

プロキシオブジェクトの導入リファクタリングの 提案とその自動化

岩谷 草紀^{1,a)} 新田 直也^{1,b)}

概要: ソフトウェアの内部構造は、機能追加や不具合修正の繰り返しによって少しずつ劣化していく。内部構造の劣化の典型的な形の一つが、ソフトウェアを構成しているモジュール間の結合度の増加である。そこで我々の研究グループは、Java プログラムを対象に、クラス間の結合度が増加したソースコードに新しいクラスを導入することによって、複数クラスの疎結合化を行う複合リファクタリングを提案した。しかしながら、このリファクタリングにはクラス間の継承関係を考慮していないという問題点があった。そこで本研究では、クラス間の継承関係を考慮に入れた疎結合化リファクタリングとしてプロキシオブジェクトの導入を提案し、そのリファクタリングを自動化するアルゴリズムの検討を行う。

1. はじめに

ソフトウェアの内部構造は、機能追加や不具合修正の繰り返しによって少しずつ劣化していく。内部構造の劣化はソフトウェア全体の保守性や拡張性、柔軟性を著しく低下させるため、できるだけ早い段階で対処がなされることが望ましい。しかしながら実際のソフトウェア開発では、内部構造の改善による信頼性の低下やコストの超過を懸念して、適切な対処がなされないまま放置されることが少なくない。内部構造の劣化の典型的な形の一つが、ソフトウェアを構成しているモジュール間の結合度の増加である。結合度が高いソフトウェアでは、一部のモジュールの変更が全体の振る舞いに影響を及ぼす傾向が強くなる。

そこで我々の研究グループは、Java プログラムを対象に、複数クラスを疎結合化する複合リファクタリング^[1]を提案した。この複合リファクタリングは、疎結合化の対象となるクラスを、多数のクラスがさまざまな形で同時に参照している状況でも、新たに作成したクラスを間に介在させることによって、一度に分離することができるという特長を持つ。ただし、このリファクタリングにはクラス間の継承関係を考慮していないという問題点があった。

そこで本研究では、継承関係が存在する場合でも適用できるような疎結合化リファクタリングを拡張することを試みる。継承関係を考慮に入れた場合、新たなクラスの導入による疎結合化の方法として以下の2つが考えられる。

1つ目は、プロキシオブジェクトの導入である。プロキシオブジェクトはクライアントに対して、疎結合化の対象となるオブジェクトの代理として働くことができるオブジェクトであり^[2]、対象オブジェクトと同一のインターフェースを持つ必要がある、対象オブジェクトよりも生存期間が長いか等しいといった特徴を持つ。

2つ目は、ラッパーオブジェクトの導入である。ラッパーオブジェクトは疎結合化の対象となるオブジェクトのインターフェースを必要に応じて隠蔽する働きを持ち、対象オブジェクトとインターフェースを共通化させる必要がない、対象となるオブジェクトに対していつでも付け外しができるといった特徴を持つ。

本稿では、プロキシオブジェクトの導入に着目し、疎結合化リファクタリングと同様、どれだけ多くのクラスがどのような形で参照していたとしても疎結合化できるような汎用なアルゴリズムを考える。

2. 疎結合化のための複合リファクタリング

Java プログラムにおいてクラスは様々な形で結合する。例えば、以下のような形式でクラス A がクラス B を参照している場合、クラス A はクラス B に依存する形で結合する。

- (1) クラス A の内部で型 B のフィールドが宣言されている。
- (2) クラス A のあるメソッドが型 B の仮引数を持つ。
- (3) クラス A のあるメソッドの戻り値の型が B である。
- (4) クラス A のあるメソッドの内部で型 B の局所変数が宣言されている。

¹ 甲南大学大学院 自然科学研究科 知能情報学専攻
Konan University, Kobe, Hyogo 658-8501, Japan
a) m1924002@s.konan-u.ac.jp
b) n-nitta@konan-u.ac.jp

- (5) クラス A の内部でクラス B のコンストラクタ呼び出し式が記述されている。
- (6) クラス A の内部で戻り値の型が B であるメソッド呼び出し式が記述されている。
- (7) クラス A の親クラスが B である。

疎結合化リファクタリングは、新たなクラス C を作成して、A から B への参照の間に C を介在させる、すなわち A に C を C に B を参照させることによって、A と B の間の疎結合化を実現する。しかしながら、疎結合化のためのソースコードの書き換え方法は、元の A から B への参照の形式によって大きく異なる。また、B を参照しているクラスが A 以外にも存在し、それらすべてクラスから同時に B を疎結合化したい場合に、どこにどのような順序でどのような書き換えを行えば全体を正しく疎結合化できるかは自明ではない。

そこで我々の研究グループは、戻り値オブジェクトの導入、オブジェクトそのものの返却、隠蔽による型の置き換えという 3 つの新たなリファクタリングを導入し、これらを含む 6 つの基本リファクタリングを適切な場所に適切な順序で適用することによって疎結合化を完成させる疎結合化のための複合リファクタリング^[1](以下、疎結合化リファクタリングと略)を提案した。疎結合化リファクタリングでは、疎結合化の対象となるクラス c_{subject} 、現在 c_{subject} を参照している c_{subject} と疎結合化したいクラス群 C_{from} 、 C_{from} と c_{subject} の間に介在させるクラス新規クラス c_{new} を指定することによって、 c_{subject} を C_{from} から分離する。しかしながら、文献 [1] では、

- 上記 (7) の場合のような、クラス間に継承関係を持つような状況を考慮していなかった、
- その後の調査で、新たに導入した隠蔽による型の置き換えリファクタリングが正しく動作しない場合があることが判明した、

という問題があった。そこで本研究では、これらの問題を解決できるよう疎結合化リファクタリングの改良および拡張を試みる。

3. 継承関係を考慮した疎結合化

疎結合化の対象となるクラス c_{subject} が親クラスまたはインタフェースを持つ場合を考慮したとき、疎結合化リファクタリングを拡張する方法として考えられるのは、以下の 2 通りである。

まず、 c_{subject} の親クラスもしくはインタフェースを新規クラス c_{new} に継承させて疎結合化を行う方法である。これは C_{from} から見たときに、 c_{new} オブジェクトが c_{subject} オブジェクトの代理として働くことを意味する。このとき導

入する c_{new} オブジェクトをプロキシオブジェクトと呼び、プロキシオブジェクトを導入して行う疎結合化リファクタリングをプロキシオブジェクトの導入リファクタリングと呼ぶ。プロキシオブジェクトの導入リファクタリングは以下のような特徴を持つ。

- 新規導入するプロキシオブジェクトは疎結合化の対象となるオブジェクトと同じインタフェースを持つ。
- プロキシオブジェクトは対象オブジェクトよりも生存期間が長い。多くの場合、対象オブジェクトはプロキシオブジェクトが必要とするまで生成されない。
- 対象クラスの子クラスからでも対象クラスを疎結合化することができる。

一方、 c_{subject} の親クラスやインタフェースを c_{new} に継承させずに疎結合化する方法も考えられる。これは C_{from} から見たときに、 c_{new} オブジェクトによって c_{subject} オブジェクトのインタフェースが隠蔽されることを意味する。このとき導入する c_{new} オブジェクトをラッパーオブジェクトと呼び、ラッパーオブジェクトを導入して行う疎結合化リファクタリングをラッパーオブジェクトの導入リファクタリングと呼ぶ。ラッパーオブジェクトの導入リファクタリングは以下のような特徴を持つ。

- 新規導入するラッパーオブジェクトは疎結合化の対象となるオブジェクトとインタフェースを共有しない。多くの場合、ラッパーオブジェクトは対象オブジェクトよりも安定したインタフェースを提供する。
- 対象オブジェクトの生存期間中にラッパーオブジェクトをいつでも付け外しすることができ、一般にラッパーオブジェクトは対象オブジェクトよりも生存期間が短い。
- 対象クラスの子クラスからは対象クラスを疎結合化することができない。ラッパーオブジェクトを介在させても対象クラスのインタフェースを子孫クラスから隠蔽することができないためである。

本稿ではプロキシオブジェクトの導入リファクタリングに着目し、その自動化について考える。

4. プロキシオブジェクトの導入

本節では、プロキシオブジェクトの導入リファクタリングを自動化するアルゴリズムを提案する。疎結合化のための複合リファクタリングでは、6 つの基本リファクタリングを組み合わせて適用することで複数クラスに跨る大規模な疎結合化を実現していた。しかしながら、基本リファクタリングの 1 つである隠蔽による型の置き換えにおいて、ソースコードの書き換えが少数クラスの内部で完結

しない場合があることが判明した. そこでプロキシオブジェクトの導入リファクタリングでは, 基本リファクタリングを組み合わせるのではなく, 関係するクラスを同時に書き換えるアプローチをとる. 以下では, ソースコード全体の全クラスを C , 疎結合化される対象となるクラスを $c_{\text{subject}} (c_{\text{subject}} \in C)$, プロキシクラスとなる新規クラスを $c_{\text{proxy}} (c_{\text{proxy}} \notin C)$, c_{subject} の代用としてプロキシクラスを使用したいクラス群を $C_{\text{from}} (C_{\text{from}} \subseteq C)$ とおく. プロキシオブジェクトの導入は, c_{subject} , c_{proxy} , C_{from} を入力することで c_{subject} と C_{from} を疎結合化し, C_{from} が c_{subject} の代わりに c_{proxy} を参照するようなソースコードを生成する. ただし, 一般に型の整合性を考えたとき, 型の置き換え範囲は C_{from} 内で完結しないため, C_{from} 以外のクラス ($C \setminus (C_{\text{from}} \cup \{c_{\text{subject}}\})$) も c_{subject} の代わりに c_{proxy} をしなければならない可能性がある. そこで型の整合性を満たすように型の置き換え範囲を C_{from} から拡張する. プロキシオブジェクトの導入アルゴリズムの全体の流れは以下の通りである. 各手順の詳細は, 4.1 節以降で説明する.

- (1) c_{proxy} の作成
- (2) 型依存グラフ生成アルゴリズム
- (3) 型置換範囲決定アルゴリズム
- (4) 型置換アルゴリズム

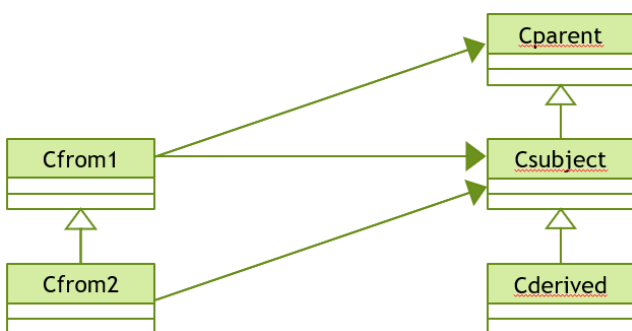


図 1: プロキシオブジェクトの導入前のクラス図の例

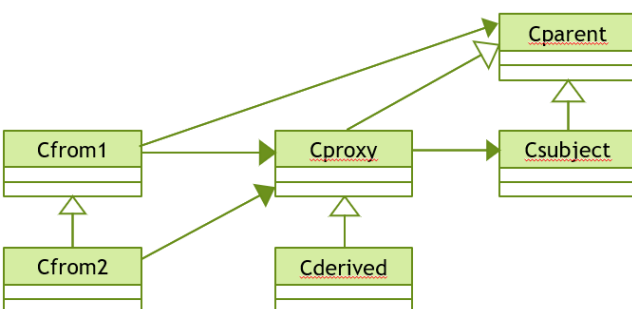


図 2: プロキシオブジェクトの導入後のクラス図の例

4.1 c_{proxy} の作成

c_{subject} の代用となるプロキシオブジェクトとして, 以下のような内部構造を持つクラス c_{proxy} を作成する.

- (1) c_{subject} 型のフィールド v を持つ.
- (2) v の値を取得するメソッド m_{get} を持つ.
- (3) c_{subject} の親クラス及びインターフェースを継承する.
- (4) c_{subject} 中の各コンストラクタ c に対して, c_{proxy} を以下のように構成する.
 - (a) c が c_{subject} 型の仮引数を含まないとき, c_{proxy} に c と同じインターフェースを持つ以下のようなコンストラクタを作成する.
 - (i) c 中の super 呼び出しと同じ super 呼び出しを行う.
 - (ii) c_{subject} 型のインスタンスを作成し, v に代入する.
 - (b) c が c_{subject} 型の仮引数を含むとき, c_{proxy} に以下のような c と同じインターフェースを持つコンストラクタ c' と,
 - (i) c 中の super 呼び出しと同じ super 呼び出しを行う.
 - (ii) c_{subject} 型のインスタンスを c と同じインターフェースを持つ以下のようなコンストラクタ呼び出しにより作成し, v に代入する.
 c の仮引数のうち, c_{subject} 型の引数を c_{proxy} 型に置換したコンストラクタ c'' を作成する.
 - (i) c の super 呼び出しと同じ super 呼び出しを行う. ただし, c_{subject} 型の実引数が求められる場合は, m_{get} 呼び出しにより c_{proxy} 型の仮引数から c_{subject} 型のインスタンスを取得したものを実引数に用いる.
 - (ii) c_{subject} 型のインスタンスを, c と同じインターフェースを持つコンストラクタ呼び出しにより作成し, v に代入する. 各実引数には c'' の各仮引数を用いる. ただし, c_{subject} 型の実引数が求められる場合は, m_{get} の呼び出しにより c_{proxy} 型の仮引数から c_{subject} 型のインスタンスを取得したものを実引数に用いる.
- (5) c_{subject} 中の各メソッド m に対して, c_{proxy} を以下のように構成する.
 - (a) m が c_{subject} 型の仮引数を含まないとき, c_{proxy} に m と同じインターフェースを持つ以下のようなメソッド m' を作成する.
 - (i) v に対して m を呼び出す. 各実引数には m' の各仮引数を用いる.
 - (b) m が c_{subject} 型の仮引数を含むとき, c_{proxy} に以下のような m と同じインターフェースを持つメソッド m'

と,

- (i) v に対して m を呼び出す. 各実引数には m' の各仮引数を用いる.

m の仮引数のうち, c_{subject} 型の引数を c_{proxy} 型に置換した以下のようなメソッド m'' を作成する.

- (i) v に対して m を呼び出す. 各実引数には m' の各仮引数を用いる. ただし, c_{subject} 型の実引数が求められる場合は, m_{get} の呼び出しにより c_{proxy} 型の仮引数から c_{subject} 型のインスタンスを取得したものを実引数に用いる.

- (6) c_{subject} が子クラスをもつ場合は, そのクラスを c_{proxy} の子クラスとする.

```

1 class Csubject extends Cparent {
2     Csubject(Csubject csubject) {
3         super(1);
4     }
5     Csubject(int a) {
6         super(a + 1);
7     }
8     int f(Cparent cparent) {
9         return cparent.f();
10    }
11 }
12 class Cproxy extends Cparent {
13     Csubject csubject;
14     Cproxy(Cproxy cproxy) {
15         super(1);
16         csubject = new Csubject(cproxy.
17             getCsubject());
18     }
19     Cproxy(Csubject csubject) {
20         super(1);
21         csubject = new Csubject(csubject);
22     }
23     Csubject(int a) {
24         super(a + 1);
25         csubject = new Csubject(a);
26     }
27     Csubject getCsubject() {
28         return csubject;
29     }
30     int f(Cparent cparent) {
31         return csubject.f(cparent);
32     }

```

図 3: Csubject クラスに対して生成される Cproxy のソースコード例

図 3 に, c_{subject} を Csubject クラス, c_{proxy} を Cproxy クラスとして生成される Cproxy クラスの例を示す. Cproxy のコンストラクタでは,

Csubject#Csubject(Csubject) に対し, Csubject 型の引数を Cproxy 型に変えた Cproxy#Cproxy(Cproxy) と, 元の引数の Cproxy#Cproxy(Csubject) が生成されていることがわかる. Cproxy#Cproxy(Cproxy) ではまず 1 行目に Csubject#Csubject(Csubject) の 1 行目にある super 呼び出しをコピーする. 2 行目では Csubject クラスのコンストラクタを呼び出すが, このコンストラクタは Csubject を引数に取るため, 仮引数の cproxy から Csubject クラスのインスタンスを取得している.

4.2 型依存グラフ生成アルゴリズム

以下では, c_{subject} 型の変数 (フィールド, 局所変数, 仮引数), c_{subject} 型の戻り値を持つメソッド本体, c_{subject} のコンストラクタ呼び出し式を c_{subject} 型を持つプログラム要素と呼び, ソースコード中の c_{subject} 型を持つプログラム要素全体からなる集合を $\mathcal{S}_{\text{subject}}$ で表す. 型依存グラフ生成アルゴリズムは, プログラム要素間の型の依存性の解決のために型依存グラフ $G_T = (V_T, E_T)$ の作成を行う. ここで, $V_T = \mathcal{S}_{\text{subject}}$, $E_T \subseteq V_T \times V_T$ とする. ソースコード中の c_{subject} 型の出現を c_{proxy} 型の出現に置換する際には, 型の整合性に注意しなければならない. 例えば右辺が c_{proxy} 型で, 左辺が c_{subject} 型である代入文の場合, 右辺から c_{subject} 型のオブジェクトを取得して, 左辺に代入することができるため, 型の不整合を解消することができる. しかしながら, 右辺が c_{subject} 型で左辺が c_{proxy} 型の場合型の不整合を解消することができない. したがって, 左辺を c_{proxy} 型に置換した場合, 右辺も c_{proxy} 型に置換する必要がある. このような, プログラム要素を c_{proxy} 型に置換する上での依存関係を表すものが, 型依存グラフである. 以下では, c_{subject} 型を持つ各プログラム要素 $e \in \mathcal{S}_{\text{subject}}$ について, e のソースコード中のすべての出現を $L(e)$ で表し, $\mathcal{L}_{\text{subject}} = \bigcup_{e \in \mathcal{S}_{\text{subject}}} L(e)$ とする. ただし, $e \in \mathcal{S}_{\text{subject}}$ がメソッド本体 m であるとき, $L(e)$ は m のすべての呼び出し式からなる集合とし, e がコンストラクタ呼び出し式 c であるとき, $L(e) = \{c\}$ とする. また, ソースコード中の e の各出現 l について, $s(l) = e$ とおく. クラス c に対して $\mathcal{S}_{\text{subject}}(c)$ を $\mathcal{S}_{\text{subject}}$ のうち c が持つものすべてとし, $\mathcal{S}_{\text{from}} = \bigcup_{c \in \mathcal{C}_{\text{from}}} \mathcal{S}_{\text{subject}}(c)$ とする. ここで, あるプログラム要素の出現が, 代入文における左辺, 仮引数宣言, メソッド呼び出し式のいずれかである場合左辺式, 代入文における右辺, 実引数, return 文の式のいずれかである場合右辺式と呼ぶものとする. ある右辺式 l_{rhs} から左辺式 l_{lhs} への値の受け渡しにおいて, $l_{\text{rhs}}, l_{\text{lhs}}$ ともに c_{subject} 型であるとき, $E_T := E_T \cup \{(s(l_{\text{rhs}}), s(l_{\text{lhs}}))\}$ という形で G_T に有向辺を追加することでプログラム要素間の依存関係を記録する.

4.3 型置換範囲決定アルゴリズム

本節では, 型依存グラフ G_T を用いて型を置換すべきプ

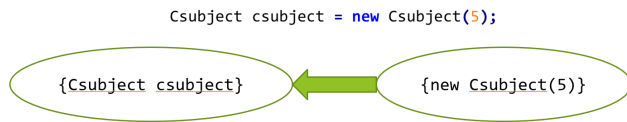


図 4: 右辺式から左辺式への依存を示す有向辺を作成するイメージ

プログラム要素を特定するアルゴリズムを導入する。このアルゴリズムにより求められる型置換の対象となるプログラム要素の集合を $S_{\text{reach}} \subseteq S_{\text{subject}}$ とし、 S_{reach} の初期集合を S_{from} とする。

4.3.1 S_{from} からの 1 段階拡張アルゴリズム

S_{from} からの 1 段階拡張アルゴリズムは、次の型置換範囲決定アルゴリズムのみでは C_{from} 内に c_{subject} 型のプログラム要素が残ることへの対処である。一般に右辺式が c_{proxy} 型で、左辺式が c_{subject} 型の場合、右辺式から c_{subject} 型のオブジェクトを取得して、左辺式に渡すことができるため、型の不整合を解消することができる。ただし、 c_{proxy} 型の実引数が C_{from} 内にある場合、 C_{from} 内では c_{subject} 型が出現することは好ましくないため、 C_{from} 内で実引数から c_{subject} 型のオブジェクトを取得して仮引数に渡すことはできない。そのため、上記の場合に限り仮引数も S_{from} に加えるように拡張することで、 c_{proxy} 型であるべきプログラム要素の範囲を拡張する。このアルゴリズムを図 6 に示す。

4.3.2 型置換範囲探索アルゴリズム

型依存グラフ G_T において、各頂点 $n \in S_{\text{reach}}$ から到達可能な範囲を調べる。ただし、到達可能範囲を調べる際は、右辺式が c_{subject} 型で左辺式が c_{proxy} 型の場合型の不整合を解消することができないことから、左辺式を基準に S_{reach} に加えるべき右辺式を探す。すなわち、 G_T における有向辺に対し逆方向にたどる。このアルゴリズムにより到達可能な範囲を加えて S_{reach} を拡張し、型を c_{subject} から c_{proxy} に置換する対象とする。

4.4 型置換アルゴリズム

ここまでの手順において作成した G_T と S_{reach} を用いて実際に型を置換する。具体的にはそれぞれの $s \in S_{\text{reach}}$ について、 s が変数である場合は s の型を c_{proxy} に、 s がメソッド本体である場合は戻り値の型を c_{proxy} に、 s がコンストラクタ呼び出し式である場合は同じ引数を持つ c_{proxy} のコンストラクタ呼び出し式に置換する。同時に、全ての $l \in L(s) (s \in S_{\text{reach}})$ の中で値の受け渡しにおいて右辺式に現れるものに対し、対応する左辺式が c_{subject} 型である場合は、右辺式に c_{subject} 型のインスタンスを取得するメソッド呼び出しを付与する。

5. 事例研究

事例研究を行うにあたり、プロキシオブジェクトの導入

の有効性を確認するため、図 7 に示す簡単な仮想プログラムを用いた検証を行った。プロキシオブジェクトの導入適用に際しての入力は、 $C_{\text{from}} = \{C_{\text{from}}\}$, $c_{\text{subject}} = C_{\text{subject}}$, $c_{\text{proxy}} = C_{\text{proxy}}$ として、著者が手作業で変更前のプログラムに対してプロキシオブジェクトの導入を行った結果を図 8 に示す。本事例において、名称変更リファクタリングは実行していないため、型名と変数名で不一致がある。なお、図 8 では紙面の都合上 C_{parent} クラス、パッケージの宣言、適用後において変化の無いクラスは省略している。図 7 のプログラムに対して、4 節で紹介したアルゴリズムから、 e をプログラム要素、 n を e が出現する行番号とし、 $e:n$ でソースコード上のプログラム要素を表すとして、 $G_T = (V_T, E_T)$ を求めると、

$$V_T = \{$$

- $\{C_{\text{subject}} c_{\text{subject}} : 13, c_{\text{subject}} : 14\}$ as s_1 ,
- $\{C_{\text{subject}} c_{\text{subject}} : 18, c_{\text{subject}} : 20,$
 $c_{\text{subject}} : 22, c_{\text{subject}} : 25\}$ as s_2 ,
- $\{new C_{\text{subject}}(3) : 18\}$ as s_3 ,
- $\{C_{\text{subject}} getC_{\text{subject}}() : 24,$
 $c_{\text{from}}.getC_{\text{subject}}() : 34\}$ as s_4 ,
- $\{C_{\text{subject}} c_{\text{subject}} : 27\}$ as s_5 ,
- $\{C_{\text{subject}} c_{\text{subject}} : 31, c_{\text{subject}} : 33\}$ as s_6 ,
- $\{new C_{\text{subject}}(4) : 31\}$ as s_7 ,
- $\{C_{\text{subject}} c_{\text{subject}}2 : 34\}$ as s_8 ,
- $\{C_{\text{subject}} c_{\text{subject}} : 36\}$ as s_9

$$\},$$

$E_T = \{\langle s_3, s_2 \rangle, \langle s_2, s_9 \rangle, \langle s_2, s_4 \rangle, \langle s_7, s_6 \rangle, \langle s_6, s_5 \rangle, \langle s_4, s_8 \rangle, \}$,
 $S_{\text{from}} = \{s_2, s_3, s_4, s_5\}$ となる。図 9 にプロキシオブジェクトの導入アルゴリズムの適用過程を示す。 S_{from} からの 1 段階拡張のアルゴリズムを用いて S_{reach} を拡張すると、 $\langle s_2, s_9 \rangle$ のエッジから S_{reach} に s_9 を追加する必要があることがわかり、 $S_{\text{reach}} = \{s_2, s_3, s_4, s_5, s_9\}$ となる。続いて型置換範囲探索アルゴリズムにより $\langle s_6, s_5 \rangle$ の有向辺から S_{reach} に s_6 を追加して、さらに $\langle s_7, s_6 \rangle$ の有向辺から S_{reach} に s_7 を追加することで、 $S_{\text{reach}} = \{s_2, s_3, s_4, s_5, s_6, s_7, s_9\}$ となる。型置換アルゴリズムを用いて型置換を実行すると図 8 のようなコードが得られる。このリファクタリング前後における各クラスの c_{subject} 型の変数の数を表 1 に示す。 C_{from} に含まれるクラスが c_{subject} と疎結合化され、 C_{from} 外のクラスの c_{subject} の参照数も減少していることがわかる。

```

function GENERATEAPPLYDEPENDENCYGRAPH( $S_{\text{subject}}$ )
   $N \leftarrow S_{\text{subject}}$ 
   $E \leftarrow \emptyset$ 
  for each  $n \in N$  do
    for each  $l \in L(n)$  do
      if  $l$  is the right hand side of assignment expression  $l2 = l$  then
        if  $l2 \in \mathcal{L}_{\text{target}}$  then
           $E \leftarrow E \cup \{(n, s(l2))\}$ 
        end if
      else if  $l$  is a return value of method  $m$  then
        for each method call  $l2$  of  $m$  do
           $E \leftarrow E \cup \{(n, s(l2))\}$ 
        end for
      else if  $l$  is an actual parameter then
        for each  $l2 \in \mathcal{L}_{\text{subject}}$  such that  $l2$  is a corresponding formal parameter of  $l$  do
           $E \leftarrow E \cup \{(n, s(l2))\}$ 
        end for
      end if
    end for
  end for
  return  $\langle N, E \rangle$ 
end function

```

▷ 代入文の右辺式から左辺式

▷ 戻り値から関数の呼び出し元

▷ 実引数から仮引数

▷ 継承を考慮すると1つとは限らない

図 5: 型依存グラフ生成アルゴリズム (GENERATEAPPLYDEPENDENCYGRAPH)

```

function EXPANDSFROM( $\mathcal{L}_{\text{subject}}, S_{\text{from}}$ )
   $S_{\text{reach}} \leftarrow S_{\text{from}}$ 
  for each  $s \in S_{\text{from}}$  do
    for each  $l \in S_{\text{from}}$  do
      if  $l$  is an actual parameter then
        for each  $l2 \in \mathcal{L}_{\text{subject}}$  such that  $l2$  is a corresponding formal parameter of  $l$  and  $s(l2) \notin S_{\text{from}}$  do
           $S_{\text{reach}} \leftarrow S_{\text{reach}} \cup \{s(l2)\}$ 
        end for
      end if
    end for
  end for
  return  $S_{\text{reach}}$ 
end function

```

▷ S_{from} 内の実引数から S_{from} 外の仮引数にわたす場合

とは限らない

▷ 継承を考慮すると1つ

図 6: C_{from} からの1段階拡張アルゴリズム (EXPANDSFROM)

```

1 class Csubject extends Cparent {
2     Csubject(Csubject csubject) {
3         super(1);
4     }
5     int f() {
6         return 2;
7     }
8     int f2(Cparent cparent) {
9         return cparent.f();
10    }
11 }
12 class Cderived extends Csubject {
13     Cderived(Csubject csubject) {
14         super(csubject);
15     }
16 }
17 class Cfrom {
18     Csubject csubject = new Csubject(3);
19     void cf() {
20         System.out.println(csubject.f());
21         Cother cother = new Cother();
22         cother.cf2(csubject);
23     }
24     Csubject getCsubject() {
25         return csubject;
26     }
27     void cf2(Csubject csubject) {}
28 }
29 class Cother {
30     void cf() {
31         Csubject csubject = new Csubject(4);

```

```

32     Cfrom cfrom = new Cfrom();
33     cfrom.cf2(csubject);
34     Csubject csubject2 = cfrom.
        getCsubject();
35 }
36 void cf2(Csubject csubject) {}
37 }

```

図 7: プロキシオブジェクトの導入適用前

```

1 class Cproxy extends Cparent {
2     Csubject csubject;
3     Cproxy(Cproxy cproxy) {
4         super(1);
5         csubject = new Csubject(cproxy.
            getCsubject());
6     }
7     Cproxy(Csubject csubject) {
8         super(1);
9         csubject = new Csubject(csubject);
10    }
11    Csubject getCsubject() {
12        return csubject;
13    }
14    int f() {
15        return csubject.f();
16    }
17    int f2(Cparent cparent) {
18        return csubject.f2(cparent);
19    }
20 }
21 class Cderived extends Cproxy {
22     Cderived(Csubject csubject) {
23         super(csubject);
24     }
25 }
26 class Cfrom {
27     Cproxy csubject = new Cproxy(3);
28     void cf() {
29         System.out.println(csubject.f());
30         Cother cother = new Cother();
31         cother.cf2(csubject);
32     }
33     Cproxy getCsubject() {
34         return csubject;
35     }
36     void cf2(Cproxy csubject) {}
37 }
38 class Cother {
39     void cf() {
40         Cproxy csubject = new Cproxy(4);
41         Cfrom cfrom = new Cfrom();
42         cfrom.cf2(csubject);
43         Csubject csubject2 = cfrom.
            getCsubject().getCsubject();
44     }

```

```

45     void cf2(Cproxy csubject) {}
46 }

```

図 8: プロキシオブジェクトの導入適用後

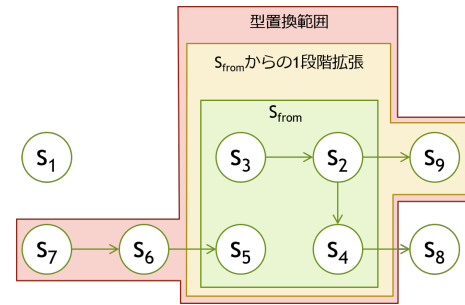


図 9: 図 7 のソースコードへのプロキシオブジェクトの導入の適用過程

6. 考察と今後の課題

本プロキシオブジェクトの導入を適用することで、正しく変更が完了し疎結合化されることが確認できた。C_{from}内のプログラム要素はすべて C_{proxy} 型となり、C_{from}に含まれないクラス内のプログラム要素も文脈次第では C_{proxy} 型となる部分があることがわかる。2 節で説明したとおり、どこにどのような順序でどのような書き換えを行えば全体を正しく疎結合化できるかは自明ではないため、手作業で疎結合化を行うと、膨大な試行錯誤が必要となる。一方、本プロキシオブジェクトの導入ではどの部分の型を置換すればよいかが一意に定まるため、疎結合化に関わる設計変更においては大規模になればなる程有用になると考えられる。今後は、本アルゴリズムを IDE 上へ実装し、OSS 等を用いて本アルゴリズムが想定する範囲における任意のプログラムに対して正常に動作が完了することを確認し、どれだけの作業量を自動化できるかを確認したい。同時に本アルゴリズムでは型の置き換えが発生することから、リネームリファクタリングを含めるかどうかの是非を含め検討する必要がある。また、本アルゴリズムの形式的な正当性の証明も今後の課題である。

7. 関連研究

リファクタリング^[3]とはソフトウェアの外的な振る舞いを変えることなく、内部構造を改善する枠組みのことである。文献[3]では72種類のリファクタリングが紹介されているが、そのいずれもが局所的な構造の変更を対象としたもので、アーキテクチャレベルの大域的な構造を対象としたものは見当たらない。文献[4]では、アーキテクチャリファクタリングの一覧が提示されており、疎結合化を目的としたリファクタリングは、この中の REDUCE DEPENDENCIES WITH FACADES に類似しているが、このリファクタリング

表 1: 各クラスの Csubject クラスへの参照数

	Csubject 型の変数の数 (導入前)	Csubject 型の変数の数 (導入後)
Cfrom	4	0
Cother	4	1

は具体的なソースコードの書換えレベルにまで詳細化はされていない。

リファクタリングの適用範囲を拡大する試みとして、いくつかの複合リファクタリングが提案されている。例えば、文献 [5], [6], [7], [8] では並列化を、文献 [9] ではライブラリ移行を目的とした複合リファクタリングが提案されているが、いずれもアーキテクチャレベルの変更を対象としているものではない。我々の研究グループの先行研究でも、疎結合化を目的とした複合リファクタリングを提案したが、このリファクタリングで使われている基本リファクタリングのうちの1つが、局所的なソースコードの書換えで完結しない場合があることが判明した。そこで、本研究ではプロキシオブジェクトの導入を単一の大きなリファクタリングとして定義することにした。

本研究と同様、文献 [10] もクラス間の依存関係を扱っているが、本研究が疎結合化を目的としているのに対し、この文献では依存関係がサイクルを形成している場合にサイクルのどの部分で切断するかを判定を扱っている。

8. おわりに

複数クラスを疎結合化する複合リファクタリングをクラス間の継承関係を考慮するよう拡張する際に、プロキシオブジェクトの導入とラッパーオブジェクトの導入という2つの拡張方法があることを示し、本稿ではプロキシオブジェクトの導入を自動化するアルゴリズムを提案した。簡単な例題プログラムに提案アルゴリズムを手動で適用し、正しく疎結合化ができることを確認した。今後、提案アルゴリズムを IDE 上へ実装し、OSS 等の設計変更例に適用することで、プロキシオブジェクトの導入リファクタリングおよび提案アルゴリズムの有効性を確認していく予定である。

参考文献

- [1] Yusuke Takahashi and Naoya Nitta: Composite refactoring for decoupling multiple classes, in *Proc. 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering ERA Track (SANER'16)*, pp. 594–598, (2016).
- [2] Erich Gamma, Richard Helm, Relph Johnson and John Vlissides: *Design patterns*. Addison Wesley, pp. 207–217, (1994).
- [3] マーチン ファウラー著, 児玉 公信, 友野 晶夫, 平澤 章, 梅澤 真史訳: リファクタリング プログラミングの体質改善テクニック. ピアソン・エディケーション, (2004).
- [4] Michael Stal: Software architecture refactoring, Tutorial of Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'07), (2007).
- [5] D. Dig, J. Marrero and M. D. Ernst: Refactoring sequential java code for concurrency via concurrent libraries, in *Proc. International Conference on Software Engineering (ICSE'09)*, pp. 397–407, (2009).
- [6] J. Wloka, M. Sridharan and F. Tip: Refactoring for reentrancy, in *Proc. Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'09)*, pp. 173–182, (2009).
- [7] F. Kjolstad, D. Dig, G. Acevedo and M. Snir: Transformation for class immutability, in *Proc. International Conference on Software Engineering (ICSE'11)*, pp. 61–70, (2011).
- [8] M. Schäfer, M. Sridharan, J. Dolby and F. Tip: Refactoring java programs for flexible locking, in *Proc. International Conference on Software Engineering (ICSE'11)*, pp. 71–80, (2011).
- [9] I. Balaban, F. Tip and R. Fuhrer, Refactoring Support for Class Library Migration, in *Proc. Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'05)*, pp. 265–279, (2005).
- [10] Tosin Daniel Oyetoan, Daniela Soares Cruzes, and Christian Thurmman-Nielsen: A decision support system to refactor class cycles, in *Proc. IEEE 31st International Conference on Software Maintenance and Evolution (ICSME)*, pp. 231–240, (2015).