# コンポーネントの責務と依存制約に基づくリファクタリング支援

陽 文樹[1]　林 晋平[1,a)]　佐伯 元司[1]

概要：ソフトウェアアーキテクチャパターンに適合するコードの記述は保守コストの削減のために重要である．しかし，パターンで定義された制約に従うコードの記述は開発者の負担となり，実際にはこれに違反するコードが記述される．本稿では，アーキテクチャ適合のためのリファクタリングプロセスを支援するために，アーキテクチャ制約における違反コードを不吉な臭いとして検出する手法を提案する．提案手法はソースコードから抽出したコード片間の依存関係グラフ，およびアーキテクチャに則した推定規則を入力とし，グラフの各ノードに付加した所属可能なコンポーネントの集合の情報を段階的に更新していく．推定規則はコンポーネントの責務と依存制約を表現しており，周辺コード片の現推定状態から制約を満たさない各ノードのコンポーネント候補を除いていく．最終結果に現所属コンポーネントが含まれていない場合，そのコード片を違反として検出する．MVC2 アーキテクチャを対象として規則を定義し，Play Framework を用いた Web アプリケーション群に対して手法を適用したところ，高精度の検出結果を得た．

## 1. はじめに

ソフトウェアシステムの基礎となる構造を定義する様々なソフトウェアアーキテクチャパターン（以下，アーキテクチャパターン）が提案されている [1]．これらは複数のコンポーネントからなり，各コンポーネントに対して責務や依存制約を定義することにより，パターンに従うアーキテクチャに対して変更容易性や再利用性といった特性を与える．ソフトウェア開発においてアーキテクチャパターンに従ったアーキテクチャを採用することは，保守コスト削減のために重要である．

アーキテクチャパターンによる特性を実現するためには，開発者が各コンポーネントのコードをその責務や依存制約に従って記述することが必要となる．これに違反したコードを記述すると特性は保証されず，アーキテクチャパターンの利点は失われる．

しかし，実際の開発においてはしばしばコンポーネントの責務や依存制約に違反したコードが発生する．責務や依存制約に従ったコードの記述は開発者にとってしばしば負担となり，特に時間的制約などがある場合には，開発者はソフトウェアの完成を優先してやむなく違反コードを記述することがある．このような場合，保守の段階においてアーキテクチャパターンによる利を得るためには，これら違反コードを除去するリファクタリング [2] が必要となる．

これまでに，コンポーネント間の依存制約に基づいたアーキテクチャ適合のリファクタリング手法が提案されている [3]．しかし，依存制約にのみ基づいたリファクタリングでは，違反コードの修正時に，責務に関する新たな違反コードを生み出してしまう可能性がある．そのため，アーキテクチャパターンに従ったアーキテクチャのリファクタリングでは，コンポーネントの責務と依存制約の両者を考慮する必要がある．

本稿では，アーキテクチャ適合のためのリファクタリングに対して，コンポーネントの責務と依存制約の両者を考慮した支援を行うことを目指す．既存手法によるリファクタリングは責務と依存制約の両者を考慮しておらず，一方の問題の修正が他方の観点で問題を引き起こす可能性があるため，提案手法では両者を同一の形式で扱うことにより，両者の影響を互いに考慮できるようにする．コンポーネントの責務や依存制約に対応したコードのパターンを定義し，提案手法に適用できることを示す．本稿では，アーキテクチャパターンとして MVC Architecture for Web Application（以下 MVC2）[4]，その実現フレームワークとして Play Framework v1 [5] を用いた場合を対象に手法を実現し，その有用性を議論する．

本稿の以降の構成を以下に示す．2 章ではアーキテクチャパターンとそのリファクタリングにおける問題点を，例題を用いて説明する．3 章で前章の問題点を解決する手

1　東京工業大学
Tokyo Institute of Technology, Tokyo 152–8552, Japan
a)　hayashi@c.titech.ac.jp

©2016 Information Processing Society of Japan

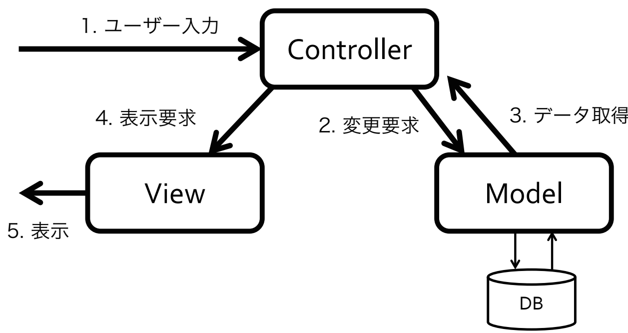


図 1　MVC2 アーキテクチャ

法を提案する．4 章では提案手法の実現について，とりわけ手法で用いる推定規則とツールによる自動化に関して述べる．5 章では提案手法を複数のプロジェクトに適用した評価について述べる．6 章では関連研究について概説し，提案手法との差異を述べる．最後に，7 章で本稿をまとめる．

## 2. 背景と問題点

### 2.1 ソフトウェアアーキテクチャパターン

アーキテクチャパターン [1] はソフトウェアシステムの基礎となる構造（アーキテクチャ）を定義する．これらアーキテクチャは複数のコンポーネントと呼ばれるカプセル化された機能単位からなり，アーキテクチャパターンは各コンポーネントに対して責務と依存制約を与えることにより，様々な非機能特性を保証する．

アーキテクチャパターンの例として MVC2 [4] がある．MVC2 は Model-View-Controller（MVC）パターンを Web アプリケーションに適した形式へと変形したアーキテクチャパターンである．MVC2 の構造を図 **1** に示す．MVC2 の処理フローは次の通りである．まずユーザ入力を Controller コンポーネントが受け取り，必要に応じて Model コンポーネントに対してデータの変更を要求する．Model コンポーネントは変更要求に応じてデータベースのデータを更新する．その後，Controller コンポーネントは表示に必要なデータを Model コンポーネントに対して要求し，Model コンポーネントは対応するデータをデータベースから取得し Controller コンポーネントへと渡す．最後に Controller コンポーネントは適切な View コンポーネントに対して表示に必要なデータを渡すとともに表示を要求する．

MVC2 では Model コンポーネントがデータベース操作を含むドメインロジックを，View コンポーネントがプレゼンテーションロジックを，Controller コンポーネントがユーザ入力に従ったこれらの制御をそれぞれの責務としている．それぞれの責務を分けることにより，各コンポーネントの置換が可能となる．例えば，View コンポーネントを置き換えることによりアプリケーションの外観を変更したり，Controller コンポーネントを置き換えたりすることによる自動テストの実行などが可能となる．また，開発者が関心事に集中できるという利点もある．

MVC2 では各コンポーネント間にアクセス可否の関係も定義している．例えば，Model コンポーネントからは View コンポーネントを参照することができない．これはコンポーネントに対する依存制約とみなすことができ，こういった制約を与えることにより，変更による影響範囲を小さく抑えられる．例えば，Model コンポーネントが View コンポーネントに依存しないことにより，アプリケーションの外観の変更時にドメインロジックのコードに変更が波及することがないという利を得られる．

### 2.2 アーキテクチャパターン適合のためのリファクタリングとその問題点

アーキテクチャパターンはこれに従うアーキテクチャに対して様々な非機能特性を保証する．これは主に保守コスト削減のために有用である．しかし，これらの特性が保証されるのは各コンポーネントのコードがその責務と依存制約に従って正しく記述されている場合に限る．時間的制約などが原因で，責務や依存制約に違反するコードが含まれている場合にはこれらの特性は失われる．保守の段階において特性を活用するためには，違反コードを取り除くリファクタリングが必要となる．

我々は，アーキテクチャパターンにおける違反を以下の 2 つに大別した．

**コンポーネントの責務に対する違反（責務違反）**　責務違反は，プログラム上でコードの果たしている役割が，その所属コンポーネントとは別のコンポーネントの責務を体現しているという違反である．例えば，MVC2 において View コンポーネントに記述すべきであるプレゼンテーションロジックが Model コンポーネントや Controller コンポーネントに所属するコードに記述されている場合などが該当する．

**コンポーネントの依存制約に対する違反（依存制約違反）**　依存制約違反は所属コンポーネントに許可された依存対象および方法以外の依存が発生しているという違反である．例えば，MVC2 において Model コンポーネントのコードが View コンポーネントや Controller コンポーネントのフィールドやメソッドを参照している場合などが該当する．

これら責務違反と依存制約違反はそれぞれ異なる観点による違反である．したがってこれらを別個にリファクタリングすると，片方の違反を解消するためのリファクタリングがもう片方の違反を生み出してしまう可能性がある．よって，アーキテクチャパターンにおける違反を正しく解消するリファクタリングのためには責務違反と依存制約違反の両方を考慮した仕組みが必要となる．

©2016 Information Processing Society of Japan

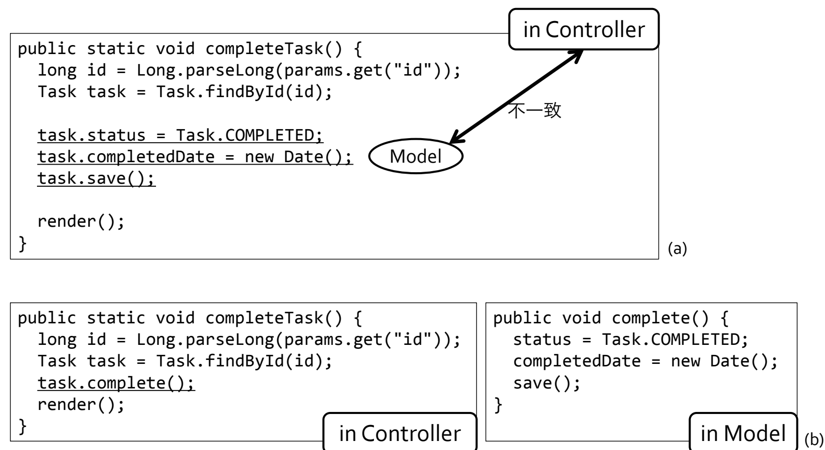


図 **2**　アーキテクチャ制約の違反の例

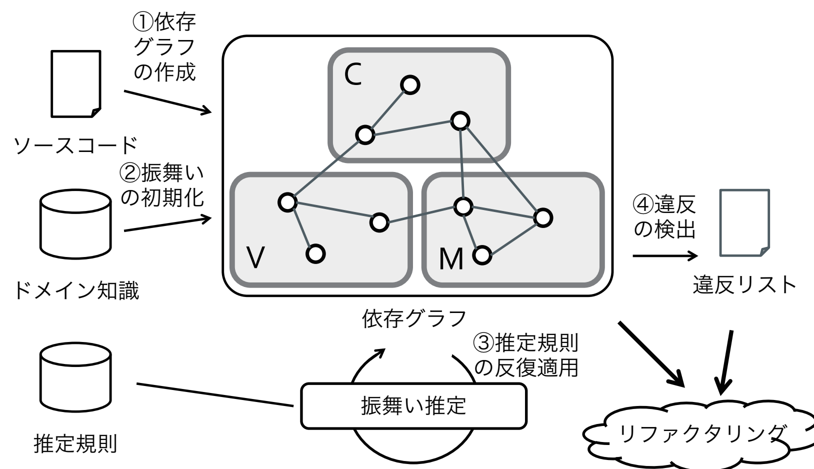


図 **3**　提案手法の概要

MVC2 における違反の例を図 **2**(a) に示す．この例は，Play framework v1 [5] を用いて開発されたタスク管理アプリケーションのうち，Controller として記述された，タスクを完了するためのアクションメソッドを示している．このメソッドでは，ユーザからの入力に基づいて Task オブジェクトを特定した後，下線で示した 3 行のコード片群により，タスクの完了状態の更新，完了日付の更新，およびタスクの保存を行っている．これらの更新・保存処理はデータベース更新に関する不可分な一連の処理とみなすことができ，Model が責務として担うべきドメインロジックと考えることができる．しかし，このメソッドは Controller に属しているため，振舞いと責務の間にギャップをなす責務違反を生じている．

この違反は，図 2(b) に示す構造へとリファクタリングにすることにより解消できる．該当のコード片は，Extract Method リファクタリングにより独立したメソッド `complete` として抽出され，また Move Method リファクタリングにより Model に配置されたため責務の不一致をもはや生じない．こういったリファクタリングを実現するためには，アーキテクチャ制約に違反するコード片を特定する手法が求められる．

## 3. 提案手法

### 3.1 アプローチ

アーキテクチャパターンにおける違反を正しく解消するリファクタリングのためには責務違反と依存制約違反の両方を考慮する必要がある．提案手法ではそのアプローチとしてコードの振舞い推定を導入する．ここで，本稿でいう振舞いとは，コードがプログラム上で果たす役割と周辺コードとの関係により決定される，コードが所属可能なコンポーネント候補の集合を指す．すなわち，振舞いの推定とは各コードの所属可能なコンポーネントの推定を意味する．振舞い推定では各コンポーネントの責務や依存制約をそれぞれ推定規則として表し，用いる．責務と依存制約を同列の推定規則として扱うことによりそれぞれが互いを考慮できる仕組みを提供する．

提案手法の概要を図 **3** に示す．提案手法の入力は，ソースコード，コード片の振舞いを初期化するためのドメイン

©2016 Information Processing Society of Japan

表 1　コード片間の依存関係

| 関係 | 依存元 | 依存先 | 概要 |
|---|---|---|---|
| Def-Use | コード片 | コード片 | 変数の定義と参照 |
| アクセス | コード片 | フィールド | フィールドの読み書き |
| 呼び出し | コード片 | メソッド | メソッド呼び出し |
| 包含 | コード片 | メソッド | コード片の包含 |

知識，および振舞い推定のための推定規則データベースである．提案手法では，まず与えられたソースコードを解析し，コード片とそれらの間の関係を抽出して依存グラフを構築する（①依存グラフの作成）．次に，得られた各コード片の振舞いをドメイン知識に基づき初期化する（②振舞いの初期化）．その後，アーキテクチャに対応した推論規則を用いて振舞いの可能性を絞る振舞い推定を行う（③推定規則の反復適用）．このプロセスにより，各コード片がどのコンポーネントに所属可能であるかを特定する．コード片の振舞いが決定すると，現所属コンポーネントとの比較により違反を検出する（④違反の検出）．すなわち，コード片の振舞いに現所属コンポーネントが含まれていなければ，コード片は所属すべきコンポーネントでないコンポーネントに所属していると考える．

このようにして検出された違反コード片を，振舞いに対応したコンポーネントへと移動することでリファクタリングを実現する．リファクタリング操作には Extract Method や Move Method [2] などを用いる．ここで，提案手法はリファクタリング操作の導出法を含まず，違反の検出までを自動化することまでを目的としていることを付記しておく．

提案手法には，ドメイン知識と推定規則の準備が必須である．一方で，フレームワークを用いた開発を仮定すれば，利用するアーキテクチャパターンとその実装上の対応関係は事前に明確化することができる．該当フレームワークの専門家がドメイン知識と推定規則を事前に定義しておくことにより，利用しているアーキテクチャパターンに明るくない，ドメイン知識を持たない開発者でも提案手法を利用できると考える．

### 3.2　依存グラフの構築

提案手法ではソースコードをコード片へと分割し，その間の関係を抽出することにより依存グラフを構築する．コード片の単位はコード片間の関係を取得する上で適切である文単位を用いる．ただし，後述するメソッド呼び出し関係の考慮のため，文中のメソッド呼び出しも個別のノードとして扱う．また，ソースコードからは同時にフィールドとメソッドの情報もノードとして取得する．

次に，各コード片間，コード片とフィールド間，コード片とメソッド間の関係を抽出する．各コード片間，コード片とフィールド間，コード片とメソッド間の抽出すべき関係を表 1 に示す．Def-Use 関係はコード片間に定義されるデータ依存の関係であり，例えば，あるコード片で定義した変数 $v$ を他のコード片内で参照していた場合，この 2 つのコード片は変数 $v$ に関する Def-Use 関係にあるとする．アクセス関係はコード片とフィールドの間に定義されるフィールドへの読み書きの関係であり，例えば，あるコード片内にてあるフィールドの読み込みあるいは書き込みを行っていた場合，このコード片とフィールドはアクセス関係にあるとする．呼び出し関係はコード片とメソッドの間に定義されるメソッド呼び出しの関係であり，例えば，あるコード片内にてあるメソッドの呼び出しを行っていた場合，このコード片とメソッドは呼び出し関係にあるとする．包含関係はコード片とそのコード片を含むメソッドの間に定義される関係である．

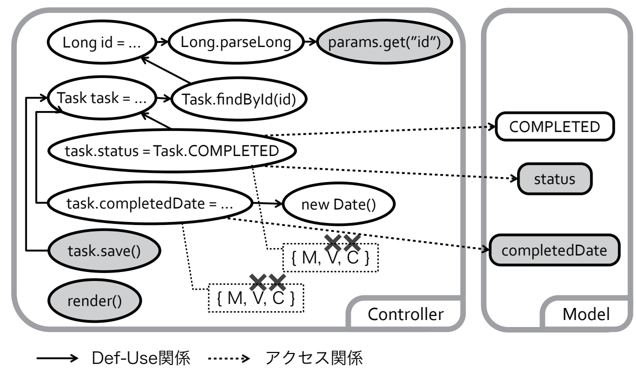


図 4　図 2 の例題に対する依存グラフと振舞い推定

図 2 に示すコードを入力とした場合に構築される依存グラフを図 4 に示す．メソッド中の各文やメソッド呼び出しに対応するコード片がノードとして抽出されていることが見て取れる．また，ノード間には依存関係が定義されており，例えば変数 `task` を定義するノード“`Task task = ...`”に対して，該当変数を使用しているノード“`task.status = Task.COMPLETED`”, “`task.completeDate = ...`” などがアクセス関係にあるノードとして定義されている．

### 3.3　振舞いの初期化

続いて，振舞いの初期化を行う．依存グラフにおける各ノードはそのノードが表すコード片，フィールド，メソッドの現推定状態における振舞いを持ち，この振舞いの最終状態が振舞い推定の出力となる．前述したように，振舞いとはコード片が所属可能なコンポーネント候補の集合を表しており，図 4 の例においては，モデル（$M$），ビュー（$V$），コントローラ（$C$）の 3 つが対象となるコンポーネントとなる．

基本的には，各ノードにはすべての可能性を割り当てる．すなわち，ノードの振舞いをすべてのコンポーネントの集合として初期化する．一方，特定のコード片の振舞いが既知であることがドメイン知識からわかっている場合は，割

表 2　Modification 規則で用いるコンポーネント間の更新可能関係

| 変更元＼変更先 | Model | View | Controller |
|---|---|---|---|
| Model | √ | | |
| View | | √ | |
| Controller | √ | | √ |

り当てる振舞いをあらかじめ絞っておく．

図 4 の例においては，白色のノードはすべての可能性，すなわち {Model, View, Controller} が割り当てられている．一方，灰色のノードは，ドメイン知識により可能性が一意に特定されている．例えば，Play Framework v1 においては，メソッド呼び出し `render()` はコントローラに記述されることがすでに知られているため，モデルやビューの可能性が削られた候補 {Controller} として初期化される．また，モデルに属するクラスに記述されたフィールド `status` や `completedDate` はモデルとして振る舞うため，{Model} として初期化される．このように，入力となるドメイン知識の多くは，メソッド名と対応するコンポーネントの辞書として機能する．

### 3.4　振舞い推定

振舞い推定では，各コード片，フィールド，メソッドに対して，所属可能なコンポーネントの候補を段階的に推定していく．扱っている依存グラフの各ノードはそのノードが表すコード片，フィールド，メソッドの現推定状態における振舞いを持っており，この振舞いの最終状態が振舞い推定の出力となる．

各ノードの振舞いの更新には各アーキテクチャに対応したコンポーネントの責務や依存制約を表す推定規則を用いる．推定規則はグラフの構造と周辺ノードの振舞いを参照して，各ノードの振舞いから候補を除去していく．ある規則による振る舞いの候補の変化が他の規則の効能に影響する場合があるため，これらの規則を変化がなくなるまで繰り返し適用することによって振舞い推定は完了する．

推定規則には，現在の推定状態に結果が依存するものとしないものがある．推定規則を適用する際には，推定状態に結果が依存しない規則を優先的に適用し，これによる変化がなくなった場合に推定状態に結果が依存する規則を適用する．適用の結果，ノードの振舞いに変化があった場合には再度，推定状態が結果に依存しない規則を優先した適用を繰り返す．

推定規則によるノードの振舞いの変化がなくなったとき，各ノードに対して振舞いが推定されたとみなす．すなわち，各コード片，フィールド，メソッドとその振舞い，所属可能なコンポーネントの対応が取得できる．これを振舞い推定の出力とする．

MVC2 における Modification 規則を例にあげ，振舞い推定の例を示す．Modification 規則は，コンポーネント間の状態変更の可否の関係を用いたコンポーネントの依存制約に基づく推定規則である．MVC2 アーキテクチャにおいては，Model の状態は Model と Controller のみが可能であり，View は行えない．この制約は，より詳細にはどのコンポーネントがどのコンポーネントを変更可能かという二項関係をなし，表 **2** のように表すことができる．表の各行は変更元，各列は変更先のコード片の所属するコンポーネントを表し，表中の印 √ は，変更が可能であることを示している．この制約を満たさない（√ の振られていない）振舞いの候補の可能性を，不適切な候補として除外する．

引き続き図 4 を用いて，Modification 規則の適用がどのように行われるかを示す．フィールド “`status`” はドメイン知識により，その振舞いが Model に特定されている．また，ノード “`task.status = Task.COMPLETED`” は，フィールド “`status`” と変更のアクセス関係にある．このノードは，Model と確定しているノードを変更しているため，表 2 を鑑みれば View ではないことがわかる．そのため，ノード “`task.status = Task.COMPLETED`” の振舞い候補より View を除外し，{Model, Controller} が残る．同様に，ノード “`task.completeDate = ...`” の振舞い候補からも View が除外される．この例では，後述する Anemic Domain Model 規則の適用も助け，最終的に Controller の可能性も除外され，2 つのノードの可能性が Model に限定されている．

### 3.5　違反の検出

振舞い推定により，各コード片，フィールド，メソッドの所属可能なコンポーネントが取得できるため，これと現所属コンポーネントを比較することで違反コードの検出を行う．すなわち，推定された所属可能コンポーネントの集合に現所属コンポーネントが含まれていなければ，コンポーネントの責務か依存制約あるいはその両方に違反しているコードであると判断する．

ここで検出できる違反は，推定された振舞いによって以下の 2 通りに分けられる．

- 振舞いに候補が 1 つ以上存在し，その候補に現所属コンポーネントが含まれない場合
- 振舞いに候補が 1 つも存在しない場合

振舞いに候補が 1 つでも存在する場合の違反は，そのコード片，フィールド，メソッドが所属すべきコンポーネントとは異なるコンポーネントに所属していることを示している．この場合，対応コードは候補のうちのいずれかのコンポーネントへと移動されることが望まれる．一方，振舞いに候補が 1 つも存在しない場合の違反は，そのコード片，フィールド，メソッドが複数のコンポーネントに対応する役割あるいは関係を有していることを示している．この場合，対応コードあるいはその周辺コードを分割し，対応す

©2016 Information Processing Society of Japan

るコンポーネントが 1 つとなるように修正することが望まれる.

図 4 の例では, 振舞い推定によりノード“`task.status = Task.COMPLETED`” および “`task.completeDate = ...`” の振舞いにおける View, Controller の可能性が除外され, 振舞いが Model と特定されている. 一方, これらのノードは Controller に属しており, 得られた振舞い候補に所属コンポーネントが含まれないため, 違反として検出される. この場合は, 残った振舞い候補である Model へ該当コード片を移動するリファクタリングの適用が検討される.

## 4. 提案手法の実現

### 4.1 推定規則

MVC2 を対象に推定規則を定義した. 規則は依存制約に基づくものと責務に基づくものの両方を用意した. 依存制約に関する規則は, 各コンポーネント間のアクセス可否の関係に従って, データ依存に関する制約を表現する Def-Use, インタフェースの可視性に関する制約を表現する Visibility, 状態変更の可否に関する制約を表現する Modification の 3 つを用意する. これは MVC2 の定義から容易に導出できるものであり, 他のアーキテクチャパターンにおいても同様の制約に基づく規則が同様に構築可能であると考えている. 例えば, Layers アーキテクチャ [1] では隣り合っていないレイヤ感のアクセスを制限しており, これは依存制約として記述できる.

一方で, MVC2 における責務の定義は曖昧であり, 責務に関する規則を定義に基づき記述することが難しい. そこで, 実際のプロジェクトにて観察された違反パターンをもとに責務に関する規則を抽出した. 情報工学を専攻する学生が作成した Web アプリケーション 11 件のソースコードを著者らのうち 1 名が手作業で分析し, コンポーネントの責務と一致していない箇所を特定したところ, それらのうちに以下の 2 種類のパターンを観察できた.

**表示用文字列の Controller での生成** 表示用の文字列生成というプレゼンテーションロジックに当たる処理を, View コンポーネントではなく Controller コンポーネントで行っているもので, MVC2 のアーキテクチャに違反している. MVC2 のアーキテクチャに則した記述とするためには, 文字列の元となるデータのみを Controller コンポーネントから View コンポーネントへと渡し, 文字列の生成自体は View コンポーネントにて行う必要がある.

**Controller でのドメインロジックの記述** Anemic Domain Model アンチパターン [6] に該当するもので, Model コンポーネントに記述すべきドメインロジックを Controller コンポーネントに記述しているため, MVC2 のアーキテクチャに違反している. MVC2 のアーキテクチャに則した記述とするためには, 可能な限りドメインロジックを Model コンポーネントに記述する必要がある.

これらを含め, MVC2 におけるコンポーネントの責務や依存制約に関する 5 規則を定義した. それらの概要を以下に示す.

**Def-Use 規則** コンポーネントのデータ依存関係を用いた, コンポーネントの依存制約に関する規則である. 各コンポーネント間にはデータ依存の関係があり, 例えば, View コンポーネントで定義した変数を Model コンポーネントや Controller コンポーネントで参照することはできない. この規則はこれらコンポーネント間の関係と Def-Use 関係を用いて, 周辺コード片の振舞いからコード片の振舞いの候補を除く.

**Visibility 規則** コンポーネント間のインタフェースの可視性を用いた, コンポーネントの依存制約に関する規則である. 各コンポーネント間では互いのインタフェースの存在を仮定できるか否かが定義されている. この規則ではこれらコンポーネント間の関係とアクセス関係および呼び出し関係を用いて, フィールド, メソッドの振舞いからコード片の振舞いの候補を除く.

**Modification 規則** コンポーネント間の状態変更の可否の関係を用いた, コンポーネントの依存制約に関する規則である. 各コンポーネント間では互いの状態を変更できるか否かが定義されている. この規則ではこれらコンポーネント間の関係とアクセス関係, 呼び出し関係, 包含関係を用いて, 変更されたフィールドの振舞いからコード片の振舞いの候補を除く.

**Visual String 規則** 表示用文字列を Controller コンポーネントで生成しているパターンを規則化したもので, コンポーネントの責務に関するものである. この規則は View コンポーネントのインタフェースをドメイン知識から取得し, Def-Use 関係を辿ることで View コンポーネントへと渡す文字列を生成しているコード片を検出し, これを View コンポーネントの振舞いであると推定する. ただし, 生成された文字列が Model コンポーネントへと渡されている場合や制御フローに影響を与えている場合には, 必ずしも表示用の文字列ではないと判断し除外する.

**Anemic Domain Model 規則** ドメインロジックを Controller コンポーネントに記述しているパターンを規則化したもので, コンポーネントの責務に関するものである. この規則はコード片の依存対象が Model コンポーネントの振舞いのコード片と Controller コンポーネントの振舞いのコード片のどちらに多いかを判定し, Model コンポーネントの振舞いのコード片に多いと判断した場合にはそのコード片も Model コンポー

ネントの振舞いであると推定する.

これらの規則を，振舞い候補に変更がでないようになるまで反復的に適用する.

### 4.2 ツールによる自動化

提案手法を統合開発環境である Eclipse [7] のプラグインとして実装した．対象とするアーキテクチャパターンには MVC2 を選択し，MVC2 に従った Web アプリケーションフレームワークである Play Framework v1 [5] を用いて構築されたソフトウェアを解析対象とした．振舞い推定の初期化処理にて用いるドメイン知識や Visual String 規則にて用いる View コンポーネントのインタフェースに関する知識にはこのフレームワークに関する知識を用意して利用した.

依存グラフの構築には同じく Eclipse のプラグインである jxplatform [8] を用いた．jxplatform はデータフロー解析の基盤となるツールであり，Eclipse JDT によって作成された Java ソースコードの抽象構文木を用いて，システム依存グラフ，プログラム依存グラフ，制御フローグラフおよびコールグラフを含む Java のモデルを生成する.

実装上の意思決定の概要を以下に示す.

- 抽出する依存グラフは，基本的には jxplatform の提供するプログラム依存グラフを採用した．プログラム依存グラフにおけるノード分割は概ね抽象構文木の文に従って行われるが，メソッド呼び出しやインスタンス生成などに関しては別のノードとして分割される．また特定条件下にてこれらの式はさらに実引数や戻り値，式自体のノードへと分割される．これは我々の目的に対して細粒度すぎるため，実引数や戻り値，式自体のノードを統合し，同一のコード片として扱うようにした．また，コードと直接対応しないノードは無視した.
- jxplatform の作成したプログラム依存グラフに含まれる情報をもとに Def-Use 関係を抽出した．jxplatform ではローカル変数の定義や代入を変数の定義 (Define)，ローカル変数やフィールドの参照，メソッド呼び出しを変数の参照（Use）とみなし，これらが行われたノード間に Def-Use 関係があるとする．またメソッド呼び出しやインスタンス生成に対してこれらの式自体のノードとこれらの戻り値を参照するノード間に Def-Use 関係があるとする．前述したように，我々のツールでは一部のノードを統合，無視しているため，対応する関係も統合して利用している.
- アクセス関係についてはコード片内の抽象構文木を探索し，フィールドへの代入を行っている場合にそのフィールドに対して書き込みのアクセス関係にあるとし，その他のフィールドの参照がある場合にはそのフィールドに対して読み込みのアクセス関係にあるとした.

## 5. 評価

### 5.1 評価方法

実装ツールを複数のプロジェクトに対して適用し，手法の評価を行った．評価の観点は以下の通りである.

**Q1** 提案手法により違反を検出できるか?

**Q2** 各推定規則は違反の検出に有効であるか?

**Q3** 振舞いの候補が存在しないパターンがあるか?

Q1 は各推定規則が表現するコンポーネントの責務や依存制約に違反したコードの検出が可能かどうかに関するものである．例えば，Visual String 規則であれば，表示用の文字列を Controller コンポーネントにて生成しているパターンを検出できるかどうか，Modification 規則であれば，View コンポーネント内のコードで Model コンポーネントの状態を変更していることを検出できるかどうか，に相応する.

Q2 は各推定規則に無意味なものが含まれていないかどうかに関するものである．各推定規則がコード片の振舞いから候補を除いており，またその候補の除去が違反の検出に影響しているかどうかを確認する.

Q3 は複数の推定規則，特に責務に関する規則と依存制約に関する規則の組み合わせによって振舞いの候補が存在しない違反が検出されるかどうかに関するものである．このような違反はコンポーネントの責務と依存制約の両者が考慮された結果といえる.

実装ツールの適用は情報工学を専攻する学生が開発した Web アプリケーション群（全 37 プロジェクト）に対して行った [*1]．これらのプロジェクトは Play Framework v1 を用いており，MVC2 アーキテクチャに従って開発されている.

評価は以下の 4 つの方法で行った.

- 適合率の計測
- 再現率の計測
- 推定規則の適用数の計測
- 違反コードの目視による調査

適合率と再現率の計測は Q1 の，推定規則の適用数の計測は Q2 の，違反コードの目視による調査は Q3 の確認を目的として行われた．適合率の計測は全プロジェクトに対してツールを適用し，検出された違反についてそれぞれ検出結果が正しいかどうかを著者らのうち 1 名が手作業で確認して行った.

ただし，Anemic Domain Model 規則が想定する違反については明確な正誤の判断基準を定めることが困難であるため，明確な場合の正，誤の判定に加えて，判断不可の判定

[*1] 4.1 節での分析に用いたプロジェクトは含まれていない.

表 3 適合率の計測結果

| No. | 検出数 | 有効検出数 | 正解数 | 適合率 |
|---|---|---|---|---|
| 1 | 5 | 5 | 4 | 0.80 |
| 2 | 7 | 7 | 7 | 1.00 |
| 3 | 7 | 7 | 7 | 1.00 |
| 4 | 8 | 7 | 7 | 1.00 |
| 5 | 8 | 8 | 6 | 0.75 |
| 6 | 8 | 8 | 8 | 1.00 |
| 7 | 8 | 8 | 8 | 1.00 |
| 8 | 8 | 8 | 8 | 1.00 |
| 9 | 8 | 8 | 8 | 1.00 |
| 10 | 10 | 10 | 10 | 1.00 |
| 11 | 11 | 11 | 11 | 1.00 |
| 12 | 11 | 11 | 11 | 1.00 |
| 13 | 12 | 12 | 12 | 1.00 |
| 14 | 12 | 12 | 12 | 1.00 |
| 15 | 13 | 13 | 13 | 1.00 |
| 16 | 13 | 13 | 13 | 1.00 |
| 17 | 13 | 13 | 13 | 1.00 |
| 18 | 15 | 15 | 14 | 0.93 |
| 19 | 16 | 16 | 16 | 1.00 |
| 20 | 16 | 16 | 16 | 1.00 |
| 21 | 17 | 14 | 14 | 1.00 |
| 22 | 17 | 17 | 17 | 1.00 |
| 23 | 19 | 13 | 13 | 1.00 |
| 24 | 19 | 17 | 14 | 0.82 |
| 25 | 19 | 19 | 19 | 1.00 |
| 26 | 19 | 19 | 19 | 1.00 |
| 27 | 21 | 21 | 17 | 0.81 |
| 28 | 24 | 24 | 24 | 1.00 |
| 29 | 25 | 25 | 25 | 1.00 |
| 30 | 26 | 26 | 26 | 1.00 |
| 31 | 28 | 15 | 14 | 0.93 |
| 32 | 28 | 28 | 28 | 1.00 |
| 33 | 30 | 25 | 24 | 0.96 |
| 34 | 35 | 32 | 28 | 0.88 |
| 35 | 44 | 32 | 28 | 0.88 |
| 36 | 63 | 32 | 9 | 0.28 |
| 37 | 71 | 27 | 24 | 0.89 |
| 合計 | 714 | 594 | 547 | 0.92 |
| 平均 | 19.3 | 16.1 | 14.8 | 0.94 |

を導入した．具体的には，Anemic Domain Model 規則が適用された結果として検出された違反コードがデータベース更新に関わる状態変更を行っていた場合にはドメインロジックであるとして正，ユーザ入力または表示の準備，あるいはそれらに関わる処理フローに影響するものであった場合にはドメインロジックではないとして誤，それ以外に対しては判断不可と判定した．

再現率の計測については，全プロジェクトの正しい検出結果を人手で用意することが困難であったため，ランダムに選択した 5 プロジェクトに対して著者らのうち 1 名があらかじめ手作業ですべての違反を特定し，その後にツールを適用して手作業で発見したもののうちどの程度検出できているかを確認した．この際，Anemic Domain Model 規則が想定する違反については適合率と同様の基準を用いて，正と判断されるものだけを違反とした．

表 4 再現率の計測結果

| No. | 違反数 | 検出数 | 再現率 |
|---|---|---|---|
| 1 | 11 | 10 | 0.91 |
| 2 | 17 | 16 | 0.94 |
| 3 | 20 | 13 | 0.65 |
| 4 | 30 | 28 | 0.93 |
| 5 | 31 | 12 | 0.39 |
| 合計 | 109 | 79 | 0.73 |
| 平均 | 21.8 | 15.8 | 0.76 |

推定規則の適用数の計測は全プロジェクトに対してツールを適用し，各推定規則が振舞いの候補を除いた回数を計測した．また，このうち最終的に違反の検出に貢献しなかったものを除外した回数も計測した．

違反コードの目視による調査は 37 のプロジェクトに対してツールを適用し，検出された振舞いの候補がない違反に対して，手作業でその原因を確認した．

### 5.2 結果

適合率の計測結果を表 **3** に示す．表は左から順にプロジェクト番号，検出された違反の数，検出された違反のうち判断不可と判定されたものを除いた数，正と判断された数，適合率の値を示しており，プロジェクトは検出数で整列されている．多くのプロジェクトでは高い適合率を示しており，平均でも 0.94 と高い値となっている．一方で，非常に低い値を示しているプロジェクトも存在している．これらのプロジェクトを調査すると，多くの誤検出が Anemic Domain Model 規則に起因するものであった．ひとつのコード片が誤って Model コンポーネントの振舞いであると判断された結果，その周辺コードまで同様に Model コンポーネントの振舞いであると誤った判断をしてしまい，誤検出が増えていた．

再現率の計測結果を表 **4** に示す．表は左から順にプロジェクト番号 [*2]，手作業で発見した違反の数，提案手法により検出された違反の数，再現率の値を示しており，プロジェクトは違反数で整列されている．再現率の値はプロジェクトによって大きく差が出ている．値が低いものを調査すると，Anemic Domain Model が想定していた違反が検出できず，低い値となっていた．これは違反箇所とその根拠となる箇所が離れていたために振舞いがうまく伝播しなかったことが原因であった．

推定規則の適用数の計測結果を表 **5** に示す．表は左から順に推定規則の名前，振舞いの候補を除いた回数，振舞いの候補を除きかつそれが最終的に違反の検出に影響した回

[*2] 表 3 のものとは対応しない．

表 5　推定規則の適用数

| 規則名 | 総適用数 | 有効適用数 |
| --- | --- | --- |
| Def-Use | 3578 | 315 |
| Visibility | 781 | 583 |
| Modification | 925 | 478 |
| Visual String | 48 | 48 |
| Anemic Domain Model | 856 | 620 |

in Controller

```
public static void signIn() {
  String status = session.get("status");
  String message = "";
  if (status != null && status.equals("wrongpass")) {
    message = "名前、もしくはパスワードが間違っています";
  }

  renderArgs.put("message", message);
  render();
}
```

{ M, V, C }　{ C }　{ V }　{ }

図 5　振舞いの候補がなくなった例

数を示している．有効適用数は高く，すべての推定規則は振舞いの推定に有効に機能していることを示唆している．

違反コードの目視による調査では，Visual String 規則が適用されたコード片に対して振舞いの候補無しとして違反となっていることが確認できた．この例を図 **5** に示す．Visual String 規則では表示用文字列の生成部分のコード片に対して，View コンポーネントの振舞いであると推定されるが，これらのコード片に同時に Def-Use 規則が適用されたことにより View コンポーネントがコード片の振舞いから除かれて振舞いの候補がなくなっていた．これはコンポーネントの責務と依存制約の両者を考慮した結果，表示用文字列の生成部分のコード片を View コンポーネントに移動させることでは違反を解決できないことを示している．コード片の修正や分割など，他のリファクタリング操作の導入が必要と考える．

### 5.3　考察

結果をもとに 5.1 節で示した観点から考察を行う．

**Q1** 提案手法により違反を検出できるか？　多くのプロジェクトにおいて高い適合率と再現率の値を得られたことから，検出できていると考える．しかし，Anemic Domain Model 規則の想定した違反についてはプロジェクトによって不適切な結果を出力する場合もあり，規則の定義に改良が求められると考える．

**Q2** 各推定規則は違反の検出に有効であるか？　すべての推定規則が違反の検出に影響を与えており，有効であると考える．

**Q3** 振舞いの候補が存在しないパターンがあるか？
Visual String 規則の適用箇所に Def-Use 規則が適用されることにより振舞いの候補がなくなるパターンを確認した．これらはコンポーネントの責務と依存制約の両方を用いた推定の結果であり，コンポーネントの責務と依存制約の両方を考慮した提案手法ならではの検出対象であると考えるが，一方で，違反解消のためのリファクタリング操作に課題が残っていることも示している．

### 5.4　妥当性の脅威

まず構成概念妥当性について，適合率と再現率の計測に用いた単位が正しくなかった可能性がある．これらの評価で用いた違反の数は違反となったコード片の数により計測しているが，複数のコード片における違反が実際には同一の原因による違反である可能性が残る．しかし，複数の違反が同一の原因であるとする明確な基準を定めることは困難である．

また内的妥当性について，適合率と再現率の計測に用いた Anemic Domain Model 規則の正誤判断基準が不十分であった可能性がある．この判断基準にはいくらかの曖昧性があり，プログラムの処理が正しく理解できていなかったために誤った判断をしている可能性がある．また，判断不可とするコードが多すぎるあるいは少なすぎるために結果が変化していた可能性がある．しかし，これらに正確な判断基準を定めることは困難である．

また外的妥当性について，ツールを適用したプロジェクトはすべて学生の記述したものであり，一般的な業務アプリケーションでは異なる結果となる可能性がある．しかし，業務アプリケーションのソースコードを得ることは難しく，またオープンソースソフトウェアでは時間的制約が少ないため違反が発生しづらい．

最後に同じく外的妥当性について，実装ツールおよび評価が MVC2 アーキテクチャのみを対象としており，他のアーキテクチャパターンでは異なる結果となる可能性がある．他のアーキテクチャパターンにおける評価は今後の課題とする．

## 6.　関連研究

Budi らは機械学習と層間のアクセス可否の関係を用いた多層アーキテクチャの違反検出手法を提案した [9]．この手法はクラスの基本情報から機械学習を用いてクラスを各層に分類し，その後クラス間のアクセス関係と層間のアクセス可否の関係を比較することにより違反を検出する．また，Hickey らは探索に基づく多層アーキテクチャのリファクタリング手法を提案した [3]．これは層間のアクセス違反を計測するメトリクスを評価に，移動のリファクタリング操作を遷移に用いて，より適切な状態を探索する．これらの手法は依存制約を用いているが，責務に関しては元のコードや訓練データに強く依存しており，アーキテクチャの責務を直接考慮してはいない点において我々の手法

©2016 Information Processing Society of Japan

とは異なる．

提案手法では違反検出後のリファクタリングとして，移動リファクタリング操作によるリファクタリングを想定としている．Tsantalis らは Move Method リファクタリングによるシステムの保守性改善手法を提案し，JDeodorant として実装した [10]．JDeodorant では結合度と凝集度を用いたシステムメトリクスにより，Move Method による保守性の改善を確認している．また，Sales らは依存集合の類似性を用いた手法により，より高い精度で Move Method による自動リファクタリングが可能であることを示した [11]．これらの手法はアーキテクチャの制約を考慮しない点で提案手法とは異なる．

## 7. おわりに

アーキテクチャパターンにおいて違反を除去するリファクタリングを行うためには，コンポーネントの責務と依存制約の両方を考慮した手法が必要となる．本稿では，各コード片の所属可能なコンポーネントを推定する振舞い推定を導入したアプローチにより違反を検出する手法を提案した．振舞い推定ではコンポーネントの責務と依存制約をともに推定規則として表現し，同列に扱うことにより互いを考慮した推定が可能となっている．MVC2 アーキテクチャ，Play Framework v1 を対象として提案手法を実装した Eclipse プラグインを実現し，複数プロジェクトに対して適用することで手法の有用性を評価した．

今後の課題として最も重要となるのは検出された違反のリファクタリング手法の考案である．本稿では違反を含むコード片の検出までは行ったが，そのコード片をどのクラスに移動させるべきであるかは扱っていない．また違反を含むコード片は周辺の関連の深いコードと一緒に移動させるほうが多くの場合に好ましい．この場合に，同時に移動させるコードを選択することもリファクタリング手法を考案する上で重要である．このとき振舞い推定の結果を利用すると同時に移動できないコード片を検出することができる．また，評価で扱ったような，単なる移動では解決できないコード片のリファクタリング方法の導入も必要である．

また，他のアーキテクチャパターンに対して適用することも重要な課題である．依存制約に関する規則は多くのアーキテクチャに対して応用できるものと考えるが，責務に関する規則はアーキテクチャによって大きく異なる可能性がある．様々なアーキテクチャに対して推定規則が責務を表現できるかどうか確認する必要がある．

**謝辞**　有益なツールをご公開頂いた立命館大学の丸山勝久教授に感謝する．本研究の一部は科学研究費補助金（15H02683，15H02685，15K15970）の助成を受けた．

### 参考文献

[1] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M.: *Pattern-oriented Software Architecture: A System of Patterns*, John Wiley & Sons (1996).

[2] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).

[3] Hickey, S. and Ó'Cinnéide, M.: Search-Based Refactoring for Layered Architecture Repair: An Initial Investigation, *Proc. 1st North American Search Based Software Engineering Symposium* (*NasBASE 2015*) (2015).

[4] Turner, J. and Bedell, K.: *Struts Kick Start*, Sams (2002).

[5] Play Framework – Build Modern & Scalable Web Apps with Java and Scala, https://www.playframework.com/ (accessed 2016-08-04).

[6] Fowler, M.: AnemicDomainModel, http://www.martinfowler.com/bliki/AnemicDomainModel.html (accessed 2016-08-04).

[7] Eclipse – The Eclipse Foundation open source community website, https://eclipse.org/ (accessed 2016-08-04).

[8] Maruyama, K.: jxplatform, https://github.com/katsuhisamaruyama/jxplatform (accessed 2016-08-04).

[9] Budi, A., Lucia, Lo, D., Jiang, L. and Wang, S.: Automated Detection of Likely Design Flaws in N-Tier Architectures, *Proc. 23rd International Conference on Software Engineering and Knowledge Engineering* (*SEKE 2011*), pp. 613–618 (2011).

[10] Tsantalis, N. and Chatzigeorgiou, A.: Identification of Move Method Refactoring Opportunities, *IEEE Transactions on Software Engineering*, Vol. 35, No. 3, pp. 347–367 (2009).

[11] Sales, V., Terra, R., Miranda, L. F. and Valente, M. T.: Recommending Move Method Refactorings Using Dependency Sets, *Proc. 20th Working Conference on Reverse Engineering* (*WCRE 2013*), pp. 232–241 (2013).

©2016 Information Processing Society of Japan 