開発体制を考慮したソフトウェアアーキテクチャ設計技術

Software architecture design considering the development system

永井 恒一郎*
Koichiro Nagai

近年、ソフトウェアの大規模化、複雑化に伴い、ソフトウェアアーキテクチャの重要性が増している。しかし、システム要件を十分に実現可能なソフトウェアアーキテクチャを設計しても、実装の結果、十分な成果を得られないことがある。この原因のひとつとして、設計段階で開発体制を考慮していないことが考えられる。そこで、本稿では開発体制を考慮したソフトウェアアーキテクチャ設計の実例を示し、開発体制を考慮したソフトウェアアーキテクチャ設計の重要性を記述する。

In recent years, the importance of software architecture has increased as a result of increase in size and complexity of the software. However, even if the software architecture is designed to fully fulfill the system requirements, the result of implementation may not be sufficient. One of the reasons for this is that the software architecture is designed without considering the development system. Therefore, in this article, I will describe an example of the software architecture design considering the development system and the importance of the software architecture design considering the development system.

1. まえがき

ソフトウェアアーキテクチャはソフトウェアの構成要素（コンポーネント）とその相互関係を定義する。そのソフトウェアアーキテクチャが定めた枠内で、詳細な設計・実装が進められる。そのため、ソフトウェアアーキテクチャが不適切な場合、ソフトウェアは無秩序に開発され、ソースコードには多くの重複が生まれ、密に結合し、複雑なソフトウェアとなる。したがって、大規模なソフトウェアほど、ソフトウェアーキテクチャの重要性が増す。

ソフトウェアアーキテクチャの影響が特に大きいのは非機能要件の実現といわれる。しかし、ソフトウェアアーキテクチャは非機能要件の実現だけでなく、ソフトウェアのライフサイクル全体に影響がある。例えば、開発プロセスの選択がある。開発中の機能の変更がリリース済みの機能に影響を与えるソフトウェアアーキテクチャの場合、機能をリリースする度に、リリース済みの機能への影響確認とリグレッションテストが必要となるため、コスト・期間が増大する。その結果、開発プロセスにアジャイル開発を選択するのは不可となる。

このように、ソフトウェアの開発においてソフトウェアアーキテクチャは重要な役割がある。しかし、システムの要件に基づき、一般的なアーキテクチャパターンや技法、様々なツールを用いてソフトウェアアーキテクチャを設計しても、コストの増大や、品質の低下など、計画よりも効果が出ないことがある。これは、後段の開発フェーズで、当初のアーキテクチャ設計とは異なる実装（いわゆるアーキテクチャ崩れ）や、設計どおりに進めようとしても、設計意図や手法の理解に期間を要してしまうことに起因する。

この原因のひとつとして、開発体制を考慮せずにソフトウェアアーキテクチャを設計することがある。もちろん、アジャイル開発などイテレーティブな開発プロセスを選択することや、十分な教育期間の確保、設計ガイドラインの整備なども対策として有効である。しかし、プロジェクトの規模やコスト、契約内容によって、これらの対策を実施できない場合がある。そのため、十分に開発体制を考慮し、その体制で実現可能なソフトウェアアーキテクチャを設計することが重要である。

そこで、本稿では、開発体制を考慮しない場合と考慮した場合のソフトウェアアーキテクチャ設計の実例を示し、開発体制を考慮することの重要性を記述する。

2. 開発システムの概要・特徴

2.1　システムの概要・特徴

論述対象のシステム（以降、本システムと称す）は、

＊鎌倉事業部 第三技術部

ユーザ組織外のシステム（以降、他システムと称す）やインターネットなどから情報を収集し、収集した情報をDB、ファイルサーバ、ビッグデータサーバなどに保存、ブラウザ上に表示する。その後、ユーザは表示された情報を分析し、業務に活用する。また、本システムが蓄積した情報は、他システムに配布する。

このように、本システムは、情報収集、配布、分析を主に行うシステムである。

本システムの主な特徴は以下の点である。

(1) 約 10 の他システムと通信する。各システムとの接続は、FTP、TCP/IP ベースの独自プロトコル、CIFS、SAMBA など、多岐にわたる。
(2) データの保存先は、DB だけでなく、ファイルサーバ、ビッグデータサーバがある。
(3) 情報収集、分析のため、市販ソフトウェアとの通信が必要である。
(4) 約 200 のサーバで構成される。また、約 250 の機能がある。
(5) ユーザインタフェースの主な処理は、開発済みの自社製ライブラリを使用する。

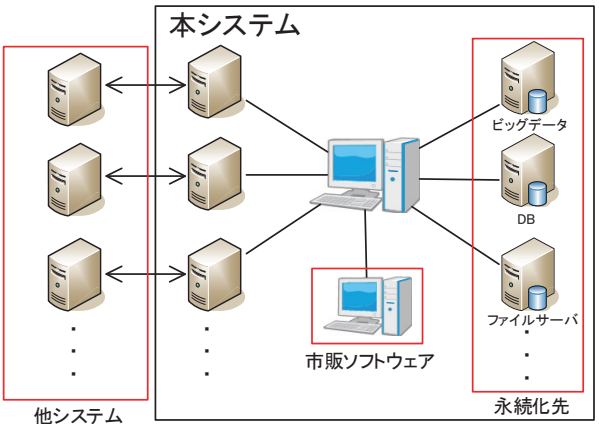

図 1　システムの特徴

2.2　プロジェクトの概要・特徴

本プロジェクトは、複数のシステムで提供していた機能を 1 つのシステムに統合する新規システム開発である。その主な特徴はステークホルダにある（図 2 参照）。

まず、プロジェクト外部に関する特徴としては、本システムは多数の部署で使用され、本システムで提供する機能は、部署間で重複して使用するものや、特定の部署独自に使用するものがある点である。加えて、各部署の所在地は日本全国に広がる。また、前述のとおり、本システムは他システム、市販ソフトウェアが多数であり、それぞれのベンダーとインタフェース、機能に関する仕様の調整が必要となる。ステークホルダが多岐にわたるため、ステークホルダへのヒアリングやステークホルダ間の要求事項の調整量が増加する。その結果、本プロジェクトでは機能の仕様調整に期間を要する。

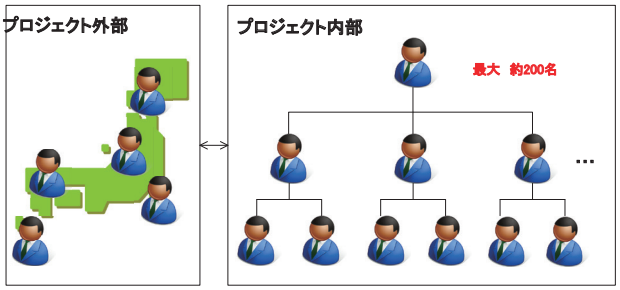

図 2　プロジェクトの特徴

また、プロジェクト内部に関する特徴としては、規模に対して開発期間が短いことから、最大で、約 200 名が本プロジェクトに従事する点である。そのため、各メンバの開発プロセス、及び経験の差異から、開発プロセスの統一に期間を要することや、プロジェクトメンバが多いことによるスキルのばらつきが大きくなる。

3. ソフトウェアアーキテクチャの課題

3.1　システムの特徴に基づく主な課題

2.1 節に記載したとおり、本システムは多様な情報を収集し、分析し、配布することに特徴がある。したがって、収集する情報を増やすことがシステムの付加価値を高めることに繋がる。また、分析においても市販ソフトウェアの積極的活用により、分析可能な範囲の拡大が見込まれる。そのため、他システム、市販ソフトウェア、及び永続化先の追加・変更を容易にすることが、本システムのソフトウェアアーキテクチャを実現する上で最も重要な課題である。

加えて、ユーザインタフェースは使用性の品質に大きく関わることから、ユーザインタフェースの主要な機能を提供する自社製ライブラリとの親和性も重要な課題である。

3.2　プロジェクトの特徴に基づく主な課題

2.2 節に記載したとおり、本プロジェクトは多数のステークホルダが関連していることから、他システムとのインタフェースや、機能ごとの仕様が確定する時期に差異が発生する。そのため、仮決定段階で設計を進め、決定後に変更を反映するというように、仕様変更が多発することが予想される。この変更によるスケジュールへの影響を低減するため、仕様変更に対して変更が容易でなければならない。

また、本プロジェクトでは、最大で 200 名ものプロジェクトメンバが同一システムを開発する。それぞれの開発経験が異なることから、そのスキル差による品質のばらつきを抑止する必要がある。すなわち、「誰がつくっても、同じような設計・製造」が可能であり、習熟が容易なソフトウェアアーキテクチャとすることが重要な課題となる。

4. ソフトウェアアーキテクチャ設計

4.1　開発体制を考慮しないアーキテクチャ設計

開発体制を考慮しない場合、前節記載の課題を解決するためには、ドメイン駆動設計をベースとしたソフトウェアアーキテクチャが適切と考える（図３①参照）。

ドメイン駆動設計は、システムのドメインをドメインモデルとして設計することにより、仕様から設計までをシームレスに繋げることができる。その結果、仕様変更時に影響範囲を特定しやすくなり、変更に強くなる。

また、ソフトウェアアーキテクチャの各レイヤと設計文書を関連付けて設計することにより、仕様変更発生時に、アーキテクチャ上のどの部位を変更すべきかを特定しやすくなる（図３②参照）。例えば、他システムとのインタフェース仕様書に変更があれば、インフラストラクチャ層を変更する、というように、設計書とレイヤの対応付けが可能となる。

なお、本システムでは、ソフトウェアアーキテクチャのベースとなる Framework に Spring Framework を、

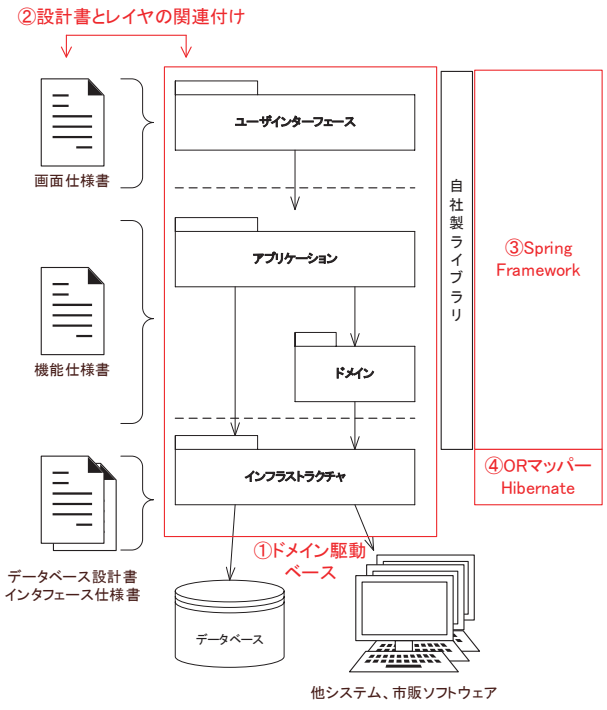

図３　開発体制を考慮しないソフトウェアアーキテクチャ

OR マッパーに Hibernate を採用する。（図３③④参照）。Spring Framework は、Web アプリケーションにおいて、普及率の高い Framework であることから、開発メンバの習熟が短時間で可能であることが期待できる。加えて、Spring Framework は、アノテーションの機能が充実しており、関連付けに設定ファイルが不要であることから、レイヤ間のオブジェクトの関連付けが簡略化できる。

また、OR マッパーである Hibernate は、テーブルの構造とオブジェクト間のマッピングに関して自由度が高く、ドメイン駆動設計との親和性が高い。このことから、Spring Framework と Hibernate を採用する。

4.2　開発体制を考慮したアーキテクチャ設計

4.2.1　ドメイン駆動設計の懸念点

前節で述べたとおり、本システムの要件を実現する上で、ドメイン駆動設計をベースとしたソフトウェアアーキテクチャが適すると考える。

しかし、ドメイン駆動設計はドメインモデルの設計品質によってその効果が大きく左右される。すなわち、システムのドメインから適切なドメインモデルを設計できない場合、その効果を期待できない。ドメインモデルを設計する場合、ドメイン知識が豊富なドメインエキスパートと開発者との密接な連携、及びオブジェクトモデリングの豊富な知識が要求される。

これに対して、本プロジェクトでは、最大 200 名もの開発メンバが参画するため、スキルの統一やドメインエキスパートとのコミュニケーションロスにより、適切なドメインモデルの設計が困難と推測する。

この問題を開発プロセスで対処する場合、アジャイル開発やスパイラル開発を行い、徐々に開発を進めることで、プロジェクトメンバの習熟度の向上やドメインモデルをリファクタリングしていくことができる。ところが、本プロジェクトでは顧客の開発プロセスに準拠して、ウォータフォールで開発することが必須であり、アジャイル開発やスパイラル開発を行うことができない。

以上の点から、開発体制を考慮すると、本プロジェクトをドメイン駆動設計ベースで進めることは、ドメイン駆動設計のメリットを活かせず、開発期間、品質へのリスクが高い。

4.2.2　トランザクションスクリプトの採用

開発体制、開発プロセスを考慮した結果、本プロジェクトでは、ドメイン駆動設計ベースのソフトウェアアーキテクチャではなく、トランザクションスクリプトを採用する（図４①参照）。

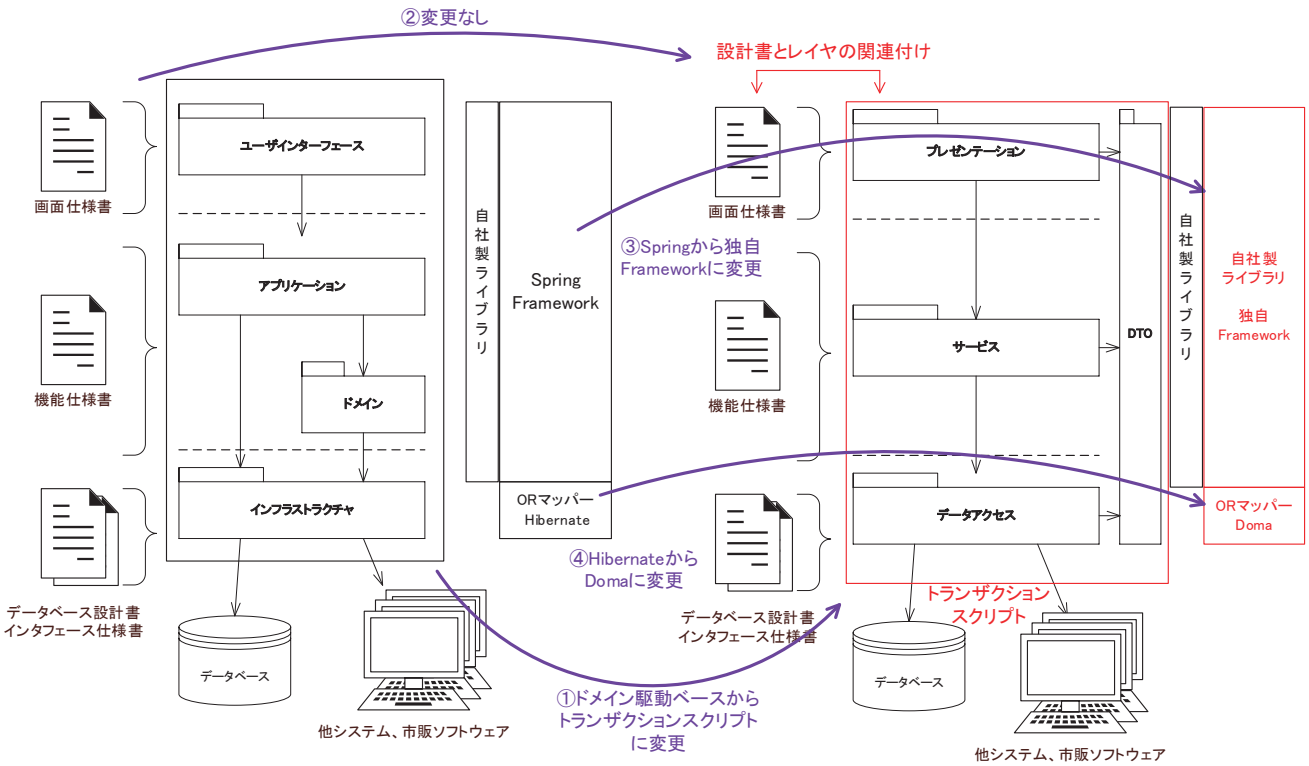

図４　開発体制を考慮したソフトウェアアーキテクチャ

トランザクションスクリプトは、ユーザの操作に関連する一連の処理（つまりトランザクション）単位で、Service 層のオブジェクトやメソッドに、機能を実装する。例えば、画面の保存ボタンを押下した際の処理を、Service 層の 1 メソッドに対応付け、メソッド内で DB への保存を実装する。ユーザの 1 操作ごとに実装が可能であるため、作業の分担が容易であり、作業効率の観点から、多数のプロジェクトメンバが従事する本プロジェクトに適していると考える。また、プロジェクトメンバにトランザクションスクリプトの経験者が多く、短期間での立ち上がりが期待できる。

以上の理由により、本プロジェクトではトランザクションスクリプトをソフトウェアウェアアーキテクチャのベースとする。ただし、仕様変更の対応を考慮し、設計文書と各レイヤの関連付けの方針は維持する（図４②参照）。

加えて、ベースの Framework に関して、本プロジェクトでは自社製ライブラリを使用するが、開発期間を考慮すると Spring Framework との併用にリスクが高い。そのため、自社製ライブラリで使用している独自 Framework を使用する（図４③参照）。Spring Framework と比較して習熟期間を要するが、自社製ライブラリの開発メンバと本プロジェクトで人材の流動性が高まることから、期間、コストへの影響は小さいと考える。

また、OR マッパーについても、Hibernate ではなく Doma を使用する。Hibernate は OR マッピングの自由度の高さからドメイン駆動設計には有用だが、機能が多く、学習コストが高い。トランザクションスクリプトの場合、Hibernate ほどの OR マッピングの自由度は不要であることから、より学習コストが低く、自社製ライブラリとの親和性が高いことから Doma を採用する（図４④参照）。

4.2.3　変更容易性の担保

ドメイン駆動設計、又はトランザクションスクリプトのいずれの場合でも、本プロジェクトにおいて最も懸念される点が、永続化先、他システム、及び市販ソフトウェアの追加・変更に対する変更容易性の担保である。3.1 節で述べたとおり、この変更容易性の実現が本システムの付加価値を左右する。そのため、ソフトウェアアーキテクチャを設計する上で必須の要件となる。

そこで、データアクセス層において、GoF のデザインパターンのうち Facade パターンを採用することでこの変更容易性を担保する。具体的な設計方針は以下のとおりである。

(1)　永続化先、他システム、及び市販ソフトウェアごとに Facade クラスを 1 対 1 で定義する。

(2) 通信を行うための DTO のシリアライズ、及びデシリアライズは、Facade クラスを境界として、データアクセス層で完全に隠蔽する。

(3) 永続化に関する Facade クラスのインタフェースは CRUD を実現するため、基本的に create、read、update、delete の 4 種類とする。

(4) 他システム、市販ソフトウェアに関するインタフェースはデータの送受信を実現するため、基本的に send、receive の 2 種類とする。

(5) Facade クラスのインタフェースの IN/OUT は、DTO の抽象クラスとし、Service 層は DTO の抽象クラスから派生した DTO で Facade クラスにデータを受け渡す。

設計方針 (1) (2) により、永続化先、他システム、及び市販ソフトウェアの追加に対しては「Facade クラスを追加する」。また、変更に対しては、該当する「Facade クラスを変更する」というように、影響範囲を限定することで変更容易性を担保することができる（図 5 参照）。

加えて、設計方針 (3) ～ (5) により、Facade の使用方法を統一できるため、サービス層は、通信先が異なっても「DTO を生成し、Facade クラスのメソッドを呼び出す」という実装に統一される。その結果、開発者のスキル差が実装に反映されにくくなる。すなわち、「誰がつくっても、同じような設計・製造」が可能なソフトウェアアーキテクチャとなる。図 6 にサービス層の疑似コードを示す。

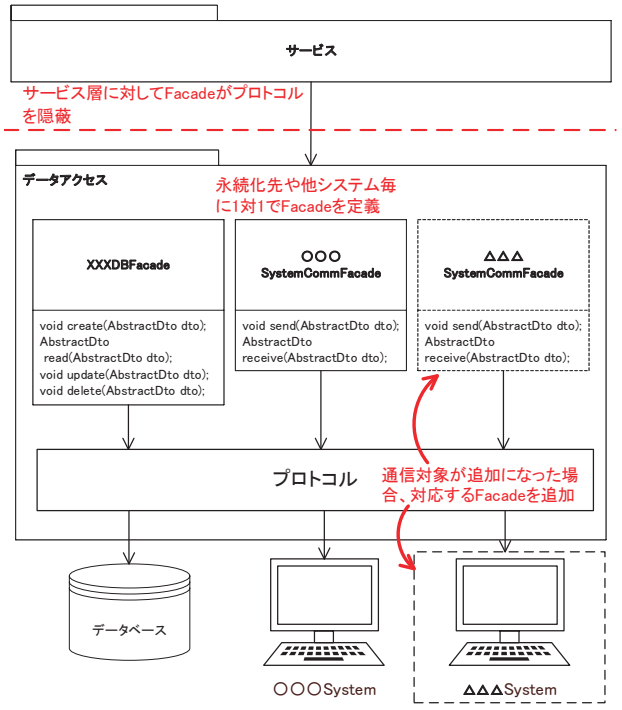

図 5　Facade パターンによる変更容易性の担保

```
void func() {
XXXDto dto = new XXXDto();

// dtoに対して値を設定

XXXDBFacade facade = new XXXDBFacade();
facade.update(dto);
}
```

通信対象や永続化先が異なる場合は、このFacadeを置き換えればよい

図 6　サービス層疑似コード

5. 効果と今後の課題

ドメイン駆動設計ベースのソフトウェアアーキテクチャとした場合、プロジェクトメンバ全員への設計方法の習熟に期間を要し、設計や実装、試験の期間を圧迫したと予想する。対して、トランザクションスクリプトベースのソフトウェアアーキテクチャを採用したことにより、そのメリットである作業分担の容易さを活かし、本システムはユーザの業務に重大な影響を与えることなく、予定どおりに運用を開始することができた。

ただし、トランザクションスクリプトのデメリットである Service 層に重複が生まれやすいという点については解消できず、多くの箇所で重複が見られる。この点については、Service 層の構造を見直す必要があり、今後、継続してリファクタリングする。また、4.2.3 項で述べた変更容易性についても、今後の永続化先、通信対象の追加・変更時に、その効果を確認する。

6. むすび

本稿では、開発体制を考慮しないソフトウェアアーキテクチャ設計と開発体制を考慮したソフトウェアアーキテクチャ設計の実例を示した。しかし、仮に本稿と同様の要件のシステムであっても、開発体制や開発期間が異なれば、本稿で示したソフトウェアアーキテクチャと異なる設計となる。

すなわち、システムの規模、期間、開発体制とそのメンバを考慮し、実現可能なソフトウェアアーキテクチャを設計すること。また、複数のソフトウェアアーキテクチャ案から、開発体制と要件のトレードオフを十分に検討することで、プロジェクトの QCD の向上に寄与することができる。

最後に、本ソフトウェアアーキテクチャの開発にあたり、短期間に度々の方針変更があったにもかかわらず、本アーキテクチャの実現に尽力いただいたアーキテクチャ開発チームメンバに改めて感謝いたします。

参考文献

(1) Evans，E.：エリック・エヴァンスのドメイン駆動設計，翔泳社（2011）

(2) Fowler，M.：エンタープライズアプリケーションアーキテクチャパターン，翔泳社（2005）

(3) Lattanze，A. J.：アーキテクチャ中心設計手法，翔泳社（2011）

(4) Bass，L.，Clements，P.，Kazman，R.：実践 ソフトウェア アーキテクチャ，日刊工業新聞社（2005）

(5) Rosenberg，D.，Stephens，M.：ユースケース駆動開発実践ガイド，翔泳社（2007）

(6) 藤原啓一：要求から詳細設計までをシームレスに行うアジャイル開発手法，MSS 技報，24（2014）
https://www.mss.co.jp/technology/report/pdf/24-04.pdf

執筆者紹介

永井 恒一郎

2004 年入社。鎌倉事業部所属。関西事業部にて、カーナビ、医用画像、衛星通信、海底ケーブル、業務系システムのソフトウェア開発業務を経て、2018 年から鎌倉事業部において、防衛分野のソフトウェア開発業務に従事。