レガシー・マイグレーションの現実解

The most recommended solution for legacy migration

中 村 修 二

要 約 昨今，レガシー・マイグレーションがIT業界およびITを活用している企業において話題となっている．ITを取り巻く環境が大きく変化している中で，10年以上前に汎用機等で構築されたレガシー・システムでは限界に達してきているからである．保守性・拡張性・迅速性を実現するにはレガシー・システムを破棄し，システムを再構築する必要がある．しかし，移行コスト・期間および本番移行リスクを考えた場合，一足飛びにシステムの再構築を実現することは難しい．日本ユニシスが考える現実的で最適なレガシー・マイグレーションは，オープン系プラットフォームに移行する場合は，リライト手法によるシステムの構築を1'stステップとし，段階的にサブシステム単位に再構築していく方法である．

Abstract Recently, the legacy migration has attracted much interest in the IT industry and enterprises that make use of IT technologies. Amid the significant changes in circumstances surrounding IT, the legacy system, consisting of technologies of a decade ago represented by mainframes, is reaching its performance limit. To achieve maintainability, scalability and promptness, it is necessary to destroy the legacy system for restructuring. However, considering the cost and time for migration works as well as the risk of shifting into operation under the production environment, restructuring the system at a single leap would be unrealizable approach. What Nihon Unisys considers to be the most suitable yet realistic way to migrate the legacy system to an open platform would be the stepwise migration, in which system build employing the rewriting methodology is done as the first step and followed by rebuild on a subsystem basis.

1. はじめに

企業での事業内容や事業構造の移り変わりが激しくなってきている．その中で，IT戦略が経営の大きな要素の一つとなってきており，経営や企画部門からの要求に瞬時に対応できる情報システムの構築が要求されている．また，ITにおける最大の経営課題はITコストの構造改革と考えられており，情報システムのコスト削減と適正化が重要視されている．従来，汎用機で構築されてきたレガシー・システムは，長年保守を繰り返してきたことによりシステムが所謂スパゲッティ状態となっていることが多い．そのため，運用・保守コストが肥大化し，さらに環境の変化に柔軟に対応できる構造となっていないことも少なくない．そこで，レガシー・システムを刷新し新たなシステムへ移行することが，企業の大きな課題の一つとなっている．

このような状況を受けて，各ベンダ・SIerが各種レガシー・マイグレーション・サービスメニュや取り組みに関して対外的に発表している．日本ユニシスも「オープン移行サービス」というレガシー・マイグレーション・サービスメニュを用意している．ユーザ状況を受けて，ベンダやSIerのレガシー・マイグレーション・サービス環境が整ってきたような感がある．

本稿では，まずはレガシー・システムにおける現状を考察し，なぜ新たな情報システムへの移行が必要なのかに言及する．また，移行手法や移行手法毎の特徴について述べ，移行方法の現実解についても合わせて言及する．最後に，資産毎の移行時の注意点についても一部言及す

る．

2. レガシー・システムの現状

レガシー・システムとは，一般的にはメインフレームやオフコン上で稼働しているシステムのことを指す場合が多いが，UNIX システム上で構築されたシステムのことを指す場合もある．これは，例え UNIX 上で構築されたシステムであっても，システム初期構築後から長年が経過し，その間の追加・改修により業務ロジックが肥大化・複雑化している可能性があるからである．本稿ではおもに 10 年以上前に構築され，その後の追加・改修などにより AP 構造が肥大化・複雑化したシステムのことをレガシー・システムと呼ぶ．

レガシー・システムの代表的な特徴は，以下の通りである．

① システムが肥大化，複雑化している
- 長年の追加改修により，システムが所謂スパゲッティ状態となっており，保守コストが増大している
- デッドコードや不要コードが増大し，システム改修コストが増大している
- 業務追加などのシステム改修に時間を要する

② システムが属人化している

2007 年問題に代表されるようにシステム保守/運用が特定の要員でのみ可能

システムの属人化・肥大化や複雑化によりレガシー・システムは一般的に，「TCO の増大」「スピード経営を実現出来ない」等の問題を抱えていると言われ，企業経営の足枷となっていることが多い．「TCO の増大」は企業収益を圧迫する要因となっており，「スピード経営を実現出来ない」ことは迅速なサービスの提供が出来ない要因となる．同業他社より新サービスの提供が遅れることにより，企業収益のみならず企業のブランドイメージまでが大幅に損なわれてしまうこともある．ROI の低下だけではなく，企業の存続すら危ぶまれることになるのである．

2.1 レガシー・システムの変遷

現在まで，企業システムの中で IT が担う役割は大きく変わってきた．1990 年以前は，業務プロセスを支援するために IT が活用され，データベースが中心の時代であった．IT が担う役割は，業務の生産性向上の支援や品質向上であった．1990 年以降になると，業務プロセスを強化するために IT が活用されるようになった．IT が担う役割は，業務の合理化や他社との差別化となっていった．2000 年代に入ると，IT の担う役割としてビジネスを先導することが求められるようになり，柔軟性や迅速性を求められるようになってきた（図 1）．

2.2 近年のサーバシェア

近年，オープン系プラットフォームでのシステム構築が盛んに行われている．これは，ハードウェアやソフトウェアの保守・メンテナンスコストが比較的安価であるという要因も挙げられるが，最新の技術や業務パッケージが UNIX や Windows などのオープン系プラットフォームで提供されることが多いということが大きな要因の一つと考えられる．世界と日本のサーバ・プラットフォーム別のシェア(出荷金額)を見ると，図 2 のような状態である．日本では，まだまだメインフレームのシェアが世界に比べると高い．欧米諸国では 1990 年代後半以降，

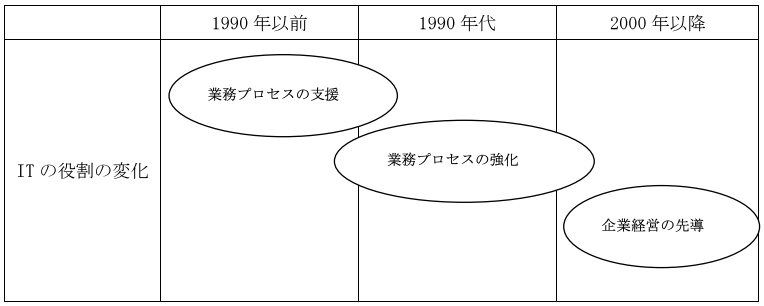

図 1　企業 IT システムの変遷

ERP[*1] やドットコム・ブームなどにより，多くのメインフレーム・システムがオープン・システムにシフトしていった．これは，オープン・システムを採用することによる IT コスト削減という観点からだけではなく，経営と IT の融合による新たなビジネスモデルの創造をオープン化と同時に進めていった結果であると考えられている．

日本市場でも中小型のメインフレームを中心にオープン・システムへの移行が実施されているが，欧米諸外国に比較するとまだまだメインフレームへの依存度が高い．この理由として，日本におけるメインフレーム・システムは，ユーザの基幹系システムを支えるためのミドルウェアが充実しており，簡単にはオープン化できないことが考えられる．ただし，ベンダ各社のメインフレームの出荷金額も毎年数%単位に減少しており，オープン化の波は確実に大きくなってきている．

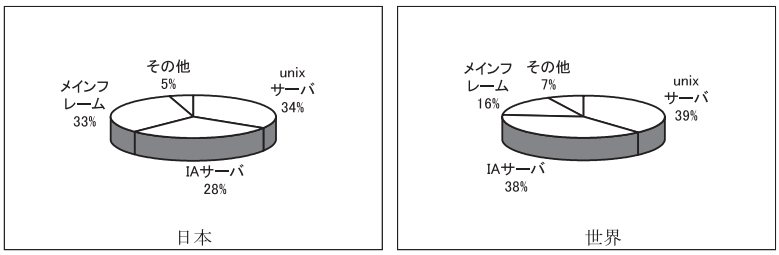

図 2　サーバシェア[1]

2.3　レガシー・システム刷新の必要性

レガシー・システムが抱えるシステムの肥大化，複雑化などの問題点や，昨今の企業 IT への要求事項は，現在のシステムで実現するには限界がある場合が多い．保守性，柔軟性が高く経営からの要求に迅速に応えるシステムを実現するためには，レガシー・システムを廃棄し，新たなシステムを構築することが求められる．

次章以降では，以下 2 点を前提に記述する．

・汎用機上で構築されたレガシー・システムを，オープン系プラットフォームへ移行する
・レガシー・システムのアプリケーションプログラムは COBOL 言語で記述されている

3.　レガシー・マイグレーションの手法

レガシー・システムを廃棄し，早期にシステムを再構築する必要性に関して今まで記述して

きた．しかし，システムの再構築には，多大な投資コストと期間が必要となる．さらに，オープン系プラットフォーム上でシステムを再構築するとなると，要員の技術力面でも不安要素は大きく移行リスクも高い．そこで，移行コストや期間を抑制し，現行資産や要員スキルを有効活用するレガシー・マイグレーション手法として，ストレートコンバージョンやリライトといった手法が登場してきた．オープン系プラットフォームへの移行手法は，移行コストやシステムの拡張性などの観点から大別すると，表1の4つに分類される．

表1　レガシー・マイグレーションの手法

移行手法	説明	特徴		
		移行コスト	拡張性	本番移行リスク
パッケージの適用	パッケージを導入して，パッケージの形態にビジネス・プロセスを適合させる方法	低	低	低
ストレートコンバージョン	ロジックは変更せずに，オープン系ソフトウェアの非互換部分のみを修正する方法	↑↓	↑↓	↑↓
リライト	ロジックは変更せずに，アプリケーション記述言語を変更する方法	↑↓	↑↓	↑↓
再構築	既存システムを廃棄し，システムを全面的に再構築する方法．既存システムからは，データのみを移行する．	高	高	高

3.1　ストレートコンバージョン手法

COBOLプログラムをそのままオープン環境のCOBOLへ移行する手法がストレートコンバージョンである．それぞれの移行対象資産における移行の特徴を記述すると図3の通りである．

ストレートコンバージョンでは，アプリケーション構造は変更せず，単純にオープン系のCOBOLにコンバージョンするのみとなる．そのため，移行コスト・期間は小さくなるという特徴がある．その反面，アプリケーションの拡張性に関しては，レガシー・システムとほとんど変わらない．

3.2　リライト手法

リライトとは，COBOLで記述されたアプリケーションプログラムを，C#やJAVAなどのオープン系で一般的に使用される言語へ変更することを言う．アプリケーションプログラム以外の資産の移行に関しては，ストレートコンバージョンと同様である．また，COBOL言語以外の各移行対象資産における移行の特徴も，ストレートコンバージョンと同様である．

またリライトは，移行後のアプリケーション構造により，3種類の移行方法に分けて考えることができる．

① リライト（パターン1）

COBOLなどの構造化プログラミングで作成されたアプリケーションを，そのままの構造で移行する方法

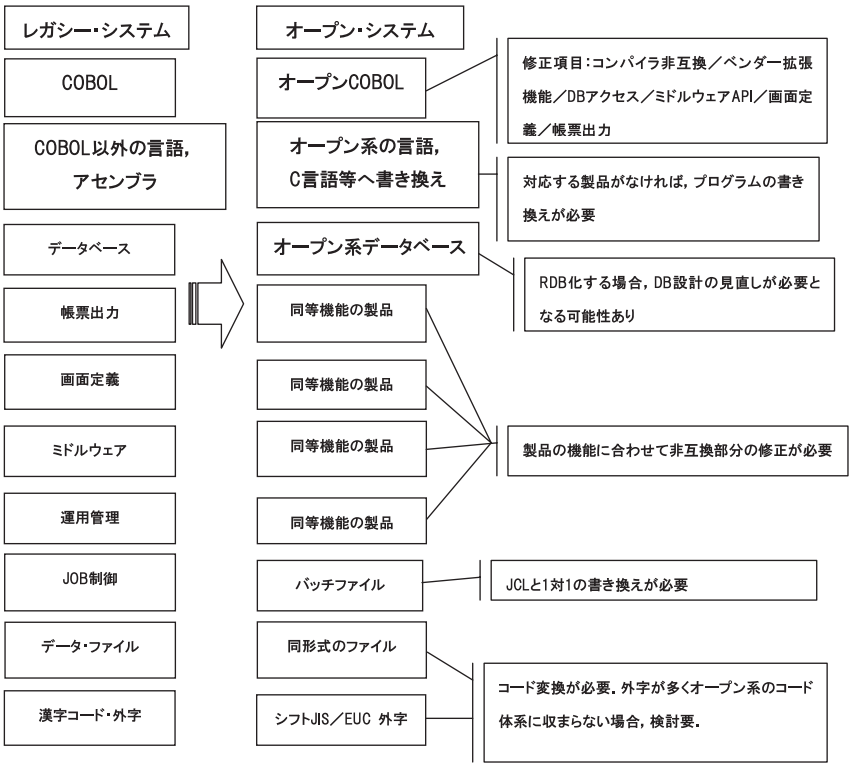

図3　ストレートコンバージョン

② リライト（パターン2）
オブジェクト指向型の構造へ移行する方法で，プレゼンテーション層・ビジネスロジック層・データアクセス層の3層構造に移行する方法

③ リライト（パターン3）
パターン2を更に進化させ，ビジネスロジック層を共有コンポーネントや拡張コンポーネントに分割し移行する方法

パターン1は，レガシー・システムから構造化プログラミングの構造を引き継いでいるため，COBOLプログラマであってもメンテナンスし易いが，C#やJAVAプログラマにとってはメンテナンスし難いアプリケーション構造になっている．ただし，移行コストは安価でありストレートコンバージョンと大差はない．

パターン2は，オブジェクト指向構造に変換されているため，C#やJAVAプログラマにとってメンテナンスし易い構造となっているが，移行コストはパターン1よりもやや高くなる．

パターン3は，ビジネスロジックを詳細に分析することにより類似性ロジックを共有コンポーネント化し，改修頻度の高いビジネスロジックを拡張コンポーネント化している．現行のビジネスロジックを踏襲するが，アプリケーション構造は再構成するという方法となる．

また，パターン2では，システムがスパゲッティ状態となっているプログラム構造は解消されないため，アプリケーションの拡張性が高くなる可能性は低い．パターン3では，現状のビジネスロジック分析～再設計を行い，サブシステム間で類似ロジックがある場合は共有コンポーネント化し，頻繁に変更が発生するロジックに関しては拡張コンポーネントとして別コンポーネントとするため，アプリケーションの拡張性が高いシステムの構築が可能となる．ただし，

パターン 1，パターン 2 に比較すると，ロジック分析と再設計を行うために移行コストが増加する．

表 2 にリライト手法の 3 パターンの特性を記述する．

表 2　リライト手法の 3 パターンの特性

移行手法	特性	特徴
リライト（パターン 1）	・アプリケーション構造の要件定義~物理設計工程不要	・C#や JAVA といった一般的なオープン系プラットフォームで使用される言語に変換されるが，アプリケーション構造は旧態依然のままである ・移行コスト小
	・アプリケーション構造を変更しないためシステムのスパゲッティ状態は解消しない	
	・ロジックが変わっていないため単体テストが不要	
リライト（パターン 2）	・オブジェクト指向型への変換	・オブジェクト指向型のアプリケーション構造体に変更することにより，保守要員調達が容易 ・リライト（パターン 1）に比べ移行コスト大
	・3 層構造に分割するが，システムのスパゲッティ状態は解消しない	
リライト（パターン 3）	・一部物理設計工程から実施	・リライトでも拡張性の高いシステム化 ・リライト（パターン 2）に比較し移行コスト大
	・アプリケーション構造を分析し，再構成する ・ビジネスロジックを分割し機能単位にコンポーネント化を実施	

3.3　再構築手法

システムの構築を要件定義から行い，既存システムを捨てて新たにシステムを構築していくことである．この手法では，移行コスト・移行期間・本番移行リスクにおいて，他の移行手法に比べると高くなることが特徴である．その反面，拡張性の高いシステムの構築が可能となり，システム構造を変化対応力に優れた構造にすることが出来る（図 4）．変化対応力に優れたシステム構造として，大きく以下の 2 点の構造にすることが重要である．

・業務プロセスを変えた時にシステム対応が迅速に可能であること

・業務プロセスを変更してもデータの整合性が保たれること

レガシー・システムでは，業務プロセスを変更すると複数のサブシステムに影響することが多い．また，同じようなデータが複数のサブシステムに分散しており，業務プロセスの変化に合わせてシステムを変更した際に，データの整合性がとれなくなってしまうケースがある．これらの問題点は，システム構造自体の変更だけではなく，業務プロセス改善も必要になってくる場合がある．アプリケーション構造を変更しないストレートコンバージョンやリライトでは，レガシー・システムが持っている本質的な問題点を解決することが出来ない．システムを再構築していくことにより，拡張性の高いシステムの構築が可能となる．また，新たな業務要件をマイグレーションと同時に取り込めるというメリットがある．

3.4　パッケージ適用

パッケージ適用は，再構築と同様に要件定義から実施し，システムを作り直すという考え方となる．この場合は，コストやリスクを抑えるためには，適用するパッケージに業務プロセスを合わせていくことが重要となる．まず，次期システムで想定する業務プロセスと，パッケージで提供されるプロセスの間の FIT&GAP を分析する．GAP のあるプロセスに関しては，極

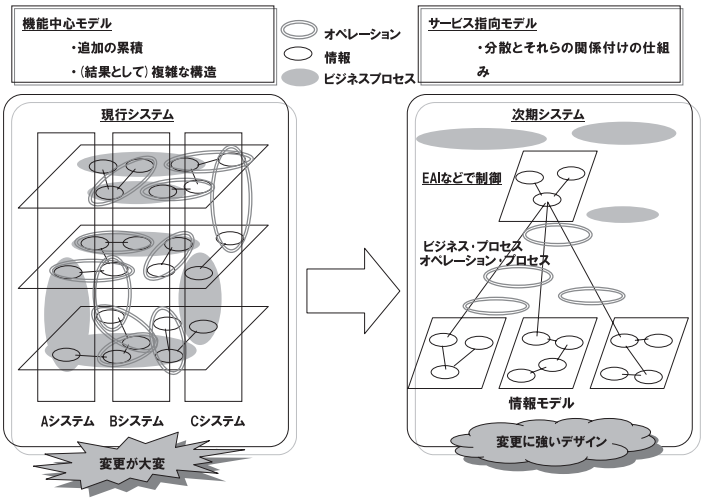

図 4　再構築

力パッケージ・プロセスを採用し，業務プロセスをパッケージ・プロセスに合わせることが望ましい. 業務プロセスをパッケージ・プロセスに合わせることが不可能なプロセスに関しては，パッケージ・プロセスを使用せず，そのプロセスのみ再構築・ストレートコンバージョン・リライトなど他の手法により移行することを考える．パッケージをカスタマイズすることも考えられるが，カスタマイズにより移行コスト・本番移行リスクならびに本番後の拡張性が低くなる可能性がある．パッケージ適用における失敗プロジェクトに見受けられる傾向としては，「FIT&GAP 分析が不十分でありパッケージのカスタマイズ要件が想定以上に多かった」というものが多い．このようにパッケージ適用では，移行作業に想定以上にコストを要し，更に実現される予定であった拡張性も現行システムに比べ低くなってしまう，ということも有り得るため，十分な FIT&GAP 分析および対応検討が必要である.

3.5　移行手法の組み合わせによるレガシー・マイグレーション

レガシー・マイグレーションでは，3.1 節から 3.4 節で説明してきた手法を単独で行う場合もあるが，いくつかの手法を組み合わせて使用する場合もある．例えば，以下のようなケースが考えられる.

① サブシステム単位に業務プロセスの変更要否を判断し，変更する必要があるサブシステムは再構築し，変更の必要のないサブシステムはストレートコンバージョンで移行する

② オンライン処理はリライトで移行し，バッチプログラムはストレートコンバージョンで移行する

どのようなケースであっても，現状分析フェーズにて次期システム像をしっかり描き，移行プランを検討することが重要である．次期システムに要求される業務機能・システムアーキテクチャだけではなく，システム保守要員の人材プランも見据えた形で，移行プランを検討することが望ましい.

3.6　レガシー・マイグレーションの理想形

移行コストや期間などに全く制約の無い場合に，最も理想的なレガシー・マイグレーション

は，再構築によるマイグレーションである．なぜなら，システムを要件定義から作り直すことにより，以下のような効果が得られるからである．

① システムのスパゲッティ状態が解消され，保守性の高いシステムとなる

② 最新のテクノロジや方法論を取り入れることが可能となり，拡張性の高いシステムとなる

保守性の高いシステムは投資するITコストの削減が可能となり，拡張性の高いシステムはスピード経営に対応することが可能となる．レガシー・システムが持っていた問題である“経営の足枷状態”が解消されるため，再構築によるレガシー・マイグレーションが理想形であると考える．

4. 現実的なレガシー・マイグレーションの考え方

4.1 日本ユニシスが考える段階的な移行

レガシー・マイグレーションの最善の方法はシステムを再構築することであるが，移行コスト・期間や移行本番時のリスクといった理由から「一足飛びに理想のマイグレーションを実現することが難しい」という場合がある．そこで，日本ユニシスでは「まずはストレートコンバージョンやリライト手法を使用したオープン化を行い，その後サブシステム毎に再構築を行っていく」という段階的な移行（図5）が現実解であると考え，推奨している．

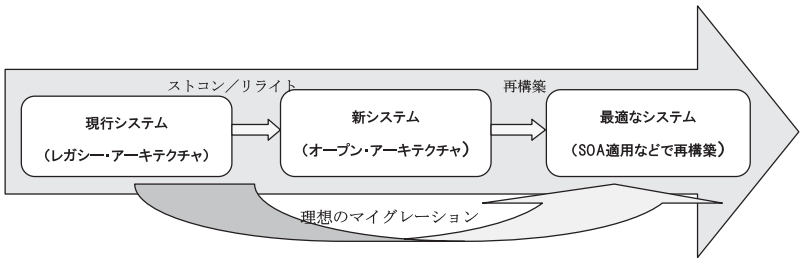

図5 段階的な移行

ストレートコンバージョンで移行後にサブシステム単位に再構築する際は，再構築時に要件定義から新たにシステムを作り直すこととなる．このため，再構築が完了するまでの投資コストを考えた場合，ストレートコンバージョンでのコスト分が過剰な投資となる場合がある．これは，ストレートコンバージョンでは移行後のアプリケーションが構造化プログラミングを踏襲しているため，オープン・システムからの移行であってもシステムは全く一からの作り直しとなるためである．また，リライト（パターン1）でもアプリケーション構造体はストレートコンバージョンと同様に構造化プログラミングを踏襲しているため，投資コストの考え方は同様であると言える．ただし，ストレートコンバージョンは比較的短期間で移行可能なため，ランニングコストという側面から考えると，ハードウェア・ソフトウェアなどの設備費用が低くなるオープン・システムの利点を早期に享受することが可能である．

リライト（パターン2）やリライト（パターン3）では，移行後のシステム構造がオブジェクト指向型となる．そのため，再構築前後でシステムアーキテクチャが同様となり，現行のビジネスロジックの有効活用を行うのであれば，再構築時の開発コストはストレートコンバージョンやリライト（パターン1）に比較すると低くなる．なぜなら，リライト（パターン2）で

は，3 層構造でのコンポーネント化が完了しており，考え方によってはビジネスロジック層の再構築のみでシステムの再構築が可能となる場合があるからである．リライト（パターン 3）の場合は，保守性や拡張性を意識し，現行ビジネスロジックを詳細分析した上でコンポーネント化を行っている．そのため，再構築時に新たな機能が必要な場合はコンポーネントの追加・改修を行うのみでよい場合があり，再構築時の開発コストを極小化できる．

日本ユニシスが推奨する段階的な移行の最適解は，リライト（パターン 2）または，リライト（パターン 3）でのオープン移行を行った上で，サブシステム単位に再構築を行い，システムの最適化を行っていくことである．

4.2　リライト（パターン 2）およびリライト（パターン 3）の特徴

リライト（パターン 2）では，アプリケーション構造をプレゼンテーション層・データアクセス層・ビジネスロジック層の 3 層に分解し，各層をコンポーネント化する．プログラム構造を 3 層に変更することにより，プログラムライブラリ数は増加することになるが，アプリケーション構造体は C＃や JAVA で新規構築した時と同様の構造となるため，オープン化の第一歩としては非常に有効な手法だと言える．

リライト（パターン 3）は，「再構築を検討しているがコストや作業期間に余裕がない」というユーザには是非お勧めしたい移行手法である．アプリケーションプログラム群の中で類似性の高いロジックを共通コンポーネントとし，さらに，変更が多いロジックを拡張コンポーネントとして別コンポーネント化する方法である．システム改修の際には，共通コンポーネントや拡張コンポーネントのみを改修すればよい場合が多くなり，拡張性の高いシステム構造となる．ビジネス・プロセスは変わらないが，拡張性という観点では限りなく再構築に近いシステムを構築することが可能となる．

リライト（パターン 2）およびリライト（パターン 3）における，移行前後のシステムアーキテクチャを図 6 および図 7 に示す．これらは Windows＋ .NET で構築した例である．

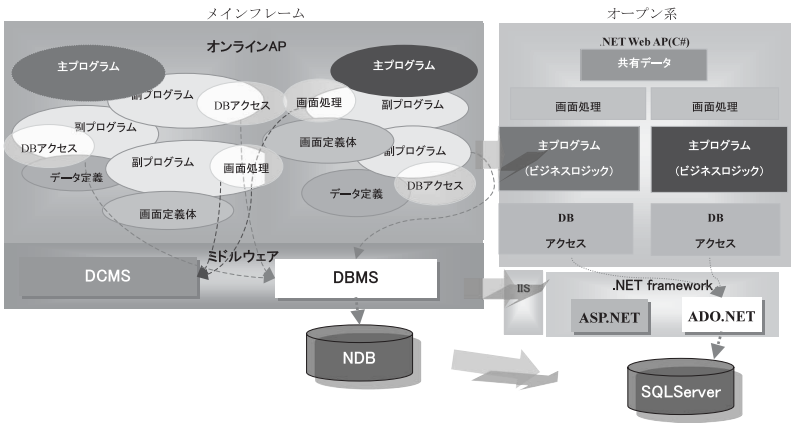

図 6　リライト（パターン 2）における移行前後のシステムアーキテクチャ

5.　移行前後のアーキテクチャとマイグレーション時の注意点

本章では，移行前後のシステムアーキテクチャの相違点やマイグレーション時の代表的な注意点について説明する．

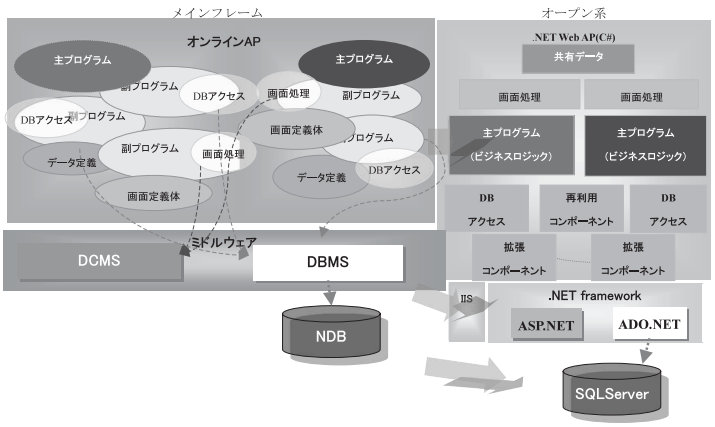

図 7　リライト（パターン 3）における移行前後のシステムアーキテクチャ

以降に記述する内容は，ストレートコンバージョンまたはリライトでの移行であり，Windows への移行パターンを例として記述している．また，端末関連制御には，ASP .NET[*2] を採用することを前提としている．

5.1　全体アーキテクチャ

移行前後での基本的なアーキテクチャで，大きく異なる点は，画面制御と DB アクセス制御である（図 8）．ホスト上の画面定義は ASP .NET 上の Form オブジェクトに相当し，端末を制御するテキスト情報と画面定義情報については ASP .NET（HTML）の Form 機能に置き換えるため，追加のプログラミングが必要となる．

データベース・マネジメント・システム（以下，DBMS）との連携は，構造的には既定のアプリケーション・プログラム・インタフェース（以下，API）を呼び出しているという点では同様であるが，API のギャップが大きく入出力処理全体の考え方が異なるため，プログラミングに与える影響範囲が大きい．

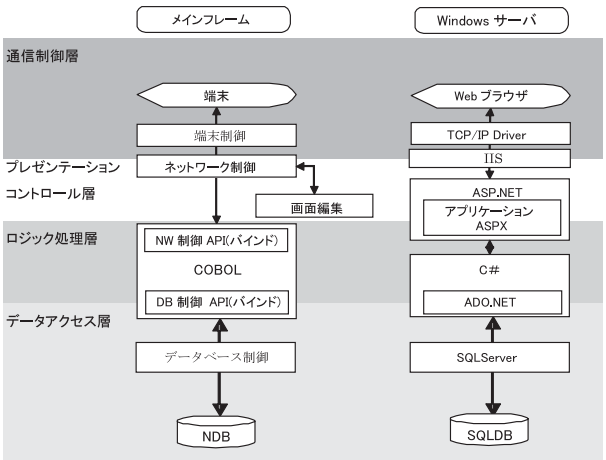

図 8　全体アーキテクチャ図

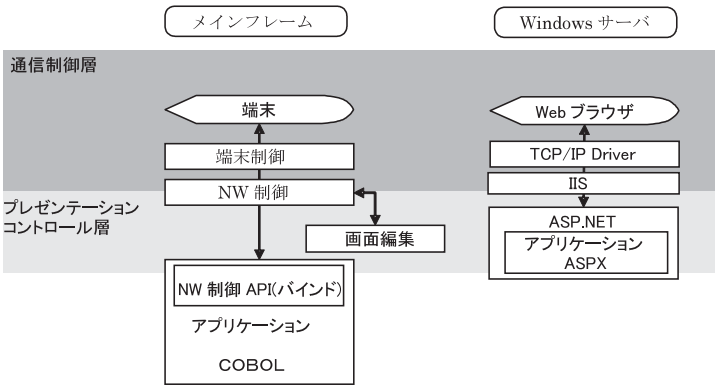

図 9　端末関連処理図

表 3　端末関連処理の相違点

端末処理	メインフレーム	Windowsサーバ
画面レイアウト生成	画面定義上に独自表現されたものを NW 制御がテキストとマージ	HTML(ASPX)にて座標・サイズ・フォント属性などで表現
画面入力項目の定義	画面定義上に独自表現したものを端末制御が端末で入力可能領域として画面出力	HTML(ASPX)にてタグ<input>表現し，ブラウザ機能が入力可能領域として画面出力
画面出力→クライアント側受信	AP で送信→NW 制御でレイアウト構成→端末が受信	ASPX 処理をぬけると自動送信
画面入力→サーバ側受信	端末の送信キーを押下→NW 制御で受信して入力項目編集→AP が受信 API で待ち受けて入手	送信ボタンクリックなど，画面出力時 ASPX 記述に依存→POST 処理によって入力項目のみを送信→ASPX 呼び出されたときにフィールドにセット
クライアント機能	特殊表現(Bright,Blink,Color)	Color,Font,Window 制御

5.2　端末関連

ホストでは画面定義ファイルで定義されたテキストを元に，端末描画仕様に応じたコマンド文字列を使用して画面処理を行う．

.NET 環境で画面処理に ASP .NET を使用する場合は，サーバ側の ASP .NET（HTML）で記述された言語を基にクライアントの Web ブラウザが画面処理を行う．

等価移行を前提とするが，端末とブラウザの基本仕様の相違が操作性に影響を与える場合があるため，画面・文字・配色などの移行先の機能定義については，ホスト仕様をベースにしつつユーザ要件による変更を可能にするべきである（図 9，表 3）．

5.3　データアクセス処理

現行システムのデータベースにネットワーク型データベース（NDB）[*3] を使用している場合は，データベースへの入出力に関するアーキテクチャが大きく異なる（図 10）．ストレートコンバージョンやリライト（パターン 1）での移行の場合は，NDB 構造をほぼ踏襲しリレーショナル型データベース（RDB）[*4] の再現が可能である．リライト（パターン 2）やリライト（パ

ターン 3）での移行の場合は，ホストデータベース仕様を継承的に .NET で採用することは移行後の保守性や拡張性の向上という本来の目的が達成できない可能性があるため，再設計が必要となる．

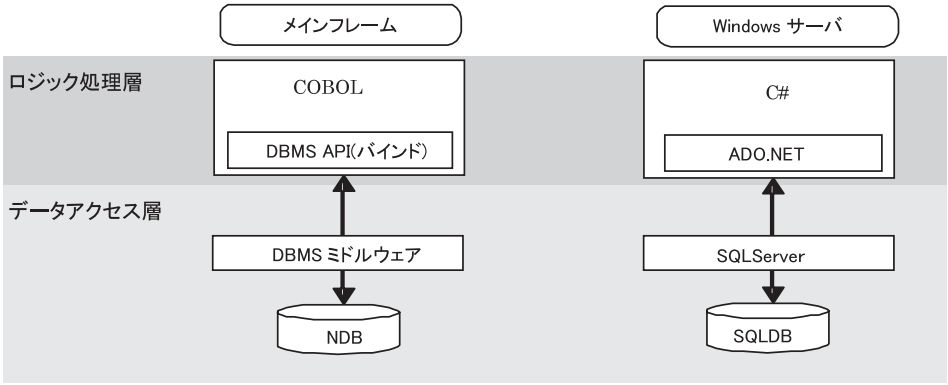

図 10　データアクセス図

5.4　ミドルウェア

一般的にメインフレームでは，ミドルウェアと呼ばれる，ベーシックソフトウェアを隠蔽するソウトウェアが使用されている．ミドルウェアの移行は，UNIX の場合では BEA　Systems, Inc. の「BEA　TUXEDO®」を，Windows で且つオープン COBOL を採用する場合は日本ユニシスの「MIDMOST®」を導入する，というような対応が考えられる．また，C# を採用する場合は「MIDMOST®　for .Net」を導入する，という選択肢もある．ただし，現行のミドルウェアの機能がオープン系のミドルウェアで実装されていない場合は，ユーザアプリケーションにて処理を実装することを検討する必要があるため，注意が必要である．

現行システムにてミドルウェアを使用している場合は，移行後に採用予定であるミドルウェアと現行ミドルウェアとの間の「FIT&GAP 分析」を十分に行い，ユーザアプリケーションでの対応も含めた検討を行うことが重要となる．

5.5　COBOL 言語

COBOL 言語は，ストレートコンバージョンの場合はオープン COBOL を，リライトの場合は C# または JAVA へ移行することが基本となる．ここでは，ストレートコンバージョンでオープン COBOL に移行する場合の代表的な注意点について説明する．

① 登録集

1 ファイル内に複数の登録集が存在する「マルチエントリ形式」は，オープン COBOL では未対応のため，1 ファイル 1 登録集に修正する必要がある．

② COBOL 規格

レガシー・システムでは COBOL は規格 74 が使用されている場合があるが，オープン COBOL は規格 85 または規格 2002 のため，規格の違いでエラーが発生する．また，OS 2200[*5] からの移行の場合は，Unisys 2200/IX シリーズが 9 ビットマシンであるため，データ定義などを 9 ビットから 8 ビットへ変換することが必要になる．

③ ビット変数（PIC 1）

データ定義のビット領域（PIC　1）を移行する場合は，移行後のデータ型を決めるために使用方法(ビットマップ，英数字，漢字，バイナリ値)の確認，解析を行う必要がある．

④　CONSOL 出力，外部入力パラメタ

CONSOL 出力，外部入力パラメタは，一般的なオープン系 COBOL では対応していない．COBOL プログラムだけでなく関連する JCL も解析し，対応方法を明確化する必要がある．

5.6　アセンブラ言語

アセンブラ言語の移行先言語としては，OS などベーシックソフトウェアに依存している処理の場合はシステム変数制御が可能な C 言語や C# 言語への移行を推奨している．また，アプリケーション処理の場合は，保守性向上の観点から COBOL 言語の移行先言語と同一の言語への移行を推奨している．

5.7　JCL

JCL を移行する際には，以下のような点に留意する必要がある．

①　極力現在と同様な処理を実装可能な移行方法を選択することが必要である．JCL コマンドと同種のコマンドが多数用意されている「シェル」コマンド(UNIX の場合)，「MS-DOS」コマンド（Windows）への移行が現実的である．

②　単純に「シェル」コマンドや「MS-DOS」コマンドで実現不可能な複雑なロジックを JCL で実装している場合は，スクリプト（VB スクリプト，JP 1 スクリプト等）で実装する．

③　JCL 実行中にエラーが発生した場合，メインフレームではエラー発生時点で処理が終了（アボート終了）する．しかし，「シェル」や「MS-DOS」のコマンドでエラーが発生した場合には処理を継続するため，コマンド実行を行うたびに正常/異常の判定を行う処理を追加する必要がある．UNIX「シェル」コマンドへ移行した場合の例を記述する．

```
cp      $path/qual_fileA $path/qual_fileB
if          $?  !=  0
            {エラー処理}
        exit 1
endif
```

5.8　その他

5.8.1　帳票処理

現行の帳票要件を明確にし，オープン系で提供されている帳票ソフトウェアと機能比較を行う．その上で現行の帳票要件にマッチした帳票ソフトウェアを選択する必要がある．一般的には，ウイングアークテクノロジーズ株式会社の Super Visual Formade（以下，SVF）や株式

会社日立製作所の EUR[*6] を使用することが多い．SVF を使用する場合，現行帳票をスキャナで取り込んだ後，SVF の設計部をオーバレイを使用して移行し，帳票定義インタフェースを使用してフォーム調整を行う．

また，マイグレーションを機に帳票要件の再検討を視野に入れることが望ましい．帳票印書や帳票発送処理に要するコストはシステム部門でもかなりの比率を占めていることが多い．電子帳票化する等の印書帳票削減を検討することにより，保守コスト削減を実現できる可能性がある．例えば，帳票出力している情報を PDF ファイル等でサーバに格納し，都度クライアント端末からアクセスするような仕組みや端末配信するような仕組みにすることが考えられる．

5.8.2 運用および要員教育

メインフレームからオープン系への移行では，プラットフォームの変更に伴いシステム運用が大きく変更となる．業務運用は，ストレートコンバージョンやリライトではバッチジョブフローを変更せずに移行することも可能であるが，通常は運用管理ミドルウェアが変更となるため各種パラメタ類が変更となる．また，障害監視，バックアップ，セキュリティ管理等のシステム運用に関しては，一般的には全面的に再構築となる．ストレートコンバージョンやリライト（パターン 1）においては，アプリケーション構造に変更がないこともあり運用が軽視されがちである．どのような移行手法を採用したとしても，運用は最も重要であり，移行プロジェクト開始当初より検討を開始する必要がある．

また，プラットフォーム変更やアプリケーション言語変更に伴う要員教育が必要となる．ベンダや SIer の教育メニュを利用する，または使用するソフトウェアの製造元ソフトウェア会社の教育サービスを利用する，等の方法により要員教育を進めていく必要がある．

6. おわりに

欧米諸国の IT 資産におけるレガシー・システムの占有率は約 16% であり，日本の約 38% と比較するとレガシー・システムからの脱却が早い．これは，欧米諸国では，IT コスト削減に向けたオープン化への取組みが 1980 年代後半あたりから行われてきたという一面もあるが，経営と IT の融合による新たなビジネスモデルの創造がオープン化に先行して取組まれてきた，とも言えるであろう．また，欧米諸国での IT 資産への投資額は，日本の約 3 倍と言われている．このことからも，欧米諸国は，単にコスト削減という観点からだけではなく，ビジネス環境の変化に素早く対応し経営を IT で支えるシステムを構築するという観点から，オープン化への積極的な IT 投資を行ったと考えられる．

ビジネス環境の変化に素早く対応し経営を IT で支えるシステム，すなわち拡張性や柔軟性の高いシステムを，コストや移行リスクを低く抑え短期間で実現することが重要となる．そこで，日本ユニシスでは 4.1 節にあげたリライト（パターン 2）またはリライト（パターン 3）からの段階的なマイグレーション手法を推奨している．レガシー・マイグレーションを，システム全体最適化に向けたシステム構築の第一歩とするためにも，是非検討していただきたい手法である．

＊1　統合基幹業務パッケージ
＊2　マイクロソフトが提供している WEB サービス開発のための言語
＊3　メインフレームで一般的に使われていたデータベース構造体．データの親子関係を階層構造で表現するデータベース．
＊4　ORACLE や SQLServer などオープン系で一般的に使用されるデータベース構造体．データベースレコードを関連付けつつ処理を行うデータベース．データベース同士を結合したり，データベースから特定のフィールドを取り出して別のデータベースを作成することが可能．
＊5　Unisys 2200 系メインフレームで稼働するオペレーティング・システム
＊6　End-User Reporting の略　帳票システム構築支援ソフトウェア

参考文献　［1］　日経 BP 社,「レガシーマイグレーションへの挑戦」，2003 年

執筆者紹介　中　村　修　二（Shuuji Nakamura）
1991 年日本ユニシス(株)入社．金融勘定系パッケージ TRITON システム開発に従事した後，地銀・系統系への TRITON パッケージ適用業務および金融系フィールド SE を経て，レガシー・マイグレーションビジネスの企画・提案支援・適用業務に従事．現在，日本ユニシス・ソリューション(株)共通技術開発本部マイグレーションサービス部マイグレーション適用技術室室長．