特別研究報告

題目

デザインパターンの適用事例に対するプログラム構造の安定性の調査

指導教員
井上 克郎 教授

報告者
齋藤 晃

平成 21 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 20 年度 特別研究報告

デザインパターンの適用事例に対するプログラム構造の安定性の調査

齋藤 晃

内容梗概

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まとめたものであり，デザインパターンを用いることでソフトウェアの保守性を向上させることができる．しかし，ソフトウェア開発過程において，デザインパターンは必要でない状況で適用される場合がある．その場合デザインパターンを適用したクラス群が，ソフトウェアの進化に伴って，デザインパターンとして示されたクラス構造が崩れ，プログラムの安定性が損なわれる．

そこで本研究では，デザインパターンの変更要因を調査することによって，将来に渡って変更の可能性のあるデザインパターンと変更の可能性の少ないデザインパターンに分類することを試みる．対象とするデザインパターンは他のパターンと比べて頻繁に用いられる Template Method パターンとする．Template Method パターンは，1 つの親クラスとそれを継承する複数の子クラスから構成される．さらに，親クラスの処理の一部を子クラスに実装を任せることにより，処理内容の切り替えを容易にしている．

本手法では Template Method パターンの抽象メソッドをオーバーライドしている子クラスの処理の規模と差異が，Template Method パターンの変更要因であると考えた．なぜなら，子クラスの処理の差異と規模が大きいほど，変更要求が親クラスまで影響しやすく，変更が生じやすいからである．

適用実験として，Java 言語で開発されたオープンソースソフトウェアに対して，デザインパターン検出ツールを用いて Template Method パターンの適用事例を検出する．検出された適用事例に対して，各リリースバージョン間に変更が生じたかを判定する．変更が生じた場合と生じなかった場合に対して，子クラスのコード行数や参照している型集合・識別子名の類似度に差異があるかを調査した．調査した結果，変更が生じたクラスの方が生じなかったクラスよりも型集合・識別子名の類似度が低くなる傾向が得られた．

主な用語

デザインパターン (Design Pattern)
変更分析 (Change Analysis)
リポジトリマイニング (Repository Mining)

目 次

1　まえがき

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，その効果がそれぞれ文書化されている．とりわけ，Gamma らの示した 23 種類のデザインパターン [9] が広く知られている．デザインパターンを知ることで得られる利点として，ソフトウェア開発コストの低減が挙げられる．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することが可能となるので，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る [17].

しかし，ソフトウェア開発過程において，デザインパターンが必要でない状況で適用される場合がある．その場合デザインパターンを適用したクラス群が，ソフトウェアの進化に伴って，デザインパターンの解法として示されたクラス構造が崩れてしまい，かえって修正範囲が増大することがある．

そこで本研究では，デザインパターンの変更要因を調査することによって，将来に渡って変更の可能性のあるデザインパターンと変更の可能性の少ないデザインパターンに分類することを試みる．

対象とするパターンは，Gamma らの示したデザインパターンのうちソースコード中で出現頻度の高い Template Method パターンとする．Template Method パターンは，1 つの親クラスとそれを継承する 1 つ以上の子クラスから構成される．親クラスは処理内容を実装しない抽象メソッド (Primitive Operation) と，それを呼び出す具象メソッド (Template Method) が定義される．抽象メソッドは親クラスを継承する子クラス内でオーバーライドされる．このように，アルゴリズムの共通な部分は親クラスの Template Method が実装し，個別の部分を子クラスで実装させることで，個別な部分の変更は新たに子クラスを追加・修正するだけでよく，既存のコードに修正を加える必要が無いという利点がある．

Template Method パターンは機能変更・追加の際は子クラスのみ修正・追加が行われることが理想的であるが，抽象メソッドである Primitive Operation の定義が変更された場合や，Template Method パターンそのものが消滅した場合は，親クラスと子クラスの両方に修正が必要となり，Template Method パターンの利点が生かせていない．

本手法では Template Method パターンの抽象メソッドをオーバーライドしている子クラスの処理の規模と差異が，Template Method パターンの変更要因であると考えた．なぜなら，子クラスの処理の差異と規模が大きいほど，変更要求が親クラスまで影響しやすく，変更が生じやすいからである．つまり，以下の状況において相関があるかを検証する．

- 子クラスの処理内容に差異が大きい場合は変更が生じやすい.
- 子クラスの処理内容の規模が大きい場合は変更が生じやすい.

実験手法としては，ソフトウェアリポジトリ公開 Web サイト SourceForge[21] などに掲載されている Java 言語で開発されたオープンソースソフトウェアの各リリースバージョンに対して，Tsantalis らのデザインパターン検出ツール [23][24] を用いて Template Method パターンを検出する．検出された Template Method パターンに対して，メトリクス計測プラグインプラットフォーム MASU[25] を用いて各リリースバージョン間に変更が生じたかを判定する．そして Template Method パターンの子クラスに対して，LOC(コード行数) 平均，LOC 分散，識別子名類似度，型名類似度の 4 つのメトリクスを計測する．識別子名類似度は，子クラス内の一時変数・参照変数の識別子名と，呼び出すメソッドの名前の類似性を数値として表す．型名類似度はクラス内の一時変数・参照変数の型名と，呼び出すメソッドの引数と戻り値の型名の類似性を表す．子クラスの処理内容が一致していなければ，これらの類似度は低くなる．この４つのメトリクスを用いて，子クラスの処理内容の差異を計測し，クラスの変更の要因となるかを調査する．

調査した結果，変更が生じたクラス群は，変更が生じなかったクラス群より，識別子名類似度と型名類似度が低くなる傾向が得られた．これにより，識別子名類似度と型名類似度が低い場合はクラス群の変更の要因の一つとして考えられる．また実際に変更が生じたクラス群が，子クラスの処理の差異に起因して変更が生じていることを確かめた．

以降，2 節ではデザインパターンの使用における利点と問題点についての研究背景を説明する．3 節では，デザインパターンの変更の調査手法を説明し，4 節では適用実験について述べる．5 節で関連研究を述べ 6 節で今後の課題について述べる．

2　研究背景

本節で研究の背景となっている問題を説明するために，デザインパターン，デザインパターン検出，ソフトウェアリポジトリについて説明し，デザインパターンの変更と問題点について述べる．

2.1　デザインパターン

Gamma らは，デザインパターンを “種々の状況における設計上の一般的な問題の解決に適用できるよう，オブジェクトやクラス間の通信を記述したもの” と定義している [10]．デザインパターン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，デザインパターンには以下に示す 4 つの要素が記述される．

パターン名 (pattern name) 設計問題とその解法および結果を 1，2 語で記述した名称．名前を付けることで，設計時に使用できる語彙を増やすことができ，議論したり文書化したりすることが容易になる．

問題 (problem) どのような状況においてパターンを適用するかを記述したもの．パターン適用前の設計状態や，満たさなければならない条件を示す．

解法 (solution) 設計の要素や，それらの関連を示したもの．ただし，解法は具体的な設計や実装の記述はせず，抽象的に記述される．

結果 (consequences) パターンを適用した結果やトレードオフを記述したもの．代替案の評価やパターンを適用する場合のコストや有効性を把握する際に重要となる．

実際に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．以下に Template Method パターンの概要を示す [22]．

パターン名 (pattern name) Template Method パターン

問題 (problem) ある一連の処理が存在し，処理の順序は決まっているが，処理の一部を可変に実装したい．

解法 (solution) 親クラスとそれを継承する子クラスからのクラス階層を構成し，共通な処理は親クラスが実装し，個々の処理は子クラスが実装する．具体的に以下の図 1 のようなクラス階層になる．

Template Method パターンは以下の構成要素を持つ．

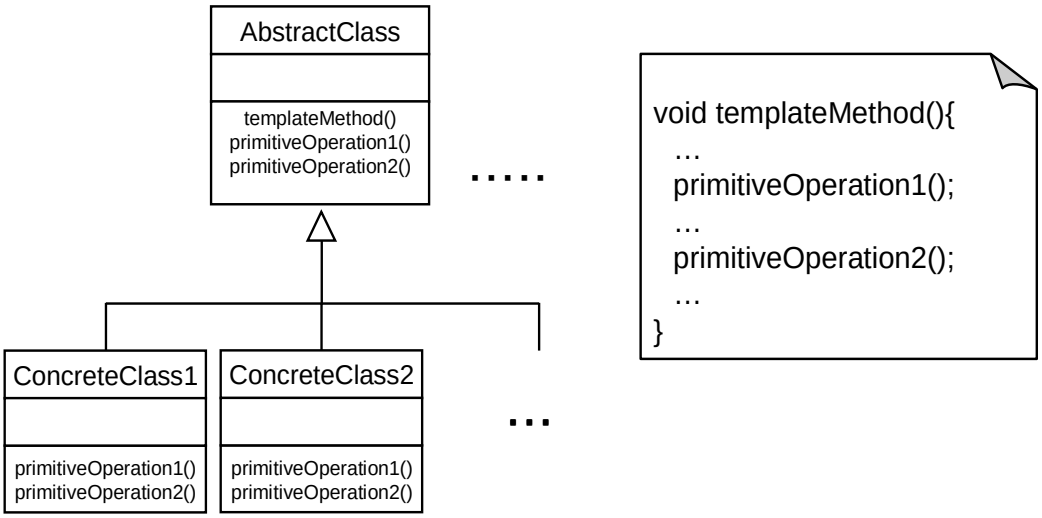


図 1: Template Method パターンのクラス階層

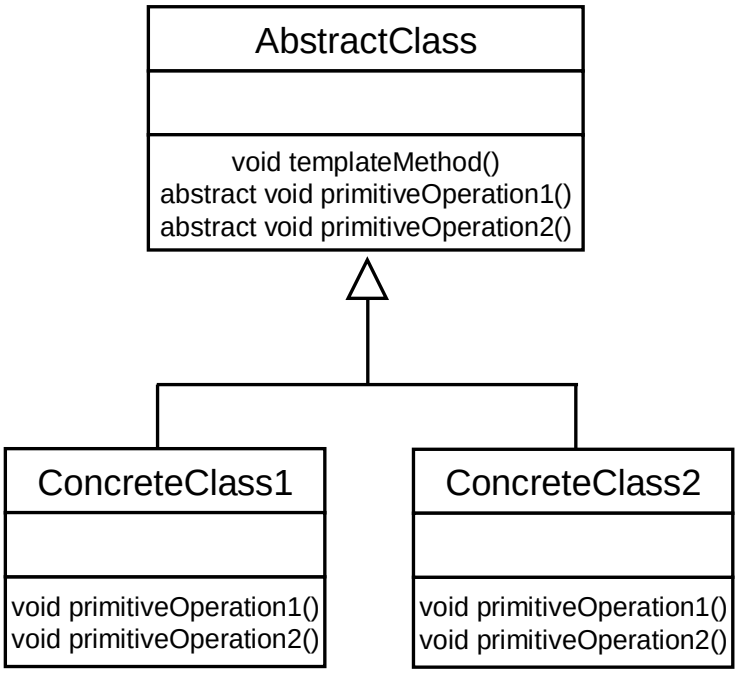
Abstract Class　抽象化された Primitive Operation を定義する．このメソッドは子クラスで定義される．また，処理の順序を定義する Template Method を定義する．Template Method は Primitive Operation を呼び出し，処理の共通な部分を実装する．

Concrete Class　Primitive Operation を実装し，各処理の可変な部分を定義する．

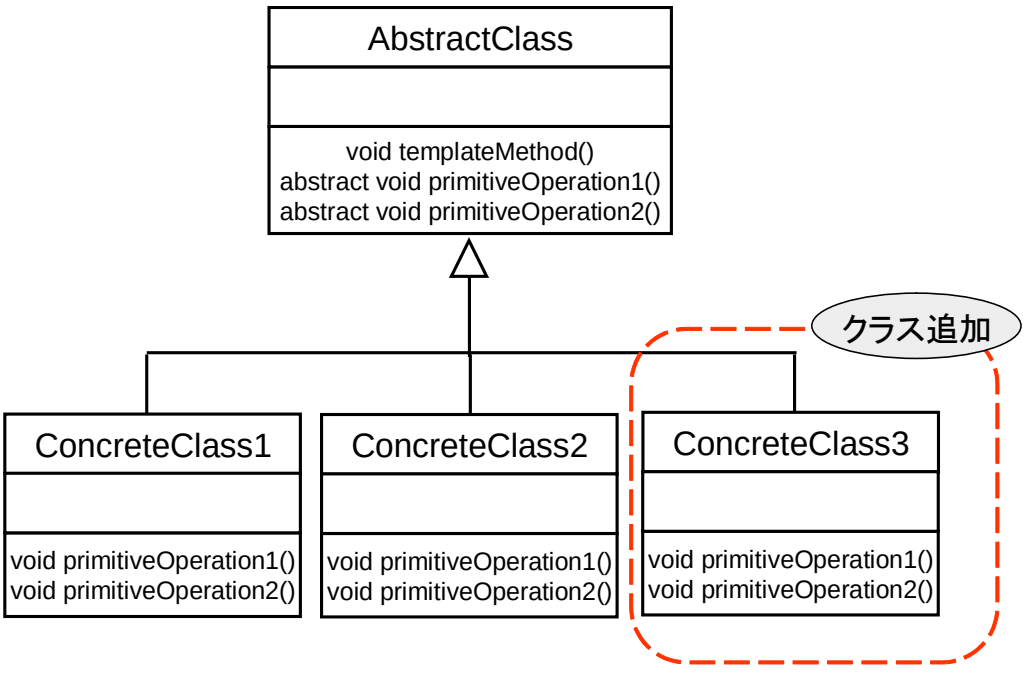
結果 (consequences) 可変な処理部を修正・追加したい時は，図 2(a), 図 2(b) のように子クラスを追加・修正するだけで良く，親クラスや他の子クラスに何ら影響を与えない．この様に，修正・追加に対して既存のクラスに影響する範囲が小さいので，保守性が向上する．

2.2　デザインパターン検出

既存のプロジェクト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．プログラムの構造理解や再利用を目的として，デザインパターンを検出する研究が行われている [11][18][20][24]．現在公開されているデザインパターン検出ツールとしては，PINOT[20] や Tsantalis らのツール [24] が挙げられる．共に Java 言語を対象としている．PINOT は，Gamma らの提案したデザインパターンを検出するためにデザインパターンの再分類を行い，23 種類のパターンのうち 17 種類を検出することができる．Tsantalis らのツールは，バイトコードからそれぞれのクラスの構成や参照関係などの類似度を計測することによって，約 10 種類のデザインパターンを検出でき



(a) 変更前のクラス構造



(b) 変更後のクラス構造

図 2: 修正におけるプログラム構造の変化

る．このツールの検出結果はデザインパターンでないパターンを誤って検出すること (False Positive) が少ないという特徴を持つ．

2.3 ソフトウェアリポジトリ

ソフトウェアリポジトリ (以下，リポジトリと書く) とは，ソースコードやドキュメント・バグレポートなどの各プロジェクトの成果物を蓄積しているデータベースである．オープンソースソフトウェアでは，リポジトリが公開されているものがあり，自由にソースコードを入手できる．リポジトリは最新のソースコードだけでなく，過去のソースコードも入手できるので，ソースコード変更分析に関する研究に用いられる [4][15]．オープンソースソフトウェアでリポジトリを公開している代表的な Web サイトとして，SourceForge[21], Apache Jakarta Project[19] が挙げられる．またこれらの Web サイトからソフトウェアの各リリースバージョンの実行ファイルも入手できる．

2.4 デザインパターンの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際に効果的な解法を与えるが，ソフトウェア進化の過程でデザインパターンの構造が変化したり，無くなったりすることがある．デザインパターンの構造が変更されるときは，複数のクラスを修正する必要があり，保守性の観点で良くないといえる．Template Method パターンにおいては，以下の場合にその構造が変化する．

- Primitive Operation のメソッド定義の変更
- Primitive Operation の数の変更
- Template Method パターンそのものの消滅

これらの変更の詳細を以下に示す．

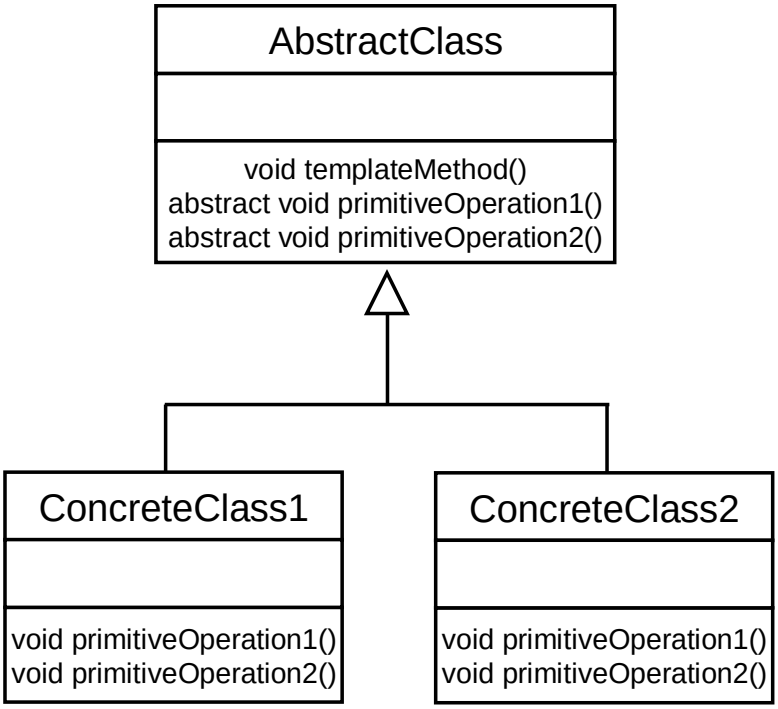
Primitive Operation のメソッド定義の変更　メソッドは以下に示す４つの要素から構成される．

- メソッド名
- 戻り値の型
- 引数の数
- 各引数の型

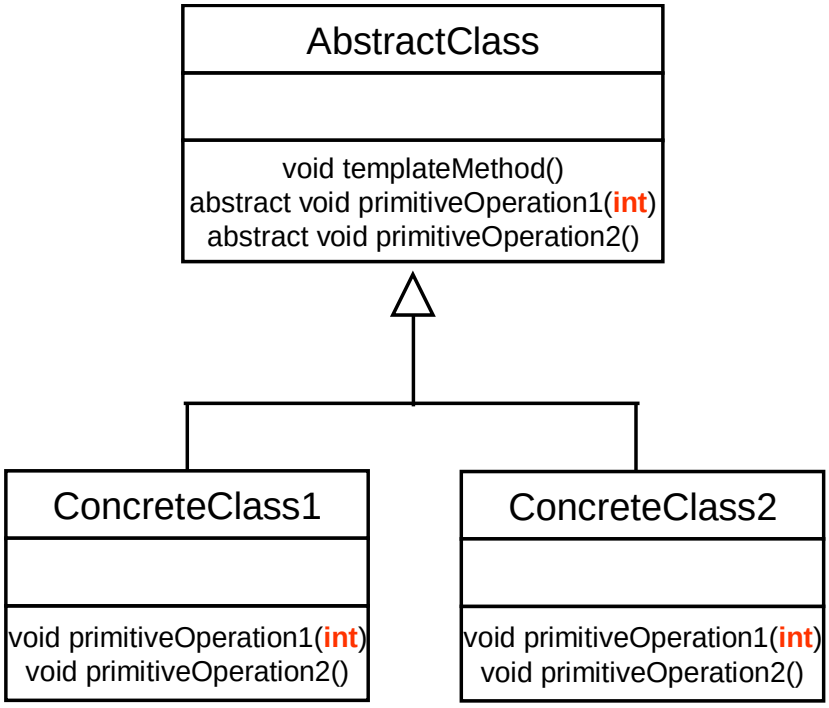
メソッド定義の変更の例として，図 3(a) で示す Template Method パターンのクラス構造に対して，Primitive Operation の引数が void から int に変わった場合のクラス構成を図 3(b) に示す．図 3(b) のように，全てのクラスに修正する必要がある．

Primitive Operation の数の変化　メソッド定義の変更と同様に，図 4(a) で示す 1 つの Template Method パターンに対して Primitive Operation の数が増減した場合も全てのクラスを変更する必要がある．図 4(b) は子クラスで定義すべき抽象メソッドが増えた場合の例である．

Template Method パターンそのものの消滅　Template Method パターンが消滅した場合も，親クラスと子クラスの両方の抽象メソッドが無くなるので，全てのクラスを修正する必要がある．

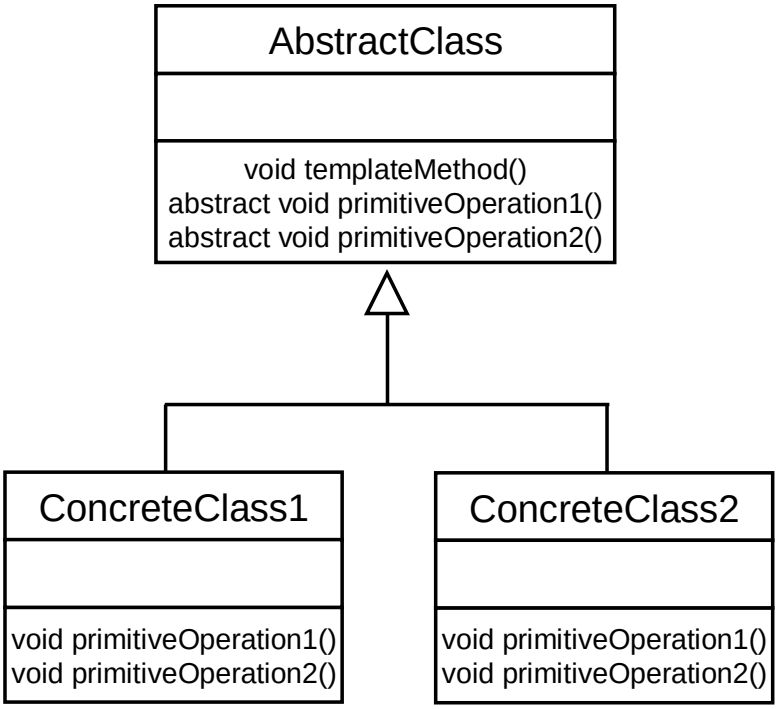


(a) メソッド定義の変更前のクラス構造

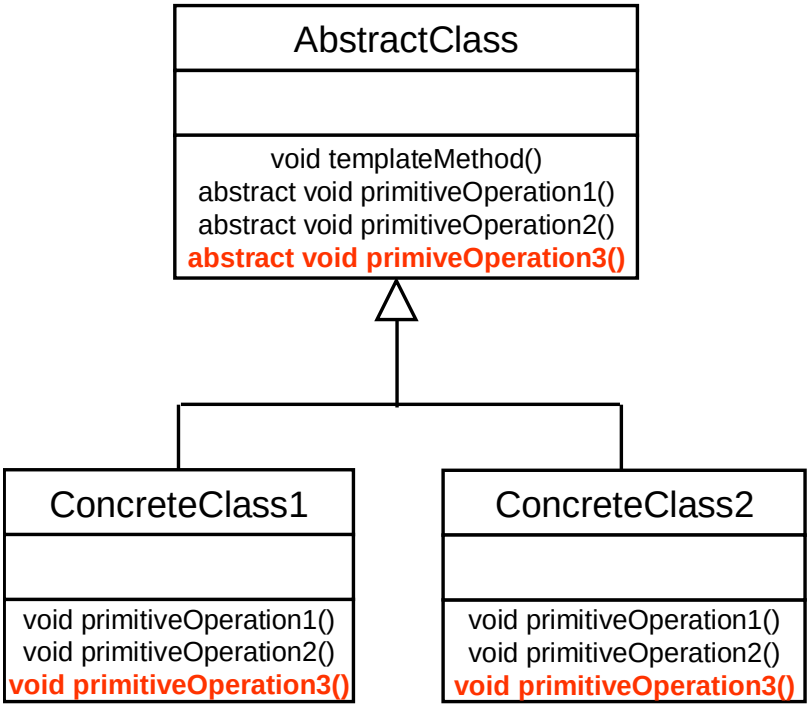


(b) メソッド定義の変更後のクラス構造

図 3: メソッド定義の変更時におけるクラス構造



(a) 抽象メソッドの数の変更前のクラス構造



(b) 抽象メソッドの数の変更後のクラス構造

図 4: 抽象メソッド数の変更時のクラス構造

3　調査手法

本節では，デザインパターンの適用事例の変更を検出する方法と，メトリクスによる比較方法について述べる．

3.1　調査内容

本手法では，デザインパターンの適用事例が変更される要因を調査する．2 節で示したような変更の原因は，オーバーライドしている子クラスの処理内容の差異が影響すると考えた．そこで，以下の仮説について検討する．

仮説 1. 子クラスの処理内容の差異が大きい場合は変更が生じやすい．

仮説 2. 子クラスの処理内容の規模が大きい場合は変更が生じやすい．

以降の節で調査方法について説明する．

3.2　調査手法の概要

全体の処理概要を図 5 に示す．調査の手順は以下の 6 つに分けられ，それぞれについて説明する．

Step1. リポジトリからソースコードと Java バイトコードを入手

Step2. Template Method パターンの検出

Step3. クラス情報の抽出

Step4. 変更の有無による分類

Step5. 子クラスの処理内容の差異を表すメトリクスを計測

Step6. 変更の有無とメトリクス値との関係を調査

3.3　[Step1] リポジトリからソースコードを入手

SourceForge[21] や Apache Jakarta Project[19] 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を入手する．Java 言語で記述されたプログラムを対象とする．また後の処理でデザインパターンツールを用いてデザインパターン検出を行なう時に必要となるので同時に Java バイトコードも入手する．ソースコード・Java バイトコードは共にリリースバージョンごとに入手する．

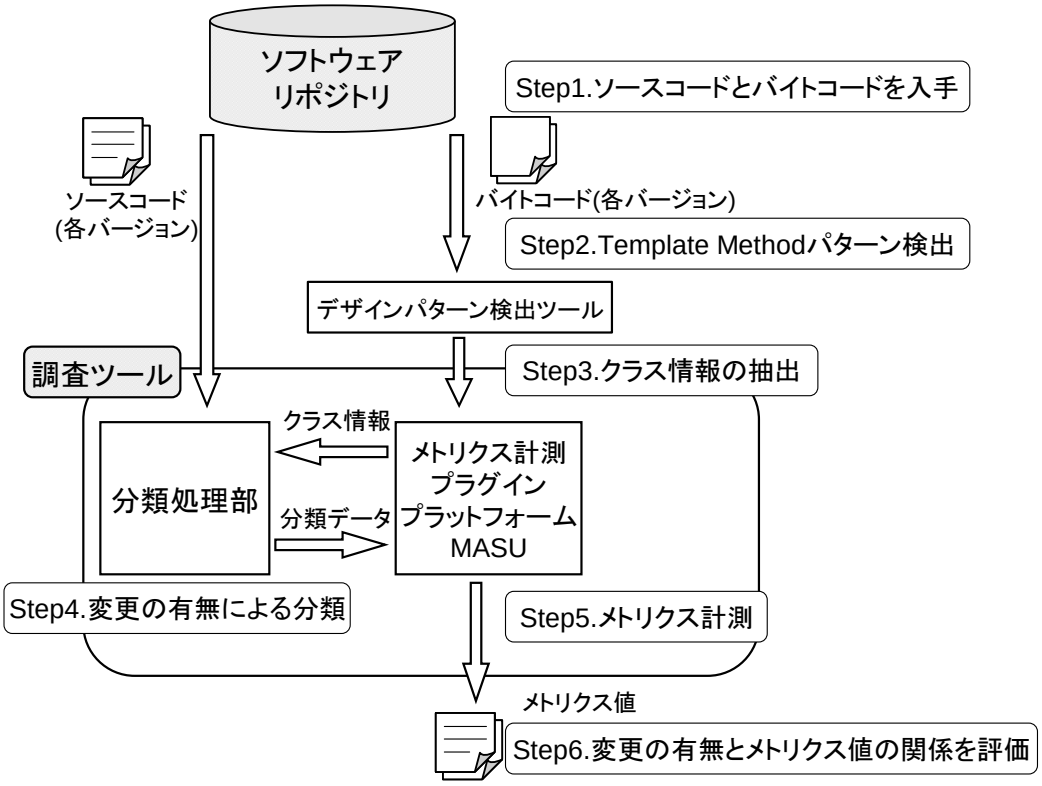


図 5: 調査手法の全体の流れ

3.4 [Step2] Template Method パターンの検出

Tsantalis らのデザインパターン検出ツール [24] を用いてデザインパターンを検出する．このツールは Template Method パターンを含めた約 10 個のデザインパターンを検出するが，Template Method パターン以外のパターンの情報は扱わない．また検出結果は XML ファイル形式で出力され，Template Method パターンの親クラスのクラス名とパッケージ内の位置が取得できる．

3.5 [Step3] クラス情報の抽出

3.4 節で示した Template Method パターンの検出は，親クラスのクラス名とパッケージ内位置しか検出できないので，メトリクス計測プラグインプラットフォーム MASU[25] を用いて以下の情報を Template Method パターンから取得する．MASU は，メトリクスを計測するためのツールであるが，メトリクス計測時にソフトウェアの構造解析を行うので，クラス情報の抽出に使用している．

- 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 パターンが存在することがあるので，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ので，Primitive Operation の数を取得する必要がある．

3.6 [Step4] Template Method パターンの変更の有無による分類

3.4 節で述べた手法で検出したパターン群を，以下の 2 つの集合に分類する．

- 各バージョンで 1 度以上変更があったパターンの集合 D
- 各バージョンで 1 度も変更がなかったパターンの集合 ND

ここで，変更があったとは，2 節で示した以下の変更を指す．

- Primitive Operation のメソッド定義の変更 (図 3)
- Primitive Operation の数の変更 (図 4)

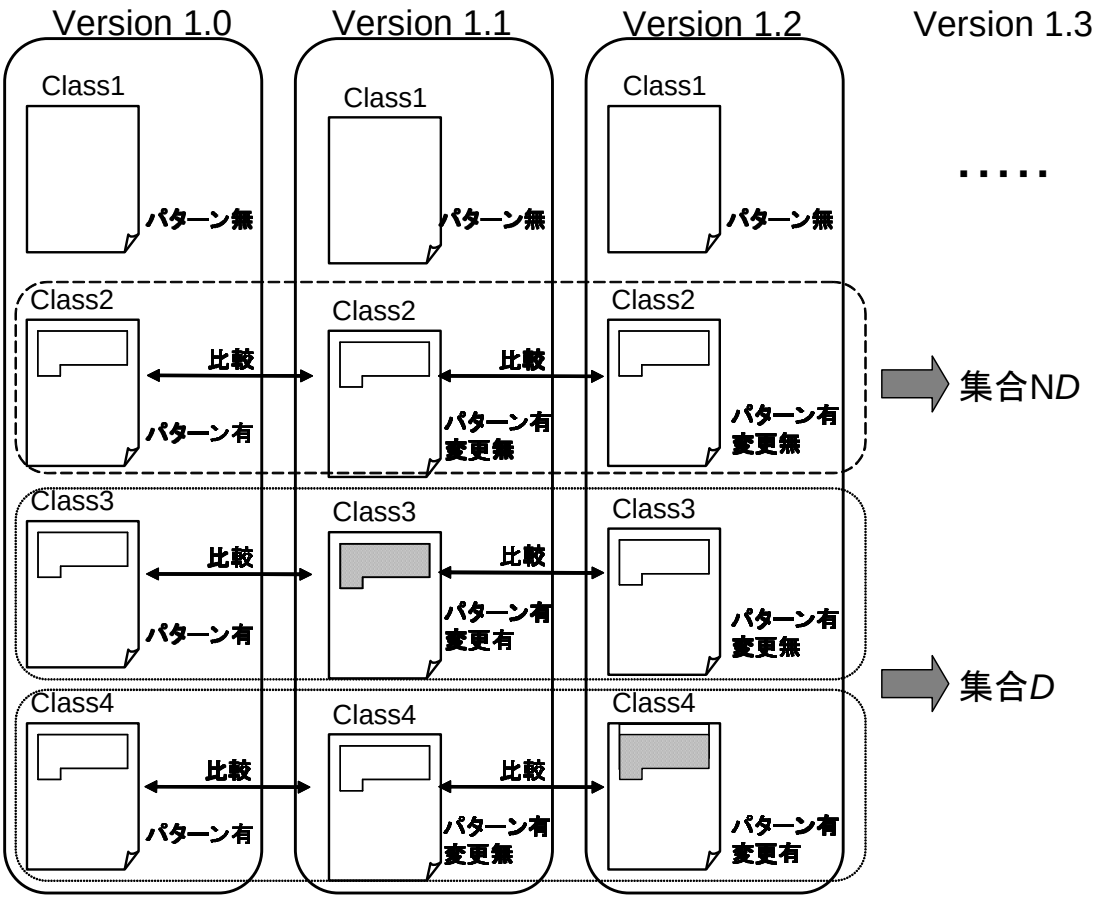


図 6: 変更の分類例

- Template Method そのものの消滅

また 1 度以上変更されるとは，あるパターンが各バージョン間で 1 度でも変更が検出されれば，その他のバージョンも変更があった集合 D に分類されることを指す．図 6 のように，バージョン 1.0, 1.1, 1.2 の 3 つの変更がある場合を考える．図 6 では，Class2, Class3, Class4 においてパターンが発見され，Class3, Class4 で変更が生じている．この場合，全てのバージョンにおいて Class2 が集合 ND に分類され，Class3, Class4 が集合 D に分類される．

それぞれのパターンの変更の判定方法を説明する．

Primitive Operation のメソッド定義の変化 メソッド定義はメソッド名・引数の型・引数の数・戻り値の型の組み合わせであるが，Primitive Operation のメソッド名が変化した場合は，バージョン変化でそれらが同一のメソッドであるかどうかを判定するのは困難であるため，戻り値と引数の型の変更のみを判定する．ただし，パッケージ名の

Version1.0

```
public SampleClass{
  //Primitive Operation
  abstract private Elem getElement();

  public void Parse(){
    …
    Elem e = getElement();
    …
}
```

クラス名：SampleClass
Template Method名：Parse
Primitive Operation名:getElement

(a) メソッド定義変化前のソースコード

Version1.1

```
public SampleClass{
  //Primitive Operation
 abstract private Elem getElement(int i);

  public void Parse(){
    …
    Elem e = getElement(10);
    …
}
```

クラス名：SampleClass
Template Method名：Parse
Primitive Operation名：getElement(引数変更)

(b) メソッド定義変化後のソースコード

図 7: メソッド定義の変更が判定される例

み変化した型は除外する．メソッドがオーバーロードされた場合を除くと，Primitive Operation は定義されているクラス名，呼び出されている Template Method のメソッド名，Primitive Operation のメソッド名の 3 つの情報から一意に識別できる．そこで，これらの情報がバージョン間で同一であり，Primitive Operation の戻り値と引数の型のみが異なっている場合，Primitive Operation のメソッド定義の変化と判定できる．Primitive Operation のメソッド定義の変化の例を図 7 に示す．

Primive Operation の数の変更 バージョン間で比較するとき，後のバージョンでは Primitive Operation が存在しなければ，Primitive Operation の数の変更と判定する．Primitive Operation の数の変更と判定される例を図 8 に示す．

Template Method の消滅 バージョン間で Primitive Operation が消滅し，かつ Template Method も無くなっていれば，Template Method の消滅と判断する．Template Method の消滅と判断される例を図 9 示す．

Version1.0

```
public SampleClass{
  //Primitive Operation
  abstract private Elem getElement();

  public void Parse(){
    …
    Elem e = getElement();
    …
}
```

クラス名 : SampleClass
Template Method名 : Parse
Primitive Operation名:getElement

(a) Primitive Operation 変更前のソースコード

Version1.1

```
public abstract SampleClass{

  //Primitive Operation消滅

  public void Parse(){
    …
  }
}
```

クラス名 : SampleClass
Template Method名 : Parse
Primitive Method名: 消滅

(b) Primitive Operation 変更後のソースコード

図 8: Primive Operation の変更が判定される例

Version1.0

```
public SampleClass{
  //Primitive Operation
  abstract private Elem getElement();

  public void Parse(){
    …
    Elem e = getElement();
    …
}
```

クラス名：SampleClass
Template Method名：Parse
Primitive Operation名:getElement

(a) Template Method 変更前のソースコード

Version1.1

```
public abstract SampleClass{

  //Primitive Operation消滅

  //Template Method消滅

}
```

クラス名：SampleClass
Template Method名：消滅
Primitive Method名: 消滅

(b) Template Method 変更後のソースコード

図 9: Template Method の消滅が判定される例

3.7 [Step5] メトリクス計測

それぞれのパターンから，子クラスの Primitive Operation に関するメトリクスを抽出する．以下に示すメトリクスを抽出する．

LOC 平均 (LOC_a) それぞれの子クラスでオーバーライドしているのメソッド (Primitive Operation) の LOC に対しての平均値．LOC 平均が高いほど，子クラスの処理規模は大きいと考えられる．

LOC 分散 (LOC_v) LOC 平均と同様に，それぞれの Primitive Operation の LOC に対しての分散．LOC 分散が高いほど，子クラスの処理内容に差異が大きいと考えられる．

識別子名類似度 (Sim_I) 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 Sim_I 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．

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

式 (1) は，子クラス同士で共通した識別子名がなければ，類似度 Sim_I が小さくなることを示している．識別子名類似度 Sim_I の計算例を図 10 に示す．

図 10 の SubClass1 中に出現する識別子名・メソッド名が “zs”, “encoding”, “defaultEncoding”, “setEncoding” であり，SubClass2 では “zs”, “defaultEncoding” であるので，識別子名類似度 Sim_I は，

$$Sim_I = \frac{\left| \bigcap_{i=1}^{2} I_i \right|}{\left| \bigcup_{i=1}^{2} I_i \right|} = \frac{|I_1 \cap I_2|}{|I_1 \cup I_2|} = 0.5 \tag{2}$$

となる．

型名類似度 (Sim_T) 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 Sim_T を以下のように定義する．

$$Sim_T = \frac{\left| \bigcap_{i=1}^{m} T_i \right|}{\left| \bigcup_{i=1}^{m} T_i \right|} \tag{3}$$

SubClass1

```
private boolean defaultEncoding = true;
private String encoding = “・・・”;
protected ArchiveScanner newArchiveScanner() {
        ZipScanner zs = new ZipScanner();
        zs.setEncoding(encoding);
        defaultEncoding = false;
        return zs;
 }
```

I_1 = {zs, encoding, defaultEncoding, setEncoding}

SubClass2

```
private boolean defaultEncoding = true;
protected ArchiveScanner newArchiveScanner() {
        TarScanner zs = new TarScanner();
        defaultEncoding = false;
        return zs;
 }
```

I_2 = {zs, defaultEncoding}

$$Sim_I = \frac{|I_1 \cap I_2|}{|I_1 \cup I_2|} = 0.5$$

図 10: 識別子名類似度 Sim_I の計算例

SubClass1

```
private boolean defaultEncoding = true;
private String encoding = “・・・”;
protected ArchiveScanner newArchiveScanner() {
        ZipScanner zs = new ZipScanner();
        zs.setEncoding(encoding);
        defaultEncoding = false;
        return zs;
 }
```

T_1 = {boolean, String, ZipScanner}

SubClass2

```
private boolean defaultEncoding = true;
protected ArchiveScanner newArchiveScanner() {
        TarScanner zs = new TarScanner();
        defaultEncoding = false;
        return zs;
 }
```

T_2 = {boolean, TarScanner}

$$Sim_T = \frac{|T_1 \cap T_2|}{|T_1 \cup T_2|} = 0.25$$

図 11: 型名類似度 Sim_T の計算例

型名類似度の計算例を図 11 に示す．図 11 の SubClass1 中に出現する型名は “boolean”, “ZipScanner”, “String” であり，SubClass2 では “boolean”, “TarScanner” であるので，型名類似度 Sim_T は，

$$Sim_T = \frac{\left|\bigcap_{i=1}^{2} T_i\right|}{\left|\bigcup_{i=1}^{2} T_i\right|} = \frac{|T_1 \cap T_2|}{|T_1 \cup T_2|} = 0.25 \tag{4}$$

となる．

3.8 [Step6] 変更の有無とメトリクス値との関係を調査

3.6 節で分類した集合ごとに 3.7 で定義したメトリクスを計測する．それぞれの分布に差異があるかどうかを確かめる．

4 適用実験

本節では3節で説明した調査手法を，実際のオープンソースソフトウェアに適用した事例について述べる．まず，システムの実行環境や適用対象を説明し，その後適用結果と考察を述べる．

4.1 概要

本節では適用実験で用いたシステムの実行環境と適用事例に用いたオープンソースソフトウェアについて説明する．

実行環境 適用実験において，使用したシステムの動作環境を以下に示す．

- CPU: Intel(R) Xeon(TM) CPU 2.8GHz
- メモリ: 3.0 GB RAM
- オペレーティングシステム: Linux (カーネルのバージョン:2.6.24-24generic)

適用対象ソフトウェア 適用実験で対象としたオープンソースソフトウェアの概要を表1に示す．ここで表1中の平均クラス数とは，各バージョンの総クラス数の平均を表した数値である．平均LOCも同様に，各バージョンに対しての総LOCの平均を表している．

計測したメトリクス 3.7節で説明した以下のメトリクスを，対象とするオープンソースソフトウェアから計測する．

- LOC 平均 LOC_a
- LOC 分散 LOC_v
- 識別子名類似度 Sim_I
- 型名類似度 Sim_T

表 1: オープンソースソフトウェアの概要

ソフトウェア名	種別	調査バージョン数	平均クラス数	平均 LOC(行)
Ant[1]	ビルドツール	12	507	136010
ANTLR[2]	パーザ生成ツール	6	165	57383
ArgoUML[3]	UML 描画ツール	10	1083	193593
FindBugs[7]	バグ検出ツール	3	902	150487
JFreeChart[12]	グラフ描画ツール	15	581	171955
JRefactory[14]	リファクタリング支援ツール	4	762	143427
JHotDraw[13]	図形描画ツール	7	368	62621
Log4j[16]	ロギングツール	8	242	39017

4.2 結果

本節では，適用実験の対象となるオープンソースソフトウェアから検出された Template Method パターンの数と，検出された Template Method パターンの中で変更が生じた数と生じなかった数を示す．また，変更の生じたパターンと変更が生じなかったパターンから計測されるメトリクス値を示す．

4.2.1 検出されたパターンの数

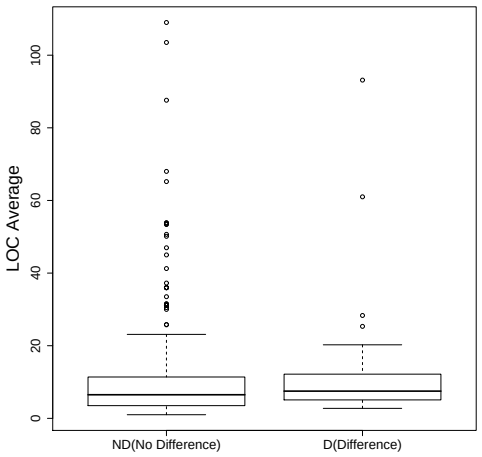
Tsantalis らのデザインパターン検出ツールとメトリクス計測プラグインプラットフォーム MASU を用いて検出したパターンの数とパターンが変更された数を表 2 に示す．表 2 中の項目は以下に示す意味を持つ．

- 検出された Template Method パターンの数 TMP_{ALL}
- 検出された Template Method パターンのうち，変更が生じなかったパターンの数 TMP_{ND}
- 検出された Template Method パターンのうち，変更が生じたパターン TMP_D
- 検出された Primitive Operation の数 PO_{ALL}
- 検出された Primitive Operation のうち，変更が生じなかった数 PO_{ND}
- 検出された Primitive Operation のうち，変更が生じた数 PO_D

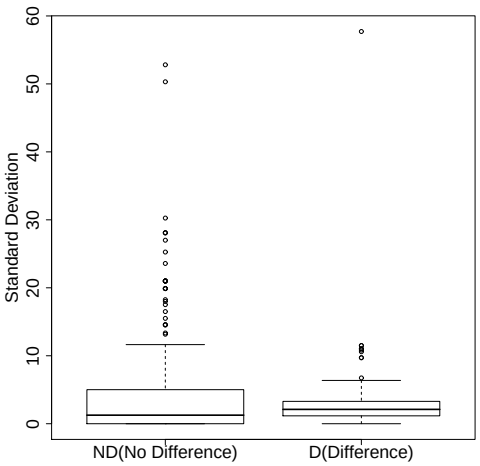
表 2 で示したように，ArgoUML, JFreeChart, JHotDraw においては，多くの Template Method パターンが検出されたが，その他のオープンソースソフトウェアでは検出数が少なかった．特に ANTLR では，検出された Template Method パターンが 1 つだけであり，変更が生じたパターンは存在しなかった．

4.2.2 メトリクス値の計測結果

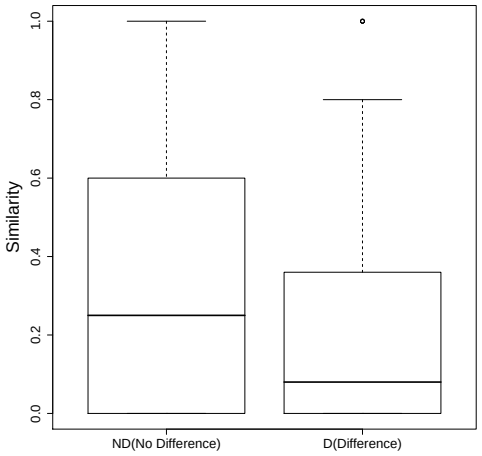
検出されたパターンから 3.7 節で示したメトリクス値を計測し，変更がなかった集合 ND と変更があった集合 D とに分けて示す．それぞれの集合ごとに計測されたメトリクス値 LOC_a, LOC_v, LOC_v, Sim_I, Sim_T を図 12 に示す．またその分布を図 13 に示す．ただし，LOC_v は，分布している値の範囲が大きかったので，グラフ上では標準偏差 (Standard Deviation) を表示している．



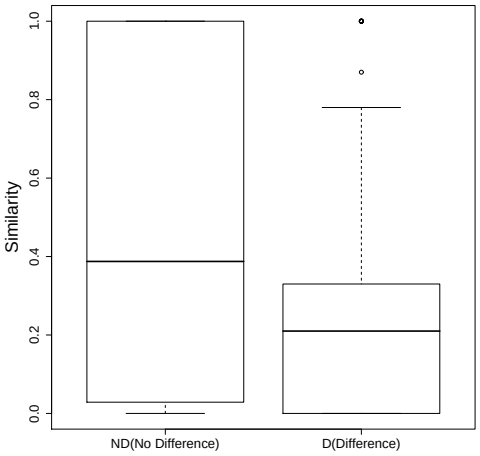
(a) LOC 平均



(b) LOC 標準偏差

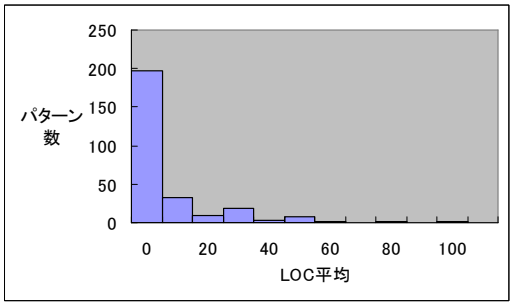


(c) 識別子名類似度 Sim_I

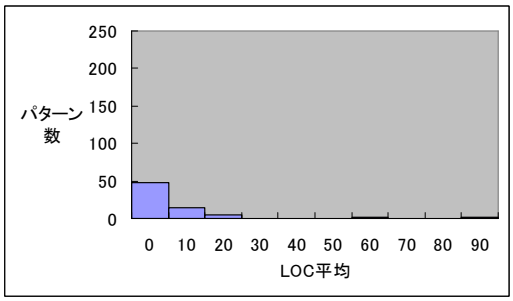


(d) 型名類似度 Sim_T

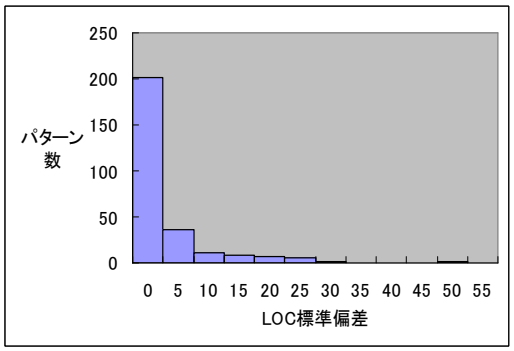
図 12: 各集合で計測されたメトリクス値の分布



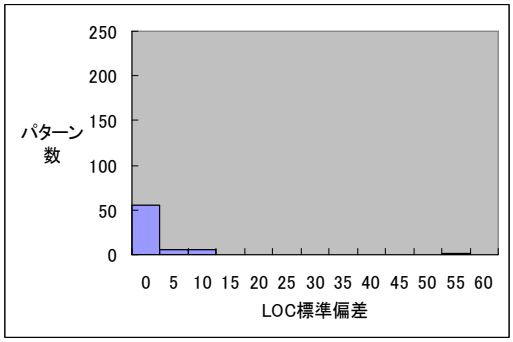
(a) LOC 平均の分布 (変更なし)



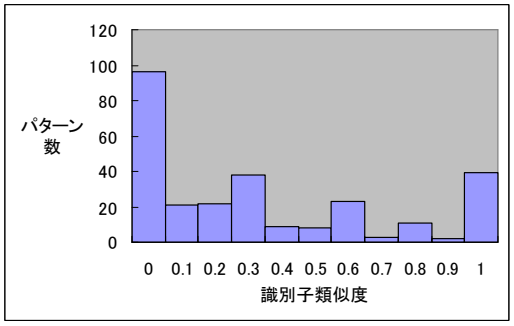
(b) LOC 分散の分布 (変更あり)



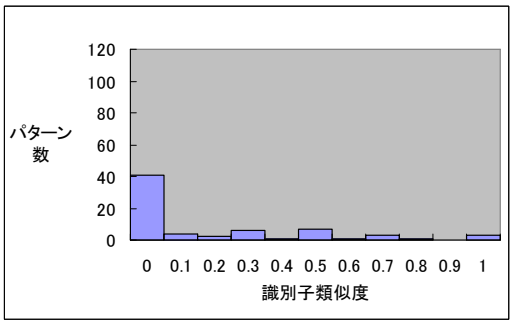
(c) LOC 標準偏差の分布 (変更なし)



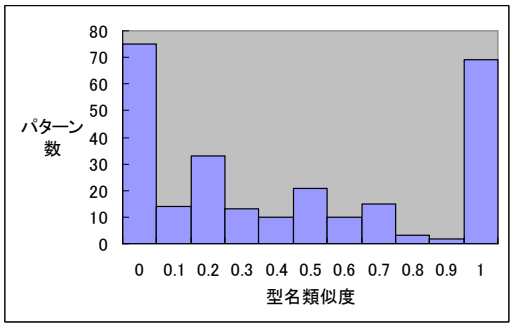
(d) LOC 標準偏差の分布 (変更あり)



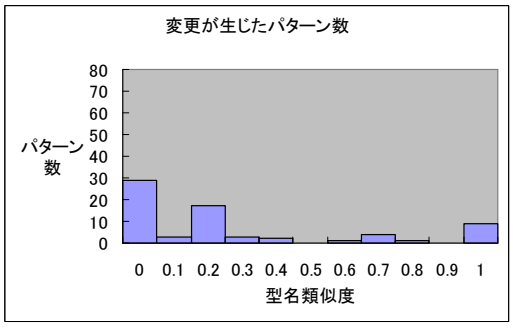
(e) 識別子名類似度の分布 (変更なし)



(f) 識別子名類似度の分布 (変更あり)



(g) 型類似度の分布 (変更なし)



(h) 型類似度の分布 (変更あり)

図 13: 計測された各メトリクスの分布

4.3 考察

本節では，4.2.2 節で計測したメトリクス値が，パターンの変更要因と関連があるかを考察する．また，変更が生じたパターンと生じなかったパターンに対して，実際にどのような変更であったかを述べる．

4.3.1 メトリクス値の分布

計測した４つのメトリクス値が，変更が生じたパターン集合 D と変更が生じなかったパターン集合 ND とで差異があるかを検討する．本節では，検定によって差異の有無を検討する．

使用する検定 図 13 で示されたメトリクス値の分布が正規分布であるということを言えないので，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 検定を使用する．変更の生じたパターン集合 D の代表値を R_1，変更の生じなかったパターン集合 ND の代表値を R_2 とすると，帰無仮説，対立仮説は以下のように設定される．

帰無仮説 $R_1 = R_2$

対立仮説 $R_1 < R_2$(片側検定)

検定結果 各メトリクス値の分布に対する検定結果を表 3 に示す．表中の P 値が小さいほど，各分布の大きさに差異があることを表す．表 3 中の結果では，LOC 平均に対して，変更が生じたパターン集合 D と変更が生じなかったパターン集合 ND の間で有意差は

表 2: 検出された Template Method パターンと Primitive Operation の数

ソフトウェア名	TMP_{ALL}	TMP_{ND}	TMP_D	PO_{ALL}	PO_{ND}	PO_D
Ant	6	4	2	7	5	2
ANTLR	1	1	0	1	1	0
ArgoUML	84	67	17	120	98	22
FindBugs	21	18	3	29	25	4
JEdit	10	3	7	11	4	7
JFreeChart	33	25	18	46	26	20
JRefactory	17	16	1	41	40	1
JHotDraw	37	26	11	50	38	12
Log4j	11	10	1	12	11	1

得られなかった．しかし，識別子名類似度，型名類似度に対しては，危険率 1 %でも有意差が得られた．これより，変更の生じたパターン集合 D の識別子名類似度，型名類似度のメトリクス値は，変更が生じなかったパターン集合 ND のメトリクス値よりも低い傾向にあることが分かった．

次に，計測したメトリクスが変更の要因と関係していることを確かめるために，識別子名類似度 Sim_I と型名類似度 Sim_T が大きい場合と小さい場合にどのような要因で変更が生じたかを調べる．

4.3.2 集合 D に属し識別子名類似度 Sim_I と型名類似度 Sim_T が小さい場合

類似度が小さく，変更が生じたパターンの例を示す．対象となるソースコードと変化の種類を以下に示す．

対象ソフトウェア ArgoUML

変化の種類 Primitve Operation の引数の型が変化

まず，変更が生じる前の Primitive Operation のメソッド定義と，Template Method のソースコードと，このクラスを継承する子クラスのソースコードを示す．また，このクラスは以下で示した子クラス以外の多くのクラスから継承されていた．

表 3: 各メトリクス値における検定結果

メトリクスの種類	統計量	P 値
LOC 平均 LOC_a	1.4786	0.1392
LOC 分散 LOC_v	1.9894	0.0467
識別子名類似度 Sim_I	2.786	0.0053
型名類似度 Sim_T	3.1996	0.0014

リスト 1: UMLModelElementListModel2.java

```
1     //Primitive Operation
2     protected abstract boolean isValidElement(MBase element);
3
4     //Tempate Methodの定義. 内部でPrimitiv Operationが呼ばれている
5     protected boolean isValidEvent(MElementEvent e) {
6         boolean valid = false;
7         if (!(getChangedElement(e) instanceof Collection)) {
8             valid = isValidElement((MBase) getChangedElement(e));
9             ....
10        }
11        return valid;
12    }
```

リスト 2: UMLAssociationAssociationRoleListModel(子クラス 1)

```
1     ....
2     protected boolean isValidElement(MBase o) {
3         return o instanceof MAssociationRole && ((MAssociation)
4                 getTarget()).getAssociationRoles().contains(o);
5     }
```

リスト 3: UMLAssociationEndAssociationListModel(子クラス 2)

```
1     protected boolean isValidElement(MBase element) {
2         return element instanceof MAssociation &&
3                 ((MAssociationEnd) _target).getAssociation().equals(element);
4     }
```

次に，変化が起こった後のソースコードを示す.

リスト 4: UMLModelElementListModel2.java(変更後)

```
1   //Primitive Operationのメソッド定義. 引数の型が変わった.
2    protected abstract boolean isValidElement(Object/*MBase*/ element);
3
4    protected boolean isValidEvent(MElementEvent e) {
5         boolean valid = false;
6         if (!(getChangedElement(e) instanceof Collection)) {
7             valid = isValidElement(/*(MBase)*/getChangedElement(e));
8             ...
9         }
10        return valid;
11    }
```

リスト 5: UMLAssociationAssociationRoleListModel(子クラス 1)

```
1       protected boolean isValidElement(Object/*MBase*/ o) {
2           return Model.getFacade().isAAssociationRole(o) &&
3               Model.getFacade().getAssociationRoles(getTarget()).contains(o);
4       }
```

リスト 6: UMLAssociationEndAssociationListModel(子クラス 2)

```
1       protected boolean isValidElement(Object/*MBase*/ element) {
2           return Model.getFacade().isAAssociation(element) &&
3               Model.getFacade().getAssociation(getTarget()).equals(element);
4       }
```

この変化において，Primitive Operation にあたるメソッドの定義が MBase 型から Object 型へ変更されている．これは変化前の子クラスが instanceof 演算子を用いた実行時の型情報に応じた処理を行っており，この型情報の比較をやめるために引数の型が変わったと考えられる．子クラスがそれぞれ異なる型情報を使用しているため，型類似度が小さくなっており，このクラスの変更は子クラスの処理の差異から起因していると判断できる．

4.3.3 集合 D に属し識別子名類似度 Sim_I と型名類似度 Sim_T が大きい場合

類似度が高いにも関わらず，変更が生じたパターンの例を示す．対象となるソースコードと変化の種類を以下に示す．

対象ソフトウェア JFreeChart

変化の種類 Template Method が消滅した

まず，変更が生じる前の Primitive Operation のメソッド定義と，Template Method のソースコードと，このクラスを継承する子クラスのソースコードを示す．この親クラスを継承している子クラスは，ここで示す 2 つのクラスだけであった．

リスト 7: ValueAxis.java

```
1       //Primitive Operationの定義
2       public abstract double translateValueToJava2D(double value,
3               Rectangle2D area,
4               RectangleEdge edge);
5
6       //Template Method 内部でPrimitive Operationを呼び出している
7       public double translateJava2DtoValue(float java2DValue,
8               Rectangle2D dataArea,
9               RectangleEdge edge) {
10
11          return translateJava2DToValue(java2DValue, dataArea, edge);
12      }
```

リスト 8: DateAxis.java(子クラス１)

```
1       public double translateValueToJava2D(double value,
2           Rectangle2D area,RectangleEdge edge) {
3           return valueToJava2D(value, area, edge);
4       }
```

リスト 9: NumberAxis(子クラス２)

```
1       public double translateValueToJava2D(double value,
2           Rectangle2D area, RectangleEdge edge) {
3           return valueToJava2D(value, area, edge);
4       }
```

次に，次のバージョンでのソースコードを示す．次バージョンでは，Template Method パターンが消滅したので，Template Method の役割を持っていたメソッドのみを示す．

リスト 10: ValueAxis.java(変更後)

```
1       //前のバージョンでTemplate Methodの役割を持っていたメソッド
2       public double translateValueToJava2D(double value,
3               Rectangle2D area,
4               RectangleEdge edge) {
5           return valueToJava2D(value, area, edge);
6       }
```

この例では，それぞれの子クラスの処理内容も短く，識別子名類似度・型名類似度が共に高いが，変更が生じている．このクラスの Primitive Operation は他のメソッドを１つ呼び出しているだけであり，このメソッド自体はその他の処理をしていない．このような他のメソッドを呼び出すだけの単純なメソッドは，識別子名類似度や型名類似度が高いにも関わらず変更されたり除去されたりする可能性があると考えられる．

4.3.4 考察のまとめ

Template Method パターンの変更要因を調べるため，4 つのメトリクス (LOC 平均，LOC 分散，識別子名類似度，型名類似度) を計測したが，そのうち識別子名類似度と型名類似度が，変更が生じたパターン集合のほうが変更の生じなかったパターン集合よりも小さくなるという傾向があった．また，実際に識別子名類似度・型名類似度が小さい場合でどのような変更が生じたかを調べると，クラスの変更要因の１つとして類似度が影響すると考えられた．また，クラスの規模が小さい場合には，類似度が高いにも関わらず，変更されるような場合も確認された．これにより，識別子名類似度と型名類似度はクラスの変更に起因していると考えられる．

5　関連研究

デザインパターン検出を用いた研究として，各バージョンでクラスの数と，デザインパターンの適用事例の数を計測し，その数の増減によって，ソフトウェアの進化の種類を判断する研究が行われている [8]．デザインパターンの適用事例の数が増加すれば，その修正はリファクタリング (プログラムの振る舞いを変えることなく内部構造を改善すること) である可能性が高くなる傾向がある．またクラス数のみ増加して，デザインパターンの適用事例の数が変化しなければ，機能追加であるという傾向を示している．

また，Bieman らはクラスの規模やデザインパターンの適用の有無と，クラスの変更要因との関連を調査している [5][6]．Bieman らは以下の仮説を調査した．

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変更され，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．

また，後の研究 [6] で 3 つの商用ソフトウェアと 2 つのオープンソースソフトウェアを対象に同様の調査を行っている．商用システムでは，メソッド数が多いほど変更数が多くなったが，オープンソースソフトウェアでは逆の結果が報告されている．また 4 つのソフトウェアにおいて，デザインパターンを適用したクラスのほうがそうでないクラスよりも変更頻度が高かったことが示されている．

6　今後の課題

本研究において考えられる課題とその解決策について述べる．

検出するパターンの変更数の増加 本研究では複数箇所に修正範囲が及ぶ変更を定義し，それらの変更が生じたパターンを調査したが，検出できたパターンの数は 30 個程度であった．検出したパターンの数を増加させる方法として，異なるデザインパターン検出ツールを用いることが考えられる．本研究では Tsantalis らのデザインパターン検出ツールを用いたが，必ずしも全てのパターンを検出できるわけではない．そのため異なるデザインパターン検出ツールを用いることで，検出数を増加させることができる．

他のデザインパターンへの適用 Template Method パターンはソースコード中で頻繁に出現し，構造も簡単であるので本研究の対象とした．Template Method パターンは他のパターンの一部として使用されることがあるので，Factory Method パターンなど，他のパターンにも提案手法を適用でき，変更要因が調査できると考えられる．

計測するメトリクスの改良 本研究で提案した識別子名類似度・型名類似度のメトリクスは，Template Method パターンを含むクラスを継承する子クラスの数の影響を受けやすいという欠点を持つ．なぜならオーバーライドする子クラスの数が多いと，共通する識別子名や型名の割合が低くなり，類似度が低くなりやすいからである．子クラスの数による影響を受けないメトリクスや他のメトリクスを計測することによって，よりクラスの変更と関連した値を得ることができると考えられる．

ソフトウェア変更予測への応用 本研究で計測したメトリクスが，クラスの変更との相関があるという結果が得られたが，この結果を応用して，ソフトウェアが次のバージョンで変更されやすいかを予測することが望ましい．そのためには，先に述べた検出パターンの増加や計測するメトリクスの改良が必要になると考えられる．

7　むすび

本研究では，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．

本研究では Template Method パターン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．そこで４つのメトリクス (LOC 平均，LOC 分散，識別子名類似度，型名類似度) を Java 言語で記述された８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などが挙げられる．

謝辞

本研究の全過程を通して，常に適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝いたします.

本研究を通して，随時適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授に深く感謝いたします.

本研究において，常時適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾 隆 助教に深く感謝いたします.

本研究に対して，逐次適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 肥後 芳樹 助教に深く感謝いたします.

本研究に対して，適時適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 早瀬 康裕 特任助教に深く感謝いたします.

本研究に対して，終始適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 吉田 則裕 氏に深く感謝いたします.

本研究での特にプログラム作成にあたって，適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 三宅 達也 氏に深く感謝いたします.

本研究での特に論文執筆にあたって，懇切丁寧な添削および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 吉田 昌友 氏に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします.

参考文献

[1] Ant. `http://ant.apache.org/`.

[2] ANTLR. `http://www.antlr.org/`.

[3] ArgoUML. `http://argouml.tigris.org/`.

[4] L. Aversano, G. Canfora, L. Cerulo, C. D. Grosso, and M. D. Penta. An empirical study on the evolution of design patterns. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-ESE'07)*, pp. 385–394, Dubrovnik, Croatia, 2007.

[5] J. M. Bieman, D. Jain, and H. J. 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.

[6] J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. 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.

[7] Findbugs. `http://findbugs.sourceforge.net/`.

[8] K. Fushida, S. Kawaguchi, and H. Iida. A method of investigate software evolutions using design pattern detection tool. In *Proceedings of the 1st International Workshop on Software Patterns and Quality(SPAQu '07)*, pp. 11–16, Nagoya, Japan, 2007.

[9] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[10] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, 本位田真一 (監訳). オブジェクト指向における再利用のためのデザインパターン. ソフトウェアパブリッシング, 1999.

[11] Y. G. Gueheneuc and G. Antoniol. Demima: A multilayered approach for design pattern identification. *IEEE Transactions on Software Engineering*, Vol. 34, No. 5, pp. 667–684, Sept.-Oct. 2008.

[12] JFreeChart. http://www.jfree.org/jfreechart/.

[13] JHotDraw. http://www.jhotdraw.org/.

[14] JRefactory. http://jrefactory.sourceforge.net/.

[15] S. Kim, E. J. Whitehead, and J. Bevan. Analysis of signature change patterns. In *Proceedings of the International workshop on Mining software repositories (MSR '05)*, pp. 1–5, St. Louis, MO, 2005.

[16] Log4J. http://logging.apache.org/log4j/.

[17] T. H. Ng, S. C. Cheung, W. K. Chan, and Y. T. Yu. Work experience versus refactoring to design patterns: a controlled experiment. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14)*, pp. 12–22, Portland, OR, USA, 2006.

[18] J. Niere, W. Schäfer, J. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pp. 338–348, Orlando, FL, USA, 2002.

[19] Apache Jakarta Project. http://jakarta.apache.org/.

[20] N. Shi and R. A. Olsson. Reverse engineering of design patterns from java source code. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*, pp. 123–134, Tokyo, Japan, 2006.

[21] SourceForge. http://sourceforge.net/.

[22] S. Steliting and O. Maassen. *applied JAVA PATTERNS*. Sun microsystems, 2002.

[23] Design Pattern Detection Tool. http://java.uom.gr/~nikos/pattern-detection.html.

[24] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, Vol. 32, No. 11, pp. 896–909, 2006.

[25] 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.