Vol.2012-SE-177 No.7
2012/7/20

フレームワークアプリケーションの副作用の特徴付け手法

久米 出[1,a)]　中村 匡秀[2,b)]　柴山 悦哉[3,c)]

概要：
近年のオブジェクト指向開発において，アプリケーションフレームワークはアプリケーションの開発効率と信頼性を大きく向上させる手段として，普及が進んでいる．しかしながら，フレームワークのメソッドを良く理解せずに，誤った形で呼び出す (或いは呼び出さない) アプリケーション固有のコード、すなわち**逸脱コード**の問題が注目され始めている．
本論文ではフレームワーク内の逸脱コードによって引き起こされる副作用を特徴付けする手法を提案する。我々はこの特徴付けによって副作用の複雑な実行過程をメソッド呼び出しとオブジェクトの参照の視点から概観する事を目指している。我々は第三者が開発した実用的なフレームワーク上のアプリケーションに対して我々の手法を適用し、その実用面からの有用性を議論する。

Characterizing Side Effects in Framework Applications

Kume Izuru[1,a)]　Nakaura Masahide[2,b)]　Shibamaya Etsuya[3,c)]

Abstract:
Application frameworks are widely used in order to increase efficiency and reliability in object-oriented software development. Recently the problem of *deviant code*, application code which invokes framework code incorrectly, becomes serious in practice. Deviant code triggers a complex error producing process and is hard to resolve.
In this paper, we propose a way to characterize side effects triggered by deviant code in framework applications. Our characterization is intended to provides an overview of a complex side effect process in terms of method invocations and object references. We discuss the practical usefulness of our characterization by applying it to a case of side effect found in an application on a practical framework developed by a third party.

1. はじめに

アプリケーションフレームワークはある特定の応用領域のアプリケーション群に共通する概念や処理の骨組を提供する再利用可能なソフトウェアである [1], [2]。近年ではアプリケーションフレームワークは Web アプリケーション開発 [3], [4] や統合開発環境の基盤 [5]、或いはソフトウェア工学ツールの開発 [6], [7] 等、様々な分野で利用されるようになった。

フレームワークには “アプリケーション固有な特徴に順応出来るようになっている” **ホットスポット** [8] が含まれている。フレームワーク内のホットスポットを表現するクラスはアプリケーション開発者に公開される。アプリケーション開発者はこれらのクラスを継承してホットスポットを実装する。フレームワーク内部で複雑なオブジェクト同士のインタラクションが発生する中でホットスポットを実装するクラスのメソッドが呼び出される。このようにフレームワーク側からアプリケーションコードを呼び出す**制御の逆転** *(inversion of control)* と呼ばれる実行形態がクラスライブラリとの最大の違いである [9]。この

1 奈良先端大学院大学
NAIST, Ikoma, Nara 630–0101, Japan
2 神戸大学
Kobe University, Kobe, Hyogo, 657–0013, Japan
3 東京大学
The University of Tokyo, Bunkyo, Tokyo, 113–0033, Japan
a) kume@is.naist.jp
b) masa-n@cs.kobe-u.ac.jp
c) etsuya@ecc.u-tokyo.ac.jp

© 2012 Information Processing Society of Japan

Vol.2012-SE-177 No.7
2012/7/20

フレームワークアプリケーション独特な設計は Pree による Metapattern[8] によって形式化の枠組みが与えられている。

アプリケーションフレームワークの普及に伴いアプリケーションの固有部分に含まれる**逸脱コード** *(deviant code)* が開発現場に於ける深刻な問題として取り上げられるようになってきた。逸脱コードとはアプリケーション開発者によるフレームワークの誤利用を実装するものであり、しばしばシステムの状態を誤った形で設定する。このような不具合を解決するためには原因となる逸脱コードを特定するだけでなくその実行過程の理解、すなわちプログラム理解も必要である [10]。

ある調査 [10] に依ればフレームワーク内部状態を変更メソッド群がアプリケーション側から誤った形で呼び出される様式の逸脱コード例が多数発見されている。一般にこの種の逸脱コードが実行されるとフレームワークの内部状態が不完全な形で更新されたり、副作用が引き起こされる。複雑なフレームワークの内部状態処理の理解という困難な作業の存在が、こうした不具合を解決する上での大きな障害となっている。

本論文ではフレームワーク内部の副作用を引き起こす逸脱コードに焦点を当てる。このような逸脱コードによって引き起こされる副作用を我々は逸脱効果と呼ぶ事にする。我々は逸脱効果の実行過程の理解作業を支援するため特徴付けを提案する。我々の提案手法は二つの特徴を有する。第一にフレームワーク側とアプリケーション固有部分側の間でのメソッド呼び出し構造に注目している。次に副作用を発生させるオブジェクトの参照構造をこれらメソッド呼び出しと関連付けている。上記の二点によって我々は逸脱効果の発生の仕組みを概観し、プログラムコードを修正する手掛かりを得られると期待している。

我々は実在するフレームワークアプリケーション上の逸脱効果に対して我々の特徴付けを適用し、その有効性を議論する。このフレームワークは第三者によって開発され実用的なアプリケーションの開発に利用されている。我々は特徴付けの結果が逸脱効果発生の仕組みの特定とコードの修正の方針決定に寄与する事を示す。

本論文の以降の構成を以下に述べる。第 2 節で我々の研究の動機となった事例と以降の議論に必要な諸概念を説明する。第 3 節で我々の提案する特徴付けを説明し、第 4 節で我々の事例に適用した結果について議論する。第 5 節で関連研究について、第 6 で結論を述べる。

2. 準備

2.1 不具合事例

本論文ではオープンソースなオブジェクト指向アプリケーションフレームワーク GEF(Graph Editing Framework) [7] 上に構築されたアプリケーション GEFDemo [11] の不具合事例を取り扱う。GEF はグラフ形式のデータを編集するエディタのフレームワークであり、実用的な CASE ツール [6] の開発にも利用されている。GEFDemo は GEF のデモンストレーション用のアプリケーションであり、簡単な UML 図式編集機能を実装している。

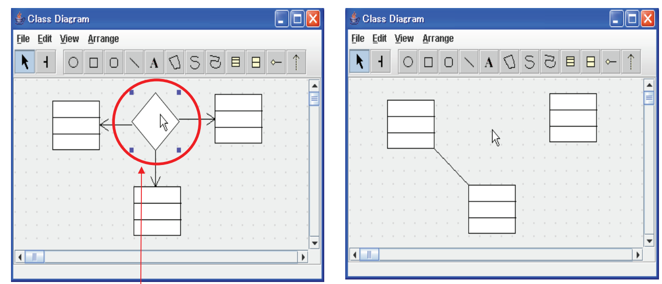

図 1 UML クラス図式の操作に伴う不具合事例

GEFDemo は UML クラスと UML 関連を表現する図形クラスを実装している。UML 関連は二項関連 (binary association) と n-項関連 (n-ary association) がそれぞれ異なる図形で表示される。一般に n-項関連は菱形の節点 (node) とそれから延びる n 本の辺 (edge) の形で表示される。図 1 には三つの UML クラスとそれらを結ぶ 3-項関連が表示されている。

この 3 項関連を UML 図式から削除するためにその節点を選択して Delete キーを押下すると、予想に反して図 1 のような誤った結果が表示されてしまう。図の右上に示されるように元の関連は二項関連に置き換えられ、図の下部に表示されるようなエラーが発生する。

実際のところ、このエラーメッセージが暗示するようにこのフレームワーク (GEF) の内部で副作用が発生している。副作用発生の引き金を引いたのはアプリケーション側のコード内で実行されたあるメソッド呼び出しである。このメッドはフレームワークで実装され、フレームワークの内部状態を変更出来るようにアプリケーション側に開示されたものである。一般にフレームワークの内部状態を変更する処理は複雑であり、それを理解する作業は多大な労力が必要とされる事が多い。この種の不具合を訂正するためにはフレームワーク内部の複雑な副作用を理解する作業を支援する手法が必要である。

Eclipse の GEF フレームワークとは別物である。

© 2012 Information Processing Society of Japan

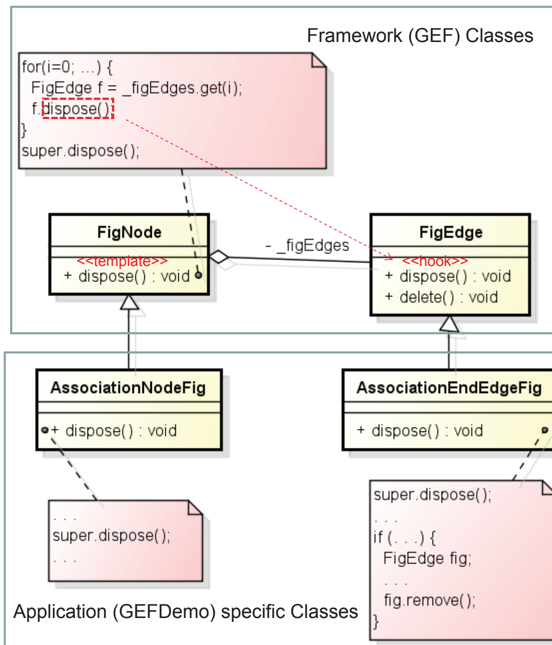

図 2 メタパターン記述例

2.2 アプリケーションフレームワーク

アプリケーションフレームワークはある特定の領域の“主要な要素”を設計実装した再利用可能なクラスの集合である [1]。フレームワークの設計には明示的或いは暗黙的に**ホットスポット**として指定される箇所が存在する。ホットスポットはアプリケーション固有の特徴を実装するために継承による拡張が出来るようになっている箇所である [8]。本論文ではフレームワークアプリケーションのうちでフレームワークに含まれているクラスを**フレームワーククラス**、アプリケーション固有な機能を実現するクラスを**アプリケーション固有クラス**と呼ぶ事にする。またフレームワーククラスで定義されているメソッドを**フレームワークメソッド**、アプリケーション固有クラスで定義されているメソッドを l **アプリケーションメソッド**と呼ぶ事にする。

フレームワークは通常複雑なオブジェクト間協調を実装している。ホットスポットを実装するアプリケーションメソッドはこうした協調の文脈で呼び出される。**制御の反転** [9] と呼ばれるこの実行形式によってアプリケーションフレームワークはクラスライブラリと区別されている。Pree はフレームワークの設計内のホットスポットをメソッド呼び出しと上書きによって特徴付けするメタパターン [8] を提案した。メタパターンはホットスポットを表現するフレームワークメソッドを `hook` メソッドとして指定し、`hook` メソッドを呼び出すメソッドを `template` メソッドとして指定する。通常 template メソッドから hook メソッドとして呼び出されるのはアプリケーションメソッドである。よってメタパターンは制御の反転として実行される可能性のある箇所を示すものととらえられる。

GEFDemo のクラス図とメタパターンの記述を図 2 に

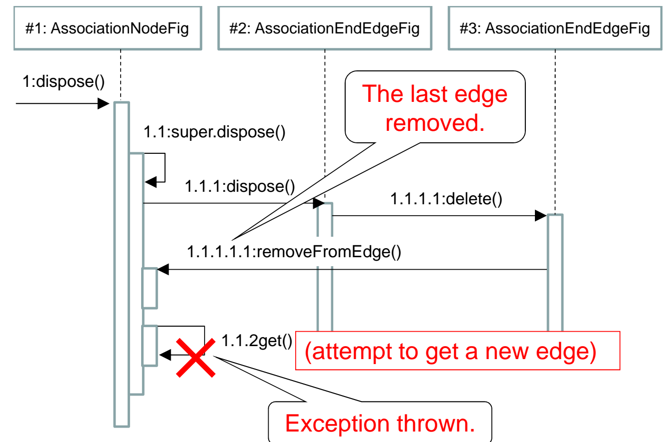

図 3 廃棄処理の順序図

示す。この図中の `FigNode` と `FigEdge` はそれぞれ一般的なグラフの節点と辺を表現するフレームワーククラスである。どちらのクラスもそれぞれのインスタンスの廃棄に伴う内部処理をメソッド `dispose()` で実装している。クラス `FigEdge()` のメソッド `delete()` は辺を両端の節点から外す処理を実装している。図 2 内の疑似コード `FigNode.dispose()` 中では辺を廃棄するメソッド `FigEdge.dispose()` の中からこのメソッドが呼び出されている。

一般にこれらのフレームワーククラスを継承するアプリケーション固有クラスは独自の廃棄処理を実行するためにメソッド `dispose()` を上書きする必要がある。よって上の疑似コードの実行の制御の反転の実行場所であり、`FigNode.dispose()` はテンプレートメソッド、`FigEdge.dispose()` はフックメソッドとして指定されるべきである。この指定を明示化するためにメタパターンのステレオタイプ ≪**template**≫ と ≪**hook**≫ がこのクラス図中のそれぞれのメソッドに付されている。

GEFDemo ではその固有クラスである `AssociationNodeFig` と `AssociationEndEdgeFig` がそれぞれフレームワーククラス `FigNode` と `FigEdge` を拡張 (継承) し、n 項関連の節点と辺をそれぞれ実装している。またこれらのアプリケーション固有クラスは n 項関連固有の廃棄処理を実装するためにメソッド `dispose()` を上書きしている。

ここで図 2 中の他の疑似コードに注意してみよう。`AssociationNodeFig.dispose()` の疑似コード中で `super.dispose()` が呼び出される事によって `FigNode.dispose()` が実行され `AssociationEndEdgeFig.dispose()` からも同様に `super.dispose()` によって `FigEdge.dispose()` が呼び出されている。n 項関係の節点と辺に対してもフレームワーク内部の廃棄処理を実行する必要があるため、このようなフレームワークメ

© 2012 Information Processing Society of Japan

ソッドの呼び出しは必須である。結果として制御の反転から逆の向き、則ちアプリケーションメソッドからフレームワークメソッドの呼び出しが発生している。我々はこの向きのメソッド呼び出しを**制御の再反転**と呼ぶ事にする。制御の反転と再反転が組み合わせられる事によりフレームワークとアプリケーション固有部分の間に複雑なメソッド呼び出し関係が構築される事になる。

我々の事例のメソッドの呼び出し鎖の中では以下のようにでは制御の反転と再反転が発生している:

```
AssociationNodeFig.dispose();  // 制御の反転
 FigNode.dispose(); // 制御の再反転
  AssociationEndEdgeFig.dispose(); //制御の反転
   FigEdge.dispose();  // 制御の再反転
```

フレームワークアプリケーション内で制御の再反転が発生する事は珍しくない。例えば Swing はアプリケーション側から呼び出される事を前提としたメソッドが API として開示されている。フレームワークの利用に関する不具合報告や質問を調査した結果、アプリケーション開発者がしばしば実装する制御の再反転が逸脱コード化し易いことが示されている [10]。

2.3 逸脱効果

図 1 に示す例外は、節点の廃棄をきっかけとしたフレームワーク内部の複雑な内部処理の副作用として発生したものである。図 3 に示す順序図を用いてこの副作用の発生過程の解説を試みてみよう。この図中のオブジェクト`#1`は件の n 項関連節点を表現し、`#2`と`#3`はこの節点と UML Class を結ぶ辺を表現する。

図 1 の実行によって菱形の図形 (`AssociationNodeFig`インスタンス`#1`) の`dispose()`メソッドが呼び出される (図 3 の`1:dispose()`)。この時制御の再反転によって自身に対して`FigNode.dispose()`が呼び出される (`1.1: super.dispose()`)。そこから`AssociationEndEdgeFig`のインスタンス`#2`に対して上書きした`dispose()`が呼び出される (`1.1.1: dispose()`)。これによって別の`AssociationEndEdgeFig`インスタンス`#3`に対する`FigEdge.delete()`が呼び出される (`1.1.1.1: delete()`)。この中で`#3`は自身を`#1`から分離する (`1.1.1.1.1: removeFromEdge()`)。この段階でインスタンス`#1`から全ての辺が外される。にもかかわらず`FigNode.dispose()`の中で "次の辺" を取得しようとする (`1.2: get()`) ために例外が発生するのである。

この順序図には直接示されていないのだが、辺`#2`から別の辺`#3`へのメソッド`delete()`の呼び出し (`1.1.1.1: deletion()`) で制御の再反転が発生している。このように制御の反転下での制御の再反転が引き起こすフレームワーク内部の副作用を我々は**逸脱作用** *(deviant effect)* と呼ぶ事にする。**逸脱作用**を引き起こすのはフレームワークの利用方法を良く理解せずに書かれたアプリケーション固有コードであり、**逸脱コード** *(deviant code)*[10] の一種と見做される。

一般に逸脱コードの問題を修正するためには逸脱挙動に関するプログラム理解が不可欠である [10] が、我々の知る限りでは逸脱挙動の理解を支援する情報を表現する手法は存在しない。上に述べた逸脱効果事例を順序図で説明しようとしても、それの引き金となる制御の反転や再反転がどこで発生しているのかも明確には示されない。オブジェクト`#2`に対するメソッド`delete()`の呼び出しが制御の再反転を引き起こしている事が一目で明らかである事が望ましい。

さらに上の順序図からは辺`#2`が何故別の辺`#3`を削除しているのかも明確ではない。`#2`にとって`#3`はどう位置付けられているのかが分かれば削除の理由も推測出来るかもしれない。そのためには`#2`から`#3`がどう関連付けられているのかを知る必要があるが、そのような情報は示されていない。オブジェクト`#2`が他のオブジェクトのインスタンス変数を経由して`#3`に至るまでの参照関係が明らかになれば、こうした疑問を解く手掛かりが得られると我々は期待している。このようにオブジェクトのインスタンス変数を介した参照を順番に辿る経路を ***R*-経路** *(R-path)* と呼ぶ事にする。

逸脱効果の発生過程は制御の反転と再反転、および R-経路によって特徴付けされるべきである。逸脱効果に対処する保守作業者がその本質を直感的に理解し、かつその解決に向けた計画を立てられるような新しい特徴付け手法が必要とされている。

3. 提案手法

本節では我々の提案する特徴付けを説明する。また第 2 節で紹介した事例の特徴付けを行う。特徴付けを実現する方法に関しては本論文の範囲外とする。

3.1 基本概念

我々は最初に特徴付けの対象を明確にする。我々はまず一般のオブジェクト指向プログラムに於ける副作用の発生様式をオブジェクト間の参照構造とメソッド呼び出しの構造によって定義する。この副作用の様式を我々は我々は**誤結合** *(Erroneous Coupling)* と呼ぶ。

次に我々はフレームワークアプリケーションで発生した誤結合を逸脱効果として特徴付けするための図式を導入する。また図式上の要素の役割を示すためのステレオタイプも導入する。これらのステレオタイプは二つの面から逸脱効果に関するプログラム理解を支援する事を目的としてい

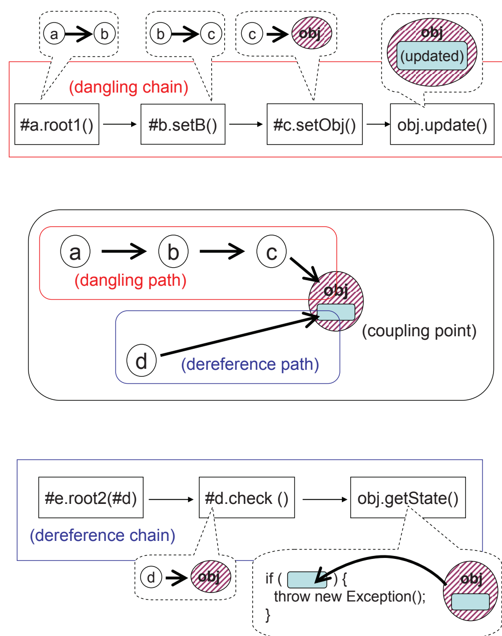

図 4　誤結合の概念

る。まず逸脱効果を誤結合としてその構造も含めて明確に特定する。次に制御の反転と再反転が一連の実行過程のどこで発生しているのかを明示する。

3.2　メソッド呼び出し鎖の誤結合

二つのメソッド呼び出し鎖同士がそれぞれの R-経路を介してある共通のオブジェクトの状態の設定と読み込みを行い、その状態の読み込みが原因で誤った結果が引き起こされる状況を我々は**誤結合**と呼ぶ事にする。誤りの遠因となる状態の設定を行うメソッド呼び出し鎖は**宙吊り鎖** *(dangling chain)*、設定された状態値によって誤りが表面化した呼び出し鎖を**参照鎖** *(dereference chain* と呼ぶ事にする。また状態の設定と読み込みが行われるオブジェクトを**結合点** *(coupling point)* と呼ぶ。

誤結合の一例を図 4 に示す。ここではオブジェクト Obj に対して root() から setB()、setObj()、update() に至る呼び出し鎖が宙吊り鎖の役割を果たし、root2()、check()) から getState() で終わる呼び出し鎖が参照鎖の役割を遂行している。オブジェクト Obj が結合点の役割を果たす

オブジェクトの状態は指定された *Dangling* 鎖の底辺で直接更新されるか、或いはその下部で実行されるメソッドの中で更新される。この状態にデータ依存する値が宙吊り鎖の底辺メソッドで誤った結果を表面化させる。図 4 では宙吊り鎖の底辺メソッド update() が直接オブジェクト Obj にアクセスし、Obj の状態値が直接参照鎖の getState() の例外発生の引き金を引いている。

宙吊り経路と参照経路は宙吊り鎖と参照鎖、そして結合点から導かれる。図 4 に示すように、呼び出し鎖中の各メソッドによって経路は段階的に辿られる。この例では宙吊り鎖の実行によって、オブジェクト a、b、c を経由して結合点 Obj が取得される。参照鎖ではオブジェクト d から Obj が取得される。

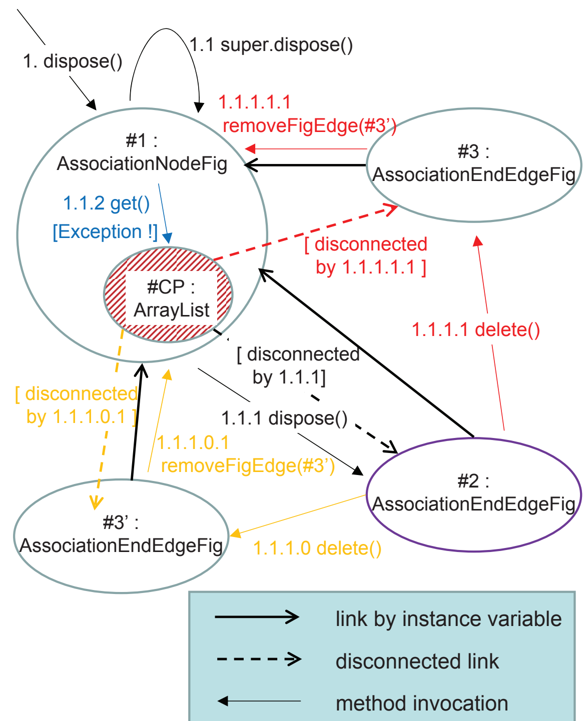

図 5　逸脱効果事例の実行過程

それぞれの呼び出し鎖に於ける R-経路をそれぞれ**宙吊り経路**と、**参照経路**と呼ぶ。誤った結果を直接引き起こす値を計算する過程で、宙吊り鎖によって設定された結合点が異なる R-経路経由で取得される可能性がある。よって一般にある誤結合の宙吊り経路は一つしか無いが、参照経路は複数存在し得る事に注意が必要である。

3.3　不具合事例の説明

ここで我々が第 2 節で紹介した逸脱効果事例の詳細を図 5 を用いて説明する。この図では図 1 の n 項関連の節点の削除を契機としたオブジェクト間のやりとりと、その中でオブジェクト同士の参照関係がどのように外されるのかが示されている。図中の細い矢印はメソッド呼び出しを、太い矢印はインスタンス変数を介したオブジェクトの参照を表している。

この図には図 3 に登場するオブジェクト#1、#2、#3 に加えて AssociationEndEdgeFig インスタンス#2' も示されている。このオブジェトは n 項関連の節点から延びる第三のの辺である。結合点 CP は n 項関連節点#1 の内部に隠蔽された ArrayList インスタンスであり、この節点から延びる三本の辺を参照している。図 5 を見易くするために、このリストオブジェクトは関連ノードの一部として表示している。

節点に対してメソッド dispose() が呼び出されると

© 2012 Information Processing Society of Japan

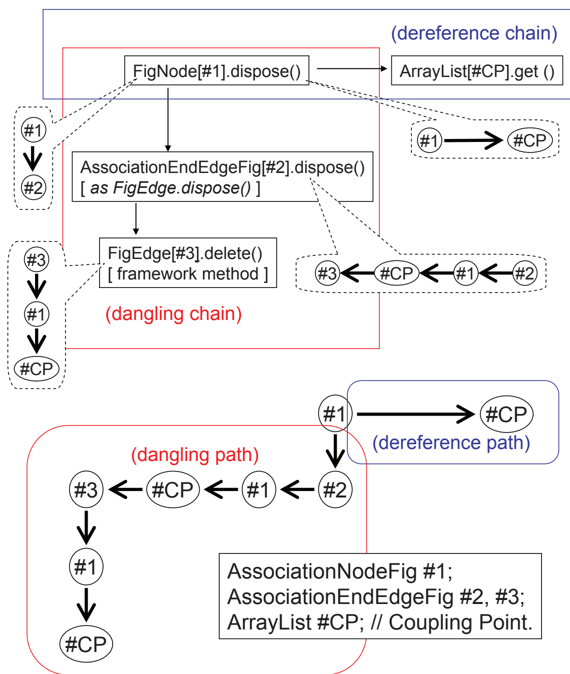

図 6 フレームワークアプリケーション内での誤結合

([1. dispose()])、その中からある辺 (#2) に対してメソッド dispose() が呼び出される ([1.1.1 dispose()])。このメソッドが返ると次には別の辺を取得しようとすると ([1.1.2 get()])、既に辺が全て削除されているために例外が発生する。辺の全削除は FigNode.dispose() の実装者にとって予期せぬ状況である事が窺われる。

第 2.1 節で紹介した事例を誤結合として、則ちメソッド呼び出しと R-経路の視点から捉えた結果を図 6 に示す。ここでアプリケーションメソッド AssociationEndEdgeFig.dispose() に注目しよう。結合点#CP の状態はこのメソッドから呼び出されるメソッド FigEdge.delete() によって変更されているからである。

このアプリケーションメソッドはフックメソッドとしてテンプレートメソッド FigNode.dispose() から呼び出されている。宙吊り経路を調べる事によって、このテンプレートメソッドが呼び出されるのは節点#1 から延びる辺の一つである事が分かる。よってこのフックメソッドのコードを調べる事によって節点に対して予期せぬ辺の全削除が実行された理由を特定出来ると期待される。我々がプログラムコードを調査した結果得た事柄を以下に述べる。

この予期せぬ辺の全削除は以下の手順を経て遂行される。まずこのアプリケーションメソッドの受け手である最初の辺 (#2) が自身を節点から切り離した後に残りの辺の数を勘定する。もしこの時残りの辺の数が二つであれば (この実行事例ではその通りなのであるが)、残りの二辺はこの節点から切り離され、新しく二項関連が作成される。

このような処理を実行するのは三項関連から一本の辺を削除した結果、二項関連になる場合に対処するためである。二項関連は n 項関連とは異なる形状であるためにこうした処理が必要となる。また実際のところ、n 項関連から辺のみを削除するような操作は問題無く実行される。メソッド AssociationNodeFig.dispose() は確かに直接逸脱効果の引き金を引いている。しかしながらそれが問題となるのは、辺を残す節点に対して FigNode.dispose() から呼び出された場合のみであり、このアプリケーションメソッド自体は問題無いと判断される。

最終的に我々は以下の方針でこの不具合を修正した。まず、我々は上記の理由から AssociationEndEdgeFig.dispose() の実装を修正しない事にした。次に我々は AssociationNodeFig.dispose() 内で全ての辺が残っている状態で super.dispose() を呼び出している箇所を逸脱コードとして認定した。メソッド super.dispose() を呼び出す前に全ての辺に対してメソッド delete() を呼び出すように実装を変更した。これによって super.dispose() の中から辺に対してメソッド dispose() が呼び出される事が無くなった。この時点で全ての辺が削除されているからである。

3.4 逸脱効果の特徴付け

我々が第 3.3 節で示したように、制御の反転と再反転の発生に注目するとアプリケーション固有の挙動が実行される文脈とその効果を容易に把握出来る。この箇所からコードの調査を開始する事によって、逸脱効果が発生したえ時にその問題の所在を特定し、解決の方針を立てる事が可能になると期待される。また R-経路からメソッドを呼び出す側と呼び出される側の関係も推測出来ると期待される。

この種の把握を可能にするために、我々は逸脱効果、すなわちフレーワークアプリケーションで発生した誤結合を制御の反転と再反転、及び R-経路によって特徴付けする記述を導入する。逸脱効果は実行時のメソッド呼び出し鎖と R-経路、およびそれらに付与される独自のステレオタイプから構成される図式によって特徴付けされる。

第 2.1 節で紹介した逸脱効果事例を特徴付けした結果を図 7 に示す。矩形はメソッドを表現している。メソッドの中で参照される R-Path は破線の角丸矩形 (dashed rounded rectangle) の中に表示される。

入れ子の矩形は字面上のメソッド呼び出しを表現している。字面上呼び出されているメソッドと実行時に呼び出されるメソッドは破線矢印によって結ばれる。例えばメソッド FigNode.dispose() 内では字面上では FigEdge.dispose() が呼び出されているが、実際には AssociationEndEdgeFig.dispose() が呼び出されている。

メソッド呼び出しの受け手や R-経路を構成するオブジェ

この特徴付けを実現するために我々が既に開発たバイトコードの可測化 (instrumentation) 技術 [12], [13] を利用した。

クトは#1 のような識別子を用いて参照される。識別子は#に続く番号として表現される。#CP は結合点の参照を表現する特別な識別子である。図 7 の下部では識別子を用いてオブジェクトの実行時のクラスを示している。また FigEdge[#2].dispose() はオブジェクト#2 に対してメソッド dispose() が呼び出される際に FigEdge のインスタンスとして参照されている事を示す。実際にはこのオブジェクトは AssociationEndEdgeFig のインスタンスである。

メソッドに含まれる R-経路 のうち、このメソッドが直接参照しない部分は角括弧で括られる。例えば AssociationEndEdgeFig.dispose() の中では関連辺オブジェクト#2 から関連ノード#1、ArrayList インスタンス#CP を経て#3 が取得されている。#3 以降の部分はこのメソッドから呼び出される FigEdge.delete() 以降で参照される。

このように記述する事によって各メソッド呼び出しによって参照される R-経路がトップダウンに表示されると共に、そのメソッドの中で R-経路が辿られる範囲も明確に出来る。さらにオブジェクトに識別子を付ける事によってメソッドのパラメータが他のオブジェクトからどう参照されるのかを特定出来る。

これらの要素が逸脱効果の形成に果たす役割を指定するために我々は以下に示す五つのステレオタイプを導入する。これらのステレオタイプのうち、初めの二つについては Pree のメタパターン [8] から概念を借用している。残りの三つで制御の再反転、及び誤結合に於ける宙吊り側と参照側を明示する:

≪template≫ テンンプレートメソッド呼び出しを示す。その使用に際して Pree のメタパターンには無かった制約を課す。このメソッドから呼び出されるフックメソッドはステレオタイプ ≪hook≫ の項で記述される条件を満たす必要がある。

≪hook≫ メタパターンで言うところのフックメソッドの呼び出しを示す。Pree のメタパターンでは hook メソッドは必ずしもアプリケーション固有のメソッドに限定されないが、我々の特徴付けではこのメソドに対して ≪dangling≫l が付与されているか、或いは以降に呼び出されるメソッドに対して付与されていなければならない。

≪dangling≫ 逸脱効果の引き金を引くメソッド呼び出し或いは宙吊り R-経路を明示化する。このステレオタイプはアプリケーションメソッドに付与され、宙吊り鎖の出発点と見做される。第 3.3 節で示したようにこのステレオタイプが付与されたメソッド呼び出しが逸脱コードを構成しているとは限らない事に注意が必要である。

≪re-inversion≫ 制御の再反転を表現する。このステレオタイプはアプリケーションメソッドから呼び出されるフレームワークメソッドに付与される。またこのステレオタイプが付与されるメソッドは宙吊り鎖に含まれており、かつこれ以降に制御の再反転が発生していてはならない。

≪dereference≫ 実行の誤りが表面化した箇所を示す。このステレオタイプはフレームワークメソッドに付与される。またこのメソッドによって参照鎖が開始される。

このような特徴付け自体がこの不具合が逸脱効果によるものである事を示している。図 6 に示された逸脱効果のメソッド呼び出しとオブジェクト参照に関する構造がステレオタイプを用いた図 7 の図式によって明示化される。ステレオタイプ ≪template≫、 ≪hook≫、そして ≪re-inversion≫ によって制御の反転と再反転の構造が明らかにされる。またステレオタイプ ≪dangling≫ and ≪dereference≫ によってメソッドや R-経路が "どちらの側なのか" を明らかにしてくれる。またデバッガ等を用いて R-経路を辿らずともトップダウンに概観出来るようになっている。

我々が先に示したように、この例では制御の再反転を実行しているメソッド AssociationEndEdgeFig.dispose() のコードから調査を開始するのが適当であると思われる。図 7 の特徴付けからこのメソッドでは同じ n 項関連の他の辺に対する削除処理を開始している事が既に明らかになっている。これによってコードを調査するメソッドの実行内容の一部が明らかになっているわけである。この情報によって我々がこのメソッドから呼び出される delete() メソッドの意味が用意に理解する事が可能となり、結果としてこの呼び出しが実行される理由の推測にもつながったのである。

4. 議論

第 3 節に示したように特徴付けを利用して間接的にプログラムコードの読解作業を支援する事は可能であるが、将来的にはより直接的な支援に利用していと考えている。プログラムコードの読解作業を直接支援するためには、保守作業者が現在表示しているコードと特徴付けの内容を関連させる必要がある。

このような作業を実現するためにはツールを用いた支援が必要な事は明らかである。またツールにはメソッド呼び出しとオブジェクトの R-経路に関する操作履歴を保存し、かつ副作用に伴うデータの流れを扱える動的解析機能が必要になると思われる。特徴付けを構築するためにはこれに加えて保守作業者が対話的にトレースの内容を検索し、段階的に特徴付けの図式の構築とソテレオタイプを付与する環境が必要であろう。

© 2012 Information Processing Society of Japan

Vol.2012-SE-177 No.7
2012/7/20

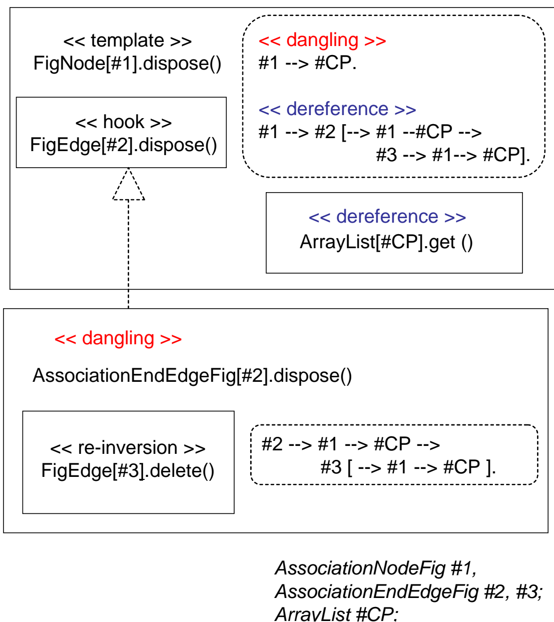

図 7 逸脱効果の特徴付け

5. 関連研究

オブジェクト指向プログラムのメソッド呼び出しと R-経路を扱う代表的な解析手法としては Lienhard 等による Object Flow Analysis [14] が挙げられる。Object Flow Analysis は一般のオブジェクト指向プログラムを対象として R-経路を含む一般的なオブジェクト同士の参照構造とその行為のグラフによる可視化を実現している。しかしながら彼等の手法によって得られる結果は、解析対象がそれほど複雑でない場合でも、グラフの節点数が三桁にのぼる非常に複雑な結果が表示されている。これは我々の特徴付けのような簡易な表現にはほど遠い。彼等の手法はより一般のプログラム実行時に於けるオブジェクトへの参照関係を対象としている事と、研究の本来の動機として必ずしも可視化を必要としない依存関係の解析 [15] も目標の含まれているためにこのような結果になっているのである。

逸脱効果の特徴付けを支援するためには動的解析機能を供えたツールの開発が必要である事は既に述べた。メソッド呼び出しと R-経路を共に扱うためにはトレースに対する検索やスライシング、流れ解析を含めた広範囲な技術 [12], [13], [16], [17], [18], [19] が必要とされると思われる。

6. おわりに

本論文ではフレームワークアプリケーション内に発生する副作用を特徴付けし、その理解を支援する手法を提案した。第三者が開発した実用的なフレームワーク上のアプリケーションで発生してた副作用を特徴付けし、その有効性を議論した。

謝辞 研究に関して御助言下さった奈良先端大学院大学の萩田紀博先生、大阪芸術大学の武村泰宏先生、甲南大学の新田直也先生、及び株式会社アーヴァイン・システムズの皆様に謹んで感謝の意を表します。本研究は文部科学省[挑戦的萌芽 (No. 23650016)、基盤 (C)(No.24500079)、(B)(No.23300009)]、及び関西エネルギー・リサイクル科学研究振興財団の支援を受けて遂行されています。

参考文献

[1] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[2] Mahamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Building Application Frameworks*. John Wiley & Sons., 1999.

[3] SptingSource. Spring framework. http://http://www.springsource.org/.

[4] Apache Software Foundation. Struts framework. http://struts.apache.org/.

[5] The Eclipse Foundation. Eclipse modeling framework project (emf). http://www.eclipse.org/modeling/emf/.

[6] ArgoUML project home page. http://argouml.tigris.org/.

[7] GEF project home page. http://gef.tigris.org/.

[8] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.

[9] Steve Sparks, Kevin Benner, and Chris Faris. Managing object-oriented framework reuse. *IEEE Computer*, 29(9):52–61, 1996.

[10] Martin Monperrus, Marcel Bruch, and Mira Mezini. Detecting missing method calls in object-oriented software. In *ECOOP*, pages 2–25, 2010.

[11] gefdemo project. http://gefdemo.tigris.org/.

[12] Izuru Kume and Etsuya Shibayama. A conceptual model for comprehension of object-oriented interactive systems. In *International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, 2009.

[13] Izuru Kume and Etsuya Shibayama. Feature interactions in object-oriented effect systems from a viewpoint of program comprehension. In *International Conference on Feature Interactions*, 2009.

[14] Adrian Lienhard. *Dynamic Object Flow Analysis*. Lulu.com, 2008.

[15] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugger. In *ECOOP*, pages l592–615, 2008.

[16] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, pages 365–383. ACM, 2005.

[17] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA*, pages 385–402. ACM, 2005.

[18] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Dynamic query-based debugging of object-oriented programs. *Automated Software Engineering*, 10(1):39–74, January 2003.

[19] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable Omniscient debugging. In *OOPSLA*, pages 535–552. ACM, 2007.

© 2012 Information Processing Society of Japan