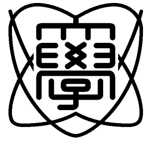

平成18年度 修士論文

リクエスタサイドから制御可能な Webサービスの分散型テスティングに関する研究

電気通信大学 大学院情報システム学研究科

情報システム設計学専攻

0550040 光林 真

指導教員 村山 隆彦 助教授
多田 好克 教授
岡本 敏雄 教授

提出日 平成19年1月30日

目 次

第 1 章 はじめに **5**

第 2 章 Web サービスの現状 **7**

2.1 Web サービス 7

2.2 Web サービスの現状 9

第 3 章 Web サービス開発における問題点 **11**

3.1 Web サービスの開発 11

3.2 既存のテスティング手法 13

3.2.1 プログラムの振る舞いに関するテスト 13

3.2.2 既存の分散型テスト技術 15

第 4 章 解決のアプローチ **16**

4.1 解決のアプローチ 16

4.2 手法 16

4.2.1 テストツール一式の送信 16

4.2.2 テストツール一式のフック 17

第 5 章 設計 **18**

5.1 システムの概要 18

5.1.1 テスト対象となるポートタイプの解析 19

5.1.2 テストツール一式の作成，送信，およびセットアップ 20

5.1.3 テスト開始から終了まで 20

5.2 構成要素となる技術 21

5.2.1 本研究との関わり 21

5.2.2 コード変換技術の概要 22
5.2.3 既存の動的コード変換技術 23
5.2.4 実行時コード変換可能な処理系 24

第 6 章 実装 25

6.1 実装環境 25
6.1.1 Java SE 25
6.1.2 Javassist 26
6.1.3 Apache Axis 26
6.1.4 Apache Tomcat 26
6.2 実装 27
6.2.1 Instrumentation Agent モジュール 27
6.2.2 Container モジュール 30
6.2.3 WSRuntime モジュール 31
6.2.4 Provider モジュール 32
6.2.5 Message Dispatcher モジュール 33
6.2.6 Controller モジュール 34
6.2.7 Messenger モジュール 38
6.3 使用方法 39
6.3.1 プロバイダ 39
6.3.2 リクエスタ 40

第 7 章 評価と考察 43

7.1 評価の概要 43
7.2 評価結果 47
7.2.1 本システム適用後のコード 47
7.2.2 達成目標の評価 48

7.3 考察 49
7.3.1 テスト要件に関係する内部情報の選定 49
7.3.2 リクエスタによるテスト要件の実現 50

第 8 章 関連技術 52

8.1 リクエスタプログラミング環境の充実に関する検討 52
8.2 セキュリティ 52

第 9 章 まとめ 54

図 目 次

2.1 Web サービスの概要 8

5.1 本システムの概要図 18

6.1 実装の構成 27

6.2 クラスローダ階層の例 (Tomcat) 28

6.3 バイトコード変換用クラスローダツリー 29

第 1 章

はじめに

新しい分散処理技術である Web サービス [1] は，その技術的発展と普及に伴い企業内システムの有機的な統合に有用であるとして注目されるようになってきた．Web サービスでは，プロバイダ (サービス提供者) が提供するサービスをリクエスタ (サービス利用者) が標準技術に基づいて利用する．このオープン性は Web サービスの重要な特徴の一つであり，プロバイダの実装に依存しない，高度に抽象化された分散システムを構築できる．

Web サービスによるシステムの抽象化は，プロバイダの内部情報を知らなくてもリクエスタがサービスを利用できるというメリットをもたらす．しかし，リクエスタ開発のテスト工程では，むしろプロバイダの内部情報を詳細に把握できる方がよく，抽象性は逆にテストを困難にする要因となる．

本研究の目的は，既存のプロバイダを利用したリクエスタ開発において，プロバイダの内部情報を取得する手段を確立し，リクエスタが主導する総合的なホワイトボックステストを実現することである．

一般に，プロバイダの内部動作をリモート監視または制御する手段はない．したがって，リクエスタはプロバイダの内部情報を必要とするテスト要件を作成できず，統合的なホワイトボックステストの実施は不可能となる．そこで，本研究では，リクエスタからプロバイダ内部にテスト用コード，ライブラリ，およびツールを Web サービスのメッセージとして送信し，実行時動的コード変換技術を用いて稼働中のプロバイダにフックするという手法を採ることで，問題の解決を図った．

以下，本論文の構成について概要を述べる．2 章では，Web サービスの技術的な概要と，企業内システムへの導入および運用状況について示す．3 章では，リクエスタ開発のテスト工程における Web サービス固有の問題を明らかにし，本研究の問題領域を定義する．更に，分散型テスティング技術およびテスティング方法論の両面から，既存のテスティング手法を検討する．4 章では，問題解決の要件を抽出し，解決のアプローチを提案する．5 章では，4 章で示したアプローチに基づき，システムを設計する．特に，本システムを構成する最も重要な要素技術である動的コード変換技術について述べ，本システムとの関わりを明らかにする．6 章では，5 章で示した設計に基づき，Java プラットフォームを用いて実装を作成する．7 章では，本システムを用い，実際にプロバイダの内部情報をテストケースの実行時に取得して，手法を評価する．8 章では，その他の関連研究について述べる．最後に，9 章で全体を総括する．

第 2 章

Web サービスの現状

2.1 Web サービス

Web サービスは，標準仕様に基づく分散技術である．

Web サービスでは，単一または複数のプロバイダを標準化されたプロトコル (SOAP [2] など) に基づいて組み合わせることで，粗粒度の複合サービスを合成できる．複合サービスでは，プロバイダとリクエスタを兼ねることもある．Web サービスの概要を図 2.1 に示す．

プロバイダは，サービス規約を Web Services Description Language (WSDL) [3] として公開する．WSDL は XML 文書 [4] として記述され，サービスのインタフェース，データ型，呼び出し手続き，バインドされるプロトコルなどを規定する．

リクエスタは WSDL の情報に基づき，SOAP メッセージをプロバイダに送信してサービスを受ける．SOAP は XML ベースのメッセージングプロトコルである．

不特定多数に向けて公開されている既存のサービスを探したい場合は，Universal Description, Discovery, and Integration (UDDI) [5] を使用する．UDDI は Web サービスのレジストリであり，Web サービスの検索を支援する．

その他の特定領域における仕様は，各標準化団体や業界団体で策定が進められている．

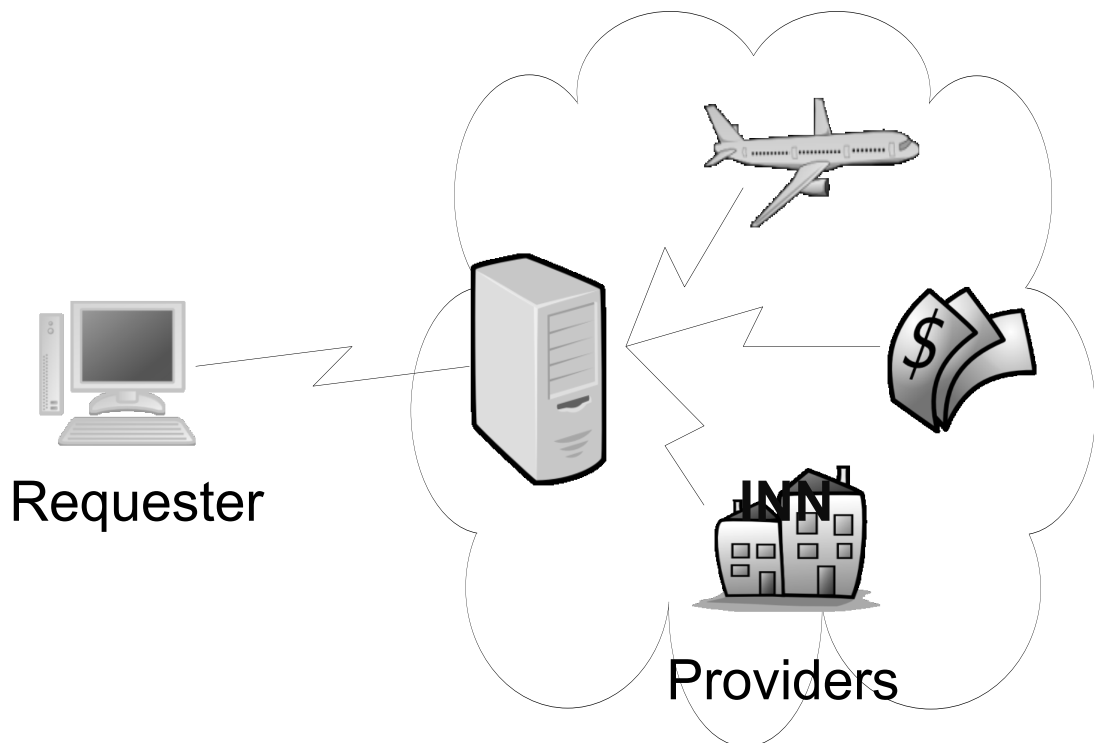

図 2.1: Web サービスの概要

2.2 Web サービスの現状

近年，Web サービス技術は企業内システムの相互接続に有用であると注目されている．

企業内システムにおいて，各部署のシステムをネットワークを介してシームレスに統合することは，部署間の連携を促し業務の効率化を図る上で重要である．

しかし，既存の企業内システムでは目的に応じて部署ごとに独立したシステムを構築しており，アーキテクチャの互換性がないケースが多い．このような異なるアーキテクチャで構成されるシステムを相互接続するための技術，すなわち分散技術は，ネットワークコンピューティングにおける重要な研究テーマである．

Web サービス以前の分散技術として，代表的なものに CORBA がある [6]．CORBA は，異機種で構成される分散環境におけるメッセージング仕様を標準化したものである．また，CORBA などの低レイヤ技術をミドルウェアとして実装し，より高レイヤでのアプリケーション統合を図る概念として，Enterprise Application Integration (EAI) がある [7]．

しかし，一般に EAI ミドルウェアの仕様はベンダによって異なっており，異なる EAI ミドルウェア間を相互接続することは困難である．このことから，従来の企業内システムは単一の EAI アーキテクチャに縛られる傾向にあった．また，EAI ミドルウェア，およびそれを用いたシステムの構築は，高いコストを要した．

一方，Web サービスは標準技術に基づいているため，単一ベンダの仕様に依存しない．この特徴は柔軟な相互運用性をもたらすので，既存の資源を多く抱える企業にとっては大きな魅力である．

また，既に豊富な Web サービス関連の実装が無償または有償で提供されており，十分に普及している．したがって，実用およびコストの観点から見ても，導入の障壁は低いと言える．

したがって，今後は新規に構築される企業内システムにおいても，Web サービ

ス技術を利用する事例が増加するものと思われる.

第 3 章

Web サービス開発における問題点

3.1 Web サービスの開発

稼働中の Web サービスを一つ以上利用して，新しいアプリケーションを構築するものとする．

プロバイダはサービスのインタフェースのみを公開しており，その情報は WSDL で記述されている．したがって，リクエスタが得られる情報は，プロバイダの WSDL に依存する．

通常，リクエスタはプロバイダの内部動作の詳細を知る必要はなく，インタフェースの規約のみを遵守すればよい．これは，コンポーネント指向の抽象プログラミングの観点から見ると，リクエスタは実装に関知せずにアプリケーションを構築できるという利点がある．

しかし，実際の開発においては，特にデバッグやテストの段階において，利用するサービスの内部動作を知りたいという要求がある．すなわち，いわゆるホワイトボックステストを Web サービス開発で実現したいという要求である．

スタンドアロンアプリケーションであれば，利用するコンポーネントはライブラリとして提供されるので，デバッガ等で追跡可能である．しかし，Web サービスは実装を公開しないので，リクエスタサイドからはいわゆるブラックボックステストしか行えない．

企業内システムなど，利用者の身元が特定可能で，かつ閉鎖的なネットワークに

おける運用であれば，開発段階の措置という前提の上で，プロバイダの実装に対する一定のアクセスを許容してもよい．この場合，プロバイダはテスト用にインタフェースを拡張し，テストに必要な実装をプロバイダに組み込む必要が生じる．このことに伴うコストは，プロバイダ管理者に課せられる．

一方で，リクエスタはプロバイダの想定するテスト要件から外れた，独自のテストケースを自由に構築したいという要求がある．しかし，リクエスタの新規開発，または既存リクエスタの仕様変更がなされるたびに，プロバイダ管理者に依頼してテスト要件に対応してもらうことは，プロバイダ管理者の負担を増大させるので，現実的ではない．また，サービスのコードを静的に変更しなければならないので，コードの品質低下に繋がる．

複合 Web サービスをテストする場合は，更に問題が複雑化する．テスト対象となるプロバイダ間に依存関係が存在する場合，リクエスタはテストケースのシーケンスに沿ってサービス要求の順序を制御する必要がある．このとき，サービス間の依存関係が正常に解決済みであると，リクエスタが確認できなければならない．しかし，この手続きはサービス間で直接行われるので，インタフェースから得られる情報のみで手続きの成否を確実に判断することは不可能である．

他，リモートデバッグにおける通信経路についても問題がある．既存のリモートデバッガは，多くの場合独自のプロトコルを使用して通信する．しかし，一般に Web サービスが稼働するホストはセキュリティ上好ましくない通信経路をファイアウォール等で遮断することが多いので，必ずしもデバッガを接続できるとは限らない．

以上の点が，リクエスタ開発における統合的なホワイトボックステストを阻害する要因となっている．

要点を以下にまとめる．まず，リクエスタからプロバイダに何らかの操作を行う上で嫁せられる，一般的な制約は以下の通りである．

1. サービスインタフェースの制約上，リクエスタからプロバイダの詳細な内部

動作を知ることは不可能である．

2. サービスインタフェースの制約上，リクエスタからプロバイダを制御することは不可能である．

3. サービスインタフェースで定義されたプロトコル以外は，セキュリティ上の問題から使用不可能である．

これらの事実から，リクエスタサイドにおけるテストにおいて，以下のような不都合が生じる．

1. リクエスタはプロバイダの内部情報を必要とするテストケースを構築できない．

2. リクエスタはプロバイダの制御権限を必要とするテストケースを構築できない．

3. Web サービス以外のプロトコルを用いて上記の不都合を回避することはできない．

3.2 既存のテスティング手法

3.2.1 プログラムの振る舞いに関するテスト

近年注目を浴びている Extreme Programming (XP) [8] や Rational Unified Process (RUP) に代表される新しいソフトウェア開発方法論では，特にテスティング方法論および技術が非常に重要視されている．

これらの方法論は，テスト実行を機械的に自動化してコストを削減することでテスト実行の繰り返しを促し，より安定かつ洗練されたコードへと改善することに役立つ．テストの問題領域はテスト対象となるモジュールの粒度および構成の

規模によって異なり，細粒度から粗粒度へ，小規模から大規模へ段階的に実施される．これらの各段階におけるテスト手法の各論については多くの既存研究がある．

Web サービスに対し既存のテスティング方法論および技術の適用を検討することは，前節で述べたホワイトボックステストの阻害要因がテストシーケンスにおいて表面化する箇所を明らかにすることに繋がる．以下，既存のテスティング方法論とそれらを実現する技術について述べ，本研究の問題領域への検討を図る．

単体テスト

単体テストとは，個々のプログラムモジュールを対象としたテストである．主にメソッドの引数および返値の型と値が仕様を満たしているかチェックする．

単一の Web サービス全体を一つのプログラムモジュールと見なすと，リクエスタによる単体テストの概念を適用できる．しかし，あくまで WSDL に従った Web サービス全体としての振る舞いのテストであり，Web サービスの内部動作には関与しない．

結合テスト

結合テストとは，プログラムモジュールを組み合わせて行うテストである．単体テストの次の段階に位置し，複数モジュールからなる総合的な振る舞いが仕様を満たしているかチェックする．

複合 Web サービス開発において，一つ以上のプロバイダからなる一連の処理シーケンスに対し，リクエスタによる結合テストの概念を適用できる．しかし，単体テストにおける概念の流用と同じく，WSDL による規約に依存する．また，プロバイダ間で直接行われるインタラクションは，リクエスタサイドで把握することはできない．

3.2.2 既存の分散型テスト技術

本研究は Web サービスを対象とすることから，特に分散型テストを実現する既存の技術について調査を行った．

リモートホストに存在するプログラムを変更可能なツールとして DJcutter [10] がある．DJcutter は分散環境でアスペクト指向プログラミングを実現することを目的としており，処理の一部をリモートホスト上に存在するプログラムで実行することができる．また，複数ホスト上にある不特定多数のプログラムで共有可能なプログラム断片を，サーバから配布してプログラムに織り込むこともできる．ただし，プログラム断片を織り込むためには，事前にホスト上のプログラムの実装について知らなければならない．すなわち，内部情報を引き出すための支援機能は持たない．また，プログラム断片の配布に使用するプロトコルは，Web サービスの標準プロトコル (SOAP) ではない．

Web サービスのテストを目的としたフレームワークに，WSUnit [11] がある．WSUnit は，リクエスタとプロバイダの間にプロキシとして介在し，SOAP 要求とその結果に介入することができる．しかし，あくまでも Web サービスの Mock として振る舞うだけなので，Web サービスにおける単体テストや結合テストの実現には有用であるが，プロバイダの内部動作に干渉することはできない．他，同種の Web サービスプロキシや Apache Axis TCPMon [12] などの SOAP モニタはすべて同様の問題を内包する．

第 4 章

解決のアプローチ

4.1 解決のアプローチ

3 章の問題点の解決にあたって，達成すべき目標を以下に示す.

1. リクエスタは，プロバイダの内部情報を取得できなければならない.
2. リクエスタは，プロバイダの内部動作を制御できなければならない.
3. リクエスタ・プロバイダ間の通信プロトコルは，Web サービス技術の枠組でシームレスに利用可能なものでなければならない.

既存の分散型テスティング技術では，これらの要件を満たせない．その主な原因は，WSDL による制約を受けず，かつ Web サービスの標準仕様に基づく，リクエスタからプロバイダの実行環境に直接リモートアクセスする手段が存在しないからである.

この問題を，以下に示す手法を用いて解決する.

4.2 手法

4.2.1 テストツール一式の送信

リクエスタが用意したテストツール一式を，テスト用ゲートウェイを通してプロバイダ内部へと送信する.

テストツール一式は，リクエスタサイドからプロバイダの制御および監視を実行するために用いられる (詳細は後で述べる).

送信時および送信後のリクエスタ-プロバイダ間通信は，テスト用ゲートウェイを通して Web サービスの標準プロトコルに基づいて行われる．すなわち，テスト用ゲートウェイは WSDL として公開し，通信プロトコルには SOAP を使用する．

テスト用ゲートウェイは，リクエスタとテストツール一式の間で交わされるメッセージングを SOAP エンコードおよびデコードする役割を担う．また，テストケースの構築に有用なユーティリティサービスへの簡便なインタフェースも提供する．

4.2.2　テストツール一式のフック

プロバイダ内部に送信されたテストツール一式を展開してセットアップし，サービスのプログラムコードへのフック，実行環境の監視などを実行する．

展開およびセットアッププロセスを起動するプログラムは，あらかじめプロバイダの実行環境に組み込んで動作させておく．

サービスコードへのフックは，サービスの制御および状態取得を目的として実行される．このとき，必要に応じてサービスコードを動的に変換する．

第 5 章

設計

5.1 システムの概要

上記のアプローチに基づき，テスティング支援システムの設計を行った．概要を図 5.1 に示す．

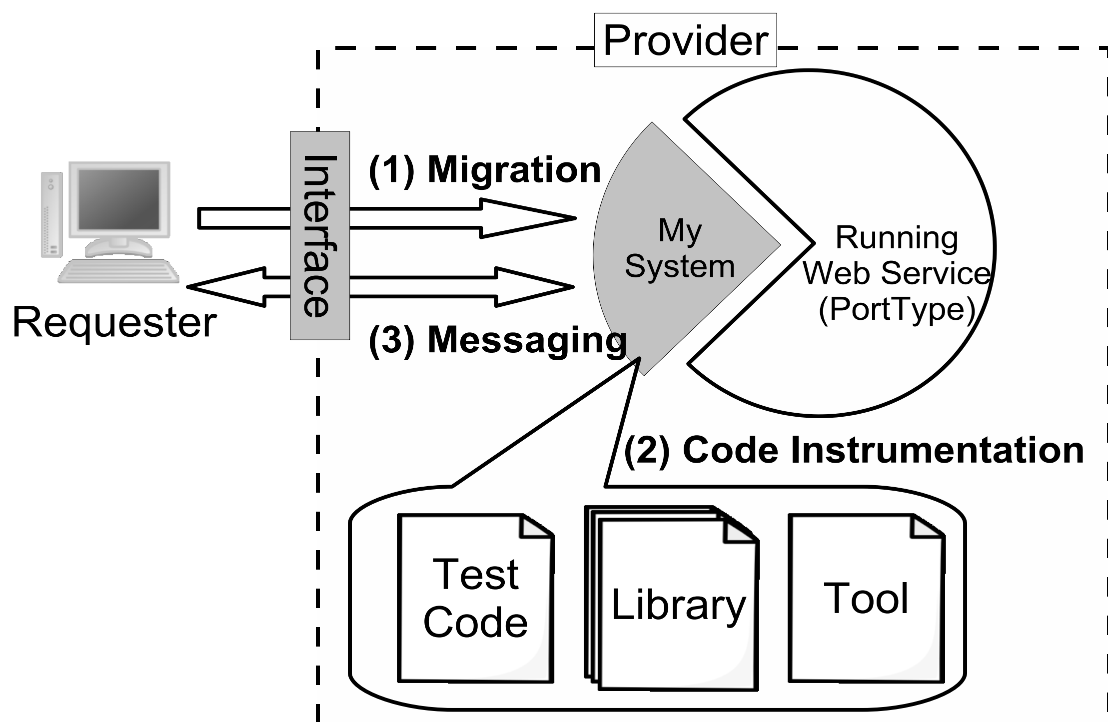

図 5.1: 本システムの概要図

本システムの動作シナリオを以下に示す．なお，プロバイダが提供する Web サー

ビスの実体をポートタイプと呼称する．ポートタイプの厳密な定義は，WSDL の portType 要素の定義に準じる．

5.1.1 テスト対象となるポートタイプの解析

まず，リクエスタが利用するポートタイプの構造を解析し，コードのメタ情報を取得する．この情報は，変換対象となるコードと変換ルールを記述するために用いられる．

以下にシーケンスを示す．

1. リクエスタは，対象となるポートタイプのプログラムコードに関する情報をテスト用ゲートウェイに問い合わせる．メッセージングプロトコルにはSOAPを用いる．
2. テスト用ゲートウェイは，プロバイダの実行環境上で動作しているコントローラにリクエスタの要求を転送する．
3. コントローラは，指定されたポートタイプのプログラムコードを探し出し，内容を解析する．このプログラムコードは，既に実行環境上にロード済みのものを使用する．解析の結果得られる情報は，当該 Web サービスの開発に用いられたプログラミング言語および処理系の仕様に依存する．
4. コントローラは，ポートタイプの解析結果をテスト用ゲートウェイに返送する．
5. テスト用ゲートウェイは，リクエスタに解析結果を SOAP メッセージとして返送する．

5.1.2 テストツール一式の作成，送信，およびセットアップ

リクエスタは，前節で取得したメタ情報を用いてプロバイダの内部情報を利用したテスト用コードを作成し，プロバイダ内部へと送信してフックする．本節の処理が完了すると，テストの準備が整ったことになる．

以下にシーケンスを示す．

1. 解析したポートタイプ情報を使用して，ポートタイプにフックするプログラム断片を作成する．
2. プログラム断片およびそれをポートタイプにフックするバインディングルール，プロバイダ実行環境を制御または監視するツール，それらの実行に必要なライブラリを，テストツール一式としてまとめる．
3. テストツール一式をシリアライズし，SOAP エンコードする．
4. テストツール一式を，テスト用ゲートウェイに SOAP メッセージとして送信する．
5. テスト用ゲートウェイは，SOAP メッセージをデコードして，コントローラに転送する．
6. コントローラは，受信したテストツールをデシリアライズし，適切にセットアップ (ポートタイプへのフック，ツールの起動，リクエスタとの通信コネクションの準備など) する．この段階では，テストツール一式はまだ無効である．

5.1.3 テスト開始から終了まで

リクエスタによって変換されたコードを有効にして，テストを実行する．

以下にシーケンスを示す．

1. リクエスタは，テスト用ゲートウェイにテストツール一式の有効化を要求する．
2. テスト用ゲートウェイは，リクエスタの要求をコントローラに転送する．
3. コントローラは，テストツール一式を有効化し，ポートタイプの処理に介入する．また，リクエスタとの通信コネクションをテスト用ゲートウェイを通して確立する．
4. リクエスタは通常のテストケースを実行する．テストツール一式が収集した情報は，テスト用ゲートウェイから SOAP メッセージとして取得する．
5. テストを一時的に中止する場合は，1〜3 と同様の手続きを踏んでテストツール一式を無効化する．
6. テスト終了後，リクエスタは 1〜3 と同様の手続きを踏んでテストツール一式の除去を要求する．

5.2 構成要素となる技術

本システムの実現にあたっては，シリアライズされたテストコードの送信および適切なセットアップと，ロード済みのポートタイプへのフックが重要な技術的課題となる．この点を解決すべく，本システムでは動的コード変換技術を適用する．

以下，本研究におけるコード変換技術の位置付けを示し，技術的概要と既存技術について述べる．

5.2.1 本研究との関わり

本システムにおけるコード変換技術の位置付けと要件について，以下に述べる．

本システムは，ロード済みのポートタイプを直接変換する．したがって，プロバイダの実行環境が実行時コード変換技術をサポートしていることが必須要件となる．

一方，リクエスタサイドでは，テスト用コードの記述容易性が重要となる．利便性の観点から見て，テスト用コードは既存のプログラミング言語を用いて，テストケースの一部としてシームレスに記述できることが望ましい．低級な記法に限定することはテストコードの難読化に繋がる．また，独自記法では，開発者に学習コストを負担させることになる．

しかし，記述性の自由度の観点から見ると，高級な記法のみに限定することは自由度を狭める要因となる．記述能力の限界はテスト用コードの処理系に強く依存するので，コード変換に既存のフレームワークを用いる場合は，その記述能力の限界を見定めた上で適用する必要がある．この点から，高級な記法に特化しているツールを本システムにそのまま適用することは得策とは言えない．開発者の利便性を損ねないという条件を満たした上で，より低級な記法もサポートできるフレームワークを選択するか，または両者の併用を検討すべきである．

上記の観点を踏まえ，コード変換技術について以下説明する．

5.2.2 コード変換技術の概要

コード変換技術は，大別して静的な変換と動的な変換に分類できる．

静的コード変換は，コンパイル済みのコードに対し，実行環境にロードされていない状態において変換処理を実行する．一方，動的コード変換は，コードのロード時か，または既にロード済みの状態において変換処理を実行する．

なお，本研究においてコード変換技術と呼称する対象は，変換ルールをプログラマブルに記述可能な手法に限定する．

変換ルールの記述形式は，対象をバイナリレベルで直接操作するものからソースコードレベルの高級言語を用いた指定が行えるものまで様々な種類がある．一

般に，前者に近いほど緻密な指定が可能であるが，記述が複雑かつ難解になりやすい．一方，後者は高級言語による記述が可能なので論理的な変換ルールを組みやすいが，コードの振る舞いを完全に制御することは不可能もしくは困難である．

コード変換技術の実現可能性は，その性質上処理系の仕様に極めて依存する．特に動的変換を実現する場合は，データ型やメソッドシグニチャなどのメタ情報を動的に取得する機能を，実行環境がサポートしていなければならない．

5.2.3 既存の動的コード変換技術

近年，コード変換技術を用いた新しいプログラミングパラダイムとして，アスペクト指向プログラミング (Aspect Oriented Programming，以下 AOP と呼称) [13] が注目されている．

AOP では，プログラム全体に散在する共通の処理概念を「横断的関心事」とし，プログラムモジュールの再利用性を損ねるものと見なす．そこで，横断的関心事は「アスペクト」と呼称される局所化されたモジュールとして抽出し，既存のプログラムコードに挿入するという手法を採る．AOP は，専らオブジェクト指向開発において，オブジェクト指向方法論ではモジュール化できない処理概念を再利用可能にするものとして，補完的に導入される．

アスペクトは AOP 処理系によって対象となるプログラムへ静的または動的に挿入される．挿入するコードポイントは設定ファイルなどの外部リソースに高級な記法で指定する．

動的な AOP は，再利用性の促進効果に加えて，元のコードに静的な影響をまったく及ぼすことなく，容易に，かつ大域的にコードを追加・除去できるという利点を持つ．例えば，デバッグ時にのみ必要なロギング処理をアスペクトとして実装しておけば，実運用に移行した際にプログラム全体をソースコードから再コンパイルすることなく，設定ファイルや起動パラメタの調整のみで除去できる．

既に多くの AOP 実装が開発されており，かつ広く普及しているが，いずれもア

スペクトの挿入を静的もしくはロード時に実行する手法を採っており，オンメモリのコードに対する変換はサポートしていないか，もしくは機能的に不完全な実装に留まっている．この制限は，実行環境の機能不足に起因する．具体的な詳細は次節で述べる．

5.2.4 実行時コード変換可能な処理系

現在，比較的コード変換を容易に実現可能で，かつ最も多く実装が開発されているプログラミング環境は Java [14] である．Java はインタプリタ型の仮想マシンを実行環境としており，コンパイル済みのコードに対して比較的容易に変更を加えられる．また，デバッグモードや Hotswap などの実行時コード変換を可能とする機能が仮想マシン仕様に含まれており [15]，動的 AOP ツールなどロード時または実行時にコード変換を行う実装にとって都合が良いという利点がある．しかし，メソッドシグニチャの動的な追加や変更は不可能であるといった制限もあり，完全な実行時コード変換を実現するには十分でない．将来的にこのような制限は撤廃される予定であるが，現状では各コード変換ツールがそれぞれ独自の手法を用いて制限を吸収している事例が殆どである．

第 6 章

実装

6.1 実装環境

第 5 章の設計に基づき，本システムを実装した．5.2 節で述べたように，実行時コード変換が比較的実現容易であることから，実装環境として Java プラットフォームを選択した．

以下，実装を構成する主要な要素技術について述べる．

6.1.1 Java SE

本システムの処理系として，Java SE を選択した．Java は，言語仕様の特性と実用性の両面において，本システムの実現に最も適していると言える．

Java を用いることによる主な利点を以下に示す．

1. Web サービス開発に最も多く用いられており，同技術に関する実装も豊富に存在する．

2. リフレクションやバイトコードインストゥルメンテーション，バイトコードの Hotswap など，実行時コード変換に有用な機能を言語仕様および仮想マシン仕様レベルでサポートしており，それらを利用した動的コード変換ツールも豊富に存在する．

3. 様々なレイヤにおけるテスティングフレームワークが豊富に存在する．

以下，Java 用語に基づき，変換の対象となるコードを「バイトコード」と呼称する．

6.1.2 Javassist

Javassist [16] はバイトコード変換器である．AOP ツールや DI コンテナの実装にも利用されており，十分な実績を持つ．

本システムにおける，実行時バイトコード変換部の基本的なフレームワークを提供する．

Javassist を用いることによる主な利点を以下に示す．

1. 挿入されるコードを Java ソースコードの断片として記述できる．
2. バイトコードレベルの操作が可能である．
3. Hotswap に対応しており，実行時にクラス単位でコードを置換できる．

6.1.3 Apache Axis

Apache Axis [17](以下，Axis と略称する) は Java で構築された代表的な Web サービスフレームワークである．SOAP メッセージングや WSDL の生成など，基本的な機能を提供する．他，ポートタイプとなる Java クラスから WSDL の自動生成する機能や，WSDL からリクエスタのスタブを自動生成する機能なども持つ．

6.1.4 Apache Tomcat

Apache Tomcat [18](以下，Tomcat と略称する) は代表的な Servlet コンテナである．Axis は Servlet として実装されており，実行の際は Servlet コンテナが必要となる．

6.2 実装

本システムは，複数のモジュールからなる．システムの構成を，図 6.1 に示す．

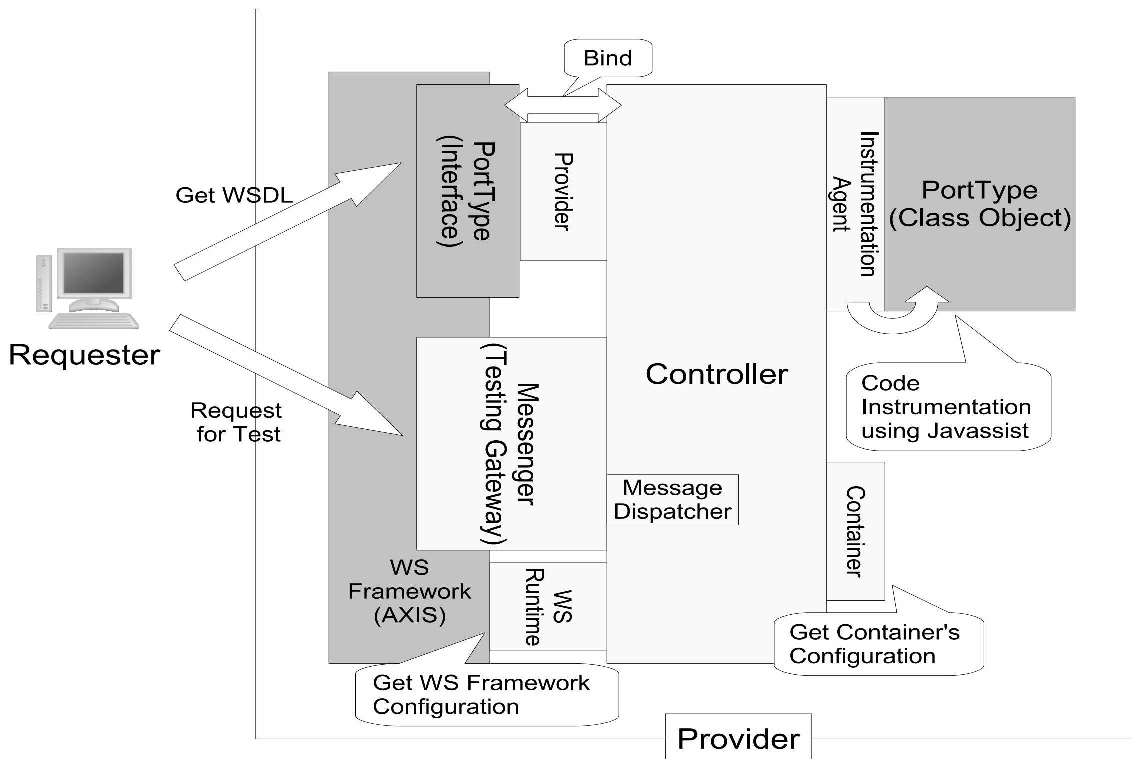

図 6.1: 実装の構成

以下，各モジュールについて述べる．

6.2.1 Instrumentation Agent モジュール

Instrumentation Agent は，仮想マシン上に存在するすべてのクラスオブジェクトを把握し，バイトコード変換の対象となるオブジェクトモデルを構築する．

Instrumentation Agent の振る舞いを説明するためには，Java におけるクラスローディングの規則を理解する必要がある．以下，JVM 階層型クラスローダについて述べる．

Java では，インスタンスを生成する前に，まずクラスオブジェクトをクラスロー

ダにロードする．クラスローダは JVM 上に複数存在し，ブートストラップローダを起点とするツリー構造を成している．図 6.2 に Tomcat における例を示す [19].

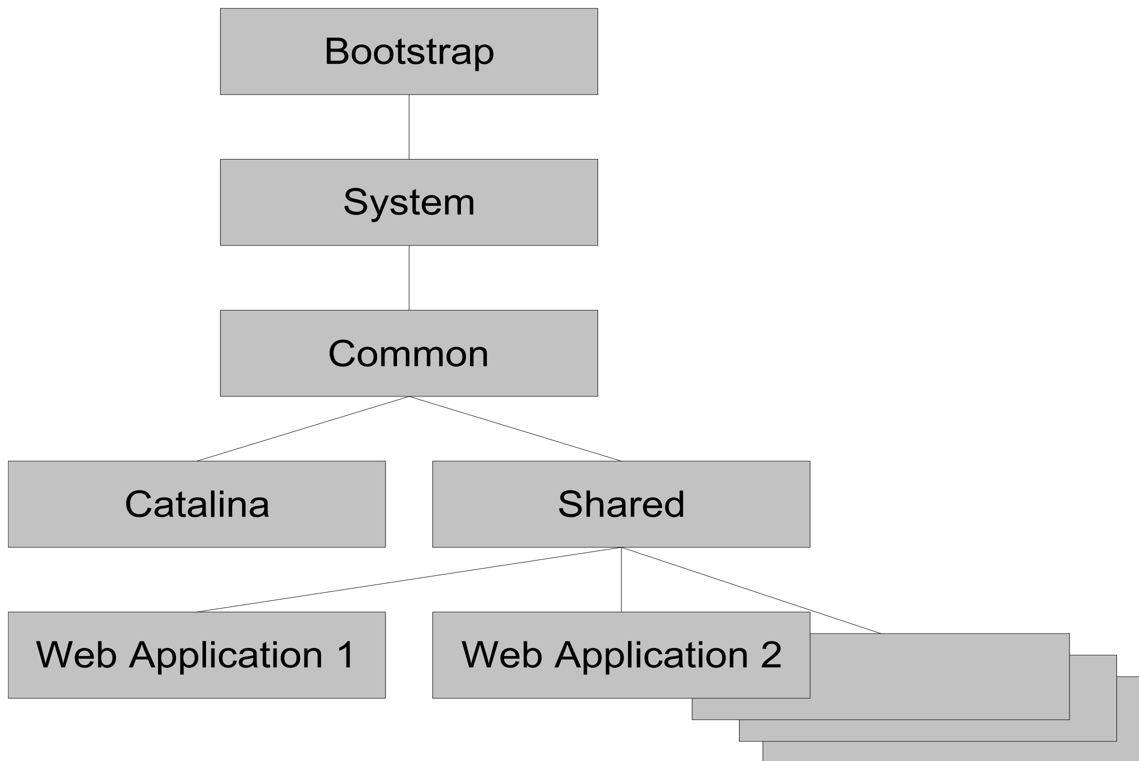

図 6.2: クラスローダ階層の例 (Tomcat)

クラスをロードする際，既に上位のクラスローダにロード済みであれば，それが使用される．そうでなければ，アプリケーションが使用しているクラスローダにロードされる．

親子関係にあるクラスローダ間ではクラスローディングの重複は発生しないが，兄弟関係のサブツリー間では，互いのツリーにぶら下がっているクラスローダが見えないので，同一のクラスがそれぞれのサブツリーにおいてロードされる可能性もある．

つまり，クラスローダ間の関係は，クラスオブジェクトのスコープを意味する．

Instrumentation Agent は，ブートストラップクラスローダレベルで起動し，システムクラスローダに常駐する．そして，その後ロードされるすべてのクラスを

監視し，必要に応じて処理に割り込む.

各クラスオブジェクトへの参照は，クラス名に基づいて保持している．しかし，先に述べた同名異スコープのクラスオブジェクトが存在すると，名前だけでは区別が付かなくなる．そこで，クラスオブジェクトが所属するクラスローダの情報も必要になる．

また，クラスオブジェクトを変換した場合，変換後のクラスオブジェクトのスコープは変換前と同一であることが望ましい．従って，変換後のクラス間の関係についても，ツリー構造のモデルで管理する必要がある．

以上の観点から，バイトコード変換用に別のクラスローダツリーを構築することとした．図 6.3 にモデルを示す．

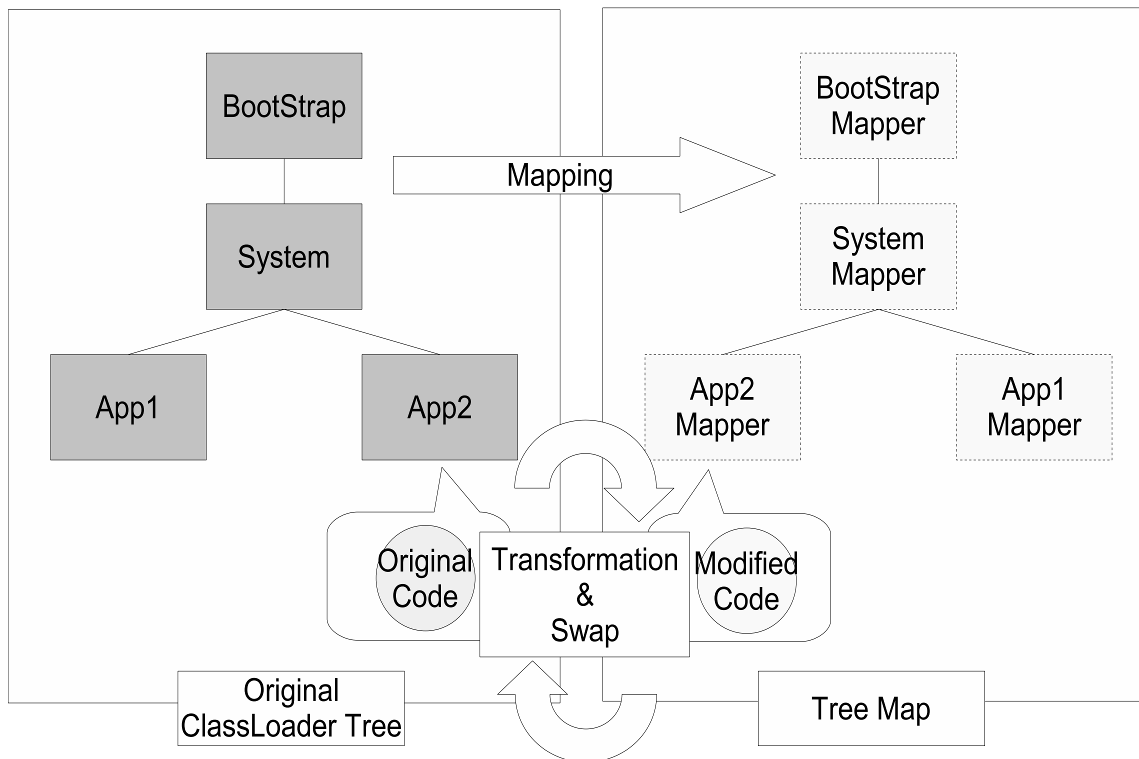

図 6.3: バイトコード変換用クラスローダツリー

すなわち，元のクラスローダツリー (オリジナルツリー) の階層関係を完全に保ったまま，バイトコード変換に適した形でツリーの写像 (コピーツリー) を構築する．

バイトコード変換はコピーツリーのクラスオブジェクトに対して実行され，必要に応じてオリジナルツリーのクラスオブジェクトと交換される．このとき，後で元に戻せるように，コピーツリー側でオリジナルコードを保持する．

コピーツリーの実装は，Javassist のオブジェクトモデルおよび API を用いて実装されている．バイトコード変換を行う際は，Instrumentaion Agent が提供するコピーツリーへのインタフェースを使用してクラスオブジェクトを取得し，Javassist を用いて変換するという手順を踏む．

6.2.2 Container モジュール

Container は，コンテナを抽象化したモジュールである．コンテナとは，Web アプリケーションフレームワークの実行環境である．本システムでは Tomcat が相当する．

一般に，設定ファイルやクラスファイルなどのコンテナによって参照されるリソースの位置およびフォーマットは，アプリケーションが配備される環境，コンテナの種類，および実行コンテキストによって異なる．Container モジュールは，このような不定な位置におかれたリソースをアプリケーションから参照するためのインタフェースを提供する．

本システムではポートタイプを特定するための識別情報の一部として，Web アプリケーション名と Web アプリケーションが配備されているファイルシステム上の位置を使用する．これは，同一のバイトコードであっても，所属するコンテキストが異なればクラスローダも異なり，すなわち別個のクラスオブジェクトと見なされるという，Java のクラスローディングのルールに対応するためである．

Container モジュールは，リクエスタから渡された Web アプリケーション名を元に，Web アプリケーションが配備されているコンテキスト情報を探索する．以下，その手順について述べる．

まず，使用中のコンテナを特定する．コンテナの種類と配備されている位置は，

Java 仮想マシンのシステムプロパティやファイルシステム上の位置などの手がかりから総合的に自動判別する．次に，特定した情報からコンテナの設定リソースを探索し，内容を解析する．解析結果から得られる情報はコンテナの種類に依存するが，少なくともコンテキスト情報のマッピングは特定できる．

コンテキスト情報は，Web アプリケーションごとに一つずつ存在し，主に Web アプリケーションのベースディレクトリ (ドキュメントベース) を調べるために用いられる．

Container モジュールは抽象クラスとして実装されており，使用するコンテナに対応したコンクリートクラスを用意しなければならない．コンクリートクラスでは，コンテナの仕様に依存するメソッドと自動判別ルーチンを実装する必要がある．現状では，Tomcat 5.5 系にのみ対応している．

6.2.3 WSRuntime モジュール

WSRuntime は，Web サービスフレームワークを抽象化したモジュールである．Web サービスフレームワークは Web アプリケーションとして振る舞うので，実行の際はコンテナが必要となる．本システムでは Axis が相当する．

ポートタイプとクラスオブジェクトのマッピング情報は，Web サービスフレームワークが管理している．しかし，一般にその設定リソースの位置やフォーマットは，Web サービスフレームワークの種類によって異なる．WSRuntime モジュールは，このような不定な位置におかれたリソースをアプリケーションから参照するためのインタフェースを提供する．

WSRuntime モジュールは，リクエスタから渡された WSDL のポートタイプ定義を元に，マッピングされているクラスオブジェクトを探索する．以下，その手順について述べる．

まず，ポートタイプの実装に用いられている Web サービスフレームワークを特定する．Web サービスフレームワークの種類は，Web アプリケーション全体の設

定リソースの内容などから総合的に自動判別する．次に，特定した情報から Web サービスフレームワークの設定リソースを探索し，内容を解析する．解析結果から得られる最も重要な情報は，ポートタイプとクラスオブジェクトのマッピング情報である．

マッピング情報は，主にクラスオブジェクトの完全修飾名 (Full Qualified Domain Name, FQDN) を調べるために用いられる．

WSRuntime モジュールは抽象クラスとして実装されており，使用する Web サービスフレームワークに対応したコンクリートクラスを用意しなければならない．コンクリートクラスでは，Web サービスフレームワークの仕様に依存するメソッドと自動判別ルーチンを実装する必要がある．現状では，Axis 1 系にのみ対応している．

6.2.4 Provider モジュール

Provider は，バイトコード変換の対象となるポートタイプのクラスオブジェクトを，本システムのバイトコード変換ルーチンから参照するためのモジュールである．

Container モジュールと WSRuntime モジュールは，それぞれが抽象化した対象のリソースへのアクセス手段を提供する．これらを用いて，実際にポートタイプとクラスオブジェクトのバインディング管理を行うのが Provider モジュールである．以下，その手順について述べる．

まず，リクエスタから渡された WSDL の URL を元に，Container モジュールと WSRuntime モジュールを取得する．そして，あるポートタイプに対するバインド要求があると，Container モジュールと WSRuntime モジュールによる解析結果に基づき，対応するクラスオブジェクトを探索してバインドする．このとき成立したバインディング情報は，Provider モジュール内で記憶される．これは，バインディングの重複を防ぐために用いられる．バインディングが成立すると，ポートタイ

プをキーとするクラスオブジェクトのルックアップが可能になる．テスト終了時など，バインディングを解除したい場合は，アンバインド操作を適切に行ってバインド情報を削除する．

ルックアップによって得られるクラスオブジェクトは，主にInstrumentation Agentモジュールのコピーツリーから対応するクラスオブジェクトを取得するためのキーとして用いられる．

6.2.5 Message Dispatcher モジュール

Message Dispatcher モジュールは，挿入されたテスト用コードによって出力されたメッセージをハンドリングするモジュールである．

例えば，標準出力を用いたデバッグプリントはプロバイダが稼働するホストのTTY に出力されてしまうので，リクエスタからメッセージを取得できない．また，テスト対象のメソッドが例外をスローした場合も，例外が伝搬する過程のどこかで意図しないキャッチが行われるか，またはプロバイダが稼働するホストの TTY にスタックトレースが出力されるかのいずれかとなり，リクエスタにエラーの発生を通知できない．

そこで，テスト用コードによるメッセージをバッファリングし，リクエスタに転送するための機構が必要となる．これが，Message Dispatcher モジュールである．

Message Dispatcher モジュールは，以下の機能を提供する．

1. メッセージバッファ
2. 例外ディスパッチ

以下，Message Dispatcher モジュールを用いたテスト用コードの書き方について概要を述べる．

テスト用コードでメッセージを出力したい場合は，所属するプロバイダのMessage Dispatcher モジュールを取得して，メッセージバッファへの書き込みメソッドを呼

び出す．

例外をディスパッチしたい場合は，`catch` した例外を Message Dispatcher モジュールの例外ディスパッチメソッドに渡す．

また，例外ディスパッチ時に，ユーザ定義ハンドラを作成して特定の処理をディスパッチャに実行させることも可能である．以下に例を示す．なお，`$e` はスローされた例外を表す Javassist 構文である．

```
ExceptionHandler eh = new ExceptionHandler();
MessageDispatcher md = Controller.getMessageDispatcher("WebappName");
md.dispatch($e, eh);
```

このとき，`ExceptionHandler` オブジェクトは Message Dispatcher モジュールによって呼び出し可能なメソッドを実装していなければならない．

本システムは，Message Dispatcher モジュールによって処理可能な例外クラスを提供している．テスト用コードが任意の例外処理を含む場合は，Message Dispatcher モジュール対応の例外オブジェクトでラッピングすればよい．

6.2.6 Controller モジュール

Controller は，本システムにおける論理的な処理を定義するモジュールである．本システムに対する制御要求はすべて Controller モジュールに集約される．

以下，Controller モジュールが提供する機能と，その内部的な振る舞いについて述べる．

Controller モジュールの生成

リクエスタから，テスト対象となるプロバイダが公開している WSDL の URL を受け取る．この WSDL，すなわちプロバイダに対して，一つの Controller モジュールが生成される．

WSDL の URL は，Controller モジュールおよび Web サービスフレームワークに関係するモジュール (WSRuntime，Provider など) の初期化に用いられる．

また，初期化時に Message Dispatcher モジュールを生成する．Message Dispatcher モジュールは一つのプロバイダに対して一つだけ生成され，静的オブジェクトとして参照される．

テスト対象となるポートタイプのバインド

リクエスタから指定されたポートタイプを，Provider モジュールを用いてバインドする．

同時に，ポートタイプをキー，ポートタイプを構成するメンバの集合 (コンストラクタ，メソッド) を値とするマップを生成する．以後，コンストラクタとメソッドを総称して「ビヘイビア」，前述のマップを「ビヘイビアマップ」と呼称する．ビヘイビアマップは，次に述べるコード挿入操作によって変更されたビヘイビアを記憶し，変更の重複を防止するために用いられる．

ポートタイプに対応するクラスオブジェクトの解析

ポートタイプ名に対応するクラスオブジェクトを取得し，そのメタデータを解析する．

まず，ポートタイプ名をキーに，WSRuntime モジュールからクラスオブジェクトの FQDN を取得する．それをキーに，Instrumentation Agent モジュールのコピーツリーから対応するコピーオブジェクトを取得し，Javassist を用いてクラス構造を解析する．解析結果として，クラスオブジェクトのメタデータ集合 (ビヘイビアのシグニチャなど) が得られる．

メタデータ集合は，ClassInfo オブジェクトとしてリクエスタに返信される．ClassInfo オブジェクトは，メタデータ集合を本システムとリクエスタで扱いやすい形式に変換し，適切なアクセサを付加したものである．

テスト用コードの挿入

任意のテスト用コードを，Javassist を用いてビヘイビアに挿入する．挿入可能なコードポイントは，ビヘイビア呼び出しの前後いずれかである．テスト用コードは，Java のソースコードとして記述する．

変更されたビヘイビアはビヘイビアマップに登録され，除去操作が行われるまで他の挿入操作を受け付けなくなる．未挿入のコードポイントへの挿入および例外ディスパッチャの有効化は可能である．

例外ディスパッチャの有効化

あるメソッド内で発生する例外をトラップし，呼び出し元に伝搬する前に処理を追加できる．ただし，`catch` ブロック内で対処が完結する例外は対象に含まれない．

主な用途は，Message Dispatcher モジュールの例外ディスパッチへ例外処理を委譲することである．詳細は Message Dispatcher モジュールの説明を参照のこと．

変更されたビヘイビアはビヘイビアマップに登録され，除去操作が行われるまで他の挿入操作を受け付けなくなる．未挿入のコードポイントへの挿入は可能である．

テスト用コードの除去

既に挿入されているテスト用コードを，Javassist を用いてビヘイビアから除去する．例外ディスパッチャも無効化される．

オリジナルツリーのコードからクラス全体を再生成するため，リクエスタによるすべての改変が無効となる．

除去操作後，当該ビヘイビアはビヘイビアマップからも除去される．

テスト用コードの有効化

挿入操作によってフックされたテスト用コードを有効にする．

挿入操作を行った段階では，コピーツリーのクラスオブジェクトが変更されただけで，まだテスト用コードは有効でない．本操作を行うことによって，実際にコピーツリーのクラスオブジェクトとオリジナルツリーのそれが入れ換わり，テスト用コードが有効になる．すなわち，この時点で実行時バイトコード変換が成立する．

クラスオブジェクトの入れ換えは，Java 仮想マシンの Hotswap 機能によって実現している．

テスト用コードの無効化

有効なテスト用コードを無効化する．

有効化操作の逆を行い，オリジナルコードに基づいてクラスオブジェクトを復元する．ロード済みのテスト用コード付きクラスオブジェクトは破棄されるが，コピーツリーには残っている．

ライブラリのロード

シリアライズされたライブラリを復元し，コンテナにロードする．ライブラリは，JAR フォーマットのバイト配列と見なす．

テスト用コードの実行にあたって外部ライブラリの追加を要する場合は，テスト用コードを挿入する前に，本機能を使用してあらかじめライブラリをセットしておかなればならない．もしライブラリが不足していた場合，実行時例外が発生する．

メッセージストリームの取得

Message Dispatcher モジュールが保持しているメッセージバッファを読み出すための，出力ストリームを提供する．

Controller モジュール自身はメッセージバッファの管理に関与しないので，メッセージバッファが更新されたことをメソッド利用者に通知できない．したがって，

リアルタイム出力を実現するためには，後述する Messenger モジュールか，または リクエスタにおいて，ポーリングしながらメッセージバッファを読み出すなどの工夫が必要となる．

6.2.7 Messenger モジュール

Messenger は，外部要求に対するインタフェースを提供する．

Controller モジュールは本システムにおける高レイヤのインタフェースを提供するが，それらは特定のクライアントに依存しないように設計された汎用的な仕様となっている．したがって実際に利用する際は，クライアントの差異を吸収するフロントエンドが必要となる．

Messenger モジュールは，Controller モジュールのフロントエンドとして機能し，以下の処理を行う．

1. 外部要求に対する応答
2. リクエスタとの通信に使用するプロトコルの変換
3. Controller モジュールの生成および初期化と，外部要求の転送

本システムにおける外部要求は，基本的に SOAP メッセージとして処理される．したがって，SOAP メッセージングに対応した Messenger モジュールをフロントエンドとし，Web サービスゲートウェイとして動作させることになる．Messenger モジュールは，WSDL を公開し Web サービスとしてテスト要求を待ち受ける．WSDL の内容は Messenger モジュールの public メソッド集合であり，Controller モジュールのインタフェースと似ている．

特に，本システムでは Web サービスフレームワークとして Axis を利用しているので，WSDL の生成や SOAP メッセージの処理は Axis に代替させることができる．実際，Messenger モジュールの内容は，Controller モジュールを生成して処理

を委譲しているのみである (ライブラリのデシリアライズやメッセージストリームのポーリングを除く).

もしWebサービス以外のプロトコルで本システムを利用したい場合も，そのプロトコルに対応したMessengerモジュールを作成し，Controllerモジュールに処理を委譲すればよい．その際，複雑なデコードやプロトコル変換などの処理は，Messengerモジュールに隠蔽すべきである．

6.3 使用方法

本節では，本システムの使用方法について述べる．

プロバイダ管理者が本システムを配備するために要するコストは，起動前の設定と起動パラメタの変更のみである．

6.3.1 プロバイダ

本システムは独立したJavaアプリケーションではなく，コンテナに付随するサブシステムである．以下に，起動前の設定とスタートアップのシーケンスを示す．

1. 本システムのInstrumentation Agentモジュールは，Javaエージェントとして実装されている．Javaエージェントとは，Javaのブートストラップクラスローダにおいて実行される特殊なモジュールである．

   Javaエージェントは`premain`という`main`の前に実行される特殊なメソッドを持ち，クラスローディングに独自の処理を付加することができる．

   JavaエージェントとしてInstrumentation Agentモジュールを有効にするためには，本システム一式を含めたJARアーカイブ (ex. wsinstrumentation.jar) を作成し，コンテナの起動コマンドに以下のようなオプションを付加する．

```
-javaagent"C:\libdir\wsinstrumentation.jar" \
```

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=9999
```

`-javaagent` オプションには Instrumentaion Agent モジュールを含む JAR アーカイブを指定する．なお，具体的な Java エージェントのクラスを指定するマニフェストファイルを，JAR アーカイブに含める必要がある．`-agentlib` は Hotswap に必要な設定で，Javassist が使用する．

2. Messenger モジュールを，テストを許可するポートタイプと同じコンテキストに配備して，テスト用ゲートウェイを開設する．Web サービスフレームワークに Axis を使用している場合は，Axis に付属している AdminClient を使用すると簡単に配備できる．
3. コンテナを起動する．同時にシステム内部で Instrumentation Agent が起動し，すべてのクラスローディングを解析してコピーツリーを生成する．
4. コンテナの起動が完了し，本システムが利用可能になる．

6.3.2 リクエスタ

テスト用ゲートウェイに対する要求は，すべて SOAP メッセージングで行われる．実用の際には，Axis に付属している WSDL2Java などのコードジェネレータを用いて，テスト用ゲートウェイとの通信に用いるクライアントプログラムを作成するなどの事前準備が必要になる．以下に，本システムを用いたテストのシーケンスを示す．

1. テスト用ゲートウェイのメソッドを呼び出し，テストしたいプロバイダの WSDL を URL 指定でセットする．すなわち，テスト用ゲートウェイの定義が含まれる WSDL を指定すればよい．
2. テスト用ゲートウェイのメソッドを呼び出し，引数にテストしたいポートタ

イプ名をセットする．本システムによって，ポートタイプのクラスオブジェクトが解析される．

3. テスト用ゲートウェイのメソッドを呼び出し，先の操作で作成されたポートタイプのメタデータ情報セットを取得する．
4. メタデータの情報を元に，ビヘイビアに挿入するコード断片を作成する．
5. テスト用ゲートウェイのメソッドを呼び出し，コード断片が使用するライブラリをシリアライズして送信する．
6. テスト用ゲートウェイのメソッドを呼び出し，任意のビヘイビアにコード断片を挿入する．挿入可能なコードポイントは，ビヘイビアの先頭および最後である．また，例外ディスパッチも，この段階で有効化する．これらの作業を，テストしたいビヘイビアすべてに対して行う．
7. 前述2から6の操作を，テストしたいポートタイプすべてに対して実行する．
8. テスト用ゲートウェイのメソッドを呼び出し，変更したコードをすべて有効化する．
9. テスト用ゲートウェイのメソッドを呼び出し，メッセージストリームを開く．
10. 実際にテスト対象となるポートタイプにサービスを要求して，テストを開始する．
11. テストを中断する場合は，テスト用ゲートウェイの無効化メソッドを呼び出す．この操作は，挿入されたコード断片の除去等は行わない．
12. 特定のビヘイビアに対する変更をすべて除去したい場合は，テスト用ゲートウェイの除去メソッドを呼び出す．

13. 特定のポートタイプに対するテストを止めたい場合は，テスト用ゲートウェイのメソッドを呼び出してポートタイプを解放する．
14. テスト用ゲートウェイのメソッドを呼び出し，テストを完了する．このとき，未解放のポートタイプはすべて解放する．

第 7 章

評価と考察

7.1 評価の概要

第 4 章で挙げた達成目標を，以下に再掲する．

1. リクエスタは，プロバイダの内部情報を取得できなければならない．
2. リクエスタは，プロバイダの内部動作を制御できなければならない．
3. リクエスタ・プロバイダ間の通信プロトコルは，Web サービス技術の枠組でシームレスに利用可能なものでなければならない．

これらの目標を達成できているか，以下の操作をトレースすることで評価する．

クラスオブジェクトの解析とシグニチャの検証

次に述べるビヘイビアシグニチャの検証プロセスを実現できるだけのシグニチャ情報を入手できることを確認する．その後，特定のビヘイビアに対し，先頭にクラス名およびビヘイビア名の出力，最後に引数の型および値の出力を行うコードを挿入する．これは，正常シナリオにおいて，達成目標の 1, 2 を検証する．

例外のディスパッチ

異常シナリオにおいて，達成目標の 1, 2 を検証する．ただし，例外処理は異常シナリオの回復を目的としているので，プロバイダの動作に副作用が生じるよう

な制御の改変は控えるべきである．

Web サービスプロトコルによる通信の検証

達成目標の 3 を検証すべく，SOAP メッセージングによる本システムの制御を試みる．Messenger モジュールは，SOAP メッセージを正確にエンコードまたはデコードし，本システムとリクエスタを仲介できなければならない．

上記の項目について本システムを検証すべく，実際に実行中のサービスに対して本システムによる制御を試みた．

評価実験では，サービスの要求呼び出しに相当するメソッドに対して以下の操作を行う．

1. メソッド呼び出し前にコードをフックし，引数の値をリクエスタに出力する．
2. メソッド呼び出し後にコードをフックし，戻り値の値をリクエスタに出力する．
3. 例外が発生した場合は，Message Dispatcher にディスパッチする．

以下に，実験に用いたサービスであるCalculatorService のコードを示す．このサービスは，任意の演算子を用いて 2 項演算を行うcompute というメソッドを公開する．サポートされていない演算子を指定された場合は，例外をスローする．上述の操作は，compute メソッドに対して実行される．

```
public class CalculatorService {

  private static HashMap<String, Operator> operatorMap;

  static {
    operatorMap = new HashMap<String, Operator>();
    operatorMap.put("+", new PlusOperator());
```

```
  operatorMap.put("plus", new PlusOperator());
  operatorMap.put("-", new PlusOperator());
  operatorMap.put("minus", new PlusOperator());
}

// 抽象的な演算子クラス．サブクラスで２項演算を実装する
abstract static class Operator{
  private String name;
  private String symbol;
  Operator(String name, String symbol){
    this.name = name;
    this.symbol = symbol;
  }
  abstract int operate(int x, int y);
}

// 加算演算子
Static Class PlusOperator extends Operator{
  PlusOperator(){
    super("plus", "+");
  }
  int operate(int x, int y){
    return x + y;
  }
}
```

```
  // 減算演算子
  static class MinusOperator extends Operator{
    MinusOperator(){
      super("minus", "-");
    }
    int operate(int x, int y){
      return x - y;
    }
  }

  // 演算子を指定して２項演算
  public int compute(int x, int y, String operator){
    if(!operatorMap.containsKey(operator))
      throw new UnsupportedOperationException("\"" +
    operator + "\" is not supported.");
    else{
      int retval = doCompute(x, y, operatorMap.get(operator));
      return retval;
    }
  }

  private int doCompute(int x, int y, Operator op){
    return op.operate(x, y);
  }
}
```

7.2 評価結果

CalculatorServiceに対して前述の操作を行い，本システムによってコードが期待通りに変換されていることを確認した．また，変換後のコードが本章の冒頭に挙げた達成目標を満たすように動作することを確認し，機能性とその限界について評価を行った．以下，それぞれについて詳細を述べる．

7.2.1 本システム適用後のコード

変換されたcomputeメソッドは，以下のソースコードと等価なバイトコードに変換された(コメントは除く)．メソッドの呼び出し前後と例外処理に，意図した通りの処理が組み込まれている．

```
public class CalculatorService {
  public int compute(int x, int y, String op) {
    MessageDispatcher dispatcher = controller.getMessageDispatcher();
    dispatcher.write("Method: setValue, x = " + x + , y = " + y);
  try{
    // サポートされていない演算子はエラーで例外処理
    if(!operatorMap.containsKey(operator))
      throw new UnsupportedOperationException("\"" +
        operator + "\" is not supported.");
    else{
      int retval = doCompute(x, y, operatorMap.get(operator));
      dispatcher.write("Method: compute, retval = " + retval);
      return retval;
  }catch(Exception ex){
    dispatcher.exec(ex);
```

```
    dispatcher.forward(ex);
  }
}
```

7.2.2 達成目標の評価

クラスオブジェクトの解析

実験結果より，ビヘイビアのシグニチャを取得できることを確認した．しかし，ビヘイビア間の関係までは解析できていない．

例えば，あるビヘイビアとそのビヘイビア内から呼び出される他のビヘイビアのみに着目し解析対象に指定する，といった使い方はできない．なぜなら，ビヘイビアのボディは解析しないので，ビヘイビア呼び出しの一覧が取得できないからである．現状では呼び出されるタイミングでビヘイビア名を出力するコードをすべてのビヘイビアに挿入して，メッセージ内容を元に呼び出し順序を追跡するなど，実行時に得られる情報から解析しなければならない．

したがって，無差別にビヘイビアの振る舞いをダンプするような使い方は可能であるが，呼び出しの連鎖に含まれるビヘイビアのみを選択することはできない．より精度の高い情報を得るためには，ビヘイビアのボディを静的解析してコールグラフを作成する必要がある．詳細は考察で述べる．

ビヘイビアシグニチャの検証

ポートタイプのクラスオブジェクトが持つビヘイビアは，private メソッド等も含めて，すべての引数と戻り値を検証できた．

この結果は，リクエスタサイドでプロバイダの振る舞いに対する事前条件および事後条件を記述できるということを意味する [20]．このことは，プロバイダで定義された両条件を満たした上で，更にリクエスタのテスト要件に適した条件に絞りこめるという有用性をもたらす．

例外のディスパッチ

異常シナリオを追跡すべく，例外オブジェクトのスタックトレースをメッセージバッファに追加する機能をディスパッチャに組み込み，ディスパッチャを有効にして故意に例外を発生させたところ，例外連鎖の原因となったクラス名とスタックトレースを得られた．

Web サービスは例外をサポートしており，WSDL で`fault`要素としてカスタム例外を定義しておくことで，ポートタイプで発生した例外をリクエスタに送信できる．しかし，例外をスローしたクラスやスタックトレースなどの情報は削られてしまう．本システムを用いれば，例外オブジェクトを Web サービスフレームワークが処理して SOAP メッセージ化する前にディスパッチするので，無劣化の Java 例外オブジェクトとして独自に処理可能である．

7.3 考察

本システムでは，ビヘイビア呼び出しの前後に任意のテストコードをフックすることで，リクエスタサイドによるプロバイダ内部情報の取得および制御を実現した．この手法によって得られるプロバイダへの干渉能力とリクエスタによるテスト要件定義能力について，考察を以下に述べる．

7.3.1 テスト要件に関係する内部情報の選定

内部情報の取捨選択は，基本的にはリクエスタのニーズに依存するため，完全に機械的に決定することはできないと考える．しかし，着目すべきサービスインタフェースに関係する情報のみに限定することは可能である．ある程度内部情報を絞り込み，かつ依存関係を明らかにすることで，リクエスタによる内部情報の選定を効果的に支援できる．

テスト要件に必要な内部情報のみを選定するためには，着目するサービスが呼

び出すビヘイビアの一覧が必要である．単純な方法であれば，すべてのビヘイビアに対して呼び出し時にシグニチャを出力するようなコード断片を挿入し，実際にサービスを要求して出力を解析すればよい．しかし，この方法ではすべてのビヘイビア呼び出しを完全に網羅できるとは限らない．ビヘイビアの相互関係を完全に明らかにするためには，クラスオブジェクトを静的解析してコールグラフを作成する必要がある．コールグラフとは，関数の呼び出し関係を有向エッジで結び付けたグラフである．

ただし，実行時に決定される呼び出し関係は，静的に作成したコールグラフでは把握できない．また，SOAP メッセージングなど本システムの解析対象に含まれない RPC も，呼び出し関係とは見なされない．

前者については，実行時に生成したコールグラフで補えばよい．この場合，まだロードされていないコードは対象とならないが，そのようなコードは解析対象となるポートタイプにとって重要でないと判断できるので，大きな問題にはならないと考える．

後者は，言語仕様外の RPC を扱うための拡張が必要となる．この実現手法については本システムの問題領域から外れるので，ここでは議論しない．

7.3.2 リクエスタによるテスト要件の実現

Meyer は「契約による設計」(Design by Contract．以下 DbC と呼称) の概念を提唱した [20]．DbC は契約プログラミングとも呼ばれ，コードが満たすべき仕様をプログラムとして記述することで，設計の頑健性を高める手法である．

DbC では，関数の呼び出し前後で満たすべき条件をそれぞれ事前条件，事後条件と呼称し，プログラムとして記述する．これは，関数の仕様をプログラムとして明文化することと同義である．

本システムの評価では，ビヘイビアの前後に引数や戻り値をチェックするコードを挿入し，検証に成功した．これは，リクエスタがプロバイダの内部動作に対する

事前条件，事後条件を表明できることを意味する．すなわち，プロバイダの設計者による事前条件，事後条件を満たす範囲内で，リクエスタが意図する仕様を条件として追加することで，より精緻なテスト要件が実現できるということである．

例えば，あるビヘイビアの引数または戻り値が，プロバイダが意図した規約には適合していたとしても，リクエスタ全体の振る舞いにおける部分動作としては不適切であるというケースが考えられる．このような状況では，リクエスタ側が意図しない値は論理的なバグとみなし，実行時エラーとしてプロバイダに処理させるように，本システムを用いて事前または事後条件を更に制限すればよい．特に非公開ビヘイビアに関してはプロバイダの処理系に処理と検証を任せざるを得ないので，本システムを用いることではじめてプロバイダ内部に独自のテスト要件を織り込むことが可能になる．

なお，DbC では事前条件，事後条件の他に，クラス，メソッド，フィールドの不変性を保証する契約として，不変条件の概念を提唱している．本システムにおいてリクエスタによる不変条件の表明を実現するためには，クラスやメンバを監視するようなコードを実行時動的コード変換技術を用いて挿入すればよい．しかし，一般にリクエスタはビヘイビアの振る舞いが及ぼす影響について知りたいのであって，ビヘイビアの不変性を保証できたところで有用性は少ないと考えられる．したがって，本システムでは不変条件をサポートしていない．

第 8 章

関連技術

8.1 リクエスタプログラミング環境の充実に関する検討

Bugdel [21] は，Javassist を用いたアスペクト指向デバッグツールである．Eclipse のプラグインとして動作し，生成したデバッグコードを元のクラスに埋め込むことができる．動的コード変換のツールとして Javassist を使用しており，コード挿入ポイントの指定可能箇所は，本システムと同等である．しかし，デバッグコードの挿入はローカルのコードに対して行われ，リモート環境で稼働中のコードを直接変換することはできない．本研究は，リモート環境のコードを解析して情報を取得することを目的としているので，そもそもツールの方向性が異なる．しかし，Bugdel が生成したコードは特殊なランタイムを必要としないので，本システムのリクエスタプログラミング環境に利用可能であると期待できる．

8.2 セキュリティ

外部からの実行環境の制御は極めて重大なセキュリティ上の危険が伴うので，テスト用インタフェースは厳重に保護され，信頼できるホストのみが暗号化された経路を用いてアクセスできるようにしなければならない．

一般に，リモートアクセスのセキュリティ技術としては，SSH が普及している．しかし，本研究は Web サービス以外の通信経路は許可されないとの前提に立って

おり，SSH の使用は認められない．そこで，Web サービスの枠組みの中で利用できるセキュリティ技術が必要となる．

現在，Web サービスではセキュリティ技術を細かく分類し，それぞれについて仕様の策定を進めている [22] [23] [24] [25] [26]．先に述べた本システムの適用とセキュリティ確保のトレードオフに関しては，これら Web サービス固有のセキュリティ技術と，SSL などの既存の暗号化経路の組み合わせで解決できる．すなわち，既に存在する標準技術を適宜組み合わせればよく，特別なセキュリティ機構を独自に実装する必要はないと考える．

第 9 章

まとめ

本研究では，まず，Web サービス開発におけるリクエスタ主導のテストにおいて，プロバイダのブラックボックス的性質が，内部情報を必要とするテストケースを作成不可能たらしめていることを指摘した．

この問題を解決すべく，Web サービスのプロトコルで利用可能な，内部情報の取得および内部動作の制御を実現するシステムを設計，実装した．本システムは，実行時コード変換技術を用いてリクエスタが用意したテストツール一式を内部にフックすることで，リクエスタ主導のテスト要件定義と，テスト要件に必要な内部情報の収集を実現する．

本システムの実装にあたり，最も重要な要素技術である実行時コード変換技術は，実行環境として Java プラットフォームを，コード変換器として Javassist を採用した．本システムを用いて作成したテストケースを実行し，内部状態の取得を試みたところ，正常および異常シナリオのいずれにおいても，期待通り結果が得られ，問題点の解決に有効であることを確認した．

ただし，本システムの解析対象に含まれない実装またはプロトコルを内部で使用している場合や，実行時にバインドが決定されるクラスオブジェクトに関しては，有効でない．また，本システムを用いることで，リクエスタサイドからプロバイダの各ビヘイビアに対し，事前条件と事後条件を追加定義できる．これは，プロバイダが定義した各条件の範囲内で，リクエスタのテスト要件を条件という形態で付加することに等しい．

謝辞

本研究を遂行するにあたっては、いろいろな方々にお世話になりました。

まず、指導教員の村山隆彦先生には日頃から熱心なご指導、そしてご鞭撻を賜わりました。ご多忙中にもかかわらず論文の草稿を丁寧に読んでくださり、大変貴重なご助言をいただきました。ここに厚く御礼申し上げます。また、厳しくも貴重なご意見、適切なご助言を頂いた多田研究室の多田好克先生と佐藤喬助手に感謝いたします

そして、本研究が行なえたことは、研究方針や方法論について議論をし、共に研究生活をおくってきた村山研究室と多田研究室の学生諸氏のおかげでもあります。最後に、これらの皆さんに感謝いたします。

参考文献

[1] Web Services,
http://www.w3.org/2002/ws/

[2] SOAP Version 1.2 part 1,
http://www.w3.org/TR/soap12-part1/

[3] Web Services Description Language (WSDL)
Version 2.0 Part 1: Core Language,
http://www.w3.org/TR/wsdl20

[4] Extensible Markup Language (XML) 1.0,
http://www.w3.org/TR/xml

[5] UDDI Version 3.0.2,
http://uddi.org/pubs/uddi_v3.htm

[6] CORBA FAQ,
http://www.omg.org/gettingstarted/corbafaq.htm

[7] Enterprise Application Integration (EAI),
http://en.wikipedia.org/wiki/Enterprise_Application_Integration

[8] Kent Beck, “Extreme Programming Explained: Embrace Change”, Addison-Wesley, 1999.

[9] Ivar Jacobson, Grady Booch, James Rumbaugh, “The Unified Software Development Process (Addison-Wesley Object Technology Series)”, Addison-Wesley, 1999.

[10] 西澤無我, 千葉滋, 立堀道昭, “分散ソフトウェアのテストに適したアスペクト指向言語”, 情処学論, Vol.46, No.7, pp.1723–1734, 2005.

[11] WSUnit, http://wsunit.dev.java.net/

[12] Apache Axis TCPMon, http://ws.apache.org/axis/java/developers-guide.html

[13] Kiczales, Gregor; John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin (1997). “Aspect-Oriented Programming”, Proceedings of the European Conference on Object-Oriented Programming, vol.1241, pp.220-242.

[14] The Java Language Specification Third Edition,
http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html

[15] Tim Lindholm, Frank Yellin, “The JavaTM Virtual Machine Specification”, Sun Microsystems, 2004.

[16] Javassist, http://www.csg.is.titech.ac.jp/ chiba/javassist/

[17] Apache Web Services Axis, http://ws.apache.org/axis/

[18] Apache Tomcat, http://tomcat.apache.org/

[19] The Apache Tomcat 5.5 Servlet/JSP Container Class Loader HOW-TO,
http://tomcat.apache.org/tomcat-5.5-doc/
class-loader-howto.html

[20] Bertrand Meyer, 『オブジェクト指向入門』, アスキー出版局, 1990.

[21] Bugdel, http://www.csg.is.titech.ac.jp/projects/bugdel/

[22] Web Services Security,
http://docs.oasis-open.org/wss/v1.1/

[23] OASIS Web Services Secure Exchange (WS-SX) TC, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-sx

[24] Web Services Policy Framework, http://www.ibm.com/developerworks/library/specification/ws-polfram/

[25] Web Services Trust Language, http://www.ibm.com/developerworks/webservices/library/specification/ws-trust/

[26] Web Services Authorization, http://msdn.microsoft.com/library/en-us/dnwssecur/html/securitywhitepaper.asp