平成18年度 学士論文

永続システムのための
アスペクト指向言語

東京工業大学 理学部 情報科学科
学籍番号 03-0368-6
内河　綾

指導教員
千葉 滋 助教授

平成19年2月7日

概要

今日、リレーショナルデータベースを利用したアプリケーションでは永続システムを利用することが多くなってきている。永続システムは、リレーショナルデータベースとクラス (永続クラス) をマッピングし、永続オブジェクトとデータベース内のレコードの一貫性 (データの一貫性) を自動的に維持する。永続クラスのスキーマとデータベーステーブルのスキーマの一貫性 (スキーマの一貫性) は手動または外部ツールを利用して維持する必要がある。

永続システムを用いたアプリケーション開発では、永続クラスに関わる機能を追加や削除する時に問題が発生してしまう。まず、永続クラスにフィールドやそれに関する操作を追加する場合、ソースコードを直接編集しなければならない。これはプログラムの再利用性や保守性を低下させる原因となる。またスキーマの一貫性を維持するためには、ソースコードを編集するだけではなく、外部ツールを用いるか手動でデータベーススキーマを変更する必要がある。

本研究では、これらの問題を解決するために、永続システムに対応した Java 言語用のアスペクト指向システムを提案する。本システムでは、永続クラスに追加する機能をアスペクトとして新たな別ファイルに記述するため、既存のソースコードを変更する必要はない。また、永続クラスにフィールドが追加された場合、それに対応するカラムをデータベーステーブルに自動的に追加する機能を実現した。これにより、永続クラスへの追加機能がモジュール化され、その機能の追加や削除がアスペクトをウィーブ・アンウィーブするという単一の作業で可能となった。

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既にデータベースにあるカラムと比較し、スキーマの一貫性を保つようにカラムの追加と削除を行う。アスペクトのクラスファイルへのウィーブはその後自動的に行われる。

既存のアスペクト指向言語では、ソースコードを直接変更せずに永続クラスにフィールドを追加することは出来る。しかし、スキーマの一貫性を保つためには、データベーススキーマをアスペクトに合わせ手動で変更するかアスペクト内にユーザーが書いたコードで変更しなければならなかっ

た。ソースコードレベルで変換を行うアスペクト指向言語の場合は、外部ツールを利用することでスキーマの一貫性を維持することが出来るが、永続クラスとデータベースの変更を別々の作業として行う問題がある。

謝辞

本研究を進めるにあたり、研究の方向付けや論文の組み立て方についての助言をしていただいた千葉滋助教授に大変感謝いたします。

論文のスタイルファイルを作成していただいた光来健一助手、アスペクト指向言語について基礎から教えていただいた西澤無我氏、データベースに関する助言をしてくださった柳澤佳里氏に感謝いたします。

また、青木康博氏にはシステムの設計や実装、そして論文の組み立て方など多岐にわたり親身になって指導していただきました。心より感謝いたします。また熊原奈津子氏、栗田洋輔氏には論文の構成について様々な助言をいただきました。感謝いたします。また、卒業研究を行う上で励ましていただいた同研究室の皆様に感謝いたします。

目 次

図 目 次

表 目 次

第1章　はじめに

近年、リレーショナルデータベースを利用したアプリケーションは数多く存在する。リレーショナルデータベースにアクセスする方法としてJavaがサポートしている JDBC やデータベースのレコードとオブジェクトを自動でマッピングする永続システム等がある。現在は永続システムが主流となってきている。永続システムとは、データベーステーブルのカラムと永続クラスのフィールを対応づけるシステムのことである。今までデータベースからデータを取得する時は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。

永続システムを用いたアプリケーション開発を行った場合、永続クラスに関係のある機能の追加や削除には困難を伴う。つまり、機能の抜き差しを容易に行うことが不可能である。永続システムでは永続クラスのフィールドとデータベースのカラムは対応していることが前提条件となっている。よって、永続クラスにフィールドを追加 (削除) した場合はデータベースにもカラムを追加 (削除) しなければならない (スキーマの一貫性)。しかし、既存のオブジェクト指向技術ではソースコードを直接編集する必要がある。これはプログラムの再利用性や保守性を低下させる原因となる。また、スキーマの一貫性を保つためには、手動または既存ツールを用いてデータベーススキーマを変更する必要がある。

本研究では、これらの問題を解決するために、永続システムに対応したJava 言語用のアスペクト指向システムを提案する。本システムでは、永続クラスに関係のある機能を追加する場合、追加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ソースファイルを変更する必要がなくなる。また、アスペクトで永続クラスに追加されたフィールドはデータベースに対応するカラムがなかったら自動で作成される。よって、スキーマの一貫性は自動で保たれる。本システムにより、永続クラスに関わる処理をアスペクトでモジュール化出来るようになり、機能の抜き差しが容易に出来るようになった。

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。まずユーザーが java ファイルをコンパイルし class ファイルに変換したら、class ファイルからアスペクトで永続クラスに追加したフィールドを読み込む。

読み込んだものとデータベースにもともとあるカラムを比較し、追加するカラムと削除するカラムを選び出し、データベーススキーマを変更する。クラスファイルへのアスペクトのウィーブはその後自動で行われる。

以下 2 章では、既存の永続システムとアスペクト指向言語とその問題点について述べ、3 章では本研究で開発したアスペクト指向システムの特徴・仕様について述べる。また 4 章では、本システムの開発にあたって GluonJ を拡張した部分と実装方法について述べる。5 章では、掲示板のアドレス欄の追加削除を本システムを用いて行った場合と既存のツールを利用して行った場合とで比較をしている。最後に 6 章は本論文のまとめとなる。

第2章　永続システムと既存のアスペクト技術

2.1　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、誰々が作った Hibernate や誰々の Spring framework、誰々の EJB 、私たちの研究室で開発されている AspectualStore などがあげられる。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、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。

2.1.1　JDBC

JDBC[7] 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。データベースへのアクセス方法は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より異なる 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 をりようすることにより、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になった。

JDBC を直接利用したプログラム例を図 2.1 に示す。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、SQL が select 文であれば SQL 発行の結果を元にオブジェクトを作成することである。

- 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う。これではどこで発生したエラーなのかがわかりにくい。。。省略？)

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 2.1.2 章で説明する永続システムである。

2.1.2　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との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。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

リレーショナルデータベースを扱う際に起こる主な問題の中に、リレーショナルデータベースとオブジェクト指向のオブジェクトとのマッピングがあげられる。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。

```
public List findOwnerPet() throws ApplicationException{
  ...
  String sql =
    "SELECT name, content FROM post";

  Connection con = null;
  PreparedStatement ps = null;
  ResultSet rs = null;
  try{
    con = ds.getConnection();
    ps = con.prepareStatement(sql);
    rs = ps.executeQuery();
    while(rs.next()){
      String name = rs.getString("name");
      String content = rs.getString("content");
      Post post = new Post(name, content);
      postList.add(post);
    }
    ...
  }catch(SQLException e){
    System.err.println("SQLException が発生しました");
  }finally{
    try{
      if(rs != null)
        rs.close();
    }catch(SQLException e){
      ...
    }
    try{
      if(ps != null)
        ps.close();
    }catch(SQLException e){
      ...
    }
    try{
      if(con != null)
        con.close();
    }catch(SQLException e){
      ...
  }  }
  return postList;
}
```

図 2.1: JDBC を利用してデータベースへアクセスするプログラム例

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。よってどちらが悪いというものではない。両方最適なアルゴリズムである。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。

これは、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出される。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 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、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ない。このマッピングを永続システムは自動化している。よってマッピングを毎回行う必要がなくなった。

2.1.3　Hibernate

Hibernate[5] 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一つである。

XDoclet と SchemeExport

XDoclet[9] は JavaDoc から XML のようなファイルを生成する。XML のようなファイルへは、ソースコードの JavaDoc に書いてあるテンプレートを元に変換している。たとえば以下のようなプログラムを書いて実行すると、図 2.2 のような XML ファイルが生成される。

```
/**
 * @hibernate.class
 * table="CATEGORY"
 */
public class Category{
  ...
  /**
   * @hibernate.id
   * generator-class="native"
   * column="CATEGORY_ID";
   */
  public Long getId(){
    return id;
  }
  ...
  /**
   * @hibernate.property
   */
  public String getName(){
    return name;
  }
}
```

SchemeExport は XML ファイルからデータベーススキーマを生成するものである。

2.1.4　AspectualStore

AspectualStore は本研究室の青木康博氏が開発している永続システムである。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。

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sourceforge.net/
     hibernate-mapping-2.0.dtd">
  <hibernate-mapping>
    <class
      name="org.hibernate.auction.model.Category"
      table="CATEGORY">
      <id
        name="id"
        column="CATEGORY_ID"
        type="long">
        <generator class = "native"/>
      </id>
      <property
        name="name"
        column="NAME"
        type="string"/>
    </class>
  </hibernate-mapping>
```

図 2.2: XDoclet により生成された XML ファイル

- アスペクト指向によるシステムの拡張

  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。[6] 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。

記述例

AspectualStore の記述例について述べる。

まず、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である。この場合は図 2.3 の read() メソッドのように記述する。これは Post テーブルからレコードを全て取り出してくる例である。

```
public List read() {
  List<Post> posts =
    new ArrayList<Post>();
  Session session =
    SessionFactory.getFactory().openSession();
  Transaction t =
    session.beginTransaction();
  try {
    Expression exp = ...
    SelectCriteria q =
      new SelectCriteria(Post.class, exp);
    q.setFetchLevel(FetchLevel.ALL_PROPERTY);
    posts =
      session.list(q);
    t.commit();
  }catch(Exception e) {
    ...
  }
  return posts;
}
```

図 2.3: AspectualStore での select 方法

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図 2.3 の

```
Expression exp = ...
```

の部分を

```
Expression exp =
  ExpressionFactory.ge("id", new Integer(11));
exp =
  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ことである。これは図 2.4 の write メソッドのように記述すれば良い。オブジェクトをセッションにストアするだけなので、リレーショナルデータベースの構文は一切考えなくてよい。

```
public void write(Post post) {
  Session session =
    SessionFactory.getFactory().openSession();
  Transaction t = session.beginTransaction();
  try {
    session.store(post);
    session.flush();
    t.commit();
  }catch(Exception e) {
    ...
  }
}
```

図 2.4: AspectualStore での update 方法

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。これは図 2.5 のように記述する。セッションから削除したい Post オブジェクトを delete すれば良い。

```
...
List list = session.list(q);
Iterator<Post> it = list.iterator();
for(int i = 0; it.hasNext(); i++) {
  Post post = it.next();
  posts.add(post);
  session.delete(post);
}
...
```

図 2.5: AspectualStore での delete 方法

2.2　既存のアスペクト技術

本章では既存のアスペクト指向言語について述べる。

2.2.1　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のである。[11]

2.2.2　AspectJ

AspectJ[4, 8] 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。

2.2.3　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。

以下にプログラム例を示す。

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。以下に @Refine を用いた簡単な例を示す。Book クラスに BookGlue クラスが変更を加えている。

```
package test;
public class Book {
    public String author;
    public void toString() {
        System.out.println("author is " + author);
    }
}
```

```
@Glue class BookGlue {
    @Refine static class Book extends test.Book {
        public String title = "book title";
        public void toString() {
            System.out.println(title);
        }
    }
}
```

この場合、まずフィールド title が付け足され、メソッド toString() が置き換えられる。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。

```
package test;
public class Book {
    public String author;
    public String title = "book title";
    public void toString() {
        System.out.println(title);
    }
}
```

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

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

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。

```
@Glue class PageGetter {
  @Refine static class Diff extends Book {
    public void getPage(int page) {
      System.out.println("Book page is " + page);
    }
  }
}
```

2.3　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

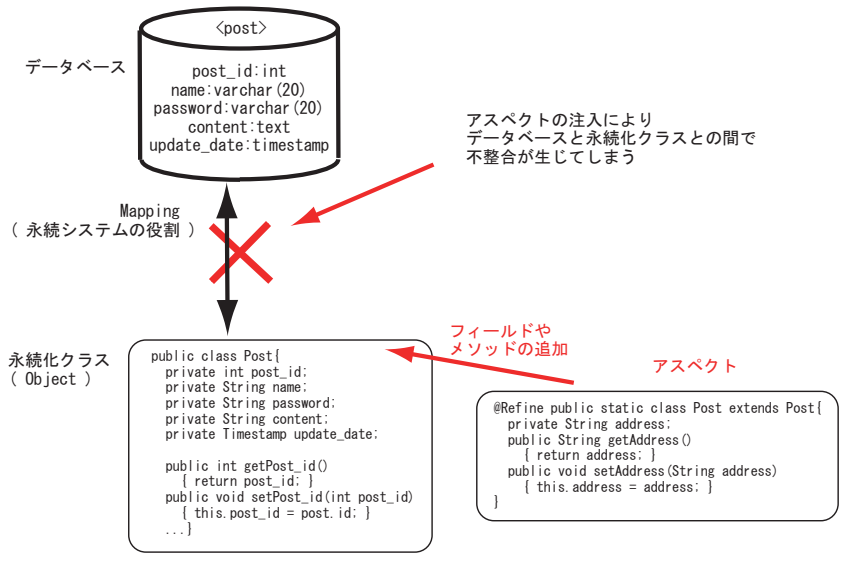

図 2.6: 問題点

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。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 $post_id$, $name$, $password$, $content$, $update_date$ を保持しているものとする。しかし、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。そこで、このアドレス欄を付け加えるにはデータベースに address カラムを追加しなければいけないとする。

もしアスペクトを使わないで実装をすると、後で実装部分を取り外すことになった時に様々な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。後で機能を取り外すかもしれない場合に適している言語が、アスペクト指向言語である。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。

データベースにカラムを追加する際に図 2.6 を見てわかるとおり、アドレス欄を追加するにあたり以下の三つの部分を変更しなければならない。最初にあげた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

上の二つの項目はアスペクトでモジュール化が可能な部分である。しかし、一番最後の項目は Java ではないので、アスペクト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の恩恵を受けにくいと言える。

アスペクト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システムでは、永続システムにカラムを追加する時にインピーダンスミスマッチが起こるだけではなく、アスペクトを削除した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変更しなければならない。アスペクトの一番のメリットは、関心ごとをモジュール化し、機能等の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メリットが生かされない。

XDoclet や SchemeExport を利用すればモジュール化は可能かもしれない。しかし、アスペクトのウィーブとスキーマの変更を別々に実行しな

ければならないという問題がある。また、この場合はソースコードレベルに限った話である。つまり、ソースコードを変更するアスペクト指向言語でないと使えないということになる。現在ではソースコードを変更するアスペクト指向言語は少ない。バイトコードを変換するアスペクト指向言語が主流である。よってこの方法も問題があると言える。

第3章　永続システムのためのアスペクト指向言語

本研究では、アスペクト指向言語を拡張し永続システムに対応する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 2.3 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に対応して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同様に自動で変更が行われる。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。

- データベースのテーブルの変更方法は二通り存在し、どちらを使うかはユーザーが選択可能

  テーブルを変更するにあたり、二通りのパターンを選択可能にする。既存のテーブルにカラムを追加する方法と、既存のテーブルとは別

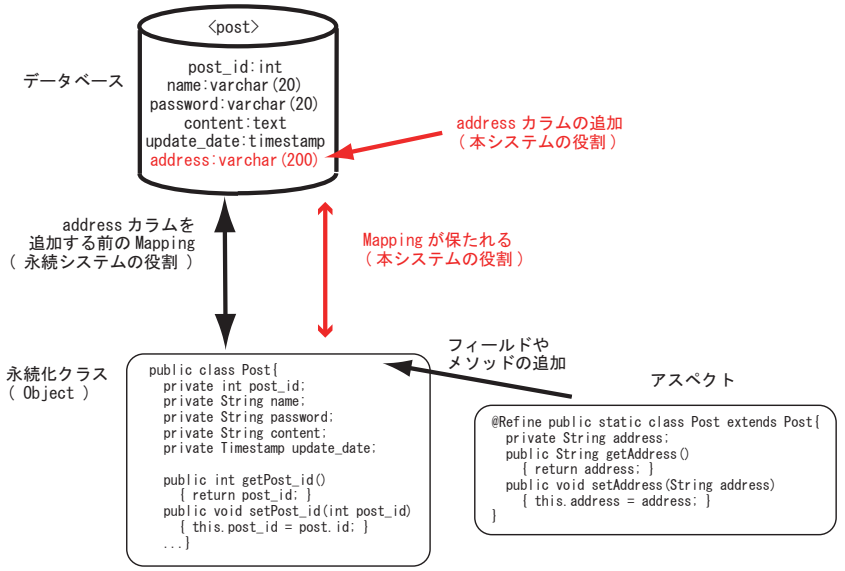

図 3.1: データベースの自動変更

テーブルを作り既存のテーブルと紐付けして使う方法である。これについては第 3.3 節で詳しく述べる。

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同一なものなので、どちらか一つだけ作成したら、もう片方は自動で生成して欲しい。そこで、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。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。データベースを先に作り永続化クラスは自動で生成する方法と、永続化クラスを先に作りデータベースは自動で生成する方法の二つである。

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。

- 削除する時にデータベースの方も変更しなければならない

  データベースの方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。

- 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、まだ永続化クラスに対応していない。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。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生成すると、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 プログラムを実装するにあたって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のほうをいじる必要がない。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

アスペクトで追加してあるので、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　二通りのデータベースの変更方法

第 3.1 章で、データベースを変更するにあたり二通りの方法があると述べた。既存のテーブルにカラムを追加する方法と、既存のテーブルとは違うテーブルを作る方法である。本節では第 2.3 章で利用した掲示板の例を使い、上で述べた二通りの設計について詳しく述べる。

3.3.1　既存のテーブルにカラムを追加する方法

既存のテーブルにカラムを追加する方法では、アスペクトで変更しているクラスに対応するテーブルに直接カラムを追加する。この方法では hbm2ddl とは違い、データベースを作り直すことはしない。元々あるデータベースにカラムを追加するだけである。アスペクトを追加した状態で何度も実行しても、前に入れたデータが初期化されることはない。アスペクトで追加したカラムの情報は新しくデータベースを作成し、そこに保存し

ておく。よって、アスペクトで追加した状態で二回実行したときは、何も起こらない。アスペクトを追加して実行した後にアスペクトを取り外して実行した時は、必要がなくなったカラムだけを削除する。この場合はカラムに入れた情報は失われれる。また、アスペクトで追加したカラムの情報用に作ったテーブルからも、削除したカラムに関するレコードは削除される。

例えばアスペクトで 3.2 のように記述すれば、データベースは図 3.3 のように変更される。赤字が変更された部分である。ただし、一番最後の行は違う場合の例である。

```
@Glue public class PostGlue {
  @Refine @PersistClass
  public static class Post extends models.Post{
    private String address;
    public String getAddress(){
      return address;
    }
    public void setAddress(String address){
      this.address = address;
    }
  }
}
```

図 3.2: アスペクトの記述例

図 3.3 にも書いてあるが、デフォルト値を設定したときとしなかったときとでは、入る値が違う。図 3.2 はデフォルト値を設定しないときの例である。設定するときは図 3.2 で追加したいカラムをフィールドで宣言するときにそのフィールドに初期値を設定すればよい。つまり、この例だと

```
private String address;
```

の部分の代わりに

```
private String address = 'aaa@aaa.com';
```

と書くと、指定した 'aaa@aaa.com' というデフォルト値が設定され、一番最後のレコードのように値が入る。

以下に既存のテーブルにカラムを追加する場合のメリットとデメリットをあげる。

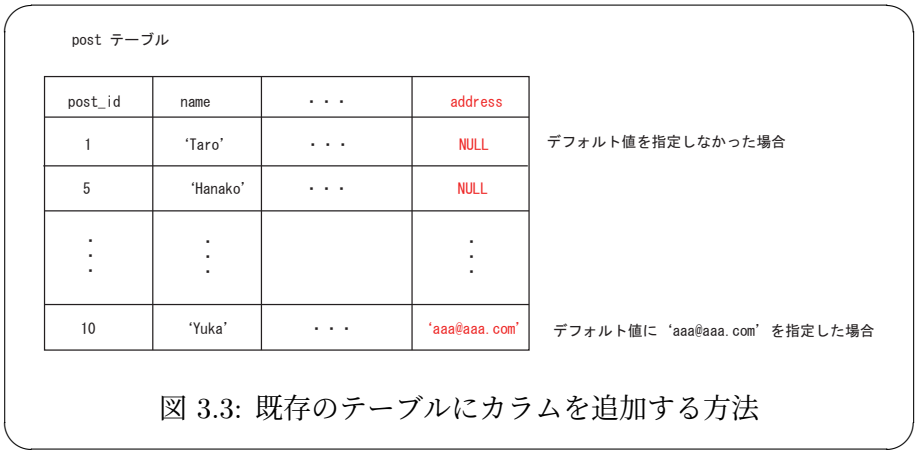

図 3.3: 既存のテーブルにカラムを追加する方法

- メリット
  - 実行速度が既存のテーブルとは別のテーブルを追加する方法に比べて速いと思われる
- デメリット
  - 一度アスペクトを取り外して実行したら、そのアスペクトに関するカラムの情報は失われる

3.3.2　既存のテーブルとは違うテーブルを作る方法

既存のテーブルとは違うテーブルを作る実装では、図 3.4 のように、元々あるテーブルとは違うテーブルを作成してカラムを管理する。ただしこの方法は実装はしていないが、以下のように実装するべきである。

まず、記述方法についてだが、これは既存のテーブルにカラムを追加する時と同様、図 3.2 のように記述する。既存のテーブルをは違うテーブルを作る方法では、アスペクトで記述されたカラムは別テーブルを用意しそこのカラムを追加していく。そこで元々あるテーブルを新しく作成したテーブルを対応付けるために、一意性のカラムが必要となる。もし primary key や unique key があったら、そのカラムを使い、なかったら新しく一意性のカラムを作り、そのカラムを元テーブルと新テーブルのマッピングに使う。

デフォルト値の設定については既存のテーブルにカラムを追加する方法と同様である。つまり、アスペクトでフィールドを初期化すればフィールドの初期値がカラムのデフォルト値となる。

この方法では既存のテーブルにカラムを追加する場合とは違い、アスペクトを取り外してもデータを残しておける。また、完全に消去したいとき

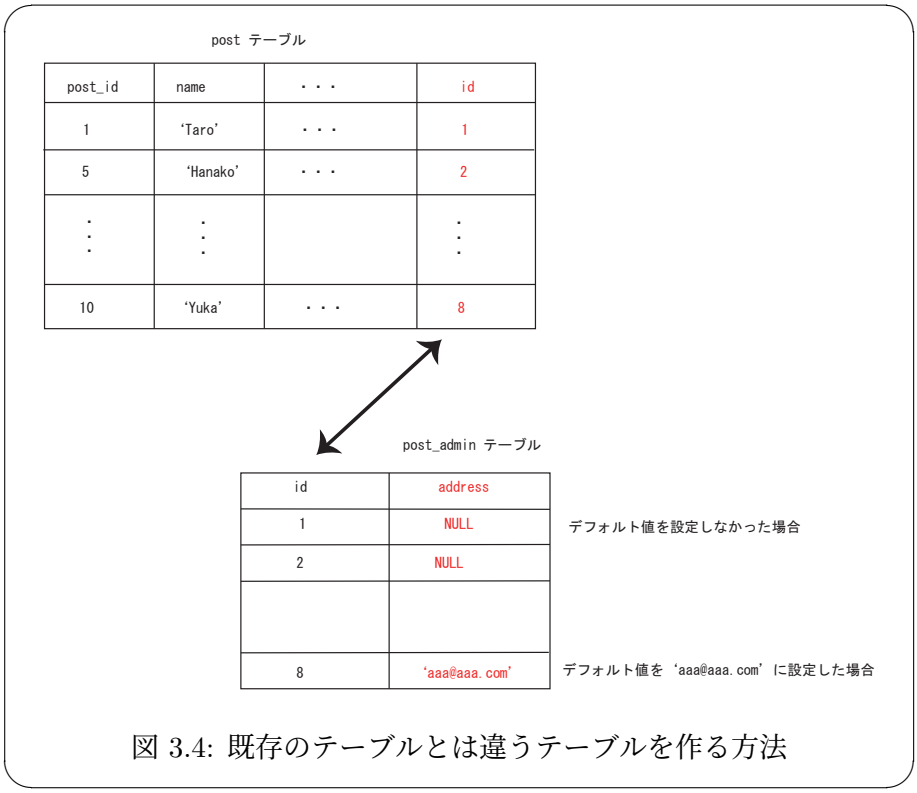

図 3.4: 既存のテーブルとは違うテーブルを作る方法

は消去することも可能である。これによって、アスペクトをつけた場合と取り外した場合を繰り返し利用したい時にデータを再入力しなくて済むので便利である。

テーブルにカラムを追加する場合、アスペクトで元々あるテーブルに追加する形で記述する。しかし、データベースのほうでは既存のテーブルとは違うテーブルにこれらのカラムを格納する。よって実行する前に SQL を変更しなければならない。それに対して第 3.3.1 章の既存のテーブルにカラムを追加する方法では SQL を変更する必要がない。そのため、既存のテーブルとは違うテーブルを作成する方法では、既存のテーブルにカラムを追加する方法に比べて実行速度が遅くなると思われる。

以下に既存のテーブルとは違うテーブルを作る場合のメリットとデメリットをまとめる。

- メリット
  - アスペクトを取り外しても追加したレコードの情報を残しておける

    不必要になったデータベースのカラムは完全に削除することも可能

- デメリット
  - 実行速度が既存のテーブルにカラムを追加する方法に比べて遅いと思われる

3.4　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。

3.4.1　設定ファイル

設定ファイルは 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。(aspectualsotre. は省略)

- connection.driver

  使用するドライバー名を指定する。PostgreSql なら 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

- connection.password

  データベースにアクセスするためのパスワード

- 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

- glue

  メインの glue クラスを指定する

3.4.2　@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、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する。つまり、永続化クラスとして使用したい場合は@PersistClass アノテーションをつける。もしこのアノテーションがついていないと、永続化クラスではないと判断する。つまり、図 3.5 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。

3.4.3　型

データベースでの型の指定方法は三通りある。一つ目はアノテーションを何も書かない方法。二つ目は @Type アノテーションを書く方法。三つ目は @SqlType アノテーションを書く方法である。以下ではそれぞれについて記述例とともに仕様について述べる。

アノテーションを書かない方法

追加するカラム、つまりフィールドには @Type や @SqlType アノテーションは記述しない方法である。

以下のように記述する。例えばこの方法では、フィールドを String で宣言したらデータベースでは varchar(200) 型となる。String 以外の本システムとデータベースとの型の対応については表 3.1 に記述してある。表 3.1 にある型以外の型を指定した場合はエラーを出して終了する。

```
@Glue public class PostGlue {
  @Refine @PersistClass
  public static class Post extends models.Post{
    private String address;
    ...
  }
}
```

表 3.1: 本システムとデータベースとの型の対応 (アノテーションなしの場合)

本システムでの型	データベースでの型
int	int
java.lang.String	varchar(200)
java.sql.Timestamp	timestamp

@Type アノテーションを書く方法

この方法は上記の型設定よりもう少し上級者向けである。上記方法ではフィールドの型を String としたら、必ず varchar(200) になってしまう。しかしデータベースについて少しでも知っている人なら、char(5) や text, varchar(100) などもう少し柔軟に設定したいと感じるときもあるだろう。そのような場合のためにこの方法がある。例えば図 3.6 のように記述する。これはデータベースでは char(200) 型になる。

@Type アノテーションでは length (int) と fixed (boolean) の二つの引数がある。length は varchar(200) なら 200、char(5) なら 5 に値するものである。varchar(200) は 200 文字までの可変長文字列を表し、char(5) は 5 文字の固定長文字列を表す。つまり、length は指定した文字列の長さを表す。fixed は可変長か固定長かを表すための引数である。ture なら固定長、false なら可変長となる。

これらの @Type アノテーションの引数は初期値が設定されている。length は 0、fixed は false である。これらの引数を指定しなかった場合は初期値が代入されて型が決定される。

表 3.2 に本システムでの型・@Type アノテーションの引数とデータベースでの型との対応を記述する。

表 3.2: 本システムとデータベースとの型の対応 (@Type アノテーションをつける場合)

本システムでの型	length	fixed	データベースでの型
int	n	true, false	int
java.lang.String	n(n>0)	true	char(n)
java.lang.String	n(n>0)	false	varchar(n)
java.lang.String	0	true, false	text
java.sql.Timestamp	n	true, false,	timestamp

@SqlType アノテーションを書く方法

@SqlType アノテーションは @Type アノテーションよりもカスタマイズしたい人用である。@SqlType アノテーションは value(String) を引数に取る。value にデータベースで設定したい型、例えば varchar(200) を記述する。つまり、図 3.7 のように記述する。

@SqlType アノテーションの引数 value に入れる型とフィールド宣言のときの型 (図 3.7 の例だと String) との対応は、AspectualStore の実装に依存する。

3.4.4　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

$$privateStringaddress = \text{”デフォルトにしたい値”};$$

と記述すればよい。

フィールド宣言を

$$privateStringaddress;$$

とした場合は初期値は入らない。よって NULL が入る。

$$privateStringaddress = \text{””};$$

とした場合は初期値に空文字が入る。

3.4.5　織り込み

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と同じである。GluonJ 等の一般のアスペクトと違うところは、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。

```
@Glue public class PostGlue {

  @Refine @PersistClass
  public static class PostGlue extends models.Post{
    private String address;
    public String getAddress(){
      return address;
    }
    public void setAddress(String address){
      this.address = address;
    }
  }

  @Refine
  public static class IndexActionGlue
      extends main.IndexAction{
    private String memo;
    public String getMemo(){
      return memo;
    }
    public void setMemo(){
      this.memo = memo;
    }
  }
}
```

図 3.5: 永続化クラスと永続化クラスではない一般のクラス

```
@Glue public class PostGlue {
  @Refine @PersistClass
  public static class Post extends models.Post{
    @Type(length = 200, fixed = true)
    private String address;
    ...
  }
}
```

図 3.6: @Type アノテーションを書いた場合の型の記述例

```
@Glue public class PostGlue {
  @Refine @PersistClass
  public static class Post extends models.Post{
    @SqlType(value = "varchar(200)")
    private String address;
    ...
  }
}
```

図 3.7: @SqlType アノテーションを書いた場合の型の記述例

第4章　実装

4.1　アスペクト指向言語 GluonJ の拡張

既存のアスペクト指向言語については第 2 章で述べた。既存のアスペクト指向言語である GluonJ を拡張することによって本システムの実装を行った。以下では GluonJ の拡張部分について述べる。

4.1.1　全体の流れ

全体の流れは図 4.1 のようになる。

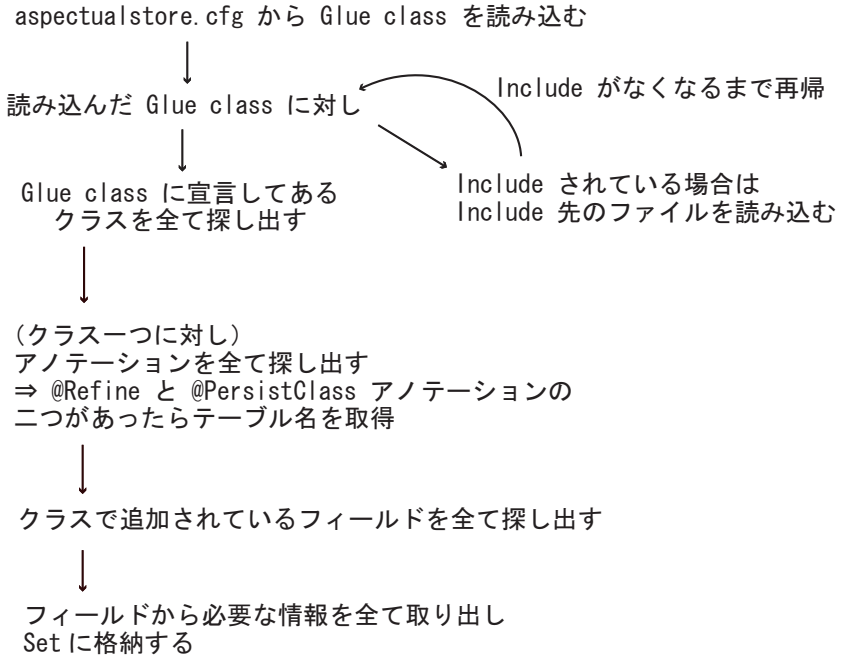

図 4.1: 全体の流れ

aspectualstore.cfg から最初の glue class を読み込んだらその glue class で宣言されているクラスを探し出す。また、Include されている glue class

があったらそのクラスに対しても上記と同じことを行う。Glue class で宣言されているクラスを探し出したら、そのクラスについているアノテーションから永続クラスかどうかのチェックを行う。第 4.1.2 章では永続クラスの取得までのプログラムについて述べる。

第 4.1.3 章では永続化クラスで追加されているフィールドからデータベースにカラムを追加するために必要な情報の取得について述べる。カラムを追加するために必要な情報がそろってから実際にデータベースに反映するまでの過程を第 4.1.5 章に記述する。

4.1.2　永続化クラスの取得

まず、aspectualstore.cfg から Glue class がある場所を読み込む。これは AspectualStore 内のメソッドを利用した。このクラスを元に、アスペクトで付け足そうとしているカラムを探し出す。アスペクトで付け足そうとしているカラムを探し出すためのアルゴリズムは以下のようになっている。基本的に解析は千葉滋氏が開発した Javassist[3, 10] を利用している。

元となる Glue class を探したら、Include されている glue class を再帰的に探す。それと同時に glue class に宣言してあるクラスを全て探し出す。プログラムは図 4.2 のようになっている。getDeclaredFieldsInfoFromCtClass メソッドで glue class に宣言してあるクラスが永続化クラスかどうかを判別している。

クラス一つに対して、アノテーションを全て探し出す。もし、クラスに @Refine アノテーションと @PersistClass アノテーションの二つがあったら、このクラスは永続化クラスである。プログラムは図 4.3 のようになっている。

分析しているクラスが永続クラスであることがわかったら、追加されているフィールドについて調べる。これについては第 4.1.3 章で述べる。

4.1.3　フィールドの情報を取得

フィールドの情報を取得するプログラムは図 4.5 のようになっている。ここで取得する情報は以下の値である。

- 追加するテーブル名
- 追加するカラム名
- 追加するカラムの型
- 追加するカラムに初期値がある場合は初期値の情報

```
public static Set getFields(
    String glueFileName, Set<...> addedColumns)
    throws SqlTypeException{
  ...
  try{
    ...
    cc = cp.get(glueFileName);
    CtClass[] ccs = null;
    ccs = cc.getNestedClasses();

    //Glue class(file) 内で宣言されているクラスをチェック
    for(int i = 0; i < ccs.length; i++){
      CtClass c = ccs[i];
      getDeclaredFieldsInfoFromCtClass(c, addedColumns);
    }
    //Include された glue class の処理
    CtField[] includedGlues = cc.getDeclaredFields();
    for(int i = 0; i < includedGlues.length; i++){
      boolean isIncludedGlues = false;
      Object[] includedGluesAnnotations =
        includedGlues[i].getAnnotations();
      for(int j = 0; j < includedGluesAnnotations.length; j++){
        Object annotationObj = includedGluesAnnotations[j];
        if( annotationObj instanceof Include ){
          isIncludedGlues = true;
          break;
      }}
      if( !isIncludedGlues )
        continue;
      CtClass glueClass = includedGlues[i].getType();
      getFields(glueClass.getName(), addedColumns);
  }}catch(...){...}
  return addedColumns;
}
```

図 4.2: Glue class に宣言してあるクラスについて調べる

図 4.3 で追加するテーブル名は取得してあるので、残りの三つの情報を得られれば良い。まず、カラム名を取得する。これは追加したフィールド名と同じであるのでフィールド名から取得する。プログラムは図 4.5 にのせる。

図 4.5 の getSqlTypeFromCtField メソッドではデータベースに保存する時の型を取得する。この型は追加したフィールドの型とアノテーションによって決定される。第 3.4.3 章で述べたとおり、アノテーションがついて

```
private static void getDeclaredFieldsInfoFromCtClass(
    CtClass ctClass, Set<...> addedColumns)
    throws SqlTypeException{
  try{
    int annotationFlag = 0;
    Object[] ctClassAnnotationObjs =
      ctClass.getAvailableAnnotations();
    for(int j = 0; j < ctClassAnnotationObjs.length; j++){
      Object obj = ctClassAnnotationObjs[j];
      if(obj instanceof Refine) {
        annotationFlag++;
        continue;
      }else if(obj instanceof PersistClass){
        annotationFlag++;
        continue;
      }
      if(annotationFlag == 2)
        break;
    }
    if(annotationFlag != 2)
      return;
    String superClassName = ctClass.getSuperclass().getName();
    int lastDot = superClassName.lastIndexOf('.');
    //テーブル名の取得
    String tableName = superClassName.substring(lastDot + 1);
    CtField[] cfs = ctClass.getDeclaredFields();
    for(int k = 0; k < cfs.length; k++){
      //フィールド情報の取得
  }}catch(...){...}
}
```

図 4.3: Glue class 内にあるクラスが永続化クラスであると判別するまでのプログラム

いない場合もある。第 3.4.3 章でも述べたが、データベースでの型はアスペクトで追加されたフィールドの型とアノテーションにより決定される。

4.1.4　初期値の指定

取得しなければならないフィールドの情報で、残っている情報は初期値の情報である。

例えば address というフィールドを追加したいとし、アスペクトで以下のように付け加えたとする。

```
CtField cf = cfs[k];
//カラム名の取得
String fieldName = cf.getName();

GetType gt = new GetType(cf);
//型を getSqlTypeFromCtField() で取得
String fieldType = gt.getSqlTypeFromCtField();

AdministrationTableForAddedColumns admin =
  new AdministrationTableForAddedColumns(
    tableName, fieldName, fieldType, null, null);
setFieldDefault(ctClass.getName(), fieldName, admin);
addedColumns.add(admin);
```

図 4.4: フィールド情報の取得

```
private String address = "aaa@aaa.com";
public void setAddress(String address){...};
public String getAddress(){return address};
```

この場合、追加されたフィールド address に対するゲッターメソッドがないと初期値を取り出せないようになっている。また、ゲッターメソッドは “get + 追加されたフィールド名の最初の文字を大文字にしたもの (Address) ” という形のメソッドに限っている。つまりこの場合、getAddress というゲッターメソッドがなかったら初期値は取りだせない。

このプログラムは図 4.5 に記述してある。

4.1.5　データベースの変更

Glue class からアスペクトで追加されているカラムを探し出すところまでは上記に説明した。ここで、アスペクトで追加されているカラムを全てデータベースに付け足してしまってはエラーが発生する。カラム名をかぶって登録してしまってはデータベースの方でエラーをはくからである。そこで、実際にデータベースにあるカラムは付け足してはいけない。

データベースを毎回チェックし、付け足そうとしているカラムがあったら付け足さず、カラムがなかったら付け足すというプログラムを組んでしまっては、データベースへの接続が本来接続すればいい回数より大幅に増えてしまい、実行速度が遅くなる。また、データベースへの接続が増えることにより、セキュリティ面でも低下するだろう。

```
private static void setFieldDefault(
    String classNameWithoutExtension, String fieldName,
    AdministrationTableForAddedColumns adminTable) {
  try{
    Class beanClass =
      Class.forName(classNameWithoutExtension);
    char fieldName0 = fieldName.charAt(0);
    if(96<fieldName0 && fieldName0<123)
      fieldName0 = (char)(fieldName0-32);
    String methodName =
      "get" + fieldName0 + fieldName.substring(1);
    Method method =
      beanClass.getMethod(methodName, (Class[])null);
    Object beanInstance = beanClass.newInstance();
    Object defaultValue = method.invoke(beanInstance);
    if(defaultValue != null)
      adminTable.setFieldValue(defaultValue + "");
  }catch(...){...}
}
```

図 4.5: フィールド情報の取得

よって、データベースにカラムを追加した場合はその情報をどこかに保持しておき、データベースに追加していないカラムだけを足すようにすればよい。データベースにカラムを追加した情報を保持しておくため、データベースに administration_table_for_added_columns テーブルを用意した。このテーブルは以下のカラムがあり、情報が収納されている。

- table_name
- field_name
- field_value
- field_type
- extra

extra カラムは今のところ使用していない。つまりすべて NULL である。今後ちょっとした情報を保持しておくためにカラムを付け加えたりカラムの変更が起こるのを防ぐために作っておいたものである。

table_name はテーブル名を、field_name は追加するカラム名を、field_value はカラムの初期値を、field_type はカラムの型を保存しておく。カラムの初期値がない場合は field_value は NULL となる。

これに対応したクラスを Java の方でも準備しておく。これが AdministrationTableForAddedColumns クラスである。

カラムの追加情報を保持しておくためのデータベースに含まれるがアスペクトで追加されたカラムには含まれていない情報はデータベースから削除する必要がある。つまり、図 4.6 の A の部分は削除する必要がある。削除とはデータベースのテーブルからカラムを削除することである。

アスペクトで追加されたカラムには含まれるが、カラムの追加情報を保持しておくためのデータベースには含まれない情報はまだカラムが付け足されていない部分なのでカラムを付け足す必要がある。つまり図 4.6 の B の部分はカラムを付け足す必要がある。

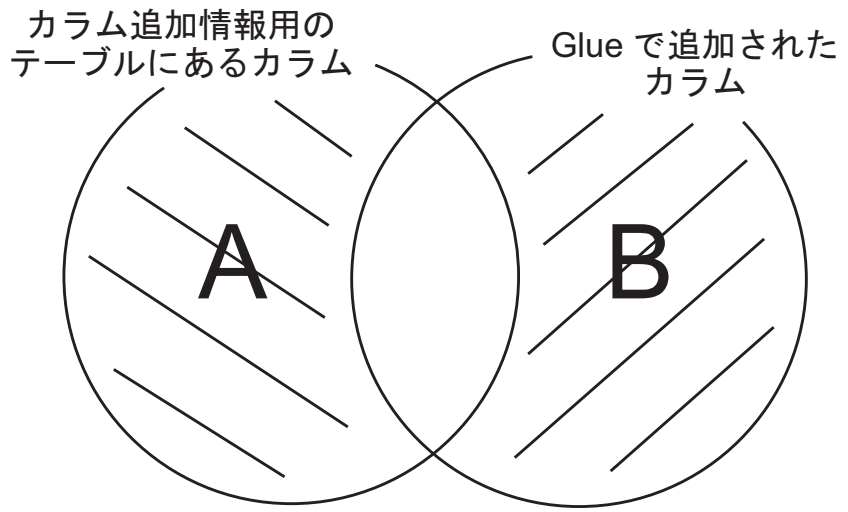

図 4.6: データベースにカラム追加するカラムとデータベースから削除するカラム

上記に述べた、A の部分と B の部分を探し出す方法として、Java の API に含まれている Set クラスの removeAll メソッドを利用した。プログラムは図 4.7 に参照してもらいたい。

図 4.6 の A と B の部分が取り出せたら、カラムの追加と削除を行う。カラムの追加と削除の順番については、まず削除を行う。その後に追加作業を行う。

例えば varchar(200) 型の address カラムを付け足そうとしていたが varhcar(100) 型の address カラムに変更したい場合、A には varchae(200) 型の address カラムが、B には varchar(100) 型の address カラムが入る。追加作業を先に行ってから削除作業を行うと、既に address カラムがデータベースには入っているのでエラーが起こる。このような場合を避け、最

初に削除作業を行い後で追加作業を行う。

```
Set<...> addedColumns =
  ConfirmationOfGlueClass.getFields(glue, addedColumns);
/*
 * Admin Table から情報取得
 * => adminInfo(Set) に入れる
 */
ResultSet rs = exe.selectAllFromTable(Constant.ADMIN_TABLE);
Set adminInfo = putRsIntoAdminInfo(rs);

/*
 * adminInfo => Admin Table から削除するべきレコード
 */
Set<...> deleteColumns =
  new ArraySet<...>(adminInfo);

if(addedColumns != null)
  deleteColumns.removeAll(addedColumns);

/*
 * addedColumns => 実際に DB に追加するべき情報
 */
addedColumns.removeAll(adminInfo);

/*
 * deleteColumns を使って
 * admin table からレコードを削除　&
 * カラムの削除
 */
if(deleteColumns.size() > 0){
  exe.deleteColumns(deleteColumns);
  exe.deleteFromAdminTable(deleteColumns);
}

/*
 * addedColumns を使って
 * admin table にレコードの挿入　&
 * カラムの追加
 */
System.out.println("DELETE_COLUMNS;" + deleteColumns);
System.out.println("ADDED_COLUMNS : " + addedColumns);
if(addedColumns.size() > 0){
  exe.addColumns(addedColumns);
  exe.insertIntoAdminTable(addedColumns);
}
```

図 4.7: フィールド情報の取得

第5章　応用例

本システムを実際のアプリケーションに適応し、機能の抜き差しの有効性を検証した。今回は WEB アプリケーションである掲示板を作成してみた。本システムを作成するにあたり、WEB に関する部分には Servlet, JSP, Struts 、これらのコンテナとして tomcat を利用した。また、永続化部分には 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述べる。掲示板に追加機能としてアドレス欄を作成する。追加機能をつけて実行する場合は、書き込みを行う際にアドレスを書く欄が設けられる。また、追加機能をつけた場合は一覧にもアドレス欄が表示される。追加機能をつけなかった場合は一覧・記入欄ともにアドレス欄は表示されない。

全体の流れは図5.1のようになる。まず、index01 ページではデータベースから今までの記事を取り出し、それが表示される。また新しい記事の書き込みも行える。index02 ページは、index01 ページで記事を書き込んだ場合に飛んでくるページである。このページは index01 で書き込んだ記事の確認である。index03 ページは、index02 ページの確認画面で”記入する” ボタンを押した場合に飛んでくるページである。このページにきたらデータベースにアクセスして記事を挿入するというしくみになっている。

以下では本システムを利用して実装した場合と、既存の方法で実装した場合の比較を行う。

5.2　既存の実装方法

既存の実装方法を用いてプログラムを組んだ場合について述べる。既存の実装方法には二つの方法があるといえる。既存のアスペクト指向言語を用いて実装する場合とアスペクト指向言語を用いずに実装する場合である。

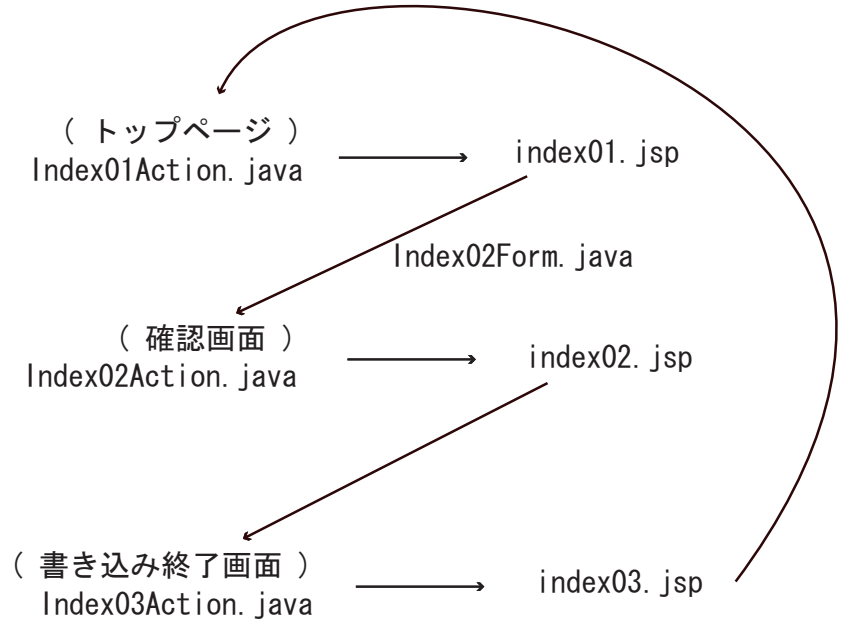

図 5.1: 全体の流れ

5.2.1　既存のアスペクト指向言語を用いる場合

まず、既存のアスペクトを用いる場合について述べる。追加機能を実装するにあたり、変更する点を以下に並べる。

- index01.jsp

  記事の書き込みの欄に address 欄を増やす。一覧に address も表示させる。

- index02.jsp

  確認画面で、address も確認する。

- Index02Form.java

  これは index01.jsp で記入した値を Index02Action.java に送るときに使用する。これは　JavaBean の形で記述されている。ここにも address を付け足す。

- Index01Action.java

  記事の一覧を出力するためにデータベースからとってくる際に、address も他のカラムと同様に取ってきて Map に入れる。

jsp の方で出力するかどうかは Flag クラスを用意し管理する。Flag クラスは JavaBean の形をしている。address というフィールドを用意し、これを出力する場合は true に、出力しない場合は false にする。(基本は false にしておく。機能を付け足したい場合のみアスペクトで ture に変える。) よって、この address フィールドを true にする必要がある。

- Index02Action.java

  Index02Form から値を受け取り、Map に変換して Index03Action に渡す。このときに address の値も Map に入れる必要がある。

- Index03Action.java

  ここではデータベースに値を代入する。値を代入するためには AspectualStore で作成した Post クラスを使う。よってこのクラスの address フィールドに Map から値を読み込んで入れる必要がある。

Java の部分はアスペクトを使って実装すれば良い。jsp はアスペクトを使うことが出来ないので、問題は jsp の部分である。追加機能を取り外しても変更があまりないようにするために、jsp の追加機能 (今回は address) に関わるところは if で場合わけをしておく。つまり、Flag の address が true だったら表示し、false だったら非表示としておく。

この glue クラスをウィーブすれば、プログラムの方の変更は終わりである。しかし、データベースも変更しなければならない。データベースのほうは、データベースに入り以下のコマンドを実行して post テーブルに address カラムを追加する必要がある。

```
ALTER TABLE post ADD COLUMN address varchar(200);
```

この作業は追加機能を付け足すときだけではなく、削除する時にも同様に以下のコマンドを実行しなければならない。

```
ALTER TABLE post DROP COLUMN address;
```

今回は追加・削除するカラムが一つだけだが、数が多くなってきた時は手間がかかる。

5.2.2　アスペクト指向言語を用いずに実装する場合

次に、アスペクトを用いない場合について述べる。変更するべき箇所は上の既存のアスペクトを用いて実装する場合とほぼ同じである。しかし追加機能を削除する時は、追加機能を付け足すために書いたコードを全て削除又はコメントアウトしなければならない。上の箇条書きを見てわかるように、変更箇所が大変多くなり大変手間がかかる。

5.3　本システムを利用した場合

最後に、本システムを利用した場合について述べる。本システムを用いた場合は、基本的に変更する箇所は既存のアスペクトを用いた場合と同じである。しかし、本システムを利用することによってウィーブするときに自動でデータベースを変更してくれるので、手動でデータベースを変更する必要がない。本アプリケーションは build.xml を起動すると実行されるが、最初にこのプロパティをユーザーのシステムに対応するように変更しておく必要があるくらいである。一回変更しておけば、追加機能の抜き差しはアスペクトの取り外しによって簡単に行える。

第6章　まとめ

本研究では、永続システムに対応したアスペクト指向システムの提案と実装を行った。本システムにより、ソースコードを変更せずに機能の抜き差しが出来るようになった。また、データベーススキーマと永続クラスの一貫性を自動で保てるようになった。ソースコードを編集しなくて良くなったことにより、コードの再利用性が増した。セキュリティも強化された。スキーマの一貫性を自動で保てるようになったことにより、今まで手動で変えていた作業がなくなった。自動化したことにより手作業がなくなり、セキュリティも強化された。

本システムはアスペクト指向言語の GluonJ を改造して作成した。GluonJ ではソースコードを編集せずに機能の追加や削除を行うことが出来るが、データベーススキーマの一貫性は保たれないというデメリットがある。そこで、本システムではウィーブする前にアスペクトのクラスファイルを読み、データベーススキーマと違う部分があればデータベーススキーマを変更するようにした。クラスファイルを読み込みデータベーススキーマと違う部分を探し出す部分は、Javassist を利用した。Javassist を利用することによって、ソースコードがなくてもクラスファイルさえあればデータベーススキーマを変更できるようになった。

応用例では実際に本システムが利用可能であることを確認した。

今後の課題

今後の課題としては以下の事柄があげられる。

- **テーブルを追加する場合の実装**

  データベースを自動で変更するときの方法として、既存のテーブルにカラムを追加する方法と既存のテーブルとは別テーブルを作りそこにカラムを追加していく方法の二つの方法を提案した。二つの方法のうちの一つである既存のテーブルとは別テーブルを作りそこにカラムを追加していく方法の実装を行う。

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] Chiba, S. and Nishizawa, M.: An Easy-toUse Toolkit for Efficient Java Bytecode Translators, *In Proceedings of 2nd International Conference on Generative Programming and Component Engineering (GPCE '03)*, pp. 364–376 (2003).

[4] 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).

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

[6] 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).

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

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

[9] XDoclet: XDoclet: Attribute-Oriented Programming, http://xdoclet.sourceforge.net/xdoclet/.

[10] 千葉滋: Javassist Home Page, http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html.

[11] 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).