Vol.2014-SE-185 No.19
2014/7/11

設計抽象化のためのリファクタリングパターン

艾 迪[1,a)]　鵜林 尚靖[1,b)]　李 沛源[1,c)]　李 宇寧[1,d)]　細合 晋太郎[1,e)]　亀井 靖高[1,f)]

概要：抽象化はソフトウェア設計において重要な役割を果たす．適切な抽象度で設計モデルとコードを明確に分離することが出来れば良いが，多くのソフトウェア開発者にとっては難しい問題である．現実的には設計とコードの間を行ったり来たりしながら，適切な関心事の分離を模索せざるを得ない．我々は，このような反復プロセスを支援するための方法として，設計抽象化のためのリファクタリングパターンを提案する．このパターンは *MoveM2C*(モデルからコードへの関心事の移動) と *MoveC2M*(コードからモデルへの関心事の移動) の 2 種類から構成される．本論文では，提案パターンの有効性を確認するための評価実験についても述べる．

キーワード：抽象化, 設計モデリング, リファクタリング, パターン

Refactoring Patterns for Design Abstraction

Di Ai[1,a)]　Naoyasu Ubayashi[1,b)]　Peiyuan Li[1,c)]　Yuning Li[1,d)]　Shintaro Hosoai[1,e)]
Yasutaka Kamei[1,f)]

Abstract: Abstraction plays an important role in software design. It is preferable to explicitly separate design models from the corresponding code at an appropriate abstraction level. Unfortunately, it is not easy for ordinary software developers to find this adequate abstraction level. In order to seek a well-balanced separation of concerns between design and code, it is not avoidable to go back and forth between them. To support this iterative abstraction refinement process, we propose refactoring patterns consisting of *MoveM2C* (Move concerns from Model to Code) and *MoveC2M* (Move concerns from Code to Model). We evaluate the effectiveness of the proposed refactoring patterns by applying them to a real application.

Keywords: Abstraction, Design Modeling, Refactoring, Patterns

1. はじめに

設計抽象化はソフトウェア工学において解決すべき重要課題の一つである [4]．様々なモデルがソフトウェア設計の抽象化表現に利用される．しかし，開発者の多くは自分が作成したモデルに対して必ずしも自信が持てないのが実情だと思われる．モデル同士の整合性やソースコードとの対応についてはある程度開発支援ツールによるチェックが可能であるが，「モデルが適切に抽象化されているか」については通常ツール支援の対象外である．一般的に，適切に抽象化された設計を行うのは容易ではなく，設計とソースコードの間を行ったり来たりしながら，何度も設計モデルとソースコードを見直していく必要がある．分かりやすく抽象化されたモデル，適切にモジュール化されたソースコードの多くは，このような過程を経て獲得されたものである．しかし，これは根気のいる繰返し作業であり，現実のモデルやソースコードは抽象化という観点では不十分な状況で終わっている場合が多い．

我々は，このような問題を解決するための方法として，設計抽象化のためのリファクタリングパターンを提案する．このパターンは大きく分けて，二つのカテゴリから

[1] 九州大学大学院システム情報科学府
[a)] aidi@posl.ait.kyushu-u.ac.jp
[b)] ubayashi@ait.kyushu-u.ac.jp
[c)] lipeiyuan@posl.ait.kyushu-u.ac.jp
[d)] liyuning@posl.ait.kyushu-u.ac.jp
[e)] hosoai@qito.kyushu-u.ac.jp
[f)] kamei@ait.kyushu-u.ac.jp

© 2014 Information Processing Society of Japan

なる．一つは，「設計モデルからソースコードへの関心事の移動 (*MoveM2C*: Move concerns from Model to Code)」であり，たとえば，実装の詳細に関わる関心事が設計モデルに含まれている場合に，モデルからその関心事を削除し，ソースコードのみにその関心事を含める場合に使用する．この場合，設計モデルの抽象度は上がる．もう一つのカテゴリは，「ソースコードから設計モデルへの関心事の移動 (*MoveC2M*: Move concerns from Code to Model)」である．これは，ソースコードに含まれる関心事で設計モデルにも反映すべきものを抽出するためのものである．設計モデルを読んでも実装のイメージが掴めないときに有効なパターンである．この場合，設計モデルが詳細化されるので抽象度は下がる．本論文では，これら二つのカテゴリを細分化し，最終的に 16 個のパターンとしてまとめた．Fowler, M. などが提案する一般的なリファクタリング [3] を事前に実施し，その後に我々のパターンを適用するとより効果的である．たとえば，設計モデル中のあるクラスの一部 (設計上あまり重要ではない部分) をサブクラス化し，設計モデルから削除する (「設計モデルからソースコードへの関心事の移動」の適用) などの組み合わせ適用が考えられる．

本論文では，リファクタリングパターンと共に，その効果を測定するためのメトリクスとして「抽象度」と「抽象度の変化」の二つを導入する．前者は設計モデルがソースコードと比較してどの程度抽象的であるかを示し，後者は抽象度がリファクタリングを行うことによってどの程度変化したのかを示す．一般的に抽象度の推奨値はプロジェクトやドメインにより異なる．しかし，抽象度の変化が小さくなった場合 (抽象度の値がある値に収束した場合) はリファクタリングの反復実行の終了目安として有効である．

本稿では，まず 2 節で設計抽象化に関わる問題点を指摘する．3 節で設計抽象化のためのリファクタリングパターンを，4 節で抽象度に関するメトリクスを提案する．5 節で評価実験の結果を示す．ここでは，三つの観点 1) 実際のプロジェクトにおいて抽象度はどの程度の値になるのか，2) 何が設計抽象度見直しのトリガーとなるのか，3) リファクタリングパターンの適用によりどう抽象度が変化していくのか，から評価を行った．最後に 6 節でまとめと今後の課題について述べる．

2. 設計抽象化に関わる課題

設計モデルには様々な側面があるが，その中でもソフトウェアの基本構造を決定づけるアーキテクチャ定義は最も重要である．設計モデルは，ソフトウェアアーキテクチャの抽象的な記述であり，ソースコードはその設計を実装したものと言える．設計とコードは関心事の分離の観点から明確に切り分けるべきであるが，これを達成するのは現実的には容易ではない．実際，抽象度はソフトウェア開発の進行と共に変化しがちである．開発者は万能ではないので，設計に関わる関心事を開発の初期段階ですべて把握できるとは限らない．どうしても，設計と実装の間で行ったり来たりせざるを得ない状況が発生する．ソフトウェア設計とアーキテクチャに関する重要な研究領域の一つとして，Taylor, R. N. らは，設計と実装の間を流動的に移動できるための支援の必要性を主張している [6] ．流動的に移動することにより設計モデルとソースコードの間のバランスを再検討する必要性が生じ，その結果として設計モデルの抽象度も流動的に変化する．すなわち，どの関心事を設計が分担に，どの関心事をプログラム記述に任せるべきかを常に判断する必要がある．しかしながら，この判断は容易ではなく，繰返し設計モデルとソースコードをリファクタリングして行く必要がある．

表 1 リファクタリングパターンの記述形式

項目	説明
名前	パターン名
問題	パターンが対象とする典型的な問題 (問題が発生する状況や条件)
適用手順	パターンを適用するための手順 (パターン適用前と適用後の設計モデル)
抽象度	パターンを適用することによる抽象度の変化

設計抽象化に関わる課題を整理すると以下のようになる．

- 課題 1: 適切な抽象度を獲得するには，設計モデルとソースコードにまたがったリファクタリングを反復的に実施する必要があるが，どのようなリファクタリングを行えばよいのかをガイドする指針がない．
- 課題 2: リファクタリングによる抽象化の改善度合い，リファクタリングの終了を判断するためのメトリクスがない．

本論文では，これらの課題を解決するためのリファクタリングパターンとメトリクスを提案する．

3. リファクタリングパターン

本節では，設計抽象化を支援するリファクタリングパターンについて述べる．このパターンは 2 節で取り上げた課題 1 を解決するためのものである．

3.1 記述形式

リファクタリングパターンの記述形式を表 1 に示す．パターンの記述は，設計モデルおよびソースコードを抽象化の観点から見直す際に繰返し利用出来るように，名前，問題，適用手順，パターンの適用による抽象度の変化，から構成される．適用手順の中では，パターン適用前後の設計モデルの例が記述される．例として設計モデルのみを使用したのは，抽象度の見直しは主に設計モデル上で発生する

© 2014 Information Processing Society of Japan

表 2 設計抽象化のためのリファクタリングパターン

カテゴリ	パターン		略称	内容	抽象度
MoveM2C	クラス図	①Remove Class	RmCs	重要でないクラスを削除	↗
		②Remove Sub Class	RmSubCs	モデル上クラスの重要でないサブクラスを削除	↗
		③Remove Library Class	RmLibCs	重要でないライブラリ (API) のクラスを削除	↗
		④Remove Method	RmMs	設計上で反映したくないメソッドを削除	↗
		⑤Remove Field	RmFd	設計上で反映したくないクラスの属性を削除	↗
	シーケンス図	⑥Remove Message	RmMsg	重要でないメッセージを削除	↗
		⑦Remove API Call	RmAPI	重要でない API 呼び出しのメッセージを削除	↗
		⑧Remove Object	RmObj	設計上で反映したくないオブジェクトを削除	↗
MoveC2M	クラス図	⑨Add Class	AddCs	コードからある重要なクラスをモデルに反映する	↘
		⑩Add Sub Class	AddSubCs	コードからあるクラスのサブクラスをモデルに反映	↘
		⑪Add Library Class	AddLibCs	ライブラリ (API) クラスをモデルに反映	↘
		⑫Add Method	AddMs	コードからあるメソッドをモデルに反映	↘
		⑬Add Field	AddFd	コードからクラスの属性をモデル反映	↘
	シーケンス図	⑭Add Message	AddMsg	重要なメッセージ送受信をモデルに反映	↘
		⑮Add API Call	AddAPI	重要な API 呼び出しをモデルに反映	↘
		⑯Add Object	AddObj	コードから重要なオブジェクトを反映	↘
Fowler の カタログ		Extract Superclass Extract Subclass	ExtSprCs ExtSubCs	共通的な要素を親クラスに移行 共通的でない要素を子クラスに移行	

からである．*MoveM2C* の場合は設計モデルからの関心事の削除 (関連するモデル要素の削除) であり，*MoveC2M* の場合は設計モデルへの関心事の追加 (コードに対応するモデル要素の追加) により実行される．

3.2 パターンのカテゴリ

設計抽象化のためのリファクタリングパターンは，大きく *MoveM2C*(モデルからコードへの関心事の移動) と *MoveC2M*(コードからモデルへの関心事の移動) の 2 つのカテゴリから構成される．各カテゴリは，更に表 2 のように細分化されたパターンから構成される．現時点では 16 個のパターンからなる．表 2 では，*MoveM2C* と *MoveC2M* 以外に Fowler, M. らのリファクタリングカタログ [3] の一部を掲載している．ソースコードを対象としたリファクタリングの中には設計モデルのリファクタリングにも応用出来るものが数多く存在するからである．また，一部は設計モデルの抽象度の改善にも適用可能である．紙面の都合上, 表 2 には *ExtSpCs*(Extract Superclass) と *ExtSupCs*(Extract Subclass) の二つのみを掲載している．

現時点では，リファクタリングの対象となる設計モデルは UML のクラス図とシーケンス図の二つである．設計における構造的な側面はクラス図で，振舞い的な側面はシーケンス図で表現できるので，最低限この二種類のダイアグラムをサポートすれば十分だと考えたからである．他のダイアグラムについては今後順次サポートしていく．

3.2.1 MoveM2C カテゴリ

MoveM2C は，設計モデル上の関心事の一部をモデルからコードに移動させることである．*MoveM2C* 適用前は設計モデル上にもソースコード上にも同じ関心事が存在するが，抽象度の観点から設計モデルに含めなくてもよいものが移動の対象となる．移動とは言え，実際には設計モデルからの削除である．*MoveM2C* は設計の抽象度を高める効果をもつ．

通常，ソースコードを修正するたびに設計モデルの方もそれに同期して修正しなければならない．しかし，それが頻繁に発生する場合は，抽象度に関する「不吉な匂い」(Bad Smells in Design Abstraction) と考えられる．多くの場合，このような関心事は設計ではなく実装に強く依存するため，ソースコード上のみに関心事を置いた方がよい．このような場合に *MoveM2C* カテゴリのリファクタリングパターンが適用できる．結果として，ソースコードに変更が発生しても設計モデルにはその影響が波及しなくなる．その他，パフォーマンスの問題や正しい API の使い方など，適切な設計を見つける為に実験的なコードを書く必要がある場合に *MoveM2C* が一時的に適用出来る．適切な設計を見つけ出した後にもう一つのリファクタリングカテゴリである *MoveC2M* を適用し，ソースコードから設計モデルに関心事を戻せばよい．

MoveM2C カテゴリに属するリファクタリングパターンは二つの種類に分けられる．一つはクラス図を抽象化するための構造的リファクタリング (表 2 の①から⑤までのパターン) で，もう一つはメーセージのシーケンスを抽象化するための振舞い的リファクタリング (⑥から⑧までのパターン) である．②と③は①の特別なケースであるが，利便性を考慮して独立したパターンとした．API 関係のクラスなどは，設計が複雑で詳細すぎる場合にモデルから削除すべきものの有力な候補になるからである．同様に，⑦は⑥の特別なケースである．

© 2014 Information Processing Society of Japan

ここでは，振舞い的リファクタリングを例に，その手順を簡単に説明する．まず，設計モデルから不要と判断したメーセージシーケンスを削除する (*RmMsg* パターン適用)．削除したメッセージに関連するオブジェクトも削除する (*RmObj* パターン適用)．メッセージとオブジェクトの削除が他のメッセージやオブジェクトにも影響が波及する場合は，これらのパターンを続けて適用し，シーケンス図内で矛盾が生じないようにする．シーケンス図内での矛盾が解消した後は，クラス図との整合性をチェックする．削除したオブジェクトが属するクラスが他でも出現しないのであれば，このクラス定義を削除する必要がある (*RmCs* パターン適用)．メッセージに対応するメソッド定義が不要な場合はこれも削除する必要がある (*RmMs* パターン適用)．このように設計モデルの抽象化は関連するモデル要素の削除に留まらず，モデル間のチェック [2] も必要となる．更にはソースコードとのトレーサビリティもチェックする必要がある．これらの作業を手動で実施するのには多大な労力を必要とする．また，チェック漏れも生じやすい．このような問題を解決するため，我々は，「設計モデル間の整合性チェック」「設計モデルとソースコード間のトレーサビリティチェック」が自動的に行えるツール *iArch*[1] を開発した．*iArch* では，設計モデル間およびソースコードの間に *Archface*[7] と呼ばれるインタフェース機構を導入することにより，これらのチェックを自動化している．このように，設計抽象化のためのリファクタリングを実践するには，*iArch* のようなツール支援が必須となる．従来のソースコードを対象としたリファクタリングでは機能を保持しつつコードの構造のみを改善する．その際，機能が本当に保持されているのかを確認するために単体テストの自動化が重要となる．設計抽象化の改善を目的としたリファクタリングでも当然機能を保持しなければならないが，それに加えて，ここで述べたような設計モデル間の整合性やソースコードとのトレーサビリティが維持されなくてはならない．

3.2.2 MoveC2M カテゴリ

MoveC2M は *MoveM2C* とは逆方向のリファクタリングで，設計の抽象度を下げる効果をもつ．*MoveC2M* は，ソースコード上にしか存在しない関心事を設計モデルに反映することである．すなわち，設計モデルにソースコードに対応する設計要素を追加する．たとえば，開発の初期段階では明確な形でモデリング出来なかったが，その後コーディングで設計方法が確定した場合などに適用できる．

MoveC2M カテゴリに属するリファクタリングパターンも *MoveM2C* と同様に，二つの種類に分けられる．一つはクラス図を詳細化するための構造的リファクタリング (表 2 の⑨から⑬までのパターン) で，もう一つはメーセージのシーケンスを詳細化するための振舞い的リファクタリング（⑭から⑯までのパターン）である．各パターンの内容

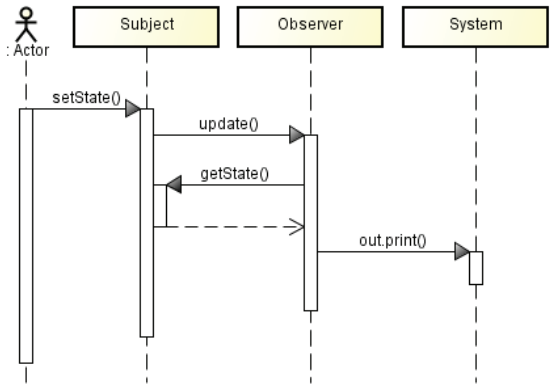

図 1 *RmAPI* パターン適用前のシーケンス図

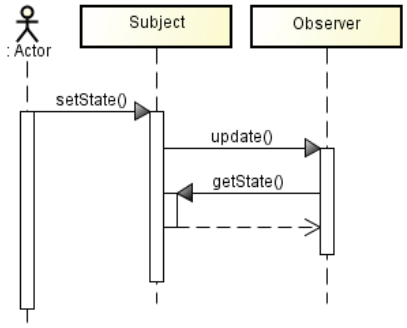

図 2 *RmAPI* パターン適用後のシーケンス図

を見て分かるように *MoveM2C* カテゴリに属するパターンと双対になっている．

3.3 パターン記述

ここでは，リファクタリングパターンの記述内容について説明する．例として，*RmAPI*(Remove API Call) パターンの紹介する．その他のパターンも同様の形で記述されている．

名前: RmAPI (Remove API Call)

問題: API の使用方法などを考慮し過ぎて，設計モデルが過度に詳細になった場合に適用する．

いま，図 1 に示すような *Observer* パターンのクラス図があるとする．`Subject` から `Observer` に通知された状態値 (State) の表示に Java の System の API を使用している．ただ，設計の抽象度を考慮すると，わざわざモデルに含める必要性は高くない．また，設計モデルが Java という特定のプログラミング言語に依存しているという問題もある．したがって，System.out.print の呼び出しを削除したい．

適用手順: 適用前後のシーケンス図を図 1 と図 2 に示す．以下の手順で print 呼び出しを削除する．

- Step 1: API 呼び出し out.print を削除する．
- Step 2: out.print の送信先オブジェクト (クラス) である System を削除する．

© 2014 Information Processing Society of Japan

表 3　設計点とプログラム点

設計点 (UML2 メタモデル)	プログラム点
クラス図	
1. Class	class
2. Operation	method
3. Property	field
シーケンス図	
4. Message -sendEvent:MessageEnd	method call
5. Message -receiveEvent:MessageEnd	method exec

- Step 3: メッセージ out.print および System の削除により当該シーケンス図に不整合が生じないかチェックする．
- Step 4: 対応するクラス図や他のシーケンス図に不整合が生じないかをチェックする．

抽象度: 重要でない API 呼び出しが設計モデルから削除されるので，抽象度が向上する．

4. 抽象化のためのメトリクス

本節では，リファクタリングパターンの適用効果を測定するためのメトリクスについて述べる．メトリクスとして「抽象度」と「抽象度の変化」の二つを導入する．前者は設計モデルがソースコードと比較してどの程度抽象的であるかを示し，後者は抽象度がリファクタリングを行うことによってどの程度変化したのかを示す．これらのメトリクスは 2 節で取り上げた課題 2 を解決するためのものである．

4.1 抽象度

設計モデルの抽象度は以下の式で計算する．

$$\text{抽象度} = 1 - \frac{\text{設計点の数}}{\text{プログラム点の数}}$$

設計点とは，モデルに含まれるクラス，属性，メソッド，メッセージ送信，メッセージ受信などの個所であり，UML2 のメタモデル上の構成要素と対応づけることができる．一個のクラスや属性は設計点一つとしてカウントされる．一方，プログラム点とは，ソースコード中の，クラス定義，属性宣言，メソッド宣言，メソッドの呼び出し点，メソッドの実行点などを指す．表 3 に今回のメトリクス定義で採用した設計点とプログラム点を示す．

もし設計モデルとソースプログラムの記述内容が同一であれば，設計点の個数とプログラム点の個数は同じになる．上記の計算式に当てはめると，この場合の抽象度はゼロとなる．一方，設計モデルが存在せずにソースコードしかない場合は設計点の個数がゼロとなり，抽象度は 1 と計算される．このように抽象度は 0 から 1 の値をとり，設計モデルがソースコードと比較してどの程度抽象的かを示す．3.3 節の例では，System.out.print メッセージ送信に対応する設計点が削除されるので，リファクタリングにより抽象度が上がることが分かる．

リファクタリングのたびに手動で抽象度を計算し直すのは大変である．そのため，我々が開発したツール *iArch* では自動的に抽象度を計算する機能を提供している．

4.2 抽象度の変化

このメトリクスは，リファクタリング実行の前と後における抽象度の差分であり，以下の式で計算する．

$$\text{抽象度の変化} = \text{リファクタリング後の抽象度} - \text{リファクタリング前の抽象度}$$

抽象度の推奨値は，開発プロジェクトやソフトウェアアプリケーションの対象ドメインにより異なる．将来，抽象度に関するデータが蓄積されれば，プロジェクトや対象ドメインの特性から推奨値を算出することが可能になるかもしれないが，現時点では難しい．しかしながら，この「抽象度の変化」メトリクスはリファクタリングの反復実行の終了目安として活用できる．抽象度の変化が小さくなった場合は，抽象度がある値に収束したこと，すなわち設計モデルの抽象化洗練が完了したことを示唆するからである．

5. 評価実験

リファクタリングパターンの有効性を確認するため，評価実験を行った．

5.1 対象システム

評価実験の対象としたのは「Web 書店システム」(名称は浅瀬書店システム) である．これは九州大学の PBL(Project-Based Learning) 授業で修士課程 1 年の学生が開発したシステムであり，その開発成果物は UML による設計モデルとソースコード (Java および JavaScript で記述) から構成される．今回は，このうちのクラス図，シーケンス図，Java プログラムのみを評価実験の対象とした．すなわち，これらの開発成果物に対して設計抽象化のためのリファクタリングパターンを適用し，その結果として抽象度がどのように変化，収束して行ったのかについて評価した．

この書店システムは，ユーザ向けの書籍検索・ランキング表示機能と店員向けの書籍情報管理機能から構成される．表 4 にシステムのモジュール構成とその規模を示す．Java で書かれた部分は，1) 各モジュールの関係のみを定義したモジュール (abs)，2) サーバ処理やデータベースへのアクセスを制御するモジュール (asasecommon)，3) ユーザ向けのモジュール (web)，4) 店員向けの管理モジュール (office)，の四つから成る．なお，ユーザーインタフェース・モジュール (UI) は JavaScript で書かれているため，今回の評価実験の対象から除外した．表 4 中，LOC(Lines of Code) は各メソッド内のコード行数の合計値を表す．

評価の対象として，この書店システムを選択したのは，

© 2014 Information Processing Society of Japan

表 4　Web 書店システムのプロジェクト規模

モジュール名	クラス数	クラス図数	シーケンス図数	LOC
abs	23	1	0	411
asasecommon	26	1	0	1393
web	9	1	2	244
office	20	1	4	438
UI	—	—	—	3170

表 5　PBL 終了時における Web 書店システムの抽象度

ユースケース ID	シーケンス図	設計点	プログラム点	プログラム点 (全)	ミスマッチ	抽象度
u1	ユーザ-ランキング閲覧	9	22	168	2	0.59
u2	ユーザ-書籍検索	16	24	236	0	0.33
u3	店員-出版社検索，更新，削除	31	76	578	0	0.59
u4	店員-出版社登録	12	24	186	0	0.5
u5	店員-書籍検索，更新，削除	37	82	618	0	0.55
u6	店員-書籍登録	12	24	186	0	0.5

第一著者が PBL に参画し実際に開発したものだからである．システムの内容について熟知しているため，リファクタリングのみに注力できるというメリットがある．

5.2　評価の観点と方法

今回の実験では，以下の三点を評価の観点とした．

- 評価観点 1: 実際のプロジェクトにおいて，抽象度はどの程度の値になるのであろうか?
- 評価観点 2: 何が設計抽象度見直しのトリガーとなるのであろうか?
- 評価観点 3: リファクタリングパターンを適用することにより，どのように抽象度が変化していくのか? 抽象度は最終的に収束するのであろうか?

評価観点 1 は，抽象度の値に一定の傾向が見られるのか否かを明らかにするためのものである．抽象度はプロジェクトや対象ドメインにより異なると考えられる．しかし，同種のプロジェクトあるいは似たようなサブシステムでは同じような傾向になる可能性がある．今回の実験ではあくまでも一つのプロジェクトでの評価なので断定的な結論は出せないが，予備実験的な意味で傾向をみたい．

評価観点 2 は，リファクタリングのトリガーとなる「不吉な匂い」を明らかにするためのものである．

評価観点 3 は，我々が提案するリファクタリングパターンの適用効果を測定するためのものである．PBL 成果物に対してパターンを適用することにより，評価実験を行う．抽象度の他に「抽象度の変化」のメトリクスを用いる．

5.3　結果と考察

5.3.1　評価観点 1 について

表 5 に評価結果を示す．PBL ではユースケース単位にシーケンス図を作成したので，ここではそれごとに抽象度を算出した．「プログラム点 (全)」は，共通パッケージに含まれるプログラム点まで全てを含めた数値 (共通ライブラリ呼び出し先の更なる呼び出しシーケンスまでをプログラム点としてカウントした値) である．この数値をそのまま抽象度の計算に使用すると抽象度が異常に高く算出されてしまう．そのため，抽象度の計算では，共通パッケージに含まれるプログラム点は除外した (4 カラム目のプログラム点数を使用)．

表 5 を見ると，今回のような教育目的でのシステム開発では抽象度は大体 0.5 程度になる．一般的なシステム開発よりは若干抽象度が低めに感じるが，設計モデルとソースコードのトレーサビリティを理解させることが教育目標の一つなので，0.5 という抽象度はその理解を助けるのには妥当な数値だと考えられる．今回の結果はあくまでも一つの事例に過ぎなく，この結果から一般化を行うことは出来ないが，少なくともプロジェクトや開発ドメインの特性により抽象度の値に傾向が見られる可能性があることが分かった．

5.3.2　評価観点 2 について

設計抽象化のためのリファクタリングのトリガーとなる「不吉な匂い」として，1) 抽象度の異常なズレ，2) 設計モデルとソースコードのミスマッチ (トレーサビリティがとれなくなっている状態) が浮かび上がった．

不吉な匂い「抽象度の異常なズレ」:　表 5 を見ると，シーケンス図 u2 の抽象度が 0.33 と低い値となっている．この数値からだけでは抽象度が適切か否かは判定できないが，他のシーケンス図と比較して，大きく値がずれているのは事実である．表 5 のユースケースはどれも似たような機能なので一つだけ抽象度が異なるのは，異常値である可能性が高い．そこで，シーケンス図 u2 を中身を確認してみたところ，「ユーザ-書籍検索」機能は「簡単検索」と「詳細検索」二つの機能を分けて書かれていた，しかし実際のプログラムの中には共通化して一つの機能として実装して

© 2014 Information Processing Society of Japan

表 6 不吉な匂いと適用すべきリファクタリングパターン

ユースケース ID	シーケンス図	不吉な匂い	問題	適用すべきパターン
u1	ユーザ-ランキング閲覧	B	API の使い方の変更によるシーケンス図とソースコードの不一致	*RmAPI*, *RmObj*, *AddObj*, *AddMsg*
u2	ユーザ-書籍検索	A	簡単検索と詳細検索機能との区別は必要ない	*RmMsg*
u3	店員-出版社検索，更新，削除	B	一部の機能がシーケンスに反映されていない	*AddObj*, *AddMsg*, *RmMsg*
u4	店員-出版社登録	B	同上	*AddObj*, *AddMsg*, *RmMsg*
u5	店員-書籍検索，更新，削除	B	同上	*AddObj*, *AddMsg*, *RmMsg*
u6	店員-書籍登録	B	同上	*AddObj*, *AddMsg*, *RmMsg*

A: 抽象度の異常なズレ，B: 設計モデルとソースコードのミスマッチ

表 7 リファクタリングの各ステップにおける抽象度変化 (抽象度の差分の変遷)

ユースケース ID	シーケンス図	リファクタリングのステップ		
		1: トレサビリティの修正	2: *MoveC2M* の適用	3: *MoveM2C* の適用
u1	ユーザ-ランキング閲覧	*RmAPI*, *RmObj* / +0.09	*AddObj*, *AddMsg* / -0.05	—
u2	ユーザ-書籍検索	—	—	*RmMsg* / +0.25
u3	店員-出版社検索，更新，削除	—	*AddObj*, *AddMsg* / -0.12	*RmMsg* / +0.15
u4	店員-出版社登録	—	*AddObj*, *AddMsg* / -0.12	*RmMsg* / +0.16
u5	店員-書籍検索，更新，削除	—	*AddObj*, *AddMsg* / -0.11	*RmMsg* / +0.14
u6	店員-書籍登録	—	*AddObj*, *AddMsg* / -0.12	*RmMsg* / +0.16

凡例: 適用したパターン / 抽象度変化

いた．この場合，*RmMsg* パターンを適用して u2 のシーケンス図をリファクタリングすることが望ましい．今回の評価実験はあくまでも一例に過ぎないが，一般的なシステム開発においても，抽象度の異常なズレが「不吉な匂い」になる可能性が高いと考えられる．

不吉な匂い「設計モデルとソースコードのミスマッチ」: 表 5 を見ると，ユーザ-ランキング閲覧 (u1) にミスマッチ (不整合) が 2 つ存在する．これは Amazon API を使っているところにある．設計時は「Java 側でランキング情報を取得するのに Amazon API を使用する」と考えていたが，実際の実装に際しては「Amazon API は JavaScript 側で書籍の表紙情報だけに取得する」ように変更したからである．API 使用方針や API そのものの仕様変更によって，シーケンス図とソースコードの間に不一致が生じるのは現実の開発現場でも起きやすい．ソースコードが変更になった時点で，設計モデルを直ぐに変更すれば済むことであるが，開発納期などの都合で設計モデルの更新がなおざりになるのはよく散見される現象である．多くの場合，ソースコードとの同期だけでなく，設計モデルの抽象度の見直しが必要とされる場合が多い．すなわち，設計モデル間あるいは設計モデルとソースコード間の不整合は「不吉な匂い」である可能性が高い．

表 6 に「不吉な匂い」と適用すべきリファクタリングパターンを示す．

5.3.3 評価観点 3 について

表 6 の問題を解消するため，今回，表 7 に示す三つのステップ (トレーサビリティの修正，*MoveC2M* の適用，*MoveM2C* の適用) に沿ってリファクタリングを行った．表 7 はリファクタリングの各ステップにおいてどのように抽象度が変化したのか，その差分を示したものである．差分が小さくなればリファクタリングの終了判定に使用できると考えたが，必ずしもそのような結果にはならなかった．各差分値が小さいこと，ステップ 2 とステップ 3 はどちらを優先してリファクタリングを行っても構わないことが起因しているのかもしれない．一方，図 3 は各リファクタリングステップの実行により抽象度そのものの値がどのように変遷したのかを示したものである．図を見て分かるように，どのシーケンス図も大体同じような抽象度に収束している様子が分かる．最初の段階 (PBL 終了時における段階) では，抽象度の最大値と最小値の差が 0.26(=0.59-0.33) であったのに対し，最終的には 0.09(=0.63-0.54) に縮まっている．リファクタリングの終了判定には，表 7 と図 3 の両方を使用し，総合的に判定した方がよいことが分かった．

6. まとめと今後の課題

本論文では，設計抽象化のためのリファクタリングパターンを提案した．また，パターン適用による抽象度の変化について，いくつかの観点から評価実験を実施した．5 節で述べたように，いくつかの興味深い知見が得られた．しかし，今回の評価実験はあくまでも一つの事例を対象にしたものであり，今回の結果をそのままの形で一般化することは出来ない．とは言え，少なくとも評価観点の設定や実験方法については，今後の抽象化に関わる研究の評価指針と成り得ると考えている．1 節の冒頭でも述べた通り，

© 2014 Information Processing Society of Japan

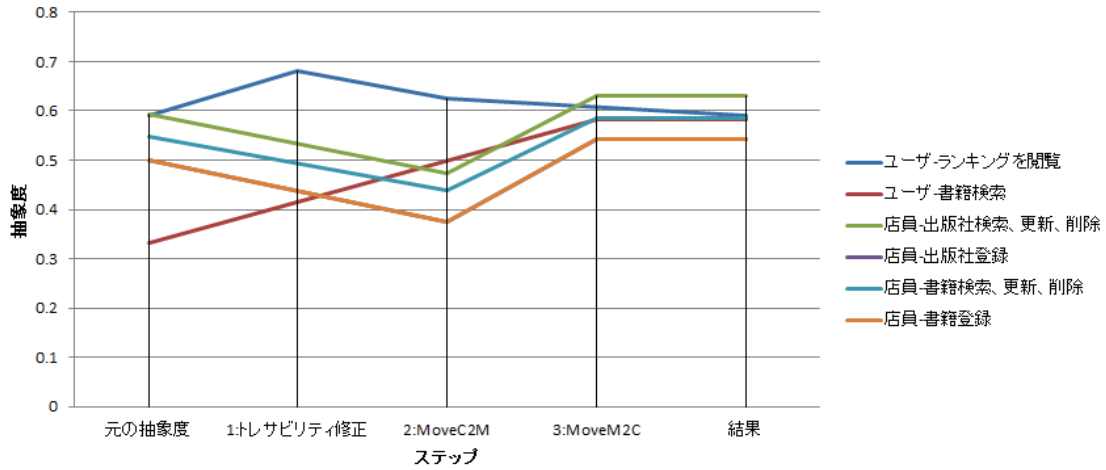

図 3 リファクタリングによる抽象度の変遷

抽象化能力はソフトウェア工学において重要な開発スキルである．「ソフトウェア開発において最も重要な能力は何か」と訊ねられたら，多くの人は「抽象化能力」と答えるであろう．しかしながら，従来のソフトウェア工学は「抽象化」に真っ正面から取り組んできたとは言い難い．もちろん，今まで抽象化支援するためのモジュール化機構，リファイメント計算など多くの研究がなされてきたのは事実である．ただ，抽象化能力の発揮支援 (抽象化支援のための知見のパターン化など)，抽象度の測定方法，抽象化の終了判定基準などの側面からの研究はほとんど行われて来なかった．理由として，理論的な取り扱いが難しかったこと，抽象度を測定するための実データがほとんど存在しなかったこと，などを挙げることが出来る．

今回，評価実験のための実データとして，九州大学で実施している PBL の開発成果物を用いた．Web 書店システムの開発は，九州大学で PBL に取り組み始めた 2007 年から 2013 年まで実施した (2014 年からは PBL 課題をリニューアルした)．7 年分の蓄積データがあるので，次のステップとして，これらを分析したいと考えている．全てではないが，開発成果物は履歴管理されているので，どのような設計モデルの変更が行われたのか，それが抽象度の変化にどう影響を与えたのかは分析可能だと考えている．一般の企業では，現在でも，必ずしもモデリングが普及しているとは言えない状況である [5]．大学では幸いなことに UML などの設計モデルが授業の一環として蓄積されており，ある意味で企業では実施が容易ではないデータ分析が可能である．学生の開発データを分析しても企業における現実の開発活動には役に立たないという意見もあるかもしれないが，5 節での分析を見て分かるように，意外にも，現実の開発現場の問題点を再現することが出来る．ソフトウェアの開発行為が存在するところは「現場」なのである．

我々は，設計抽象化を支援するモデル駆動開発ツール *iArch* を開発している．本論文で提案したリファクタリングパターンの適用自動化支援を今後この *iArch* の中に取り込んで行く予定である．

謝辞

本研究の一部は，文部科学省科学研究補助費 基盤研究 (A) (課題番号 26240007) および挑戦的萌芽研究 (課題番号 25540025) による助成を受けた．

参考文献

[1] Ai, D., Ubayashi, N., Li, P., Hosoai, S., and Kamei, Y.: iArch: An IDE for Supporting Abstraction-aware Design Traceability, 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014), pp.442-447, 2014.

[2] Egyed, A., Letier, E., and Finkelstein, A.: Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models, In *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE 2008)*, pp.99-108, 2008.

[3] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, 1999.

[4] Kramer, J.: Is Abstraction the Key to Computing? *Communications of the ACM*, Vol. 50 Issue 4, pp.36-42, 2007.

[5] Petre, M.: UML in Practice, 35th International Conference on Software Engineering (ICSE 2013), pp.722-731, 2013.

[6] Taylor, R. N. and Hoek, A.: Software Design and Architecture –The once and future focus of software engineering, In *Proceedings of 2007 Future of Software Engineering (FOSE 2007)*, pp.226-243, 2007.

[7] Ubayashi, N., Nomura, J., and Tamai, T.: Archface: A Contract Place Where Architectural Design and Code Meet Together, In *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010)*, pp.75-84, 2010.

© 2014 Information Processing Society of Japan