ソフトウェア開発技術の底流

技術主席

土屋　正登

Masato　Tsuchiya

masato-tsuchiya@exa-corp.co.jp

過去から継続している技術の底流を理解すれば、将来の動向を予想することができます。また、異なるように見える技術を統一的に理解できるメリットがあります。ここではソフトウェアの開発技術に関して底流を述べ、例をあげて解説します。底流として、手続的プログラムから宣言的プログラムへの移行につき８つの例を示します。次に、宣言的プログラムと手続的プログラムとの橋渡しとしてスクリプト言語を位置付けます。この事例として、試作した状態遷移マシンとDIコンテナを述べます。次にバイナリコード編集が活用されていることを指摘します。例えば、最新のフレームはバイナリコード編集を使用しています。また、AOPはバイナリコード編集により実装されています。つまり、部品の組み立てにバイナリコード編集を活用する傾向といえます。最後に極端から極端に走るアンチパターンの存在を述べます。このアンチパターンは技術リスクの識別と、次の展開(揺れ戻し)の予想に役立ちます。例としてメインフレーム、クライアント/サーバ、Webまでの分散システムにおける責務の分担の大きなぶれをたどります。また、実装の規約が多いフレームへの批判からPOJO(Plain Old Java Object)が奨励されましたが、逆にPOJOにこだわりすぎるため、ドメインオブジェクトの能力を殺すことになりかねない現状を指摘しました。最後に、IT技術者が長期的に成長するためには、流行を追うのではなく、継続的に新しい技術の優れた点と短所を評価し、優れた技術を吸収することが重要であることを述べました。

1. はじめに

過去から継続している技術の底流を理解すれば、将来の動向を予想することができますし、異なるように見える技術を統一的に理解できるメリットがあります。また、技術の流行は極端に走る傾向があります。流行している技術の必然性と将来性を評価するためにも底流の理解が必要です。難しいテーマですが、ソフトウェア開発に関する技術に限って挑戦してみます。なお、インターネットには語句の適切な解説がありますので、インターネットによる検索と共に読まれることを前提で説明を省略します。

2. 手続から宣言的プログラムへ

今年のJavaOneではスクリプト言語 JavaFX が発表されました。JavaFX は"プログラムを手続的に記述してUIを構築するのではなく､UIコンポーネントの構造に近い形で宣言的に記述できる"とのことです｡この"手続的(procedural)プログラムから宣言的(declarative)プログラムへ"は底流です。

宣言的プログラムでは要求(what)を記述するのみで、要求を達成する手段(how)を記述しません。つまり、宣言的プログラムはトップダウンアプローチです。例えば、SQLはかなり宣言的といえます。SQLでは欲しいデータを獲得するためのマージ、ソート、検索等のアルゴリズムを記述せずに、欲しいデータの仕様を記述します。一方、アルゴリズムは手続的です。アルゴリズムを実行するためにはC や Fortran 等の手続的プログラム言語で記述しなくてはなりません。HTML も宣言的です。HTMLはページの構成を指示しますが、ディスプレイにおける実際の表示を指示するものではありませんのでブラウザによって表示が異なります。

手続的プログラムで記述していたところから、宣言として表現できることを分離して、宣言の記述に適した表現に置き換える傾向を "手続から宣言へ"と呼びます。宣言の記述は図による表現と宣言的な言語を使用する方法があります。まず、 宣言を図で表現できれば視覚的・直感的な理解が可能となります。ただし、図を実行可能な言語に変換できることが前提です。つまり、文法を有する図の表示規則が必要です。次に、SQL のように独自の言語を考案してもよいでしょう 。最近は XML を利用して新しい宣言的な言語が大量に作られています。Object Relational Mapping（O/R mapping ）や DIコンテナ等のフレームワークにおける XML による設定ファイルは宣言的なプログラムであると考えることができます。Java 2 からはプログラム中に記したアノテーションを実行時まで記憶しておき、プログラムの振る舞いを制御することができます。アノテーションはソースコードにコンテキストが依存する一種の宣言的なプログラムです。なお、宣言が属するドメイン に着目すれば、宣言を記述する言語を一種の Domain Specific Languages（ DSL ）と考えることもできます。

なお、XML で表現されているから宣言的であるとはいえません。XML はメタ言語なのでXMLを使用して手続的なプログラム言語を作ることができます。

ここで、手続を宣言に変えることによる、我々にとっての 価値をまとめてみます。

(1) 手続よりは宣言を記述するほうが簡単である。
(2) 宣言を達成する手段を取り替えることができる。
(3) 宣言と手続を分けることは関心の分離（範囲を狭くして集中的に考えられるようにする）になる。
(4) 視覚的・直感的な宣言の表現を工夫できる。
(5) 宣言が属するドメインに対して一般解を考えることができる。宣言は問題の定式化である。

以下、"手続から宣言へ"が底流であることを例で説明します。

（例１）SQL / OCL / インタフェース

SQL はプログラム言語より宣言的です。即ち、SQL では欲しいデータの仕様を記するのみで、それを獲得するためのアルゴリズムを記述しなくても RDBMS が解釈して実行します。古くからある埋め込み SQL は手続と宣言を組合せた良い工夫です。ER図と集合演算を表すベン図は SQLを組み立てる際に視覚的な思考補助になります。

UML における Object Constraint Language（ OCL ）も宣言的です。OCL は図と共に使用して図では表現できないモデルの拘束条件を表現します。OCL は SQL のように集合を扱うことができます。

Java におけるインタフェースは実装がなく宣言のみのクラスです。何の役に立つか疑問に思われますが、インタフェースはクラスの実装者とクラスの利用者間の約束事になり、それ以上立ち入った関心を断ち切るのに

役立ちます。

（例２）Dependancy Injection（ DI ）コンテナ

DI コンテナが登場するまでは手続的プログラム言語によりオブジェクトのインスタンスを生成し、関連付けするのが一般的でした。しかし、生成と関連が静的であれば、インスタンスの生成と関連付けを DI コンテナの設定ファイルとして切り出すことができます。この設定ファイルは宣言的なプログラムと考えるべきです。設定ファイルに宣言されたオブジェクトの生成と関連付けを実行するのが DI コンテナです。DIコンテナに関しては視覚的な宣言は難しいようです。原理的には UML のインスタンス図から DI コンテナの設定ファイルを生成できます。しかし、インスタンス図の作製は煩雑なので実用的ではありません。

ソースコードにアノテーションを挿入する DI コンテナがありますが、問題をはらんでいる方法です。なぜなら､DI コンテナはインスタンスを扱います。一方、Java や C++ のソースはクラスを表現しています。ソースにインスタンスに関するアノテーションを記入すると、クラスの再利用性を制約してしまいます。

（例３）宣言的 GUI の記述

フレーム、パネル、テーブル、ボタン等の GUI の部品の構成は階層構造をしています。この構成をプログラムで記述するのは退屈な作業です。したがって、構成をプログラムから抜き出して XML 等で宣言的にしてしまうのは自然な発想です。これを実現した製品として、Windows Vista の XAML, Firefox で使用されている XUL が有名です。また、SwiXML,SwiXAT,XUI,gui4j, Xulfaces等の多数の小規模なオープンソース系の製品があります。今後、宣言的に画面部品を構成する GUI フレームワークが主流になるのは確実です。

GUIの構成とレイアウトを直接XMLで記述するのではなく、グラッフィックにより、実物を見ながら構成するほうが直感的で視覚的になります。即ち、GUI ビルダーを使用したWYSIWYGです｡GUI ビルダによりXML ファイルを作成します｡GUI フレームワークはこのXML ファイルを受け取り実行します。 なお、これまでの製品における XML の設計は GUI 部品の構成とレイアウトと GUI フレームワークのアーキテクチャ(イベント、モデル､コントロール等)を分離していない問題があります。つまり、２種類の宣言が混在しているのです。GUI 部品の構成とレイアウトは GUI ビルダーで、イベントリスナー、モデル、コントロールの組み立ては DI コンテナと分離すべきです。これによって GUIビルダーを GUIフレームワーク のアーキテクチャから独立させることができますし、DIコンテナを活用することができます。

（例４）状態遷移 / ワークフロー

ワークフロー、GUIの画面遷移等は状態遷移図で記述することができます。また、Stephen Mellor氏は ExecutableUML として実行可能なモデルを提唱しています。この実行エンジンは一種の状態遷移マシンではないかと思っています。このようにオブジェクトの動的振る舞いを表現する手段として状態遷移図の応用は広いので、JBoss jBPM のような汎用的な状態遷移マシンが製品化されています。状態遷移図は宣言的に記述できます。例えば、UML を使用してもよいし、jBPMでは専用のGUIを用意しています。Bussiness Process Modeling Notation（ BPMN ）は図によるワークフローの表現です。BPMN を Businesss Process Modeling Language（ BPML ）に変換してからワークフローエンジンによりプロセスを実行します。

（例５）Object Relational Mapping（ O/R mapping ）

Hibernate や EJB3.0 ではクラスとエンティティのマッピングを設定ファイルに記述しておけば、オブジェクトと RDB 間の変換を自動化してくれます。これにより、マッピングのための機械的なプログラムを排除することができます。

（例６）Model Driven Architecture（ MDA ）

UMLで表現されたモデルは実装より直感的・視覚的に理解することができます。そのモデルは宣言に属するものだと考えています。実際、オープンソースの MDA ツールとして有名な AndroMDA は宣言的な部分に積極的に MDA を適用しています。例えば、クラス図 から、インタフェースやクラスのスケルトンを生成したり、O/R mapping の設定ファイルを生成したりします。また、状態遷移図で表した画面遷移から Struts の設定ファイルを生成します。なお、 モデルは宣言に属しているため、手続を取り込むことは難しいと考えています。例

えば、計算式や複雑な論理は手続に属し、UML では表現できません。MDA は手続的なプログラムとの共存を考えるべきです。

（例７）宣言的なビジネスルール

手続プログラムでビジネスルールを表現した場合、ビジネスルールを理解したり修正できる業務担当者は稀です。また、プログラム言語に依存しているので再利用性が低くなります。ビジネスルールが宣言的に表現でき、それをシステムで実行可能であれば、ビジネスルールを企業の知的財産として管理して利用することが可能になります。手軽に利用できるビジネス・ルールの表現と実行エンジンが望まれます。

上記の例に示した解決法は特殊解ではなく一般解を与える能力があります。それだけに、習得するための時間が必要で、プロジェクト期間内で習得することは難しいです。実際、時間的な制約のため、当面の業務要件を満足するだけの特殊解に留まるプロジェクトを見かけました。しかし、例に示した解決法により柔軟で発展性があるソフトウェアとすることができます、プロジェクトを越えた組織的な取り組みが必要であると考えています。

一方、"手続から宣言へ"には弊害もあります。

(1) 宣言を記述する言語の氾濫

オープンソースのフレームワークでは設定ファイルの記述に大きな負担が伴います。設定ファイルを宣言的プログラムと考えると負担があるのは当然です。フレームワーク固有の言語の習得が必要になるからです。複数のオープンソースを使用すると XML 地獄を感じます。アノテーションを併用すると設定ファイルの記述が簡単になりますが、アノテーションも独自言語です。標準化が強く望まれます。

(2) プログラムの分散化

１つのプログラム言語で記述していたのが、プログラム言語と XML やアノテーション等の宣言に分散してしまうので全体的な見透しが悪化します。この分散化は開発だけでなく維持を難しくします。構成管理ツールの使用は当然ですが、これからは分散した開発成果物間の依存関係を管理できる高度な構成管理が必要になるでしょう。

(3) トラブルシュートが困難

トラブルがあると宣言を実行する仕組みに立ち入らなくてはならないことがあります。

例えば、 SQL のチューニングです。性能問題があるとRDBMS による SQL の解釈を制御することまで踏み込むことになります。悪いことに、宣言を実行する仕組みは一般解を与えるために高度で複雑です。

3. スクリプト言語とコンパイル言語の組合せ

Java SE 6 では Java から JavaScript を使用することができるようになりました。Java SE 7 では、 更に、Beanshell,JRuby,Jython,Groovy,JavaFX などのスクリプト言語のサポートが予定されているようです。スクリプト言語から Java を使用する効用は理解できます。Unix では C 言語で開発したコマンドをシェルスクリプトにより連携してさせています。プログラム言語によるコマンドは単機能の実現に徹して、シェルスクリプトにより複雑な機能を実現します。 同様に JavaScript や Beanshell から Java を使用することができます。逆に、筆者にはプログラム言語からスクリプトを使用する意義が明確ではありませんでした。Java を多言語化するだけの目的なら、スクリプト言語だけでなく、C / C++ や C# もサポートすべきであると思っていました。

ところが最近、スクリプト言語が宣言から手続への橋渡しになることに気が付きました。宣言から手続を起動したい場合を考えてください。例えば、XML で宣言を表現して、かつ、XML 中には手続を起動するスクリプトが含まれるとします。JAXB のような XML バインディングツールを使用すれば XML を Java のインスタンスに変換(unmarshal)することができます。この際、スクリプトは文字列としてインスタンスに含まれます。このスクリプトを実行することは、即ち、Java からスクリプト言語を使用することになります。今後、宣言的なプログラムで骨格を作り、それを実行可能とするためにスクリプト言語で手続的な言語に接続するプログラムパラダイムが興隆するでしょう。

以下、Swing をベースとした GUI フレームワークの試作におけるスクリプト言語の活用例を示します。

（例１）状態遷移マシン

状態遷移は状態、イベントとアクションで構成されます。イベントが生じた時、状態が変化してアクションが起動されます。アクションの起動をスクリプトで記述することにより、XML で記述した状態遷移が実行可能になります。スクリプトを使用できないと、アクションを記述する言語を定義してインタプリタを実装しないと実行可能にはなりません。Stephen Mellor氏によるExecutableUML ではモデルの振る舞いを状態遷移と独自なアクション言語で表現しています。Java SE 6を使用すれば独自なアクション言語の代わりに、使い慣れたスクリプト言語と Javaを組合せることができるのではないかと考えています。

試作した GUI フレームワークでは画面遷移や入力のナビゲーションに状態遷移マシンを利用します。実際のアプリケーション：スキル管理システムにおけるフレームの状態の１つを示します。フレームの状態 "S3" では個人のスキルを表示します。

```
<state id="S3">
  <onentry>
    <action>controller.showContl.init()</action>
    <action>showMySkill.getSkillTreeTable().
                        expandAllNode()</action>
    <action>behavior.setVisible( showMySkill )
                                        </action>
  </onentry>
  <onexit>
    <action>behavior.setInvisible( showMySkill )
                                        </action>
  </onexit>
    <transition event="logout" target="S1"/>
    <transition event="return menu"
                                    target="S2"/>
  </state>
```

<onentry> は状態に入った時のアクションをまとめたものです。ここでは３つのアクションをスクリプトで記述しています。そして、スクリプトを経由して Java を使用しています。例えば、behaviorはフレームワークが提供している Javaのインスタンスです。setVisible (showMySkill)により showMySkill 画面（パネル）で個人のスキルを表示しています。 <onexit> は他の状態に遷移する時のアクションをまとめています 。この場合は、showMySkill画面を非表示にしています。<transition> はイベントを受けた時に遷移すべき状態を指示します。例では不必要なので記述がありませんが、要素として <action> を記述することができます。また、スクリプトから Java のインスタンスを使用するためには、スクリプトエンジンに Javaのインスタンスを注入しておく準備が必要です。このためにDIコンテナから状態遷移マシンにインスタンスを受け渡しています。

（例２）DI コンテナ

フレームワークでは DI コンテナを次ぎの目的で使用しています。

(1) アクションリスナを Swing のコンポーネントに設定する。
(2) テーブル等のコンポーネントにモデルを設定する。
(3) フレームやパネルのようにまとまった大きな画面部品を組み立てる。
(4) フレームワークに属するオブジェクトをアプリケーションに注入する。

DI コンテナが管理するオブジェクトは例１の状態遷移マシンが使用するスクリプトエンジンでも利用することが可能です。

スクリプトエンジンを使用することができる独自 DI コンテナを試作しました。理由は、製品化されているDI コンテナはコンテナが管理していないオブジェクトにアクセスできないからです。DI コンテナが管理しているオブジェクトからナビゲーションできる全てのオブジェクトにアクセスするためです。この要求が生まれた理由を説明します。

試作したフレームワークは NetBeans 5.5 の Matisse のような GUI ビルダを使用して GUI 部品を構成してレイアウトされることを前提としています。NetBeans 5.5は構成とレイアウトに関するJavaプログラムを生成します。生成されたフレームやパネルのような大物部品は DI コンテナで管理します。一方、生成されたプログラムにはパネルに含まれるボタンやテーブルを生成するコードが書かれています。残念ながら、プログラムが独自に生成したインスタンスはDI コンテナの管理外になってしまいます。したがって、ボタンやテーブルなどの小物部品には DI コンテナを活用した設定ができません。

例えば、ボタンにアクションリスナを、また、テーブルにテーブルモデルを設定することができません。この問題を、JavaScript の記述ができる DI コンテナにより解決しました。DI コンテナの設定は次のようになります。

```
<component name="jpanel1" script="new
                                    JPanel()">
    <operation script="getTable1().setModel
                                ( model1 )"/>
</component>
<component name="model1" script="new
                                MyTableModel()"/>
```

ここでは パネル："jpanel1" と テーブルモデル："model1" の２つのコンポーネントを DI コンテナに登録していますscript="…"の 部分が JavaScript です。"jpanel1" は テーブル:"table1" を含んでいます。<operation> の script は "jpanel1" から "table1" にナビゲーションして、テーブルモデル："model1" を設定しています。DI コンテナは "table1" を管理していません。

しかし、開発者が宣言的プログラムの意義を理解してない場合、宣言においてスクリプトの使用を許すと、宣言に手続を記述してしまう逸脱の危険があります。例えば、例１の状態遷移の <onentry> における <action> の始めの２つには手続が混入している疑いがあります。

```
<action>controller.showContl.init() ; </action>
<action>showMySkill.getSkillTreeTable().
                        expandAllNode()</action>
<action>behavior.setVisible( showMySkill )
                                </action>
```

始めの２つの <action> は記述した順に実行されることを期待していて、その順序が異なるとエラーになる可能性があります。この２つは controllerのメソッドとしてまとめたほうが良いでしょう。製品化されている DIコンテナである Searers2 はスクリプトより制限が強いOGNL（ Object Graph Navigation Language ）をサポートしています。スクリプトの危険性を理解しているようです。今回、我々が試作したDI コンテナでは逸脱を制限するため、完全なスクリプトを書けないようにしています。例えば、例２の DI コンテナの設定リストを、完全なスクリプトで書き換えると次のようになります。

```
<operation script="jpanel1.getTable1().
                        setModel( model1 )"/>
```

DIコンテナが内部の処理で "jpanel1"を追加しています。つまり、メッセージを送る対象を<component>タグのname属性で指定したオブジェクトに限定しています(例２の DI コンテナの設定リスト参照)。なお、現在はJakarta BSFを使用して Beanshell を使えるようにしました。全て、Javaで書いている感覚です。

全てスクリプト言語で開発してもよいのではないかとの意見もあります。現状、スクリプト言語は実行速度が懸念されますし、既存のソフト資産を活用しないと限界に突き当たるかもしれません。しかし、Ruby やPythonを筆頭にしたスクリプト言語の隆盛を見ていると全てスクリプト言語で開発する時代がくるかもしれません。しかし、筆者には、Javaより Ruby のほうが簡単だとは思えません。Ruby は Java　より純粋なオブジェクト指向言語です。また、RubyはJavaにはないクロージャがあります。また、Javaコンパイラのエラーチェックのおかげで論理的なエラーを事前に避けることができました。正直、スクリプト言語が好かれる理由が分かりません。

4. フレームワークにおけるバイナリコード編集の活躍

開発者がフレームワークを利用するために、フレームワークが要求するインタフェースの実装を強いられた時代がありました。例えば、OODB(オブジェクト指向データベース)により永続化したいクラスは特定のクラスを継承するとか、EJB 2.0 では実装すべきインタフェースの多さが非難されました。 最新の　Seaser2, HibernateやEJB 3.0 等のフレームワークでは POJO（ Plain Old Java Object ）による開発 が可能です。POJOは Martin Fowler 氏の造語であり、簡単な昔ながらのJava オブジェクトの意です。POJOはフレームワークの都合による実装の取り決めを強要されない、フレームワークから切り離しても試験と再利用できるプログラム開発を目指しています。

POJO による開発を可能にしたのは、バイナリコードを実行時(正確にはクラスをロードする時)に編集する技術をフレームワークに採用したからです。例えば、フレームワークはフレームワークが必要とする機能をフレームワークのクライアントのクラスに追加する、または、クライアントのクラスを継承して機能を追加することを実行時に行っています。そのため、フレームワークで動作するクライアントのクラスは開発者が作ったとおりのコードではありません。バイナリコード編集は以前からありましたが、フレームワークがそれに活躍の場を与えました。プログラムの構造をプログラムが知ることができるリフレクション、実行時の振る舞いの指示をコードに埋め込むことができるアノテーション、それと、バイナリコード編集を組合せたパラダイムはフレームワークに限らず広い適用分野があるように思います。

ビジネスアプリケーションの開発に、直接、バイナリコード編集を使用することは稀でしょう。ただし、ソースコードがないライブラリを変更したい、または、統合開発環境製品が生成するコードを変更したい場合に、切り札として、バイナリコード編集を使用することが考えられます。前述の GUI フレームワークの試作 では、バイナリコード編集により、 NetBeans 5.5 の Matisse が生成したコードにメソッドを追加しています。

さて、バイナリ編集のツールですが、Seaser2 と JBoss が使用しているJavassist[1] をお勧めします。Java のコードの断片を入力すればバイナリ編集してくれるので、Javaの機械語の知識が不要です。編集したクラスをファイルに書き出し、これを javap で逆アセンブルするか、jad 等のディコンパイラで Java に戻すことで、編集したコードを確認することができます。

オブジェクト指向の問題である横断的関心の分離を実現する技術 AOP(Aspect Oriented Programming)が注目されています。AOP の実装はバイナリコード編集です。AOP はアプリケーションが使用できるように制御されたバイナリコード編集といえます。AOP では、実行時だけでなく、バイナリファイルを編集することもできます。

Fortran 77 のころはプログラムとプログラムの実行時のイメージが同じでした。C言語 は実行時に動的にメモリを割り当てて使用することができます。これにより、予め配列の大きさを決めておく制約がなくなりました。オブジェクト指向言語ではプログラムは型になりました。そして、プログラムとプログラムの実行の間にインスタンス化のプロセスが割り込んできて、プログラムとプログラム実行時のイメージの間に隔たりができました。この相違は、鯛焼きを作るのと、鯛焼きを想定して鯛焼きの金型を作るほど大きいです（Cobol から Javaへの移行が難しい原因ではないかと思っています）。更に、バイナリコード編集により実行時のプログラムの動作が本当に分からなくなりました。このように、決定をプログラム時点から実行時に遅らせることで柔軟性を獲得するのが継続的な傾向です。

現在、バイナリコード編集の知識がなければフレームワークは魔法使いのブラックボックスです。しかし、ブラックボックスとして完全に無視できるまでには成熟していません。例えば、オブジェクトをシリアライズして他のシステムに転送する場合は、バイナリコード編集が問題になります。プログラマがPOJOであると思っていてもバイナリコード編集により変更されていることがあるからです。フレームワークの開発者はバイナリコードをどのように編集しているか説明する責任があります。

5. アンチパターン：極端から極端へ

ここでは技術が極端から極端に振れるアンチパターンについて述べます。このパターンではユーザ企業の情報システム部門にIT投資のための理論武装を与えたいメーカや問題を指摘しないマスコミにより振れが増幅されることがあります。アンチパターンは技術リスクの識別と、次の展開(揺れ戻し)の予想に役立ちます。

5.1. リッチクライアントに至るまで

現在、リッチクライアントが注目されています。ここでは、クライアントのアプリケーションがコンテキストを有することをリッチクライアントの条件とします。すると、JSF （ JavaServer Faces)はリッチクライアント技術から外れます。Ajax はサーバと非同期通信ができるので、リッチクライアントに属します。筆者は、リッチクライアントを歓迎しますが、そこに至る経緯には疑問を感じます。これからアプリケーションの実行形態についての歴史を述べます。中庸に止まってほしいのですが、クライアントとサーバの責務の分担が正反対に振れるのです。

(1) メインフレームにおけるダム端末型のアプリケーション

アプリケーションは常にメインフレーム上で動作していました。端末は単純な表示機能しかなくダム(dumb)端末と呼ばれました。

(2) クライアント・サーバ型のアプリケーション

アプリケーションはクライアントPC のみで動作します。サーバでは DBMS 等のサーバプロセスが動作します。メインフレームが不要になり撤去できるような過大広告を記憶しています。しかし、ネットワークがボトルネックになるスケーラビリティがないアーキテクチャでした。例えば、開発したアプリケーションを納入したら、応答が悪く実用に耐えないというクレームを受けたことがありました。調べると、納入先では電話線のモデム経由で使用していました。開発で利用したイントラネットの環境とは 1000倍程度の速度差がありました。その他にもクライアントソフトウェアの配布の負荷が新たに情報システム部門の負担になることもありました。

(3) 3階層型のアプリケーション

このタイプは、クライアントPCではプレゼンテーションを実行するアプリケーションが、アプリケーションサーバでは業務ロジックが、DBサーバでは DBMS 等のサーバプロセスが動作します。スケーラビリティと負荷分散を考慮したバランスの取れた良いアーキテクチャでしたが、分散トランザクションをサポートするTPモニターが難しいため、短命でした。ただし、DB サーバでストアード・プロシージャを使用する論理的な3階層は生き延びたようです。

(4) Webブラウザ活用型のアプリケーション

Web はハイパーリンクが売りで、ビジネスアプリケーションを動作させる環境ではないと筆者は考えていましたが、流行して今日に至っています。ダム端末と同様にアプリケーションはサーバのみで動作します。ブラウザはマルチメディア化したダム端末です。当然、サーバに負荷が集中します。また、多数のクライアントのコンテキストを保持するため、サーバのメモリ資源を必要とします。また、ブラウザとサーバ間の通信の遅延による応答の鈍さは予想されるはずです。Excel 等のアプリケーションを使い慣れたユーザの要求を満足させるにはHTMLによる静的な表現では足りなく、JavaScript によるDOM(Document Object Model)操作と非同期通信、即ち、Ajax に移行しつつあります。巨大なコモンデータであるDOMを多数の関数が操作するため、Ajaxは保守が難しいことが予想されます。Ajax にはオブジェクト指向以前の状態に戻ってしまわないようにする仕組みが必要です。

(5) リッチクライアント型のアプリケーション

3階層のアーキテクチャに戻ったように思います。ここに留まってほしいと筆者は考えています。

このように、技術が正反対に振れるのは過去のアーキテクチャの問題を反省しないからだとも思います。例えば、EJB における Entity Bean は DB の1レコードに相当する小さな粒度のオブジェクトです。このように小さな、かつ、多数のオブジェクトをリモートからアクセスすれば、通信がボトルネックになることは当然予想できます。CORBA でも同様の問題があり、オブジェクトの値渡し(Object By Value)で対処しました。EJB ではオブジェクトの値渡しが DTO (Data Transfer Object) パターンと改名されて推奨されています。 CORBA の経験から、Entity Bean を作るべきはなかったのです。新しいEJB 3.0では、新たな永続化フレームワークの出現によりEntity Beanは無用化される運命です。

5.2. ドメインオブエジェクトはPOJOでよいのか

データを保持しているだけで機能が乏しいドメインモデルの問題を Martin Fowler 氏が Anemic Domain Model (貧血のドメインモデル)としてBlog[2] に投稿しています。ドメインオブジェクトに実装されるべき機能がサービス層に実装されるため、サービス層が肥大化する問題です(筆者も経験しています)。氏はドメインモデルのメリットを理解しない、データ中心の設計者を非難しています。また、Entity Bean のようなフレームワークの欠陥が貧血のドメインを助長しているとして、POJO を推奨しています。筆者は不勉強な設計者の問題は賛成しますが、POJO については"過度のPOJOへのこだわりが貧血のドメインモデルを招く"のでは ないか と懸念しています。以下にその理由を説明します。

ドメインオブジェクトから永続化機能(通常､スタティックなメソッドになる)を取り出ししたDAO (Data

Access Object)クラスは有用です。 SQLが得意な開発者の作業を切り出すのに便利だからです。これにより、ドメインオブジェクトは POJO になることが期待されます。

しかし、ドメインオブジェクトから DAO を使用したいこともあります。例えば、ドメインオブジェクトがRDBの集約関数を使用する場合です。また、ビジネスルールをドメインオブジェクトに実装した場合、ビジネスルールの判定に従ってDBを検索することもあります。更に、ドメインオブジェクトが通信により他システムのオブジェクトの機能を利用することもあるでしょう。結局、ドメインオブジェクトはRDBや通信と無関係な POJO であってはならないのです。

残念ながら、EJB 3.0 は、POJO が DAO（ Data Access Object)を使用することを想定していないようです。なぜなら、EJB 3.0 でPOJOからDAOを使用するためには、DAOをローカルなStateless Session Beanとして登録しておき、loookup で引き当てる複雑な手順が必要になるからです。筆者は次のような構文を期待しています。

```
@Entity
public class Customer implements java.io.
                                    Serializable {
    @In
    transient CustomerDao customerDao;
    private String id; //  主キー
    private double discountRatio;

    public double getAccount( Date date ) {
        return customerDao.getAccount( this.id,
                date ) * ( 1.0 - discountRatio );
    }
}
```

ここでは、コンテナがCustomer のインスタンスをRDBやシリアライズから復元する際に、DAO のインスタンスであるcustomerDao を注入してくれることを前提としています。メソッド getAccount では このDAO を使用しています。このメソッドはビジネスルールである割引を記述しています。このクラスはDAOを有するのでPOJOではありません。しかし、このような実装ができないと、ドメインオブジェクトに属するべきビジネスルールがファサードに移動してしまいます。貧血のドメインモデルの出現です。なお、プログラムでCustomer のインスタンスを明示的に生成する際にもDAOの注入が必要です。このために自分でファクトリクラスを作り、DAOを注入ことになるでしょうが、この程度の開発負荷は我慢できます。ローカルなStateless Session Bean でファクトリクラスを実装すると簡単になるかもしれません。なお、軽量DI/AOPコンテナとして有名なSpring Framework 2.0 はAOP とDI コンテナを組合せて、DAO をドメインオブジェクトに注入することができるようです。

オブジェクト指向のメリットは分析モデルを素直に実装できることであると考えます。ドメインモデルは共通の語彙として、また、ビジネスルールをクラスの責務として整理できる等のメリットがあります。フレームワークの設計者は POJO だけでなく、分析におけるドメインモデルが素直に実装できることを目標としてほしいものです。

6. おわりに

消えてしまったと思われる技術が形を変えて復活することがあります。例えば、SGMLは XML として復活しました。表舞台からCORBA は消えてしまったようですが、RMI over IIOP、分散トランザクションやディレクトリサービスは EJBを支える技術になっています。IT技術者が長期的に成長するためには、流行を追うのではなく、継続的に新しい技術の優れた点と問題点を過去の技術と照らし合わせて評価し 、その流れの中で優れた技術を吸収していくことが重要です。安易に新しいことは全て良いことだと思ってはいけません。以上ソフトウェアの開発に携わる技術者の刺激になれば幸いです。

参考文献

1) http://www.csg.is.titech.ac.jp/~chiba/notes/javapress03/index.html
2) http://capsctrl.que.jp/kdmsnr/wiki/bliki/?AnemicDomainModel
http://martinfowler.com/bliki/ 日本語訳へのリンクがあります。
