平成17年度修士論文

アスペクト指向プログラミング と Dependency Injection の融合

東京工業大学 情報理工学研究科
数理・計算科学専攻

学籍番号 04M37024

石川　零

指導教官

千葉 滋 助教授

平成18年1月20日

概要

プログラムは一度完成した後にもバグの修正や機能の拡張などのために変更が必要になることが多い。そのためプログラムを後から簡単に変更できるように設計しておくことは重要である。コンポーネントを用いた開発は、プログラムを構成する部品を組み替えるだけで新しいプログラムを作ることを可能にする。コンポーネントとは関連する機能をまとめた部品のことである。コンポーネントを用いた開発では別々に開発された細かい部品を組み立てることでプログラムを作る。

コンポーネントを組み替えて新たなプログラムを作るには、一度結びつけたコンポーネントの結合を変更する必要がある。しかし既存のプログラミング技術を用いて設計したプログラムでは、コンポーネントの結合を変更することは簡単ではない。例えばオブジェクト指向プログラミングを用いて設計されたプログラムでは、コンポーネントの結合を変更するにはそのコンポーネントと結合したコンポーネントの書き換えが必要となる。書き換えが必要となる原因は、コンポーネントの内部にコンポーネントを結合するためのコードを記述することである。デザインパターンや Dependency Injection といった技術はコンポーネントを結合するためのコードの一部をコンポーネントから分離して記述することを可能にするが、結合するためのコードの全てを分離することはできない。

本論文ではプログラムを後から簡単に変更できるように設計する手法として、glue コードを用いた設計を提案する。glue コードを用いた設計では、アスペクト指向プログラミングを利用してコンポーネントを結合するための全てのコードをコンポーネントの内部から分離することが可能である。コンポーネントを結合するためのコードは glue コードにまとめて記述する。glue コードとはアスペクト指向プログラミングにおけるアスペクトの一種だが、コンポーネントの結合の記述に特化しているという特徴を持つ。

また glue コードを用いた設計を実現するアスペクト指向システムとして GluonJ を開発した。GluonJ は glue コードを用いた設計に特化しており、既存のアスペクト指向システムに比べてコンポーネントの結合をより直感的に記述することを可能にする。GluonJ の glue コードは従来のアスペクトの機能に Dependency Injection の機能を取り込み、再設計し

たものである。

謝辞

東京工業大学 数理・計算科学専攻の千葉滋助教授は、研究の様々な面について非常に丁寧かつ親身に私のことを指導してくださりました。本研究においては研究の進め方や方針について数多くの助言をいただきました。また論文や発表用の資料を作る際には、要旨のまとめ方や分かりやすい説明の仕方、細かな文法についてまで非常に細かくご指導していただきました。さらに国内外の会議へ参加する機会を積極的に与えていただきました。会議への参加は自分の研究を見直し、そして自らの見聞を広める良い機会となりました。また研究生活を過ごすにあたり非常に快適な環境を用意していただきました。さらに研究についてだけではなく、社会人の先輩として貴重な意見を幾度となくいただきました。ここに深く感謝いたします。

三菱総合研究所の佐藤芳樹氏、千葉研究室の西沢無我氏は、同一の分野を研究する後輩である私に、研究に必要な基礎的な知識や関連研究、論文や発表資料の作り方など、様々な点で助言をくださりました。また私の研究テーマについて深い議論を交わしてくださりました。深く感謝いたします。また東京工業大学 数理・計算科学専攻の光来健一助手は、本研究の内容や発表について多くの助言をくださりました。

そして昼夜を問わず共に学んだ千葉研究室の方々のお陰で、私は充実した研究生活を過ごすことができました。ここに感謝いたします。

目 次

図 目 次

表 目 次

第1章　はじめに

プログラムは一度完成した後に機能の追加や変更が必要になることが多い。そのためプログラムを後から簡単に変更できるように設計することは重要である。コンポーネントを用いた開発はプログラムを構成する部品を組み替えるだけで新しいプログラムを作ることを可能にする。コンポーネントとは関連する機能を1つにまとめた部品のことで、コンポーネントを用いた開発とはコンポーネントを組み合わせてプログラムを作るという開発手法である。コンポーネントを用いた開発の利点は、プログラムの機能を後から変更することが簡単になることである。コンポーネントを用いて作られたプログラムはコンポーネントごとに機能がまとまっているため、機能を変更するには、コンポーネントを別なものに交換すればよい。

コンポーネントを用いた開発では、コンポーネントの内部を変えずにコンポーネントの結合を変更できるようプログラムを設計するべきである。コンポーネントの結合の変更は、プログラムを後から変更するときに必要となる。もしコンポーネントの結合を変えるたびにコンポーネントの内部を変更しなければならないとすると、そのコンポーネントの再利用性は低い。しかし既存のプログラミング技術を用いて開発したプログラムは、コンポーネントの内部を変更せずにコンポーネントの結合を変えることが難しかった。例えばオブジェクト指向プログラミングを用いて設計されたプログラムでは、コンポーネントの結合を変更するにはそのコンポーネントと結合したコンポーネント内部の書き換えが必要となる。書き換えが必要となる原因は、コンポーネントの内部に結合のためのコードを記述することである。デザインパターンや Dependency Injection といった技術はコンポーネントを結合するためのコードの一部をコンポーネントから分離して記述することを可能にするが、結合するためのコードの全てを分離することはできない。

我々はコンポーネントの内部を変更せずにコンポーネントの組み合わせを変えることができるプログラムの設計手法として、glue コードを用いた設計を提案する。glue コードを用いた設計はアスペクト指向プログラミングに基づいているが、アスペクトをコンポーネントの結合の記述のみに用いるという特徴がある。このコンポーネントの結合を記述するアスペクトが glue コードである。また我々は glue コードに適したアスペクトを

提供するアスペクト指向システムとして GluonJ を開発した。GluonJ が提供する glue コードは従来のアスペクトの機能に Dependency Injection の機能を取り込んだものである。GluonJ は従来のアスペクト指向システムに比べて、より直感的にコンポーネントの結合を記述することを可能にする。

以下 2 章ではコンポーネントの内部を変えずにコンポーネントの結びつけを変更できることの重要性と既存技術の限界を述べ、3 章では glue コードを用いた設計と GluonJ について示す。4 章では GluonJ の機能と実装について述べ、最後に 5 章で本稿をまとめる。

第2章　研究の目標と既存技術の限界

この章ではまず、プログラムを構成するコンポーネントの内部を書き換えずにコンポーネントの組み合わせを変更できるようプログラムを設計することが重要であることを示す。次に既存技術ではこの設計をうまく実現することが困難であることを示す。

2.1　コンポーネントを用いた開発

コンポーネントを用いた開発とはプログラムの開発手法の１つで、別々に開発された細かい部品を組み立てることでプログラムを開発するという手法である。コンポーネントとは関連する機能を１つにまとめた部品のことである。

コンポーネントを用いた開発は、プログラムを構成する部品を組み替えるだけで新しいプログラムを作ることを可能にする。一般にプログラムは完成後にバグの修正や機能の拡張のために変更が必要になることが多い。コンポーネントを用いて作られたプログラムはコンポーネントごとに機能がまとまっているため、機能を変更するには、コンポーネントの組み合わせを変えて、その機能を提供するコンポーネントを別なものに交換すればよい。またプログラムの一部を再利用して新しいプログラムに利用することもある。コンポーネントを用いて作られたプログラムの機能は、コンポーネントごとにまとまって記述されているため、プログラムの一部を新しいプログラムの開発時に再利用することも簡単である。

2.1.1　目標：コンポーネントの内部を変えずにその組み合わせを変更する

コンポーネントの結合の変更は、コンポーネントの内部を書き換えずに行えるべきである。コンポーネントを組み替えて新たなプログラムを作るときには、一度結びつけたコンポーネントの結合を変更する必要がある。もしコンポーネントの結合を変更するたびにコンポーネントの内部を書き換えなければならないとすると、そのコンポーネントの再利用性は低い。なぜなら内部の書き換えが必要なことは、コンポーネントを修正せずその

まま再利用できないことを意味するからである。プログラムがそのような再利用性の低いコンポーネントから構成されているとき、プログラムの機能を変更する作業はコンポーネントの内部の修正を必要とするため困難となる。

コンポーネントを用いて開発したプログラムとそれに対する変更の例を、以下の 2 つのコンポーネントから構成されるプログラムを用いて示す。このプログラムはオンライン書店プログラムの一部で、利用者の注文をデータベースへ送信するプログラムである。このプログラムは図 2.1 のように 2 つのコンポーネントから構成される。

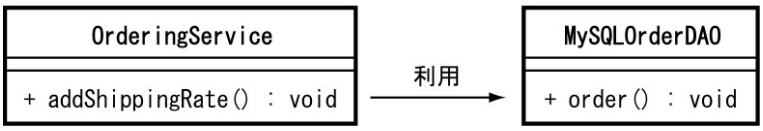

図 2.1: 注文発注プログラムを構成するコンポーネント

このプログラムは OrderingService コンポーネントと MySQLOrderDAO コンポーネントから構成される。OrderingService コンポーネントの addShippingRateAndOrder メソッドは利用者が購入しようとしている本の総額を調べ、総額が一定以下ならば総額に送料を足して発注管理データベースに注文を送信する。注文の送信には MySQLOrderDAO コンポーネントの order メソッドを用いる。

このプログラムに対する変更例として、以下の 2 つの変更を考える。

- 変更 1 : データベースシステムの変更と新たな機能の利用

  この変更は、注文発注プログラムが利用するデータベースシステムを別のデータベースシステムに置き換える場合を想定している。また新たに利用するデータベースシステムがデータベースアクセスの最適化アルゴリズムにヒントを与えられる機能を提供しており、その機能を利用することを想定する。新たに利用するデータベースシステムにアクセスするためのコンポーネントは OracleOrderDAO である。最適化アルゴリズムにヒントを与えるには OracleOrderDAO コンポーネントの sendHint メソッドを呼ぶ。

- 変更 2 : 安全性の強化

  この変更はプログラムの安全性を強化するために、プログラムを実行するユーザにデータベースへのアクセス権があるかを調べる機構を追加する場合を想定している。アクセス権の検査はデータベースをア

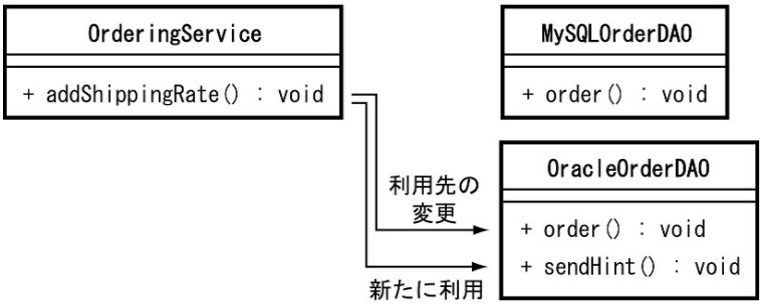

図 2.2: 注文発注プログラムに対する変更の例 1

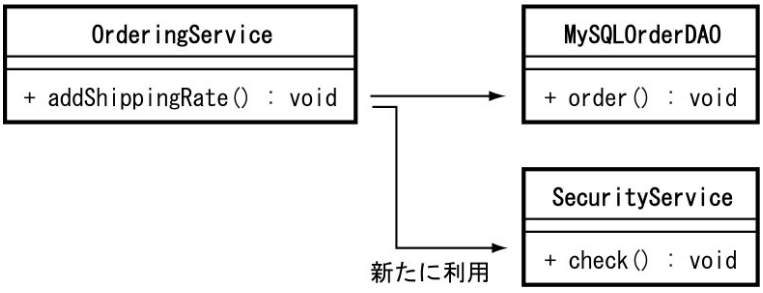

図 2.3: 注文発注プログラムに対する変更の例 2

クセスする処理の直前で、SecurityManager コンポーネントの check メソッドを用いて行う。アクセス権を検査するのに必要なユーザの ID は、OrderingService コンポーネントの id が指しているとする。SecurityManager コンポーネントを生成するには SecurityManager クラスのファクトリメソッドを用いる。

これら 2 つの変更は、図 2.3 のようにコンポーネントの結びつけを変えることで実現できる。変更 1 を実現するには、OrderingService コンポーネントと結びついた MySQLOrderDAO コンポーネントを OracleOrderDAO に置き換えればよい。また変更 2 を実現するには、OrderingService コンポーネントに SecurityManager コンポーネントを新たに結びつければよい。

2.1.2　通常のオブジェクト指向による設計の問題

オブジェクト指向を用いた設計では、コンポーネント同士を結びつけるためのコードをコンポーネントの内部に記述するため、コンポーネントの組み合わせを変更するにはコンポーネントの内部の変更が必要になって

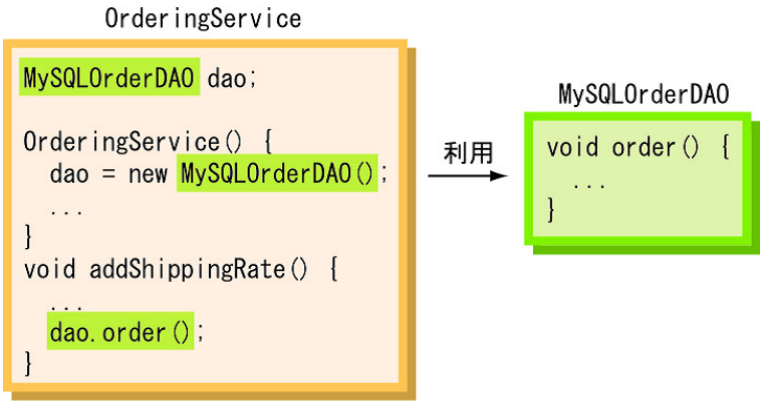

図 2.4: 通常のオブジェクト指向を用いて設計した注文発注プログラム

しまう。このことを具体的に説明するためにオブジェクト指向を用いて設計した注文発注プログラムの例を示す。図 2.4 はこのプログラムにおけるコンポーネントの関係を表している。またこのプログラムを構成する OrderingService クラスと MySQLOrderDAO クラスのソースコードを以下に示す。

```
1   public class OrderingService {
2     private MySQLOrderDAO dao;
3     private String id;
4     public OrderingService() {
5       dao = new MySQLOrderDAO();
6     }
7     public void addShippingRateAndOrder (
8         User user, Book[] books) {
9       int total = 0;
10      for (int i = 0; i < books.length; i++)
11        total += books[i].getPrice();
12      if (total < 1500)
13        total += getShippingRate(user);
14      dao.order(user, books, total);
15    }
16  }
```

```
1   public class MySQLOrderDAO {
2     public void order (User user,
3         Book[] books, int total) { ... }
4   }
```

オブジェクト指向を用いて設計した OrderingService コンポーネントの内部には、OrderingService コンポーネントと MySQLOrderDAO コンポーネン

トを結びつけるコードが存在する。結びつけるコードは 3 箇所にあり、1 つは MySQLOrderDAO 型の dao フィールドの宣言、1 つは MySQLOrderDAO コンストラクタ内の、SubCa コンポーネントのコンストラクタを呼び出すコードである。もう 1 つは addShippingRateAndOrder メソッド内の、dao フィールドが指すオブジェクトの order メソッドを呼び出すコードである。

このオブジェクト指向を用いて設計した OrderingService コンポーネントの再利用性は低い。なぜなら OrderingService コンポーネントと MySQLOrderDAO コンポーネントの結び付けを変えるには、OrderingService コンポーネントに記述された MySQLOrderDAO コンポーネントとを結びつける記述の修正が必要である。例えば 2.1.1 節で述べた「変更 1：データベースシステムの変更と新たな機能の利用」を実現するには、OrderingService コンポーネントの内部を次のように変更しなければならない。

- dao フィールドの型を OracleOrderDAO に変更する
- MySQLOrderDAO のコンストラクタ呼び出しを OracleOrderDAO のコンストラクタ呼び出しに変更
- addShippingRateAndOrder メソッドに含まれる、order メソッドの呼び出しの直前に、 dao フィールドの指す OracleOrderDAO コンポーネントの sendHint メソッドの呼び出しを追加

また 2.1.1 節で述べた「変更 2：セキュリティ機能の追加」を実現するには、OrderingService コンポーネントの内部を次のように変更しなければならない。

- OrderingService コンポーネントに SecurityManager コンポーネントを指すフィールドを追加
- OrderingService コンポーネントのコンストラクタに SecurityManager コンポーネントのコンストラクタ呼び出しを追加
- addShippingRateAndOrder メソッドの冒頭に、SecurityManager コンポーネントのメソッド呼び出しを追加

コンポーネント内部を変更せずにコンポーネントの結びつきを変えることは、プログラムを構成するコンポーネントが多い場合に特に重要となる。もしプログラムを構成するコンポーネントの数が少ないときには、コンポーネントの内部を修正する手間は比較的小さい。しかし、例えば OrderingService コンポーネントの他に MySQLOrderDAO コンポーネントを利用するコンポーネントが複数種類存在するとき、変更 1 を行うには MySQLOrderDAO コンポーネントのコンストラクタ呼び出しの変更と

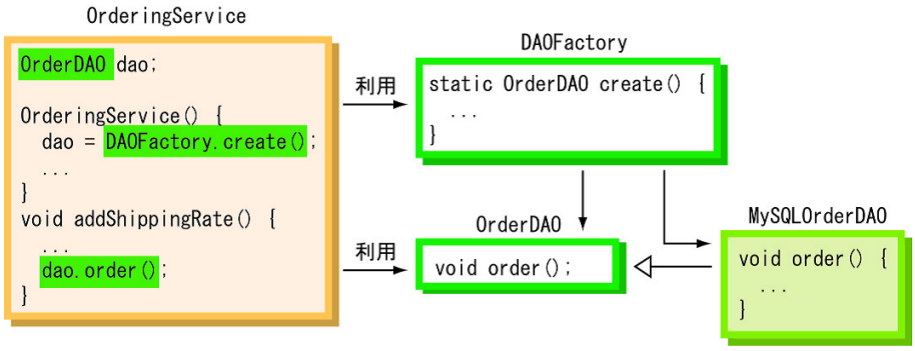

図 2.5: デザインパターンを用いて設計した注文発注プログラム

sendHint メソッド呼び出しの追加を、MySQLOrderDAO コンポーネントを利用する全てのコンポーネントに対して行わなければならない。変更 2 でも同様に、上に挙げた OrderingService コンポーネントに対する変更と同じ変更を、MySQLOrderDAO コンポーネントを利用する全てのコンポーネントに対して行わなければならない。

2.2　既存技術の限界

2.2.1　デザインパターン

デザインパターン [6] を用いても、プログラムに起こる変更をあらかじめ予測し、それに備えてプログラムを設計しない限り、コンポーネントの内部を変えずにコンポーネントの組み合わせを変更することはできない。デザインパターンとは、オブジェクト指向プログラミングにおいて頻繁に起こる典型的な問題を解決するための設計集である。デザインパターンの中には、コンポーネントの内部を変更せずにその組み合わせを変更することを可能にする設計のパターンも存在する。

デザインパターンを用いた設計の問題点を示すために、デザインパターンを用いて設計した注文発注プログラムの例を示す。ここで示す注文発注プログラムは、あらかじめデータベースシステムの変更が起こると予測し、それに備えて設計したプログラムである。データベースシステムの変更に備えるために、ここではデザインパターンの 1 つであるファクトリパターンを用いている。ファクトリパターンは、コンポーネントを生成するコードをファクトリメソッドとしてコンポーネントの外側に分離して記述するという設計手法である。ファクトリパターンを用いて設計した注文発注プログラムを表す図を図 2.5 に、その具体的なコードを以下に示す。

```
1   public class OrderingService {
2     private OrderDAO dao;
3     private String id;
4     public OrderingService() {
5       dao = OrderDAOFactory.factory();
6     }
7     public void addShippingRateAndOrder (
8         User user, Book[] books) {
9       int total = 0;
10      for (int i = 0; i < books.length; i++)
11        total += books[i].getPrice();
12      if (total < 1500)
13        total += getShippingRate(user);
14      dao.order(user, books, total);
15    }
16  }
```

```
1   public class OrderDAOFactory {
2     public static OrderDAO factory() {
3         return new MySQLOrderDAO();
4     }
5   }
```

```
1   public interface OrderDAO {
2     public void order();
3   }
```

ファクトリパターンを用いて設計した注文発注プログラムと 2.1.2 節で示したオブジェクト指向を用いた注文発注プログラムの違いは、OrderingService コンポーネントがインタフェースとファクトリメソッドを用いて間接的に MySQLOrderDAO コンポーネントを参照することである。OrderingService クラスのソースコードに見られる違いは 2 つあり、1 つは OrderingService コンポーネントの dao フィールドの型を、MySQLOrderDAO 型から OrderDAO 型に変更したことである。もう 1 つは MySQLOrderDAO コンポーネントのコンストラクタ呼び出しを OrderingService コンストラクタの中から取り除き、代わりに OrderDAOFactory クラスのファクトリメソッドを呼ぶように変更したことである。

上のデザインパターンを用いたプログラムでは、利用するデータベースシステムを変更してもコンポーネントの内部を変更する必要はない。データベースシステムを変更するには OrderingService コンポーネントと結びつく MySQLOrderDAO コンポーネントを OracleOrderDAO コンポーネントに変更する必要があるが、これを実現するには OrderDAOFactory のファクトリメソッド内の MySQLOrderDAO コンポーネントを OracleOrderDAO コンポーネントに変更するだけで十分である。OrderingService コンポーネントは変更せずに再利用できる。

しかしこのプログラムに対して設計時に想定していない変更を行うには、コンポーネントの内部を変更しなければならない。例えば 2.1.1 節で示した変更 1 を実現するには、前段落で述べたデータベースシステムの変更に加えて、OrderingService コンポーネントに OracleOrderDAO コンポーネントの sendHint メソッド呼び出しを追加しなければならない。またこのプログラムに対して 2.1.1 節で示した変更 2 を実現するには、2.1.2 節で示した通常のオブジェクト指向を用いて設計したプログラムと同様に、コンポーネント内部の変更が必要になる。具体的には OrderingService クラスの内部に OracleOrderDAO コンポーネントの sendHints メソッド呼び出しを追加し、また OrderDAO インタフェースにも sendHints メソッドを用意する必要がある。

コンポーネント内部の変更が必要になる原因は、このプログラムが変更 1 が起こることを想定して設計されていないためである。もし起こりうる全ての変更をあらかじめ想定できるならば、デザインパターンを用いることでどの変更が起こってもコンポーネントを変更せず再利用することが可能なプログラムを設計することができる。しかしプログラムに対して起こる全ての変更を設計時に想定することは困難である。

2.2.2 Dependency Injection

Dependency Injection [5] は、サブコンポーネントを生成してフィールドに代入するコードをコンポーネントの外側に分離して記述することを可能にする言語機構である。サブコンポーネントの生成と代入をおこなうコードは、DI コンテナの設定ファイルに記述する。DI コンテナとは Dependency Injection を提供するシステムであり、代表的な DI コンテナとして Spring Framework [9]、Seasar2 [17]、PicoContainer [4] がある。

Dependency Injection を用いてプログラムを設計することで、コンポーネントの組み合わせの変更のうち一部の変更をコンポーネントの内部の修正なく行うことができるが、全ての変更をコンポーネント内部の修正なく行うことはできない。このことを具体的に示すために DI コンテナを用いて設計した注文発注プログラムを用いる。DI コンテナを用いた注目発注プログラムのコンポーネントの関係は図 2.6 に表す。DI コンテナを用いて設計したとき、MySQLOrderDAO クラスはオブジェクト指向を用いて設計した 2.1.2 節のプログラムと同じものを用いる。Dependency Injection を利用した場合の OrderingService クラスは以下のようになる。

```
1   public class OrderingService {
2     private OrderDAO dao;
```

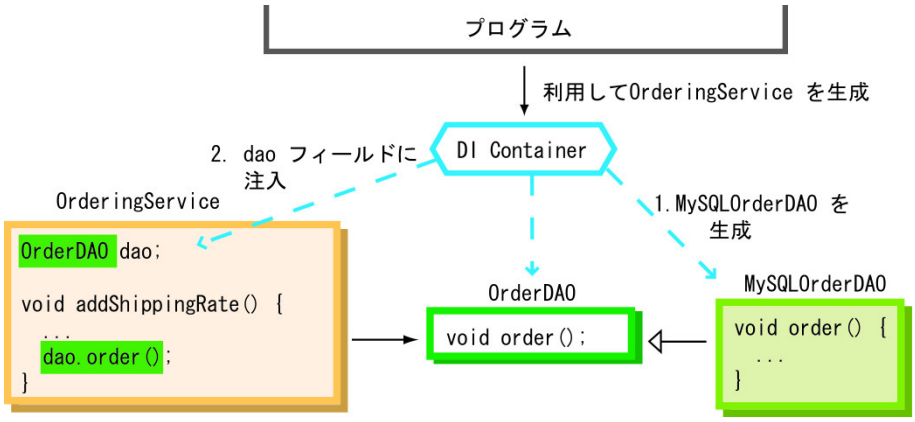

図 2.6: DI コンテナを用いて設計した注文発注プログラム

```
3     private String id;
4     public OrderingService() {}
5
6     public void setOrderDAO(OrderDAO d) {
7       dao = d;
8     }
9     public void addShippingRateAndOrder (
10        User user, Book[] books) {
11      int total = 0;
12      for (int i = 0; i < books.length; i++)
13        total += books[i].getPrice();
14      if (total < 1500)
15        total += getShippingRate(user);
16      dao.order(user, books, total);
17    }
18  }
```

Dependency Injection を用いたプログラムでは、MySQLOrderDAO コンポーネントを生成して dao フィールドへ代入するコードが OrderingService コンポーネントから取り除かれている。MySQLOrderDAO コンポーネントの生成と代入をおこなうコードは DI コンテナの設定ファイルに記述する。代表的な DI コンテナの 1 つである Spring Framework を用いて記述した設定ファイルは以下である。

```
1   <beans>
2     <bean name="service"
3           class="OrderingService">
4       <property name="dao">
5         <ref bean="dao"/>
6       </property>
```

```
 7      </bean>
 8      <bean name="dao"
 9            class="MySQLOrderDAO" />
10    </beans>
```

OrderingService コンポーネントを使うプログラムは DI コンテナが提供するファクトリメソッドを呼ぶことで、この設定ファイルに従って MySQLOrderDAO コンポーネントが dao フィールドに代入された OrderingService コンポーネントを生成することができる。

Dependency Injection を用いることで、コンポーネントの組み合わせの変更のうち一部の変更はコンポーネント内部の変更無しに行うことができる。例えば上の例で、OrderingService コンポーネントが利用するデータベースシステムの変更を考える。このとき OrderingService コンポーネントと結びつく MySQLOrderDAO コンポーネントを OracleOrderDAO コンポーネントに変更するには、設定ファイルを変更すればよい。OrderingService コンポーネントの内部を変更する必要はない。

しかし Dependency Injection を用いても、コンポーネントの結びつきの変更の全てをコンポーネントの内部を変えずに行うことはできない。例えば 2.1.1 節で挙げた変更 1 を実現するには、利用するコンポーネントを MySQLOrderDAO コンポーネントから OracleOrderDAO コンポーネントへ変更するのに加え、OracleOrderDAO コンポーネントのメソッド呼び出しを OrderingService コンポーネントに追加する必要がある。Dependency Injection を用いて設計した注文発注プログラムでこのメソッド呼び出しを追加するには、2.1.2 節で示したオブジェクト指向を用いた通常の実装と同様に OrderingService クラス内部に sendHints メソッドを呼び出すコードを新たに記述しなければならない。

2.2.3　アスペクト指向プログラミング

アスペクト指向プログラミングとは横断的関心事を独立したコンポーネントとしてモジュール化するための技術である。横断的関心事とは、オブジェクト指向など従来のプログラミング技法でモジュール化することの難しい処理群のことである。一般的な横断的関心事の例として、ログ処理や例外処理、プログラムの認証処理などが知られている。例えばオブジェクト指向プログラミングでログ処理をおこなうには、ログを出力するコードをプログラム中に記述する必要があるが、多くの場合この出力処理はプログラム中の様々な箇所に散らばってしまう。このことはログを出力する箇所を変えたりログの出力を止めたりする作業を困難にし、処理の変更のし忘れを招く原因となる。アスペクト指向プログラミングは、このようなプ

ログラム中に散らばってしまう処理を 1 つのモジュールにまとめることを目的としている。

アスペクト指向プログラミングを用いてプログラムを設計することで、コンポーネントの内部を変更せずにコンポーネントの結びつけを変更することはできるが、様々な制限によりコンポーネントの再利用性が低下してしまう。この問題点について、代表的なアスペクト指向言語である AspectJ [10, 1] を用いて以下で説明する。

AspectJ

AspectJ では横断的関心事を 1 つのモジュールにまとめるためにアスペクトというコンポーネントを用いる。アスペクトはクラスを拡張したもので、フィールドやメソッドに加えてポイントカットやアドバイスといった要素から構成される。ポイントカットとは実行中のプログラムのある時点を指定するための記述である。プログラムの実行がポイントカットが指定する時点に差し掛かったら、そのポイントカットと結びついたアドバイスが実行される。アドバイスは横断的関心事を実装するための要素で、ポイントカットと組み合わせて用いる。この他にアスペクトはインタータイプ宣言、デプロイメント指定子といった要素から構成される。これらアスペクトを構成する要素について順に説明する。

- ポイントカット

  ポイントカットとはプログラムの実行中のある時点を指定するための記述である。ポイントカットを用いて指定できる時点のことをジョインポイントと呼ぶ。ポイントカットを用いて指定できるジョインポイントの種類は、アスペクト指向言語によって異なる。AspectJ では、メソッドを実行する時点やフィールドをアクセスする時点を指定することができる。

  AspectJ ではポイントカットをポイントカット指定子の組み合わせで表す。表 2.1 は、AspectJ で用いることができるポイントカット指定子の一部を表している。例えば OrderingService クラスの addShippingRateAndOrder メソッドを実行する時点を指定するポイントカットは、以下のように書ける。

  ```
  execution(void OrderingService.addShippingRateAndOrder(User, Book[]))
  ```

  ポイントカット指定子を組み合わせるには論理演算子を用いる。AspectJ は &&、||、！ という 3 種類の論理演算子を提供している。

- アドバイス

ポイントカットが指定した特定のジョインポイントで実行される処理をアドバイスと呼ぶ。AspectJ のアドバイスには幾つか種類があり、それぞれ処理を実行するタイミングが異なる。代表的なアドバイスとしては before アドバイス、after アドバイス、around アドバイスがある。それぞれのアドバイスは記述されたコードを、ジョインポイントの直前、ジョインポイントの直後、またジョインポイントの代わりに実行する。

- インタータイプ宣言

  AspectJ のアスペクトは、既存クラスにフィールドやメソッドなどを新たに追加する機能を持つ。これをインタータイプ宣言と呼ぶ。

- デプロイメント指定子

  AspectJ のアスペクトは、通常のオブジェクトと異なり、new 式を用いて生成することはできない。アスペクトはプログラム中に自動的に生成される。生成の方法は、あらかじめ用意されたデプロイメント指定子を用いて選ぶことができる。AspectJ の提供するデプロイメント指定子を表 2.2 に示す。

AspectJ を用いた設計の例

注文発注プログラムの例を代表的なアスペクト指向言語である AspectJ を用いて記述し、それを用いて AspectJ のプログラミングモデルについて説明する。アスペクト指向プログラミングを用いた設計では、結びつけるコンポーネントのペアのうち利用されるコンポーネントをアスペクトとして記述する。注文発注プログラムの例では MySQLOrderDAO コンポーネントがアスペクトになる。なお OrderingService コンポーネントは通常のクラスとして設計する。以下は AspectJ を用いて設計したときの OrderingService コンポーネントである。

```
1   public class OrderingService {
2     private User userdata;
3     private Book[] booksdata;
4     private int total;
5     public void addShippingRateAndOrder (
6         User user, Book[] books) {
7       total = 0;
8       for (int i = 0; i < books.length; i++)
9         total += books[i].getPrice();
10      if (total < 1500)
11        total += getShippingRate(user);
```

表 2.1: AspectJ の代表的なポイントカット指定子

指定子の名前	指定するジョインポイントの種類
execution (*behavior-pattern*)	*behavior-pattern* が表すメソッド・コンストラクタを実行する時点
call (*behavior-pattern*)	*behavior-pattern* が表すメソッド・コンストラクタを呼び出す時点
set (*field-pattern*)	*field-pattern* が表すフィールドへ値を代入する時点
get (*field-pattern*)	*field-pattern* が表すフィールドの値を参照する時点
within (*class-pattern*)	*class-pattern* がクラスの内部にあるジョインポイント
cflow (*pointcut*)	*pointcut* が表すジョインポイントに含まれるジョインポイント
this (*type or id*)	実行中のオブジェクトが、*type* が表すクラスもしくは *id* の表す変数の型であるジョインポイント
target (*type or id*)	処理の対象のオブジェクトが、*type* が表すクラスもしくは *id* の表す変数の型であるジョインポイント
args (*types or ids*)	処理の引数のオブジェクトが、*type* が表すクラスもしくは *id* の表す変数の型であるジョインポイント

```
12        userdata = user;
13        booksdata = books;
14      }
15    }
```

2.1.2 節で示した通常のオブジェクト指向プログラミングを用いた設計との違いは、MySQLOrderDAO コンポーネントに関する 3 つのコードが取り除かれていることである。3 つのうち 1 つは MySQLOrderDAO コンポーネントを指すフィールドであり、1 つは MySQLOrderDAO コンポーネントのコンストラクタ呼び出しである。もう 1 つは MySQLOrderDAO コンポーネントの addShippingRateAndOrder の最後にあった order メソッドの呼び出しである。これらのコードに対応する記述は、以下の MySQLOrderDAO アスペクトの中に記述される。

```
1    aspect MySQLOrderDAO {
```

表 2.2: AspectJ で利用可能なデプロイメント指定子

指定子	機能
issingleton	プログラムに対してアスペクトを 1 つインスタンス化する
perclass	各クラスに対してアスペクトを 1 つインスタンス化する
perthis(*pointcut*)	オブジェクトごとにアスペクトを 1 つインスタンス化する。インスタンス化は引数のポイントカットが指定するジョインポイントに達したときに行われる。生成されたインスタンスは、ジョインポイントにおける this 変数が指すオブジェクトと結び付けられる
pertarget(*pointcut*)	オブジェクトごとにアスペクトを 1 つインスタンス化する。インスタンス化は引数のポイントカットが指定するジョインポイントに達したときに行われる。生成されたインスタンスは、ジョインポイントで target 変数が指すオブジェクトと結び付けられる
percflow(*pointcut*)	引数のポイントカットが指定する実行フローに入ったときにアスペクトをインスタンス化する。インスタンス化は指定する実行フローに入るたびに生成される。生成されたインスタンスは実行フローから出るまで有効である

```
2     after(OrderingService s) :
3         execution(void OrderingService.
4         addShippingRateAndOrder(..))
5         && this(s) {
6       User user = s.userdata;
7       Book[] books = s.booksdata;
8       int total = s.total;
9       // 注文処理の実行
10    }
11  }
```

AspectJ を用いた設計では MySQLOrderDAO アスペクトに order メソッドを呼び出すアドバイスが含まれているが、この処理は通常のオブジェクト指向による設計での OrderingService クラスの addShippingRateAndOrder メソッドの最後にある order メソッド呼び出しに対応する。これらは異

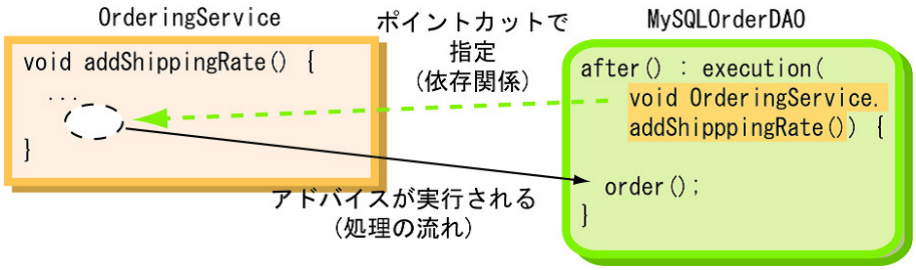

図 2.7: AspectJ を用いて設計した注文発注プログラム

なる位置に記述されているが、AspectJ を用いたプログラムが実行する処理の内容は 通常のオブジェクト指向を用いて設計したプログラムの処理の内容と変わらない。このアドバイスと結びついたポイントカットは addShippingRateAndOrder メソッドの最後をアドバイスを呼び出す箇所に指定する。

MySQLOrderDAO アスペクトはプログラム中で自動的に生成されるため、通常のオブジェクト指向による設計にあった MySQLOrderDAO コンポーネントを生成するコードや、MySQLOrderDAO コンポーネントを指すフィールド宣言に対応するコードを記述する必要はない。生成の方法はアスペクトの冒頭に記述された perthis 指定子が指定する。

一般的にアスペクト指向プログラミングでは、元のプログラムとの関係が薄い処理をアスペクトとして記述することが多い。そのためデータベースへアクセスするための処理をアスペクトとして記述することは適切でないと考えるかもしれないが、それは正しい考え方とは言えない。例えば MySQLOrderDAO のメソッド呼び出しを取り除きやすくすることで OrderingService クラスの単体テストが容易になる。また商品の売れ行きを管理するために異なるデータベースへのデータ送信処理が必要になったときには、データ送信の処理をアドバイスとして記述することで、元のプログラムへの変更を防ぐことができる。そして実際のオンライン本屋プログラムでは、注文履歴の表示や注文した商品の発送状況の確認など、複数の処理が発注管理データベースへアクセスすることが考えられる。このような場合、発注管理データベースへアクセスするためのメソッド呼び出しをアドバイスとして記述することで、このメソッド呼び出しのコードを 1 箇所にまとめ、コードの変更を容易にすることができる。

AspectJ を用いた設計の問題点

AspectJ を用いたプログラムを構成するコンポーネントの組み合わせを変更しても、OrderingService コンポーネントは変更せずに再利用することができる。なぜなら order メソッドの呼び出しや MySQLOrderDAO コンポーネントを指すフィールド宣言など、結合のためのコードが OrderingService コンポーネントから取り除かれているためである。例えば 2.1.1 節で示した変更 1 は MySQLOrderDAO アスペクトの代わりに OracleOrderDAO アスペクトを定義して用いることで実現できる。また変更 2 は新たに SecurityManager アスペクトを定義することで実現できる。

しかし AspectJ を用いた設計では、コンポーネントを通常のクラスでなくアスペクトとして設計しなければならない。アスペクトとして設計すると様々なプログラミング上の制約を受ける。この制約はアスペクト指向言語により異なる。例えば AspectJ のアスペクトは、アスペクトを独自の文法を用いて記述しなければならない、アスペクトをコンストラクタを直接呼んで生成することができない、抽象アスペクトではないアスペクトを定義するときに別な抽象アスペクトではないアスペクトを親アスペクトに利用できないといった制約を伴う。また通常のクラスとして設計された既存のコンポーネントをアスペクトとして用いることもできない。

また AspectJ が開発者に与える制約の 1 つに、アスペクトを用いて設計したコンポーネントと別のコンポーネントの結びつけを自由に指定できないという制約がある。これはアスペクトとコンポーネントとの結びつけ方がデプロイメント指定子によって限定されているからである。AspectJ では issingleton、perthis といった指定子を提供しているが、これらの指定子がサポートしていない結合をおこなうのは困難である。例えば、あるフィールドに同じ値を持った複数の OrderingService コンポーネントに 1 つの MySQLOrderDAO アスペクトを結びつけることは難しい。

また AspectJ を用いた設計では、アスペクトが結びつくコンポーネントを変更しようとしたときにアスペクトの内部の変更が必要となってしまう。例えば注文発注プログラムの例で、予約登録を行う ReservingService コンポーネントも MySQLOrderDAO コンポーネントを使ってデータベースアクセスを行うよう変更するとする。このとき MySQLOrderDAO アスペクトを修正せず再利用することはできない。アドバイスのポイントカットを変更する必要がある。

2.2.4　関連研究

AspectJ のアスペクトを用いた設計では、開発者は様々な制約に従ってコンポーネントを設計しなければならない。開発者に与える制約が AspectJ

に比べて少ないアスペクト指向システムは存在するが、全く制約を与えないシステムはない。

AspectJ が与える制約の 1 つに、コンポーネントの結びつけ方が限定されるという制約がある。これを解決しようとするアスペクト指向システムとして、Association Aspects [14] や Eos [12] 、Eos-U [13] がある。これらのアスペクト指向システムは、複数のコンポーネントに対してアスペクトを結びつけることを可能にする。Association Aspects は新たなインスタンス化指定子として perobjects という指定子を持つ。また Eos、Eos-U では instance-level aspects というアスペクトを提案している。これはクラスのインスタンスごとに異なるアスペクトを結びつけることを可能にする言語機構である。またこれらのシステムは new 式を用いてアスペクトを明示的に生成することを可能にしている。さらに Eos-U では、アドバイスを通常のクラスのメソッドとして実装する。そのためより通常のクラスに近い形でアスペクトを記述することができる。

しかしこれらのシステムが提供するコンポーネントの生成と結びつけの機能には制限がある。例えばある 1 つの OrderingService コンポーネントと結びつけた MySQLOrderDAO アスペクトを、その結びつきを保ったまま別の OrderingService コンポーネントと結びつけることはできない。またこれらのシステムでは AspectJ と同様に、アスペクトをそれぞれのアスペクト指向システムに依存した文法で実装しなければならない。Eos-U ではアドバイスを通常のメソッドとして記述できるが、ポイントカットはアスペクトの中に記述しなければならない。そのため言語拡張が必要となる。

また AspectJ では、アスペクトが結びつくコンポーネントを別なものに変更しようとしたときにアスペクトの内部の変更が必要になるという問題があった。これを解決しようとするアスペクト指向システムとして、AspectWerkz [2] や JBoss-AOP [8]、JAsCo [15]、CaesarJ [11] がある。これらのアスペクト指向システムは AspectJ のアスペクトの機能を、再利用が可能な部分とそうでない部分の 2 つに分けて記述する。

AspectWerkz や JBoss-AOP といったアスペクト指向システムではアスペクトをクラスとして、アドバイスをそのクラスのメソッドとして記述させ、ポイントカットなどを XML ファイルやアノテーションを用いて分離して記述させる。しかしアスペクトとして用いるクラスは文法上の制限を受ける。例えばアドバイスとして用いるメソッドの引数の型や個数、また投げる例外はシステムが定めたものに限定される。また AspectJ と同様にコンポーネントの結びつけ方が限定されるという問題がある。

JAsCo は再利用性の高いアスペクトを記述することを目標としたアスペクト指向システムである。JAsCo ではアスペクトの機能を aspect bean

と コネクタの 2 つに分けて記述する。aspect bean は AspectJ のアスペクトのアドバイスとポイントカットの機能の一部に対応し、コネクタはポイントカットの機能の一部とデプロイメント指定子に対応する。aspect bean にはポイントカットの機能のうち、アドバイスを実行するジョインポイントの種類のみを記述し、アスペクトが結びつくコンポーネントの具体的なシグネチャはコネクタに記述する。そのためアスペクトとコンポーネントの結びつき方が代わったときでも aspect bean の内部を変更する必要はない。しかし JAsCo では、横断的関心事を実装する aspect bean を JAsCo 独自の文法で記述する必要がある。そのため、あらかじめ JAsCo で用いることが想定されていないコンポーネントをアスペクトとして用いることはできない。

CaesarJ [11] は、アスペクトの機能のうち一部を再利用可能な形で記述することができる。CaesarJ は AspectJ と同じくポイントカット＆アドバイスの機構を持つが、既存のクラスにメンバを追加するインタータイプ宣言の機構は持たない。既存のクラスへのメンバの追加は、virtual class と多重継承という機構を用いて実現する。CaesarJ ではこの 2 つの仕組みを用いることで、追加したいメンバと追加先のクラスの指定を分離して記述することが可能である。そのためもし追加先のクラスが変わっても、追加したいメンバが書かれたコンポーネントを修正する必要はない。しかし CaesarJ では、アスペクトに用いるコンポーネントはシステムに依存した文法で記述しなければならない。

第3章　glue コードを用いた設計

コンポーネントの内部を変更せずにコンポーネントの組み合わせを変えられるようにするプログラムの設計手法として、glue コードを用いた設計を提案する。glue コードを用いて設計したプログラムを構成するコンポーネントの再利用性は高い。また glue コードを用いた設計を実現するためのシステムとして GluonJ を提案する。

3.1　glue コード

glue コードを用いた設計は、コンポーネントの内部を変更せずにコンポーネントの組み合わせ方を変えることを可能にする。glue コードを用いた設計ではオブジェクト指向を用いた設計とは異なり、コンポーネントを結びつけるコードをコンポーネントの外側に分離して記述する。この分離したコードを記述するためのモジュールが glue コードである。一般にコンポーネントを結びつけるためには以下の 3 通りのコードが用いられるが、glue コードはこれらのコード全てをコンポーネントから分離して記述する機能を持つ。

- フィールド宣言

  関連する別のコンポーネントを参照するフィールド。オブジェクト指向を用いた注文発注プログラムの例で OrderingService コンポーネントの dao フィールドは MySQLOrderDAO コンポーネントを指す。

- サブコンポーネントの生成と代入

  関連するコンポーネントを生成し、それをフィールドに代入するコード。コンポーネントの生成方法にはコンストラクタを用いた方法やファクトリメソッドを用いた方法がある。注文発注プログラムの例で OrderingService コンポーネントは、コンストラクタ内で MySQLOrderDAO 型のコンポーネントを生成し、dao フィールドへ代入している。

- サブコンポーネントのメソッド呼び出し

別のコンポーネントのメソッドを呼び出すコード。注文発注プログラムの例で OrderingService コンポーネントは addShippingRateAndOrder メソッドの最後で MySQLOrderDAO コンポーネントの order メソッドを呼び出している。

glue コードはアスペクト指向プログラミングにおけるアスペクトの一種であるだが、コンポーネントの結びつける機能に特化しているという特徴がある。一般にアスペクトは横断的関心事を実装し、それを実行するコンポーネントを結びつける働きを持つ。例えば AspectJ [1, 10] では、アドバイスを用いて横断的関心事を実装し、ポイントカットを用いて横断的関心事とコンポーネントを結びつける。しかしこの glue コードを用いた設計では、アスペクトの利用を、横断的関心事とそれを実行するコンポーネントとの結びつけのみに限定する。glue コードを用いた設計では、横断的関心事の実装にはアスペクトではなく通常のオブジェクトを用いる。

glue コードを用いた開発では、コンポーネント間の結合のうち後から変更される可能性が高い結合を glue コードに記述する。これはデザインパターンを用いた設計におけるファクトリメソッド、Dependency Injection を用いた設計における設定ファイルに対応するが、これらの設計ではコンポーネントを結びつけるための 3 種類のコードのうち、生成と代入のコードしか分離して記述することができなかった。一方で glue コードを用いた開発は、アスペクト指向プログラミングを用いることで、3 種類のコード全てをコンポーネント内部から分離することを可能にする。

3.2　GluonJ：glue コードを用いた設計を支援するアスペクト指向システム

我々は glue コードを用いて再利用性の高いコンポーネントを設計することを支援する Java 言語用のアスペクト指向システムとして GluonJ を開発した。AspectJ のアスペクトを glue コードの代用に使うことも可能だが、GluonJ を用いることでコンポーネントの結びつきをより直感的に記述することが可能になる。

3.2.1　GluonJ を用いたプログラムの例

2 章で例として用いた注文発注プログラムを GluonJ を使って記述し、それを用いて GluonJ が提供する glue コードの機能とその使い方を示す。GluonJ を用いて設計した注文発注プログラムを図 3.1 に載せる。GluonJ を用いた注文発注プログラムは、 MySQLOrderDAO コンポーネ

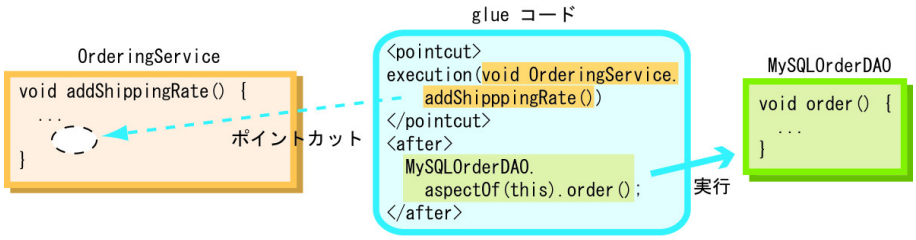

図 3.1: GluonJ を用いて設計した注文発注プログラム

ントと OrderingService コンポーネント、それらの組み合わせを記述した glue コードの 3 つから構成される。MySQLOrderDAO コンポーネントと OrderingService コンポーネントのソースコードを以下に載せる。

```
1   public class OrderingService {
2     private User userdata;
3     private Book[] booksdata;
4     private int total;
5     private String id;
6
7     public void addShippingRateAndOrder (
8         User user, Book[] books) {
9       total = 0;
10      for (int i = 0; i < books.length; i++)
11        total += books[i].getPrice();
12      if (total < 1500)
13        total += getShippingRate(user);
14      userdata = user;
15      booksdata = books;
16    }
17  }
```

```
1   public class MySQLOrderDAO extends OrderDAO {
2     public void order (User user,
3         Book[] books, int total) { ... }
4   }
```

オブジェクト指向による設計との違いは、コンポーネント同士を結びつけるコードがコンポーネント内に含まれていないことである。OrderingService コンポーネントと MySQLOrderDAO コンポーネントと結びつけるコードは以下の glue コードに記述する。GluonJ の glue コードは XML を用いて記述する。

```
1   <glue>
2     <reference>
```

```
3       MySQLOrderDAO OrderingService.
4           aspect = new MySQLOrderDAO();
5     </reference>
6     <behavior>
7       <pointcut>
8         execution(void OrderingService.
9             addShippingRateAndOrder(..))
10      </pointcut>
11      <after>
12        MySQLOrderDAO.aspectOf(this).order(
13            userdata, booksdata, total);
14      </after>
15    </behavior>
16  </glue>
```

glue コードに含まれる <reference> タグと <behavior> タグが、コンポーネントを結びつける役割を持つ。

<reference> タグ

<reference> タグは、コンポーネントを結合するために用いられる 3 通りのコードのうち、フィールド宣言と、サブコンポーネントを生成し代入するコードの 2 つを分離して記述するためのタグである。このタグは既存のコンポーネントにフィールドを新たに追加する機能と、コンポーネントの生成時に別のコンポーネントを生成してそれをフィールドに代入する機能を持つ。このうちコンポーネントの生成は Java コードを用いて明示的におこなう。上の例の <reference> タグの記述は、OrderingService コンポーネントに新しくフィールドを追加することと、 OrderingService コンポーネントの生成時にコンストラクタを呼んで MySQLOrderDAO コンポーネントを生成し、それを追加したフィールドに代入することを意味する。

上の glue コードに含まれる <reference> タグの記述は MySQLOrderDAO コンポーネントを指すフィールドとして OrderingService クラスの aspect というフィールドを指定している。これは OrderingService クラスの既存のフィールドではなく、匿名フィールドという特別なフィールドである。コンポーネントを代入するフィールドとしてあるコンポーネントの匿名フィールドを指定すると、そのコンポーネントには新たにフィールドが追加される。そして追加されたフィールドにコンポーネントが代入される。aspect は匿名フィールドを表すキーワードで、実際に追加されるフィールドの名前はシステムによって自動的に決められる。匿名フィールドの値を取得するには aspectOf メソッドを用いる。aspectOf メソッドについては 3.2.1 節で詳しく説明する。

匿名フィールドではなく特定の名前を持つフィールドを追加して、そのフィールドにコンポーネントを代入することも可能である。この機能は AspectJ のインタータイプ宣言と似た機能である。これらの機能の違いについては、 3.2.4 節で述べる。次の記述は匿名フィールドの代わりに link フィールドを追加して、そのフィールドにサブコンポーネントを代入する例である。

```
<reference>
  OrderDAO OrderingService.link = new MySQLOrderDAO();
</reference>
```

匿名フィールドの追加と特定の名前を持つフィールドの追加との違いは、既存のフィールドとの名前の衝突を意識する必要があるかどうかである。特定の名前を持つフィールドを追加する場合、そのフィールドには既存のフィールドと異なる名前をつけなければならない。なぜなら追加したフィールドと既存のフィールドとの区別ができなくなるからである。一方、匿名フィールドを追加する場合は名前の衝突を意識する必要はない。匿名フィールドには既存のフィールドの名前と衝突しない名前が、システムにより自動的に与えられるからである。この衝突を回避する機能は AspectJ における private フィールドのインタータイプ宣言の持つ機能と同様である。

一方、<reference> タグでは、生成したコンポーネントを既存のフィールドに代入することも可能である。 生成したコンポーネントのメソッドをコンポーネントから直接呼びたいときには、この既存のフィールドを用いた方法が有効である。OrderingService クラスに OrderDAO 型の dao フィールドが用意されていた場合には、次の記述により MySQLOrderDAO コンポーネントを dao フィールドに代入できる。なお dao フィールドの型は定義してもしなくてもよい。

```
<reference>
  OrderingService.dao = new MySQLOrderDAO();
</reference>
```

<behavior> タグ

<behavior> タグはコンポーネントを結びつけるために用いられる 3 種類のコードのうち、メソッド呼び出しをコンポーネントから分離して記述するためのタグである。この機能は AspectJ のアドバイスに似ている。<behavior> タグとアドバイスの違いは 3.2.4 章で詳しく示す。<pointcut> タグ内の記述は、コンポーネント内のどこでメソッドを呼び出すかを指定

する。<pointcut> タグ内の記述に用いる言語は AspectJ の提供するポイントカット言語とほぼ同じである。

<after> タグは <pointcut> タグが指定した箇所の直後に呼び出すブコンポーネントのメソッドを指定する。実行する処理は Java コードで指定する。このタグの中では任意の Java コードに加えて、aspectOf メソッドを利用できる。これは匿名フィールドが指すコンポーネントを取得するためのメソッドで、以下のような仕様をしている。

static *Aspect Aspect*.aspectOf(*Target* target);

aspectOf メソッドの戻り値は、引数の *Target* コンポーネントの匿名フィールドが指す *Aspect* コンポーネントである。この aspectOf メソッドは、*Aspect* 型の匿名フィールドの追加にともなって *Aspect* クラスへ追加される。

上の <behavior> タグの記述は OrderingService コンポーネントの addShippingRateAndOrder メソッドの実行をポイントカットを用いて指定し、その処理の直後に、OrderingService コンポーネントの匿名フィールドが指す MySQLOrderDAO コンポーネントの order メソッドを呼び出すことを意味している。

3.2.2　GluonJ による結合の分離

3.1 節で、glue コードはコンポーネントを結合するために用いられる 3 通りのコードを分離する機能を持つものだと定義した。ここでは GluonJ の提供する glue コードがこの条件を満たしていることを示す。

- フィールド宣言

  <reference> タグを用いることで、フィールドをコンポーネントから分離して記述することができる。<reference> タグは別のコンポーネントを指すためのフィールドをコンポーネントへ新たに追加できる。追加できるフィールドには特定の名前を持つフィールドと匿名フィールドがある。匿名フィールドを用いれば、既存フィールドとの名前の衝突を意識する必要はなくなる。

- サブコンポーネントの生成と代入

  <reference> タグを用いることで、コンポーネントを生成し代入するコードをコンポーネント内部から分離して記述することができる。コンポーネントの生成方法は Java コードを用いて明示的に記述できる。Java コードを用いるため複雑な生成も可能である。生成に複雑な処理を必要とする場合には、glue コードとは別に Java でファ

クトリメソッドを定義し、それを <reference> タグ内で利用することが可能である。

また <reference> タグでは、生成したコンポーネントを代入するフィールドとして、既存のフィールドと新たに追加されたフィールドのどちらも指定することができる。

- サブコンポーネントのメソッド呼び出し

  <behavior> タグを用いることで、コンポーネントの処理の呼び出しをコンポーネント内部から分離して記述することができる。<behavior> タグは、処理を呼び出す箇所の指定にはポイントカット記述を、呼び出す処理の指定には Java コードを用いる。匿名フィールドが指すコンポーネントの処理を呼ぶときには、aspectOf メソッドを用いてコンポーネントを取得する。

  なおポイントカット記述では指定ができない箇所で別のコンポーネントの処理を呼び出したい場合には、行アノテーションというコーディング技術が利用できる。これについては 4.1.2 節で詳しく説明する。

3.2.3　glue コードを用いて再利用性が向上したコンポーネントの例

glue コードを用いてプログラムを設計することでコンポーネントの再利用性が向上することを、GluonJ を用いて設計した注文発注プログラムの例を用いて具体的に示す。 2.1.1 節では 2 つの変更例を挙げたが、どちらの変更も GluonJ を用いて設計したプログラムに対しては glue コードを変更するだけで実現できる。まず変更 1 を実現するには、glue コードを次のように書き換えればよい。

```
1   <glue>
2     <reference>
3       OracleOrderDAO OrderingService.
4           aspect = new OracleOrderDAO();
5     </reference>
6     <!-- フィールドと生成方法の変更 -->
7     <behavior>
8       <pointcut>
9         execution(void OrderingService.
10            addShippingRateAndOrder(..))
11      </pointcut>
12      <after>
13        OracleOrderDAO.aspectOf(this).sendHint(this);
```

```
14          <!-- sendHint 呼び出しの追加 -->
15          OracleOrderDAO.aspectOf(this).order(
16              this.userdata, this.booksdata, this.total);
17        </after>
18      </behavior>
19    </glue>
```

変更した glue コードの箇所は 2 つある。1 つは <reference> タグにある MySQLOrderDAO のフィールドの追加とコンストラクタ呼び出しを OracleOrderDAO に書き換えたことである。もう 1 つは <behavior> タグに sendHint メソッドの呼び出しを追加したことである。

また変更 2 を実現するには glue コードを次のように書き換えればよい。

```
1   <glue>
2     <reference>
3       MySQLOrderDAO OrderingService.aspect =
4           new MySQLOrderDAO();
5       SecurityService OrderingService.aspect =
6           SecurityService.factory();
7     </reference>
8     <!-- フィールドと生成のコードを追加 -->
9
10    <behavior>
11      <pointcut>
12        execution(void OrderingService.
13            addShippingRateAndOrder(..))
14      </pointcut>
15      <after>
16        MySQLOrderDAO.aspectOf(this).order(
17            this.userdata, this.booksdata, this.total);
18      </after>
19    </behavior>
20
21    <!-- check メソッドの呼び出しを追加 -->
22    <behavior>
23      <pointcut>
24        execution(void OrderingService.
25            addShippingRateAndOrder(..))
26      </pointcut>
27      <before>
28        SecurityService.aspectOf(this).check(this.id);
29      </before>
30    </behavior>
31  </glue>
```

変更した glue コードの箇所は 2 つある。まず <reference> タグを用いて SecurityManager コンポーネントのフィールドを追加し、ファクトリメソッ

ドを呼び出して SecurityManager コンポーネントを生成、代入するコードを追加する。次に <behavior> タグを用いて check メソッドの呼び出しを記述する。

また 2.1.1 節で挙げた変更例以外の予期せぬ変更に対しても、コンポーネントの内部を変えずに行うことができる。それはコンポーネントを結びつけるコードが全て glue コードに分離されているためである。デザインパターンを用いて設計したプログラムで予期せぬ変更を行うには、コンポーネントの内部を変更する必要があった。

3.2.4　AspectJ のアスペクトを glue コードの代わりに用いることの問題点

AspectJ のアスペクトを glue コードとして用いることもできるが、結びつきの記述が煩雑になるという欠点がある。AspectJ のアスペクトを glue コードの代わりに用いて設計した注文発注プログラムは、GluonJ の glue コードを用いた設計と同様に OrderingService コンポーネントと MySQLOrderDAO コンポーネント、それらを組み合わせる glue コードの 3 つから構成される。OrderingService コンポーネントと MySQLOrderDAO コンポーネントは GluonJ を用いて設計した場合の Java クラスと同じものを用いる。glue コードの代用となる Glue アスペクトは以下に示す。

```
1   public privireged aspect Glue {
2     private MySQLOrderDAO OrderingService.dao =
3         new MySQLOrderDAO();
4
5     after(OrderingService s) :
6         execution(void OrderingService.
7         addShippingRateAndOrder(..)) && this(s) {
8
9       s.dao.order(s.userdata, s.booksdata, s.total);
10    }
11  }
```

この Glue アスペクトの機能は、GluonJ を用いて設計した注文発注プログラムの glue コードの機能と同等である。Glue アスペクトは 1 つのインタータイプ宣言と 1 つのアドバイスから構成される。インタータイプ宣言とは既存のクラスに対してフィールドやメソッドを追加する機能である。Glue アスペクトのインタータイプ宣言は、GluonJ を用いて記述した glue コードの <reference> タグに対応する。またアドバイスは GluonJ を用いて記述した glue コードの <behavior> タグに対応する。

AspectJ のアスペクトの持つ機能は glue コードの代用に使うには不十分である。なぜなら AspectJ のアスペクトは、コンポーネントを既存の

フィールドに代入するコードを直接分離するための機能を備えていないからである。AspectJ のアスペクトを使ってコンポーネントを既存のフィールドへ代入するには、アドバイスを用いて代入するコードを記述しなければならない。このとき代入のコードは長く煩雑になってしまう。例えば注文発注プログラムの例で、dao フィールドが既に OrderingService コンポーネントに存在しているとする。このとき MySQLOrderDAO コンポーネントを dao フィールドに代入するコードは以下のようにアドバイスを用いたものになる。

```
after(OrderingService s) :
    execution(OrderingService.new(..)) && this(s) {

  s.dao = new MySQLOrderDAO();
}
```

一方 GluonJ の glue コードでは、<reference> タグを用いて既存フィールドへ代入するコードを短く簡潔に記述することができる。上のアドバイスと同等の機能を持つ <reference> タグの記述は以下のようになる。

```
<reference>
  OrderingServoce.dao = new MySQLOrderDAO();
<reference>
```

<reference> タグはサブコンポーネントの生成と代入の分離を目的として設計された言語機構であるため、追加されたフィールドへも既存のフィールドへも同じようにコンポーネントを代入することができる。

また AspectJ のアスペクトを glue コードの代わりに使った場合、ジョインポイントの文脈情報を取得するための記述が煩雑になってしまう。ジョインポイントの文脈情報は、glue コードから他のコンポーネントのコンストラクタやメソッドを呼び出すときに必要となるため、文脈情報の取得が煩雑なことは問題である。本節の例では、 Glue アスペクトのアドバイスは OrderingService コンポーネントの userdata フィールドや booksdata フィールドの指すコンポーネントを取得し、それを引数に渡して MySQLOrderDAO コンポーネントの order メソッドを呼び出している。これらの文脈情報を取得するためには OrderingService コンポーネントの参照を取得する必要があるが、そのためにはまずポイントカット変数 s を宣言する必要がある。そして this ポイントカットを用いて OrderingService コンポーネントを s に束縛するという手順を踏まなければならない。

一方 3.2.1 節で示した GluonJ の glue コードでは、Java コードの中で this 変数を使うことで OrderingService コンポーネントへの参照を取得できる。ポイントカット変数や this ポイントカットを使う必要はない。

結びつけの記述が間接的になる原因は、AspectJ のアスペクトが独立したコンポーネントであるためである。Glue アスペクトから見て OrderingService コンポーネントは異なるコンポーネントである。そのため OrderingService コンポーネントやそのメンバにアクセスするには、まず this ポイントカットなどを用いてコンポーネントへの参照を取得し、そしてポイントカット変数を通じて間接的にアクセスしなければならない。一方 GluonJ の glue コードは独立したコンポーネントではなく、他のコンポーネントの一部である。そのため <behavior> タグの Java コードは OrderingService コンポーネント内部のコードと同じスコープを持つので、メンバに直接アクセスすることができる。

また同様の問題として、ジョインポイントの文脈情報が private フィールドに保存されているとき、AspectJ のアスペクトはこの文脈情報へアクセスできないという問題がある。注文発注プログラムの例では OrderingService コンポーネントの private フィールドがこれにあたる。アクセスできない原因は、上で述べた問題の原因と同じく、AspectJ のアスペクトが独立したコンポーネントであることである。一方 GluonJ の glue コードは、ジョインポイントを含むコンポーネントの private フィールドにアクセスできる。これは GluonJ の glue コードがジョインポイントを含むコンポーネントの一部だからである。

なお glue コードに privileged アスペクトを用いれば、glue コードが横断的関心事と結びつくコンポーネントの private フィールドへアクセスできるようになるが、privileged アスペクトの利用はプログラムの情報隠蔽を無効化するという問題を起こす。AspectJ の提供する privileged アスペクトは全てのクラスの private メンバをアクセスすることが可能である。Glue アスペクトは privileged アスペクトとして宣言されているため OrderingService コンポーネントの private メンバへアクセスできる。しかし privileged アスペクトとして実装した glue コードは横断的関心事を実行するコンポーネント以外の private メンバにアクセスすることも可能になってしまう。例えば privileged アスペクトとして実装した Glue アスペクトは MySQLOrderDAO コンポーネントの private メンバへのアクセスも可能になる。これはプログラムの情報隠蔽を無効化してしまう。GluonJ の glue コードからアクセスできる private メンバは上の例では OrderingService の private メンバのみであり、プログラムの情報隠蔽を無効にすることはない。

第4章　GluonJ の機能と実装

この章ではまず GluonJ の glue コードの機能と使い方について説明する。次に GluonJ の glue コードを用いて設計したプログラムを動かすために必要な、処理系の実装について説明する。

4.1　glue コードの機能

GluonJ の glue コードは XML (Extensible Markup Language) [18] を用いて記述する。glue コードの最上位のタグは <glue> タグであり、<glue> タグの中には <reference> タグ、<behavior> タグ、<pointcut-decl> タグ、<import> タグを記述することができる。これら 4 種類のタグの機能と使い方を以下で説明する。

4.1.1　<reference> タグ

<reference> タグは，コンポーネントを結びつけるために用いられる 3 通りのコードのうち，コンポーネントを指すフィールドと，コンポーネントを生成し代入するコードを分離して記述するためのタグである．以下のコードは OrderingService クラスに Logger 型の匿名フィールドを宣言することを意味する。

```
<reference>
  Logger OrderingService.aspect = new Logger($this);
</reference>
```

匿名フィールドの初期値には宣言の右側に書かれた式の計算結果が用いられる。匿名フィールドの初期化は通常のフィールドの初期化が行われるタイミングと同じタイミングで行われる。初期化に用いる式では、通常の Java 式に加えて $this という変数を用いることができる。この変数は、匿名フィールドが追加されたオブジェクトの値を指す変数である。上の例では$this は OrderingService 型の変数である。

匿名フィールドへのアクセスには aspectOf メソッドを用いる。aspectOf メソッドは glue コード内からのみ利用できる。例えば上の <reference>

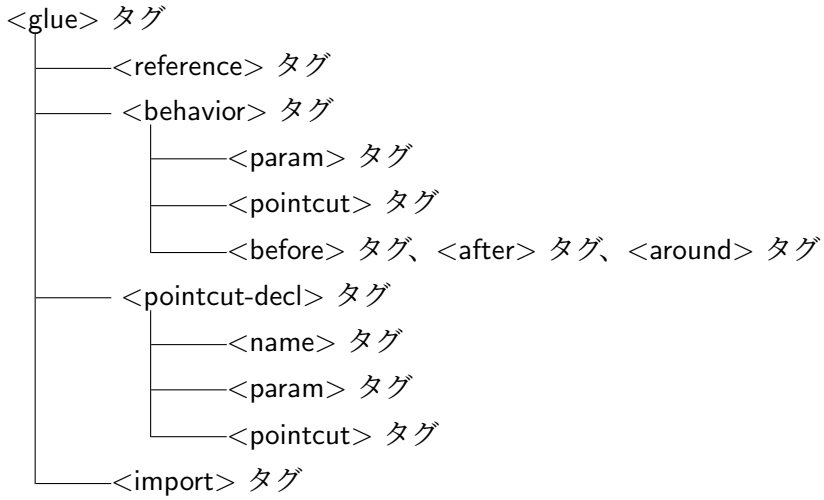

図 4.1: glue コードで利用できるタグの階層構造

タグで宣言された匿名フィールドにアクセスするには、Logger クラスに宣言された次のシグネチャを持つ aspectOf メソッドを用いる。

```
public static Logger aspectOf(OrderingService o);
```

このメソッドは変数 o が指す OrderingService オブジェクトに宣言された Logger 型の匿名フィールドの指す値を返す。

また匿名フィールドを static キーワードを付けて宣言すると、クラスには static フィールドが追加される。static 宣言された匿名フィールドは、通常の static フィールドが初期化されるタイミングで初期化される。static 宣言された匿名フィールドの指す値を取り出すには、aspectOf メソッドの引数にクラス名を表す String 型の変数を渡せばよい。

<reference> タグでは匿名フィールドではなく通常のフィールドを宣言することもできる。通常のフィールドを宣言したときには、aspect の代わりに指定された名前のフィールドが追加される。もし追加先のクラスに指定した名前のフィールドが既に存在するときにはそのフィールドの初期化のみが実行される。

特定の実行フロー内で有効な匿名フィールド

<reference> タグは，特定の実行フローの中でのみ有効な匿名フィールドを追加する機能も持つ．この機能は AspectJ が提供する percflow 指定子と同等の機能である．特定の実行フローの中でのみ有効な匿名フィールドを追加するには，次のように匿名フィールドの追加先のクラスとして

Cflow クラスを指定する．Cflow クラスは実行フローを表す特別なクラスである．

```
<reference>
  OrderDAO Cflow(call(void OrderingService.
      addShippingRateAndOrder(..))).aspect =
      new MySQLOrderDAO();
</reference>
```

Cflow クラスの後ろに書かれた括弧の中の記述は，どの実行フローの中で匿名フィールドを有効にするかを指定する．この括弧の中の記述には <pointcut> タグの内容を記述するときに用いるポイントカットの記述言語を用いる．この <reference> タグの記述は， addShippingRateAndOrder メソッドの呼び出しの間だけ有効になる OrderDAO 型の匿名フィールドを追加する．

このフィールドの初期化は、引数のポイントカット記述が指定するジョインポイントを実行する直前に行われる。初期化に使われたオブジェクトは、ジョインポイントを実行した直後に破棄され、初期化が行われる直前の値に戻される。例えば上のコード例で addShippingRateAndOrder が再帰的に呼び出されたとき、匿名フィールドは呼び出しのたびに初期化される。

このように追加された匿名フィールドの値を取得するには，aspectOf を次のように利用する．

```
<after>
  OrderDAO.aspectOf($thisCflow).order();
</after>
```

aspectOf メソッドの引数に渡されている $thisCflow 変数は，現在の実行フローを表す Cflow 型のオブジェクトである．aspectOf の引数に $thisCflow オブジェクトを渡すことで，現在の実行フローで有効な匿名フィールドの値を取得できる．$thisCflow は glue コードの <behavior> タグに書かれた Java コードの中のみで有効な変数である。

クラスのグループへの匿名フィールドの追加

<reference> タグは，クラスのグループに対して匿名フィールドを追加することもできる．クラスのグループに対して追加された匿名フィールドは，グループ内のクラスをそれぞれインスタンス化したオブジェクトのグループごとに異なる値を持つ．この匿名フィールドの構造はオブジェクトのグループをキーとする多重ハッシュテーブルのようなものである．

この機能は Association Aspects [14] が提供する perobjects 指定子と同等の機能である．次の <reference> タグの記述は OrderingService クラスと Session クラスのペアに OrderDAO 型の匿名フィールドを追加する．

```
<reference>
  OrderDAO OrderingService.aspect(Session) =
      new MySQLOrderDAO($this, $arg0);
</reference>
```

この <reference> タグの記述により OrderingService クラスに追加された匿名フィールドは，複数の OrderDAO コンポーネントを指すことができる．この匿名フィールドが指すコンポーネントは，Session コンポーネントをキーとして識別される．キーの型は aspect の後ろの括弧に記述された型により決まる．

この匿名フィールドが指すコンポーネントを取得するには，<behavior> タグの中で aspectOf を次のように利用する．

```
<after>
  OrderDAO.aspectOf(service, session).order();
</after>
```

この aspectOf メソッドは，service 変数の匿名フィールドが指す複数の OrderDAO コンポーネントのうち， session 変数をキーとする OrderDAO コンポーネントを返す．service 変数の型は OrderingService クラス，session 変数の型は Session クラスである．もし service 変数の匿名フィールドに session 変数がキーとなる OrderDAO コンポーネントが存在しないときには，aspectOf メソッド内でこれをキーとする OrderDAO コンポーネントが新たに生成される．この生成には <reference> タグ内に書かれた MySQLOrderDAO コンストラクタが用いられる．

上の例で MySQLOrderDAO コンストラクタへの引数となっている $this と $arg0 は特別な変数である．$this 変数は aspectOf メソッドの 1 番目の引数を表す変数であり、$arg0 変数は aspectOf メソッドの 2 番目の引数を表す変数である。もし 2 種類以上のオブジェクトをキーとする匿名フィールドを宣言したときには、キーの個数に応じて$arg1 変数、$arg2 変数などが利用可能になる。

4.1.2　<behavior> タグ

<behavior> タグは、他のコンポーネントのメソッドを呼び出す箇所とその箇所で呼び出すメソッドを指定する。このうちメソッドを呼び出す箇所を指定するには、<pointcut> タグを用いてポイントカットを記述す

る。ポイントカットを記述するための言語は AspectJ のものを利用している。GluonJ では AspectJ で利用可能なポイントカット指定子のうち、if ポイントカット指定子、adviceexecution ポイントカット指定子などを除いた主要なものが利用可能である。また GluonJ ではポイントカット指定子を連結させるのに用いる && 演算子と || 演算子の代わりに、それぞれ AND と OR を用いる。XML ファイルの中では & 記号のエスケープが必要になるためである。

また <behavior> タグ内では <param> タグを使うことでポイントカットパラメータを指定することができる。このポイントカットパラメータは AspectJ のポイントカットパラメータの働きと同じである。例えば以下の <behavior> タグでは、String 型の message というポイントカットパラメータを宣言し、ポイントカットとアドバイスの内部で利用している。

```
<behavior>
  <param><name>message</name><type>String</type></param>
  <pointcut>
    execution(void OrderingService.*(..)) AND
          args(message)
  </pointcut>
  <after>
    Logger.aspectOf(this).log(message);
  </after>
</behavior>
```

また呼び出すメソッドを指定するタグには <before> タグ，<after> タグ、<around> タグの 3 種類がある。<before> タグは <pointcut> タグが指定した箇所の直前に呼び出すメソッドを指定し，<after> タグは <pointcut> タグが指定した箇所の直後に呼び出すメソッドを指定する。<around> タグは <pointcut> タグが指定した箇所で実行される処理の代わりにおこなう処理を指定する．これらのタグでは任意の Java コードが記述できるが、<、> 、& 記号は <、>、& とエスケープして記述する必要がある。

処理を指定するタグの中では、通常の Java コードと aspectOf メソッドに加え、文脈情報を取得するための特別な変数を用いることができる。文脈情報は this、args などのポイントカット指定子を用いて取得することもできるが、特別な変数を用いることでさらに詳しい情報を取得することができる。利用できる変数は Javassist [19, 3] が提供するものに準じている。Javassist とは Java コードを表す文字列を Java のバイトコードに変換し、それを既存のクラスファイルに埋め込むことを可能とするライブラリである。GluonJ の処理系は内部で Javassist を用いている。またクラスファイルにデバッグ情報が付加されているときには、タグの中でジョインポイントから参照できるローカル変数を利用することも可能である。

なお <around> アドバイスで戻り値を指定するときには、AspectJ とは違い $_ 変数への代入という形で記述する。$_ 変数は Javassist が提供する変数の 1 つで、置き換える処理の戻り値を表している。例えば以下の <around> タグでは処理の戻り値に TestOrderDAO オブジェクトを指定している。

```
<behavior>
  <pointcut>
    execution(void OrderDAOFactory.getInstance())
  </pointcut>
  <around>
    $_ = new TestOrderDAO();
  </around>
</behavior>
```

行アノテーション

GluonJ は別のコンポーネントのメソッドを呼び出す箇所を指定するために <pointcut> タグを用いるが，<pointcut> タグでは指定できないプログラム中の箇所も存在する．例えば GluonJ のポイントカット記述言語を用いて，特定の for 文の直前を指定することはできない．

このような場合は行アノテーションと我々が呼ぶコーディング技術を用いて，コンポーネント内にある，サブコンポーネントの処理を呼び出したい箇所に目印をつければよい．行アノテーションとはコード内に記述する注釈のことである．例えば先に挙げた注文発注プログラムで，行アノテーションを用いて MySQLOrderDAO コンポーネントの order メソッドの呼び出し箇所に目印をつけるには，次のようにして OrderingService コンポーネントの addShippingRateAndOrder メソッドに行アノテーションを記述する．

```
public void addShippingRateAndOrder (
    User user, Book[] books) {
  ...
  DAO.send();
}
```

DAO.send() が行アノテーションである．行アノテーションは空の static メソッドの呼び出しとして記述する．行アノテーションに用いる static メソッドとクラスは開発者が用意する．このクラスやメソッドの名前は開発者が自由に決められる．

行アノテーションによって目印のついた箇所は <pointcut> タグを用いて指定できる．これには static メソッドの呼び出しを指定するポイント

カット記述を用いる．次の <pointcut> タグの記述は上の行アノテーションを指定する．

```
<pointcut>
  call(static DAO.send())
</pointcut>
```

なお行アノテーションは GluonJ にのみ有効な手法ではなく，他のアスペクト指向言語を用いたときにも有効な手法である．

ただし行アノテーションは、コンポーネントの設計時にコンポーネントの内部に記述しておかなければならない。そのため、もしコンポーネントの結合を変えるときになって初めて行アノテーションが必要となった場合、コンポーネントの内部の変更が必要となってしまう。

この方法はファクトリパターンを用いた設計で、OrderingService クラスに OrderDAO 型の dao フィールドを用意し、dao.order(user, books, total) と呼び出しを記述する方法と変わらないように見えるかもしれない。しかし行アノテーションを用いた方法は、メソッド呼び出しによる結合をより柔軟に変更することを可能にする。例えばファクトリパターンを用いた設計の場合、order メソッドの引数の型や個数が変わったときには OrderingService クラスの内部を修正して与えるパラメータを変更する必要がある。一方、行アノテーションを用いた方法では、order メソッドの引数の型や個数が変わったときでも OrderingService クラスの内部を修正する必要はない。これは行アノテーションを用いた方法では order メソッドの呼び出し位置のみが OrderingService クラスに記述されており、実際の呼び出しは glue コードに記述されているためである。

4.1.3　<pointcut-decl> タグ

<pointcut-decl> タグは、既存のポイントカット指定子を組み合わせて新たなポイントカット指定子を定義することを可能にする。これは AspectJ のポイントカット宣言と同等の機能である。新たなポイントカット指定子を定義するには、指定子の名前とポイントカット式を与える。指定子の名前は <name> タグで、ポイントカット式は <pointcut> タグで指定する。例えば以下の記述は execService というポイントカット指定子を新たに定義する。

```
<pointcut-decl>
  <name>execService</name>
  <param><name>message</name><type>String</type></param>
  <pointcut>
    execution(void OrderingService.*(..)) AND args(message)
```

```
  </pointcut>
</pointcut-decl>
```

pointcut-decl タグではポイントカットパラメータを宣言することもできる。ポイントカットパラメータを利用するには param タグを用いる。この定義されたポイントカットは、定義が記述された glue コードの中で有効である。

4.1.4　<import> タグ

<import> タグは、glue コードの中で Java クラスやメソッドのパッケージ名を省略して書くことを可能にする。これは Java の import 文と似た機能である。<import> タグの中にパッケージ名を記述すると、そのパッケージのクラスはパッケージ名の省略が可能になる。省略はその glue コードの中でのみ有効である。例えば以下の glue コードでは、java.util パッケージのクラスをインポートすることで、java.util.Map インタフェースと java.util.HashMap のパッケージ名を省略して記述している。

```
<glue>
  <import> java.util </import>
  <reference>
     Map OrderDAO.cache = new HashMap();
  </reference>
</glue>
```

<imoprt> タグを使うことでパッケージ名を省略できる箇所は、reference タグのフィールド宣言やメソッド宣言、pointcut タグのポイントカット記述、 before タグ、after タグ、around タグ内の Java コード内にあるクラスの記述である。

4.1.5　その他の機能

この節では GluonJ を用いて設計したプログラムで、コンポーネントから匿名フィールドにアクセスする方法と、コンポーネントから <around> アドバイスを用いて置き換えた処理を実行する方法を説明する。ただしこれらの方法を取るとコンポーネントの実装が GluonJ に依存してしまう。そのためコンポーネントの再利用性が低下してしまうという欠点もある。

リフレクション

GluonJ は、glue コードの外から匿名フィールドにアクセスすることを可能にするリフレクション機構を持つ。リフレクション機能を用いること

表 4.1: Aspect クラスの提供するメソッド

void add(Object target, Object aspect, Class clazz) target オブジェクトの clazz 型の匿名フィールドに、 aspect オブジェクトを代入する
Object get(Object target, Class clazz) target オブジェクトの clazz 型の匿名フィールドの指すオブジェクトを得る
void add(Object target, Object aspect, Class clazz) target オブジェクトの clazz 型の匿名フィールドが aspect オブジェクトを指していたら、そのリンクを外す

でコンポーネントから匿名フィールドの値を利用できる。リフレクション機能を可能にする Aspect クラスの static メソッドを表 4.1 に載せる。

proceed 機構

GluonJ の持つ proceed 機構は、<around> タグを使って置き換えた処理をコンポーネントの中で実行することを可能にする。この機能は AspectpJ の around アドバイス内で利用できる proceed メソッドの機能と同等である。proceed 機構の具体的な利用例として、OrderDAO の getData メソッドの実行を、その戻り値をキャッシュする Cache コンポーネントのメソッド実行に置き換えるという例を示す。getData() メソッドの実行を置き換える glue コードは以下のようになる。

```
<glue>
  <reference>
    Cache OrderDAO.aspect = new Cache();
  </reference>
  <behavior>
     <param><name>key</name><type>Object</type></param>
     <pointcut>
        execution(Object OrderDAO.getData(Object)) AND
        args(key)
     </pointcut>
     <around>
       $_ = Cache.aspectOf(this).getValue(key);
     </around>
  </behavior>
</glue>
```

置き換えた OrderDAO.getData() の戻り値をキャッシュする Cache クラスの実装は以下のようになる。

```
public class Cache {
  private Map cache = new HashMap();
  public Object getValue(Object arg) {
    if(!cache.containsKey(arg)) {
        Closure c = ClosureStack.peek();
        try {
          cache.put(arg, c.proceed());
        } catch(Throwable t) {}
    }
    return cache.get(arg);
  }
}
```

getValue メソッドはメソッドの戻り値を引数の値ごとにキャッシュし、メソッドが同じ引数で呼ばれたときにはキャッシュした値を返すという機能を持つクラスである。この Cache クラスは特定のクラスに依存しない形で実装されている。

Cache クラスでは ClosureStack.peek() を呼ぶことで <around> タグにより置き換えられた元の処理を表す Closure オブジェクトを取得している。取得した Closure オブジェクトの proceed() を呼ぶことで元の処理を実行できる。なお ClosureStack.peek() メソッドは、その処理が around タグの中から呼ばれていないときには null を返す。

4.2　GluonJ の処理系

4.2.1　GluonJ の提供する weaver

GluonJ は glue コードを Java プログラムに織り込むための weaver を 3 種類提供する。Weaver とは、アスペクト指向システムにおいてアスペクトを元のプログラムに織り込む (weave) 機能を持つ処理系を意味する。GluonJ の提供する weaver は 3 種類あり、それぞれ compile-time weaver、load-time weaver、JBoss 上で動かすアプリケーションのための load-time weaver である。

compile-time weaver は glue コードに書かれた処理を Java プログラムのコンパイル時に織り込む。compile-time weaver は Java のソースファイル (*.java) をコンパイルすることで生成されたクラスファイル (*.class) に変更を加え、glue コードの処理を織り込んだクラスファイルを出力する。

load-time weaver は glue コードに書かれた処理を、Java プログラムを実行する仮想機械がクラスをロードしようとするときに埋め込む。Java

プログラムを動かす仮想機械は、クラスをロードする際にクラスローダと呼ばれるモジュールを利用する。GluonJ の load-time weaver は、クラスをロードする際に glue コードの処理を埋め込む機能を持つ拡張クラスローダを用いてロード時に glue コードの処理を埋め込む。ロード時に織り込みをおこなう利点の 1 つは、glue コードの処理を織り込むときにプログラムの再コンパイルが不要だということである。

JBoss 上で動かすアプリケーションのための load-time weaver とは、アプリケーションサーバである JBoss [7] 上で動くアプリケーションに glue コードの処理を織り込むための weaver である。JBoss とは Java 用のアプリケーションサーバである。この JBoss 用 weaver は、JBoss を用いて動かすアプリケーションに対して glue コードに書かれた処理を織り込むことができる。Jboss 用 weaver を用いることで、glue コードの織り込みや織り込む glue コードの変更を、織り込む対象のアプリケーションを再コンパイルせずにおこなうことが可能になる。またこれは JBoss が提供するクラスファイルを変更する仕組みを用いて実装しているため、JBoss 自体を動かすためのクラスローダを変更したり、特別な仮想機械を用いたりする必要はない。

4.2.2　weaver の実装

GluonJ の提供する 3 種類の weaver は、クラスファイルへ glue コードの処理を織り込むために共通のコアモジュールを用いている。コアモジュールに含まれるクラスの 1 つが AOPWeaver クラスで、これはクラスファイルを入力として受け取り、glue コードに書かれた処理を織り込んだクラスファイルを出力する機能を持つ。特定の glue コードに書かれた処理を織り込む AOPWeaver オブジェクトを生成するには AOPWeaverFactory クラスを用いて生成する。AOPWeaverFactory が AOPWeaver オブジェクトを生成する過程を表したのが図 4.2 である。AOPWeaverFactory クラスは glue コードをコンパイルし、得られた情報を AOPWeaver オブジェクトに追加する。

コアモジュールに含まれる、クラスファイルを編集する部分の実装には Javassist [19] を利用している。Javassist は Java のクラスファイルを編集するためのライブラリである。Javassist の特徴は、バイトコードを直接編集するための API のほかにソースコードレベルの API を用意していることである。後者の API はバイトコードの知識を持たずに利用することができる。また load-time weaver の実装で、クラスのロード時にクラスファイルを編集する部分の実装にも Javassist を用いている。

また JBoss 用 weaver の実装には、UnifiedClassLoader の持つクラスファ

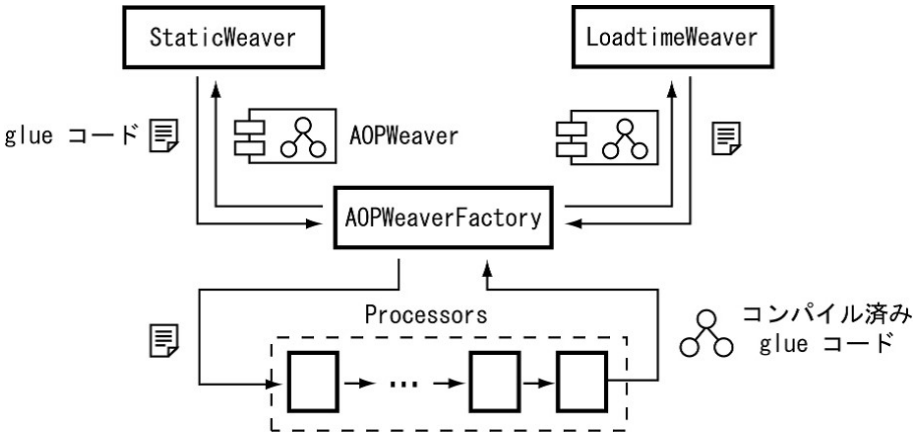

図 4.2: weaver が AOPWeaver オブジェクトを取得する流れ

イルを変換する機能を用いている。UnifiedClassLoader とは、JBoss がアプリケーションに含まれるクラスをロードするときに使う拡張クラスローダである。UnifiedClassLoader には、クラスをロードする処理をトラップし、クラスファイルを変更するための機能が用意されている。GluonJ の JBoss 用 weaver はこの機能を利用して glue コードの処理を織り込む。この機能を利用するために GluonJ は、クラスファイルを変換するクラスに共通のインタフェースである Translator インタフェースを実装した GluonJTranslator クラスを用意している。JBoss 用 weaver はこの GluonJTranslator クラスのオブジェクトを UnifiedClassLoader を束ねるオブジェクトである LoaderRepository オブジェクトにセットする。UnifiedClassLoader はクラスをロードするときに、自身が登録された LoaderRepository オブジェクトにセットされたトランスレータを用いてクラスファイルの変換をおこなう。

また JBoss 用 weaver を実装するにあたり、UnifiedClassLoader がクラスファイルに付属した正しい証明書を用いてクラスをロードしないというバグを修正している。UnifiedClassLoader はクラスをロードするために親クラスである ClassLoader クラスの defineClass メソッドを用いるが、この引数としてクラスファイルの配布元の証明書やアクセス権を表すオブジェクトである ProtectionDomain オブジェクトが必要になる。UnifiedClassLoader は、Translator オブジェクトがセットされていないときには ProtectionDomain オブジェクトを生成する処理を親クラスに任せるが、Translator オブジェクトがセットされているときにはこの処理を UnifiedClassLoader の中で行う。しかし UnifiedClassLoader の中で行われている処理には、ProtectionDomain オブジェクトを生成するときにクラ

Xerces	実行時間 (ms))
オリジナル	10692
GluonJ static メソッド	11975
AspectJ	13072

表 4.2: アドバイスの実行にかかるオーバヘッドの比較

スファイルに付属した証明書を取得する処理が抜けている。これにより証明書が付属したクラスファイルのロードに失敗することがある。そのため JBoss 用 weaver では、正しい証明書を取得する処理を GluonJTranslator の中で行っている。

4.3　実験

GluonJ は、アドバイスを static メソッドを用いて実装することを可能にする。static メソッドを用いる利点は、アドバイスの実行にかかるオーバヘッドを減らせることである。AspectJ ではアドバイスを static メソッドとして実装することはできない。AspectJ のアドバイスは weaver によりコンパイルされて Java のメソッドに変換されるが、このメソッドは 常に非 static のメソッドとなるためである。一方 GluonJ では、static メソッドの呼び出しをジョインポイントに織り込むことができる。

アドバイスに static メソッドを用いることでオーバヘッドを減らせることを確認するために実験をおこなった。この実験では、XML パーザである Xerces [16] にアドバイスを織り込み、その実行時間を計測している。アドバイスはカウンタを入れるためのものであり、Xerces ライブラリに含まれる全てのメソッド実行の先頭に織り込まれる。

実験では、アドバイスの織り込み方が異なる 3 種類の Xerces ライブラリを用意した。それぞれ static メソッドで実装したアドバイスを GluonJ の compile-time weaver を用いて織り込んだ Xerces ライブラリ、AspectJ の compile-time weaver を用いてアドバイスを織り込んだ Xerces ライブラリ、またアドバイスを織り込まなかった Xerces ライブラリである。これら 3 種類のライブラリを用いてサンプルプログラムを実行するのにかかった時間を比較した。サンプルプログラムは、XML ファイル (およそ 7MB) を読み込んでその要素の数を数えるプログラムである。実験には XML パーザである Xerces 2.6.1 を用いている。実験環境は、マシンが Sun Enterprise 450 、CPU が UltraSPARC-II 300MHz × 4、メモリが 1GB、OS が Solaris 8、JDK のバージョンが 1.4.0_01 である。

この実験の結果を表 4.2 に示す。static メソッドを用いて実装することで、サンプルプログラムに織り込まれたアドバイスの実行にかかるオーバヘッドは約 54% 減っている。このプログラムはアドバイスの実行回数が非常に多い。非常に多くのアドバイスを挿入する必要があるときに、この手法は特に有効である。

また static メソッドとしてアドバイスを実装することには、アスペクトをインスタンス化して使うことができないという欠点がある。しかしアドバイスのオーバヘッドをできるだけ小さくしたいときには有効な手法である。

第5章　まとめ

本稿はコンポーネントの内部を変えずにコンポーネントの組み合わせを変更できるプログラムの設計手法として、glue コードを用いた設計を提案した。glue コードを用いた設計では、アスペクト指向プログラミングを用いてコンポーネントの結合をコンポーネントの外側に記述する。コンポーネントの結合には 3 種類あり、それらをコンポーネントの内部から分離して記述するためのアスペクトが glue コードである。従来のアスペクト指向プログラミングではアスペクトをコンポーネントとして用いていたが、アスペクトの設計はアスペクト指向システムにより様々な制約を受けるため、開発者はそれらの制約を意識してコンポーネントを開発しなければならなかった。一方、glue コードを用いた設計では、コンポーネントを通常のオブジェクトとして実装するため、開発者はアスペクトとして実装されるかどうかを意識せずにコンポーネントを開発できる。また既存のコンポーネントをそのまま利用することも可能である。

従来のアスペクト指向言語を用いても glue コードの記述は可能だが、本稿はより glue コードに適したアスペクトを提供するアスペクト指向システムとして GluonJ を提案した。GluonJ では コンポーネントを結びつけるための 3 種類のコードを <behavior> タグと <reference> タグを用いて記述する。<behavior> タグは AspectJ の言語機構であるアドバイスを、<reference> タグは Dependency Injection と AspectJ の言語機構であるインタータイプ宣言を、コンポーネントの結び付けを記述するために再設計したものである。

glue コードを用いた設計はコンポーネントの内部を変えずにコンポーネントの結合を変更できるが、コンポーネントを見ただけでは別なコンポーネントとの関係が分からなくなるという欠点もある。そのため、コンポーネント間の結合を変更する可能性が低いときには、通常のオブジェクト指向を用いて記述したほうが良い場合もある。また 3 種類あるコンポーネント間の結合のうち、コンポーネントの生成と代入による結合のみが変更される可能性が高いときには、デザインパターンや Dependency Injection といった既存の技術による設計で十分な場合もある。

また GluonJ を用いることで横断的関心事を実装するコンポーネントを通常のクラスとして実装でき、またコンポーネントの結びつけを Java

コードで細かく指定することができるが、一方で GluonJ は、単純な生成や結びつけをおこなうときにもこのような指定を必要とする。単純な生成や結びつけで十分な場合には、AspectJ のように用意された指定子から選ばせる機構を持つアスペクト指向システムのほうが便利である。しかし、アスペクト指向の研究が進み、新たな利用例が見つかるにしたがって GluonJ の提供する細かな指定が必要になる機会は増えると我々は考えている。

参考文献

[1] AspectJ Project. Aspectj. `http://eclipse.org/aspectj/`.

[2] Jonas Boner and Alexandre Vasseur. Aspectwerkz 1.0. `http://aspectwerkz.codehaus.org/`.

[3] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 364–376, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[4] Codehaus. Pico Container. `http://www.picocontainer.org/`.

[5] Martin Fowler. Inversion of control containers and the dependency injection pattern. `http://www.martinfowler.com/articles/injection.html`.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[7] JBoss Inc. Jboss 4.0.2. `http://www.jboss.org/`.

[8] JBoss Inc. Jboss aop 1.0.0 final. `http://www.jboss.org/`.

[9] Rod Johnson and Juergen Hoeller. *Expert One-on-One J2EE Development without EJB*. Wrox, 2004.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP' 01*, pages 327–353, 2001.

[11] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03*, pages 90–99. ACM Press, 2003.

[12] Hridesh Rajan and Kevin Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE 2003*, pages 297–306. ACM Press, 2003.

[13] Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05*, pages 59–68. ACM Press, 2005.

[14] Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Kimoya. Association aspects. In *AOSD '04*, pages 16–25. ACM Press, Mar. 2004.

[15] Davy Suvee, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03*, pages 21–29. ACM Press, 2003.

[16] The Apache Software Foundation. Xerces 2.6.1. `http://xerces.apache.org/xerces2-j/`.

[17] The Seasar Project. Seasar. `http://homepage3.nifty.com/seasar/`.

[18] World Wide Web Consortium. Extensible markup language (xml). `http://www.w3.org/XML/`.

[19] 千葉 滋. Javassist home page. `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.