2009 年度　卒業論文

アスペクト指向を用いた
リッチインターネットアプリケーションの
変更要求モジュール化に関する研究

2010 年 2 月 2 日(火)提出

指導：鷲崎 弘宜 准教授

早稲田大学 理工学部

コンピュータネットワーク工学科 鷲崎研究室

学籍番号：1G06R196-4

村上 真一

目次

第 1 章 はじめに

近年、クラウドコンピューティングや SOA への注目を背景に、サーバーで生成した HTML をクライアント側に送信する従来の Web アプリケーションに代わって、Ajax[1]や Flash[2]といった技術を利用し非同期通信を行うリッチインターネットアプリケーション（Rich Internet Application：RIA）の需要が高まっている。また、要求の変更に素早く正確に対応できることが一層強く求められてきている。しかし一方で、RIA の設計は従来の Web アプリケーションより複雑になっており、変更が必要な箇所はプラットフォームを越えて横断的に拡散することがあるため、変更要求への対応は必ずしも容易ではない。

実装が設計に横断して必要となる機能のことを、横断的関心事と呼ぶ。横断的関心事の実装をモジュール化する技術として、アスペクト指向プログラミング（Aspect-Oriented Programming：AOP）[3]がある。アスペクト指向プログラミングは、処理とその処理があるべき箇所の記述を組にした上で、アスペクトと呼ばれるモジュールに実装し、ウィーバと呼ばれる合成器で後から指定された箇所に自動で織り込む技術である。AOP により、本来モジュール群へと横断的に散らばる処理を局所化することができる。また、プログラミングの手法である AOP を発展させ、従来のオブジェクト指向ソフトウェア開発では実装が設計を横断してしまう機能を、一つの側面としてまとめて扱う手法として、アスペクト指向ソフトウェア開発（Aspect-Oriented Software Development：AOSD）があり、変更要求への対応に必要な実装を、一括管理することができる。

しかし、従来の AOP、AOSD によるアプローチは、Java や JavaScript など単一プラットフォーム内での開発を前提としており、RIA の、クライアント、サーバー、外部サービス、DB と複数のプラットフォームにまたがる設計に対応することはできなかった。

また、AOP、AOSD は、必要となる変更箇所を単一のモジュールにまとめることはできるが、具体的に必要な変更箇所を特定する方法が含まれていないので、変更要求に素早く正確に対応するための解決方法として十分ではない。

本研究では、RIA への変更要求に素早く正確に対応するために、変更要求への対応に必要な実装がアーキテクチャを横断する場合において、アスペクトを含む、変更に必要となる具体的なモジュール群の生成方法を定型化してまとめ、あらかじめパターンとして整理しておく方法を提案する。

第 2 章 RIA の登場背景と問題点

2.1. 従来の Web アプリケーションと MVC アーキテクチャ

Web アプリケーションとは、Web の技術を利用して構築されたアプリケーションソフトウェアの総称である。典型的な Web アプリケーションのアーキテクチャを図 2.1.に示す。図 2.1.に示される Web アプリケーションでは、通信プロトコルに HTTP を用い、クライアントは Web ブラウザで Web サーバーにアクセスし、必要な情報や処理を要求する仕組みになっている。

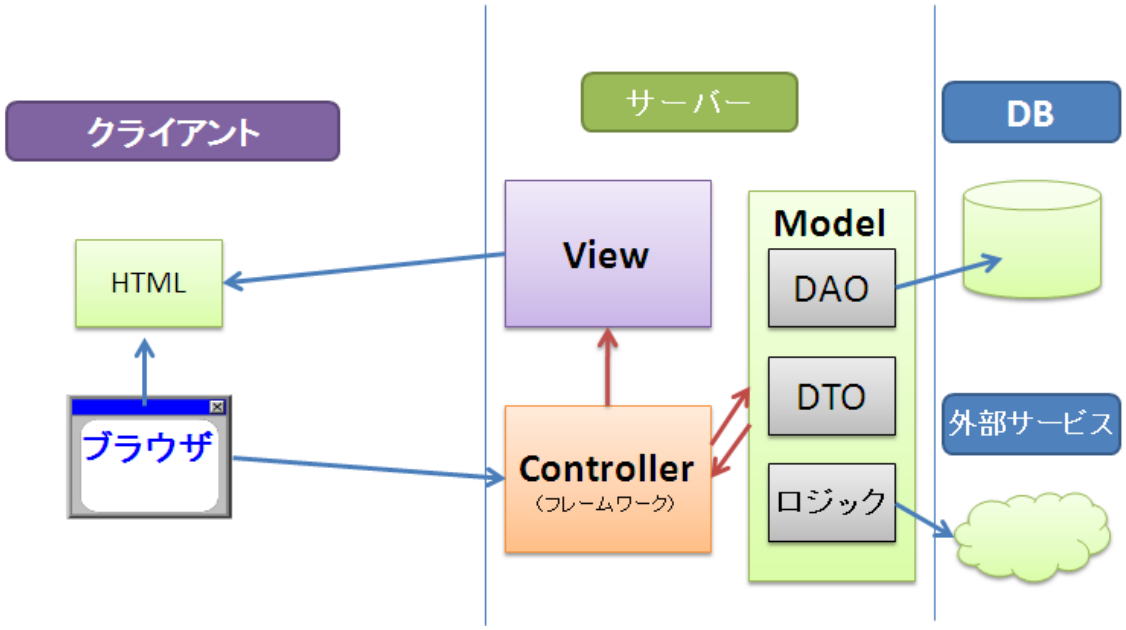

図 2.1. 従来の Web アプリケーションのアーキテクチャ

Web アプリケーションの設計は MVC アーキテクチャに基づいて行われることが多い。MVC アーキテクチャとは、Model、View、Controller の 3 つに責務を分けるアーキテクチャのことである。図 2.1.に示される Web アプリケーションを例に、Model、View、Controller のそれぞれの責務を以下に説明する。

Model

Model は、アプリケーションが使用するデータとロジックである。Model はサーバー内で使用するデータやロジックのクラスのみならず、Data Base Management System（DBMS）に格納するデータを扱う Data Transfer Object

（DTO）や、データベースとの接続を行う Data Access Object（DAO）、Web API を用いて外部サービスを使用するロジッククラスなども含まれる。

View

View は、Controller からの起動を受けて、クライアントが表示する HTML を作成する処理である。

Controller

Controller は、クライアントからのイベントを受けて、対応する Model のロジックと View の起動に振り替える、橋渡し的な処理である。実際の開発においては、Java EE[4]や Ruby on Rails[5]といったコンポーネント、フレームワークによって Controller の生成は自動化されることが多く、Model の変更に伴い手動で Controller を修正する必要はほとんどない。

従来の Web アプリケーションにおいては、クライアントが表示する HTML の生成をすべてサーバーが請け負っている。表示に関しては、クライアントはサーバーから受け取った HTML を表示するのみとなっている。

2.2. Ajax を利用した典型的な RIA のアーキテクチャ

RIA も Web アプリケーションの一種である。しかし RIA は、リッチなユーザーインターフェースを実現するために、クライアント内での簡単な処理、非同期な通信と再描画による HTML の更新といった、従来の Web アプリケーションでは不可能であった機能を持っている。それらの機能を実現するために、従来の Web アプリケーションでは見られなかった技術やアーキテクチャが使われている。典型的な RIA のアーキテクチャを図 2.2.に示す。図 2.2.に示される RIA では、Ajax（Asynchronous JavaScript + XML）を用いている。またクライアント内において、サーバーとは独立した MVC アーキテクチャが構築されている。以下にそれぞれの説明を行う。

Ajax

Ajax は、Web ブラウザに実装されている JavaScript の XMLHttpRequest という機能によって、HTTP 通信と XML を使ってページ遷移と非同期に通信を行い、通信結果を元に HTML をダイナミックに更新することで、ページ遷移を伴わずにページの一部を更新する仕組みのことである。

従来の Web アプリケーションにおいては、表示する HTML の生成をサーバーの View がすべて請け負っていたため、通信と再描画はページ遷移に伴う必要があり、レスポンスの悪さを主とするユーザーインターフェース上の問題となっていた。

Ajax を利用することで、ユーザーは従来の GUI アプリケーションに近い、リッチなユーザーインターフェースを得ることができる。

クライアント内 MVC

クライアント内に新たに構築した MVC を用意するということは、クライアント内にプログラムを設置するということに他ならない。従来の Web アプリケーションのアーキテクチャでは、クライアントはサーバーの View が生成した HTML を、Web ブラウザの機能を使って表示するだけにすぎなかった。一方で RIA は、従来の Web アプリケーションが持つサーバーの View によるページ遷移を伴った HTML の更新を廃し、HTML の更新処理をクライアント内の View に一任している。RIA のクライアント内における MVC は、クライアントに専用のクライアントプログラムをインストールさせることなく、クライアントサーバー型の GUI ネットワークアプリケーションに近いユーザーインターフェースを提供することを可能にしている。クライアント内 MVC の各責務を以下に説明する。

Model

Model は、クライアント内で扱われるデータとロジックである。クライアント内で動的に更新されるデータの実体を Model としてクライアント内に保持する。また、クライアント内で完結する簡単なロジックを Model としてクライアント内に保持する。こうすることによって、クライアントはサーバーと通信の必要がない処理については、独自に実行することが可能になる。

View

View は、クライアント内の Model のデータを動的に描画する処理である。クライアント内に View を設置することによって、再描画のために必ずサーバーと通信をし、新たな HTML を受け取らねばならないという制約を回避している。

サーバーのロジックを使う必要がないような簡単なロジックと再描画の処理をクライアント内で完結させることにより、レスポンス性の高いリッチなユーザーインターフェースを提供できる。また、サーバーのロジックを使う必要のある処理においても、Ajax による非同期通信を行い、結果をクライア

ント内の View で動的に更新することにより、ページ遷移を伴わない素早い反映が可能となっている。

Controller

Controller は、クライアント内で発生したイベントを、適切な処理に振り替える処理である。振り替えられる処理は、クライアント内の Model のロジック、クライアント内の View のみでなく、サーバーの Controller を含む。サーバーの Controller への振り替えを含むことで、サーバーのロジックを非同期に利用することができる他、ページ遷移のような従来の Web アプリケーションに可能な処理を網羅することができている。

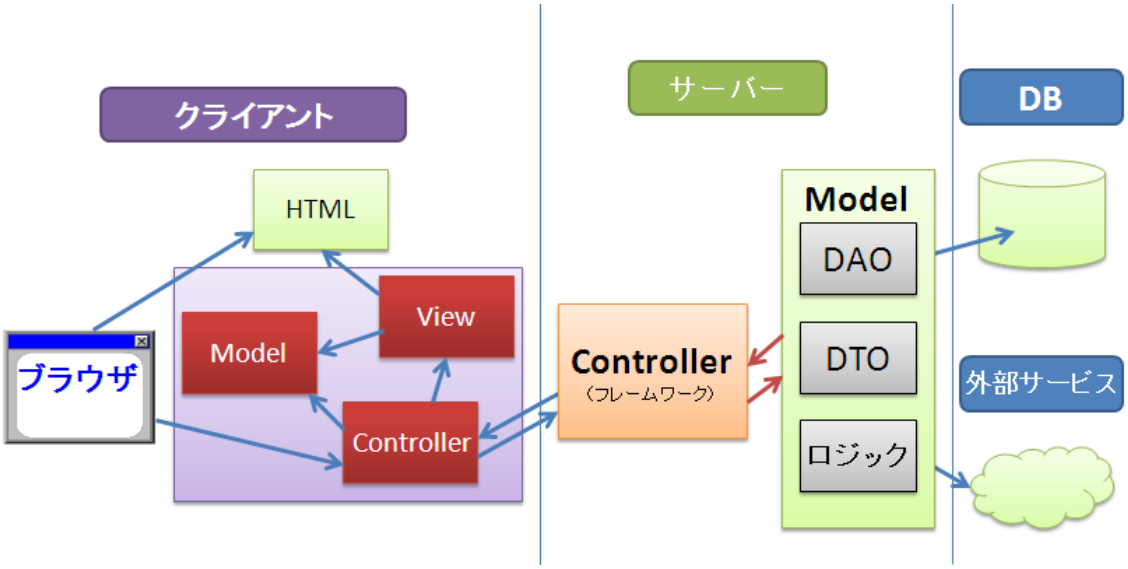

図 2.2. RIA のアーキテクチャ

2.3. RIA への変更要求における問題点

Web アプリケーションとしてサーバーのロジック、DB、外部サービスを利用しつつ、高度なユーザーインターフェースを提供する RIA は、クラウドコンピューティングや SOA への注目を背景に需要が高まっている。そのような RIA への変更要求について考える。

RIA のアーキテクチャは、従来の Web アプリケーションのアーキテクチャと異なり、MVC の責務がサーバー内に集中しておらず、クライアントとサーバーのプログラムが協調して一つの機能を実現することが多い。特に、非同期通信を行う RIA の機能の実現は、複数のモジュールに横断して存在する。しかし、

RIA への変更要求に対応するために必要な変更箇所が、必ずしも複数モジュールに横断するとは限らない。なぜなら、変更要求は特定のモジュールの変更に集約される場合があるからである。

変更要求が影響するモジュールの組み合わせの例を、表 2.1.に示す。表 2.1.では、変更が及ぶモジュールの組み合わせとして 5 つの例を示している。

		1	2	3		x	y
Server	DB, DTO					✓	✓
	Model						✓
Client	Model	✓			.....	✓	✓
	View		✓			✓	✓
	Controller			✓		✓	✓

表 2.1. 変更要求が影響するモジュールの組み合わせの例

表 2.1.に示されている組み合わせのうち、「(1)Model にのみ影響する」場合の例として、画面内の要素をソートするアルゴリズムを変更するなどといった、ロジックのアルゴリズムの変更が考えられる。「(2)View にのみ影響する」場合の例として、表示の形式やタイミングなどといった、描画処理の変更が考えられる。「(3)Controller のみに影響する」場合の例として、クリックによって起動されていた処理を、ダブルクリックによって起動されるように変更するなどといった、操作性の変更が考えられる。このように、変更要求が影響するモジュールが限られている変更要求が存在する一方で、(x)や(y)のような、変更箇所がサーバーとクライアントのモジュールに横断する場合がある。「(x)変更箇所がサーバーの DB、DTO、クライアントの Model、View、Controller に及ぶ」場合の例として、以下のような場合が考えられる。Yahoo!地図[6]のような地図アプリケーション上で、任意の地点に名前を付けて DB に登録しユーザー間で共有する機能を持つ RIA において、地点に関連するコメントを保存できるように拡張を行いたいとする。このような、クライアント側から DB のデータ追加、参照、変更、削除の CRUD 処理と呼ばれる処理を行う RIA において、扱うデータの属性を拡張したいというような変更要求は一般に考えられる。

変更要求には素早く正確に対応することが望ましいが、(x)の場合のように、変更要求が影響する範囲がサーバーからクライアントまで複数のモジュールに渡る場合、以下に挙げる 2 つの問題が発生すると考えられる。

P.1. 横断的に存在する変更箇所を、開発者が一箇所ずつ巡回して変更を加えていく必要があると、変更漏れが発生し、欠陥となる恐れがある。

P.2. 変更箇所が横断的に必要となる変更要求において、具体的に変更が必要となる箇所を、開発者が全体の実装を確認しながら一箇所ずつ特定していく必要があると、特定に抜けが生じ、欠陥となる恐れがある。

本研究においては、上記 2 つの問題に対する解決のための手法を提案する。

第 3 章 変更要求に対する解決のモジュール化とパターン化

3.1. アスペクト指向プログラミング

前章における問題点 ***P.1.***の解決として、アスペクト指向プログラミング（Aspect-Oriented Programming：AOP）が考えられる。アスペクト指向プログラミングとは、ソフトウェアの特定の振る舞いをアスペクトとして分離し、モジュール化するプログラミングの手法である。

ソフトウェアを開発する際、モデリング、設計、実装に登場する様々な観点を「関心事」と呼ぶ。関心事には、機能や非機能といった開発上の留意点が広く該当する。オブジェクト指向で開発を進める上で、単一のモジュールに落とし込めず、複数のモジュールに実装が横断して必要となる関心事がある。そのような関心事を、「横断的関心事」と呼ぶ。横断的関心事の例としては、ロギングやトランザクション管理などがある。

図 3.1.に従来のオブジェクト指向プログラミングによる横断的関心事の実現を示す。オブジェクト指向プログラミングで横断的関心事を実現するためには、基本となる処理を記述しておき、そのコードを様々なモジュールから呼び出す必要がある。この呼び出しを行うコードのことを「グルーコード」と呼ぶ。オブジェクト指向プログラミングにおいて横断的関心事を実現するためには、グルーコードが必須であり、横断的関心事をモジュール化し、局所化することができない。このため、横断的関心事に関する変更が生じた場合、すべてのグルーコードに変更が必要となり、保守性をはじめとする品質を低下させる原因になる。

図 3.2.にアスペクト指向プログラミングによる横断的関心事の実現を示す。アスペクト指向プログラミングでは、横断的関心事自体に、自らが必要となる場所を示すコードを含んで記述することができる。この記述形態を「アスペクト」と呼ぶ。アスペクトは、ウィーバと呼ばれる合成器を使って、対象となるコードに自動的に織り込むことができる。アスペクト指向プログラミングは、グルーコードなしに横断的関心事をモジュール化して記述することを可能にする。

アスペクト指向プログラミングでは、アスペクトを対象のコードに埋め込む代表的な仕組みとして、ジョインポイントモデルとインタータイプ宣言の 2 つがある。それぞれについて以下に説明する。

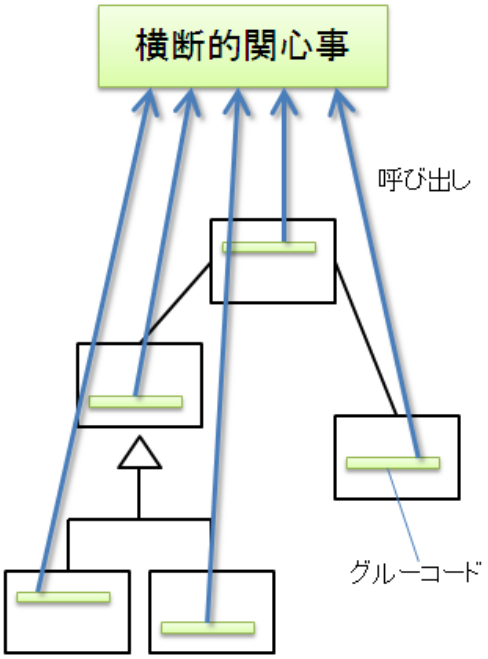

図 3.1. オブジェクト指向プログラミング

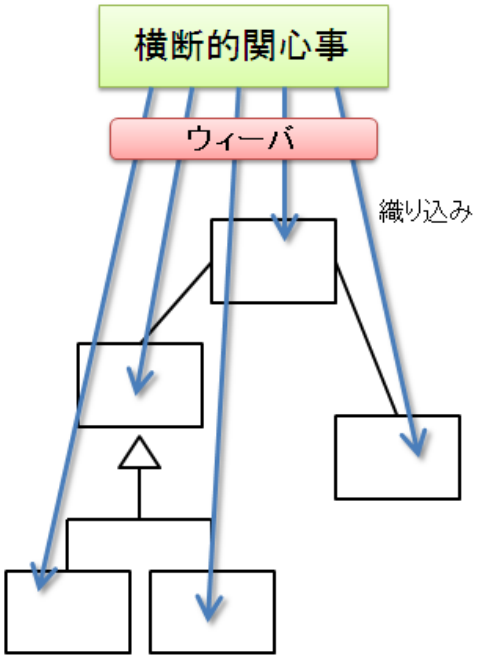

図 3.2. アスペクト指向プログラミング

ジョインポイントモデル

ジョインポイントモデルとは、プログラムの振る舞いの中にアスペクトを割り込ませ、動的な振る舞いを変更させるための仕組みのことである。ジョインポイントモデルを実現するための要素として、ポイントカット、ジョインポイント、アドバイスの 3 つがある。

ポイントカット

ポイントカットは、コード上の特定の位置を指定する条件であり、ジョインポイントを指定するために使う。プログラム中に存在するすべてのジョインポイントの集合から、条件を一つ、または複数指定することによって部分集合を切り出し、対象とすることができる。

ジョインポイント

ジョインポイントは、アドバイスの実行を織り込むことのできるコード上の位置のことである。代表的なものとして、特定の関数が呼び出される位置、特定の変数に値が代入される位置などがある。ジョインポイントの種類は、アスペクト指向プログラミングの処理系によって差がある。

アドバイス

アドバイスは、ジョインポイントに織り込まれる処理のことである。アスペクトが織り込まれる対象となるコードにおいて、ポイントカットによって選択されるジョインポイントに実行がさしかかったときに、実行されるコードの本体となる。コードを実行する代表的なタイミングとして、ジョインポイントの事前、事後、置き換えて実行の 3 種類があり、それぞれ before アドバイス、after アドバイス、around アドバイスと呼ぶ。

JavaScript 用のアスペクト指向プログラミングを可能にする処理系として、AOJS（Aspect-Oriented Programming Framework for JavaScript）[7]がある。以下に、AOJS における before アドバイス、after アドバイス、around アドバイスの例を示す。

```
<var varname="対象となる変数">
    <before><![CDATA[ジョインポイントの事前に織り込む処理]]></before>
</var>
```

リスト 3.1. before アドバイス

```
<function functionname="対象となる関数" pointcut="call">
    <after><![CDATA[ジョインポイントの事後に織り込む処理]]></after>
</function>
```

リスト 3.2. after アドバイス

```
<function functionname="対象となる関数" pointcut="execution">
    <around><![CDATA[
        __proceed__();
      任意の処理
    ]]></around>
</function>
```

リスト 3.3. around アドバイス

リスト 3.1.は対象となる変数に値が代入される時点をジョインポイントとし、その事前に任意の処理を織り込むアスペクトである。

リスト 3.2.は対象となる関数が呼び出される時点をジョインポイントとし、その事後に任意の処理を織り込むアスペクトである。

リスト 3.3.は対象となる関数が宣言されている箇所をジョインポイントとし、その実行に置き換える処理を指定しているアスペクトである。アスペクト中の「__proceed__();」は、もともとジョインポイントにあった処理の実行を示す。つまり、リスト 3.3.は、もともとジョインポイントにあった処理を実行した後に、任意の処理を行う処理を追加して置き換えるアスペクトとなっている。

インタータイプ宣言

インタータイプ宣言とは、プログラムの構造の中にアスペクトを割り込ませ、静的な構造を変更させるための仕組みのことである。インタータイプ宣言によって、任意のクラスに任意の変数や関数を追加することができる。

Java 用のアスペクト指向プログラミングを可能にする処理系として、AspectJ[8]がある。以下に、AspectJ におけるインタータイプ宣言の例を示す

```
public aspect Sample
{
    private String クラス名.変数名 = “初期値”;
    public 返値 クラス名.関数名(引数)
    {
        処理
    }
    …..
}
```

リスト 3.4. インタータイプ宣言

リスト 3.4.は任意のクラスに初期値によって初期化された String 型の変数を、private でフィールドに追加している。またその後、任意のクラスに、引数、返値の関数を public で追加している。

RIA への変更要求のうち、変更箇所が複数のモジュールに横断して必要になる場合、変更箇所をアスペクトとして単一のモジュールに落とし込むことで、前章で挙げた問題 ***P.1.***の解決に役立つと考えられる。

しかし、アスペクト指向プログラミングによるアプローチだけでは、***P.1.***を解決するには不十分である。なぜなら、前章で挙げた「クライアント側から DB の CRUD 処理を行う RIA において、扱うデータの属性を拡張したいというような変更要求」においては、変更箇所はクライアントの JavaScript からサーバーの Java までプラットフォームを横断して散らばるため、AOJS や AspectJ といった単一の処理系では対応しきれないからである。その問題を解決するために、アスペクト指向ソフトウェア開発の視点を取り入れる必要がある。

尚、以後 RIA の問題に対するアスペクト指向プログラミングを用いたアプローチを考えるために、RIA のクライアントサイドの実装を JavaScript、サーバーサイドの実装を Java で行うことと仮定し、それぞれの AOP 処理系に AOJS と AspectJ を用いることとする。

3.2. アスペクト指向ソフトウェア開発

アスペクト指向によるソフトウェア開発の流れを、アスペクト指向ソフトウェア開発（Aspect-Oriented Software Development：AOSD）という。オブジェクト指向プログラミングに対して、オブジェクト指向ソフトウェア開発がより広義な意味を持つのと同様、ジョインポイントモデルとインタータイプ宣言を基盤とした、横断的関心事のモジュール化プログラミング技法であるアスペクト指向プログラミングに対して、AOSD はモデリング、分析、設計を含んだより広い領域を対象とし、横断的関心事をアスペクトとして扱う開発手法である。

RIA への変更要求のうち、変更が必要な箇所が設計に横断して存在する場合、変更要求をアスペクトとしてとらえ、AOSD の考え方を適用することができる。

AOJS や AspectJ といった従来のアスペクト指向プログラミング処理系では、単一のプラットフォームにおける横断的関心事はアスペクトとしてモジュール化することができたが、クライアント、サーバー、DB といったプラットフォームを横断した変更要求を一つのアスペクトとしてとらえ、複数の処理系を併用した上で一括して扱う手法は提案されていなかった。

本研究では、AOSD の考え方から、要求の粒度でアスペクトをとらえ、AOP によってモジュール化されるプラットフォームごとのアスペクトのみならず、

AOP レベルではアスペクトとは呼べないが考える要求の実現には必要となるモジュールまでも一括してアスペクトとして扱う枠組みを提案する。(図 3.3.)

この考え方を適用することで、問題 ***P.1.***の解決とすることができる。具体例は後章にて挙げる。

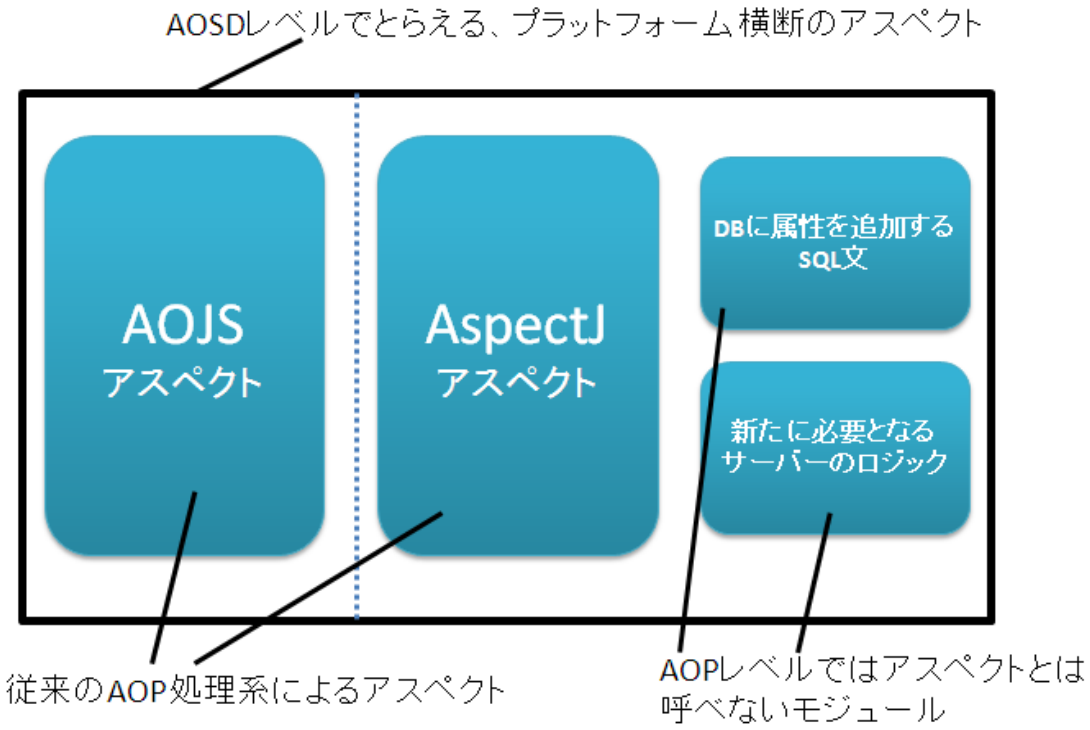

図 3.3. プラットフォーム横断の変更要求をモジュール化したアスペクト

3.3. 変更箇所の特定

AOP、AOSD は前章の問題 P.1.の解決には対応しているが、問題 P.2.の解決にはならない。なぜなら、AOP、AOSD はあくまでモジュール化の技術であり、何をモジュール化すべきかという、変更箇所の特定には貢献しないからである。

開発者が全体の実装を確認しながら、必要となる変更箇所を一箇所ずつ特定していく作業を避けるために、変更箇所特定の作業を定型処理化する方法が考えられる。

本研究では、具体的な変更要求の種類を「問題」とし、その変更要求の実現のために必要な具体的な変更箇所を特定するための、変更箇所と変更内容のルールを「解決」として、問題と解決をパターンとして扱う方法を提案する。

パターンを利用して行う定型処理の流れを図 3.4.に示す。開発者は対象となる RIA のコードと、アスペクトの要素を生成するためのルールをパターンとしてまとめたものを入力とし、パターンにしたがって定型処理を行うことで、各プラットフォームに必要なアスペクトの要素を半完成の状態で得ることができる。本手法は、RIA のマルチプラットフォームなアーキテクチャにおいて、「クライアント側から DB の CRUD 処理を行う RIA において、扱うデータの属性を拡張したいというような変更要求」のような変更箇所がプラットフォームを横断して必要となる変更要求が来た際に、必要なすべての変更箇所を定型処理によって特定し、半完成のアスペクトとしてまとめることで、開発者が変更要求に素早く正確に対応することを支援するものである。開発者は、半完成のアスペクトに状況に応じた具体的な処理を追記し完成させることで、変更適用の漏れや、変更箇所特定の抜けなどを心配せずに、変更を実現することができる。

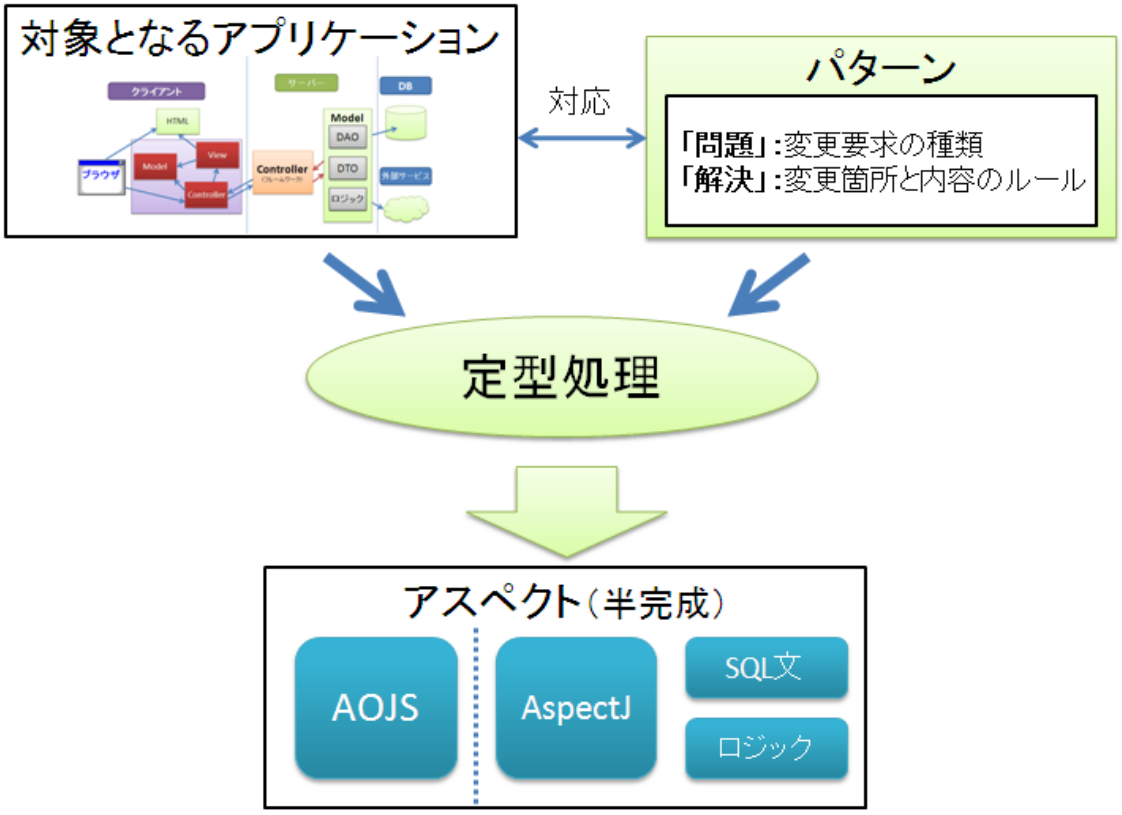

図 3.4. アスペクト生成の定型処理化

しかし、変更箇所がプラットフォームを横断して必要となる変更要求のうち、アスペクト生成のためのルールをうまくパターン化できない場合がある。例として以下のような場合を考える。ホテルの宿泊予約を行う RIA を考える。部屋

の予約要請がきたとき、部屋に空きがあれば予約を作成、部屋に空きがなければ失敗とする仕様になっている。この RIA に、部屋に空きがない場合は、キャンセル待ちリストに追加する処理を加えるような変更要求が来たとする。この変更は、ユースケースを使用して、図 3.5.に示すように表すことができる。

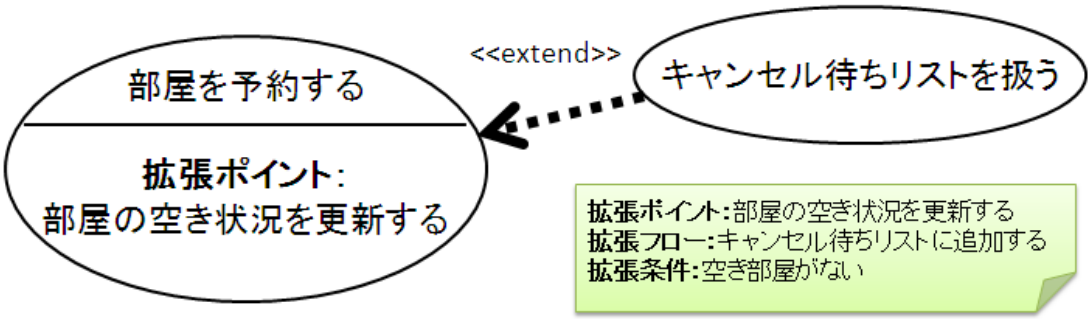

図 3.5. 「キャンセル待ちリストを扱う」変更要求

このような、処理のフローの特定の部分に新たなフローを割り込ませるような、拡張ユースケースで表現することがふさわしい変更要求の例においては、変更箇所の特定が具体的なフローの内容に強く影響されるので、パターンとして一般化することが困難である。また、一連のフローに新たなフローを割り込ませて拡張する場合、合成された後のフローの全体像が重要となるので、実装をアスペクトとして分離したままにしておくことは、モジュール化によるメリットよりもデメリットの方が目立つと考えられる。

本研究で提案する手法は、抽象化された汎用性の高い変更要求の種類に、必要となる変更箇所と内容のルールを組み合わせ、半完成のアスペクトを定型処理によって生成することを目的としている。よって対象となる変更要求は、変更箇所がプラットフォームを横断して必要となる変更要求のうち、ユースケースフローの特定の部分に拡張のフローを割り込ませるようなフローの内容に強く依存するものではなく、抽象化してパターンとすることが可能なものに限られる。変更要求の分類を図 3.6.に示す。本手法で対象となる変更要求は、③に分類されるものとなる。

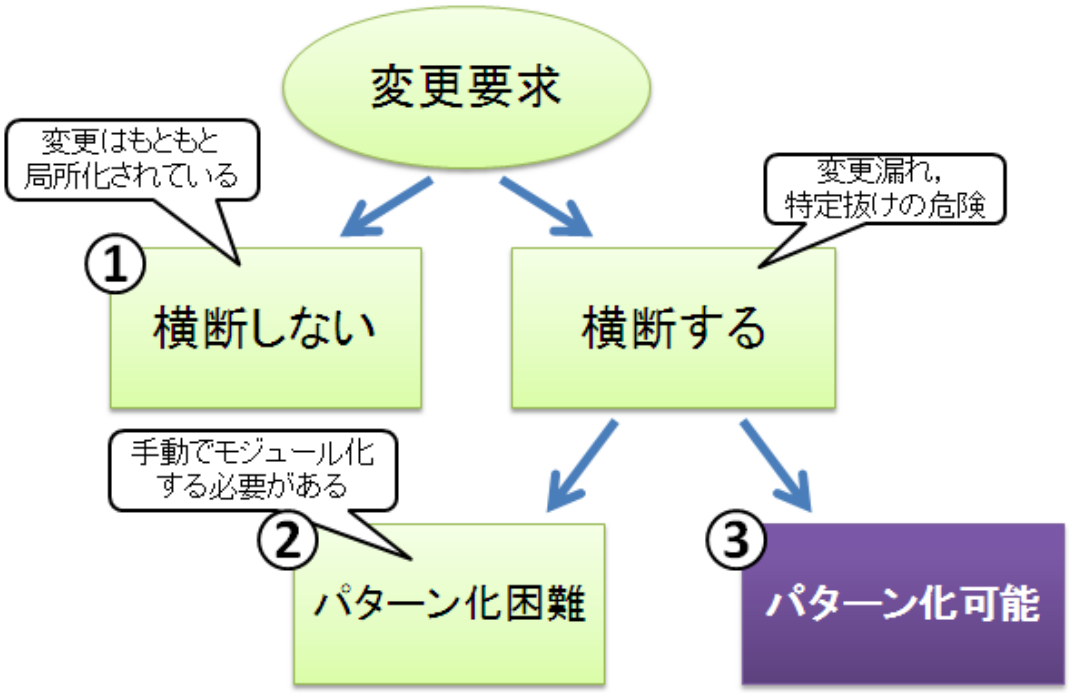
変更要求
変更はもともと
局所化されている
変更漏れ，
特定抜けの危険
①
横断しない
横断する
手動でモジュール化
する必要がある
②
パターン化困難
③
パターン化可能

図 3.6. 変更要求の分類

第 4 章 データベースへの属性追加

前章にて提案した、抽象化された汎用性の高い変更要求の種類に、必要となる変更箇所と内容のルールを組み合わせ、半完成のアスペクトを定型処理によって生成する手法は、手法のみでは機能しない。実際に本手法を利用するためには、条件を満たす具体的なパターンが必要となる。

本研究では、「RDBMS の CRUD 処理を行う RIA において、扱う属性を追加する」問題を対象に、必要な変更箇所の特定と内容を、具体的なパターンとしてまとめることに成功した。その内容をまとめたものを図 4.1.に示す。

図 4.1. パターンの具体例

以下では、本パターンについて、各モジュールにおける具体的な処理の記述を、クライアントの JavaScript については AOJS、サーバーの Java については AspectJ、RDBMS につては MySQL[9]を利用することを前提に説明する。

4.1. Data Base への変更

本節における Data Base は、Relational Data Base Management System (RDBMS)を前提として話を進める。

まず、DB に追加される属性を格納するために、DB テーブルに列を追加する必要がある。追加のための命令は、SQL 文を利用して、リスト 4.1.のように記述できる。

このような SQL 文は AOP におけるアスペクトとは全く関係ないが、変更要求に必要な要素全体をアスペクトとしてとらえる AOSD を行う際に、アスペクトの 1 要素として整理することができる。

```
ALTER TABLE 【テーブル名】ADD [COLUMN] 【列名】【データ型】;
```

リスト 4.1. DB テーブルへ列追加のための SQL 文

4.2. Data Transfer Object への変更

DB に保存するデータの形式を、サーバー内でエンティティとして扱うクラスが Data Transfer Object（DTO）である。DTO はフィールドに private な変数としてデータを持ち、それぞれに対応する getter と setter のアクセサを備える。

DB を利用する簡単なシステムでは、サーバーの Model に DTO の DB への CRUD 処理を、SQL 文を直接扱うことで行うロジックとして用意することができる。しかし、サーバーの Controller としてしばしば使われる EJB や Spring といったフレームワークは、DTO の構造を元に SQL 文を扱う処理を Data Access Object（DAO）として自動生成する機能を持っていることが多く、属性追加のような DTO への変更に伴い、SQL 文を扱うロジックの変更を省略することができる。

本研究においては、そのようなフレームワークの DAO 自動生成機能を利用することを前提に、DB に新たな属性を追加する際、DTO にフィールドとアクセサの追加を行えば、SQL 文を扱う CRUD 処理を手動で変更する必要はないとする。

AspectJ を利用し、DB に追加する新たな属性に対応する DTO のフィールドとアクセサを、インタータイプ宣言によって DTO に織り込む実装を、リスト

4.2.に示す。

```
public aspect Sample
{
    private 【型】【DTO のクラス名】.【変数名】 = 【初期値】;

    public 【型】【DTO のクラス名】.get【変数名】()
    {
        return 【変数名】;
    }

    public 【型】【DTO のクラス名】.set【変数名】()
    {
        this.【変数名】= 【変数名】;
    }
}
```

リスト 4.2. DTO へフィールド，アクセサを追加するためのアスペクト

4.3. クライアント内 Model への変更

DB への属性の追加に伴い、クライアントで追加された属性を表示、操作できるよう一連の変更を行う必要がある。クライアントの View で描画される実体はクライアントの Model として存在するので、Model のメンバを追加する必要がある。Model のメンバ追加を実現するために、追加の内容を書いた JavaScript ファイルを用意し、AOJS の機能を使って、対象となる JavaScript ファイルに合成する。

追加メンバを記述した JavaScript ファイルをリスト 4.3.に、追加メンバを記述した JavaScript ファイルを対象となる JavaScript ファイルに合成するために必要なアスペクトの記述を、リスト 4.4.に示す。

```
var　【追加先のオブジェクト、クラス名】.【メンバ名】;
…
```

リスト 4.3. 追加メンバを記述した JavaScript ファイル

```
<initializeFile>extension.js</initializeFile>
```

リスト 4.4. JavaScript ファイルを合成するためのアスペクト

4.4. クライアント内 View への変更

DB への属性追加に伴い、クライアント側で表示の操作を変更する必要がある。この変更には以下の二つの種類がある。

(1) 追加された Model のメンバを、CRUD 処理に対応させるために、入力フォームを新たに設置する。

(2) Model のメンバ追加を受けて、Model の描画処理を再考し、必要に応じて変更を加える。

(1)の変更のために、DHTML の機能を用いて、View の JavaScript から HTML の内容を指定するように記述する。具体的な記述方法をリスト 4.5.に示す。この例において、関数名である”createSampleView”は、画面の初期化時に呼び出され、HTML 内の任意のタグの位置に、HTML の記述を挿入する関数である。この関数の内容を変更することにより、入力フォームを含む画面初期化時の構成を変更することができる。

(2)の変更のために、Model の描画処理を行っている関数内の処理を、新しいものに置き換える。具体的な記述方法をリスト 4.6.に示す。この例において、関数名である”updateModelView”は、Model の描画を行う際に呼び出される関数である。単純に描画する処理を追記するのみでよい場合は、around アドバイス

内で「__proceed__();」関数を用いて、変更前の処理を流用し、追加の処理を追記する方法が有効である。

```
<function functionname="createSampleView" pointcut="execution">
  <around><![CDATA[
      var obj = document.getElementById("【HTML 内のタグ】");
            obj.innerHTML = '【HTML の記述】';
  ]]></around>
</function>
```

リスト 4.5. 画面初期化時の構成を記述する View の置き換えを行うアスペクト

```
<function functionname="/updateModelView" pointcut="execution">
  <around><![CDATA[
      【描画処理】
  ]]></around>
</function>
```

リスト 4.6. Model を描画する View の置き換えを行うアスペクト

4.5. クライアント内 Controller への変更

DB への属性追加に伴い、各イベントに応じて Model を操作する Controller の処理の内部で、Model の追加されたメンバに対応するために、処理を追記する必要がある。

この処理は、Controller が Model の描画を View に指示する時点がジョインポイントとされ、View に描画を指示する直前に、Model の追加されたメンバに適切な値を代入するという処理を行う。具体的な記述方法を、リスト 4.7.に示す。

View の変更が、Model 全体のデータを元に描画を行う場合のように、従来の記述に単純に追記するわけにはいかない例が考えられるのに対して、Controller への追記は Model の追加されたメンバに適切な値を代入するのみというシンプ

ルなものになっており、定型化の効果が高い。

しかし、Model に追加されたメンバに代入すべき値が、クライアント内のロジックだけでは求められず、サーバーの DAO やロジックを利用する必要があることが考えられる。そのような場合は、必要に応じて、クライアントやサーバーの Model にロジックを追加し、利用する必要がある。

```
<function functionname="/【対象とする Controller 関数】/updateModelView" pointcut="call">
    <before><![CDATA[
        【モデル名】.【追加のメンバ】=【適切な値、それを得るための処理】;
    ]]></before>
</function>
```

リスト 4.7. Model の追加されたメンバに適切な値を代入する処理を追記するアスペクト

第 5 章 地図情報共有アプリケーションへの適用実験による評価

前章で示した、「RDBMS の CRUD 処理を行う RIA において、扱う属性を追加する」問題を対象に、必要な変更箇所の特定と内容をまとめた具体的なパターンを、地図情報共有アプリケーションに適用する実験を行った。適用前と適用後のスクリーンショットの比較を、図 5.1.に示す。

対象とした地図情報共有アプリケーションの仕様を説明する。

本アプリケーションでは、Yahoo!地図のサービスを利用し、Ajax を利用したリッチなユーザーインターフェースを備えた地図を利用することができる。地図表示の下には、「Name」「Latitude（緯度）」「Longitude（経度）」の入力フォーム兼表示領域があり、その下に「登録」「削除」「更新」の 3 つのボタンが並んでいる。

ユーザーが地図を操作すると、地図の中心の緯度と経度が、「Latitude」「Longitude」フィールドにリアルタイムに反映される。

「Name」フィールドに任意の文字列を入力し、「登録」ボタンを押すと、「Name」「Latitude」「Longitude」の値を情報として持つ地点が登録される。登録された地点は、地図表示の上でアイコンが表示される。地点は DB に登録されており、複数のユーザーで地点の情報を共有することができる。

このような仕様の地図情報共有アプリケーション上で、地点に関連するコメントを、地点情報として付加して保存できるように拡張を行いたいとする。

具体的には、図 5.1.に示すように、「Name」「Latitude」「Longitude」のフィールドの下に、新たに「Comment」フィールドを設け、地点情報の一つとして扱えるようにしたい。

このような変更要求の実現に、前章で示したパターンを利用することができる。パターンを利用した結果を図 5.2.に示すとともに、以下に説明する。

DB への変更として DB テーブルに「Comment」属性を格納するための列を追加する SQL 文を 1 つ生成した。DTO への変更として DTO のフィールドに String 型の comment 変数を 1 つ追加、その getter と setter を 1 つずつ追加した。クライアント内 Model への変更として Model のフィールドに var 型の comment 変数を 1 つ追加した。クライアント内 View への変更として画面初期化時の構成変更を 1 つ、Model の描画方法の変更を 1 つ行った。クライアント内 Controller への変更として、各イベントに対応する Controller 関数の内部に

Model に追加されたメンバに適切な値を代入する処理を 8 つの関数に渡って追加した。

パターンによって特定することができた以上の変更点は、全部で 14 カ所である。特定された変更のみを行うことで、実際に機能が拡張されたことを確認した。

本実験により、本研究によって提案したパターンの有効性が確認された。

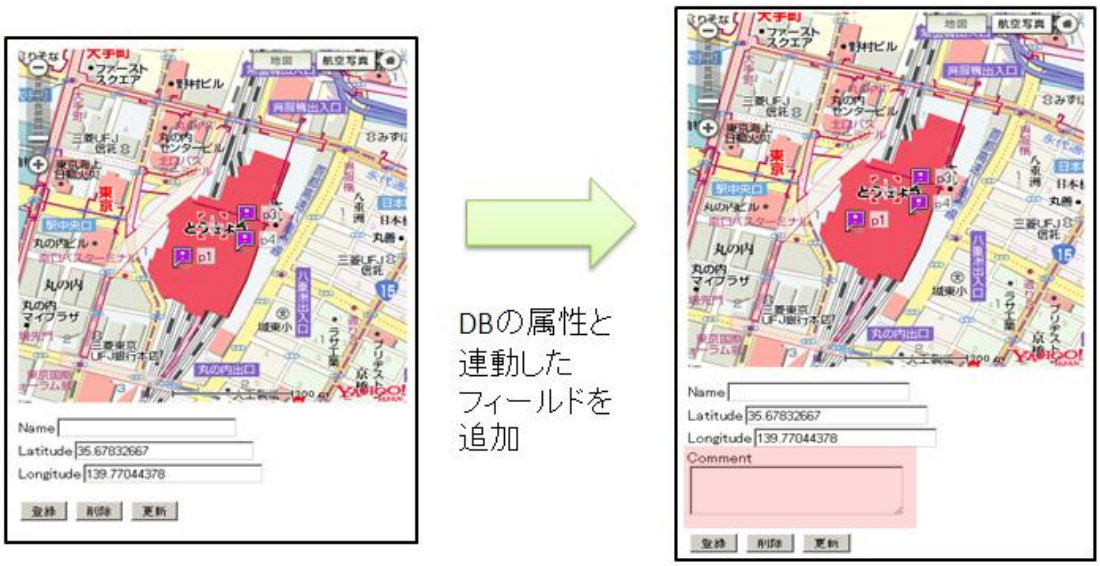

図 5.1. 地図アプリケーションへの適用例

		処理	抽出箇所数
Server	DB	テーブルに列を追加	1
	DTO	フィールドを追加，アクセサを追加	3
Client	Model	フィールドを追加	1
	View	追加されたフィールドの描画	1
	Controller	追加されたフィールドに値を代入	8

図 5.2. パターンの適用により特定された各モジュールにおける変更

第 6 章 関連研究

Aspect-Oriented Software Development with Use-Cases [Jacobson ら 05][10]

Jacobson らは、ユースケース単位で AOSD を行う手法を提案している。ユースケースごとに必要な実装をユースケーススライスとしてまとめ、ユースケーススライスの重ね合わせによって、実装を完成させることができる。多くのユースケースで共通に利用されるデータ構造などは、ユースケース独立スライスとして別に扱う。Jacobson らの言うユースケーススライスこそが、まさに AOSD で考えるアスペクトに相当する。

Jacobson らは、トランザクション管理などの非機能由来の機能を基盤ユースケースとして扱い、提案手法にとりいれているが、Java などの単一のプラットフォーム内での開発を前提としており、機能がマルチプラットフォームに横断する場合については言及していない。

また、Jacobson らは、実装の全てをユースケーススライスにて行うことを提案しているが、それらを生成するための定型処理化やパターン化といったことについてはまったく言及していない。

第 7 章 おわりに

7.1. 総括

本研究では、マルチプラットフォームに横断する変更要求の変更箇所をパターンに従って特定し、アスペクトとしてまとめて扱うことに成功した。

7.2. 今後の課題

定型処理の自動化

本研究では、変更要求と、その実現に必要な変更箇所を特定するためのルールをパターンとしてまとめ、定型処理によって半完成のアスペクトとして変更のモジュール群を得る手法を提案した。しかし、その定型処理の自動化までは至っていない。

今後、対象となるアプリケーションを解析し、パターンに従って変更箇所を自動出力する機能を備えたツールに昇華することが望ましいと考えられる。

MDA の導入

本研究では、変更要求を AOSD のモデリングレベルの視点からアスペクトとしてまとめ、一括して扱うことを行った。これにより、モデリングレベルのアスペクトと対応づけて、変更箇所をパターンとしてまとめることができた。

しかし、現在の状態では、変更を一括して適用することを強制する仕組みが取り入れられていないため、変更が別個に適用されてしまい、同期を確認できずに欠陥となる恐れがある。

このような状況を避けるために、変更の適用を手動で行うのではなく、MDA(Model Driven Architecture)[11]によるモデル管理を行い、ツールによって変更の実装への織り込みまでを自動で扱うようにすることが望ましいと考えられる。

非機能要求のパターン化

本研究では、具体的なパターンの例として、「RDBMS の CRUD 処理を行う RIA において、扱う属性を追加する」という機能要求を扱った。しかし、本手法は機能要求においてのみ利用可能というわけではない。ビジネスロジックの

処理のフローと強く関係する機能要求はパターン化が難しいと考えられるが、例えば通信の際のセキュリティ確保のように、非機能要求を機能に転化させて実装する場合などに応用できる可能性がある。

謝辞

本研究を進めるにあたり、数々のご指導を頂いた早稲田大学理工学部の鷲崎弘宜准教授に深く感謝いたします。

また、共に研究に励み、様々な面でご協力いただいた鷲崎研究室の皆さまに深く感謝致します。

参考文献

[1] Ajax: A New Approach to Web
http://www.adaptivepath.com/ideas/essays/archives/000385.php

[2] Adobe Flash
http://www.adobe.com/jp/products/flash

[3] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin: Aspect-oriented programming,
Proc. European Conference on Object-Oriented Programming (ECOOP), pp220--242, 1997.

[4] Java EE
http://java.sun.com/javaee/

[5] Ruby on Rails
http://rubyonrails.org/

[6] Yahoo!JAPAN デベロッパーネットワーク-地図 .
http://developer.yahoo.co.jp/webapi/map/

[7] AOJS(Aspect-Oriented JavaScript Programming Framework)
Hironori Washizaki, Atsuto Kubo, Tomohiko Mizumachi, Kazuki Eguchi, Yoshiaki Fukazawa, Nobukazu Yoshioka, Hideyuki Kanuka, Toshihiro Kodaka, Nobuhide Sugimoto, Yoichi Nagai, Rieko Yamamoto, "AOJS: Aspect-Oriented JavaScript Programming Framework", Proc. 6th Asian Workshop on Foundations of Software (AWFS 2009), 2009. (to appear)

[8] AspectJ
http://www.eclipse.org/aspectj/

[9] MySQL
http://www.mysql.com/

[10] Ivar Jacobson, Pan-Wei Ng, "Aspect-Oriented Software Development With Use Cases", Addison-Wesley Professional, 2005

[11] Object Management Group. MDA
http://www.omg.org/mda/