修 士 論 文

# レガシーシステムから<br>機能追加・修正が容易なシステムへの再構成

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

高橋 悠

2009 年 3 月

修 士 論 文

# レガシーシステムから<br>機能追加・修正が容易なシステムへの再構成

指導教官　鈴木正人 准教授

審査委員主査　鈴木正人 准教授
審査委員　落水浩一郎 教授
審査委員　青木利晃 特任准教授

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

710042 高橋 悠

提出年月: 2009 年 2 月

## 概 要

現在のシステム開発では，コスト低減のために既存システムに追加・拡張を行うことが多い．しかし，再利用を繰り返すことにより，ソースコード量は肥大化し，構造も複雑になってしまう．ソースコードが理解不能であったり，変更の適用手法が不明であったりするため，システムの保守作業でコストが増大している．これらの問題は，有効な解決策がいまだ見つかっておらず，アドホックに対応しているのが現状である．そのため，意味を踏まえた理解支援や変更手法が必要となっている．

研究目的は，保守担当者が既存の複雑なソースコードを理解しやすく，保守作業を効率的に行える構造に再構成にする手法を提案することである．

本研究では，ソースコードを対象とするシステムの再構成支援を行うため，層の分割手法と有用なオブジェクトの抽出手法の２つを提案した．層の分離とは，レガシーシステムの構造を Web システムの標準的構成である３階層アーキテクチャに分割する手法である．この手法により，レガシーシステム内のオブジェクトは機能が混合されていたのだが，プレゼンテーション層とデータベース層の記述を分離することにより，オブジェクト内の機能が一括化され，機能の追加・修正を行うことが容易な構成にすることができた．また，追加・修正の繰り返されたレガシーシステムでは，使われていないオブジェクトが多く含まれている．そのため，レガシーシステムをフロー解析して運用時に用いられているオブジェクトのみ抽出する手法を考案した．その結果，保守をする際に，不用なオブジェクトを減らし，システムの理解を行いやすくさせるられるようになった．

# 目 次

# 図 目 次

# 第1章　はじめに

## 1.1　研究の背景

現在のシステム開発は新規に構築を行うよりも既存システムに追加・拡張を行うことで実現することが多い．これはシステムを再利用することで開発コストを低減するためである．しかし，再利用を繰り返すことにより，ソースコード量は肥大化し，構造も複雑になっている．

そのため，システムの保守作業においてコストの増大を招いているのが現状である．コストの増大化する要因は，ソースコードが理解不能なことと，変更の適用手法が不明であることがあげられる．これらの問題は，有効な解決策がいまだ見つかっておらず，アドホックに対応しているのが現状である．これらの問題を解決するため，意味を踏まえた理解支援や変更手法が必要となっている．

## 1.2　研究の目的

システムの追加・修正などの要求に対する保守のコストを削減するためには，変更箇所が少なくなるようにシステムを再構成した後で修正を適用する方法が必要である．

本研究の目的は，保守担当者が既存の複雑なソースコードを理解しやすく，保守作業を効率的に行える再構成にする手法を提案することである．層の分離を行うことで役割の明確化を，通信の最適化を行うことで構造の単純化を図る．その結果，システム構造の複雑度を軽減させ，再構成容易性の向上を図る．


図 1.1: 本研究の目的概要

# 第2章　保守の現状と問題点

## 2.1　再構成の必要性

現在のシステム開発は，ソースコードを最初から書くことは少なく，多くの場合はすでにあるシステムのソースコードを再利用して行われている．

システムは再利用されるため，作り終えた段階でライフサイクルが終了するわけではない．新機能の追加やバグフィックスのための修正など，保守作業を行うことになる．しかし，構造や通信の複雑なシステムに対し作業を行おうとしても，複雑化したソースコードから変更するべき箇所を特定することは困難である．

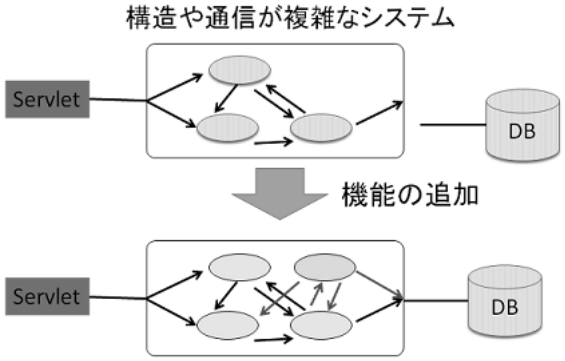


図 2.1: レガシーシステムの構成

それに対し，通信や構造が単純なシステムではソースコードが要素ごとにまとまっているため，変更するべき箇所の特定は容易に行うことができる．

このように，保守担当者がスムーズに開発に加わることができるようにシステムの再構成を行う必要がある．

## 2.2　既存技術

現在，再構成の支援として利用することのできる研究がいくつか提案されている．以下にそれらについて述べる．

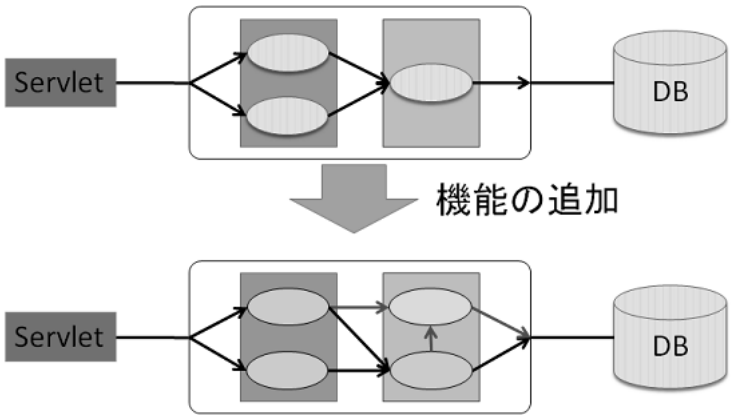


図 2.2: モダンシステムの構成

### 2.2.1 リファクタリング

リファクタリング [2] とは，ソースコードの改善手法である．外部から見たプログラムの振る舞いを変更せずに，プログラム内部の構造を改善する．リファクタリングを行うと，プログラムが整理され，潜んでいるバグの発見が容易になったり，可読性が向上したりする．

そして，リファクタリングを行うと，機能の追加を容易に行えるようになる．一般に機能追加を重ねるとソースコードは複雑になっていく．複雑なコードに対して機能追加を続けると，さらに複雑度が増してしまう．リファクタリングを行うと，構造が崩れかかったコードをきちんと整えることができる．

しかし，リファクタリングで行える機能の追加は意味を持って行われるものではない．開発者がプログラミングの熟練者であった場合は，ソースコードに対してどの部分にリファクタリング操作を加えればよいのかを考えることができる．しかし，開発者が初心者であった場合，どこに対してソースコードを適用すればよいのかが分からないことが多い．また，リファクタリング操作の適用範囲は，メソッドやクラスなど，小規模な場合が多い．システム全体の構造を変更するようなリファクタリング操作は定義されていない．

### 2.2.2 クラス間の通信形態に着目した設計レベルの再構成手法の研究

一般に機能の追加・拡張が行われる際には，既存のシステムを再利用することが多い．追加拡張が頻繁に行われると，システムは複雑さを増し，また新たな機能の拡張要求に対応することができない．大尾 [1] は，機能の追加・拡張がしやすいように設計レベルでの再構成を行う手法について研究した．

再構成手法を提案するために，通信の形態 (style)・メッセージに含まれるデータ量 (Amount)・メッセージの頻度 (Frequency) の 3 つに着目して解析を行い，システムの分析を行った．その結果，再構成手法としてクラスの分割の移譲，クラスの従属化，オブジェクトとデー

タの不整合の解消の３手法を提唱した．それらの手法に対して評価実験を行った結果，クラスの役割の移譲において既存部分の変更箇所を減少することができた．

大尾の研究対象は設計に対して行われている．しかし，現在システム開発を行っている企業などでは，設計書が書かれていなかったり，不備やドキュメントの修正を怠っていたりする．設計書を対象にした大尾の研究は，現状と合致していない．また，設計書を修正したとしても，実際にシステムを動作させているソースコードの修正は行われていない．

本研究では，ソースコードを対象に再構成を行う．そのため，設計書に不備があったり存在しなかったりするようなシステムに対しても手法の適用を行うことができる．また，ソースコードに対して再構成を行うため，設計図から修正する大尾の研究よりも，実際に稼働するシステムに対する修正時間の短縮を図ることができる．

## 2.3 変更要求技術への要望

2.2 節で示したように，これらのツールは再構成支援技術として利用するには不十分である．その理由として，リファクタリングでは大規模な修正が行えず，大尾の研究ではソースコードの再構成が行えない，などが挙げられる．

しかし，2.1 節で述べられたようなケースは多く存在する．そのため，再構成支援にはソースコードに対して，意味を持った再構成を行う必要がある．

# 第3章　レガシーシステムの構造と再構成方針

本章では，再構成の対象となるシステムの構造と，その再構成方針について説明する．既存のシステムに追加・拡張を行う場合，新技術を利用してサービスの追加を行うことが多い．しかし，システムの構成が古く，そのサービスを導入することが困難なことがある．そのような，構成の古いシステムのことを本論文ではレガシーシステムと呼ぶことにする．

3.1 節ではレガシーシステムの構造について，3.2 節ではレガシーシステムから新システムに再構成する際の方針について記述する．

## 3.1　システムの構造

本研究では，図 2.1 のような構造や通信が複雑なシステムを対象とする．そのようなシステムをレガシーシステムと呼ぶ．レガシーシステムでは，機能の追加・修正を行う際に変更するべき箇所の特定が困難であるという欠点がある．構造が複雑なため，修正すべき箇所が複数にわたり存在し，保守作業の低下を招く．

本研究では，特に Java で作成された Web アプリケーションを対象とする．Java は言語仕様の追加や変更が短期間に頻出しているため，以前のシステムの再利用や修正が困難になることが多い．また，業務用アプリケーションを構築する際に Java を用いられることが多い．業務用アプリケーションでは業務内容に合わせてシステムの保守作業が頻繁に行われる．

特に Web アプリケーションは複数の標準的構成が存在する．Web アプリケーションとは，Web ブラウザに対して入力を行うと，システムからレスポンスとして動的にページを出力するものである．

初期の Web アプリケーション開発では，標準的な構成が定まっておらず，アドホックに開発が進められてれていた．しかし，そのようなシステムに対し，保守作業を行うことは困難である修正箇所が散在しており，箇所の特定に時間を要してしまうからである．次節で再構成の方針について記述する．

## 3.2 再構成の方針

レガシーシステムではアドホックに開発が進められていった結果，機能追加や修正のしにくい物となっている．このようなレガシーシステムの保守を行うために問題が生じるが，本論文では下記の３点に着目した．

1. ひとつのオブジェクトに複数の役割が混在している
2. 再構成の手掛かりを見つけだすことができない
3. 類似したクラスが複数個存在する．

レガシーシステムに対して追加・修正要求が来た場合を考える．１では，複数の役割が混在しているオブジェクトに対して追加・修正要求が発生したとしても，どのオブジェクトを変更するべきかを特定することが困難である．そのため，階層ごとに分割を行うことにする．階層ごとに分割されているため，追加・修正要求に対して修正すべきオブジェクトの特定が容易に行うことができる．

２の場合，追加・修正要求が生じた際に，メッセージ通信が複雑化しているためどこを変更すべきかを特定することが困難である．そのため，オブジェクト間のメッセージ通信を解析してシーケンス図にまとめることにする．メッセージ通信が図示されるため，修正すべきオブジェクトの特定を容易に行うことができる．

３の場合，追加・修正要求が生じた際に，類似した複数のクラスすべてを検査する必要がある．そのため，類似したクラスを１つにまとめる．１つのクラスに凝縮されるため，追加・修正要求に対して変更すべきオブジェクトを最小に抑えることができる．

3.2.1 節では階層の分割について，3.2.2 節ではオブジェクト間の通信解析について，3.2.3 節ではエンティティクラスのインライン化について述べる．

### 3.2.1 階層の分離

階層の分割について述べる．階層を分割する際に，Web アプリケーションの標準的仕様としてよく用いられている３階層アーキテクチャを用いることにした．以下に３階層アーキテクチャについて述べる．

３階層アーキテクチャとは，アプリケーションを３つの階層に分けて実装する仕組みである．それぞれの機能を別個のマシンで実行することができる．ユーザーインターフェース部分を実現するのが「プレゼンテーション層」．プレゼンテーション層ではクライアント側の GUI を実現する．データに対して処理を行うのが「ビジネスロジック層」．ビジネスロジックに該当する．データベースを置くのが「データベース層」．サーバー側の RDBMS がこの階層にあたる．3 階層アーキテクチャの利点として，

- データと処理の最適分散により処理性能が向上する

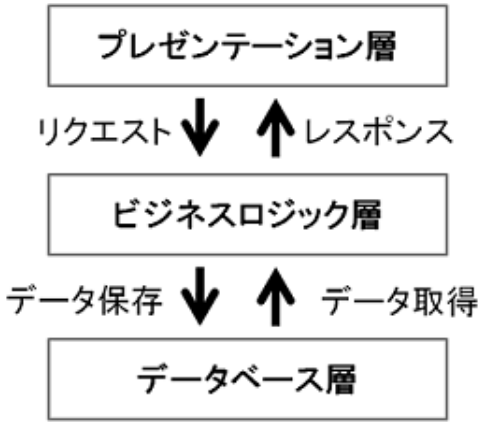


図 3.1: ３階層アーキテクチャの構成

- 処理ロジックの集中管理によりアプリケーションの保守性が向上する
- 各モジュールを並行して開発すれば、生産性も向上する

などがあげられる．

レガシーシステムが３階層アーキテクチャにしたがって構成されいていない場合，３階層アーキテクチャを採用するために機能の分割を行う必要がある．

### 3.2.2 オブジェクト間の通信解析

オブジェクト間の通信解析について述べる．レガシーシステムでは，ドキュメントをきちんと残しておらず，正確な情報はソースコードしか無い場合が多い．そのため，オブジェクト間の通信を解析する必要がある．

レガシーシステムでのメッセージ通信は例えば図 3.2 のようになっている．

このように複雑化したメッセージ通信では，新たに追加・拡張作業を行う際にどこに追加をすればよいのか把握することが難しい．そのため，メッセージ通信を可視化することでどこを再構成するべきかを明確化する．メッセージ通信の解析を行うために，まず通信の解析を行う．通信の解析方法はデータフローを用いることにする．そして，解析結果を UML のシーケンス図にまとめることでオブジェクト間のメッセージ通信の可視化を行う．次に，フロー解析について述べる．

プログラム中の依存関係を明らかにする解析をフロー解析と言う．依存関係には，コントロールフローとデータフローが存在する．コントロールフローとは，各行の実行がどの行に依存するかを表現したものである．データフローとは，各行内の識別子の値が，どの行のどの識別子の値に依存するかを表現したものである．フロー解析は，構文解析よりは意味的な要素が関係してくる．

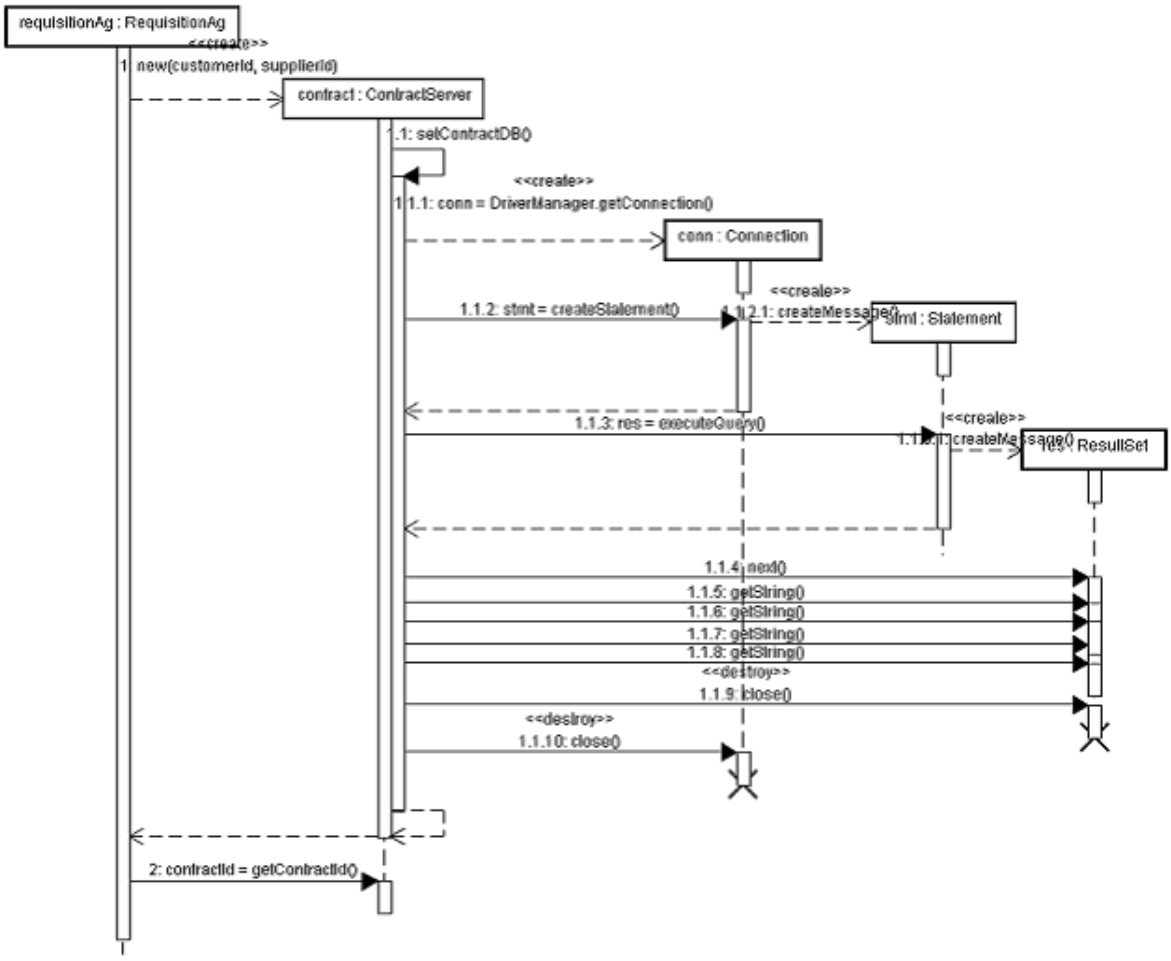


図 3.2: 複雑なメッセージ通信例

### 3.2.3　エンティティクラスのインライン化

エンティティのインライン化について述べる．階層の分割を行った結果，クラスが大量に作成されることがある．特に，類似したデータを保持するエンティティクラスが複数存在すると，データの追加・修正を容易に行うことができない．そのため，類似したエンティティクラスをインライン化する必要がある．リファクタリング操作のクラスのインライン化を用いる．

# 第4章　再構成手法

再構成手法を考案するため，実際にレガシーシステムの解析を行った．4.1 節では再構成手法を抽出するために用いたレガシーシステムの構成について述べる．4.2 節ではレガシーシステムの改善点について述べる．4.3 節ではレガシーシステムから抽出された再構成手法について述べる．

## 4.1　対象システムの構成

再構成手法を考案するにあたり，電子商取引システムを対象とする．本節では，電子商取引システムが持つサービスの種類とその内容について説明する．

### 4.1.1　システムの概要

今回対象にする電子商取引システムの概要について述べる．図 4.1 に電子商取引システムの概要図を示す．

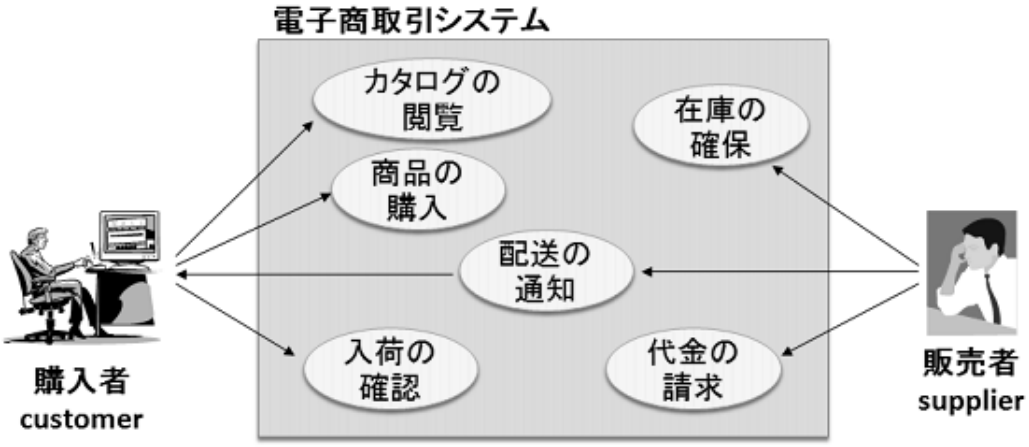


図 4.1: 電子商取引システムの概要

対象システムは，業者間での電子商取引を行う，いわゆる BtoB の E-commerce システムである．しかし，実際の金銭取引は行わず，信用取引とする．金銭取引を行うバンクシステムが別に存在する．

システムのアクターは，購入者（customer）と販売者（supplier）のふたつである．
購入者は

- カタログの閲覧
- 商品の購入
- 配送の確認

のサービスを利用できる．以下，各サービスについての詳細を述べる．

カタログの閲覧

購入者は Web ページでカタログの検索，商品の選択を行うことができる．カタログの検索を行うとシステムは販売者のカタログデータベースからデータを取得し，検索結果ページを作成して購入者に表示する．購入者は購入したい商品を選択し，ショッピングカートに入れる．

商品の購入

購入者はショッピングカートに入れられた商品の注文を行える．システムは発注書（Requisition）を作成し，販売者に発注情報を送信する．販売者側のシステムで注文書 (DeliveryOrder) を作成する．

配送の確認

購入者が商品を受け取った際，Web ページから配送到着通知を行う．システムは購入者の発注データに対して発注状態の更新を，販売者の注文書データに対して注文状態の更新を行う．

販売者は

- 在庫の確保
- 出荷の確認
- 配送の確認

のサービスを利用できる．以下，各サービスについて詳細を述べる．

在庫の確保

システムで在庫の確保を行い，在庫データの更新をする．注文状態の更新を行う．

出荷の確認

配送を行う際，販売者は出荷確認をシステムに対して行う．システムは販売者の注文書データと購入者の発注書データを更新する．

代金の請求

システムは購入者の配送確認メッセージを取得した際，バンクシステムに金銭取引を依頼する．販売者の注文書データと購入者の発注書データを更新する．(バンクシステムは対象外)

システムの利用者は購入者と販売者である．販売者はシステム上にカタログを配備している．購入者は，そのカタログを Web 上で閲覧し，購入商品を選択し発注を行う．発注書が販売者に届いたら，販売者は在庫の確認をし，在庫がある場合は発送を行う．購入者は商品が届いたら受領通知を行う．受領通知が行われたら，販売者は請求書を購入者に送る．販売者が銀行に対して支払いを行う．

本論文では，システムが大きすぎるため，以下商品の購入サービスについてのみ取り扱うことにする．

### 4.1.2 システムのクラス構成

電子商取引システムで用いられている要素技術と構成について述べる．

**サーブレット** Web ブラウザと連動してクライアントと情報のやり取りを行いながら動的にページを生成する Java オブジェクト．

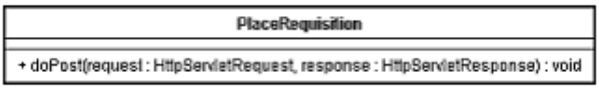


図 4.2: サーブレット

**ビジネスロジッククラス** ビジネスロジッククラスは，ビジネスルールを記述するクラスである．表示やデータの保存とは独立して論理に関する記述をまとめる．

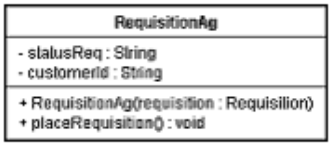


図 4.3: ビジネスロジッククラス

**エンティティクラス**　エンティティクラスとは，データの保持をつかさどるクラスである．エンティティクラスは，データベースに永続化されることが多い．また，クラスの凝集度が高く，変更を行うことが少ない．

プリミティブなエンティティクラスはデータとそれにともなうアクセッサのみで構成されているが，レガシーシステムではデータベースアクセスクラスに記述されるべき箇所やサーブレットが行うべき画面の表示に関する記述が含まれていた．

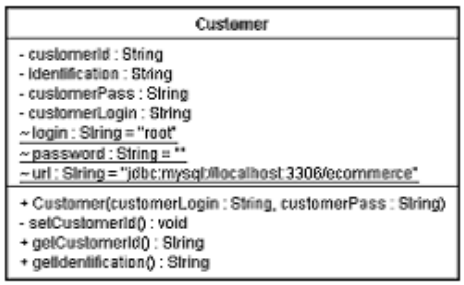


図 4.4: エンティティクラス

**データベースアクセスクラス**　データベースにアクセスするための記述をまとめたクラス．以下，データベースアクセスクラスのことをDBA クラスと表記する．基本的なDBA クラスでは，データベースに対して，データの作成・更新・削除・追加を行う．

プリミティブなDBA クラスはデータベースアクセスにかかわる記述のみで構成されるが，レガシーシステムでは，エンティティクラスが行うべきデータの保存や，サーブレットが行うべき画面の表示に関する記述が含まれていた．

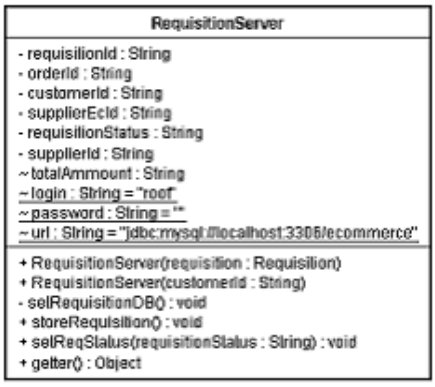


図 4.5: データベースアクセス

**コンテナ**　Web コンテナとは，Web システムの実行環境を提供するものである．コンテナ内のオブジェクトに対し，セッション状態の記録や，マルチスレッド管理を行うものである．本研究ではコンテナに対しての解析は行わないことにする．

### 4.1.3　システムの相互作用

商品の購入サービスのコミュニケーション図を図 4.6 に示す．商品の購入を行うのに対

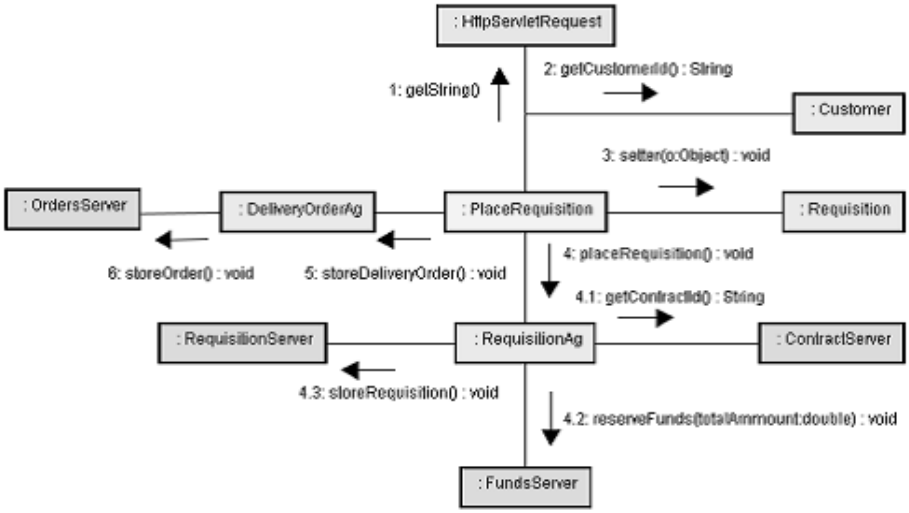


図 4.6: 商品の購入サービスのコミュニケーション図

し，関連するクラスを次に述べる．

**PlaceRequisition**　商品の購入画面を作成するサーブレット．

**HttpServerRequest**　利用者が入力したデータを保持するクラス．

**RequisitionAg**　請求書（Requisition）を作成するためのエージェント．

**DeliveryOrderAg**　注文書（DeliveryOrder）を作成するためのエージェント．

**Customer**　顧客データを保持するエンティティクラス．再構成前のシステムでは，顧客情報を保持するサーバとやり取りする DBA の記述が含まれている．

**Requisition**　請求書データを保持するエンティティクラス．再構成前のシステムでは，請求書情報を表示するサーブレットの記述が含まれている．

**RequisitionServer**　発注書情報を保持するサーバとやり取りするDBAクラス．再構成前のシステムでは，請求書データを保持するエンティティクラスと一体化している．

**OrderServer**　注文書情報を保持するサーバとやり取りするDBAクラス．再構成前のシステムでは，注文書データを保持するエンティティクラスと一体化している．

**ContractServer**　契約情報を保持するサーバとやり取りするDBAクラス．再構成前のシステムでは，契約データを保持するエンティティクラスと一体化している．

**FundsServer**　資金情報を保持するサーバとやり取りするDBAクラス．再構成前のシステムでは，資金情報データを保持するエンティティクラスと一体化している．

## 4.2　システムの改善理由

電子商取引の問題点について述べる．

本システムは，サーブレットオブジェクトを介してユーザとやり取りを行うシステムである．サーブレットで動的にページの作成が行われるように作成するべきであるが，他のJavaオブジェクト内でもページ生成の記述がある．また，データベースアクセス専用のクラスを作成するべきであるが，サーブレット同様他のJavaオブジェクト内でデータベースアクセスに関する記述がある．１つのオブジェクトが機能を複数持ちすぎているため，構造を把握することが困難である．

## 4.3　再構成ルールの作成

本節では，構成の改善点をもとに検討した再構成手順をルールの形式で述べる．

### 4.3.1　階層の分割

#### 動機

保守が容易に行えるような構造になっていない．１つのオブジェクトに役割が複数ある．構造が複雑化しており，システムを把握しづらい．例えば，データベースにアクセスしているオブジェクト内に，動的ページ作成を行っている記述がされているような場合などである．そのようなシステムでは，保守を行う際にどのオブジェクトを調べればよいのかが分かりにくい．保守をしやすいように，機能ごとに層の分離を行う必要がある．

**解法**

Web アプリケーションの標準的構成である３階層アーキテクチャを元に記述の分離を行う．オブジェクトからデータベース層，プレゼンテーション層の特徴を持つ記述を分離し，別オブジェクトにする．ここで，データベース層の特徴を持つ記述とは，SQL 構文[2]およびその構文から波及するすべての変数を指す．プレゼンテーション記述とは，HTML 構文[3]およびその構文から波及するすべての変数を指す．

**効果**

層の分割を行うことにより，オブジェクトごとの役割が明確化する．その結果，保守担当者はシステムの構造を把握しやすくなり，再構成にかかる時間が短縮する．しかし，分離しなくてもよいオブジェクトまで分離してしまうという欠点もある．

**手順**

手順１　特徴的な文字列を含む行の検出層の分離を行う最初の手掛かりとして，各層の特徴的な文字列を含む行の検出を行う．データベース層に記述するべきソースコードは SQL 構文[2]が，プレゼンテーション層に記述するべきソースコードは HTML 構文[3]が記述されている．それらを検索し，検索結果を解析の始点とする．後にこれら記述を移行する際の目印となるように，プレゼンテーション層には p を，データベース層には d を検出した行に付与する．図 4.7 に SQL 構文を含む行の検出例を示す．

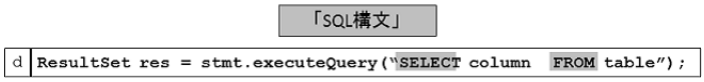


図 4.7: SQL 構文を含む行の検出例

手順２　変数を対象とするフロー解析

手順１で検出された行のうち，代入文の左辺に出現する変数を対象としてフロー解析を行う．対象となる変数の検出範囲によって以下の４つの場合にわけて本手順を繰り返す．

1. 変数が別の代入文の右辺に含まれる場合
   レシーバを解析対象に追加する．代入文を含む行に p または d の目印を伝搬させる．図 4.8 に例を示す．

[2]SELECT, CREATE, UPDATE, DELETE, WHERE, FROM など
[3]$\langle \cdots \rangle$ で始まり $\langle / \cdots \rangle$ で終わる文字列

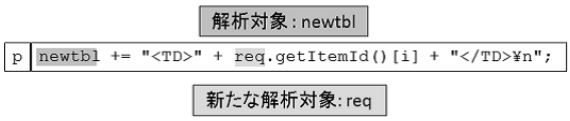


図 4.8: 別の代入文の右辺に含まれる変数のフロー解析例

2. 変数が制御ブロックに含まれる場合

   制御ブロック内部のすべての変数を解析対象に追加する．追加されたすべての変数を右辺に含む行に p または d の目印を伝搬させる．図 4.9 に例を示す．

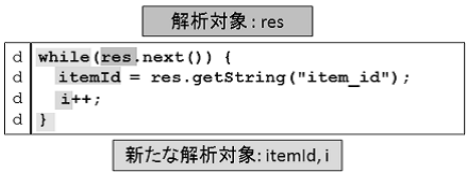


図 4.9: 制御ブロックに含まれる変数のフロー解析例

3. 変数がフィールド変数の場合

   分離を行う際に単純に移行を行えないため，変数に e の目印を付与する．また，フィールド変数を参照している行に p または d の目印を伝搬させる．図 4.10 に例を示す．

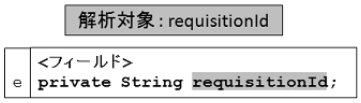


図 4.10: フィールド変数のフロー解析例

4. 変数がフロー解析結果でそれ以上検出されなかった場合

   フロー解析を終了する．

手順 3　フィールド変数のカプセル化

e の目印が付いているフィールド変数に対しカプセル化を行う．リファクタリング操作 [2] の ⟨⟨encapsulate field⟩⟩ を用いる．

手順 4　メソッドの抽出

p または d の目印が付いている行に対し，別メソッドとして抽出を行う．リファクタリング操作 [2] の ⟨⟨move method⟩⟩ を用いる．

手順 5　抽出したメソッドの移動

手順 4 で p 及び d の目印が付いたメソッドを作成した場合，新たにクラスを作成する．p または d の目印が付いている行に対し，別メソッドとして抽出を行う．リファクタリング操作 [2] の ⟨⟨move method⟩⟩ を用いる．

### 4.3.2　オブジェクト間の通信解析

#### 動機

構造が複雑なため，データフローが把握できない．再構成の手掛かりを見出すことができない．

#### 解法

ユーザがアクセスするページを始点に，フロー解析を行う・解析結果をシーケンス図にまとめる．

#### 効果

データフローが明確化する．不適切な通信のある箇所が特定できる．

#### 手順

手順 1　フロー解析の始点の決定

設定ファイル（web.xml など）に記述されているサーブレット定義記述[4]をキーワードにサーブレットの検出を行う．検出されたサーブレット内にあるメソッド doGet 及び doPost 内に存在する変数をフロー解析の始点とする．

index.html など、ユーザが最初に起動するページを開始点とし，そこから呼び出される Java オブジェクトを有用なオブジェクト候補として検出を行う．

手順 2-1　サーブレットを始点とするフロー解析

変数が左辺に出現する行に対し，右辺に出現する変数を解析対象とする．

解析対象となる変数によって解析手順が変更する．以下の 4 つの場合に分けて手順 2 を繰り返す．

1. 変数が別の代入文の右辺に含まれる場合
   代入式のレシーバを解析対象の変数に追加する．

[4] ⟨ servlet-class ⟩ · · · ⟨/servlet-class⟩ で終わる文字列

2. 変数がコンストラクタに含まれる場合
   コンストラクタ内に含まれる関数を解析対象に追加する.
3. 変数がメソッド呼び出しを行っていた場合
   メソッド内に含まれる関数を解析対象に追加する.
4. 変数が初期化部分に到達した場合
   フロー解析を終了する.

### 4.3.3 エンティティクラスのインライン化

**動機**

類似したクラスが複数個存在する．昨日の追加・修正を行う際，すべてのクラスに対し修正を行わなくてはならない.

**解法**

類似した複数のクラスをひとつにまとめる.

**効果**

昨日の追加・修正を行う際に，修正箇所が１箇所にまとまっている．変更が容易になる.

**手順**

手順１　フィールド変数の共通性検査

フィールド変数が以下の条件を満たした場合，インライン対象のクラスと判定する.

1. フィールド変数が完全一致する場合
   あるオブジェクトと別のオブジェクトのフィールド変数が完全に一致する場合，同一オブジェクトと判定する.
2. フィールド変数が包含的に一致する場合
   あるオブジェクトが別のオブジェクトのフィールド変数を包含する場合，変数の多いオブジェクトに対してインライン化する.
3. フィールド変数が部分一致する場合
   オブジェクトの類似度が高い場合，両方のフィールド変数を統合する．ここで，類似度の測定法は未だ確立しておらず，定性的に判断を行っている．具体的には，1,2 個程度の違いなら同一のオブジェクトと判断する.

# 第5章　適用実験と有用性の評価

第 4 章で提案したルールの有用性を検査する．適用実験の例として電子商取引システムに適用することにより，有用性の調査を行う．5.1 節で適用実験を行い，5.2 節で手法の評価を行う．

## 5.1　適用事例

4.3 章で提案したルールの適用実験を行う．実験例として，商品の購入に関するに新サービスの追加要求があったと仮定する．

従来のサービスでは，購入時に資金の確認を行っていた．しかし，新サービスでは非常用の資金プールから支払うサービスを導入する．システムは限度額を超過していないかを確認するだけでよい．

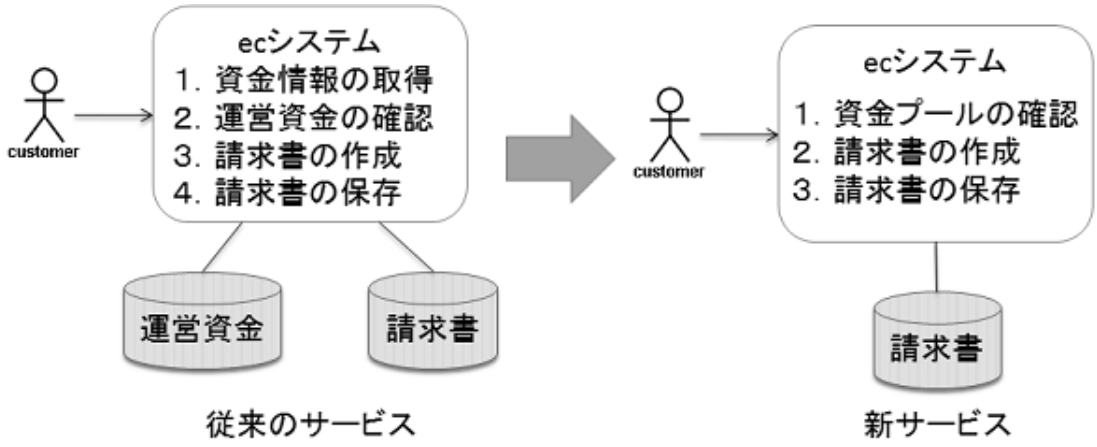


図 5.1: 新サービスの概要図

従来のサービスでは，運営資金サーバから資金情報を取得し，購入資金と運営資金の確認を行い，その後請求書を作成し請求書データベースに格納するという手順で行われていた．

それに対し，新サービスでは，運営資金サーバから資金情報を取得せず，資金プールを新たに作成することにする．資金プールのデータは請求書オブジェクトのフィールドに格納する．新サービスではシステムは限度額を超過していないかを確認するだけでよい．

次に，それぞれの手法についての適用実験の説明を行う．

### 5.1.1 層の分割手法の適用事例

#### 手順 1　特徴的な文字列を含む行の検出

SQL 構文をキーワードにソースコードの検出を行った．その結果，次の解析対象となる変数 res,stmt を得ることができた．後に抽出する際の目印となるように，d のマークをその行に追加した．

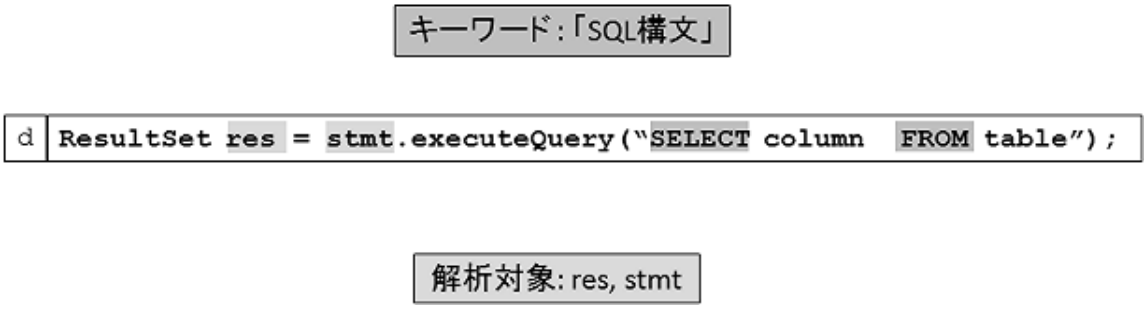


図 5.2: 特徴的な文字列を含む行の検出

#### 手順 2　変数を対象とするフロー解析

res,stmt の変数を対象にフロー解析を行った．両変数は代入文の右辺に出現するため，操作 1 を適用した．その結果，新たな解析対象 conn, column を得た．

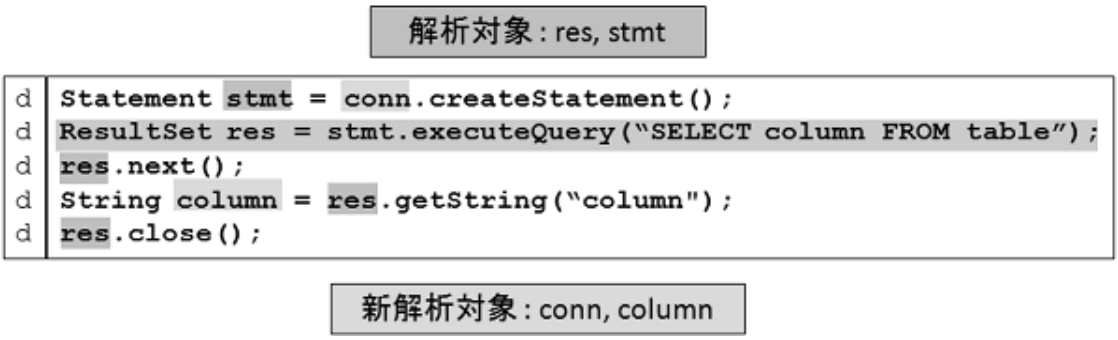


図 5.3: 代入文の右辺に出現する変数 res, stmt を始点とするフロー解析

次に，conn の変数を対象にフロー解析を行った．conn は制御ブロックに含まれるため，操作 2 を適用した．その結果，新たな解析対象制御構造文 if を抽出した．まず，if ブロッ

ク内のすべての行に対して移行を行う際の目印となるようにdのマークを追加した．そしてifブロック内に含まれるすべての変数を解析対象に追加する．今回はすべての変数が解析対象にすでに追加されていた．


図 5.4: 変数が制御ブロックに含まれる場合のフロー解析

次に，conn と column の変数を対象にフロー解析を行った．conn と column は代入文の右辺に含まれているため，操作 1 を適用した．その結果，新たな解析対象 url, login, password を得ることができた．後に抽出する際の目印となるように，d のマークをその行に追加した．


図 5.5: 代入文の右辺に出現する変数 conn, column を始点とするフロー解析

次に，url, login, password の変数を対象にフロー解析を行った．url, login, password はフィールド変数となっているため，操作 3 を適用した．この操作では新たな解析対象変数を得ることができなかった．後にフィールド変数のカプセル化を行う際の目印となるように e のマークを追加した．


図 5.6: フィールド変数を始点とするフロー解析

以上の操作ですべての解析対象のフロー解析を行うことができたため，手順 3 に移行する

### 手順 3　フィールド変数のカプセル化

手順 2 で e マークを追加した箇所のカプセル化を行う．この操作はリファクタリング操作の ⟨⟨ フィールド変数のカプセル化 ⟩⟩ を用いた．


図 5.7: フィールド変数のカプセル化

### 手順 4　メソッドの抽出

手順 2 で d マークを追加した箇所をメソッドとして抽出する．この操作はリファクタリング操作の ⟨⟨ メソッドの抽出 ⟩⟩ を用いた．

```
<storeRequisitionメソッド内>
tempStoreRequisition();
```

```
<メソッドtempStoreRequisitionの作成>
private void tempStoreRequisition(){
  Connection conn = DriverManager.getConnection
                        (getUrl(), getLogin(), getPassword());
  if (conn != null) {
    Statement stmt = conn.createStatement();
    ResultSet res = stmt.executeQuery("SELECT column FROM table");
    res.next();
    String column = res.getString("column");
    res.close();
  }
}
```

図 5.8: メソッドの抽出

**手順 5　メソッドの移動**

手順 4 で作成したメソッドを新たなクラスに移動する．この操作はリファクタリング操作の ⟨⟨ メソッドの移動 ⟩⟩ を用いた．

```
<storeRequisitionメソッド内>
RequisitionServerDB reqDB = new RequisitionServerDB();
reqDB.storeRequisition();
```

```
<新クラスRequisitionServerDB>
public class RequisitionServerDB {
  public void storeRequisition() {
    (省略)
  }
}
```

図 5.9: メソッドの移動

### 5.1.2　オブジェクト間通信の解析の適用事例

オブジェクト間通信の解析を行い，シーケンス図を作製した．この手法を，再構成を行う前のレガシーシステム・階層の分割を行った後のシステム・エンティティクラスのインライン化を行ったクラスに対して行った．

### 5.1.3　エンティティクラスのインライン化の適用事例

エンティティクラスのインライン化を行い，類似したクラスのインライン化を行った．

## 5.2　手法の評価

上記メッセージを行った．以下にそれぞれの手法の評価を行い，最後に新サービスの追加を行った際の評価を行う．

### 5.2.1　階層の分割の評価

手法を適用する前後のクラス図を比較する．図 5.10 に再構成前のクラス図を，図 5.11 に再構成後のクラス図をそれぞれ示す．

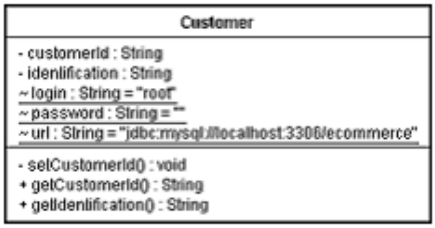


図 5.10: 再構成前のクラス図

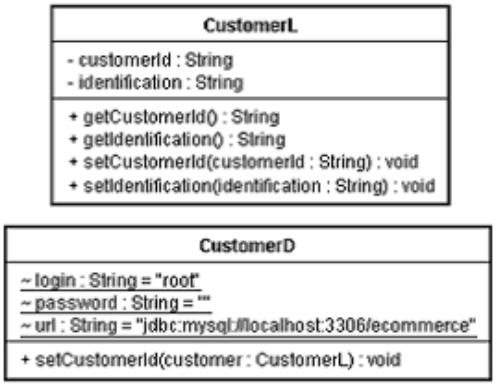


図 5.11: 階層の分離後のクラス図

再構成前のクラス図では，エンティティとデータベースアクセスの役割が混在していたのに対し，分割後は 1 つのクラスに 1 つ役割をもつようになった．

### 5.2.2　オブジェクト間の通信解析の評価

層の分割を行った後にフロー解析を行い，シーケンス図を作成した．その結果，類似したメッセージ通信が頻繁に行われていることが分かった．図 5.12 に層の分割後のシーケンス図を示す．

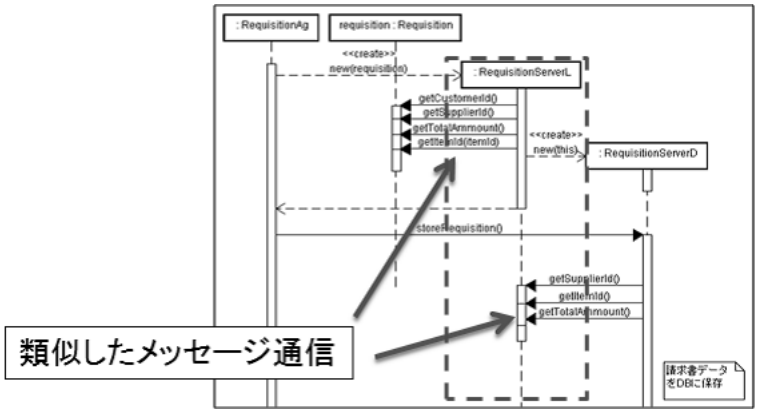


図 5.12: 階層の分離後のシーケンス図

Requisition クラスと RequisitionServerL クラスを比較したところ，フィールドがほぼ同等なエンティティオブジェクトであることが分かった．この手法により，次に再構成するべき箇所の特定を容易に行えるようになった．

### 5.2.3　エンティティクラスのインライン化の評価

5.2.2 章で見つけたほぼ同等なエンティティオブジェクト 2 つに対してクラスのインライン化を行った．

片方の不用なオブジェクトの削除を行ったため，追加・修正要求に対して，簡単に答えることができるようになった．

### 5.2.4　新サービスの導入の評価

Requisition クラスに対して，資金プールを扱う変数を追加した．

再構成前のクラスは役割が不明確だったため，追加箇所の特定が困難であった．しかし，再構成後はエンティティとメソッドが役割ごとにまとまっているため，追加箇所の特定を容易に行うことができた．

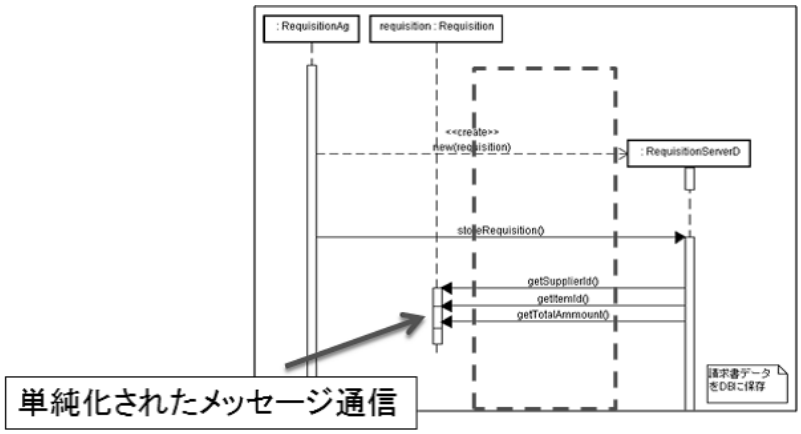


図 5.13: 階層の分離後のシーケンス図

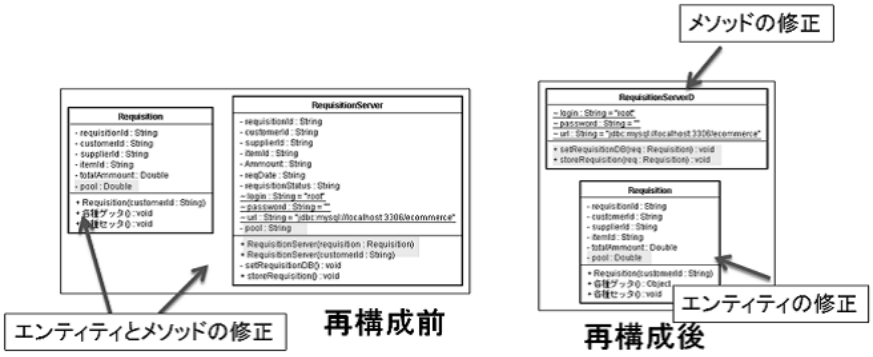


図 5.14: 新サービス追加の

# 第6章　おわりに

## 6.1　まとめ

本研究では，ソースコードを対象とするシステムの再構成支援を行うため，階層の分割と通信の最適化を行うための手法を考案した．階層の分割として，レガシーシステムの構造を Web システムの標準的構成である 3 階層アーキテクチャに分割する手法を提案した．レガシーシステム内のオブジェクトは機能が一括していなかった．そのため，本研究ではプレゼンテーション層とデータベース層の記述を分離する．それにより，オブジェクト内の役割が明確化し，機能の追加・修正を行う際に変更箇所の特定を容易に行えるようにした．

通信の最適化では，複数の類似したクラス間でメッセージ通信が複雑化していたため，類似したクラスをインライン化する手法を考案した．それにより，オブジェクト間の通信が単純化し，機能の追加・拡張を行う際に変更箇所の特定を容易に行えるようにした．

そして，手法の有用性を確認するための実験を行った．再構成を行ったシステムの変更容易性を検査して，変更すべき箇所の減少を確認した．

## 6.2　今後の課題

### 6.2.1　最適化基準の定量的即手と評価

通信の最適化を行う場合，現在の手法では通信量を測定していない．現在の手法では通信の回数のみを対象にメッセージ通信を測定しているが，String 型，Array 型，Map 型などでは通信量が異なる．

また，評価基準の閾値画定まっていないという問題がある．閾値を決定するための調査を行う必要がある．

### 6.2.2　多様なアーキテクチャへの適用

現在の手法では，3 階層アーキテクチャを用いて再構成を行った．それ以外のアーキテクチャへ再構成する手法を検討する必要がある．

# 謝辞

本研究を行うにあたり，終始ご指導していただきました鈴木正人准教授に深く感謝申し上げます．また，研究を行うにあたり有益なご助言をいただきました落水浩一郎教授に心よりの感謝を申し上げます．研究を勧めるにあたり貴重なご意見をいただきました研究室の皆様に心よりの感謝をいたします．最後に，大学院での生活を援助していただいた両親に感謝をいたします．

# 参考文献

[1] 大尾健介, クラス間の通信形態に注目した設計レベルの再構成手法の研究, 北陸先端科学技術大学院大学 情報科学研究科 修士論文, 2007.03.

[2] Martin Fowler 著, (児玉公信，友野晶夫，平澤章，梅沢真史 訳)，リファクタリング, ピアソンエデュケーション，2000