Vol.2009-SE-166 No.2
2009/11/5

同時変更が生じたTemplate Methodパターンの適用事例の調査

齋藤 晃，吉田 則裕，松下 誠，井上 克郎

デザインパターンはプログラム設計時に生じる問題の解決策をまとめたものであり，プログラムの保守性の向上に役立つ．しかし，デザインパターンの 1 種である Template Method パターンにおいて，Template Method パターンが必要で無い状況でプログラムに適用された場合，同時変更 (1 つの変更要求を満足するために行う，複数部分に渡る変更) が生じ，かえってプログラムの保守性が低下することがある．本稿では，このような同時変更が生じる原因は Template Method パターンの実装の差異によると考え，調査を行った．調査の結果，3 つのオープンソフトウェアのうち 2 つが，これらの差異と関連があることが分かった．

An Investigation of Simultaneously Changed Programs Using Template Method Pattern

AKIRA SAITO, NORIHIRO YOSHIDA, MAKOTO MATUSHITA, KATSURO INOUE

Design patterns are general repeated solutions to common problems in object-oriented design. Developers can design software systems easily by applying design pattern. However, when we use template method pattern, which is one of the design patterns, simultaneously changes occur if template method pattern is unsuitable to apply problems. In this paper, we focused on cause of simultaneous changes. we assumed the cause is differences of implementation of template method pattern. As a result, these differences showed the relationship with simultaneous changes in two of three open source software.

†1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

1. はじめに

デザインパターンとは，プログラム設計時に生じる典型的な問題の解決策を名前を付けてまとめたものであり，解決すべき問題に対し，有効な解法，その効果が文書化されている．とりわけ，Gamma らが示した 23 種類のデザインパターン[2] が広く知られている．

デザインパターンを用いることで，ソフトウェア開発における熟練者が考えた解決策を非熟練者が使用できるため，ソフトウェアの品質が向上することがある．実際に，デザインパターンはソフトウェア開発の生産性の向上に寄与していた事例が報告されている[6]．

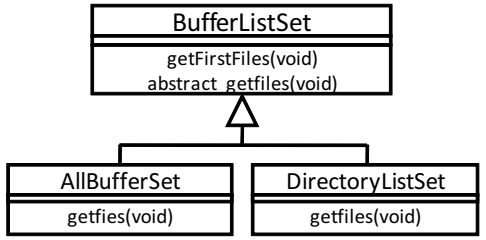
しかし，デザインパターンが適用可能な問題であっても，デザインパターンの適用により必ずしもプログラムの品質が向上するとは限らない．そこで本稿では，頻繁に適用されるデザインパターンの 1 つである Template Method パターン[2] に着目する．Template Method パターンに着目する理由として，

- パターンの構造が比較的単純であり，静的なクラス構造を分析することで検出可能であること
- Factory Method パターンや Strategy パターンなど，他のパターンの一部として使われていること

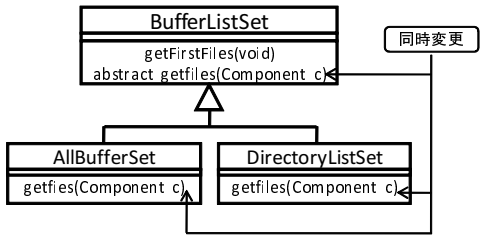
が挙げられる．

Template Method パターンを用いた設計では，1 つのアルゴリズムを 1 つの Template Method と 1 つ以上の Primitive Operation に分割する．Template Method は，アルゴリズムの各ステップの実行順序を表し，Primitive Method は各ステップを表す．典型的には，抽象クラスにおいて Template Method の実装と Primitive Operation の抽象メソッドを宣言し，複数の子クラスで Primitive Operation を実装する．図 1 では Template Method パターンを適用したために，保守性が低下すると考えられる場合を示す．図 1 中のメソッド *getFirstFiles* が Template Method，メソッド *getfiles* が Primitive Operation に相当する．この例では，バージョン変更によって，メソッド *getfiles* の引数が void から Component に変更された場合を示している．この様な変更は，Template Method パターンに関連するすべてのクラスに影響を及ぼしてしまう．図 1 の変更のように 1 つの変更要求を満足させるために，複数の部分を変更する必要が生じるプログラムは，保守性が低いプログラムと考えられる．

デザインパターンを適用したプログラムの保守性を正確に評価するためには，図 1 のような**同時変更**（1 つの変更要求を満足するために行う，複数部分に渡る変更）が行われやすい部分を特定できることが望ましい．しかし，各デザインパターンを適用したプログラムの



(a) 変更前のクラス



(b) 変更後のクラス

図 1 同時変更が生じた Template Method パターンの適用事例の例
Fig. 1 Example of simultaneously changed programs using Template Method Pattern

中で，同時変更が行われやすいものが持つ特徴について調査した研究は，我々が知る限り存在しない．

本稿では，Template Method パターンに着目し，Template Method パターンが適用されたプログラムの中で，同時変更が行われやすいものが持つ特徴を調査した．調査では，Primitive Operation の実装の間に差異が大きいほど，具象クラスおよび抽象クラスが同時変更されやすいと仮定し，調査を行った．本調査を行うために，Primitive Operation の実装間の差異を表すメトリクスを定義し，差異が大きい Primitive Operation を持つ適用事例ほど同時変更がされやすい傾向にあるかどうかを確認した．そのために，3 つのオープンソースソフトウェアについて調査を行ったところ，2 つのオープンソースソフトウェアについては，その傾向があった．また実際に発生した同時変更が，Primitive Operation の実装間の差異に起因していることを確認した．

2. 背　　景

2.1 デザインパターン

デザインパターンとはオブジェクト指向設計において生じる問題の解法を集めたものであり，頻繁に用いられるデザインパターンの代表例として，Template Method パターンが挙げられる．Template Method パターンは図 1 のような 1 つの抽象クラス (Abstract Class) が複数の具象クラス (Concrete Class) を継承する．それぞれのクラスは以下の役割を持つ[9]．

抽象クラス　抽象化された Primitive Operation を定義する．このメソッドは子クラスで定義される．また，処理の順序を定義する Template Method を定義する．Template Method は Primitive Operation を呼び出し，処理の共通な部分を実装する．

具象クラス　Primitive Operation を実装し，各処理の可変な部分を定義する．

可変な処理を修正・追加したい時は，新たに子クラスを追加・修正するだけで良く，親クラスや他の子クラスに影響を与えない．この様に，修正・追加に対して既存のクラスに影響する範囲を小さくできるので，保守性が向上するとされる．

2.2 適用事例の検出

既存のプログラム内にどのようなデザインパターンが適用されているかが分かれば，プログラム構造や処理内容の理解が容易になる．プログラムの理解を目的として，デザインパターンの適用事例を検出する研究が行われている[3)7)]．現在公開されているデザインパターンの適用事例を検出するツール (以降，適用事例検出ツールと書く) として，PINOT[8)] や Tsantalis らの適用事例検出ツール[10)] がある．Tsantalis らの適用事例検出ツールは，Java バイトコードからそれぞれのクラスの構成やオブジェクト間の参照関係などの類似性を計測することによって，10 種類のデザインパターンを検出できる．この適用事例検出ツールは検出結果の適合率が高いことが知られている．実際に，評価実験として JHotDraw，JRefactory，JUnit の 3 種類のオープンソースソフトウェアに対して Template Method パターンの検出を行っているが，いずれも適合率は 100%であったと報告されている．

2.3 適用事例の変更と問題点

デザインパターンは特定の問題を解決するための解法を与えるが，ソフトウェア進化の過程でデザインパターンを適用したプログラムの構造が変化する場合，同時変更が生じる場合がある．Template Method パターンにおいては，以下の場合に同時変更が生じる．

- Primitive Operation の定義の変更
- Primitive Operation の追加もしくは削除

- 適用事例が Template Method パターンの構造を持たなくなる

1 節の図 1 のように， Primitive Operation の引数が変化した場合，Template Method に関連するすべてのクラスを変更する必要がある．この様な複数のクラスの同時変更は，保守性の観点で望ましくないと言える．

3. 調 査 内 容

本節では，Template Method パターンの同時変更を検出する方法と，同時変更を推定するためのメトリクスについて述べる．

3.1 仮　　説

本節では，デザインパターンの適用事例が変更され，前節で示した同時変更が生じる要因の調査方法を説明する．このような同時変更が生じる要因として，以下の仮説を提案し，これを検証する．

仮説　Primitive Operation の処理内容の差異が大きいほど，同時変更が生じやすい

この仮説を設定した理由は，Primitive Operation の処理差異が大きければ，変更要求が親クラスまで影響しやすく，同時変更が生じやすいと考えたからである．

3.2 調査手法の概要

全体の処理概要を図 2 に示す．調査の手順は以下の 6 つに分けられる．

Step1. ソースコードとバイトコードを入手
Step2. Template Method パターンの適用事例を検出
Step3. クラス情報の抽出
Step4. 変更の有無による分類
Step5. Primitive Operation の処理の差異を表すメトリクスの計測
Step6. 変更の有無とメトリクス値との関係を調査

以下，上記の Step1 から Step6 までの処理内容をそれぞれ説明する．

3.2.1 ソースコードとバイトコードを入手

SourceForge などのオープンソースソフトウェアのリポジトリ公開 Web サイトからソースコードと Java バイトコードを入手する．入手した Java バイトコードは適用事例検出ツールの使用時に使用し，ソースコードはメトリクス計測時に使用する．Java バイトコード・ソースコードは共にリリースバージョンごとに入手する．

3.2.2 Template Method パターンの適用事例を検出

Tsantalis らの適用事例検出ツールを用いてオープンソースソフトウェアから Template

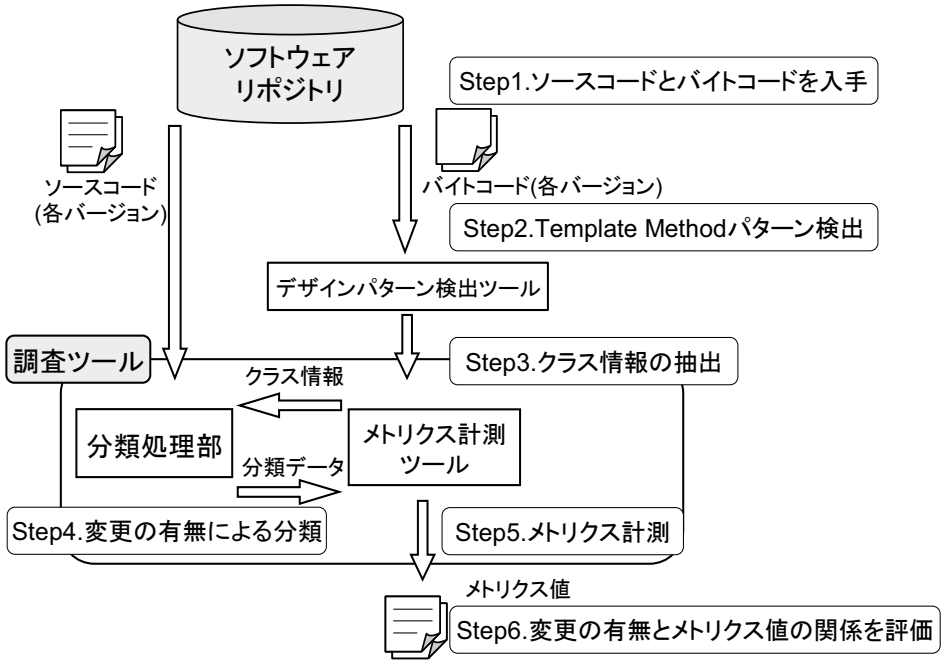


図 2　調査手法全体の流れ
Fig. 2　Overview of our investigation

Method パターンの適用事例を検出する．またツールの出力結果から，適用事例の抽象クラスのクラス名とパッケージ内の位置を取得する．

3.2.3 クラス情報の抽出

Step2 で用いた適用事例検出ツールは，抽象クラスの名前と抽象クラスが所属するパッケージ名のみを検出する．これらの情報だけでは Template Method パターンに関連するクラスが完全に特定できないため，メトリクス計測ツール[5)11)] のソースコード解析機能を用いて以下の情報を Template Method パターンの適用事例から取得する．

- 子クラスのクラス名とパッケージ内の位置
- Template Method の役割を持つメソッドの数とメソッド定義
- Primitive Operation の役割を持つメソッドの数とメソッド定義

1 つのクラスには複数の Template Method パターンの適用事例が存在することがあるため，Template Method の役割を持つメソッドの数を取得する必要がある．同様に，1 つの Template Method パターンには複数の Primitive Operation が存在することがあるため，Primitive Operation の数を取得する必要がある．

3.2.4 変更の有無による分類

検出されたデザインパターンの適用事例に同時変更が生じた適用事例とそうでないものに分類する．同時変更が生じたかどうかを判定するには，リリースバージョン間で対応する Primitive Operation に変化がないことを調べればよい．Primitive Operation はクラス名，Template Method 名，Primitive Operation のメソッド定義の３つの情報から一意に識別できるため，バージョン間で以下のことがすべて成り立てば，同時変更は生じていないと判定できる．

条件 1. 同一のクラス名を持つクラスで定義されている．
条件 2. 同一のメソッド名を持つ Template Method から呼び出されている．
条件 3. 同一のメソッド定義を持つ．

ここで，同一のメソッド定義を持つとは，Primitive Operation が同一のメソッド名，同一の引数をとるということを指す．図 3 は 2 つのバージョン間で，同時変更の生じたパターンの適用事例とそうでない適用事例を分類するアルゴリズムである．

このアルゴリズムでは，以下に示す３つの関数を含む．

$detect(V)$　特定のバージョンのクラス群 V から，適用事例検出ツールを用いて，Template Method パターンを適用したプログラムとそれらを一意に識別する情報を抽出する．

$extractByClassName(D, d)$　Template Method パターンを適用したプログラム群 D から，

```
1:  input V1, V2 ; 各バージョンのクラスの集合
2:  return C, NC ; 同時変更が生じたクラスの集合, 同時変更が生じていないクラスの集合
3:  d := (Cname, T, P)；Primitive Operation
4:  Cname：クラス名
5:  T：Template Method 名
6:  P ：Primitive Operation のメソッド定義
7:  begin
8:    D1 ← detect(V1), D2 ← detect(V2)
9:   for all d ∈ D1 do
10:    D2' ← extractByClassName(D2, d)
11:    if |D2'| = 0 them
12:      NC ⇐ d ; クラス消滅
13:      continue
14:    end if
15:    D2'' ← extractByTemplateName(D2', d)
16:    if|D2''| = 0
17:      C ⇐ d ; TemplateMethod 消滅
18:      continue
19:    end if
20:    dP' ← extractByPrimitiveName(D2'', d)
21:    if|D2'''| = 0
22:      C ⇐ d ; PrimitiveOperation 消滅
23:      continue
24:    end if
25:    ifDiff(dP, dP') = true
26:      C ⇐ d ; PrimitiveOperation の定義が変更
27:      continue
28:    end if
29:    NC ⇐ d ; 変更なし
30:  end for
31: end
```

図 3　同時変更を検出するアルゴリズム
Fig. 3　Algorithm of simultaneous change detection

指定されたクラス名を持つプログラム群 d のみ抽出する．アルゴリズム中の extractByTemplateName, extractByPrimitiveName も同様に，指定された Template Method を持つプログラム群と指定された Primitive Method のメソッド定義をもつプログラムのみ抽出する．

$Diff(d_{P1}, d_{P2})$　2 つの Primitive Operation のメソッド定義の差異を検出する．メソッドの引数と戻り値の方が同一であれば，Diff は false を返す．そうでなければ true を返す．

3.2.5 Primitive Operation の処理内容の差異を表すメトリクスの計測

変更の生じたパターンの適用事例とそうでない適用事例に対してメトリクスを計測する．ここでは，Primitive Operation の処理の差異を表すために 2 種類のメトリクスを提案する．

識別子名類似度 (IS)　m 個の子クラスのそれぞれの Primitive Operation 内で定義されている一時変数・参照変数の識別子名と，メソッド内で呼び出している関数名を含む集合を I_i とし，識別子名類似度 IS を以下のように定義する．ここで，$|I_i|$ は集合 I_i の要素数を表す．

$$IS = \frac{\left| \bigcap_{i=1}^{m} I_i \right|}{\left| \bigcup_{i=1}^{m} I_i \right|} \tag{1}$$

識別子類似度 IS の計算例を図 4 に示す．図 4 の SubClass1 のメソッド中に出現する識別子名は *zs*, *encoding*, *defaultEncoding*, *setEncoding* であり，SubClass2 のメソッド中に出現する識別子名は *zs*, *efaultEncoding* である．これらの共通する要素の割合が識別子名類似度として計算される．この定義より，識別子名類似度は子クラス同士で共通した識別子名がなければ値が小さくなる．

型名類似度 (TS)　m 個の子クラスのそれぞれの Primitive Operation 内で定義されている一時変数・参照変数の型名と，メソッド内で呼び出されている関数の引数，戻り値の型名を集合 T_i とし，型名類似度 TS を以下のように定義する．

$$TS = \frac{\left| \bigcap_{i=1}^{m} T_i \right|}{\left| \bigcup_{i=1}^{m} T_i \right|} \tag{2}$$

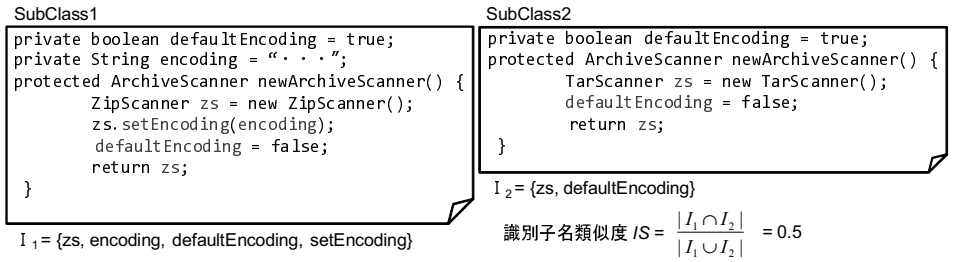


図 4　識別子類似度 IS の計算例
Fig. 4　Example of how to calculate IS Simirality

3.2.6 変更の有無とメトリクス値との関係を調査

同時変更の生じたクラス群とそうでないクラス群に対して，計測したメトリクス値の分布に差異があるかどうかを確かめる．

4. 調 査 結 果

本節では，ArgoUML, JFreeChart, JHotDraw の 3 種類のオープンソースソフトウェアを対象に調査した結果について述べる．

4.1 同時変更の生じた数

図 5 と表 1 は検出された Template Method パターンの数と Primitive Operation の数を示す．棒グラフの上部は同時変更が生じた適用事例の数であり，下部は同時変更が生じなかった適用事例の数である．

4.2 メトリクス値の計測結果

図 6 は 3.2.5 節で説明した識別子類似度 (IS) と型名類似度 (TS) を表すメトリクスの分布を箱ヒゲ図で表したものである．それぞれの図の左側が同時変更の生じなかった適用事例のメトリクスの分布を表し，右側が同時変更の生じたメトリクスの分布を表す．また，図中の長方形の下側がメトリクスの分布の第 1 四分位を表し，上側が第 3 四分位を表し，中央の直

表 1　検出された Template Method パターンの詳細
Table 1　Detail of detected Template Method

名前	バージョン数	Template Method の適用事例の数	同時変更の生じた Template Method の適用事例の数	Primitive Operation 数	同時変更の生じた Primitive Operation 数
ArgoUML	10	84	17	98	22
JFreeChart	15	43	18	26	20
JHotDraw	7	37	11	38	12

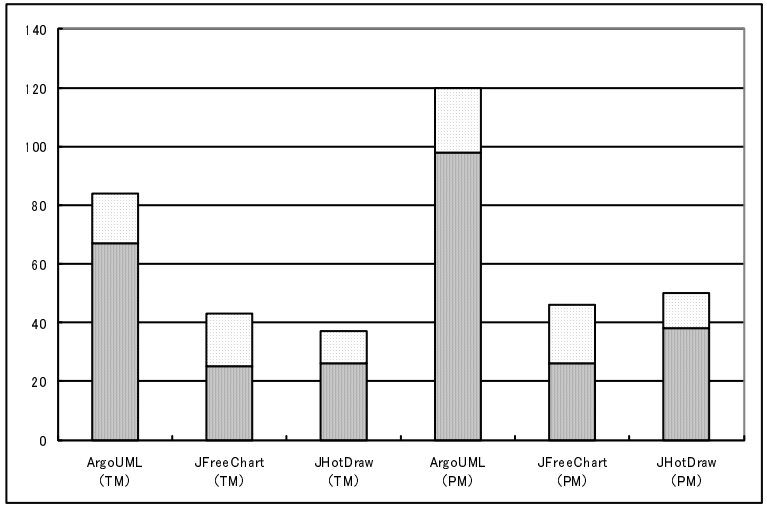


図 5 Template Method と Primitive Operation の同時変更が生じた数
(下部：同時変更が生じなかった数，上部：同時変更が生じた数)

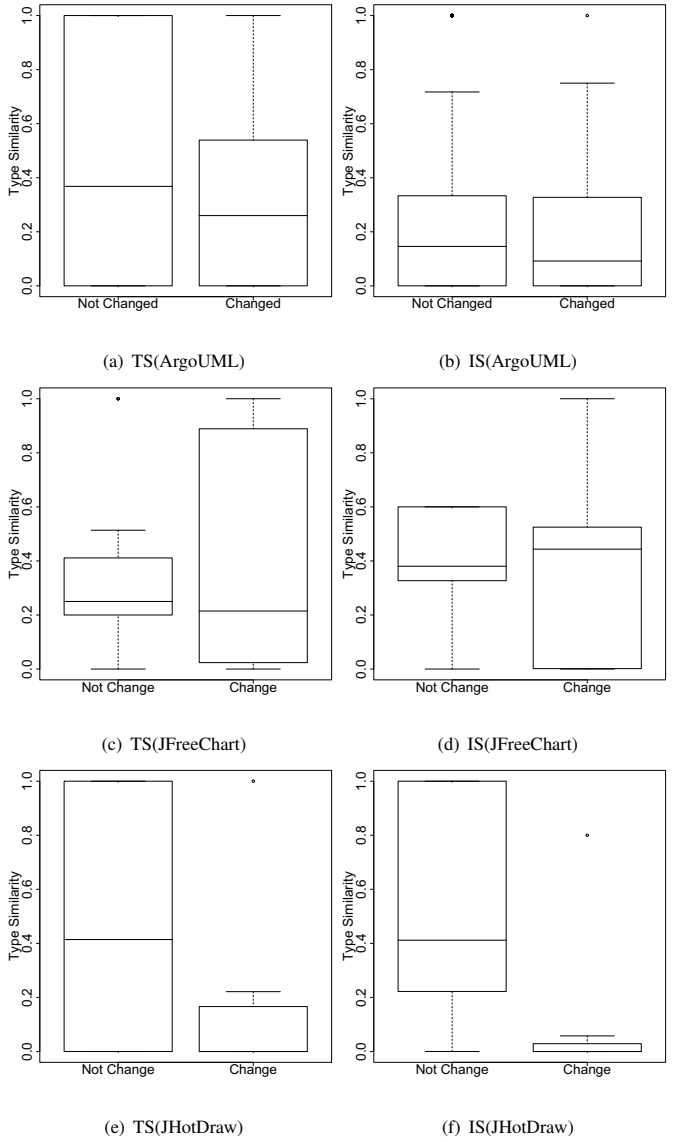
Fig. 5 Number of Template Methods and Primitive Operation changed between versions (The lower part: the number of simultaneous change, the upper part: the number of no simultaneous change)

線が中央値 (メディアン) を表している．

JFreeChart に対して計測した識別子名類似度 (図 6(d)) を除き，同時変更の生じたパターンの適用事例のほうが同時変更の生じていないパターンの適用事例よりも類似度が低下する傾向があった．また，同時変更が生じた適用事例とそうでない適用事例との間で計測したメトリクスに差異があるかどうかを調べるために，マン・ホイットニーの U 検定による検定を行ったところ，表 2 に示す結果が得られた．

表 2 U 検定による検定結果
Table 2 Result of U test

	型名類似度 (TS)	識別子名類似度 (IS)
統計量	2.26	1.80
P 値	0.02	0.07



(a) TS(ArgoUML) (b) IS(ArgoUML)

(c) TS(JFreeChart) (d) IS(JFreeChart)

(e) TS(JHotDraw) (f) IS(JHotDraw)

図 6 識別子名類似度と型名類似度の分布
Fig. 6 Distributions of Type Similarity and Identifier Similarity

5. 考　　察

ここでは，分析結果の評価と，実際に検出結果から同時変更が生じた例を示す．

5.1 分析結果の評価

図 6 で示したメトリクス値の分布に対してコルモゴロフ-スミルノフ検定を行い正規分布であるかどうかを調べたが，正規分布であると言えなかったため，ノンパラメトリック検定の 1 種であるマン・ホイットニーの U 検定を使用した．表 2 より，型名類似度に関しては，有意水準 5%(P=0.05) において，有意差が得られた．このことより，仮定で示した Primitive Operation の定義に差異が大きいほど，同時変更が生じやすいと言える．

実験結果の妥当性に関する問題として，使用した適用事例検出ツールの誤検出に強く影響を受けることが挙げられる．検出した Template Method パターンの適用事例が間違っていればこれらの結果の信頼性は低下する．しかし，本研究で対象とした Template Method パターンはクラスの静的構造を分析することで検出できるため検出が比較的容易であり，さらに使用したデザインパターンの適用事例は適合率が高いことが報告されているので，検出間違いにおける結果の信頼性の低下は少ないと考えられる．

また，今回用いた同時変更の有無を判定する基準は，3.2.4 節に示した 3 つの条件を用いたが，親クラスの名前のみが変更された場合でも，そのクラスが消滅したと見なされ，同時変更されたと判定される．この場合は親クラスのみしか変更されないので同時変更と判定するのは望ましくないが，名前が変更されたクラスの対応関係を取得することは容易でないので，今回の実験ではそのような誤差を含んでいる．今回はオープンソースソフトウェア 3 種類を実験対象としたが，このようなクラス名の変更が多いソフトウェアであれば，同様の結果が得られない可能性がある．また，Template Method パターンの処理ロジックは言語固有の機能を使用していないので，継承概念が扱うことができ，デザインパターンの適用事例が検出できれば，他の言語でも同様の結果が得られると考えられる．

5.2 検出された同時変更の例

計測された類似度が低い例で，実際にどのような変更が生じたかを説明する．以下に示す例は，識別子名類似度 IS, 型名類似度 TS がともにゼロで同時変更が生じた例である．図 7 は同時変更が生じたコードの前後の状態を表す．図 7 の上部は同時変更が生じる前のソースコードであり，下部は同時変更が生じた後のソースコードである．この例は instanceof 演算子によって型情報の比較を行っている．引数の型”Mbase” が次のバージョンで，“Object” に変更されている．

```
//UMLAssociationRoleListModel
protected boolean isValidElement(MBase o){
  return o instanceof MAssociationRole &&
  ((MAssosiation)getTarget()).・・・.contains(o);
}
```

```
//UMLAssociationEndAssociationListModel
protected boolean isValidElement(MBase element){
  return element instanceof MAssociation &&
  ((MAssosiationEnd)_target).・・・.equals(element)
}
```

```
//UMLAssociationLinkListModel
protected boolean isValidElement(MBase element){
  return o instanceof Mlink &&
  ((MAssociation) getTarget()).・・・.contains(o);
}
```

(a) 同時変更が生じる前のソースコード

```
//UMLAssociationRoleListModel
protected boolean isValidElement(Object o){
  return Modek.getFacade().isAssociationRole(0) &&
   .getFacade().getAssociationRoles().・・・
   contains(o);
}
```

```
//UMLAssociationEndAssociationListModel
protected boolean isValidElement(Object element){
 return Model.getFacade().isAssociation(element) &&
  Model.getFacade().getAssociation(getTarget())
  .equals(element)
}
```

```
//UMLAssociationLinkListModel
protected boolean isValidElement(Object element){
  return Model.getFacade().isALink(o)
  && Model.getFacade().getLinks(getTarget())
  .contains(o);
}
```

(b) 同時変更が生じた後のソースコード

図 7 同時変更が生じた例
Fig. 7 Example of actual simultaneous modifications

変更された原因は．この比較をやめるために引数が変化し，同時変更が生じていると考えられる．このような例は，子クラスの処理の差異が同時変更に起因しているといえる．

6. 関 連 研 究

デザインパターンの適用事例を用いたプログラムの評価に関する研究として，Bieman らの研究が挙げられる．Bieman らは，デザインパターンの適用事例の検出を用いて，デザインパターンの適用の有無とクラスの変更頻度との関係を調査し，以下の仮説を調査した．[4)1)].

H1. 規模の大きなクラスは変更されやすい．規模の大きさは，クラスに含まれている属性の数やメソッドの数などを用いて計測する．

H2. デザインパターンが適用されているクラスはそうでないクラスより変更されにくい．

H3. 継承によって再利用されているクラスはそうでないクラスより変更されにくい．

2 つの商用ソフトウェアに対して調査した結果，規模の大きさに比例して変更数も増加しており，仮説 H1 に従う傾向が得られている．しかし，デザインパターンを適用しているクラス群のほうがデザインパターンを適用していない暮らす群よりも変更頻度が高く，また継承によって再利用されているクラスのほうが変更頻度が高くなり，仮説 H2，H3 に反する結果となった． Bieman らはデザインパターンを適用しているクラス群と適用していないクラス群を比較しているが，デザインパターンの適用事例の中での評価は行ってない．本稿ではデザインパターンの適用事例の評価基準として変更頻度を考え，Template Method パターンに着目して変更頻度が高くなる要因の調査を行っている．

7. お わ り に

本稿では，デザインパターンの安定性を調査するため，実際のソースコード中で頻出する Template Method パターンを取り上げ，オープンソースソフトウェアから変更具合を調査した．

本稿では Template Method パターンの適用事例が変更される要因として，子クラスの処理内容の差異に起因すると考えた．差異を表すメトリクス２つ (識別子名類似度，型名類似度) を提案し，Java 言語で記述された 3 つのオープンソースソフトウェアから計測した．その結果，識別子名類似度・型名類似度は変更が生じたパターンの適用事例のほうが，変更が生じなかったパターンの適用事例よりも小さくなる傾向が得られた．今後の課題として，計測したメトリクスを組み合わせて，ソフトウェアが次のバージョンにおいて変更されやすいかどうか予測するなどの手法につなげることが考えられる．

謝　　辞

本研究は一部，日本学術振興会科学研究費補助金基盤研究 (A)(課題番号:21240002)，文部科学省科学研究費補助金若手研究 (B)(課題番号:20700024)，日本学術振興会特別研究員奨励費 (課題番号:20・1964) の助成を得た．

参 考 文 献

1) J.M. Bieman, G.Straw, H.Wang, P.Munger, and R.Alexander. Design patterns and change proneness: An examination of five evolving sysftems. In *Proceeding of the 9th IEEE International Software Metrics Symposium(METRICS '03)*, pp. 40–49, Sydney, Australia, 2003.
2) E.Gamma, R.Helm, R.Johnson, and J.Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
3) Y.G. Gueheneuc and G.Antoniol. DeMIMA: A multilayered approach for design pattern identification. *IEEE Trans. Softw. Eng.*, Vol.34, No.5, pp. 667–684, 2008.
4) J.Bieman, D.Jain, and H.Yang. OO design patterns, design structure, and program changes: an industrial case study. In *Proceedings of the 17th IEEE International Conference on Software Maintenance (ICMS '01)*, pp. 580–589, Florence, Italy, 2001.
5) MASU. `http://sourceforge.net/projects/masu/`.
6) T.Ng, S.Cheung, W.Chan, and Y.Yu. Work experience versus refactoring to design patterns: a controlled experiment. In *Proc. of SIGSOFT'06/FSE-14*, pp. 12–22, Portland, OR, USA, 2006.
7) J.Niere, W.Schäfer, J.Wadsack, L.Wendehals, and J.Welsh. Towards pattern-based design recovery. In *Proc. of ICSE'02*, pp. 338–348, Orlando, FL, USA, 2002.
8) N.Shi and R.Olsson. Reverse engineering of design patterns from java source code. In *Proc. of ASE'08*, pp. 123–134, Tokyo, Japan, 2006.
9) S.Steliting and O.Maassen. *Applied Java Patterns*. Sun microsystems, 2002.
10) N.Tsantalis, A.Chatzigeorgiou, G.Stephanides, and S.Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Softw. Eng.*, Vol.32, No.11, pp. 896–909, 2006.
11) 三宅達也, 肥後芳樹, 井上克郎. メトリクス計測プラグインプラットフォーム MASU の開発. ソフトウェアエンジニアリング最前線 2008, pp. 63–70, 2008.