アスペクト指向技術を用いたORBミドルウェア開発方法

蜂巣吉成[†] 野呂昌満[†] 張漢明[†]

本論文ではアスペクト指向技術を適用したORBミドルウェアの開発方法を示す．ORBミドルウェアを階層モデルで表現すると複数の層を横断した構成要素が存在してしまう．我々は層をアスペクトとしてとらえてアスペクト間の関係を整理したソフトウェアアーキテクチャを提案する．アーキテクチャに基づいて，ORBのフレームワークとスタブ，スケルトン生成系の言語独立な設計方法を示す．ORBに関するアスペクトは独立したフレームワークとして実現する．スタブ，スケルトン生成系はウィーバの役割を果たし，ORBフレームワークを利用するためのコードが織り込まれたスタブ，スケルトンを生成する．提案した開発方法に基づいてXML-RPCのORBミドルウェアを実現し，ORBミドルウェアの開発が省力化できること，ORBミドルウェアの機能の追加が容易なことを確認した．

An Aspect-oriented Approach to ORB Middlewares Development

Yoshinari Hachisu,[†] Masami Noro[†] and Han-Myung Chang[†]

The paper describes an aspect-oriented approach to ORB middlewares development. In general, ORB middlewares are modeled as a layered model. We, however, detected crosscutting components in the layered model. To solve the crosscutting problem, we designed and employed an aspect-oriented software architecture for ORB middlewares. The architecture shows language independent design of a framework and a stub-skeleton generator. The generator is conceived as an aspect weaver which yields codes including the framework code. We demonstrated, through the implementation of ORB middleware for XML-RPC, that the architecture contributes to reduce the effort for the implementation and to build flexible and easy-to-modify middlewares.

1. はじめに

近年，分散オブジェクト技術を用いたアプリケーション開発が一般化している．代表的な分散オブジェクト技術としてCORBA[1)]，Java RMI[2)]，XML-RPC[3)]，SOAP[4)]などが提案され，ORBミドルウェア☆（**図1**）が開発されている．しかし，ORBミドルウェアの構造の整理が不十分であり，開発が個々の開発者の知識や技術，経験に依存するという問題がある．ORBミドルウェアの構造を階層モデルとして整理する試みがあるが[2)]，統一的かつ標準的な階層モデルが定義されてきたとはいえない．特定の技術に依存しない一般化した階層モデルを考えると，階層を横断した構成要素が存在し，適切にモデル化できないという問題が顕在化する．

本研究ではORBミドルウェアの構造を整理し，開発を省力化するために，アスペクト指向の概念を適用したORBミドルウェアの開発方法を提案する．アスペクト指向では，あるアスペクトを実現するコードが複数のアスペクトに散在するときに，それらをひとまとめにして分離することができる．ORBミドルウェアにおける複数の層を横断した構成要素をアスペクト指向の概念を用いて整理し，ソフトウェアアーキテクチャとして示す．

アーキテクチャに基づいて，ORBのフレームワークとスタブ，スケルトン生成系を設計・実現する．ORBに関するアスペクトは独立したフレームワークとして実現する．スタブ，スケルトン生成系はアスペクトを結合するウィーバの役割をし，ORBフレームワークを利用するためのコードが織り込まれたスタブ，スケルトンを生成する．ORBミドルウェアをアスペクトにより整理したことで，従来，各分散オブジェクト技術において個別に開発がなされていたスタブ，スケルトン生成系の構造を整理して，言語独立な実現方法を

† 南山大学数理情報学部
Faculty of Mathematical Sciences and Information Engineering, Nanzan University

☆ 本論文ではORBとスタブ，スケルトンのことをORBミドルウェアと呼ぶ．

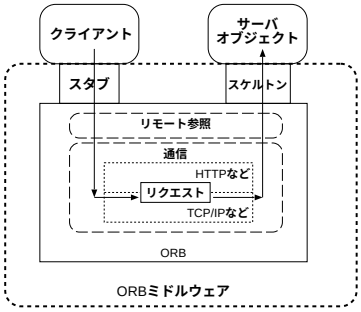


図 1 ORB ミドルウェア
Fig. 1 ORB middleware.

考察する．

XML-RPC の ORB ミドルウェアをフレームワークとスタブ，スケルトン生成系を利用して実現した．フレームワークの可変部をあつらえることで ORB を開発できた．スタブ，スケルトン生成系により ORB フレームワークを利用するためのコードをサーバオブジェクトの仕様から生成することができ，ORB ミドルウェアの開発が省力化できることを確認した．さらに，作成した XML-RPC の ORB ミドルウェアにデータの暗号処理を追加することを試みた．その結果，アスペクト指向を用いて構造を整理したことによって，機能の追加が容易であることを確認した．

2. 分散オブジェクト技術

表 1 に示す分散オブジェクト技術が考慮すべき性質[5),6)]や，ORB を特徴づけるメッセージ交換形式の観点から，CORBA，Java RMI，XML-RPC，SOAP を概観し，階層モデルとしてとらえた場合の問題点を指摘する．

2.1 既存の分散オブジェクト技術

CORBA

CORBA（Common Object Request Broker Architecture）は OMG によって提案された規格である[1)]．言語独立性，位置透過性を満たすためにサーバオブジェクトのインタフェースは IDL と呼ばれる言語で記述する．IDL コンパイラによってアプリケーション開発に用いる言語（Java や C++など）で書かれたスタブ，スケルトンのコードを生成する．

整列形式は CDR（Common Data Representation）と呼ばれるバイト順の選択が可能なバイナリ表現が用いられる．TCP/IP での通信規約には，IP アドレスやポート番号，CDR のバイト順などが格納さた IIOP（Internet Inter-ORB Protocol）が用いられる．

表 1 分散オブジェクト技術の性質と交換形式
Table 1 Characteristics of distributed object environments.

言語独立性	特定のプログラミング言語に依存せずに ORB ミドルウェアを利用できる性質
位置透過性	分散アプリケーションの開発者が，サーバオブジェクトの位置を意識せずに利用できる性質．スタブ，スケルトンが，サーバオブジェクトの位置決めやメソッド呼び出しの中継を行うことで位置透過性が実現される
整列形式	CPU などに依存しないデータの表現形式
通信規約	メッセージを送受信するための通信規約

Java RMI

Java RMI（Remote Method Invocation）は Sun Microsystems によって提案された位置透過性を満たした分散オブジェクト技術である[2)]．Java 言語での利用を前提としており，言語独立性は考慮されていない．サーバオブジェクトは Java 言語で記述され，JavaRMI コンパイラによって Java 用のスタブ，スケルトンのコードが生成される．

データの整列化には Java 言語仕様として定義されたオブジェクト直列化形式（Serialization）が用いられる．通信には JRMP と呼ばれる TCP/IP を利用した Java RMI 用の通信規約，または HTTP が用いられる．

XML-RPC, SOAP

XML-RPC と SOAP は言語独立性を考慮して，整列形式に XML，通信規約に HTTP を用いた遠隔手続き呼び出しの規約である[3),4)]．XML-RPC，SOAP は整列形式に異なる語彙（タグ）を用いている．また，スタブ，スケルトンを生成するための処理系については言及されておらず，位置透過性は考慮されていない．

CORBA や Java RMI が通信規約と整列形式，ORB の実行時環境，スタブ，スケルトンまでを定義しているのに対し，XML-RPC と SOAP は通信規約と整列形式を規定しているだけであり，厳密には ORB ミドルウェアとはいえない．

2.2 階層モデルにおける問題点

Java RMI は，スタブ/スケルトン層，リモート参照層，トランスポート層の 3 つの層からなる階層モデルとして実現されている．スタブ/スケルトン層は分散アプリケーションと ORB の間でデータの受渡しを行う．リモート参照層はサーバオブジェクトの管理を，トランスポート層はデータ通信処理を行う．

このモデルではデータの整列化，非整列化はスタブ/スケルトン層で行われる．しかし，整列化と非整列化は CORBA，Java RMI，XML-RPC などの各技術に

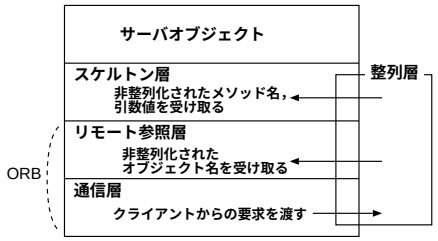


図 2 整列層と他の層との関係
Fig. 2 Relationship between the marshal layer and others.

依存した処理なので，データの受渡し処理とは分離して新たな階層（整列層）にすべきである．XML-RPC と SOAP は通信層と整列層だけを定義した規約なので，この分離は妥当である．

階層モデルにおける整列層の位置を決定するために，ここではサーバ側における整列層と他の層との関係を考える．通信層では受信したクライアントからの要求を整列層に渡す．リモート参照層では整列層から非整列化されたサーバオブジェクトの名前を受け取り，要求されているサーバオブジェクトを検索する．スケルトン層では整列層から非整列化されたメソッド名や引数の値を受け取る．したがって，整列層は通信層，リモート参照層，スケルトン層を横断して関係を持ち（**図 2**），階層構造として表現できない．整列層をサーバオブジェクト名の整列層とメソッド名，引数値の整列層などに分割することも考えられるが，既存の技術ではこれらは包括して扱われており，整列化アルゴリズムなどの共通する部分も多く，整列層の分割は困難である．

3. ORB ミドルウェアのアスペクト指向ソフトウェアアーキテクチャ

本研究では ORB ミドルウェアにおいて複数の層を横断する整列化/非整列化処理をアスペクト指向の概念を用いて整理する．

アスペクト指向では，ある特性や機能により規定されたコンサーンによって定義されたソフトウェアの一部分をアスペクトとする．アスペクトは合流点（Join Point）で他のアスペクトと結合する．ここでは，アスペクト指向ソフトウェアアーキテクチャを，ソフトウェアを複数のアスペクトで表現し，合流点におけるアスペクト間の関係を示したものとして定義する．

一般に ORB ミドルウェアの階層モデルではクライアントとサーバを同じ階層構造，すなわち層の数が同じで対応づけが可能な構造としてとらえていた[2]．しかし，クライアントとサーバではコンサーンが異なるので，それぞれの構造を独立して考える．

表 2 クライアントのアスペクトとコンサーン
Table 2 Aspects and concerns of client.

コンサーン	説明	アスペクト
トランスポートプロトコル	OSI 参照モデルの下位 4 層に従ってデータを送受信する．たとえば，TCP/IP など．	通信
アプリケーションプロトコル	OSI 参照モデルの上位 3 層に従ってデータを送受信する．たとえば HTTP など．	
整列化	データを整列化する．	整列
非整列化	整列化されたデータを計算機の内部表現に復元する．	
サーバ情報	サーバオブジェクトの位置情報を管理する．	リモート参照
インタフェース提供	クライアント側の ORB 利用者に対して，サーバオブジェクトと同一のインタフェースを提供する．	スタブ
メソッド中継	ローカルのメソッド呼び出しを遠隔メソッド呼び出しに変換する．	

3.1 クライアント

位置透過性を実現するためのスタブやサーバオブジェクトの位置情報管理，およびメッセージ交換形式の基準からクライアントのコンサーンを**表 2** のように認識し，従来の階層モデルにおける各層をアスペクトとしてとらえた☆．

クライアント側ではスタブがサーバオブジェクトと同じメソッドをアプリケーションに提供し，スタブのメソッドを呼び出すと要求がサーバに中継される．メソッドの中継処理の実現を考えると，サーバオブジェクト名の整列化，引数の整列化，サーバオブジェクトへのデータ通信，戻り値の非整列化の処理がスタブの各メソッドに散在するので，これらをアスペクトとして整理する．

図 3 にアスペクト間の合流点を表現したクライアントにおけるアスペクト間の関係を示す．図の四角はアスペクト，アスペクト内のテキストはそのアスペクトを規定するコンサーンを表す．アスペクトが重なっている部分が合流点であり，矢印はアスペクト間のメッセージ通信を表す．4.1 節で詳細を述べるが，これらの合流点はすべてスタブコードに含めることができるので，スタブ生成系によりアスペクトを結合させることができる．**表 3** には代表的な合流点と，合流点における処理☆☆を示す．

☆ 以降ではトランスポートプロトコルコンサーンはトランスポートコンサーン，アプリケーションプロトコルコンサーンはプロトコルコンサーンと略記することがある．

☆☆ AspectJ[7] の用語ではアドバイス（advice）に相当する．

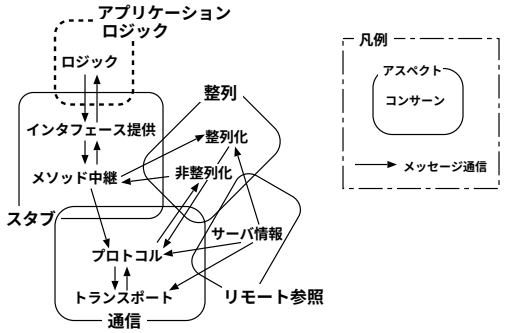


図 3 クライアントのアスペクト間の関係
Fig. 3 Relationships between aspects of client.

表 3 クライアントの合流点における処理
Table 3 Join points and advices of client.

合流点	合流点における処理
ロジック → インタフェース提供	サーバオブジェクトのメソッドを呼び出す.
メソッド中継 → 整列化	メソッド名，引数を渡して整列化する.
サーバ情報 → 整列化	サーバオブジェクト名を渡して整列化する.
整列化 → プロトコル	整列化されたデータを渡す.
サーバ情報 → トランスポート	通信先の情報を渡す.
メソッド中継 → プロトコル	遠隔メソッド呼び出しの要求をサーバに送信する.

表 4 サーバのアスペクトとコンサーン
Table 4 Aspects and concerns of server.

コンサーン	説明	アスペクト
待受け通信	クライアントからの接続を待つ.	通信
トランスポートプロトコル	OSI 参照モデルの下位 4 層に従ってデータを送受信する.	
アプリケーションプロトコル	OSI 参照モデルの上位 3 層に従ってデータを送受信する.	
整列化	データを整列化する.	整列
非整列化	データを非整列化する.	
レジストリ	サーバオブジェクト名とスケルトンの対応を管理する.	リモート参照
メソッド振分け	クライアント側の要求をサーバオブジェクトに振り分けてメソッドを実行する.	スケルトン

3.2 サーバ

クライアントと同様にしてサーバのソフトウェアアーキテクチャを構築した．**表 4** にアスペクトとコンサーンを，**図 4** にアスペクト間の関係を，**表 5** に代表的な合流点と合流点における処理を示す．これらの合流点はすべてスケルトンコードに含めることができるので，スケルトン生成系によりアスペクトを結合させることができる．

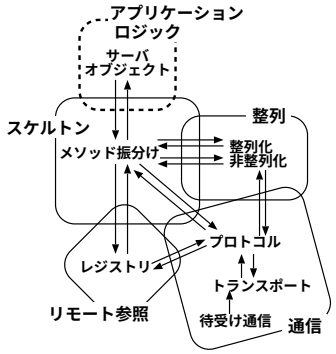


図 4 サーバのアスペクト間の関係
Fig. 4 Relationships between aspects of server.

表 5 サーバの合流点における処理
Table 5 Join points and advices of server.

合流点	合流点における処理
プロトコル ↔ 非整列化	クライアントの要求を渡して非整列化されたサーバオブジェクト名を受け取る.
プロトコル ↔ レジストリ	サーバオブジェクト名を渡して対応するスケルトンを受け取る.
プロトコル → メソッド振分け	クライアントの要求をスケルトンに渡す.
メソッド振分け ↔ 非整列化	メソッド名，引数のデータを渡して非整列化されたデータを受け取る.
メソッド振分け ↔ サーバオブジェクト	クライアントから要求されたサーバオブジェクトのメソッドを呼び出して，戻り値を受け取る.

サーバ側ではスケルトンがクライアントからの要求をサーバオブジェクトのメソッド呼び出しに振り分ける．メソッドの振分け処理の実現を考えると，受信した要求の非整列化，戻り値の整列化の処理がスケルトンのコードに散在するので，これらをアスペクトとして整理する．

4. アーキテクチャに基づいた ORB ミドルウェアの設計と実現

提案したアスペクト指向ソフトウェアアーキテクチャに従って，ORB ミドルウェアを設計・実現する．

スタブアスペクトとスケルトンアスペクトの実現には生成系を用いる．スタブはサーバオブジェクトと同じインタフェースを持ち，メソッド呼び出しをサーバに中継する．スタブは分散アプリケーションに依存したメソッドを持つので，コンポーネントやフレームワークとしてインタフェースを標準化することはできない．一方，スタブのメソッドは決まった手順で整列化や通信などの処理を行うので，処理内容を定型化することは可能である．以上のことからサーバオブジェクトのインタフェースからスタブを自動生成する方法

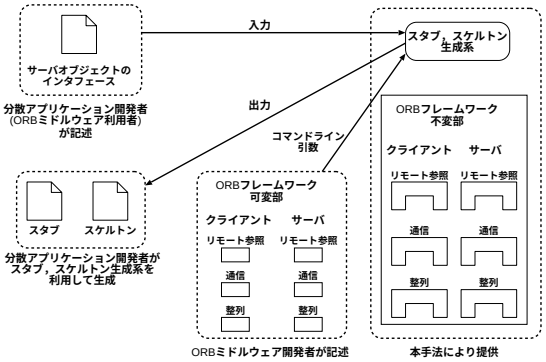


図 5 分散アプリケーション開発の概要
Fig. 5 Development for distributed application.

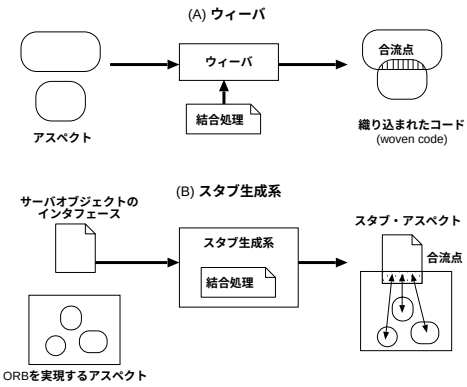


図 6 ウィーバとスタブ生成系
Fig. 6 Weaver and stub generator.

を選択した．スケルトンも同様である．

ORB を実現するリモート参照，通信，整列アスペクトはフレームワークとして実現する．各アスペクトは独立したフレームワークとして実現し☆，スタブ，スケルトン生成系がアスペクト間の関係を生成する．フレームワークにより，インタフェースを標準化し，分散オブジェクト技術に依存した開発箇所を明確にして局所化できる．ORB をコンポーネントとして実現する方法も考えられるが，ORB では整列化などの技術固有の箇所が多く，コンポーネントの組合せはほとんど起こらないので，適切ではないと考えられる．

本手法による分散アプリケーション開発の概要を**図 5** に示す．我々はスタブ，スケルトン生成系と ORB フレームワークの不変部を提供する．開発者☆☆は分散オブジェクト技術に応じて ORB フレームワークの可変部を記述する．スタブ，スケルトン生成系は，ORB フレームワークの可変部として記述されたクラスの名前をコマンドライン引数として受け取り，分散オブジェクト技術に対するスタブ，スケルトンのコードを生成する．

4.1 ウィーバとスタブ，スケルトン生成系

アスペクト指向言語では，アスペクトコードとそれらの結合のためのコードは独立に記述され，ウィーバ（weaver）と呼ばれる処理系がアスペクトが結合されたコードを生成する[7]（**図 6** (A)）．アスペクトは既存のプログラミング言語で，結合処理は専用のプログラミング言語で記述されることが多い．ORB ミドルウェアでは結合処理として，アスペクトの順序を指定して起動することや，起動したアスペクト間でのデータの受渡しを記述する必要がある．たとえば，スタブでは整列（整列化），通信，整列（非整列化）のような順序でアスペクトを起動してメソッド呼び出しの中継を行う．また，各アスペクトはメソッドの実引数や戻り値によりデータの受渡しを行う．しかし，一般のアスペクト指向言語およびそのウィーバの処理ではこれらは考慮されていない．

本研究では，位置透過性を満足した ORB ミドルウェアにはスタブ，スケルトンの生成系が存在することに注目し，スタブ，スケルトン生成系をウィーバと見なしてアスペクトを結合する方法を提案する．スタブ，スケルトンはサーバオブジェクトのインタフェースから生成され，ORB を利用してメソッドの中継やメソッドの振分けを行う処理が記述される．これは，スタブアスペクトとスケルトンアスペクトに通信アスペクトや整列アスペクトが結合されたと考えられる．応用領域を ORB ミドルウェアに限定することで，汎用のウィーバを用いた場合では考慮されていなかったアスペクトの起動順序の指定やアスペクト間のデータの受渡し処理を定型化して記述し，自動生成できるようになる（4.3.2 項参照）．

図 6 (B) にスタブ生成系の処理の概略を示す．ORB ミドルウェアではアスペクトの合流点はスタブ，スケルトンのメソッドに限定でき，結合されるコードは通信処理や整列処理などのメソッド呼び出しに限定される．したがって，汎用のウィーバでは開発者が手作業で記述していた結合処理（アドバイス）を，定型化することが可能である．スタブ，スケルトン生成系は定型化されたアスペクトの結合処理を内包したウィーバ

☆ クライアントとサーバそれぞれでリモート参照，通信，整列アスペクトが独立したフレームワークとして実現されるので，合計 6 個のフレームワークが存在する．本論文ではこれらのフレームワークを総称して ORB フレームワークと呼ぶ．

☆☆ 開発者は，ORB ミドルウェアの開発者と分散アプリケーションの開発者（ORB ミドルウェアの利用者）の 2 種類に分けられるが，本論文における開発者は ORB ミドルウェア開発者を指す．

```
1: public interface Sample {
2:   int compare(int a, int b);
3:   int[] sort(int ar[]);
4: }
```

図 7 サーバオブジェクトのインタフェース
Fig. 7 Interface of server object.

```
 1: public class SampleStub implements Sample {
 2:   private ServerInfo sinfo;
 3:   public SampleStub(ServerInfo sinfo) {
 4:     this.sinfo = info;
 5:   }
 6:   public int compare(int param_0, int param_1) {
 7:     /*** メソッド名，引数の整列化処理 ***/
 8:     ClientMarshalStream ms
 9:       = new XmlrpcClientMarshalStream();
10:     ms.open();
11:     // 整列とリモート参照の合流点
12:     ms.writeObjectName(sinfo.getObjectName());
13:     // 整列とスタブの合流点
14:     ms.writeMethodName("compare");
15:     ms.writeInt(param_0);
16:     ms.writeInt(param_1);
17:     ms.close();
18:     /*** 通信処理 ***/
19:     // スタブと通信の合流点
20:     Connection con = new HttpConnection();
21:     // 通信とリモート参照の合流点
22:     con.connect(sinfo);
23:     // 整列と通信の合流点
24:     byte[] req = ms.getBytes();
25:     con.sendRequest(req);
26:     byte res[] = con.recieveResponce();
27:     con.disconnect();
28:     /*** 戻り値の非整列化処理 ***/
29:     ClientUnmarshalStream ums
30:       = new XmlrpcClientUnmarshalStream(res);
31:     ums.open();
32:     // 整列とスタブの合流点
33:     int result = ums.readInt();
34:     ums.close();
35:     return result;
36:   }
37:   public int[] sort(int[] param_0) {
38:     /*** メソッド名，引数の整列化処理 ***/
39:     ...
40:     /*** 通信処理 ***/
41:     ...
42:     /*** 戻り値の非整列化処理 ***/
43:     ...
44:     reeturn result;
45:   }
46: }
```

図 8 スタブのコード
Fig. 8 Stub code.

であると考えることができる．

4.2 スタブ，スケルトン生成系の入出力

スタブ，スケルトン生成系の入力であるサーバオブジェクトのインタフェースの例を**図 7** に，生成するスタブコードの例を**図 8** に，スケルトンコードの例を**図 9** に示す．例には Java 言語を用いた．

スタブの各メソッドは整列化，通信，非整列化の順序で中継処理を行う．スタブのメソッドごとに中継処理が行われるので，整列化，通信，非整列化の処理が各メソッドに散在することになる．同様にスケルトンでは dispatch のメソッド名を判定する各 if 文の本体に，非整列化と整列化の処理が散在している．

```
 1: public class SampleSkeleton implements Skeleton {
 2:   private Sample serverObj;
 3:   public SampleSkeleton(Sample obj) {
 4:     serverObj = obj;
 5:   }
 6:
 7:   public byte[]
 8:   dispatch(ServerUnmarshalStream unmar) {
 9:     byte[] res = null;
10:     unmar.open();
11:     // 非整列化とスケルトンの合流点
12:     String methodName = unmar.readMethodName();
13:     if (methodName.equals("compare")) {
14:       /*** 引数の非整列化処理 ***/
15:       // 非整列化とスケルトンの合流点
16:       int p_0 = unmar.readInt();
17:       int p_1 = unmar.readInt();
18:
19:       /*** メソッド呼出し ***/
20:       // スケルトンとアプリケーションロジックの合流点
21:       int ret = serverObj.compare(p_0, p_1);
22:
23:       /*** 戻り値の整列化処理 ***/
24:       ServerMarshalStream mar
25:           = new XmlrpcServerMarshalStream();
26:       mar.open();
27:       // 整列化とスケルトンの合流点
28:       mar.writeInt(ret);
29:       mar.close();
30:       res = mar.getBytes();
31:     }
32:     else if (methodName.equals("sort")) {
33:         /*** 引数の非整列化処理 ***/
34:       int[] p_0 = unmar.readIntArray();
35:       /*** メソッド呼出し ***/
36:       int[] ret = serverObj.sort(p_0);
37:       /*** 戻り値の整列化処理 ***/
38:       ...
39:     }
40:     unmar.close();
41:     return res;
42:   }
43: }
```

図 9 スケルトンのコード
Fig. 9 Skeleton code.

我々のアーキテクチャではこれらの処理はアスペクトとして整理され，スタブ，スケルトン生成系がアスペクトが結合されたコードを生成する．たとえば，クライアントではスタブアスペクトと整列アスペクト（図 8 の 14〜17 行），スタブアスペクトと通信アスペクト（図 8 の 20 行），通信アスペクトと整列アスペクト（図 8 の 24〜25 行）などが結合されたコードがスタブとして生成される．

アーキテクチャに基づいて忠実にアスペクトの結合を実現するならば，通信アスペクトと整列アスペクトの結合は通信アスペクトを実現しているフレームワー

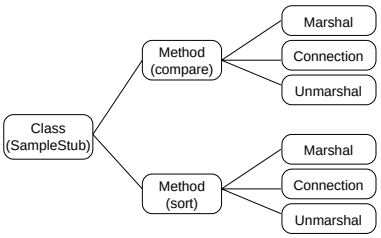


図 10 スタブの抽象構文木
Fig. 10 Abstract syntax tree of stub code.

ク（4.4 節参照）のソースコードを変更することで行う．しかし，この方法ではフレームワークのソースコードの再コンパイルが必要となり，フレームワークの独立性が低下する．我々はアスペクトの合流点をスタブ，スケルトンのコード内に設定して，結合されたコードを生成する方法を選択した．これにより，ORB を実現するフレームワークを変更する必要がなく，位置透過性を満たすために生成されるスタブ，スケルトンにアスペクトの結合を局所化させることができる．

4.3 スタブ，スケルトン生成系の設計と実現

スタブ，スケルトン生成系の再利用可能かつ言語独立な実現のために，以下の指針に従って設計する．

（A） 特定のプログラミング言語に依存せずに，スタブ，スケルトン生成系を設計・実現するためにデザインパターン[8] を用いる．

（B） コード生成処理を，コードの記述言語に依存する部分と独立した部分に分離する．

（C） 結合処理記述の再利用性を高めるために，各合流点における結合処理を個別に記述する．

4.3.1 スタブコード，スケルトンコードの構造

スタブ，スケルトン生成系は特定のプログラミング言語のためのコードを生成する．指針（B）に従って言語独立部分を考えた場合，スタブコードおよびスケルトンコードの構造が独立部分としてあげられる．図 8 で示したスタブコードは**図 10** の抽象構文木で，図 9 で示したスケルトンコードは**図 11** の抽象構文木で表現できる．スタブコード，スケルトンコードにはさらに文や式といった構文要素が出現するが，文法を細粒度で表現すると言語への依存度が高くなり，木も大きくなる．我々は木の節点として，標準的なオブジェクト指向言語が持つ言語要素であるクラスやメソッド，および，通信，整列アスペクトの合流点を選んだ．スタブ，スケルトン生成系はサーバオブジェクトのインタフェースから図 10，図 11 で示した抽象構文木を作成し，抽象構文木を深さ優先探索して節点に対応するコードを生成するように設計した．

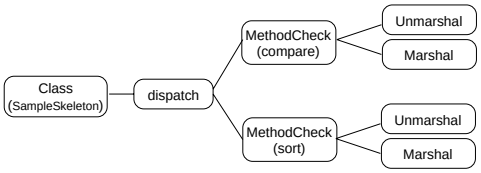


図 11 スケルトンの抽象構文木
Fig. 11 Abstract syntax tree of skeleton code.

4.3.2 コード生成部の設計

木の探索処理には，抽象構文木を解釈実行するためのパターンである Interpreter パターンを用いる．

木探索時に各節点のコード生成処理を記述するには以下の 3 つの方法が考えられる．

(1) Visitor パターン
(2) Command パターン
(3) サブクラスと Template Method パターン

Visitor パターンは節点に対応した処理を Visitor クラスのメソッドとして記述する．(1) は Interpreter パターンで木を探索し，節点訪問時に Visitor クラスのメソッドを呼び出してコードを生成する．しかし，1 つの Visitor クラスにすべての合流点の結合処理が記述されるので指針（C）に反する．たとえば，HTTP で通信を行う XML-RPC と SOAP では通信アスペクトの結合処理記述を再利用したいが，Visitor パターンでは XML-RPC 用の XmlrpcVisitor クラス，SOAP 用の SoapVisitor クラスに個別に記述しなければならない．

Command パターンは要求，すなわちコード生成処理を 1 つのクラスにカプセル化することができる．(2) では抽象構文木の節点の種類に対応したコード生成処理が記述された Command クラスを定義する．コード生成は，Interpreter パターンで木を探索し，節点の種類に応じた Command オブジェクトを生成し，呼び出すことで行われる．Command オブジェクトの生成は Abstract Factory パターンを用いることでカプセル化できる．

(3) では木の節点クラスに木を探索する処理と抽象メソッドとして定義されたコード生成処理を記述する．実際のコード生成処理はサブクラスでメソッドを再定義して記述する（Template Method パターン）．サブクラスのオブジェクトの生成は Abstract Factory パターンを用いることでカプセル化できる．

(2) はコード生成に委譲を用いる方法，(3) は継承を用いる方法といえる．スタブ，スケルトン生成系はどちらの方法でも同じように実現できるが，継承を用いた場合ではコンパイラによる多相型の型検査や実行

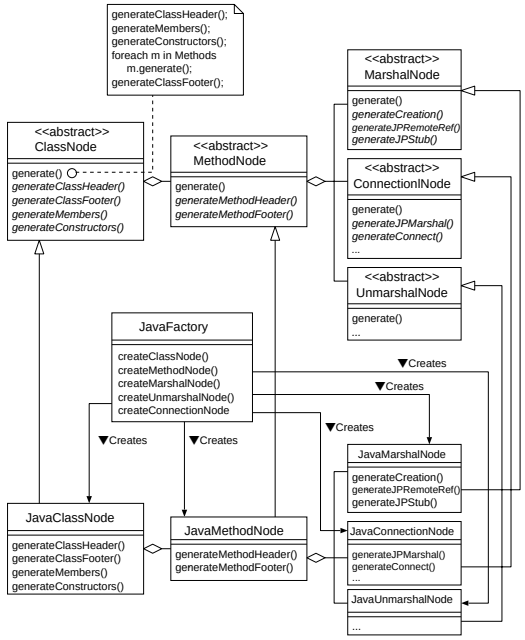


図 12 スタブ生成系のクラス図
Fig. 12 Class diagram for stub generator.

コードの最適化が期待できるので，(3) の方法を採用する．

設計したスタブ生成系のクラス図を**図 12** に示す．ClassNode，MethodNode などが木を探索する処理（generate）とコード生成のための抽象メソッドが定義されたクラスである．JavaClassNode，JavaMethodNode などが Java 言語用のコード生成処理が記述されたクラスである．合流点における結合処理は JavaMarshalClassNode などのメソッド（generateJPRemoteRef）として記述する．構文木は Abstract Factory パターンに従った JavaFactory クラスが節点のオブジェクトを生成して構築される．スケルトン生成系も同様にして設計した．

表 6 に，図 8 で示したスタブコードを生成するときの，スタブ生成系の呼び出されるメソッドと生成されるコード（行番号）の対応を示す．メソッドは表の順序で呼び出される．

汎用のウィーバでは困難であったアスペクトの起動順序の指定やアスペクト間のデータの受渡し処理は次のように解決した．アスペクトの起動順序は，抽象構文木では Marshal，Connection，Unmarshal ノードの出現順序として表現し，コード生成処理では Template Method パターンによりコード生成を行うメソッドの呼び出し順序として記述した．データの受渡し処理はあらかじめ決められた名前の変数，たとえば図 8 の 24 行目の req や 26 行目の res を利用してメソッドの戻り値を受け取り，別のメソッドの実引数として渡すコードを生成することで実現した．

表 6 スタブ生成系のメソッド呼び出し順と生成されるコード
Table 6 Order of method invocation of stub generator and generated code.

節点クラス	呼び出されるメソッド	コード
Class (SampleStub)	generateClassHeader()	1
	generateMembers()	2
	generateConstructors()	3〜5
Method (compare)	generateMethodHeader()	6
Marshal	generateCreation()	8〜10
	generateJPRemoteRef()	12
	generateJPStub()	14〜17
Connection	generateJPStub()	20
	generateJPRemoteRef()	22
	generateJPMarshal()	24
	generateConnect()	25〜27
Unmarshal	generateCreation()	29〜31
	generateJPStub()	33〜34
Method (compare)	generateMethodFooter()	35〜36
Method (sort)	generateMethodHeader()	37
Marshal Connection Unmarshal	compare の場合と同様であるので省略	38〜43
Method (sort)	generateMethodFooter()	44〜45
Class (SampleStub)	generateClassFooter()	46

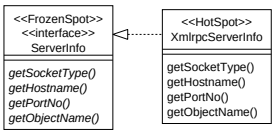


図 13 リモート参照アスペクトのフレームワーク
Fig. 13 Framework for remote reference aspect.

4.4 ORB フレームワークの設計と実現

ORB を実現するリモート参照，通信，整列アスペクトは技術によって処理内容が大きく異なる場合が多いので，ホワイトボックスフレームワークとして設計・実現する．

クライアントのリモート参照アスペクトと通信アスペクトを実現するフレームワークのクラス図を，それぞれ**図 13**，**図 14** に示す☆．

リモート参照アスペクトはクラスの継承関係を利用してフレームワークの不変部と可変部を定義した．

☆ 紙面の都合上，図 14 では接続と送信に関するメソッドだけを記述している．また，整列アスペクトとサーバフレームワークのクラス図は省略するが，同様にして設計できる．

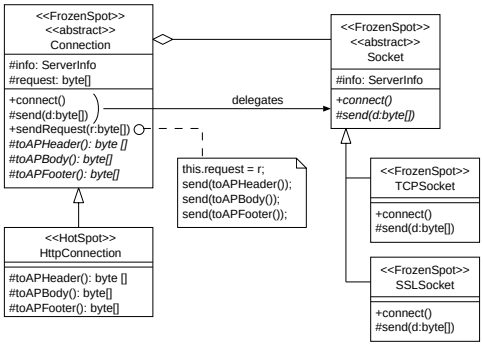


図 14 通信アスペクトのフレームワーク
Fig. 14 Framework for connection aspect.

通信アスペクトは TCP や SSL などの OSI 参照モデルの下位 4 層に関するトランスポートプロトコルコンサーンと HTTP や SMTP などの上位 3 層に相当するアプリケーションプロトコルコンサーンによって規定される．トランスポートプロトコルとアプリケーションプロトコルを独立に選択できるようにフレームワークを設計することで，再利用性を向上させることができる．我々は，継承と委譲を用いて通信アスペクトのフレームワークを設計した．多重継承による設計も考えられるが，Java などの単一継承の言語には適用できない．Connection クラスでそれぞれのプロトコルのメソッドを定義し，トランスポートプロトコルの処理は Socket クラスに委譲し，アプリケーションプロトコルの処理は Connection クラスのサブクラスで再定義する．Socket クラスは抽象クラスで，そのサブクラスでトランスポートプロトコルでの通信処理が記述される．

フレームワークを分散オブジェクト技術から独立して利用するには，利用者には Connection クラスなどの不変部のクラスだけを提供し，HttpConnection などの可変部のクラスを隠蔽しなければならない．可変部のオブジェクトの生成には Factory Method パターンや Abstract Factory パターンを用いることが多い．しかし，これらのパターンを用いると独立性が低下する．

我々が提案する ORB ミドルウェアの実現法ではスタブ，スケルトン生成系が存在するので，可変部オブジェクトの生成処理は生成系がスタブ，スケルトンに記述することにした（図 8 の 8，20，29 行，図 9 の 24 行）．スタブ，スケルトン生成系はこれらの可変部のクラス名をコマンドライン引数として受け取り，コードを生成する．これにより，フレームワークのクラスによる機能の実現とオブジェクトの生成処理を分離することができる．また，アスペクト間の関係，すなわちフレームワーク間のデータの受渡し処理もスタブ，スケルトン生成系が生成することで（図 8 の 22，25，30 行），フレームワークの独立性を高めることができる．

表 7 ORB フレームワークの行数
Table 7 Lines of ORB Framework.

	クラス数	行数
フレームワーク 合計	39	2180
不変部 合計	26	790
整列	7	139
通信	8	463
可変部 合計	13	1390
整列	9	1170
通信	2	150

5. 議　　論

5.1 ORB フレームワークの実現

Java 言語を用いて ORB フレームワークを実現し，XML-RPC 用の ORB を実現して，以下の観点から ORB ミドルウェア開発が省力化されることを確認した．

- 整列化などの技術に強く依存した処理記述は再利用できないが，その他の処理は再利用可能である．

  ORB フレームワークの不変部は約 790 行，可変部は約 1,390 行である（**表 7**）．可変部の通信に関する部分が約 150 行，整列化・非整列化に関する部分が約 1,170 行である．アプリケーションプロトコルに HTTP を用いた通信部は可変部としてコード量を計算しているが，SOAP などの他の技術でも再利用可能な部分である．XML-RPC では整列形式に XML を用いるが，XML パーサを利用した処理や int や int の配列などの型に応じた整列処理記述が必要なのでコード量が多くなっている．
- 開発箇所が明確化される．

  可変部の実現では，データの受渡し処理やオブジェクトの生成処理を考える必要がなく，注目しているアスペクトの機能の実現箇所が明確であった．すなわち，4.4 節で示したように各アスペクトにおいて実現すべき箇所がクラスやメソッドとして明示されていた．

5.2 スタブ，スケルトン生成系の実現

Java 言語を用いて Java 言語用のスタブ，スケルトンを生成する生成系を実現した．サーバオブジェクトの構文解析には Java の Reflection API を利用した．生成系の規模は約 1280 行である（**表 8**）．生成系により，図 7 のサーバオブジェクトのインタフェースから，60 行のスタブコード，56 行のスケルトンコードが生

表 8 スタブ，スケルトン生成系の行数
Table 8 Lines of stub/skeleton generator.

	クラス数	行数
生成系	29	1280

成される．

技術独立性

スタブ，スケルトン生成系が生成するコードで ORB に依存した部分は，フレームワークの可変部オブジェクトを生成する処理である（図 8 の 8，20，29 行，図 9 の 24 行）．実現した生成系では生成するオブジェクトのクラス名をコマンドライン引数として与えることにより，特定の分散オブジェクト技術に依存しないようにした．

言語独立性

たとえば C++言語のスタブ，スケルトンを生成系により生成するには，図 12 の ClassNode，MethodNode などのサブクラスとして C++言語のコードを生成するクラス CppClassNode，CppMethodNode などを定義し，それらのインスタンスを生成する CppFactory クラスを定義すればよい．

我々は特定のプログラミング言語に依存しないようにデザインパターンを組み合わせて，スタブ，スケルトン生成系を設計した．したがって，生成系自体を C++などで実現することも容易である．なお，この場合は抽象構文木の作成には，サーバオブジェクトのインタフェースを構文解析する必要がある．

5.3 アスペクト指向に関する議論

本研究では ORB ミドルウェアの構造をアスペクト指向を用いて整理し，スタブ，スケルトン生成系を応用領域が ORB ミドルウェアに限定されたウィーバと考えてアスペクトの結合処理を行った．合流点と結合処理を定型化し，自動生成することができた．本手法は汎用のウィーバを用いたアスペクトの結合方法と比較し，以下の特徴がある．

特徴 1 アスペクトの合流点がスタブとスケルトンのコード内にまとめられる．

特徴 2 アスペクトの結合処理が定型化され，生成系の内部コードとして記述される．

特徴 3 アスペクトの起動順序の指定やアスペクト間のデータの受渡しができる．

汎用のウィーバを用いる方法では合流点や結合処理を自由に記述できるが，開発者がそれらを手作業で記述しなければならない．我々の方法は，合流点や結合処理を限定することで，それらをスタブ，スケルトンの内部コードとしてあらかじめ記述しておくことができ，開発者が記述する必要はない．

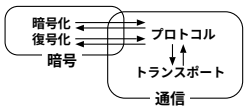


図 15 暗号処理におけるアスペクトの関係
Fig. 15 Relationships between cryptography and connection aspects.

```
23: // 整列と通信の合流点
24: byte[] req = ms.getBytes();
追：// 通信と暗号の合流点
追：Crypto enc = new Crypto(Crypto.ENCRYPT_MODE);
追：req = enc.encode(req);
25: con.sendRequest(req);
26: byte res[] = con.recieveResponce();
27: con.disconnect();
追：// 通信と暗号の合流点
追：Crypto dec = new Crypto(Crypto.DECRYPT_MODE);
追：res = enc.decode(res);
```

「追：」は追加された行を表す

図 16 暗号化と復号化を行うスタブのコード
Fig. 16 Stub code with encryption and decryption.

ここでは，ORB ミドルウェアに通信データを暗号化する機能を追加することを題材にして，その設計・実現方法を示し，合流点の追加，アスペクトと結合処理記述，アスペクトの干渉について議論する．

5.3.1 暗号処理の設計と実現

暗号処理とソフトウェアアーキテクチャ

暗号処理は SSL などのトランスポートプロトコルでの暗号化とアプリケーションプロトコルで通信されるデータの暗号化に分類される．トランスポートプロトコルでの暗号化は 4.4 節で示したように，Socket クラスのサブクラスである SSLSocket で SSL に従って通信をする処理を記述すればよい．

アプリケーションプロトコルでの暗号化は，ソフトウェアアーキテクチャを拡張することで行う．暗号処理（暗号化と復号化）を機能コンサーンとして認識し，それらのコンサーンによって規定される暗号処理を 1 つのアスペクトとして考える．ORB ではアプリケーションプロトコルにおけるデータの送信前に暗号化，データ受信後に復号化する必要があるので，暗号アスペクトと通信アスペクトは関係を持つ（**図 15**）．

合流点と合流点における処理

クライアントではスタブの通信処理の前後を合流点として，暗号化，復号化の処理を追加すればよい．図 8 のスタブコードに暗号処理を追加したコードを**図 16** に示す．

サーバではスケルトンの整列化処理の後を合流点とし，暗号化処理を追加した．復号化処理の合流点は通

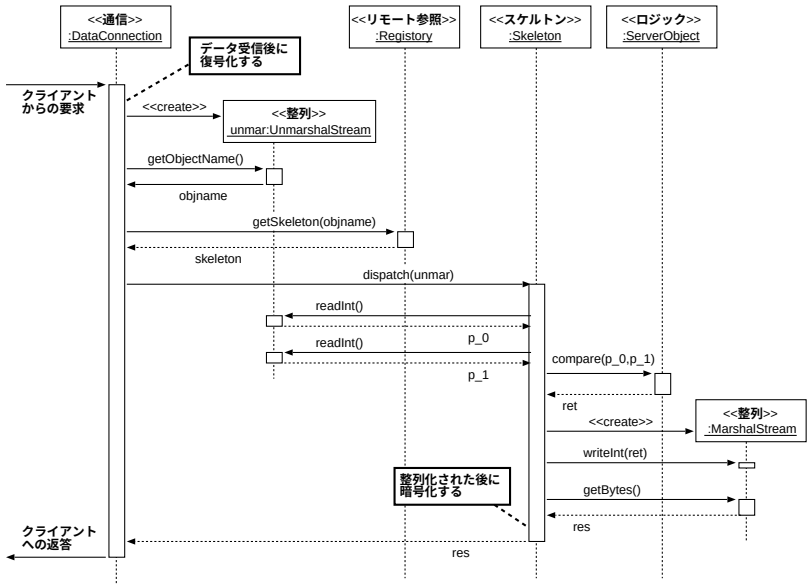


図 17 サーバのシーケンス図
Fig. 17 Sequence diagram of server.

信アスペクトのクライアントからのメッセージ受信の後に設定した．これは，メッセージ受信後にデータを復号化して，スケルトンを検索する必要があるからである（**図 17**）．

暗号処理の実現

暗号処理の実現は ORB フレームワークとスタブ，スケルトン生成系を変更することで行われる．スタブ，スケルトン生成系の入力（サーバオブジェクトのインタフェース）は変更する必要はない．

ORB フレームワークの変更

暗号アスペクトは独立したフレームワークとして設計・実現できる．暗号フレームワークは Java2 SDK1.4 の Java 暗号化拡張機能（JCE）を利用して作成し，80 行程度である（**表 9**）．

クライアント側では暗号アスペクトと通信アスペクトのデータの受渡し処理はスタブコード内にスタブ生成系が生成するので，通信フレームワークを変更する必要がない．サーバ側では復号化に関する処理を通信フレームワークに追加する必要があるので，サブクラスを用いて受信後に復号化処理を行うメソッドを再定義した（**図 18**）．追加に必要なクラスは 2 個で，行数は 30 行であった（表 9）．

スタブ，スケルトン生成系の変更

スタブ生成系では図 12 の JavaConnectionNode のサブクラス JavaCryptoConnectionNode で generateConnect メソッドなどを再定義して，通信アスペクトと暗号アスペクト間のデータの受渡し処理（図 16 の「追：」の行）を生成する（**図 19**）．スケルトン生成系も同様にして暗号化を行うコードを生成する．

暗号化にともないスタブ，スケルトン生成系に 6 個のクラス（110 行）を追加した（表 9）．

表 9 暗号処理にともなう追加行数
Table 9 Added lines for cryptography.

	クラス数	行数
暗号フレームワーク	2	80
通信フレームワークへの追加	2	30
スタブ，スケルトン生成系への追加	6	110

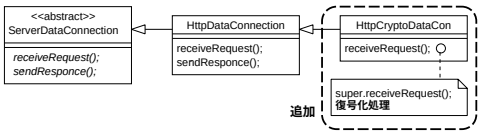


図 18 復号処理を追加したサーバ側の通信フレームワーク
Fig. 18 Class diagram of server-side connection framework with decryption.

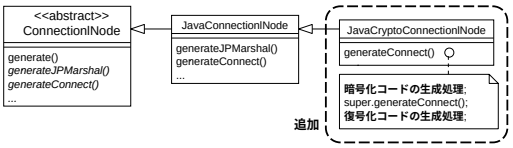


図 19 暗号処理を追加したスタブ生成系のクラス図
Fig. 19 Class diagram of stub generator with encryption.

5.3.2 合流点に関する議論

暗号処理の追加では，合流点をスタブ，スケルトン

のコード内だけに設定することはできなかった．サーバの通信アスペクトと暗号アスペクトの合流点の設定と結合処理の記述は，通信フレームワークにサブクラスを追加し，メソッドを再定義することで行った（図 18）．変更は 30 行程度のクラスを追加するだけで済んだ．

通信フレームワークに合流点を設定する場合でも変更箇所が明確化，局所化されていることを確認した．これは，アスペクト指向の概念を用いたことで，煩雑になりがちなデータの受渡し処理やオブジェクトの生成処理をフレームワークとして実現する必要がなくなり，フレームワークの構造を簡潔にできたことによる．

5.3.3 アスペクトと結合処理に関する議論

我々の手法では，ORB フレームワークがアスペクトの記述に相当し，スタブ，スケルトン生成系の内部コードがアスペクトの結合処理記述に相当する．

ORB フレームワークでは分散オブジェクト技術に依存した処理を可変部としてサブクラスを用いて記述する．この場合は，5.2 節で述べたようにクラス名をコマンドライン引数として与えることで，スタブ，スケルトン生成系の内部コードを変更する必要はない．

暗号処理の追加のように，新たにアスペクトを追加する場合には，ORB フレームワークを変更し，それに合わせてスタブ，スケルトン生成系の内部コードを変更する必要がある．暗号処理では，暗号アスペクトを暗号フレームワークとして通信フレームワークなどとは独立して記述し，暗号フレームワークを呼び出すようにスタブ，スケルトン生成系の内部コードを変更した．スタブ，スケルトン生成系の変更においては，図 19 で示したように，再定義すべきメソッドが明確化されていることを確認した．追加したコードは 110 行で（表 9），比較的少ない行数で変更が行えたと考える．

5.3.4 アスペクトの干渉に関する議論

我々の提案したソフトウェアアーキテクチャではスタブ，スケルトンに複数のアスペクトが結合されるので，アスペクトが相互に動作を阻害するアスペクトの干渉が起こる可能性がある[9),10)]．本手法では，アスペクトの起動順序やアスペクト間のデータのやりとりを定型化し，干渉が起こらないようにスタブ，スケルトン生成系を設計・実現した．

暗号アスペクトなど新たにアスペクトを追加する場合は，他のアスペクトと干渉が起こらないように，スタブ，スケルトン生成系のコード生成部を変更する必要がある．1 つの合流点にアスペクトが複数存在する場合には Decorator パターン[8)] などを用いることで，コードの生成処理が容易になると考えられる．

5.3.5 ま と め

本論文で提案した処理方法は，汎用のウィーバと比較すると合流点や結合処理記述の自由度は低いが，3 章で示したソフトウェアアーキテクチャに従う限り，開発者が結合処理を記述する必要がない．また，新たにアスペクトを追加する場合でも，変更箇所が明確に示されており，変更する行数も比較的少なく，追加は容易であった．今回は暗号処理の追加を例にあげたが，実時間処理やトランザクション処理などの追加でも同様である．

6. おわりに

本論文ではアスペクト指向の概念を適用した ORB ミドルウェアの開発方法を提案した．ORB ミドルウェアを複数のアスペクトで表現し，アスペクト指向ソフトウェアアーキテクチャを定義した．ORB を構成している各機能は ORB フレームワークとして独立に実現される．アスペクトの結合はスタブ，スケルトン生成系によって行われ，その言語独立な実現方法を考察した．Java 言語を用いて ORB フレームワークとスタブ，スケルトン生成系，XML-RPC 用の ORB ミドルウェアを実現し，開発が省力化されることおよび機能の追加が容易なことを確認した．

今後の課題としては以下があげられる．

- SOAP などの他の分散オブジェクト技術を実現して，フレームワークにおける再利用率などを計測し，提案したアーキテクチャの有効性を示す．
- アスペクト指向を用いてスタブ，スケルトン生成系を設計・実現する．スタブ，スケルトン生成系のアスペクトを構文解析，木走査，コード生成などとして構造を整理し，各アスペクトを独立して実現することで，ソースコードの読解性の向上や機能追加に対する変更の容易さが期待できる．
- 実時間処理やトランザクション処理などを実現し，拡張性についてさらに考察する．
- ORB ミドルウェア開発で得られた手法を他のソフトウェア開発に応用する．
  我々の手法の特徴は以下のように一般化できる．
  - ソフトウェアを複数のアスペクトで表現し，アスペクト指向ソフトウェアアーキテクチャを定義する．
  - 生成系を用いて開発されるソフトウェアに対して，生成系をウィーバと見なす．
  - アスペクトの合流点がまとめられた構成要素を見つけ出し，生成系によってその構成要素

を生成する．

生成系を用いるソフトウェアとして，JSP（JavaServer Pages）やデータベースの埋込み SQL などがある．これらに対する応用を考察する．

アスペクトの起動順序の指定やアスペクト間のデータの受渡しはアスペクトの干渉に関する問題であり，これらを一般化した解決策を考察する．

参 考 文 献

1) OMG: *The Common Object Request Broker: Architecture and Specification* (1999).
2) Sun Microsystems: *Java Remote Method Invocation Specification*.
3) UserLand Software: XML-RPC Home Page. http://www.xmlrpc.com/
4) W3C Note: *Simple Object Access Protocol* (*SOAP*) 1.1 (2000). http://www.w3.org/TR/SOAP/
5) Chin, R.S. and Chanson, S.T.: Distributed, Object-Based Programming Systems, *ACM Computing Surveys* (*CSUR*), Vol.23 (1991).
6) Tanenbaum, A.S.: *Distributed Operating Systems*, Prentice Hall (1994).
7) AspectJ Team: AspectJ Programming Guide. http://aspectj.org/
8) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns*, Addison-Wesley (1994).
9) Aßmann, U.: AOP with design patterns as meta-programming operators, Technical Report 28, Universität Karlsruhe (1997).
10) 石尾 隆，楠本真二，井上克郎：プログラムスライスを用いたアスペクト指向プログラムのデバッグ支援環境，オブジェクト指向最前線 2003, pp.129–136, 近代化学社 (2003).

(平成 15 年 9 月 22 日受付)
(平成 16 年 4 月 5 日採録)



蜂巣 吉成（正会員）

1971 年生．1994 年名古屋大学工学部情報工学科卒業．1999 年同大学院工学研究科情報工学専攻博士後期課程修了．1999 年南山大学経営学部情報管理学科助手，2000 年より同大学数理情報学部情報通信学科講師．工学博士．構造化文書の記述，ソフトウェアの開発環境に関する研究に興味を持つ．日本ソフトウェア科学会，電子情報通信学会，IEEE Computer Society，ACM 各会員．



野呂 昌満（正会員）

1958 年生．1981 年慶應義塾大学工学部管理工学科卒業．1983 年同大学院工学研究科管理工学専攻前期博士課程修了．1986 年同後期博士課程単位取得退学．1986 年より南山大学経営学部情報管理学科講師，助教授，教授を経て，2000 年より同大学数理情報学部情報通信学科教授，現在に至る．この間 1988 年から 1990 年まで米国メリーランド大学計算機科学科客員研究員．工学博士．ソフトウェアアーキテクチャ，アスペクト指向計算，プログラミング言語の意味論および処理系等の研究およびそれらの技術の応用に興味を持つ．日本ソフトウェア科学会，電子情報通信学会，IEEE Computer Society，ACM 各会員．



張 漢明（正会員）

1963 年生．1989 年同志社大学工学部卒業．同年より 1992 年までオムロンソフトウェア（株）勤務．1999 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．（財）九州システム情報技術研究所研究員を経て，現在，南山大学数理情報学部情報通信学科助教授．工学博士．形式的仕様記述の研究に従事．日本ソフトウェア科学会会員．