

オブジェクト指向クラス間依存解析に基づく コンポーネント抽出

Component Extraction based on OO-Class Relation Analysis

鷺崎 弘宜[†]

Hironori WASHIZAKI

深澤 良彰[†]

Yoshiaki FUKAZAWA

[†] 早稲田大学理工学部

School of Science and Engineering, Waseda University

{washizaki, fukazawa} @fuka.info.waseda.ac.jp

既存のオブジェクト指向プログラムを、クラス間の継承関係や参照関係、および生成関係について静的に解析し、ソフトウェア部品の単位として独立して再利用可能なクラス群を特定し、単体で生成可能なコンポーネントにまとめる手法を提案する。同時に、コンポーネントを利用する周辺部分を修正することでコンポーネントを生成・利用する例を得る手法を実装した結果について述べる。

1 はじめに

ソフトウェアを機能単位で部品に分解し、得られたソフトウェア部品を組み合わせることによって開発を進めるコンポーネント指向開発技術が、開発コストの削減とソフトウェア全体の信頼性の向上を目的として注目されている。コンポーネント指向開発技術は、コンポーネント群を合成する技術と、既存のソフトウェアをコンポーネント群に分解する技術から成り立つ。

本稿では、既存ソフトウェアからのコンポーネントの自動抽出を実現するために、コンポーネントを明確に定義し、コンポーネントとすべき領域(部分)を自動的に特定し、さらに、特定された領域をコンポーネントとして使用する場合に必要となる周辺部分の修正手法を提案する。

2 オブジェクト指向とコンポーネント

コンポーネントはオブジェクト指向言語で実装することが自然であると指摘されるため [1]、本稿ではオブジェクト指向言語に基づくコンポーネントを扱う。コンポーネント指向開発を進めるにあたっては、必要なコンポーネントが既に存在する必要がある。しかしながら、再利用可能なコンポーネントを全ての状況に応じて事前に準備することは不可能である。一方、目的とする機能要件を(部分的に)満たすソフトウェアが存在することは一般にあるが、そのソフトウェア全体のままでは用いるべき部分を特定するの

が困難であり、利用者が開発したソフトウェアでなければ効果的に再利用できない。そこで、既存のオブジェクト指向ソフトウェアから目的に合致するコンポーネントを容易に抽出することが重要である。

コンポーネントだけを抽出しても、適切に再利用することは難しいため、利用方法を説明する例としてのコンポーネント利用プログラムを提供することがある。このために、既存のソフトウェアに対して外部に対する振舞いを保ったまま内部設計を変更するリファクタリング操作 [2] によってコンポーネントを抽出することで、コンポーネントと共にコンポーネントを利用するプログラム例を得ることができる。しかし、コンポーネントとすべき領域の特定について開発者の主観的判断によるところが大きい。本稿では、既存のオブジェクト指向ソフトウェアを解析することでクラス関連グラフを作成し、同グラフ上で領域到達可能を自動的に判断することでコンポーネント化すべき領域を特定する。

2.1 コンポーネントの性質

コンポーネント一般の構造的な定義は、独立して交換可能な、一つ以上のオブジェクト指向クラス群である [2]。しかし、この定義は交換可能であることの条件が不明確であり、この曖昧さが再利用可能なコンポーネントの開発・再利用の妨げとなっている。そこで、コンポーネントの明確な定義を行うに辺り、コンポーネントが持つべき性質として再利用性と独

立性を考える。

オブジェクト指向開発において、構造的な再利用の単位はクラス・クラスライブラリ・フレームワーク・コンポーネントの 4 種である。各単位において一般に満される再利用目的・独立性の対比を表 1 に示す。構造とは、クラス間の静的な関連構成と動的な制御の流れを意味する。

表 1: 再利用性と独立性の対比

単位	再利用目的	独立性
クラス	部分的な機能	他クラスに依存
ライブラリ	機能	個別生成可能
フレームワーク	機能・全体構造	部分実装が必要
コンポーネント	機能・部分構造	個別生成可能

2.2 コンポーネントの定義

前節のコンポーネントが満すべき性質に基づき、コンポーネントの明確な定義を以下に与える。

定義 1 (コンポーネント)

次の条件 1 ~ 条件 4 を満たして、Java 言語に従って作成されたクラス群をコンポーネントと呼ぶ。本稿ではコンポーネントシステムとして JavaBeans[3] を前提とする。しかし、以下の定義はこれに拘束されるものではない。以降において、クラスとは Java の実行系と共に標準で配布される各種パッケージ群 (java.*, javax.* などの基本クラス群) 中のクラス以外のものを指す。コンポーネント化にあたって、ソフトウェア開発者が独自に開発したものではない標準的なクラス群は考慮しない。

条件 1: JavaBeans コンポーネントである

引数を持たない外部から呼出し可能なコンストラクタ (public なデフォルトコンストラクタ) を持ったクラスを Facade パターンにおける Facade 役クラスとして一つ持ち、そのクラス自身が java.io.Serializable インタフェースを実装するか、または Serializable インタフェースを実装したクラスを継承している。

コンポーネントは、Facade 役クラスの上記コンストラクタの起動により単独で生成可能な必要がある。そのため、コンポーネントを利用する実行環境中に生成に必要な全ての関連クラス・インタフェースが存在する必要がある。コンポーネントを配布する際には、Facade 役クラスと共にそれらの関連クラス・インタフェースをまとめて Jar ファイルへパッケージ化する。

条件 2: 外部へ Facade 役を唯一公開している

コンポーネントに対するメソッド呼出しは、全てコンポーネントが唯一持つ Facade 役クラスを通

して行われる。ただし、コンポーネントを構成する Facade 役クラス以外のクラス群が持つメソッドは、コンポーネントから取得した取得した参照を介して呼出し可能であっても構わない。

これにより、利用者にとってコンポーネントの内部構造が隠蔽される。

条件 3: インタフェース定義が実装から分離している

Facade 役インタフェースを準備し、同インタフェースにおいて Facade 役クラスが公開するメソッドを宣言し、Facade 役クラスが同インタフェースを実装する。

これにより、同一のインタフェース定義に対して異なる実装を準備し、実装を変更する際の影響を抑えることができる。

条件 4: 内部のクラスは外部から生成されていない

Facade 役クラス以外のコンポーネントを構成・関連するクラスは、コンポーネントを構成するクラス群以外から生成されていない。

これにより、コンポーネントを構成するクラスが、コンポーネントの構成要素という意味とは別に、クラスライブラリ (の一部) としての意味を持ってしまうことを防ぐ。

3 クラス関連グラフ

既存のオブジェクト指向プログラムを解析することでクラス関連グラフ (以降 CRG: Class Relation Graph と呼ぶ) を作成し、同グラフ上で領域到達可能を自動的に判定することでコンポーネント化すべき領域を特定する。ただし、CRG を作成するにあたって、基本クラス群 (java.*, javax.* など) が使用されていても CRG を構成する頂点とはしない。

定義 2 (クラス関連グラフ)

次の条件を満たす Java 言語に従ったクラス群を表現した有向辺を持つ多重グラフ $\Gamma = (V, \Lambda, E)$ をクラス関連グラフと呼ぶ。

$$1. V = VC \cup VI$$

Γ の頂点の集合 V は、クラスに対応したクラス頂点の集合 VC と、インタフェースに対応したインタフェース頂点の集合 VI からなる。

$$2. E = EE \cup EI \cup EU$$

Γ の有向辺の集合 E は、継承辺の集合 EE 、生成辺の集合 EI 、利用辺の集合 EU からなる。

$$3. EE \subseteq VC \times V \cup VI \times VI$$

継承辺の集合 EE は、クラスが他のクラス・インタフェースを継承している、またはインタフェース

が他のインタフェースを継承していることを表す。

$$4. EI = EID \cup EIP$$

$$5. EID \subseteq V \times VC \times \Lambda, EIP \subseteq V \times VC \times \Lambda$$

生成辺の集合 EI は、引数を伴わずにインスタンス生成 (new) を行うことを解析対象中で唯一なラベルを持って表す無引数生成辺の集合 EID と、引数を伴ってインスタンス生成を行うことをラベルを持って表す有引数生成辺の集合 EIP からなる。

$$7. EU \subseteq V \times V \times \Lambda$$

利用辺の集合 EU は、クラス・インタフェースの内部で、他のクラス・インタフェースを参照していることを表す。

Λ は解析対象のソースコード名と開始トークン位置を組み合わせた名前の集合である。

以上の定義に基づいた CRG を図示する場合は、頂点を長方形、継承辺を矢印 \rightarrow 、無引数生成辺を矢印 \xrightarrow{ind} 、有引数生成辺を矢印 \xrightarrow{inp} 、利用辺を矢印 \xrightarrow{use} でそれぞれ表わす。下記のクラス・インタフェース群の CRG を図 1 に示す。

```
public class AAA {
  public AAA(){ } public BBB b1 = new BBB(); }
public class BBB extends CCC {
  public BBB(){ } public BBB(Object o){ }
  public void foo(CCC c){ } }
public class CCC implements FFF {
  public CCC(){ } public BBB b2;
  public void baz(){ b2=new BBB("test"); } }
public class DDD extends AAA {
  public DDD(String s){ }
  public BBB bar(){ return new BBB(); } }
public class EEE {
  public EEE(){ new AAA().toString(); } }
public interface FFF {
  public void baz(); }
```

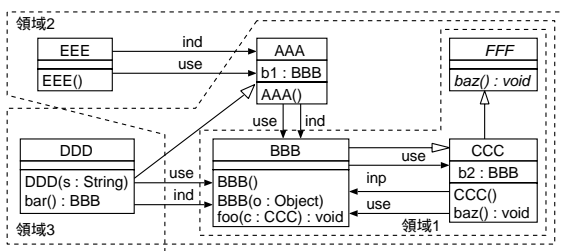


図 1: CRG の図示例

定義 3 (依存到達可能)

任意の $CRG \Gamma = (V, \Lambda, E)$ において、任意の頂点 u から v へ 1 つ以上の生成辺・利用辺・継承辺を経由して到達できるか、または $u = v$ の時、 $u \xrightarrow{*} v$ と表

わし、 u から v へ依存到達可能であると呼ぶ。同様に、任意の頂点 u から v へ 1 つ以上の継承辺を経由して到達できるか、または $u = v$ の時、 u から v へ継承到達可能であると呼ぶ。

定義 4 (領域化可能)

頂点 u から依存到達可能な頂点集合 V' を部分集合とする頂点集合 V_f があり、 V_f 中のあるクラス頂点 v_f に対して V_f に含まれない頂点からの無引数生成辺が存在し、 V_f に含まれない頂点からの生成辺がつながる先として v_f 以外の頂点が V_f に含まれない場合に、 V_f は領域基準 v_f について領域化可能であると呼ぶ。頂点 u を元に、領域化可能な頂点集合 V_f と領域基準 v_f を得る手続き $GR(u, \Gamma)$ を以下に示す。

//入力: 任意の頂点 $u \in V_\Gamma, CRG \Gamma$

//出力: (領域基準 v_f , 頂点集合 V_f)

$GR(u, \Gamma)$

return $ER(\{v \mid (u \xrightarrow{*} v) \text{ in } \Gamma\})$

//入力: 任意の頂点集合 $V \subseteq V_\Gamma$

//出力: (領域基準 v_f , 頂点集合 V_f)

$ER(V)$

$v_f := \text{null}; VO := \Phi;$

for_each $v \in V$ do

$VT := \{v' \mid (v', v, l) \in EI_\Gamma \wedge v' \notin V \wedge l \in \Lambda_\Gamma\};$

if $VT \neq \Phi$ then

if $VO \neq \Phi$ then

$VO := VT;$

if $v \in VC_\Gamma$ then $v_f := v;$

else $v_f := \text{null};$ end_if

else $v_f := \text{null}; VO := VO \cup VT;$

end_if end_if end_for

if $v_f \neq \text{null}$ then

if $\exists v' \in V_\Gamma - V, l \in \Lambda_\Gamma [(v', v_f, l) \in EIP_\Gamma]$ then

for_each $v \in VO$ do

$V := V \cup \{v' \mid (v \xrightarrow{*} v') \text{ in } \Gamma\};$

end_for

return $ER(V);$

end_if

return $(v_f, V);$

else if $VO \neq \Phi$ then

for_each $v \in VO$ do

$V := V \cup \{v' \mid (v \xrightarrow{*} v') \text{ in } \Gamma\};$

end_for

return $ER(V);$

end_if

return $(\text{null}, \Phi);$ //領域が得られなかった

4 コンポーネント 抽出手順

既存ソフトウェアからコンポーネントを抽出し、同時にコンポーネントを利用するソフトウェアを得るための手順を以下に示す。

手順 1: クラス関連グラフの作成

対象とするソフトウェアの *CRG* を作成する。

手順 2: コンポーネント領域の特定

コンポーネントとするクラス・インタフェース群をコンポーネント領域と呼ぶ。コンポーネントを抽出したい利用者は、抽出基点となるクラス・インタフェースを *CRG* から 1 つ指定する。このとき利用者は、対象ソフトウェアの詳細を知る必要はなく、クラス名などの限られた情報から判断できる範囲内で基点の選択を行う。指定されたクラス・インタフェースをもとに領域化可能なクラス・インタフェース群と、領域基準となるクラス 1 つ (以降、領域基準クラスと呼ぶ) を得る。

手順 3: インタフェースの作成・実装

Facade 役インタフェースを新たに 1 つ作成し、その名前を “I <領域基準クラス名>” とし、同インタフェースを領域基準クラスに実装する。

同インタフェースにおいて、*CRG* 上で領域基準クラスから継承到達可能な全てのクラス・インタフェースの集合 V_{EE} が持つ全ての公開メソッドを宣言する。

V_{EE} 中のクラス・インタフェースが公開属性を持つ場合は、その属性に対応した取得メソッド (public 属性の型 get <属性名> ();) と設定メソッド (public void set <属性名> (属性の型 属性名);) を、Facade 役インタフェースに宣言し、領域基準クラスにおいて両メソッドの実装を行う。ただし、両メソッド名中の属性名は先頭の一文字を大文字に置き換える。

さらに、上述の公開メソッド・設定/取得メソッドの宣言・実装にあたり、メソッドの引数・戻り値・例外の型が領域基準クラスである場合は、その型を Facade 役インタフェースに変更する。

領域基準クラスが *Serializable* インタフェースを実装していない場合はその実装を行う。

手順 4: パッケージ化

コンポーネントとして配布する場合を想定して、領域中の全クラス・インタフェースと作成した Facade 役インタフェースを一つの Jar パッケージにまとめる。

手順 5: 外部の利用部分の変更

CRG 上で領域基準クラスをつながる先とする利用辺のつながる元の領域に含まれないクラス・インタフェースについて、領域基準クラスへの参照の型を

全て Facade 役インタフェースへ変更する。また、その参照を介して集合 V_{EE} 中の公開属性の値を参照または代入している場合は、手順 3 で領域基準クラスへ追加した取得・設定メソッドの呼出しへ変更する。ただし、*CRG* 上で生成辺のつながる元のクラス・インタフェース中におけるインスタンス生成には領域基準クラスの型を用いる。

以上の手順によって、他のクラスに依存せずに独立して生成可能なコンポーネントと、コンポーネントを実際に生成・利用するプログラム例が得られる。コンポーネント抽出の例として、図 1 の *CRG* について、構成する全てのクラス・インタフェースを基点として領域化を行った結果、クラス BBB を領域基準とする領域 1・領域 2 と、クラス AAA を領域基準とする領域 3 が得られた。領域 1 をコンポーネントとして抽出した例を図 2 に示す。インタフェース IBBB を作成し BBB に対して IBBB の実装が行われ、IBBB, BBB, CCC, FFF を 1 つのコンポーネントとしてまとめて、BBB を利用している外部の AAA と DDD に対して修正が行われる。得られたコンポーネントは外部から IBBB を経由して操作される。

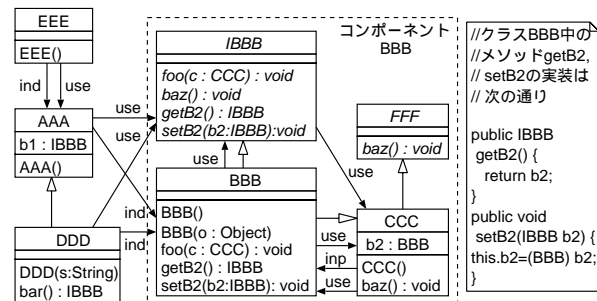


図 2: コンポーネント抽出後の *CRG*

5 実装・評価

Java 言語のソースコードを解析して *CRG* を作成し、指定されたクラス・インタフェースを元に領域を判定し、Facade 役インタフェースの生成と、元のソースコードの修正、およびコンポーネントのパッケージ化を行うツールを実装した。実装には Java2 SDK 1.4.0 および JavaCC を用いた。実行例を図 3 に示す。クラス・インタフェース頂点は UML におけるクラス図と同様に属性とメソッドを表示する。また、有向辺は UML におけるステレオタイプ (<<use>>, <<instantiate>>) に対応させる形で出力する。

CRG 解析ツールを用いて、内部がコンポーネント化されていない Java ソフトウェアに対してコンポーネント化を試みた。解析対象サンプルは、BDK1.1

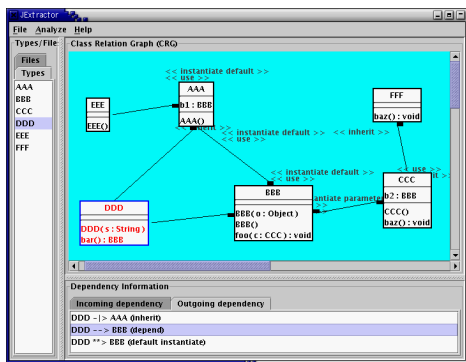


図 3: CRG 解析ツールの実行例

付属の BeanBox (総クラス数:67, 総インタフェース数:2), Java2SDK1.4.0 付属の Metalworks (19,0), 単体テストフレームワーク JUnit 3.7 (84,10) である。全サンプルについて、構成する全クラス・インタフェースを基点として領域化を行った。得られた全コンポーネントのうちで、コンポーネントを構成する元のクラス・インタフェース数が 2 つ以上のものの元の構成クラス・インタフェース数を表 2 に示す。

表 2 より, JUnit についてはテストケースやビューといった機能単位で複数のコンポーネントを抽出できた。各コンポーネントを構成するクラス・インタフェース数は全体からみて大きくないため、得られたコンポーネント群は JUnit とは関係なく異なる状況において再利用できると考える。

Metalworks および BeanBox については、得られたコンポーネント数が少なく、各コンポーネントを構成するクラス・インタフェース数が大きい。本手法は、Facade 役インタフェースの実装と利用部分の修正以外のクラス内部のメソッドや属性の移動などのリファクタリング操作を行わないため、得られるコンポーネントの数・大きさが解析対象とする元のプログラムの設計品質に左右される結果となった。

表 2: サンプルからの抽出結果

領域基準クラス	クラス	インタフェース
(BeanBox)	(67)	(2)
ErrorDialog	43	2
(Metalworks)	(19)	(0)
MetalworksFrame	16	0
MetalworksHelp	2	0
(JUnit)	(84)	(10)
WasRun	2	0
TestCase	5	3
TestCaseClassLoader	4	1
StandardTestSuiteLoader	1	1
TestCaseClassLoader	3	1
TestSuitePanel	2	0
FailureRunView	1	2
TestHierarchyRunView	3	2

6 関連研究

プログラムスライシングによって既存のプログラムから実行可能な部分系列を抽出する試みがある [4]。この試みは、主に抽出対象プログラムに対して利用者が詳しいという前提をおくが、クラスよりも細かくて冗長な部分を排除した部品を得ることができる。

文献 [2] では、各クラスが持つ役割の判断によってコンポーネントの候補を識別し、リファクタリング操作を行う手法を提案している。コンポーネントの候補とインタフェースの決定を行うにあたって、作業従事者による主観的な判断によるところがある。

文献 [5] では、クラス群の主に集約関係に着目して静的なクラス階層を再構築する手法を提案している。その再構築における抽象頂点の追加操作は、我々の手法における Facade 役インタフェースの追加操作に類似するが、我々の手法ではその操作の許される条件が生成関係に基づいている。

文献 [6] では、既存のオブジェクト指向プログラムを手作業でコンポーネント群に置き換える手順とコストについて報告している。我々の手法はその変換作業にかかるコストの低減に役立つと考える。

7 おわりに

明確な定義のもとでコンポーネントとすべき領域を自動的に特定し、特定された領域を使用している周辺部分を自動的に修正する手法を提案した。本手法は、ソースコードの静的な解析に基づくものであり、使用されるクラスが実行時に決定されるようなプログラムについて扱うことができない。今後、このような問題について考えていく予定である。

参考文献

- [1] J.Hopkins: Component Primer, Communications of the ACM, Vol. 43, No. 10 (2000)
- [2] 落合竜一, 鈴木正人: 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, 情処研報, SE136-12 (2002)
- [3] G.Hamilton: JavaBeans 1.01 Specification, Sun Microsystems (1997)
- [4] L.Larsen and M.Harrold: Slicing Object-Oriented Software, 18th International Conference on Software Engineering (1996)
- [5] 黄錫炯 他: クラス階層モデルにおける再構成法について, コンピュータソフトウェア, Vol. 15, No. 4 (1997)
- [6] M.Goulao and F.Abreu: From Objects to Components: a Quantitative Approach, 6th ECOOP Workshop on QAOOSE (2002)