Vol.2014-SE-183 No.9
2014/3/19

# シーケンス図に対するモデルリファクタリングの試み

石川 敦啓[1]　上田 賀一[1]

**概要**：UML の品質を向上させる手法としてモデルリファクタリングがあるが，シーケンス図に対してはまだその手法が示されていない．そこで，本研究では，ソースコードにおける不吉な匂いのうち，シーケンス図で自動検出して有効な不吉な匂いを定義し，その不吉な匂いを取り除くためのモデルリファクタリングを提案する．より有効な手法を定義するためクラス図も併用する．また，不吉な匂いに対しては自動で検出するためのアルゴリズムを，モデルリファクタリングに対してはその適用手順と，外部から見たシステムの振る舞いを変更しないことを示す．これらの手法により，品質の高いシーケンス図のモデルを作成できる．公開されている UML のモデルを用いて，提案した 4 つの不吉な匂いのアルゴリズムの妥当性を評価し，3 つの妥当性を確認し，確認できなかったアルゴリズムについては別の方法を示した．

**キーワード**：UML，シーケンス図，モデルリファクタリング，不吉な匂い

# An Attempt to Refactor Models of Sequence Diagrams

***Abstract:*** Although there are model refactorings to improve the quality of UML, there is no refactoring approach for sequence diagrams. Thus, to the diagrams, we apply bad smells of source codes that are effective and can be detected automatically in them. In addition, we propose model refactorings to remove the smells for them. To define more effective approach, we also use class diagrams. Besides, we show algorithms to detect bad smells, procedures of model refactorings and that they do not change observable behaviors of a system. By this approach, sequence diagrams with high quality can be made. To an open UML model, we evaluated the validity of the algorithms to detect four bad smells we propose, and confirmed it for three of them. For the other one, we show an other approach for it.

***Keywords:*** UML, Sequence Diagram, Model Refactoring, Bad Smell

## 1. はじめに

システムの設計モデルの品質を向上させる方法としてモデルリファクタリングが挙げられる．モデルリファクタリングとは，外部からみたシステムの振る舞いを変更することなく内部構造を変更し，再利用性や理解容易性を向上させるリファクタリングを，モデルに適用した手法である．モデリング言語 UML (Unified Modeling Language) に対しては，クラス図を始めとして様々な研究がなされている [1] [2] [3]．UML のモデリング対象であるオブジェクト指向は，オブジェクト同士のメッセージのやり取りでシステムの機能を実現するように考えられ，シーケンス図などの相互作用図に記述される．しかしながら，シーケンス図に対するモデルリファクタリングの手法は提案されていない．

そこで本研究では，シーケンス図を対象としたモデルリファクタリングの手法について提案する．より有効なリファクタリングを適用するために，構造モデルであるクラス図も利用する．また，シーケンス図で自動検出できる不吉な匂いについて定義し，どの要素に対してモデルリファクタリングを適用できるかも示す．これにより，シーケンス図の再利用性や理解容易性を向上させることができる．

## 2. 関連知識

### 2.1 UML

UML とは，OMG (Object Management Group) が提案したオブジェクト指向のモデリング言語である [4]．本研究では，静的な構造を表すクラス図と，動的な振る舞いを

[1] 茨城大学
Ibaraki University

表すシーケンス図を対象にする．

### 2.2 モデルリファクタリング

リファクタリングとは，“外部から見たときの振る舞いを保ちつつ，理解や修正が簡単になるように，ソフトウェアの内部構造を変化させること” である．リファクタリングを行う際には，修正したことによって，ソフトウェアの外部から見た振る舞いが変更されていないことを保証しなければならない．Fowler ら [5] はオブジェクト指向のソースコードのために 70 種類のリファクタリングを，また，リファクタリングを適用できる可能性のあるソースコードの部分である，22 種類の不吉な匂いを提案した．

一方モデルリファクタリングとは，ソースコードで有効性が示されたリファクタリングの考えを，モデルに対して応用した手法である．モデルはソースコードよりも抽象度の高い表現であるため，より粒度の大きいリファクタリングを設計段階で適用することが可能となり，設計モデルの再利用性や理解容易性を向上させることができる．

## 3. 提案手法

シーケンス図のモデルに対する不吉な匂いとリファクタリングの手法を提案する．より有効な手法を適用するため，構造モデルからクラスや操作などの情報を参照，または追加，削除する．

本研究では，以下のことを前提に手法を説明する．

(1) 相互作用フラグメントとは，あるライフラインが送信するメッセージ，あるいは関わる複合フラグメント，相互作用の利用のことを指す．
(2) 委譲操作とは，メッセージの送信を 1 度しか行わず，送信先が他のオブジェクトである操作と定義する．
(3) ゲッタは，外部から見たシステムの振る舞いに影響を与えない．

### 3.1 シーケンス図中の不吉な匂い

Fowler らがソースコードに対して提案した不吉な匂いをシーケンス図に適用する [5]．シーケンス図中で自動検出可能な不吉な匂いを次に示す．ただし，重複したコード，長すぎるメソッド，メッセージの連鎖以外の不吉な匂いは，属性の参照にゲッタを利用するなど，一定の規則に従ってシーケンス図が記述されている場合にのみ検出可能である．

(1) 重複したコード
(2) 長すぎるメソッド
(3) 属性，操作の横恋慕
(4) 基本データ型への執着
(5) スイッチ文
(6) 怠け者クラス
(7) 疑わしき一般化
(8) 一時的属性

```
func DuplicatedInteractionFragments ()
  interactionSet = getAllInteractionSet ()
  similarInteractionFragmentSequenceOfSetOfSet =
    getSimilarFragmentSequenceOfSetOfSet (
      interactionSet)
  result = {}
  foreach similarInteractionFragmentSequenceOfSet in
      similarInteractionFragmentSequenceOfSetOfSet
    if evaluate (
          similarInteractionFragmentSequenceOfSet)
      add similarInteractionFragmentSequenceOfSet
        to result
    end if
  end foreach
  return result
end func
```

**図 1** 重複した相互作用フラグメントを検出するためのアルゴリズム
**Fig. 1** An algorithm to detect duplicated interaction fragments

(9) メッセージの連鎖
(10) 仲介人
(11) 不適切な解決

上記の不吉な匂いの中で，本研究では自動検出して効果が期待できるものとして，次の不吉な匂いを対象にする．

(1) 重複した相互作用フラグメント (重複したコードに対応)
(2) 長すぎる操作 (長すぎるメソッドに対応)
(3) メッセージの連鎖
(4) 仲介人

#### 3.1.1 重複した相互作用フラグメント

重複した相互作用フラグメントでは，シーケンス図において類似する複数の相互作用フラグメントが異なる部分で表れることを指す．重複したコードと同様に，一方が変更された時，他方も変更する必要がある可能性が高く，保守性を悪くしている．なお，本研究では複数のライフラインが関わる重複した相互作用フラグメントは取り扱わない．

重複した相互作用フラグメントを検出するためのアルゴリズムを**図 1** に示す． ここで，getAllInteractionSet () は相互作用すべてを取得する関数，getSimilarFragmentSequenceOfSetOfSet (interactionSet) は，相互作用の集合 interactionSet に含まれる，類似した相互作用フラグメントの列を要素に持つ集合の集合を取得する関数，evaluate (similarInteractionFragmentSequenceOfSet) は類似している相互作用フラグメントの列の集合が，不吉な匂いかを判定する関数である．本研究では等価な相互作用フラグメントのみを検出した．また，相互作用フラグメントが等価であるとは，次の条件を満たしていることと定義した．

- メッセージの場合，メッセージが表す操作のシグネチャと，送信先のライフラインが表すオブジェクトの

クラスが一致していること．

- 複合フラグメントの場合，その複合フラグメントの種類が一致し，そのオペランドに含まれる相互作用フラグメントが全て等価であること．
- 相互作用の利用の場合，参照先の相互作用に含まれる相互作用フラグメントが全て等価であること．

また，evaluate 関数の評価方法として，今回は式 (1) を用いる．

$$(ndif \geq MINNDIF) \wedge (\forall cf \in cfSet \\ (nm \times CM + calcCf(cf) > THRESHOLD)) \quad (1)$$

ここで，ndif は重複した相互作用フラグメントが記述されている個数，cfSet はネストされていない複合フラグメントの集合，nm はメッセージの数，calcCf は複合フラグメントのオペランドに含まれる相互作用フラグメントを考慮して値を評価する関数である．

ここで，calcCf の関数を次のように定義した．

$$calcCf(cf) = \forall innerCf \in cf.cfSet \\ (CCF \times (cf.nm \times CM + calcCf(innerCf)))$$

cf は複合フラグメント，cf.nm は，複合フラグメント cf の全てのオペランド中に含まれるメッセージの数，cf.cfSet は複合フラグメント cf 中に含まれる複合フラグメントの集合である．また，MINNDIF，CM，CCF と THRESHOLD は，どの程度の相互作用フラグメントが類似していれば重複していると検出するかを決定する定数である．

#### 3.1.2 長すぎる操作

長すぎる操作では，ある操作を実行するためのメッセージを受信してから，その操作が終了するまでに送信するメッセージの数や，その操作が関わる複合フラグメントの数が多い．そのため，その操作は多くの振る舞いを持っている場合が多く，その一部のみを利用できない問題がある．また，その操作を理解するのが困難になっている．

長すぎる操作を検出するためのアルゴリズムを**図 2** に示す． ここで，interaction.getLifelineSet () は相互作用 interaction に含まれるライフラインを取得するための操作，lifeline.getOperationSet () はライフライン lifeline が持っている操作の集合を取得する操作，evaluate (operation) は操作 operation 中に含まれる相互作用フラグメントから，この操作が長すぎる操作の不吉な匂いであるかを判定する関数である．ただし，図 2 では，同じ操作が複数回出現する場合も，それぞれ判定を行なっている．

また，evaluate 関数は式 (2) を用いる．ただし，calcCf は式 (1) と同じ定義の式を，各定数は異なる値を利用する．

$$\forall cf \in cfSet \\ (nm \times CM + calcCf(cf) > THRESHOLD) \quad (2)$$

```
func LongOperations ()
  result = {}
  foreach interaction in getAllInteractionSet()
    foreach lifeline in
        interaction.getLifelineSet ()
      foreach operation in
          lifeline.getOperationSet ()
        if evaluate (operation)
          add operation to result
        end if
      end foreach
    end foreach
  end foreach
  return result
end func
```

**図 2** 長すぎる操作を検出するためのアルゴリズム
**Fig. 2** An algorithm to detect long operations

#### 3.1.3 メッセージの連鎖

メッセージの連鎖とは，あるオブジェクトから別のオブジェクトにメッセージを送り，取得できたオブジェクトに別のメッセージを送り，そのメッセージによって取得できたオブジェクトにさらに別のメッセージを送る，といったシーケンスが起こっている操作に対する不吉な匂いである．これは送信先のオブジェクトが内部構造をカプセル化していないため，送信元のオブジェクトと関連のないオブジェクトとの結合度が強くなっている．

メッセージの連鎖を検出するためのアルゴリズムを**図 3** に示す．図 3 に示すように，この不吉な匂いを検出するためには，相互作用中に表れるクラスの集合が必要となる．getClass (classSet, lifeline) はクラスの集合 classSet から，ライフライン lifeline が表すクラスを取得する関数，lifeline.getOperationSet () はライフライン lifeline 中で実行されている操作の集合を取得するための操作，evaluate (class, operation) は，クラス class の操作 operation がメッセージの連鎖かを判定する関数である．

なお，メッセージの送信先のオブジェクトが他のオブジェクトから取得したことを，本研究では，送信先のオブジェクトが，送信元のオブジェクトの属性に含まれないことで判定している．また，evaluate 関数は，そういった一時オブジェクトの数が一定値以上であることでメッセージの連鎖と判定している．

#### 3.1.4 仲介人

仲介人とは，あるオブジェクトから別のオブジェクトに対してメッセージを委譲する操作を，多く持っているオブジェクトのことである．これは必要以上にカプセル化した結果であり，本来依存関係にあるべきオブジェクト同士の関係がなく，オブジェクトの関係が不適切となっている．

仲介人を検出するためのアルゴリズムを**図 4** に示す． こ

```
func MessageChains (classSet)
  result = {}
  foreach interaction in getAllInteractionSet ()
    foreach lifeline in
        interaction.getLifelineSet ()
      class = getClass (classSet, lifeline)
      foreach lifelineOperation in lifeline
          .getOperationSet ()
        if evaluate (class, lifelineOperation)
          add operation to result
        end if
      end foreach
    end foreach
  end foreach
  return result
end func
```

**図 3** メッセージの連鎖を検出するためのアルゴリズム
**Fig. 3** An algorithm to detect message chains

```
func MiddleMen (classSet)
  result = {}
  foreach class in classSet
    operationSet = getOperationSet (class)
    if evaluate (operationSet, class)
      add class to result
    endif
  end foreach
  return result
end func
```

**図 4** 仲介人を検知するためのアルゴリズム
**Fig. 4** An algorithm to detect middle men

こで，getOperationSet (class) は，全ての相互作用中に含まれるクラス class の操作を取得するための関数，evaluate (operationSet, class) はクラス class と，シーケンス図中で使用されている操作の集合 operationSet を元に仲介人かを判定する関数であり，本研究では式 (3) の条件全てを満たすクラスと判定する．

$$\begin{cases} ndops/|ops| \geq MINRATE \\ ndops \geq MINDOPS \end{cases} \tag{3}$$

where

$$ndops = |\{dop | dop \in ops, \\ dop.nsm > 0 \wedge dop.nsm \leq MAXDNSM\}|$$

ここで, ops は該当するクラスが持つ操作の集合, dop.nsm は操作 dop が送信するメッセージの数である．また，MINRATE，MINDOPS，MAXDNSM はそれぞれクラスが持つ委譲操作の割合，委譲操作の最小数，委譲操作が送信するメッセージの最大数で，MAXDNSM は本研究では 1 である．ここでは操作が委譲しているかどうかは送信するメッセージ数のみで判定しているが，生成メッセージは含めない．これは，Abstract Factory パターンとの競合を防ぐためである． [*1]

### 3.2 シーケンス図に対するモデルリファクタリング

各リファクタリングについて次のことを説明する．

(1) パラメータ
(2) 事前条件
(3) 事後条件
(4) リファクタリング手順と外部から見た振る舞いを変更しないことの証明

パラメータは，そのリファクタリングを適用するために必要な情報であり，事前条件，事後条件はリファクタリングによって外部から振る舞いが変更されないために満たされているべき条件である．なお，事前条件，事後条件が複数存在した場合，それら全ての条件を満たす必要がある．

シーケンス図に対するリファクタリングは，内部の振る舞いを変更するかしないかで分類できる．提案するリファクタリングのうち，内部の振る舞いを変更しないリファクタリングは相互作用の抽出のみで，他のリファクタリングは振る舞いを変更するリファクタリングに分類される．

なお，内部の振る舞いを変更しないリファクタリングでは，外部から見たシステムの振る舞いが変更しないことの証明はしない．

#### 3.2.1 操作の抽出

操作の抽出は，ある操作中の相互作用フラグメントの列を，他の操作として作成するリファクタリングである．

シーケンス図上では，別の操作を抽出しても記述量は増え，簡潔にはならない．しかし，複数の相互作用フラグメントを一つにまとめ，それらがどのような処理をしているのかを把握することが可能になる．また，操作の抽出を行った後に，操作の移動や操作の引き上げなどのリファクタリングを行うことも可能になる．

(1) パラメータ
- 操作を抽出するクラス
- 抽出する操作の引数, 戻り値の型を含めたシグネチャ
- 新しく作成される操作の抽出元の操作
- 抽出する相互作用フラグメントの列

(2) 事前条件
  (a) 構造モデルにおいて，抽出する操作を持つクラスとその先祖クラス，子孫クラスが既に同じ操作を所有していない．
  (b) 抽出元の操作を抽出元のクラスが所有している．
  (c) 抽出元の操作が，抽出する相互作用フラグメントを所有していない．

[*1] Abstract Factory パターンとは，あるグループ中のオブジェクトを生成するインタフェースを持つデザインパターンである [6].

(3) 事後条件
  (a) 構造モデルにおいて，操作を抽出したクラスが，新たに作成した操作を持っている．
(4) リファクタリングの手順と，外部から見た振る舞いを変更しないことの証明
  (a) 相互作用の利用の参照先に存在する相互作用フラグメントを含めて操作を抽出する必要がある場合は，その相互作用を参照している全ての相互作用で，参照先のモデルをインライン化する．
    [証明] 参照している相互作用の内部の動作は，インライン化する前とした後で変化しない．このため，システムの振る舞いは変更されない．
  (b) 構造モデルにおいて，指定したクラスに新しく操作を追加する．
    [証明] 新しく空の操作を作成することは，モデルの振る舞いに影響を与えない．
  (c) 抽出する相互作用フラグメントを新しい操作に追加する．
    [証明] 新しい操作は，どこからも参照されていないため，システムの振る舞いを変更しない．
  (d) 抽出する相互作用フラグメントを，新しく作成した操作に置き換える．
    [証明] 新しく作成した操作は，メッセージの受信後，置き換える相互作用フラグメントと同じ振る舞いをする．このため，システムの振る舞いは変更されない．

#### 3.2.2 委譲の隠蔽

委譲の隠蔽とは，あるクライアントクラスから委譲クラスへのメッセージ送信を，クライアントクラスからサーバクラスを仲介させて委譲クラスにメッセージを送信する流れに置き換えるリファクタリングである．これはクライアントクラスから委譲クラスへのメッセージ送信の前に，委譲クラスを取得するためにサーバクラスに対してクライアントクラスがメッセージを送信している場合に有効である．

(1) パラメータ
- クライアントクラス
- サーバクラス
- 委譲クラス
- 隠蔽する操作の列
- サーバの委譲操作の列
- 隠蔽の関係の集合
  隠蔽の関係の集合は，ある相互作用にクライアント，サーバ，委譲クラスのオブジェクトが存在し，かつサーバオブジェクトが複数存在した場合にのみ必要である．どのクライアントクラスのオブジェクトが，どの委譲クラスのオブジェクトに対して，どのサーバクラスのオブジェクトを仲介させるか，といった関係を表す．

(2) 事前条件
  (a) クライアント，サーバ，委譲クラスが全て異なる．
  (b) 隠蔽する操作の列は，全て委譲オブジェクトのクラスが所有している．
  (c) サーバの委譲操作と同じシグネチャの操作をサーバが持っていない．
  (d) 隠蔽する操作の列とサーバの委譲操作の列の要素数が同じである．
  (e) 隠蔽の関係の集合に含まれるオブジェクトは，全てクライアント，サーバ，委譲クラスである．
(3) 事後条件
  (a) サーバオブジェクトが委譲操作を所有している．
(4) リファクタリングの手順と，外部から見た振る舞いを変更しないことの証明
  (a) 全ての隠蔽する操作と，サーバの委譲操作に対して，次の処理を追加する．
    (i) 構造モデルにおいて，サーバクラスに新たに委譲操作を追加する．
      [証明] 追加した委譲操作は使用されていないため，システムの振る舞いを変更しない．
    (ii) 全ての相互作用中に存在するクライアント，サーバ，委譲クラスを検索する．クライアントクラスのオブジェクトから委譲クラスのオブジェクトに対して，隠蔽する操作に含まれる操作を実行するためのメッセージ送信があった場合，そのメッセージ送信を委譲クラスのオブジェクトを取得したサーバクラスのオブジェクトを経由したメッセージ送信に置き換える．
      [証明] 変更前の委譲クラスのオブジェクトに対するメッセージ送信は，サーバクラスのオブジェクトを介して送信される．サーバの操作は新しく作成したため，委譲クラスのオブジェクトに対するメッセージ送信以外に振る舞いはない．そのため，システムの振る舞いは変更されない．
  (b) クライアントオブジェクトから委譲オブジェクトに対するメッセージ送信が他にない場合，置き換えたメッセージの前で利用されている，委譲オブジェクトをサーバオブジェクトから取得するためのゲッタを削除する．
    [証明] クライアントオブジェクトから委譲オブジェクトに対するメッセージ送信は，全てサーバオブジェクトが委譲するため，クライアントオブジェクトは委譲オブジェクトを取得しても，オブジェクトに対してメッセージの送受信を行わない．そのため，ゲッタを削除してもシステムの振る舞いは変更されない．

#### 3.2.3 仲介人の削除

3.1.4 節で示した仲介人の不吉な匂いを取り除くためのリファクタリングである．

(1) パラメータ
- クライアントクラスの集合
- 仲介人のクラス
- 委譲クラス
- 委譲先のオブジェクトを取得するゲッタのシグネチャ
- 委譲操作の集合

(2) 事前条件
- (a) クライアントクラス，仲介人のクラス，委譲クラスが全て異なる．
- (b) 仲介人のクラスが，委譲操作の集合を持っている．
- (c) 仲介人が委譲クラスのオブジェクトを返すためのゲッタを持っていない．
- (d) 委譲操作の集合はメッセージの委譲のみを行う．
- (e) クライアントクラスは，仲介人が仲介している委譲クラスの操作に対してメッセージが送信できる．

(3) 事後条件
- (a) 仲介人が委譲操作を所有していない．
- (b) 仲介人がゲッタを所有している．

(4) リファクタリングの手順と，外部から見た振る舞いを変更しないことの証明
- (a) 構造モデルにおいて，仲介人にゲッタを追加する．
  [証明] 追加されたゲッタは使用されていないため，外部システムからの振る舞いを変更しない．
- (b) クライアントオブジェクトが仲介人に委譲メッセージを送信する前に，クライアントオブジェクトから仲介人にゲッタのメッセージを送信する．
  [証明] ゲッタはメッセージを送信せず，取得したオブジェクトは使用されないため，外部から見たシステムの振る舞いは変更されない．
- (c) 仲介人を経由したクライアントオブジェクトから委譲オブジェクトへのメッセージ送信を，クライアントオブジェクトから委譲オブジェクトへの直接のメッセージ送信に置き換える．
  [証明] 元々の委譲メッセージと新しいメッセージは，委譲オブジェクトに同じメッセージを送信しており，それ以外の振る舞いをしないため，システムの振る舞いは変更されない．

#### 3.2.4 相互作用の抽出

相互作用では，別の相互作用を相互作用の利用により参照することができる．それを利用して，相互作用中に含まれる相互作用フラグメントを別の相互作用に移動させるリファクタリングである．これによりメッセージのやり取りを再利用することができるようになる．

(1) パラメータ
- 新しく作成する相互作用の名前
- 抽出する相互作用フラグメントの列
- ライフラインの集合

(2) 事前条件
- (a) 抽出する相互作用の名前と同じ名前の相互作用が存在しない．

(3) 事後条件
- (a) 抽出した相互作用が存在する．
- (b) 抽出した相互作用中に，指定したライフラインのみが存在する．
- (c) 抽出した相互作用中に，指定した相互作用フラグメントのみが利用されている．

(4) リファクタリングの手順
- (a) 相互作用を作成する．
- (b) 作成した相互作用にライフラインをコピーする．
- (c) 作成した相互作用に抽出する相互作用フラグメントの列をコピーする．
- (d) 抽出した相互作用フラグメントの列が表れる全ての相互作用で，その列を相互作用の利用に置き換え，参照先を作成した相互作用にする．

### 3.3 不吉な匂いとリファクタリングの対応関係

**表 1** に，本研究で提案した不吉な匂いと，それを取り除くためのリファクタリングの対応関係を示す．

**表 1** 不吉な匂いとリファクタリングの対応関係
**Table 1** Bad smells and corresponding refactorings

| 不吉な匂い | リファクタリング |
|---|---|
| 重複した相互作用フラグメント | 相互作用の抽出 |
| | 操作の抽出と相互作用の抽出 |
| 長すぎる操作 | 相互作用の抽出 |
| | 操作の抽出と相互作用の抽出 |
| メッセージの連鎖 | 委譲の隠蔽 |
| 仲介人 | 仲介人の削除 |

## 4. 実験

### 4.1 実験概要

本手法で提案した不吉な匂いを検出するアルゴリズムの妥当性を確認するための実験を行う．実験を評価するために，UML モデルから不吉な匂いを検出するためのツールを開発し，不吉な匂いを検出するためのクエリは形式言語 OCL で定義した [7]．また，今回実験に使用した UML モデルは，Eclipse 上のプラグインであるモデリングツール Papyrus で作成した [8].

実験の対象とするモデルは，Web 上で公開されている飛行船制御システムの PSM モデルである [9]. [*2]ただし，実験のために Papyrus を利用して同じモデルを作成したが，

[*2] PSM とは，ソフトウェアの動作環境を考慮したモデルである．

**表 2** 各不吉な匂いのパラメータの値
**Table 2** Values of the parameters of each bad smell

| 不吉な匂い | パラメータ | 検出するための値 | 検出できないことを確認するための値 |
|---|---|---|---|
| 重複した相互作用フラグメント | MINNDIF | 2 | 2 |
| | CM | 1 | 1 |
| | CCF | 2 | 2 |
| | THRESHOLD | 1 | 2 |
| 長すぎる操作 | CM | 1 | 1 |
| | CCF | 1 | 1 |
| | THRESHOLD | 10 | 11 |
| メッセージの連鎖 | 一時オブジェクトの数 | 1 | 2 |
| 仲介人 | MINRATE | 0.3 | 0.4 |
| | MINOPS | 10 | 10 |
| | MAXDNSM | 1 | 1 |

元のモデルに対して次の変更を加えた．

- 他の相互作用を参照している操作を，相互作用の利用を使って記述した．
- ガード条件が付加されたメッセージ送信は，その条件をメッセージから取り除いて，その条件を持つ alt か opt の複合フラグメント中に記述した．
- メッセージの戻り値を記憶するための変数は記述していない．

このモデルの各不吉な匂いの特徴を抽出した．その特徴は次の通りである．

(1) 重複した相互作用フラグメント
- 相互作用 startTestFlight と flight 中の操作 getSensorInfo，setSensorInfo のメッセージ送信が重複している．

(2) 長すぎる操作
- クラス Blimp の操作 startAutomaticFlight がメッセージを 11 回送信している．

(3) メッセージの連鎖
- クラス BlimpSystem の操作 setUSSensors と setFlightPath が他のオブジェクトからオブジェクトを 1 つだけ取得している．

(4) 仲介人
- クラス Blimp と FlightController の操作数は 13 で，それぞれ委譲操作を 5 つ持っている．それ以外のクラスは操作数を 10 個以上は持っていない．

抽出した特徴を元に， 3.1 節で示した各不吉な匂いの定数を決定した．各定数の値を**表 2** に示す．ここで，定数は上記のモデル要素のみが予想通り検出できるか，あるいは検出できないかを確認できるように決定した．

### 4.2 実験結果

不吉な匂いを検出できない定数に設定した場合は，予想通りどの不吉な匂いも検出されなかった．しかし，定数を検出する値に設定した場合，上記で挙げたモデルの要素を検出できたが，**図 5** に示したクラス AltitudeData, DirectionData, PositionData の操作 write もメッセージの連鎖と判定された．これらのクラスは，クラス SimpleLock との関連はあるが，その関連端名は lock である．

### 4.3 考察

実験結果から，メッセージの連鎖を除き提案した不吉な匂いを検出するアルゴリズムの妥当性を確認できた．メッセージの連鎖と誤検出された操作を含む，図 5 に示したシーケンス図中では，同じクラスが他の複数のクラスと関連を持ち，その関連端名が同じであり，かつそれらのオブジェクトを 1 つのシーケンス図に記述するために，実際とは異なったオブジェクト名を使用したと考えられる．このようなシーケンス図に対応するためには，属性名ではなく，属性のクラスを判定することで防ぐことができる．しかしながら，この方法ではあるクラスやインタフェースと関連を持っている場合，その子孫クラスを一時オブジェクトと判定するかを考慮する必要がある．

## 5. 関連研究

クラス図を対象として，Stolc らは不吉な匂いとモデルリファクタリングをグラフィカルな環境で定義するための手法を説明している [10]．また，Astels は UML を利用してソースコードのリファクタリングを可視化している [1]．ここではクラス図に対する不吉な匂いとリファクタリング，シーケンス図に対する仲介人の不吉な匂いのみを述べている．また, Einarsson らはリファクタリング適用後のモデルとダイアグラムの一貫性を保つことの有用性を，アクティビティ図に対してリファクタリングを適用して示している [2]．開発の初期のモデルのために，Arendt らはクラス図，ユースケース図，ステートマシン図中の不吉な匂いとリファクタリングを定義し，それらがモデルのどのような品質と関わっているのかを定性的に評価している [3]．

また，リファクタリングの手法の 1 つとしてデザインパターンの導出が挙げられる．増田らは，デザインパターンが適用できる箇所を探し出し，Observer パターンを適用した例を示している [11]．また，El-Sharqwi らは，デザインパターンを適用できる箇所を示す問題の仕様，デザインパターンの構造を表すターゲットの仕様，そして変換を表す変換仕様の 3 つを説明し，Abstract Factory パターンの適用例を示している [12]．

これらの手法と本研究を組み合わせることで，UML モデルの様々なビューに対してより有効なリファクタリングの適用が可能になると考えられる．なお，El-Sharqwi らは，抽象度の低い, メソッドの抽出のようなリファクタリングはモデルに対しては適用できないと述べているが，振

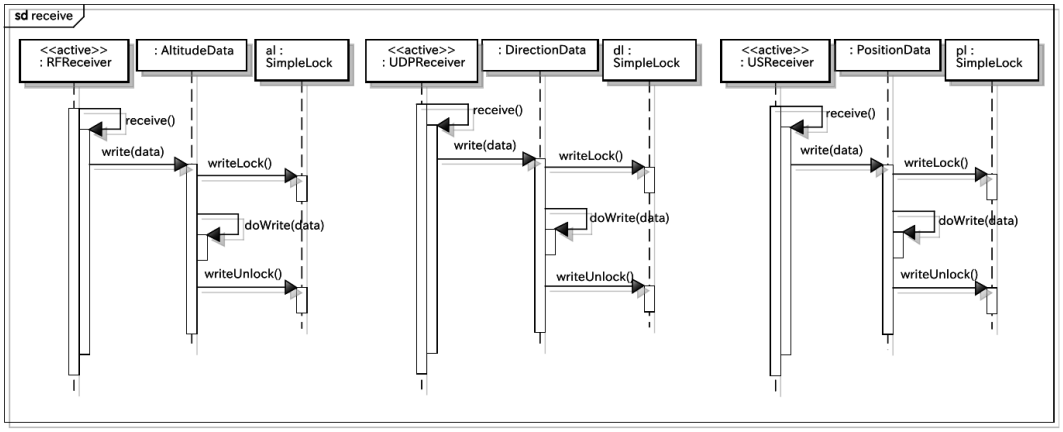


**図 5** メッセージの連鎖と誤検出されたモデルを含むシーケンス図
**Fig. 5** A sequence diagram which contains models that were incorrectly detected as message chains

る舞いモデルを考慮した場合そのリファクタリングは適用できると考え，本研究で提案した．

## 6. おわりに

本研究では，UML シーケンス図の再利用性や理解容易性を向上させるため，構造のモデルも考慮したモデルリファクタリングの手法を提案した．また，従来ソースコードに対して適用されてきた不吉な匂いをシーケンス図に対しても適用し，リファクタリングが必要なモデルの要素を検出するために使用できることを示した．実際に公開されているモデルに対して不吉な匂いを適用し，提案した不吉な匂いのアルゴリズムの妥当性について確認したが，メッセージの連鎖では誤検出されたモデルの要素もあり，誤検出されたモデルを考慮した別の方法について提案した．

本研究で提案した，不吉な匂いを自動で検出して，そのモデルの要素に対してリファクタリングを適用する手法により，シーケンス図のモデルの品質を高めることが可能となる．また，不吉な匂いは，モデリングのガイドラインとして利用することも可能である．リファクタリングについては事前，事後条件のチェックや適用は現在手動で行わなければならないが，今後自動化して評価する予定である．

また，今後の課題として，次の点などが挙げられる．

- 本研究で考慮しなかった，複合フラグメントの種類や同期メッセージ以外のメッセージに対応する．
- レイアウトやデザインパターンとの関係を示す．
- モデルに対するメトリクスにより品質を測定し，リファクタリングの効果を定量的に評価する．

### 参考文献

[1] Astels, D.: Refactoring with UML, Proc. 3rd Int ’l Conf. eXtreme Programming and Flexible Processes in Software Engineering, pp. 67–70 (2002).
[2] Einarsson, H. T. and Neukirchen, H.: An Approach and Tool for Synchronous Refactoring of UML Diagrams and Models Using Model-to-model Transformations, *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT ’12, New York, NY, USA, ACM, pp. 16–23 (2012).
[3] Arendt, T. and Taentzer, G.: UML Model Smells and Model Refactorings in Early Software Development Phases, *Universitat Marburg* (2010).
[4] OMG: OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, Object Management Group (online), available from ⟨http://www.omg.org/spec/UML/2.4.1⟩ (accessed 2013-01-09).
[5] Fowler, M.: 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，ピアソンエデュケーション (2000).
[6] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).
[7] OMG: OMG Object Constraint Language (OCL), Object Management Group (online), available from ⟨http://www.omg.org/spec/OCL/2.3.1⟩ (accessed 2013-04-15).
[8] The Eclipse Foundation: Papyrus, The Eclipse Foundation (online), available from ⟨http://www.eclipse.org/papyrus/⟩ (accessed 2014-01-26).
[9] 立命館大学情報理工学情報システム学科ソフトウェア基礎技術研究室：MDD ロボットチャレンジ 2005，立命館大学（オンライン），入手先 ⟨http://www.fse.cs.ritsumei.ac.jp/mdd/⟩（参照 2014-02-16）.
[10] Stolc, M. and Polasek, I.: A Visual Based Framework for the Model Refactoring Techniques, *Applied Machine Intelligence and Informatics (SAMI), 2010 IEEE 8th International Symposium on*, pp. 72–82 (2010).
[11] 増田敬史，吉田則裕，浜口 優，井上克郎：UML モデルを対象としたリファクタリング候補検出の試み，電子情報通信学会技術研究報告. KBSE, 知能ソフトウェア工学, Vol. 108, No. 65, pp. 25–30 (2008).
[12] El-Sharqwi, M., Mahdi, H. and El-Madah, I.: Pattern-Based Model Refactoring, *Computer Engineering and Systems (ICCES), 2010 International Conference on*, pp. 301–306 (2010).