卒 業 研 究 報 告 書

題　目　　オープンソースソフトウェア開発における
版更新と不具合出現の関連分析

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　　07122005

氏　　名　　出原　真人

平成23年2月15日提出

オープンソースソフトウェア開発における版更新と不具合出現の関連分析

平成 23 年 2 月 15 日　　07122005　出原　真人

概　要

ソフトウェア開発において不具合を出さないことは不可能と言っても問題ないほど難しいことである．不具合の発生はスケジュール遅延，動作不良などソフトウェア開発に数々の弊害を引き起こす．そのため，ソフトウェア開発においては第 1 に不具合の発生数を抑えること，第 2 に不具合の早期発見と修正が重要となる．不具合を発見する為の研究としては，ソフトウェアメトリクスという概念を用いてソフトウェアを定量的に測定し，測定したメトリクスがどのように不具合と関係しているかを分析するものが盛んである．

ソースコードの行数や大きさ，発見されたバグの数や開発言語の種類などもメトリクスである．これらのように測定が簡単なものを始めとして，測定に大規模な準備が必要な難しいものまで様々なメトリクスが提案されては分析されている．

本研究では版更新の情報から簡単に測定できるメトリクスを提案し，いくつかのオープンソースソフトウェアの開発プロジェクトのデータに対して，提案したメトリクスを計測して不具合出現との関連性を分析する実験を行った．分析の結果，不具合の発生率とモジュールの安定性，不具合の持続性について版更新との間の関連性を冪乗則を用いてモデル化することが出来た．

本研究の成果を利用することで，簡単に計測できるメトリクスからソフトウェアモジュールに不具合が残存する期間をモデル化することができる．これにより，不具合の残存や出現に関して新たなモデルを提案することも可能になると期待している．

目　次

1. 緒言

ソフトウェア開発において，不具合の発生は避けては通れない道であり，できるだけ不具合の発生を回避する，または発生した不具合の早期発見し迅速に対処する術は貴重な財産となる．不具合が発生しそうなモジュールの予測や，起こりうる不具合の総数を予測することができれば効率の良いデバッグが可能になる．それによってソフトウェア開発工数の削減や品質の向上に繋がる．

本研究では版更新の観点からメトリクスを定義し，不具合の出現との関連分析を行う．また，可能ならばその関連性について何らかのモデル化を行う．研究対象となるのは，バージョン管理システム，バグ管理システムで管理された 5 つのオープンソースソフトウェアの開発データである．分析するプロジェクトのデータは，ソフトウェア開発のリポジトリから SZZ と呼ばれるアルゴリズムを用いてデータを取得し，SQLite3 の形式のデータベースに纏めたものを使用する [1]．分析の結果，不具合出現に関係するいくつかの興味深い知見を得ることが出来た．具体的には，次の 2 つの仮説が正しいか否かについての分析を行うため，必要なメトリクスを定義し，計測した：(1) 版数の増加に伴って欠陥の新規発生率は下がる．(2) 欠陥モジュール，非欠陥モジュールの生存率は版数でモデル化できる．5 つのプロジェクトから得られたデータに対して，これたの仮説を検証したところ，仮説 (1) については，一部のプロジェクトのみで成立するにとどまったものの，仮説 (2) はほとんどのプロジェクトで成り立つことが実証された．

最後に，本報告書の以降の構成を説明する．第 2 章で本研究の分析対象となるオープンソースソフトウェアと，それを管理しているバージョン管理システムの基本的な意味を説明する，第 3 章ではソフトウェアメトリクスを用いて不具合のあるモジュールを予測する方法について，代表的なソフトウェアメトリクスを説明してから，予測モデルを構築，検証する方法について説明する．第 4 章では研究目的と示すべき仮説，定義したメトリクス等を説明する．第 5 章では定義したメトリクスを用いて実際にオープンソースソフトウェアプロジェクトの解析を行う．第 6 章では実施した実験に対する考察を行い，第 7 章で本研究のまとめと今後の課題を述べる．

2. オープンソースソフトウェアとバージョン管理システム

ここでは，オープンソースソフトウェアとバージョン管理システム，その関連性について説明する．

2.1 オープンソースソフトウェアとは

オープンソースソフトウェアとは，広義の意味ではソースコードが公開されているソフトウェアのことを指す．この中には非営利活動では無償だが，商用利用する場合は有償になるような疑似オープンソースソフトウェアも含まれる．狭義の意味では OSI(Open Source Initiative) の提唱するオープンソースの 10 の定義を満たしているソフトウェアのことを指す．本研究では「オープンソースソフトウェア」をこの狭義の意味を満たすものとして扱う．

OSI の提唱するオープンソースの 10 の定義を以下に示す．

1. Free Redistribution：ソースコードや実行ファイルの再配布の自由
2. Source Code：ソースコードと実行ファイルの公開
3. Derived Works：派生ソフトフェア作成の自由
4. Integrity of The Author's Source Code：作者のソースコードの完全性の保持
5. No Discrimination Against Persons or Groups
   ：個人やグループに対する差別の禁止
6. No Discrimination Against Fields of Endeavor
   ：使用する分野に対する差別の禁止
7. Distribution of License：ライセンスの継承
8. License Must Not Be Specific to a Product
   ：特定製品のみで有効なライセンスの禁止
9. License Must Not Restrict Other Software
   ：ほかのソフトウェアに干渉するライセンスの禁止
10. License Must Be Technology-Neutral ：テクノロジーニュートラルなライセンス

オープンソースソフトウェアの利用者はこの定義によって，誰でも平等に，使用目的を問わず，あらゆる方法でソースコードやバイナリの提供が保証されている．

オープンソースソフトウェアの利点の一つとして，ソースコードが公開されているという特徴から，ユーザはそれぞれが開発者となることができるというものがある．複数人の開発者が協力して，参加者を限定せず，各々の独自性を尊重して開発を進める手法にバザール方式というものがある．バザール方式で開発された代表的なオープンソースソフトウェアとしては Linux や Netscape などがあげられる．

本研究で分析するオープンソースソフトウェアは，実験手順との関係から，バージョン管理システム・バグ管理システムを使用したソフトウェア開発のプロジェクトとする．

2.2 バージョン管理システム

オープンソースソフトウェアを含むほとんどのソフトウェアの開発にはバージョン管理システム (Version Control System) が使用されている．ここではバージョン管理システムが提供する機能と種類について説明する．

バージョン管理システムとは

バージョン管理システムは，ファイルの変更履歴を管理するシステムである．

ファイルに変更がある度に，編集者や変更内容などの変更に付随する情報をデータベースに記録している．このデータベースのことをリポジトリと呼び，普通，1 つのソフトウェア開発プロジェクトごとに 1 つのリポジトリを作成して管理を行う．バージョン管理システムには大別して分散型と集中型の 2 種類が存在する．

バージョン管理システムを使うメリット

バージョン管理システムは，ソフトウェア開発に以下のようなメリットを提供できる．

- プロジェクト全体に Undo 機能を提供する
  いつでも巻き戻すことができるので安心して作業できる．
- 複数の開発者が同じコードを基に作業できる
  ロック方式の場合，他者の編集中に編集可能になるという事はない．

マージ方式の場合，どちらかの作業が完全に失われて無駄になるという事はない．

- 変更の記録を保存する
  いつ，誰が，変更を行ったのかがすぐわかる．適切なコメントをつけていれば何のために変更を行ったのかも分かる．バグ管理システムも導入することで，さらに詳しい情報が残せるようになる．
- メインの開発とともに複数のリリースをサポートできる
  リリース前の期間でも開発を止めずにどんどん進めていくことができる．

バージョン管理システムはプロジェクト全体におけるタイムマシーンのようなものである．ある日付の状態に戻したい，そこからさらに進めて経過をみたい，とある地点での正確な情報を知りたい等の開発者側の要望を叶えることができる．

複数人が同時に作業しても，基本的にはどちらの作業も反映されるというのも強い機能である．これにより，いくら人数が増えても他の人の作業を気にすることなく開発がすすめられる．

集中型のバージョン管理システム

リポジトリはサーバで管理されている．そのため，リポジトリを使った作業を行う場合はサーバと通信可能な環境が必要である．

ユーザがリポジトリのファイルを編集したい場合は以下の手順を踏む．

1. ファイルをサーバからチェックアウトする
2. ローカル環境で編集する
3. 編集した結果をリポジトリにコミットする

集中型はクライアント・サーバ型とも呼ばれる．

以下にいくつかの集中型のバージョン管理システムを紹介する．

CVS（Concurrent Versions System）

- 1990 年代から使われているバージョン管理システム．
- ファイルごとにバージョン管理され，リビジョン番号も独立してつけられる．そのため，ファイルの履歴はわかるがリポジトリの履歴を調べることは難しい．

- ファイルの圧縮をしないので，リポジトリが大きくなる傾向がある．
- ファイル名やディレクトリ名の変更削除が上手く扱えない．
- アトミック・コミットをサポートしていない．

SVN（Subversion）

- CVS の欠点を改善して開発されたバージョン管理システム．
- コミットごとにリビジョン番号が割り当てられる．変更情報はリビジョンごとに 1 つのファイルに纏められている．
- ディレクトリの移動や削除をサポートしている．
- アトミック・コミットをサポートしている．

分散型のバージョン管理システム

リポジトリはサーバで管理されているが，チェックアウトによってユーザの作業ディレクトリに複製することができ，それに対して作業をすることができる．サーバにリポジトリを複製するとき以外はネット環境が無くてもコミット等の操作が可能である．

ユーザがサーバのリポジトリに対して作業したい場合は以下の手順を踏む．

1. リポジトリを作業ディレクトリに複製する（clone）．
2. 作業ディレクトリのファイルに対して変更を加える．
3. コミットする（ネット環境がなくても可能）．
4. 作業ディレクトリのリポジトリをサーバに複製する（push）．

以下に分散型のバージョン管理システムの 1 つである Git を紹介する．

Git

- Linus Torvalds によって開発された．Linux のカーネル管理にも Git が使われている．
- コミット毎にリビジョン番号が作られる．ファイルは commit,tree,blob,tag の 4 つのオブジェクトファイルで管理される．
- ファイルの移動，ファイル名の変更を自動検出できる．

- ファイルが増えると，古くなったいくつかをパックしてリポジトリの容量を小さくする．
- 大きなプロジェクトにおける処理の高速化が行われている．
- ノンリニアな開発を強力にサポートできる（ブランチやマージが高速）．

3. ソフトウェアメトリクスを用いた不具合モジュールの予測

この章では，一般的に用いられているソフトウェアメトリクスと, それを使用して不具合モジュールを予測する方法について説明する.

3.1 ソフトウェアメトリクスとは

ソフトウェアメトリクスは，1970 年代後半頃から提唱され始めた概念であり，具体的には，ソースコードの規模，複雑さ，保守性などの「属性」の情報を定量的に示すことができる指標である．早初期ではプログラムのステップ数やバグの数がこれにあたる.

実際に開発中に測定することが困難なメトリクスも存在するが，そういったメトリクスは新規開発よりもソフトウェアの保守の時に測定，利用される.

最近では，ソフトウェアメトリクスを用いて，ソフトウェアの品質の測定や，誤り傾向の強いモジュールを判別する方法について研究されている.

3.2 ソフトウェアメトリクスの種類

ここでは，今までに研究されてきたソフトウェアメトリクスのうちのいくつかについて説明する.

- LOC（lines of code：コード行数）
  ソフトウェアの規模を表すメトリクスで，ソースコードの論理的・物理的行数を表す．最も古いソフトウェアメトリクスの一つである．コピー＆ペーストで簡単に増える，使用するプログラミング言語によって LOC が大きく変わる，仕様が同じでも使用するアルゴリズムによって LOC が変化するなどの問題がある為，コスト指標としては正しくても実際の規模を測るのは難しい.
- Defect Density（欠陥密度）
  ソースコードに存在したバグの数/KLOC と計算し，ソースコードに存在するバグの密度を表す．プロジェクトの品質の測定などに使われる.

- Cyclomatic Complexity[2]（循環的複雑度）

  T. McCabe が開発したソフトウェアの複雑性を測るメトリクスで，プログラムの制御フローをグラフとして描くことで計算できる。この値はソースコード中の独立なパスを表しており，これが大きいと凝集度が低く複数の機能を詰め込もうとしていることが多いと判断できる．ただし，モジュールの種類によっては必然的に大きくなることもあるので注意が必要である．また，制作者自身が複雑度を定義していないため，猜疑的な声も多い．

- Function Point[3]（ファンクションポイント：機能規模）

  1979 年に A.J.Albrecht が考案したソフトウェアの規模を測定するためのメトリクスである．先ず，計測したいソフトウェアについて５種類のファンクション数を調べてポイントを振って基準値を出す．次にシステム特性について複雑さを調べて６段階に評価し，調整値を算出する．この基準値と調整値を用いることでファンクションポイントが算出できる．ファンクションポイントはソフトウェアの工数を予測できるメトリクスであり，実際にコーディングを始める前の仕様書が固まった段階で適用することができる．

- Code Churn[4]（コードチャーン）

  ソフトウェアの変更の度合いを表すメトリクスである．２つのリビジョン間で単純にコード量の差を測るのではなく，実際に追加，変更，削除された量を計測することでソースコードの変化量と不具合との関連を調べたものである．それをもとに Nagappan が RelativeCodeChurn[5] を提案し，コード行数以外にもいくらかのメトリクスをあげ，それらの絶対測量を相対測量にすることで，欠陥密度と深く関連していると発表している．

  また，複雑さのメトリクスで測定を行った CodeComplexityChurn，デバッグに関連するメトリクスで測定を行った DebugCodeChurn[6]，開発に関連するメトリクスで測定を行った DevelopmentCodeChurn 等の複数のコードチャーンが提唱されている．

3.3 ソフトウェアメトリクスを用いた予測モデルの構築と検証

この節では，前節で説明したメトリクス等を用いて目的の変数を予測，検証する方法について説明する．

予測モデルの構築

1. 予測したいメトリクス（欠陥の数や欠陥密度など）を決める．
2. ソフトウェア開発プロジェクトのデータから必要なメトリクスを計測する．
3. 予測したいメトリクスを目的変数として，必要に応じて回帰分析，重回帰分析を行って予測モデルを得る．この時，目的変数と相関が低いと判断されたメトリクスは削除される．

上記3. のフェーズで説明変数として選ぶのは予測したいメトリクスを除いた残りのメトリクスであるが，PCA（Principal Component Analysis，主成分分析）を行って抽出した主成分を使う場合もある．この場合，抽出された主成分はただの数学的な値であることに十分注意が必要である．

予測モデルの検証

得られたモデル式を利用して目的変数の予測値を算出する．この予測値と実測値の間で相関関係を調べることで，得られたモデル式がどの程度フィットしているのかを調べることができる．

相関係数 R は-1 から 1 の間で求められるが，モデルフィットを確かめるときはこの値を 2 乗した R^2 値を使う．この値が 1 に近ければ近いほど良い．使用されていないメトリクスによって高い相関関係が引き起こされ，全く相関の無い 2 変数が相関しているように見えている（疑似相関）可能性があるため，たとえ相関が高かったとしても無相関の可能性があるということに注意しておく必要がある．

判別分析を用いた予測

求めたい事象が欠陥密度のように連続的な変数ではなく，欠陥が有るか無いかのように状態的なものの場合は判別分析等の分析手法を用いる．入力には予測モデルで求めた予測値や，計測したソフトウェアメトリクスなどが考えられる．

4. 版更新と不具合出現の関連

この章では，本研究の研究目的と示すべき仮説，そして，その仮説を実証する手順について説明する．

4.1 研究の目的

ソフトウェアプロジェクトと関係していそうなメトリクスについては，前章で述べたメトリクス以外にも様々なものが提唱されている．欠陥数の予測に関するメトリクスも例には漏れないが，計測が大変であったり高度な統計的知識を必要とするものも多い．

本研究ではもっと単純な，簡単に計測できるメトリクスとしてソースコードの版数を提案する．実際にオープンソースのソフトウェアプロジェクトを分析して，ソフトウェアプロジェクトの不具合出現の情報とソースコードの版数との間に相関関係があるかを調べることが，本研究の目的である．

4.2 計測したメトリクス

この節では，分析するにあたって定義したメトリクスについて説明する．

- Renew number（RN，更新回数）
  バージョン管理されている連続したソースコードについて，それソースコードが何回更新されているかを表す．リビジョン番号と似ているが，RN は初めてソースコードのサイズが 0 ではなくなったリビジョンを 1 としてそこから数え始めるという点でリビジョン番号と異なる．これは，とりあえずファイルだけ作ってコミットされたソースコードと少し作ってからコミットされたソースコードの条件を揃える為に行った処理である．
- Code generation（CG，コード世代）
  最初にソースコードが作られたときを第一世代とする．1 つ以上のバグを持つ状態から更新によってバグを持たない状態になったときに世代が進むと考える．バージョン管理システムの履歴からどのバグがどのリビジョンからどのリビジョンまであったと判断することは非常に手間がかかり，単純，簡単なメトリ

クスを探すというテーマに反する為，ソースコードをバグを持っているか持っていないかの 2 つの状態のみに分類する．本研究では前者を欠陥モジュール，後者を非欠陥モジュールとしている．

- Code age（CA，コード年齢）

  RN= 1 のソースコードか，ソースコードの状態（欠陥モジュール，非欠陥モジュール）が更新によって切り替わったときを 1 として，更新されるたびに 1 ずつ増加させる．対象のソースコードがどれだけの間，欠陥モジュール，又は非欠陥モジュールであり続けているかを表す．

- Code life（CL，コード寿命）

  RN=1 のソースコードのみで計測する値で，対象のソースコードの状態が切り替わるまで，又は更新が止まるまでにどれだけの版数を要するかを表す．

- Fault birth Renew number（FBR，欠陥誕生版数）

  そのモジュールが欠陥モジュールになった時の RN．

  通算何回目の更新で欠陥モジュールになったかを表す．

- Fault birth Code age(FBA，モジュール故障年齢)

  そのモジュールが欠陥モジュールになる直前の CA+1．

  その時の CG になってから，何回目の更新で欠陥モジュールになったかを表す．

  FBA=1 になるのは最初にファイルを作った時に欠陥モジュールだった時のみ．

表 4.1，表 4.2 に，メトリクスの計測例を示しておく．

4.3 仮説

Todd L. Graves らはソフトウェア変更履歴を用いたソフトウェアの欠陥密度の予測研究において，モジュールの履歴を見て、全ての変更に重みをつけて足し合わせたもので欠陥を予測するモデルがうまくいったと論じている．その中で，過去の変更が現在のモジュールに与える衝撃の重みは，変更が為されてから 1 年経過するごとに半減するとしている．

本研究ではそこに着目し，過去の変更が重み付きと言えども “すべて”足し合わされるならば，ソースコードの版数を見ることで同様の効果が期待できるのではないかと考えて分析を行った．

表 4.1　コードサイズが 0 から始まるモジュールのメトリクス計算例

revision	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7
codesize（Byte）	0	850	2321	2449	6720	6519	9807	10267
バグ	無	無	有	有	無	有	無	無
RN	—	1	2	3	4	5	6	7
CG	—	1	1	1	2	2	3	3
CA	—	1	1	2	1	1	1	2
CL	—	1	2	—	1	1	2	—
FBR	—	—	2	—	—	5	—	—
FBA	—	—	2	—	—	2	—	—

表 4.2　状態が欠陥から始まるモジュールのメトリクス計算例

revision	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7
codesize（Byte）	19851	21020	20989	19101	20502	21981	23410	24203
バグ	有	有	有	無	無	無	有	無
RN	1	2	3	4	5	6	7	8
CG	1	1	1	2	2	2	2	3
CA	1	2	3	1	2	3	1	1
CL	3	—	—	3	—	—	1	1
FBR	1	—	—	—	—	—	7	—
FBA	1	—	—	—	—	—	4	—

本論文では以下の仮説について検証した結果を報告する．

仮説 1: 版数の増加に伴って欠陥の新規発生率は下がる

一般に，更新が為されるほどプロジェクトは成熟していき，安定するとされている．もし版数がメトリクスとして使えるならば、両者の間には負の相関関係があるはずである．可能ならばモデル化を試みる．

仮説 2: 欠陥モジュール，非欠陥モジュールの生存率は版数でモデル化できる

すなわち，一度欠陥モジュールになってからバグが取り除かれるまで（すなわち，非欠陥モジュールになるまで）に要する版の数には何らかの法則があることを示す．

4.4 分析の手順

1. モデルの決定

   先ずは分析を始める前に，取得したメトリクスのデータをグラフにプロットする．ここでグラフの特徴を確認し，適用できそうなモデルを検討する．

2. モデルフィット

   モデルを決定したら，計測したメトリクスを用いてモデルをフィットさせる．この作業には統計解析用の言語である R 言語 [7] を使用する．

   フィットさせる手順は，取得したデータを csv 出力し，R で読み込んで回帰分析を行うというものである．

3. R^2 値の確認

   上記の手順で得られたモデルを使用して，実際の値を推測する．

   相関係数 R は (4.1) 式によって求められる．

$$R = \frac{\text{推測値の平方和}}{\text{実測値の平方和}} \tag{4.1}$$

   よって，寄与率 R^2 値は (4.2) のようにして求められる．

$$R^2 = \left(\frac{\text{推測値の平方和}}{\text{実測値の平方和}}\right)^2 \tag{4.2}$$

このようにして求めた R^2 値が 1.0 に近ければ近いほど，このモデルは推測したいデータによくフィットしていることになる．

5. 適用実験

いくつかのバージョン管理されたオープンソースのソフトウェア開発プロジェクトのリポジトリを入手し，メトリクスを計測，分析して 4.3 節で提示した仮説を検証する．

5.1 対象プロジェクト

本実験では以下の 5 つのオープンソースのソフトウェアを分析対象とする．

- Eclipse BIRT
  Eclipse 用のビジネス文書開発プラグインで，CVS で管理されている．Java 言語で開発されており，欠陥モジュールの解析対象は Java ファイルとする．
- Eclipse
  統合開発環境であり，CVS で管理されている．Java 言語で開発されており，欠陥モジュールの解析対象は Java ファイルとする．
- Netbeans
  統合開発環境であり，Mercurial で管理されている．Java 言語で開発されており，欠陥モジュールの解析対象は Java ファイルとする．
- Chrome
  ウェブブラウザであり，SVN で管理されている．複数の言語を用いて開発されており，欠陥モジュールの解析対象は拡張子が c,cc,cpp,h,m,mm,py のファイルとする．
- Firefox
  ウェブブラウザであり，CVS で管理されている．複数の言語を用いて開発されており，欠陥モジュールの解析対象は拡張子が c,cpp,h,java のファイルとする．

5.2 データ取得

実験を行うためには，バージョン管理システムのリポジトリから全ての変更の情報を取得し，全モジュールの全リビジョンに対して欠陥モジュールと非欠陥モジュールを判別する必要がある．

本研究では，バージョン管理システムとバグ管理システムのデータからSZZというアルゴリズムを用いてそれらを判別し，SQLite3のデータベースにまとめたデータを使用した．

SZZ アルゴリズム

SZZアルゴリズム[8]は，バージョン管理システムとバグ管理システムの保有する情報を相互に結び付け，自動的に不具合モジュールを識別するアルゴリズムである．その処理を行うための基本となるアイデアは以下のようになっている．

- バグ管理システムから不具合修正を行っているログを抜き出す．
- 不具合修正ログとバージョン管理システムを結び付け，不具合修正が行われた変更の情報を抜き出す．
- その変更が行われたより以前の変更を，不具合挿入の原因となる変更だと決定する．

不具合修正がなされたリビジョンより以前の変更は，不具合混入の原因の一つだと考えられる．よって、以前の変更の中から修正された部分の変更を行っているモジュールを識別できれば，それがfix-inducing change（不具合が混入された変更）となる．

実際のアルゴリズムの手順は以下のようになる．

1. バグ管理システムで使われているBugIDに対して，バージョン管理システムの管理しているファイル変更のコメントからそのBugIDがあるものを見つけ出し，そのファイル変更を不具合の修正のための変更とする．
2. ファイル変更の情報と不具合の情報を使って，構文的と意味的な観点から本当に不具合の修正のための変更なのか判定する．
3. 不具合の修正のための変更をfix changeと呼ぶこととし，そのrevisionをr_iとする．ここでfix changeの一つ前のrevision r_{i-1}とr_iの差分を取得しておく．
4. 差分の情報からr_{i-1}とr_iの間で変更・削除された行の行番号をすべて取得し，それらの行番号をLと置く．
5. バージョン管理システムのannotateコマンドをr_iに対して実行し，行Lが追加，変更されたrevisionをすべて調べ，それらのrevisionをRとする．

6. revision R の中から不具合の報告日より前に変更された revision を見つけ出し，その revision を fix-inducing change とする．

解析結果データベースのスキーマ

本実験でメトリクスを計測するために使用したデータセットは，前述の手順で解析した結果が SQLite3 で扱うデータベースの形で保存されている．その形式は次のようになっている．

- TABLE log バージョン管理システムのファイル編集履歴を各ファイルのリビジョンごとに格納する．
  - id INTEGER：リビジョンの識別番号
  - file TEXT：リポジトリ内のファイルパス
  - revision TEXT：リビジョン番号
  - date TEXT：ファイル編集日時
  - author TEXT：ファイル編集者名
  - message TEXT：コミットメッセージ
- TABLE bugdb バグ管理システムのデータベースの情報を格納する．
  - bug_id INTEGER：バグ管理システムで使用している BugID
  - opendate TEXT：不具合の報告日
  - assigned_to TEXT：不具合の報告者名
  - resolution TEXT：不具合の状況
  - short_desc TEXT：不具合の状況説明文
  - bug_severity TEXT：不具合の深刻度
- TABLE buglink fix change（欠陥の修正の為の変更）の情報を格納している．
  - file TEXT：リポジトリ内のファイルパス
  - revision TEXT：リビジョン番号
  - opendate TEXT：不具合の報告日
  - id INTEGER：リビジョンの識別番号
  - bug_id INTEGER：不具合の BugID

- TABLE faulty fix-inducing changes の情報を格納している．
  - id INTEGER：リビジョンの識別番号
  - file TEXT：リポジトリ内のファイルパス
  - revision TEXT：リビジョン番号
  - date TEXT：ファイル編集日時
  - faulty INTEGER：faulty の場合 1，そうでない場合 0
  - fixin INTEGER：fix-inducing change の場合 1，そうでない場合 0
  - line TEXT：不具合の原因となった行番号
- TABLE code ソースコードを格納している．
  - id INTEGER：リビジョンの識別番号
  - code BLOB：ソースコード

リビジョンの識別番号とは，ファイル編集履歴を取得するときに，リビジョンごとに取得した順につける 1 から始まる十進整数である．テーブル code にはファイル名などのフィールドが無いが，リビジョンの識別番号 id を用いて，他のテーブルから取得することができる．異なるバージョン管理システムによる解析結果の違いは，リビジョン番号の形式にしか現れず，他の情報は全て統一化されている．

表 5.1 に，本実験で使用したプロジェクトの基本的なデータを纏めておく．

メトリクスの計測

4.2 節で説明したメトリクスの測定方法について簡単に説明する．

ある 1 種類のファイル（モジュール）について，今回定義したメトリクスを計測するには以下のデータが必要になる．

- そのファイルの最初のリビジョンから最後のリビジョンまでのソースコード
- それらのソースコードが欠陥を含んでいるかいないかの情報
- それらのソースコードのサイズ（RN の計算に必要）

解析結果データベースを利用してメトリクスを計算する手順は以下のようになる．

1. テーブル code 内の全ての code を参照して，それぞれについてファイルサイズを計算する．

表 5.1　分析するプロジェクトの基本データ

	Birt	Eclipse	Netbeans	Chrome	Firefox
file	8973	24426	53359	13113	3988
revision	70142	308966	393531	131059	47376
BugID	21372	254068	184767	36892	457277
fixin	10178	36357	57337	14020	1795
faulty	32508	159818	165560	45490	17902

2. テーブル log 内の file を参照してファイル名のリストを作成する．
3. ひとつのファイル名を選び，テーブル faulty からファイル名と file の等しいカラムを抜き出す．抜き出す情報は id,faulty の 2 種類で，それらを date でソートすることによって変更された順に並べる．
4. 一番最初のリビジョンから最後のリビジョンまで順に faulty を確認しながら辿り，必要なメトリクスを計算する．

この一連の手順を，本研究では C 言語を用いて実装した．

取得したデータは，以下のようなスキーマで SQLite3 の形式のデータベースに記録した．

- TABLE codedata リビジョンの識別番号に対応するソースコードのサイズを記録する．
  - id INTEGER：リビジョンの識別番号
  - size INTEGER：ファイルの大きさ
  - loc INTEGER：ファイルのコード行数
- TABLE metrics リビジョンの識別番号に対応するソースコードについてのメトリクスを記録する．
  - id INTEGER：リビジョンの識別番号
  - faulty INTEGER: 不具合を含むなら 1，含まないなら 0
  - rn INTEGER：Renew number（更新回数）
  - cg INTEGER：Code generation（コード世代）
  - ca INTEGER：Code age（コード年齢）
  - cl INTEGER：Code life（コード寿命）
  - fbr INTEGER：Fault birth Renew number（欠陥誕生版数）
  - fba INTEGER：Fault birth Code age（モジュール故障年齢）

5.3 データの分析

ここでは，分析したデータを基に 4.3 節で提示した以下の仮説を検討する．

仮説 1: 版数の増加に伴って欠陥の新規発生率は下がる

仮説 2: 欠陥モジュール，非欠陥モジュールの生存率は版数でモデル化できる

仮説 1 の検証

RN=x のときの新規欠陥発生率は，RN=x である全モジュールの中で，faulty=1 かつ CA=1 の物が占める割合と定義する．RN=1 が大きくなるにつれ標本数が少なくなるため，新規欠陥発生率が 5 回目に 0 になったところまでのデータを使用した．

表 5.2 は RN と新規欠陥発生率の間の相関係数を求めたものである．この表から，多少弱いが負の相関があることが分かる．

そこで，最初だけ極端に高く，一気に下がって 0 に近づいていくという特徴のデータ分布から，式（5.1）のような冪乗モデルに載るのではないかと考えて，R を用いて回帰分析を行った．

$$y = ax^b \tag{5.1}$$

その結果求められた回帰直線と，元のデータの分布を図 5.1 から図 5.5 に示す．このグラフは，縦軸を新規欠陥発生率，横軸を RN としたグラフであり，求めた回帰直線と，実際に計測したメトリクスがプロットされている．

また、表 5.3 に求めたモデル式のパラメータと，モデルへの適合率 R^2 を示す．これを見ると，netbeans はうまく適合しているが，それ以外は大した相関はなく，特に firefox はほとんどモデル化出来ていないことになる．

これらのデータから「版数の増加に伴って欠陥の新規発生率は下がる」という仮説は，RN と新規欠陥発生率の間には負の相関がみられるので一応正しいが，冪乗モデルを用いて，RN の値で新規欠陥発生率を予測するモデルを構築するのは難しいと考えられる．

表 5.2　RN と新規欠陥発生率の相関係数

	birt	eclipse	chrome	firefox	netbeans
RN	-0.48207	-0.51436	-0.45353	-0.31615	-0.36314

表 5.3　新規欠陥発生率のモデル式と適合率

	birt	eclipse	chrome	firefox	netbeans
a	0.1560	0.1986	0.1786	0.0414	0.2148
b	-0.5727	-0.7040	-0.9678	-0.3005	-0.9665
R^2	0.66	0.74	0.87	0.25	0.94

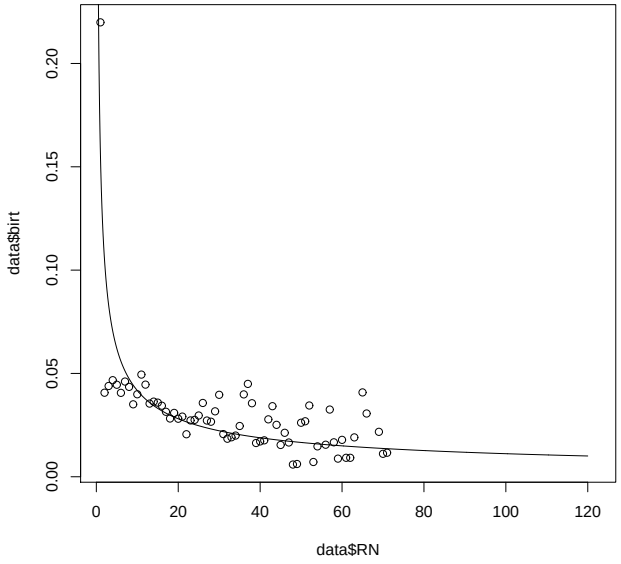

図 5.1　Birt の新規欠陥発生率

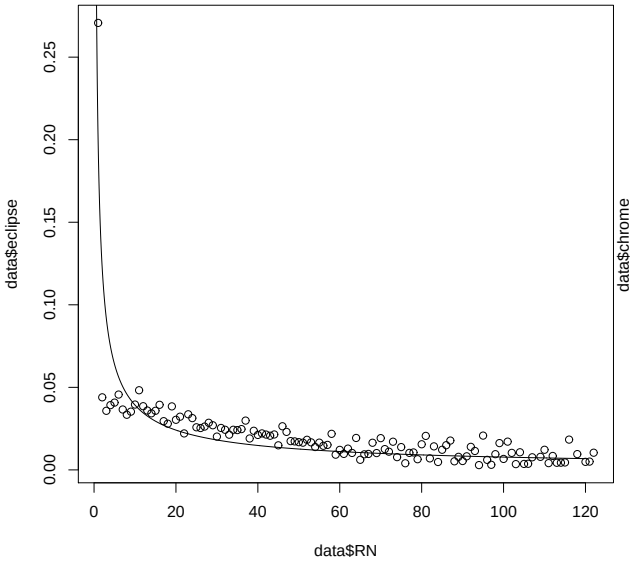

図 5.2　Eclipse の新規欠陥発生率

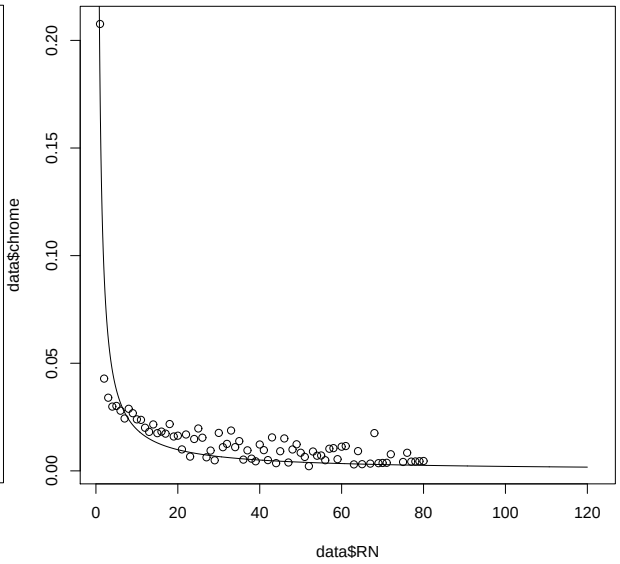

図 5.3　Chrome の新規欠陥発生率

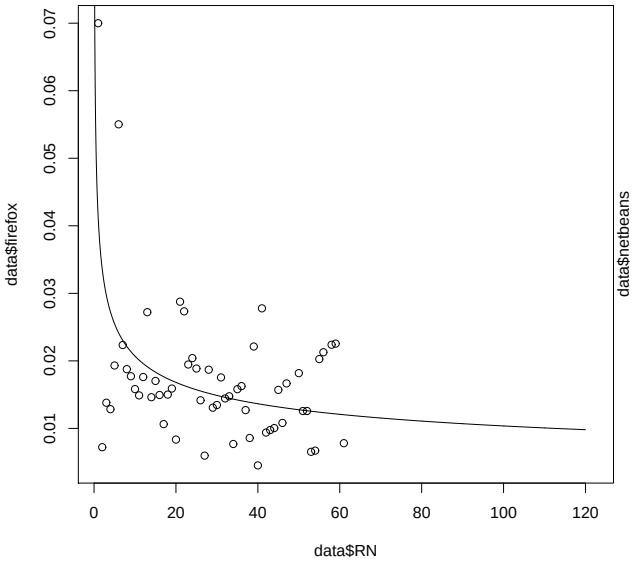

図 5.4　Firefox の新規欠陥発生率

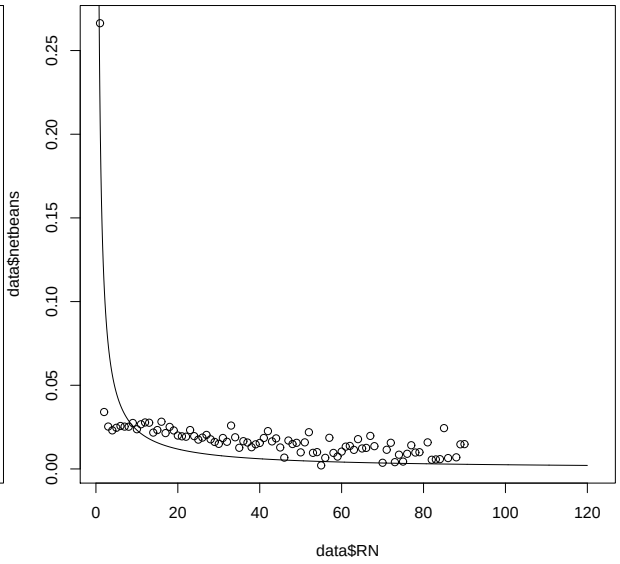

図 5.5　Netbeans の新規欠陥発生率

仮説 2 の検証

欠陥のあるモジュールを faulty, 欠陥のないモジュールを n-faulty と記述する．

先ずは，fauty モジュールと n-faulty モジュールの寿命（Code life メトリクス）に違いがあるかを確認する．表 5.4 は，CA=1 のソースコードで faulty なものと n-faulty それぞれについて CL の平均と分散を計算したものである．この表から，faulty なモジュールは寿命が長くばらつきがあるのに対して，n-faulty なモジュールは総じて短命なことが読みとれる．

次に，生存率を見てみることにする．CA=x の時の faulty(n-faulty) モジュールの生存率とは，faulty(n-faulty) モジュールの総数に対して，その時点でまだ n-faulty(faulty) モジュールに切り替わっていないモジュールの数の割合を表す．これも，仮説 1 と同じく冪乗モデルにフィットさせることにした．

モデルをフィットさせた結果を図 5.6 から図 5.15 までに示す．グラフは，横軸に CA（Code age)，縦軸に生存率をとり，＊が実計測データの散布図であり実線がモデルの方程式を表している．表 5.5 と表 5.6 では求めたモデル式のパラメータと適合率を示している．

適合率が 0.95 を超えているものが多いことから，このモデルは版数の生存率の予測に十分に相関関係があると思われる．よって，仮説 2 は確からしいと考えられる．

表 5.4　ソースコードの平均寿命

	faulty		n-faulty	
	average	variance	average	variance
birt	6.608	185.119	3.770	15.518
eclipse	10.485	563.293	4.464	27.570
chrome	15.229	1242.507	3.640	18.521
firefox	20.205	2060.211	6.336	93.996
netbeans	7.354	199.722	3.697	6.354

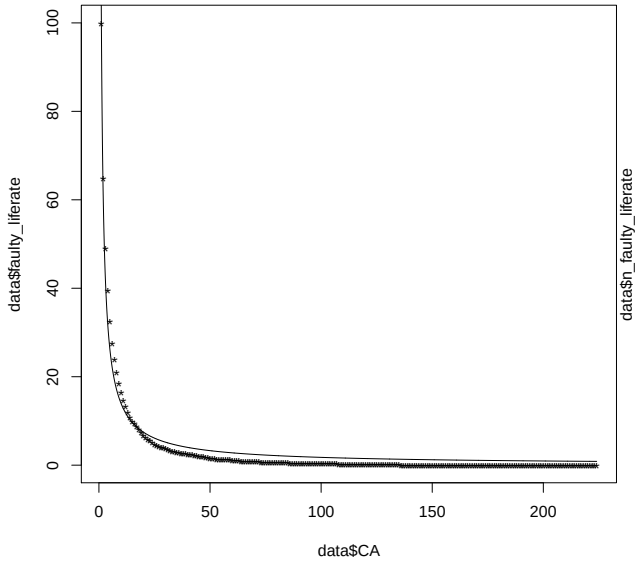

図 5.6

Birt:fauty モジュールの生存率

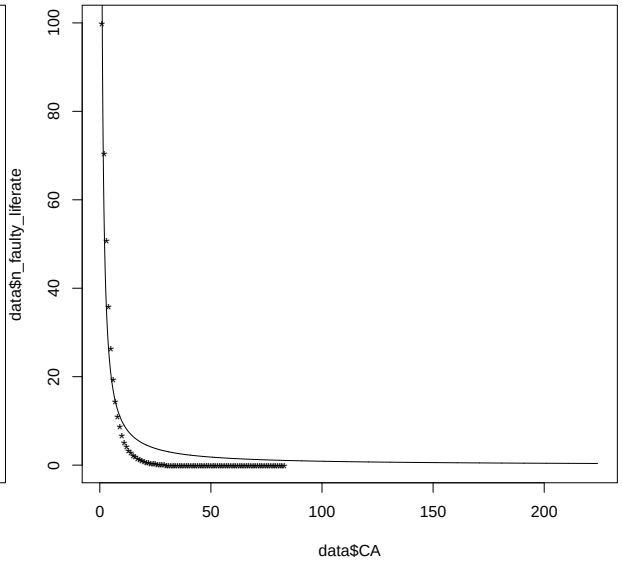

図 5.7

Birt:n-fauty モジュールの生存率

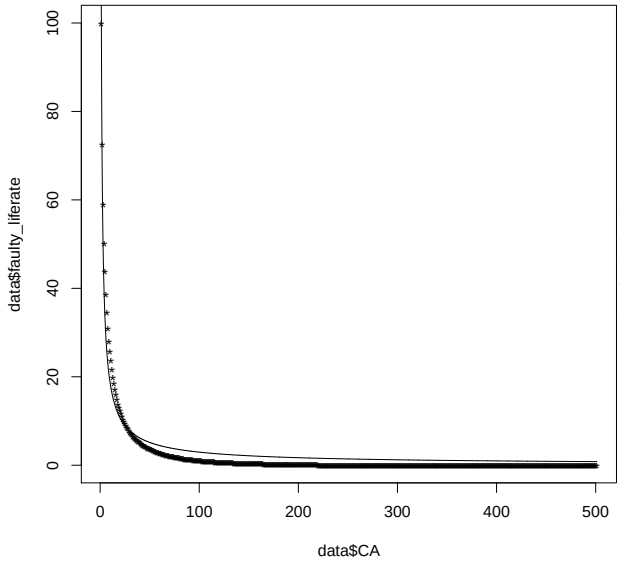

図 5.8

Eclipse:fauty モジュールの生存率

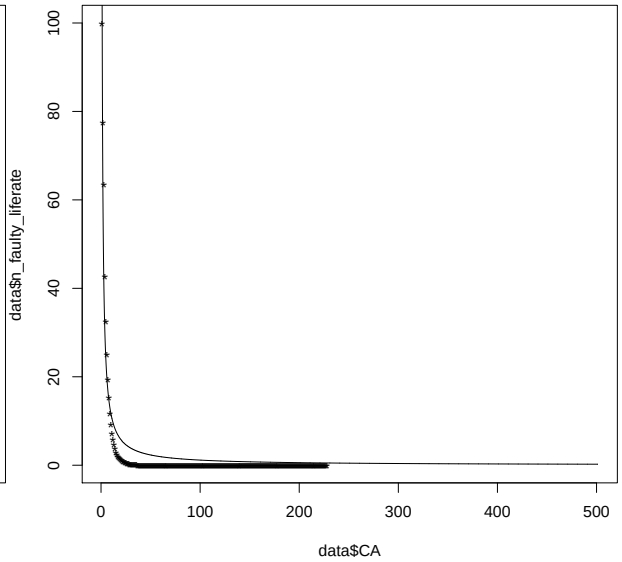

図 5.9

Eclipse:n-fauty モジュールの生存率

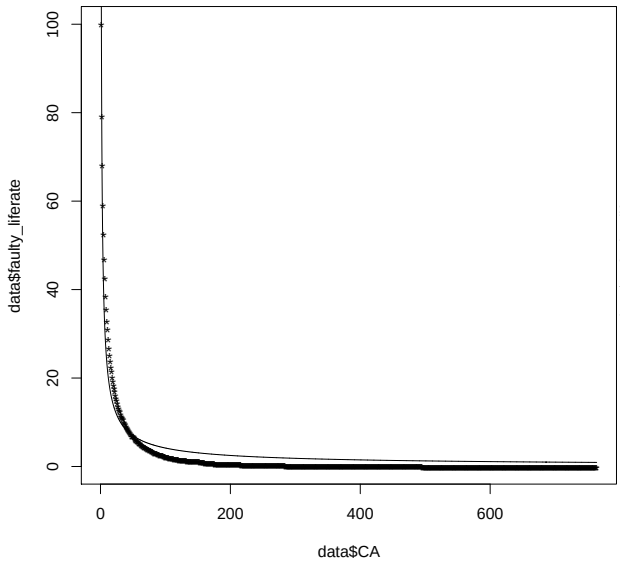

図 5.10

Chrome:fauty モジュールの生存率

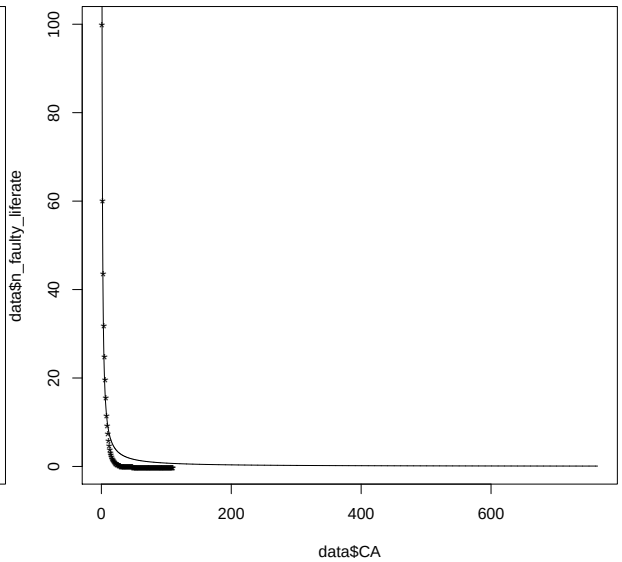

図 5.11

Chrome:n-fauty モジュールの生存率

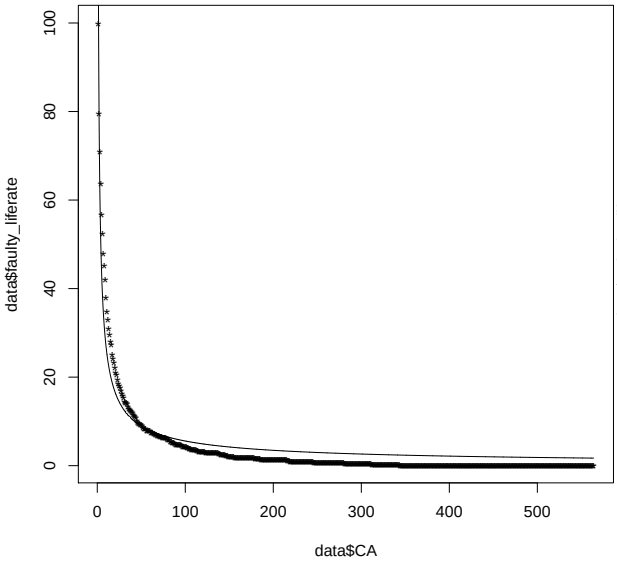

図 5.12

Firefox:fauty モジュールの生存率

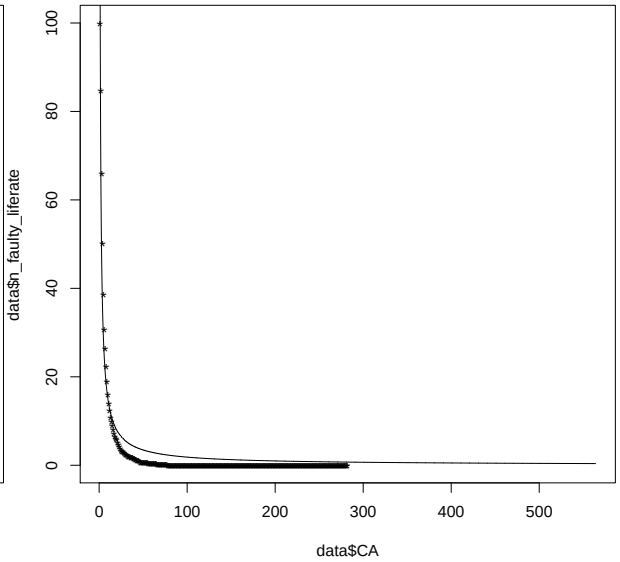

図 5.13

Firefox:n-fauty モジュールの生存率

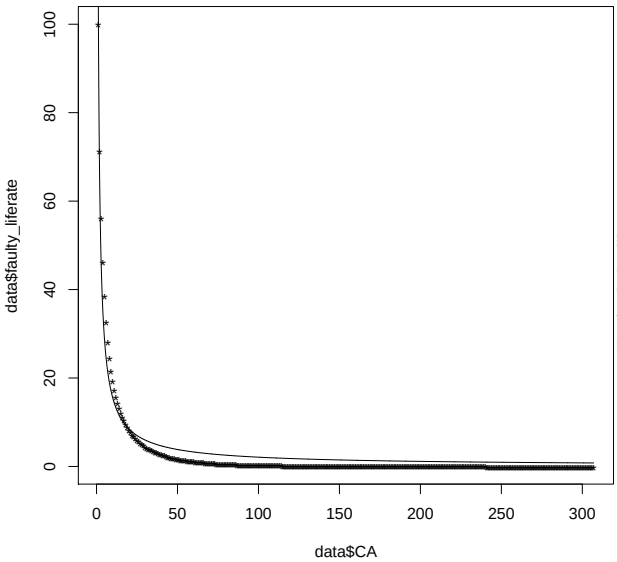

図 5.14

Netbeans:fauty モジュールの生存率

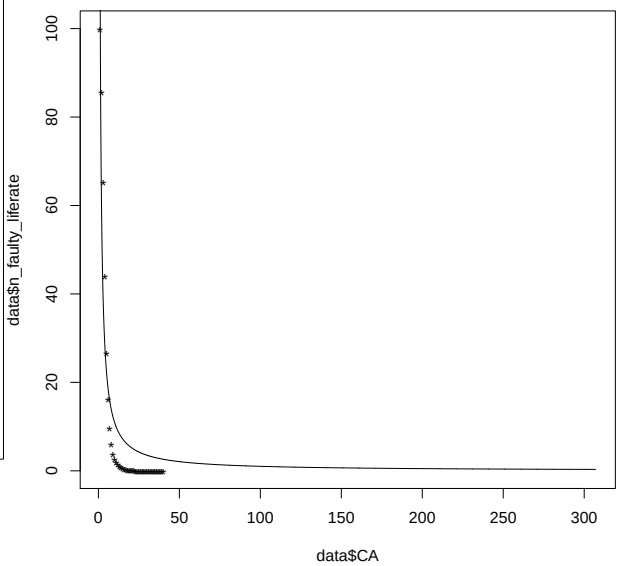

図 5.15

Netbeans:n-fauty モジュールの生存率

表 5.5　fauty モジュールの生存率のモデル式と適合率

	birt	eclipse	chrome	firefox	netbeans
a	109.6046	118.0378	127.0622	129.7978	114.8543
b	-0.8967	-0.8004	-0.7417	-0.6838	-0.8692
R^2	0.98	0.95	0.93	0.93	0.96

表 5.6　n-fauty モジュールの生存率のモデル式と適合率

	birt	eclipse	chrome	firefox	netbeans
a	111.464	116.86	107.608	120.9531	117.407
b	-1.052	-1.076	-1.076	-0.9089	-1.031
R^2	0.95	0.92	0.97	0.93	0.88

6. 考察

RenewNumber と CodeAge

今回の実験でこのような結果になった原因について考えてみる．仮説 1 は欠陥の発生について，仮説 2 は欠陥（または非欠陥）モジュールが状態変化を起こすまでの期間（生存率）を見たものである．

どちらもモジュールの欠陥の存在に関わる仮説であったが，モデルへの適合率はかなり開きがある．しかし，プロジェクト全てが適合しなかったわけではなく，netbeans は $R^2 = 0.94$ という十分高い値を記録している．反対に Firefox は $R^2 = 0.25$ と低い値をとっている．

プロジェクトにおける違いは，メンバーや使用言語，用途や運用年数による成熟度などのパラメータの違いなどが考えられるが生存率の適合度はそれほど変わらなかったことから，それほど大きな影響は無いように思われる．そこで，予測に使ったメトリクスの違いに着目する．

仮説 1 で使用した RN と仮説 2 で使用した CA は，どちらも更新回数に関連するメトリクスであるが，RN は最初からずっと増え続け，CA はモジュールの状態が欠陥と非欠陥を切り替わったときにリセットされるという点で異なる．今回の分析では、この違いが重要であろうと思われる．表 5.4 を見るとわかるように，非欠陥モジュールの寿命は短くどんどん消滅している．欠陥モジュールは分散が大きくばらつきが大きいが，取得データを確認すると頻繁に消滅していくのが確認できた．更新があるたびに何らかの重みが蓄積されていき，欠陥が挿入されるか，又は削除された段階でソースコードは生まれ変わって（世代が変わって）重みがリセット，あるいは軽減される．

こうしたライフサイクルがソースコードにも適用されていると考えるが，仮説 1 で用いた RN は CA と違い，一度欠陥モジュールになった後に非欠陥モジュールに戻っても初期化されることはない．すなわち，重みがリセットされることが無いために，実際の振る舞いとの間に齟齬が出て，相関の低いモデルになってしまったのではないかと考える．

妥当性の検証

最後に，本研究の妥当性において言及しておく．

- 疑似相関の問題

  本研究で関連性があると検討したメトリクスについて，実は全く相関性がなく，偶然一致してしまったり，両方が別のメトリクスと強く相関していて結果的に相関性があるように見えてしまっている場合がある．
- プログラムのバグ

  自分で定義したメトリクスを測定する為のプログラムにバグがある可能性があり，欠陥モジュールと非欠陥モジュールを間違えていたり，測定するメトリクスを間違えている可能性がある．
- データベースの不備

  本実験ではSZZのアルゴリズムを用いて，バージョン管理，バグ管理システムのデータベースから必要な情報を取り出して制作されたデータベースを使用した．SZZアルゴリズムの実装が間違っていて正しい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側の問題

  バージョン管理システムの管理履歴に不備があったり，使用者が適切な情報を入力しなかった等の理由で元々のデータが正しくない可能性がある．

7. 結言

本研究では，版更新と不具合に関連したメトリクスをいくつか提案した。それらのメトリクスを用いて実際にオープンソースソフトウェアの開発プロジェクトに対して分析実験を行うことで，版更新と不具合出現の間に何らかの相関関係があり，相関関係のうち一部はモデル化が可能であることを示した．

今後の課題としては，不具合出現を予測するような他のメトリクスと同時に適用することで予測精度を上げられるか実験を行うことや，オープンソースのプロジェクト以外でも提案したメトリクスが有効であるか調査することがあげられる．

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，研究生 梁軍偉先輩，情報工学課程 川本公章君，西村祐輔君，中井道君，山田悠太君，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] 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.

[2] T.J. McCabe, “A complexity measure,” IEEE Trans. Softw. Eng., vol.2, no.4, pp.308–320, 1976.

[3] A.J. Albrecht and J.E. Gaffney, “Software function, source lines of code, and development effort prediction: A software science validation,” IEEE Trans. Softw. Eng., vol.9, no.6, pp.639–648, 1983.

[4] J.C. Munson and S.G. Elbaum, “Code churn: A measure for estimating the impact of code change,” IEEE International Conference on,Software Maintenance, p.24, IEEE Computer Society, Los Alamitos, CA, USA, 1998.

[5] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” ICSE ’05: Proceedings of the 27th international conference on Software engineering, pp.284–292, ACM, New York, NY, USA, 2005.

[6] T.M. Khoshgoftaar, E.B. Allen, N. Goel, A. Nandi, and J. McMullan, “Detection of software modules with high debug code churn in a very large legacy system,” International Symposium on, Software Reliability Engineering, p.364, 1996.

[7] The R Project, The R Project for Statistical Computing, （オンライン）, 入手先 ⟨http://www.r-project.org/index.html⟩（参照 2010-12-7）.

[8] 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, New York, NY, USA, 2005.