卒 業 研 究 報 告 書

題　目　　リポジトリマイニングに基づく
バグ混入者と修正者との関連分析

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　08122055

氏　　名　向井　弘記

平成24年2月15日提出

リポジトリマイニングに基づくバグ混入者と修正者との関連分析

平成 24 年 2 月 15 日　　**08122055　向井　弘記**

概　要

ソフトウェア開発においては，バグが含まれてしまうのは避けられないことである．バグが混入されてしまうと，開発スケジュールの遅延や不正な動作を引き起こしてしまう．より効率的にソフトウェア開発を行い，ソフトウェアの安定した動作を追求するためには，バグが混入してしまうことをできる限り避け，バグの修正をできる限り多く行うことで，バグの総数を減らす必要がある．

バグが混入される原因はさまざまであるが，その多くは開発者の不注意によって混入されることが多い．ソフトウェア開発に携わるのは人間である以上，それは避けられないことであるが，バグの総数をできる限り少なくするためには，バグの混入ができる限り少ない開発者，または，バグの修正をできる限り多く行う開発者がプロジェクトに参加するのが望ましい．従って，よりよいソフトウェア開発を目指すためには，ソフトウェアプロジェクトに携わる開発者に注目し分析することが重要である．

本研究では，リポジトリマイニングを利用し，バグを混入した開発者とバグを修正した開発者を調べることで，開発者相互の関係を抽出することを目的とする．不具合混入モジュールの特定を行うアルゴリズムが適用され，バグ混入情報およびバグ修正情報が埋め込まれた，バージョン管理システム Git で管理されたリポジトリに対して，バグの混入者およびバグの修正者の情報を抽出する．得られた情報から，一つのバグに対する一つのバグの修正という一対一関係を，バグの混入者とバグの修正者という一対一関係にあてはめることによって分析を行う．これを複数の種類のソフトウェアリポジトリに対して行うことで，バグの混入者とバグの修正者との一般的な関係を調査する．

その結果，ソフトウェア開発プロジェクトにおいては，混入されたバグは，多くの場合同一の開発者が修正することが分かった．

目　次

1.　緒言 **1**

2.　研究の目的 **3**

3.　ソフトウェアリポジトリ **5**

3.1　バグデータベース 6

3.2　構成管理システム (Git) 6

4.　実験と分析 **8**

4.1　実験対象プロジェクト 9

4.2　メトリクスの定義 9

4.3　Research Question の定義 12

4.4　グラフの作成 16

4.5　実験結果 16

5.　考察 **25**

6.　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は，バグが含まれてしまうのは避けられないことである．バグが混入されてしまうと，開発スケジュールの遅延や不正な動作を引き起こしてしまう．より効率的にソフトウェア開発を行い，ソフトウェアの安定した動作を追求するためには，バグが混入してしまうことをできる限り避け，バグの修正をできる限り多く行うことで，バグの総数を減らす必要がある．

バグが混入される原因はさまざまであるが，その多くは開発者の不注意によって混入されることが多い．ソフトウェア開発に携わるのは人間である以上，それは避けられないことであるが，バグの総数をできる限り少なくするためには，バグの混入ができる限り少ない開発者，または，バグの修正をできる限り多く行う開発者がプロジェクトに参加するのが望ましい．従って，よりよいソフトウェア開発を目指すためには，ソフトウェア開発プロジェクトに携わる開発者に注目し分析することが重要である．

本研究では，リポジトリマイニングを利用し，バグを混入した開発者とバグを修正した開発者を調べることで，開発者相互の関係を抽出することを目的とする．具体的には，不具合混入モジュールの特定を行うアルゴリズムである SZZ アルゴリズム [1] が適用され，バグ混入情報およびバグ修正情報が埋め込まれた，バージョン管理システム Git で管理されたリポジトリに対して，バグの混入者およびバグの修正者の情報を抽出する．得られた情報から，一つのバグに対する一つのバグの修正という一対一関係を，バグの混入者とバグの修正者という一対一関係にあてはめることによって分析を行う [2][3][4]．これを複数の種類のソフトウェアリポジトリに対して行うことで，バグの混入者とバグの修正者との一般的な関係を調査する．

その結果，ソフトウェア開発プロジェクトにおいては，混入されたバグは，多くの場合同一の開発者が修正することが分かった．

本報告の以降の構成は次の通りである．第 2 章では，本研究を行う目的を記述し，Reserch Question について説明する．第 3 章では，実験に使用するソフトウェアリポジトリを管理するバージョン管理システムについての基本的な知識を説明する．第 4 章では，ソフトウェアリポジトリを解析を行い，Reserch Question に答えるための実験の手順と，メトリクスの定義を行う．さらに，実験に対する結果を述べる．第

5 章では，実験に対する考察を行う．第 6 章では，本研究のまとめと今後の課題を述べる．

2. 研究の目的

以下，Research Question をいくつか定義することによって，バグ混入者と修正者の関係や，ソフトウェア開発プロジェクトに有益な開発者に関する情報を調査する．

RQ1: 最もバグを混入しやすい開発者は誰か．最もバグを修正しやすい開発者は誰か．

ソフトウェアリポジトリから，最もバグを混入しやすい開発者および，最もバグを修正しやすい開発者を特定したい．

リポジトリに対して開発者は様々な形でコミットするが，大まかに，次の三種類の開発者に分類できる．

- どちらかといえば，バグを混入する方が多い開発者 = 害のある開発者
- どちらかといえば，バグを修正する方が多い開発者 = 有益な開発者
- バグを混入する量も修正する量もほぼ等しい開発者 = 有益な開発者

ソフトウェア開発において，バグの総数を最小限にとどめるためには，バグを混入しやすい開発者の数を出来る限り減らし，バグを修正しやすい開発者の数を出来る限り増やせばよい．

最もバグを混入しやすい開発者および，最もバグを修正しやすい開発者を特定し，その情報を与えることによって，今後ソフトウェアプロジェクトへ多大な被害を与えうる開発者や，有益なソフトウェアの改善を行う開発者を特定できる．

またソフトウェアリポジトリの規模の大きさによって，それらがどう異なるかを調べる．

RQ2: 開発者が自分で混入したバグを自分で修正する割合はどの程度か．

開発者が自分で混入したバグを自分で修正するというコミットのパターンが，ソフトウェアリポジトリにコミットした全ての開発者のコミットに対して，どの程度の割合があるかを調べる．

これによって，混入されたバグが誰によって直されやすいか，今後バグが発見されたとき，発見されたバグをどの開発者が最も適切に修正することができるかを特定できる．

またソフトウェアリポジトリの規模の大きさによって，それがどう異なるかを調

べる.

RQ3: 最も頻繁に発生したバグ混入と修正のパターンは何か.

特定の開発者が混入したバグを特定の開発者が修正するパターンのうち，最も数が多いパターンは，どの開発者によって混入されたバグであり，どの開発者によって修正されたものであるか調べる.

ソフトウェアリポジトリ内で開発者同士が，最も関連が深いのか，また，最も影響を与えているのかを調べる.

これによって，今後行われるソフトウェア開発プロジェクトにおいて，最も頻繁に活躍する開発者同士の関係を見ることができる.

また，ソフトウェアリポジトリの規模の大きさによって，それがどう異なるかを調べる.

3. ソフトウェアリポジトリ

ソフトウェアリポジトリとは，ソフトウェアプロジェクトの，ソースコードを蓄積することができる巨大なデータベースである．本研究ではソフトウェアリポジトリを調査するために，バージョン管理システムを使用する．

バージョン管理システムとは，ファイルの開発変更履歴を記録するシステムである．ファイルを変更する度に，変更内容，変更した日付，変更した人などの情報をリポジトリと呼ばれるデータベースに記録する．これによって，過去の変更内容を閲覧したり，必要ならばファイルを過去の状態に戻すことができる．

また，ディレクトリ構造を1つの単位としてリポジトリを作成して管理を行うことができるので，大規模なソフトウェア開発プロジェクトに存在するような，ディレクトリの異なる複数のファイルにまたがる，論理的には同一の変更も記録することができる．

バージョン管理システムには集中型と分散型の2種類がある [5]．以下に特徴をまとめる．

- 集中型のシステム
  - 全てのユーザがソフトウェアプロジェクトの内容を変更したり閲覧することができるリポジトリが1つ存在する．
  - 1つのリポジトリが置いてあるサーバへアクセスし，必要なファイルをチェックアウトして，ファイルの編集を行ったら，編集した結果をサーバのリポジトリへコミットする．
  - リポジトリへのコミットはネットワーク環境が必要になる．
  - コミットによる変更の記録は即時にリポジトリへ反映されるので，他のユーザはリアルタイムで変更を見ることができる．
  - システムを低コストで運用できる．
- 分散型のシステム
  - ユーザごとにリポジトリのコピーをローカルに置くので，ファイルの変更やリポジトリへのコミットが，全てローカル内で完結することができる．

– リポジトリのコピーをサーバへ置くので，他のユーザと変更の履歴を共有することができる．
– 変更の履歴を公開するときにサーバへアクセスするだけなので，コミットをするときはネットワーク環境が必要ではない．
– プロジェクトに参加する他のユーザに影響を与えることなく，大規模に変更を加えることができる．
– サーバ上のリポジトリに障害が起こったときでも，致命的な問題には至らない．
– システムの管理が複雑になってしまう．

3.1 バグデータベース

バグデータベースには，ソフトウェアプロジェクトのバグ情報が記録されており，バグがどのように修正されていくかを追跡することができる．ここに，ソフトウェアプロジェクトに携わる開発者の情報や，変更履歴，バージョンなどが管理されている．

3.2 構成管理システム (Git)

Git[6][7] は Linus Torvalds によって開発された，オープンソースのバージョン管理システムであり，分散型を採用している．Linux カーネル開発のソース管理にも使われている．

以下に Git の主な特徴を列挙する

- コミット (commit)，ツリー (tree)，ブロブ (blob)，タグ (tag) の 4 つの異なるオブジェクトファイルを使ってファイルを管理している．
- commit，tree，blob，tag の 4 つのオブジェクトファイルのファイル名にもそのオブジェクトの値から計算されたハッシュ値が用いられ，ファイルの内容は全て圧縮されている．
- コマンドの出力フォーマットをコマンドラインオプションで指定することができる．

- ファイルの移動やファイル名の変更の検出を自動的に行うことができる．
- 古いバージョンのファイルをいくつか圧縮して１つのファイルとして管理する．これによって，リポジトリの容量を小さくすることができる．

4. 実験と分析

実験対象とする各 Git リポジトリには，予め SZZ アルゴリズム [1] を適用してあり，FIX 情報と BUG 情報が tag として記録されている．

シェルスクリプトによるプログラムを使用し，Git リポジトリから以下のタグがついた履歴を抽出し，履歴に対する開発者の情報を得た．

- FIX-(issueid)-(fixid)
- BUG-(issueid)-(fixid)-(bugid)
  - (issueid)：バグトラッキングシステムにおける問題番号
  - (fixid)：バグ修正の通し番号
  - (bugid)：バグの通し番号

ここで，issueid と fixid の通し番号が等しい BUG-(issueid)-(fixid)-(bugid) に対する修正が FIX-(issueid)-(fixid) であるとする．

以下に必要なデータを Git リポジトリから抽出する手順を示す．

1. Git リポジトリから FIX のタグがついた履歴を抽出する．Git コマンド
   `git tag | grep FIX`
   で FIX のタグがついた履歴を result_fix.txt へ出力した．
2. Git リポジトリから BUG のタグがついた履歴を抽出する．Git コマンド
   `git tag | grep BUG`
   で BUG のタグがついた履歴を result_bug.txt へ出力した．
3. result_fix.txt へ出力して得られた全ての FIX-(issueid)-(fixid) に対して，Git コマンド
   `git show FIX-(issueid)-(fixid) | grep Author > result.txt`
   を適用して，全ての FIX-(issueid)-(fixid) の開発者の情報を result.txt へ出力した．
4. result_bug.txt へ出力して得られた全ての BUG-(issueid)-(fixid)-(bugid) に対して，Git コマンド
   `git show BUG-(issueid)-(fixid)-(bugid)| grep Author >> result.txt`

を適用して，全ての BUG-(issueid)-(fixid)-(bugid) の開発者の情報を result.txt へ出力した.

5. result.txt の内容を，(issueid) の値について昇順でソートした．これによって，(issueid) の値が等しい BUG-(issueid)-(fixid)-(bugid) と FIX-(issueid)-(fixid) の組み合わせが分かる.
6. 開発者ごとに，BUG-(issueid)-(fixid)-(bugid) または FIX-(issueid)-(fixid) が何回出現したかを数えた.

4.1 実験対象プロジェクト

実験の対象としたプロジェクトは以下の 6 つである.

- org.eclipse.ecf
- org.eclipse.emf.compare
- org.eclipse.webtools.incubator
- org.eclipse.xpand
- org.eclipse.birt
- hadoop-common

4.2 メトリクスの定義

Research Question に答えるために，いくつかのメトリクスを定義する.

- $Author$

  実験の対象となる Git リポジトリにコミットした開発者を表す.
- $NumAuthor$

  実験の対象となる Git リポジトリにコミットした開発者の総数を表す.
- $BUGcommit[Author]$

  開発者 Author の BUG コミット数を表す．Git リポジトリに対して開発者 $Author$ がバグを混入した回数である．具体的には，抽出できた全ての BUG-(issueid)-(fixid)-(bugid) の $Author$ のうち，ある開発者 $Author$ が出現した回数である.

- $FIXcommit[Author]$
  開発者 Author の FIX コミット数を表す．Git リポジトリに対して開発者 $Author$ がバグを修正した回数である．具体的には，通し番号 `(fixid)` および `(bugid)` が等しい BUG コミット数に対応した全ての `FIX-(issueid)-(fixid)` の $Author$ のうち，ある開発者 $Author$ が出現した回数である．

例えば，表 4.1 のような場合は

$$BUGcommit[rsuen] = 13$$

$$FIXcommit[rsuen] = 3$$

$$BUGcommit[slewis] = 9$$

$$FIXcommit[slewis] = 19$$

である．

$FIXcommit[Author]$ を単純に，抽出できた全ての `FIX-(issueid)-(fixid)` の $Author$ のうち，ある開発者 $Author$ が出現した回数としなかった理由は，一つの修正は複数のバグに対して行われる場合があり，このとき，開発者 $Author$ がどの程度バグの修正に貢献したかを定量化することを考えると，バグコミット数に応じて修正の数を定めた方が良いからである．

例えば，1 個の BUG コミット数に対する 1 個の修正 A と，10000 個の BUG コミット数に対する 1 個の修正 B とを比較することを考える．いずれの修正も，Git リポジトリのタグ付け上では 1 個の修正にあたるが，実際は，修正 A は 1 個分のバグに対して，修正 B は 10000 個分のバグに対して行われる．つまり修正 A は 1 個分のバグに対する 1 個分の修正であり，修正 B は 10000 個分のバグに対する 10000 個分の修正であると見るのが妥当である．このとき，修正 B のほうがバグの修正に大きく貢献したと言える．

- $commit[AuthorX \rightarrow AuthorY]$
  - 開発者 $AuthorX$ の BUG コミット数 $BUGcommit[AuthorX]$ のうち，開発者 Y が修正した $BUGcommit[AuthorX]$ の回数を表す．もしくは，開発者

表 4.1 Git タグと Author の出力例

Git tag	Author
FIX-107173-21	rsuen
BUG-107173-21-380	rsuen
BUG-107173-21-379	slewis
FIX-107173-22	slewis
BUG-107173-22-1418	slewis
FIX-112545-18	rsuen
BUG-112545-18-1252	slewis
FIX-112597-19	slewis
BUG-112597-19-1975	slewis
BUG-112597-19-1974	slewis
BUG-112597-19-1973	rsuen
BUG-112597-19-1972	rsuen
BUG-112597-19-1971	rsuen
BUG-112597-19-1970	rsuen
BUG-112597-19-1969	rsuen
BUG-112597-19-1968	rsuen
BUG-112597-19-1967	rsuen
BUG-112597-19-1966	rsuen
BUG-112597-19-1965	rsuen
BUG-112597-19-1964	rsuen
FIX-125572-30	slewis
BUG-125572-30-377	slewis
BUG-125572-30-376	slewis
BUG-125572-30-375	slewis
BUG-125572-30-374	slewis
BUG-125572-30-373	slewis
BUG-125572-30-372	slewis

$AuthorY$ の FIX コミット数 $FIXcommit[AuthorY]$ の回数のうち，開発者 X がバグを混入した $FIXcommit[AuthorY]$ の回数である．

- $totalcommit[Author]$
  - 開発者 $Author$ による総コミット数を表す．Git リポジトリに対して開発者 $Author$ がバグを混入または修正した回数の合計である．すなわち，式 (4.1) である．

$$totalcommit[Author] = BUGcommit[Author] + FIXcommit[Author] \tag{4.1}$$

4.3 Research Question の定義

以上のメトリクスから，各 Research Question に関して，詳細なメトリクスを定める．

RQ1: 最もバグを混入しやすい開発者は誰か．最もバグを修正しやすい開発者は誰か．

開発者を以下のような三種類に定義する．

- どちらかといえば，バグを混入する方が多い開発者 = バグコミッター
- どちらかといえば，バグを修正する方が多い開発者 = 修正コミッター
- バグを混入する量も修正する量もほぼ等しい開発者 = 中立コミッター

実験の対象とするソフトウェアリポジトリに関わる全ての開発者の中で，最もバグを混入しやすい開発者は誰か，あるいは最もバグを修正しやすい開発者は誰かを特定するために，上記三種類の開発者に分類するための指標として，コミット傾向値 $p[Author]$ を定義する．

以下，コミット傾向値 $p[Author]$ の定義について説明する．

コミット傾向値を単純に BUG コミット数と FIX コミット数との比率を使うとする．つまり，$p1[Author]$ を式 (4.2) で定義する．

$$p1[Author] = \frac{BUGcommit[Author]}{FIXcommit[Author]} \tag{4.2}$$

こうすれば，

$$p1[Author] > 1 \text{ のとき} \cdots Author \text{ はバグコミッター}$$

$$p1[Author] < 1 \text{ のとき} \cdots Author \text{ は修正コミッター}$$

$$p1[Author] = 1 \text{ のとき} \cdots Author \text{ は中立コミッター}$$

と判断できる．さらに，コミット傾向値が大きいほどバグコミッターの傾向が強く，コミット傾向値が 0 に近いほど修正コミッターの傾向が強いといえる．

しかしこのままだと，例えば

- 開発者 X : $BUGcommit[X] = 3, FIXcommit[X] = 1$
- 開発者 Y : $BUGcommit[Y] = 300, FIXcommit[Y] = 100$

のとき，コミット傾向値は $p1[X] = p1[Y] = 3$ であるために，開発者 X と開発者 Y は同程度にバグコミッターとしての傾向が強いと判断できてしまう．

直感的に，開発者 X と開発者 Y は同程度にバグコミッターであるとは判断しがたい．

さらに，FIX コミット数が 0 値のときには，0 除算のために $p1[Author]$ の値が計算できないという問題もある．

バグコミッターか修正コミッターか判断するためには，BUG コミット数と FIX コミット数の値の大きさを比較すれば明らかなので，バグコミッターと修正コミッターと分けて区別し，バグコミッターならばどの程度その傾向が強いか，修正コミッターならばどの程度その傾向が強いかを考える．

そこで，コミット傾向値を総コミット数のうち，多い方のコミットの割合を使うとする．つまり，$p2[Author]$ を式 (4.3) と定義する．

$$p2[Author] = \begin{cases} \frac{BUGcommit[Author]}{totalcommit[Author]} & (BUGcommit[Author] > FIXcommit[Author]) \\ \frac{FIXcommit[Author]}{totalcommit[Author]} & (BUGcommit[Author] < FIXcommit[Author]) \\ 0.5 & (BUGcommit[Author] = FIXcommit[Author]) \end{cases} \tag{4.3}$$

こうすれば，

$$BUGcommit[Author] > FIXcommit[Author] \text{ のとき} \cdots Author \text{ はバグコミッター}$$

$$BUGcommit[Author] < FIXcommit[Author] \text{ のとき} \cdots Author \text{ は修正コミッター}$$

$$BUGcommit[Author] = FIXcommit[Author] \text{ のとき} \cdots Author \text{ は中立コミッター}$$

と判断できる．

このとき，$p2[Author]$ の値が 1 に近いほどバグコミッターもしくは修正コミッターとしての傾向が強く，$p2[Author]$ の値が 0.5 に近いほどその傾向が弱いといえる．

しかしこのままだと，例えば

- 開発者 X : $BUGcommit[X] = 1000, FIXcommit[X] = 800$
- 開発者 Y : $BUGcommit[Y] = 1, FIXcommit[Y] = 0$

のとき，$p[Author]$ の値は，$p[X] = 0.5556, p[Y] = 1$ となり，開発者 X は傾向が弱いバグコミッターであり，開発者 Y はバグコミッターとしての傾向が強いと判断できてしまう．

これも直感に反してしまう．むしろ開発者 Y より開発者 X のほうが，バグコミッターとしての傾向が強いと感じる．

ここまで，$p[Author]$ を $p1[Author]$ および $p2[Author]$ の二通りに定義したが，いずれも直感と反してしまう．何故そう感じるかというと，両者はコミット数が大きく異なるからである．

直感と一致させるために，BUG コミット数と FIX コミット数の差を，上記にて定義した $p2[Author]$ に重み付けする．すなわち $p3[Author]$ を式 (4.4) と定義する．

$$p3[Author] = \begin{cases} \frac{BUGcommit[Author]}{totalcommit[Author]} \times (BUGcommit[Author] - FIXcommit[Author]) \\ (BUGcommit[Author] > FIXcommit[Author] \text{ のとき}) \\ \frac{FIXcommit[Author]}{totalcommit[Author]} \times (FIXcommit[Author] - BUGcommit[Author]) \\ (BUGcommit[Author] < FIXcommit[Author] \text{ のとき}) \\ 0 \\ (BUGcommit[Author] = FIXcommit[Author] \text{ のとき}) \end{cases} \quad (4.4)$$

こうすれば，

$BUGcommit[Author] > FIXcommit[Author]$ のとき…$Author$ はバグコミッター

$BUGcommit[Author] < FIXcommit[Author]$ のとき…$Author$ は修正コミッター

$BUGcommit[Author] = FIXcommit[Author]$ のとき…$Author$ は中立コミッター

と判断できる．さらに，$p3[Author]$ の値が大きいほどバグコミッターもしくは修正コミッターとしての傾向が強く，$p3[Author]$ の値が 0 に近いほどその傾向が弱いといえる．

このとき

- 開発者 X : $BUGcommit[X] = 1000, FIXcommit[X] = 800$
- 開発者 Y : $BUGcommit[Y] = 1, FIXcommit[Y] = 0$

であるので，$p3[Author]$ の値は $p[X] = 111.12, p[Y] = 1$ となり，ある程度直感と一致させることが出来る．

これより，コミット傾向値 $p[Author]$ を $p3[Author]$ で決定し，以下に RQ を定める．

- 最もバグを混入しやすい開発者 $Author$ は，バグコミッターの中で最大の $p[Author]$ の値を持つ $Author$ である．
- 最もバグを修正しやすい開発者 $Author$ は，修正コミッターの中で最大の $p[Author]$ の値を持つ $Author$ である．

これらメトリクスに関して，6 つの異なるリポジトリでどのように変化するかを調べた．

RQ2: 開発者が自分で混入したバグを自分で修正する割合はどの程度か．

全ての開発者の総コミット数の合計のうち，全ての開発者の自分の BUG コミットを自分で FIX コミットした回数の合計の割合を定義する．

一つのソフトウェアプロジェクト内で，開発者同士の関係は複雑である．とある開発者 $Author$ が混入したバグを自分自身が修正する以外は，他のどの開発者がどれだけ修正するかは様々である．従って，メトリクスの対象を開発者ごとに定めるのではなく，全ての開発者に関して定めた．

開発者が自分で混入したバグを自分で修正する割合 r は

$$r = \frac{\sum_{X}^{NumAuthor} commit[AuthorX \rightarrow AuthorX]}{\sum_{X}^{NumAuthor} totalcommit[X]} \tag{4.5}$$

と定義する．

また，このメトリクスに関して，6 つの異なるリポジトリでどのように変化するかを調べた．

RQ3: 最も頻繁に発生したバグ混入と修正のパターンは何か．

対象のリポジトリに関する全ての $commit[AuthorX \rightarrow AuthorY]$ のうち，最大の $commit[AuthorX \rightarrow AuthorY]$ をとるときの $AuthorX$ および $AuthorY$ を特定する．

さらに，$AuthorX$ および $AuthorY$ が，どの程度強いバグコミッターもしくは修正コミッターであるかを調べる．

また，このメトリクスに関して，6 つの異なるリポジトリでどのように変化するかを調べた．

4.4 グラフの作成

バグの混入者と修正者との関係を，有向グラフで視覚化する．

グラフの作成には，Graphviz Version:1.01[8] を使用した．以下記述例に沿って，Graphviz の使い方を解説する．

- `pschonbac [style=filled colorscheme=reds9 color=4 fontsize=50 ];`
  ノードを作る．ノードの色セットを `colorscheme` で，ノードの色を `color` で，ノードの文字の大きさを `fontsize` で指定する．本実験では，エッジに $Author$ を指定し，$Author$ がバグコミッターなら `colorscheme=reds9` を，修正コミッターなら `colorscheme=gnbu9` を指定した．また，グラフのノードを $AuthorX$ および $AuthorY$ にし，`color` には $p[Author]$ の自然対数をとった値を，`fontsize` には $totalcommit[Author]$ の自然対数をとった値を指定した．
- `pschonbac -> sefftinge [label = 11 penwidth=3];`
  ノードとノードとをエッジで結ぶ．エッジに添える値を `label` で，エッジの太さを `penwidth` で指定する．本実験では，$AuthorX$ の混入したバグを $AuthorY$ が修正することを `AuthorX -> AuthorY` のように指定した．また，`label` には $commit[AuthorX \rightarrow AuthorY]$ の値を，`penwidth` には $commit[AuthorX \rightarrow AuthorY]$ の自然対数をとった値を指定した．

4.5 実験結果

6 つのソフトウェアリポジトリに対して，以下，実験によって得られたデータを整理し，Research Question に答えるための実験結果を示す．

RQ1: 最もバグを混入しやすい開発者は誰か．最もバグを修正しやすい開発者は誰か．

6 つの Git リポジトリごとに，最もバグを混入しやすい開発者を表 4.2 にて，最もバグを修正しやすい開発者を表 4.3 にて与える．

RQ2: 開発者が自分で混入したバグを自分で修正する割合はどの程度か．

6 つの Git リポジトリごとに，開発者が自分で混入したバグを自分で修正する割合 r の値を表 4.4 にて与える．

RQ3: もっとも頻繁に発生したバグ混入と修正のパターンは何か．

6 つの Git リポジトリごとに，最大の $commit[AuthorX \rightarrow AuthorY]$ と，そのときの $AuthorX$ および $AuthorY$ を表 4.5 にて与える．また，そのとき $Author$ がバグコミッターなら B を，修正コミッターなら F を，そしてコミット傾向値 $p[Author]$ の値を併記した．

バグの混入者と修正者との関係を視覚化したグラフを図 4.1 から図 4.6 に示した．

表 4.2　最もバグを混入しやすい開発者

Git リポジトリ	$NumAuthor$	最もバグを混入しやすい開発者	$p[Author]$
emf.compare	3	cbrun	161.6869
webtools.incubator	11	david_williams	54.0351
ecf	12	slewis	111.3220
xpand	17	bkolb	52.6190
hadoop	37	Owen O'Malley	185.5534
birt	73	Dazheng Gao	496.2544

表 4.3　最もバグを修正しやすい開発者

Git リポジトリ	$NumAuthor$	最もバグを修正しやすい開発者	$p[Author]$
emf.compare	3	lgoubet	187.2259
webtools.incubator	11	sclarke	33.6362
ecf	12	mkuppe	141.6955
xpand	17	kthoms	37.6129
hadoop	37	Todd Lipcon	60.9231
birt	73	xizhang	396.9882

表 4.4　開発者が自分で混入したバグを自分で修正する割合

Git リポジトリ	$NumAuthor$	r
emf.compare	3	0.5706
webtools.incubator	11	0.7536
ecf	12	0.7744
xpand	17	0.1220
hadoop	37	0.1632
birt	73	0.3979

表 4.5　もっとも頻繁に発生したバグ混入と修正のパターン

Git リポジトリ	$NumAuthor$	最大の $commit[Author \rightarrow AuthorY]$	$AuthorX$ (BorF,$p[AuthorX]$)	$AuthorY$ (BorF,$p[AuthorY]$)
`emf.compare`	3	581	lgoubet (F,187.2259)	lgoubet (F,187.2259)
`webtools.incubator`	11	423	sclarke (F,33.6362)	sclarke (F,33.6362)
`ecf`	12	1164	slewis (B,111.3220)	slewis (B,111.3220)
`xpand`	17	20	bkolb (B,52.6190)	sefftinge (F,16.6667)
`hadoop`	37	41	Owen O’Malley (B,185.5534)	Thomas White (F,35.1351)
`birt`	73	824	yulin (F,28.5739)	yulin (F,28.5739)

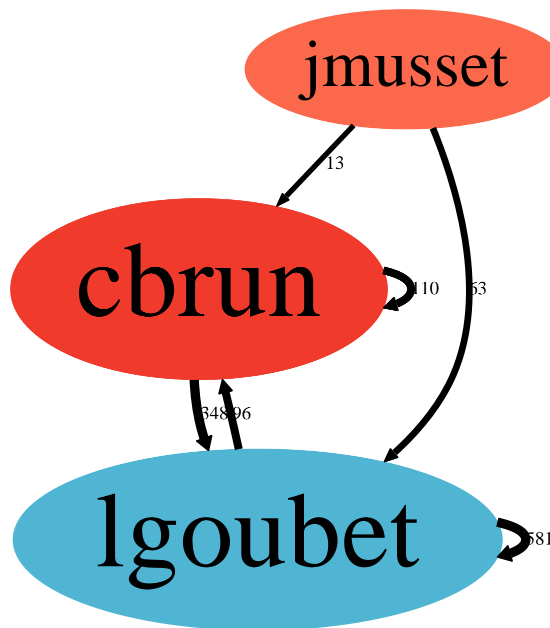

図 4.1　eclipse.emf.compare のバグの混入者と修正者との関係グラフ

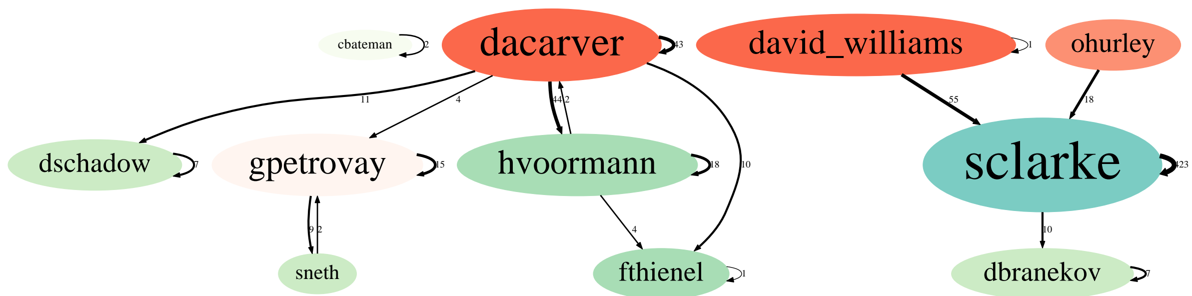

図 4.2　eclipse.webtools.incubator のバグの混入者と修正者との関係グラフ

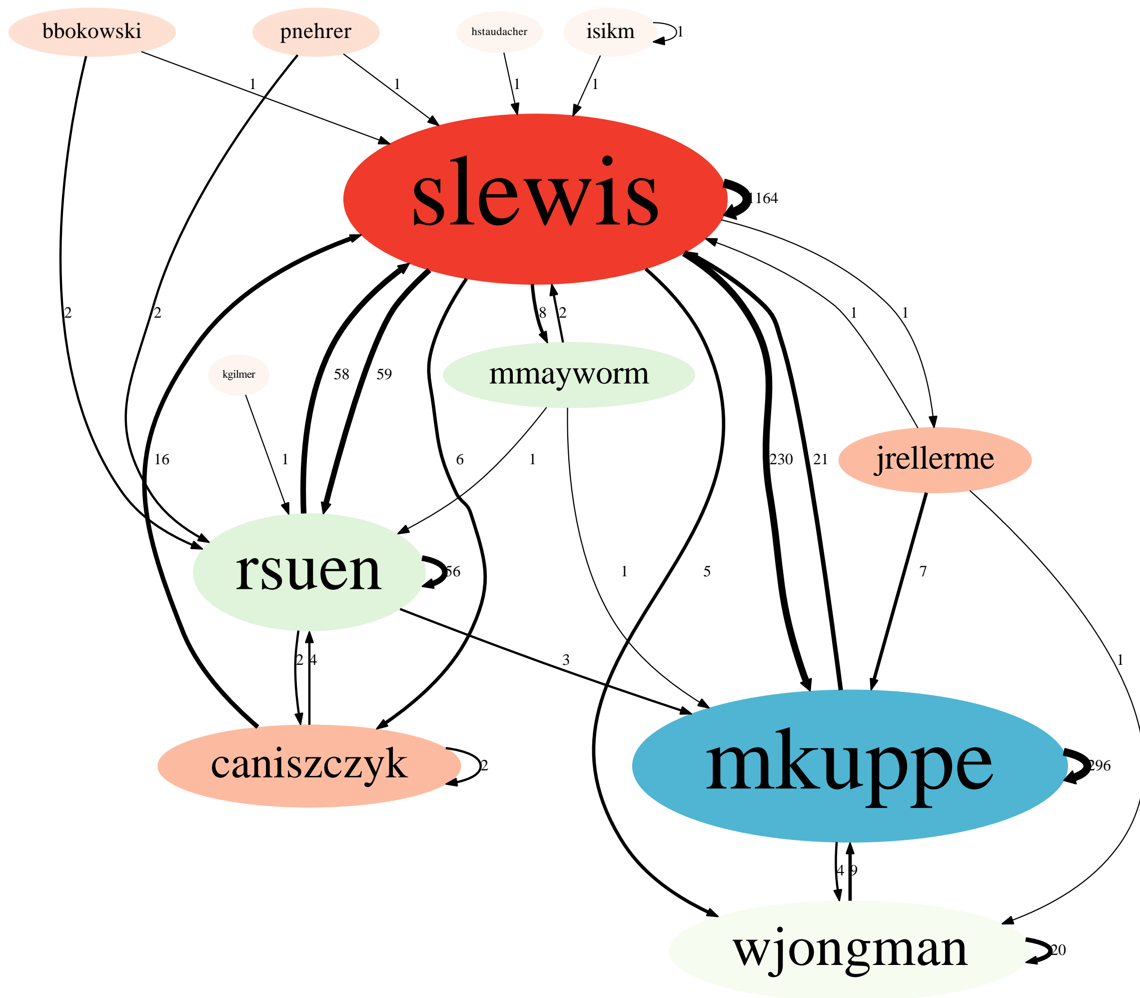

図 4.3　eclipse.ecf のバグの混入者と修正者との関係グラフ

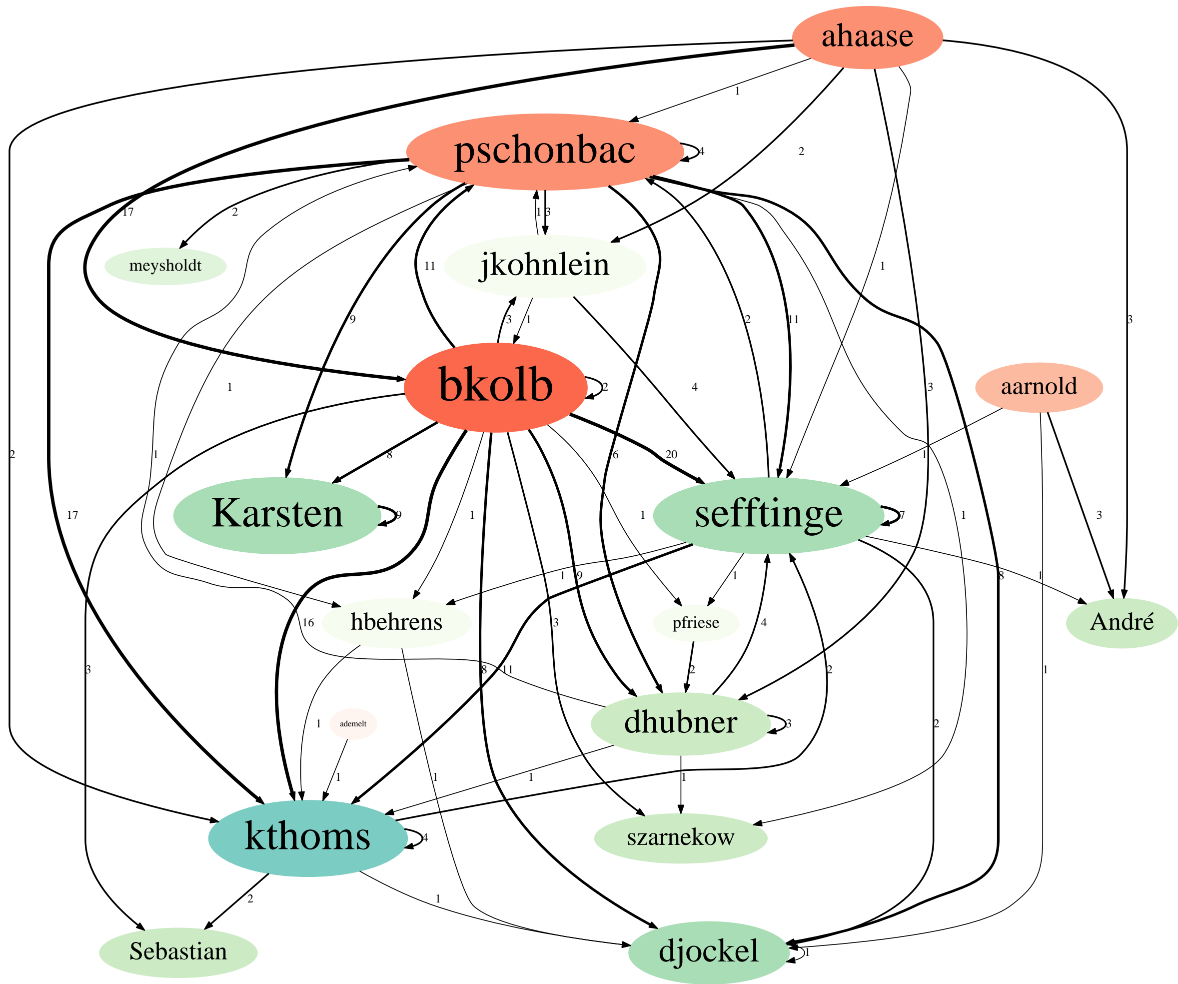

図 4.4　eclipse.xpand のバグの混入者と修正者との関係グラフ

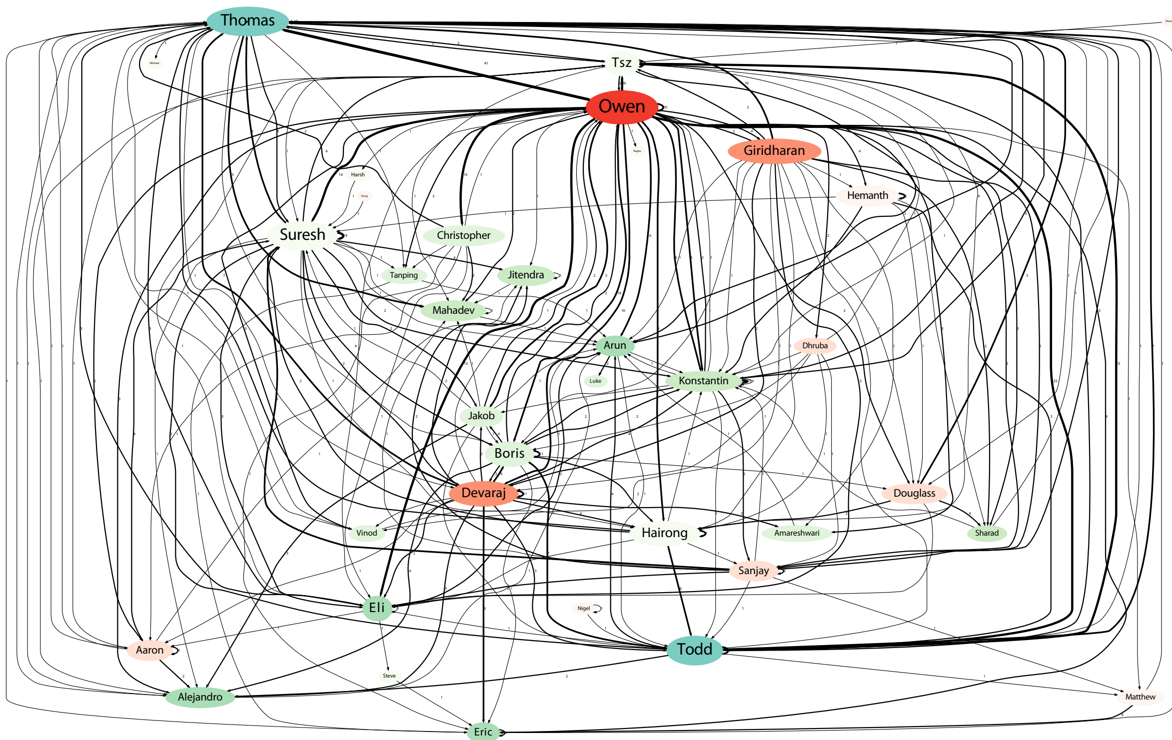

図 4.5　hadoop-common のバグの混入者と修正者との関係グラフ

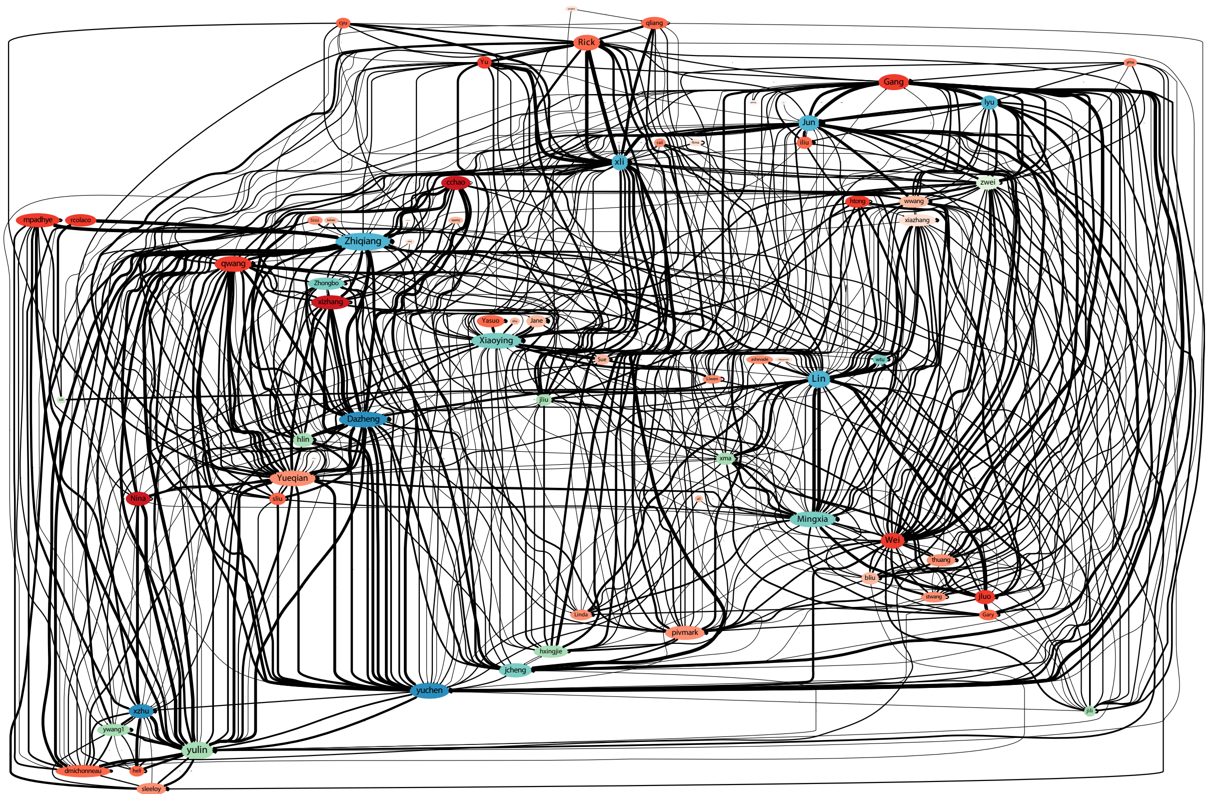

図 4.6　eclipse.birt のバグの混入者と修正者との関係グラフ

5. 考察

実験結果から Reserch Question に答える前に，全般的な実験結果の妥当性についていくつか議論する．

- バグリポジトリの不備
  本実験では，SZZ アルゴリズムによって，バージョン管理およびバグ管理システムのリポジトリからバグ混入情報とバグ修正情報の *Author* 情報を抽出した．SZZ アルゴリズムの実装に問題があるために，正しく *Author* の情報が得られていない可能性がある．
- バージョン管理システムの不備
  バージョン管理システム自体が管理体制に不具合があったり，ユーザが正しくシステムを運用せず，誤った情報を入力した等の理由で，元々のデータが正しくない可能性がある．
- プログラムのバグ
  Git リポジトリから *Author* の情報を抽出するために実装したプログラムに不具合が含まれている可能性もあり，抽出した *Author* の情報に誤りが存在する可能性があると考えられる．

以上の実験結果の妥当性について注意し，実験結果から Reserch Question に答え，それに対して考察をする．

RQ1: 最もバグを混入しやすい開発者は誰か．最もバグを修正しやすい開発者は誰か．

表 4.2 と表 4.3 のように，各ソフトウェアリポジトリごとに最もバグを混入しやすい開発者=最も害のある開発者および，最もバグを修正しやすい開発者=最も有益な開発者を特定できた．

最も $NumAuthor$ が大きい `birt` は，他の Git リポジトリの場合と比較して，最もバグを混入しやすい開発者および最もバグを修正しやすい開発者の両方について，コミット傾向値 $p[Author]$ が特に大きくなった．

しかし，最も $NumAuthor$ が小さい `emf.compare` のコミット傾向値 $p[Author]$ も比較的大きいため，$NumAuthor$ が大きいほど $p[Author]$ が大きくなるとは言えない．

確かに，$NumAuthor$ が特に大きい `birt` のときは他のリポジトリと比較して $p[Author]$ が非常に大きいが，今回の実験では $NumAuthor$ が特に大きいリポジトリのサンプルが `birt` だけだったので，必ずしも $NumAuthor$ が大きければ $p[Author]$ が大きくなるとは限らない．

RQ2: 開発者が自分で混入したバグを自分で修正する割合はどの程度か．

$NumAuthor$ が小さいほど，開発者が自分で混入したバグを自分で修正する割合 r が大きくなる傾向があり，$NumAuthor$ が大きい，開発者が自分で混入したバグを自分で修正する割合 r が小さくなる傾向があった．

コミットした開発者の数が少ないということは，同じ開発者がバグの混入やバグの修正を繰り返していることが多いため，開発者が自分で混入したバグを自分で修正する割合が高くなると考えられる．

RQ3: 最も頻繁に発生したバグ混入と修正のパターンは何か．

$NumAuthor$ の大きさに関わらず，開発者が自分で混入したバグを自分で修正する場合が比較的多かった．特にここで，最もバグを混入する $Author$ や最もバグを修正する $Author$ が挙げられることが多かった．

ソフトウェアプロジェクトで，最もバグを混入する開発者および，最もバグを修正する開発者は，自分で混入したバグを自分で修正することが多いことが分かった．

つまり，最もバグを混入する開発者は，自分の混入した大量のバグを放置せずに，多くの場合は自分で修正し，最もバグを修正する開発者は，他の開発者が混入したバグの修正よりも，多くの場合は自分が混入したバグの修正を行うことが分かった．

6.　結言

本研究では，リポジトリマイニングに基づくバグ混入者と修正者との関連を分析し，最もバグを混入する開発者や最もバグを修正する開発者を特定することで，今後のソフトウェア開発プロジェクトに携わる開発者に関する研究に有益な情報の収集を行った．

ソフトウェア開発プロジェクトにおいては，混入されたバグは，多くの場合同一の開発者が修正することが分かった．

今後の課題としては，より多くの種類のソフトウェアリポジトリに対して同様の実験を行うことで，リポジトリの規模の違いによる開発者同士の関係をより明確にすること，バグ混入者やバグ修正者を定義するメトリクスの有効性を調査することが挙げられる．

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，梁軍偉先輩，川本公章先輩，出原真人先輩，中井道先輩，情報工学課程，西村祐輔先輩，椋代凛君，小野木祐太君，松村好剛君，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] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," Proceedings of the 2005 international workshop on Mining software repositories, pp.1–5, ACM, St. Louis, Missouri, 2005.

[2] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pp.13–23, SIGSOFT '08/FSE-16, ACM, Atlanta, Georgia, 2008.

[3] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?," Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pp.2–12, SIGSOFT '08/FSE-16, ACM, Atlanta, Georgia, 2008.

[4] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," Proceedings of the 31st International Conference on Software Engineering, pp.1–11, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009.

[5] 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，2011.

[6] Git, Git - Fast Version Control System, Git - Fast Version Control System（オンライン), 入手先 ⟨http://git-scm.com/⟩（参照 2012-02-12）.

[7] T. Zimmermann and P. Weißgerber, "Preprocessing cvs data for fine-grained analysis," Proceedings of the First International Workshop on Mining Software Repositories, pp.2–6, May 2004. Edinburgh, United Kingdom.

[8] Graphviz, Graphviz - Graph Visualization Software, Graphviz - Graph Visualization Software（オンライン), 入手先 ⟨http://www.graphviz.org/⟩（参照 2012-02-14）.