平成20年度 修士論文

WEB アプリケーションのモジュール化のための言語

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻
学籍番号 07M37051
内河　綾

指導教員

千葉 滋 教授

平成21年2月6日

概要

今日, デスクトップアプリケーションに代わり, WEB アプリケーションが主流となってきている. WEB アプリケーションはさまざまな言語で開発可能であるが, Java 言語で開発されることが多くなってきている. WEB アプリケーション開発において, 拡張機能をモジュールとして追加することを考えると, まずアプリケーションのロジック部分等を修正しなければならない. Java 言語用のアスペクト指向言語を使えば, ロジック部分の修正箇所に関しては簡単にモジュールとしてまとめて追加できる. 元のコードを修正する必要もない.

WEB アプリケーションの場合, データの保存先としてデータベースを使用することが多い. データベースを Java から呼び出すときは O/R Mapper と呼ばれる永続システムを使用することが主流である. データベースのテーブルに対応する永続クラスを作成すると, ロジック部分からデータベースへ容易にアクセスできるようになる. よって, データベースの設計変更を伴う拡張機能を追加する場合, ロジック部分以外に永続クラスとデータベースを修正しなければならない. ロジック部分と永続クラスに関しては, 既存のアスペクト指向言語でモジュール化が可能であるが, データベースは Java 言語の外部にあるため, モジュール化が出来ない.

そこで, 本研究ではこれらの問題を解決するために, Java 言語と永続システムの両方に対応したアスペクト指向言語を提案する. 本システムでは, 既存のアスペクト指向言語同様, 追加機能はアスペクトとして別ファイルに記述するため, 元のコードを修正する必要がない. また, 永続クラスがアスペクトで修正されていたら, それに対応するデータベースのテーブルを自動で修正する機能を持つ. これにより, データベースの変更もモジュール化され, 機能の追加や削除がアスペクトをウィーブあるいはアンウィーブするという単一の作業で可能となる.

本システムは, Java 言語のクラスファイルを処理対象にする. クラスファイルからアスペクトで永続クラスに追加されたフィールドを読み込み, データベースと比較しカラムの追加, 修正, 削除を行う. アスペクトのウィーブはその後自動で行う. また, Tomcat へ組み込むことにより, Tomcat を起動, 再起動することでデータベースの更新とアスペクトのウィーブを自動で行えるようにした. また, 本システムを使った場合の実行時間は使わ

なかった場合に比べて, 起動は遅いものの一定時間経過すると変わらないことがわかった.

謝辞

本研究を支えていただいた多くの方々への感謝の意をここでは表したいと思います. 東京工業大学 千葉滋教授には, 私が千葉研究室に所属した 3 年間の間, 様々な面でご指導いただきました. 本研究を進めるにあたり, 研究の方向付けから論文の組み立て方, 発表資料に至るまで多岐にわたってご指導していただきました. 深く感謝いたします.

千葉研究室の堀江倫大氏にはアスペクト指向言語の知識や論文の組み立て方について教えていただきました. 竹内秀行氏には Tomcat やデータベースについて教えていただきました. また, 戸部敦氏にはシステムの設計や実装など, 様々な助言をいただきました. 心より感謝いたします.

最後に, 本研究に関して貴重な意見をくださった千葉研究室の方々に心から感謝いたします.

目 次

第 1 章 はじめに **8**

第 2 章 WEB アプリケーションと既存のアスペクト指向言語 **10**

2.1 WEB アプリケーション 10
2.1.1 Tomcat 11
2.1.2 MVC モデル 11
2.2 永続システム 12
2.2.1 JDBC 13
2.2.2 永続システム 14
2.2.3 Hibernate 16
2.2.4 AspectualStore 17
2.3 拡張機能をモジュールとして追加 19
2.3.1 オブジェクト指向言語 19
2.3.2 アスペクト指向言語 19
2.3.3 AspectJ 20
2.3.4 GluonJ 20
2.4 既存のアスペクト指向言語では永続化システムへの対応が困難 23

第 3 章 WEB アプリケーションのモジュール化のための言語 **26**

3.1 特徴 26
3.2 データベースの自動変更 27
3.2.1 永続化クラスを先に実装しデータベースは自動で生成 28
3.2.2 既存の自動生成ツール 29
3.3 仕様と記述例 29
3.3.1 設定ファイル 30
3.3.2 @PersistClass アノテーション 31
3.3.3 @ColumnDef アノテーション 32
3.3.4 カラムの削除 33
3.3.5 カラムのデフォルト値の設定 34
3.4 織り込み 34
3.4.1 実行前 34

3.4.2　ロードタイムウィービング 35

第 4 章　実装 **36**

4.1 処理の全体の流れ 36

4.1.1　データベースの更新 37

4.2 AspectualStore の拡張 38

第 5 章　応用例と実験結果 **41**

5.1 実装するアプリケーション 41

5.2 変更ファイル数の比較 44

5.2.1　オブジェクト指向言語で実装する場合 44

5.2.2　既存のアスペクト指向言語で実装する場合 46

5.2.3　本システムを利用した場合 47

5.3 平均応答時間の比較 48

第 6 章　まとめ **49**

図 目 次

2.1 WEB アプリケーションの構図 10
2.2 MVC モデルとは 12
2.3 拡張機能を追加する際の修正箇所 19
2.4 問題点 23

3.1 データベースの自動変更 27

4.1 全体の流れ 36
4.2 38

5.1 掲示板アプリケーション 42
5.2 全体の流れ 43
5.3 ファイル構造 44
5.4 拡張機能 45
5.5 平均応答時間の比較 48

表 目 次

3.1 Type 列挙型の定数一覧 33
3.2 DeleteTiming 列挙型の定数一覧 34

5.1 修正ファイル・フォルダ数 47

第1章　はじめに

近年, デスクトップアプリケーションに代わり WEB アプリケーションが主流となってきている．WEB アプリケーションは, さまざまな言語での開発が可能であるが, Java で開発されることが多くなってきている．また, データの保存先としてデータベースを使用することが主流となっている．Java からデータベースにアクセスする方法はたくさんある．例えば, JDBC や Hibernate, AspectualStore などがあげられる．中でも O/R Mapping と呼ばれる, 永続システムを使う方法が主流となっている．永続システムとは, データベースのカラムと永続クラスのフィールドを対応付けるシステムのことである．今までの方法では, データベースからデータを取得するには取得したデータをオブジェクトにマッピングする必要があったが, 永続システムにより透過的にオブジェクトを取得できるようになった.

ここで, WEB アプリケーションにデータベースの変更を伴う拡張機能をモジュールとして追加することを考える．まずオブジェクト指向言語で追加しようとすると, 修正箇所が多岐に渡ってしまいモジュール化出来ない. 次にアスペクト指向言語で追加しようとすると, 大半はモジュール化出来るものの, データベースまでをモジュール化することはできず, 毎回手動で変更しなければならない．永続クラスとデータベースは同等のものであるので, 永続クラスの修正とデータベースの修正は作業内容として重複している．これは時間の無駄であると同時に, バグを増やす要因にもなる.

そこで, WEB アプリケーションのモジュール化のための言語を提案する．本システムでは, データベースの修正を伴う拡張機能を追加する場合, 拡張機能部分をアスペクトとして記述することにより既存のコードを修正する必要がなくなる．また, 永続クラスにアスペクトとして追加されたフィールドは, データベースに対応するように自動で修正される．これにより, データベースにまたがる処理をアスペクトでモジュール化できるようになり, 機能の抜き差しが容易に出来るようになった．また, Tomcat を起動または再起動することで, 本システムが起動され, データベースを適切な形に修正しアスペクトのウィーブが行われる.

本稿の残りは以下のような構成からなっている．第 2 章では, 研究背景

として, 永続システムと既存のアスペクト指向言語の問題点について述べる. 第 3 章では, WEB アプリケーションのモジュール化のための言語を提案する. 第 4 章では, 本システムの実装について述べる. 第 5 章では, 本システムの応用例として実際に作成した掲示板と実験について述べる. 第 6 章は, まとめである.

第2章　WEB アプリケーションと既存のアスペクト指向言語

2.1　WEB アプリケーション

近年は, デスクトップアプリケーションに代わり, WEB アプリケーションが主流となってきている. WEB アプリケーションとは, インターネット上で使えるアプリケーションのことである. デスクトップアプリケーションと違い, 個々のコンピュータにソフトをインストールする必要がない. また, URL さえわかっていてインターネットに接続することができればすぐに利用することができる, というメリットがある.

通常 WEB アプリケーションは, 図 2.1 のようにブラウザ・アプリケーションサーバー・データベースの 3 層で成り立っている. ユーザが URL を入力することにより, アプリケーションサーバーへ処理が依頼される. アプリケーションサーバーは依頼された処理を実行し, ブラウザに返す. ブラウザは受け取った処理結果を表示する. よって, ブラウザは, 基本的に何か処理等を行うことはなく, 表示するのみであり, 実際に処理を行うのは, アプリケーションサーバーである. また, アプリケーションサーバーが処理する際に必要なデータや, 処理して出来たデータを保存しておく場所として, 一般的にデータベースが用いられる.

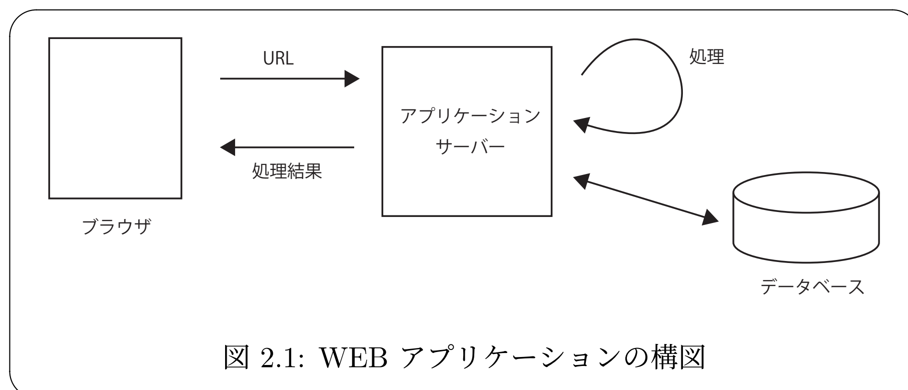

図 2.1: WEB アプリケーションの構図

アプリケーションサーバーは, フリーのものから有料のものまで, さまざまなものがある. たとえば, jakarta Project の Tomcat [10] や IBM 製

の WebSphere Application Server [5], BEA 製の WebLogic Server [7], Microsft 製のインターネット・インフォメーション・サービス (IIS) [6] などがある.

データベースも, フリーのものから有料のものまでさまざまである. 例として, MySQL, PostgreSQL, Oracle, DB2, SQLServer などがあげられる.

WEB アプリケーションは, PHP や Ruby, C 言語などさまざまな言語を利用して開発されているが, 近年では, 大規模基幹システムやミッションクリティカルシステムでも Java EE を利用して開発されることが増えてきている. WEB アプリケーションが Java 言語で開発されるようになった理由として, サーブレットが JDBC や EJB など, Java API 全般を利用することができることがあげられる. つまり, 豊富な Java ライブラリが既に存在していたからである.

2.1.1 Tomcat

Tomcat は Java で書かれたサーバーアプリケーションである. 多くのライブラリやフレームワークが, Tomcat に組み込むことのできるように開発されている. 後述する Hibernate や AspectualStore も, そのようなフレームワークの一つである.

Tomcat では, コンテキストファイルを変更することによって, WEB アプリケーションごとに Tomcat の機能を拡張したり変更したりすることができる. また, Tomcat は WEB アプリケーションごとに異なるクラスローダーを使用する. これによって, WEB アプリケーション間で互いに影響を及ぼすことなく安全な WEB アプリケーションを実現している点も特徴である.

2.1.2 MVC モデル

MVC (Model View Controller) モデルは, WEB アプリケーション開発のための手法で用いられているモデルの一つであり, 近年ではこのモデルを使用した開発が主流となってきている.

処理の流れとそれぞれの役割

MVC モデルを利用した場合の処理の流れは図 2.2 のようになる. クライアントからのリクエストは, サーブレットが受け取り, 処理を行う. この時にデータベースなどのデータが使われることもある. 処理を行い, どのページを表示するかを決定すると, 結果を JSP に送信する. この結果を元に JSP は画面表示する.

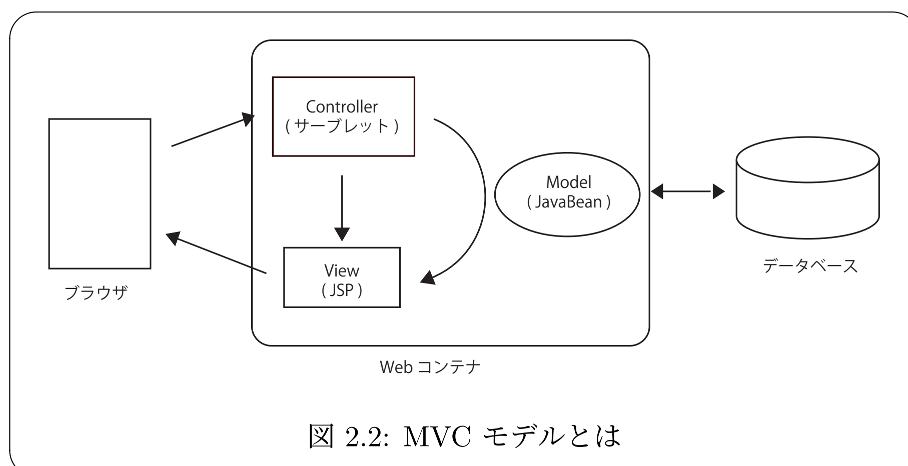

図 2.2: MVC モデルとは

Model, View, Controller の役割は以下のようになっている．

- Model
  アプリケーションで必要なデータと, そのデータを操作するためのメソッドを提供している．通常, EJB や JavaBean で実装される．
- View
  表示と入出力を担当している．通常, HTML や JSP で実装される．
- Controller
  Model と View を制御している．ユーザーからの入力に応じて, 必要な処理を Model に依頼し, その結果を View に依頼する．通常, Java サーブレットで実装される．

MVC モデルのメリット

上記のように, 機能ごとに役割が分担されることで, 開発する上での分業が楽になる．また, コンポーネント間の依存性が少なくなるため, 他の部分の変更による影響を受けにくくて済む．これにより, コンポーネントの再利用性が高まる．また, 他のコンポーネントが変更されることによって, 同一のソースに対して複数の人がメンテナンスを行うということを防ぐことも出でき, 保守性も確保される．

2.2　永続システム

データベースを利用して開発するアプリケーションでは, データベースにアクセスするために何かしらツールを使用する方法が一般的である．こ

のツールの例としては, Hibernate や Spring framework, EJB, AspectualStore などがあげられる. また, JDBC は, これらのデータベースへのアクセスの基礎となっている.

本章ではまず JDBC について説明し, O/R Mapper と呼ばれる永続システムが登場してくるまでの流れを簡単に説明する. そして最後に永続システムについて説明する.

2.2.1 JDBC

JDBC[9] とは Java プログラムからデータベースにアクセスするための基礎をなすもので, Java API の一部である. 昔は, データベースへのアクセス方法がデータベースにより異っていた. よって, プログラムからデータベースにアクセスするためには, 使用するデータベースに対応する API を用いなければならなかった. このため, データベースへのアクセスを標準化するために現れた API が, JDBC である. JDBC を利用することにより, どのデータベースに対しても同じ手順でアクセスすることが可能となった.

JDBC を直接利用したプログラム例を以下に示す.

```
1  public List findOwnerPet() throws ApplicationException{
2    ...
3    String sql = "SELECT name, content FROM post";
4
5    Connection con = null;
6    PreparedStatement ps = null;
7    ResultSet rs = null;
8    try{
9      con = ds.getConnection();
10     ps = con.prepareStatement(sql);
11     rs = ps.executeQuery();
12     while(rs.next()){
13       String name = rs.getString("name");
14       String content = rs.getString("content");
15       Post post = new Post(name, content);
16       postList.add(post);
17     }
18     ...
19   }catch(SQLException e){
20     System.err.println("SQLException が発生しました");
21   }finally{
22     try{
23       if(rs != null)
24         rs.close();
25     }catch(SQLException e){
26       ...
27     }
28     try{
29       if(ps != null)
```

```
30            ps.close();
31        }catch(SQLException e){
32            ...
33        }
34        try{
35            if(con != null)
36                con.close();
37        }catch(SQLException e){
38            ...
39        }
40    }
41    return postList;
42 }
```

この例を見てもわかるとおり, この方法では以下の問題点があげられる.

- 本質的ではないコードをたくさん書かなくてはならない
  この場合の本質的なコードとは, SQL を発行することである. 例えば SQL が select 文であれば SQL 文発行の結果を元に, オブジェクトを作成しなければならない.

- Connection 等の close 処理
  データベースへアクセスするために取得した Connection や ResultSet などは必ず close しなければならない. この時, close したいものが null でないかのチェックや例外処理を行う必要がある.
  また, SQLException がたくさん発生してしまい, これではどこで発生したエラーなのかがわかりにくくなってしまう.

以上の問題点のほかに, データベースはオブジェクト指向とは別の概念であり, オブジェクト指向の Java ユーザーにはわかりにくいという問題がある. そこで JDBC に修正を加え, 出来上がってきたものが次の第 2.2.2 章で説明する, 永続システムである.

2.2.2　永続システム

永続システムとは, オブジェクトとリレーショナルデータベースのマッピングを対応付けするシステムのことである. データベースの変更は開発にはつき物である. しかし, データベースのスキーマを変更したら, システム内の SQL 文を大幅に変更しなければならない. 一つでも間違っていてはいけないので, 全て見直す必要がある. これはとても膨大な作業になってしまう. そこでこのような煩雑な作業を軽減し, データベースのレコードを一つのオブジェクトとしてみなし, オブジェクト指向ユーザー向けに使いやすくしたものが, 永続システムである.

JDBC のシステムでは主に以下の二つの問題点があった.

- インピーダンスミスマッチの発生
- データベースが非オブジェクト指向言語である

そこで, 永続システムでは上の問題点を解決した.

インピーダンスミスマッチ

オブジェクト指向言語でリレーショナルデータベースを扱う際には, リレーショナルデータベースのデータをオブジェクトにマッピングしなければならない. もしくは, オブジェクトをリレーショナルデータベースに対応するデータにマッピングしなければならない. このマッピング作業のことをインピーダンスミスマッチと呼んでいる.

この煩雑なマッピング作業は, それぞれの目的の違いから発生している. リレーショナルデータベースは, データベースの検索や更新削除を迅速に行うためのアルゴリズムが適応されている. それに対し, オブジェクト指向言語は, データを現実世界のモデルになぞらえた考え方である.

どちらもそれぞれの目的にあった最適なアルゴリズムをしている. よってどちらが良い悪いというものではない. 両方最適なアルゴリズムなのである. しかし, 両者を共存させることは極めて厳しい. リレーショナルデータベース作成時にオブジェクト指向を持ち込むと, パフォーマンスの悪化や, 更新時に複数のテーブルに更新がまたがりかねない.

永続システムでは上記の問題を, データベースのレコードを一つのオブジェクトに対応付けることによって解決した. データベースのカラムとオブジェクト指向のクラスのフィールドをマッピングしておく. このマッピングにより, 個々の検索結果はそれぞれオブジェクトにマッピングされる. また, 挿入や削除もオブジェクトを挿入/削除することにより, データベースからレコードの挿入/削除が行えるようになった.

データベースの非オブジェクト指向手続き

JDBC では直接 SQL 文を発行しなければならなかった. SQL は非オブジェクト指向手続きであり, 結果も非オブジェクトとして返ってくる. そこで, 非オブジェクトの結果をオブジェクトとして扱うために, 毎回自らの手でマッピングを行わなくてはならず, 大変な作業となっていた. また, このマッピング作業の大半は, 単純なコードの繰り返しであるため, 気付きにくいバグを埋め込んでしまう危険性も含んでいる. 永続システムでは, このマッピング作業を自動で行ってくれる. よってマッピング作業を毎回自らの手で行う必要がなくなった.

2.2.3 Hibernate

Hibernate[4] は Gavin King 氏が中心となって開発を進めている永続システムの 1 つである．現在ではデータベースにアクセスするためのフレームワークの中で, 最も多く利用されていると思われるフレームワークの 1 つである．

Hibernate では, まずモデルクラスを作成し, モデル間の関係をマッピングファイルに記述する．Hibernate は, モデルクラスとマッピングファイルを読み込み, 自動的に関連するテーブルのレコードを取得してくれる．レコードの条件などをカスタマイズしたいときには, HQL と呼ばれる SQL を簡易化した言語を使用する．以下に簡単なモデルの記述方法と, データの取得方法を示す.

```
1  // モデルの記述
2  public class Paper implements java.io.Serializable {
3    private Integer id;
4    private String title;
5    private Integer authorId;
6    private Author author;
7
8    public Integer getId() { return id; }
9    ... setter & getter
10 }
11
12 // マッピングファイルの記述
13 <?xml version="1.0" ?>
14 <!DOCTYPE hibernate-mapping PUBLIC "
15   -//Hibernate/Hibernate Mapping DTD//EN"
16   "http://hibernate.sourceforge.net/
17   hibernate-mapping-2.0.dtd">
18 <hibernate-mapping>
19 <class name="Paper" table="papers">
20 <id name="id" column="id" type="java.lang.Integer" >
21   <generator class="assigned" />
22 </id>
23 <property name="title" type="string" column="title" />
24 <property name="authorId"
25   type="java.lang.Integer" column="author_id" />
26 <many-to-one name ="author" column="authorId"
27   class="Author" cascade="all" outer-join="auto"
28   update="false" insert="false" />
29 </class>
30 </hibernate-mapping>
31
32 // レコードの取得
33 SessionFactory sessionfactory = config.buildSessionFactory();
34 Session session = sessionfactory.openSession();
35 List list = session.find(" FROM Paper ");
```

2.2.4 AspectualStore

AspectualStore は, 本研究室にて青木康博氏が開発した永続システムの 1 つである．AspectualStore には以下の特徴がある．

- オブジェクト指向の永続システム
  例えば Gavin King 氏を中心に開発を行っている Hibernate では select 文を記述するために HQL で記述する．しかし, AspectualStore ではリレーショナルデータベースがわからないオブジェクト指向ユーザーのために, オブジェクト指向言語で書けるように工夫されている．
- アスペクト指向による機能の拡張
  Rashid らは, アスペクト指向を用いた永続システムの設計を推奨した [8]．AspectualStore もその理念にのっとり, アスペクトを記述することでデータベースへのアクセスをするアルゴリズムを容易に変更できるようにした．

Tomcat との連携

AspectualStore は, Tomcat への組み込みが可能なシステムである．これは, Tomcat のクラスローダーを差し替えることによって実現している．

記述例

AspectualStore の記述例について述べる．
まず, Post テーブルから name カラムに "hoge" と入っているレコードを取り出してくる場合の記述例を以下に示す．select 文を書きたい場合, つまりデータベースにあるテーブルからレコードを取得したい場合は, 以下のコードの read() メソッドのように記述する．

```
1  public List read() {
2    List<Post> posts = new ArrayList<Post>();
3    Session session =
4      SessionFactory.getFactory().openSession();
5    Transaction t = session.beginTransaction();
6    try {
7      Expression exp = ExpressionFactory.eq("name", "hoge");
8      SelectCriteria q = new SelectCriteria(Post.class, exp);
9      q.setFetchLevel(FetchLevel.ALL_PROPERTY);
10     posts = session.list(q);
11     t.commit();
12   }catch(Exception e) {
13     ...
14   }
15   return posts;
16 }
```

コードを見てわかるとおり, リレーショナルデータベースからデータを取り出す際に, 自動でオブジェクトにマッピングされているため, 手動でマッピングを行う必要がない.

また, 例えば id が 11 から 20 のものだけを取り出してきたい場合は, 上記のコードの Expression exp = ExpressionFactory.eq("name", "hoge"); の部分を

```
1 Expression exp =
2   ExpressionFactory.ge("id", new Integer(11));
3 exp =
4   exp.and(ExpressionFactory.lt("id", new Integer(20)));
```

のように変更すればよい.

次に update 文の例を示す. update 文とはレコードを変更するための SQL 文のことである. これは以下のコードのの write(Post post) メソッドのように記述すれば良い. これも select 文同様, オブジェクトとリレーショナルデータベースとのマッピング作業は AspectualStore が自動で行っているため, 意識する必要はない. オブジェクトをセッションにストアするだけで済む.

```
1  public void write(Post post) {
2    Session session =
3      SessionFactory.getFactory().openSession();
4    Transaction t = session.beginTransaction();
5    try {
6      session.store(post);
7      session.flush();
8      t.commit();
9    }catch(Exception e) {
10     ...
11   }
12 }
```

最後に delete 文について述べる. delete 文とはデータベースからレコードを削除することである. これは以下のように記述する. 削除したい Post オブジェクトをセッションから削除 (delete) するだけである. select 文や update 文同様, リレーショナルデータベースの構造を意識する必要はない.

```
1 ...
2 List list = session.list(q);
3 Iterator<Post> it = list.iterator();
4 for(int i = 0; it.hasNext(); i++) {
5   Post post = it.next();
6   posts.add(post);
7   session.delete(post);
8 }
9 ...
```

2.3　拡張機能をモジュールとして追加

WEB アプリケーションに, 拡張機能をモジュールとして追加することを考えてみる. ここでは, 拡張機能の追加方法として, オブジェクト指向で追加する方法と, アスペクト指向言語を用いて追加する方法の 2 種類を提案する.

2.3.1　オブジェクト指向言語

オブジェクト指向言語で拡張機能を追加する場合, Model, View, Controller, データベースのすべての箇所を修正する必要がある. 図 2.3 でいうと赤色の部分をすべて修正する必要がある. しかし, これでは修正箇所が多岐に渡ってしまい, 機能を追加したり削除したりする際に修正が複雑になってしまうという欠点がある.

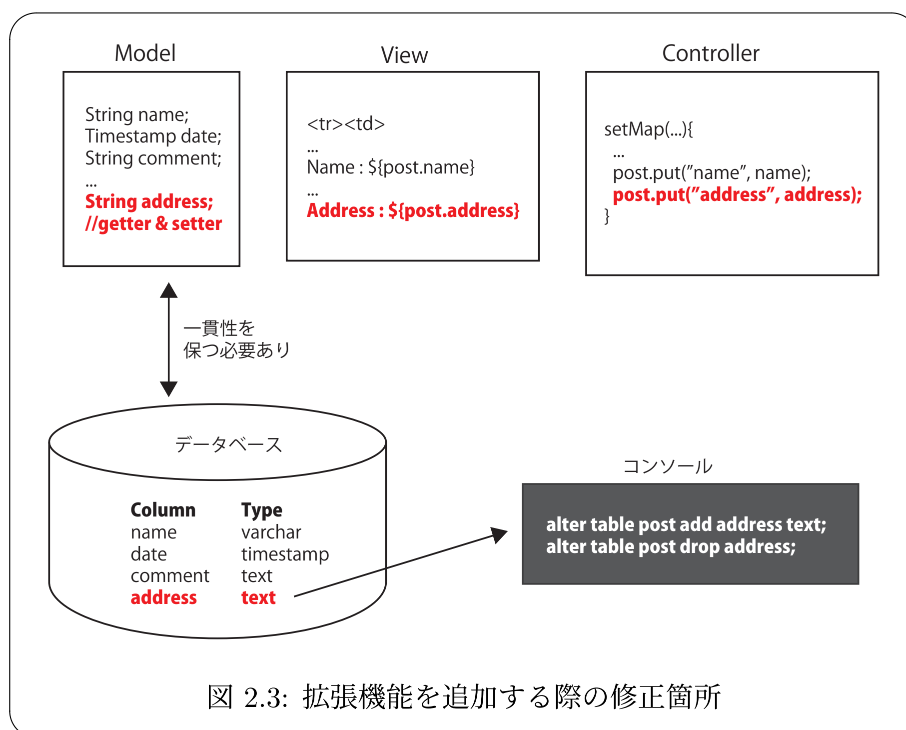

図 2.3: 拡張機能を追加する際の修正箇所

2.3.2　アスペクト指向言語

アスペクト指向言語とは横断的関心事をモジュール化出来る言語である. オブジェクト指向言語でも関心事をモジュール化し, オブジェクトと

いう単位にまとめた．しかし, どんなにきれいにオブジェクトにまとめても, モジュール化出来ない関心事も現れてくるだろう．例えばロギングの例である．いろいろなメソッドの後にログを出したい時である．

つまりアスペクト指向言語は, オブジェクト指向ではモジュール化出来ず横断的関心事となってしまった関心事をモジュール化するための言語である．アスペクト指向言語により, 本質的な関心事とは別の横断的関心事もモジュール化できるようになった [12]．

このアスペクト指向言語を利用すると, オブジェクト指向言語と違い, Controller, View, Model に関してはモジュール化することが可能になる．しかし, データベースに関してはモジュール化することが出来ないため, オブジェクト指向言語の時同様, コンソール上で毎回修正をする必要がある．

最後に, アスペクト指向言語の一つである, AspectJ と GluonJ について説明をする．

2.3.3　AspectJ

AspectJ[3, 11] はアスペクト指向言語の一つであり, 今では数多くの人が利用している言語である．これは Eclipse Fundation が主に開発を行っている．現在では最も成熟した言語と言っても過言ではないだろう．

AspectJ は Java を拡張した言語であり, 独自の文法で記述される．また, 織り込みは静的に行われる．つまり, アスペクトを織り込むためのコンパイルが必要となる．よってアスペクトの動的な追加や削除は行えなくなってしまう．

しかし, このタイプのフレームワークはアスペクトを追加や削除するためにプログラムを停止する必要がある．これは開発段階では不便なことがある．そこで動的なアスペクトの織り込みをするタイプのフレームワークが現れてきた．例えば GluonJ や SpringAOP, JBossAOP などが挙げられる．

2.3.4　GluonJ

動的にアスペクトを織り込むタイプのフレームワークの一つとしてここでは GluonJ[2, 1] を取り上げる．GluonJ は千葉滋氏が中心となって開発を進めている, アスペクト指向言語システムである．GlueonJ は Java の文法であるアノテーションを使うことによって, プログラムを組むことが出来る．よって, アスペクト指向言語用の新しい文法を覚えなくて済むというメリットがある．

次に, リファインメントと, ポイントカット・アドバイスについて説明する．

リファインメント

@Refine アノテーションを利用すると, 既存のクラスを改良することが可能となる. 既存のクラスの改良とは, フィールドやメソッドを追加・上書きすること等である. 以下に BookGlue クラスが Book クラスに変更を加えている例を示す.

```
1 package test;
2 public class Book {
3   public String author;
4   public void toString() {
5     System.out.println("author is " + author);
6   }
7 }
```

```
1 @Glue class BookGlue {
2   @Refine static class Book extends test.Book {
3     public String title = "book title";
4     public void toString() {
5       System.out.println(title);
6     }
7   }
8 }
```

上記のコードを見ると, @Glue クラスは, static な Book クラスを含んでいる. この Book クラスは, @Refine アノテーションがついている. このような, @Refine アノテーションがついた, static なクラスのことを, @Refine クラスと呼ぶ. 修正元のクラスは, @Refine クラスに extends されている. 上記の例だと, test.Book クラスが修正元のクラスである. @Refine クラスの名前付けに規則はなく, どのような名前でも大丈夫である. また, 1 つの @Glue クラスの中に @Refine クラスを複数個書くことも可能である.

上記コードの例では, まず title フィールドが付け足され, メソッド toString() が, アスペクトで記述した toString() メソッドに置き換えられる. つまり, toString() メソッドが上書きされる. よって元の Book クラスが以下のようなクラスに変更されたと考えればよい.

```
1 package test;
2 public class Book {
3   public String author = "hogehoge";
4   public String title = "book title";
5   public void toString() {
6     System.out.println(title);
7   }
8 }
```

新しいメソッドの追加も同様に行える. 例えば元の Book クラスに

```
1 public void getPage(int page) {
2   System.out.println("Book page is " + page);
```

```
3 }
```

というメソッドを付け加えたいとすると, 以下のような Glue クラスを書けばよい.

```
1 @Glue class PageGetter {
2   @Refine static class Diff extends Book {
3     public int page = 100;
4     public void getPage(int page) {
5       System.out.println("Book page is " + page);
6     }
7   }
8 }
```

ポイントカットとアドバイス

指定したタイミングに, 指定した処理を行うことも可能である. 以下のコードは, test.Paper クラスの getInfo メソッドが呼び出されるときに, System.out.println("before getInfo"); を実行する例である.

```
1 package test;
2 public class Paper {
3   public String author;
4   public void getInfo() {
5     System.out.println("author is " + author);
6   }
7 }
```

```
1 @Glue public class PaperGlue {
2   @Before("{ System.out.println('before getInfo');}")
3   Pointcut pc = Pcd.call("test.Paper#getInfo(..)");
4 }
```

ポイントカット・アドバイスを表しているフィールド (上記の例だと, pc) のことを, ポイントカット・フィールドと呼ぶ. このポイントカット・フィールドは, Pointcut 型で, @Before らのアノテーションを付加しなければならない. そうでない場合は, ポイントカット・フィールドとしてみなされない.

指定されたタイミングで呼び出されるブロックのことを, アドバイス・ボディと呼ぶ. 上記コードの例だと, System.out.println('before getInfo'); のことである. GluonJ では, アドバイス・ボディ内で使われている ' を, \" として処理している. アドバイス・ボディが呼び出されるタイミングを 3 行目の call (メソッドが呼ばれるとき) で指定している. ポイントカット指定子には, call 以外にも get (フィールドの値を読み込むとき) や within (メソッドが実行されている間) などがある.

また，ポイントカット・フィールドに付加するアノテーションは，@Before(adviceBody) 以外にも After(adviceBody) や Around(AdviceBody) がある．アドバイス・ボディを実行するタイミングは以下のとおりである．

- @Before(adviceBody)
  ポイントカット・フィールドによって指定されたタイミングの直前で，アドバイス・ボディを実行
- @After(adviceBody)
  指定されたタイミングの直後で，アドバイス・ボディを実行
- @Around(adviceBody)
  ポイントカット・フィールドによって指定された計算の代わりに，アドバイス・ボディを実行

2.4 既存のアスペクト指向言語では永続化システムへの対応が困難

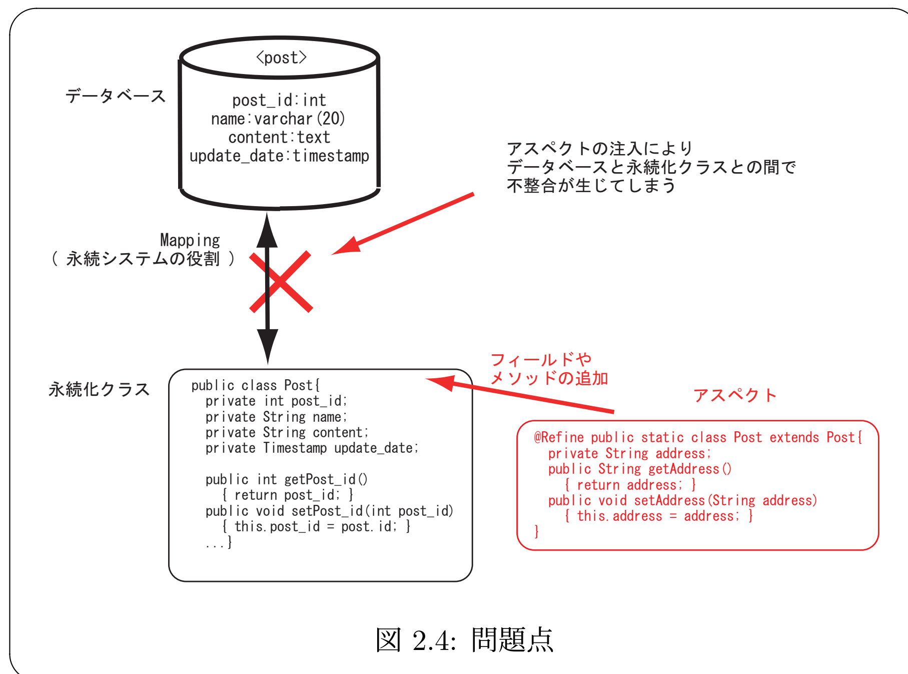

図 2.4: 問題点

例えばある企業が掲示板を作ること考えているとしよう．まず，簡単ではあるが一通りプログラムを作ったとする．掲示板への書き込み等のデータを保持しておく場所として，データベースに post テーブルを作成する．

このテーブルにはカラムとして post_id, name, content, update_date を保持しているものとする．ここに, ユーザーが入力した, 名前や本文等を保持しておくものとする．

ここで, 一通り作った後で実験的に掲示板にアドレスの欄を付け加えてみようとする．しばらく様子を見て周りの反応が良かったら採用し, 悪かったらまた元通りに戻すことを考えているとする．つまり, 追加機能を削除する可能性があるということである．追加機能で必要なアドレス欄を付け加えるにはデータベースに最低でも address カラムを追加しなければいけないとする．

もしアスペクト指向言語を使わないでオブジェクト指向言語で実装をすると, 後で実装部分を取り外すことになった場合にさまざまなな箇所のソースコードを修正しなければならず, 大変な作業となる．そこで, アスペクト指向言語を使うことを考えてみる．アスペクト指向言語は, 第 2.3.2 項でも記述したとおり, 横断的関心事をモジュール化することが可能な言語だからである．そこで, 以下はアスペクト指向言語を利用して実装することを前提として話を進める．

アドレス欄を追加するにあたり, 図 2.4 を見てわかるとおり以下の三つの部分を変更しなければならない．最初にあげる二つの部分だけでは不十分である．

- 永続化クラスに address カラムに対応するフィールドとメソッドを追加
- ロジックの部分の変更（アドレス欄に対応する部分の追加）
- 修正した永続化クラスに対応するデータベースの修正（address カラムの追加）

上の二つの項目はアスペクト指向言語でモジュール化が可能な部分である．しかし, 一番最後の項目に関しては, データベースの修正でありJava のプログラムの修正ではないので, 既存のアスペクト指向言語では対応不可能である．つまり, 追加案件の実装をアスペクト指向言語でモジュール化可能にし後で取り外しをしやすくしようと思っても, データベースや Java のプログラムに実装（変更）が散在してしまう．これではアスペクト指向言語の恩恵を受けにくいと言える．

アスペクト指向言語の恩恵を受けにくいのは, 新機能を追加する時だけではない．既存のアスペクト指向言語のシステムでは, インピーダンスミスマッチが, 永続システムにカラムを追加する時だけでなく, アスペクトを削除した時（追加機能を削除する時）にも起こってしまう．アスペクトを取り外して実行するときには, アスペクトを削除し, さらにデータベースのほうも永続化クラスに対応するように修正しなければならない．アス

ペクト指向言語の一番のメリットは, 関心事をモジュール化し, 機能等の抜き差しを容易にするところである. しかし既存のアスペクト指向言語では, 永続化システムを扱う際にはこのメリットが生かされない.

第3章 WEB アプリケーションのモジュール化のための言語

本研究では, アスペクト指向言語である GluonJ を拡張し, WEB アプリケーションのモジュール化のための言語の設計と実装を行った. 以下では本システムの設計について述べる.

3.1 特徴

本システムはアスペクト指向言語で永続化クラスに変更を加えると, それに伴いデータベースが自動的に変更されるシステムである. 第 2.4 節で述べたように, 永続システムを横断する関心事はアスペクトでモジュールすることは不可能であった. しかし本システムは, 図 3.1 のようにアスペクト指向言語に対応しており, データベースの構造を自動で変更することが出来る. アスペクトを削除した時も追加時と同様にデータベースの変更が自動で行われる. つまり, アスペクト指向を拡張して永続システムを横断するような関心事でもアスペクトでモジュール可能にした言語が本システムである.

本システムには以下の特徴があげられる.

- 自動でデータベースを変更
  まず, アスペクト指向言語 GluonJ でデータベースのテーブルに対応している Bean クラスを変更するプログラムを記述する. ここでの変更とは, データベースのカラムに対応するフィールドの追加やメソッド追加のことである. Bean クラスにアスペクトでフィールド等を追加した状態で本システムを実行すると, データベースのテーブルが永続クラスに応じて変更される. また, アスペクトを取り外して実行した時は, データベースからカラムを取り除いて実行する. これについては第 3.2 節で詳しく述べる.

- カラムの削除のタイミングを 3 通り用意
  本システムでは, カラムを削除するタイミングを 3 通り用意した. EXIT, RESTART, PERSIST の 3 種類である. アスペクトをアン

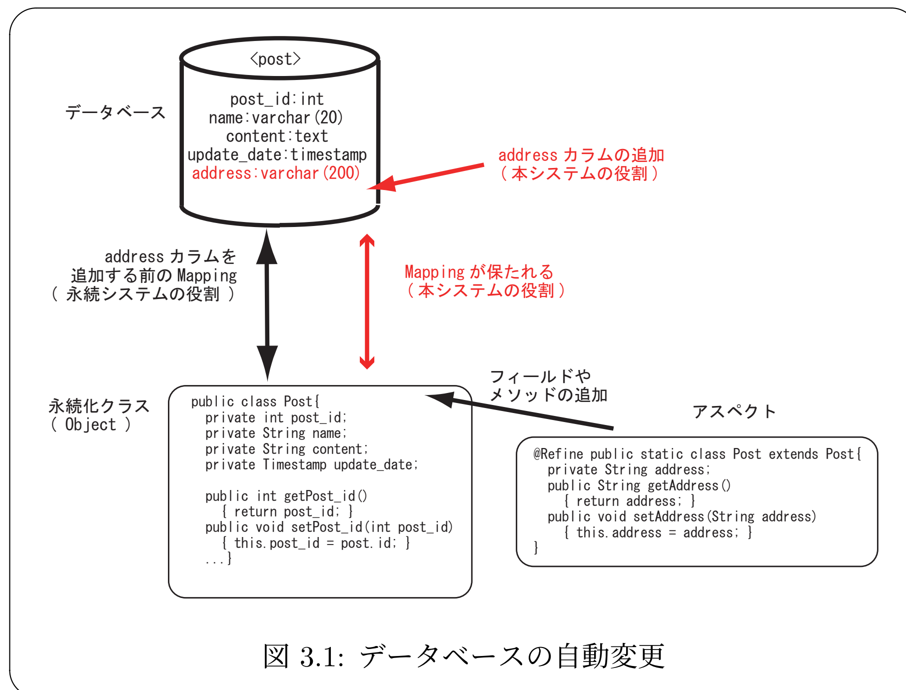

図 3.1: データベースの自動変更

ウィーブしても，データベースのデータを残しておけるようにしておくためである．これは第 3.3.4 項で詳しく説明する．

- Tomcat への組み込み
  Tomcat を起動・停止すると，本システムが起動しデータベースを修正しアスペクトをウィーブするようにした．これは第 3.4.2 節で説明する．

3.2　データベースの自動変更

データベーステーブルのカラムと永続化クラスのフィールドは一貫性を保つべきものであるので，どちらか 1 つを変更したら，もう片方は自動で修正して欲しい．ここで，上記の条件を満たす実装方法は大まかに二つあげられる．データベースを変更すると自動で永続クラスが修正される方法と，永続クラスを変更すると自動でデータベースが修正される方法の 2 つである．本システムでは，アスペクトをウィーブする前にデータベーススキーマを変更することにした．また，ウィーブは実行前に行うことにした．

3.2.1　永続化クラスを先に実装しデータベースは自動で生成

アスペクトを利用して永続化クラスを実装し, 自動でデータベースに反映させるという方法が一般的な考え方であろう. これは”アスペクトを利用する”というのが前提条件にあるからである. また, データベースを先に生成する方法では以下の問題点が発生するからである.

- アスペクトを削除する際に, データベースを手動で変更しなければならない
  データベースを直接変更するにはアスペクトは使えない. よって, 自らの手でデータベースを変更しなければならない. この方法は新機能を追加する時は, 永続化クラスを先に変更する方法と手間はたいして変わらないかもしれない. しかし, 新機能を削除する時, また自らの手で削除しなければならない. また, どれを追加したかがわからなくなるので, わかるようにメモを残しておかなければならない. これではバグも多くなってしまうであろう.

- 変更後, プログラムを書く前にデータベースから永続化クラスを生成する必要がある
  データベースを変更しただけでは, まだ永続化クラスは変更されていない. そこで, 永続化クラスを利用したプログラムを作る前には必ずデータベースから永続化クラスを生成しなければならない. そうでないとプログラム内でエラーが起きてしまうからである. これはデータベースを変更する回数が多いと大変な作業となってしまう.

- 追加したカラムがどれかがわかりにくい
  これは最初の項目とかぶるが, カラムを追加または変更した際に毎回追加したカラム名をメモしておかなければならない. また, プログラムを作成している際に追加したカラムだと気づかず, アスペクト以外のプログラム内で使用してしまった場合, 新機能を削除したとき, つまりデータベースからカラムを削除した場合にコンパイルエラーが起こってしまう.

よって, まず永続化クラスの方を先に変更する方法が良いといえる. 永続化クラスを先に変更すると, データベースを先に変更する方法と比べて以下の利点がある.

- 削除するときはアスペクトだけで十分である
  データベースとは違い新機能追加として作成したアスペクトを削除するだけで十分となる.

- 永続化クラスを変更後, 実行するまでデータベースを変更する必要がない
  Java プログラムを実装している最中は, データベースに接続することはない．よって, 実行するまではデータベースを修正する必要がない．これにより, カラム名の変更等がデータベースを先に生成する方法に比べてしやすくなったといえる．

- 追加したカラムが目で見てすぐにわかる
  アスペクトで追加する場合, 追加するプログラムは元のプログラムとは別のファイルに記述する．よって, データベースを変更した時とは違い, 追加したカラムが一目瞭然である．

3.2.2　既存の自動生成ツール

永続化クラスからデータベースを自動生成するツールや永続化クラスからマッピングメタデータを自動生成, マッピングメタデータからデータベースを自動生成するツールは既に存在する．例えば XDoclet や hbm2ddl, SchemeExport などがあげられる．しかし, これらのツールはバイトコードを変換するアスペクト指向言語に対応していない．ソースコードを変換するアスペクト指向言語なら対応しているかもしれないが, コメントを大量に書かなければならず, 大変である．また, hbm2ddl ではデータベースを作成し直した時, 一からテーブルを作成し直してしまうので元々入っていたデータが全部なくなってしまう．前のデータを使うにはテーブルをダンプしバックアップをしてから新しく作ったテーブルに入れる必要があり, 手間がかかるというデメリットもある．

本システムは以上のデメリットを克服したツールである．つまり, 既存の自動生成ツールのメリットに加えて以下のメリットがある．

- バイトコードを変換するアスペクトに対応している

- テーブルにカラムを追加・削除しても, 削除したカラムとは無関係な情報は失われない．

3.3　仕様と記述例

本システムは GluonJ を拡張し, AspectualStore に対応するように作成した．以下では本システムの使用及び記述例について述べる．

3.3.1 設定ファイル

設定ファイルは, 基本的に AspectualStore での設定ファイルと同じものを使う. つまり, aspectualstore.cfg を AspectualStore の仕様通りに設定すればよい. ここで, AspectualStore の aspectualstore.cfg の主な仕様部分を説明する.

- connection.driver
  使用するドライバー名を指定する.
  PostgreSql を使用する場合は, org.postgresql.Driver と指定する
- connection.url
  データベースがある場所を指定する
- connection.user
  データベースにアクセスするためのユーザー名を指定する
- connection.password
  データベースにアクセスするためのパスワードを指定する
- class.directory
  クラスファイルがある場所を指定する
- model.package
  永続クラスが保存してあるフォルダを指定する

本システムを使用する場合には, 上記の設定に加えて, 以下のプロパティを追加する必要がある.

- aspectualstore.persistent.filter [1]
  永続クラスにバイトコード変換をするクラスを指定する.
  aspectq.aspectualstore.AQPersistentFilter を指定する.
- aspectq.glue
  永続クラスにウィーブするアスペクトを指定する.

また, Tomcat が入っているフォルダのパスを TOMCAT とすると, 以下のように aspectq.jar, aspectq-common.jar, gluonj.jar を指定されたフォルダに入れなければならない.

- TOMCAT/server/lib/
  aspectq.jar

[1] ロードタイムウィービングにて本システムを組み込むときに設定が必要となる.

- TOMCAT/common/lib
  aspectq-common.jar, gluonj.jar

aspectq-common.jar には, @ColumnDef アノテーションなど WEB アプリケーション内で使うクラスが入っている.

3.3.2　@PersistClass アノテーション

永続化クラスであるということを明示させるために, 永続化クラスには @PersistClass というアノテーションをつける. これは AspectualStore の仕様と同じである.

JavaBean の形をしたクラスは永続化クラスに限ったものではない. よって JavaBean クラスの形をしていたからと言ってデータベースにテーブルを作るようなことをしてはいけない. よって永続化クラスかそうでないクラスかどうかをチェックしなければならない. チェックの方法として, @PersistClass アノテーションを利用している. つまり, 永続化クラスとして使用したい場合は, 永続クラスとして扱いたいクラスに @PersistClass アノテーションをつけ, そうでない場合は @PersistClass アノテーションをつけなければよい. 下にコード例を記す. このコード例では, PostGlue クラスは @PersistClass アノテーションがついているので永続化クラスと判断される. それに対し, IndexActionGlue クラスは @PersistClass アノテーションがついていないので永続化クラスではないと判断される.

```
1  @Glue public class PostGlue {
2
3    @Refine @PersistClass
4    public static class PostGlue extends models.Post{
5      private String address;
6      public String getAddress(){
7        return address;
8      }
9      public void setAddress(String address){
10       this.address = address;
11     }
12   }
13
14   @Refine
15   public static class IndexActionGlue
16       extends main.IndexAction{
17     private String memo;
18     public String getMemo(){
19       return memo;
20     }
21     public void setMemo(){
22       this.memo = memo;
23     }
```

```
24    }
25  }
```

3.3.3 @ColumnDef アノテーション

本システムでカラムを定義する場合, カラムと関連付けられたフィールドに @ColumnDef アノテーションを付加する. @ColumnDef アノテーションには, 型の定義・ NULL を許すかどうか・カラムを削除するタイミングなどを記述する.

@ColumnDef アノテーションの nullable プロパティには, カラムが NULL を許すかどうかを指定する. 例えば, nullable = false と指定した場合, カラムを定義する際の SQL 文に NOT NULL が付加されたことを意味する. デフォルトではこの値は true であり, 定義されたカラムは NULL を許すものと解釈する.

さらに, @ColumnDef アノテーションには, ユーザーがカラムに任意の情報を付け加えられるように extra プロパティが存在する. 例えば, MySQL などのデータベースサーバーでは, 値を自動的に 1 つ増やす auto_increment を指定できる. auto_increment されるカラムを作成する場合には, extra = "auto_increment" と指定すればよい.

型の定義方法

本システムでは, 2 通りの型の指定方法を提供する. 本システムで提供する型の定義を使用する方法と, ユーザーが直接型の定義を記述する方法の 2 つである.

本システムで提供される型を使用する場合, @ColumnDef アノテーションの type プロパティに, Type 列挙型で定義された定数を指定する. Type 列挙型に定義されている定数の一覧を表 3.1 に示す.

表 3.1 に示された型を使用する場合には, 必要な引数を @ColumnDef アノテーションに与える必要がある. 例えば, NUMERIC を使用する場合には, 精度と位取りを指定しなければならない. この場合の記述方法は次の通りである.

```
1  @ColumnDef(type=Type.NUMERIC, pricision=24, scale=23)
2  private BigDecimal pi;
```

型の定義に必要な引数が与えられていない場合, 本システムはエラーを表示して終了する.

カラムに表 3.1 に示された型以外の型を使用する場合には, @ColumnDef アノテーションの typeDef プロパティに型の定義を直接記述することができる. 例えば, ENUM 型の group というフィールドを追加したい場合, アスペクト内に次のようなフィールドを作成すればよい.

定数	データの種類	必要な引数
SMALLINT	整数	なし
INTEGER	整数	なし
BIGINT	整数	なし
FLOAT	浮動小数点数	なし
REAL	浮動小数点数	なし
DOUBLE_PRECISION	浮動小数点数	なし
DECIMAL	固定小数点数	precision, scale
NUMERIC	固定小数点数	precision, scale
CHAR	固定長文字列	length
VARCHAR	可変長文字列	length
NCHAR	多国語固定長文字列	length
NCHAR_VARYING	多国語可変長文字列	length
CLOB	バイナリ文字列	length
NCLOB	多国語バイナリ文字列	length
BLOB	バイナリ	length
DATE	日付	なし
TIME	時刻	なし
TIMESTAMP	タイムスタンプ	なし

表 3.1: Type 列挙型の定数一覧

```
1 @ColumnDef(typeDef="ENUM('admin', 'foo', 'hoge')")
2 private String group;
```

3.3.4 カラムの削除

本システムでは, @ColumnDef アノテーションの deleteOn プロパティに値を設定することで, 追加されたカラムがいつ削除されるかを指定することができる. deleteOn プロパティには, DeleteTiming 列挙型の定数を指定する. 指定できる定数を表 3.2 に示す.

deleteOn プロパティに DeleteTiming.EXIT が指定された場合, 本システムは Runtime#addShutdownHook(Thread) を使用して, シャットダウンフックにカラムの削除処理を追加する. まれに, Java 仮想マシンが異常終了した場合には, この処理は呼び出されないことがあるので, DeleteTiming.EXIT を使用する際には注意が必要である.

定数	削除されるタイミング
EXIT	プログラム終了時
RESTART	次回プログラム実行時
PERSIST	削除しない

表 3.2: DeleteTiming 列挙型の定数一覧

DeleteTiming.RESTART が指定された場合には, 次回プログラム実行時に, 同名カラムがアスペクトによってウィーブされなければ, そのカラムは削除される．また本システムは, deleteOn プロパティが指定されていない場合は, DeleteTiming.RESTART が指定された場合と同等に扱う.

アスペクトによってウィーブされたカラムを削除せず, 保存しておきたい場合には, DeleteTiming.PERSIST を指定すればよい．このようなカラムは, 本システムで削除されることがないので, 削除は手動で行う必要がある．ただし, 後にウィーブされたアスペクトによって削除されるタイミングが上書きされた場合, 本システムは後からウィーブされたアスペクトの記述内容に従ってカラムを削除する.

3.3.5　カラムのデフォルト値の設定

カラムにデフォルト値を設定したい場合はフィールド宣言の時に
private String address = "デフォルトにしたい値";
と記述すればよい.

フィールド宣言を
private String address;
とした場合は初期値は入らない．よって NULL が入る.
private String address = "";
とした場合は初期値に空文字が入る.

3.4　織り込み

この節では, 織り込みについて説明する.

3.4.1　実行前

本システムは実行前の織り込みをサポートしている．本システムをウィーブすると, まずデータベーススキーマの変更が行われ, その後にアスペクトがウィーブされる．これは GluonJ のようなアスペクト指向言語のウィー

ブと同じである．GluonJ 等の一般のアスペクト指向言語と違うところは，永続クラスとデータベーススキーマとの整合性を自動で整えてくれるところである．

3.4.2　ロードタイムウィービング

本システムでは，Tomcat 上で本システムを使用するために，独自のクラスローダーを提供している．クラスローダーは，クラスをロードするときにアスペクトを自動的にウィーブするため，ユーザーはアスペクトの記述を書き換えるだけでデータベースとの一貫性を保つことができる．

Tomcat 内で本システムのクラスローダーを使用するためには，Tomcat の設定ファイル (context.xml) に次のように記述する．

```
1 <Context privileged="true">
2   <Loader
3     className="AQLoader"
4     loaderClass="AQClassLoader" />
5 </Context>
```

上のように記述すると，Tomcat のクラスローダーが変更され，本システムが有効化される．

第4章　実装

4.1　処理の全体の流れ

本システムは, アスペクト指向言語である GluonJ を利用して永続クラスを修正すると, 自動でデータベースを修正してくれるシステムである. 本節では, 自動でデータベースを修正する部分について, どのように実装したかを説明をする.

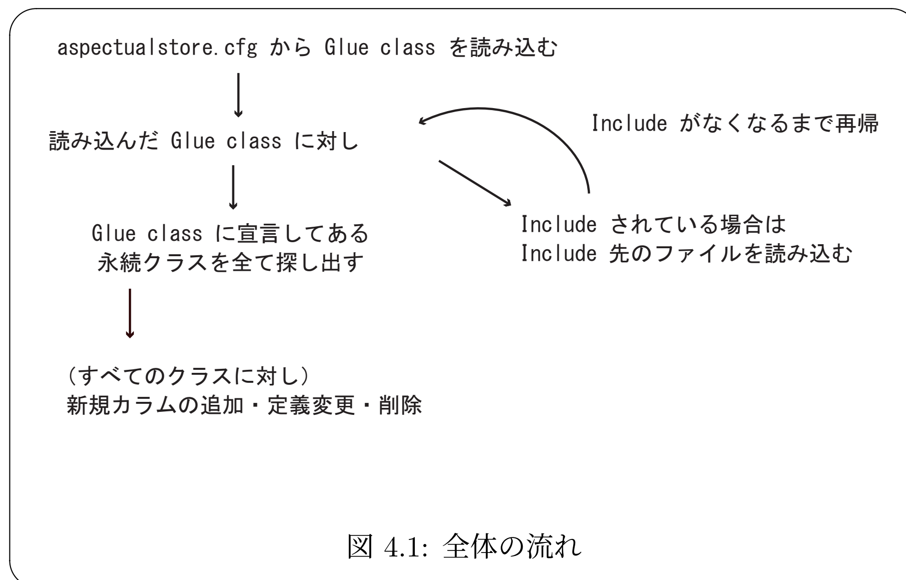

図 4.1: 全体の流れ

まず, 図 4.1 は処理の全体像である. この図のように, 設定ファイルである aspectualstore.cfg を読み込み, 指定された Glue クラスを読み込む. 指定された Glue クラスを元に, ウィーブする Glue クラスをすべて探し出す. ウィーブする Glue クラスをすべて読み込み, 永続クラスを探し出す. これは @PersistClass アノテーションがついているかどうかで判断する. 永続クラスを探し出したら, データベースの更新を行う. データベースの更新時の処理内容については, 第 4.1.1 節で説明する.

4.1.1　データベースの更新

データベースの更新では, データベースやアスペクトから必要な情報を取得し, その情報を元にデータベースの修正を行っている. コードは以下のようになっている.

```
1  /* 現在のカラム一覧 */
2  Set<DBColumn> currents = DBColumn.getColumns(clazz, con);
3  /* アスペクト内に定義されたカラム */
4  Set<RefineColumn> refines = getAddColumns(clazz);
5  /* 前回のアスペクトで追加され, RESTART が指定されたカラム */
6  Set<PastColumn> pasts = pastTable.getRestartColumns(clazz);
7
8  /* 追加するカラム */
9  Set<RefineColumn> adds = new HashSet<RefineColumn>();
10 adds.addAll(refines);
11 adds.removeAll(currents);
12 /* カラム定義を変更するカラム */
13 Set<RefineColumn> modifys = new HashSet<RefineColumn>();
14 modifys.addAll(refines);
15 modifys.retainAll(currents);
16 /* 削除するカラム */
17 Set<PastColumn> deletes = new HashSet<PastColumn>();
18 deletes.addAll(pasts);
19 deletes.removeAll(refines);
20
21 String table = DBUtil.getTableName(clazz);
22
23 /* カラムの追加 */
24 addColumns(table, adds);
25 /* カラムの変更 */
26 modifyColumns(table, modifys);
27 /* カラムの削除 */
28 deleteColumns(table, deletes);
```

上記のコードの 1 ～ 6 行目が必要な情報の取得に対応する部分である. ここでは, 以下の 3 つの情報を取得している.

A データベースから, 現在あるテーブルとカラムの情報の取得

B ウィーブするアスペクトから, 永続クラスに追加するカラムの情報の取得

C 前回アスペクトで永続クラスに追加され, deleteOn プロパティで RESTART が指定されたカラムの情報の取得
(データベース内の管理テーブルより取得)

8 ～ 19 行目で, 追加するカラム・カラムの定義を変更するカラム・削除するカラムを, それぞれ Set に格納している. 追加するカラム・定義変更をするカラム・削除するカラムは以下のように決めている (図 4.2 を参照).

- 新規追加するカラム
  $\bar{A} \cap B$ に対応するカラム
- 定義変更するカラム
  $A \cap B$ に対応するカラム
- 削除するカラム
  $\bar{B} \cap C$ に対応するカラム

21 行目ではテーブル名を取得している．テーブル名と Set に格納した情報を元に, データベースの修正を行っている (23 ～ 28 行目).

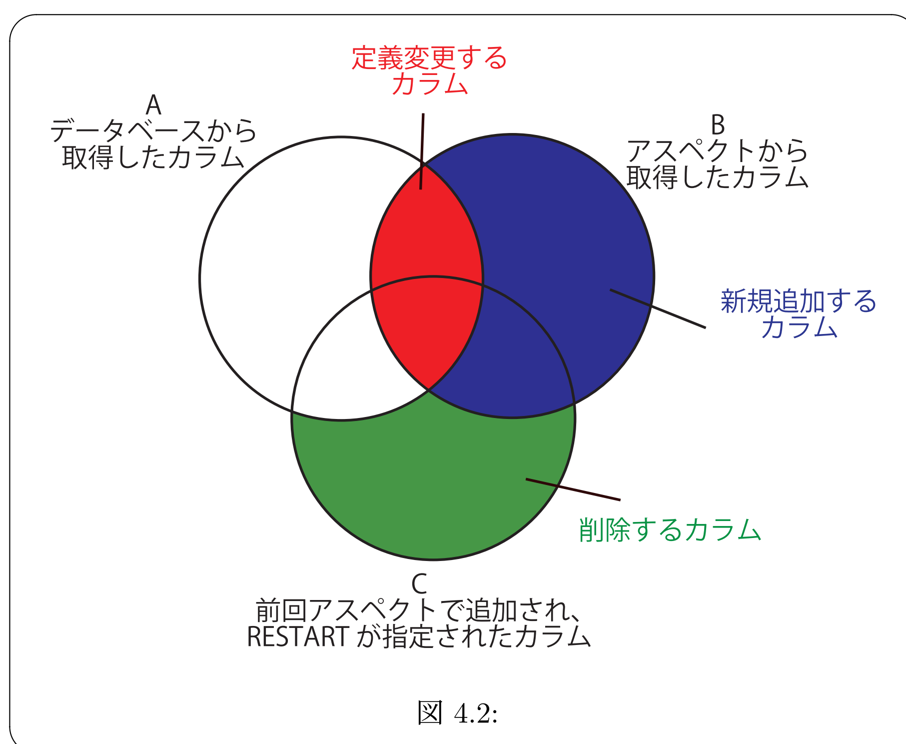

図 4.2:

4.2　AspectualStore の拡張

Tomcat が実行されると，AspectualStore が Javassist を使用して，永続クラスを AspectualStore に対応した永続クラスに変換しているが, AspectulaStore の永続クラスに変換する前に, Filter を通すようにした．この Filter では, 以下の 2 つの処理を行っている．

- データベースの更新
  トランザクションを開始し，第 4.1.1 節で説明したデータベースの

更新を行う. データベースの更新が終わったら, トランザクションを終了する.

- アスペクトのウィーブ
  GluonJ のロード時の織り込みを利用して実装している.

具体的には, AspectualStore の aspectualstore.javassist.EntityGenerator class の pickPersistent メソッドを拡張した (以下のコードを参照). pickPersistent メソッドの最初で, Filter の初期化を行う (1 ～ 12 行目). 次に, Filter がセットされていたら beforeGenerate メソッドを実行するように拡張した (19 ～ 20 行目). beforeGenerate メソッドでデータベースの更新とアスペクトのウィーブを行っている.

```
1  PersistentFilter filter = null;
2  ApplicationEnv env = Environment.getApplicationEnv();
3  String filterName =
4    env.get("aspectualstore.persistent.filter");
5  if (filterName != null) {
6    try {
7      ClassLoader loader =
8        PersistentFilter.class.getClassLoader();
9      Class<?> filterClass = loader.loadClass(filterName);
10     filter = (PersistentFilter) filterClass.newInstance();
11   } catch ...
12 }
13
14 File[] files = directory.listFiles();
15 for (int i = 0; i < files.length; i++) {
16   ...
17   if (isPersist(cc)) {
18     ...
19     if (filter != null)
20       cc = filter.beforeGenerate(cc);
21     // AspectualStore の実装
22     PersistCtClass pcc = new PersistCtClass(cc, schema);
23     persistClasses.put(fileName, pcc);
24   }
25 }
```

beforeGenerate メソッドのコードは以下のようになっている. まず, トランザクションを開始し第 4.1.1 節で説明したデータベースの更新を行う. このデータベースを更新している最中に異常が発生など Exception が発生してしまったら, rollback をしてデータベースを更新する前の状態に戻す. 異常発生など起こらずデータベースの更新が正常に終了したら, commit をしてデータベースに反映させる. 最後にアスペクトのウィーブを行っている.

```
1  /* トランザクションの開始 */
2  boolean autoCommit;
3  try {
4    autoCommit = con.getAutoCommit();
5    con.setAutoCommit(false);
```

```
 6  } catch ...
 7
 8  try {
 9    //データベースの更新を実行
10    try {
11      con.commit();
12      con.setAutoCommit(autoCommit);
13    } catch ...
14  } catch (AspectQException e) {
15    try {
16      con.rollback();
17      con.setAutoCommit(autoCommit);
18    } catch ...
19    ...
20  }
21  try {
22    /* ウィーブ */
23    CtClass weaved = weaver.transform(clazz.getName());
24    return weaved;
25  } catch ...
```

第5章　応用例と実験結果

WEB アプリケーションである掲示板を作成し, 本システムを使った場合と使わない場合とで拡張機能の抜き差しの容易さについて比較を行った. 掲示板を作成するにあたり, WEB に関する部分は Servlet, JSP, Struts, これらのコンテナとして Tomcat を利用した. また, 永続システムの部分は AspectualStore, PostgreSQL を利用した.

5.1　実装するアプリケーション

今回応用例として作成したアプリケーションである, 掲示板について説明する. 図 5.1 に今回作成した掲示板の概観を示す.

この掲示板は, トップページで今まで書き込まれた情報をすべて見ることができる. また, 名前と本文を入力することができるフォームがあり, 書き込みを行うことができるようになっている. このアプリケーションの拡張機能として, アドレス欄の取り外しを考える. 拡張機能をつけて実行した場合は, 名前と本文の入力欄に加えて, アドレス欄も表示される. それに対して, 拡張機能を外して実行した場合は, 入力欄は名前と本文だけで, アドレス欄は表示されない.

システムの全体の流れは図 5.2 のようになっている. まず, トップページである index.jsp では, データベースから今までの記事の情報を取り出し, 表示している. index.jsp で, 入力フォームに書き込みを行い書き込みボタンを押すと, confirm.jsp に飛ぶ. このページでは, index.jsp で書き込んだ内容の確認である. もし書き込みの内容が正しく, 記入ボタンを押すと, データベースにアクセスして新しい情報を書き込み, complete.jsp に飛ぶ. ここでは, 正しくデータベースにアクセスして情報が書き込まれたかどうかが表示される.

以下に, 主なファイルの説明を示す.

- index.jsp
  トップページ. 入力フォームがある.

- confirm.jsp
  入力フォームで入力した内容の確認画面.

図 5.1: 掲示板アプリケーション

- complete.jsp
  データベースに接続し，正しく書き込めたかどうかを表示．

- IndexAction.java
  index.jsp に対応するアクション．投稿されたコメントの一覧を index.jsp に渡す．

- ConfirmAction.java
  confirm.jsp に対応するアクション． 投稿されたコメントの内容を confirm.jsp に渡す．

- CompleteAction.java
  complete.jsp に対応するアクション．データベースを更新して，complete.jsp を表示する．

- ConfirmForm.java

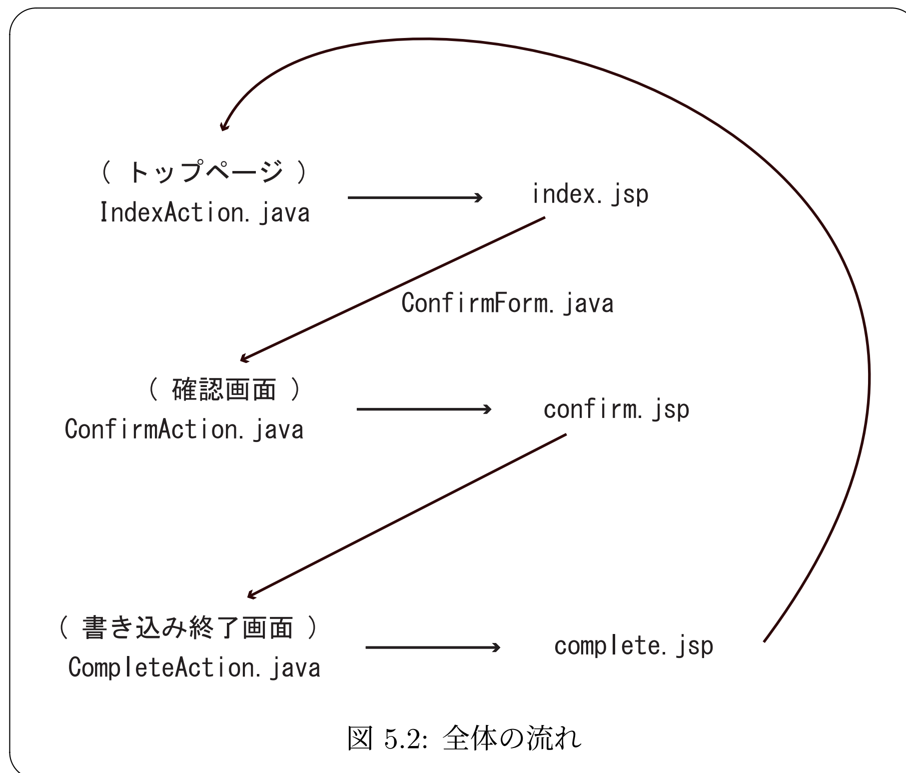

図 5.2: 全体の流れ

入力フォームに書かれた内容を保持.

- Flag.java
  入力フォームに表示する項目を, true または false で保持.

- Post.java
  永続クラス. データベースの post テーブルに対応.

また, Java クラスのファイル構造は以下の図 5.3 のようになっている. 背景が水色のものはフォルダを意味している.

トップページの index.jsp で入力フォームを表示する項目を, Flag.java で決定している. Flag.java には, boolean name, boolean content, boolean address などのフィールドがあり, true または false が入っている. index.jsp は以下のようなコードになっており, if 文の結果 (test="$flag.address" の結果) によって入力フォームを表示するかどうかが変わる. 例えば, Flag.java の中で address = true にすると, index.jsp で入力フォームにアドレス欄が表示される.

```
1 <c:if test="${flag.address}">
2   <tr>
3   <td width="100" colspan="2">
4     Address : ${post.address}
5   </td>
```

```
6    </tr>
7  </c:if>
```

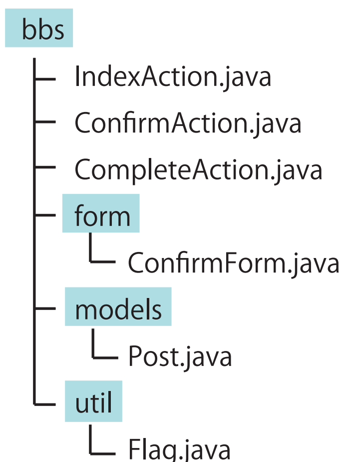

図 5.3: ファイル構造

以下では既存の方法で拡張機能を実装した場合と, 本システムを用いて実装した場合の比較を行う.

5.2　変更ファイル数の比較

本節では, 既存の実装方法と本システムを用いた場合の変更ファイル数を比較する. 今回は拡張機能として, 以下の図 5.4 のように, 入力フォームにメールアドレス欄とパスワード欄を表示することを考える. 既存の実装方法には 2 つの方法があるといえる. オブジェクト指向言語で実装する場合と, アスペクト指向言語を用いて実装する場合の 2 通りである.

5.2.1　オブジェクト指向言語で実装する場合

まず, オブジェクト指向言語で実装する場合について述べる. オブジェクト指向言語で開発する場合, 修正する箇所は以下のようになる.

- IndexAction.java
  記事一覧を表示する際に, address に関しても他のカラム同様に取得し, Map に入れる.

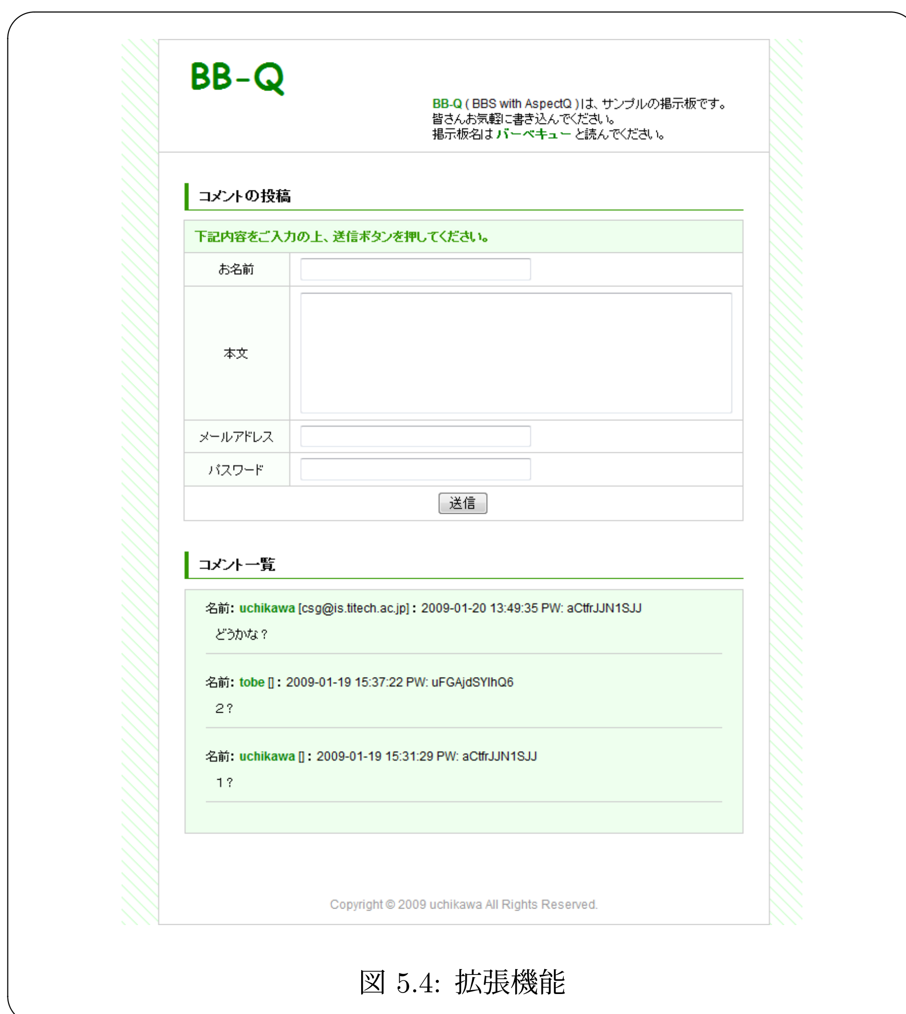

図 5.4: 拡張機能

- ConfirmAction.java
  ConfirmForm から入力フォームに入れた値を受け取り, Map に変換して CompleteAction.java に渡す. この時に address の値も他同様 Map に入れる.

- CompleteAction.java
  データベースに情報を入れる際に, 他カラム同様 address の値も入れる.

- ConfirmForm.java
  address フィールドおよび getter, setter を追加.

- Post.java
  address フィールドの追加.

- データベースの post テーブル
  address カラムの追加.
  例えば, コンソール上で以下のように実行.
  ALTER TABLE post ADD COLUMN address varchar(200);

以上のように 5 個のファイルとデータベースを修正しなければならない．また, フォルダ数は 3 個に及ぶ.

次に, 別の拡張機能としてパスワード入力フォームを追加することを考えよう．この場合も, アドレス欄を追加するとき同様 5 個のファイルとデータベースを修正しなければならない．フォルダ数も同じく 3 個に及ぶ.

最後に, 拡張機能であるパスワード入力フォームを削除することを考える．この場合は, 追加した箇所をすべて削除しなければいけないため, 拡張機能追加時と同様 5 個のファイルとデータベースを修正しなければならない．フォルダ数も同じく 3 個である．データベースからカラムを削除するときは, 例えばコンソール上で
ALTER TABLE post DROP COLUMN password;
と実行しなければならない.

5.2.2 既存のアスペクト指向言語で実装する場合

本項ではアスペクト指向言語を用いて実装する場合について考える．修正する箇所は以下の通りである.

- Glue クラス
  Java ファイルである IndexAction.java, ConfirmAction.java, CompleteAction.java, ConfirmaForm.java, Flag.java, Post.java への修正は, アスペクトとして Glue クラスにまとめることが可能.

- データベースの post テーブル
  オブジェクト指向言語使用時と同様, コンソールから修正する必要がある.

以上のように, 1 個のファイルとデータベースを修正しなければならない．フォルダ数は 1 個となる．別の拡張機能を追加する際も, 1 個のファイルとデータベースを修正しなければならない．フォルダ数は 1 個となる．削除する場合も, 1 個のファイルとデータベースを修正しなければならない．フォルダ数は 1 個となる.

5.2.3　本システムを利用した場合

最後に, 本システムを利用して実装した場合について述べる. 本システムを用いた場合は, 永続クラスである Post.java にフィールドを追加すると, データベースも自動で修正される. よって, 修正する箇所は以下の通りである.

- Glue クラス
  既存のアスペクト指向言語を用いて実装する場合同様, Java 言語で書かれたクラスへの修正は Glue クラスにまとめることが可能.

- aspectualstore.cfg
  本システムを使うために設定を行う必要がある. ただし, 最初の 1 回だけ設定すればよい.

本システムを用いた場合, 修正箇所は 2 個のファイルとなり, フォルダ数は 2 個となる. 別の拡張機能を追加する場合は, aspectualstore.cfg は修正しなくてよいため, Glue クラスの 1 ファイル, 1 フォルダのみの修正となる. 拡張機能を削除する場合も, 1 ファイル, 1 フォルダのみの修正となる.

	オブジェクト指向言語	アスペクト指向言語	本システム
ファイル数 (個)	6 / 6 / 6	1 / 1 / 1	2 / 1 / 1
DB の修正	要 / 要 / 要	要 / 要 / 要	不 / 不 / 不
フォルダ数 (個)	3(4) / 3(4) / 3(4)	1(2) / 1(2) / 1(2)	2 / 1 / 1

表 5.1: 修正ファイル・フォルダ数

これらの結果をまとめたものが表 5.1 である. スラッシュでの区切りは, 1 回目の拡張機能の追加の場合 / 2 回目以降の拡張機能の追加の場合 / 拡張機能の削除の場合 を意味している.

これを見ると, 既存のアスペクト指向言語に比べ, 修正ファイル数の増加はあったものの, データベースを含めたモジュール化は成功しているといえる. また, 運営後の WEB アプリケーションでは, 追加機能を試験的に試すことが頻繁に行われる. もし, 機能の追加と削除を一連の流れで行うとすると, オブジェクト指向で機能を追加する場合には, 複数ファイルを編集しなければならないため, 修正し忘れなどのミスが生じやすい. 本システムを使用すると, オブジェクト指向で機能を追加する場合に比べ, 修正ファイル数が格段に少なくなっていることもわかる.

5.3　平均応答時間の比較

本システムを用いて実行した場合とオブジェクト指向言語を用いて実行した場合とで, 平均応答時間の比較を行った. 図 5.5 は、この結果である.

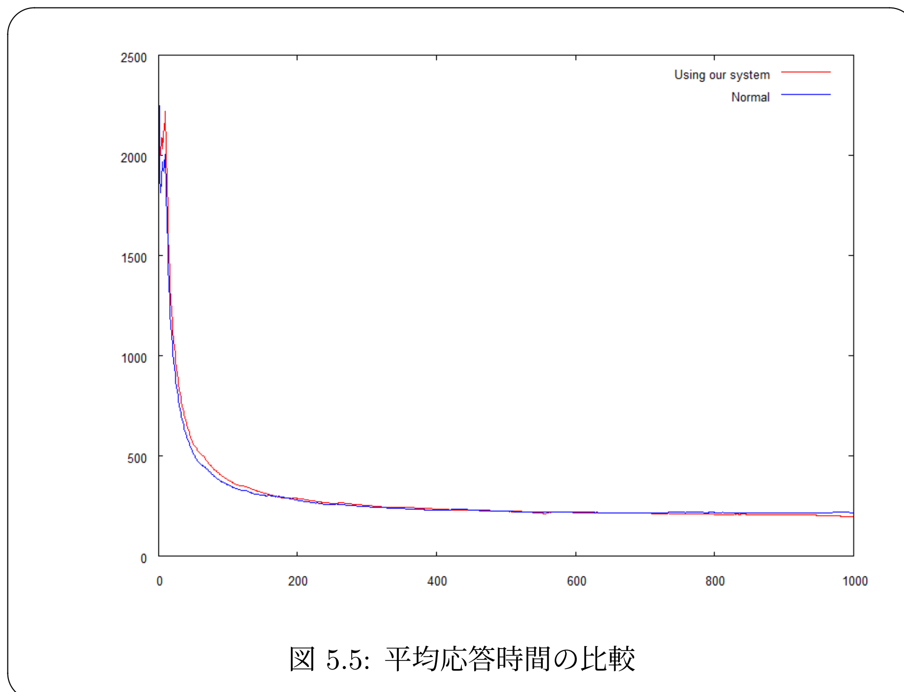

図 5.5: 平均応答時間の比較

縦軸の単位は ms で, 横軸はサンプル数を表している. 赤い線が本システムを使った場合で, 青い線が本システムを使わなかった場合になっている.

図を見てわかるとおり, 最初はデータベースの修正がある分, 本システムを使わなかった場合に比べてオーバーヘッドがあるものの, 起動してしばらく経った後はシステムを使った場合と使わなかった場合とでほとんど差は見られなかった.

第6章　まとめ

本稿では, WEB アプリケーションのモジュール化のための言語について提案した. 本システムでは, 永続クラスをアスペクト指向言語である GluonJ を使って修正すると, 修正した永続クラスに対応するデータベースのテーブルを自動で修正するシステムである. これにより, ユーザーはデータベースまでを含めモジュール化することが可能となった.

WEB アプリケーション開発において, データの保存先としてデータベースを使うことは多々ある. データベースの設計変更を伴う拡張機能をモジュールとして追加する場合を考える. オブジェクト指向言語で拡張しようとすると, 修正箇所が多岐に渡ってしまい, モジュール化出来ない. アスペクト指向言語で拡張しようとしても, Java に関する部分はモジュール化出来ても, データベースまではモジュール化出来ない. 本システムを利用すると, データベースにまでまたがる関心事をモジュール化することができ, 修正箇所は Glue クラスと設定ファイルのみで済む. 設定クラスは最初の 1 回のみ修正が必要であるが, 2 回目以降は修正する必要はなくなる.

本システムではアスペクトで永続クラスのフィールドが編集されると, ウィーブ時に自動で永続クラスに対応するようにデータベースを修正するシステムである. データベースに追加したカラムを削除するタイミングを 3 通り用意している. プログラム終了時, 次回プログラム実行時, 削除しない, の 3 通りである. これにより, ユーザーはアスペクトをアンウィーブしても一度データベースに入れたデータを保存しておけるようになった.

また本システムは, Tomcat に組み込んだことにより, Tomcat の起動または再起動で実行できるようにした. まず Tomcat が起動すると, AspectualStore が Javassist を使用して, 永続クラスを AspectualStore に対応した永続クラスに変換しているが, AspectulaStore の永続クラスに変換する前に, Filter を通すようにした. この Filter 内では, 永続クラスとデータベースが一貫性を保つようにデータベースを修正し, アスペクトをウィーブしている.

また本稿では, 応用例として掲示板を作成し, 拡張機能として新しい入力フォームの追加と削除を行った. このとき, 最終目標であるモジュール化, つまりデータベースの修正をなくすことが達成されたことがわかった.

実験では本システムを使った場合と使わなかった場合とでは実行時間にあまり差がなく, スピードが落ちないことが証明された.

参考文献

[1] Chiba, S.: GluonJ, http://www.csg.is.titech.ac.jp/projects/gluonj/.

[2] Chiba, S. and Ishikawa, R.: Aspect-Oriented Programming beyond Dependency Injection, *In Proceedings of the 19th European Conference on Object Oriented Programming (ECOOP 2005)*, pp. 121–143 (2005).

[3] Gregor Kiczales, Erik Hilsdale, J. H. M. K. J. P. and Griswold., W. G.: An Overview of AspectJ, *In Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP 2001)*, pp. 327–353 (2001).

[4] Hibernate: hibernate.org, http://www.hibernate.org/.

[5] IBM: IBM WebSphere ソフトウェア, http://www-06.ibm.com/jp/software/websphere/.

[6] Microsoft: インターネット インフォメーション サービス, http://www.microsoft.com/japan/windowsserver2003/iis/default.mspx.

[7] Oracle: BEA WebLogic Server, Application Server, App Server, Java Application Server, Web Application Server, http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/web%logic/server/.

[8] Rashid, A. and Chitchyan, R.: Persistence as an aspect, *In Proceedings of 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pp. 120–129 (2003).

[9] Sun: Java SE Technologies - Java Database Connectivity(JDBC) -, http://java.sun.com/javase/technologies/database/index.jsp.

[10] the Apache Software Foundation: Apache Tomcat, http://tomcat.apache.org/index.html.

[11] the AspectJ project: The AspectJ Project, http://www.eclipse.org/aspectj/.

[12] 千葉滋: アスペクト指向入門 Java・オブジェクト指向から AspectJ プログラミングへ, 技術評論社 (2005).