

オープンソース開発におけるソースコードの安定性予測について

阿 萬 裕 久†

本稿ではオープンソース開発におけるソースコードの安定性に着目し、その状態を4段階に分類している。そして、実際のオープンソースソフトウェア Eclipse について測定実験を行い、安定性に関する基礎データの収集と部分的な解析を行っている。結果として、100日間を追跡の単位期間とし、67行以上の変更を大きな変更とした場合、次の2つの傾向が確認されている：(1) 保守に長い期間（今回の実験では400日を超える）をかけたとしても必ずしも安定したコードが得られるわけではなく、むしろ逆に安定しにくい傾向にある。(2) リリース後100日の間で完全に安定状態にあったコードは、その後で仕様変更が行われたとしても高い安定性を維持できる傾向にある。

On Source Code Stability Prediction in Open Source Development

HIROHISA AMAN†

This paper focuses on the stability of source code in open source development, and categorizes source code into four levels of stability. An empirical study is performed using a practical open source software, Eclipse, for collecting empirical data and analyzing them on source code stability. The empirical results show the following two tendencies: (1) a long-time maintenance (more than 400 days in this case) would produce a low-stability code; (2) a code that was stable during the first 100 days after the initial release, would keep its high-stability even if the external specification was changed.

1. はじめに

近年、ソースコードをインターネット等において公開し、その上で開発を進めていくオープンソース開発¹⁾が盛んに行われるようになってきている。この種の開発では、ネットワークを介して開発コミュニティを形成し、所属企業や組織等の枠を越えて開発を行うというスタイルがとられる。誰でもソースコードを入手・利用でき、なおかつ開発に参加できるという特徴から、高品質なソフトウェアを短期間に低コストで開発できる場合も多いとされている。しかしその一方、特定の企業や組織の管理下で開発が行われるわけではないため、そこに投入された資源や工数、コストといった管理データの収集が難しい²⁾。必然的に工数やコストの見積りも困難となる。そこで我々はソースコードの安定性に着目している。これは、ソースコードに対する開発及び保守作業が収束している状態をソースコードの“安定”と考え、安定性の評価と予測を定量的に行うことで工数やコストの見積りに役立てようとするものである。本稿は、安定性の評価と予測を行っていく上で重要となる基礎データの収集と解析を目的とする。

2. コードの安定レベル

本節では議論の準備として、CVS等のバージョン管理システム³⁾を用いて管理されているソースコードの安定状態に関する定義を行う。便宜上、以下ではJava言語で記述されたコードに限定する。

2.1 定義

これまで、ソフトウェアの設計特性をメトリクスを使って定量化し、バージョン間でその変化が小さい場合に“安定”と見なすといった研究が報告されている⁴⁾。これに対し本稿では、設計特性の変化のみならずソースコードそのものの変化についても考慮できるよう、より段階的に次の3種類の安定状態を定義する。

定義1 (外部安定 (安定レベル1))

あるクラスCのソースコードが与えられたとき、次のすべての要素が所定の期間内に不変ならばそのコードを“外部安定 (安定レベル1)”の状態にあると呼ぶ：

- C内で宣言されているコンストラクタの仮引数の数及び型とそれらの順序。ただし、非公開 (private) コンストラクタを除く。
- C内で宣言されているメソッドの名前、戻り値の型、仮引数の数及び型とそれらの順序。ただし、非公開メソッド並びに継承メソッドを除く。
- C内で宣言されているフィールドの名前及び型。

† 愛媛大学大学院理工学研究科
Graduate School of Science and Engineering, Ehime University

ただし、非公開 (private) フィールド並びに継承フィールドを除く。

- C のスーパークラス。
- C が実装しているインターフェース。

□

定義 2 (準内部安定 (安定レベル 2))

安定レベル 1 の条件を満たすソースコードのうち、所定の期間内にそのコードに対して施された変更の総量 (コード差分の量) が一定量未満のものを “準内部安定 (安定レベル 2)” の状態にあると呼ぶ。

なお、ここでいう変更量とは、変更前のコードから x 行を削除し、 y 行のコードを追加ないし挿入することで変更後のコードが得られるとした場合の $x+y$ を意味する*。一般には一定の期間内に複数回の変更が施されるため、ここでは各々での $x+y$ を考え、その合計値を総変更量とする。

□

定義 3 (内部安定 (安定レベル 3))

安定レベル 2 の条件を満たすソースコードのうち、所定の期間内に一切の変更が施されないものを “内部安定 (安定レベル 3)” の状態にあると呼ぶ。

□

外部安定 (安定レベル 1) は、他のクラスから見た外部仕様、すなわちメソッドの呼び出し方法及びフィールドの型、キャスト可能な型がそれぞれ不変に保たれている状態を意味する。準内部安定 (安定レベル 2) は、外部仕様が固まった後、ソースコードの変化もある程度の範囲に収まり、徐々に完全な安定状態へ移行しようとしている状態を意味する。そして、内部安定 (安定レベル 3) は、完全にソースコードが変更されなくなった、完全な安定状態を意味する。以下では特に断らない限り、安定レベル 2 ~ 3 の状態にあることをソースコードの “安定” と考える。

上述の 3 つのレベルに加え、いずれにも属さない非安定な状態も定義しておく。

定義 4 (非安定 (安定レベル 0))

あるクラスのソースコードが与えられたとき、それが外部安定の条件を満たさない (即ち、安定レベル 1, 2, 3 のいずれにも属さない) ものを “非安定 (安定レベル 0)” の状態にあると呼ぶ。

□

1 つのクラスのソースコードにおける安定レベルは、例えば 図 1 のようなイメージで時間とともに成長していくと考えられる。また、コードによっては、途中の過程で安定レベルが低下してしまう場合やレベル 0 から 3 へ急激に成長するような場合も考えられる。

2.1.1 複数のクラスが同一ファイルに含まれる場合

Java 言語の場合、複数のクラス宣言を 1 つのソースファイルにまとめて記述できるため、1 つのファイルに 1 つのクラスが対応するとは限らない。しかし

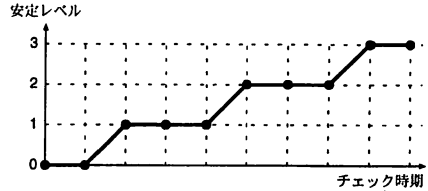


図 1 安定レベル成長のイメージ

Fig.1 Image of stability level growth.

ながら、1 つのファイル内で宣言可能な公開 (public) クラスは 1 つに限定されており、そのクラス名がソースファイル名と対応付けられている。例えば、Foo と Bar という 2 つのクラスが 1 つのファイルの中で記述されていたとしても、公開クラスとしてはそのいずれか一方のみが許される。いま、Foo が公開クラスであるとする、ファイル名はクラス名に対応した “Foo.java” でなければならない。このことは、ソースファイル “Foo.java” の主たる目的が公開クラス Foo の宣言及び定義であり、クラス Bar の内容は Foo を補助するものであると解釈できる。

そこで便宜上、各ソースファイルの修正はすべて公開クラスに対する修正と見なすことにする。なお、外部安定 (安定レベル 1) を評価するには、クラス内のメソッド宣言やそのクラスのスーパークラス宣言等に変化が起っていないかを調べることになるが、このチェックは公開クラスに対してのみ行うこととする。すなわち、公開クラス以外のクラスにおけるメソッド宣言等の変更は、公開クラスの外部仕様変更ではなく、そのコード修正の範疇に相当すると考える。

Java 言語では、クラスの内部に別のクラスを入れ子のかたちで宣言及び定義した “内部クラス” や同様のかたちでクラス名を与えない “無名クラス” を作成することができる。本稿では、内部クラス及び無名クラスに対する変更についても上述の場合と同様に公開クラスの外部仕様変更ではなく、そのコード修正の一種と見なす。

2.1.2 抽象クラス

抽象クラスはその子孫クラスに対して基本構造を提供するものであり、振る舞いの詳細は子孫クラスで決定される。それゆえ、抽象クラスの場合、“コードが修正されながら徐々に完全化されていく” という過程よりも、“仕様の確定に合わせて内容も定まり、その後は仕様変更が起こらない限り修正されることは無い” という場合も多いと推察される。つまり、コードの安定性を論じる上で抽象クラスのデータは例外となり得る可能性がある。そこで本稿では抽象クラスを分析の対象外とする。同様の理由により、インターフェースも対象外とする。換言すれば、本稿では具象クラスのみを分析対象とする。

* 例えば 1 行の内容が書き換えられた場合、その 1 行が削除され、その後で同位置に新しい内容の 1 行が挿入されたと解釈される。

表 1 BuildActionGroup.java の更新過程
Table 1 Upgrade process for BuildActionGroup.java

チェック日	更新数	変更量	主な変更内容
2002/04/17	—	—	新規開発
2002/07/26	8	75	公開メソッドの追加
2002/11/03	1	8	コメント文の修正
2003/02/11	0	0	—
2003/05/22	0	0	—
2003/08/30	0	0	—
2003/12/08	0	0	—
2004/03/17	1	12	メソッドの内容修正
2004/06/25	2	8	メソッドの内容修正
2004/10/03	0	0	—
2005/01/11	1	8	非公開フィールドの削除
2005/04/21	2	10	メソッドの内容修正
2005/07/30	1	2	コメント文の修正
2005/11/07	0	0	—
2006/02/15	0	0	—
2006/05/26	0	0	—
2006/09/03	0	0	—

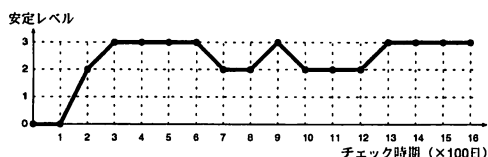


図 2 BuildActionGroup.java の安定レベルの変化
Fig. 2 Changes in the stability level of BuildActionGroup.java.

2.2 例

実例として、オープンソースソフトウェア “Eclipse”⁵⁾ に含まれているソースコードの 1 つについて、安定レベルの変化を以下に示す: “org.eclipse.jdt.ui.actions” パッケージの “BuildActionGroup.java” について、その更新過程を追跡した。なお、ここでは一例として、追跡期間を 100 日単位とし、コード変更量の閾値を 50 行 (50 行以上ならば安定レベル 2 と見なさない) とした。このコードは 2002 年 4 月 17 日に開発されたものであり、その 100 日後の 7 月 26 日までに 8 回の更新が行われ、公開メソッドの追加等を経て 75 行の変更が施された。その次の期間、即ち 2003 年 11 月 3 日までの 100 日間には 1 回だけコメント文の修正が行われ、その変更行数は 8 であった。これらを含め、2006 年 9 月 3 日までの更新内容を表 1 に示す。

表 1 より、このソースコードの安定レベルは図 2 に示す通りとなる。リリース後 1 回目のチェック日 (2002 年 7 月 26 日) までに多くの更新が施されているが、その後の変更量は最大でも 12 行と小さい。途中に 1 度だけ非公開フィールドの削除が行われているが、外部仕様の変更には至っておらず比較的安定した状態にあると考えられる。

3. 実験

本節では、実際に広く使われているオープンソースソフトウェアについて、その変更履歴に関するデータ収集を行い、コードの安定性について考察する。

3.1 実験対象

本稿ではオープンソースソフトウェア Eclipse のソースコードに対し、その変更履歴データの収集を行うこととした。Eclipse は、ソフトウェア開発技術者を中心に広く普及している統合開発環境であり、専用のプラグインも含めて多量のソースコードが CVS を使って管理され、公開されている。なお、ソースコードの記述言語は Java 言語である。

今回は 2006 年 10 月 1 日以前に開発された 8,427 個のソースファイルを実験の対象とした。ただし、この 8,427 個のソースコードというのが Eclipse の開発サイトで公開されているすべてではない。今回は時間の都合によりすべてを調査するには至らなかった (一部が未調査のままとなっている) ことを付記しておく。

3.2 データ収集

我々は次の手順でデータ収集を行った:

- (1) Eclipse の開発サイト⁶⁾ から各クラスのソースコードを入手する。
- (2) 同サイトから更新履歴情報を入手する。これには更新日時、版本号、変更行数 (追加行数と削除行数) が含まれる。
- (3) 更新履歴情報を解析し、各クラスのソースコードの初版から最新版までを順次入手する。
- (4) 各版のクラスのソースコードを解析し、版間での外部仕様の変更を調べる。
- (5) 各クラスについて、版ごとの版本号、更新日時、変更行数、外部仕様変更の有無をそれぞれデータベースへ格納する。

前節で述べたように、本稿では分析対象を具象クラスに限定している。結果として 5,814 個のクラスについて 97,906 件の更新データを収集できた。

以下ではデータ収集の各手順について概説する。

3.2.1 開発サイトからソースコードを入手

Eclipse 開発サイトでは CVS を使ってソースコードを管理及び公開している。今回は “org.eclipse.ant.ui” や “org.eclipse.core.tools” といった 55 個のパッケージ (付録参照) について、リポジトリからソースファイルをチェックアウトすることでそれぞれ最新版のソースコードを入手した。具体的には図 3 のように cvs コマンドを利用した。

```
cvs -d リポジトリ名 co パッケージ名
```

図 3 CVS におけるソースコードのチェックアウト
Fig. 3 Checkout of source code on CVS.

```

$ cvs log ./src/org/eclipse/core/tools/metadata/Dump.java

RCS file: /cvsroot/eclipse/org.eclipse.core.tools/src/org/eclipse/core/tools/metadata/Dump.java,v
Working file: ./src/org/eclipse/core/tools/metadata/Dump.java
head: 1.4
branch:
locks: strict
access list:
symbolic names:
    after-osgi-api-refactor: 1.2
    pre-osgi-api-refactor: 1.2
    v20050225: 1.2
    v20050221_pre_copyright_fix: 1.1
    v20041013: 1.1
    v20041012: 1.1
keyword substitution: kv
total revisions: 4;    selected revisions: 4
description:
-----
revision 1.4
date: 2006/07/14 15:24:51; author: johna; state: Exp; lines: +0 -9
delete unused code
-----
revision 1.3
date: 2006/05/10 18:54:19; author: dj; state: Exp; lines: +2 -2
Updated copyrights.
-----
revision 1.2
date: 2005/02/21 23:10:04; author: johna; state: Exp; lines: +8 -8
Convert copyrights from CPL to EPL
-----
revision 1.1
date: 2004/10/12 18:00:25; author: rchaves; state: Exp;
Moved metadata dumping fwk from org.eclipse.core.tools.resources into org.eclipse.core.tools

```

図 5 更新履歴情報の取得例

Fig.5 Example of change history acquisition.

```
cvs log ソースファイル名
```

図 4 ソースコードの変更履歴の入手

Fig. 4 Acquisition of source code change histories.

3.2.2 更新履歴の入手

入手した各ソースファイルについて、その更新履歴情報を入手した。これについても具体的には cvs コマンドを 図 4 のように利用した。

一例を 図 5 に示す。この例では、“Dump.java” というソースファイルが 2004 年 10 月 12 日に開発され、その後 2005 年 2 月 21 日に 8 行の追加と 8 行の削除（合計 16 行の変更）が行われている。同様に、2006 年 5 月 10 日、7 月 14 日にそれぞれ 4(= 2+2) 行、9(= 0+9) 行ずつ変更されていることが分かる。

3.2.3 各クラスの初版から最新版までを入手

各ソースファイルについて、図 5 のように更新履歴情報を入手すると、それまでの版本号（図 5 の例では “revision”）を抽出できる。この情報をもとに初版から最新版までの各版のソースコードを入手した。各

```
cvs -r 版本号 ソースファイル名
```

図 6 ソースコードの入手（版指定）

Fig. 6 Acquisition of a specified version of source code.

版のソースコードは 図 6 のように cvs コマンドを使用することで入手可能である。

3.2.4 外部仕様変更の調査

各ソースファイルについて、ある版と次の版とを比較し、その間で外部仕様に変更が起っていないか調査した。ここでいう外部仕様の変更とは、定義 1 で示した外部安定の条件が満たされないことをさす。この調査結果と変更行数を用いることで、安定性（レベル 0～3）の評価が可能となる。

外部仕様の変更を調べるには、Java ソースコードの構文解析が必要となる。今回は Java ソースコードをいったん JavaML⁷⁾へ変換し、JavaML を解析することでその調査を行った。JavaML とは XML の一種で、Java ソースファイルのためのマークアップ言語である。JavaML ではメソッドやフィールドの宣言

がそれぞれ専用のタグ “<method> ~ </method>” や “<field> ~ </field>” に対応しているため、外部仕様変更の検査では特定のタグを調べればよい。

3.2.5 データベースへの格納

各ソースファイルについて、次の 4 項目をデータベースへ格納した：

- ソースファイル名
- 公開クラス名
- クラスであるかインタフェースであるか
- 抽象クラスであるか具象クラスであるか

さらに各ソースファイルの各版について、次の 5 項目もデータベースへ格納した：

- 版番号
- 更新日時
- 追加行数
- 削除行数
- 外部仕様変更の有無

次節より、以上の情報を分析し、ソースコードの安定性について検討する。

3.3 安定性を論じる単位期間

ソースコードの安定性を調査していく上で、調査の単位期間、すなわち“どれだけの期間を単位としてソースコードの更新内容をチェックしていくのか”を決定しておく必要がある。単位期間が短いと、その期間内に更新が起らなかった場合にそれがソースコードの安定を意味するのか、それとも単に期間が短すぎるだけなのか区別が難しい。それゆえ、平均的に複数回の更新が期待されるような期間を単位期間にすべきと考える。本実験において収集したデータのうち、更新間隔の分布及び代表的な統計量をそれぞれ図 7 及び表 2 に示す。ただし、期間の粒度は“日”単位とする。なお、図 7 は更新間隔が 400 日未満のもののみを示している。400 日以上の場合は極めて少数の事例しか存在しなかったため図 7 では割愛した。表 2 の統計量には同図で割愛した部分も含まれている。

図 7 及び歪度 = 3.21（表 2）からも分かるように、更新間隔の分布は非対称で右裾の長いものとなっている。これはソフトウェア開発に関する多くの定量データに見られる傾向である⁸⁾。

単位期間を決定するため、まずは 1 回の更新が期待される更新間隔を求める。いま、更新間隔を確率変数 X とし、ある更新から X 日後に次の更新が起こる確率を $P(X)$ とする。このとき、更新間隔の期待値 $E(X)$ は

$$E(X) = \sum_{x=0}^{\infty} xP(x) \quad (1)$$

となる。更新間隔について真の確率分布が得られない限り上式による期待値の算出は不可能であるが、今回

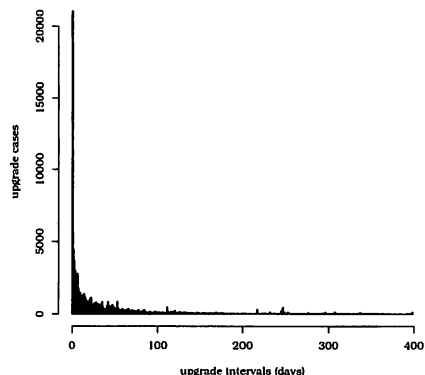


図 7 更新間隔（日数）の分布（400 日未満のみ）

Fig. 7 A part of the upgrade intervals (days) distribution (limited to less than 400 days).

表 2 更新間隔（単位：日）に関する統計量

Table 2 Statistics about upgrade intervals (days).

最小値	第 1 四分位数	中央値	第 3 四分位数	最大値
0	2	14	53	938
平均値		標準偏差	歪度	
50.1		88.6	3.21	

は図 7 に示した件数の分布*を経験的確率に見立てて期待値を算出する。(1) 式の値は算術平均に等しく、更新間隔の期待値は 50.1 日（表 2 参照）となる。

よって、本稿では複数回の更新が期待される 100 日（ $\approx 50.1 \text{ 日} \times 2 \text{ 回}$ ）を単位期間として用いる。

3.4 変更行数の閾値

次に準内部安定（安定レベル 2）と外部安定（安定レベル 1）との境界となる変更量の閾値を決定する。この閾値が低すぎると些細な変更でも外部安定状態と見なされてしまい、逆に高すぎると大幅な変更であっても（外部仕様が不変ならば）準安定状態と見なされかねない。それゆえ適度な閾値が必要である。

本実験において収集したデータより、1 回の更新における変更行数の分布及び代表的な統計量をそれぞれ図 8 及び表 3 に示す。なお、ここでも更新間隔の場合と同様にデータの一部を割愛して図示している：図 8 は変更行数が 150 行未満のもののみを示している。150 行以上の変更は極めて少数の事例しか存在しなかったため図 8 では割愛しているが、表 3 の統計量には同図で割愛した部分も含まれている。やはり更新間隔の場合と同様に変更行数の分布非対称で右裾の長いものとなっている（図 8 参照；歪度 = 29.5）。

3.3 節での議論と同様にして、1 回の更新で期待される変更行数は 33.3 行ということが導かれる。いま、本稿では 2 回の更新が期待される期間を単位期間とし

* これは、割愛した 400 日以上の部分も含めた上での議論である。

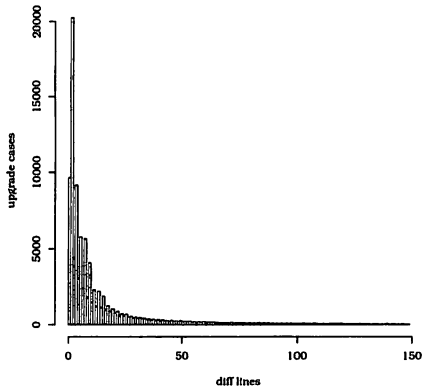


図 8 変更行数の分布 (150 行未満のみ)

Fig. 8 A part of the changed lines distribution (limited to less than 150 lines).

表 3 変更行数に関する統計量

Table 3 Statistics about changed lines.

最小値	第 1 四分位数	中央値	第 3 四分位数	最大値
0	2	6	18	14900
平均値	標準偏差	歪度		
33.3	171	29.5		

ているため、単位期間内に 67 行 (≈ 33.3 行 $\times 2$ 回) の変更が施されると期待される。それゆえ本稿では、単位期間内に施された総変更行数が 67 行未満であれば“平均以下の変更量”と判断し、準内部安定 (安定レベル 2) の状態にあると見なす。逆に 67 行以上の変更が施されていた場合は、より下の安定レベルである外部安定 (安定レベル 1) の状態にあると見なす。ただし、いずれの場合も外部仕様が不変であった場合に限る。言うまでもなく、外部仕様に変更が起こった場合は非安定 (安定レベル 0) の状態となる。

3.5 データ解析

前述したように、100 日間を単位としてソースコードの更新を追跡した。そして、各単位期間内での安定性をレベル 0～3 (非安定, 外部安定, 準内部安定, 内部安定) に分類した。レベル 1 (外部安定) とレベル 2 (準内部安定) との境界は、総変更行数が閾値 (= 67 行) 以上であるか否かとした。今回の収集データから、100 日間を単位とすることで 65,983 件の安定性評価データを得ることができた。

まず、収集した 5,814 個のクラスのうち、2006 年 10 月 1 日現在で 1 度も内部安定 (レベル 3) に達することの無かったものが 835 個あった。また、最後のチェック時に初めてレベル 3 に到達したためその後の動向を観察できなかったクラスが 549 個だけ存在していた。これら 1,384 (= 835 + 549) 個に対する解析は今後の課題として対象外とし、以下では残りの 4,430 個のクラスについて解析を行う。

表 4 内部安定 (レベル 3) に初めて到達した時期

Table 4 The first times to be stability level 3.

時期	チェック期間	クラス数
1	リリース後 1～100 日	885
2	101～200 日	1,609
3	201～300 日	844
4	301～400 日	303
5	401～500 日	244
6	501～600 日	183
7	601～700 日	78
8	701～800 日	95
9	801～900 日	49
10～	901～ 日	140

3.5.1 内部安定 (レベル 3) に到達する時期

各クラスについて、リリース後に初めてレベル 3 に到達した時期、すなわち何回目のチェック時にレベル 3 に分類されたのかを表 4 に示す。

最も多いのが第 2 回目のチェック時 (リリース後 101～200 日目が対象) であり、1,609 個のクラスがこれに該当した。次いで第 1 回が 885 個、第 3 回が 844 個、第 4 回が 303 個、第 5 回が 244 個、第 6 回が 183 個となっており、最初の 6 回まで (リリース後 600 日目まで) で全体の約 9 割 (91.8 %) を占めていた。すなわち、リリースされてから 600 日目までに約 9 割のコード*はレベル 3 へ到達していたことになる。同様に、約半数 (56.3%) のコードはリリース後 200 日目までにいったんはレベル 3 の状態へ到達していることが分かる。

3.5.2 内部安定 (レベル 3) 状態の持続性

内部安定 (レベル 3) の状態にあるコードが、その後も同状態を保持しているかどうかについて解析を行った。表 5 に各チェック時期及びその時期に初めてレベル 3 へ分類されたクラス数 (= a)、それ以降もレベル 3 であり続けたクラス数 (= b) とその割合 (= b/a) をそれぞれ示す。なお、本実験では 1 回以上の更新が施されているコードのみを分析対象として収集しているため、第 1 回目のチェック時期にレベル 3 として分類されたコードの中でその後もレベル 3 を持続するというケースはデータとして収集していない**。

それゆえ表中では“—”としている。レベル 3 という完全な安定状態が長期に渡って持続するという事象は決して起こりやすいものではないが、表 5 から第 2～4 回目のチェック時期 (リリース後 200～400 日) において安定状態に到達したコードは約 10% の割合でその後もレベル 3 を持続できていることが分かる。一方、より後半になってレベル 3 に到達したコードに着目すると、第 7 回が 7.69% となっているが、その他はいずれも 5% 未満であり、第

* 上述の対象外コードも含めると約 65 %

** そのままレベル 3 を維持することは、初期リリースを最後として更新回数が 0 であることを意味する。

表 5 内部安定（レベル 3）の持続性
Table 5 Continuousness of stability level 3.

時期	レベル 3 の クラス数	その後もレベル 3 で あり続けたクラス数	割合
1	885	—	—
2	1,609	183	11.4%
3	844	99	11.7%
4	303	32	10.6%
5	244	12	4.92%
6	183	7	3.83%
7	78	6	7.69%
8	95	4	4.21%
9	49	1	2.04%
10 ~	140	3	2.14%

表 6 内部状態（レベル 2, 3）の持続性
Table 6 Continuousness of stability level 2 or 3.

時期	レベル 3 の クラス数	その後もレベル 2/3 であり続けたクラス数	割合
1	885	0	0%
2	1,609	748	46.5%
3	844	365	43.2%
4	303	140	26.2%
5	244	62	25.4%
6	183	59	32.2%
7	78	35	44.9%
8	95	21	22.1%
9	49	10	20.4%
10 ~	140	47	33.6%

2 ~ 4 回に比べて割合は半減している。確認のため、第 2 ~ 4 回のチェック時にレベル 3 へ到達したコードがその後もレベル 3 を維持できている割合を r_1 、第 5 回以降における同様の割合を r_2 とおき、“ $r_1 = r_2$ ”を帰無仮説、“ $r_1 > r_2$ ”を対立仮説とした片側検定を実施した。その結果、 p 値は 9.28×10^{-10} となり、統計的にも有意な差を確認できた。

このことから、時間をかけて保守されていたコードが必ずしも安定しているわけではなく、むしろ保守に時間がかかりすぎるとコードも安定しないという傾向が見てとれる。

3.5.3 安定状態（レベル 2, 3）の持続性

3.5.2 節ではレベル 3 の持続性について調べた。ここではその条件を緩和し、“レベル 2 ~ 3 の状態で持続しているか”、すなわち、“大きな変更（レベル 1）や外部仕様の変更（レベル 0）は起こっていないか”についても調べた。結果を表 6 に示す。

表 6 から分かるように、いったん内部安定（レベル 3）の状態になると、その後も比較的高い割合でレベル 3 若しくはレベル 2（準内部安定）の状態を維持できている。例えば、第 2 回のチェック時（リリース後 200 日目）にレベル 3 へ到達したコードは、その後も 46.5% の割合でレベル 2 ~ 3 を維持できている。ここでも表 5 ほど顕著ではないが、第 2 ~ 3 回目のチェック時にレベル 3 に到達したコードは、その

他に比べて比較的安定しやすい傾向が見てとれる。これについても 3.5.2 節と同様に統計的検定による確認を行った。すなわち、第 2 ~ 3 回のチェック時にレベル 3 へ到達したコードがその後もレベル 2 ~ 3 を維持できていた割合を r_1 、第 4 回目以降における同様の割合を r_2 とし、3.5.2 節の場合と同じ仮説検定を実施した。その結果、 p 値は 1.11×10^{-15} となり、ここでも統計的に有意な差を確認できた。

また、表 6 について特筆すべき点として、第 1 回のチェック時（リリース直後から 100 日目までが対象）にレベル 3 とされたコードの安定性の変化が挙げられる。これらのコードのうち、101 日目以降もレベル 2 以上を維持できたものは皆無となっている。すなわち、大きな変更（レベル 1）や外部仕様の変更（レベル 0）が必ず施されていたことを意味する。この点について、さらに詳しい調査を行った。

3.5.4 リリース直後から 100 日間は内部安定（レベル 3）であったコードについて

上述したように、第 1 回のチェック時（リリース後 100 日目）までに一切の更新が行われなかった、つまりレベル 3 の状態にあったコードが 885 個存在していた。そしてその後、それらはすべてレベル 1 以下の不安定な状態へと移行していた。追跡調査を行ったところ 885 個すべてのコードにおいてレベル 3 からレベル 0 への移行が見られた。つまり、最初のレベル 3 の状態から安定性が崩れるのは、いずれのコードの場合も外部仕様の変更によるものであった。一例として“org.eclipse.core.tools”パッケージの“BaseTextView.java”の場合を図 9 に示す。図 9 のように 1 回目のチェック時にレベル 3 となるも、その後はレベル 0 へ直接落ちるという現象が確認された。コードによっては最初のレベル 3 の状態がしばらく持続することはあるが、いずれの場合も崩れる時にはレベル 0 まで直接落ちている。これは議論の対象となっている 885 個すべてに対して成り立つことが確認できた。つまり、いずれのコードも外部仕様の変更が起こるまでは完全な安定状態を維持できていた。

また、レベル 0 へ安定性が下落した後、再びレベル 2 ~ 3 の安定状態へ回復するケースは多く、レベル 0, 2, 3 のいずれかのみで推移していくコードは 799 個（全体の 90.3%）であった。ゆえに、このようなコー

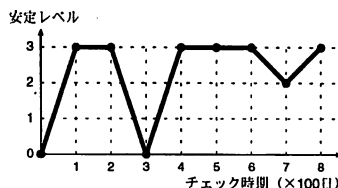


図 9 BaseTextView.java の安定レベルの変化
Fig. 9 Changes in the stability level of BaseTextView.java.

ドは安定しやすい傾向にあると考えられる。

3.6 考 察

今回のデータ解析によって得られた知見とそれらに対する考察を以下に述べる。

- リリースの後、比較的早期（200～400日程度）に内部安定（レベル3）の状態となったコードは、その後も安定状態を維持しやすいことが見てとれた。逆に、安定状態に到達するまでの時間が長くなると、その後の安定性も低い傾向にあった。このことから、保守に長い期間（今回の実験では400日を超える期間）をかけたとしても必ずしも安定したコードが得られるわけではなく、むしろ逆に安定しにくい傾向にあると考えられる。
- リリース後100日の間に一切修正されなかったコードは、外部仕様の変更が起らない限り、その後も内部安定状態（レベル3）を維持していた。また、外部仕様の変更が施された後もレベル2～3の安定状態へ回復しやすい傾向にあった。このようなコードは、リリースと同時に安定していることから、高い信頼性を有しているといえよう。さらに、外部仕様の変更後も安定していることから保守性も高いと考えられる。それゆえ、リリース後100日の間で完全に安定していたようなコードは、仮に仕様変更が行われたとしても、高い安定性を維持できる傾向にあると考えられる。

4. おわりに

本稿ではオープンソース開発におけるソースコードの安定性に着目し、その状態を非安定（レベル0）、外部安定（レベル1）、準内部安定（レベル2）、内部安定（レベル3）の4段階に分類した。そして、実際のオープンソースソフトウェアEclipseについて測定実験を行い、安定性に関する基礎データの収集と部分的な解析を行った。100日間を追跡の単位期間とし、67行以上の変更を大きな変更とした場合、次の2つの傾向が見てとれた：(1) 保守に長い期間（今回の実験では400日を超える）をかけたとしても必ずしも安定したコードが得られるわけではなく、むしろ逆に安定しにくい傾向にある。(2) リリース後100日の間で完全に安定していたようなコードは、その後に仕様変更が行われても安定性を維持できる傾向にある。

今後、より多くのソースコードについてデータ収集及び解析を進め、各種メトリクスによる測定・解析も行いながらソースコードに対する安定性予測法の開発を行っていく。また、これに関連して仕様変更やリファクタリングによる影響、コードクローンによる影響についても解析していく予定である。

謝辞 本研究の一部は「プロジェクト定量分析に関するテーマ型調査研究」として独立行政法人 情報処理推進機構ソフトウェア・エンジニアリング・センター

による援助を受けている。ここに感謝の意を表す。

参 考 文 献

- 1) 可知 豊：オープンソースのことがわかる本，日本実業出版社，東京（2005）。
- 2) 加藤みどり：企業戦略としてのオープンソース，技術報告，科学技術庁科学技術政策研究所 Discussion Paper（2000）。
- 3) Karl Fogel（竹内里佳訳）：CVS —バージョン管理システム—，オーム社，東京（2000）。
- 4) Kelly, D.: A Study of Design Characteristics in Evolving Software Using Stability as a Criterion, *IEEE Trans. Software Eng.*, Vol.32, No.5, pp.315-329（2006）。
- 5) Eclipse.org: <http://www.eclipse.org/>.
- 6) Eclipse 開発サイト: <http://dev.eclipse.org/>.
- 7) Badros, G.: JavaML: A markup language for Java source code, *Proc. 9th Int'l World Wide Web Conf.*（2000）。
- 8) 独立行政法人情報処理推進機構ソフトウェア・エンジニアリング・センター：ソフトウェア開発データ白書 2006，日経 BP 社，東京（2006）。

付 録

A.1 実験に使用したパッケージ

実験に使用したパッケージの一覧を以下に示す。

- org.eclipse.ant.{core, ui}
- org.eclipse.compare
- org.eclipse.core.{applicationrunner, commands, contenttype, expressions, filebuffers, filesystem, jobs, resources, runtime, tools, variables}
- org.eclipse.debug.{core, ui}
- org.eclipse.equinox.{app, common, device, ds, event, http, log, metatype, preferences, registry, supplement, useradmin, wireadmin}
- org.eclipse.help
- org.eclipse.help.{appserver, base, ui}
- org.eclipse.jdt.core
- org.eclipse.jdt.core.manipulation
- org.eclipse.jdt.debug
- org.eclipse.jdt.junit
- org.eclipse.jdt.junit.runtime
- org.eclipse.jdt.junit4.runtime
- org.eclipse.jdt.launching
- org.eclipse.jdt.ui
- org.eclipse.jface
- org.eclipse.jface.{databinding, snippets, text}
- org.eclipse.ltk.core.refactoring
- org.eclipse.ltk.ui.refactoring
- org.eclipse.osgi
- org.eclipse.pde
- org.eclipse.pde.{build, core, junit, junit.runtime, runtime, ui}

なお、いずれのパッケージについても、リポジトリ名は“psrver:anonymous@dev.eclipse.org:/cvsroot/eclipse”である。