卒 業 研 究 報 告 書

題　目　　ソースコード静的解析結果のテキスト分類による
不具合混入モジュールの予測手法

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　　07122032

氏　　名　　中井　道

平成23年2月15日提出

ソースコード静的解析結果のテキスト分類による
不具合混入モジュールの予測手法

平成 23 年 2 月 15 日　　07122032　中井　道

概　要

スパムフィルタリングの理論に基づいた Fault-prone モジュールの検出手法である Fault-prone フィルタリングでは、これまでソースコードモジュール以外のモジュールを学習・分類に用いてこなかった。だが、Fault-prone フィルタリングに与えるモジュールはテキスト情報であれば何でも良いため、本実験では静的コード解析ツール PMD の出力をモジュールとして与えて Fault-prone フィルタリングを行い、その影響を調査する。その結果、従来の手法に近い結果が出たが、PMD の出力を用いた提案手法の方が再現率に若干の向上が見られ、精度、適合率等に若干の低下が見られた。そのことから、PMD の出力は十分 Fault-prone フィルタリングに与えるモジュールとして利用可能であり、また、少しでも不具合の発見率を上げたいときは、従来手法よりも効果的であることが分かった。[1]

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。

そこで、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、Fault-prone フィルタリング [2] という手法が存在する。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もとのする。そして、スパムフィルタリングの理論に基づき、与えられたモジュールが Fault-prone モジュールであるのか、Fault-prone モジュールでないのかを判定するものである。

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。しかし、本来 Fault-prone フィルタリングで学習・分類に用いるモジュールは、テキスト情報であればどのようなものでも良かった。

また静的コード解析ツール 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ているツールであり，その出力は不具合を検出するための強力な特徴な特徴でになると予想される。

そこで本研究では、Fault-prone フィルタリングに与えるモジュールとして、ソースコードモジュールの代わりに PMD の出力をモジュールとして用いて予測実験を行う。

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

では、Fault-prone フィルタリングの 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

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，それによって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る．それにより，ソフトウェアの品質を向上させることができ，保守にかかるコストを削減することができる．

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．

2.1 PMD の概要

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．

これはオープンソースなソフトウェアで，Java を用いて作成されている．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，自分で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．

ちなみに，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．

2.2 PMD ルールセット

数多く用意されているルールセットの中で，以下の実験で使用する 10 個のルールセットについて概要を述べる．

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を検出，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．

3. Fault-proneフィルタリング

Fault-proneフィルタリングとはモジュールの不具合の検出にスパムフィルタリングの理論を導入したものである．

現在使用されている一般的なスパムフィルタは，その仕組みにベイズの定理を利用しているものが多い．これらのスパムフィルタでは受信したメールの特徴を学習し，新たにメールを受信した際にそれらの学習に従ってメールを本人に有用なメール（ham）と迷惑なメール（spam）に分類するといった仕組みを取っている．

基本的なスパムフィルタの動作は，メールの特徴を学習するステップと，学習した結果を用いてメールを分類するステップの2ステップで構成されている．

学習ステップ

1. 学習用にhamメールなのかspamメールなのかが分かっているメール群を用意する．
2. 各メールから単語（トークン）を抽出し，それらのトークンを登録した辞書を作成する．この際に，hamメールから抽出したトークンとspamメールから抽出したトークンは別々の辞書に登録する．したがって，hamメール用とspamメール用の2つの辞書が作成されることになる．

分類ステップ

3. 作成した2つの辞書を用いて，新しく受信したメールをhamメールであるか，spamメールであるかに分類する．

このようなスパムフィルタの動作は，hamメールとspamメールには文中に含まれる語や文の傾向といったものにそれぞれ異なる特徴見られるといった仮定に基づいている．

水野らは，この仮定が不具合を含むソースコードと含まないソースコードに対しても当てはまると考え，スパムフィルタをFault-proneモジュールの検出に導入したものを提案し，「Fault-proneフィルタリング法」と名付けた．Fault-proneフィルタリングでは，不具合を含まないモジュールをhamメール，不具合を含むモジュールをspamメールとそれぞれ対応させ，スパムフィルタの処理と同様に辞書を学習してい

く．そして，学習した辞書を用いることで，新しいモジュールが不具合ありモジュールである確率を計算し，Fault-prone モジュールであるかの判定を行う．

Fault-prone フィルタリングはいろいろな観点から研究が進められている [4][5][6][7].

3.1 PMD を利用した Fault-prone モジュールの検出

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 [5] 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 [6] ではモジュール中のコメントに該当する部分を削除して実験が行われた．そして，研究 [7] では研究 [6] とは逆に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．

このようにソースコードモジュールを学習・分類の対象とした研究は数多く行われているが，ソースコードモジュール以外を対象とした研究はまだ行われていない．Fault-prone フィルタリングへの入力はテキスト情報であれば何でも構わないため，ソースコードから得られる別の情報を学習・分類の対象とすることでも Fault-prone モジュールの予測が可能ではないかと考えた．

こうした予想の下に，第 2 章で述べた静的コード解析ツール PMD を Fault-prone フィルタリングの入力として与えることを考える．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，本研究では新たな観点として，第 2 章で述べた静的コード解析ツール PMD の出力をソースコードモジュールの代わりに入力として与えることで，Fault-prone フィルタリングの予測精度に与える影響を調査する実験を行う．

3.2 Fault-prone フィルタの実装

本実験では，文献 [7] で用いられた Fault-prone フィルタを使用する．この Fault-prone フィルタはベイズの定理を利用して作成された単純なものである．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．以下，PMD の出力のトークン分け及び，学習，分類の両ステップについて説明する．

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，本実験ではソースコード及び PMD の出力を用いるが，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) PMD 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 学習ステップ

このステップは後の分類ステップで必要となる辞書を作成するステップである．

ここでは不具合を含むモジュールと不具合を含まないモジュールからトークンを抽出し，それぞれを別々にトークンの出現数を数える．そして，それにより次に定義する 2 つのハッシュテーブルを得る．

- $faulty(t)$：全ての Fault-prone モジュールにおけるトークン t の出現数
- $nonfaulty(t)$：全ての Fault-prone でないモジュールにおけるトークン t の出現数

3.2.3 分類ステップ

このステップでは学習ステップで作成した辞書を用いて，与えられたモジュールが Fault-prone モジュールであるのか，そうでないのかの判定を行う．

このステップは次の 3 つの手順から構成されている．

1. まず，トークン t を含むモジュールが Fault-prone モジュールである確率を $P_{ft|t}$ と定義し，それを得るハッシュテーブルを作成する．不具合を含むモジュールの数を N_{ft}，不具合を含まないモジュールの数を N_{nf}，不具合を含むモジュール中にトークン t が出現する確率を r_{ft}，不具合を含まないモジュール中にトークン t が出現する確率を 2 倍したものを r_{nf} とそれぞれ定義することで，確率 $P_{ft|t}$ は式 (3.3) で表される．

$$r_{nf} = \min\left(1, \frac{2 \times nonfaulty(t)}{N_{nf}}\right) \tag{3.1}$$

$$r_{ft} = \min\left(1, \frac{faulty(t)}{N_{ft}}\right) \tag{3.2}$$

$$P_{ft|t} = \max\left(0.01, \min\left(0.99, \frac{r_{ft}}{r_{nf} + r_{ft}}\right)\right) \tag{3.3}$$

2. 次に，分類されるモジュールからトークンを抽出し，n 個の特徴的なトークン $t_1 \cdots t_n$ を決定する．ここで，これらのトークンについて，式 (3.4) の値が大きいほど特徴的であると定義する．また，今までに一度も出現したことのないトークンであった場合は $P_{ft|t}$ は 0.4 と定義する．

$$\mathrm{abs}\left(0.5 - P_{ft|t}\right) \tag{3.4}$$

しかし，本実験では n 個の特徴的なトークンを全てのトークンとしている．

3. そして，式 (3.5) を用いて，分類されるモジュールが Fault-prone モジュールである確率を計算する．

$$P_{ft} = \frac{\prod_{i=1}^{n} P_{ft|t_i}}{\prod_{i=1}^{n} P_{ft|t_i} + \prod_{i=1}^{n} \left(1 - P_{ft|t_i}\right)} \tag{3.5}$$

判定基準として，式 (3.5) で求めた P_{ft} を用いる．P_{ft} が 0.9 以上の値を取れば，分類されるモジュールは Fault-prone モジュールであると判定する．なお，ここで使用した 0.9 は変更可能な閾値である．

4. 適用実験

この実験では，PMD の出力をソースコードモジュールの代わりに入力として与えた際に，Fault-prone フィルタリングの予測精度に与える影響を調査することを目的とする．そのため，実験はソースコードモジュールを入力とした分類実験と PMD の出力を入力とした分類実験の 2 パターンを行う．

4.1 実験対象

本実験では実験対象として次の 2 つのオープンソースプロジェクトのソースコードモジュールを用いる．

- **Eclipse BIRT**
  Eclipse のビジネス文章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．
- **Eclipse**
  統合開発環境であり，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．

モジュールは学習用と評価用に分けるが評価用は最終更新時間の新しいものから順に全体の 10 % と定義する．各プロジェクトにおけるモジュールの詳細を表 4.1 に示す．

4.2 プロジェクトモジュールの取得

これらのプロジェクトモジュールの取得には，SZZ アルゴリズムに基づいて作成された 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，

表 4.1　実験で用いる各モジュールの詳細

	Eclipse BIRT		Eclipse	
	評価用	学習用	評価用	学習用
不具合を含まない	6268	36235	20665	128483
不具合を含む	746	26895	10231	149587
合計	7014	63130	30896	278070
	70144		308966	

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．

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，PMD の出力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．

1. まず，SQlite 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，モジュールは学習用と評価用に分け，且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. 次に PMD 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．その際に用いるルールセットは文献 [9] に従い，第 2.2 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．PMD 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，それらは分類に悪影響を与え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．

この実験手順を，Eclipse BIRT，Eclipse の各プロジェクトに対して実行する．

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（False_positive），フォールス・ネガティブ率（False_negative）の 6 つを用いる．

精度（Accuracy）

精度（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をどれだけ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_positive）

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．

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

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_negative）

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．

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

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 4.3，4.4 に各プロジェクトのモジュールの分類の予測結果を示す．また，表 4.5，4.6 に上で挙げた各評価指標の計算結果を示す．これらの表では表 4.2 で用いた N_1〜N_4 の表記を用い，フォールス・ポジティブ率，フォールス・ネガティブ率をそれぞれ $False\_P$，$False\_N$ と表記する．

表 4.3 から読み取れる特徴を示す．N_1 と N_3 に着目すると，ソースコードをモジュールとした方が大きな値を示していることが読み取れる．N_2 と N_4 に着目すると，PMD の出力をモジュールとした方が大きな値を示していることが読み取れる．

また，表 4.4 を見てみるが，上で示した表 4.3 の特徴と同様の傾向が出ていることが分かる．

次に，表 4.5 から読み取れる特徴を示す．この表を見ると，再現率と $False\_P$ に関しては PMD の出力をモジュールとした方が大きい値を示しているが，その他の精度，適合率，F_1 値，$False\_P$，$False\_N$ はソースコードをモジュールとした方が大きい値を示していることが読み取れる．

また，表 4.6 からも上で示した表 4.5 の特徴と同様の傾向が出ていることが分かる．

表 4.2　実験結果の凡例

		予測	
		Not Fault-prone	Fault-prone
実測	不具合を含まない	N_1	N_2
	不具合を含む	N_3	N_4

表 4.3　分類されたモジュール数の詳細（Eclipse BIRT）

	N_1	N_2	N_3	N_4
ソースコード	3719	2549	77	669
PMD 出力	3118	3150	49	697

表 4.4　分類されたモジュール数の詳細（Eclipse）

	N_1	N_2	N_3	N_4
ソースコード	8916	11749	811	9420
PMD 出力	8131	12534	743	9488

表 4.5　モジュールに対する各評価指標（Eclipse BIRT）

	精度	再現率	適合率	F_1 値	$False\_P$	$False\_N$
ソースコード	0.626	0.897	0.209	0.338	0.363	0.011
PMD 出力	0.544	0.934	0.181	0.304	0.449	0.007

表 4.6　モジュールに対する各評価指標（Eclipse）

	精度	再現率	適合率	F_1 値	$False\_P$	$False\_N$
ソースコード	0.593	0.921	0.445	0.600	0.380	0.026
PMD 出力	0.570	0.927	0.431	0.588	0.406	0.024

5. 考察

表 4.3，4.4 の結果から読み取れた特徴について考える．これによると，PMD の出力を用いた方はソースコードモジュールを用いた方と比べ，与えられたモジュールを Fault-prone モジュールと判断する確率が大きいことが分かる．

次に，表 4.5，4.6 から読み取れた特徴について考える．PMD の出力を用いたことで起きた変化に焦点を当てると，精度，適合率が小さく，$False\_P$ が大きくなっていることから，多くの評価指標が悪化を示していることが分かる．

精度が下がるということは，Fault-prone フィルタリングが間違った判断をする可能性が上がることを意味する．そして，適合率が下がるということは，Fault-prone モジュールを誤検出してしまう可能性が上がり，不具合を含まないモジュールを調べることに無駄な工数を取られる．ここで $False\_P$ つまりフォールス・ポジティブ率が誤検出した割合を示しているのでそれによると，半数近くが誤検出であることが分かり，かなり多くの無駄な工数がかかることが予想される．

しかし，すべてが悪化したわけではない．再現率と $False\_N$ に注目すべきである．再現率の値が大きくなっていることから，Fault-prone モジュールの検出数が純粋に多くなっていることが分かる．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が増加したことを示している．$False\_N$ つまりフォールス・ネガティブ率が減少したことにより，不具合を含むモジュールの検出漏れが減少し，よりくまなく不具合を検出できることが予想される．

そして，F_1 値に注目する．元々再現率と適合率はトレードオフの関係であったため，再現率が向上した代わりに適合率が悪化したのは仕方のないことである．だが，総合的な評価指標である F_1 値が減少したため，総合的に見ると PMD の出力をモジュールとした場合，Fault-prone フィルタリングの性能が低下すると考えられる．

また，Eclipse BIRT に比べて Eclipse での実験の方が各評価指標の低下量が少ないのは，元々Eclipse のプロジェクトに含まれる不具合を含むモジュールの数が多かったため，全体として Fault-prone モジュールであると予測した場合に予測が外れる可能性が低かったためと考えられる．

よって，これらの情報をまとめると，PMD の出力を入力モジュールとして Fault-prone フィルタリングを使用した提案手法では，ソースコードモジュールを入力モ

ジュールとした従来手法と比べ，不具合の誤検出は増加するが，より多くの不具合を検出することが可能となる．したがって，工数がかかってでもできるだけ不具合を取り除きたいようなプロジェクトでは提案手法を用いることでより高い効果が期待できる．

ここで，考察しなければならないことがある．たしかにPMDの出力を用いた場合は総合的にはFault-proneフィルタリングの性能が低下した．しかし，表4.5，4.6から読み取れるように，数ある評価指標の悪化や向上は微々たるものということである．これは言い換えると、提案手法の結果は従来手法にきわめて近い値が出ていると言え、PMDの出力はソースコードと同等の判断材料となり得ることが分かる。

また、PMDの出力は十分Fault-proneフィルタリングで分類可能であり，冗長な出力の中から本当に重要な単語を抽出することができるとも言える。

ところで，提案手法の若干ではあっても多くの評価指標が悪化してしまったことには原因があると考える．考えられる原因を以下に挙げる．

- PMDのルールセットの選択
- 学習用及び評価用モジュールの定義
- トークン分割規則
- プロジェクトの選択
- 提案手法の限界

まず，本実験ではPMDのルールセットとして第2.2節で示した10個を用いたが，その選択を変更すれば良い結果が得られた可能性も考えられる．学習用及び評価用のモジュールの定義も実験に大きく影響を与える．トークンの分割に関してもPMDの出力のトークン分けを単語単位ではなくもう少し意味のある単位で分割すれば良い結果が得られたかもしれない．また，今回用いたEclipse BIRT及びEclipseのプロジェクトが例外で，他のプロジェクトで実験した場合に良い結果が得られる可能性は，低いとは考えるが，ないと断言できるものではない．そして，最も望まない原因としてこの結果が提案手法の限界であるという可能性である．

本実験は，Eclipse BIRTとEclipseといった2つのプロジェクトについて実験を行い，それぞれの結果に同じ傾向が見られたことにより実験結果の信頼性が向上した．しかし，より多くのプロジェクトについて実験を行う方がより信頼性を得ることが

できると考える．

5.1 妥当性の検証

本実験の妥当性について検証を行う．

- **プログラムの不具合**

  実験で使用した各種プログラムに不具合が含まれている可能性がある．仮に学習や分類のプログラムに不具合が含まれている場合，モジュールの学習や分類が正しく行われていない可能性が生じる．また，それ以前にデータベースからソースコードモジュールを取得するプログラムに不具合が含まれている場合，正しくソースコードモジュールを取得できていなかったり，不具合を有無の情報が間違っている可能性が生じる．

- **データベースの不備**

  本実験に用いたモジュールはSZZアルゴリズムに基づいて作成されたSQLite3のデータベースから取得したものであるので，データベースの作成に不備がある場合，正しいモジュールが取得できない可能性が考えられる．また，プロジェクトの開発者は人間であるので，データベースが作成される以前に，バージョン管理システム及びバグ管理システムへの入力を誤っている可能性がある．

- **トークン分割規則の不備**

  トークンを分割する規則に不備がある可能性がある．その場合，正しいトークンが抽出されなくなり，実験結果に悪影響を与える．

6. 結言

本研究は、Fault-prone フィルタリングの入力モジュールとして静的コード解析ツール PMD を用いる手法を提案した。それにより従来手法に近い結果が出るも、従来手法と比べ、精度、適合率、F_1 値などに若干の低下が見られ、再現率に若干の向上が見られるといった結果が得られた。今後の課題しては，多くの評価指標が低下した原因を検証と，より多くのプロジェクトに対しての実験の実行が挙げられる．また，PMD 以外の静的コード解析ツールについて実験を行うことも望ましい．

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，研究生 梁軍偉先輩，情報工学課程 出原真人君，川本公章君をはじめとする，ソフトウェア工学研究室の皆さん，学生生活を通じて著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] 川本公章，“ほげほげ,” 卒業研究報告, 京都工芸繊維大学，2011.

[2] 水野　修，菊野　亨，“Fault-prone フィルタリング：不具合を含むモジュールのスパムフィルタを利用した予測手法,” SEC journal，vol.4，no.1，pp.6–15，Feb. 2008.

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

[4] O. Mizuno and T. Kikuno, “Training on errors experiment to detect fault-prone software modules by spam filter,” The 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE2007), pp.405–414, Sept. 2007. Dubrovnik, Croatia.

[5] O. Mizuno and T. Kikuno, “Prediction of fault-prone software modules using a generic text discriminator,” IEICE Trans. on Information and Systems, vol.E91-D, no.4, pp.888–896, April 2008.

[6] H. Hata, O. Mizuno, and T. Kikuno, “Fault-prone module detection using large-scale text features based on spam filtering,” Empirical Software Engineering, vol.15, no.2, pp.147–165, April 2010.

[7] 平田幸直，水野　修，“テキスト分類に基づく fault-prone モジュール検出法におけるコメント行の影響の分析,” 情報処理学会研究報告 ソフトウェア工学 (SE)，第 2010-SE-170 巻，no.10，pp.1–8，Nov. 2010．大阪大学．

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

[9] O. Mizuno and H. Hata, “A hybrid fault-proneness detection approach using text filtering and static code analysis,” International Journal of Advancements in Computing Technology, vol.2, no.5, pp.1–12, Dec. 2010.