博 士 論 文

リフレクションを利用した CORBA アプリケーション開発環境に関する研究

指導教官　落水 浩一郎 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

藤枝 和宏

平成 12 年 4 月 13 日

要旨

本論文では、CORBA のアプリケーション開発における問題点として、サーバとクライアントが異なるプラットフォームや異なる言語で開発される際のインタフェースの不一致、およびスタブ/スケルトンファイルの生成とアプリケーションへの組み込みのために生じる開発手順の煩雑さを取り上げ、これらを解決する手法を提案する。

インターフェースの不一致の問題は、インタフェースリポジトリを中心とする開発環境によって解決される。この開発環境は、更新日時を記録するインタフェースリポジトリと、そのインタフェースリポジトリを操作するためのツール、およびインタフェースリポジトリからアプリケーションに必要なスタブ/スケルトンを生成するツールから構成される。

開発手順の煩雑さは、リフレクションの可能なプログラミング言語を利用して、このインタフェースリポジトリを元にアプリケーションの実行時に必要なスタブ/スケルトンを自動的に生成して取り込むことで大幅に軽減される。

リフレクションによって提供される二つの能力のうち、Linguistic リフレクションはスタブ/スケルトンの実行時生成と、すでに存在するスタブ/スケルトンを必要に応じて改変するために用いる。

Behavioral リフレクションは、プログラミング言語の処理系の振る舞いを変更することで、特別なプログラミングや開発手順なしで、アプリケーションに必要なスタブ/スケルトンを決定し、生成したスタブ/スケルトンをアプリケーションに組み込むために用る。

本論文では、上記のアプローチをリフレクションをサポートしており、対応する CORBA の実装が存在する二つのプログラミング言語、Python および Java を用いて実装する方法を示す。Python による実装では、比較的広範囲にわたってリフレクションを提供する言語の有用性を示し、Java による実装では、狭い範囲のリフレクションしか提供しない言語でも、方法次第では上記のアプローチが実現可能であることを示す。

目 次

図 目 次

表 目 次

第 1章

はじめに

1.1 CORBA とは

CORBA (Common Object Request Broker Architecture) [1] は、オブジェクト指向技術の標準化を進めている非営利団体である OMG (Object Management Group) によって策定された、分散オブジェクトを実現するためのミドルウェアである ORB (Object Request Broker) の標準規格である。

ORB を利用することで、ネットワークを介してサービスを提供するサーバを分散オブジェクトの集まりとして実装し、クライアントは分散オブジェクトのメソッドを実行する形でサービスを利用することが可能になる。分散オブジェクトを操作する際のネットワークプロトコルや、複雑なデータ型のネットワーク上での表現形式は ORB の実装に隠蔽されるので、アプリケーション開発者はそれを意識する必要がない。特に複雑な通信プロトコルを必要とするサービスを実現する際には、従来のネットワーク・プログラミングと比べて設計、実装の手間を大幅に削減できる。

1991 年に策定された CORBA 1.1 の主要な目的は、ORB の提供するオブジェクトモデルやプログラミング方法の共通化であった。このバージョンの規格で定められていたのは、分散オブジェクトのオブジェクトモデル、分散オブジェクトの仕様を記述する IDL (Interface Description Language)、C 言語による分散オブジェクトの実装と利用を可能にする言語マッピング、および ORB の提供する基本的な API 群である。

CORBA 1.1 によって、CORBA 準拠の ORB であれば利用する ORB が異なっていても、IDL を用いて記述した分散オブジェクトの仕様は同じ物が利用できるようになり、C 言語を用いる場合にはアプリケーションをほぼ同様に記述できるようになった。しかし、CORBA 1.1 は分散オブジェクトの通信プロトコルを定めていなかったため、異なる ORB で実装されたクライアントと分散オブジェクトを組み合わせて稼動させることはできなかった。

1994 年に策定された CORBA 2.0 では、実装間のインターオペラビリティを実現するために、分散オブジェクトの通信プロトコルの枠組み GIOP (General Inter-ORB Protocol) と、その TCP/IP 上へのマッピング IIOP (Internet Inter-ORB Protocol) が定められた。さらに CORBA 2.0 では、C 言語の他に C++ と Smalltalk の言語マッピングが追加された。その後規格が更新されるごとに言語マッピングは増えており、CORBA 2.3 の時点では Ada、C、C++ 、COBOL、Java、Smalltalk の言語マッピングが定められている。

オブジェクト指向言語に対する言語マッピングでは、基本的に分散オブジェクトはその言語の持つオブジェクトに対応付けられる。分散オブジェクトを操作するプログラムを記述する際には、通常のオブジェクトの操作と同じ構文が利用できる。また、分散オブジェクトの実装はクラス定義の構文を用いて行うことができる。それ以外の言語では、分散オブジェクトとプログラミング言語のマッピングは関数呼び出しと適当なデータ型を用いて行われる。

これらの言語マッピングにより、さまざまなプログラミング言語で分散オブジェクトの操作や実装を容易に行えるようになり、さらに IIOP により、異なるプログラミング言語や ORB を用いて実装された分散オブジェクトとクライアントを組み合わせて稼動させることも可能になった。

1.2 CORBA の特徴と位置付け

CORBA は特定のプラットフォームを想定した規格ではないので、組み込み機器用の OS からメインフレームまで、さまざまなプラットフォーム上に実装されている [2]。CORBA 2.0 で IIOP が策定され、多くの言語マッピングが追加されたことで、CORBA はさまざまなプラットフォーム上で稼動する、さまざまなプログラミ

ング言語で記述されたアプリケーション間のインターオペラビリティを実現する枠組みとして認識されるようになった。CORBA 以外の ORB として知られている Java RMI [3] や DCOM [4] との間の決定的な差も、このプラットフォーム非依存性と多言語サポートにある。Java RMI はプログラミング言語として Java しかサポートしておらず、DCOM はプラットフォームとして基本的に Microsoft Windows しかサポートしていない。

CORBA の持つ役割がオブジェクトモデルや API の共通化から、アプリケーション間のインターオペラビリティの実現に移ってきたことで、CORBA に準拠しつつも、まだ言語マッピングが規格化されていないプロトタイピングに向いたスクリプト言語をサポートしたり、独自にプログラミングのより容易な言語マッピングや API を提供する ORB が増えつつある。

スクリプト言語である Python [5] や Tcl [6] や Perl [7] をサポートする CORBA 準拠の ORB の実装はいくつか存在する [8, 9, 10, 11, 12]。このうち Python をサポートするものは複数あり [9, 10, 12]、ORB 間でアプリケーションの記述を共通化するために、Python の言語マッピングの規格化が OMG によって進められている [13]。CORBA のアプリケーション開発用に新たにプログラミング言語を開発している例もある [14, 15]。CORBA で定められているものとは異なる言語マッピングと API を提供する、Java をサポートする ORB も存在する [16]。いずれのアプローチにおいても、分散オブジェクトのオブジェクトモデルや通信プロトコルなどの基本的な要素は CORBA に準拠しているので、他の CORBA 準拠の ORB で実装された分散オブジェクトやクライアントとのインターオペラビリティは保たれている。

1.3 CORBA のアプリケーション開発における問題点

さまざまなプログラミング言語で分散オブジェクトを容易に実装、操作できるという特徴を実現するために、CORBA のアプリケーション開発の手順はいくらか煩雑になっている。具体的には、IDL で記述した分散オブジェクトの仕様を格納したファイルから、各 ORB ごとに用意されている IDL コンパイラを用いて、スタブとスケルトンと呼ばれる言語マッピングを支援するソースコードを生成し、開発者の記述するアプリケーションに取り込む必要がある。CORBA のアプリケーショ

ン開発では、この開発手順が存在するために、場合によっては開発効率が大きく低下したり、開発手順の誤りからサーバとクライアントが分散オブジェクトの仕様を正確に共有しなくなることがある。

1.3.1 煩雑な開発手順

CORBA のアプリケーション開発手順の中でも、スタブとスケルトンをアプリケーションに取り込むための手順は特に煩雑である。プログラミング言語によらず、スタブやスケルトンをアプリケーションに取り込むためには、他のファイルで記述された定義を取り込むためのプログラムへの記述や開発手順が必要になる。たとえば、アプリケーションの記述に C++ を用いる場合には、ヘッダファイルを取り込む記述やコンパイルとリンクの手順がスタブとスケルトンの分だけ増えることになる。Java の場合には取り込むための記述やリンクが不要な代わりに、スタブとスケルトンのクラスファイルの中から必要なものをサーバ、クライアントの実行環境にそれぞれ配置しなければならない。Java の場合には生成されるスタブとスケルトンのファイル数が特に多く、この作業はかなり煩雑である。

スタブとスケルトンを生成する際に生成方針の指定が必要な場合もあり、それも開発手順を煩雑にする要因となる。CORBA の一部の API は、アプリケーションプログラムから利用する際に、スタブやスケルトンとして生成される特別なクラスやメソッドを必要とする。任意のデータ型の値を格納できる Any 型や、非同期メソッド呼び出しをサポートする CORBA Messaging [18] がこれに当てはまる。これらの API をすべてのアプリケーションが利用するわけではない。そこでスタブが不必要に肥大化するのを押さえるために、必要な場合にだけ IDL コンパイラに指示して生成する実装が一般的になっている。

IDL ファイルが変更されない場合には、スタブとスケルトンを生成する手順は一度だけ行えばよい。また、アプリケーションの実装・利用する分散オブジェクトのインタフェースや、アプリケーションの利用する API について変更がない場合には、アプリケーションに取り込むべきスタブとスケルトンが変わることもない。このような条件下であれば、手順の煩雑さはさほど問題にはならない。

しかし、アプリケーションの仕様が確立していない開発の初期段階では、分散オ

ブジェクトの仕様やアプリケーションの仕様が変更されるたびに煩雑な手順が必要になる。分散オブジェクトをテストするためのクライアントの開発では、テストケースの変更に伴いクライアントの仕様が変更されやすい。また、テストケースごとにクライアントを多数作成する場合もある。このような条件下では、開発手順の煩雑さが開発効率を下げる大きな要因となる。

1.3.2　インタフェースの不一致

CORBA のアプリケーション開発では、サーバとクライアントで利用するプログラミング言語や開発プラットフォームが異なることは珍しくない。利用するプログラミング言語が異なれば、サーバとクライアントで同じ IDL コンパイラの生成したスタブとスケルトンを共有することはできない。また、開発プラットフォーム間で同一ファイルを共有するのが難しい場合には、IDL ファイルをすべてのプラットフォームで共有することも難しくなる。このような状況下では、IDL ファイルが更新されたときの開発手順の誤りから、同じ IDL ファイルの異なる版から生成されたスタブやスケルトンを誤って用いられて、サーバとクライアントが分散オブジェクトの仕様を正確に共有しなくなることがある。これを本論文ではインターフェイスの不一致と呼ぶ。

CORBA では、IDL で各定義ごとにバージョン番号を振ることができて、アプリケーションの実行時にそのバージョン番号を利用してインタフェースの不一致を検出することもできる。しかし、バージョン番号は開発者が定義ごとに指定しなければならず、不一致を検出できる範囲もインタフェースの名前に振られたバージョン番号のみに限られている。そのため、開発手順の誤りから生じるインタフェースの不一致を防ぐ目的には適さない。

CORBA 以外のアプリケーション開発でも、アプリケーションを構成する複数のモジュール間のインタフェースの不一致として同じ問題は起こりうる。一般的なアプリケーションでは、この種の誤りがコンパイルやリンクの段階、つまりアプリケーションの開発時に比較的容易に見つかるのに対して、CORBA のアプリケーション開発では、実際にアプリケーションを稼動させるまで発見できない上、その発見も容易でないことが多く、より問題は深刻である。

1.4　本研究の目的

アプリケーションの開発手順の煩雑さを軽減する既存のアプローチとしては、特定のプログラミング言語と CORBA の実装を統合した開発環境 [19, 20, 21] や、CORBA のアプリケーション開発用に設計された、言語マッピングの機構をその処理系に内蔵するプログラミング言語 [14] がある。統合開発環境によるアプローチは、その環境内ですべての開発が行われる場合には有効に機能するが、利用する開発ツールや開発スタイルが環境が想定しているものと異なる場合には有効に機能しない。専用のプログラミング言語を設計するアプローチは、言語の学習コストが大きくなることや、既存のクラスライブラリを利用できないといった欠点がある。いずれのアプローチも、クラスプラットフォームの開発におけるインタフェースの不一致の問題については考慮されていない。

本研究では、まずクロスプラットフォームの開発でインタフェースの不一致を防ぐことを目的に、開発に利用するプログラミング言語や CORBA の実装に依存せずに支援可能な、CORBA のインタフェースリポジトリを利用した開発環境を実現する。この環境を元に、CORBA のアプリケーションの開発可能なプログラミング言語の中でも、リフレクションをサポートする言語を対象に、リフレクションの持つ能力を利用して、CORBA の実行時ライブラリの一部として CORBA の言語マッピングの機能を提供することで、開発手順の煩雑さを軽減する手法を提案する。

1.4.1　インタフェースリポジトリ中心の開発環境

CORBA では IDL ファイルに記述された内容を格納し、アプリケーションから実行時に定義を参照できるようにする、**インタフェースリポジトリ**と呼ばれるサーバが規定されている。CORBA のプラットフォーム非依存性により、CORBA の実装が存在しているプラットフォームであれば、どんなプラットフォームからでもインタフェースリポジトリを参照できる。インタフェースリポジトリで IDL で記述された分散オブジェクトの仕様を管理して、アプリケーション開発時に利用することができれば、クラスプラットフォームのアプリケーション開発でインタフェースの不一致を防ぐことが容易になるはずである。

しかし、ほとんどの実装では、インタフェースリポジトリをアプリケーション

開発に利用することはできない。インタフェースリポジトリに格納された定義からスタブとスケルトンを生成する方法はおろか、定義を IDL ファイルの形で取り出す方法すら提供されていないからである。さらに、CORBA のインタフェースリポジトリは、格納されている定義が更新されたことをバージョン番号に依らずに確認する方法を提供してない。そのため、インタフェースリポジトリからスタブとスケルトンを生成する方法があったとしても、定義とスタブとスケルトンの間の一貫性を保つことが容易ではない。

本研究では、インタフェースリポジトリをアプリケーション開発に利用できるようにするために、更新日時を記録できるインタフェースリポジトリと、インタフェースリポジトリに格納された定義とスタブとスケルトンの間の一貫性を保つツールを開発する。このツールはスタブとスケルトンを生成する際に、インタフェースリポジトリに格納された定義から必要な IDL ファイルを生成して、それに対して既存の IDL コンパイラを実行する。この実装方法により、利用するプログラミング言語や CORBA の実装に依存せずに、このツールを利用したアプリケーション開発が可能になる。

この環境により、異なる開発プラットフォームの間で、インタフェースリポジトリを利用して IDL によって記述された定義を共有し、その定義とスタブとスケルトンの間の一貫性を保つこともできるので、クラスプラットフォームでのアプリケーションの開発中に、開発手順の誤りから生じるインタフェースの不一致を防ぐことが容易になる。

1.4.2 リフレクションによるスタブとスケルトンの実行時自動生成

リフレクションの可能なプログラミング言語では、必要なスタブとスケルトンを生成してアプリケーションに取り込む手順を、リフレクションの持つ能力を利用して、CORBA の実行時ライブラリの一部としてアプリケーションに組み込むことで、アプリケーションの実行時にこの手順を自動的に行うことができる。

計算システムが自分自身の構成や計算過程に関する計算を行うことを**リフレクション**という [22]。リフレクションは記述力を高めることを目的にプログラミング言語に導入されることが多く、広く用いられているプログラミング言語でリフ

レクションをサポートするものも少なくない。

リフレクションの可能なプログラミング言語では、プログラム自身を参照、改変するプログラムやプログラムを生成して自身に追加するプログラムを記述したり、プログラムの実行環境の持つ情報や振る舞いを参照、改変するプログラムを記述することができる。本論文では前者のプログラム自身に関するリフレクションの能力を **Linguistic リフレクション**、後者の実行環境に関するリフレクションの能力を **Behavioral リフレクション**と呼ぶ。

Linguistic リフレクションは、アプリケーションに必要なスタブとスケルトンを実行時に生成するために利用する。実行時にスタブとスケルトンを生成する際に必要な定義は前述したインタフェースリポジトリから取得できる。Behavioral リフレクションは、必要なスタブとスケルトンの特定とアプリケーションへの取り込みを、開発者による特別なプログラミングや手順なしで可能にするために、プログラミング言語の実行環境を操作する際に利用する。

このアプローチは、CORBA のアプリケーション開発が可能であり、リフレクションの可能なプログラミング言語である Python と Java について、既存の言語マッピングを変更することなく実現できる。その他のプログラミング言語と組み合わせてアプリケーション開発を行う場合には、前述したインタフェースリポジトリを利用した開発環境を利用することで、インタフェースの不一致が生じるのを押さえることができる。

1.5 本論文の構成

本論文では、第 2 章で CORBA のアプリケーション開発の概要を述べ、開発手順の煩雑さとインタフェースの不一致に関する問題を明らかにする。第 3 章では、インタフェースの不一致を防ぐことができる、インタフェースリポジトリを利用した開発環境について論じる。

第 4 章では、リフレクションの可能なプログラミング言語の持つ能力を利用して、スタブとスケルトンを実行時に自動的に生成する手法を述べると共に、そのために必要となるリフレクションの能力について分析する。第 5 章および第 6 章では、それぞれ第 4 章で論じた手法を Python と Java の持つリフレクション能力を

利用して実装する方法を述べる。

第 7 章では、CORBA のアプリケーション開発を容易する既存のアプローチと本研究のアプローチを比較し、クラスプラットフォームでのアプリケーション開発の支援や、プログラミングの互換性の点で、本研究のアプローチが優位であることを示す。

最後に、第 8 章でまとめと今後の展望について述べる。

第 2章

CORBAのアプリケーション開発の概要と問題点

CORBA のアプリケーションを開発する際には、IDL ファイルから IDL コンパイラが生成したスタブとスケルトンをアプリケーションに取り込む必要があるため開発手順が煩雑になる。アプリケーション開発に複数の CORBA の実装、プログラミング言語や開発プラットフォームを利用する場合には、インタフェースの仕様を正確に共有できずにインタフェースの不一致を起こすこともある。この章では、CORBA のアプリケーション開発の概要を述べた後、これらの問題点とそれを解決するアプローチについて議論する。

2.1　アプリケーション開発の概要

CORBA のアプリケーション開発は以下の手順で行われる。

- IDL ファイルの記述
- IDL コンパイラを用いたスタブとスケルトンの生成
- サーバの記述
- クライアントの記述
- 構築と配置

以下各手順について説明する。

2.1.1 IDL ファイルの記述

CORBA のアプリケーション開発は、分散オブジェクトのインタフェースを IDL (Interface Description Language) で記述したファイルを作成することから始まる。既存の分散オブジェクトを利用するクライアントを作成する場合には、すでに記述されているものを入手する。

IDL ファイルの例を図 2.1 に示す。これは名前と値の対を管理する分散オブジェクトのインタフェースを記述したものである。IDL は C++ [23] の文法を参考に設計されている。この例で用いられているキーワード `module` は C++ の `namespace` に相当し、名前空間の分離のために用いられる。C++ の `class` に相当するのは `interface` であり、ここで分散オブジェクトのインタフェースを定義する。インタフェースでは、主にメソッド (CORBA ではオペレーションと呼ばれる) と属性を定義する。継承するインタフェースは C++ と同様に複数指定できる。また、IDL ファイルは C++ と同じプリプロセッサで処理されることが規格で規定されているので、`#include` 指令を用いて関連する定義を複数のファイルに分割することもできる。

IDL ファイルには、分散オブジェクトのインタフェースだけでなく、インタフェースの使用する複雑なデータ型、例外や定数値の定義も記述する (図中に太字で示した)。複雑なデータ型としては構造体や共用体、配列、シーケンス (可変長の列) などが利用できる。共用体の仕様は C++ とはまったく異なっており、どのメンバを利用しているかを示すタグの型と、タグの値とメンバの対応を定義できる。これらの定義はインタフェース内で行ってもよい。

2.1.2 スタブとスケルトンの生成

必要な IDL ファイルがそろったら、利用するプログラミング言語ごとに CORBA の実装が提供している IDL コンパイラを実行して[1]、IDL ファイルからスタブとスケルトンを生成する。スタブは IDL ファイルで定義されたインタフェースを持つ分散オブジェクトとユーザ定義型の操作に必要なプログラムであり、スケルトン

[1]提供している IDL コンパイラは一つで、オプションで利用するプログラミング言語を指定する実装もある

```
module NameTable {
  const string dummy_key = "**dummy**";
  enum EntryType { T_BOOL, T_INT, T_FLOAT, T_STRING };
  union EntryValue switch (EntryType) {
    case T_BOOL: boolean b;
    case T_INT: long i;
    case T_FLOAT: float f;
    case T_STRING: string s;
  };
  exception NotMatch {};
  exception AlreadyExist {
    EntryValue ent;
  };
  interface NameTableServer {
    void insert(in string key, in EntryValue ent,
                in boolean force)
      raises (AlreadyExist);
    void delete(in string key) raises (NotMatch);
    EntryValue lookup(in string key) raises (NotMatch);
  };
};
```

図 2.1: IDL ファイルの例 (nametable.idl)

は分散オブジェクトの実装に必要なプログラムである。

IDL コンパイラによって生成されるファイルの数や種類は、CORBA の実装や対象とするプログラミング言語によって異なる。C++ を対象とする一般的な実装の IDL コンパイラは、IDL ファイル一つに対して、スタブとスケルトンとしてヘッダファイルとソースファイルを一つずつ生成する。たとえば、VisiBroker for C++ [24] の IDL コンパイラは、nametable.idl に対してスタブとして nametable_c.h と nametable_c.cc を、スケルトンとして nametable_s.cc, nametable_s.h を生成する。サーバとクライアントに共通して必要なものとそうでないものを細かく分けて生成する実装もあり [25]、そのような実装ではファイルの数がいくらか増える。

Java を対象とする場合には、IDL コンパイラは Java の慣習に従い生成するクラスごとにファイルを作成する。したがって、IDL ファイルの名前と生成されるファイルの名前の間には直接的な関係はない。IDL コンパイラによって生成されるクラ

スには、IDL ファイルで定義されたインタフェースやユーザ定義型に直接対応するスタブクラス、インタフェースのスケルトンクラスの他に、すべての型の **Helper クラス**と **Holder クラス**が含まれる。これらはスタブの一部であり、Helper クラスはその型について CORBA の API を実行する際に、Holder クラスは引数の参照渡しを行う際に用いられる。このような構成を取っているために、IDL ファイルに定義されている内容が多い場合には非常に多くのファイルが生成される。たとえば、先の `nametable.idl` に対して VisiBroker for Java [26] の IDL コンパイラは 18 個のファイルを生成する。

2.1.3 サーバの記述

CORBA のサーバは、IDL ファイルで定義されたインタフェースを持つ分散オブジェクトの実装と、分散オブジェクトのインスタンスをセットアップするメインプログラムからなる。

分散オブジェクトの実装

オブジェクト指向言語を利用する場合には、インタフェースに対応するスケルトンクラスを継承したクラスを定義することで、そのインタフェースを持つ分散オブジェクトを実装できる。図 2.1 の IDL ファイルで定義した `NameTableServer` インタフェースの Java 言語による実装の一部を図 2.2 に示す。図中の *`_NameTableServerImplBase`* がスケルトンクラスである。IDL ファイルで定義されたデータ型や例外は、図中に太字で示したスタブクラスを介して扱う。

メインプログラムの記述

サーバのメインプログラムの記述方法は CORBA の実装によって若干異なる。この実装間の互換性の低さは、ORB と分散オブジェクトのインスタンスの間を仲介するオブジェクトアダプタとして、CORBA 2.0 で定められた Basic Object Adaptor の規格の曖昧さと能力の低さに起因するものである。CORBA 2.2 で導入された Portable Object Adaptor によって、この問題は規格上ではすでに解決されているが、

```
import NameTable.*;
public class NameTableServerImpl
    extends _NameTableServerImplBase {
  private java.util.Hashtable
    table_impl = new java.util.Hashtable();
...
  public void insert(String key,
                     EntryValue ent,
                     boolean force)
              throws AlreadyExist {
    if (!force && table_impl.containsKey(key)) {
      throw new AlreadyExist((EntryValue)table_impl.get(key));
    }
    table_impl.put(key, ent);
  }
...
}
```

図 2.2: NameTableServer の実装

現時点では一部の実装でしかサポートされていない。

ここでは Visibroker for Java の場合のメインプログラムを図 2.3 に示す。このプログラムでは、ORB の初期化、Basic Object Adaptor の初期化、分散オブジェクトを実装したクラスのインスタンスの生成とオブジェクトアダプタへの登録を経て、最後にリクエストを待つループに入っている。

2.1.4 クライアントの記述

分散オブジェクトを操作するクライアントプログラムは、まず ORB を初期化した後、分散オブジェクトのオブジェクトリファレンスを取得する。オブジェクトリファレンスの取得方法はおおむね以下の 3 通りである。

- NamingService を利用する

  ORB オブジェクトの持つメソッド `resolve_initial_reference` を用いて NamingService のオブジェクトリファレンスを取得し、NamingService に問い合わせてオブジェクトリファレンスを取得する。

- Stringified IOR(Interoperable Object Reference) を利用する

```
import NameTable.*;
public class Server {
 public static void main(String[] args) {
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
   org.omg.CORBA.BOA boa = orb.BOA_init();
   NameTableServer srv = new NameTableServerImpl("NameTable");
   boa.obj_is_ready(srv);
   boa.impl_is_ready();
 }
}
```

図 2.3: サーバのメインプログラム

オブジェクトリファレンスを文字列化したものをファイルなどを経由して読み込み、ORB オブジェクトのメソッド `string_to_object` によりオブジェクトリファレンスに変換する。

- Parsistent Object Reference を利用する

  サーバ側でインスタンスを生成するときか、オブジェクトアダプタに登録するときにに指定したオブジェクトキーを元に、オブジェクトリファレンスを取得する。この方法は現在の規格では定められていないため、実際の利用方法は実装により異なり、サポートしていない実装もある。

取得したオブジェクトリファレンスを用いて、分散オブジェクトのオペレーションを呼び出す方法は二つある。一つは、インタフェースのスタブを利用した Static Invocation Interface (以降 SII と略記) であり、もう一つはインタフェースのスタブを利用しない Dynamic Invocation Interface (以降 DII と略記) である。

Static Invocation Interface

Persistent Object Reference を利用してオブジェクトリファレンスを取得し、SII を用いてオペレーションを呼び出す、Java で記述したクライアントプログラムを図 2.4 に示す。

SII では、インタフェースのスタブクラス (`NameTableServer`) を利用することで、分散オブジェクトのオペレーションを通常のオブジェクトのメソッドと同様

```
import NameTable.*;
public class Client {
  public static void main(String[] args) {
    org.omg.CORBA.ORB orb =
      org.omg.CORBA.ORB.init(args, null);
    // Persistent Object Referenceを用いた
    // オブジェクトリファレンスの取得
    NameTableServer srv =
      NameTableServerHelper.bind(orb, "NameTable");
    // union EntryValueの値の生成
    EntryValue val = new EntryValue();
    val.i(1000);
    String key = "hoge";

    try {
      // オペレーションinsertの実行
      srv.insert(key, val, true);
    } catch (AlreadyExist e) {
      System.out.println("'" + key + "' already exist.");
    }
  }
}
```

図 2.4: クライアントプログラムの例

にメソッド呼び出しの構文を用いて呼び出すことができる。`insert` オペレーションの送出する例外 `AlreadyExist` の処理は、通常の例外と同様に `try–catch` 構文で処理できる。共用体 `EntryValue` は、Java では `EntryValue` クラスに対応付けられており、IDL コンパイラの生成したメソッドを用いて値を容易に設定できる。

Dynamic Invocation Interface

同じ呼び出しを DII で行うプログラムを図 2.5 に示す。図 2.4 に灰色の背景で示した部分を DII で置き換えたものが、図 2.5 の灰色の背景で示した部分である。この例では、インタフェースのスタブクラス `NameTableServer` と Helper クラス `NameTableServerHelper` を利用せずに、`NameTableServer` インタフェースを持つ分散オブジェクトを操作している。

```
// オブジェクトリファレンスの取得
org.omg.CORBA.Object srv =
  orb.bind("IDL:NameTable/NameTableServer:1.0",
           "NameTable", null, null);
EntryValue val = new EntryValue();
val.i(1000);
String key = "hoge";

// オペレーションを指定しRequestオブジェクトを生成
org.omg.CORBA.Request req = srv._request("insert");
// 引数の設定
req.add_in_arg().insert_string(key);
org.omg.CORBA.Any val_any = req.add_in_arg();
EntryValueHelper.insert(val_any, val);
req.add_in_arg().insert_boolean(false);
// 例外の設定
req.exceptions().add(AlreadyExistHelper.type());
// 実行
req.invoke();
// 例外の処理
org.omg.CORBA.UnknownUserException e =
  (org.omg.CORBA.UnknownUserException)
    req.env().exception();
if (e != null &&
  e.except.type().equals(AlreadyExistHelper.type())){
  System.out.println("'" + key + "' already exist.");
}
```

図 2.5: Dynamic Invocation Interface の実行例

DII はスタブを必要としない代わりに、オペレーションの実行と例外処理にプログラミング言語のメソッド呼び出しや例外処理の構文を利用できないため、プログラミングがかなり複雑になる。この例では共用体 `EntryValue` および例外 `AlreadyExist` の操作にはスタブを利用している。これらの操作を DynAny API を利用してスタブを用いずに行うことも可能だが、さらにプログラミングが煩雑になる。DII はプログラミングが煩雑になるだけでなく実行速度も遅く、同じ呼び出しを行う SII と比べて数倍から 10 数倍程度遅いことが知られている [27]。

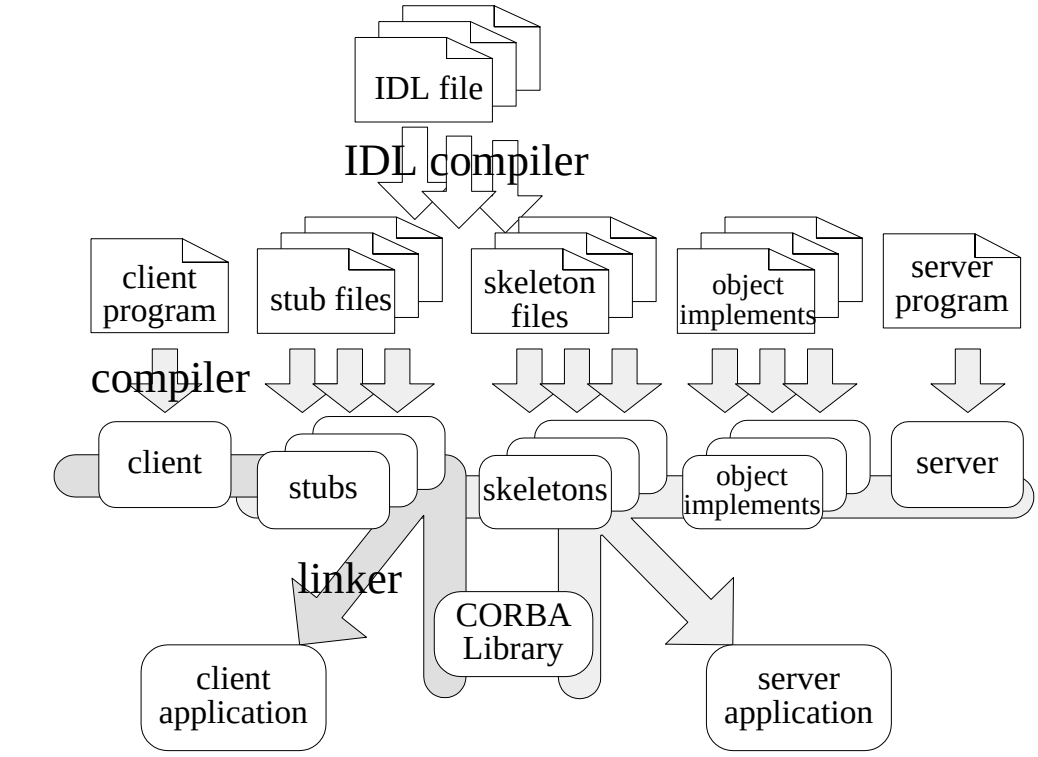

図 2.6: 構築手順の概要

2.1.5 構築と配置

記述したプログラムを実際にアプリケーションとして稼動させるには、IDL コンパイラの生成したソースコードと、開発者の記述したソースコードのすべてをコンパイルし、C++ の場合には生成されたオブジェクトファイルの中からサーバとクライアントについてそれぞれ必要なものと、CORBA の実行時ライブラリをリンクして実行ファイルを生成する必要がある (図 2.6)。

Java の場合には、リンクする必要がない代わりに、コンパイラによって生成されたクラスファイルのうち、クライアントとサーバに必要なものをそれぞれの実行環境に配置する必要がある。

図 2.1 の IDL ファイルから生成された 18 個のスタブとスケルトン、および開発者の記述したソースコードから生成されたクラスファイルのうち、図 2.3 のサーバと図 2.4 のクライアントを稼動させるために必要なファイルをまとめたものを表 2.1 に示す。この表では、スタブとスケルトンのクラスファイルを灰色の文字で示

サーバ側	クライアント側
Client.java	Server.java
NameTableServer.class	NameTableServer.class
	NameTableServerHelper.class
EntryType.class	EntryType.class
EntryValue.class	EntryValue.class
EntryValueHelper.class	EntryValueHelper.class
AlreadyExist.class	AlreadyExist.class
AlreadyExistHelper.class	AlreadyExistHelper.class
NotMatch.class	NotMatch.class
NotMatchHelper.class	NotMatchHelper.class
	_st_NameTableServer.class
_NameTableServerImplBase.class	
NameTableServerImpl.class	

表 2.1: アプリケーションの稼動に必要なクラスファイル

した。このうち、開発者の記述したソースコードから直接参照されているものは濃い灰色で、スタブとスケルトンから間接的に参照されているものは薄い灰色で示している。

2.2 スタブとスケルトンの役割

スタブとスケルトンを利用することで IDL で記述された定義がプログラミング言語の持つ要素に対応付けられるため、プログラミングが非常に容易になる。他にもスタブとスケルトンは、IDL で記述された定義に依存する通信プロトコルに関する処理をアプリケーションから隠蔽したり、CORBA の API を実行するために必要なメソッドや定数をアプリケーションに提供する役割も持っている。以下、それぞれの役割について考察する。

2.2.1 プログラミングの効率化と誤り検出

図 2.1 に示したように、IDL ファイルには分散オブジェクトのインタフェースそのものの定義の他に、ユーザ定義型や例外の定義や定数値の定義が含まれている。スタブの持つ役割は、これらの定義を実装に用いるプログラミング言語から利用

できる形でアプリケーションプログラムに提供することである。表 2.1 において、濃い灰色で示したアプリケーションプログラムから直接参照されるスタブクラスは、この役割のために用いられている。

スタブに含まれている定義によって、IDL による定義と開発者の記述したプログラムの間の矛盾はプログラミング言語の処理系で検出できる。クライアントがオペレーションの引数として誤った型の値を渡した場合には、静的な型を持つプログラミング言語であれば、スタブクラスの対応するメソッドの定義を元に処理系によって型の不一致として検出される。同様にスケルトンに含まれている定義によって、IDL によるインタフェースの定義と分散オブジェクトの実装の間の矛盾を検出される。動的な型を持つプログラミング言語においても、IDL による定義に基づいて IDL コンパイラがスタブとスケルトンに型検査の処理を埋め込むことで、実行時に型の不一致を検出することができる。

2.2.2 通信プロトコルの処理

スタブとスケルトンには、クライアントと分散オブジェクトの間で用いられる通信プロトコルの処理が含まれている。インタフェースに対応するスタブクラスにはメソッド呼び出しからリクエストプロトコルへの変換操作が、スケルトンクラスにはその逆の操作が含まれている。

ユーザ定義型の値とネットワークストリームとの相互変換は、ユーザ定義型に対応するスタブクラスに含まれている。これらはインタフェースに対応するスタブ、スケルトンクラスの双方から利用されるので、サーバクライアントの両方に必要である。Java の場合には Helper クラスにこの処理が含まれている。変換処理の実装だけが用いられる Helper クラスはアプリケーションからは直接参照されないので、表 2.1 において薄い灰色で示されている。

一般に、クライアントと分散オブジェクトの間の通信はそのものは CORBA の実装の実行時ライブラリによって行われ、通信プロトコルに至るまでの処理のうち、IDL による定義に依存する大きく部分がスタブやスケルトンとして生成される。どの範囲までスタブとスケルトンとして展開するかは CORBA の実装によって異なっている。範囲を広げれば、スタブとスケルトンのサイズが大きくなる代

```
// 引数の設定
req.add_in_arg().insert_string(key);
org.omg.CORBA.Any val_any = req.add_in_arg();
// Any 型に EntryValue 型の値を挿入
EntryValueHelper.insert(val_any, val);
req.add_in_arg().insert_boolean(false);
// 例外の設定 (TypeCode を利用)
req.exceptions().add(AlreadyExistHelper.type());
```

図 2.7: Any 型の利用例

わりに、プロトコルに関する処理が各定義ごとに最適化された形で展開できるので、アプリケーションの性能を向上させることができる [28]。

2.2.3 CORBA の API の支援

CORBA の提供する API のうち、IDL による分散オブジェクトの定義に強く依存するものは、それを実行する際に IDL コンパイラがスタブとして生成するメソッドや定数の定義を必要とする。このような API の例としては Any 型や CORBA Messaging がある。

Any 型

Any 型は IDL で分散オブジェクトの仕様を記述する際に利用できる標準のデータ型の一つであり、任意のデータ型の値を扱うために用いられる。プログラミング言語で Any 型を扱う際には、格納されている値の型を識別する TypeCode と値のバイト列の組として扱われる。TypeCode はその型の持つ名前や構造を表現したデータであり、IDL コンパイラによってスタブの一部として生成される。Any 型に値を格納したり取り出したりする操作は、基本型については実行時ライブラリに含まれており、IDL で定義された型については IDL コンパイラによってスタブの一部として生成される。たとえば、Java の場合にはこれらは Helper クラスに含まれており、先に示した図 2.5 でも一部利用している。該当する個所を図 2.7 に示す。

```
// オペレーションを指定しRequestオブジェクトを生成
org.omg.CORBA.Request req = srv._request("insert");
// 引数の設定
req.add_in_arg().insert_string(key);
org.omg.CORBA.Any val_any = req.add_in_arg();
EntryValueHelper.insert(val_any, val);
req.add_in_arg().insert_boolean(false);
// 例外の設定
req.exceptions().add(AlreadyExistHelper.type());
// 実行
req.send_deferred();
// 返値を待たずに行う処理
...
// 返値を受け取る
req.get_response();
// 例外の処理
org.omg.CORBA.UnknownUserException e =
 (org.omg.CORBA.UnknownUserException)req.env().exception();
if (e != null &&
  e.except.type().equals(AlreadyExistHelper.type())){
```

図 2.8: Deferred Synchronous Interface の実行例

CORBA Messaging

クライアントから分散オブジェクトのオペレーションを呼び出すと、IDL で定義する際に oneway を指定したオペレーション以外は、サーバから返値が戻るまで待つ同期呼び出し (Synchronous Invocation) になる。CORBA では、返値を待たずに別の処理を行い、後で結果を受け取る遅延同期呼び出し (Deferred Synchronous Invocation) もサポートされている。これを利用するには DII を用いなければならないため、プログラミングが非常に煩雑になる。図 2.4 の insert オペレーションの呼び出しを遅延同期で行うプログラムを図 2.8 に示す。

CORBA 3.0 に含まれる予定の CORBA Messaging [18] と呼ばれる非同期呼び出しの規格では、IDL コンパイラが遅延同期呼び出し用の特別なメソッドをスタブとして生成するので、このメソッドを利用することでプログラミングが非常に簡潔になる。CORBA Messaging では IDL コンパイラがスタブを作成する際に、オペレーションごとに通常のスタブメソッドの他に poll モデル、callback モデルの 2 種類の

```
...
    AMI_NameTableServerPoller poller;
    // オペレーション insert の実行
    poller = srv.sendp_insert(key, val, true);
    // 返値を待たずに行う処理
    ...
    // 返値を受け取る
    try {
      poller.insert(-1 /* タイムアウトの指定 */);
    } catch (AlreadyExist e) {
      System.out.println("'" + key + "' already exist.");
    }
...
```

図 2.9: CORBA Messaging による遅延同期呼び出し

非同期呼び出しのためのメソッドと、これらの非同期呼び出しをサポートする補助クラスを生成する。poll モデル用に生成されたメソッドと補助クラスを利用することで、SII と同様に通常のメソッド呼び出しの構文を用いて、遅延同期呼び出しを実行することが可能になる (図 2.9)。図中の `AMI_NameTableServerPoller` が補助クラス、`sendp_insert` が poll モデル用のスタブメソッドである[2]。

2.2.4 スタブとスケルトンを利用しないプログラミング

CORBA ではスタブとスケルトンをまったく利用せずに、分散オブジェクトの利用と実装を可能にする API が一通り用意されている。前述したように Dynamic Invocation Interface を利用すれば、スタブを用いずに分散オブジェクトのオペレーションを実行できる。スタブを用いないユーザ定義型の操作は DynAny API によって可能になる。Dynamic Skeleton Interface を利用すれば、スケルトンを用いずに分散オブジェクトを実装することもできる。

しかし、これらの API を利用する場合には、スタブとスケルトンとして展開されている操作の一部を開発者が記述しなければならないため、プログラミングが非常に煩雑になる上、プログラミングの誤りを処理系で検出することもできなく

[2]CORBA Messaging の実装はまだ利用できないので、このプログラムは規格を元に想像したものである

なる。また、スタブとスケルトンではIDLによる定義に基づいて最適化されている操作が、これらのAPIでは各定義に依存しない汎用のライブラリで実行されるため性能の劣化が避けられない。

一般には、これらのAPIはIDLファイルの定義に本質的に依存できないアプリケーション、たとえばIIOPとは異なるプロトコルを用いるORBとの間のブリッジや、汎用のデバッグツールやテストツールの実装に用いられる。

2.3 アプリケーション開発における問題点

CORBAのアプリケーション開発では、IDLコンパイラの生成したスタブとスケルトンを利用することでプログラミングが容易になる代わりに、開発手順が煩雑になる傾向がある。状況によってはこの開発手順の煩雑さが、開発効率を大きく低下させたり、サーバとクライアントが分散オブジェクトの仕様を正確に共有しない状態を引き起こすことがある。

2.3.1 煩雑な開発手順

スタブの生成方針の指定

IDLコンパイラでスタブを生成する際には、アプリケーションの利用するCORBAのAPIに応じてスタブの生成方針の指定が必要になる。先に示した、Any型やCORBA Messagingを利用するためにスタブとして生成される定義は、すべてのアプリケーションに必要なわけではない。不要な定義によってスタブが大きくなると、コンパイルの必要な言語ではスタブやそれを取り込むアプリケーションのコンパイルに時間がかかる、アプリケーションの実行ファイルが肥大化する、インタプリタ方式の言語ではアプリケーションの起動に時間がかかるなどの問題が生じる。そのため、スタブを生成する際にIDLコンパイラのオプションを用いて、各APIのためのコードを生成するかを指定する実装が一般的である。特にスタブファイルが大きくなりがちなC++では、ほとんどすべての実装で、Any型に関するスタブの生成方針をIDLコンパイラに指示できるようになっている。

CORBA Messagingの実装は現時点では存在しないが、実装されるときには同様

に指定が必要になるはずである。CORBA Messaging を実現するためには、先に述べたようにオペレーションごとに poll モデルと callback モデルの二種類のメソッドおよび補助クラスをスタブとして生成する必要がある。通常の呼び出しよりも多くのコードが各オペレーションについて必要になるため、すべてのインタフェースについてこれらを生成すると、スタブがかなり大きくなってしまう。

スタブとスケルトンのアプリケーションへの取り込み

IDL コンパイラの生成したスタブやスケルトンをアプリケーションに取り込むには、プログラムにモジュールを取り込むための記述や、モジュールをアプリケーションに結合する手順、あるいはモジュールをアプリケーションの実行環境に配置する手順が必要である。

たとえば、C++ ではスタブとスケルトンのヘッダファイルを取り込むための記述と、アプリケーションにスタブとスケルトンを結合するためのコンパイルおよびリンクの手順が必要である。C++ では、IDL ファイル単位でスタブとスケルトンが生成されるので、一つの IDL ファイルに必要な定義をすべて記述することで生成されるファイルの数を減らすことができる。この方法により、取り込むファイルの指定やコンパイルやリンクの手間を減らすことは可能である。しかし、多くの定義を含む IDL ファイルから生成されるファイルは非常に大きくなり、IDL ファイルが変更されたときの再構築のコストが非常に大きくなる。また、大きな IDL ファイルは他のプロジェクトで再利用する際に不便であり、IDL ファイルは複数のファイルに分割されることが多い。

Java ではスタブやスケルトンを取り込む記述やリンクの手順が不要な代わりに、スタブとスケルトンをコンパイルして生成されたクラスファイルの中から、アプリケーションに必要なものをサーバとクライアントの実行環境に配置する必要がある。Java の場合には、IDL ファイルから生成されるスタブとスケルトンの数が非常に多く、それぞれの間の依存関係も複雑なため、必要なクラスファイルを実行環境に配置する手順は非常に煩雑になる。

コンパイルの不要なスクリプト言語を CORBA のアプリケーション開発に利用することで、開発手順をいくらか簡略化することもできる。しかし、依然として取り込むファイルの指定や、生成されたスタブやスケルトンを適切に配置する手順

は必要である。たとえば、Python を利用する場合には、取り込むスタブやスケルトンのモジュールの指定と、そのモジュールを配置する手順の両方が必要である。

開発効率の低下する場合

スタブとスケルトンの生成、コンパイル、リンクあるいは配置といった一連の作業は、make などの構築支援ツールを利用することで自動化が可能である。これらのツールを利用するためには、事前にアプリケーションを構築するためのルールの記述が必要である。IDL ファイルが変更されず、アプリケーションの利用する分散オブジェクトの種類や CORBA の API が変更されない場合には、このルールに変更が加えられることはないので、ルールの記述にかかる手間はさほど問題にはならない。

しかし、開発の初期の段階で適切なインタフェースを設計するためにプロトタイピングを行う際や、既存の IDL と分散オブジェクトの実装を元に、再利用性を高めるために共通性の高いインタフェースと実装を抽出するなどの refactoring [29] を行う際には、IDL ファイルの変更が多くなるのは避けられない。また、分散オブジェクトをテストするクライアントを作成する場合には、テストケースの変更によって、利用する API や分散オブジェクトの種類も変更される。このような状況では、ルールの記述を変更する手間や、スタブやスケルトンの再コンパイルや再配置にかかるコストが開発者に大きな負担になる。

2.3.2 インタフェースの不一致

CORBA のアプリケーション開発では、サーバとクライアントで実装に用いるプログラミング言語、ORB や開発/実行プラットフォームが異なることは珍しくない。サーバの開発プラットフォームが UNIX でクライアントは Microsoft Windows であるとか、サーバの実装に C++ をクライアントには Java を用いるといった組み合わせが典型的なケースであり、CORBA の多くの実装がこれらの組み合わせに対応している [2, 30]。また、分散オブジェクトをテストするクライアントを作成する場合や、クライアントのプロトタイピングの際には、コンパイルの手順の不要なスクリプト言語が用いられることもある。

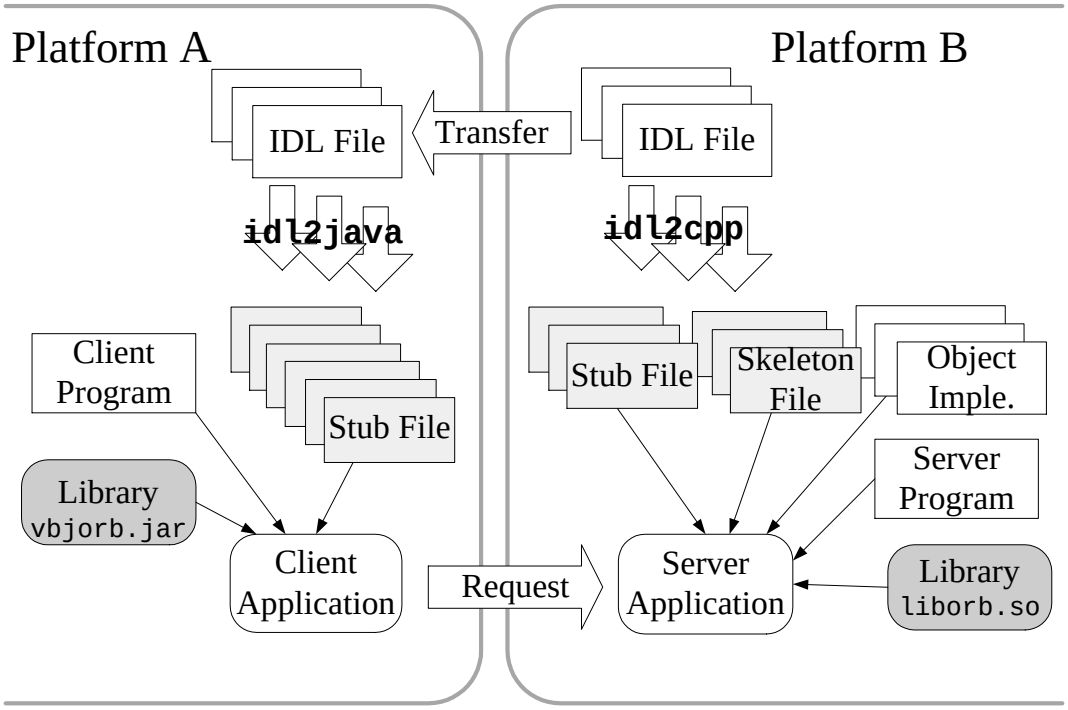

図 2.10: クロスプラットフォームでの CORBA のアプリケーション開発

サーバとクライアントでプログラミング言語や CORBA の実装が異なる場合には、図 2.6 のようにスタブを両者で共有することはできない。開発プラットフォームが異なる場合には、同じファイルを共有するのことが難しい場合もあり、その場合には IDL ファイルまでも共有できなくなる。IDL ファイルやスタブが共有できない場合には開発手順はさらに煩雑になる (図 2.10)。

このようなクロスプラットフォームの開発では、IDL ファイルが更新されたときに、それがサーバとクライアントの両方に正しく反映されずに、サーバが実装している分散オブジェクトのインタフェースと、クライアントが分散オブジェクトの操作に用いるインタフェースが一致しなくなる確率が高くなる。

プラットフォームや言語などが混在しないアプリケーション開発では、**同じ IDL ファイルから同時に生成されたスタブとスケルトン**により、IDL ファイルの変更にアプリケーションプログラムが正しく追随していなければ、プログラミング言語の処理系でそれを検出できる。しかし、クロスプラットフォームの開発ではこの前提が崩れるため、サーバとクライアントのいずれかで古い IDL ファイルや古

いスタブやスケルトンを用いるミスが生じ易くなり、アプリケーションプログラムが IDL ファイルの変更に追随していることを検出できなくなる。

インタフェースの不一致があると、アプリケーションを稼動させたときにサーバとクライアントの間の通信プロトコルに矛盾が生じる。この矛盾は運が良ければ CORBA の実装によって検出されて、`MARSHAL` や `BAD_OPERATION` といったシステム例外の形で捕捉することができるが、運が悪い場合にはアプリケーションの挙動が意図した通りにならないだけで、一見正常に動作してしまうので検出が非常に難しくなる。エラー検査の不十分な、あるいは性能を向上させるためにあえてエラー検査を減らしている CORBA の実装では、インタフェースの不一致がアプリケーションをクラッシュさせることもある。

2.4 解決へのアプローチ

2.4.1 開発手順の煩雑さ

開発手順の煩雑さを軽減する既存のアプローチとしては、CORBA のアプリケーション開発に対応した統合開発環境 [20, 21, 19] がある。統合開発環境は、すべてのアプリケーション開発がその環境の中で閉じている場合には非常に有用である。しかし、これらの環境では開発スタイルに関する制約がきつく、統合開発環境が直接サポートしていない開発ツール、ライブラリやミドルウェアを利用する場合には、かえって開発手順が煩雑になることもある。利用するプログラミング言語や CORBA の実装や開発プラットフォームが混在することの多い CORBA のアプリケーション開発においても、これは同様である。

統合開発環境ではなくプログラミング言語の処理系だけで解決するアプローチとして、言語マッピングの機構をその処理系に内蔵した CorbaScript [14] がある。新しいプログラミング言語を設計するアプローチには、プログラミング言語の学習コストがかかることや、既存のクラスライブラリを利用できないといった問題がある。このような専用言語は利用可能なアプリケーションの範囲が限られているため、汎用の言語に見られるような、さまざまなクラスライブラリが広く開発されることも期待できない。

```
interface NameTableServer {
  void insert(in string key, in EntryValue ent,
              in boolean force)
    raises (AlreadyExist);
  void delete(in string key) raises (NotMatch);
  EntryValue lookup(in string key) raises (NotMatch);
};
#pragma version NameTableServer 1.1
```

図 2.11: インタフェースにバージョン番号を振る

2.4.2 インタフェースの不一致

CORBA では、分散オブジェクトのインタフェースを定義する際にバージョン番号を付けることができて、これを利用してインタフェースの不一致を検出することもできる。たとえば、図 2.1 のインタフェース `NameTableServer` にバージョン番号 1.1 を振る場合には、プリプロセッサの `#pragma` 指令を用いて図 2.11 のように記述する。このバージョン番号はスタブとスケルトンの両方に埋め込まれて、CORBA の標準のオペレーションの一つである `is_a` オペレーションによって照合される。`is_a` オペレーションはオブジェクトリファレンスを扱う API によって暗黙のうちに実行されるので、開発者が明示的に実行を指示する必要はない。たとえば、図 2.4 ではオブジェクトリファレンスを取得する `bind` メソッドの中で `is_a` オペレーションが実行される。

バージョン番号は IDL で定義する識別子ごとに指定できるが、この方法で照合できるのはインタフェースのバージョン番号だけである。インタフェースの不一致を起こさないようにするためには、ユーザ定義型が更新されたときには、それを利用するインタフェースの番号をすべて更新する必要がある。また、仕様が変更されたインタフェースが別のインタフェースに継承されている場合には、そのインタフェースのバージョン番号も更新する必要がある。IDL ファイルが更新されることが多い状況で、開発手順の誤りから生じるインタフェースの不整合を防ぐ目的に使用するには、この機構はあまりにも開発者にかかる負荷が大きすぎる。

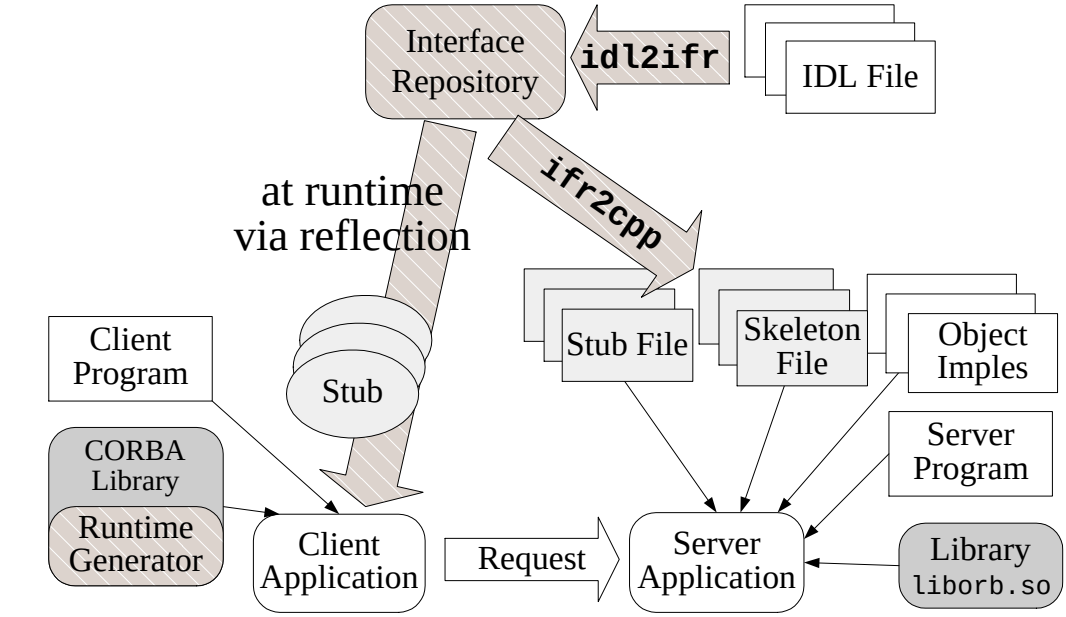

図 2.12: インタフェースリポジトリを利用した開発環境

2.4.3 本研究のアプローチ

本研究では、インタフェースの不一致については、起きてしまったインタフェースの不一致を検出するのではなく、開発手順の誤りが起こる確率を下げることで、これを防ぐアプローチを取る。具体的には、IDL で記述された定義を格納できるインタフェースリポジトリを利用して、プラットフォーム間でインタフェースの仕様を共有し、インタフェースリポジトリを元にスタブとスケルトンを生成するツールを提供することで、開発手順の誤りが生じる確率を下げる。

開発手順の簡略化は、既存のプログラミング言語の持つリフレクションの能力を利用して実現する。CORBA のアプリケーション開発の可能なプログラミング言語のうち、リフレクションの可能なプログラミング言語については、アプリケーションの実行時に必要なスタブとスケルトンを自動的に生成して取り込む機構を CORBA の実行時ライブラリの一部として提供できる。この自動生成の際に、やはりインタフェースリポジトリを利用することで、インタフェースの不一致を防ぐこともできる。本研究のアプローチの概観を図 2.12 に示す。以降の章ではこのアプローチの詳細を述べる。

第 3章

インタフェースリポジトリを利用した開発環境

本章では前章で述べたインタフェースの不整合を防ぐ一つの方法として、CORBAで定められているインタフェースリポジトリを利用した、アプリケーション開発環境を提案する。

インタフェースリポジトリの現状の規格と既存の実装は、アプリケーション開発に用いることについてほとんど考慮されていない。本章では、現状のインタフェースリポジトリについて概説するとともに、アプリケーション開発に用いるために必要なインタフェースリポジトリの拡張とツールについて議論する。

3.1 インタフェースリポジトリの概要

インタフェースリポジトリは IDL で記述された定義を格納し、各定義を分散オブジェクトとして提供する CORBA のサーバであり、主に CORBA のアプリケーションから IDL による定義を実行時に参照するために用いられる。インタフェースリポジトリの仕様は CORBA の規格に含まれており、ほとんどの CORBA の実装がインタフェースリポジトリの実装と、IDL ファイルの内容をインタフェースリポジトリに格納可能な IDL コンパイラを提供している。前章に示した図 2.1 の IDL ファイルを格納した状態のインタフェースリポジトリを図 3.1 に示す。

CORBA の実行時ライブラリによって、すべての分散オブジェクトが暗黙のう

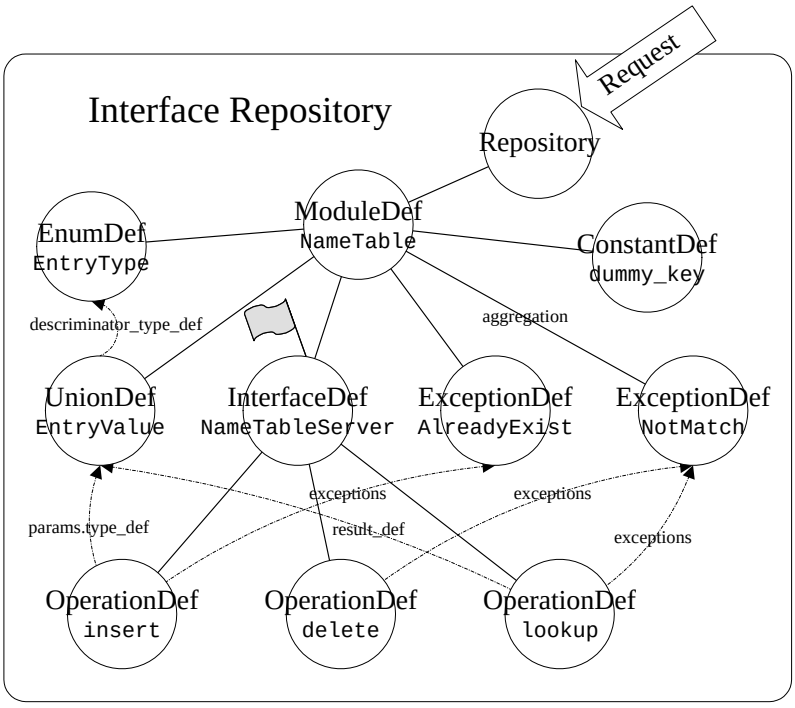

図 3.1: インタフェースリポジトリの構成

ちに実装する `CORBA::Object` インタフェースには、分散オブジェクトのインタフェースの定義を取り出す `_get_interface` というオペレーションが含まれている。このオペレーションは、サーバがインタフェースリポジトリの位置を知っている場合には、その分散オブジェクトのインタフェースの定義を格納している分散オブジェクトのオブジェクトリファレンスを返す。たとえば、前章に示したクライアント (図 2.4) の取得したオブジェクトリファレンスに対して、`_get_interface` は図中に旗で示した分散オブジェクトのオブジェクトリファレンスを返す。

このオブジェクトリファレンスを起点に、インタフェースリポジトリ内の分散オブジェクト間の関連をたどることで、分散オブジェクトの操作に必要な情報は一通り取得できる。取得した情報は、2.2.4 節で述べた DII や DynAny API を用いて、特定のインタフェースに依存しない汎用のデバッグツールやテストツールを実装に用いられるのが一般的である。

分散オブジェクトの `_get_interface` を介さずに、`Repository` オブジェクトの提供するオペレーションを利用して定義を取り出すこともできる。`Repository`

オブジェクトのオブジェクトリファレンスは、NamingService と同様に ORB オブジェクトの resolve_initial_reference メソッドで取得できる。

Repository オブジェクトのオペレーションで定義を取り出すには、定義の絶対スコープ名かリポジトリ ID が必要である。CORBA の規格は、どちらを用いた場合でもインタフェースリポジトリ内で定義を一意に特定できることを保証している。絶対スコープ名で取り出す場合には lookup オペレーションを、リポジトリ ID の場合には lookup_id オペレーションを利用する。たとえば、NameTableServer インタフェースの定義を取得する際には、以下のいずれかが必要になる。

絶対スコープ名　::NameTable::NameTableServer
リポジトリ ID　IDL:NameTable/NameTableServer:1.0

リポジトリ ID の最後のフィールドはバージョン番号である。図 2.11 のように IDL ファイルでバージョン番号を指定した場合には、NameTableServer のリポジトリ ID は “IDL:NameTable/NameTableServer:1.1” となる。ただし、同じ定義の異なるバージョンを一つのインタフェースリポジトリに格納することはできないので、絶対スコープ名だけでも一意に定義を特定できる。

3.2 インタフェースリポジトリを利用したアプリケーション開発

前述したように、インタフェースリポジトリは CORBA のアプリケーションの実行時に用いられるサーバだが、アプリケーションの開発時に利用することで、クロスプラットフォームでの開発時にインタフェースの不一致を起こす原因となる開発手順の誤りを防ぐことができる。

開発プラットフォーム間でファイル共有を実現するのが難しい場合でも、インタフェースリポジトリを共有することは難しくはない。インタフェースリポジトリを操作するツールは CORBA のクライアントであり、サーバとクライアントの運用プラットフォーム間の相違は CORBA の実装が隠蔽するからである。開発プラットフォーム間にファイアウォール [31] が存在する場合には、ファイアウォールを介した CORBA のアプリケーション間の通信をサポートする製品 [32, 33] を利

用することで、安全にファイアウォールを越えさせることもできる。

開発プラットフォーム間で IDL ファイルの代わりにインタフェースリポジトリを共有し、スタブとスケルトンをインタフェースリポジトリから直接生成できれば、インタフェースの不一致の原因となる開発手順の誤りが生じる確率を下げることができるはずだ。しかし、既存の CORBA の実装は、インタフェースリポジトリをアプリケーション開発に利用することを想定していないため、インタフェースリポジトリを利用するツールが整備されていない。また、CORBA の規格が定めているインタフェースリポジトリの仕様にも、アプリケーション開発に利用する際には不便な点がある。

- インタフェースリポジトリを利用してスタブとスケルトンを生成できない

  ほとんどの実装が IDL ファイルをインタフェースリポジトリに格納するツールを提供しているが、逆にインタフェースリポジトリから IDL ファイルを再生するツールや、インタフェースリポジトリから直接スタブとスケルトンを生成するツールを提供している実装はまれである。

- インタフェースリポジトリ内の定義の更新日時を記録できない

  CORBA の規格で定められているインタフェースリポジトリでは、定義が更新された日時を記録することができない。そのため、インタフェースリポジトリ内の定義が更新されたときに、対応するスタブあるいはスケルトンのみを再生成するツールを実装することができない。

- インタフェースリポジトリ内の定義を上書きできない

  一部の実装では、IDL ファイルを構文解析する際に、インタフェースリポジトリ内の分散オブジェクトを直接構文木として用いることで、インタフェースリポジトリに定義を格納するツールを実現している。そのため、インタフェースリポジトリ内に以前の定義が残っていると、同一名の重複定義としてエラーになってしまう．さらに、IDL ファイルに構文誤りがある場合には、インタフェースリポジトリが一貫性の取れていない状態で放置されることになる。

```
module CORBA {
  interface IRObject {
    readonly attribute DefinitionKind def_kind;
    attribute long last_modified;
    void destroy();
  }
}
```

図 3.2: IRObject インタフェースに更新日時を追加

3.3 インタフェースリポジトリと周辺ツールの改善

前節で述べた問題を解決できれば、インタフェースリポジトリを中心にアプリケーション開発を行うことは十分可能である。これらの問題は、CORBA の実装や利用するプログラミング言語にほぼ依存しない形で解決できる。

3.3.1 更新日時の記録可能なインタフェースリポジトリ

インタフェースリポジトリ内の分散オブジェクトのほとんどが継承している `Contained` インタフェースには、バージョン番号を記録する属性が含まれている。これは IDL ファイルで `#pragma version` で指定されたバージョン番号を格納するためのものであり、定義の更新を記録するために利用することはできない。そこで、更新日時を記録できるようにするために、インタフェースリポジトリの仕様を一部変更する。

IDL の定義を格納する分散オブジェクトはすべて、`IRObject` インタフェースを継承している。この `IRObject` インタフェースに図 3.2 のように更新日時を格納する属性 `last_modified` を追加する。値は、C 言語の `time` ライブラリ関数で用いられている、グリニッジ標準時の 1970 年 1 月 1 日 0 時から経過した秒数を用いる。この属性の追加によって、既存のインタフェースを想定しているクライアントとインタフェースリポジトリの間でインタフェースの不一致が起こることはない。

このインタフェースの修正に対応するインタフェースリポジトリの実装の修正はごくわずかである。インタフェースリポジトリが C++ で実装されているなら、

IRObject インタフェースを実装しているクラスのコンストラクタに、time() 関数で得られる数値を初期値として設定する操作を追加し、この属性を読み書きする accessor メソッドを追加するだけである。

3.3.2 IDL ファイルを格納するツール

前述したように、IDL ファイルをインタフェースリポジトリに格納するツールには、既存の定義を上書きできない問題がある。そこで、定義を上書きと更新日時の変更が可能な同様のツールを別途用意する。

このツールは、IDL ファイルを構文解析する際に一度ツール内に構文木を作成して、構文誤りがない場合のみインタフェースリポジトリ側に転送する形で実装できる。構文木の内容をインタフェースリポジトリに転送する際には、定義が変更されているかを比較して、変更されている定義だけを上書きした上で更新日時を記録する。

定義が変更されているかは、モジュール、インタフェース、オペレーション以外については、同じ絶対名を持つオブジェクトの type 属性に格納された TypeCode を比較するだけで確認できる。オペレーションについては、引数の名前、オペレーションの返値、引数や例外として用いられる型、oneway の有無を比較する。インタフェースとモジュールについては、その中で行われている定義の追加、削除、更新があった場合に更新日時を書きかえる。インタフェースについては、継承しているインタフェースの追加、削除、変更があった場合にも更新日時を書きかえる必要がある。

ところで、モジュールの定義は複数の IDL ファイルにまたがることが許されていて、同じモジュール内でも関連のないインタフェースは別のファイルに分割するのが一般的である。各ファイルについて別々に処理を行うと、インタフェースリポジトリに残っているモジュール内の定義を安全に削除することができない。したがって、このツールを利用する際には、アプリケーション開発に用いられる IDL ファイルのうち、同じモジュール内の定義を含むものは、同時にこのツールで処理しなければならない。

3.3.3　インタフェースリポジトリを利用する IDL コンパイラ

インタフェースリポジトリからスタブとスケルトンを直接生成することができる IDL コンパイラは、ほとんどの実装で提供されていない。そこで、先に拡張したインタフェースリポジトリを利用して、スタブとスケルトンを生成するツールについて述べる。

生成するスタブとスケルトンの指定方法

従来の IDL コンパイラでは、IDL ファイルを指定することでスタブとスケルトンの生成が行われている。しかし、インタフェースリポジトリを利用する場合には、IDL ファイルという概念を利用できないので別の指定方法が必要である。インタフェースリポジトリからスタブとスケルトンを直接生成できるツールを提供している既存の実装 [12, 8] では、モジュールのリポジトリ ID を指定してモジュール内の定義をすべて生成するという方法が取られている。しかし、一つのモジュールに非常に多くの定義が格納されている場合には、アプリケーションが利用しないスタブとスケルトンが生成されることがある。

そこで、生成ツールに与える情報として、アプリケーションが利用するインタフェースの絶対スコープ名かリポジトリ ID も指定可能にする。前述したように、インタフェースの絶対スコープ名が与えられれば、インタフェースリポジトリからインタフェースの定義と、そのインタフェースを利用する際に必要な定義を取り出すことができるので、必要なスタブとスケルトンを生成するためのヒントとしては十分である。

アプリケーションがまったく依存関係のないインタフェースを複数利用する場合には、絶対スコープ名かリポジトリ ID を列挙する必要がある。インタフェースの定義を持つ分散オブジェクトには、継承しているインタフェースを示す属性は存在するが、逆にそのインタフェースを継承しているインタフェースのリストは存在しないので (図 3.3)、この方向の依存関係をあてにすることはできない。生成ツールにはアプリケーションの利用するインタフェースのうち一番下位のインタフェースを指定する必要がある。実際には、継承以外の依存関係から下位インタフェースをたどることができる場合も多い。たとえば、図 3.3 のインタフェース

ModuleDef、ConstantDef、InterfaceDef を利用するアプリケーションは、絶対スコープ名 "::Repository" を指定するだけでよい。

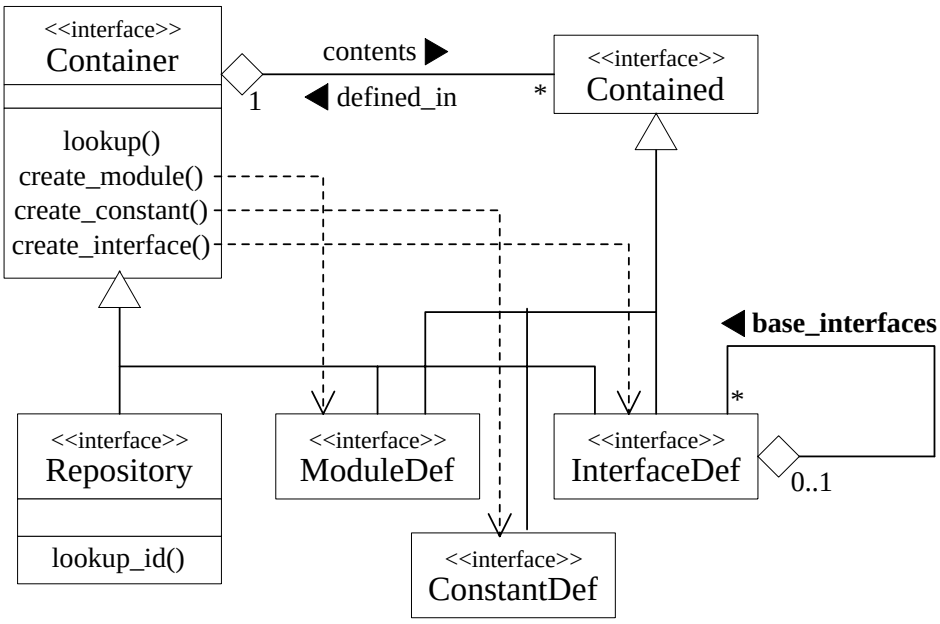

図 3.3: インタフェースリポジトリを構成するインタフェースの関連

定数の定義だけは、型が合っていれば任意のインタフェースで利用できるので、あるインタフェースの操作にどの定数の定義が必要かを正確に決定することはできない。実際には、まったく関連のないモジュールやインタフェースで定義されている定数が用いられることはないので、関連するモジュールやインタフェースで定義をされている定数をすべて必要であると見なすことで、この問題を回避できる。

スタブとスケルトンの生成

スタブとスケルトンとして生成される内容やファイルの名前は、CORBA の実装やプログラミング言語によって大きく異なっている。Java の場合には "Java ORB Portability Interface"[34] として、IDL コンパイラの生成するスタブとスケルトンの内容とファイル名が規定されているので実装間の差は小さくなるはずだが、この規格は最近策定されたため対応している実装は少ない。インタフェースリポジトリから真に直接スタブとスケルトンを生成するよりも、一時的に IDL ファイルを

生成して既存の IDL コンパイラをツールから起動する方が、さまざまな実装やプログラミング言語に対応できる点で有利である。

IDL ファイルをどのように生成するべきかは、開発に用いる CORBA の実装やプログラミング言語によって異なる。コンパイルの必要なプログラミング言語では、必要な定義を一つの IDL ファイルにすべて格納するアプローチは望ましいものではない。ごく一部の定義が更新された場合でも、IDL コンパイラはその IDL ファイルから生成されるスタブとスケルトンをすべて更新し、make のようなファイルの更新日時に基づく構築支援ツールは、更新されたスタブとスケルトンとそれに依存するソースコードを再コンパイルするためターンアラウンドが非常に長くなる。特に C++ では非常に大きなヘッダファイルが生成され、ヘッダファイルは分散オブジェクトを利用あるいは実装するすべてのソースコードに取り込まれるため、各ソースコードのコンパイル時間が長くなる。そこで、もう少し粒度の細かい生成方法を採ることにする。

IDL ファイルを生成する際には、モジュール名を持つディレクトリを作成して、インタフェース名を持つファイルに、インタフェースの定義とそのスコープで行われている定義を格納する。インタフェースに属さない定義が存在する場合には、適当な名前 (たとえば `Common.idl`) を持つファイルに格納する。他の IDL ファイルの定義を参照している IDL ファイルには、`#include` 指令で IDL ファイルを取り込む記述を挿入しておく必要がある。生成される IDL ファイルの粒度を小さくすれば、インタフェースリポジトリ内の定義の更新により細かく追随できる。しかし、IDL の文法上の制約からインタフェースよりも細かい単位でファイルを分割することはできない。また、生成される IDL ファイルの数を必要以上に増やさないために、インタフェースに属さない定義はモジュール単位でまとめるべきである。

IDL ファイルに定義を生成したら、その定義の名前と更新日時を特別なファイルに記録しておいて、次回に起動されたときには、記録された更新日時とインタフェースリポジトリ内の定義の持つ `last_modified` 属性の値を比較して、更新された定義の格納されたファイルのみを再生成する。

IDL ファイルを生成したら、IDL コンパイラを起動してスタブとスケルトンを生成する。生成した IDL ファイルが `#include` によって他の IDL ファイルを取り

込んでいる場合には、`#include` 指令で取り込んだ定義に対応するスタブやスケルトンの生成を抑制する必要がある。さもないと、更新されていない IDL ファイルに対応するスタブとスケルトンが上書きされる可能性があり、IDL ファイルを細かく分割した意味が失われてしまう。幸いほとんどすべての実装の IDL コンパイラで、起動時のオプションで抑制できるので常にこのオプションを指定する。

3.4 実装と運用に関する考察

この環境は既存の CORBA の実装のソースコードを利用することで、容易に実装できる。インタフェースリポジトリおよび周辺ツールのソースコードを公開している実装は多く [35, 8, 36, 9, 12]、いずれも UNIX 互換のさまざまな OS と Microsoft Windows に対応している。現在の実装は、Python に対応した実装である Fnorb [12] を利用している。

更新日時を記録可能にするためのインタフェースリポジトリの実装の修正はごくわずかであり、どの実装を用いても非常に容易に実現できる。

IDL ファイルをインタフェースリポジトリに格納するツールを実装する際には、インタフェースリポジトリ内の分散オブジェクトの実装を構文木として利用している IDL コンパイラを元にすることで、変更箇所の検査を容易に実現できる。このような実装になっているのは、Fnorb と ORBacus [36] だけである。このツールの実装には、インタフェースの依存関係の追跡などや更新日時の検査などの処理が必要であり、既存の実装に対する追加・変更箇所はやや多くなる。

この環境は、開発プラットフォームや開発に用いるプログラミング言語や CORBA の実装にはほとんど依存しない。完全に統合された開発環境上でしか利用できない CORBA の実装 [19] を除けば、どのような組み合わせで開発を行う場合でも、この環境を利用したアプリケーション開発は可能である。

インタフェースリポジトリは、開発/運用プラットフォームのいずれかで運用できればよい。IDL ファイルをインタフェースリポジトリに格納するツールと、インタフェースリポジトリからスタブとスケルトンを生成するツールは、すべての開発プラットフォームで運用可能でなければならない。現在の Fnorb を利用した実装は、Python のインタープリタを実行できるプラットフォームであれば利用で

きる。格納ツールは開発に用いるプログラミング言語やCORBAの実装にはまったく依存しないが、生成ツールには起動するIDLコンパイラの名前などの依存する要素がある。依存する要素はパラメータとして生成ツールに与える。

開発手順については、IDLファイルをインタフェースリポジトリに格納する必要があるため手順は一つ増える。IDLによる定義とスタブとスケルトンの間の依存関係の追跡とスタブとスケルトン再生成は、インタフェースリポジトリからスタブとスケルトンを生成するツールが行う。これによりインタフェースの不整合が生じる確率を下げることができるだけでなく、開発手順の煩雑さもいくらか軽減される。

以降の章では、この環境を前提とした上で、インタフェースリポジトリからスタブとスケルトンを生成して、アプリケーションに組み込むまでの開発手順を自動化する手法を示す。

第 4章

リフレクションを利用した開発手順の簡略化

リフレクションの可能なプログラミング言語を CORBA のアプリケーション開発に利用する場合には、通常のプログラミング言語を利用する際に必要となる開発手順を、開発者の手を煩わせることなくアプリケーションの実行時に自動的に実行することが可能になる。本章では、まずリフレクションの概要を述べ、次にこの手法の実現方法を述べると共に、その際に必要となるリフレクションの能力を分析する。

4.1 リフレクションの概要

4.1.1 リフレクションとは

計算システムが、自分自身の構成や計算過程に関する計算を行うことを**リフレクション**という [22]。リフレクションは柔軟性と拡張性の高いシステムを構成する技術として、オペレーティングシステム [37]、ウィンドウシステム [38]、CSCW ツールキット [39] など、さまざまなシステムに応用されている。

リフレクションのもっとも一般的な利用方法は、記述力を高めるためにプログラミング言語に導入することである。プログラミング言語にリフレクションを導入する際には、まずプログラムの構成要素やプログラムの実行環境の構成要素を

プログラミング言語自身で表現する。そして、その表現にアクセスする方法をプログラムに提供した上で、その表現が変更された場合にはプログラム自身にその結果が反映されるようにする。その結果、プログラムが「自分自身の構成や計算過程に関する計算」を行うことが可能になる。

4.1.2　リフレクションによって得られる能力の分類

リフレクションを導入することで、大きく分けて以下の二つの能力をプログラミング言語に与えることができる。

- **Linguistic リフレクション**

  プログラム自身を**参照**、**改変**するプログラムや、プログラムを**生成**してプログラム自身に追加するプログラムを記述できる。

- **Behavioral リフレクション**

  プログラムの実行環境を**参照**、**改変**するプログラムを記述できる。すなわち、実行環境の持つ情報を参照したり、プログラムの実行機構を利用したり、それらを改変するプログラムを記述できる。

プログラムも実行環境が持つ情報の一部であり、Linguistic リフレクションはこれを参照、改変能力なので、Linguistic リフレクションは Behaviral リフレクションによって得られる能力の一つでもある。

プログラミング言語の持つリフレクションの能力を二種類に分類する考え方は、Graham Kirby らによる [40] に基づくものである。彼らは、プログラムを生成してプログラム自身に追加する能力だけを Linguistic リフレクションに分類し、他の能力をすべて Behavioral リフレクションに分類している。

プログラムを生成して追加する能力のみを別に分類しているのは、この能力が実行環境の持つ情報の参照、改変なしに実現できるからである。プログラムから実行環境の持つ評価器を利用できるようにする、つまり、Lisp [41] の `eval` 関数に相当するものを用意するだけで実現できる。彼らはこの能力の応用を中心に議論しているので、この分類には意味があるが、リフレクションの能力全体を議論するには荒すぎる。

本論文では、実行環境の持つプログラムの字面上の情報の参照と改変および生成を Linguistic リフレクションに、実行環境が持つプログラムの字面以外の情報の参照と改変、および実行機構の利用と改変を Behavioral リフレクションに分類する。この分類により、たとえば Java の Reflection API [42] が提供する能力を

> Linguistic と Behavioral の両方について参照のみを提供する。Behavioral については、メソッドの実行機構とインスタンスの生成機構の利用が可能である

と分析することが可能になる。

4.2 既存のリフレクションの可能な言語の持つ能力

Java の Reflection API の例にもあるように、リフレクションの可能なすべての言語がこの分類のすべての能力を提供しているわけではない。また、各能力の幅についてもそれぞれである。プログラミング言語の持つリフレクションの能力は、リフレクションを導入する際に、プログラムの構成要素や実行環境の構成要素をどの程度詳細にその言語自身で表現するか、そしてその表現の改変をどこまでプログラムに許すかによって決まる。

4.2.1 リフレクションの実装方法と得られる能力の関係

Common Lisp [43] では関数定義をリスト型のデータとして作成し、`eval` 関数に与えることでプログラムの生成するプログラムを記述できる。一度定義された関数は `function` 型の値となるが、その値の内容は `function-lambda-expression` により参照できる。既存の関数を定義し直すことで、その内容を改変することもできる。したがって、すべての Linguistic リフレクションが提供されていると言える。

また、Common Lisp では言語の実行機構を `read`、`eval`、`print` の 3 つの関数として参照できる。フック変数を通じてこれらの関数の詳細な振る舞いを改変することもできる。たとえば `eval` については、フック変数 `*evalhook*` や `*applyhook*` を通じて、各フォームを評価するときの振る舞いや、関数を適用するときの振る

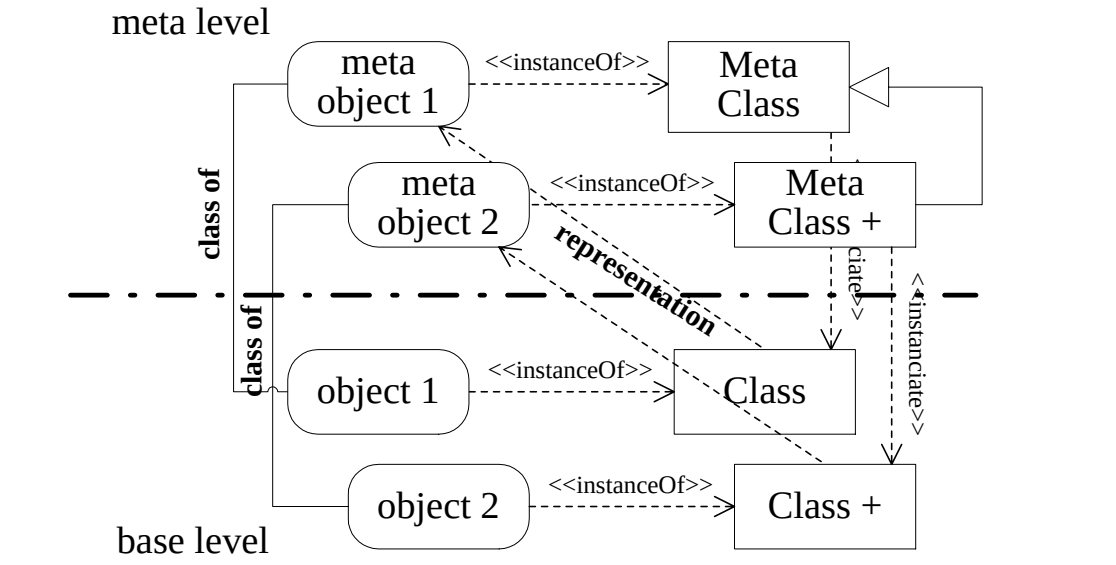

図 4.1: メタクラスとメタオブジェクト

舞いを改変できる。したがって、すべての Behavioral リフレクションが提供されていると言える。

Common Lisp のためのオブジェクトシステムである CLOS [44] では、クラスやメソッドが CLOS 自身のオブジェクトとして表現されている。一般に、リフレクションの可能なオブジェクト指向言語において、プログラムの構成要素をあらわすオブジェクトを**メタオブジェクト**と呼び、そのクラスを**メタクラス**と呼ぶ (図 4.1)。

CLOS では、インスタンスに `class-of` 関数を適用する、クラス名やメソッド名から直接取り出すなどの手段で、プログラムからメタオブジェクトを参照できる。さらに、メタオブジェクトの属性にアクセスすることで、クラスやメソッドの詳細な定義について参照と改変が可能である。メタクラスのインスタンスを生成することでプログラムの生成もできる。したがって、すべての Linguistic リフレクションが提供されているといえる。

CLOS では、インスタンスの生成、メソッドの実行、属性へのアクセス、クラス継承など、オブジェクトシステムの持つさまざまな振る舞いが、メタクラスのメソッドとしてプログラムに公開されている。また、メタクラスのサブクラスを定義する形で、それらを安全に改変することもできる。したがって、すべての Behavioral リフレクションが提供されているといえる。このプログラムに公開されているメ

タクラスのメソッドのことを**メタオブジェクトプロトコル**と呼ぶ [45]。

4.2.2 Linguistic リフレクションの実際

プログラムにプログラム自身の広範な改変や生成を許すと、実行環境がプログラムの実行効率を向上させるために、プログラムについて何らかの仮定を置くことがことが難しくなる。プログラミング言語に Linguistic リフレクションを導入する際には、参照については広い範囲で可能にしても、改変や生成についてはある程度制限するのが一般的である。

たとえば、一度生成したクラス定義の改変は一切許さないとか、メソッドの追加のみが可能といった具合である。Python ではクラスが Python のオブジェクトとして表現されており、その属性を改変することでクラス定義の改変が可能だが、継承しているクラスのリストを保持する属性の改変を禁止しているので、実行時にクラス継承の階層を改変することはできない。Dylan [46] では実行時にメソッドの追加や削除が可能だが、あらかじめ追加や削除を行わない領域を開発者がプログラム中で宣言することで、実行環境に最適化の機会を与えることが可能になっている。

一般にメソッドの定義に関する Linguistic リフレクションは、かなり制限されている。たとえば、既存のメソッドのシグネチャ[1]は参照できても改変は許さない、メソッドボディ[2]の詳細については参照も改変も許さないといった具合である。Dylan はメソッドの生成を制限しており、まったく新規にメソッドを生成することができない代わりに、関数合成やカリー化関数を用いた既存のメソッドの組み合わせによる生成が可能になっている。

4.2.3 Behavioral リフレクションの実際

Behavioral リフレクションにより、プログラムの実行環境の持つほとんどすべての情報と振る舞いの参照と改変が可能な例は CLOS のみである。他の言語ではプログラムから参照可能な実行環境の範囲は限られており、改変可能な範囲はさら

[1]引数や返値の仕様
[2]メソッドで行う処理を記述したプログラム

に狭くなる。

実行環境の持つ情報のうち、リフレクションの可能なほとんどすべてのオブジェクト指向言語で参照可能なものとして、インスタンスとそれを生成したクラスの関係がある。この関係の改変が可能なのは CLOS のみであり、一般には改変は許されない。

デバッガをその言語のクラスライブラリとして提供できるようにするために、プログラムの実行時の名前空間やスタックフレームの情報を、その言語のオブジェクトとして表現している言語もある [5, 47]。これらのオブジェクトの参照と改変により、プログラムの字面上の情報の操作だけでは実現できない、名前空間の参照や改変が可能になる。

プログラムの実行機構に関するリフレクションとして、多くの言語がメソッドの実行機構の利用と改変を可能にしている。そのような言語では、任意のクラスの任意の名前のメソッドを呼び出し可能なプログラムや、メソッドの移譲の機能をメソッド呼び出しの機構に追加するプログラムを記述できる。

インスタンスの生成機構の利用と改変の可能な言語も多い。インスタンスの生成機構を利用すると、任意のクラスのインスタンスを生成可能なプログラムを記述できる。C++ や Java ではコンストラクタで実現されているインスタンスの初期化を、メタクラスの持つインスタンスの生成機構を改変して行う言語もある [44, 48]。

名前解決や名前空間の移入など、実行環境の持つ実行時の名前空間に関する情報や振る舞いの参照や改変が可能な言語は多い [5, 49, 50, 47]。Java では、アプリケーションが未定義のクラスを参照したときに、クラス定義をファイルから取り込む機構の利用と改変が可能である。Web ブラウザ上で動作する Java アプレットは、クラス定義をファイルではなくネットワーク経由で取り込むように、この機構を改変することで実現されている。Python では、クラスを定義する際に特別なメソッドを定義することで、未定義のメソッドが呼び出されたときの実行環境の振る舞いを改変できる。

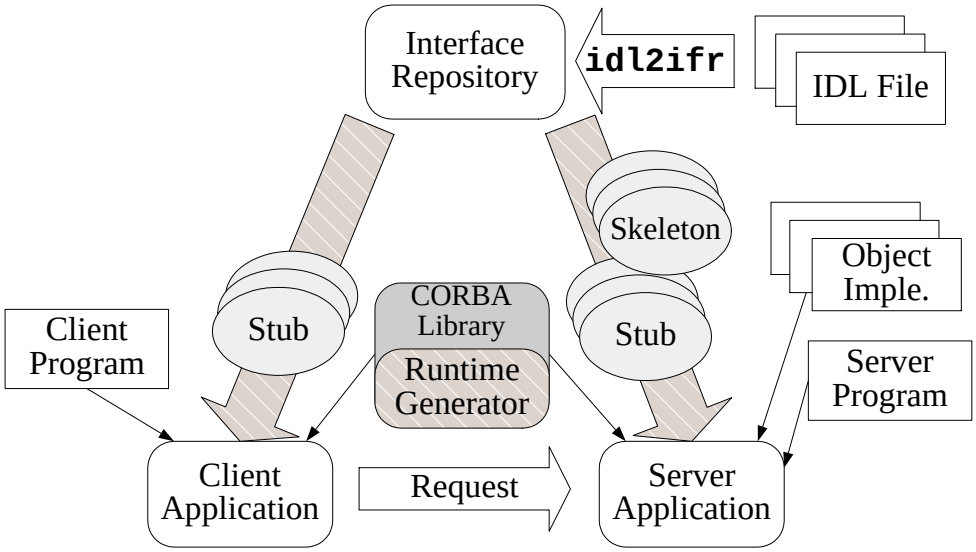

図 4.2: スタブとスケルトンの実行時自動生成を用いた開発

4.3 リフレクションによるスタブとスケルトンの実行時自動生成

リフレクションによって提供される能力は言語によってまちまちである。少なくとも、Linguistic リフレクションによる新たなプログラムの生成と追加が可能であれば、アプリケーションに必要なスタブとスケルトンを実行時に生成して追加するプログラムを、CORBA の実行時ライブラリの一部として提供できる。これが可能になれば、開発者が扱うファイル数と開発手順が減るので、CORBA のアプリケーション開発における開発手順の煩雑さを軽減できる。さらに、スタブとスケルトンの生成に必要な情報をインタフェースリポジトリから獲得することで、クラスプラットフォームの開発時にインタフェースの不整合を防ぐこともできる (図 4.2)。

この開発スタイルでは、実行時にスタブとスケルトンを生成するために、アプリケーションの性能低下が避けられないように見える。実際には、最初に必要なスタブとスケルトンを生成して取り込むためのコストがかかるだけであり、その後は通常の開発スタイルで開発したアプリケーションと同様に振る舞うので、アプリケーションの性能には影響しない。一度生成したスタブとスケルトンをキャッ

シュすることで、次に生成するときには、追加あるいは更新された定義に対応するスタブとスケルトンを生成するだけでよい。定義の更新は 3 章で述べたインタフェースリポジトリを用いることで確認できる。

以降の節では、特定の言語を仮定せずにリフレクションの可能なオブジェクト指向言語の全般を想定して、この環境の実現方法を述べると共に、この環境を実現するために実際に必要となるリフレクションの能力を明らかにする。

4.4 スタブとスケルトンの実行時生成

まず最初に、スタブとスケルトンをリフレクションを利用して生成する手法と、その際に必要となるリフレクションの能力について議論する。

4.4.1 Linguistic リフレクションによるスタブとスケルトンの生成

型やクラスの定義の提供

スタブとスケルトンに含まれている IDL による定義に対応する型やクラスの定義をアプリケーションプログラムに与えるには、インタフェースリポジトリから取得した定義を元に、新たに型やクラスの定義を生成してアプリケーションプログラムに追加する Linguistic リフレクションの能力が必要である。

クラスのメソッドを任意の時点で追加できる言語では、必ずしも、スタブクラスを生成する際にメソッドの定義をすべて生成する必要はない。Behavioral リフレクションにより、メソッドの実行機構を改変できる言語であれば、実際にメソッドが呼ばれた時点で必要なメソッドの定義を提供することができる。この手法により、無駄なメソッドが生成されるのを防ぐことができるので、実行時にスタブクラスを生成する際のコストを下げることができる。

通信プロトコルの処理

スタブとスケルトンの持つ役割の一つである通信プロトコルの処理は、スタブクラスやスケルトンクラスのメソッドボディに記述されている。これを実行時に生

成するには、Linguistic リフレクションにより任意のメソッドボディを生成できる必要がある。その際には、文字列などのデータ型で記述したソースコードを、Lisp の `eval` に相当するインタフェースを用いて実行環境におけるメソッドボディの表現に変換するか、それが公開されているのならメソッドボディの内部表現を直接作成する必要がある。

特定の実装への依存度

生成するクラスや型の定義と関数あるいはメソッドのシグネチャは、CORBA の言語マッピングで規定されているので、それに従って生成することで CORBA の特定の実装に依存することは避けられる。IDL コンパイラの生成する関数やメソッドのボディは CORBA の実装に依存する形で最適化されているので、同じものを生成しようとすると特定の実装への依存が避けられない。

通信プロトコルの処理を、実装に依存しない CORBA の API である Dynamic Invocation Interface、Dynamic Skeleton Interface や DynAny API を利用して行うプログラムを生成することで、メソッドボディが特定の実装に依存することは避けることができる。ただし、前述したようにその代償としてアプリケーションの性能が低下することになる。

4.4.2 リフレクションを用いた通信プロトコルの処理の実装

通信プロトコルの処理については、Linguistic リフレクションでそれを行うメソッドボディを生成する代わりに、Linguistic の参照と Behavioral の参照の能力を利用して、CORBA の実行時ライブラリの一部として提供することもできる。メソッドボディの生成に制約がある言語でスタブとスケルトンの自動生成を実現する場合には、この手法が必要になる。

メソッド呼び出しからリクエストプロトコルへの変換

メソッド呼び出しをリクエストプロトコルに変換する処理は、

1. Behavioral リフレクションにより取得した呼び出されたメソッドの名前と引

数の値

2. インタフェースリポジトリから得たオペレーションの定義

を利用することで、特定のインタフェースの定義に依存しない単一のプログラムとして記述できるので、ライブラリとしてあらかじめ用意しておくことができる。

Behavioral リフレクションによるメソッドの実行機構の改変が可能なら、プログラムがメソッドを実行したときに本物のメソッドを実行する代わりに、呼び出されたメソッドの名前や引数の値などの情報をこのプログラムに渡すことでメソッドボディの生成はまったく不要になる。

リクエストプロトコルからメソッド呼び出しへの変換

リクエストプロトコルをメソッド呼び出しに変換するプログラムは、

1. リクエストプロトコルに含まれる要求されたオペレーションの文字列
2. インタフェースリポジトリから得たオペレーションの定義

を元に、実行環境の持つメソッド呼び出しの機構を利用してメソッドを実行することで、特定のインタフェースやオペレーションに依存せずに記述できる。

ネットワークストリームとユーザ定義型の値の相互変換

ユーザ定義型の値とネットワークストリームの間の相互変換については、

1. インタフェースリポジトリから得たユーザ定義型の定義
2. Behavioral リフレクションにより取得した値の型あるいはクラスと、Linguistic リフレクションにより取得したその定義

を元に、特定のユーザ定義型に依存せずに相互変換を行うプログラムを記述できる。ネットワークストリームからユーザ定義型の値を取り出す際には、任意のクラスのインスタンスを生成するために、インスタンスの生成機構を利用する必要がある。

リフレクションによる実装の利点と制約

この方法には、メソッドボディの自由な生成が許されない言語において、スタブとスケルトンの実行時生成を実現できるという利点がある。ただし、リフレクションを利用して記述した通信プロトコルの処理を行うプログラムは、各オペレーションやユーザ定義型に依存して生成したものよりも性能は低くなる。

特定のオペレーションに依存せずに、リクエストプロトコルとメソッド呼び出しの間の相互変換を行うプログラムを記述するには、リフレクション以外の言語仕様もかなり選ばれる。これらのプログラムでは、任意のクラス、型、例外を扱う必要があるため、静的な型を持つプログラミング言語では記述がその難しくなる。もし、任意のメソッドボディの生成が許されているのなら、この処理はオペレーションやユーザ定義型の定義に依存した、専用のプログラムとして生成するべきである。

4.5 スタブとスケルトンの自動生成

アプリケーションに必要なスタブとスケルトンの内容を決定し、前節の手法を用いてスタブとスケルトンを生成してアプリケーションに追加するまでの一連の処理は、リフレクションを利用することで、開発者に特別な開発手順や特別なプログラミングスタイルを強いることなく実現できる。

4.5.1 必要なスタブとスケルトンの決定に用いるヒント

開発者による特別な指示なしをまったく必要とせずに、アプリケーションに必要なスタブとスケルトンを決定するためのヒントとしては以下の二つが考えられる。

- クライアントが取得したオブジェクトリファレンス
- アプリケーションプログラムの参照したクラスや型名

さらに、開発者による特別な指示なしに、CORBA の API の実行に必要なスタブを生成したり、逆に不要なスタブの生成を押さえたりするためのヒントとしては

- アプリケーションプログラムの呼び出したメソッドの名前

が必要である。

通常の CORBA のアプリケーション開発で、スタブやスケルトンを取り込むためにモジュールやパッケージを取り込む構文を用いる必要がある言語では、

- モジュールを取り込む構文に指定された名前

をヒントとして使うこともできる。この場合には、開発者が明示的に取り込むモジュールを指定する必要があるという点で、他のヒントを利用する場合と比べて開発者の負荷が大きくなる。

以下それぞれのヒントを元に一連の手順を実行する手法と、その際に必要なリフレクションの能力について述べる。

4.5.2 オブジェクトリファレンスを起点とするスタブ生成

分散オブジェクトを操作するためにクライアントが取得したオブジェクトリファレンスは、クライアントが必要とするスタブを決定する手段として用いることができる。クライアントがオブジェクトリファレンスを取得する方法は複数存在するが、どの方法も CORBA の実行時ライブラリに含まれている、ネットワークストリームからオブジェクトリファレンスを取り出す変換ルーチンを実行する。この変換ルーチンに手を入れることで、クライアントがオブジェクトリファレンスを取得した時点で、クライアントに必要なスタブを自動的に生成することが可能になる。

クライアントが取得したオブジェクトリファレンスに対して、`_get_interface` オペレーションを実行することで、インタフェースリポジトリからそのインタフェースの定義を取得できる。生成する必要があるスタブは、取得したインタフェースの定義と依存関係のあるすべての定義と、同じモジュールで定義されている定数に対応するものである。ただし、インターフェイスについては親インタフェースの定義以外は生成する必要はない。そのインタフェースのオブジェクトリファレンスを取得したときに、改めて生成すればよいからである。

オブジェクトリファレンスを取り出すルーチンで必要なスタブを生成した後、インターフェイスに対応するスタブクラスのインスタンスを生成してクライアントプログラムに返すことで、クライアントプログラムは続く分散オブジェクトの操

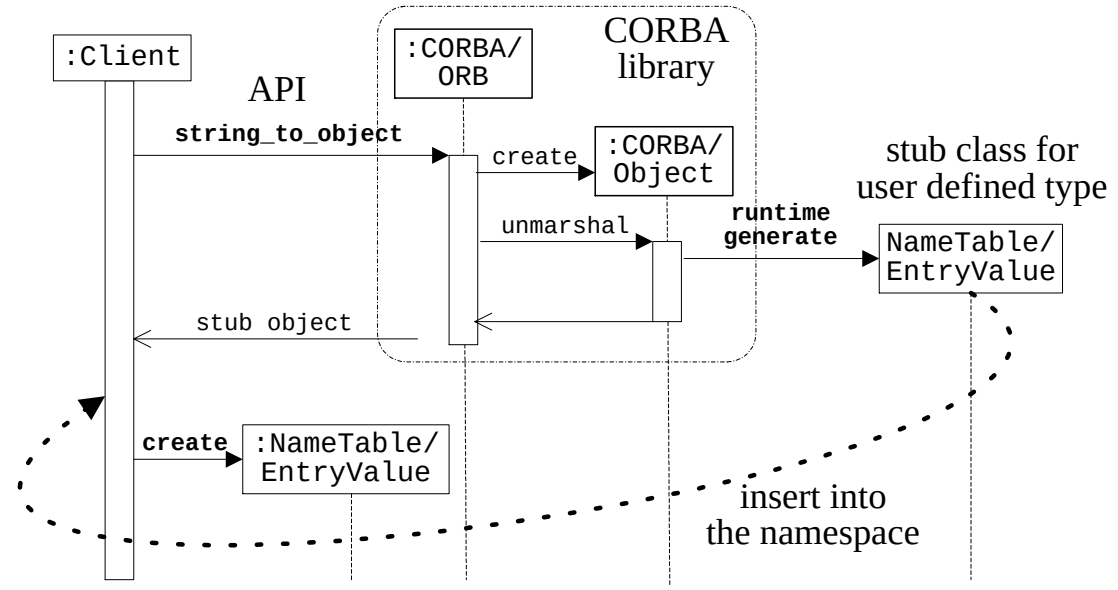

図 4.3: 名前空間をさかのぼってスタブを挿入する

作を行うことができる。生成したスタブのうち、ユーザ定義型および定数に対応するものは、ユーザ定義型の値を生成したり、定数を参照する際に、クライアントプログラムからその名前が参照されるので、クライアントプログラムの名前空間に挿入しておく必要がある。

プログラミング言語の仕様によっては、CORBA の実行時ライブラリから挿入するべき名前空間が見えない場合もある。たとえば、モジュールやパッケージなどの機構で、一つのアプリケーションの大域的な名前空間が分割されている場合がこれに相当する。その場合には、分散オブジェクトを取得する操作を実行した名前空間にさかのぼって挿入する必要がある (図 4.3)。この操作を実現するには、クライアントの名前空間を見つけるために、Behavioral リフレクションによりスタックフレームの内容を参照できる必要がある。

このクライアントの取得したオブジェクトリファレンスを利用する手法では、プログラムの実行時の振る舞いのみに基づいて、必要なスタブを決定して取り込むことができる。ただし、クライアントがスタブを取り込む際にしか利用できない、アプリケーションプログラムを記述する際に、記述スタイルに若干制限が生じるという欠点もある。この手法を利用する場合には、分散オブジェクトを操作するプログラムを記述する際に、ユーザ定義型や定数の名前を参照する前に、必ず分散

オブジェクトのオブジェクトリファレンスを取得するようにしなければならない。これに反したプログラムでは、定義されていない名前への参照が起こりエラーになってしまう。また、アプリケーションの実行前にすべての名前解決が行われるプログラミング言語では、この手法を実装することはできない。

4.5.3 未定義のクラス名を起点とするスタブとスケルトンの生成

Behavioral リフレクションにより、アプリケーションプログラムが未定義のクラスや型の名前を参照した際の実行環境の振る舞いを改変できるなら、CORBA の実行時ライブラリからその振る舞いを改変することで、未定義の名前に対応するスタブやスケルトンを生成してアプリケーションに渡すことができる。この手法は、未定義の場合だけでなく名前を解決する機構そのものの改変が可能な言語でも実現できる。以下、未定義の名前が参照されたときに行う処理の実装方法を述べる。

インタフェースリポジトリの検索

名前空間を階層化できるモジュールやパッケージの機構を持っている言語では、最上位のモジュールからその名前に至るまでのモジュールの名前を“::”で結合することで絶対スコープ名を生成できる。ただし、アプリケーションプログラムが相対的な名前でクラスを参照している場合や、モジュール名のショートカットを定義して短い名前で参照している場合には、Behavioral リフレクションにより、実行環境の持つ現在のモジュール名の情報やショートカットの情報を参照する必要がある。絶対スコープ名がわかれば、あとはインタフェースリポジトリの `lookup` オペレーションで対応する定義を取り出すだけである。

モジュールの機構を持たない言語では、モジュールの名前を“_”で結合することになっている。IDL によるモジュールやインタフェースの名前は“_”を含むことを許しているので、単に“_”を“::”に変換するだけでは必要な絶対スコープ名を得ることはできない。名前からインタフェースリポジトリ内の定義を取り出す際には、“_”で区切られた名前のどこまでがモジュール名かを確認しながらインタフェースリポジトリをたどっていくことで、その名前に対応する定義に到達できる。

スタブかスケルトンの生成

インタフェースリポジトリから未定義の名前に対応する定義が得られなかった場合には、それはプログラミングの誤りなのでエラーにする。定義が得られた場合には、それに対応するスタブかスケルトンを生成して、名前を参照した結果としてアプリケーションプログラムに返すことで、アプリケーションは分散オブジェクトに関する処理を継続できる。

プログラムがスケルトンクラスを参照するときには、一般に特殊な名前が用いられるので、それをインタフェースの名前に変換してインタフェースリポジトリを検索して、スケルトンを生成することができる。たとえば、図 2.1.3 の Java による分散オブジェクトの実装では、では `NameTableServer` インタフェースに対応するスケルトンクラスが`_NameTableServerImplBase` として参照されている。このような特殊な名前が用いられていない場合にはスタブを生成すればよい。

IDL の定数に関する処理

アプリケーションの参照した未定義のクラス名や型の名前に基づく生成では、IDL で定義された定数の値をアプリケーションに提供できない場合がある。Java の言語マッピング [34] では、定数もクラスに対応付けられるので特に問題は生じない。たとえば、図 2.1 の IDL ファイルにおける定数 `dummy_key` は `dummy_key` クラスに対応付けられ、`dummy_key.value` として値の参照が行われる。ところが、定数 `dummy_key` が文字列型の変数 `dummy_key` に対応付けられる言語もあり、この場合にはクラス名や型の名前に基づく生成では、定数の定義をアプリケーションに与えることはできない。

クラスや型だけでなく任意の未定義名を参照したときの振る舞いを Behavioral リフレクションで改変できるのなら、この場合にも対処できるが、そのような改変が行える言語はほとんどない。後述するモジュール名を起点とする生成方法が可能であれば、定数についてはそちらを経由して行うこともできる。どちらも不可能な場合には、定数に関する言語マッピングを変更して、単なる変数名以外のものに対応付けられるようにする必要がある。

4.5.4　未定義のメソッド名を起点とするスタブメソッドの生成

Linguistic リフレクションによりクラスのメソッドを任意の時点で追加できる言語で、かつ Behavioral リフレクションにより、未定義のメソッドをアプリケーションが呼び出した際の実行環境の振る舞いを改変できるなら、その時点で必要なメソッドの定義をスタブクラスに追加してアプリケーションの実行を継続することができる。

これが可能なら、スタブクラスを生成する際に、最初はスタブクラスの持つべきメソッドを生成せずに、実際にメソッドが呼び出された時点で生成することで、そのアプリケーションが利用しないメソッドの生成にかかるコストを削減できる。さらに、特定の API を実行するために必要な特別なコードを、アプリケーションが利用する時点で生成することも可能になる。

メソッドの生成方法

スタブクラスの未定義のメソッドを生成する際には、そのスタブクラスに対応するインタフェースかユーザ定義型の定義をインタフェースリポジトリから取り出す必要がある。そのためには、スタブクラスを生成する際に、リポジトリ ID か絶対スコープ名を記録しておく必要がある。実際には、ほとんどすべての実装において、通常の IDL コンパイラがスタブクラスの属性としてリポジトリ ID を生成している。実行時生成もそれに習うことでこの要件を満たすことができる。

スタブクラスに埋め込まれたリポジトリ ID がユーザ定義型の定義を指している場合には、呼び出されたメソッド名を元にユーザ定義型の値の操作に必要なメソッドか、ネットワークストリームとの相互変換に必要なメソッドを生成する。インターフェイスの定義を指している場合には、メソッド名を元にさらにインタフェースリポジトリを検索して、対応するオペレーションの定義を取り出し、その定義を元に必要なメソッドを生成すればよい。

動的生成により削減されるメソッド

この手法により、無駄なメソッドの生成を押さえることができる。まず明らかに、クライアントが利用しないオペレーションに対応するスタブメソッドは生成

```
   module NameTable {
      interface NameTableServer {
       enum EntryType { T_BOOL, T_INT, T_FLOAT, T_STRING };
       union EntryValue switch (EntryType) {
         ...
       };
       exception NotMatch {};
       exception AlreadyExist {
         EntryValue ent;
       };
       void insert(in string key, in EntryValue ent,
                   in boolean force) raises (AlreadyExist);
...
     };
```

図 4.4: インタフェースの中で型を定義する (nametable.idl)

しないですむ。これにより、非常に多くのオペレーションを持つインタフェースの一部をテストするクライアントを作成する際には、スタブの生成コストがかなり小さくなる。

ユーザ定義型がオペレーションの返値に利用されていないなら、クライアント側で用いられる対応するスタブクラスでは、ネットワークストリームから値への変換メソッドを、サーバ側ではその逆のメソッドを生成しないですむ。たとえば、図 4.4 の IDL ファイルでは、`EntryValue` 型はオペレーションの返値として用いられていないので、クライアント側では `EntryValueHelper` クラスの `read` メソッドが、サーバ側では `write` メソッドが不要になる。

また、図 4.4 では図 2.1 とは異なり、モジュールではなくインタフェースの中でユーザ定義型を定義している。アプリケーションがこれらのユーザ定義型だけを利用しており、インタフェースそのものは利用しない場合には、オペレーションに対応するスタブメソッドの生成がまったく不要になる。

動的生成による CORBA Messaging

未定義のクラス名やメソッド名が参照された時点で、それらに対応するプログラムを生成するアプローチを採ることで、2 章で述べた CORBA Messaging のため

の補助クラスやメソッドを、開発者による指示なしでアプリケーションが必要とする場合にのみ生成できる。

たとえば、CORBA Messaging の poll モデルをサポートする補助クラスは、`AMI_`*`interface`*`Poller` という名前で参照される。アプリケーションがこの形式のクラス名を参照したときには、そこからインタフェースの名前を取り出してインタフェースリポジトリを検索し、取得した定義を元に poll モデルの補助クラスを生成すればよい。poll モデルの呼び出しを実行するスタブメソッドは `sendp_`*`operation`* という名前で呼び出される。これもアプリケーションがこの形式のメソッド名を参照したときには、対応するオペレーションの定義を元に、poll モデルの呼び出しを実装するメソッドを生成すればよい。

4.5.5 モジュールを起点とするスタブとスケルトンの生成

一般に、IDL のモジュールと同等な名前空間を分割する機構を持つプログラミング言語では、モジュール内で定義された内容を取り込むための構文が用意されている。たとえば、Python や Java の `import` 文がこれに相当する。Behavioral リフレクションにより、モジュールを取り込む構文の振る舞いを改変できる言語では、モジュールを取り込む構文の中で指定されたモジュールに含まれるべきスタブやスケルトンを生成することで、アプリケーションに必要なスタブやスケルトンを取り込むことができる。

具体的には、指定された名前のモジュールがその言語の通常の検索方法では見つからない場合には、インタフェースリポジトリからその名前の IDL のモジュールを検索して、そのモジュールが含んでいる定義に対応するスタブやスケルトンを生成した後、通常のモジュール取り込みの手順を踏むことで、アプリケーションにスタブとスケルトンを取り込むことができる。

この方法では、取り込むスタブとスケルトンをモジュール単位で指定することしかできないので、無駄なスタブやスケルトンが生成されることがある。また、開発者が明示的に取り込むモジュールの名前を指定する必要がある点で、他の方法と比べて開発者の負荷が大きくなる。クラス名やメソッド名を利用した自動生成が可能な場合には、それらを併用するべきである。

4.6　実行時自動生成とリフレクションの能力の関係

本章では、リフレクションの可能なプログラミング言語をCORBAのアプリケーション開発に利用する際に、リフレクションによって提供される能力を利用して、**開発者に特別な開発手順や特別なプログラミングスタイルを強いることなく**、アプリケーションに必要なスタブやスケルトンを、実行時に自動的に生成して取り込む手法について述べてきた。

本章で述べた手法と必要とされるリフレクションの能力をまとめたものが表4.1である。メソッドボディの生成については、同じ処理を単一のプログラムで実装できるかもしれないので必須ではない。誰もメソッド定義を参照していないのは、ほとんど同じ情報がインタフェースリポジトリからオペレーションの定義として取得できるからである。

category	target	generate stub skelton	add stub method	implement common routine	use object reference	use undefined class	use undefined method	use import statement
linguistic generate	class & type	must			must	must		must
	medhod body	should	must		should	should	should	should
linguistic refer	class & type		must	must				
	method definition							
linguistic modfiy	add method		must				must	
behavioral refer	type-of or class-of			must				
	name space				must			
	stack frame				may be			
	generate instance			must				
	method invoke			must				
behavioral mododify	undefined class					must		
	undefined method			should			must	
	import statement							must

表 4.1: 実行時自動生成手法とリフレクション能力の関係

第 5 章

Python を利用した実装

本章では、前章で述べたリフレクションを利用して、アプリケーションに必要なスタブとスケルトンを実行時に自動的に生成してアプリケーションに組み込む手法を、インタプリタ型で対話的に実行可能なオブジェクト指向言語である Python [5] で実装する方法を述べる。

5.1 Python の提供するリフレクション

まず最初に、Python の提供しているリフレクションの能力を明らかにする。

5.1.1 Linguistic リフレクション

Python のプログラムの構成要素、すなわちモジュールやクラスやメソッドなどは、Python のオブジェクト[1]として実装されている。プログラムからこれらのオブジェクトの属性を参照/改変することで、プログラム自身の参照/改変が可能になる。プログラムの構成要素をあらわす各オブジェクトは、標準ライブラリとして提供されている new モジュールで生成できる。また、Python では任意の文字列をプログラムとして実行する exec 文も用意されているので、これを利用して新しいプログラムを生成することもできる。プログラムの構成要素の改変に関する制約は非常に少ない。Lingustic リフレクションの実用上問題になりそうなのは、一度確

[1]正確には組込み型の値だが、オブジェクトと同様に扱える

立されたクラスの継承関係の操作が禁止されていることくらいである。

5.1.2 Behavioral リフレクション

Python の処理系は比較的広範囲にわたって Behavioral リフレクションを許している。Python の名前空間はすべて Python 自身の連想配列型の値としてプログラムから参照できて、連想配列型の操作を用いて名前空間の参照/改変が可能である。制約が設けられているのは、関数オブジェクト内のローカルスコープだけである。スタックフレームも `Frame` オブジェクトとしてプログラムから参照できて、その内容を書きかえることもできる。

未定義名が参照されたときの処理系の振る舞いの変更は、クラスのメソッドと属性の名前についてのみ可能である。クラスを定義する際に `__getattr__` という名前でメソッドを定義しておくと、そのクラスのインスタンスについて未定義の名前が参照されたときに、実行エラーの例外を発生させる代わりに、その名前を引数としてこのメソッドが実行される。

Python にはファイルやディレクトリをモジュールとして扱うことができて、モジュールの内容を取り込む際には `import` 文を使用する。この `import` 文の実装が `__import__` という組み込み関数としてプログラムに公開されており、この関数を再定義することで `import` 文の振る舞いを変更することができる。`__import__` 関数の内部で `import` 文の実装に用いられている関数も、すべてアプリケーションから利用できる。さらに、 `__import__` 関数の実装をクラスに置き換えて、そのクラスを継承する形で `import` 文の振る舞いを安全に変更できるようにするクラスライブラリも用意されている。

5.2 スタブとスケルトンの実行時自動生成の実装

5.2.1 利用する CORBA の実装

Python に対応した CORBA の実装としては、CRC for Distributed Sytems Technology によって開発された Fnorb [12] がある。Fnorb は Python 用の CORBA 2.0 の実装であり、Python の基本型の値をネットワークストリームに変換する部分と、IDL

のパーザはC言語で記述されているが、他はすべてPythonで記述されている。

現在のCORBAの規格にはPythonの言語マッピングは含まれていないが、すでにOMGによるPythonの言語マッピングの規格化は最終段階にあり[13]、CORBA 3.0がリリースされるときに正式な規格になる可能性が高い。

5.2.2 生成するスタブとスケルトンの内容

インタフェースリポジトリを参照しながら、FnorbのIDLコンパイラが生成するものとほぼ同じ内容をLinguisticリフレクションを利用して生成する。生成した各定義には生成した時刻をクラス属性やローカル変数に埋め込んでおき、後でインタフェースリポジトリ内の分散オブジェクトと更新時刻と比較できるようにする。

インタフェースに対応するスタブクラスの内部については、未定義名の参照を起点とした生成が可能なので、スタブクラスについてはコンストラクタと`__getattr__`メソッド、およびリポジトリIDを保持するクラス属性だけを生成する。

5.2.3 オブジェクトリファレンスを起点とする生成

Pythonでは名前空間の操作が可能なので、前章で述べたオブジェクトリファレンスを起点とするスタブ生成が可能である。この操作は、Fnorbの実行時ライブラリにおけるネットワークストリームをオブジェクトリファレンスに変換する操作の一環として行う。

ただし、Pythonはアプリケーション全体で共有する名前空間がないので、オブジェクトリファレンスを取得する操作を行った名前空間を決定するために、スタックフレームをさかのぼる必要がある。

まず、その時点のスタックフレームをあらわす`Frame`オブジェクトを取得して、スタックフレームをさかのぼって、オブジェクトリファレンスを要求したアプリケーションの名前空間を特定する図5.1。Pythonでは例外処理のBehavioralリフレクションとして`Frame`オブジェクトをプログラムに見せるので、最初にダミーの例外を生成する必要がある。名前空間に`_FNORB_ID`が存在する間はスタブか実行時ライブラリの名前空間である。

```
  try:
   raise Dummy
except Dummy:
  frame = sys.exc_info()[2].tb_frame
  while frame.f_back:
    frame = frame.f_back
    if not frame.f_globals.has_key('_FNORB_ID') or \
           not frame.f_locals.has_key('_FNORB_ID'):
      break
```

図 5.1: スタックフレームの探索

こうして得られた名前空間に、インタフェースリポジトリ探索しながら、アプリケーションが利用する可能性があるユーザ定義型や定数値の定義を挿入する。最後に、オブジェクトリファレンスに対応するスタブクラスのインスタンスを生成してアプリケーションに返却することで、アプリケーションが分散オブジェクトを操作する準備がすべて整うことになる。

5.2.4 未定義名の参照を起点とする生成

前述したように、インタフェースの中で定義された内容に対応するスタブは、実際にアクセスされたとき呼び出される__getattr__で生成する。具体的には、スタブクラスに保存したリポジトリ ID と__getattr__の引数の名前を元に、インタフェースリポジトリから必要な定義を取り出して、対応するスタブを生成し、スタブクラスに挿入した後、挿入した値を__getattr__の返値として返せば良い。

__getattr__の引数の名前が sendp_で始まっている場合には、CORBA Messaging の遅延同期呼び出しの可能性があるので、sendp_を取り去った名前でインタフェースリポジトリを検索し、対応するオペレーションが見つかった場合には、遅延同期呼び出し用のメソッドを生成する。このときに、呼び出した返値を受け取るために用いる AMI_*interface*Poller クラスがまだ生成されていなければ生成する。

5.2.5　モジュールを起点とするスタブとスケルトン生成

モジュールを取り込む `import` 文の振る舞いを以下のように変更することで、`import` 文を利用してスタブとスケルトンをアプリケーションに取り込むことができる。

1. 指定されたモジュールが存在する場合には通常通り振る舞う
2. 存在しない場合には、モジュール名 (例えば `NameTable`) を IDL の絶対スコープ名 (`::NameTable`) に変換して、インタフェースリポジトリから IDL のモジュール定義を取得する。
3. モジュールに含まれる定義のスタブかスケルトンを生成する (Fnorb の IDL コンパイラに習い、モジュール名の最後が`_skel` の場合はスケルトンとする)
4. 生成したスタブやスケルトンを格納した Python のモジュールを作成する

5.2.6　各手法の得失

Python では未定義名が参照されたときの振る舞いを変更できるのはクラスだけなので、未定義名の参照を起点とするスタブの生成は、インタフェースの内部で定義された内容に対応するスタブを生成する場合にしか利用できない。アプリケーションが実装/操作するインタフェースの決定し、スタブクラスかスケルトンクラスを生成するには、別の方法を併用する必要がある。

オブジェクトリファレンスを元にした生成では、ユーザ定義型のスタブを利用するよりも先に、オブジェクトリファレンスを取得しなければならない、というプログラミングに関する制約が生じる。Python の場合には、アプリケーション全体で共有する名前空間がないので、取得したオブジェクトリファレンスを別の名前空間で定義された関数に渡してしまうと、そちらの関数ではオブジェクトリファレンスの操作に必要なスタブにアクセスすることができなくなるため、さらにプログラミングに関する制約がきつくなる。

とはいえ、対話的に Python の処理系を利用する場合には、基本的には分散オブジェクトを操作する名前空間は対話的な操作に用いているものだけになるので、十

分便利に活用することができる。

一般的なアプリケーション開発にスタブとスケルトンの自動生成を用いる場合には、振る舞いを変更した `import` 文を利用する方法がもっとも有効である。ただし、この場合にはモジュール単位の生成になるので、アプリケーションが利用しないスタブとスケルトンを取り込む可能性がある。未定義名の参照を利用したスタブ生成を併用することで、この無駄を最小限に押さえることは可能である。

第 6 章

Java を利用した実装

本章では、4 章で述べたリフレクションを利用して、アプリケーションに必要なスタブとスケルトンを実行時に自動的に生成してアプリケーションに組み込む手法を、コンパイル型で静的な型を持つオブジェクト指向言語である Java [51] で実装する方法を述べる。

6.1 Java の提供するリフレクション

まず最初に、Java の提供してるリフレクションの能力を明らかにする。

6.1.1 Reflection API の提供するリフレクションの能力

Linguistic リフレクション

Java の Reflection API [42] を利用すると、プログラムの構成要素、すなわちクラスやメソッド、フィールドなどを Java のオブジェクトとして参照できる。プログラムからこれらのメタオブジェクトを参照する際の最初の鍵となるのは、Java のすべてのオブジェクトが継承する `java.lang.Object` クラスの `getClass()` メソッドか、クラスのメタクラス `java.lang.Class` のクラスメソッド `forName()` である。これらにより、一度クラスのメタオブジェクトを取得したあとは、メタクラスのメソッドを利用してプログラムの字面上の情報は、メソッドボディの詳細を除いてすべて参照できる。Reflection API の提供する Linguistic リフレクション

は参照だけである。メタクラスには改変するためのメソッドや、生成するためのコンストラクタは用意されていない。

Behavioral リフレクション

クラスのメタクラスには、そのクラスのインスタンスを生成する `newInstance()` が用意されており、インスタンスの生成機構をプログラムから利用できる。メソッドのメタクラスには、そのメソッドを実行するためのメソッド `invoke()` が用意されており、メソッドの実行機構もプログラムから利用できる。Behavioral リフレクションについても提供されるのは実行機構の利用だけであり、改変することはできない。

6.1.2 ClassLoader の提供するリフレクションの能力

Java において、プログラムを実行時に生成して追加する能力と実行環境の振る舞いを改変する能力は、Reflection API ではなくクラスローダの機構によって提供される。Java のクラス定義は、プログラムがクラス名を参照したときに、クラスローダによってファイルから読み込まれる [52]。

クラスローダの振る舞いは `ClassLoader` クラスとしてプログラムから参照できる。Java の仮想機械によって用いられている `ClassLoader` のインスタンスは、このクラスのクラスメソッド `getSystemClassLoader()` で取り出すことができる。ただし、`ClassLoader` クラスのメソッドはサブクラスにしか見えないように保護されているので、クラスローダを利用するプログラムは `ClassLoader` のサブクラスに記述しなければならない。

実行時のプログラム生成

`ClassLoader` の `defineClass()` メソッドを用いると、プログラムの実行時にクラス定義を生成して取り込むことが可能になる。Java の仮想機械のバイトコードを用いてクラス定義をバイト列であらわして、`defineClass()` メソッドの引数に渡すと、それを仮想機械に読み込ませることができる。この実行時生成はク

ラス単位でしか行えず、一度読み込まれたバイトコードの変更は不可能なので、後でメソッドの定義を追加するなどの操作は実現できない。

クラス定義をあらわすバイトコードを生成するプログラムの記述には、かなりの手間がかかる。この手間は、Java のバイトコードをオブジェクトとして表現して、そのオブジェクトを介してバイトコードの編集を可能にする Javassist [53] を利用することで比較的容易に記述できる。ただし、メソッドボディの編集については、Javassist はメソッドボディが参照している名前の置換しか提供しないため、任意のプログラムを記述する際には、やはりバイトコードを生成する必要がある。

[40] では、クラス定義をあらわすバイトコードを生成するのではなく、文字列で表現しておいて Java のコンパイラを実行してバイトコードに変換するアプローチを採っている。彼らは Java のコンパイラを起動する方法として、クラス定義を中間ファイルに出力して本当に Java のコンパイラを実行する方法と、Java のコンパイラを実装している Java のクラスを直接利用する方法を述べているが、どちらのアプローチでも実行コストは非常に大きくなる。

クラスローダの振る舞いの改変

`ClassLoader` のサブクラスを定義する際にメソッドをオーバーライドすることで、既存のクラスローダの振る舞いを利用しつつ、異なる振る舞いをするクラスローダを定義できる。この `ClassLoader` のサブクラスのインスタンスを用いてクラス定義を読み込むと、読み込んだクラスが参照しているクラスを読み込む際にも、同じクラスローダが用いられるようになる。

Java の仮想機械は、あるクラスの定義がまだ定義が読み込まれていないクラスを参照している場合には、そのクラスを読み込んだクラスローダの持つ `loadClass()` メソッドをその名前を引数として実行する。`ClassLoader()` のサブクラスを定義するときに、まず親クラスの `loadClass()` を呼び出して、これが `ClassNotFoundException` 例外を返したときの振る舞いを記述することで、定義の存在しないクラス名が参照されたときの振る舞いを記述できる。

6.2　スタブとスケルトンの実行時自動生成の実装

6.2.1　スタブとスケルトンの実行時生成

前述したようにクラスローダの機構をプログラムから利用することで、アプリケーションの実行時にスタブとスケルトンを生成することは可能である。ただし、クラス単位での生成になるので、4.5.4 節で述べた、メソッドの実行機構とあわせて不要なメソッドの生成を防ぐアプローチは簡単には実現できない。

生成するスタブとスケルトンの内容

Java に対応している CORBA の実装は非常に多いので、リフレクションを利用して実行時に生成するスタブやスケルトンの内容は、各実装について互換性のあるものにしたい。これを実現するために、メソッドボディの内容を DII、DSI や DynAny API を利用して記述すると性能の劣化が避けられないことは 4.4.1 節で述べた通りである。Java の場合には、スタブとスケルトンと CORBA の実行時ライブラリの間のインタフェースを共通化することを目的に、“Java ORB Portability Interface” として IDL コンパイラの生成するスタブとスケルトンの実装を共通化する方法が言語マッピングの規格に示されている [34]。この方針に沿うことで、DII や DSI を利用する場合のように性能を大きく劣化させずに、実装への依存度の低いスタブとスケルトンを生成することができる。

生成したスタブクラスやスケルトンクラスをあらわすバイトコードは、`ClassLoader` クラスの `defineClass()` でアプリケーションに取り込むことができる。このバイトコードはクラスファイルとして保存しておいて、アプリケーションが次に起動したときには、ファイルの日付と対応するインタフェースリポジトリの定義の更新日時と比較して、更新されているものについてだけ生成することで、スタブやスケルトンの生成コストを押さえることもできる。

6.2.2　スタブとスケルトンの自動生成の実装

クライアントの取得したオブジェクトリファレンスを利用して、クライアントに必要なスタブを判断する手法は、Java では用いることができない。オブジェクトリファレンスを処理する CORBA の実行時ライブラリから、`ClassLoader` の `defineClass()` メソッドを用いて、クライアントプログラムから参照できるようにスタブを挿入することは、確かに可能である。しかし、Java のデフォルトのクラスローダが、その前にクラスに関する名前解決を行ってしまうので、オブジェクトリファレンスを取得した時点では間に合わない。

図 2.4 に示したクライアントプログラムを、オブジェクトリファレンスを取得する前にスタブのクラス名に触れないように修正したものを図 6.1 に示す。このプログラムを普通に起動すると、`Client` クラスの定義を読み込んだ仮想機械のデフォルトのクラスローダは、実行を開始する前にこのクラスが参照しているクラスの名前 (図中に太字で示した) について名前の解決を始めてしまうので、クラスの定義を取得できずにエラーになってしまう。

したがって、Java で自動的にアプリケーションに必要なスタブとスケルトンを生成して組み込むには、定義の存在しないクラス名に対応して、スタブかスケルトンを生成するように振る舞いを変更したクラスローダを用いるしかなく、しかもこのクラスローダはアプリケーションを起動するときに用いる必要がある。これを実現するには、アプリケーションを起動するときに `java` コマンドに直接アプリケーションのクラス名 `Client` を指定するのではなく、

```
$  java CORBALoader Client
```

のように `Client` を振る舞いを変更したクラスローダで読み込むためのフロントエンド (`CORBALoader`) を介して、起動する必要がある。

6.2.3　コンパイル時のスタブとスケルトンの自動生成

Java はコンパイルの必要な言語なので、スタブとスケルトンはアプリケーションを実行するときだけでなくコンパイル時にも必要である。Sun の提供している開発キットに付属している Java のコンパイラは、Java で実装されていて、コンパ

```
public class Client {
  public static void main(String[] args) {
    org.omg.CORBA.ORB orb =
      org.omg.CORBA.ORB.init(args, null);
    // オブジェクトリファレンスの取得
    org.omg.CORBA.Object obj =
              orb.bind("IDL:NameTable/NameTableServer:1.0",
                       "NameTable", null, null);
    NameTable.NameTableServer srv =
      NameTable.NameTableServerHelper.narrow(obj);
    NameTable.EntryValue val =
      new NameTable.EntryValue();
    val.i(1000);
    String key = "hoge";

   try {
      // オペレーション insert の実行
      srv.insert(key, val, true);
    } catch (NameTable.AlreadyExist e) {
      System.out.println("'" + key + "' already exist.");
    }
```

図 6.1: クライアントプログラムの例

イルしているプログラムが参照しているクラス名の定義を、コンパイラ自身が起動するときに用いられたクラスローダを利用して取得するようになっている。この性質を利用して、スタブとスケルトンを自動生成するクラスローダを用いてコンパイラを起動することで、コンパイル時に必要なスタブとスケルトンも生成できる。

```
$  java CORBALoader sun.tools.javac.Main Client.java
```

この方法でコンパイラを起動すると、スタブとスケルトンを自動生成するクラスローダがキャッシュとしてコンパイル時にもクラスファイルを生成してしまう。もし、デプロイメント (配置) の手順をいとわないなら、このときに生成されたクラスファイルを実行環境に配置してもよい。デプロイメントが煩雑な場合には、アプリケーションのクラスファイルだけを実行環境に持ち込んで、コンパイラと同様にクラスローダを置き換えるフロントエンドを介して起動すれば、必要なスタブとスケルトンが再び自動的に生成される。

6.2.4　本手法の得失

CORBA のアプリケーション開発に Java を利用する場合には、扱う必要のあるクラスファイルが非常に多くなるため、特にデプロイメントの手順が煩雑になる傾向がある。この手法を利用することで、アプリケーションに必要なスタブとスケルトンはアプリケーションの実行環境において自動的に生成されるため、この手順を大きく削減できる。

しかし、スタブとスケルトンの自動生成を実現するために Java で利用可能なリフレクションの能力が、クラス定義をあらわすバイトコードの取り込みと、クラスローダの振る舞いの変更しかなかったために、この手法自体の実装方法がやや煩雑になっている。

コンパイラの持つ振る舞いを実行環境が持つ振る舞いと、まったく同じ方法で改変できるという、Sun の開発キットに付属しているコンパイラの性質は、静的な型を持つプログラミング言語で本手法を実現する上で大きな役割を果たしている。

しかし、Sun の JDK に付属しているコンパイラの持つ、自身を読み込んだクラスローダでクラス定義の名前解決を行う仕様は、Java の言語仕様で定められているものではなく、しかもコンパイラを起動するときに用いている、コンパイラの `main` メソッドを持つクラスの名前 `sun.tools.javac.Main` も、正式に公開されているものではないという点で非常に危うい手法となってしまっている。

第 7 章

関連研究

CORBA のアプリケーション開発における開発手順の煩雑さを軽減するために、インタフェースリポジトリとプログラミング言語の実装を変更するアプローチはすでにいくつか行われている。本章では、これらのアプローチと本論文のアプローチを比較する。

7.1 既存のアプローチ

7.1.1 ILU

ILU (Inter-Language Unification) [10] は、様々なプログラミング言語で記述されたモジュール間のインタラクションを可能にすることを目的として開発された、CORBA の ORB とほぼ同様な機能を提供するシステムである。

ILU の Python 用の実行時ライブラリでは、`import` 文の振る舞いを変更して、モジュールの名前の代わりに IDL ファイルの名前を直接指定できるようにしている。この `import` 文は IDL ファイルが指定されると、IDL コンパイラを起動してテンポラリファイルにスタブとスケルトンを出力して、それをアプリケーションに取り込む。

実行時に起動された IDL コンパイラは構文解析からスタと/スケルトンの生成まで、すべて行うため実行コストが非常に大きい。また、IDL ファイルを利用するためクロスプラットフォームの開発における、インタフェースの不一致の問題に

は対処できない。IDL ファイルの形式上の制約から受ける影響も大きい。このアプローチがうまく機能するのは、一つのモジュールの内容が一つの IDL ファイルに格納されていて、かつファイル名がモジュールの名前と同じ場合だけである。

7.1.2 LuaORB および TclMico

LuaORB [54] は Lua というリフレクションの可能なプログラミング言語を CORBA に対応させたものである。Lua では Python と同様に、未定義メソッドが呼ばれたときのプログラミング言語の振る舞いを、リフレクションを利用して変更することができる。LuaORB では、これを利用してメソッドが呼び出されたときに、インタフェースリポジトリから必要な情報を取り込んで、スタブを生成せずに分散オブジェクトのオペレーションを DII で実行する。したがって、オペレーションの実行にかかるコストが非常に大きくなる。

LuaORB では IDL で定義されたユーザ定義型や定数を扱うために必要なスタブも生成されない。ユーザ定義型を扱うプログラムを記述する際には、リストや連想配列を利用して IDL の定義に合うように値を作成する必要があり、プログラミングが非常に煩雑になる。そもそも、ユーザ定義型や定数値はメソッドが呼ばれる前に必要になるので、メソッドが呼ばれたときにインタフェースリポジトリを参照するのでは、これらを扱うスタブを生成することはできない。

よく似たアプローチとしては TclMico がある [55]。TclMico は CORBA の実装の一つである MICO を利用して、スクリプト言語 Tcl [6] を用いて CORBA のアプリケーションを開発できるようにしたものである。インタフェースリポジトリを利用する、スタブを生成しないという点は LuaORB とまったく同じであり、性能とユーザ定義型に関する問題も同じように抱えている。

7.1.3 CorbaScript

CorbaScript [14] は CORBA のアプリケーションを記述するために新たに設計されたオブジェクト指向スクリプト言語である。CorbaScript はその処理系の実装で、プログラミング言語の名前空間とインタフェースリポジトリの名前空間を結合し

ていて、プログラミング言語側で未定義名にアクセスすると、対応する定義が自動的にインタフェースリポジトリから取り込まれる。

CorbaScript ではユーザ定義型や定数値に対する配慮もなされており、プログラミングを煩雑にすることなく、開発手順の煩雑さを軽減することに成功している。しかし、CorbaScript は新たに開発されたプログラミング言語であるため、既存のクラスライブラリを利用することができない。本論文では、リフレクションの能力を持つ既存のプログラミング言語を利用しているので、このような問題は生じない。

7.2 本論文のアプローチの特徴

7.2.1 クロスプラットフォーム開発への対応

サーバとクライアントの開発プラットフォームや用いるプログラミング言語が異なっている場合に起こりやすいインタフェースの不一致の問題を、改良したインタフェースリポジトリと、それを操作するためのツールを提供することで解消している。

インタフェースリポジトリを利用することで開発手順の煩雑さを解消することは、既存のアプローチでも行われている。しかし既存のアプローチでは、その要であるインタフェースリポジトリとそれを操作するツールの不備については、まったく考慮されていない。

7.2.2 ユーザ定義型への配慮

本研究のアプローチでは、スタブ/スケルトンについては従来のアプリケーション開発と同じ物を提供している。したがって、複雑なユーザ定義をプログラミング言語の型に対応付けることで、値の操作を容易にする CORBA の特徴がまったく損なわれない。

現状の CORBA ではオペレーションや属性の引数や返値に用いることができるのはデータ型でだが、CORBA 2.3 [1] で導入された**Object by Value** という規格によって、ユーザ定義型としてオブジェクトも扱えるようになろうとしている。ユー

ザ定義型の扱いを軽視しているアプローチでは、この流れに対応することはできない。

第 8章

おわりに

8.1　まとめ

本論文では、CORBA のアプリケーション開発における問題点として、サーバとクライアントが異なるプラットフォームや異なる言語で開発される際のインタフェースの不一致、および、スタブ/スケルトンファイルの生成とアプリケーションへの組み込みのために生じる開発手順の煩雑さを取り上げ、これらを解決する手法を提案した。

インタフェースの不一致の解消

インターフェースの不一致の問題は、インタフェースリポジトリを中心とする開発環境によって解決される。この開発環境は、更新日時を記録するインタフェースリポジトリと、そのインタフェースリポジトリを操作するためのツール、およびインタフェースリポジトリからアプリケーションに必要なスタブ/スケルトンを生成するツールから構成される。

サーバとクライアントの稼動するプラットフォーム、開発に用いるプログラミング言語、CORBA の実装の各要素がどのように混在していても、この環境を利用することでサーバとクライアントは正確に同じインタフェースの定義を共有することが可能になり、スタブ/スケルトンはこの共有している定義に基づいて自動的に生成されるので、インタフェースの不一致を防ぐことができる。

開発手順の煩雑さの軽減

リフレクションの可能なプログラミング言語を利用する場合については、アプリケーションの実行時に必要なスタブ/スケルトンを自動的に生成して取り込むことで、開発手順の煩雑さを大幅に軽減する手法を示した。

リフレクションによって提供される二つの能力のうち、Linguistic リフレクションはスタブ/スケルトンを実行時に生成するためと、すでに取り込まれているスタブを必要に応じて改変するために用いている。これにより、実行時のアプリケーションの振る舞いに基づいて、必要なスタブを段階的に生成することが可能になる。

スタブ/スケルトンを取り込むために必要な定義は前記のインタフェースリポジトリから取得する IDL ファイルの形式上の制約に縛られずにアプリケーションに必要な定義だけを取り込むことが可能になると同時に、インタフェースの不整合を防ぐことも可能になる。

Behavioral リフレクションは、プログラミング言語の処理系の振る舞いを変更することで、特別なプログラミングや開発手順なしで、アプリケーションに必要なスタブ/スケルトンを決定し、生成したスタブ/スケルトンをアプリケーションに組み込むために用いている。

本論文では、上記のアプローチをリフレクションをサポートしており、対応する CORBA の実装が存在する二つのプログラミング言語、Python および Java を用いて実装する方法を示した。Python による実装では、比較的広範囲にわたってリフレクションを提供する言語の有用性を示し、Java による実装では、狭い範囲のリフレクションしか提供しない言語でも、方法次第では上記のアプローチが実現可能であることを示した。

8.2 今後の展望

本論文で示した、リフレクションをサポートするプログラミング言語を利用してアプリケーションの開発手順を簡略化する手法は、CORBA に限らず、開発に用いるプログラミング言語と異なるオブジェクトモデルやデータモデルを持ったシステムを扱うアプリケーション開発一般に有効であることが予想される。CORBA 以外のそのようなシステムとしては、オブジェクト指向データベースシステムが

ある [56]。

4 章では CORBA のアプリケーション開発を簡略化するために必要なリフレクションの機能を示したが、既存のリフレクションをサポートするプログラミング言語でこれらをすべて満たすものは存在しない。これらのリフレクション能力を過不足なく、利用の容易な形で提供する新たなプログラミング言語を作成するアプローチも有用だろう。新たな言語を作成する際に常に問題となるライブラリの欠如の問題は、Java の仮想機械の上に、Java のクラスライブラリへのインタフェースを持つ形で実装することで回避できる [57, 58]。

謝辞

本研究を行うに当たり、終始ご指導を賜りました落水 浩一郎教授に深く感謝します。CORBA とリフレクションの関係を追及するというテーマを与えていただき、ご指導いただいた渡辺 卓雄助教授に深く感謝します。

本研究を進める上で有益なアドバイスをいただき、いくつかのミドルウェア製品に触れる機会を与えていただくなど、さまざなご支援をいただいた株式会社 PFU の熊谷 章氏および東田 雅宏氏には心からの感謝の意を表します。

本論文の審査委員に加わっていただいた本学の片山 卓也教授、篠田 陽一助教授および東京工業大学の柴山 悦哉助教授には、本論文を完成させるに当たり有益なアドバイスをいただき感謝します。また、本研究について有益なアドバイスをいただいた柴山研究室のメンバーの皆さんにも深く感謝します。

最後に、学業だけでなく私生活の面からもサポートしていただいた、落水研究室および篠田研究室のメンバー一同に感謝します。

参考文献

[1] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.3*, June 1999. formal/98-12-01.

[2] Ben Eng. CORBA vendor platform matrix. available at `http://www.vex.net/~ben/corba/platmatrix.html`, 1999.

[3] Sun Microsystems, Inc. *Java Remote Method Invocation Specification*, October 1998. Revision 1.50 JDK 1.2.

[4] Microsoft Corporation. *DCOM Technical Overview*, November 1996. included in the MSDN Library.

[5] Guido van Rossum. *Python Reference Manual*. Corporation for National Research Initiatives(CNRI), release 1.5.2 edition, April 1999. see `http://www.python.org/`.

[6] John Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[7] Larry Wall, Tom Christiansen, and Randal L. Schwarts. *Programming Perl*. O'Reilly & Associates, Inc., 2nd edition, 1996.

[8] Arno Puder and Kay ömer. *MICO is CORBA: A CORBA 2.2 Compliant Implementation*. Morgan Kaufmann Publishers, 1998. see `http://www.mico.org/`.

[9] AT & T Laboratories Cambridge. omniORB2. available at `http://www.uk.research.att.com/omniORB/`.

[10] Bill Janssen, Mike Spreitzer, Dan Larner, and Chris Jacobi. *ILU 2.0alpha14 Reference Manual*. Xerox Corporation, 1998. available at `ftp://ftp.parc.xerox.com/pub/ilu/2.0a14/manual-html/manual_toc.html`.

[11] Red Hat, Inc. *ORBit*. see `http://www.labs.redhat.com/orbit/`.

[12] CRC for Distributed Systems Technology. *Fnorb Version 1.0*, February 1999. see `http://www.dstc.edu.au/Fnorb`.

[13] Object Management Group. *CORBA Scripting With Python*, Aug 1999. OMG TC Document orbos/99-08-02.

[14] Laboratoire d'Informatique Fondamentale de Lille and Object-Oriented Concepts, Inc. *OMG CORBA Scripting Language RFP Revised Submission*, December 1998. OMG TC Document orbos/98-12-09.

[15] David Flater. Manufacturer's CORBA interface testing toolkit. available at `http://www.mel.nist.gov/msidstaff/flater/mcitt/`, 1999.

[16] ObjectSpace, Inc. *Voyager ORB 3.1 Developer Guide*, 1999. see `http://www.objectspace.com/products/vgrorb.htm`.

[17] Object Management Group. *CORBA 2.3 – Chapter 21 – Interceptors*, June 1999. formal/99-07-25.

[18] Object Management Group. *CORBA Messaging*, May 1998. OMG TC Document orbos/98-05-06.

[19] Object Share, Inc. *Distributed Smalltalk Application Developer's Guide*, April 1998.

[20] INPRISE Corporation, Inc. *Borland JBuilder 3 Developing distributed applications*, 1999.

[21] INPRISE Corporation, Inc. *Borland C++Builder 4 developing distributed applications*, 1999.

[22] 渡部卓雄. チュートリアル リフレクション. コンピュータソフトウェア, Vol. 11, No. 3, pp. 5–14, May 1994.

[23] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[24] INPRISE Corporation, Inc. *VisiBroker for C++ 3.3 Programmer's Guide*, 1998.

[25] 富士通株式会社. FUJITSU ObjectDirector プログラミングガイド, February 1998.

[26] INPRISE Corporation, Inc. *VisiBroker for Java 3.3 Programmer's Guide*, 1998. available at `http://www.inprise.com/techpubs/books/vbj/vbj33/index.html`.

[27] Aniruddha S. Gokhale and Douglas C. Schmidt. Measuring and optimizing CORBA latency and scalability over high-speed networks. *IEEE Transaction on Computers*, Vol. 47, No. 4, pp. 391–413, April 1998.

[28] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, June 1997.

[29] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992. available at `ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z`.

[30] Ben Eng. ORB Core feature matrix. available at `http://www.vex.net/~ben/corba/orbmatrix.html`, 1999.

[31] D. Brent Chapman and Elizabeth D. Zwicky. *Building Internet Firewalls*. O'Reilly & Associates, Inc., 1996.

[32] IONA Technologies PLC. *Orbix Wonderwall Administrator's Guide*, June 1999.

[33] INPRISE Corporation, Inc. *Gatekeeper Guide*, 1998.

[34] Object Management Group. *IDL to Java Language Mapping Specification*, June 1999. formal/99-07-59.

[35] Douglas C. Schmidt. Real-time CORBA with TAO (the ACE ORB). available at `http://siesta.cs.wustl.edu/~schmidt/TAO.html`, 1999.

[36] Object Oriented Concepts, Inc. *ORBacus for C++ and Java Version 3.2*. see `http://www.ooc.com/ob/`.

[37] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, October 1992.

[38] Ramana Rao. Implementational reflection in silica. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP '91)*, 1991.

[39] Paul Dourish. Developing a reflective model of collaborative systems. *ACM Transactions on Computer-Human Interaction*, Vol. 2, No. 1, pp. 40–63, 1995.

[40] Graham Kirby, Ron Morrison, and David Stemple. Linguistic reflection in Java. *Software: Practice and Experience*, Vol. 28, No. 10, pp. 1045–1077, 1998. see `http://www-ppg.dcs.st-and.ac.uk/Research/LinguisticReflection/`.

[41] J. McCarthy, P. W. Abrahams, D. J. Edwards, T.P. Hart, and M. I. Levin. *The Lisp Programmers' Manual*. The MIT Press, 1962.

[42] Java Software. Reflection. In *Java 2 SDK, Standard Edition Documentation*. Sun Microsystems, Inc., 1999. available at `http://java.sun.com/jdk/`.

[43] Guy L. Steele Jr. *Common Lisp The Language*. Digital Equipment Corporation, 1990.

[44] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective*. The MIT Press, 1993.

[45] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[46] Andrew Shalit, David Moon, and Orca Starbuck. *The Dylan Reference Manual*. Addison-Wesley, 1997.

[47] Søren Brandt and Réne W. Schmidt. The design of a meta-leval architecture for the BETA language. In *Proceedings of Workshop on Advances in Metaobject Protocol and Reflection (META '95)*, August 1995. available at `http://www.daimi.au.dk/~beta/Papers/sbrandt/meta95.ps.Z`.

[48] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Publishing, 1987.

[49] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA'98 Proceedings*, October 1998.

[50] Yukihiro Matsumoto. *Ruby Language Reference Manual version 1.4.3,* 1998. available at `http://www.ruby-lang.org/en/man-1.4/index.html`.

[51] Ken Arnold and James Gosling. *The Java Programming Language, Second Edition.* Addison-Wesley, 1998.

[52] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[53] Shigeru Chiba. Javassist — a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[54] Roberto Ierusalimschy, Renato Cerqueira, and Noemi Rodriguez. Using reflexivity to interface with CORBA. In *IEEE International Conference on Computer Languages (ICCL'98)*, May 1998.

[55] Frank Pilhofer. *TclMico*, March 1999. see `http://www.informatik.uni-frankfurt.de/~fp/Tcl/tclmico/`.

[56] R.G.G. Cattell, editor. *The Object Data Standard: ODMG 2.0.* Morgan Kaufmann Publishers, 1997.

[57] Moses DeJong and Cameron Laird. Tcl + Java = a match made for scripting. *SunWorld*, Nov 1999. available at `http://www.sunworld.com/sunworldonline/swol-11-1999/swol-11-jacl_p.html%`.

[58] Jim Hugunin. Python and java: The best of both worlds. In *Proceedings of the 6th International Python Conference*, Oct 1997.

本研究に関する発表論文

[1] 藤枝 和宏, 渡部 卓雄, 落水 浩一郎, “リフレクションを利用したCORBA 言語インタフェースの改善”, コンピュータソフトウェア 小論文 (投稿中)

[2] 藤枝 和宏, 渡部 卓雄, 落水 浩一郎, “CORBA アプリケーション開発におけるリフレクションの有効性”, 情報処理学会論文誌:プログラミング (投稿中)

[3] Kazuhiro Fujieda, Watanabe Takuo, and Koichiro Ochimizu, “CORBA Application Development Environment Using Reflection”, In Proceedings of International Symposium on Future Software Technology ’99, pp. 149–154 , Oct. 1999.

[4] 藤枝 和宏, 落水 浩一郎, “メタレベル・アーキテクチャを利用したオブジェクト管理システムの構成法について”, 情報処理学会研究報告 94-SE-101, Vol.94, No.99, pp. 41–48, Nov. 1994.

[5] 藤枝 和宏, 落水 浩一郎, “メタレベル・アーキテクチャを利用したオブジェクト管理システムのバージョン管理機構の構成法”, 日本ソフトウェア科学会第12 回大会論文集, pp. 109–112, 1995.

[6] 藤枝 和宏, 渡部 卓雄, 落水 浩一郎, “リフレクションを利用したCORBA アプリケーション実行環境の実現法”, 信学技報 SS99-31, Vol.99, No.287, pp.9–16, Sep. 1999.

[7] 藤枝 和宏, 渡部 卓雄, 落水 浩一郎, “リフレクションを利用したCORBA 言語インタフェースの改善”, 日本ソフトウェア科学会第 16 回大会論文集, pp. 93–96, Sep. 1999.

[8] 藤枝 和宏, 渡部 卓雄, 落水 浩一郎, “リフレクションを利用したCORBA API の改善”, 情報処理学会研究報告 99-SE-124, Vol.99, No.89, pp. 3–10, Oct. 1999.

[9] 藤枝 和宏, 東田 雅宏, “リフレクションを利用したCORBAのアプリケーション開発環境”, SEAMAIL, Vol.12, No.2, pp. 2–18, Nov. 1999.