卒 業 研 究 報 告 書

題　目　CodeLamp:

フォールトプローンフィルタリングの

Eclipse Plugin開発とその適用

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　12122014

氏　名　黒田　翔太

平成28年2月15日提出

CodeLamp:フォールトプローンフィルタリングの Eclipse Plugin 開発とその適用

平成 28 年 2 月 15 日　　　　12122014　黒田　翔太

概　要

ソフトウェア開発過程において、バグは発生し、開発者はその修正を要求されている。このバグは気付かれず放置されてしまうと、後々に大きな修正が必要となったり、開発に悪影響を及ぼすことになる。

これらのバグを早期に発見できれば、修正の労力は少なく済むので、バグの発生を「予測」することができれば、ソフトウェア開発に貢献できる。

これらバグ予測についての研究は数多く、その研究の一つとして研究されている「Fault-prone フィルタリング」という、スパムメールの判別に用いられるベイズ識別を用いた仕組みのシステムを用いて、実際の開発に利用できるようなツールに作成することを目的とした。

この研究において開発したツール、「CodeLamp」は Eclipse プラグインとして利用することができるもので、ソースファイル、及び Git コミット差分について、バグ予測を行うことができる。また、ソースファイル・Git コミット差分を学習することもでき、ツールの使用者のバグコーディングの特徴を判別するために最適化したデータベースの作成につながる。

しかしながら、「Fault-prone フィルタリング」は学習量が少ない状態では予測精度は望めないため、利用者が自分のコーディングに合ったデータベースを 1 から作成することは難しい。そのため、利用者が学習する場合にはあらかじめ大量の学習済みデータベースを用意しておくなどの、用意が必須であることが分かった。また、学習時に「ノイズ」となってしまう情報も多く含まれてしまうので、学習時の仕組みにも修正が必要である。これらの対応を行うことで、予測精度の向上が見込めれば、実際にバグ予測ツールとして利用できるようになる。一方、ツールとしては、マルチプラットフォームで利用するためには利用者にも準備が必要であったり、機能・インターフェースともに改良の必要が大きい。

後日、共同研究先で試用評価を予定しており、その結果も踏まえて、修正を行う予定となっている。

目　次

1. 緒言 **1**

2. 研究背景 **3**

2.1 ツールとしての要件 3

3. 実装の準備 **4**

3.1 Eclipse [1] 4

3.2 Eclipse プラグイン 4

3.3 JGit [2] 4

3.4 Git 5

3.5 Fault-prone フィルタリング法 5

3.6 Orthogonal Sparse Bigram (OSB) 6

3.7 ベイジアンフィルタ (ベイズ推定) [3] 7

3.8 CRM114 Descriminator [4] 7

4. CodeLamp **9**

4.1 CodeLamp 9

4.2 ツールの構成 9

4.3 動作環境 9

4.4 使い方について 10

4.4.1 CodeLamp-Creator 10

4.4.2 CodeLamp-E 12

4.4.3 CodeLamp-J 15

4.5 CodeLamp で使用している外部モジュール 15

4.6 CodeLamp 中の仕組み 16

4.6.1 CodeLamp-Creator における仕組み 16

4.6.2 CodeLamp-E 及び CodeLamp-J における仕組み 17

5. 動作実験 **18**

5.1 実験目的 18

5.2 実験準備 18
5.3 実験対象 18
5.4 確認する要件 19
5.5 実験の結果 19
5.6 考察 20
5.7 要件に対する結果の達成具合 22
5.7.1 要件の達成評価と課題 22

6. 試験利用による評価の予定 24

6.1 製作途中にいただいた意見とその反映 24
6.2 評価項目 24
6.3 予想される評価と対応の予定 25

7. 結言 27

謝辞 27

参考文献 28

1. 緒言

ソフトウェア開発において、「バグ・不具合」というものはどうしても発生してしまう。それら「バグ」の発生は軽度のものから重度のものまで様々であり、それらを発見する機会も様々である。

コーディング中にバグが発見されることなく開発が進行し、ソフトウェアテストの際にに入り組んだモジュールからバグがようやく発見されて、そのバグを修正する。そしてまたソフトウェアテストを行ったときに、バグが発見され、修正が必要になるかもしれない。それでもなお、使用時まで見つからなかったバグもあるかもしれない。このように、バグの混入は開発工程に悪影響を及ぼし、バグの未発見はソフトウェアの質の低下を招く。しかし、開発中の全リソースを人力で精査することは極めて大きな負担となる。

このような状況への対処として、開発過程におけるバグの絞り込みと早期発見が必要である。バグがありそうな箇所が絞り込めれば、個別に精査・修正するにしても負担は減る。バグが開発途中に修正できるのであれば、テスト時に発見される不具合も少なくなる。バグがソフトウェアテストの時点で減っているのであれば、修正と再テストに要する回数も少なくなると期待されている。

一般に、これらソフトウェアへのバグの混入を知ることは非常に難しく、出現して初めて認知されることになる。そこで、事前の情報からバグの混入を予測する研究が行われている。そのための手段の一つとしてで研究されているのが「バグ予測」である。これまでに蓄積されてきた開発工程の記録をもとに、バグが発生していそうなモジュールを予測する。これを「Fault-prone モジュール予測」と呼ぶ。

これまでに「Fault-prone モジュール予測」に関する多くの研究がなされてきた。[5][6][7] 例えば、2011 年末には、Google がバグ予測アルゴリズム [8] を公開し、それを実装した「bugspots」[9] というツールが登場するなど、様々な研究が行われている。

本研究では、それら「Fault-prone モジュール予測 [10]」の一つ、スパムメールフィルタリングの仕組みで実現する「Fault-prone フィルタリング」という手法を用いる。Fault-prone フィルタリングの研究は数多くなされ [10][11][12][13][14]、多くの検証ツール・スクリプトも作られたが、それらを利用者を選ばずに実用化することは難しかった。本研究はこの「Fault-prone フィルタリング」を「開発現場」で試用できるツー

ルとして実現することを目標としたものである。

本研究はバグ予測に関するものであり、説明や操作において「バグ」の対義語が必要となる。そこで本研究においては、「Bug、バグ、不具合」を「Faulty」なものとまとめ、その対義語「NonBug、バグでない、不具合のない」を「Non-Faulty」なものとして総称する。

本研究では「ツール」の実装が目的であるので、それに伴い、ツール化するにあたっての要件を 5.4 に、4.4 で作成したツールの利用方法を、4.6 で、ツールの詳細をまとめた。当ツールの実働実験については 5 章にまとめている。

当初の予定では、共同研究先にて試用評価を行う予定であったが、開発と先方の都合で評価することができず、本報告にまとめることはできなかったが、評価予定の項目、及び予想される評価に対する改良案などを 6 章にまとめた。

2. 研究背景

過去の「Fault-prone フィルタリング」に関する研究では、様々な入力対象に対して、Fault-prone フィルタリングを検証してきた。それらの研究ごとに測定ツール・スクリプトが利用されてきたが、それらは検証の意味合いが強く、それらを直接、開発現場に導入することは難しかった。

そこで、本研究は共同研究の一環として、Fault-prone フィルタリングを実装した、「現実的な」開発ツールの補助アプリケーションを作成する。

本研究は、補助アプリケーションの実装環境案として、フトウェア開発で広く使われている統合開発環境「Eclipse」[1] において利用する Eclipse プラグインとして Fault-prone フィルタリングを実装を目指したものである。

2.1 ツールとしての要件

現実的な開発ツールの補助アプリケーションとして実装するために、不特定多数の開発者の利用を想定し次の要件を挙げる。

- Eclipse 本体をよび Java 実行環境を除く外部ツールの利用、特にインストールの必要な外部ツールの利用を避ける。
- Eclipse はマルチプラットフォームアプリケーションであるので、同様に、なるべくマルチプラットフォームで利用できるツールとする。

また、Eclipse プラグインとして作成するにあたり、次の点を考慮する。

- シンプルである。つまり、使い方が過剰に複雑化しない。
- 使用者によってツールの利用方法は一定ではないことに配慮したインターフェースを作る。

一方で、Fault-prone フィルタリングに用いるデータベースを、初めて作成するときには、膨大なソースコードから学習すると想定されるために、非常に時間がかかることが想定されるため、Eclipse プラグインとは独立したツールとして用意する。

3. 実装の準備

3.1 Eclipse [1]

Java で実装された統合開発環境。オープンソースソフトウェア (OSS)。数ある統合開発環境の中でも特にユーザ数が多い。

3.2 Eclipse プラグイン

「Eclipse」において、「Eclipse プラグイン」が機能拡張を果たす。基本となる Eclipse Platform 以外の機能、例えば、Java も含め各開発言語への対応も Eclipse プラグインによるものである。つまり、プラグインを実装することで、Eclipse は様々な機能を追加できる大きな汎用性を持っている。

Eclipse プラグインの開発もまた、Eclipse プラグインである PluginDevelopment Environment(PDE) という専用のツールを使用する。Eclipse プラグインで UI を作成するためには、Eclipse の GUI 作成に使われている SWT や JFace を利用することができる。SWT も JFace も、Swing・AWT や JavaFX といった GUI ライブラリと同様に、GUI を持つ Java アプリケーションの作成に利用でき、Eclipse プラグイン専用のライブラリというわけではない。PDE を利用して、Eclipse の UI ライブラリ (org.eclipse.ui.*) 上に SWT・JFace を用いて GUI を構成する。

本研究では SWT で UI を構成した Java アプリケーションを作成し、その上でプラグイン化する。

3.3 JGit [2]

Eclipse では、Git バージョン管理として EGit による UI が提供されているが、その EGit が Git 操作に使用している API が JGit である。

JGit とは、Git リポジトリに対して Git バージョン管理を Java から行える機能を実現する JavaAPI である。Java プログラマが Git 操作を行う機能を実装するためには、ProcessBuilder クラスを用いてコマンドプロンプトやシェルなどの外部プロセス経由で Git コマンドを使用することもできるが、そのためは実行環境に Git がインス

トールされている必要があり、万全ではない。一方で、JGit を使えば、Git がインストールされていない実行環境においても Java コードによる Git 操作が可能となる。

3.4 Git

開発などで広く使われる「分散型バージョン管理システム」なるもので、ファイルの変更履歴を保存・追跡するための仕組み。Git を適用した Git リポジトリと呼ばれるディレクトリは「.git」フォルダを持ち、その下にファイル変更時に関する記録 (変更者、時間、変更理由) や、ハッシュで分類されて変更差分などの様々な記録が保存されている。

変更を記録することは「コミット」と呼ばれ、このコミット単位ごとに履歴が管理・追跡される。

Git は一本道に変更履歴を保存・追跡するだけではなく、「ブランチ」として履歴を分岐して記録することもできる。「ブランチ」を分ける目的は、他のブランチの影響を受けない環境で、開発・修正して他のブランチに合流したり、別の案・プロジェクトとして独立するために利用される。

また、作業中の最新のコミットは「HEAD」として保存されており、ここから作業中のブランチも判別する。

特に、本研究で利用する相当する Git コマンドは、

- git log
  - Git の更新情報を確認する。
- git diff /a /b
  - Git の 2 コミット間の差分を出力する。本来、Git は log のオプションで対象コミットを指定することで差分の出力もできるのだが、JGit では diff を用いて差分を出力する。

3.5 Fault-prone フィルタリング法

モジュール (例えばソースコード) 中にバグが含まれていそうなことを予測することを「Fault-prone モジュール予測」といい、Fault-prone フィルタリング法とは、

Fault-prone モジュール予測の手法の一つ。Fault-prone モジュール予測の例では、「bugspots」というツールが利用している Google のバグ予測アルゴリズムなども知られている。

Fault-prone フィルタリングとは、Orthogonal Sparse Bigram (OSB) を用いてベイズ推定を行うという、スパムメールフィルタリングに用いられる手法をソースコードに対して応用するという、水野らの論文 [10] に基づく手法である。本研究では、この Fault-prone フィルタリングをバグ予測に利用する。

3.6 Orthogonal Sparse Bigram (OSB)

OSB とは、ある単語に対して 5 文字目までの各単語との組み合わせで分割した単位をもとにする方法である。これを用いたベイジアンフィルタ、OSBF[15] を本研究で使用する。

例えば、”WORA is a slogan of the Java language . . . .” という文章を OSB で解釈する分類すると、先頭の「WORA」に対して次のような組み合わせが成り立つ。

表 3.1　単語対生成例

	WORA	is	a	slogan	of	the	Java	language
単語対 1.1	WORA	is						
単語対 1.2	WORA		a					
単語対 1.3	WORA			slogan				
単語対 1.4	WORA				of			
単語対 1.5	WORA					the		
単語対 2.1		is	a					
単語対 2.2		is		slogan				
⋮								

3.7 ベイジアンフィルタ (ベイズ推定) [3]

ベイジアンフィルタは、対象に対しベイズの定理 (Bayes' theorem) に基づく条件付き確率の推定を行うもので、スパムメールフィルタなどで用いられている。

スパムメールフィルタを参考に、条件付き確率の推定のための計算を整理すると、カテゴリ (Cat) をスパム (spam)、スパムでない (ham) とし、対象となる入力 (Doc) は単語 (word) で形成されている ($Doc = \Sigma_i word_i$) とすると、

$$P(spam|Doc) = P(Doc|spam) \times \frac{P(spam)}{P(doc)} = \Pi_i P(word_i|spam) \times \frac{P(spam)}{P(doc)} \tag{3.1}$$

のようにして、各単語がスパムカテゴリに含まれているの積算に対して、係数 $\frac{1}{P(Doc)}$ とカテゴリの事前確率 $P(spam)$ が乗算されて、条件付き確率が推定される。

CRM114 が利用する OSBF というフィルタ方式では、独自の係数を条件付き確率に乗じて予測値として算出している。[15]

3.8 CRM114 Descriminator [4]

OSB を実現するツールとして CRM114 がある。本研究で使用しているのは次のパッケージである。

- CRM114crm114-20081111-BlameBarack-Ger-4560 [16]

OSB 自体は SQL のようなデータベースでも実装できるが、仮に SQL データベースで実装した場合、単純に重複なしと考えた場合、OSB の仕組み上、約 5 倍のテキスト情報が必要となってリソースを要し、また、処理時間も大きくなる。

一方、CRM を用いれば、データベースのリソースは抑制できる。CRM ではバケットとみなしたバイナリをデータベースとして利用するので、データベース作成時のリソースを上回ることがない。また、C 言語ベースで実装されており動作も軽快である。

しかし、もともとスパムフィルタとして設計されているように、特に Windows で様々に利用することを想定されていないのか、(非 Cygwin の)Windows 向けのバイナリ生成が極めて困難のため、本研究では、hebbut.net にてビルドされたバイナリを

使用するが、本家の最新のものではない。同時に、マルチプラットフォームでの動作の統一を確保するために、他の環境でも上記と同じバージョンの CRM114 を用いる必要があるのだが、Windows 以外の環境ではビルド時に要求される tre(laurikari.net) のライブラリが必須となるため、実行バイナリを同梱するだけでは動作しないので、crm114 自体をインストールする必要がある。

4. CodeLamp

4.1 CodeLamp

本研究にあたり作成した「Fault-prone フィルタリング」ツールを「CodeLamp」と呼ぶ。

本ツールでは、ソースコードに対する「Fault-prone フィルタリング」を用いた判別・学習を行う。

開発当初は Java のみで Git コミット差分を取得することが困難であると見込まれたため、ソースファイルに対する自動判別機能の実装にとどまり、「ぼやっとした(ソースファイルという広い) 範囲で」バグの可能性を判断できる程度の予定であったためこのような名称となった。

4.2 ツールの構成

CodeLamp は次の 3 つのツールから構成されている。

- CodeLamp-Creator
  CodeLamp を利用するための準備を行う。
- CodeLamp-E
  ソースファイル、もしくは Git コミットを対象として、バグ予測を行う本体の Eclipse プラグイン版。
- CodeLamp-J
  バグ予測を行う本体の Java アプリケーション版。

上記の各ツールについて、利用方法を 4.4、機能詳細を 4.6 にてまとめる。

4.3 動作環境

動作条件として、少なくとも、CodeLamp-E は Eclipse、CodeLamp-Creator・CodeLamp-J については Java RuntimeEnvironment[17] の導入が必要となる。

Windows 環境下での動作を想定しているが、UNIX(OSX 及び Linux) で動作させるためには、別途ツールをインストールする必要がある。

4.4 使い方について

デフォルトでデータベースも用意しているが、OSBF の特性上、同じ作成者やプロジェクトの方が、コーディングの際に同じ単語の組み合わせで記述する傾向があると考えられ、バグ予測の精度を見込めるため、第一に、CodeLamp-Creator を用いてデータベースの作成を行い、そのデータベースを使用して CodeLamp 本体を利用する。

4.4.1 CodeLamp-Creator

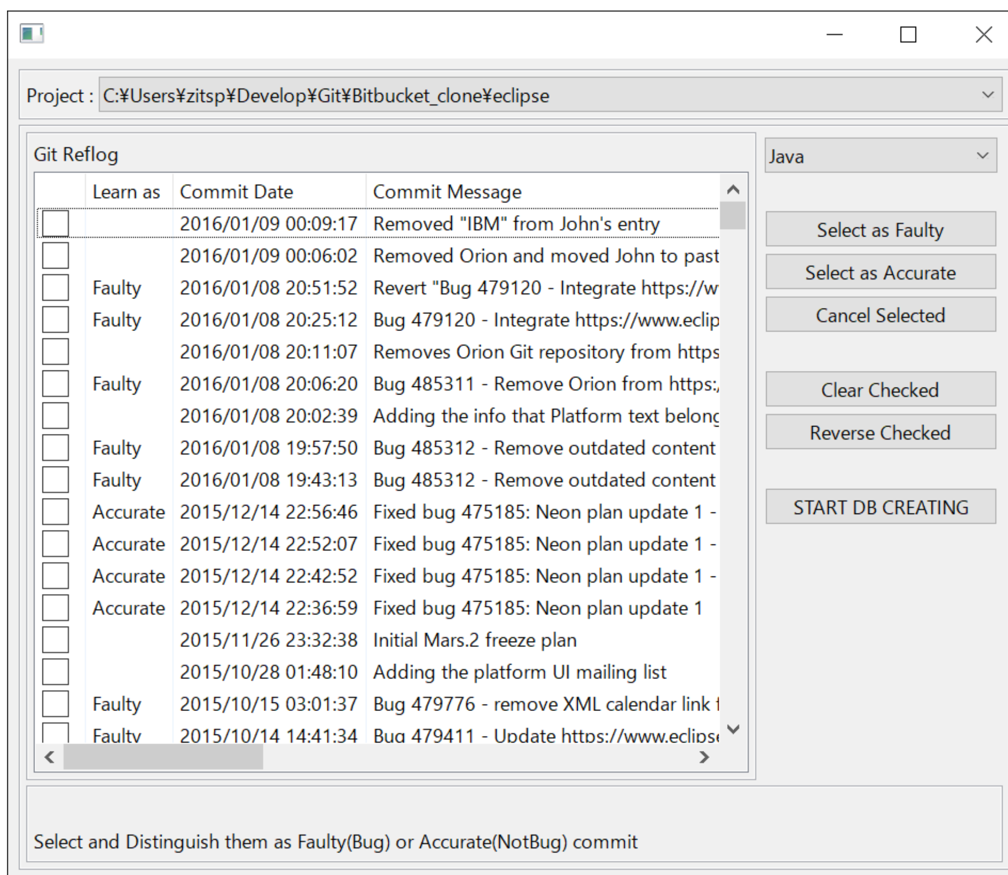

図 4.1 CodeLamp-Creator に Git リポジトリを読み込ませた様子

CodeLamp で使用するフィルタ用データベースを作成する。データベースの作成のために、Git リポジトリのコミット情報を取得し、それを利用者がコミットメッセージをもとに分類したものを学習に用いる。

1. (ユーザ) JRE を利用して、CodeLamp-Creator を起動する。

2. (ユーザ) プロジェクトフォルダ (Git リポジトリ)、もしくは.git フォルダを選択する。

   - (ユーザ) 起動 JAR に対してフォルダをドラッグアンドドロップする。
   - (ユーザ) 起動時にコマンドライン引数を利用して指定する。
   - (ユーザ) コンボウィジェットから「Add project」を選択し、ダイアログウィンドウからフォルダを指定する。
   - (ユーザ) ウィンドウに対象フォルダをドラッグアンドドロップする。

3. (ツール) Git リポジトリであれば、表示テーブルに対して作業中 (HEAD) のブランチのコミット一覧を表示する。
4. (ツール) コミットメッセージを参考にして、「Faulty(バグ)」か「Non-Faulty(正常)」か自動的にラベリングしている。
5. (ユーザ) コミット一覧に表示されたコミットメッセージを参考にして、「Faulty(バグ)」か「Non-Faulty(正常)」かを登録する。
6. (ユーザ)「Start」を選択する。

   (a) (ユーザ) ディレクトリ選択画面に従い、データベースの作成先を選択する。
   (b) (ツール) そのディレクトリに対して、「BUG.fpdb」、「NotBUG.apdb」の各データベースを生成する。
   (c) (ツール) 「Faulty」か「Non-Faulty」かの登録をもとに、学習を行う。
   (d) (ツール) 動作が終了すればボタンが解放される。

7. (ユーザ) 生成したデータベースはリネーム・移動に制限はないので、学習させたプログラミング言語がわかるように編集することが推奨される。
8. (ユーザ) 続けて学習したいリポジトリがあれば、項目 2 と同じく次のいずれかの操作でフォルダを選択し、同様に手順を進める。

   - (ユーザ) コンボウィジェットから「Add project」を選択し、ダイアログウィンドウからフォルダを指定する。
   - (ユーザ) ウィンドウに対象フォルダをドラッグアンドドロップする。

9. (ツール) 項目 5.2 のとき、すでに「BUG.fpdb」、「NotBUG.apdb」の存在するフォルダを選択すれば、そのデータベースに対して続けて学習する。

4.4.2 CodeLamp-E

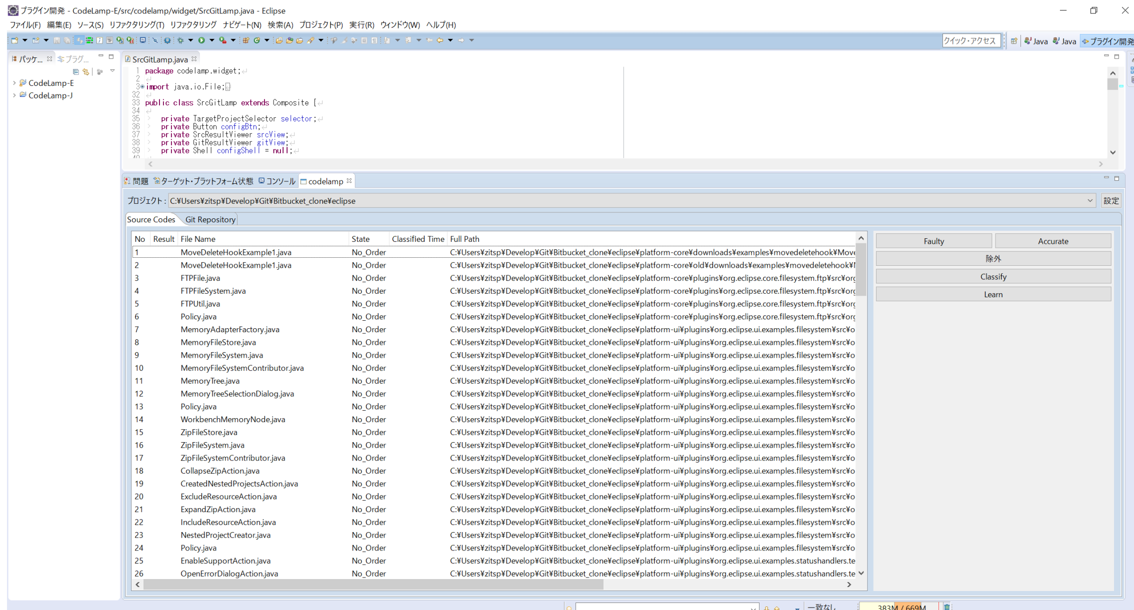

図 4.2 CodeLamp-E を Eclipse 上で起動した様子 (ソースファイルビュー)

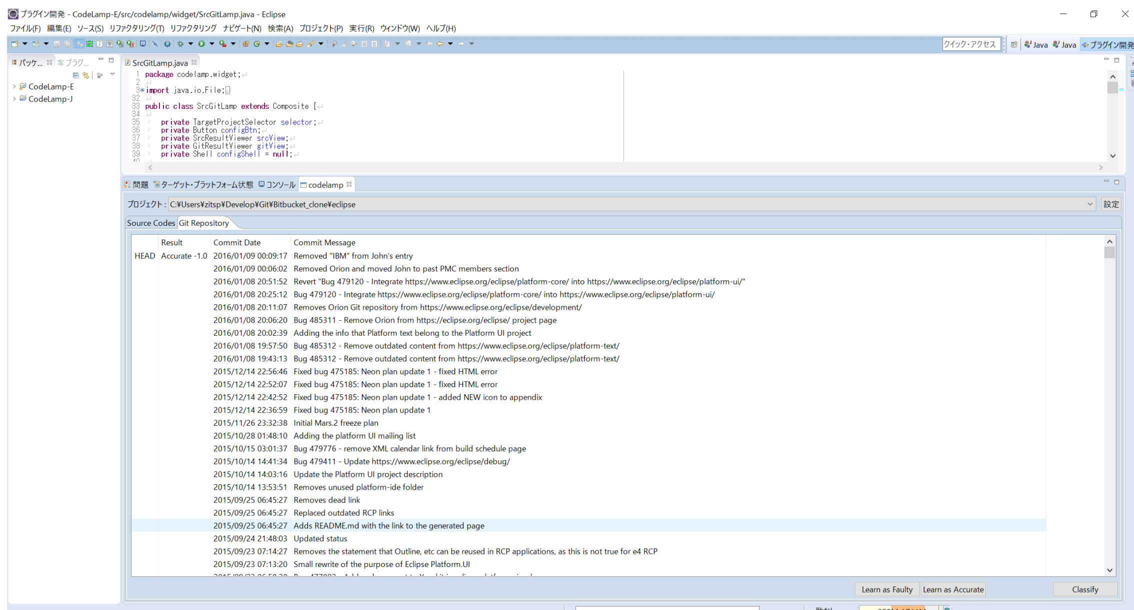

図 4.3 CodeLamp-E を Eclipse 上で起動した様子 (Git コミットビュー)

Eclipse プラグインとして、開発補助ツールとしてバグ予測を行う。フィルタ用デー

タベースをもとに、ソースファイル・Git コミット差分について学習・判別を行う Fault-prone フィルタリングツールである。

1. (ユーザ) Eclipse をインストールしたディレクトリのプラグインディレクトリにこのフォルダをコピーする。
2. (ユーザ) Eclipse を新たに起動するとメニューバーから「Window - ShowWindow - other」とたどり、「Other」の分類に「CodeLamp」というものがあるのでそれを選択すると、ビュースペースに CodeLamp が表示される。
   - (ユーザ) この段階でデータベースを登録・切り替えすることはできない。
3. (ユーザ) プロジェクトを選択する。
   - (ユーザ) コンボウィジェットから「Add project」を選択し、ダイアログウィンドウからフォルダを指定する。
   - (ユーザ) ウィンドウに対象フォルダをドラッグアンドドロップする。
4. (ツール) プロジェクトのトップディレクトリの下に「.codelamp」というフォルダを生成する。
   - (ツール) 「.codelamp」内に xml 形式でファイルの学習情報・データベースの登録情報を保存する。
5. (ツール) データベースの登録情報がなければ、「.codelamp/db/」内に、デフォルトのデータベースをコピーする。
6. (ツール) CodeLamp には「SourceCode」ビューと「Git Repository」ビューの切り替えタブがあり、それぞれ選択されたプロジェクトのディレクトリ構造から情報を出力している。
7. (ツール) フィルタに利用するデータベースについては、「Config」ボタンを押すと表示されるコンフィグからの変更を受け付ける。

CodeLamp の「Source Code」ビューに関して

ソースファイルに対して Fault-prone フィルタリングを行うインターフェースに相当する。

当インターフェースは次の機能を備える。

- ソースファイル一覧の表示
- 動的にソースファイルに対する判別・学習を実施
- ソースファイルの変更監視に基づく自動的な判別の実施
- 自動的なソースファイルの判別実施の拒否
- 判別が正しくない場合の動的な分類

次のように利用する。

1. (ツール) プロジェクトが選択された時、フィルタに対応する言語の拡張子を持つフォルダを当テーブルに一覧表示する。
2. (ツール) コンフィグ情報があればそれを反映する。
3. (ツール) 初期状態では各ソースファイルは自動監視されない。
4. (ツール) 一度、動的にソースファイルの判別を行うと、それ以降は変更監視の対象になる。
   - (ツール) 監視対象になった場合、ソースファイルが変更されると判別を行うようになる。
   - (ユーザ) 監視対象から外すためには、ファイルを選択し、「Exclude」ボタンを選択する。
5. (ツール) 判別の新しい順にソート・表示する
   - 判別結果に誤りがあると分かった場合、「Faulty」、「Non-Faulty」を選択する
     - (ツール) 結果表示に「*」が付き、誤りのあったファイルとして認識・学習する。
   - (ツール) 誤りのあったファイルとして認識しても、以降で判別が行われた場合には、認識を解除する。

CodeLamp の「Git リポジトリ」ビューに関して

Git コミット差分に対して Fault-prone フィルタリングを行うインターフェースに相当する。

当インターフェースは次の機能を備える。

- HEAD 情報をもとに、作業中のブランチのコミット一覧を表示
- HEAD 情報の更新に伴う自動的な判別を実施
- HEAD 以外のコミットに対しては、動的にコミット差分に対する判別・学習を実施

次のように利用する。

1. (ツール) プロジェクトが選択された時、作業中 (HEAD) のブランチのコミット一覧を当テーブルに一覧表示する。
2. (ツール) 初期状態では直前のコミット (HEAD のコミット) に対して判別を行う。
3. (ツール) 新たにコミットされると判別を行う。
4. (ツール) 以前のコミットに対する判別結果は当作業中は継続して保持する。
5. (ユーザ) 以前のコミットに対する判別・学習はコミットを選択し、ボタンを押すことで行う。

CodeLamp の「Config」ウィンドウに関して

フィルタ用データベースと対応言語に関して登録・変更する。

使用するフィルタには「*」が付いており、変更する場合には、対象をダブルクリックする。

4.4.3 CodeLamp-J

Eclipse プラグイン化する前の状態で、操作・機能は CodeLamp-E と同じ。

4.5 CodeLamp で使用している外部モジュール

- JavaAPI
  - SWT(ecllipse.org)
    - GUI 作成のため
  - JGit(eclipse.org)
    - Java で Git 操作を行うため

  - SLF4J(slf4j.org)
    * JGit を利用するために必須
    * ロギング JavaAPI である log4j を含む
- その他システム
  - CRM114 (crm114.sourceforge.net/hebbut.net)
    * Fault-prone フィルタリングを行う外部システム
  - tre/libtre (laurikari.net)
    * crm114 をコンパイルするため
  - flex
    * Java 用 tokenizer を作成 (京都工芸繊維大学ソフトウェアテスト工学研究室)

4.6 CodeLamp 中の仕組み

4.6.1 CodeLamp-Creator における仕組み

Git コミットごとの学習について

「Faulty」「Non-Faulty」に区分されたコミットはリストとして保持され、GUI とは別のスレッドで随時コミット差分を出力・学習する。別のスレッドを用意したのは、学習には時間がかかるため、アプリケーションが止まったと誤解されないようにするため。

コミットメッセージをもとに自動的にラベリングしているが、単純にメッセージに「bug」と含まれていれば Faulty なコミット、「fix」と含まれていれば Non-Faulty なコミットとしてみなしているため、厳密な判別はしていない。ラベリングは手動で変更できる。

コミット差分は生成するフィルタ用データベースと同じディレクトリを作業場としてに生成される。このコミット差分はコミット時刻がひとつ前のコミットと比較して出力されている。コミット差分の出力時に、対象言語以外のソースコードやドキュメントの差分は、学習におけるノイズになりかねないので取得しない。

なお、JGit は diff 経由で差分を表示する必要があるので、初回のコミットについては出力することができない。

4.6.2　CodeLamp-E 及び CodeLamp-J における仕組み

ソースファイルの自動判別、及び手動判別・学習

別スレッドにて判別キュー、学習キューの監視を行っている。各キューが要素を持っていた時はそのソースファイルを判別、もしくは学習する。

また別のスレッドにて対象プロジェクトを監視し、ソースファイルの変更・追加・削除を監視している。なお、この監視は、リソースを少しでも少なく抑えるために、フィルタに関連付けられた言語の拡張子以外のファイルを無視している。

自動判別は、ソースファイルが待機状態 (No order)・除外指定 (exclude) ではないときに、最終判別時間よりも最終更新時間が遅いときに、用判別とみなして学習キューに追加する。

Git コミットごとの自動判別、及び手動判別・学習について

別スレッドにて、該当 Git リポジトリの HEAD の変更を監視している。HEAD が変更された場合、Git コミット一覧の再取得と、新しい HEAD に関して判別キューに追加する。

ソースファイルの方と異なり、同じスレッドで判別・学習キューの監視も行っている。これは、コミットの方がファイル更新よりも頻度が低いため、干渉する可能性は低いと考えられるため。こちらもデータベース作成時と同じく、対象言語以外のソースコードやドキュメントの差分は、学習におけるノイズになりかねないので取得しない。

5. 動作実験

5.1 実験目的

一連の動作が行えることを確認し、本実装における要件を満たしているか確認する。

5.2 実験準備

動作確認実験として次の 2 種類を行った。

1. 実験 1 Java で実装されたあるプロジェクトについて、過去のコミットをもとに学習したデータベースから、特定の 5 コミットを対象として判別する。
    - CodeLamp-Creator を使用して自動ラベリングに従って学習したデータベースを使って、CodeLamp-J にて対象コミットについて手動で判別する。
2. 実験 2 複数の GIt リポジトリのコミットから学習データベースを作成し、それをもとに実験 1 と同じ対象について判別する。
    - CodeLamp-Creator を用いて自動ラベリングをもとに学習・生成したデータベースを用いて、CodeLamp-E にて当プロジェクトを判別する。

5.3 実験対象

1. 実験 1
    - バグデータベースと対応して、すでに Faulty/Non-Faulty 箇所を該当コミットにタグ付けしている Git リポジトリを対象とする。
        - mina[18]
        - openjpa[19]
        - james[20]
    - コミットメッセージ中に含まれる「bug」「fix」の単語に従った自動分類を学習内容とし、バグタグ付け済みコミットについて判別結果を取得する。
2. 実験 2

- 学習用プロジェクト
  - mina
  - openjpa
  - james
  - eclipse[21]
  - junit[22]
  - tomcat[23]
  - jetty[24]
- 実験 1 と同じ判別対象のコミット差分について、判別結果を取得する

5.4 確認する要件

ツールに関して実験時に確認すべき点として、特に次の要件を挙げる。

- 要件 1 Eclipse 本体をよび Java 実行環境を除く外部ツールの利用、特にインストールの必要な外部ツールの利用を避ける。
- 要件 2 Eclipse はマルチプラットフォームアプリケーションであるので、同様に、なるべくマルチプラットフォームで利用できるツールとする。
- 要件 3 シンプルである。つまり、使い方が過剰に複雑化しない。
- 要件 4 使用者によってツールの利用方法は一定ではないことに配慮したインターフェースを作る。
- 要件 5 安定した動作をする。

5.5 実験の結果

対象となるコミットは、コミットメッセージから自動判別して学習したところ、表 5.1 の量の学習量が得られた。

対象となったコミット数が多くても、内容 (ソースコード) の文量や、学習済み単語数との重複で一回の学習量は異なる。作成したデータベースのバケットは最大 3396996 個で、比較的学習量が多かった james で全てのバケットが使用されたが、各

表 5.1　フィルタ用データベースごとの学習量

学習対象	分類	使用バケット数
mina	Faulty	45395
	NonFaulty	2027588
openjpa	Faulty	111894
	NonFaulty	524288
james	Faulty	3396996
	NonFaulty	3396996
実験 2	Faulty	3396996
	NonFaulty	3396996

バケットはそれぞれ異なる値を保有するため、データベースの最大まで学習した訳ではない。

これらのデータベースを用いて行った実験結果表 5.2 のように求まった。

予測結果である probably 値 (Classified val) が 0 であるものというのは、「Faulty」データベースに予測対象の単語対があまり学習されていなかったため、「Faulty」予測が非常に小さく出てしまったため、probably 値が 0 で「Faulty」と判別されてしまっている。

実験 2 では、より学習量の多いデータベースを使用したにも関わらず、結果からはその影響を見ることが出来なかった。

5.6　考察

これら実験 1、2 はツールの動作確認が目的だったが、利用時における課題として、次のことが確認できた。

- 判別精度
  - 学習量の少ないリポジトリでは特に判別できなかった。一方で、学習量が実験対象の中では比較的多くても、今回の実験で行った程度の学習量では不足しているように見えた。

表 5.2　対象のバグコミットについての判別実験 1,2 の結果

Reposiory Name	Commit ID	Time	実験 1 の結果 ClassifiedVal	Result	実験 2 の結果 ClassifiedVal	Result
mina	af37d31	2012/2/1515:22:50	0	NonFaulty	0	NonFaulty
	cc20296	2011/9/2317:29:04	0.5	Faulty	0	NonFaulty
	7c581a8	2011/8/2620:21:42	0	NonFaulty	0	NonFaulty
	e33bd1c	2010/9/1023:03:56	0	NonFaulty	0.5	Faulty
	10e9280	2010/9/1011:20:27	0	NonFaulty	0	NonFaulty
openjpa	7063dd5	2012/3/171:43:34	0	NonFaulty	0	NonFaulty
	b68cf1d	2012/3/116:26:31	0	NonFaulty	0	NonFaulty
	ecf492a	2012/3/80:20:04	0.5	Faulty	0	NonFaulty
	ce44b7d	2012/2/255:35:02	0	NonFaulty	0	NonFaulty
	c275da6	2012/2/251:09:31	0	NonFaulty	0	NonFaulty
james	34a875d	2012/2/2117:30:43	0.5	Faulty	0	NonFaulty
	8c512fb	2012/2/1420:23:19	0.5	Faulty	0.5	Faulty
	d7ecf44	2012/2/520:12:58	0.5	Faulty	0.5	Faulty
	f968934	2012/1/272:12:06	0	NonFaulty	0.5	Faulty
	73cec66	2012/1/2019:57:41	0	NonFaulty	0	NonFaulty

– とは言え、この実験程度の学習量でも、データベース作成には非常に時間がかかったため、個人で多くのソースコードを学習したデータベースを作成するのは負担が大きい。

– 学習データにも問題は多く、コミットメッセージから「bug」か「fix」かで分類しただけなので必ずしも適切なコミットを取得できていないという懸念はあった。

– 学習対象として、バグの発生が確認されたとバグデータベースからタグ付けられたコミットを対象としたが、それぞれタグ付けされた時点の 1 コミットについて判別しただけなので、必ずしもバグコードが含まれている

部分ではなかった (それ以前のコミット時にバグが含まれたが発見されていなかったものなど) 可能性もある。

5.7 要件に対する結果の達成具合

ツールとしての要件

(5.4 より再掲)

- 要件 1 Eclipse 本体をよび Java 実行環境を除く外部ツールの利用、特にインストールの必要な外部ツールの利用を避ける。
- 要件 2 Eclipse はマルチプラットフォームアプリケーションであるので、同様に、なるべくマルチプラットフォームで利用できるツールとする。
- 要件 3 シンプルである。つまり、使い方が過剰に複雑化しない。
- 要件 4 使用者によってツールの利用方法は一定ではないことに配慮したインターフェースを作る。
- 要件 5 安定した動作をする。

5.7.1 要件の達成評価と課題

先に提示した要件について確認する。

- 要件 1・2
  - Fault-prone フィルタリングを速く・効率的に行うためには crm114 を使用する必要があった。Windows ではバイナリファイル単体で利用できたため、ツール内のディレクトリに配置すればよかったが、Unix で同じバージョンの crm114 を使用するためには crm114/tre のインストールが必須となってしまうため、達成できたとは言えない。
- 要件 3・4
  - 基本的に表示やボタン数を抑えて作成したが、使用者がシンプルと受け止めるほどに収まったか、少なすぎてわかりにくいととらえるかは、評価 (6 章) を受けて修正する必要がある。

- Eclipse プラグインとして作成した本ツールは、他のツールの邪魔にならないように配置することになる。即時判別結果が必要でない (コミットした際に分かれば十分) という人は、ワークスペースのタブを切り替えて利用するし、即時確認したければ、左右で縦に長く・下で左右に広く・あるいはそれらを分断するように小さく、なるように配置する。
  - 大きく使用したとき、ボタンの位置が隅になるため、利用者によっては操作しにくく感じると思われる
  - 左右を狭くしたとき、にボタンや一部の表示がつぶれてしまう。
- 操作を限定しすぎて現在の対象 (フィルタ用データベースなど)・進行状況が分かりにくいので、修正が必要である。

• 要件 5
  - 実験において、CodeLamp-Creator を使用時に、特定のタイミングから、学習終了まで表示が固まることが確認された。この現象は差分が特定のサイズを超えて、判別に時間がかかると発生しているように思われる。
  - 判別・学習に時間がかかるのだが、その間に取り消し処理が出来ない。少なくとも、待ち状態の対象に関して取り消し機能を追加することが必要である。

6.　試験利用による評価の予定

本研究はツールの開発であり、Eclipse プラグインとして利用されることを想定した開発である。ツールの最適化のためには利用者からの感想を受けて改善を図ることが必要である。

本研究報告では、開発と共同開発先との時期の都合もあり、完成版の評価をまとめることができなかったため、

- 製作途中にいただいた意見とその反映
- 完成版の評価項目の予定
- 完成版の評価に対する反映の予定

について、次のようにまとめた。

6.1　製作途中にいただいた意見とその反映

製作過程中のアプリケーションを見ていただく機会があり、その際に、「Git のコミット差分についても判別できるようにしてほしい」との意見を伺った。

開発初期において、「コミット差分の出力を行う」ことが困難であると認識したため、当時の予定では「ソースコードについて判別・学習する」、「ソースコードが更新された場合、そのソースコードについて判別する」という機能を目的として製作していた。その上で、

「開発過程では Git で管理していることが多く、コミットの差分について判別することで、『どの編集部分でバグが生じた可能性があるか』分かる方が都合がよい」

との率直な意見を頂き、再検討した結果現完成版において「Git のコミットについても判別・学習する」機能を追加するに至った。

6.2　評価項目

主に UI 面についての評価と、使用時の不都合・不具合、利用状況と予測精度から考慮すべき事象を予定している。

1. ユーザインターフェースについて

   (a) 表示情報のわかりやすさ
   (b) Eclipse の画面に対する本ツールの表示位置についての調査
   (c) 本ツールの基本 2 画面 (ソースコードビュー・Git コミットビュー) についての使用比率
   (d) ウィジェットの配置に関する意見

2. 使用時の不都合
   (a) 使用した Git リポジトリの構成 (ブランチなど) について
   (b) データベースの設置個所について　(サーバーに設定して利用した、など)
3. 使用時の不具合 1. ツールがエラー終了した場合の状況について
4. 利用状況と予測精度
   (a) データベース作成時に利用したコミットの数に対する予測精度
   (b) データベース作成時に利用したプロジェクトと、使用者の関係
   (c) オープンソースプロジェクトから生成した配布データベースを使用した場合の予測精度精度

6.3 予想される評価と対応の予定

- 表示のわかりやすさ
  - コミットの一覧しか表示されていないので、構造がわかりにくい
- 判別について
  - もっと詳細な判別結果がわかった方がいい。対象コミットは複数のファイルの変更が記録されているので、それぞれについて判別結果が分かるように修正したい。
- ウィジェットの配置に関する意見・本ツールの基本 2 画面についての使用比率
  - 表示数は少なくてもいいので「ソースコードについて判別・学習する」と「Git のコミットについても判別・学習する」ことに関して一つの画面で見れた方がいいという意見が想定される。そのような意見が散見される場合、それぞれから表示を抽出する画面を追加することが考えられる。

- 使用した Git リポジトリの構成 (ブランチなど) について
  - 現モデルでは「全コミット」という枠組みで利用しているので、ブランチが多い場合などに、コミット情報が入れ交う・ブランチの境界で差分を取ろうとして、想定していない差分を取得してしまうことが考えられる。ブランチの切り替え機能を追加するという対処は考えられる。
  - 特にデータベースの作成時において、コミットばかり多くて分類しにくい、メッセージしか参考となる目印がなくて分かりにくいという意見も考えられる。その場合は、コミットの一覧表示をブランチやタグを基準に、ツリー形式にまとめることについて検討する。
- データベースと予測精度
  - 先行研究でデータベースに学習した量が多いほど予測精度は上がることは判明しているので、実際に使用者に任せて学習する量がどの程度のものかによって、この場合の実用的な予測精度を見込めるか否かが推測できる。ここから、推奨するコミット数を見込むことができると使用者に一任する是非が推測できる。
  - また、使用者以外が作成したソースコードにおけるデータベースでも予測精度が見込めるのであれば、データベースの作成において、新規作成するよりも、ある程度の精度のデフォルトデータベースに対して使用者が自分のソースコードを学習させて利用する形式の方が現実的であるという判断が行える場合がある。

7. 結言

本報告では Fault-prone フィルタリングの実用化を目指し、開発環境 Eclipse 用のプラグインとしたツール「CodeLamp」の開発を行った。事前に定めた仕様を満たす実装を行い、また、現場からの意見による仕様改訂を通じて、実践的なツールとして整備した。

今後の課題として、以下のようなことが挙げられる。

概ね動作を確認することはできたが、Git リポジトリの構造をわかりやすくビューに反映させる必要がある。

実験 1 の結果からは、1 プロジェクトほどの大きさでは学習量が圧倒的に少なく、判別精度が低いことがわかる。しかし、個々の所持するプロジェクトの量はそれほど多くないので、実験 2 のように、まったく別のプロジェクトを利用して結果を得られることが望ましい。

まず、学習段階において大量のノイズが混入しているので、それをいかに取り除くかという問題が認識されている。一方で、使用者がノイズなく学習させることができるような仕組みの開発も必要である。、非常に困難であるのも事実である。

一方で、一連のツール群として CodeLamp を整備してため、系統的に評価実験を行うことが可能となった。また、開発補助ツールとして開発現場での使用の道を拓くこともできるため、今後の共同研究における活用が期待できる。

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、本学情報工学部門工織一郎教授に厚く御礼申し上げます．貴重なデータをご提供頂きました、本学大学情報工学・人間科学系 水野修准教授に深く感謝致します．本報告書執筆にあたり貴重な助言を多数頂きました、本学情報工学専攻氏、氏、釆野友紀也氏、河端駿也氏、山田晃久氏、藤原剛史氏、森啓太氏、Nicolas Fruy 氏、Anais Tournois 氏、情報工学課程 田中健太郎氏、西浦生成氏、原田禎之氏をはじめとする、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] eclipse.org, Eclipse, （オンライン）, 入手先 ⟨https://eclipse.org/⟩（参照 2016-2-14）.

[2] eclipse.org, JGit (JavaAPI), （オンライン）, 入手先 ⟨http://www.eclipse.org/jgit/⟩（参照 2016-2-14）.

[3] H. Yukinao, “Application of trend of tokens in source code modules to fault-prone module prediction,” 2012.

[4] crm114.source.forge.net CRM114 Discriminator, （オンライン）, 入手先 ⟨http://crm114.sourceforge.net/⟩（参照 2007-4-19）.

[5] P.Nesi P.Bellini, I.Bruno and D.Rogai, “Comparing fault-proneness estimation models,” Proc. of 10th IEEE International Conference of Engineering of Complex Computer Systems, pp.205–214, 2005.

[6] W.L.Melo L.C.Briand and J.Wust, “Assessing the applicability of fault-proneness models across object-oriented software projects,” IEEE Trans. on Software Engineering, vol.28, no.7, pp.706–720, 2002.

[7] B.Cukie L.Guo and H.Singh, “Predicting fault prone modules by the dempster-shafer belief networks,” Proc. of 18th International Conference on Automated Software Engineering, pp.249–252, 2003.

[8] Google.com, Bug Prediction at Google, （オンライン）, 入手先 ⟨http://google-engtools.blogspot.jp/2011/12/bug-prediction-at-google.html⟩（参照 2011-12-15）.

[9] I. Grigorik, github igrigorik/bugspots (MIT License), （オンライン）, 入手先 ⟨https://github.com/igrigorik/bugspots⟩（参照 2011-12-15）.

[10] O. Mizuno and T. Kikuno, “Training on errors experiment to detect fault-prone software modules by spam filter,” Proc. of 5th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundation of software engineering, pp.405–414, 2007.

[11] S.N. Osamu Mizuno, Shiro Ikami and T. Kikuno, “Fault-prone filtering: Detection of fault-prone modules using spam filtering technique,” 2007.

[12] O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno, "Spam filter based approach for finding fault-prone software modules," Proc. of Fourth International Workshop on Mining Software Repositories (MSR07), p.4, May 2007. Minneapolis, MN, USA.

[13] 森　啓太，水野　修，"スパムフィルタに基づく即時バグ予測ツールの試作," ソフトウェア・シンポジウム 2015，pp.37–46，June 2015.

[14] 藤原剛史，水野　修，"バイトコードを用いたテキスト分類による不具合予測," ソフトウェア・シンポジウム 2015，pp.80–88，June 2015.

[15] osbflua.luaforge.net, Exponential Differential Document Count, （オンライン）, 入手先〈http://osbf-lua.luaforge.net/papers/osbf-eddc_slides.pdf〉（参照 2006-03-28）.

[16] hebbut.net, CRM114 for win32/UNIX, （オンライン）, 入手先〈http://hebbut.net/Public.Offerings/crm114.html〉（参照 2009-2）.

[17] Oracle.com, Java Runtime Environment, （オンライン）, 入手先〈https://java.com〉（参照 2016-2-5）.

[18] apache.org, github apache/mina, （オンライン）, 入手先〈https://github.com/apache/mina〉（参照 2012-2-18）.

[19] apache.org, github /openjpa, （オンライン）, 入手先〈https://github.com/apache/openjpa〉（参照 2012-4-31）.

[20] apache.org, github /james, （オンライン）, 入手先〈https://github.com/apache/james〉（参照 2012-4-29）.

[21] eclipse.org, github eclipse/eclipse, （オンライン）, 入手先〈https://github.com/eclipse/eclipse〉（参照 2016-2-9）.

[22] apache.org, github /junit, （オンライン）, 入手先〈https://github.com/junit-team/junit〉（参照 2016-2-11）.

[23] apache.org, github apache/tomcat, （オンライン）, 入手先〈https://github.com/apache/tomcat〉（参照 2016-2-12）.

[24] eclipse.org, github eclipse/jetty, （オンライン）, 入手先〈https://github.com/eclipse/jetty〉（参照 2016-2-12）.