6

エンタープライズ・システムにおけるソフトウェアプロダクトラインの適用

石田 裕三　野村総合研究所

SPLC などの公開情報では，組込み系の事例が圧倒的に多い．市場の大きさや従事する企業の数が決して小さくないエンタープライズ系において，なぜ事例が少ないのか．一般論として両者の違いを整理し，その違いから生じる SPL 適用上での課題を明らかにする．

その上で，その課題への解決策として過去 8 年にわたり NRI が蓄積／洗練を続けるコア資産，SPL プラットフォームのアーキテクチャ的特長を解説し，それを用いた 7&i グループシステムの再構築の事例を述べる．

組込み系との違い

開発規模，人材，そして企業文化など，再利用を現実に機能させるための条件は多く，また障壁は高い．再利用技術の集大成が SPL（ソフトウェアプロダクトライン）とするならば，再利用へのハードルの違いが，SPL との親和性の違いとも考えられる．ここでは，エンジニアリングの観点から組込み系とエンタープライズ系の違いを整理する．

(1) プロダクト——シリーズ製品とワンオフ開発

組込み系では，消費者と生産者の双方が共通認識を持てる明確な相似性 (Similarity) が一連のプロダクトに対して存在する（シリーズ製品）．一方のエンタープライズ系では，プロジェクトごとの要件にあわせて個別に開発する（ワンオフ開発）．

(2) ライフサイクル——モデルチェンジと改修

「仕様変更」は，シリーズ製品であれば次期モデルあるいはモデルチェンジを意味する一方，ワンオフ開発では既存システムへの改修にあたる．1 プロダクトのライフサイクルの定義が異なるといえる．品質属性的には，生産性と保守性の関心事の違いといえる．

(3) 要件定義——製造業とサービス業

事業の生業にも違いがある．「製品」は不特定多数の利用者に提供するが，「受注製品」は 1 社向けのサービス提供という付加価値である．プロダクト要件は市場の動向やユーザの意向に左右されるが，最終決定権がプロダクトの生産者か，利用者かの違いがある．

(4) 関心事——ハードウェアとデータ

ソフトウェアを実装する際のハードウェアを意識する度合いの違いがある．リソース面の制約が相対的に緩いエンタープライズ系は，ハードウェアそのものではなくデータへの意識が強い．主要な関心事が異なる．

(5) フィーチャ——具象と抽象

目に見えるハードウェアの存在やリソース面の制約は，特徴的な機能（フィーチャ）を切り出す上で，意思決定の枠組みを与えてくれる．ハードウェアという具体的なモノから離れた抽象世界では，「受注製品」に対する立場により，フィーチャの粒度や捉え方が異なる．

エンタープライズ系 SPL の適用障壁

上述の違いは，そのままエンタープライズ系への SPL 適用の難しさを表している．さらに，90 年代初頭から加速した“オープンシステム化”により，業界構造が，ハードウェアからアプリケーションまで 1 社で提供する垂直統合型から特定領域に特化した水平統合型にシフトした．その変化は，オープン化のメリットと引換えに，ドメイン最適化を困難にしている側面もある．

◉移り変わる実装技術と非機能要求の高度化

プロダクトの“モデルチェンジ”ではなく“改修”により，10 年以上のライフサイクルを持つことも珍しくないエンタープライズ系において，実装技術の寿命はシステムの寿命に比べて相対的に短くなってきている．

実装技術の進化は，それを使いこなす人間の技の習得を伴って価値が生まれるが，顧客の機能要件を実装に落

とすために重要な事項は，実装技術に直結するわけではない．また，操作性やリアルタイム性，24 時間稼働など非機能要求の高度化は，システムを完成する上でのハードルを高め，製品固有技術への依存度が高まるのである．

すなわち，変移する実装技術と非機能要求の高度化は，長期にわたり保守性を維持し，プロダクトをまたがり再利用する上での障壁となっている．

◉RDBMS への依存度

エンタープライズ系の主要な関心事であるデータ管理において，Relational Database Management System（RDBMS）は中心的な役割を担っている．また，標準化された SQL 言語の手軽さもあり，機能要求の多くは SQL で表現される場合も多い．

しかしながら，増え続けるデータ量や同時トランザクション数の増加，機能要求の変化と高度化により，RDBMS が処理のボトルネックになることは避けられない．レプリケーションやクラスタリングにより RDBMS への負荷を水平分散させると，データ鮮度と同期のオーバヘッドという課題が生じ，解決には多額の追加投資が必要となる．

また，SQL というデータ操作言語には，特徴的な機能（フィーチャ）の組合せで，より複雑な機能を構成するという「分割統治」の機構が備わっていない．

O/R マッピング・ツールで SQL をカプセル化し，RDBMS をオブジェクト指向的に処理する試みがされているが，厳しい性能要件やリソースの制約から適用範囲が限定され，複雑な機能ほど SQL 文で表現することになる．つまり，機能（フィーチャ）とデータが密結合になり，SQL 主体の開発は両者の独立した進化および最適化の妨げとなっている．

◉個別案件主体とレガシーシステムの存在

ワンオフ開発のように要員がプロジェクトごとにアサインされ，新規構築時と保守時で入れ替わることが多い環境下では，短期的なコスト最適化を求めやすい．そのため，将来の保守性は損なわれ，再構築時の移植には高いリスクが生じる．つまり，現行保障という制約の中で新たなプロダクトを開発しなければならない．そこでは「製品」の買い替えや，サポートの打ち切りといった不連続点を設ける手段は取れず，段階的な“移行”か，現行仕様の“凍結”の選択を迫られる．

このようにプロダクトのライフサイクルが曖昧であり，さらに発注者側の予算によりコスト面で大きな制約を受ける「受託開発」の繰り返しでは，生産者側が長期的な視点からアーキテクチャへ投資するのは難しい．長期にわたり生業を続ける中で根付いた組織文化を変えるには，長い時間と大きな外的環境変化が必要になる．

NRI のコア資産：SPL プラットフォーム

前章で述べた課題を認識した上で，基幹業務システムの受託開発を生業とする筆者らの組織（NRI：野村総合研究所）では，2001 年から SPL に取り組み，まずドメイン最適なアーキテクチャを構築／整備した．

そのアーキテクチャは，既存の基盤製品や部品の積み上げで構築するのではなく，アプリケーション開発保守効率を高めるための視点から，プラットフォームとアプリケーションの境界線を規定している．

そうして整備したプラットフォームは，小売業を始めさまざまな事業ドメインをまたがり共通で利用できるコア資産となり，各事業ドメインで SPL を実践する上での土台，「SPL プラットフォーム」として日々進化している．

◉SPL プラットフォームへの要件

大小さまざまな規模のシステムをまたがって再利用可能な部品を用意しようとすると，扱うデータ量や同時アクセス数など性能やスケーラビリティに関する非機能要求の違いが再利用を困難にしている．これは，機能と非機能の要求が混在して実装されるためである．

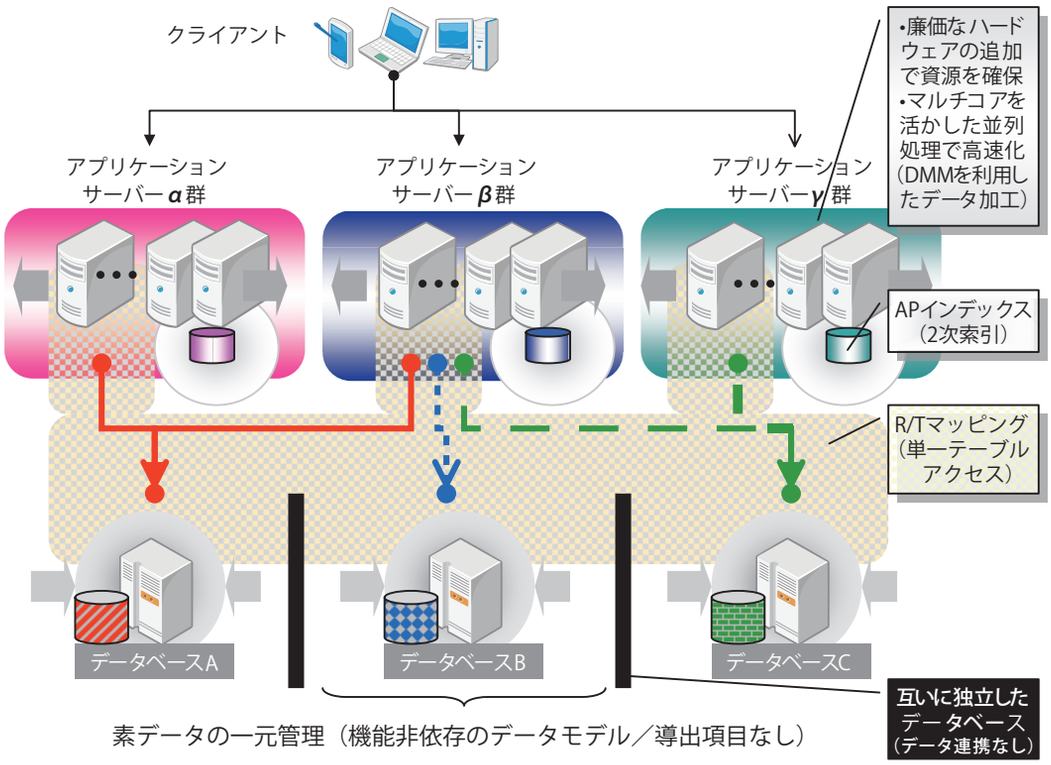
非機能要求の違いをハードウェアの構成の違いで表し，要求の変化にあわせて必要なハードウェア増強が適宜行え，高い投資対効果の水準を維持することが SPL プラットフォームのアーキテクチャ的要件となる．

また，システムの寿命に対して，実装技術や基盤製品の寿命は短い．アプリケーションがオープンソースも含め特定のプロダクトに依存すると，長期的な再利用は難しくなる．そこで変移するプロダクトの変化を吸収し，アプリケーションの開発生産性および保守性に寄与する機能を提供することが SPL プラットフォームの責務となる．

◉スケーラビリティの実現

RDBMS は，データの整合性管理という責務から高い信頼性が求められ高価なハードウェアおよび基盤ソフトウェアを用いる必要がある．そこでのボトルネックを回避するために，我々はあえて RDBMS の責務を最低限にした．そして廉価なアプリケーション・サーバの追加でメモリ空間と CPU 資源を確保し，マルチコアを活かす並列処理で応答性能／スケーラビリティを担保している．NRI は，**図 -1** に示す負荷処理配置を特定プロダクトに依存せずに実現し，コア資産化している．

再利用性を追求する際に物理およびコスト面での制約



※1. R/T(Record/Table)マッピング：単一テーブル単位の入出力操作. RDBMSでは一切のデータ加工をしない.
※2. APインデックス：検索条件をユニーク・キーに変換するための情報. RDBMSの2次索引に相当.
※3. DMM(Data Model Management)：データモデル（静的関連）からコードを自動生成する開発基盤と，オブジェクト・グラフ（動的関連）を管理する実行基盤からなる. SQLのDML(Data Manipulation Language)に相当.

図 -1　SPL プラットフォームの特徴と負荷分散戦略

を切り離して考えることができれば，変化し難いものと変化しやすいものを分離するなど可変性管理に最適な論理構造の普遍性を高めることが可能になる．

アプリケーションの可変性管理

生産者側に要件の決定権がなく，将来の変化を予測できない環境下では，変化を予測するより，変化に対応する／受け入れるといった戦略が必要と考えられる．そこで我々は，できるだけ重複を排除して小さく作り，異なる関心事を分離し，できるだけ変化に俊敏に対応する備えを行った．「重複の排除」と「関心事の分離」の 2 つを追及したアプリケーション・アーキテクチャの論理構造を図 -2 に示す．通常の MVC（Model-View-Controller）パターンとは異なり，厳密な 5 層階層構造を取る．この構造は，「View は Model のバリエーション」という考えに基づいており，View（画面）単位の開発で生じる冗長な実装を排除する狙いがある．さらに，各層の責務を明確にし，依存関係を一方向にすることで，変更時の影響範囲の特定を容易にし，修正個所の極小化を目指した構造である．

◉関心事の多次元分離

オブジェクト指向ではデータを中心に振舞い（ロジック）を持たせるデータ中心のモジュール構成になる．しかしながら，データとロジックでは異なる関心事が存在するため，データとロジックを不可分とする“古典的”オブジェクト指向では，それぞれの関心事で最適化を追及できない問題を Ossher 氏らは指摘している [1]．

SPL プラットフォーム上のアプリケーション開発ではデータとロジックの次元に加え，階層（レイヤ）等の関心事を用いて多次元でモジュール分割し，重複の排除や処理効率の最適化を追及する．以下に，我々のドメインでは関心事の多次元分離が不可欠であることを述べる．

クラスの価値と弊害

将来の変化を正しく予測することはできない．そのため，長期にわたり普遍的なクラスとは，外部の変化要因に影響を受けず，静的な要素を整理するための分類手段（Classification）に限定される．すなわち，最適なクラス粒度や分割基準が変化する中でクラスを固定することは，長期的には弊害となる可能性が高いと考えられる．このような状況下で再利用性を高めるには，再利用の粒度を

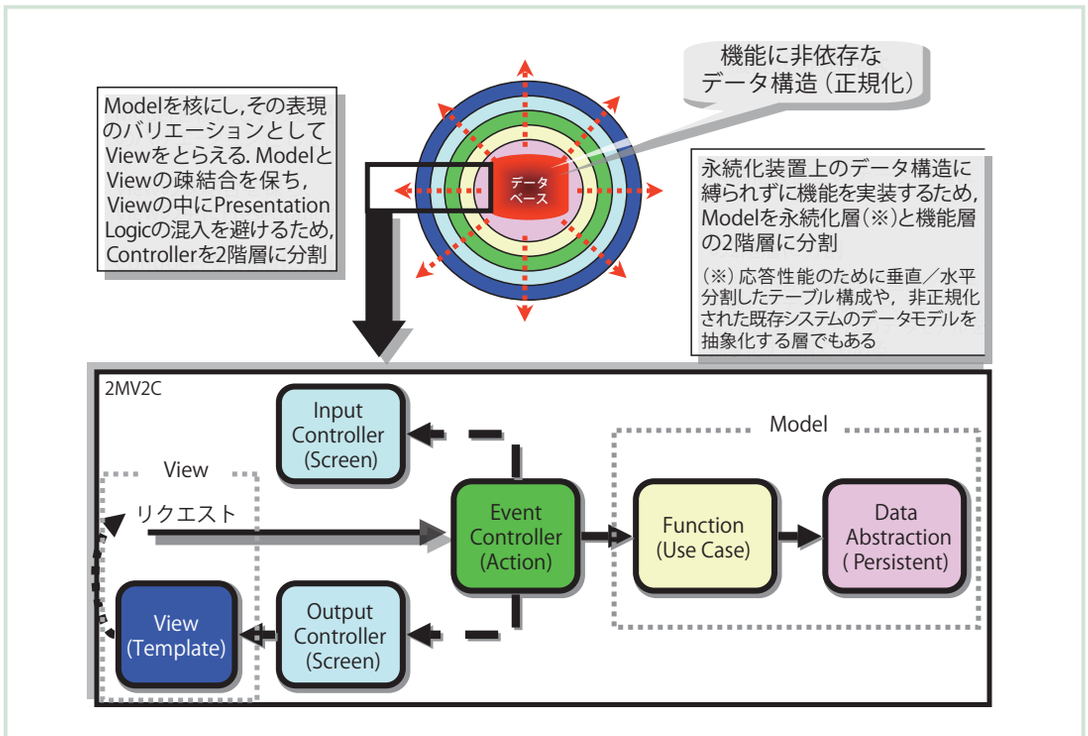


図 -2　アプリケーション・アーキテクチャ（5層 MVC）

小さくするのが 1 つの方向性となる．すなわち，メソッド（関数）を束ねるクラスの拘束を必要最小限にし，関数の合成で複雑さを表現する方向性である．

データの正規化と処理の正規化

データ整理術の 1 つに正規化がある．データの重複や冗長さを避ける上で有効であり，特定の機能に依存しないデータモデルを構築することが原則となる．なぜなら，機能とデータモデルが密な関係では機能要求の変化がデータモデルの変更を誘発し，そのモデルに基づく処理の修正が広範囲に及びやすいからである．すなわち，重複を排除し再利用性を高める処理の正規化と，特定の機能に依存しないデータモデルの正規化という 2 つの異なる関心事を分離することが可変性管理の土台となる．

情報の隠蔽とカプセル化

特徴的な機能要求の単位であるフィーチャを具現化するには，処理ロジックとそれが扱うデータ構造の規定が必要である．フィーチャの組合せでシステムを作るには，個々のフィーチャの独立性を高め，変更時の影響の極小化が前提条件となる．そのためには，データ構造の変化がロジックに影響を与えることがないようにデータを抽象化し，一方向の参照関係を持つことが鍵である．

データの抽象化とは，**図 -3** に示す永続化にかかわる関心事の抽象化と（情報の隠蔽），データ構造そのものの抽象化である（カプセル化）．このような普遍的クラスの分割ルールを規定することが分割統治の要である．

通常は 1 つのフィーチャは複数のテーブルに関連を持ち，また 1 つのテーブルは複数のフィーチャから利用される．つまり，フィーチャとテーブルは“メッシュ状”の関係を持つ．

この複雑に絡み合った依存関係を整理するには，各層を特定のルールに従い，層内を垂直方向に複数に分割する「スコープ」の適用が有効である．規定された範囲内で変化や実装の詳細を閉じ込めることができれば，システム全体の依存関係の複雑さを軽減できるからである．

スコープの適用

スコープ分割の基準は層ごとに異なる．機能層はユースケースの粒度に従い，永続化層は独立して存在し得る核となるエンティティ（コア・エンティティ）に従う．つまり，永続化層はコア・エンティティを中心に関連の強いテーブルをパッケージングする．各スコープ内で管理されるテーブルの数は異なるがメタ構造は共通であり，ロジック視点では**図 -4** のようなモジュール分割になる．

図 -3 の 2 次元に，「スコープ」という新たな次元を加えた 3 次元で Model を表現したのが**図 -5(a)** である．永続化層の安定度を高めるため，互いに排他なパッケージの集合とする．すなわち，同一層内のいかなるパッケージとも依存関係を持たず，独立性を保つようにすることを示したのが**図 -5(b)** である．一方，機能層は層内の依存関係を持つ．機能間に生じる冗長性を排除し，永続化層が機能要求に左右されないようにするためである．よって，永続化層のパッケージをまたがるデータ連携については機能層で行うことを図示したのが，**図 -5(c)** である．

この疎結合化は，レガシーシステムの非正規化されたデータ構造を永続化層の特定のパッケージに閉じ込める

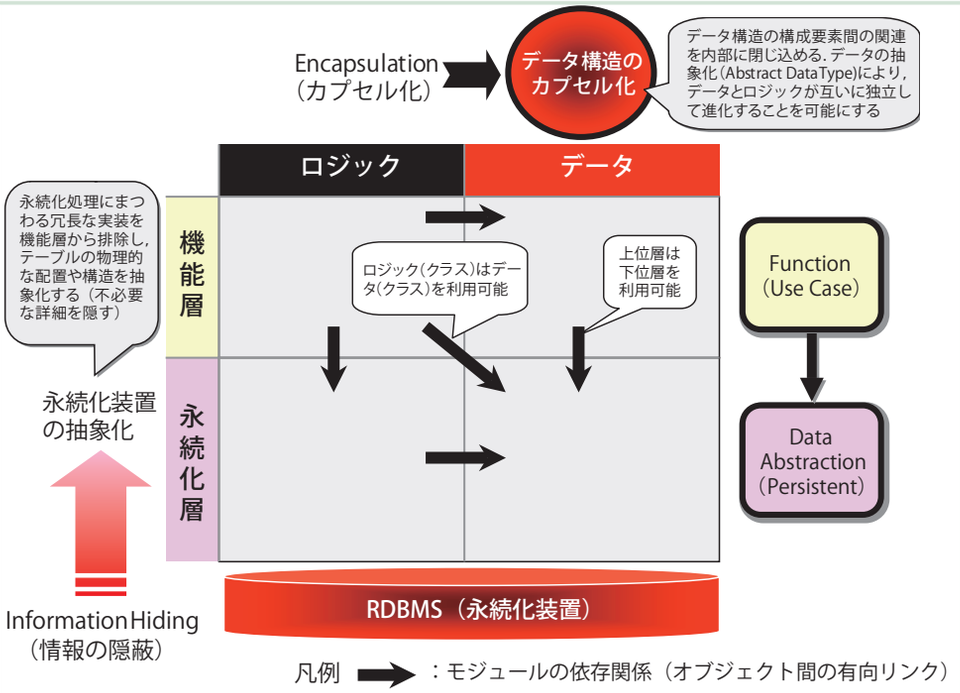


図 -3　データ抽象化における情報の隠蔽とカプセル化

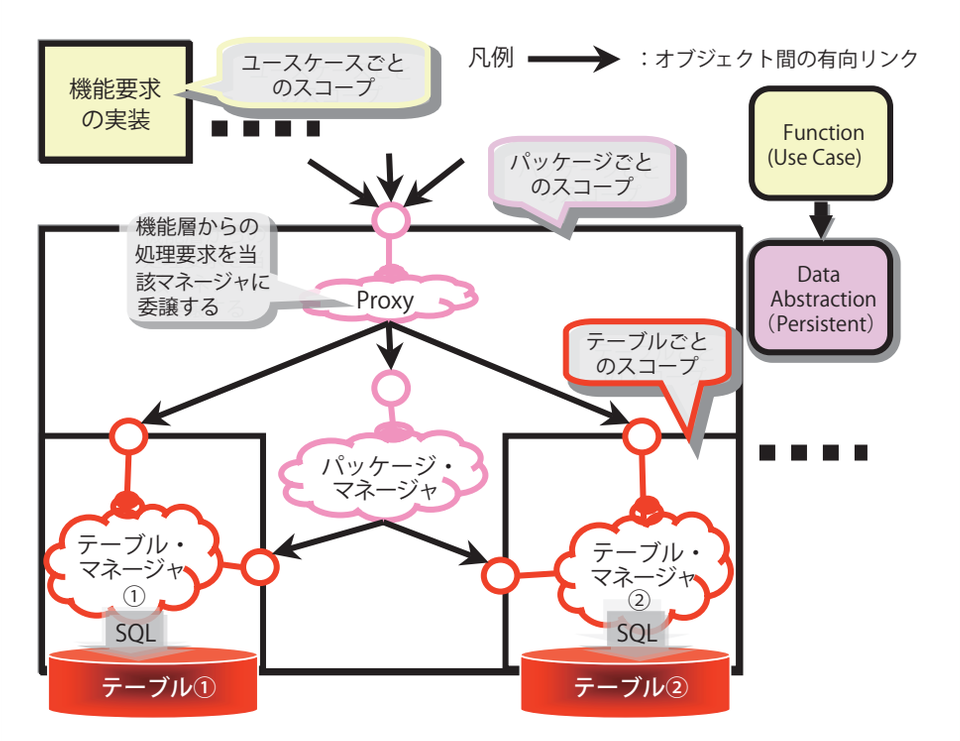


図 -4　永続化層ロジックのメタ構造

際にも有効である．なぜなら新たな機能はレガシーなデータ構造と切り離して開発することができるからである．

図 -4 で示した構造を実装する段階では，定型処理は自動生成し，個別の要件は手作業で実装している．つまりテーブルのスコープ，複数のテーブルを束ねるパッケージのスコープ，それぞれにおいて「自動」と「手動」というコードレベルの関心事の違いがあり，カラムやテーブルの増減に対応するためには，コードの再自動生成が可能な構造とする必要がある．そのために，自動生成したコードと手作業で実装したコードをクラス（ファイル）レベルで分離し，依存関係を表したのが**図 -5(d)** である．

◉可変性管理手法の使い分け

関心事の多次元分離によりモジュールをスライスして

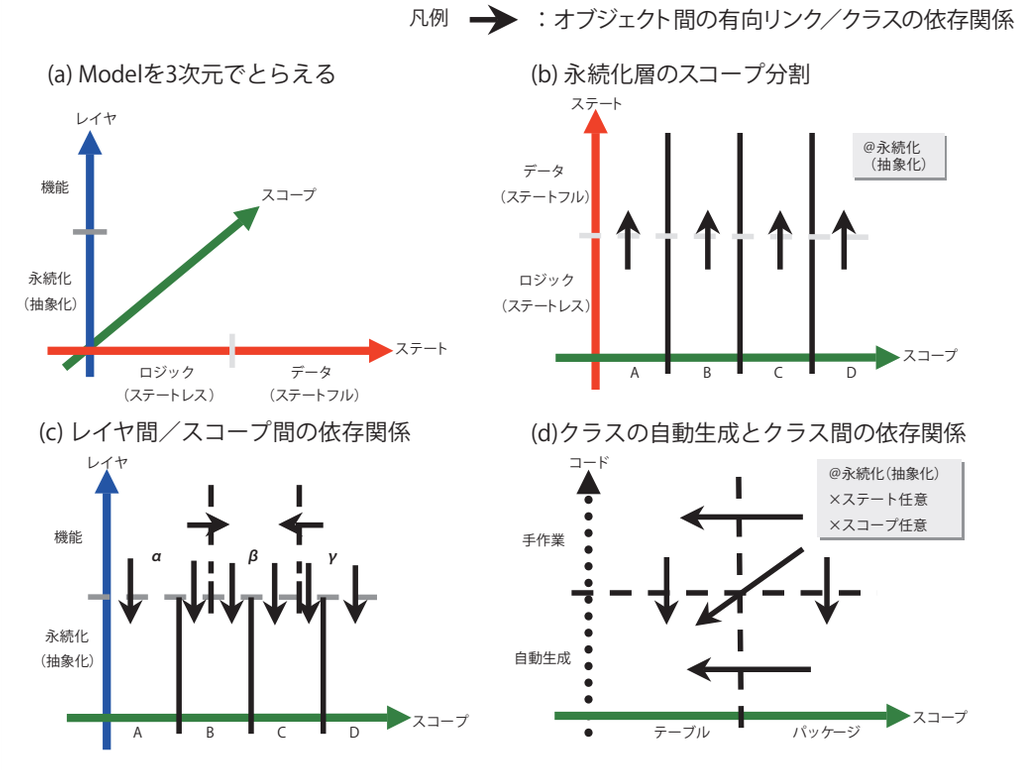


図 -5　関心事の多次元分離

いく規律を規定することで，開発者からクラス設計の責務を排除できる．つまり，適用する設計パターンは統一され，モジュール粒度の設計と組合せに集中できる．

一方，外的要因に左右され，変移する機能要求を前提におくと，事前に用意されたフィーチャの組合せだけでシステムを構築することはできない．そこで，柔軟性を確保するために，我々は複数の可変性管理手法を使い分けて，アプリケーションの可変性を管理している．

図 -2 で示したようにデータを中心に正規化を行い，永続化層は機能要求に左右されない相対的に安定した層を作ることは可能だが，外部システムとのデータ連携や機能要求の変移またはバリエーションにより，根幹を成すデータ構造そのものを固定することはできない．

Generative Programming

テーブル構造の可変性を管理するには，Generative Programming（GP）[2] が有効である．前述の DMM の開発基盤は GP の概念を使い，テーブル定義とテーブルのパッケージングの静的な定義から，永続化層全体を自動生成している．そのためには，汎用的な処理やクラスの関連をテンプレートとして管理する必要がある．

このテンプレートに記述されたメタなソースコードが共通性であり，バリエーションは XML など静的なインプットで管理する．この GP で管理する可変性は，図 -5(d) では，「自動生成」の行に当たる部分である．

Late Binding

「手作業」の行は，データアクセスの最適化ロジックを記述するために必要である．なぜなら，テーブルの連結順序やインデックス適用の最適化は，データ量や連結対象などによって異なり，テーブルの参照関係だけでは規定できないためである．このテーブル間の動的な関連を実行時に管理するのが DMM の実行基盤である．

手作業で実装する際の表現手段の自由度を高めると，開発者間でバラつきが大きくなり，可変性管理が困難になる．そこで，開発者から不必要な自由度を奪い，必要最小限の自由度を与えるライブラリの整備が必要になる．つまり，DMM で共通性を管理し，個別の要件にあわせて記述するコードで可変性を表現するのである．

その両者をランタイム時に遅延結合（Late Binding）することで，「手作業」の行が成り立っている．配列操作に必要な「反復」処理（for 文）や，キー項目の値の判定に必要な「分岐」（if 文）は，共通性として切り出せる．同じ配列の組合せでも機能によって最適な順番は異なるため「順次」だけは共通性として管理しないことが望ましい．そこで，「順次」，「反復」，「分岐」が絡み合う一連の処理を制御の反転（Inversion of Control）[3] によって，「反復」，「分岐」を共通性として切り出す高階関数的アプローチ／コールバック・ファンクションを有効に使ったのが DMM の実行基盤（フレームワーク）である（**図 -6** 参照）．

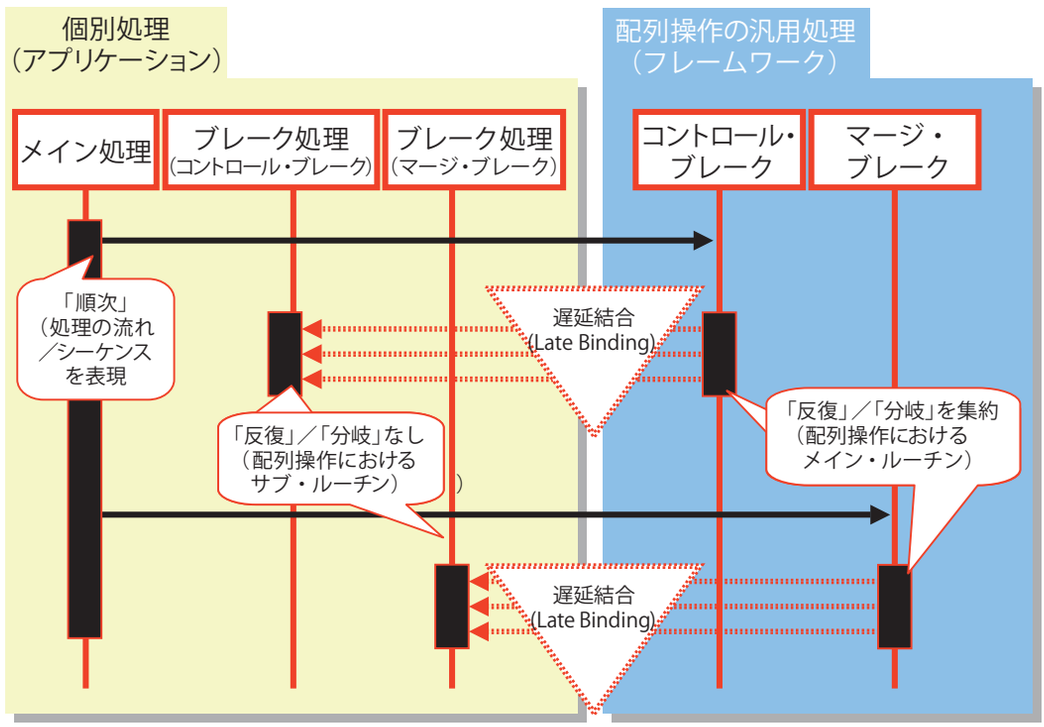


図 -6　制御の反転～「分岐」と「反復」処理のコア資産化

一方で，InnerJoin や LeftOuterJoin などの定型的な表連結処理や Sum や Average など簡易な集合演算であれば，個別に実装するコードを共通性として管理し，関数の引数に可変性を表現した“コマンドの呼び出し”によって記述を簡略化する．つまり，普遍的なコマンドを DMM として蓄積／洗練するのである．

DSL

パッケージをまたがったデータ連携や集約処理が個別機能を実装する機能層の主な責務である．そこで記述すべき内容は，データ処理の流れ，データ・フローである．COBOL 言語などでバッチ処理を設計する際のフローそのものである．そこには左から右，上から下という一方向の処理の流れが記述されており，「反復」や「分岐」はない．すなわち，簡潔な「順次」を表現することが求められ，既存の部品を連携する「グルー（糊）・コード」が最良の表現手段となる．そこで，Scala4) や JRuby を用いて独自の軽量言語 (Domain Specific Language) をコア資産化している．DSL により，DMM の活用時の記述の最適化と，バグ摘出が困難な並列処理の弊害を排除している．

機能層の上位である制御層 (Controller) では，画面の状態管理や画面遷移制御にまつわる複雑な「分岐」が必要になる．そこでは，分岐条件と処理の関係を，メソッドへのアノテーションや，“ルール・オブジェクト”として管理する別の DSL をコア資産化している．

7&i グループ統合システムの事例

筆者らのドメインでは，システムを外部からパラメタ制御するような「静的」な可変性管理では可変性を表現できない．かといって，パッケージソフトのカスタマイズには多大な労力と時間を要し，費用対効果を上げることが難しい．そこで，上述した SPL プラットフォーム上で「静的」と「動的」な可変性管理を使い分け，顧客企業の要求に柔軟に対応している．

2003 年から始まった 7 & i グループシステムの統合は 2008 年 3 月に完了したが，本部系のシステムではこの SPL プラットフォーム上に構築されている [5]．

◉開発手法とプラットフォームの統一

グループ各社のシステムの再構築案件が，2003 年頃に重なった．これまで事業会社ごとに最適化されたシステムを構築／運用してきたのだが，取り巻く事業環境の変化に素早く対応するために，縦割りなシステムつくりを見直した．

まず着手したのが，事業から切り離せる実装面での共通化である．変化の激しい要素技術に振り回されることを避け，寿命の長い土台作りに配慮した．つまり，特定の製品やオープンソースに依存することのないように，アプリケーションと基盤ソフトウェアを分離している．

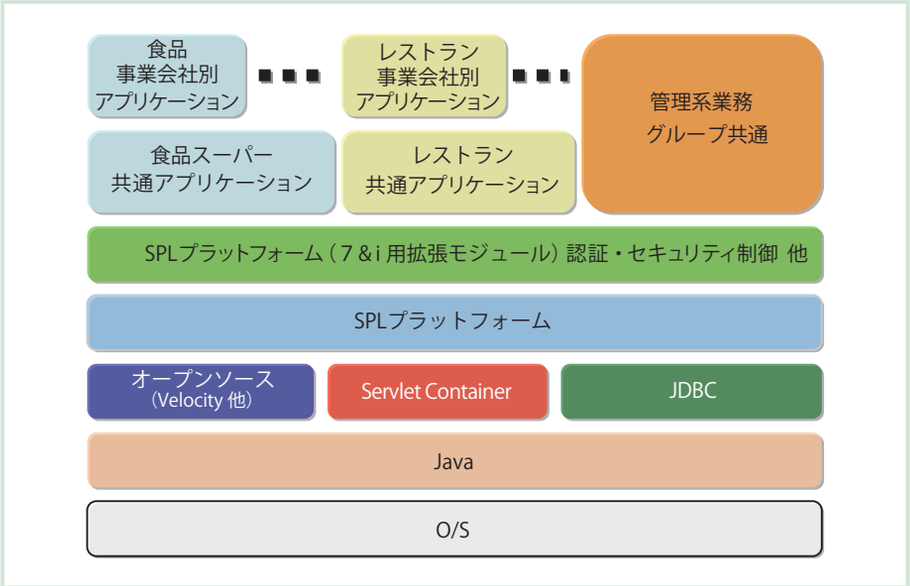


図 -7　アプリケーション・サーバ内のソフトウェア構成

たとえば，RDBMS 固有の“方言”が入る SQL の発行は SPL プラットフォームが担う．仮に将来 RDBMS が他の製品に変わったとしてもアプリケーションに手を入れずに移行できる．また，増え続けるデータ量，高度化する機能要求にタイムリーに対応するには性能問題は大きな関心事であるが，RDBMS 固有の技術に依存することなく応答性能を担保している．

◉要求仕様の共通化と個別要求の取り込み

実装面の共通化と平行して，顧客側も事業会社の枠を超えて業務面の共通化，システム側への機能要求の共通化を進めた．その際，食品スーパー，レストラン，管理業務系の 3 つのドメインにわけて仕様の検討を行った．

一方で，既存の他システムとの連携や，各事業会社固有の必須要件などは，バリエーションとして取り込んでいる（**図 -7** 参照）．

上述した関心事の多次元分離でシステム全体をスライシングした構造を持つので，局所的な部品の交換が可能となる．たとえば，帳票の項目やレイアウトの違いであれば，View からテーブルまで異なるが，データとロジックの双方を抽象化すれば多くのモジュールは共通化できる．部品間をインタフェースで抽象化し，事業会社ごとに実装クラスを切り替えることで，可変性を管理している．

顧客と経営と技術の融合が必須条件

エンタープライズ系の SPL の適用事例は，海外でもまだ数少ない．適用の難しさは，エンジニアリングの技術力だけでは解決できない課題が数多くあるからである．特に受託開発業務が中心のドメインにおいては，発注者側である顧客企業の理解と長期的なパートナーシップなくしては大規模な SPL 適用は難しいと考えられる．

また，生産者側はトップダウンとボトムアップの双方のアプローチから組織文化を変えていく継続的な努力が不可欠である．プロジェクト単位のコスト最適化の連続では，ドメイン最適なアーキテクチャは作れない．そしてそのアーキテクチャを使いこなす人材を育成することが鍵である．顧客の賛同，経営層の支援，そして志を共にする仲間，そのどれか 1 つでも欠けては形にならない．

参考文献

1) Tarr, P., Ossher, H., Harrison, W. and Sutton, Jr., S. M. : N Degrees of Separation : Multi-Dimensional Separation of Concerns, *In International Conference on Software Engineering*, IEEE Computer Society Press, pp. 107-119 (1999).
2) Czarnecki, K. and Eisenecker, U. : Generative Programming : Methods, Tools, and Applications, Addison-Wesley (2000).
3) Avalon/Apache Team, *WhatIsIoC* (2004), available as : http://wiki.apache.org/avalon/WhatIsIoC
4) The Scala Programming Language, Available as: http://www.scala-lang.org/
5) Ishida, Y. : Software Product Lines Approach in Enterprise System Development, *11th International Software Product Line Conference*, IEEE Press, pp.44-53 (2007).

（平成 21 年 1 月 31 日受付）

石田 裕三
y-ishida@nri.co.jp
1970 年生．1993 年野村総合研究所入社．1999 年（米）カーネギメロン大学留学．2001 年卒業（経営学修士・ソフトウェア工学修士）．