特別研究報告

題目

変数使用とメソッド呼び出しに着目した Fault-Prone メソッド特定手法の提案と評価

指導教員
楠本 真二 教授

報告者
兼光　智子

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

変数使用とメソッド呼び出しに着目した Fault-Prone メソッド特定手法の提案と評価

兼光　智子

内容梗概

近年，ソフトウェア開発の大規模化・複雑化・開発期間の短縮化に伴い，ソフトウェアのすべてのモジュールに同様の力を注ぐことは困難になってきている．Fault-Prone モジュール，つまりフォールトを含んでいる可能性が高いモジュールを予測し，そのモジュールに力を注ぐことで効率よく開発や保守が行える．

一般的な Fault-Prone モジュールの予測手法では，ソフトウェア保守性を評価するための複雑度メトリクスの値を説明変数として予測モデルを作成し，予測したいモジュールに適用することが多い．しかし，予測に使用するメトリクス値を測定するモジュールの単位は，ファイルやクラスの場合が多い．モジュールの単位は小さい方が，問題を特定しやすくより適切な箇所に力を注ぐことができ開発や保守の効率が上がる．

そこで，本研究ではメソッド単位でのメソッド呼び出しと変数使用に着目し，Fault-Prone メソッドを特定する手法を提案する．具体的には，メソッド呼び出しは，内部メソッド呼び出しと外部メソッド呼び出しの 2 種類に分けて考え，そのうちの内部メソッド呼び出しの割合が大きいものほどバグが含まれる可能性が高いと考える．変数使用情報は，変数の使用可能な範囲と実際に使用されている範囲に着目し，その差が大きいものほどバグが含まれる可能性が高いと考える．

次に，提案手法を実際のソースコードに適用しバグ情報との関連を見ることで提案手法の有効性を調査した．その結果，メソッド呼び出しを用いた予測とバグとの相関は低く従来のメトリクスであるコード行数やサイクロマチック数と比べてもバグとの相関は低かった．しかし，変数使用情報を用いた予測は，従来のメトリクスであるコード行数やサイクロマチック数よりバグとの相関があることを確認した．また，変数の使用可能な範囲と実際に使用されている範囲の差が大きいものが，簡単なリファクタリングによってその差を小さく出来ることを示した．

主な用語

Fault-Prone
複雑度メトリクス
バグ予測
ソフトウェア保守

目 次

1　まえがき

近年，ソフトウェアの応用分野の拡大と共にソフトウェアが大規模・複雑化し，ソフトウェアの保守に要するコストが増加してきている．それに伴い，開発期間の短縮やコストの削減が求められており，ソフトウェアのすべてのモジュールに同様の力を注ぐことは困難になってきている．ソフトウェアテストは多くの開発コストが費やされており，その削減はソフトウェア開発の効率化において有効な手段である．Fault-Prone モジュール，つまりフォールトを含んでいる可能性が高いモジュールを予測し，そのモジュールに力を注ぐことで効率よく開発や保守が行え，品質の向上にもなる．

一般的な Fault-Prone モジュールの予測手法では，ソフトウェア保守性を評価するための複雑度メトリクスの値を説明変数として予測モデルを作成し，予測したいモジュールに適用することが多い．しかし，予測に使用するメトリクス値を測定するモジュールの単位は，ファイルやクラスの場合が多い．モジュールの単位は小さい方が，問題を特定しやすくより適切な箇所に力を注ぐことができ開発や保守の効率が上がる．

そこで，本研究ではメソッド単位でのメソッド呼び出しと変数使用に着目し，Fault-Prone メソッドを特定する手法を提案する．具体的には，メソッド呼び出しは，内部メソッド呼び出しと外部メソッド呼び出しの 2 種類に分けて考え，そのうちの内部メソッド呼び出しの割合が大きいものほどバグが含まれる可能性が高いと考える．変数使用情報は，変数の使用可能な範囲と実際に使用されている範囲に着目し，その差が大きいものほどバグが含まれる可能性が高いと考える．

次に，提案手法を実際のソースコードに適用しバグ情報との関連を見ることで提案手法の有効性を調査した．

以降，2 節では本研究に関する種々の用語に関する説明を行う．続いて，3 節では本研究に関連する研究を紹介する．4 節では，提案手法の具体的な定義について述べ，5 節では，4 節の提案手法を実装したツールについて説明を行う．6 節では，4 節の提案手法を用いた実験を説明し，その考察を 7 節で行う．最後に 8 節で，まとめと今後の課題について述べる．

2　準備

2.1　複雑度メトリクス

この研究では，ソースコードの複雑さを計測するメトリクスを単に複雑度メトリクスとよぶ．複雑度メトリクスは多く提案されており，代表的な複雑度メトリクスとして以下のものがある．

2.1.1　コード行数

基本的な複雑度メトリクスとして行数がある．ソースコードの行数が多いということはそれだけ規模が大きく，ゆえに複雑であるといえる．空行とコメントのみの行を除いた行をもってコード行数とすることが多い．

2.1.2　サイクロマチック数

サイクロマチック数とは，McCabe によって提案されたメトリクスである [1]．プログラム制御の流れを有向グラフで表現したときの枝の数 e，節点の数 n を用いて $e - n + 2$ で表される．この値は直観的にはソースコードの分岐の数に 1 を加えた数を表す．サイクロマチック数が多いと，テストケース（ホワイトボックステスト）を作成する手間がかかり，保守性が下がると指摘されている．サイクロマチック数は経験的に 10 以下に抑えることが望ましいといわれている [2]．

2.1.3　CK メトリクス

CK メトリクスとは，Chidamber と Kemerer らの提案したメトリクスである [3]．オブジェクト指向ソフトウェアに対する代表的なメトリクスであり，クラスの複雑度を静的に評価する．計測する内容ごとに下記の 6 つのメトリクスが定義されている．全てのメトリクスにおいて，計測値が大きいほうが，複雑なクラスであり好ましくないことを示す．

WMC(weighted methods per class)

クラスのメソッドの重さ（複雑さの総和）を計測する．WMC の値が大きいほど，複雑であり保守のコストがかかることを示唆する．WMC では，メソッドをどのように重みづけするかは定義されていない．実際は，サイクロマチック数や Halstead のソフトウェアサイエンス [4] が用いられることが多い．

LCOM(lack of cohesion of methods)

クラスの凝集性の欠如を計測する．共通のインスタンス変数を使用しないメソッドのペア（つまり 2 つのメソッド）の数から，共通のインスタンス変数を使用するメソッドのペアの数を引いた値で表される．LCOM の値が大きいほど，凝集性が低く，保守性が低いことを示唆する．

DIT(depth of inheritance tree)

クラスのスーパークラスの数を計測する．そのクラスから継承木の根への長さで表される．DIT の値が大きいほど，継承されている変数やメソッドが多いことを表す.

NOC(number of children)

クラスのサブクラスの数を計測する．子孫ではなく子の数である．NOC の値が大きいほど，サブクラスへの影響力が大きいので，保守をする場合には注意をする必要があることを示唆する.

CBO(coupling between objects)

クラスに関係しているクラスの数を計測する．ここで，関係しているとは，一方のメソッドが他方のメソッドかインスタンス変数を使用することを言う．CBO の値が大きいほど，他のクラスに依存していることを物語り，複雑で保守のコストがかかることを示唆する.

RFC(response for a class)

クラスが呼び出すメソッドの数を計測する．そのクラスのメソッドの数と，そのクラスのメソッドが呼び出す他クラスのメソッド全ての和集合の要素の数で表される．RFC の値が大きいほど，受信したメッセージを遂行するために発信しなければならないメッセージの数が多いことを物語り，複雑なクラスであることを示唆する.

2.2 Fault-Prone

Fault-Prone とは，フォールトを含んでいる確率が高いという意味である．ソフトウェア開発において，Fault-Prone モジュールを予測する研究が広く行われている．例えば Basili らは，オブジェクト指向プログラムに対する代表的なメトリクス CK メトリクス [3] を用いて，ソフトウェア開発の早期段階で Fault-Prone クラスの予測が行えることを示した [5].

Fault-Prone モジュールを特定できれば，テスト工程において，Fault-Prone モジュールにより多くのテスト工程を割り当てることで，無駄なテスト工程を省き，ソフトウェア開発の効率化を図れ信頼性も向上する [6]．Arisholm らは，レガシーシステムにおいて過去のバージョンのフォールトや変更情報を用いて新しいバージョンにおける Fault-Prone モジュールを予測したとき，大きくテスト効率が上がったと報告している [7].

モジュールから計測されたメトリクス値（コード行数，サイクロマチック数等）を説明変数とし，モジュールのフォールトの有無を目的変数とする Fault-Prone モジュール予測モデルが多数提案されている．予測に使用するメトリクス値を測定するモジュールの単位は，ファイルやクラスの場合が多い.

2.3 ソフトウェアの保守性

ソフトウェアの品質には様々なものがあるが，ISO/IEC9126 ではソフトウェアの保守性を次のように定義している [8].

保守性

> 修正のしやすさに関する能力．修正は，是正もしくは向上，又は環境の変化，要求仕様の変更及び機能仕様の変更にソフトウェアを適応させることを含めてもよい．

一般に，ソフトウェアの保守性を評価するためには複雑度メトリクスやクローンメトリクスが用いられることが多い [9][10].

3　関連研究

Fault-Prone モジュールを特定できれば，テスト工程において，Fault-Prone モジュールにより多くのテスト工程を割り当てることで，無駄なテスト工程を省き，ソフトウェア開発の効率化を図れ信頼性も向上する [6]．Arisholm らは，レガシーシステムにおいて過去のバージョンのフォールトや変更情報を用いて新しいバージョンにおける Fault-Prone モジュールを予測したとき，大きくテスト効率が上がったと報告している [7].

プログラム自体の規模や複雑さに起因するバグは運用前の単体レベルのテストで見つけやすく修正されている．一方，多くのプログラムを呼び出したりデータを参照したりするプログラムを修正する場合，その影響は広く及ぶため確認するため保守に要する工数が増え，見落としが起こるリスクも高くなる．つまり，保守時の品質に与える要因の 1 つとして，修正時の影響が及ぶ範囲の大きさが考えられる．早瀬らは，プログラムから呼び出されるプログラムや参照・更新されるデータを辿り影響範囲の大きさを求め保守ポイントを特定し，保守作業の労力見積りに用いることができるメトリクスを提案した [11].

Basili らは，オブジェクト指向で開発されたソフトウェアに対して，メトリクス CK メトリクス [3] が，従来の複雑さメトリクスよりもエラーの個数と相関が高いことを示した．また，ソフトウェア開発の早期段階で Fault-Prone クラスの予測が行えることを示した [5].

門田らは，20 年以上前に開発され，拡張 COBOL 言語で記述された大規模なレガシーソフトウェアを題材とし，代表的なソフトウェア品質である信頼性・保守性とコードクローンとの関係を定量的に分析した．改版により多くの保守コストが費やされるため，改版数が多いほど保守コストが費やされる．分析の結果，モジュールに含まれるコードクローンのサイズが大きいほど改版数がより大きい傾向にあることがわかった [9].

馬場は，従来の複雑度メトリクスの他にコードクローンに関するメトリクスも説明変数に加えたロジスティック回帰分析によって Fault-Prone モジュールを予測した [10]．しかし，予測したモジュールの単位はコンポーネントとファイルであり，メソッド単位では行われていない．COBOL のような言語ではプログラムとファイル・データベースを粒度とし，Java のような近代的な言語ではメソッドやフィールドを粒度として扱うのが良いといわれている [12].

4 提案手法

4.1 提案手法の概要

本研究では，メソッド呼び出しと変数使用の情報を用いて Fault-Prone メソッドを特定する．メソッド呼び出しは，内部メソッド呼び出しと外部メソッド呼び出しの 2 種類に分けて考え，その割合とバグとの関係を調査する．変数使用情報は，変数の使用可能な範囲と実際に使用されている範囲に着目し，その範囲の差とバグとの関係を調査する．

4.2 提案手法の定義

4.2.1 メソッド呼び出し

この研究では，メソッド呼び出しを以下の 2 種類に分けて考える．

内部メソッド

対象システム内で定義したメソッドの呼び出し

外部メソッド

あらかじめライブラリなどで定義されているメソッドの呼び出しであり，次のようなものが当てはまる．

- System.out.println(“ Hello ”);
- java.io.File file=new java.io.File(filename);
  file.getName();

この 2 種類のメソッドの割合によってそのメソッドの主な役割がわかる．内部メソッド呼び出しの割合が多ければ対象システム内の処理制御，外部メソッド呼び出しの割合が多ければ実際に行う基本的な機能処理が主な役割だと思われる．

外部メソッドはライブラリで提供されているため仕様の変更がほとんどないが，内部メソッドは開発によって仕様の変更が生じる可能性がある．内部メソッドの仕様が変更された場合そのメソッドを呼び出している側でも変更が必要な可能性がある．そこで内部メソッドに重点を置き，内部メソッド呼び出しの割合を式 (2) と定義し，内部メソッド呼び出しの割合が高いほどバグが含まれる可能性が高いと考える．

$$\text{総メソッド呼び出し数} = \text{外部メソッド呼び出し数} + \text{内部メソッド呼び出し数} \tag{1}$$

$$\text{内部メソッド呼び出しの割合} = \text{内部メソッド呼び出し数}/\text{総メソッド呼び出し数} \tag{2}$$

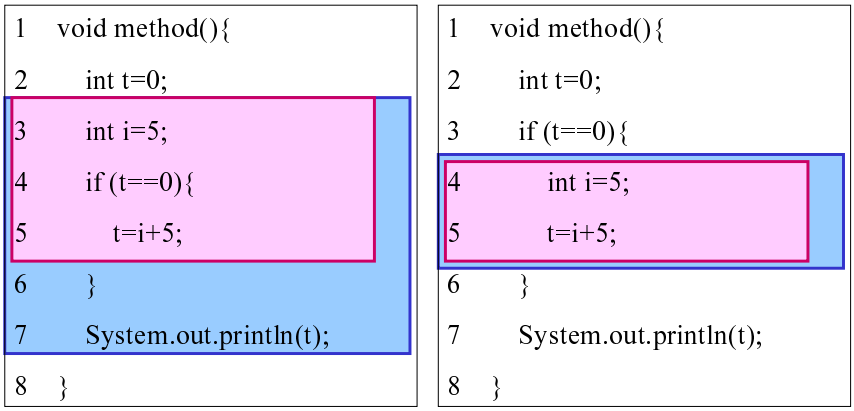


図 1: 変数のスコープと実際に使用されている範囲の例
青は変数のスコープを表し，赤は実際に使用されている範囲を表す

4.2.2 変数使用

変数の使用可能な範囲と実際に使用されている範囲について考える．

例として図 1(a) のメソッドを考える．変数 i は 3 行目で宣言されており，7 行目まで使用可能である．しかし，実際に最後に使用されているのは 5 行目である．このように変数の使用可能な範囲と実際に使用されている範囲には差がある場合がある．

変数 i は if 文の中でのみ使用されているので，図 1(b) のように変数 i の宣言を if 文の中で行うとする．変数 i は 4 行目で宣言されており，5 行目まで使用可能である．実際に最後に使用されているのは 5 行目である．このように変数の宣言位置を変更することによって，変数の使用可能な範囲と実際に使用されている範囲の差が小さくなる．

変数の使用可能な範囲と実際に使用されている範囲について以下のように定義する．

変数のスコープ

変数の使用可能な範囲であり，図 1(a) の変数 i では，3 行目から 7 行目である．

変数が実際に使用されている範囲

変数が宣言されてから最後に使用されるまでであり，図 1(a) の変数 i では，3 行目から 5 行目である．

変数が実際に使用されている範囲は常に変数の使用可能な範囲に含まれる．変数の使用可能な範囲に比べ変数が実際に使用されている範囲が極端に小さい場合，変数のスコープが無駄に広く変数の誤った使用が起こる可能性がある．

変数の使用可能な範囲と実際に使用されている範囲の差を評価する指標として次の 2 つを定義する．

変数使用行率

変数のスコープのうち実際に使用されている範囲の割合であり，式 (3) のように定義する．

$$\text{変数使用行率} = \text{実際に使用されている範囲の行数}/\text{変数のスコープ行数} \tag{3}$$

この値が小さいほど，変数の使用可能な範囲と実際に使用されている範囲の差が大きくなる．

図 1(a) の変数 i では，実際に使用されている範囲の行数は 3 行，変数のスコープ行数は 5 行なので，変数使用行率は

$$\begin{aligned}\text{変数使用行率} &= 3/5 \\ &= 0.6\end{aligned}$$

よって，0.6 となる．

変数使用行差

変数の使用可能な範囲と実際に使用されている範囲との差の行数であり，式 (4) のように定義する．

$$\text{変数使用行差} = \text{変数のスコープ行数} - \text{実際に使用されている範囲の行数} \tag{4}$$

この値が大きいほど，変数の使用可能な範囲と実際に使用されている範囲の差が大きくなる．

図 1(a) の変数 i では，変数のスコープ行数は 5 行，実際に使用されている範囲の行数は 3 行なので，変数使用行差は

$$\begin{aligned}\text{変数使用行差} &= 5 - 3 \\ &= 2\end{aligned}$$

よって，2 となる．

変数使用行率が小さい変数や変数使用行差が大きい変数を含むメソッドの方がバグが含まれる可能性が高いと考える．

変数使用行率・変数使用行差は変数ごとに算出する値であるが，1 つのメソッドには複数の変数が含まれる．メソッドの特性としての値を 1 つに決めるため，その変数ごとの値から平均をとるなどして算出する．変数使用行率・変数使用行差は，その変数のスコープ行数が小さいと変数使用行率は大きく変数使用行差は小さくなる．しかし，変数のスコープ行数が大きいほど与える影響は大きいと考え，通常の平均だけでなく変数のスコープ行数で重みを付けた加重平均と変数使用行率では最低値，変数使用行差では最大値も評価する．各値を次のように定義する．なお各値の例として図 1(a) のメソッド内の変数を使用する．変数 i の変数使用行率は 0.6，変数使用行差は 2，スコープ行数は 5 であり，変数 t の変数使用行率は 1.0，変数使用行差は 0，スコープ行数は 6 である．

平均変数使用行率

変数ごとの使用行率の和を変数の数で割る

$$\text{平均変数使用行率} = \frac{1}{n}\sum_{i=1}^{n} \text{変数 } a_i\text{の変数使用行率} \tag{5}$$

図 1(a) のメソッドでは,

$$\begin{aligned}\text{平均変数使用行率} &= \frac{1}{2}(0.6 + 1.0) \\ &= 0.8\end{aligned}$$

よって，0.8 となる.

加重平均変数使用行率

変数の使用行率にその変数のスコープ行数をかけたものの和を変数のスコープ行数の和で割る

$$\text{加重平均変数使用行率} = \frac{\sum_{i=1}^{n}(\text{変数 } a_i\text{の変数使用行率} \times \text{変数 } a_i\text{のスコープ行数})}{\sum_{i=1}^{n} \text{変数 } a_i\text{のスコープ行数}} \tag{6}$$

図 1(a) のメソッドでは,

$$\begin{aligned}\text{加重平均変数使用行率} &= \frac{0.6 \times 5 + 1.0 \times 6}{5 + 6} \\ &= \frac{9}{11} \\ &= 0.\dot{8}\dot{1}\end{aligned}$$

よって，$0.\dot{8}\dot{1}$ となる.

最低変数使用行率

メソッドに含まれる変数の使用行率の最低値

図 1(a) のメソッドでは,

$$0.6 < 1.0$$

よって，0.6 となる.

平均変数使用行差

変数ごとの使用行差の和を変数の数で割る

$$\text{平均変数使用行差} = \frac{1}{n}\sum_{i=1}^{n} \text{変数 } a_i\text{の変数使用行差} \tag{7}$$

図 1(a) のメソッドでは,

$$\begin{aligned}\text{平均変数使用行差} &= \frac{1}{2}(2 + 0) \\ &= 1\end{aligned}$$

よって，1 となる.

加重平均変数使用行差

変数の使用行差にその変数のスコープ行数をかけたものの和を変数のスコープ行数の和で割る

$$\text{加重平均変数使用行差} = \frac{\sum_{i=1}^{n}(\text{変数 } a_i\text{の変数使用行差} \times \text{変数 } a_i\text{のスコープ行数})}{\sum_{i=1}^{n} \text{変数 } a_i\text{のスコープ行数}} \tag{8}$$

図 1(a) のメソッドでは,

$$\begin{aligned}\text{加重平均変数使用行差} &= \frac{2 \times 5 + 0 \times 6}{5 + 6} \\ &= \frac{10}{11} \\ &= 0.\dot{9}\dot{0}\end{aligned}$$

よって，$0.\dot{9}\dot{0}$ となる．

最大変数使用行差

メソッドに含まれる変数の変数使用行差の最大値

図 1(a) のメソッドでは,

$$2 > 0$$

よって，2 となる．

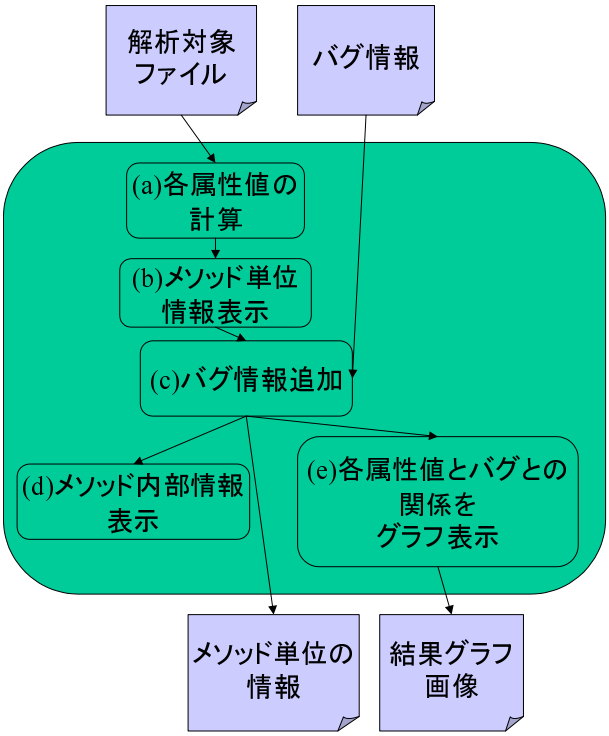


図 2: ツールの処理の流れ

5 実装

提案手法の計算を自動で行い，情報を表示するためにツールを実装した．Java 言語で書かれたプログラムを対象としている．

5.1 処理の流れ

実装したツールの処理について説明する．図 2 に全体の流れを示す．各ステップの詳しい内容は以下の通りである．

1. 入力
   (a) 解析対象ファイル
   (b) バグ情報ファイル
2. ツール内部処理
   (a) 各属性値の計算
      i. メソッド単位で種類別メソッド呼び出し数の合計を求める
      ii. メソッド内のローカル変数スコープ情報と平均変数スコープ行数を計算
      iii. メソッド内のローカル変数スコープ情報に最後に使用されている行情報を追加

iv. メソッドのサイクロマチック数を計算

(b) メソッド単位情報表示

i. 最低変数使用率順にメソッドをソート

ii. メソッド情報をテーブルに一覧表示

(c) バグ情報追加

i. バグ情報からバグの含まれていたクラス名メソッド名を取得

ii. 同じクラス名を検索

iii. 同じメソッド名を検索

iv. 該当メソッドのバグ数を 1 増加

(d) メソッド内部情報表示

i. メソッドの含まれるソースファイルからソースコードを読み込みソースコード表示テキストフィールドにセット

ii. メソッド開始行までスクロール

iii. 変数情報をテーブルに一覧表示

(e) 各属性値とバグとの関係をグラフ表示

i. 属性値の最大値を求め，値を区切る区間幅を計算

ii. 区間ごとにメソッド行数とバグ数の合計を求める

iii. 区間ごとにメソッド行数とバグ数から 1 行あたりのバグ数を求めグラフ表示

3. 出力

(a) メソッド単位の情報 (CSV 形式)

(b) グラフ画像 (png 形式)

5.2 GUI の説明

解析し表示する情報は次の通りである．

- メソッド単位
  - メソッド呼び出し情報
    - 内部メソッド呼び出し数
    - 総メソッド呼び出し数
    - 内部メソッド呼び出し割合
  - 変数情報

        * 変数の数
        * 平均変数スコープ行数
        * 変数使用行率
            - 平均変数使用行率
            - 加重平均変数使用行率
            - 最低変数使用行率
        * 変数使用行差
            - 平均変数使用行差
            - 加重平均変数使用行差
            - 最大変数使用行差
    - 行数
    - サイクロマチック数
    - メソッド名
    - クラス名
    - メソッド開始行
    - バグ数
- メソッド内の各変数単位
    - 変数名
    - 宣言開始行
    - 使用可能な最後の行
    - 変数スコープ行数
    - 最後に使用した行
    - 使用行率
    - 使用行差

メソッド単位での情報のうち，平均変数スコープ行数, 平均変数使用行率, 加重平均変数使用行率, 最低変数使用行率, 平均変数使用行差, 加重平均変数使用行差, 最大変数使用行差については，そのメソッド内の変数情報を用いて計算した値である．変数の数は，ローカル変数・引数・メソッド内で使用したフィールド変数の合計である．

メソッド単位のバグ数は，別ファイルとして用意したバグ情報を読み込みメソッドごとのバグ数を数えた値であり，各属性値とバグとの関係を見ることで属性値の有効性を調査するために追加した項目である．

ツールの実行画面を図 3 に示す．メソッド情報には 1 行に 1 つのメソッドに関する情報が表示される．

ソースコードを解析し結果を表示したあとは，テーブルヘッダをクリックすることによって各項目の値をキーとして行をソートすることが可能である．メソッド呼び出し情報や変数情報の値でソートすると上位に Fault-Prone メソッドが表示される．注目したいメソッドがある場合，そのメソッドの行を選択することでソースコードと変数情報の具体的な内容が表示される．各変数ごとの情報も表示されるため変数使用行率・変数使用行差の値が悪い変数を特定し，ソースコードで実際の使用状況を確認することができる．

また，メソッド情報とバグとの関係もグラフ表示する．

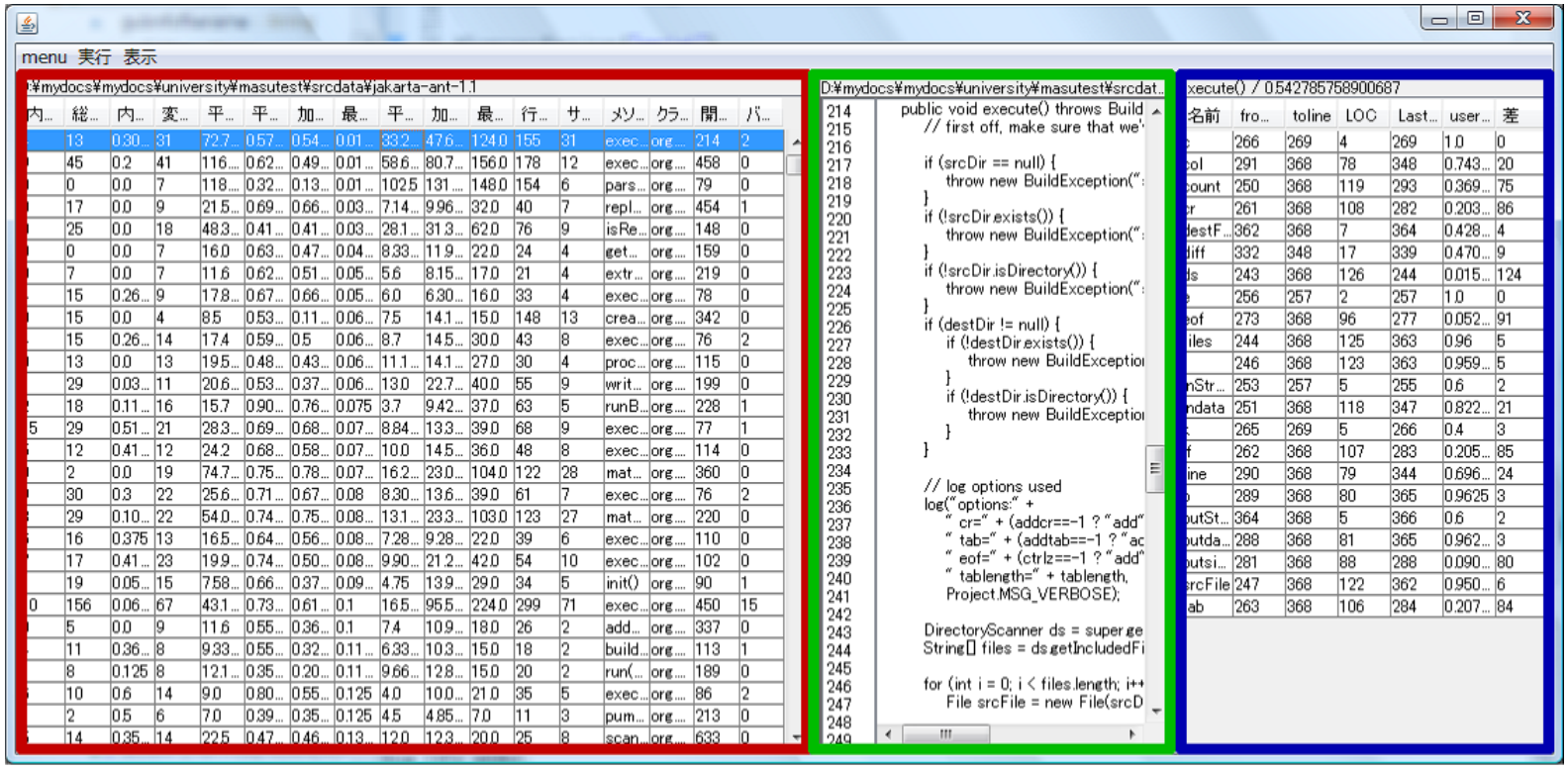


図 3: 作成したツールの実行画面

6 実験

提案手法の有効性を調査するため実験を行った.

6.1 実験対象

本実験では，Apache Ant(v1.1) と Ant のリポジトリよりメソッド単位で構築したバグ情報を用いた．Apache Ant(v1.1) の規模を表 1 に示す.

バグ情報はリポジトリの変更内容を見て，変更がバグの修正であるか，バグの修正ならばどのメソッドかを 1 つずつ確認し作成した．バグ情報を構築するのに使用した Ant のバージョンは 1.1 から 1.7.1 であり，1514 個のバグがメソッド単位で発見された．Ant 1.1 から Ant1.7.1 の開発期間は 7 年 11 カ月である.

6.2 実験方法

実験には本研究で作成したツールを用いた．メソッドの行数が多いほどバグを含みやすい．そこでメソッド行数によるバグの公平さを取るため 1 行あたりのバグ数と各属性値との関係をグラフにして比較した．また，各属性値とバグ数・メソッド 1 行あたりのバグ数について Kendall と Spearman の順位相関係数も求めた.

6.3 実験結果

実験結果のグラフは付録として本論文の最後に添付する.

外れ値のためにグラフの区間に対応するメソッド数に偏りがあるものもあり，1 行あたりのバグ数がない区間にはもともと対応するメソッドが存在しないものもあるので，提案手法により得た数値とバグ数を比較したものと個数のグラフを並べた.

各グラフの横軸は提案手法により得た各数値であり，0 から最大値までを 20 分割し，軸の数値で区切られた間の値を取るメソッドがその区間に含まれる．同じ属性値についてのメソッド数のグラフと 1 行あたりのバグ数のグラフでは横軸は一致する.

縦軸については，メソッド数のグラフは区間内の数値を取るメソッドの数である．バグ数のグラフは区間内の数値を取るメソッドの 1 行あたりのバグ数であり，区間内のメソッドに含まれたバグ数を

表 1: Apache Ant(v1.1) の規模

ソースファイル数	87
クラス数	112
行数	17,412
メソッド数	647

区間内のメソッド行数の和で割ったものである．

図 5(a) は内部メソッド呼び出し数とメソッド数の関係をグラフにしたもの，図 5(b) は内部メソッド呼び出し数と 1 行あたりのバグ数の関係をグラフにしたものである．図 6(a) は総メソッド呼び出し数とメソッド数の関係をグラフにしたもの，図 6(b) は総メソッド呼び出し数と 1 行あたりのバグ数の関係をグラフにしたものである．図 7(a) は内部メソッド呼び出しの割合とメソッド数の関係をグラフにしたもの，図 7(b) は内部メソッド呼び出しの割合と 1 行あたりのバグ数の関係をグラフにしたものである．各属性値の最大値と区間の幅を表 4 に示す．

内部メソッド呼び出し数と総メソッド呼び出し数が多いほど 1 行あたりのバグ数がやや多い．しかし，内部メソッド呼び出しの割合と 1 行あたりのバグ数には関連性は認められなかった．

図 8(a) は平均変数スコープ行数とメソッド数の関係をグラフにしたもの，図 8(b) は平均変数スコープ行数と 1 行あたりのバグ数の関係をグラフにしたものである．図 9(a)・図 10(a)・図 11(a) は変数使用行率のそれぞれ平均・加重平均・最低値とメソッド数の関係をグラフにしたもの，図 9(b)・図 10(b)・図 11(b) は変数使用行率のそれぞれ平均・加重平均・最低値と 1 行あたりのバグ数の関係をグラフにしたものである．図 12(a)・図 13(a)・図 14(a) は変数使用行差のそれぞれ平均・加重平均・最大値とメソッド数の関係をグラフにしたもの，図 12(b)・図 13(b)・図 14(b) は変数使用行差のそれぞれ平均・加重平均・最大値と 1 行あたりのバグ数の関係をグラフにしたものである．各属性値の最大値と区間の幅を表 5 に示す．

平均変数スコープ行数と 1 行あたりのバグ数には関連性は認められなかった．

変数使用行率・変数使用行差ともに平均では差があまりないものにバグ数が多いが，加重平均・変数使用行率の最低値・変数使用行差の最大値では，差があるものほど 1 行あたりのバグ数が多い．

メソッドの各属性値とバグ数・メソッド 1 行あたりのバグ数の相関係数を表 2 に示す．

内部メソッド呼び出し数・総メソッド呼び出し数・内部メソッド呼び出し割合の相関係数は，行数やサイクロマチック数の相関係数より絶対値がやや小さく相関も低めである．

平均変数スコープ行数・最低変数使用行率・平均変数使用行差・加重平均変数使用行差・最大変数使用行差の相関係数は，行数とサイクロマチック数の相関係数より絶対値が大きい．

7　考察

7.1　メソッド呼び出し

図 5(b)・図 6(b) より，内部メソッド呼び出し数と総メソッド呼び出し数が多いほど 1 行あたりのバグ数がやや多い．しかし，図 7(b) より内部メソッド呼び出しの割合と 1 行あたりのバグ数には関連性は認められなかった．また，表 2 より内部メソッド呼び出し数・総メソッド呼び出し数・内部メソッド呼び出し割合の相関係数は，行数やサイクロマチック数の相関係数より絶対値がやや小さく，特に内部メソッド呼び出し割合の相関係数は低い．これより，内部メソッド呼び出しの割合が Fault-Prone メソッドの特定に関してあまり有効ではないことがわかる．内部メソッド呼び出し数が多いほど 1 行あたりのバグ数が多く，内部メソッド呼び出しの割合とバグとの関連性がないということは，内部メソッド・外部メソッドに関係なくメソッド呼び出し数が多いほどバグが多いのだと考えられる．

7.2　変数使用

変数使用行率・変数使用行差ともに平均では差があまりないものにバグ数が多いが，加重平均・変数使用行率の最低値・変数使用行差の最大値では，差があるものほど 1 行あたりのバグ数が多い．つまり，変数使用行率が低い変数，変数使用行差が大きい変数が含まれるメソッドに 1 行あたりのバグ数が多いといえる．

また表 2 より，最低変数使用行率・平均変数使用行差・加重平均変数使用行差・最大変数使用行差の相関係数は行数とサイクロマチック数の相関係数より絶対値が大きいので，よりバグとの相関があり，Fault-Prone メソッドの特定に有効であるといえる．

7.2.1　変数使用行率が低いメソッドの実例

変数使用行率が低いメソッドの実例を 1 つ示す．表 3 がメソッド情報であり，図 4(a) がソースコードである．

最低変数使用率となった変数は 137 行目の変数 e である．この変数 e は 137 行目で定義されて 153 行目まで使用可能である．しかし，実際に最後に使用されているのは while 文の中の 139 行目である．while 文でのみ使用する変数を while 文の外で定義しているがため，無駄に変数のスコープが大きくなっている．

そこで，図 4(b) のように while 文を for 文に変更し，変数 e を前提条件として定義するようリファクタリングを行う．変数 e の使用可能な範囲は 137 行目から 141 行目と小さくなり，変数使用行率が高くなった．このような容易なリファクタリングにより変数使用行率を高めることができる．

図 4(a) の 137 行目から 142 行目の while 文は v1.5 のリビジョン 270105 にて新たなメソッドとして定義されており，変数使用行率が低い変数 e がメソッド execute() から除去された．リビジョン

270105 以前でメソッド execute() のバグは 2 つであったが，ともに変数 e に関連のあるバグではなかった.

表 2: Kendall と Spearman の順位相関係数

		Kendall の順位相関係数		Spearman の順位相関係数	
		バグ数	1 行あたりのバグ数	バグ数	1 行あたりのバグ数
内部メソッド呼び出し数		0.232	0.204	0.248	0.223
総メソッド呼び出し数		0.243	0.212	0.276	0.248
内部メソッド呼び出し割合		0.151	0.135	0.163	0.148
平均変数スコープ行数		0.297	0.261	0.325	0.297
変数使用行率	平均	0.236	0.211	0.258	0.237
	加重平均	0.233	0.209	0.255	0.235
	最低値	-0.306	-0.274	-0.328	-0.301
変数使用行差	平均	0.313	0.279	0.335	0.307
	加重平均	0.318	0.283	0.340	0.312
	最大値	0.314	0.277	0.335	0.306
行数		0.281	0.244	0.320	0.292
サイクロマチック数		0.288	0.250	0.310	0.280

表 3: 変数使用行率が低いメソッドの実例情報

クラス名	org.apache.tools.ant.taskdefs.Ant
メソッド名	execute()
最低変数使用率	0.176470
バグ数	8

```
130   public void execute() throws BuildException {
131       if( dir==null) dir=".";
132
133       p1.setBasedir(dir);
134       p1.setUserProperty("basedir" , dir);
135
136       // Override with local-defined properties
137       Enumeration e = properties.elements();
138       while (e.hasMoreElements()) {
139           Property p=(Property) e.nextElement();
140           //System.out.println("Setting " + p.getName()+ " " + p.getValue());
141           p.init();
142       }
143
144       if (antFile == null) antFile = dir + "/build.xml";
145
146       p1.setUserProperty( "ant.file" , antFile );
147       ProjectHelper.configureProject(p1, new File(antFile));
148
149       if (target == null) {
150           target = p1.getDefaultTarget();
151       }
152
153       p1.executeTarget(target);
154   }
```

(a) リファクタリング前

```
130   public void execute() throws BuildException {
131       if( dir==null) dir=".";
132
133       p1.setBasedir(dir);
134       p1.setUserProperty("basedir" , dir);
135
136       // Override with local-defined properties
137       for(Enumeration e = properties.elements();
138       e.hasMoreElements();) {
139           Property p=(Property) e.nextElement();
140           //System.out.println("Setting " + p.getName()+ " " + p.getValue());
141           p.init();
142       }
143
144       if (antFile == null) antFile = dir + "/build.xml";
145
146       p1.setUserProperty( "ant.file" , antFile );
147       ProjectHelper.configureProject(p1, new File(antFile));
148
149       if (target == null) {
150           target = p1.getDefaultTarget();
151       }
152
153       p1.executeTarget(target);
154   }
```

(b) リファクタリング後

図 4: execute() のソースコード

青は変数 e のスコープを表し，赤は実際に使用されている範囲を表す

8 あとがき

本研究では，メソッド呼び出しの種類と変数使用情報に着目して Fault-Prone メソッドを特定する手法を提案し，その手法を実現するためのツールを実装した．メソッド呼び出しは内部メソッド呼び出しと外部メソッド呼び出しの 2 つに分けて考え，内部メソッド呼び出しの割合の有効性を調査した．変数使用情報は，変数のスコープと変数の実際に使用されている範囲の差に着目し変数使用行率・変数使用行差の有効性を調査した．

実装したツールを用いて実験を行った．実験の結果，変数使用情報では変数使用行率が低い変数が含まれるメソッドに 1 行あたりのバグ数が多いことがグラフで確認できた．また，平均変数スコープ行数・最低変数使用行率・平均変数使用行差・加重平均変数使用行差・最大変数使用行差の Kendall と Spearman の順位相関係数は，行数とサイクロマチック数の順位相関係数より絶対値が大きく，よりバグとの相関があると確認した．

更に，簡単なリファクタリングによって変数使用行率を向上させることができることを示した．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究に関して，的確なご助言を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

本研究に多大なるご助言およびご指導を頂きました柿元健特任助教に深く感謝致します.

その他の楠本研究室の皆様のご協力に心より感謝致します.

また，本研究に至るまでに，講義，演習，実験等でお世話になりました情報科学科の諸先生方にこの場を借りて心から御礼申し上げます.

参考文献

[1] T.McCabe. A software complexity measure. *IEEE Transactions on Software Engneering*, Vol. SE-2, pp. 308–320, 1976.

[2] 山田茂, 高橋宗雄. ソフトウェアマネジメントモデル入門ーソフトウェア品質の可視化と評価法. 共立出版, 1993.

[3] S.R.Chidamber and C.F.Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engneering*, Vol. 20, No. 6, pp. 476–493, 1994.

[4] M.H.Halstead. Elements of software science. *Elsevier*, 1977.

[5] V.R.Basili, L.C.Briand, and W.L.Melo. A validation of object oriented metrics as quality indicators. *IEEE Transactions on Software Engineering*, Vol. 22(10), pp. 751–761, 1996.

[6] P.L.Li, J.herbsleb, M.Shaw, and B.Robinson. Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc. *28th Int'l Conf. on Software Engineering*, pp. 413–422, 2006.

[7] E.Arisholm and L.C.Briand. Predicting fault-prone components in a java legacy system. *Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering*, pp. 8–17, 2006.

[8] JIS X 0129(ISO/IEC 9126). ソフトウェア製品の評価-品質特性及びその利用要領. 日本規格協会, 1994.

[9] 門田暁人, 佐藤慎一, 神谷年洋, 松本健一. コードクローンに基づくレガシーソフトウェアの品質の分析. 情報処理学会論文誌, Vol. 44, No. 8, pp. 2178–2187, 2003.

[10] 馬場慎太郎. コードクローン情報を用いた fault-prone モジュールの予測. Master's thesis, 2008.

[11] 早瀬康裕, 松下誠, 楠本真二, 井上克郎, 小林健一, 吉野利明. 保守請負時を対象とした労力見積りのためのメトリクスの提案. 電子情報通信学会技術研究報告. SS, ソフトウェアサイエンス, Vol. 105, No. 491, pp. 61–66, 2005.

[12] 鎌倉潤一, 井口正之, 竹田学, 濱本勇人, 笛田重喜, 多門宏晋, 田中珠貴, 木下直樹, 松尾昭彦, 小林健一. 運用・保守を定量的に「見える化」する方法. システム開発ジャーナル, Vol. 8, pp. 26–45, 2009.

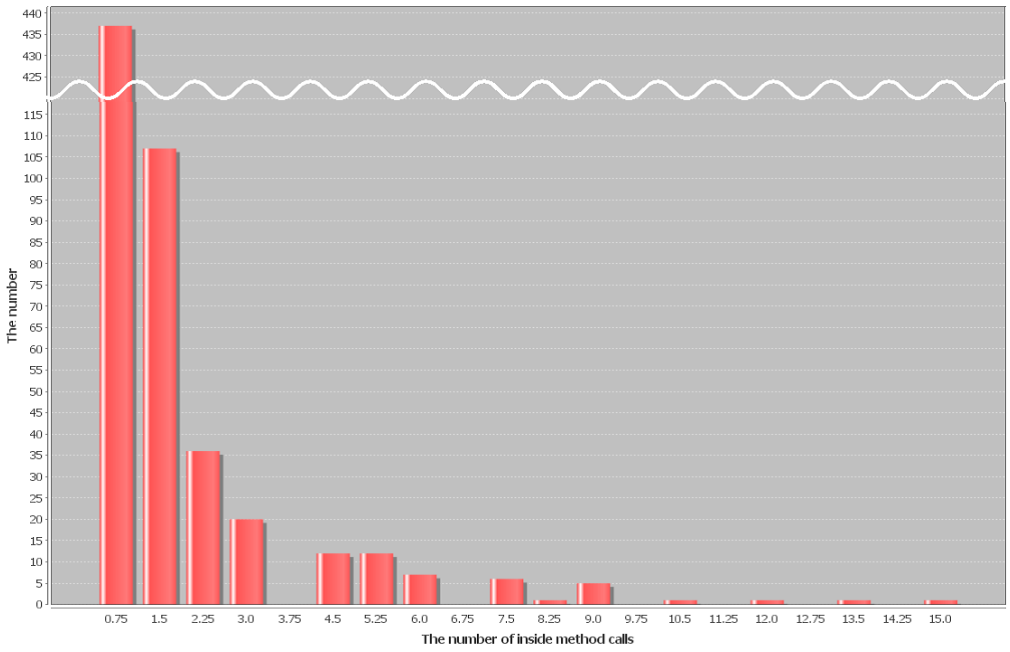
付録

表 4: メソッド呼び出しに関する属性値の最大値と区間の幅

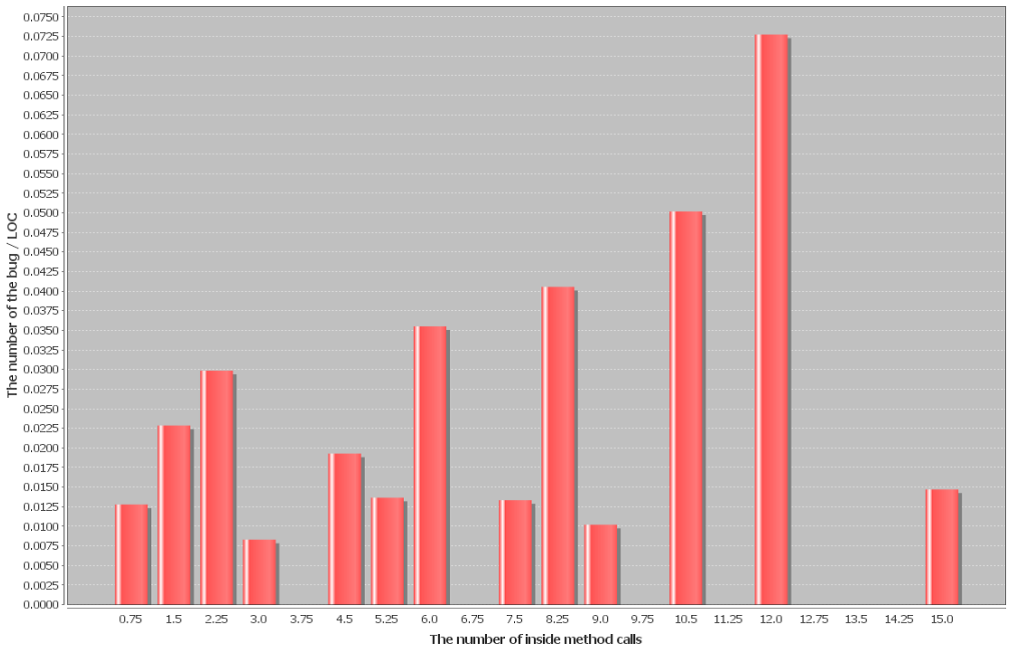
属性値	最大値	区間の幅
内部メソッド呼び出し数	15	0.75
総メソッド呼び出し数	156	7.8
内部メソッド呼び出しの割合	1.0	0.05

表 5: 変数使用に関する属性値の最大値と区間の幅

属性値		最大値	区間の幅
平均変数スコープ行数		118.25	5.9125
変数使用行率	平均	1.0	0.05
	加重平均	1.0	0.05
	最低値	1.0	0.05
変数使用行差	平均	102.5	5.125
	加重平均	131.71458773784354	6.585729386892178
	最大値	224	11.2

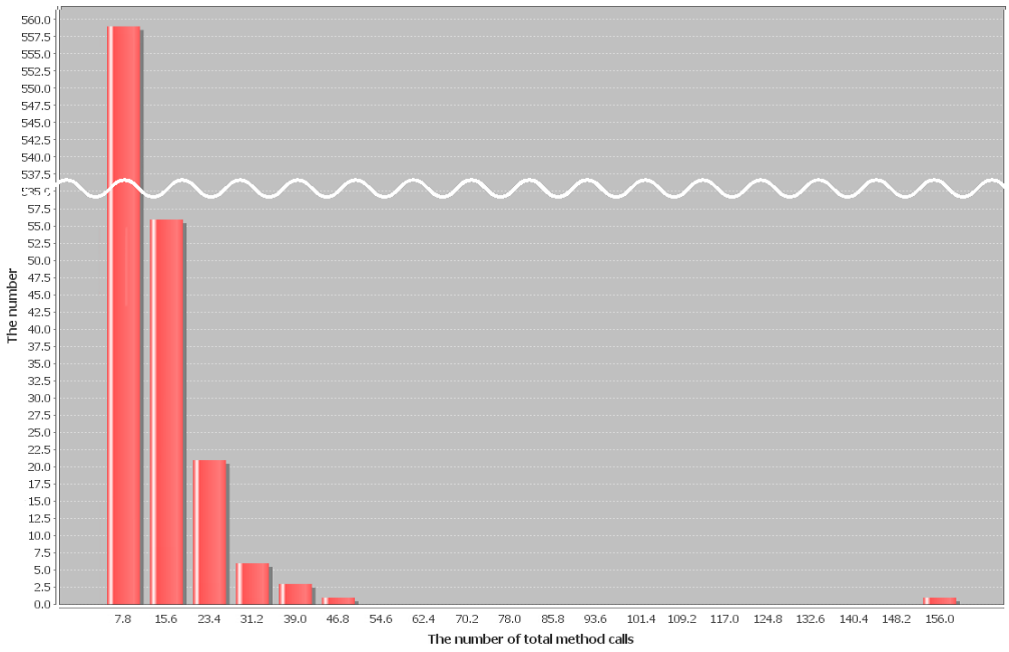


(a) メソッド数

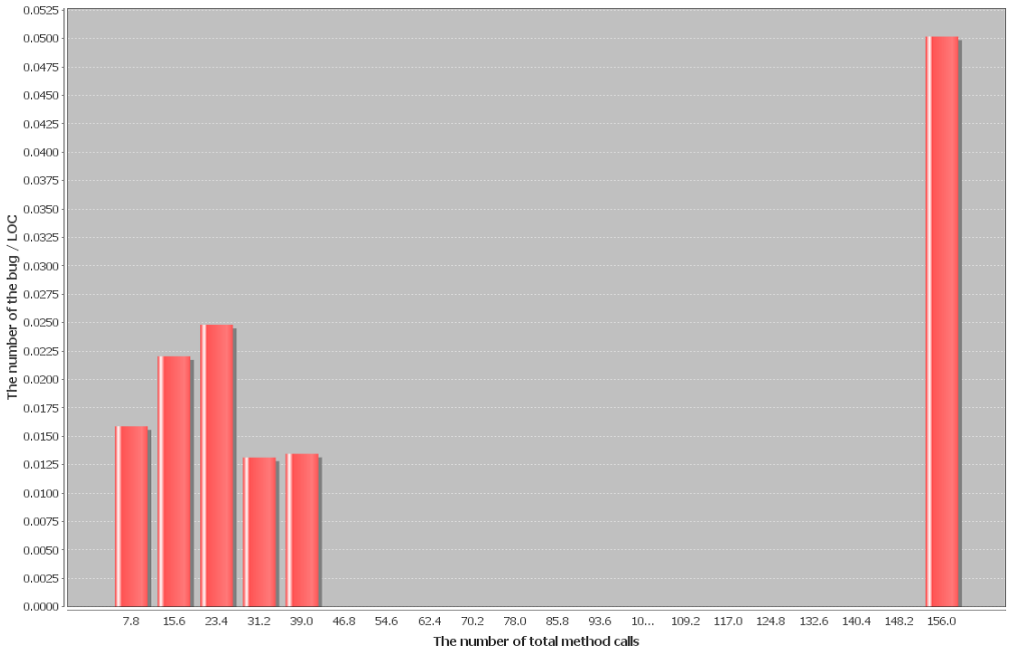


(b) 1 行あたりのバグ数

図 5: 内部メソッド呼び出し数

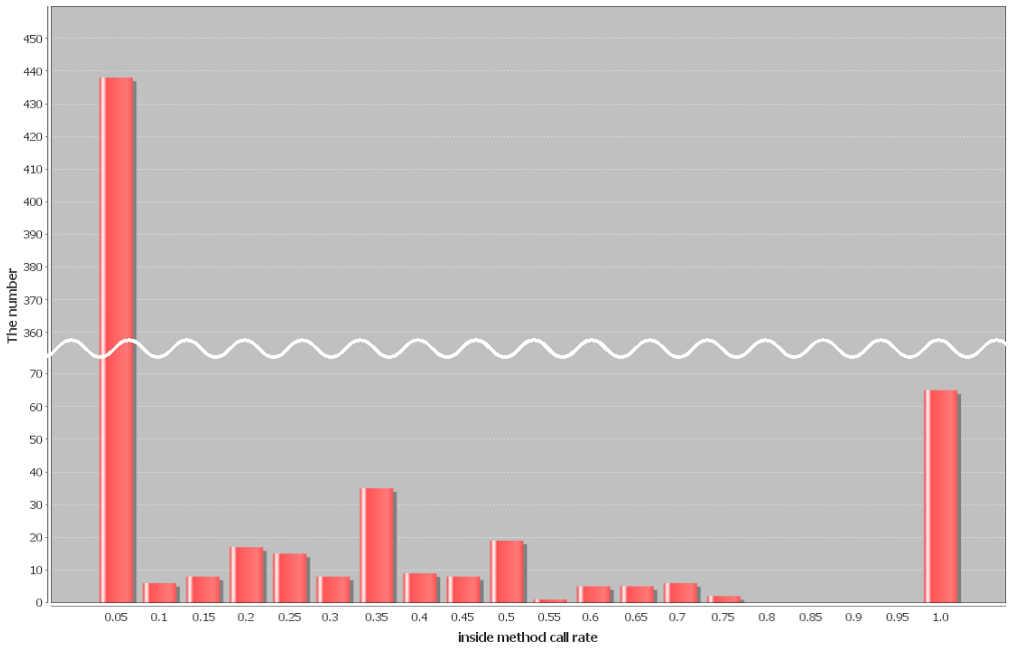


(a) メソッド数

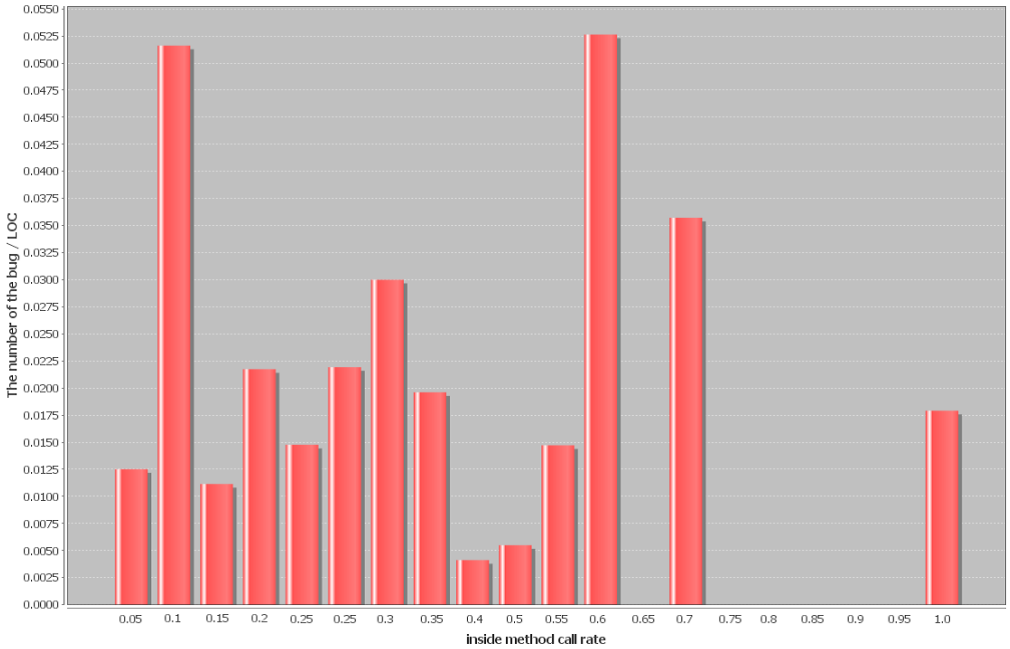


(b) 1 行あたりのバグ数

図 6: 総メソッド呼び出し数

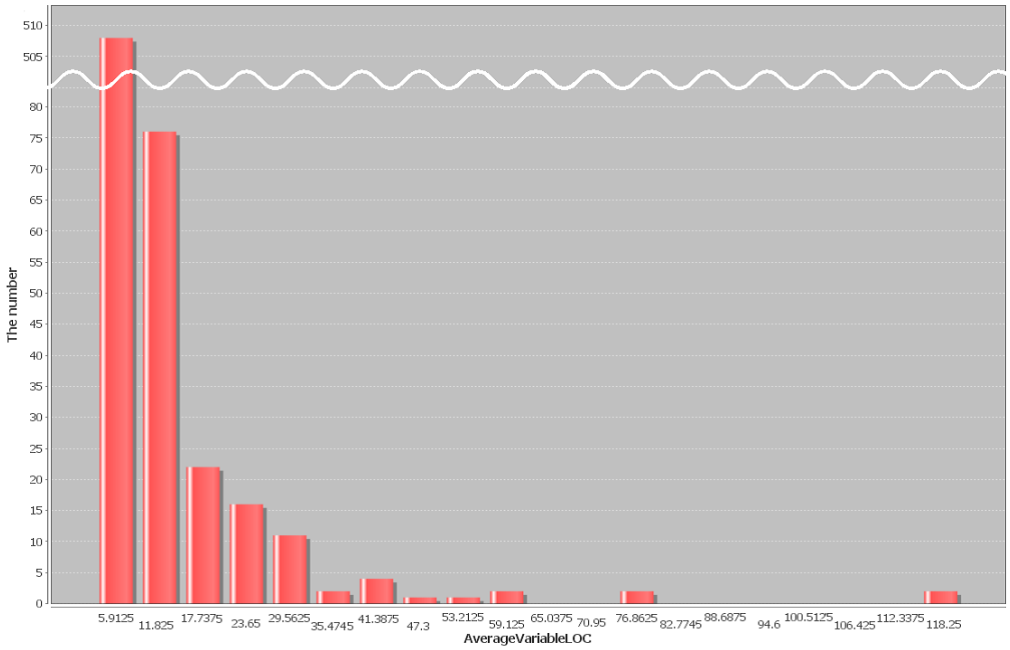


(a) メソッド数

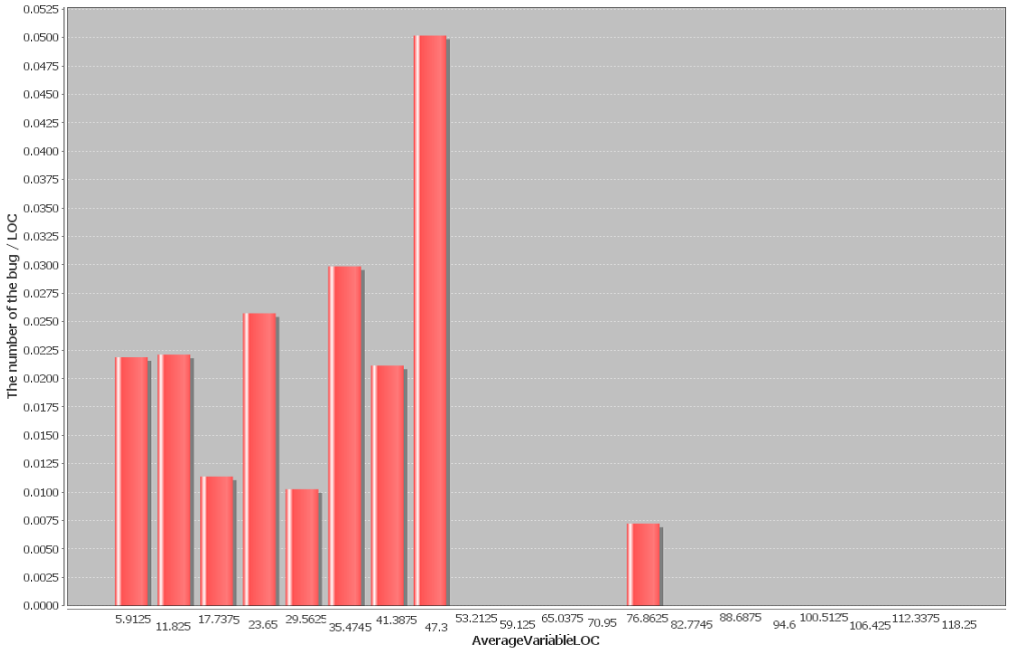


(b) 1 行あたりのバグ数

図 7: 内部メソッド呼び出しの割合

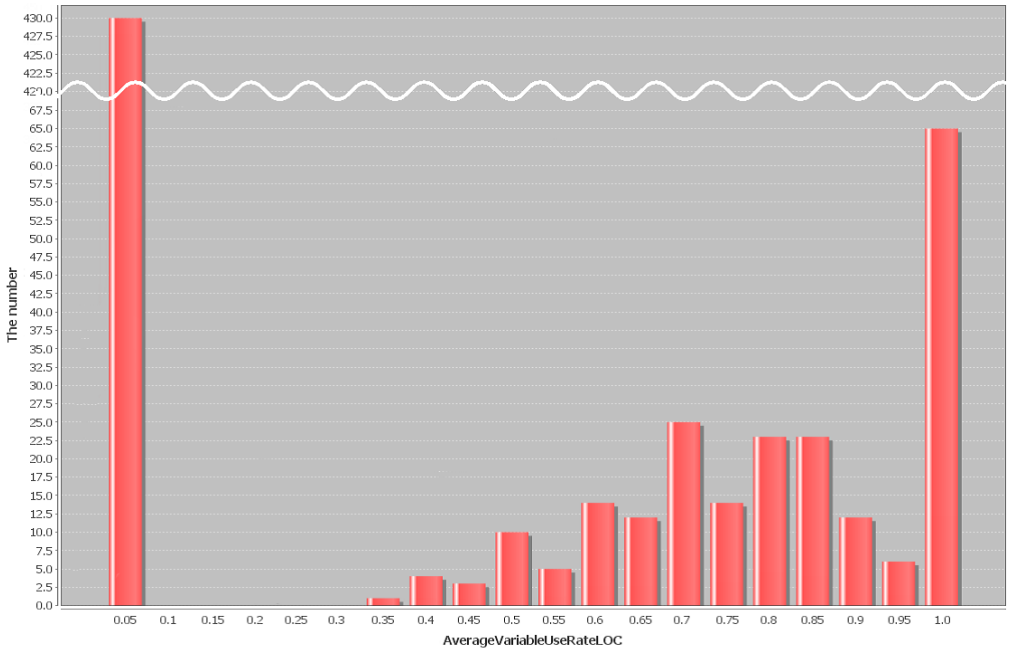


(a) メソッド数

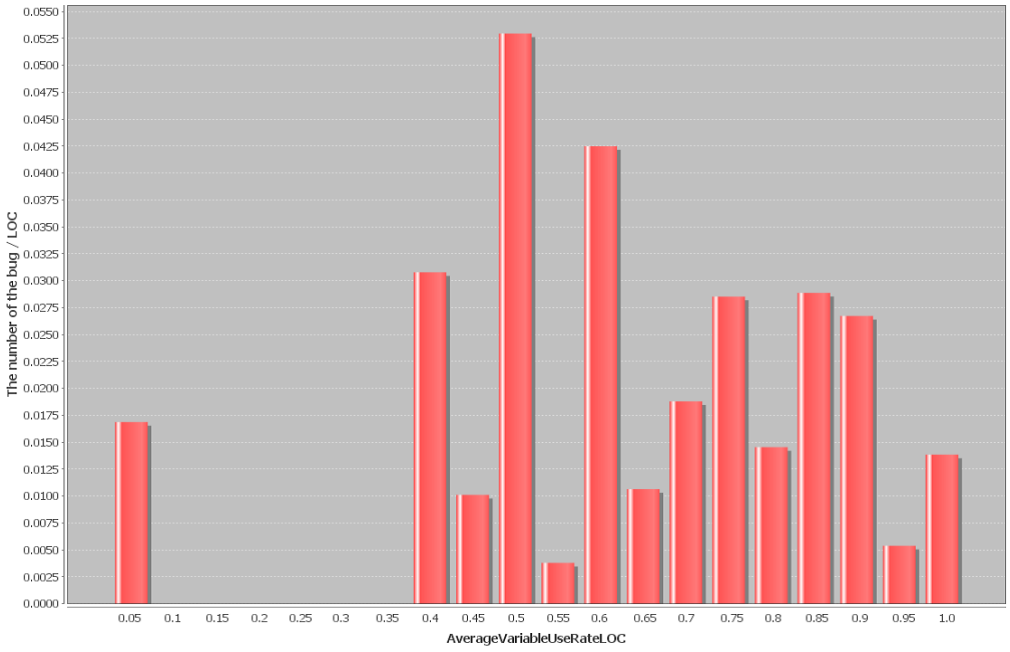


(b) 1 行あたりのバグ数

図 8: 平均変数スコープ行数

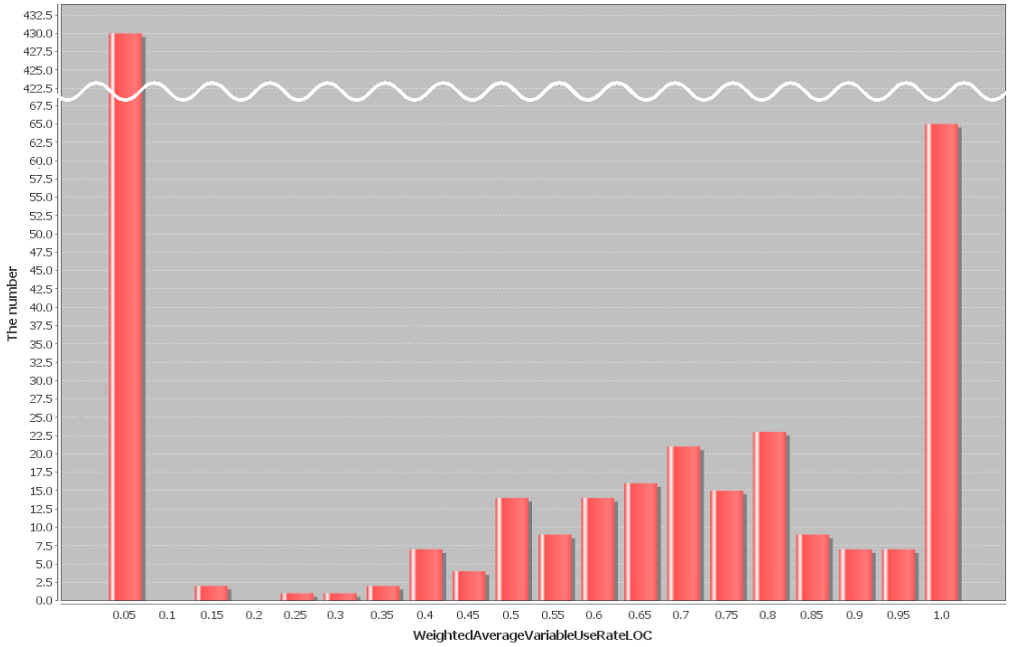


(a) メソッド数

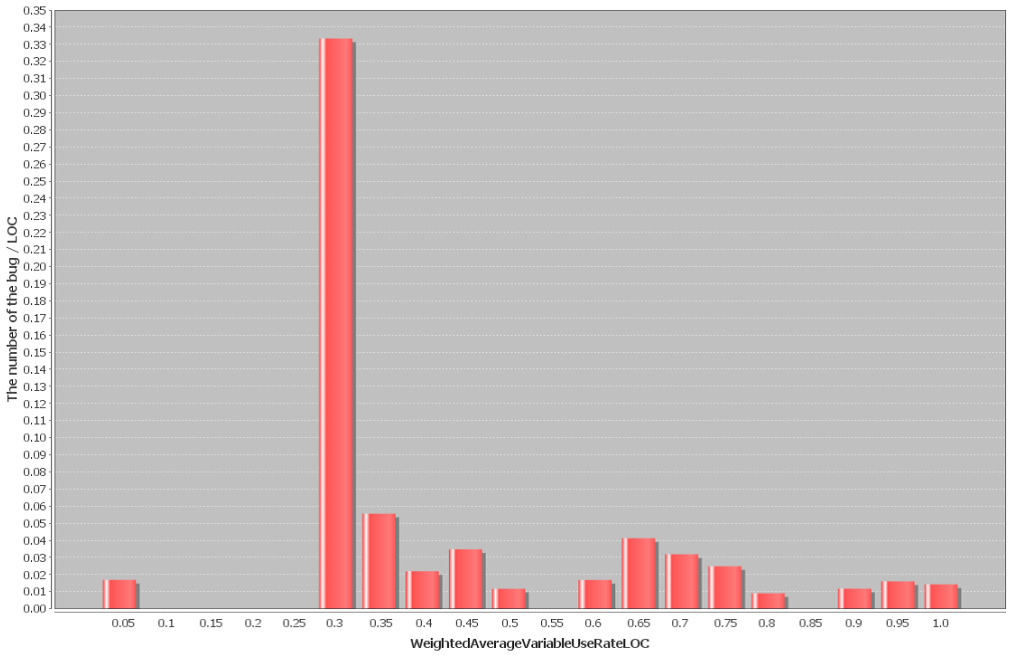


(b) 1 行あたりのバグ数

図 9: 平均変数使用行率

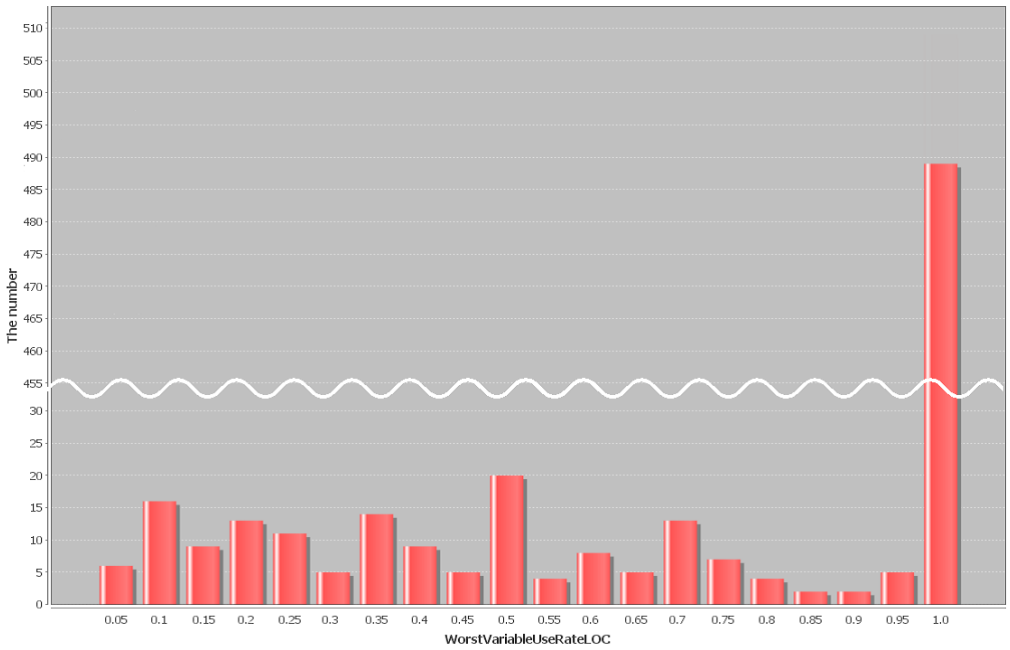


(a) メソッド数

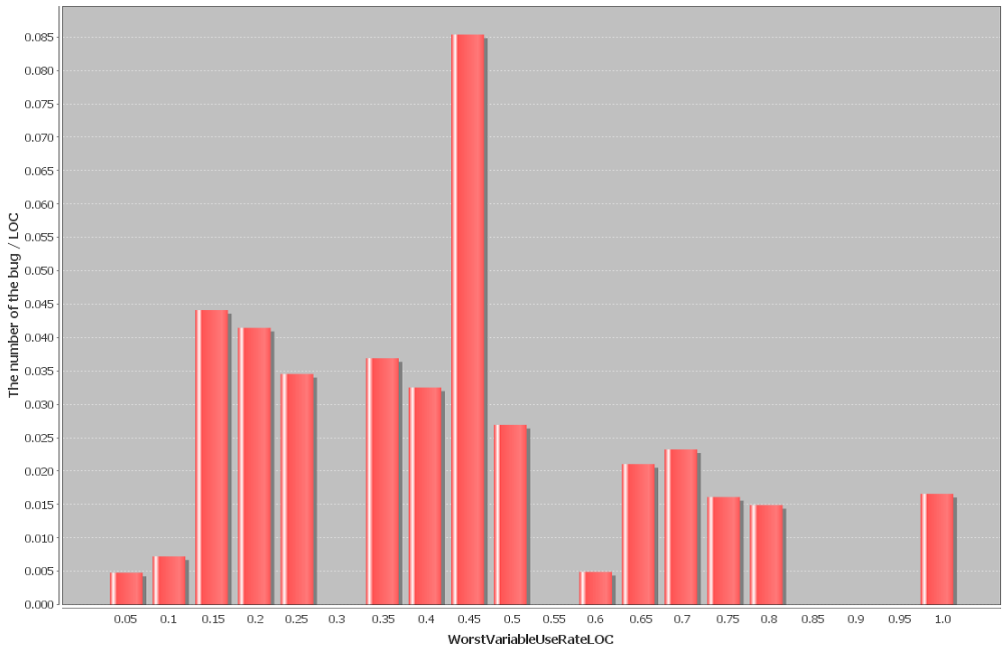


(b) 1 行あたりのバグ数

図 10: 加重平均変数使用行率

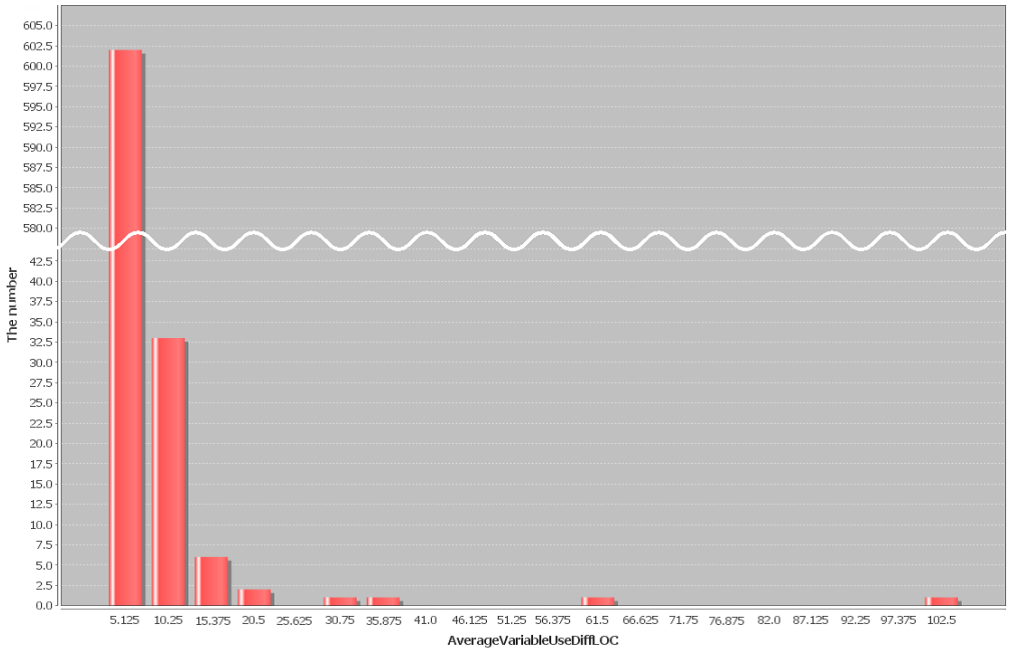


(a) メソッド数

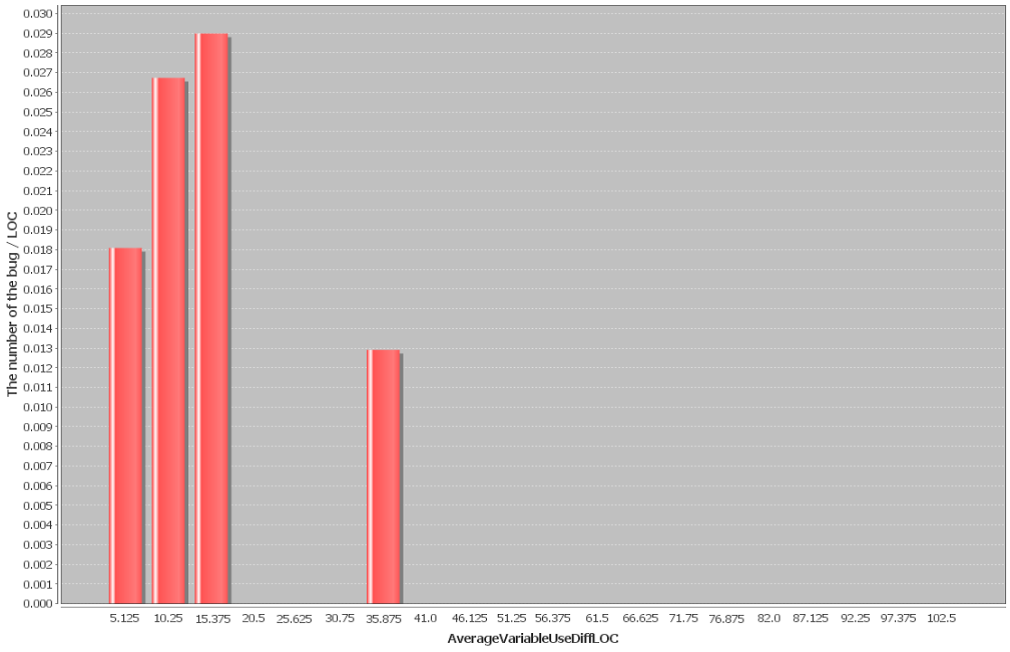


(b) 1 行あたりのバグ数

図 11: 最低変数使用行率

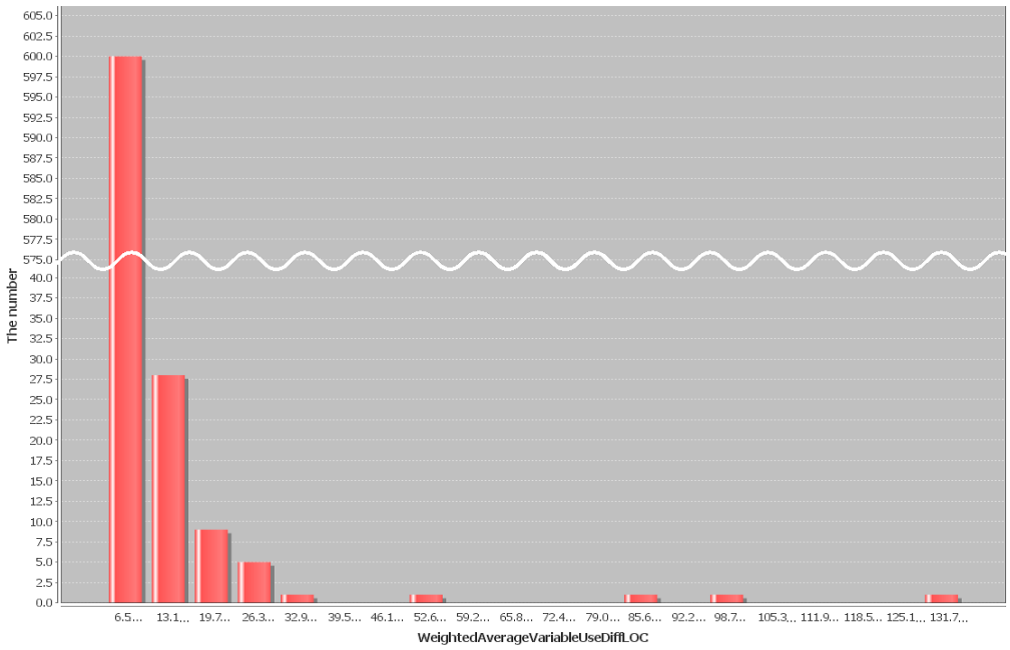


(a) メソッド数

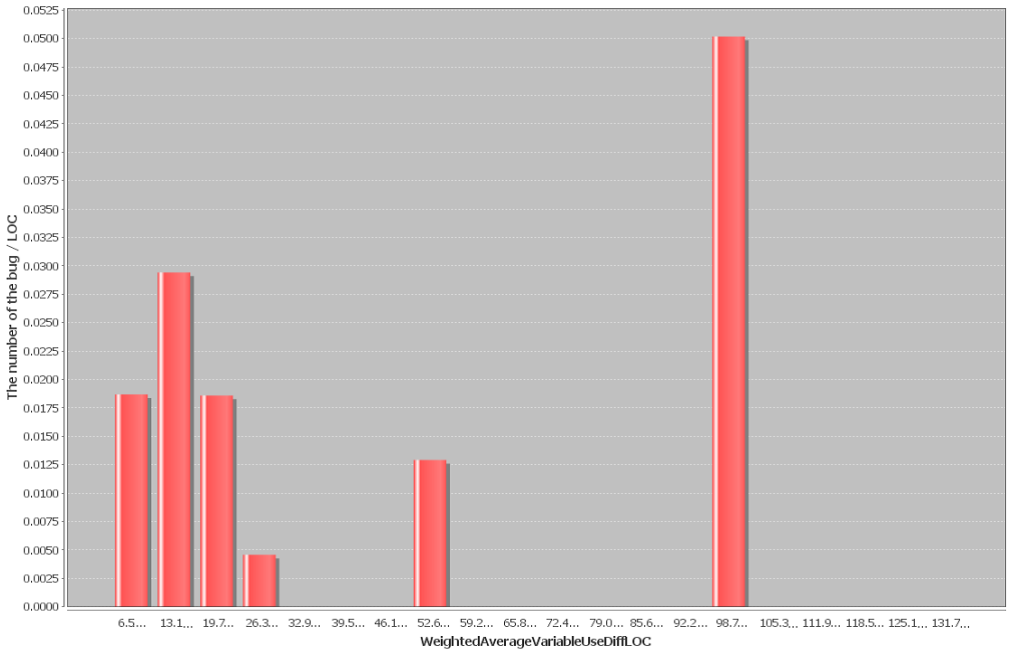


(b) 1 行あたりのバグ数

図 12: 平均変数使用行差

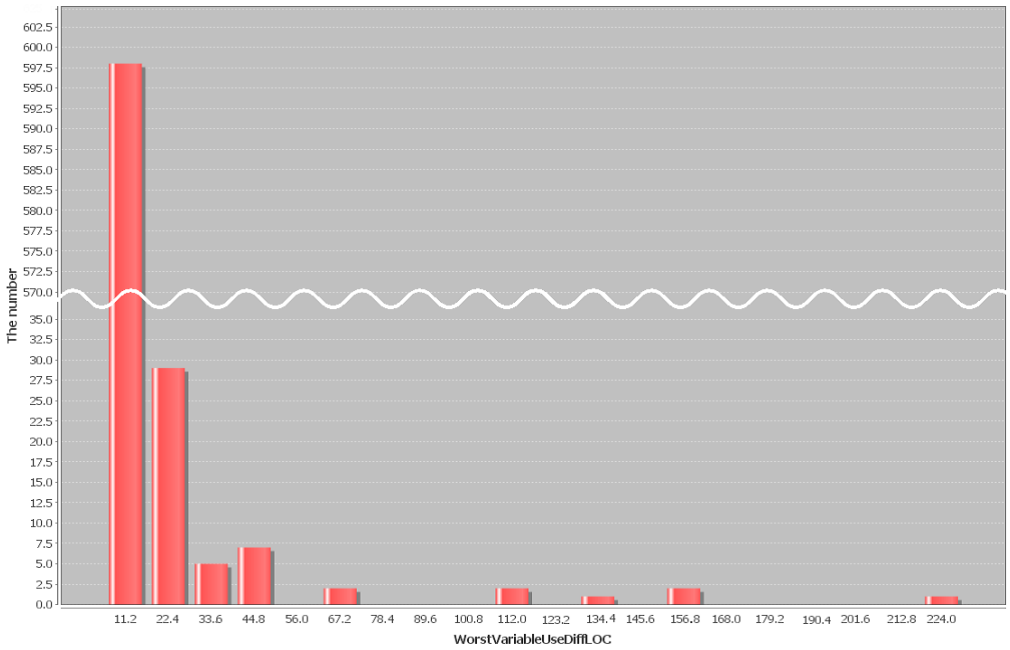


(a) メソッド数

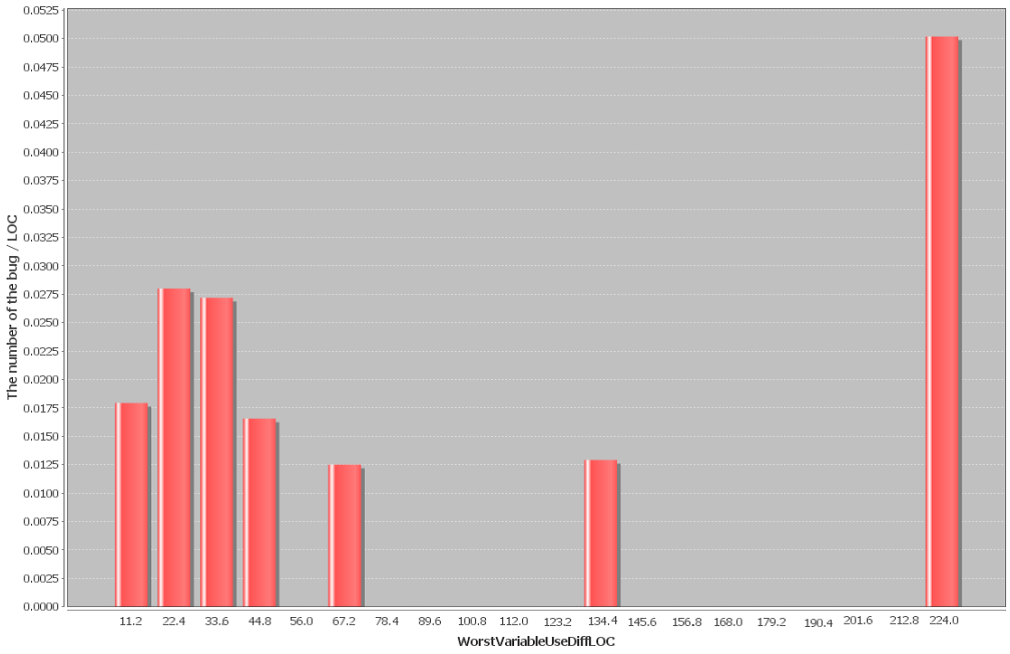


(b) 1 行あたりのバグ数

図 13: 加重平均変数使用行差



(a) メソッド数



(b) 1 行あたりのバグ数

図 14: 最大変数使用行差