拘るのは他のツールにはない以下の 3 点の優れた特長があるからである．

- COBOL のサポート
- 技術力の高くない技術者でも SQL が設計できる UI(User Interface)のサポート
- 定義した SQL を一括管理しており，SQL の品質分析や開発進捗を分析可能

以上，MDA ツールとの比較においては，実装の自由度を求めるスクラッチ型開発というニーズに合致し実用的であり，また OR マッピングツールとの比較においても，D 層はあまり技術力の高くない技術者をサポートする UI を持ち，一定レベルの技術者を多く集めるのが困難なエンタプライズシステム開発の現状に沿っていて実用的である．また COBOL 言語のサポートを含むマルチアーキテクチャ対応という点においても本ツールが優れているといえる．

6．5　本ツールの利用を前提としたソフトウェア規模見積ツールの実現

第 5 章で述べた FP から LOC への精度の良い変換方式を生かし，見積ツールを開発した．例として F 層の見積について EI タイプの画面で説明する．

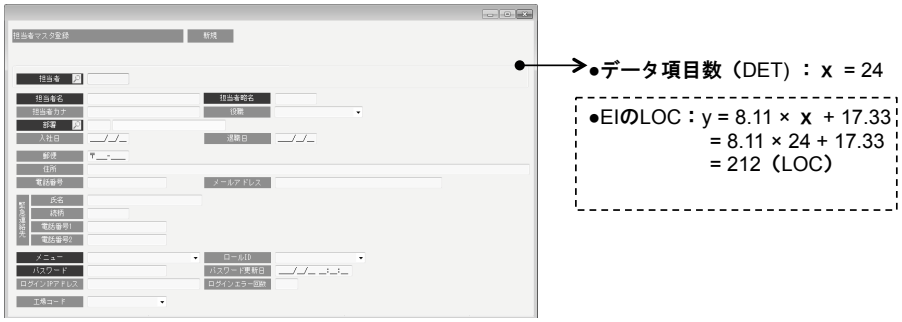


図 6-17　DET から LOC の算出

Figure 6-17　Example of calculating LOC from DET.

第 5 章では，FP を求める構成要素のうち，データ項目数（DET）からの変換精度が良いとの結論を得た．図 6-17 ではその変換式である y = 8.11 × x + 17.33 を用いて x に 24(DET)を代入して y = 212(LOC)を求めている．

ツールを利用すれば x を求めれば，あとは自動で計算を行ってくれる．図 6-18 はツールで FP から LOC を算出している例で，図 6-17 の計算は，楕円で囲った部分で行われている．

●トランザクションファンクションの測定
画面項目数,ボタンイベントにより,P層，F層のFPを測定

機能ID	機能階層 大機能	中機能	小機能	画面項目数	画面数		画面数内訳					帳票数	FP計	ファンクションポイント			STEP数（ks）			
					論理画面数	物理画面数	通常画面	大画面	巨大画面	小画面				外部入力（EI）	外部出力（EO）	外部照会（EQ）	P層（画面）	P層（帳票）	F層	P/F計
計	1 機能	1 機能	113 機能		175.0	128	82	35	7	4		22	1597				128.80	16.19	93.04	238.03
1	Project-A	Package	ED 取り込み	11	1.0	1	1						6	6			0.74		0.11	0.84
2			受注入力_明細情報タブ	49	2.0	1		1					23	14	3	6	1.47		1.53	3.01
3			受注入力_納入情報タブ	16	1.0	1	1						5	5			0.74		0.15	0.88
4			受注承認	40	2.0	1		1					9	5		4	1.47		0.46	1.93
5			受注照会	68	3.0	1			1				10		6	4	2.21		0.39	2.60
6			売上返品入力	25	1.0	1	1						19	10	3	6	0.74		0.60	1.34
7			受注内示確定照会	13	1.0	1	1						10		6	4	0.74		0.10	0.84
8			ED 受注検収照会	49	2.0	1		1					10		6	4	1.47		0.29	1.76
9			品番マスター覧	51	2.0	1		1					10		6	4	1.47		0.30	1.77
10			品番マスタ登録	59	2.0	1		1					11	6	3	2	1.47		0.67	2.14
11			設備ライン能力マスタ登録	5	1.0	1	1						12	9		3	0.74		0.20	0.94
12			担当者マスタ登録	24	1.0	1	1						5	5			0.74		0.21	0.95

P層，F層，D層のFP測定結果を集計

	FP	STEP数(ks)
全体FP	2948	292.07
マスタ分FP	1351	54.04
機能分FP	1597	238.03

図 6-18　FP 集計ツールの概要

Figure 6-18　Outline of FP calculation tool.

個々の計算は今回の例に限らず非常に単純であり，一方で大規模システムを見積もる場合は一覧表にまとめても巨大なサイズとなる．ツールに求められるのは入力アシストよりも一覧性とデータの 2 次加工のしやすさである．そのため以前は.NET 言語で開発したクライアント・サーバ型システムを提供していたが，廃止して EXCEL をベースにリニューアルし，一覧性と 2 次加工のしやすさ，可搬性に優れた仕掛けにした．

6．6　まとめ

この章では，モデルベース設計ツールと，FP 集計ツールの具体的な操作を追いながら実装されている機能を説明した．また，モデルベース設計ツールでは他の類似機能を持つツールとの機能比較も行った．

モデルベース設計ツールは，第 3 章で定義した 8 つの基本的なシステム構成に対応すべく，プログラミング言語，層 (P 層，F 層，D 層)，バッチやオンラインといった処理形態，クライアント・サーバや Web といったシステムアーキテクチャを意識したプログラム生成を行っていることを 6．2 で示した．また第 4 章で体系化したモデルベース開発でありながら適所では手コーディングを行う反復型開発を実現するために，6．3 でモデルからプログラムの生成・再生成の操作を，一部仕組みを含め示した．

6．4 では，まず OMG が MDA 提唱後，最もその理想に近く，かつ先進的だったツールとの機能比較を行い，エンタプライズシステムのスクラッチ型開発においては，本ツールが実用性の面で優位であることを示した．続いて OR マッピングツールとの比較では，10 年以上も開発が続く，今日でも最もポピュラーな 2 つのオープンソースと比較し，機能面で見劣りなく，一方で SQL に対する知識が不足する技術者にも利用しやすい UI を備える点で優位であることを示した．

第 7 章　本ツールの適用実績と評価

7. 1　生産性と品質に対する評価

ステップ数（LOC）を評価軸としてプログラム自動生成の効果を調査した.

(1)　生産性

図 7-1 に Java 言語の生産性を示す．ソースコード行数を，ソフトウェア詳細設計からソフトウェア結合までに要した工数で除し，相対比較したものである．2007 年に本番リリースを迎えたプロジェクト 40 のデータで，本ツールを適用して自動生成を適用したプロジェクト群(Use automatic generation)標本数 17 と，用いないプロジェクト群(Nouse)標本数 23 のデータをプロットしたもので，自動生成を適用したほうが，1.6 倍ほど生産性が高い.

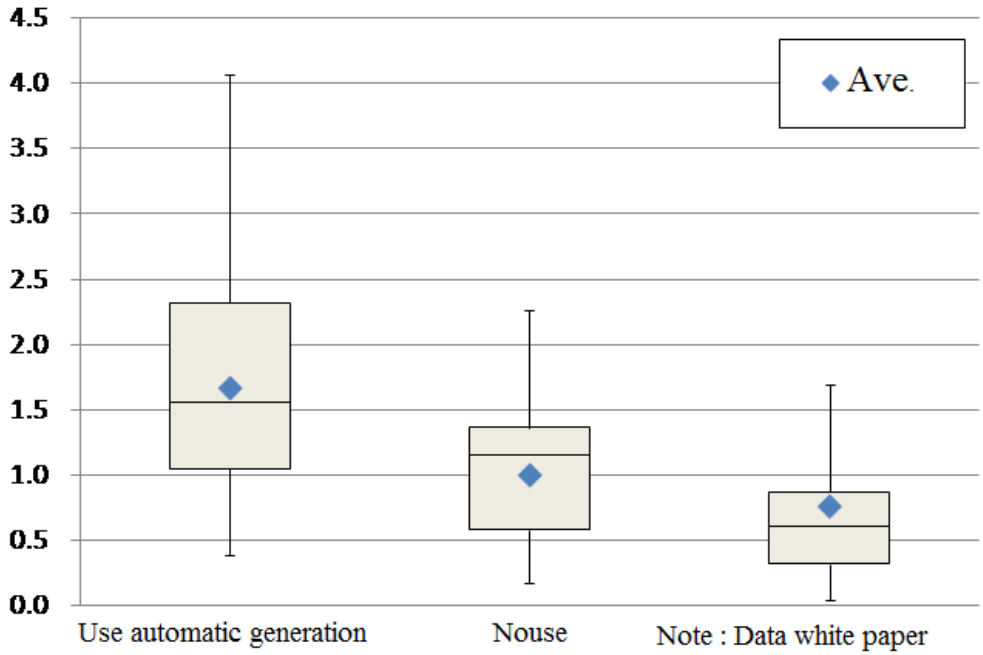


図 7-1　自動生成の効果

Figure 7-1　Effect of automatic code generation.

図 7-1 の右端は公開されているデータ[17]から同時期に開発されたもの(Data white paper)を示している.

生産性が向上した要因としては自動生成の効果があげられるが，層ごとに開発者を分けることが可能な開発プロセスを構成したことも一つの要因と考えられる.

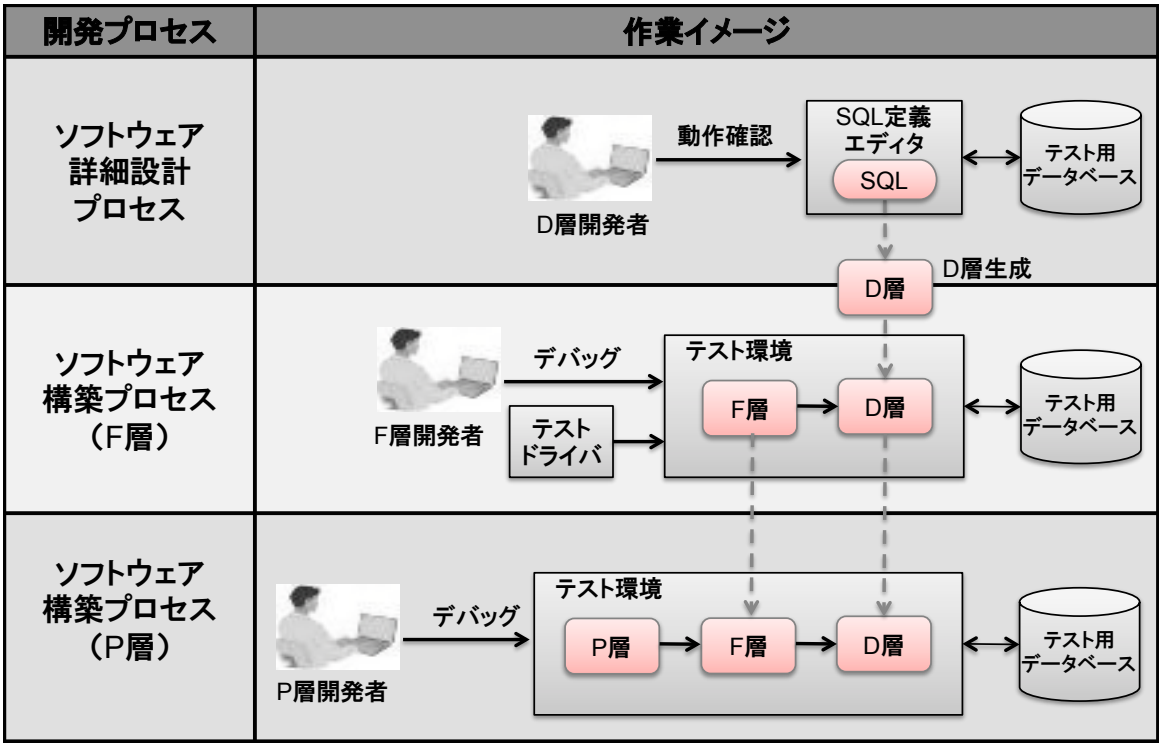


図 7-2　3 層に分かれた開発プロセス

Figure 7-2　Development process that divides into three layers.

図 7-2 はソフトウェア構造が 3 層となっていることから可能となった開発プロセスの流れを示している.

まず D 層の開発であるが，D 層は SQL の設計をソフトウェア詳細設計プロセスで行う際，SQL の機能テストまで完了させる．D 層は 100%自動生成であることから，以降の開発，テストは必要がない.

次に F 層であるが，F 層は自動生成された F 層内に手で追加コーディングを行い，続いて単体テストを実施する．この際，動作が保証されている D 層をテストスタブとして利用する．一般には F 層のデータベースアクセス部分はテストスタブを作成して単体テストを行う必要があるが，D 層は事前に作成済みであり，テストスタブの作成工数の削減と，品

質の保証されたD層を直接利用する効果で，F層開発の生産性も向上する．またF層は図3-36に示すようにサーバサイドのミドルウェア上に実装されるケースが多いが，F層を自動生成する際に，テストドライバも併せて生成されるため，テストの準備工数が大幅に削減される．

なお，一般的に単体テストは開発言語の開発環境内に閉じてホワイトボックステストを行い，ミドルウェアやデータベースに接続してのテストは組合せテストの最初に行うとする開発プロセスが多いが，本ツールを利用すればこれらのテスト作業を先取りして実施することが可能であり，これも生産性に大きく寄与する．

最後にP層であるが，F層と同様に自動生成されたP層内に手で追加コーディングを行い，続いて単体テストを実施する．この際，先に開発を終えたF層をテストスタブとして利用する．一般にはP層もF層をアクセスする部分はテストスタブを作成して単体テストを行う必要があるが，既に単体テストを終えたF層をテストスタブとして利用することで，テストスタブの作成工数の削減と，品質の保証されたF層を直接利用する効果で，P層開発の生産性も向上する．

一般に多層に分割する開発は分割損が発生し，設計コストもテスト工数も上昇すると考えられているが，3 層を独立に，かつ並行に開発し，テスト時は下位層をテストスタブとして利用することで分割損を押さえ，生産性を向上させられる．

また，各層の開発者を専任化することで，習熟が早くなり，さらに生産性を上げることが可能となる．1．3（1）においてツール開発の背景で述べたように，エンタプライズシステムの開発は，大量の人員を必要とし，ツールになれた技術者だけでプロジェクトを構成することは困難である．本ツールは層ごとに作業もツールも分かれるため，専任化しやすく，習熟効果を発揮しやすい．

(2)　品質

プログラムユニットごとのバグ発生数を自身のステップ数で除したものをバグ発生密度とし，横軸にプログラムの自動生成率をとって比較した．対象はP層，F層である．D層は100%自動生成であるためバグはないことから，この評価の対象外としている．

図7-3は2005年のCOBOLでの開発データでプログラム本数は約2000本，規模は約1.2MLOCである（その他にD層が約1MLOCある）．

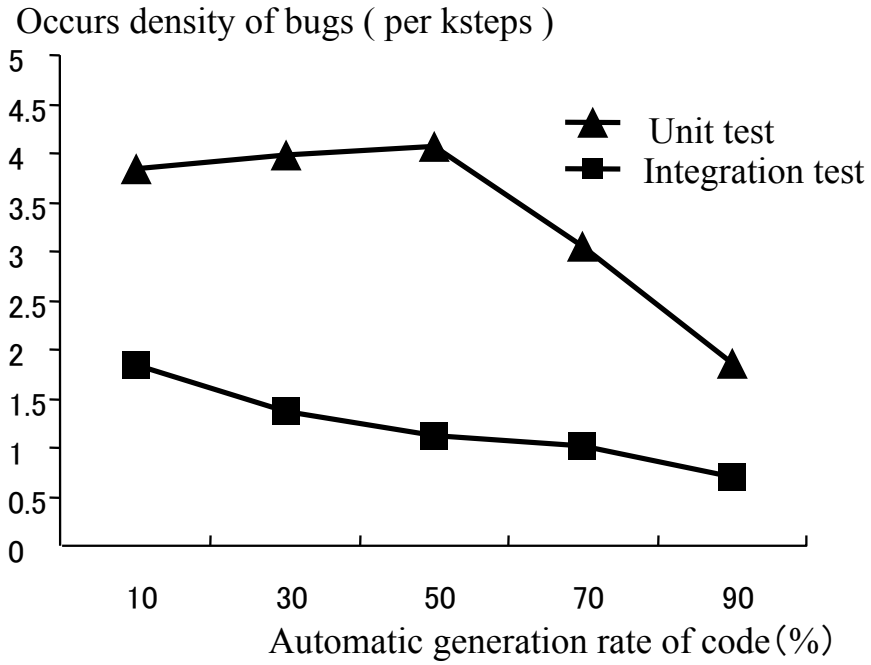


図 7-3　バグ発生密度

Figure 7-3　Occurs density of bugs.

自動生成率が高いほどバグ密度は減少する．特に 70%を超えると単体テスト(Unit test)で顕著である．

結合テスト(Integration test)でバグ密度減少がユニットテストより早く現れるのは，図 7-2 で示したように先行開発された D 層，F 層が，それぞれ F 層，P 層のテストスタブとして有効に機能したためと推定される．

7．2　自動生成の実績

2007 年の同時期に本番リリースをむかえた 4 プロジェクトのソース自動生成状況をコード行数で比較したデータを示す(図 7-4).

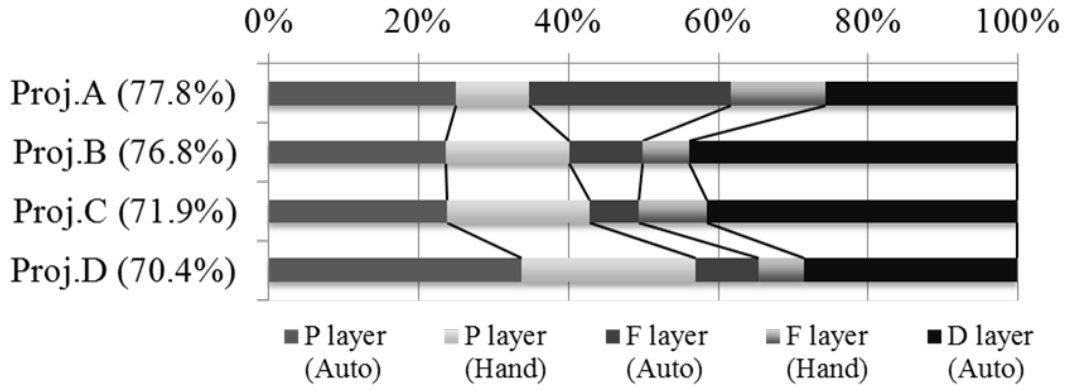


図 7-4　自動生成の実績値

Figure 7-4　Actual value of automatic code generation.

これらは流通(Proj.A，Proj.B)と公共分野(Proj.C，Proj.D)のシステムであり，言語はいずれも Java で，全てのシステムが標準提供している Struts をベースとした P 層生成ツールを採用している．規模は自動生成されたコードを含め 350KLOC から 1.7MLOC と幅がある．

プロジェクト全体の自動生成率は，いずれも 70%を超えており，安定して良好な自動生成率を達成している．（Auto はモデルから自動生成されたコード部分を示し，サンプルやテストコードとして生成したものは含めていない）．

P 層 F 層 D 層のコード行数比はプロジェクトの特徴を反映して異なっている．例えば Proj.D などはデータエントリ画面が多く，また入力項目数も多いことから P 層の規模が大きい．

7．3　マルチアーキテクチャの実現の実績

本ツールは，企業の IT 戦略に基づき，2002 年当時，需要が伸びると想定された COBOL，Java，.NET の 3 つの言語をサポートして来た．2011 年，約 150 プロジェクトでの利用実績を図 7-5 に示す．

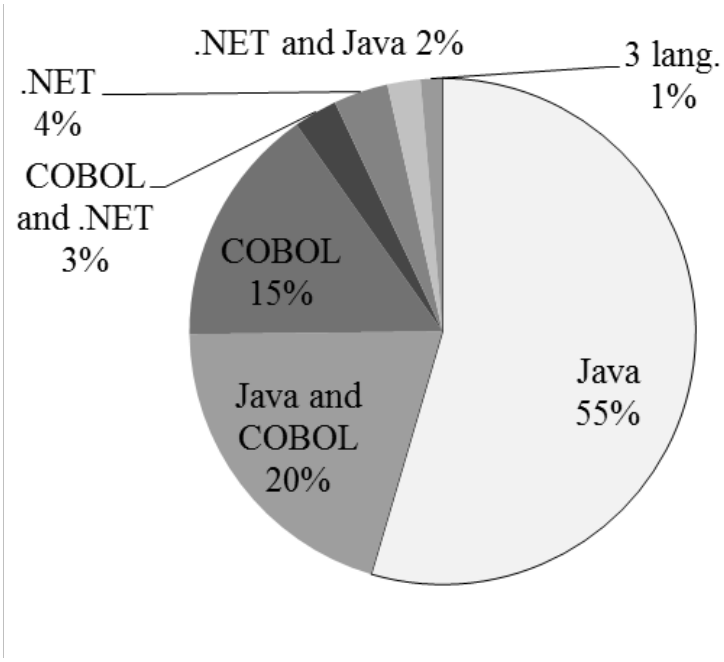


図 7-5　使用言語の比率

Figure 7-5　Percentage of the programming languages being used.

割合をみると Java が 78%，COBOL も 39%使われている．Java と COBOL など複数言語を利用しているシステムが 25%あるが，Java と COBOL の組合せの場合，ほとんどがバッチ処理を COBOL で，Web によるオンライン処理を Java でという組合せだった．マルチアーキテクチャを実現している一例である．

このような組合せでは第 6 章で述べた開発ツールの機能で D 層のモデル（設計情報）を共有することでバッチ処理向けに COBOL の D 層を生成し，オンライン処理向けに Java の D 層を生成することが可能で，開発生産性向上に寄与する．

ミドルウェア間の相性により制約はあるが，別々の言語で作られたコンポーネントを連携させて利用することも可能である．

典型的な例では，クライアントを.NET で，サーバを COBOL でという組合せで，初心者ユーザに配慮して UI を充実させると同時に，従来の専用端末のようにテンキーや PF キー操作が中心のオペレータが望む高速タイピングを実現させるシステム構成である（図 7-6）．

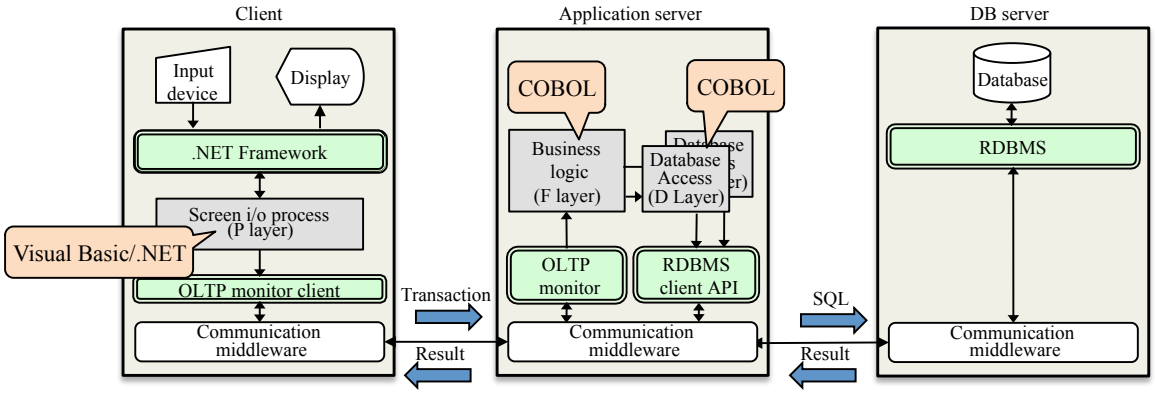


図 7-6　.NET と COBOL の混在システム

Figure 7-6　Coexistence system of .NET and COBOL.

ある事例では図 7-7 にあるように既存の COBOL 資産を活かしながら，Web ブラウザの使い勝手を求めて，OLTP モニタと Web サーバ連携機能を持つミドルウェア上に P 層が Java，F,D 層が COBOL という組合せがみられた．マルチアーキテクチャ機能が活かされていることがわかる．

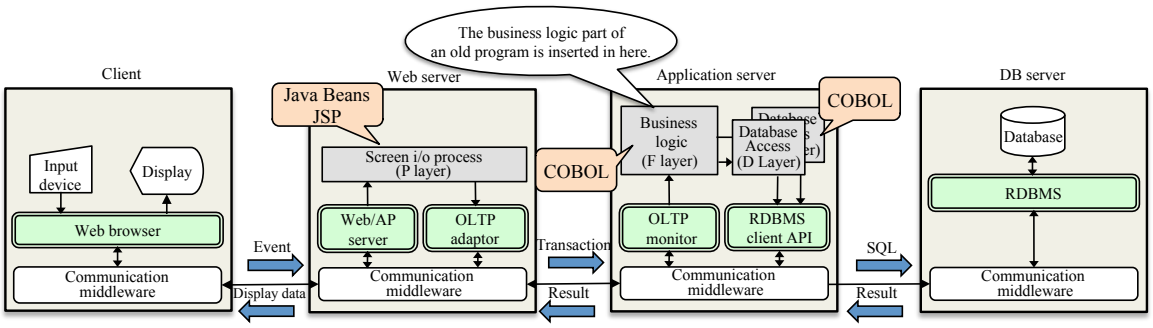


図 7-7　Java と COBOL の混在システム

Figure 7-7　Coexistence system of Java and COBOL.

COBOL 資産を活かす方法は，大きく分けて 2 通りあり，一つは今述べた図 7-7 にあるように，新しく COBOL の F 層，D 層を作り，ロジックだけを F 層内に移植する方法である．もう一つは，既存の COBOL 資産の状態が良く，新しい F 層に移植する必要はないケースで，データベースアクセス処理部分だけを RDBMS に対応するために D 層に置き換える方法である．いずれの方法も実際のプロジェクトで適用された方法である．

.NET と Java の組合せでは，基本的に Java でシステム全体を構築し，特別な操作者向けにリッチな画面（P 層）を.NET クライアントで構築した事例がある．この場合，.NET クライアントと Java が動く AP サーバの間は SOAP で接続する．

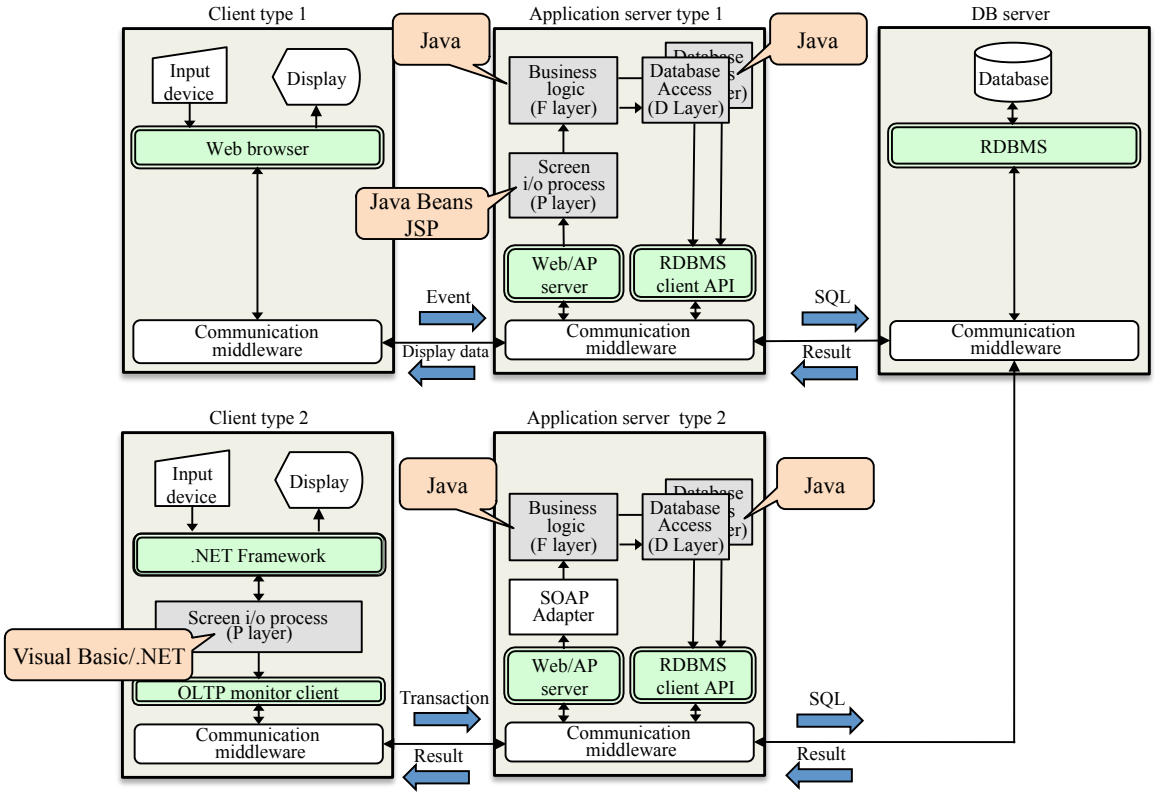


図 7-8　Java と.NET の混在システム

Figure 7-8　Coexistence system of Java and .NET.

Type1 では典型的な Java の実装で Web ブラウザをサポートし，Type2 はリッチクライアントを Visual Basic .NET で実現しているが，F 層，D 層は同じ Java の実装を利用している．

SOAP での通信を可能とするため，本ツールから図 6-5 でも触れた SOAP アダプタ生成を行って Type2 のシステムの F 層に実装している．これにより.NET クライアントとの通信を可能とした．

SOAP アダプタのサポートは，マルチアーキテクチャのサポート範囲を大きく拡大させた．SOAP は通信相手を.NET クライアントに限定する必要はなく，SOA の象徴である ESB(Enterprise Service Bus)に接続することはもちろんのこと，ERP パッケージからの

接続も可能となった.

加えて F 層から SOAP，REST でのサービス呼出し機能もサポートする．これにより ERP パッケージの機能を呼び出すことも可能であるし，また ESB 上のビジネスプロセスも呼出せる.

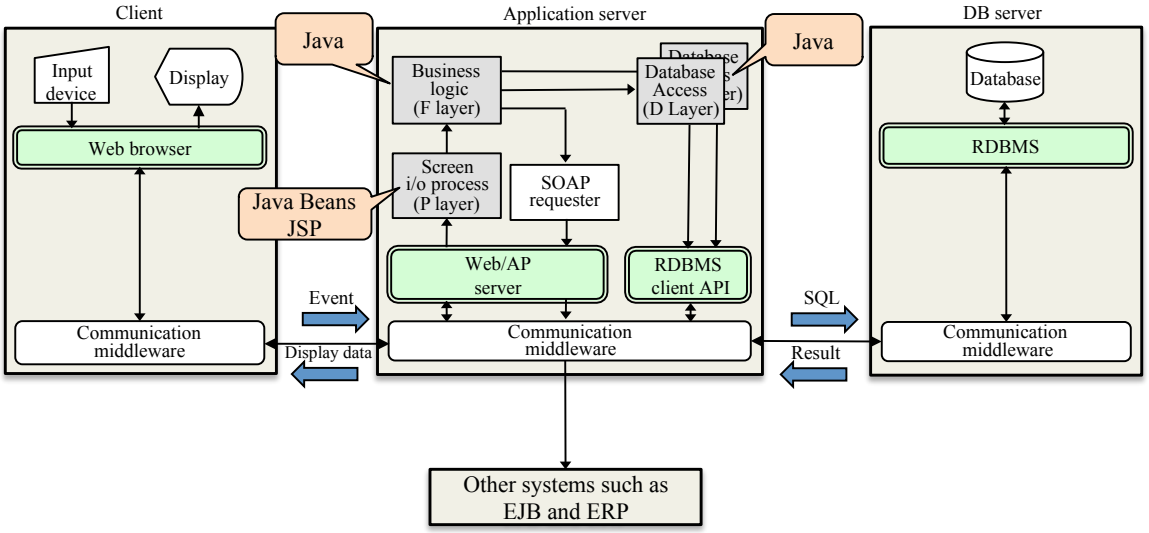


図 7-9　SOAP による他システムの呼出し

Figure 7-9　Call of another system by SOAP.

図 7-9 は，F 層から SOAP によって他システムとの連携を行った事例を示している.

接続先は SOAP 呼出しの口を持っているものであれば接続可能であり，具体的事例では，認証サーバへ登録情報の更新依頼や，購買システムから財務システムへ購買情報の転送など，緊急性は高くないが，ニアリアルで処理されることでビジネスプロセスが加速されるような使い方がこれまで多い．なお ESB の場合は BPEL で定義されたビジネスプロセスを呼出すことも可能である.

SOA は当初，ESB にのみ注目が集まった．ESB 上のビジネスプロセスをバッチ処理的に構成することも，また対話処理や対話処理の中に織り込んで使うことも仕様上は可能である．しかし，少なくとも登場当初は EJB 登場当初を思わせるように設定が複雑で，処理が重く，リアルタイムでの利用に難があった．従って対話処理やオンライン処理の中に織り込んで使う例は少なく，非同期処理やバッチ処理を制御するコンポーネントとして利用される例が多かった.

近年，本ツールはSOAPに加えREST(Representational State Transfer)もサポートした．COTS（既製品）として提供されるESBもRESTをサポートするものが増えてきており，今後はリアルタイム処理での利用も増える可能性がある．

SOAPやRESTのインタフェースをサポートするWeb向けの開発環境やサービス，スマートフォンなどのデバイスは日に日に増加しており，システムの拡張性を確保する武器となる．

以上，述べたように本ツールの利用によりマルチアーキテクチャは実現可能であることが示された．

7．4　長期にわたる反復型開発の実践

本開発ツールは，2002年から適用を開始し，2012年までに300を超えるシステム開発に適用されている．

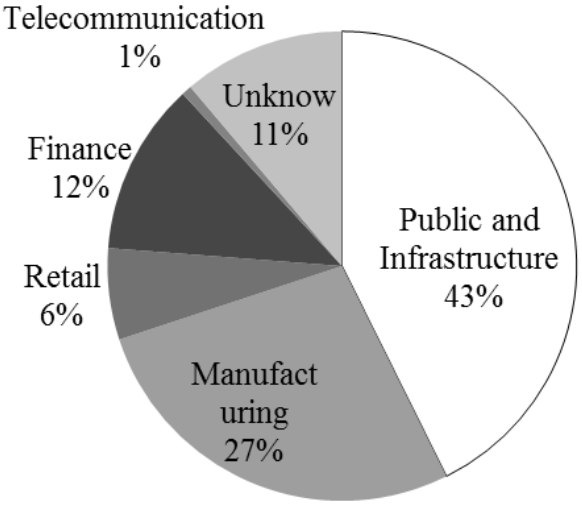


図7-10　適用先の業種

Figure 7-10　Industry which provides.

図7-10は2011年時点で継続してツールが利用されている約150システムの適用分野を示している。公共分野(Public and Infrastructure)と製造分野(Manufacturing)から適用をスタートしたが，現在は各業種向けエンタプライズシステムの開発に使用されている．

図 7-11 はツールを利用した期間を年数で示した．利用期間が 6 か月以下のデータは外してある（集計は 2012 年）．

現在もツールを利用しているシステムと既に利用を終えたシステムの数は半々でそれぞれ約 150 システムある．

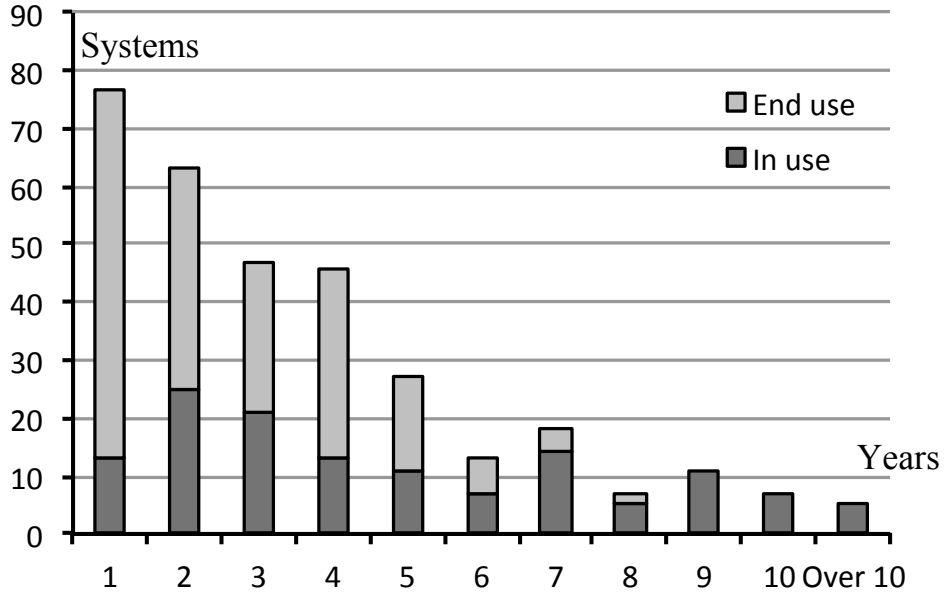


図 7-11　ツールの利用年数

Figure 7-11　Years using the tool.

利用が 2 年を超える約 10 のプロジェクトにヒアリングを行ったところ，どのプロジェクトも機能追加, 仕様変更が発生する度に反復型開発機能を利用して, ソース再生成を行っていた．ツールを導入して 10 年を超えるプロジェクトも出てきており，当初の想定通り長期間利用される基幹システムの開発，保守に継続的に使われている．

本ツールをパッケージ製品そのものの開発に利用しているプロジェクトもあり，パッケージ本体の製品機能強化を本ツールで継続的に行っている．

また，パッケージ製品に機能追加を行う場合は，パッケージが提供する API を利用して機能を作り込むことが一般的であるが，パッケージが提供するモデルを変更してコードを再生成し，機能拡張，追加を行う事例もみられた．これはドメイン特化機能再利用の一例を示しているといえる．

7．5　マネジメントへの活用の試み

本研究の成果であるモデルベース開発ツールを採用した2008年から2011年にかけての600～7700FPの規模の5つのプロジェクト（製造業4システム，流通業1システム）で，第5章に述べた見積手法を適用したデータを取得出来た．

各プロジェクトでは以下のように手法を適用した．

(a)　各プロジェクトで採用される言語，システムアーキテクチャ，モデルベース開発ツールのバージョンはそれぞれ異なるためLOCからFPへの換算はそれらを考慮して計算されている．

(b)　LOCは単純に計算で求めた訳ではなく，FPで計測不可能な複雑なビジネスロジックの考慮や，共通化可能なP層，F層内の検証機能，編集処理の集約などクラス単位の共通化設計は，それぞれのプロジェクトが独自に行っている．

（単位：KLOC）

	FPからの換算値	実測値（実装プロセス終了時）	見積誤差
製造業A	366	341	-7%
製造業B	605.6	694.8	+15%
製造業C	71	65	-8%
製造業D	50	55.5	+11%
流通業E	79.4	83.9	+6%

図7-12　見積との誤差

Figure 7-12　Error margin with estimate.

結果，求めたLOCは50～606KLOCで，実装プロセス終了時の結果は，いずれのプロジェクトも-10～＋15%の見積誤差の範囲におさまり，実用に供すことが確認できた．

7. 6　まとめ

本章では，本研究で提案し，実現したモデルベース開発ツールのフィールドでの適用実績と成果をまとめた.

10 年以上に渡り，300 以上のシステム開発に利用され，生産性，品質向上に効果が見られた．また，マルチアーキテクチャの対応を裏付けるように，複数言語を利用したシステムが 25%もあり，また 8 つの基本システム構成に当てはまらない，複数の基本構成を利用したシステムや，SOAP や REST の対応により，SOA 型システムの実現や ERP との接続を実現する事例も出てきた.

適用された約 300 のうち，まだ半数の 150 のシステムが本ツールを継続して利用しており，保守の場面においても本ツールは有効であることが示せた．生成・再生成機能を利用した反復型開発を継続して利用していることが明らかになった．最長のものでは 10 年を超える利用がみられた.

見積手法についても一部プロジェクトで実際に試行利用の形ではあるが成果が見られるようになってきた.

第８章　結言

8. 1　総括

日本においては，企業の基幹業務に IT を導入して既に 40 年以上が経過し，IT の中に企業のノウハウが盛り込まれている．一方で今日においては激変する経済や社会情勢，トレンドに追従すべく，IT システムもスクラッチ開発のみに固執することなく，ERP パッケージや SaaS などを組合せてエンドユーザの期待に柔軟かつ迅速に対応することが求められている．

本論文では，このような状況に柔軟に対処可能なマルチアーキテクチャ対応を基本とし，日々発生する新たな要求を柔軟に取り入れる反復型開発に応えられるモデルベース開発ツールについて提案した．

まず 2．4 で設定した本論文が達成すべき目標の達成状況について総括する．

- ■ マルチアーキテクチャを実現できるアプリケーションのソフトウェア構成を提案する．その提案したソフトウェア構成が実際のエンタプライズシステム実現で利用され，成果を得ることが出来たかを示す．
- ■ トップダウンアプローチとボトムアップアプローチという相反する側面のあるモデルベース開発と反復型開発が共存できる開発プロセスと，それをサポートするツールの関係を含めて体系化し，さらにツールが生成すべきソフトウェア機能について明らかにする．
- ■ 本研究の提案するモデルベース開発ツールが，ソフトウェア規模の見積精度向上に寄与するとの仮説を掲げ，実際に変換精度の向上に取組んだ結果について報告する．
- ■ 上記 3 点で示した機能を備えたツールを実現し，フィールドでの長年の適用を経て，長期に渡るツールの価値の証明，生産性と品質への寄与を明らかにする．

それぞれ掲げた目標に対して以下のような成果を得た．

- ✓ 第 3 章において，マルチアーキテクチャを実現するために最低限サポートしなければならない 8 つの基本システム構成を定義し，次に 3 層のソフトウェア・コ

ンポーネントからなるアプリケーション構造を提案し，それらコンポーネントの組合せで全てのマルチアーキテクチャを満たす構造を定義した.

- ✓ 第 4 章において，モデルベース開発と反復型開発を統合した開発プロセスを体系化した. はじめにモデルベースで開発する範囲を一般的な開発プロセスの標準モデルから導きだし，モデルベースによる反復型開発と手作業による反復型開発の境界を決めた．開発ツールが生成する部分はモデルベースの反復型開発を行い，手作業でコーディングする部分は手作業で反復型開発を行う. 次にモデルベースの開発ツールが生成するソフトウェアの機能について規定した. ツールが一貫して生成する基本部分と，P,F,D 層の各々で特徴ある生成部分について明確化した.
- ✓ 第 5 章において，本研究の提案するモデルベース開発ツールが，ソフトウェア規模の見積精度向上に寄与するとの仮説を裏付ける変換方法を確立することが出来た.
- ✓ 第 6 章において，第 3 章から第 5 章までに提案した要求事項を満たす構造，機能を備えたツールを実現し，具体的な操作例を示しながら特徴的な機能の実現方法を示した.
- ✓ 第 7 章において，本ツールの実績と評価を示した．本ツールを 10 年以上継続して利用しているシステムも確認され，生産性と品質の向上に明らかな成果がみられた．また実践投入のなかで，想定通りの利用方法に加えて，当初は想定していなかった特定コンポーネントだけを利用する形態や，開発当初は予定していなかった SOAP や REST を無理なく追加サポートしたことで，活用範囲が広がり，SOA や RIA，スマートフォンなどへの対応も可能となってきた.

以上，本論文で掲げた目標は全て達成できたと考える.

次に本論文の特長的な成果について述べる.

第一章で，今日のエンタプライズ開発を支えるモデルベース開発ツールは，3 つのポイントを満たす必要があるとした.

1 点目は，今日エンタプライズシステムで求められる処理方式，システム構成を実現可能なことであったが，現在多くのシステムで採用されている 3 つの処理形態，5 つのシステムアーキテクチャを元に，ツールでサポートすべきシステム構成を明確化し，その上で動作させるアプリケーション構造を第 3 章で示した．P 層，F 層，D 層という 3 層の単純

なソフトウェア・コンポーネントの組合せで，定義したシステム構成をカバー可能であることを述べた．このアーキテクチャを採用したことで複数の言語，複数の処理形態を持ついわゆるマルチアーキテクチャで構成されるシステムの開発に多く適用された．

また近年では，完全に新規のシステム開発を行うプロジェクトより，既存システムの再構築が多く，このようなシステム開発では資産を如何に再利用するかも，効率的にシステムを構築するポイントとなるが，ビジネスロジック部分だけを移植するケースや，D 層のみ新規開発してリレーショナルデータベース対応を果たしたシステムなど，コンポーネントアーキテクチャのメリットを最大限活かした事例が多くうまれた．

さらにコンポーネント間の境界を適切に設けていたことからツールの開発当初は想定していなかった SOAP や REST といった今日標準あるいは業界標準的なインタフェースを後からサポートすることも可能となり，SOA への対応や，ERP パッケージ，RIA クライアントとの連携事例も出てきた．スマートフォンとの接続も可能となった．

2 点目は，ユーザと開発者がターゲットとなるシステムを描き，議論するために必要十分なモデルを採用することであったが，エンタプライズシステム開発に長年利用されてきた仕様記述を主として採用し，UML は適所に利用することを 4．1 で述べた．これにより本ツール導入の障壁は低く押さえられた．ツールの採用が特定の部門，業種に縛られず 300 を超え，広範囲の業種に広がった一因とも考えられる．

そして，3 点目は，1 点目で決めたシステム構造に沿って 2 点目で採用したモデルから自動でソフトウェアを生成可能とすることだが，4．2 で自動生成する機能を明確化し，第 6 章でそれを実現するツールを説明した．2002 年より 300 を超えるシステム開発への適用を通して，

- コードの高い自動生成率(70%超)
- コード自動生成による 1.6 倍の生産性向上
- 同様にコード自動生成がもたらす品質の向上

という成果を得た．

また反復型開発機能のサポートにより，半数のプロジェクトが保守フェーズに入ってもツールを継続使用しており，中には 10 年を超えるプロジェクトも登場している．

ベンチマークとして，典型的な MDA ツールとの比較においては，実装の自由度を求めるスクラッチ型開発というニーズに本ツールがより合致し実用的であり，また今日最もポピュラーな OR マッピングツールとの比較においても，D 層はあまり技術力の高くない技

術者を支援出来る UI を持ち，一定レベルの技術者を多く集めるのが困難なエンタプライズシステム開発の現状に沿っていて実用的である．また COBOL 言語のサポートを含むマルチアーキテクチャ対応という点においても本ツールが優れているといえる．

ソフトウェア開発プロジェクト，特に新規にソフトウェアを開発するケースにおいては，ソフトウェアの規模がコストや進捗を代替的に表現可能とする重要な尺度である．本研究の成果であるモデルベース開発ツールを採用した場合，ソフトウェアは構造が統制され，コーディングに制約を設けられる．この制約を利用し， FP 計測時に取得する係数をうまく利用することで LOC への変換精度を高めることが可能となった．

この技術を利用すればプログラムコンポーネント単位で規模を推定できる．これにより進捗が把握しやすくなり，外れ値の解析による不良検出にも利用できるなど，マネジメントへも大きく寄与することを示した．

長年取り組んできた中で，多くの課題が見つかり都度対処も行ってきたが，今後は継続した機能拡張に加え，並行して軽量化にも取り組み，さらに適用範囲を広げていく．

8．2　今後の課題

(1)　新たな開発プロセスへの対応

エンタプライズシステムの開発においても Agile 型の開発プロセスを試行するプロジェクトが多く見られるようになってきた．エンタプライズシステム開発において，Agile 開発のプラクティスを実施するには，長く日本型ウォータフォール開発で呪文のように唱えられてきた以下の「呪縛」に打ち勝たなければならない．

- 手戻りは時間と工数の無駄．だから仕様は上流できっちり決定する．
- ひとつひとつ工程の完成度（上流であれば設計書の完成度）を高めて次の工程に進めることが，システム全体の品質向上につながる．

しかし，ウォータフォール型開発で成功している人は，工程のあちこちで小さな改善プロセスを回して，工程全体が遅れないように未然に防止している．たとえば大きな工程が開始される前に準備プロセスを起こしてリスクを未然に低減させ，また工程の中途でも思

い通りの進捗が得られないならば，大きくサポート内容，契約内容，作業内容を見渡し，実行可能な案に作り変えて，ステークホルダと積極的に調整している．プロジェクトの開始時に描いたスケジュールどおりに進むプロジェクトは無いことから，「呪縛」は単なる原則を述べたに過ぎない．

また，Agile に対しては，以下のような偏見があった．

- Agile はドキュメントを作らない（あるいは軽視している）．
- Agile はいつでも変更を取り入れなければならない．ルールも計画もない好き勝手な手順で作業を行なう．

少し表現を誇張しすぎたかもしれないが，当初は，このような受け止められ方をしていたと思う．

実際には Agile はプログラムの開発量（スループット）が最大になるようにするための開発手法であり，プログラマの生産性を最適化するようにプログラム開発以外の作業に従事する人は阻害要因の排除に注力する．これらサポートを行う人もプログラム開発を行なう人と同じスピードで走る必要がある．たとえば開発環境を整備する人，テスト自動化を受け持つ人は，プログラマが開発を開始する前，テストする前にそれぞれの環境を整備する必要があるということである．仕様を調整する人はプログラマが手持ち無沙汰にならないように，必要とされる時期から開発の優先順位を計算してユーザと調整し，仕様を決めていく必要がある．

また変化を積極的に受け入れるという考え方は，顧客が本当にほしいシステムは常に変化するのだから，その思いに応えられるように開発の優先順位を機敏に変れる柔軟性を持とうということである．対応するためには柔軟に開発作業を組み替えられる必要がある．また示された仕様を全て鵜呑み丸呑みすることは物理的に不可能であるのだから，タイムボックスを意識して期日に間に合うよう，積極的に交渉すべきということになる．

柔軟に開発の優先順位を変更できるようにするためには，仕様が決まってからプログラムが出来るまでの期間が短いほど良い．よって後から見ることのないドキュメンテーションはやめて，ソースをきれいに（コメントを含め）書いて，一日でも早く動くシステムをユーザに見せてフィードバックをもらえということを説いているに過ぎない．プログラムをきれいに書くことで必要の無くなるドキュメントは書くべきではなく，一方でソース

コードでは語られない仕様はきっちりドキュメントを作成するべきである.

ウォータフォール，Agile どちらが優位な開発プロセスであるかというよりは適所があるし，また一方を軽視するのも間違っている.

近年，CI(Continuous Integration)，CD(Continuous Delivery)，DevOps(Development-Operations)といった IT 用語が盛んに語られるようになった．CI は継続的インテグレーションであり，テストの自動化を含め，構成管理から最新のソースを抜き出し，ビルド，デプロイ，テストまでの工程を自動化することである．CD ではさらに本番環境までのデプロイまでを視野に入れている． DevOps は開発と運用の一体化を標榜した言葉であり，CD に加え，本番時のアプリケーションの監視，稼働分析までも視野に入れ，開発のみでなく運用までを含めた開発・運用ライフサイクルの自動化を指している．ユーザの利益を最大化するために開発と運用は協力，連携して，いち早く求められる IT サービスを開発し，提供しようという考え方である.

これらの用語は，Agile が目指していることを実現するために奨励される具体的行動を述べている.

今日まで Agile がエンタプライズシステムの開発にあまり取り入れられなかったのは，これまでオーダーメイド的なスクラッチ開発が主だったため，要件定義⇒設計⇒構築⇒テストという繰返しのない一方向の開発プロセスが関係者全員の脳裏にあり，Agile のような反復型になじまなかったのも主因であった．しかしここ 1，2 年で B to B の分野でも SaaS を提供する企業が多くなり，またそのサービスを受ける企業も多くなってきた. CI，CD，DevOps といった考えは，主に B to C のサービスを行なう IT サービス企業が取り入れ発展させてきたものだが，今は B to B の分野でも浸透しつつある.

これらはデリバリサイクルの長短はあるものの，いずれ SaaS 自身の開発だけでなく，パッケージ適用開発，そしてスクラッチ開発においても取り入れられていく可能性が高い.

以前は IT システムの開発というと最初のリリース時に必要な機能が全てそろっていることが常識と捉えられ，ウォータフォール型の開発プロセスを採用せざるをえなかったが，その後のメンテナンスコストを考えれば，最初に必要最大限の開発を行なうのではなく，必要最小限からスタートして徐々に機能を拡充させる Agile 型の開発スタイルもライフサ

イクルコスト低減の観点から受け入れられる可能性がある.

このような開発プロセスが主流になっても本開発ツールは十分適用可能と考えている. 本開発ツールは，反復型の開発プロセスを定義しており，そのプロセスの中でモデルベースによる反復型開発と手作業による反復型開発があり，その間に明確な境界がある.

CI ではビルドからテストの自動化までを対象としているが, 開発ツールをその中に加えることで，プログラムの自動生成からテストまでを自動化するという考え方も出来ると捉える.

今後は, スクラッチ開発にも CI や CD といった考え方が大いに取り入れられると考え, 私の開発ツールもその一員として活躍できるよう取組んでいきたい.

⑵　MDE への取組み

本研究で開発したモデルベース開発ツールが提供するモデル検証とは主に以下の機能を指す.

- CASE ツールから培ってきたデータ項目辞書を中心とする各モデル間の整合性検証機能
- 同様にデータ項目辞書に基づいたチェック，編集処理の生成処理機能
- P,F,D 層と命名したソフトウェア・コンポーネント間インタフェースの整合性検証機能
- SQL 定義エディタによって制約された環境での SQL 生成機能

これらは有効に機能していると考えるが，いずれも各データ項目，各クラス，インタフェース，データベース構造などの静的構造に着目した検証機能である．一方でエンタプライズシステムでも，複雑なビジネスルールやビジネスプロセス，複雑なイベント制御・画面遷移，アプリケーション間の通信プロトコル，データフローなど動的な処理の流れを仕様としてまとめることが必要なケースもあり，いずれもモデル化する対象である.

これらは近年，形式手法の導入などでまた研究が盛んになりつつあるが，多くの開発者が利用するにはまだまだ障壁が高い．多くの技術者が利用できるドメイン固有の形式的表現に取り組み, いつしか動的モデルの検証機能についても本ツールに取り入れていきたい.

(3)　NoSQL データベースへの対応

現時点において事例はないが，機会さえあればすぐにでも対応が求められるものにNoSQLデータベースを使ったアプリケーション開発がある. 現時点においてNoSQLデータベースはビッグデータの蓄積・解析処理に使われる例が多いため，解析処理を行うシステムと従来のエンタプライズシステムとはデータベース間のバッチ的なデータ転送で運用されている例が多い.

しかし, 解析処理に要す時間, あるいは求められる時間が短くなってくると, データベース間の同期間隔が短くなり，さらにはエンタプライズシステムから直接解析処理を呼出すようになってくる可能性がある.

NoSQL データベースはビッグデータ解析処理以外の活用方法も広く考えられ，現在のリレーショナルデータベースの代替となるケースもあり得る. 継続して注目していきたい.

謝辞

本研究を進めるにあたっては，多くの方々にお世話になりました．ここに感謝の辞を述べさせていただきます．

まず，本研究を直接ご指導いただいた田野俊一教授に深く感謝申し上げます．田野先生には，時として方向性を失う私の論旨を常に正し，冷静に取組めるよう熱心に暖かくご指導頂きました．

本論文をまとめるにあたり，様々なご助言，提言をいただきました渡辺俊典名誉教授，舩橋誠壽氏，増位庄一氏に厚く御礼申し上げます．

審査を快く引き受けてくださいました大学院情報システム学研究科の小池英樹教授，末廣尚士教授，大須賀昭彦教授，栗原聡教授に感謝申し上げます．また，本論文を熟読頂き，誤りを見つけていただいた情報メディア学講座事務員の岸本雅代氏に感謝申し上げます．

本論文をまとめるきっかけと電気通信大学というすばらしい研究環境を紹介してくださった株式会社日立製作所インフラシステム社中野利彦氏に感謝します．また職場より快く本学への入学を認めてくださった前横浜研究所所長の堀田多加志氏に感謝します．加えて，異動で職場に慣れていない私を気にかけ，本研究の継続を支援頂いた株式会社日立システムズ大野治専務，生産技術本部の皆様に感謝いたします．

本論文で述べたツールの開発および多くのプロジェクトへの 10 年以上に及ぶ適用成果は，株式会社日立製作所情報・通信システム社プロジェクトマネジメント統括推進本部，並びに生産技術本部，横浜研究所を中心に多くの方々の長年に渡る継続的な取組みが実ったものです．これらの方々の努力と知恵と粘り強さがなければ実現することはありませんでした．深く感謝申し上げます．

最後に平日の夜には私の好きなコーヒーを煎れ，休日には茶店に長く籠る自由を与えてくれた妻，潤子に感謝します．

関連論文

第２章 2．１および2．2，第３章，第４章，第６章，第７章

・ 石川貞裕，田野俊一, "反復型開発を実現するマルチアーキテクチャ対応モデルベース開発ツールの本格適用，電子情報通信学会論文誌，Vol.J96-D,No.10,pp.2226-2239, 東京，Oct.2013.

第２章 2．3，第５章

・ Sadahiro Ishikawa and Shun'ichi Tano, "Improvement of Software Size Estimation Method That Contributes to Project Management，6th International Conference on Project Management (ProMAC2012)，Honolulu, Hawaii, USA, 2012.

・ 石川貞裕，田野俊一, "プロジェクトマネジメントに寄与するソフトウェア規模評価手法，電子情報通信学会論文誌．＊投稿中

参考文献

[1] 社団法人日本情報システム・ユーザー協会(JUAS), "3.2.1 平成 21 年度議論内容の背景(平成 20 年度の課題認識), "IT 経営普及促進に向けた調査研究 -報告書-, JUAS, p.38, 東京, 2010.

[2] 一般社団法人 日本情報システム・ユーザー協会(JUAS), "3 IT 投資マネジメント, "企業 IT 動向調査報告書 2012, JUAS, pp.72-102, 日経ＢＰ社, 東京, 2012.

[3] 一般社団法人 日本情報システム・ユーザー協会(JUAS) , "7.7 代表的な基幹業務システムのライフサイクル, "企業 IT 動向調査報告書 2012, JUAS, pp.228-233, 日経ＢＰ社, 東京, 2012.

[4] C. Zetie, "MDA is DOA, party thanks to SOA, "Forrester Research, March Inc., Cambridge, March 22, 2006.

[5] D.C. Schmidt, "Model-Driven Engineering, "Proc. IEEE Computer, Vol. 39, No. 2, pp.25-31, February 2006.

[6] 中井奨, "バージョンアップを賢く乗り切る, "日経コンピュータ 2010.4.14, pp.21-41, 日経ＢＰ社, 東京, 2010.

[7] P. Mohagheghi, V Dehlen, "Where is the Proof? – A review of experiences from applying MDE in industry, "Proc. 4th European Conference on Model Driven Architecture Foundations and Applications (ECMDA '08), no.LNCS5095, pp.432-443, 2008.

[8] J. Hutchinson, J. Whittle, M. Rouncefield, S. Kristoffersen, "Empirical assessment of MDE in industry, "Proc. the 33rd International Conference on Software Engineering(ICSE '11), no.ss, pp.471-480, Honolulu, USA, May 21-28, 2011.

[9] 大野治, 小室彦三, 降旗由香里, 渡部淳一, 今城哲二, "多次元部品化方式によるソフトウェア開発の自動化, "信学論, D-I, J84-D-I, no.9, pp.1372-1386, 東京, Sep, 2001.

[10] M. Tsuda, S. Ishikawa, O. Ohno, A. Harada, M. Takahashi, S. Kusumoto, K. Inoue, "Effectiveness of an integrated CASE tool for productivity and quality of software developments, "Proc. IEICE Trans. Inf. & Syst., vol.E89-D, no.4, pp.1470-1479, April 2006.

[11] 石川貞裕，北川誠，宮崎肇之，生形知一, "企業システムの構造改革を加速するアプリケーションアーキテクチャ, "日立評論 2004.6，vol86, no.6, pp.407-410，日立評論社，東京，2004.

[12] 経済産業省, "I.総論, "情報システムの信頼性向上に関するガイドライン 第2版, p.1, 東京，2009.

[13] 社団法人日本情報システム・ユーザー協会(JUAS) , "3.2.1 平成 21 年度議論内容の背景(平成 20 年度の課題認識), "IT 経営普及促進に向けた調査研究 -報告書-, JUAS, pp.28-37，東京，2010.

[14] 社団法人日本情報システム・ユーザー協会(JUAS), "5. 2 プロジェクト特性, "ユーザー企業ソフトウェアメトリックス調査 2013，pp.43-45，JUAS，東京，2013.

[15] 立行政法人情報処理推進機構(IPA) 技術本部 ソフトウェア・エンジニアリング・センター(SEC), "4. 4 システム特性, "ソフトウェア開発データ白書 2012-2013, pp37-41, SEC，東京，2012.

[16] C. Jones, "Software Technology Adjustment factors, "Estimating software costs: Bringing realism to estimating, 2E., pp.340-343, McGraw-Hill, New York, 2007.

[17] 社団法人日本情報システム・ユーザー協会(JUAS) , "5. 2 プロジェクト特性, "ユーザー企業ソフトウェアメトリックス調査 2013，p.41，JUAS，東京，2013.

[18] Software Engineering Center, Information-technology Promotion Agency, Japan (IPA/SEC), "Principle 5: Multistage estimation reduces risks for the acquirer as well as the supplier, "The Seventeen Principles for System Development A "Cho-joryu" Approach, p.7, May 2, 2012, http://www.ipa.go.jp/english/sec/reports/20120502.html.

[19] M. Kurashige, A. Harada, "A practical use of an estimate method at early stage of business application software development, ProMAC2006, The Society of Project Management，Tokyo，2006.

[20] C. Jones, "Source Code Sizing, "Estimating Software Costs: Bringing Realism to Estimating Second Edition, pp.269-275, The McGraw-Hill Companies, New York, 2007.

[21] Software Engineering Center, Information-technology Promotion Agency, Japan (IPA/SEC), "8.1.3 Phase-Based Effort: Development, "IPA/SEC White Paper 2007

on Software Development Projects in Japan (Fully translated English Edition of the White Paper 2007 in Japanese)， p.226， Tokyo， 2010, http://www.ipa.go.jp/english/sec/reports/20100507a_1.html.

[22] 大野治，小室彦三，降旗由香里，今城哲二，古宮誠一，"多次元部品化方式によるソフトウエア開発の自動化−バッチプログラム用スケルトンの作成とその十分性−，"子信学会，論文誌 J83-D-INo.10，pp.1055-1069, 2000.

[23] A. ウマー，"5.2.2.3 分散データおよびトランザクション管理，"情報システムテクニカルガイド，pp.271-273，株式会社トッパン，Tokyo，1995.

[24] J. Edwards，"第 1 章なぜクライアント/サーバが注目されているか，"3 層 C/S コンピューティングケーススタディ，pp.4-19，株式会社朔泳社，東京，1998.

[25] 吉川和巳, "GoogleAppEngine 入門, "日経 SYSTEMS 2010.9，pp.86-90，日経ＢＰ社，東京，2010.

[26] 社団法人日本情報システム・ユーザー協会(JUAS), "5. 2 プロジェクト特性, "ユーザー企業ソフトウェアメトリックス調査 2013，pp.44-45，JUAS，東京，2013.

[27] 立行政法人情報処理推進機構(IPA) 技術本部 ソフトウェア・エンジニアリング・センター(SEC), "4. 4 システム特性, "ソフトウェア開発データ白書 2012-2013， p.41，SEC，東京，2012.

[28] I. Jacobson, M. Griss and P. Jonsson, "Software reuse, ACM Press, Inc., 1997.（邦題："ソフトウェア再利用ガイドブック，pp.76-77，株式会社トッパン，東京，1999.）.

[29] R. オーファリ，D. ハーキー，J. エドワーズ，"第 2 章クライアント/サーバコンピューティングの世界へようこそ，"SE のためのサバイバルガイド，pp.22-30，日経 BP 社，東京，2000.

[30] F. Buschmann, R. Mounier, H. Rohnert, P. Sommerlad and M. Stal, "2. 4 対話型システム, "ソフトェアアーキテクチャ, pp.120-139, 株式会社トッパン, 東京, 1999.

[31] 鷲崎弘宜, "ソフトウェアパターン 概観, "情報処理 Vol.52 No.9 Sep. 2011，p.1123，情報処理学会，東京，2011.

[32] F. Buschmann, R. Mounier, H. Rohnert, P. Sommerlad and M. Stal, "2. 4 対話型システム, "ソフトェアアーキテクチャ, pp.140-164, 株式会社トッパン, 東京, 1999.

[33] Sun Microsystems，"Core J2EE Patterns - Data Access Object，2000-2001，

http://www.oracle.com/technetwork/java/dataaccessobject-138824.html.

[34] F. Marinescu, "Version Number, "EJB デザインパターン, pp.90-96, 日経 BP 社, 東京, 2003.

[35] ISO, "Software specific process, "ISO/IEC 12207:2008(E) Systems and software engineering - Software life cycle processes, pp.57-66, ISO, Geneva, 2008.

[36] 立行政法人情報処理推進機構(IPA) 技術本部 ソフトウェア・エンジニアリング・センター(SEC), "2. 4 ソフトウェア実装プロセス, "共通フレーム 2013, pp.152-168, IPA, 東京, 2012.

[37] 立行政法人情報処理推進機構(IPA), "第 4 章 超上流工程でやるべきことと役割分担, "経営者が参画する要求品質の確保, pp.41-69, IPA, 東京, 2006.

[38] 独立行政法人 情報処理推進機構(IPA) ソフトウェア・エンジニアリング・センター(SEC) エンタプライズ系ソフトウェア開発力強化推進委員会 要求・アーキテクチャ領域 機能要件の合意形成技法 WG, "機能要件の合意形成ガイド, IPA, 東京, 2010.

[39] 石川貞裕, 向坂太郎, 宮崎肇之, 後藤卓司, 福士有二, "機能要件設計書だけで 20 種類 役割を知り, 書き方をつかむ, "日経 SYSTEMS 2007 年 11 月号, pp.34-41, 日経 BP 社, 2007.

[40] P. Marian, "UML in practice, "35th International Conference on Software Engineering (ICSE 2013), San Francisco, CA, USA 18-26 May 2013.

[41] K. Beck and 16 other people, "Manifesto for Agile Software Development, http://agilemanifesto.org/, 2001.

[42] R. Miles, K. Hamilton, "UML の度合い, "入門 UML2.0, p.12, オライリー・ジャパン, 東京, 2007.

[43] 石川貞裕, 田野俊一, "反復型開発を実現するマルチアーキテクチャ対応モデルベース開発ツールの本格適用, "電子情報通信学会論文誌, Vol.J96-D, No.10, pp.2226-2239, 東京, Oct. 2013.

[44] INTERNATIONAL STANDARD, "ISO/ICE 20926 Software and systems engineering – Software measurement – IFPUG functional size measurement method 2009, ISO/ICE, 2009.

[45] J.J. Edmunds, "SAA/LU6.2, McGraw-Hill, Inc., New York, 1992.

[46] 片山卓也, 深谷哲司, 篠原郁二, 亀尾和弘, 銀林純, 丸山勝巳, 中島震, 神谷慎吾, 塚本英昭, "1.1. エンタプライズ・ソフトウェアの位置付け, "エンタプライズ・ソフトウェア生産革新プロジェクト, http://www.cocn.jp/common/pdf/0904_EnterpriseSoftware_v2.pdf, pp.2-3, 産業競争力懇談会(COCN), 2010.

[47] 総務省情報通信国際戦略局, 経済産業省大臣官房調査統計グループ, "第 5 章 情報サービス業, "情報通信業基本調査速報 平成 25 年情報通信業基本調査, http://www.meti.go.jp/statistics/tyo/joho/result-2/h25sokugaiyo.pdf, pp.39-43, 総務省情報通信国際戦略局, 経済産業省大臣官房調査統計グループ, Oct. 2013.

[48] 社団法人日本情報システム・ユーザー協会(JUAS) , "5.2.1 業務種別, "ユーザー企業ソフトウェアメトリックス調査 2013, p.36, JUAS, 東京, 2013.

[49] 立行政法人情報処理推進機構(IPA) 技術本部 ソフトウェア・エンジニアリング・センター(SEC), "4. 3 利用局面, "ソフトウェア開発データ白書 2012-2013, p.34, SEC, 東京, 2012.

[50] Anneke Kleppe, Jos Warmer, Wim Bast, "OptimalJ DEMO Edition の使用方法, "MDA 導入ガイド, pp.213-227, 株式会社インプレス, 東京, 2003.

[51] Jboss Community, "Hibernate, Hibernate.org, http://www.hibernate.org/, Red Hat Inc., 2013.

[52] The MyBatis Team, "MyBatis , http://blog.mybatis.org/ , Apache Software Foundation, 2013.

著者略歴

石川　貞裕（いしかわ　さだひろ）

1980 年 4 月　青山学院大学　理工学部経営工学科入学

1984 年 3 月　青山学院大学　理工学部経営工学科卒業

1984 年 4 月　株式会社日立製作所入社

2012 年 4 月　電気通信大学　大学院情報システム学研究科　博士後期課程入学

2013 年 4 月　株式会社日立システムズ入社

2014 年 3 月　電気通信大学　大学院情報システム学研究科　博士後期課程修了