卒 業 研 究 報 告 書

題　目　　複数の静的解析ツールによる
　　多数決を利用した不具合判別法の提案

指導教員　　水野 修　准教授

京都工芸繊維大学 工芸科学部 情報工学課程

学生番号　　08122502

氏　　名　　小野木　祐太

平成24年2月15日提出

複数の静的解析ツールによる 多数決を利用した不具合判別法の提案

平成 24 年 2 月 15 日　　08122502　小野木　祐太

概　要

開発の早期段階でソースコード中の fault-prone モジュールを特定することはプロダクトの品質向上につながる．これまでにも fault-prone モジュールを予測する多くの研究が行われてきたが，それらは全てメトリクスベースによるもので，ソフトウェアメトリクスの測定に余分な工数やコストがかかってしまう場合もある．我々は，簡単に適用でき，かつ効果の高い fault prone モジュールの検出手法として，spam フィルタに基づく fault prone モジュール検出法「fault prone フィルタリング」を提案している．この手法は全く事前の知識がない状態からでも開発プロジェクトに適用できるという特徴を持っている．

こうした fault-prone モジュール予測手法は，ソフトウェアのモジュールを単位として予測を実施する．しかし，このモジュールの単位が大きすぎるとせっかく不具合があると予測できても、そのモジュール内のどこにあるのかを特定できない．そのため，細粒度のモジュールを利用した不具合予測が必要とされている．

本研究では，細粒度モジュールを扱えるリポジトリを利用して，fault-prone filtering 法を細粒度モジュールに対して適用する．その際，研究設問として，細粒度モジュールによる予測は従来の予測精度よりも良くなるか否か，そして，それは実用的な値が得られるかどうかを調査する．

実験の結果，予測精度はやや低くなるものの，その値は高い水準を維持しており，実用的に意味のある値であることを確認した．この結果，細粒度モジュールによる fault-prone モジュール予測は，早期の不具合発見に有効であることを確認した．

目　次

1. 緒言

ソフトウェア開発において，開発が大規模になればなるほどモジュールに不具合が混入することは避けられないことであり，いかにして不具合を取り除くかがソフトウェア開発を行う上で重要な鍵となる．不具合を含みそうなモジュール（Fault-proneモジュール）を予測することができれば，不具合を効率よく取り除くことができ，ソフトウェア開発にかかるコストを削減することができると考えられている．そのため，いかにFault-proneモジュールを高い精度で予測するかという研究が数多く行われている．

従来の研究では，ソフトウェアの不具合はソフトウェアの構造の複雑さに関連していると仮定し，ソフトウェアの複雑さを定量化したメトリクスを利用することによって，不具合を推定するモデルを構築していた．これに対し，我々はFault-proneフィルタリング [1] という手法を提案している．この手法では，不具合はソフトウェアモジュール中に含まれる語や文脈に関連していると考え，与えられるモジュールをテキスト情報として扱う．そして，スパムフィルタリングで利用されるベイズ理論を用いて，あらかじめ不具合が混入していたものを分類して学習させた辞書に基づいてあるモジュールに対する不具合混入確率を計算する．この確率に基づいて，与えられたモジュールがFault-proneモジュールであるのか，否かを判定する．

この手法に対して，静的コード解析ツールをソースコードに適用し，そこで出力される警告メッセージを利用してFault-proneモジュールの予測を行う試みが提案されている [2]．これまで行われてきたFault-proneフィルタリングの研究では，学習・分類の対象となるモジュールは常にソースコードモジュールであった．これに対して，この手法では静的解析ツールが出力する警告メッセージというソースコードとは全く違うものを用いた予測を行っている．その結果，ソースコードを入力とした場合とほとんど同等の予測精度が得られることを確認している．

この研究では，この手法に着想を得て，複数の静的コード解析ツールを利用した不具合予測を行う手法の提案を行う．具体的には2つの静的コード解析ツールを利用して，それぞれが出力した警告メッセージに対して予測を行い，これにソースコードを用いたオリジナルの手法の予測結果を合わせた3系統の予測結果を得る．そして，3系統の多数決を採ることで，最終的は不具合の予測を行おうとするもの

である．

この手法の提案のために，本研究では 2 つの静的コード解析ツール PMD [3] と CheckStyle [4] を用いた．これらはいずれも静的コード解析ツールとしては広く利用されており，また，ソースコードに対して直接適用可能であるという特徴を持つ．これらを用いた実験を通じて，多数決による予測が有効かどうかを分析する．

本報告の以降の構成を述べる．第 2 章では，静的コード解析の概要や解析ツールである PMD の概要及び利用できるルールセットについて説明する．第 3 章では，Fault-prone フィルタリングの概要や学習・分類ステップ，Fault-prone フィルタの実装方法等について説明する．第 4 章では，Fault-prone フィルタリングに PMD の出力をモジュールとして与えて実験を行う．実験対象や実験方法，評価指標など，そして実験結果について説明する．第 5 章では，実験結果に対する考察を述べる．最後に第 6 章では，本研究のまとめ及び，今後の課題について述べる．

2. 静的コード解析

2.1 静的コード解析の背景

静的コード解析とはソフトウェアの解析手法の 1 つであり，ソフトウェアを実際に実行せずに解析を行い，プログラムの問題点や不具合を発見する手法である．解析は一部例外としてオブジェクトコードに対して行うものがあるが，基本的にソースコードに対して行われる．ソースコードを構造的に解析することで，潜在的な不具合や保守性の低下の原因となるコーディング規約違反，性能劣化の原因となるコードなどを検出する．静的コード解析は，ソフトウェアの安全性や信頼性，保守性などを向上させる手法であり，ソフトウェア品質向上や，保守にかかるコストの削減を目指すことができる．

こういったプログラムの問題点や不具合を発見する作業はテストの役割だと思われがちだが，テストでは発見できないものも存在する．テストは設計どおりにソースコードが組まれているか，つまり不具合が存在しないかを確認する作業とも言えるためである．例えば，ソースコードのコーディング規則違反や，保守性に関する問題も多くがテストでは発見できない．

また，性能劣化の問題などは大抵の場合すべてのソフトウェアを結合し，テストを行うことで発見されるものであるので，そのときになって問題を発見していたのでは，手戻りが非常に多くなってしまう．そのため，静的コード解析を開発早期に用いることでテストの前に各種問題や不具合を発見し，開発にかかるコストや時間を削減することに繋がる．

近年，潜在するセキュリティホールの検出などの必要性が増していることで，静的コード解析が重要視されている．静的コード解析を行うツールは，非商用，商用様々で，対象となるプログラミング言語も様々なものがある．

2.2 PMD

静的コード解析ツールの 1 つとして，PMD [3] というソフトウェアが存在する．PMD はオープンソースとして提供されているソフトウェアで，Java を用いて実装されている．対象言語は Java である．PMD は，Java のソースコードを解析して，未使

用の変数や空の catch ブロック，到達不可能なステートメントなどの，潜在する不具合の原因となるコードを検出することができる．PMD には多種多様なルールセットが用意されており，使用するルールセット次第でコーディング規約の検査から，潜在的な不具合を検出まで幅広い用途で使用することができる．

また，独自のルールセットを作成することができるため，特殊なルールを使用することも可能である．なお，ルールセットの作成には XPath か Java のクラスファイル形式を用いる．PMD にはオープンソースの統合開発環境である Eclipse のプラグインも公開されており，Eclipse で Java 開発を行う際に利用すると効果的に用いることができるとされている．

2.2.1 PMD ルールセット

PMD に標準で用意されているルールセットの中で，本研究では以下に示す 10 個のルールセットを利用する．これらのルールはソフトウェアの品質を調べる観点でよく利用されるものとなっている [5].

Basic

もっとも基本的なルールセット．catch，if，while，switch などの空ブロックや必要のない文字列変換処理，連続して記述された単項演算子，IP アドレスのハードコーディングの検出など，幅広く多種多様なルールが用意されている．

Braces

括弧に関するルールセット．if や else，for や while などステートメントに対して，括弧が使用されてないものを検出するルールが用意されている．

Code Size

コードのサイズに関するルールセット．複雑なメソッドや異常に長いメソッドの検出や，大量の public なメソッドやフィールドを含むクラスなど検出するルールが用意されている．

Coupling

カップリングに関するルールセット．オブジェクト-パッケージ間のカップリングが高すぎたり，不適切なカップリングがないかを検査する．例えば，大量のインポート・ステートメントが定義されているクラスや，使用しているフィールドやローカル変数，戻り値の型の種類が多すぎるクラス，インタフェースではなく実体クラスを宣言しているなどの検出を行うルールが用意されている．

Design

コードの設計に関するルールセット．設計を良くするための様々な原則を検査する．深すぎる if ブロックのネストや，default ブロックのない switch ステートメント，メソッドのパラメータへの代入などを検出するルールが用意されている．

Naming

標準的な名前付けに関するルールセット．ループ変数以外で極端に短い，または長い名前の変数，クラス名と同名のメソッド，先頭が大文字でないクラス名などを検出するルールが用意されている．

Optimizations

最適化に関するルールセット．1 回しか代入されないのに final として宣言されていないローカル変数，ループ内でのオブジェクトの生成，不必要なラッパーオブジェクトの使用などを検出するルールが用意されている．

Strict Exception

例外に関する厳密なルールセット．フロー制御に例外を使用している，Throwable を捕らえている，Exception を投げている，java.lang.Error を継承したクラスなどを検出するルールが用意されている．

Strings

文字列に関するルールセット．重複した文字列定義，String インスタンスの生成，String オブジェクトに対する toString() の呼び出し，StringBuffer インスタンスの生成方法が適切でないところなどを検出するルールが用意されている．

Unused Code

使用されていないコードに関するルールセット．使用されないローカル変数や，private フィールド，メソッドなどを検出するルールが用意されている．

2.3 CheckStyle

Checkstyle も静的コード解析ツールの一種で有り，コーディング標準に従った Java コードを書くことを支援するツールである．このツールの利用により，Java コードをチェックするプロセスを自動化することが可能となっている．このため Checkstyle はコーディング標準を強制したいと考えているプロジェクトにとって理想的なものになっており，Eclipse のプラグインとして提供されている最も有名なコード解析ツールの 1 つである．

Checkstyle は歴史的にはコードのレイアウトをチェックすることが主目的とされてきたが，それ以外にもソースコードのさまざまな側面をチェックすることができる．多くのコード標準に対応するため，Java においては必須の静的解析ツールの一つであると言える．

Checkstyle はさまざまな構成設定が可能であり，設定次第ではどのようなコーディング標準でもサポートしている．標準では，「Sun コーディング規約 [6]」をサポートする構成ファイルのサンプルが提供されており，他にもよく知られた規約のサンプル構成ファイルも同梱されている．

本研究では，Sun コーディング規約の構成ファイルを利用した実験を行う．この規約が最も標準的なコード規約の 1 つであることから，本研究での利用に適していると考えた．

3. Fault-proneフィルタリング

3.1 概要

Fault-proneフィルタリングはスパムメール(迷惑メール)の判別をおこなうスパムフィルタで用いられるテキスト分類フィルタ技術を利用する．スパムフィルタは，過去に受信した電子メール内の単語群を利用して，スパムメールと通常のメールを判別するための辞書を作成する．そして，新たに受信した電子メールについては，ベイズ識別などの技術により，スパムか否かを判定する．学習は随時行われ，辞書は常にその時点の状況を反映したものになるため，新種のスパムメールなどにも柔軟に対応できるとされている．この考えは，スパムメールには特定の単語群や文章が頻繁に含まれている，という事実に基づいている．

我々は，この考え方がソースコード内の不具合についても適用できるのではないかと考えた．もちろん，元々悪意を持って作成されたスパムメールと，意図的ではないがバグが混入したソースコードを全く同じものと見なすのは無理があるかもしれない．しかし，一連のソフトウェア開発においては，同じ開発者が同じ文脈でバグを混入することや，類似の関数やAPIの呼び出しなどにおいてバグを混入してしまうことは良くあることだと考えられる．すなわち，スパムメールの中の特定の単語のように，バグが存在するところには特定のコード片が存在するのではないかと類推した．

3.1.1 スパムフィルタ: CRM114

本研究ではテキスト分類フィルタとしてCRM114を用いた[7]．主にスパムフィルタとして開発されているが，汎用的な用途，例えば，計算機のログ監視やネットワークのトラフィック監視などにも活用できるとされている．また，現在開発されているメールフィルタの中でも高い予測精度をあげているものの1つである．

CRM114は基本的にはベイズ識別を利用したテキスト分類フィルタであるが，複数の単語を組み合わせたものをトークンと呼び，学習・分類の単位として利用することが大きな特徴である．従来のテキスト分類フィルタは1単語をトークンとしているのに対し，複数単語の組をトークンとすることで，より複雑な学習が可能と

表 3.1　OSB で生成されるトークン例

a	=			
a	b			
a		+		
a			1	
	=	b		
	=		+	
	=			1
	=			return
		b	+	
		b		1
		b		return
		b		a
			+	1
			+	return
			+	a
				1 return
				1 a
				return a

なっている．本研究では，CRM114 のデフォルトの分類手法である “Othogonal Sparce Bigrams Marcov model (OSB)” を使用する．OSB は任意の連続する 5 単語の組合せのうち，2 単語からなるものだけをトークンとする手法である．

OSB によるテキスト処理について以下に簡単に示す．表 3.1 は “`a = b + 1 ; return a;`” という文について，トークンを生成した様子である．OSB ではある単語を起点として 5 単語からなる単語列に対し，正確に 2 単語のみを含むもののみを学習・分類の対象として抽出する．なお，本研究ではプログラム言語中の区切り文字をあらかじめ排除するため，“;” は単語群に含まれていない．

3.1.2 利用したツール

本研究では，研究室において開発・保守されている fault-prone filtering 用ライブラリ (`FPFilter.pm`) を利用した．FPFilter.pm は perl 言語で書かれたライブラリで，上記 CRM114 をソースコードの予測に特化した形で利用できるようになっている．

3.2 Fault-prone フィルタの実装

本実験では，文献 [1] で用いられた Fault-prone フィルタを使用する．この Fault-prone フィルタは CRM114 を利用して作成されたものであるが，基本的な学習ステップおよび分類ステップのアルゴリズムは文献 [8] で提案されたスパムフィルタの基礎理論を踏襲したものとなっている．

3.2.1 トークンの分割規則

Fault-prone フィルタリングの入力となるモジュールとして，本研究ではソースコード及び 2 種類の静的解析ツールの出力を用いる．それらは全てトークン単位に分割される必要がある．

(1) ソースコードモジュール

ソースコードモジュールは Java のコードが書かれている部分と，コメントが書かれている部分で構成されている．しかし，コメント部分はコードの部分と違い，厳密な構文が決まっておらず人間が自由に記述できるということに注意する必要がある．制約が緩いため，ダブルクォート等が正しく綴じられていない場合などが生じる．また，アポストロフィなのかシングルクォートで囲まれた文字列なのか判定が困難であると言ったような問題も生じる．そのため，ダブルクォートやシングルクォートで囲むといった規則が正しく機能しない可能性がある．したがって，コメントであるか否かでトークンの分割規則を変える必要がある．

そのためには，まずコメントである部分を検出する必要がある．Java 言語には次の 3 種類のコメントが存在する．

- “//”で始まるコメント

- “/*”で始まり，“*/”で終わるコメント
- “/**”で始まり，“*/”で終わるコメント

よって，モジュール中にこれらが検出されたときにコメントであると判断し，コメント用の規則を使用することとする．それ以外の部分はコード用の規則を使用することとする．

コードの部分のトークン分割規則は次のように定義する．

- アルファベット，数字，ドットからなる文字列
- Java 言語の演算子
- シングルクォートで囲まれた文字列
- ダブルクォートで囲まれた文字列

コメントの部分のトークン分割規則は上記の理由により，次のように定義する．

- アルファベット，数字，ドットからなる文字列
- Java 言語の演算子

(2) 静的解析ツールの出力

PMD の出力は英語で書かれた文章に所々Java のコードに関連する部分が含まれているものである．それらを混同しないようにするために次の 4 つの規則を考える．

以下にトークンの分割規則を定義する．

- アルファベット，数字からなる文字列
- 各種括弧とセミコロン，カンマ
- Java 言語の演算子とドット
- それ以外の文字列

3.2.2 CRM114 による分類の例

この節では Fault-prone フィルタリングがどのように不具合のあるソフトウェアモジュールを検出するのかを，単純な例を用いて説明する．図 3.1(a) と (b) はそれぞれ，不具合を含む (FP) モジュールと含まない (NFP) モジュールの例である．以降で

```
1: public int fact(int x) {
2:     return(x==1?1:x*fact(++x));
3: }
```

(a) 不具合を含む (FP) モジュール m_{FP}

```
1: public int fact(int x) {
2:     return(x==1?1:x*fact(--x));
3: }
```

(b) 不具合を含まない (NFP) モジュール m_{NFP}

図 3.1　FP と NFP モジュールの例

```
1: public int sigma(int y) {
2:     return(y==1?1:y+sigma(++y));
3: }
```

図 3.2　新しく作成したモジュール m_{new}

はそれぞれを m_{FP}，m_{NFP} と表記する．fact() は与えられた自然数 x に対してその階乗を返すことを意図しているが，図 3.1(a) の実装では--x とすべきところを++x と誤記しているため，正常に動作しない．図 3.1(b) はその不具合を取り除いた状態である．この 2 つのモジュールのみが辞書に学習された時点で，図 3.2 に示すモジュール m_{new} が新たに作成されたとし，このモジュールが不具合を含む確率を計算することを考える．なお，モジュール m_{new} は与えられた正整数 y に対してその総和を求めるつもりであるが，m_{FP} と同様に本来--とすべきところを++としている．

まず，m_{FP} と m_{NFP} をそれぞれ FP, NFP として学習する．この時，CRM114 によってそれぞれのモジュールについて図 3.3(a), (b) に示すようなトークンの集合 T^{FP}, T^{NFP} が生成される．m_{FP} から生成されたトークンの集合 T^{FP} は Fault-prone モジュールの特徴として FP 辞書に格納される．同様に，m_{NFP} のトークンの集合 T^{NFP} は，NFP 辞書に格納される．

新たなモジュールが与えられると，その時点で辞書に学習されている全てのトークンとのマッチングがとられ，確率の計算が行われる．図 3.3 は，m_{FP}, m_{NFP}, m_{new} について生成されるトークンの集合，T^{FP}, T^{NFP}, T^{new} を列挙したものである．図 3.3(a) と (b) が現時点でそれぞれ FP と NFP の辞書に格納されている全てのトークンであり，図 3.3(c) は新たに生成されたトークンである．下線を引いた部分はモジュー

ル m_{new} と同一のトークンであることを表す．この図から，m_{FP} と m_{new} の間で同一なトークンの数は 14 であり，m_{NFP} と m_{new} の間で同一なトークンの数は 13 であることが分かる．この情報から新規モジュール m_{new} が不具合を含む確率 $P(T^{FP}|T^{new})$ を算出する．ベイズの定理によって，確率 $P(T^{FP}|T^{new})$ は次のように求められる．

$$\frac{P(T^{new}|T^{FP})P(T^{FP})}{P(T^{new}|T^{FP})P(T^{FP}) + P(T^{new}|T^{NFP})P(T^{NFP})}$$

まず，学習されているのは T^{FP} と T^{NFP} だけなので，それぞれのトークンの存在する事前確率は $P(T^{FP}) = P(T^{NFP}) = 1/2$ となる．次に，m_{FP} のトークン T^{FP} 内に m_{new} のトークン T^{new} が存在する確率 $P(T^{new}|T^{FP}) = 14/45$ である．また，m_{NFP} のトークン T^{NFP} 内に T^{new} が存在する確率 $P(T^{new}|T^{NFP}) = 13/45$ である．よって，確率は次式で求められる:

$$P(T^{FP}|T^{new}) = \frac{\frac{14}{45} \times \frac{1}{2}}{\frac{14}{45} \times \frac{1}{2} + \frac{13}{45} \times \frac{1}{2}}$$

$$= 0.519$$

この結果，新規モジュール m_{new} が不具合を含む確率は 0.519 となる．本研究では確率の閾値を 0.50 と定め，0.50 以上であれば FP モジュール，0.50 未満であれば NFP モジュールと判定する．よって，この例では FP モジュールと判定されることになる．

3.3 静的解析ツールを利用した Fault-prone モジュールの予測

Fault-prone フィルタリングは，モジュール中から検出したトークンに基づいて分類を行うものであるが，過去の研究 [1] ではモジュール中の全てのトークンを用いて分類が行われていた．また，研究 [9] ではコメントに該当する部分のみのトークンに対して実験が行われたりもしている．

一方で，研究 [2] では静的コード解析ツール PMD の出力を Fault-prone フィルタリングの入力として与える手法が提案されている．PMD は元々ソフトウェアの不具合を未然に防ぐことを目的としたツールであるため，その出力は不具合を検出するための強力な特徴であると考えられる．しかし，一方で PMD の出力は冗長であり，かつ，非常に多くの情報を含むために本当に必要な情報が人間の目に入らないという問題を抱えている．そこで，テキスト分類をかけることにより，機械的に学習と予測を行う方法を提案している．この試みは一定の成果を挙げている．

本研究では，研究 [2] で用いられた PMD の他に，Checkstyle も用いたフィルタを実装する．

3.4 多数決を用いた Fault-prone モジュールの予測

本研究では，オリジナルの Fault-prone フィルタリング (以降，FPF と呼ぶ)，PMD を用いたフィルタリング (以降，PMD と呼ぶ)，そして，CheckStyle を用いたフィルタリング (以降，CHK と呼ぶ) の 3 系統の予測手法を用いた Fault-prone モジュールの予測を行う．

具体的には，各系統による予測を一通り実施し，それぞれの手法での不具合混入確率を計算する．そして，その確率に基づき，fault-prone(FP) か not fault-prone(NFP) かの判定を行う．そして，3 つの手法の判定結果を集め，多数決により最終的な予測結果を得る．

public	int
public	fact
public	x
int	fact
int	int
int	x
int	return
fact	int
fact	x
fact	return
int	==
x	return
x	x
x	==
x	1
return	x
return	==
return	1
return	?
x	?
x	:
==	1
==	?
==	:
==	x
1	?
1	:
1	x
1	*
?	:
?	x
?	*
?	fact
:	x
:	*
:	fact
:	++
x	*
x	fact
x	++
*	fact
*	++
*	x
fact	++
++	x

(a) m_{FP} から生成されるトークン T^{FP}

public	int
public	fact
public	x
int	fact
int	int
int	x
int	return
fact	int
fact	x
fact	return
int	==
x	return
x	x
x	==
x	1
return	x
return	==
return	1
return	?
x	?
x	:
==	1
==	?
==	:
==	x
1	?
1	:
1	x
1	*
?	:
?	x
?	*
?	fact
:	x
:	*
:	fact
:	--
x	*
x	fact
x	--
*	fact
*	--
*	x
fact	--
--	x

(b) m_{NFP} から生成されるトークン T^{NFP}

public	int
public	sigma
public	y
int	sigma
int	int
int	y
int	return
sigma	int
sigma	y
sigma	return
int	==
y	return
y	y
y	==
y	1
return	y
return	==
return	1
return	?
y	?
y	:
==	1
==	?
==	:
==	y
1	?
1	:
1	y
1	+
?	:
?	y
?	+
?	sigma
:	y
:	+
:	sigma
:	++
y	+
y	sigma
y	++
+	sigma
+	++
+	y
sigma	++
++	y

(c) m_{new} から生成されるトークン T^{new}

図 3.3　各モジュールに対して生成されるトークン

表 4.1　実験で用いるモジュールの個数

	Eclipse BIRT	
	評価用	学習用
hline 不具合を含まない	6268	36235
不具合を含む	746	26895
合計	7014	63130
	70144	

4. 適用実験

本実験では，前章で述べた多数決を用いた Fault-prone モジュールの予測手法の有効性を実験を通じて確認する．

4.1 実験対象

本実験では実験対象として次のオープンソースプロジェクトを用いる．

Eclipse BIRT Eclipse のビジネス文書作成プラグインであり，Java 言語を用いて開発されている．バージョン管理システムは CVS を使用している．

今回の実験では，十重交差検証を用いて予測精度の評価を行う．交差検証とは，対象となるデータの一部分を予測用に確保し，残りの部分を学習したモデルで予測を実施する実験をデータの全体が予測を網羅できるように実行する手法である．今回の実験では，全データを十等分した実験を行う．これを十重交差検証と呼ぶ．交差検証の 1 試行における利用モジュールの個数を表 4.1 に示す．

4.2 モジュールの取得

ソースコードモジュールの取得には，SZZ アルゴリズム [10] の一実装 [11] によって作成された SQLite3 のデータベースを用いる．

SZZ アルゴリズムとは，バージョン管理システムとバグ管理システムの情報をお互いに結びつけることで，自動的に不具合を含むモジュールを識別するアルゴリズ

ムである．ソフトウェア開発において，バージョン管理システムとバグ管理システムは重要なツールだが，今現在これらの 2 つのシステムの情報をお互いに結びつけることで，バージョン管理されているファイルの中から不具合を含むファイルを見つけるという機能は，これらの 2 つのシステムに含まれていない．SZZ アルゴリズムはその機能を提供する理論である．

このアルゴリズムの実装 [11] を利用すると，指定したオープンソースソフトウェアのソースコードリポジトリに対してリポジトリマイニングを行い，各リビジョンのファイル全てを取得し，さらにそれらのファイルに対して，その時点で不具合が混入されていたか否かのマークを付けたものをデータベースに格納する．

4.3 実験方法

ソースコードモジュールを入力とした分類実験と，静的解析の警告メッセージを入力とした分類実験との 2 つの実験方法について説明する．これらの 2 つの実験は手順に共通する部分が存在するため，以下に示すように 2 つの実験の手順を合わせて説明する．

4.3.1 FPF 法による予測

1. まず，SQLite3 を用いてプロジェクトのデータベースから全てのソースコードモジュールを取得する．この際，モジュールを十等分して学習用と評価用に分け，かつ，それぞれ不具合が含まれるモジュールと含まれないモジュールに分けておく．
2. 各モジュールをトークンに分割する．
3. 学習用のトークンに分割されたモジュールからトークンを抽出して，辞書に学習させる．この辞書は不具合を含むモジュールと含まないモジュール別々に作成する．
4. 作成した辞書を用いて評価用のトークンに分割されたモジュールを分類する．
5. これを全ての分割に対して実施する．

4.3.2 静的解析ツールによる予測

5. 静的解析ツールを用いて手順 1 で用意したモジュールに対して静的コード解析を行う．PMD を用いるときは，ルールセットに第 2.2.1 節で挙げた Basic, Braces，Code Size，Coupling，Design，Naming，Optimizations，Strict Exception, Strings，Unused Code の 10 個のルールを使用する．また，CheckStyle を用いる際は，Sun コーディング規約 [6] の構成を用いる．
   静的解析の出力にはファイル名と警告元の行番号も含まれているが，それらは今回の実験には不要であると考えられるため，取り除いておく．
6. ソースコードモジュールの代わりに手順 5 で作成した PMD の出力をモジュールとして手順 2〜4 を実行する．
7. ソースコードモジュールを入力した分類実験の結果と PMD の出力を入力とした分類実験の結果を比較する．

4.4 評価指標

本実験では予測の評価指標として精度（Accuracy），再現率（Recall），適合率（Precision），F_1 値, フォールス・ポジティブ率，フォールス・ネガティブ率の 6 つを用いる．

正答率（Accuracy）

正答率（Accuracy）とは，全モジュールに対して，実際に不具合を含むモジュールを Fault-prone，不具合を含まないモジュールを Fault-prone でないと正しく予測できた割合を示す．したがって，表 4.2 の凡例の値を用いると式 (4.1) のように定義される．

$$Accuracy = \frac{N_1 + N_2}{N_1 + N_2 + N_3 + N_4} \tag{4.1}$$

この値は用いるデータの偏りなどの影響を受けやすい指標である．そのため，この値のみではなく以下の再現率，適合率，F_1 値といった評価指標を併用する．

再現率（Recall）

再現率（Recall）とは，実際に不具合を含むモジュールを Fault-prone であると予測できた割合を示す．したがって，表 4.2 の凡例の値を用いると式 (4.2) のように定義される．

$$Recall = \frac{N_4}{N_3 + N_4} \tag{4.2}$$

この値は実際に存在する不具合のうち，どれだけのものを予測できるかということを示すもので，非常に重要な評価指標である．

適合率（Precision）

適合率（Precision）とは，Fault-prone であると予測したモジュールの内，実際に不具合を含んでいたモジュールの割合を示す．したがって，表 4.2 の凡例の値を用いると式 (4.3) のように定義される．

この値が低いということは，実際には不具合でないものを Fault-prone であると誤った予測をする確率が高いことを示し，本来調べる必要のないモジュールを調べる手間が増えることに繋がる．そのため，この値は不具合を発見するために必要なコストを表す指標であるといえる．

$$Precision = \frac{N_4}{N_2 + N_4} \tag{4.3}$$

F_1 値

再現率と適合率の調和平均を F_1 として式 (4.4) と定義する．

$$F_1 = \frac{2 \times Recall \times Precision}{Recall + Precision} \tag{4.4}$$

再現率と適合率はトレードオフの関係にあるため，両者を総合的に判断するための指標としてこの F_1 値を用いる．

フォールス・ポジティブ率（False_p）

フォールス・ポジティブとは，誤検出，すなわち，本来検出するべきでないものを誤って検出してしまうことを意味する．ここで用いるフォールス・ポジティブ率

(False_positive) とは誤った検出が起きた割合とし，表 4.2 の凡例の値を用いて式 (4.5) と定義する．

本実験では式 (4.5) は全モジュール中で，Fault-prone モジュールであると予測されたにもかかわらず，不具合を含まないモジュールだった確率を示す．

$$False\_p = \frac{N_2}{N_1 + N_2 + N_3 + N_4} \tag{4.5}$$

この値は実際に Fault-prone モジュールをどれだけ誤検出したかを示す．

フォールス・ネガティブ率（False_n）

フォールス・ネガティブとは，検出漏れ，すなわち，本来検出するべきのものを誤って検出しないことを意味する．ここで用いるフォールス・ネガティブ率（False_negative）とは誤って検出しなかった割合とし，表 4.2 の凡例の値を用いて式 (4.6) と定義する．

本実験では式 (4.6) は全モジュール中で，Fault-prone モジュールではないと予測されたにもかかわらず，不具合を含むモジュールだった確率を示す．

$$False\_n = \frac{N_3}{N_1 + N_2 + N_3 + N_4} \tag{4.6}$$

この値は実際に不具合を含むモジュールをどれだけ見逃したかを示す．

4.5 実験結果

表 4.3〜表 4.6 に，FPF 法による予測結果，PMD 法による予測結果，CHK 法による予測結果，多数決による予測結果をそれぞれ示す．また，それぞれの結果において評価指標を計算したものを表 4.7 に示す．

表 4.2　実験結果の凡例

		予測	
		Not Fault-prone (NFP)	Fault-prone (FP)
実測	不具合を含まない (NFT)	N_1	N_2
	不具合を含む (FT)	N_3	N_4

表 4.3　Fault-prone filtering 法による予測結果

FPF		予測	
		NFP	FP
実測	NFT	15,609	26,891
	FT	3,571	24,069

表 4.4　PMD の警告メッセージを利用した予測結果

PMD		予測	
		NFP	FP
実測	NFT	10,658	31,842
	FT	924	26,716

表 4.5　CheckStyle の警告メッセージを利用した予測結果

CHK		予測	
		NFP	FP
実測	NFT	15,413	27,087
	FT	1,372	26,268

表 4.6　3 手法の多数決を利用した予測結果

多数決		予測	
		NFP	FP
実測	NFT	12,998	29,502
	FT	1,091	26,549

表 4.7　それぞれの手法を比較した評価指標

手法	正答率	再現率	適合率	F_1	$False\_p$	$False\_n$
Fault-prone Filtering	0.566	0.871	0.472	0.612	0.383	0.051
PMD	0.533	0.967	0.456	0.620	0.454	0.013
CheckStyle	0.594	0.950	0.492	0.649	0.386	0.020
3 手法の多数決	0.564	0.961	0.474	0.634	0.421	0.016

5. 考察

実験結果より読み取れた特徴について記す。表 4.3〜表 4.6 より、各手法による出力結果は似たような特性を持っていることが分かる。どの手法も N_2 の値が高く、N_3 の値が低いという結果になっている。これは FP と予測したものが必ずしも正しく予測できているわけではないが，NFP と予測したものについては高い確率で正しく予測できるということを示している。またこれらの結果より、静的解析における今後の課題として、N_2 を下げることが一番の課題として挙げられるのではないかと考えられる。

表 4.7 より、正答率，適合率，$F1$ 値，$Faulse_N$ では CheckStyle が一番良い値を示す結果となり、再現率，$Faulse_P$ では PMD が一番良い指標を示した。再現率の値が大きくなっているということは、Fault-prone モジュールの検出数が純粋に高くなっていることを示す。つまり、実際に存在する不具合を含むモジュールの検出数では CheckStyle が一番であるといえる。しかしながら、一番有効的な手法であったものは総合的な評価指標を示す $F1$ 値が最も高い CheckStyle であったことが言える。

3 手法の多数決を取った結果はそれぞれの評価指標で 3 つの手法の中間を示す位置となった。通常，1 つの手法における再現率と適合率はトレードオフの関係となり，片方が極端に高いという状態はあまり好ましくない．多数決により，そうした再現率と適合率のバランスを取ることが可能であることをこの結果は示している．

本実験では多数決による性能比較を行ったが、他の組み合わせ方によってはもっと良い結果が生み出せるかもしれない。また、それぞれの静的解析ツールを用いた予測手法自体の性能を高めることも必要であると言える。

いずれにしても、本実験の結果より、複数の静的解析ツールにおる多数決を利用した不具合判別は、それに用いる静的解析の性能に大きく左右されるものであり、特定のモジュールに対する性能が特化したものを組み合わせることで有効的に使えるのではないかと期待できる。

6. 結言

本研究では，複数の静的コード解析ツールから得られる警告メッセージを用いた fault-prone モジュール予測手法から得られた不具合予測結果を多数決によって選別する手法を提案した．適用実験の結果，多数決によって各手法が持つ特徴を平均した形での予測結果が得られることを確認した．

今回は単一のソフトウェアのみに対して実験を行ったが，より多くのプロジェクトに対して実験を行ったり，プロジェクト横断的な予測実験を行うことが考えられる．また，静的コード解析ツールとして，今回利用した PMD や CheckStyle だけではなく，そのほかのツールも利用することを検討すべきである．ただし，多くの静的コード解析ツールは Java のバイトコードを対象にしているため，ソースコードへの適用が可能なものに比べて多くの時間を要する．そうした課題の解決も行わねばならない．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学部門水野 修准教授に厚く御礼申し上げます．本報告書執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻平田幸直先輩をはじめとする，ソフトウェア工学研究室の皆さん，学生生活を通じて筆者の支えとなった家族や友人に深く感謝致します．

参考文献

[1] O. Mizuno and T. Kikuno, “Training on errors experiment to detect fault-prone software modules by spam filter,” Proc. of 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pp.405–414, 2007.

[2] M. Nakai and O. Mizuno, “Fault-prone module prediction by filtering warning messages of static code analyzer,” Proc. of the Joint Conference of the 21th International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement (IWSM/MENSURA2011), Fast abstracts, pp.18–21, Nov. 2011. Nara, Japan.

[3] T. Copeland, PMD Applied, Centennial Books, Arexandria, VA, 2005.

[4] CheckStyle 5.5, （オンライン）, 入手先 ⟨http://checkstyle.sourceforge.net/⟩（参照 Feb. 2012）.

[5] H. Hata, O. Mizuno, and T. Kikuno, “Comparative study of fault-proneness filtering with PMD,” Proc. of 19th International Symposium on Software Reliability Engineering (ISSRE2008), pp.317–318, Nov. 2008. Seattle/Redmond, WA, USA.

[6] Sun Microsystems, Inc. “Java code conventions,”, 1997.

[7] W.S. Yerazunis, CRM114 – the Controllable Regex Mutilator, （オンライン）, 入手先 ⟨http://crm114.sourceforge.net/⟩（参照 Feb. 2012）.

[8] P. Graham “Hackers and painters: Big ideas from the computer age,”, chapter 8, pp.121–129, O’Reilly Media, 2004.

[9] Y. Hirata and O. Mizuno, “Do comments explain codes adequately? – investigation by text filtering –,” Proc. of 8th Working Conference on Mining Software Repositories (MSR2011), pp.242–245, May 2011. Honolulu, HI, USA.

[10] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes? (on Fridays.),” Proc. of 2nd International workshop on Mining software repositories, pp.24–28, 2005.

[11] 川本公章，“複数の版管理システムを対象とした不具合混入モジュール特定アルゴリズムの実装,” 卒業研究報告, 京都工芸繊維大学，Feb. 2011.