Vol.2012-SE-175 No.18
2012/3/16

メソッド境界を越えた呼び出しパターン抽出のためのコールグラフ探索戦略

大場光明[†1]　渥美紀寿[†1]
小林隆志[†1]　阿草清滋[†1]

本論文では，コードの分離が行われたソースコードから，メソッド境界を越えてコードテンプレートを抽出する手法を提案する．オブジェクト指向プログラミングにおけるコードの分離をいくつかの形態に分類し，各形態におけるコールグラフ探索戦略を定義する．各形態のコードの分離が行われている箇所を自動的に検出し，分離されたコードを 1 つの連続したコードに集約して，それを抽出候補としてコードテンプレートを抽出する．

A Strategy of Call Graph Exploring for Mining Inter-method API Patterns

Mitsuaki Oba,[†1] Noritoshi Atsumi,[†1]
Takashi Kobayashi[†1] and Kiyoshi Agusa[†1]

In this paper, we propose a method to extract interprocedural code templates We classify code separations on the object-oriented program into several categories, and define the strategies of call graph exploring for each category. Detecting separated codes of each category, we extract code templates from the set of unified code.

1. はじめに

ソフトウェア開発において，開発者は様々なライブラリ，フレームワークを利用する．その中には，そのソフトウェア開発において使用しているプログラミング言語の標準ライブラリや，広く一般に用いられることを目的として公開されているライブラリ，ソフトウェア開発企業が自社の製品で利用するために独自に作成したライブラリなどが含まれうる．それらのライブラリを利用することにより，開発者はより短時間に目的のプログラムを実装できる．再利用ベースの開発では，部品の利用割合によっては，開発生産性を 10 倍以上向上させることができる[1)]．すでに多くのソフトウェアで利用され，十分にテストされたソフトウェア部品を再利用することで，新しいソフトウェアの信頼性も確保できる．

多くのライブラリ，特に広く一般に使われることを意図したライブラリは，より高度な処理もできるように大規模化し，様々な状況で利用できるよう汎用化する傾向にある．しかしその結果，ライブラリを利用したい開発者がその利用方法を理解することに困難になってきている．ライブラリに付属されるドキュメントは，ライブラリを利用する開発者がその機能や利用方法を知る基礎的な資料であるが，ライブラリに修正や機能追加があっても，ドキュメントはすぐには修正されず，全く修正されないことさえある[2)]．

この問題に対し，コードテンプレートと呼ばれる典型的なコードの書き方を既存のソフトウェアから自動的に抽出する手法が存在する．このような手法には ソフトウェアのソースコードを静的に解析して得られる情報からコードテンプレートを抽出する手法[3)] や， ソフトウェアの実行情報を動的に解析して得られる情報から抽出する手法[4)] などがある． 前者がソースコードさえあれば適用できるのに対し，後者はソフトウェアを実行できる環境と十分なシステムテストを必要とする．本研究では静的に解析する手法に着目する．

ソフトウェア開発ではコードの分離が推し進められる傾向にある．すなわち，1 つの機能を実現するコードが長くなる場合，それをそのままソースコード上で連続して記述するのではなく，一定のまとまったサブ機能ごとに新しい関数として抽出し，分割してコードを記述する．コードの分離が行われ，連続するコードが短くなると開発者にとっては可読性や保守性が向上する．しかし，静的解析によるコードテンプレート抽出手法の多くは，連続して記述されたコードを抽出候補としている．コードの分離が進んだソースコードからは，マイニングの結果得られるコードテンプレートも短いものに限られる．

本論文では，コードの分離が行われたソースコードから，メソッド境界を越えてコードテンプレートを抽出する手法を提案する．オブジェクト指向プログラミングにおけるコードの分離をいくつかの形態に分類し，形態ごとにコールグラフを探索する戦略を定義した．各形態のコードの分離が行われている箇所を自動的に検出し，分離されたコードを 1 つの連続したコードに集約して，それを抽出候補としてコードテンプレートを抽出する．

†1 名古屋大学大学院 情報科学研究科
Graduate School of Information Science, Nagoya University

提案手法の有用性を評価するため，提案手法を実装し，オープンソースソフトウェアからコードテンプレートを抽出する実験を行った．実験によって，メソッド境界を越えて現れるコードテンプレートを抽出できることを確認した．提案手法を適用せずにコードテンプレートを抽出した場合と比べ，より長いコードテンプレートが抽出できることを確認した．

2. 関連研究

本章では，コードテンプレート抽出手法など，典型的に書かれるコードをデータマイニングを用いて抽出する既存の手法について，静的解析によるものと動的解析によるものに分けて説明する．また，我々の提案手法との相違点について述べる．

2.1 静的解析

Xie と Pei が開発した MAPO[3] は，開発者がライブラリ API の使い方を理解するのを助けるため，ソースコード検索エンジンでの検索結果を元に頻出する API 呼び出しの系列パターンを抽出して提示する．利用者からメソッド名やパッケージ名などのクエリを受け取ると，MAPO はソースコード検索エンジンに問い合わせて，そのクエリを含むソースコードを取得する．取得したソースコードからメソッド呼び出し系列を取得し，系列パターンマイニングを用いて頻出する系列パターンを抽出する．MAPO はマイニングの事前処理として，メソッド呼び出しのインライン展開を行う．ただし，我々の提案手法がメソッド間の関係を考慮して展開するか判断するのに対し，MAPO は同じソースコード内のメソッドを無条件に 3 段階まで展開する．

Zhenmin らによる PR-Miner[5] は，C 言語で書かれたソースコードを対象に，頻出する関数呼び出しの組み合わせをプログラミングルールとして抽出する．さらに PR-Miner は抽出されたプログラミングルールを用いてバグ検出を行う．高い頻度で出現する関数の組み合わせがあったとき，その組み合わせのうち一部だけが欠けている箇所は，バグを含んでいる可能性が高い．PR-Miner はそのような箇所を自動的に検出し，開発者に警告する．PR-Miner はプログラミングルールの抽出に頻出アイテムセットマイニングというアルゴリズムを用いており，関数呼び出しの順序関係を考慮しない．また，関数をまたいでプログラミングルールが用いられる可能性を考慮していない．

Acharya らは，ライブラリ API 間の利用シナリオを半順序として自動的に抽出する手法を提案した[6]．Acharya らはプッシュダウンモデル検査の処理を応用し，静的解析でありながら，動的解析で得られるような関数境界を超えたトレースを収集する方法を考案した．ただし，あらかじめユーザーが指定したいくつかの API 呼び出しだけからなるトレースを収集するものであり，任意の API を対象としてはいない．

Nguyen らは，複数のオブジェクトが関わる利用パターンをマイニングする GrouMiner[7] を開発した．Nguyen らは，メソッド呼び出しや制御構造をノード，利用順序とデータ依存関係をエッジとする有向グラフ表現 Groum を定義し，複数の Groum に頻出する部分グラフをコードテンプレートとしている．GrowMiner は制御構造を含むコードテンプレートを抽出することができるが，メソッド境界を越えたコードテンプレートは抽出できない．

2.2 動的解析

Salah らが考案した Scenariographer[8] は，実行時に行われるメソッド呼び出しからクラス利用シナリオを表すメソッド呼び出し系列を生成する．対象ソフトウェアの実行時に生成されたオブジェクトごとに，それに対するメソッド呼び出しを系列として抽出する．メソッド呼び出し系列の canonical set と呼ばれるものを計算することで，一定の利用シナリオで用いられるメソッド呼び出しのパターンを抽出する．

Pradel らは，複数のオブジェクトからメソッド呼び出し系列の仕様を自動生成することで，API の典型的な使い方を提示する手法を提案した[4]．Object collaboration と呼ばれるオブジェクト間の関係を検出し，関連のある複数のオブジェクトのメソッド呼び出し系列から有限状態機械の形で API 仕様を導出する．

動的解析では，連なって抽出されるメソッド呼び出し系列のどこからどこまでが 1 つの機能を実現する処理であるか判断する必要がある．Scenariographer では，メソッド呼び出しがどのインスタンスに対するものか，という観点からメソッド呼び出し系列を分解している．この方法だと複数のオブジェクトが協調して実現される処理は抽出できない．一方 Pradel らの手法では，メソッド呼び出しを一定の深さまでだけたどることで，メソッド呼び出し系列を一定の長さに抑えている．このとき，外部に対し非公開のメソッドの呼び出しは深さの計算に入れない，という考慮をしており，我々の提案手法と類似している．

3. メソッド境界を越えたコードテンプレート抽出手法

3.1 手法の概要

本手法の目的は，ソースコードを解析し，そこからライブラリの利用方法の理解に役立つコードテンプレートを抽出することである．本手法では，保守性などの理由から，一定の機能を実現するコードがメソッドの抽出などを用いて複数の箇所に分離されている，と仮定する．そのようなコードの分離が行われている箇所を検出し，分離されたコードを 1 つの連続したコードに集約する．集約されたコードをコードテンプレートの候補としてマイニングを

行うことで，メソッド境界を越えて現れる長いコードテンプレートを抽出できる．

オブジェクト指向プログラミングにおけるコードの分離の形態として，次の 3 つを考える．

- クラス内でのコードの分離
- 継承によるコードの分離
- 委譲によるコードの分離

以下，それぞれの形態の詳細と，その形態を検出する方針，各場合におけるコールグラフ探索戦略を述べる．

3.2 クラス内でのコードの分離の集約

オブジェクト指向プログラミングにおけるコードの分離は，分離したコードを同じクラスに置くか，異なるクラスに置くかで大別できる．ここでは，同じクラスの複数のメソッドにコードを分離している場合を考える．

多くのオブジェクト指向プログラミングでは，メソッドにアクセス権が設定できる．例えば Java において，**private** 修飾子が付けられたメソッドはそのメソッドの属するクラスの外部に対し非公開となり，同じクラスの他のメソッドからしか呼び出すことができない．一方，**public** 修飾子が付けられたメソッドはクラスの外部に公開され，任意のクラスから呼び出すことができる．

外部に公開されたメソッドは，そのメソッドの属するクラスにおいてインターフェースとなり，そのクラスに割り当てられた一定のまとまった機能を受け持っている．一方，非公開のメソッドは，クラスのインターフェースを実装する上で不要であるが，公開されたメソッドのコードが冗長となり，可読性や保守性が低下するのを抑制するために作成される．

このようなメソッドの関係から，**private** や **protected** などの非公開のメソッドのコードを，公開されているメソッドのコードに集約させることで，一定のまとまった機能を実装する上で必要になるすべてのコードを 1 つに集約させることができる．

3.3 継承を用いた分離の集約

継承は，オブジェクト指向プログラミングを象徴する概念の 1 つであるとともに，委譲など様々な技法に活用される機構でもある．まず，継承を直接利用してコードを分離するケースについて考える．

継承を用いたコードの分離には，分離されたコードの参照関係から，2 つのケースに分けられる．

- 親クラスで実装されているメソッドを子クラスが利用するケース（**図 1**）
- 親クラスが利用している（抽象）メソッドを子クラスがオーバーライドするケース（**図 2**）

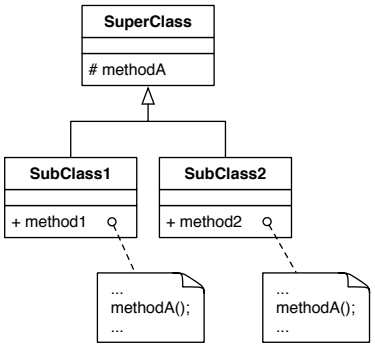

図 1 継承を用いたコードの分離 (1) 親クラスで実装されているメソッドを子クラスが利用するケース
Fig. 1 Code Separation by Inheritance (1) Subclasses Use Methods Implemented by Superclass

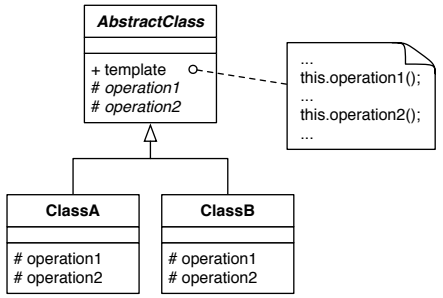

図 2 継承を用いたコードの分離 (2) 親クラスが利用している（抽象）メソッドを子クラスでオーバーライドするケース
Fig. 2 Code Separation by Inheritance (2) Superclass Uses (Abstract) Methods Overrided by Subclasses

以下，それぞれについて議論する．

3.3.1 親クラスで実装されたメソッドを子クラスが利用するケース

共通の親クラスを継承する複数の子クラスにおいて，様々な機能を実現するために同じようなコードを用いている場合，そのコードをメソッドに抽出し，親クラスに引き上げることがある．例えば**図 3** において，JxElement クラスを継承している JxField クラスと JxMethod

```
public class JxElement {
        protected Element getTypeElem(elem) {
                return (Element) elem.getElementsByTagName("Type").item(0);
        }
}
```

```
public class JxField extends JxElement {
        public JxType getType() {
                Element typeElem = getTypeElem(getElem());
                if (typeElem == null) {
                        return null;
                } else {
                        return new JxType(typeElem);
                }
        }
}
```

```
public class JxMethod extends JxElement {
        public String getSignature() {
                StringBuffer buf = new StringBuffer();
                for (Element paramElem : getParameterElems()) {
                        Element typeElem = getTypeElem(paramElem);
                        buf.append(",␣");
                        buf.append(new JxType(typeElem).getFqn());
                }
                ...
        }
}
```

図 3 親クラスで定義されたメソッドを子クラスが利用している例（抜粋）
Fig. 3 An Example of Subclasses Using Methods Implemented by Superclass

クラスは，それぞれ異なる目的で，JxElement クラスの getTypeElem メソッドを利用している．このような場合，親クラスのメソッドは子クラスの機能を実現するためだけにあるので，メソッドのアクセス権は限定的なものとなる．図 3 の例でも，getTypeElem メソッドは **protected** 修飾子によって，継承された子クラスからのみ呼び出せるようにしている．

図 3 の例では，JxField クラスや JxMethod クラスのそれぞれのメソッドの機能を実現するために，外部ライブラリの API である Element クラスの getElementsByTagName メソッドが間接的に利用されている．しかし，JxField クラスや JxMethod クラスのソースコードを解析するだけでは，getElementsByTagName メソッドが利用されるという情報は得られない．

そこで，親クラスのメソッド，特に子クラスに限定公開しているメソッドのコードは，そのメソッドを呼び出している子クラスのコードに集約する．そうすることで，子クラスの特定の機能を実現するために必要なコードをまとまった形で処理できるようになる．

3.3.2 親クラスが利用しているメソッドを子クラスがオーバーライドするケース

親クラスを継承している子クラスは，親クラスが定義しているメソッドの一部を定義し直し，動作を上書き（オーバーライド）することができる．オーバーライドによって，親クラスで実装された機能の一部を子クラスの特徴に合わせて修正することができる．

特に，ある一連の処理の中で，振る舞いが変わりうる部分を，子クラスでオーバーライドできる別のメソッドの呼び出しとしておくことで，処理の基礎部分を 1 つのコードにまとめつつ，様々なバリエーションを容易に作成できるようになる．Gamma らは，このような設計は「コードの再利用のための基本的な方法」であるとし，Template Method パターンと呼んだ[9]．

図 4，**図 5** は Template Method パターンの例である．親クラスである AttributeFigure クラスの draw メソッドは，処理の途中で抽象メソッドである drawBackground メソッドと drawFrame メソッドを呼び出している．これらのメソッドは AttributeFigure クラスを継承した各クラスで定義され，例えば RectangleFigure クラスなら四角形の描画，TextFigure クラスなら文字列の描画をするライブラリ API が実行されている．

このようなクラス構造では，親クラスだけ，あるいは子クラスだけを解析しても，処理の全容はつかめない．そこで，親クラスのコードで呼び出されているメソッドが，子クラスでオーバーライドされている場合，オーバーライドしているそれぞれの子クラスで，子クラスのメソッドのコードと親クラスのコードを集約する．

3.4 委譲を用いた分離の集約

委譲は一般的に**図 6** のような構造で実装される．すなわち，委譲を行うオブジェクトは委譲先のオブジェクトをインスタンスフィールドに保持し，自身の機能の一部を委譲する際に，そのフィールドのオブジェクトに対しメソッド呼び出しを行う．委譲先となるオブジェクトを状況に応じて入れ換えるようにすることで，同じ委譲元オブジェクトのメソッド呼び出しを，それぞれの状況に合った処理に変えることができる．

しかし，インスタンスフィールドに保持したオブジェクトに対しメソッド呼び出しを行う，という操作は，オブジェクト指向プログラミングにおいて極めて一般的なものであり，処理

```
public abstract class AttributeFigure extends AbstractFigure {
        /**
         * Draws the figure in the given graphics. Draw is a template
         * method calling drawBackground followed by drawFrame.
         */
        public void draw(Graphics g) {
                Color fill = getFillColor();
                if (!ColorMap.isTransparent(fill)) {
                        g.setColor(fill);
                        drawBackground(g);
                }
                Color frame = getFrameColor();
                if (!ColorMap.isTransparent(frame)) {
                        g.setColor(frame);
                        drawFrame(g);
                }
        }

        protected void drawBackground(Graphics g) {
        }
        protected void drawFrame(Graphics g) {
        }
}
```

図 4 Template Method パターンが適用された親クラスの例（抜粋）
Fig. 4 An Example of Template Method Pattern Application (Superclass)

```
public class RectangleFigure extends AttributeFigure {
...
        public void drawBackground(Graphics g) {
                Rectangle r = displayBox();
                g.fillRect(r.x, r.y, r.width, r.height);
        }

        public void drawFrame(Graphics g) {
                Rectangle r = displayBox();
                g.drawRect(r.x, r.y, r.width−1, r.height−1);
        }
...
}
```

```
public class TextFigure
        extends AttributeFigure
        implements FigureChangeListener, TextHolder {
...
        public void drawBackground(Graphics g) {
                Rectangle r = displayBox();
                g.fillRect(r.x, r.y, r.width, r.height);
        }

        public void drawFrame(Graphics g) {
                g.setFont(fFont);
                g.setColor((Color) getAttribute(FigureAttributeConstant.TEXT_COLOR));
                FontMetrics metrics = Toolkit.getDefaultToolkit().getFontMetrics(fFont);
                Rectangle r = displayBox();
                g.drawString(getText(), r.x, r.y + metrics.getAscent());
        }
...
}
```

図 5 Template Method パターンが適用された子クラスの例（抜粋）
Fig. 5 An Example of Template Method Pattern Application (Subclass)

を委譲する場合に限られない．前述した構造だけでなく，より詳細な条件をもって委譲を用いたコードの分離であるか判定すべきである．本研究では，委譲の場合によくみられる特徴として，呼び出される委譲先のメソッドは，呼び出し元のメソッドと同名に設定される点に着目した．前述した構造に加え，呼び出し元と呼び出し先のメソッド名が同一の場合に，呼び出し先のメソッドのコードを呼び出し元に集約する．

3.5 提案手法の実装

本節では，提案手法を実装したコードテンプレート抽出ツールについて述べる．本ツールは以下の 4 つのルーチンから構成される．

構文解析ルーチン ソースコードの解析を容易にするため，ソースコードを構文解析し，抽象構文木に変換する．変換には Sapid/XML Tool Platform[10] を用いた．

構文木書き換えルーチン クラス内でのコードの分離，および継承によるコードの分離を集約するため，メソッド呼び出しのインライン展開を行う．
各クラスについて，まず，継承している先祖クラスのメソッドのコードを複製し，挿入する（オーバーライドしている場合を除く）．次いで，先祖クラスから複製してきたものも含めた各 **public** メソッドについて，同じクラス，または先祖クラスの **public** では

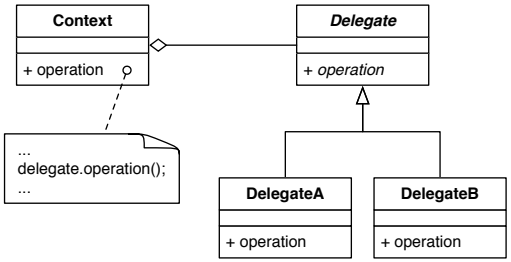

図 6 委譲を用いたコードの分離
Fig. 6 Code Separation by Delegation

ないメソッドへの呼び出しをインライン展開する．すなわち，抽象構文木において呼び出し先のメソッドの本体に当たる部分木を複製し，呼び出し部分に挿入する．

特徴抽出ルーチン ソースコードからコードテンプレートの候補となるコードの特徴を抽出する．本ツールでは，メソッド呼び出しの系列をコードテンプレートの候補とした．メソッド呼び出し系列は，1 つのメソッド定義を一単位として，その中に含まれる全てのメソッド呼び出しを一つのトランザクションとして扱った．各メソッド呼び出しは，呼び出されるメソッドの名前と，そのメソッドの属するクラスの完全修飾名で区別した．委譲によるコードの分離は，このルーチンの過程で集約した．本ツールでは，委譲先のオブジェクトが委譲元のオブジェクトのフィールドに保存され，そのフィールドを通じて，委譲元で呼び出されたのと同名のメソッドが呼び出される場合のみ，委譲によるコードの分離が行われていると判定する．このような構造が検知された場合，委譲先へのメソッド呼び出しを，呼び出されるメソッドに記述されたメソッド呼び出しの系列に置換する．

マイニングルーチン 抽出されたコードテンプレートの候補に対し，系列パターンマイニングを適用し，頻繁に用いられるコードテンプレートを発見する．マイニングアルゴリズムには Prefixspan[11] を用いる．

4. 評価実験

4.1 実験方法

提案手法の有用性を評価するため，提案手法の実装を用いて，オープンソースソフトウェアからコードテンプレートを抽出する実験を行った．実験の対象ソースコードとして，オープンソースソフトウェアの JHotDraw 5.4b1[⋆1]（608 クラス，5357 メソッド，72,922 LOC）を用いた．これに 3.5 節で述べたコードテンプレート抽出ツールを適用し，分離されたコードの集約を行ったうえでコードテンプレートを抽出した．ミニマムサポートを 2 から 300 まで変化させながら，2 つ以上の要素からなる頻出系列パターンを抽出した．また，コードの集約を行わずに特徴抽出とマイニングだけを行う対照実験も行い，分離されたコードの集約を行ったことでどのようなコードテンプレートが取れるようになったか調べた．

表 1 抽出されたメソッド呼び出し系列の長さ
Table 1 The Lengths of Extracted Method Call Sequences

	平均値	標準偏差
集約しない場合	4.22	32.71
クラス内でのコードの分離を集約	4.49	39.29
クラス内+継承を用いた分離の集約	4.69	76.92
クラス内+継承+集約を用いた分離の集約	5.77	126.93

4.2 実験結果

提案手法を用いてコードの集約を行った場合と行わなかった場合とで，コードテンプレートの候補として抽出されるメソッド呼び出し系列の長さを比較した．**表 1** に示されるように，提案手法を適用したことで平均値，標準偏差のどちらも増加した．

さらに，コードテンプレートとして抽出される出現頻度の閾値（ミニマムサポート）を変えながら，いくつコードテンプレートが得られるか調査した．**図 7** に示されるように，集約するコードの分離の形態が増えるごとにどのミニマムサポートにおいてもコードテンプレートの数が増えていることがわかる．

抽出されたコードテンプレートの中から，メソッド境界を越えて複数のメソッド呼び出しが抽出されたものを調査した．ここでは，分離されたコードの集約を行った場合のみ抽出された，次のコードテンプレートについて説明する．なお，“#”の左が呼び出されているメソッドの属するクラス，“#”の右がメソッド名を表す．

- javax.swing.JFrame#getJMenuBar
  → javax.swing.JMenuBar#getMenuCount
  → javax.swing.JMenuBar#getMenu
  → javax.swing.JMenu#getItemCount

⋆1 JHotDraw as Open-Source Project ⟨http://www.jhotdraw.org/⟩

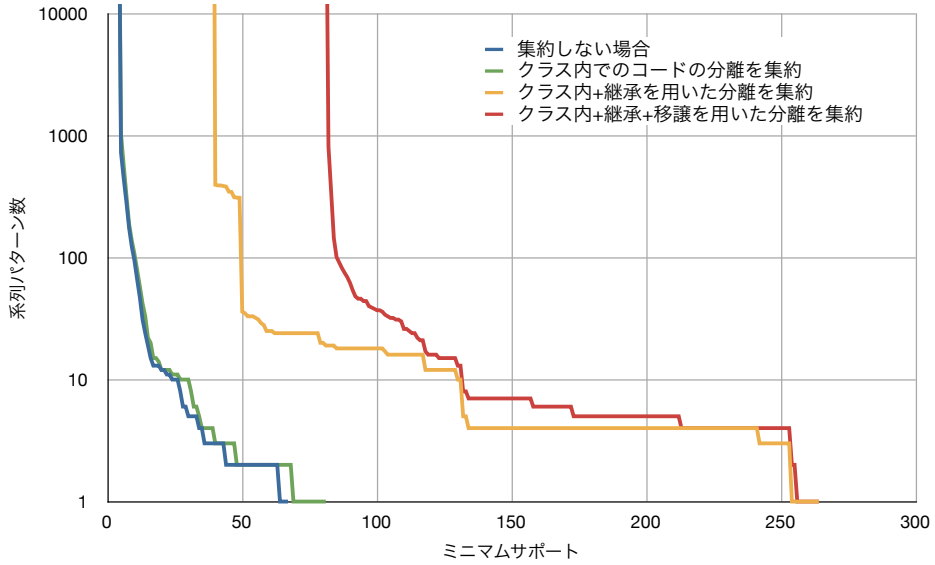

図 7 抽出されたコードテンプレートの数
Fig. 7 Amounts of Extracted Code Templates

→ javax.swing.JMenu#getItem

これは，Swing のウィンドウフレームからメニューバーを取得し，各メニューバーからメニュー項目を取得する処理を表すコードテンプレートである．

このコードテンプレートは JHotDraw の org.jhotdraw.application.DrawApplication クラスに存在した．**図 8** に出現箇所のソースコードを抜粋して示す．中央の checkCommandMenus メソッドで呼び出されている公開メソッドは getJMenuBar と getMenuCount，getMenu メソッドのみであり，その下の checkCommandMenu メソッドで呼び出されている公開メソッドは getItemCount と getItem メソッドのみである．このため，ソースコード上で連続したコードのみをコードテンプレートの抽出候補としている場合，これら 5 つのメソッドがすべて含まれるコードテンプレートは得られない．しかし，checkCommandMenus メソッドの中に公開メソッドではない checkCommandMenu メソッドの呼び出しがあったことから，本手法によりコードの集約が行われ，メニューバーを取得してからメニュー項目を取得するまでのまとまった一連の処理を示すコードテンプレートとして抽出できた．

```
public class DrawApplication
    extends JFrame
    implements DrawingEditor, PaletteListener, VersionRequester {
...
    public void figureSelectionChanged(DrawingView view) {
        checkCommandMenus();
    }

    protected void checkCommandMenus() {
        JMenuBar mb = getJMenuBar();

        for (int x = 0; x < mb.getMenuCount(); x++) {
            JMenu jm = mb.getMenu(x);
            if (CommandMenu.class.isInstance(jm)) {
                checkCommandMenu((CommandMenu)jm);
            }
        }
    }

    protected void checkCommandMenu(CommandMenu cm) {
        cm.checkEnabled();
        for (int y = 0; y < cm.getItemCount();y++) {
            JMenuItem jmi = cm.getItem(y);
            if (CommandMenu.class.isInstance(jmi)) {
                checkCommandMenu((CommandMenu)jmi);
            }
        }
    }
...
}
```

図 8 JMenubar に関するコードテンプレートが利用されているコード（抜粋）
Fig. 8 An Application of a Code Template About JMenubar

4.3 考　　察

前節で示したように，分離したコードの集約を行うことで，従来のアプローチでは得られないコードテンプレートが抽出できることが確認できた．また，コードテンプレートの候補となるメソッド呼び出し系列がより長くなっていることがわかった．このことから，抽出されたコードテンプレートも従来のアプローチで得られるものより長くなっている可能性がある．

しかし，コードの集約を行った結果，抽出されるコードテンプレートの数が著しく増大している．この原因として，より長いコードテンプレートが抽出されるようになったことで，その部分パターンもまたコードテンプレートとして抽出されてしまっているとみられる．

5. おわりに

5.1 まとめ

本研究では，静的解析を用いながらメソッド境界を越えてコードテンプレートを抽出する手法を提案した．オブジェクト指向プログラミングにおいて保守性を高めるために行われる，コードの分離をいくつかの形態に分類し，形態ごとにコールグラフを探索する戦略を定義した．提案手法では，コードの分離が行われている箇所を自動的に検出し，分離されたコードを 1 つの連続したコードに集約し，それを抽出候補としてコードテンプレートを抽出する．

提案手法を実装したツールを作成し，オープンソースソフトウェアからコードテンプレートを抽出する実験を行った．実験の結果，メソッド境界を越えて現れるコードテンプレートを抽出できることを確認した．提案手法を適用しない場合と比べ，より長いコードテンプレート候補が得られることがわかった．抽出されるコードテンプレートの数が増加することに関する課題を議論した．

5.2 今後の課題

今後の課題として，抽出されたコードテンプレートのさらなる評価が必要である．しかしそのためには，長いコードテンプレートが抽出されることに伴う，部分コードテンプレートの爆発的増加が問題となる．そのような部分パターンをより長いパターンにまとめて取り扱う，飽和系列パターンマイニングアルゴリズムを採用する予定である．

今回の実験では単一の小規模なソフトウェアを対象に行なったが，より大規模なソフトウェアでの評価も必要である．あるいは複数のソフトウェアをまとめて解析することで，プロジェクトをまたがって現れる普遍的なコードテンプレートを抽出できるはずである．

また，今回実装したツールでは特に委譲を用いたコードの分離に対し，単純な実装でのみ検出されるようなかなり強い条件を設けた．今後，例えばデザインパターンに関する知見を用いて，より柔軟な判定条件に修正するべきだろう．

さらに，コードテンプレートの候補としてメソッド呼び出し系列を用いたが，これに限らず，制御構造や他の構文要素も含めることができる．関連研究に挙げた構文木や依存グラフを用いる手法を本論文の提案手法と組み合わせることで，より有用性の高いコードテンプレートが抽出できるようになると期待される．

参考文献

1) 竹下　亨：ソフトウェアの保守・再開発と再利用，共立出版 (1992).
2) Lethbridge, T.C., Singer, J. and Forward, A.: How software engineers use documentation: the state of the practice, *IEEE Software*, Vol.20, No.6, pp.35–39 (2003).
3) Xie, T. and Pei, J.: MAPO: mining API usages from open source repositories, *Proc. MSR*, Shanghai, China, ACM, pp.54–57 (2006).
4) Pradel, M. and Gross, T.R.: Automatic Generation of Object Usage Specifications from Large Method Traces, *Proc. ASE*, Washington, DC, USA, IEEE Computer Society, pp.371–382 (2009).
5) Li, Z. and Zhou, Y.: PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code, *Proc. ESEC/FSE*, Lisbon, Portugal, ACM, pp.306–315 (2005).
6) Acharya, M., Xie, T., Pei, J. and Xu, J.: Mining API patterns as partial orders from source code: from usage scenarios to specifications, *Proc. ESEC/FSE*, New York, NY, USA, ACM, pp.25–34 (2007).
7) Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M. and Nguyen, T.N.: Graph-based mining of multiple object usage patterns, *Proc. ESEC/FSC*, Amsterdam, The Netherlands, ACM, pp.383–392 (2009).
8) Salah, M., Denton, T., Mancoridis, S. and Shokouf, A.: Scenariographer: A Tool for Reverse Engineering Class Usage Scenarios from Method Invocation Sequences, *Proc. ICSM*, Budapest, Hungary, IEEE Computer Society, pp.155–164 (2005).
9) Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).
10) Maruyama, K. and Yamamoto, S.: A CASE Tool Platform Using an XML Representation of Java Source Code, *Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, Washington, DC, USA, IEEE Computer Society, pp.158–167 (2004).
11) Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U. and Hsu, M.-C.: PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth, *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, Los Alamitos, CA, USA, IEEE Computer Society, pp.215–224 (2001).