ソフトウェア工学　136－12
（2002．3．7）

リファクタリングとコンポーネント技術による既存ソフトウェアの拡張手法

落合 竜一 †　　鈴木 正人†

本稿では既存の即時的拡張を繰り返して複雑になったソフトウェアの拡張のための手法を提案する。提案手法では、まずリファクタリングを行い拡張を行い易い構造に変更する。これにより更なる拡張のコストの上昇をおさえる。次にコンポーネント化により以後の拡張をコンポーネントを単位としたものに限定し、拡張作業のコストを低減する。拡張の容易性を客観的に表すメトリクス RF(Refactoring Factor) を導入し、即時的な拡張の例としてレンタル料金計算システムに“ レンタル物件の種類増加 ”という同一の拡張を、行う場合について提案手法の適用前と適用後の各システムに施した際の変更コストを RF により比較した。本手法の適用後のソフトウェアへの拡張では機能の複製が存在せず、RF の減少、及び拡張コストの低減が確認された。

An extension method of software by refactoring and component-based techniques

RYUUICHI OCHIAI†　　MASATO SUZUKI†

In this paper, we will introduce a new method applicable to complex softwares which are result of successive ad-hoc extensions. Refactoring is used for making structure of the software simple and easy to understand. Component technologies are used to restrict the complexity of extension to that of exchanging components. As the consequence, we can reduce the costs for extension.

Refactoring Factors(RF), which is calculated from several parameters of dependencies among classes, is introduced as a metrics showing “easiness of extension.” We will show the effectiveness and feasibility of this method and metrics using a system for video-rental shops. Comparing RF’s by applying the same extension that is “add a new rental” to the system both the original and simplified by our method,we found that RF and replication of functions are also reduced.

1. はじめに

ソフトウェア開発の過程では変更要求が絶えず発生している。従来の開発手法では要求が発生する度に即時的な拡張を繰り返していたが、その結果としてソフトウェアの構造が複雑化する。このようなソフトウェアに更なる拡張を行う場合、拡張コストの増大が問題となっている。

本稿では既存ソフトウェア拡張のための手法を提案する。複雑化したソフトウェアを再構成する技術としてリファクタリングを用い、提案する適用基準はクラス間の依存関係の減少、メソッドからのプリミティブの抽出、ポリモーフィズムの導入、及びソフトウェアの構成要素の分類を行う。また、リファクタリング適用中構成要素をユーザインターフェース (UI) とデータストアに分類し、データストアを利用するメソッドを統合したクラス (コントロール) を作成し、コントロールからデータストアへアクセスする部分 (アクセッサ) を分離することで各々にコンポーネント技術 (インターフェースの定義) の適用を容易にし拡張の体系化が促進される。

本稿は以下のように構成される。2 節ではソフトウェアの拡張と従来手法における問題、提案手法の要素技術であるリファクタリングとコンポーネントを説明する。3 節では提案する拡張手法を説明し、4 節では例題としてレンタルビデオ店の料金計算システムをあげ、“レンタル物件の種類増加” という同一の拡張を拡張手法の適用前と適用後の各システムに施した際の変更コストを比較することによって提案手法の有効性を確認する。5 節でまとめと今後の課題を述べる。

2. ソフトウェアの拡張

従来の開発手法ではソフトウェアの拡張は即時的に繰り返され、その結果としてソフトウェアの構造が複雑になっている。図 1 はこの様子を模式的に表したものである。この図では拡張はソフトウェアの底辺に追加することで表現される。S や S' に対しては拡張を問題なく行えるが、S'' の底辺にそのままでは C を追加できない為、C を変形する必要がある。これが拡張コストの増加である。

† 東京工業大学 大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology

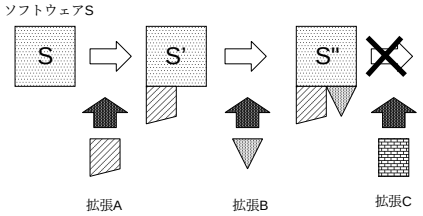

図 1　即時的な拡張により複雑になるソフトウェア

2.1　リファクタリング

リファクタリング[1] とは統制された方法でコードを変更し設計を改善する技術であり、「外部から見たときの振る舞いを保ちつつ、理解や修正が簡単になるように、ソフトウェアの内部構造を変化させること」と定義されている。

また、リファクタリングの個々の技術を**リファクタリング操作**と呼ぶことにする。なお、本稿で**「操作」**は特に断らない限りリファクタリング操作を指すものとする。

2.2　コンポーネント

コンポーネント[2] とはソフトウェア部品でありさまざまな定義があるが、本稿では以下の特徴をもつものをコンポーネントと呼ぶことにする。

- 一つ以上のクラスからなる
- 独立して交換可能である。
- 何らかの機能又はサービスを提供している。

3.　提案する拡張手法

本手法は即時的な拡張の繰り返しで構造が複雑になったソフトウェアの拡張コストを低減することを目的としている。そのため、リファクタリングを行い、拡張を行い易い構造に変更する。これにより拡張を繰り返しても更なる拡張のコストが上昇しなくなる。そして、コンポーネント化によってコンポーネントを単位とした拡張を可能にし、拡張作業のコストを低減する。

本手法は以下の手順によって行う。

(1)　リファクタリングの適用によってプログラムの意味を変えることなく構造を単純化する。

(2)　1 の結果の構成要素の中から、コンポーネントの候補となるべき部分を抽出する。

(3)　各コンポーネントの候補をコンポーネント化する。

図 2 が提案手法の概要である。手法の詳細を例題を用いて説明する。

3.1　例　　題

ビデオレンタル店の料金計算システムを用いる。

仕様　これは、顧客の名前を入力するとその顧客が現在借りている映画のタイトルと料金を記したレシートを表示するものである。料金は映画の分類とレンタル期間によって以下のように計算される。

レンタル期間が３日未満　基本料金

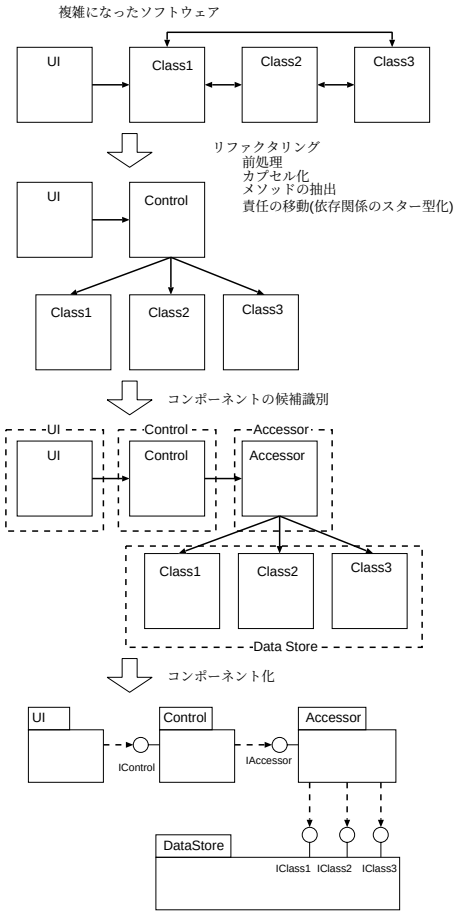

図 2　提案手法の手順

レンタル期間が３日以上　基本料金
+(レンタル期間−2)× 追加料金

表 1　料金表

映画の分類	基本料金	追加料金
New_Release	300 円	300 円
Regular	200 円	200 円
Childrens	200 円	200 円

プログラム　図 3 に示すクラス図を持つプログラムが与えられているとする。

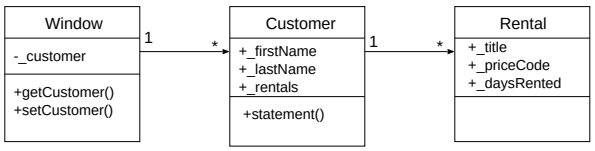

図 3　料金計算システムのクラス図

Customer　顧客を表すクラス

- `_firstName`,`_lastName`:顧客の名前を表す属性

- _rentals:レンタル状況を表す属性
- statement():料金を計算し、レシートを作成するメソッド

Rental　レンタルの情報を表すクラス。

- _title:借りている映画のタイトルを表す属性
- _priceCode:借りている映画の分類を表す属性
- _daysRented:レンタル期間を表す属性

Window　画面の表示を行うクラス。

- _customer:接続する Customer を表す属性

実行例　顧客”山田太郎”が以下の映画を借りているとする。

Top Secret　分類 REGULAR、期間３日

Flying High　分類 NEW_RELEASE、期間３日

この場合の各クラスのインスタンスを図 4 に示し、実行例を図 5 に示す。

山田太郎:Customer

属性	値
_firstName	taro
_lastName	yamada
_rentals	{ rental1,rental2 }

rental1:Rental

属性	値
_title	Top Secret
_priceCode	REGULAR
_daysRented	3

rental2:Rental

属性	値
_title	Flying High
_priceCode	NEW_RELEASE
_daysRented	3

図 4　例題のインスタンス

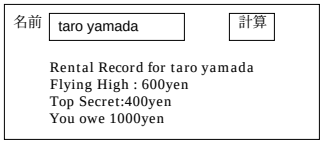

図 5　料金計算システムの実行例

3.2　リファクタリング

[1] ではリファクタリングはデザインパターンの構成を目標としていたが、本手法ではこれを拡張し、コンポーネント化が容易になるようにリファクタリングを適用する。このため適用基準を以下のように定める。

依存関係の減少　変更の影響が及ぶ範囲は、クラス間の依存関係が存在する部分である。従って、依存関係を減少させることによって変更の局所化が可能となる。変更の局所化によってソフトウェアの対象となる構成部品のみの変更で、変更要求に対応することが可能となる。従って、変更の局所化はソフトウェア部品のプラグイン可能化のために必要であると言える。本手法では、依存関係を減少し、変更を局所化するために依存関係の限定を行う。クラス間の依存関係をスター型にすることで、ソフトウェア中の依存関係の総和を減少させる。

カプセル化　変更を体系化する為にカプセル化を行い、構成要素の内部を隠蔽する。

プリミティブなメソッドの抽出　プリミティブとは以下のものである。

- 状態の問い合わせを行うメソッド
- 状態変化を発生するメソッド

プリミティブを抽出することによって以下の効果が期待できる。

- 依存関係とメソッドの影響の局所化を行う。
- 機能の複製の作成を防ぐ。

ポリモーフィズムの導入　ポリモーフィズムの導入は条件判定の変更を容易にする。

3.2.1　前　処　理

前処理では以下の操作を適用可能な箇所全てに行う。

変数の整理　変数の扱い方を改善することでメソッドの抽出を行いやすくする。

変数の整理には以下の操作が含まれる。

「一時変数の分離」　一時変数の中で複数回代入される必要があるものはループ変数と一時変数を集める変数のみである。これ以外の一時変数が複数回代入される場合、代入ごとに別の一時変数に分ける。

「パラメタへの代入の除去」　パラメタへの代入とは、引数で与えられた参照に別のオブジェクトを参照させるように変更する事である。これが問題になることが多いのでパラメタへの代入を除去する。

条件分岐の単純化　条件ロジックは分かりにくくなり易いので、これを単純化しポリモーフィズムの導入を容易にする。

条件分岐の単純化には以下の操作が含まれる。

「条件記述の統合」　条件判定文は異なるが結果のアクションが同じである一連の条件判定文がある場合、一まとめにする。一まとめにした部分をメソッドの抽出で抽出し、ポリモーフィズムを導入し易くする。

「重複した条件記述の断片の統合」　条件記述のアクション部で実行される全てのコードが同じである場合、その部分を条件記述の外に出す。これにより、状態によって振る舞いが変化する部分

と変化しない部分を明確にする。

本例題では、前処理は既に行われているものと仮定する。

3.2.2 カプセル化

本手法ではコンポーネントの構成を目的としているので、構成要素の内部を隠蔽するために全てのフィールドをカプセル化する。カプセル化は以下の操作を用いて行う。

「フィールドのカプセル化」 表 2 がフィールドーカプセル化の対象である。

表 2 フィールドのカプセル化

Customer クラス	_firstName _lastName
Rental	_title _priceCode _daysRented
Window	_customer

「コレクションのカプセル化」 Customer クラスの_rentals が対象となる。

図 6 がカプセル化後のクラス図である。

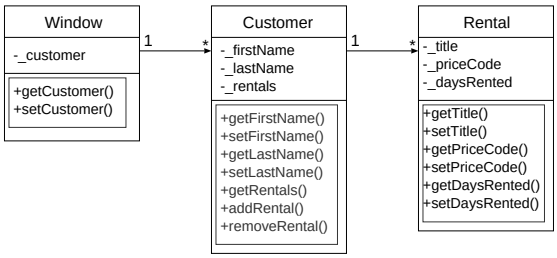

図 6 カプセル化後のクラス図

3.2.3 メソッドの抽出

式の結果を保持する一時変数 代入回数が 1 度の一時変数全てに**「問い合わせによる一時変数の置き換え」**を適用する。前処理でループ変数、一時変数を集める変数以外のすべての一時変数の代入回数が一回となっているので、プリミティブである状態への問い合わせが全て抽出できる。

状態変化を行う部分 プリミティブである状態変化はフィールドの変更で実装される。よって、この部分を**「メソッドの抽出」**によって抽出する。

値を集める変数 **「メソッドの抽出」**によって値を集める変数を返り値として返すメソッドを抽出する。値を集める変数は、複数回のプリミティブを呼び出して 1 つの値を得る為に存在しているので、この変数を利用している部分が 1 つの複雑な作業を行っていると考えられる。この部分を抽出することによって、複数の複雑な作業を行う長いメソッドを作業ごとに分割することが可能となる。

条件分岐 **「メソッドの抽出」**によってタイプコードによる条件分岐の部分を抽出する。この部分を抽出することによって、ポリモーフィズムの導入が可能となる。

ファイルへのアクセス ファイルへのアクセスは 1 つの作業と見なせるのでこの部分を抽出する。この部分を抽出することによって、複数の複雑な作業を行う長いメソッドを作業ごとに分割することが可能となる。

本例題では、式の結果を保持する一時変数として Customer クラス、statement メソッドの fullName が getFullName メソッドとして抽出され、値を集める変数として Customer クラス、statement メソッドの thisAmount と totalAmount がそれぞれ getThisAmount、getTotalAmount として抽出される。図 7 がメソッドの抽出後のクラス図である。

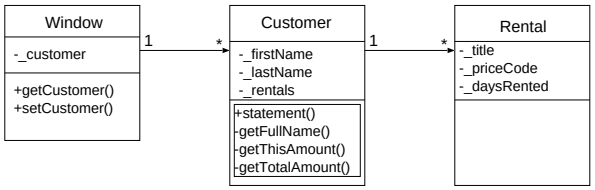

図 7 メソッドの抽出後のクラス図

メソッドの抽出の効果を測定するために関数 $depM$ を定義する。$depM$ のパラメタの意味を表 3 に示す。

定義 3.1

$$depM = \sum_{i=1}^{N} \frac{A_i + B_i + C_i}{N}$$

$depM$ はメソッドのクラスへの依存関係の局所性を表す関数である。$depM$ が小さい場合、各メソッドが扱う情報が局所化され、責任が分担されていると言える。

表 4 は本例題での Customer クラスのメソッドにおける $depM$ の比較である。メソッド抽出後は $depM$ の値が減少しており、メソッドの責任が分担されていることが確認できる。

表 3 $depM$ のパラメタ

パラメタ	意味
A_i	メソッド i が扱うフィールドの数
B_i	メソッド i が呼ぶアクセサメソッドの数
C_i	メソッド i が扱う参照の数

表 4 $depM$ の比較

	$depM$ の値
メソッドの抽出前	2
メソッドの抽出後	1.7

3.2.4 責任の移動

責任の移動を行う事によってクラス間の依存関係をスター型に変更する。責任の移動は以下の手順で行う。

(1) **「クラスの抽出」**によってクラスの責任を分割する。

本稿では、参照とするべきだが値として扱われているフィールドをクラスとして抽出する。この時、対象となるフィールドは本来与えられるべきものとは異なる多重度が与えられている。これらのフィールドと、そのフィールドにのみ依存するメソッドを**「クラスの抽出」**を用いて抽出する。

(2) コントロールクラス作成。
全てのオブジェクトへの参照を持つ新クラスを作成し、これをコントロールクラスと呼ぶ。各オブジェクトへの参照は、オブジェクトの固有名とオブジェクトの固有名からオブジェクトへの参照を得る関数で実現する。クラスがオブジェクトの固有名として利用できるフィールド又はメソッドを持たない場合、作成する。このクラスは複数のクラスに依存するメソッドの統合に用いる。ユーザインターフェースの機能を提供するクラスをコントロールクラスとのみ関連を持つように変更する。

(3) 依存関係のスター型化
依存関係の移動を行い、ユーザインターフェースとコントロールクラス以外のクラスが互いに依存関係を持たないように変更する。これは、各クラスが持つ参照をオブジェクトの固有名に変更することで実現する。次に**「メソッドの移動」**を用いて一つのクラスにのみ依存するメソッドを、依存するクラスに移動し、複数のクラスに依存するメソッドをコントロールクラスに移動する。

(4) 3でコントロールクラスに移動したメソッドに**「引数によるメソッド呼び出しの置き換え」**を適用する。これにより一つのクラスに依存するメソッドが抽出された場合、依存クラスに移動する。**「引数によるメソッド呼び出しの置き換え」**はこの手法のために新しく作成した操作である。
複数のクラスのメソッドを呼ぶメソッド M から、特定のクラス A のメソッドの呼び出しを取り出す。A のメソッドの呼び出しの結果を引数として与えるように残りの部分を**「メソッドの抽出」**で抽出する。
これが適用できる条件は以下の通りである。A のメソッドの呼び出しの返り値が全て基本型、かつ M の実行中 A のメソッドの呼び出しの結果が変化しない

(5) ポリモーフィズムの導入
タイプコードによる条件分岐全てに
「State/Strategy によるタイプコードの置き換え」
「ポリモーフィズムによる条件記述の置き換え」
を適用してポリモーフィズムを導入する。責任の移動によって各クラスはコントロールクラス又はポリモーフィズムで導入されたクラスとのみ依存関係を持つことになるので依存関係がスター型になる。

本例題では以下の事を行う。

- クラスの抽出
  `Rental`クラスのインスタンスで、映画を表す`_title`及び`_priceCode`の組が同じで`_daysRented`のみ異なるものが数多く存在する。`_title`、`_priceCode`、及びそのアクセサメソッドを新クラス`Movie`として抽出する。また、`Rental`クラスに`Movie`クラスへの参照を追加する。
- コントロールクラス作成
  `Customer`クラス、`Rental`クラス及び`Movie`クラスへの参照を持つコントロールクラス`Control`を作成する。`Customer`クラスの個有名として`getFullName()`で得られるフルネーム、`Movie`クラスの固有名として`_title`で表現されるタイトルを用いる。`Rental`クラスにはオブジェクトの固有名として利用できるフィールドが存在しないので、レンタル`ID`を表現する`String`型のフィールド`_rentalID`を追加し、これを固有名とする。
- 依存関係のスター型化
  - `Customer`クラスが持つ`Rental`クラスへの参照の集合`_rentals`を`Rental`の固有名の集合`_rentalIDs`に変更
  - `Rental`クラスから`Movie`クラスへの参照`_movie`を`Movie`の固有名`_title`に変更
- 引数によるメソッド呼び出しの置き換え
  `Control`クラスの`getThisAmount()`メソッドに**「引数によるメソッド呼び出しの置き換え」**を適用する。`Rental`クラスに依存する部分は`getDaysRented()`メソッドの呼び出しである。このメソッドの返り値は`int`型であり、メソッドの実行中に値が変化しない。従って、`getDaysRented()`の呼び出しを引数とするように**「引数によるメソッド呼び出しの置き換え」**を行う。この時抽出されるメソッドを`getCharge()`と呼ぶことにする。`getCharge()`は`Movie`クラスにのみ依存するので`Movie`クラスに移動する。
- ポリモーフィズムの導入
  `Movie`クラスの`_priceCode`はタイプコードである。この`_priceCode`に対してポリモーフィズムの導入を行う。`_priceCode`を表すクラス`priceCode`、各タイプコードの値に相当するクラス`RegularPrice`、`New_ReleasePrice`、及び`ChildrensPrice`クラスを作成し、タイプコードによって変化するメソッド`getCharge()`を移動する。

責任の移動の結果、図8を得る。

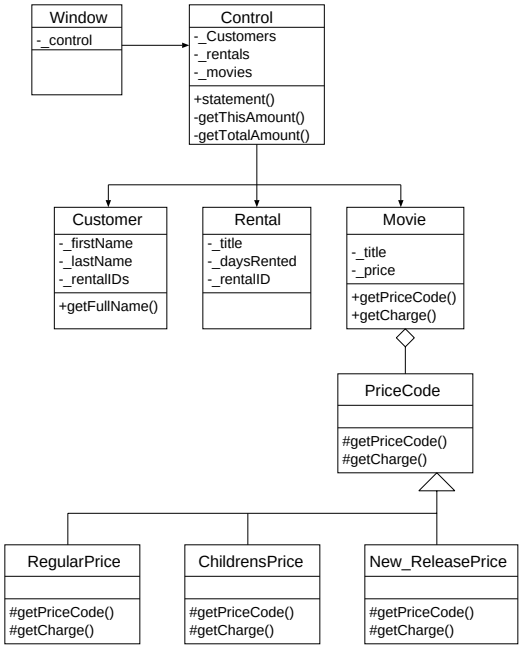

図 8　リファクタリング後のクラス図

3.3　コンポーネントの侯補識別

リファクタリングで結果のクラス図 8 の構成要素を役割毎に分類し、各々をコンポーネントの侯補とする。

注目する役割はとしては以下の四つがある。この分類は Jacobson の BCE(Boundary-Control-Entity)[4] を参考にしたものであり、UI は Boundary、コントロールとアクセッサは Control、データストアは Entity に相当する。

UI　UI とはユーザインターフェースを提供する部分である。ウィンドウ、ボタンといった機能を提供するクラスを UI として識別する。

コントロール　コントロールは、データストアのデータの加工を行う部分である。リファクタリングのメソッドの移動時に抽出されるコントロールクラスをコントロールとして識別する。

アクセッサ　コントロールからデータストアにアクセスする部分をメソッドとして抽出し、それらをアクセッサクラスとして統合する。この作業は**「メソッドの抽出」**、**「クラスの抽出」**を用いて行う。アクセッサはコントロールからデータストアの変更の影響を受ける部分を抽出したものである。アクセッサがデータストアの交換を容易にする。

データストア　データストアとは、情報を扱うクラスである。ポリモーフィズムの場合を除いて、全てのメソッドが他のクラスに依存しないクラスをデータストアとして識別する。メソッドの移動後の既存のクラスがデータストアとして識別される。

本例題では、UI として Window クラス、データストアとして Cusotmer、Rental、Movie、priceCode、RegularPrice、New_ReleasePrice、及び ChildrensPrice クラス、コントロールとして Control クラス、アクセッサとして Accessor クラスを識別し、図 9 を得る。

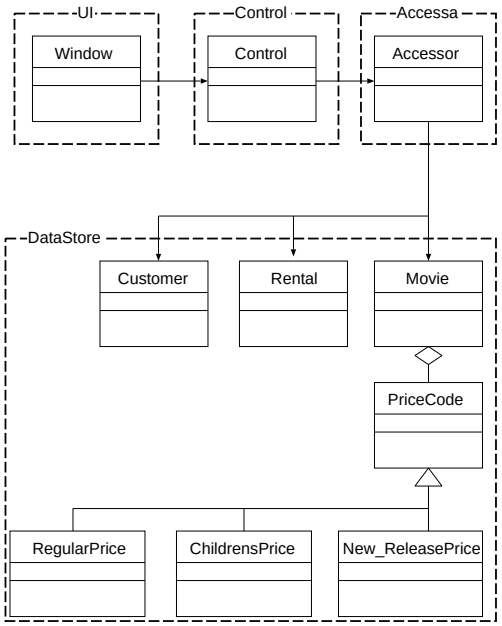

図 9　コンポーネントの候補識別後のクラス図

3.4　コンポーネント化

コンポーネント化の目的は以下の通りである。

- 構成部品の境界の共通化
- 部品のプラグイン可能化

コンポーネント化にはインターフェースを定義する必要がある。インターフェースとはクラスが外部に提供する機能の情報である。

インターフェースの定義は以下の手順で行う。

(1)　**インターフェースの決定**
各部品が提供するインターフェースを決定する。本手法では、可視性が公開となっているメソッドを全てを外部に提供する。

(2)　**インターフェースの作成**
1 で決定したメソッドを全て含むインターフェースを作成する。

(3)　**ソースコードの変更**
2 で作成したインターフェースを利用するようにソフトウェアのソースコードを変更する。

本例題では、表 5 に示すインターフェースを定義し、図 10 を得る。

表 5　公開するインターフェース

分類	公開するインターフェース
UI	IWindow
コントロール	IControl
アクセッサ	IAccessor
データストア	ICustomer IRental IMovie

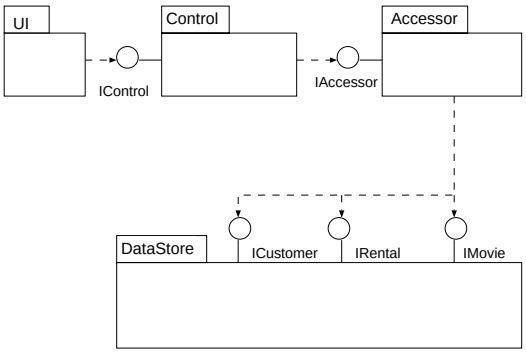

図 10　コンポーネント化後のクラス図

4. 評　　価

本章では前章で用いたビデオレンタルの料金計算システムについて、拡張手法適用前、及び拡張手法適用後に実際に拡張を施し、そのコストをメトリクスを用いて計算する。これによって提案する拡張手法の評価を行う。

4.1 メトリクス

この節では提案する拡張手法を評価するためのメトリクスの説明を行う。

拡張による構造の複雑化として、機能の複製の量を相対的に測るメトリクスを提案する。これを RF と呼ぶ事にする。RF は以下の式で定義できる。

定義 4.1　$RF = A_1 + A_2 + B_1 + B_2 - C$

表 6 は RF に影響を与えるパラメタの特徴である。

表 6　パラメタの特徴

パラメタ	特徴
フィールドへのアクセス数の拡張による増分	新しい機能の追加 A_1 機能の複製 B_1
アクセサメソッドの呼び出し数の拡張による増分	新しい機能の追加 A_2 機能の複製 B_2
アクセサメソッド以外のメソッド呼び出しの数の拡張による増分	機能の複製の回避 C

ここで、同じ機能を持つ二つのソフトウェア X、及び Y に同じ拡張を行うとすると、この拡張における X のメトリクスを $RF(X)$、Y のメトリクスを $RF(Y)$ とする。$RF(X)$、$RF(Y)$ は以下の式で計算される。

$$RF(X) = A_1(X) + B_1(X) + A_2(X) + B_2(X) - C(X)$$

$$RF(Y) = A_1(Y) + B_1(Y) + A_2(Y) + B_2(Y) - C(Y)$$

$A = A_1 + A_2$、$B = B_1 + B_2$ とすると、X と Y は同じ機能を持つので $A(X) + A(X) = A(Y) + A(Y)$ が成り立つ。従って、$RF(X)$ と $RF(Y)$ の大小関係は $B(X) - C(X)$ と $B(Y) - C(Y)$ の大小関係と一致する。これは直観的には複製の量-複製の回避の量を表す値である。RF 値が低いほど拡張コストが低いことを意味する。

4.2 メトリクスの適用

現在のシステムでは映画のレンタルのみを扱っている。これに CD レンタルの料金計算の機能を追加する。CD のレンタル料金は以下の通りである。

レンタル期間が 2 日未満　100 円

レンタル期間が 2 日以上　100 円+50 円 × レンタル期間

4.2.1 提案手法適用前

このシステムへの CD レンタルの料金計算機能の追加は以下の手順で行う。

(1) CD のレンタル情報を表す RentalCD クラスを作成

(2) Customer クラスに RentalCD への参照を保存する集合_rentalCDs とビデオレンタルの料金を計算する関数の後に CD レンタルの料金を計算する関数とレシートを作成する機能を追加

フィールドへのアクセス数の増分　5

アクセサメソッドの呼び出し数の増分　0

メソッド呼び出しの数の増分　0

以上より RF 値は 5 となる。

4.2.2 提案手法適用後

このシステムへ CD レンタルへの料金計算機能の追加は、IMovie インターフェースを実装する CD クラスの作成によって行う。

フィールドへのアクセス数の増分　アクセサメソッドが二つ増えるので 2 である。

アクセサメソッドの呼び出し数の増分　コンストラクタの呼び出しが二つ増えるので 2 である。

メソッド呼び出しの数の増分　Control クラスの getThisAmount() メソッドでの getCharge() の呼び出しが増加するので 1 である。

4.3 考　　察

拡張手法適用前と拡張手法適用後で同じ拡張を行い、機能の複製の量を測るメトリクスを定義し、評価を行った。

この比較において最も注目すべき点は、本手法適用後のシステムへの拡張では Control クラスの statement() メソッドが呼ぶ getCharge() が追加されている事である。これは、メソッドの抽出によってメソッドの責任が分担され、責任の移動によって情報を保持する部分と情報を利用する部分に分割されているために可能になって

いる。

拡張前はこの分類が明確に行われていないので、Customer クラスの statement() メソッドにレシートのフォーマットの整合性を保持したまま情報を利用する部分であるレシートの作成機能と情報を保持する部分である料金計算関数の追加するという複雑な構造な拡張を行う必要があるが、拡張手法を適用した場合は情報を保持する部分である CD クラスのみの追加で十分である。これはメソッドからのプリミティブの抽出、依存関係のスター型化、及びインターフェースの導入が有効に働いているためである。

メソッドからプリミティブを抽出することで、プリミティブは利用される側、残りの部分は利用する側という分類を行っている。依存関係のスター型化により、利用される側は他に依存しなくなるので変更のコストが低減される。インターフェースを定義することにより、利用される側を変更しても利用する側に影響を与えなくしている。

提案手法を利用してソフトウェアの構成要素を分類し、特に利用される部分の変更の影響を小さくすることによって、この部分の拡張のコストを低減していることが定性的、定量的に確認された。

5. おわりに

本稿では、ソフトウェアの拡張による構造の複雑化とそれに伴う開発・保守コストの増大の問題について、リファクタリングを用いた拡張コスト低減のための手法を提案した。この手法はリファクタリングの適用プロセス、コンポーネント化のプロセスを提供する。これらによって、ソフトウェアの構造を単純化し、コンポーネント単位の容易な拡張を導入することでソフトウェアの拡張コストを低減することを可能にした。また、例題によってその有効性を示した。

下に挙げられる幾つかの点が課題として残る。

- 結果の最適化
  メソッドの抽出は間接層の挿入である。メソッドの間接層の数に関する新たなメトリクスを発見し、間接層の数を抑える。これにより、メソッドの数を適度に減らすことが期待される。
- 新たなコンポーネントの候補の発見
  メソッドの依存関係をグラフ理論を応用して調べることにより、新たなコンポーネントの候補の発見を目指す。

参 考 文 献

1) Martin Fowler,Kent Beck, John Brant,William Opdyke,don Roberts, Refactoring:Improving the Design of Existing Code, ADDISON-WESLEY,1999

2) Clemens Szyperski,
Component Software - Beyond Object-Oriented Programing, ADDISON-WESLEY, 1997

3) コンポーネントウェア 青山幹雄・中所武司・向山博 編, 共立出版, 1998

4) Ivar Jacobson,
Object-Oriented Software Engineering: A Use Case Driven Approach,
Association for Computing Machinery,1992