技術レポート

JAVAアプリケーションフレームワークの開発について

産業システム第三事業部
淵　靖英

産業システム第三事業部
角谷信太郎

1．はじめに

様々な雑誌等でフレームワークが紹介され、注目されるようになって久しい。しかし、我々が携わっている開発の現場における認識や体制が追い付いていかないケースも多く発生するようになり、見過ごせない状況になってきている。例えば、Javaなどに代表されるオブジェクト指向技術を用いたシステム構築を行っても、スキルや認識等の不足により、フレームワークを適用できなかったり、有効活用できなかったりするケースがある。そこで、フレームワークそのもの、および我々の取り組み方について、本稿で改めて紹介する。フレームワークがどのようなものなのか、今まではイメージが掴めていなかった方にも理解していただければ幸いである。

2．フレームワークについて

2.1　フレームワークとは

『増補改訂Javaプログラミングデザイン』(参考文献1.)には、フレームワークについて、以下のように記載されている。

「フレームワーク（framework）とは、特定の目的のために再利用できるように設計されたモジュール群のことである。…（中略）…フレームワークの利用者、つまり一般のアプリケーション開発者が、カスタマイズ（部分的に変更）したり、他の部品と組み合わせたり、フレームワーク自体を拡張したりすることによって、個々の要求を満たすアプリケーションを少ない労力で開発できるようにすることを目指している」

つまり、フレームワークとは開発工数を削減するためのモジュール群なのである。

部品のライブラリ化、コードの再利用、スケルトン利用等、開発工数を削減するための手法は、今までにも多種多様にわたって数多く存在した。しかし、ここに紹介するフレームワークという手法は、既存の手法とは少し毛色が異なっている。手続き型言語としての適用も可能であり、さらにJavaに代表されるオブジェクト指向の開発言語の特性も持っている手法なのだ。

2.2　フレームワークの基本コンセプト

基本的にフレームワークは、以下の3つのコンセプトに基づいて構築を行う。

⑴　**オブジェクトのロールの分離**

UNIXのコマンドに代表されるように、単純な機能の部品を組み合わせて複雑な機能を実現する場合、複雑な機能そのものの再利用性は低いかもしれないが、各部品の再利用性は高い。また、部品そのものが単純なので、構造を理解することが容易であり、メンテナンス性も向上する。フレームワークでも、設計段階からオブジェクトとロールを分離し、各部品の目的を単純かつ適正なものとすることで、高い再利用性とメンテナンス性を実現することが可能となる。

⑵　**階層化することによるアーキテクチャと実装の分離**

アーキテクチャと実装との間に直接的な関連はない。逆に考えれば、アーキテクチャを変更せずに実装を変更することは可能なはずである。すなわち、階層化することでアーキテクチャを表現し、さらに、その階層に従って実装を行い、多様性のあるフレームワークを実現する。

⑶　**抽象化を進めることによる柔軟性の向上**

フレームワークが利用されて得られたフィードバック、技術の進歩や変化等により、よりよいフレームワークの実装方法が見つかるかもしれない。しかし、フレームワーク

の実装と構築されたアプリケーションの実装が密接な関係にある場合、フレームワークの実装を変更することは、広範囲なアプリケーションに影響を及ぼしてしまう恐れがある。そこで、実装を容易にすることを目的として、アーキテクチャに抽象化された階層を設ける。そして、その階層に基づいてアプリケーションを実装することで、相互に密接した状態からの分離を実現する。これにより、抽象化された階層が変化しない限りは「フレームワークの実装は、アプリケーションの実装に影響しない」という利点が発生する。

これら3つのコンセプトからもわかるように、フレームワークは「柔軟性」と「再利用」を最大の目標としているのである。

2.3 従来のスケルトンを利用した開発との違い

部品のライブラリ化やコードの再利用という既存の開発手法は、フレームワークとは基本的に別次元の手法である。つまり言い換えれば、これら手法とフレームワークとを同時に開発に適用することは可能だ。しかし、スケルトンを利用した開発手法とフレームワークとは根本的に考え方が異なっている。従来の開発において、数多く適用されてきたスケルトンとフレームワークとが、どう違うのかを以下に述べる。図1を併せて参照されたい。

いまさら解説するまでもないが、スケルトンを利用する開発手法は、まずプログラムの基本的な流れをスケルトン（骨格）として作成する。あくまでも、基本的な流れであるので、ロジックはほとんど含まれない。開発者は、スケルトンを複製してからロジックを肉付けしていく。複数のプログラムを作成する場合は、複数のスケルトンを複製して、それぞれに肉付けしていく。肉付けするロジックには共通のロジックも含まれるので、それらを部品として抽出したうえでライブラリ化することで効率化をはかる。

しかし、スケルトンそのものが、あまりにもスリムなものであった場合、付けるべき肉が、ひじょうに多くなってしまい、メリットが少なくなるという問題が生じる。スケルトンは、流れを実装するものであり、カスタマイズや拡張性を重点的に考慮して実装されるわけではないので、柔軟性を持たせるには、よりスリムにならざるを得ない。したがって、スケルトンそのものに変更が発生した場合には、すべての作成したものに対して変更が発生するのである。特に検証を行ったわけではないのだが、スケルトン利用によって削減できる工数というのは、作業を開始する段階において、立ち上げが速くなったことで生じる減少分ではないだろうか。

これに対して、フレームワークを利用した開発というのは、「従来のスケルトンに当たる“フレーム”部分に、ある特定のアプリケーション構築のためだけに作成された“ワーク”部分が追加されたもの」であるといえる。フレー

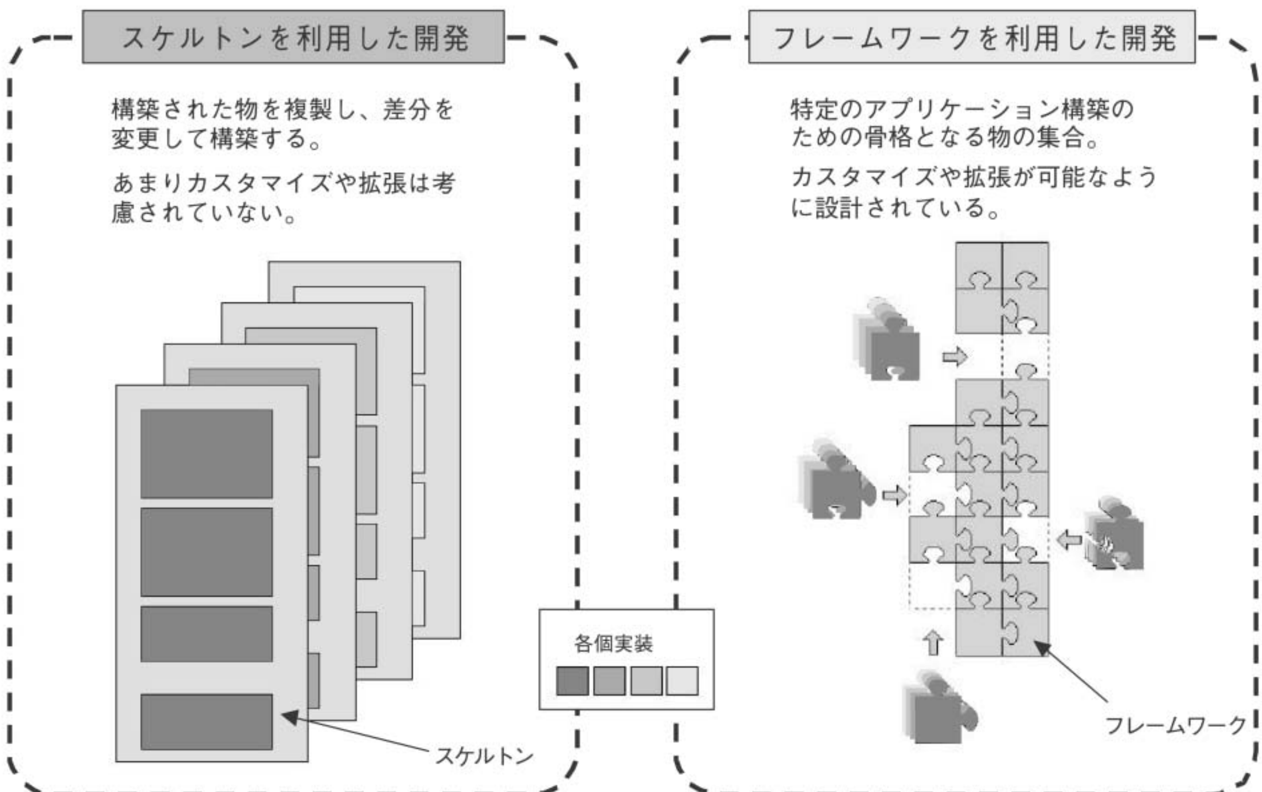

図1　スケルトンとフレームワークのイメージ

ム部分があるので、スケルトンと同様の流れを実装しているが、同時にワーク部分であるロジックも実装している。実際は、フレームワークにはロジックの各要所が既に実装済みなので、欠けている部分のみを個別に埋めるために、フレームワークをカスタマイズ、または拡張していくイメージになる。つまり、フレームワークそのものは、拡張性を十分に考慮したうえで設計を行っているのである。

また、オブジェクト指向の開発言語を利用してフレームワークを構築しておけば、複数のアプリケーションを構築する場合でもスケルトンのように複製する必要はない。メモリー上に展開する複数の実体を作るように実装すればよいので、フレームワーク部分のメンテナンス性は向上するのである。

2.4 特徴

フレームワークをより深く理解していただくために、以下に4つの特徴を紹介する。

(1) ライブラリの性質とアプリケーションの性質

フレームワークを利用したシステム構築は、策定したアウトラインに沿った設計から始まる。アウトラインの想定範囲を外さずに設計する限り、フレームワークをカスタマイズ、または拡張するだけで容易にシステム構築ができる。これは、フレームワークそのものを、1つのアプリケーションと考えるとわかりやすい。アプリケーションは、ある程度はカスタマイズできるように実装されているものが大部分である。アプリケーションで実行できることは、設計段階で想定されているので、それを外れたカスタマイズは不可能だが、想定された範囲内においてのカスタマイズは極めて容易に行うことができる。同様に、設計段階で想定された適用範囲を外れてフレームワークを利用しようとすることは難しいが、想定された範囲を外れない限りでの利用は極めて容易となる。これがアプリケーションの性質である。

フレームワークは、ライブラリとしての性質も持っている。フレームワークを構築する段階で適用範囲を想定するため、事前に共通化すべき部品の想定や実装が可能になる。それらの部品をライブラリ化しておき、利用して実際に実装することで、フレームワークの有用性がいっそう高まる。これがライブラリの性質である。この性質により、同じフレームワークを利用して実際にシステムが構築された場合、フレームワーク構築の際には予想されていなかった共通化すべき部品を、フレームワークへフィードバックすることが可能になる。さらに、後に利用するシステムにおいて、より充実したライブラリを利用できるようになる。

(2) 制御の反転

フレームワークには、「フレームワークそのものが設計・実装の主であり、アプリケーションは従である」という、極めて重要な特徴がある。既存の手法では、アプリケーションそのものは、設計・実装の主であり、スケルトンなどが従となっているので、フレームワークはかなり違和感があって感覚的に摑みにくい部分であろう。このように、既存の手法とフレームワークとは制御が反転しているので、フレームワークを利用して構築するということは、フレームワークをメンテナンスして新規システムを構築することだというイメージを持っていた方が違和感が少ないだろう。

(3) ドメイン分析の必要性

前述のとおり、フレームワークで想定されている範囲を外れてアプリケーションを構築することは難しい。しかし、あえて範囲外の実装を行いたい場合は、アプリケーション全体が範囲内に収まる、別のフレームワークを利用することが必要である。この想定範囲、つまりドメインを分析することにより、フレームワーク適用の可能性を検証する必要性がある。しかし、逆に捉えると、類似したアプリケーションを構築する場合には、同じフレームワークを利用できるので、新規に構築するよりも格段に少ない工数で実現できる。

(4) オブジェクト指向との親和性

オブジェクト指向の概念とフレームワークは、親和性が高い。オブジェクト指向のカプセル化、継承、多様性のそれぞれが、フレームワークの抽象性の向上、柔軟性の増大、理解度の増加につながり、より良いフレームワークの構築に役立つ。よって、フレームワークを構築する上でオブジェクト指向が選択されるのは、ごく自然なことである。ただし、これはオブジェクト指向でなければフレームワークが構築できないということではない。

2.5 利点と欠点

当たり前のことだが、フレームワークは万能ではなく、利点もあれば、もちろん欠点もある。

最たる利点は「アプリケーションの開発・保守のコスト減少」である。フレームワークを利用することで、主要部分の実装とテストを省くことができ、あとは追加した部分の実装と機能のテストのみを行えばよくなる。つまり、構築コストや期間の削減を実現できる。また、アーキテクチャが標準化されるので、保守コストの削減にもつながる。さらに、新規に作成する部分が減るので、実装に際しての品質が向上し、テクニカル面でも容易になるので、構築に要する特別なスキルを必要としない。したがって、構築コスト、期間、保守コスト、要求スキルなどを総合的に縮小できるという、最大級の利点となる。

しかし、フレームワーク利用の利点が裏返って、そのまま欠点になってしまう場合もある。アプリケーションの設計は、フレームワークが想定しているアウトラインに則して行われるので、フレームワークを利用するには、まずフ

レームワークそのものが事前に完成している必要がある。その時点で、フレームワークが想定範囲外であった場合には、別のフレームワークを用意するか、フレームワークの利用そのものを諦めることになる。そこで、フレームワークそのものの種類を増やしていくか、またはフレームワークの柔軟性を強め、想定範囲を拡大するようにメンテナンスしていくことが重要となる。だが、フレームワークそのものの自由度、保守性、品質が高いレベルで要求されるということは、フレームワークそのものの構築やメンテナンスにも、より高いスキルを要求されるのである。

これでは、あまり利点がないようだが、それはフレームワーク単独で考えた場合であり、前述のとおり、トータルコストに注目すると、やはり多大なメリットがある。開発という作業を切り出して考えた場合、フレームワーク部分を同時に構築すると、スケルトン等を利用した通常の開発手法よりもコストが増加する。しかし、単一のフレームワークを数多く適用できれば、メンテナンスコストが発生しても、それ以上に開発工数やテスト工数などが削減できるため、結果的にはスケルトン等を利用するよりもトータルコストは減少するのである（図２参照）。

３．オリジナル・フレームワークについて

3.1　基本アーキテクチャ

オリジナル・フレームワークを構築するにあたって、基本アーキテクチャには、Web アプリケーションでは一般的になっている MVC-model 2 パターンを採用した。このアーキテクチャ・パターンでは、アプリケーションを中核機能とデータを扱うモデル（M）、情報をユーザーに表示するビュー（V）、ユーザーからの入力と処理の制御を扱うコントローラ（C）の３つのコンポーネントに分割する。今回、フレームワークを構築したサーバーサイドの Java 技術においては、モデルは Bean[*1]、ビューは JSP[*2]（JavaServer Pages）、コントローラは Servlet[*3]にマッピングされる。

3.2　目標

今回のフレームワークを構築するにあたり、以下の項目を達成目標とした。

①各オブジェクトの責務の明確化
②レイヤ化と、隣接するレイヤ間の結合を疎に保つ
③インタフェースと実装の分離の実践
④定型的・共通的な処理のフレームワークへの隠蔽

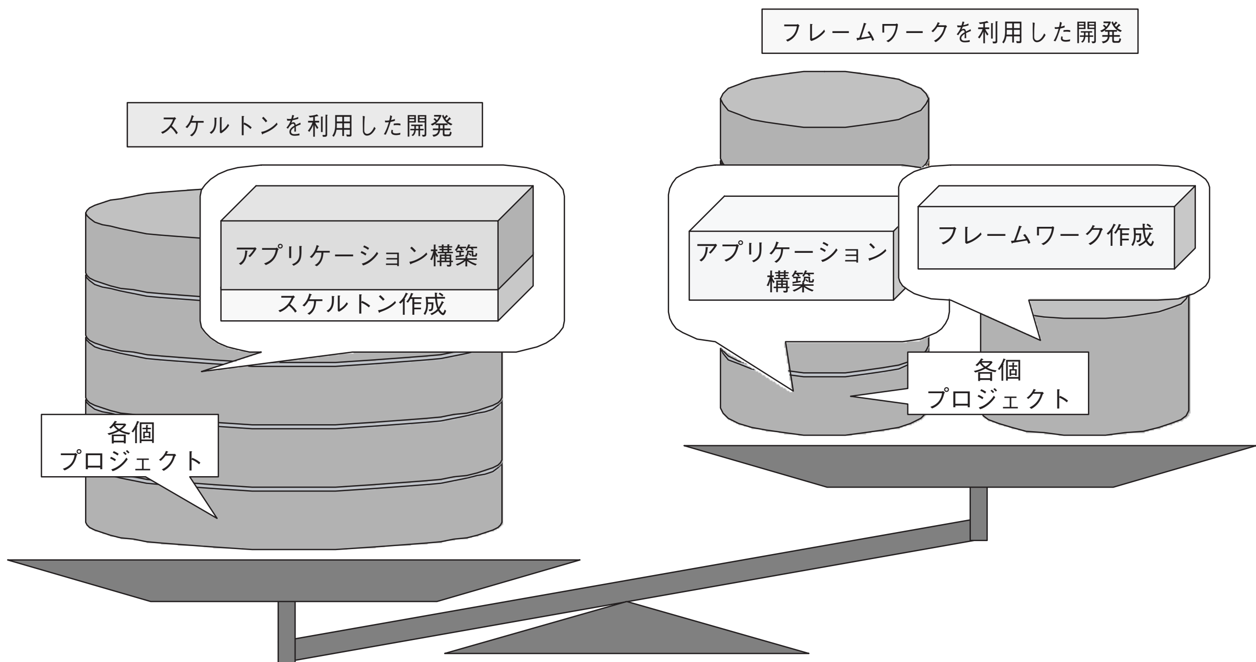

図２　コストの対比

＊１）Bean : Java Bean のこと。Java の拡張 API の一つで、Java 言語を使って再利用可能なソフトウエア部品を作成するためのフレームワーク。

＊２）JSP : Web アプリケーション構築のためのサーバーサイド Java 技術で、サーバー側で動的に Web ページを生成できるようにする。

＊３）Servlet : Web アプリケーションを支援するために策定されたフレームワークで、CGI と同じ動作と原理を持つが、Java を使っているのでマルチプラットフォームで動作する。

これらの目標を念頭におくことで、柔軟で発展性のあるフレームワークの構築を狙った。以降では、目標のそれぞれを、実際のフレームワーク構築において、どのように意識していたかを述べる。

3.3　フレームワークの構築

3.3.1　各オブジェクトの責務の明確化

MVC-model 2 アーキテクチャにおいても、役割の分割は行われている。しかし、業務アプリケーションを構築するにあたっては、MVC-model 2 による責務の分割では粒度が大きすぎる。フレームワークを業務アプリケーションの基盤とするためには、業務アプリケーションを記述するオブジェクトの粒度を、より細かく規定する必要がある。

今回は、その分割の目安として Web アプリケーションのトランザクション単位に着目した。1つのトランザクションは、比較的粒度の細かい処理のステップからなる。その処理のステップは、以下の4つの部品から構成される。

①ユーザーがシステムに要求とデータを送る
②システムが要求とデータを確認する
③システムがその内部状態を変更する
④システムがユーザーに結果を返す

これらの処理は、以下のようなオブジェクト群にマッピングされる。

・上記①で発行されたユーザーからのリクエストの低レベルの解析を行うオブジェクト
・リクエストの解析結果の妥当性と整合性を確認するオブジェクト
・妥当性検査の結果に基づいてデータベースへのアクセスを行うオブジェクト
・トランザクション処理の流れを制御するオブジェクト

ユーザーに対する結果の表示は MVC－model 2 アーキテクチャでのビューコンポーネントにあたる JSP を通じて行う。このとき JSP は、あくまでも、モデルに格納されている情報の表示を行うビューとしての役割に関連する処理のみが記述されるように配慮した。ビューにおいて必要となるロジックは、モデルにあたる Bean に隠蔽した。

3.3.2　レイヤ化と、隣接するレイヤ間の結合を疎に保つ

アプリケーションのレイヤ化は、アプリケーションを、その下位のレイヤであるアプリケーションサーバーやデータベースサーバーの変更から受ける影響を低減するという観点において重要である。アプリケーション構築の基盤となるフレームワークは、自身の上に構築されるアプリケーションを、こういった下位レイヤの変更による影響から守るための支援を行う。また、レイヤ化したアプリケーション構造の標準化も支援する。そのためには、フレームワークそのものもまたレイヤ化されている必要がある。影響を低減するためには、レイヤ化するとともに、隣接するレイヤ間のインタフェース仕様を定めて、レイヤ間通信が行われるポイントにおける相互依存性を疎に保つ必要がある。また、レイヤ間の結合を疎に保つことは、各レイヤ間での単体テストや動作検証を容易にするという利点もある。

今回のフレームワークでは、J 2 EE[*4]（Java 2 Platform, Enterprise Edition）Blue Print に述べられている、クライアント層、プレゼンテーション層、ビジネスロジック層、インテグレーション層、リソース層といった考え方を参考に、レイヤ分けを行った。今回のフレームワーク構築では、EJB[*5]（Enterprise JavaBeans）は利用していないが、J 2 EE Blue Print や J 2 EE Patterns は、フレームワークの設計にあたって大いに参考になった。

3.3.3　インタフェースと実装の分離の実践

レイヤ間の結合を疎に保つことにも関連するが、レイヤ間のみならず、各レイヤ内の細かい粒度に分割されたオブジェクト同士の通信に関してもインタフェースを定義し、コンポーネント使用に関する仕様を定めたインタフェースと、その仕様を実現する実装とを分離することに努めた。これにより、フレームワーク内部やフレームワークを利用する側へ必要な情報だけを知らせることができ、情報隠蔽と利用者側にとって理解しやすい API[*6]（Application Programming Interface）の提供を実現できる。また、実装がインタフェースから独立することで、実装を変更する際の容易性も確保できる。

3.3.4　定型的・共通的な処理のフレームワークへの隠蔽

実際の開発プロジェクトにおいて、どのプロジェクトにも出現する、必要な処理が存在する。そのような共通的な処理は、ライブラリとして提供することが望ましい。ユーザーインタフェースとの入出力の際に利用されるフィルタや、各種データのフォーマット、入力値の検査ロジックなどが、これに該当する。また定型的な処理は、その流れも含めてフレームワーク化する。画面遷移、ユーザー認証、データベースアクセス、エラーや例外の処理、ロギング、キャッシング等が、これに該当する。こうした処理は、フレームワークの機能として取り込み、利用者側に対しては透過的な API や設定ファイルへの記述を提供する。これにより、各アプリケーションの開発者は業務ロジックの設計・実装に注力できるようになる。

＊4）J 2 EE : Sun Microsystems 社のテクノロジーによる、多層エンタープライズアプリケーションを開発するための仕様のセット。
＊5）EJB：再利用可能なコンポーネントとして多層アプリケーションのビジネスロジックを実装するための仕様
＊6）API : OS やプログラムの機能を利用するための接続仕様。

3.4　構築の成果と課題

今回のフレームワーク構築で、フレームワークによるオブジェクトの責務とレイヤの明確化がなされ、フレームワークの上に構築されるアプリケーション構成の見通しをよくした。これが今回の構築において、最も顕著な成果であった。

その一方で、残された課題も多い。今回のフレームワーク構築に投入できたリソースは、時間的にも人的にも、非常に制限されてしまい、適用対象の案件も１件のみであった。このことから、抽象化や拡張性といった部分は、プロジェクトの現実的なスケジュールを前にして犠牲にされがちであった。

機能的な面においても、定型的な処理をフレームワークに「隠蔽」するレベルにまでは達していない。特に画面遷移やロギング、データのキャッシング等には設計レベルから、多くの改善余地が残されている。今後、ユーザー認証やデータベースアクセスに関しても適用可能なプロジェクトを増やしていくために、可搬性や拡張性を考慮した設計・実装へと洗練させていく必要がある。

共通処理をまとめたコンポーネント部品についても、粒度の細かい再利用性の高いものを、より多く収録していき、ライブラリを充実させる必要がある。

全体を通しては、コンポーネント化されたオブジェクトを組み合わせるコンポジションよりも、継承を利用した安易な機能拡張が多く見られた。コードの拡張性や変更の容易性について将来性を考慮すると、継承だけではなく、コンポジションも積極的に利用していく必要がある。この場合、デザインパターンを適用するなどして、当初の設計を大幅に見直すことになるが、今回は、きちんとした単体テストを用意できなかったため、リファクタリング[*7]が行えなかった。フレームワークに限った話ではないが、拡張や保守といった継続的なメンテナンス作業が発生するコード群については、その作業を通じて設計を洗練させていくためにも、自動化された単体テストを揃えながら作業を行っていくことが重要である。

４．おわりに

昨今、アプリケーションを構築するためだけに要求されるスキルが日々増大している反面、顧客の条件が厳しくなっている。なんらかの対応策を考慮する必要があり、その答えの１つがフレームワークではないだろうか。ただし、フレームワークとは直接関係しないが、見過ごせない問題が２つ残されている。

⑴　**技術者の二極化**

フレームワークを使う者と作る者に分かれてしまうという、技術者の二極化だ。フレームワークを使う者は、あくまでもフレームワーク・ユーザーであって、フレームワークが充実すればするほど、フレームワーク以外の知識を得る機会が減少してしまう。その一方で、フレームワークを構築する者は、よりいっそうスキルを高めていく必要がある。経験と知識が乏しい初心者にとって、フレームワーク・ユーザーになるのには初期環境として適しているが、ステップアップしてフレームワーク構築メンバーへになるためのスキルとのギャップが、中間ステップが存在しないために、とても大きくなってしまう。ステップアップのための教育手法という問題についても、既存とは違った解決方法が必要になる。

⑵　**設計の品質**

既存の構築においては、開発期間の問題や明確な基準が存在しない等の理由で、設計段階で行っておくべき事項が満たされないケースが存在した。その場合、不足した事項を省略することができず、止むを得ず実装段階に持ち越されてしまっていた。これに対し、フレームワークを利用する場合は、事前条件が明確に決まるため、実装段階へ持ち越すことができなくなる。これは、工程の区切りが明確になるので利点であるとも言える。

容易に想定できる問題点を挙げたが、それ以外にも種々の問題の発生が予想される。しかし、そうした問題を一つひとつ解決していかなければ、前進は有り得ない。フレームワークとは、やはり銀の弾になるものではなく、単なるオブジェクト指向に適した手法の１つでしかないが、有効なツールであることに疑問の余地はない。

〈参考文献〉

１．戸松豊和著：『増補改訂 Java プログラミングデザイン』，ソフトバンクパブリッシング（1998）

２．F. ブッシュマン他著：『ソフトウェアアーキテクチャ ソフトウェア開発のためのパターン体系』，J2EE Blue Prints

＊7）リファクタリング：動作しているシステムの外部的な振る舞いを変更せずに、実装コードを抽象度の高い設計に基づいたものに洗練していくプロセス。