メトリクス測定に基づくオープンソースプロジェクトにおける実証的研究

An Empirical Study of Open Source Projects based on Measurement Metrics

高澤 亮平* 坂本 一憲† 鷲崎 弘宜‡ 深澤 良彰§

あらまし
オープンソースプロジェクトのリポジトリを解析することで、ソフトウェア開発に有益な知見を収集する研究が盛んに行われている。しかし、ソフトウェアテストに関する情報を収集する試みは少ない。本稿では、オープンソースリポジトリマイニングツールを実装し、GitHub 上の 495 のオープンソースプロジェクトを対象として、ソフトウェアテストに関するいくつかのメトリクスを測定する。また、テストを実行し、その結果を収集する。以上の結果から、オープンソースプロジェクトにおけるテストの現状を明らかにする。実験の結果、テストコードが存在するプロジェクトは 88%に上ったものの、テスト実行が正常に行われるプロジェクトは 34%にとどまり、さらにはテスト実行が成功するプロジェクトは 21%にとどまることが明らかになった。

1 はじめに

オープンソースプロジェクトとは、プログラムのソースコードなどをインターネットなどを通して無償で公開し、ソフトウェアの改良、再配布を可能にしているプロジェクトである。プロジェクトの成果物を共有することで、複数人で協調して開発を進める形態を可能にしている。近年では、インターネット上にプロジェクトのリポジトリを置く、リポジトリ共有ホスティングサービスが普及し、それに伴い、オープンソースのプロジェクト数も増加傾向にある [1]。

オープンソースプロジェクトの普及とともに、オープンソースプロジェクトを対象とした研究も盛んに行われている。例えば、Hars ら [2] は、開発者がオープンソースプロジェクトに参加する要因を分析している。Bonaccorsi ら [3] は、オープンソースプロジェクトが普及し、成功した要因を分析している。Schryen ら [4] は、オープンソースプロジェクトとコードを公開しないクローズドソースソフトウェアを、脆弱性の観点から比較している。

また、商用ソフトウェアに比べソースコードの入手が容易であることから、ソフトウェアメトリクスやソフトウェアテストの研究において、提案手法をオープンソースプロジェクトに適用し、評価を行っているものが数多く存在する。例えば、Ciupa らの研究 [5] では、筆者らが提案したテスト手法をオープンソースのライブラリに適用し、従来手法よりも必要時間を減少できたことを示している。

しかし、オープンソースを対象とした大規模な実証的研究の実例は少ないのが現状である。Fraser らの研究 [6] における調査によると、既存のオープンソースを対象としたテストデータ生成ツールの実証的研究においては、対象とするプロジェクト数が多くとも 10 個であったことが示されている。また、対象とするプロジェクトの選び方も不透明であり、手法の適用への制限が少ないなどの、同じような特徴を持つプロジェクトが多く選ばれている傾向があると述べられている。

*Ryohei Takasawa, 早稲田大学
†Kazunori Sakamoto, 国立情報学研究所
‡Hironori Washizaki, 早稲田大学
§Yoshiaki Fukazawa, 早稲田大学

また、テストコードに着目した実証的研究は少ないのが現状である。Shah らの研究 [7] によると、実際の開発現場においても、プロジェクトにおけるテストコードはあまり注目されておらず、開発者の中でも、テストは開発の二の次であるという認識が広まっていると述べられている。

そこで本研究は、複数のオープンソースプロジェクトを対象とし、大規模な実験を行う。具体的には、複数のオープンソースプロジェクトに対して、メトリクス値の測定を行う。また、テストコードに着目し、テストカバレッジやプロジェクト内のテストコードの閉める割合を算出する。これらの実験を通して、現状のオープンソースに関する知見を得ることを目的とする。

本研究におけるリサーチクエスチョンは、以下の通りである。

RQ1 テストコードが存在しているオープンソースプロジェクトはどれぐらいの割合で存在するか？

RQ2 総コード行数におけるテストコードの割合はどのぐらいか？

RQ3 テスト記述者の数はテストの結果と関係があるか？

本稿による貢献は以下の通りである。

1. オープンソースプロジェクト向けのリポジトリマイニングツールを作成した
2. 現状のオープンソースプロジェクトにおけるソフトウェアテストに関するメトリクス値を測定し、テストの現状を明らかにした
3. 現状のオープンソースプロジェクトのテストにおける問題点を明らかにした

本稿の構成は以下のとおりである。2 節で実験の概要、3 節で実験手法の詳細を述べ、4 節で測定結果と考察、5 節で妥当性への脅威、6 節では制限を述べる。7 節では関連研究を、8 節ではまとめと今後の展望を述べる。

2 リポジトリマイニング手法の概要

本節では、マイニング手法の概要として、収集するデータの内容を説明する。

2.1 リポジトリの収集

リポジトリとは、プロジェクトに必要なファイルなどがひとまとめになっているものである。インターネット上に公開されているオープンソースプロジェクトを探し、実験手法を適用させるため手元の実行環境への複製を行う。

2.2 テスト実行

テストの存在の有無を調査するため、各プロジェクトに対してテスト実行を行う。テスト実行時のメッセージから、テストケースの成功数や失敗数の情報を収集する。また、テスト実行がスキップされたプロジェクトや、失敗したプロジェクトに対して、原因を調査する。

2.3 コード行数の測定

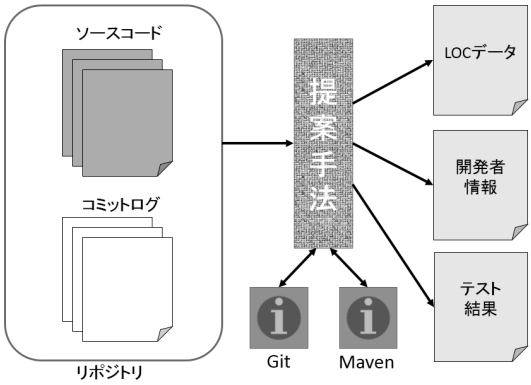
テストの記述量を調査するため、各プロジェクトに対してコード行数の測定を行う。テストコードを含むソースコードの総物理行数と、テストコードのみの総物理行数を測定する。これらの値を用いて、プロジェクト内にテストコードが存在するか否かの判別と、プロジェクト内のコードにおけるテストコードの占める割合を算出する。

2.4 開発者数の測定

テストアクティビティに対する開発者の行動を調査するため、各プロジェクトの開発者の数の測定を行う。テストコードを含むソースコードの開発者数と、テストコードのみの開発者数を測定する。これらの値を用いて、複数の開発者によりテストが行われている割合を調査する。

3　実験手法の詳細とツールの実装

本節では、実験手法の詳細と、実装方法について説明する。実験に使用したツールの概要を図 1 に示す。



1　ツールの概要

3.1　リポジトリの収集

プロジェクト共有ホスティングサービスである GitHub [10] を利用し、プロジェクトのリポジトリを収集する。プロジェクト共有ホスティングサービスとは、オープンソースを含むリポジトリを保管し、プロジェクトの分散開発をサポートする機能や、条件に合わせた検索機能などが利用可能となっているサービスである。

本実験で対象とするリポジトリは、以下の条件を満たしている必要がある。

1. Git を使用している
2. Java 言語、JUnit を使用している
3. Maven を利用している

1. の Git [8] とは、バージョン管理システムと呼ばれるシステムの一種である。バージョン管理システムとは、ファイルの変更履歴などを保管しておくシステムであり、後述のテスト記述者の数の測定の際に用いる。GitHub 上のリポジトリは、すべて Git を利用しているため、1. に関してはすべてのプロジェクトが既に満たしていると考えられる。

2. は、後述のテスト実行において、Java 言語を対象としたテストツールである JUnit [12] を利用しているため、必要となる。GitHub での検索条件としては、JUnit が Java 言語のテストツールにおけるデファクトスタンダードであることから、Java 言語を利用していることを条件としている。

3. の Maven [9] とは、プロジェクト管理ツールと呼ばれるツールの一種である。プロジェクト管理ツールとは、プロジェクトの依存するライブラリの管理や、ビルドやテストなどのプロジェクトにおけるタスクの支援など、プロジェクトのライフサイクル全体を管理するツールである。本手法では、テスト実行において Maven を利用している。

Maven を利用する際には、ライブラリの依存関係の情報や、プラグインの設定などを pom.xml という設定ファイルに記述する必要がある。このことから、GitHub での検索条件としては、pom.xml ファイルがプロジェクト内に存在することを条件としている。

以上の 3 つの条件に当てはまるリポジトリを GitHub で検索する。GitHub が提供している API では、リポジトリ内に特定のファイルが存在するか、等の条件を指定した検索を行うことができない。このため、Web アプリケーション用テストツールである Selenium [11] を用い、ブラウザを操作し検索を行う。検索結果のページから、リポジトリの入手に必要なアドレスを取得する。

3.2 コード行数の測定

各プロジェクト内の、コードが記述されているすべてのファイルの物理行数を測定する。また、テストコードがあるコードを判別し、テストコードのあるすべてのファイルについて、物理行数を測定する。

リスト 1 に、JUnit [12] で記述されたサンプルテストケースを示す。これを用いて、テストコードの要素について説明する。

リスト 1　サンプルテストケース

```
1  public class SampleTestCase {
2    @Test
3    public void testEqual1() {
4      sc.set(3);
5      assertEquals(sc.getResult(), 3);
6    }
7
8    @Test
9    public void testEqual2() {
10     assertEquals(sc.getResult(), null));
11   }
12 }
```

リスト 1 において、@Test の部分はアノテーションと呼ばれ、以下のメソッドの注釈を表す記述である。@Test から始まる部分が、テストメソッドと呼ばれる部分であり、以下の記述が 1 つのテストケースを表す。テストメソッド内の assertEquals 文は、アサート文と呼ばれ、テストの対象とする値と期待される結果が一致するか確認している。アサート文の結果により、テストの成功と失敗が判定される。

本稿では、プロジェクト内のソースコードのうち、“test”という名前のフォルダ内に存在するコードをテストコードとして扱う。これは、テストメソッドの記述以外にも、外部の関数としてテストに必要な記述が存在する場合に対処するためである。

3.3 テスト実行

Maven に用意されている test コマンドを利用する。プロジェクト内に存在するテストメソッドを含むファイルが自動的に検索され、すべてのテストが自動的に実行される。テスト実行の結果として、リスト 2 のような出力が得られる。

リスト 2　サンプルテスト結果

```
1  -------------------------------------------------------
2   T E S T S
3  -------------------------------------------------------
4  Running com.googlecode.mycontainer.commons.file.RegexFileFilterTest
5  Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.028sec
6  Running com.googlecode.mucontainer.commons.file.FileComparatorTest
7  Tests run: 3 Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.028sec
8  Running com.googlecode.mucontainer.commons.io.TimeoutInputStreamTest
```

```
9  Tests run: 2 Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.028sec
10
11 Results :
12
13 Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
```

テストを実行すると、テストメソッドを含むファイルごとにテストメソッドが実行され、その結果が得られる。実行結果はRun、Failure、Error、Skipの4つのパラメータを含む。各パラメータは実行したテストケースの数、失敗したテストケースの数、エラーが発生したテストケースの数、実行されなかったテストケースの数を表している。また、テスト実行にかかった時間がTime elapsedとして出力される。最後の行に、パッケージごとに結果を集計した結果が出力される。

本手法では、出力結果の文字列から各パラメータごとに値を集計した。また、Runが1以上となった場合にテスト実行が正常に行われたと判断し、FailureとErrorが0の場合のみをテストが成功したと判断した。

3.4 開発者数の測定

開発者の情報を取得するために、Gitに用意されているblameコマンドを用いる。blameコマンドを用いることで、ソースコードの各行に対し、最新の変更情報を出力として得ることができる。例として、リスト1に対してblameコマンドを用いた出力結果の一部を、リスト3に示す。

リスト 3　blame コマンドの結果

```
1  654ad4a4 (Ryohei Takasawa   2013-07-10 17:12:21 +0900  9) @ Test
2  654ad4a4 (Ryohei Takasawa   2013-07-10 17:12:21 +0900 10) public void testEqual1() {
3  654ad4a4 (Ryohei Takasawa   2013-07-10 17:12:21 +0900 11)   sc.set(3);
4  654ad4a4 (Ryohei Takasawa   2013-07-10 17:12:21 +0900 12)   assertEquals(p.getX(), 1);
5  654ad4a4 (Ryohei Takasawa   2013-07-10 17:12:21 +0900 13) }
6  654ad4a4 (Ryohei Takasawa   2013-07-10 17:12:21 +0900 14)
7  ^d3096dd (Kazunori Sakamoto 2013-07-12 09:20:59 +0900 15) @Test
8  ^d3096dd (Kazunori Sakamoto 2012-07-12 09:20:59 +0900 16) public void testEqual2() {
9  ^d3096dd (Kazunori Sakamoto 2012-07-12 09:20:59 +0900 17)   assertEquals(sc.getResult
10 ^d3096dd (Kazunori Sakamoto 2012-07-12 09:20:59 +0900 18) }
```

リスト3では、ソースコードの各行の左側に、行番号に加え最新の変更日時や、最後に変更を加えた開発者の名前が記録されている。最左端の文字列は、各変更を識別するためのものである。変更の一単位のことをコミットと呼ぶ。

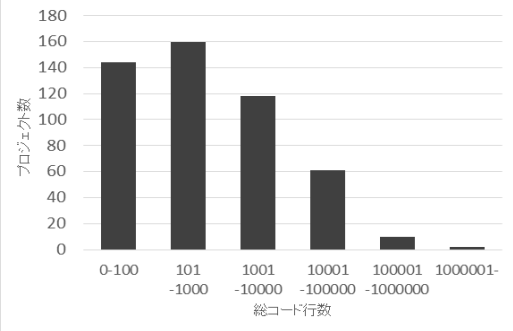
この出力結果から開発者名を抽出し、異なる開発者名の数を開発者の人数としてカウントする。本手法では、すべてのコードにおける異なる開発者の数、テストコードにおける異なる開発者の数を算出する。これを用いて、テストが成功したプロジェクトに対して、すべての開発者の数、テストコードの開発者の数を算出し、全開発者におけるテストコードの開発者の割合を算出する。

4 実験結果

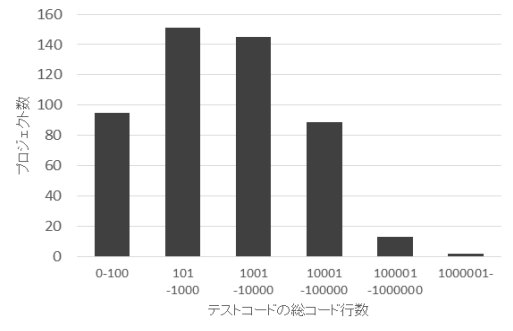
本章では、実験の結果とそれに基づく考察を述べる。実験対象は、3.1で述べた条件でGitHub上で検索を行い、検索結果の上位100ページに表示されたプロジェクトのうち、重複を除き、入手が可能であった495プロジェクトを対象とした。

4.1 コード行数

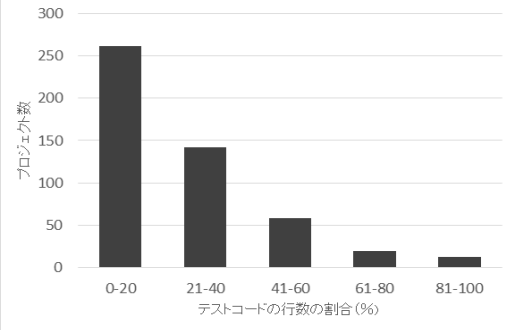
コード行数とプロジェクトの数の関係を図 2 に、テストコードとプロジェクト数の関係を図 4 にそれぞれ示す。また、総コード行数におけるテストコードの行数の割合とプロジェクト数の関係を図**??**に示す。



2 総コード行数とプロジェクト数



3 テストコードのコード行数とプロジェクト数



4 総コード行数におけるテストコードの行数の割合とプロジェクト数

コード行数に関しては、全プロジェクトの平均は 131974.1、テストコードのコード行数に関しては、全プロジェクトの平均は 22020.07 となった。テストコードの行数が 0、つまりテストコードが存在しないプロジェクトの数は 57 となり、11.5%のプロジェクトにはテストコードが存在しないという結果になった。

総コード行数におけるテストコードの行数の割合においては、全体の約 5 割を超える 261 のプロジェクトが、20%以下に収まるという結果になった。また、テストコードが総コード行数の半分以上を占めるプロジェクトも 53 プロジェクト存在した。

以上の結果より、RQ1 への回答として、88.5%のプロジェクトにテストコードが存在するという結果が明らかとなった。また、RQ2 への回答として、コード行数に対するテストコードの割合の平均は 23.4%となることが明らかとなった。

4.2 テスト実行

テスト実行の結果、テスト実行が正常に行われるプロジェクトの数は 168 にとどまり、割合としては全体の 33.9%にとどまるという結果になった。

また、テスト実行が成功したプロジェクトの数は 104 となり、割合としては全体

の 21.0%のみがテストが成功するという結果になった。

テスト実行が行われなかった理由としては、以下の原因が存在した。

1 テストコードが存在しなかった。
2 ビルドが成功していなかった。
3 Maven の設定でテストがスキップされるようになっていた。

1. に関しては、4.1 でも述べたように、プロジェクト内にテストコードがそもそも存在していないということが原因である。このようなプロジェクトには、テストを実行することは最初から意図していないということが考えられる。

2. に関しては、テストの前にビルドが行われるが、その時点で失敗したことで、テスト実行まで到達しなかったというのが原因である。このようなプロジェクトが 194 存在し、全体の 39.2%を占めるという結果になった。このようなプロジェクトは、入手時点で動作しないプロジェクトということになる。

3. に関しては、Maven の設定によりすべてのテストがスキップされるようにプロジェクトが設定されていたものである。このようなプロジェクトでは、テストコードはあらかじめ用意しておき、開発が進んだ段階でテスト実行を行っていくという開発手法がとられていることが考えられる。

また、テスト実行が成功しなかった要因として、プロジェクトのファイルを入手した際に、依存関係のあるファイルなどがリポジトリに含まれていない、もしくは設定ファイルに記述されていないことで、テスト実行の際にエラーが発生してしまうというのが主な原因であった。

以上の結果より、プロジェクト中にテスト記述はあるものの、正常に実行できない、あるいはテストが失敗するプロジェクトが数多く存在することが明らかとなった。

4.3 テスト記述者の特定

テストが成功した 104 プロジェクトのうち、テストコードが単一の開発者のみによって記述されていたプロジェクトは 43 プロジェクトとなり、残りの 61 プロジェクト、割合にすると 58.7%のプロジェクトにおいて複数の開発者によるテストが行われているという結果になった。また、すべてのプロジェクトにおいて開発者の特定を行った結果と、テストが成功したプロジェクトにおける結果の比較を表 1 に示す。

	プロジェクト数	テストコードの割合（平均）	テスト記述者数（平均）
成功	104	27.70%	4.18 人
すべて	492	23.37%	4.09 人

} 1 テストコードの割合、テスト記述者数の比較

テストが成功しているプロジェクトは、すべてのコード中のテストコードの割合、テスト記述者数ともに多くなっていることが分かる。この結果か、ら RQ3 への回答として、テストが成功するプロジェクトは、テスト記述者の数が多いという傾向があるのではないかと考えられる。

正確なテスト記述者の特定ができないプロジェクトが複数存在した。この原因としては、Git 以外のバージョン管理システムが使用されていたことがある。バージョン管理システムには、Git のほかにも Subversion [14] や Mercurial [15] など、さまざまな種類が存在する。GitHub 上のプロジェクトは通常 Git を使用しているが、一部のプロジェクトにおいて Subversion が使用されていたため、本手法では情報の取得が行えなかったものである。

5 妥当性への脅威

本節では、本手法の妥当性への脅威を説明する。

5.1 外的妥当性への脅威

本手法では GitHub を利用したが、他にもさまざまなプロジェクト共有サービスが存在する。一例を挙げると、SourceForge [13] や Google Code [16] があげられる。これらの他のサービスにおいても GitHub と同様な結果が得られるかは不明であり、更なる実験を行っていきたい。

5.2 内的妥当性への脅威

本手法では無作為に対象となるプロジェクトを選定したが、プロジェクトの規模や開発の内容によって結果に差異が生じることが考えられる。例えば、プロジェクトの規模が大きいほど開発者の数が多いことや、テストコードが存在する可能性が大きいことが考えられ、これらを考慮したプロジェクトの選定を行い、傾向の分析を行っていきたい。

6 制限

本節では、本手法の制限を説明する。本手法では、テスト実行結果の取得のために Maven、開発者の特定のために Git を使用している。このため、これらのツールが正しく実行できなかったプロジェクトは、各項目において評価対象から除外している。

Maven においては、設定ファイルが正しく記述されていないものにおいては、ビルドやテスト等が正確に行えず、結果の収集ができないという制限がある。この場合は、プロジェクトのファイルの入手時点でビルドやテストなどが行えないということであり、開発者側におけるプロジェクト管理の問題点であると考えられる。

Git においては、バージョン管理システムとして意図されていない使い方をされている場合、正確な測定ができないという制限がある。意図されていない使い方の例として、ソースコードの内容は変わらないが、他の開発者により書き直されてしまった場合が考えられる。例えば、他の開発者が、元から存在するソースコードをすべてコピーし、そのまま貼り付けた場合、ソースコードの記述者を正しく判別することが不可能になる。これに対応するためには、コミット間の差異を時系列順に追うなどの対策が必要である。

7 関連研究

オープンソースプロジェクトを対象とした実証的研究がいくつか存在する [6] [17]。

Fraser らは、複数のオープンソースプロジェクトを用いた実験結果を元に、オープンソースを対象とした研究向けのコーパスを提案している [6]。Fraser らの手法では、SourceForge 上のプロジェクトに対して、自作のテストデータ自動生成ツールを用い、ソースコード中の分岐の網羅率を測定した。測定結果を元に、テストデータ自動生成ツールの網羅率向上に必要とされる改良点を明らかにした上で、今後の研究に利用可能なオープンソースのプロジェクトのセットを提案した。しかし、提案手法ではブランチカバレッジのみに着目しており、複数のメトリクスからの評価は行われていない。また、プロジェクトの選定の手法や規模の大きさは考慮されていない。

Gousion らは、オープンソースプロジェクトにおける各開発者の貢献を測定する手法を提案している [17]。 Gousios らの手法では、プロジェクト内での各開発者の行動を分類し、その結果を元に各開発者の貢献度を数値化している。しかし、提案手法ではプロジェクトからの定量的な情報の取得方法が述べられておらず、実際のプロジェクトにおける評価もなされていない。一方、本手法においては、バージョン管理システムから開発者の機械的な特定が可能であり、実際に測定した定量的な値から各開発者の行動を分類することが可能である。

その他にも、オープンソースプロジェクトに関する研究が存在する。Pham ら [18]

は、オープンソースプロジェクトにおいて、テストコードが存在するプロジェクトのほうが、部外者による貢献がなされやすいということを示している。

また、Mclntosh ら [19] は、オープンソースプロジェクトにおいて、テスト開発者の多くがプロジェクトのビルド設定も行っていることを示している。本稿の実験の結果より、テスト記述者の不足がビルドの失敗を招いているという仮説を立てることもできる。

上述の研究から、オープンソースソフトウェアにおいて、テストコードの存在は大きな価値を持つものであり、推奨されるべき行為であると考える。しかし現状では、テストコードは存在するものの、テストが実行されない、もしくは成功しないプロジェクトが多く存在することが明らかとなった。今後のオープンソースプロジェクトの発展のためには、テストに関する不具合のあるプロジェクトが多いことは大きな問題であり、改善されるべきであると考える。

8 まとめと展望

本稿では、実際のオープンソースプロジェクトに対し、コード行数の測定と、テスト実行を行い、結果を収集した。また、テスト実行が成功したプロジェクトに対して、テストコードの開発者数を測定した。これらの結果をもとに、リポジトリ共有ホスティングサービス上のオープンソースプロジェクトにおける、現状の問題点を提示した。

1 節であげたリサーチクエスチョンについて、結果をもとに回答する。

RQ1 テストコードが存在しているオープンソースプロジェクトはどれぐらいの割合で存在するか？
—対象としたプロジェクトのうち、88.5%にテストコードが存在した。

RQ2 総コード行数におけるテストコードの割合はどのぐらいか？
—総コード行数に対するテストコードの割合の平均は 23.4%であった。

RQ3 テスト記述者の数はテストの結果と関係があるか？
—テストが成功するプロジェクトは、テスト記述者の数が多いという傾向があるのではないかと考えられる。

以降、今後の展望として、課題と考えられる改良点を述べる。

1. 対象とするプロジェクトの拡充
   対象とするプロジェクトの数をさらに増やすことが考えられる。今回の実験では、検索結果の上位のプロジェクトを対象に用いたが、GitHub の検索機能により、さらに細かい条件でのプロジェクトの検索が可能である。例えば、リポジトリの作成日時や、ファイルのサイズなどの条件を指定して検索することで、より詳細なデータの取得が可能になると考えられる。また、対象とするプロジェクトの数を増加させることで、データの有用性および妥当性の向上を図ることができると考える。
2. 測定するメトリクスの拡充
   今回の実験では、コード行数のみを測定したが、テストカバレッジやサイクロマティック複雑度など、他のメトリクスを測定することが考えられる。また、複数のメトリクスによる測定を行うことで、実際のオープンソースプロジェクトにおけるメトリクスの相関を調査することができるのではないかと考える。
3. オープンソースプロジェクト特有の機能の利用
   メトリクスだけでなく、オープンソース特有の機能の値を利用することが考えられる。例えば、プロジェクトへのお気に入りを示す star や、他の開発者のリポジトリへの貢献を意図する fork などの回数、さらには開発者が平行して行っているプロジェクトの数などが考えられる。これらの値を利用することで、開発者の行動の調査や、オープンソースでの開発に特有な知見を得ることができるのではないかと考える。

4. オープンソースプロジェクト検索ツールの実装
　特定の条件を満たすオープンソースプロジェクトを検索するツールを実装することが考えられる。3.1 で述べた通り、オープンソースプロジェクトの詳細な条件を指定した検索は Web ブラウザ上でしか行えないのが現状である。そこで、今回作成した検索の仕組みを利用することで、今後のオープンソースプロジェクトを対象とした研究に有用となるのではないかと考える。

参考文献

[1] Amit Deshpende, Dirk Riehle, ”The Toal Growth of Open Source”, Open Source Development, Communities and Quality, IFIP - The International Federation for Information Processing, Vol.275, pp197-209, 2008.
[2] A. Hars and S. Ou, “Working for free? motivations for participating in open-source projects”, International Journal of Electronic commerce, Vol.6, pp25-69, 2002.
[3] A. Bonaccorsi and C. Rossi, “Why open source software can succeed”, Reserach Policy, Vol.32, pp.1243-1258, 2003
[4] G. Schryen, R. Kadura, “Open source vs. closed source software: toworads measuring security”, ACM symposium on Applied Computing (SAC 2009), pp2016-2013, 2009.
[5] I. Ciupa, A. Leitner, M. Orils, and B. Meyer, “ARTOO: adaptive random testing for object-oriented software”, International Conference on Software Engineering (ICSE 2008), pp71-80, 2008.
[6] Gordon Fraser, Andrea Arcuri, “Sound Empirical Evidence in Software Testing”, International Conference on Software Engineering (ICSE 2012), pp178-188, 2012.
[7] Hina Shah and Mary Jean Harrold, “Case Study: Studying Human and Social Aspects of Testing in a Service-Based Software Company”, ICSE Workshop on Cooperative and Human Aspects of Software Engineering pp102-108, 2010.
[8] Git, http://git-scm.com/.
[9] Apache Maven Project, http://maven.apache.org/.
[10] GitHub - Build software better, together., https://github.com/.
[11] Selenium - Web Browser Automation, http://docs.seleniumhq.org/
[12] JUnit, http://junit.sourceforge.net/.
[13] SourceForge - Find, Create, and Publish Open Source software for free, http://sourceforge.net/.
[14] Apache Subversion, http://subversion.apache.org/.
[15] Mercurial SCM, http://mercurial.selenic.com/.
[16] Google code, http://code.google.com/.
[17] Georgios Gousios, Eirini Kalliamvvakou, Diomidis Spinellis, “Measuring Developer Conribution from Software Repositor Data”, Working Conference on Mining Software Repository (MSR 2008), pp129-132, 2008.
[18] Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider “Creating a Shared Understanding of Testing Culture on a Social Coding Site”International Conference on Software Engineering (ICSE 2013), pp112-121, 2013.
[19] Shane Mclntosh, Bram Adams, Thanh H. D. Nguyen, Yasutaka Kmei, and Ahmed E. Hassan “An Empirical Study of Build Maintenance Effort”International Conference on Software Engineering (ICSE 2011), pp141-150, 2011.