On Incomplete Bug Fixes and Programmers’ Intuition on These

Sabih Agbaria
On Incomplete Bug Fixes and Programmers’ Intuition on These

Research Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Sabih Agbaria

Submitted to the Senate of the Technion - Israel Institute of Technology

Av 5773 Haifa August 21, 2013
The research thesis was done under the supervision of Assoc. Prof. Yossi Gil in the Department of Computer Science.

Acknowledgment

I would like to express my deepest gratitude to Assoc. Prof. Yossi Gil, his devoted guidance and encouragement made this work possible.

I am thankful for everything I learned from Yossi. His insights, ideas and standard of work will surely accompany me well beyond the scope of research.

To my dear family, Miriam, Reem and Mohammad, thank you for the great time.
# Contents

Abstract 1

I Introduction 2

1 Overview 3
  1.1 Method and results 4
  1.2 Other applications 4
  1.3 Related work 5

2 Mathematical model for preference 6
  2.1 A system of equations for the unknown preference values 8
  2.2 A fixed point numerical solution 9

3 Power Law 11

4 Preferential Attachment 15
  4.1 Evolution Monitoring 16

II On Incomplete Bug Fixes 18

5 Bug-Fixes data set 19

6 Metrics of effort invested in bug fixing and refixing 23

7 Results 28
  7.1 Impact of bug properties and the first fix effort on the selection of the next bug to fix 28
7.2 Impact of future refixing effort on the selection of the next bug to fix  

8 Conclusions, discussion, and directions for further research

III How Do Classes Choose Their Parents?

9 Data Corpus

10 Power Law Distribution of the Number of Immediate Extenders
   10.1 Cautions ..................................................... 46
   10.2 Distribution of #Extenders in the Corpus ................... 48

11 Examining the Preferential Attachment Hypothesis
   11.1 Binning ..................................................... 52
   11.2 Branches ................................................... 53
   11.3 Generalized Preferential Attachment ....................... 53
   11.4 Confirming the Weak Preferential Attachment Hypothesis  55

12 Other Candidate Predictors of Parenting Probability

13 Conclusions

IV appendices

14 Method of Interpolation and Removal of Outliers

15 Kendall-tau Rank Correlation Coefficient

Bibliography
List of Figures

5.1 Subversion log messages ........................................ 20
5.2 combining information obtained from the bugs repository and in-
formation from the source code repository .......................... 21
5.3 Distribution of number of bug births per month ................. 21

6.1 Complementary cumulative distribution of $\text{RFIX}^{R+1}$ (the total
number of revisions required to fix a bug) ......................... 24
6.2 Complementary cumulative distribution of $\text{FIX}^{F}$ (number of files
changed in the first revision fixing a bug) ............................ 25
6.3 Complementary cumulative distribution of $\text{FIX}^{T}$ (number of to-
kens changed in the first revision fixing a bug) ..................... 26
6.4 A redraw of Figure 6.3 on a semi-logarithmic scale ............ 27

7.1 Preference value vs. severity ..................................... 29
7.2 Preference value vs. priority ...................................... 30
7.3 Preference value vs. $\text{FIX}^{F}$ (number of files changed in the first
revision fixing the bug) .............................................. 31
7.4 Preference value vs. $\text{FIX}^{T}$ (number of tokens added in the first
revision fixing the bug) .............................................. 31
7.5 Preference value vs. $\text{RFIX}^{R}$ (number of future refixing revisions) . 32
7.6 Preference value vs. $\text{RFIX}^{S}$ (number of files modified during re-
fixes, but left intact during initial fix) ............................. 33
7.7 Preference value vs. $\text{RFIX}^{F}$ (number of file modification opera-
tion during refixes) .............................................. 34
7.8 Preference value vs. $\text{RFIX}^{T}$ (number of tokens added during refixes) 35

10.1 Complementary cumulative distribution of the simulated prefer-
ential attachment process with $b_0 = 0; a = 1; c = 1; n = 1,000.$ 47
10.2 Complementary cumulative distribution of \#Extenders in the corpus (doubly logarithmic planes). .......................... 50
10.3 Complementary cumulative distribution of \#Extenders for the rest of the corpus. ................................................ 51

11.1 Preference value, as computed by evolution monitoring, vs. \#Extenders in the corpus (log-log scale) .......................... 57
11.2 Preference values of artifacts Eclipse (A) and Mojarra (E) in a linear scale (classes with at most 40 children). ............... 58
11.3 Preference values of artifacts Eclipse (A) and Mojarra (E) in a linear scale (all classes). ................................. 58
List of Tables

5.1 The three large projects studied by this research ............... 22
6.1 Pearson correlation of metrics of bug fixing effort ............... 26
8.1 Kendall \( \tau \)'s rank correlation coefficient of preference values with
respect to all metrics in the studied projects ............... 39
9.1 Software artifacts comprising the data corpus ............... 43
10.1 Skewed power-law interpolation (of all data, and outliers removal)
of the CCDF \#Extenders in the artifacts of the corpus. ............... 49
11.1 Kendall \( \tau \) and Pearson-\( \rho \) coefficients computed from the preference values of the evolution of \#Extenders in the artifacts in the corpus. 55
12.1 Kendall \( \tau \) coefficients of the correlation of each of \#Ctors, \#Fields,
\#Methods, \#Tokens, Age, Dust and \#Changes class properties with the respective preference values, as computed by monitoring the evolution of \#Extenders ............... 60
Abstract

Recent studies indicate that multiple patches to software are found in a hefty portion of resolved bugs. It is also known that bugs that require multiple patches take longer to resolve, that their severity tends to be higher than the average and that they induce programmers to engage more in bug discussions.

This work examines programmers' decisions regarding bugs of this sort, and in particular whether fixing these is more likely to be put off later than bugs not requiring multiple fixes.

A mathematical model is developed for a retrospective analysis of bugs maintenance history. In this model we compute the impact of an array of bug properties on the likelihood that a specific bug is chosen, among all open bugs, to receive its first fix. The studies we conduct on a sizable portion of the history of the Eclipse code base indicate that programmers tend to attend first to bugs which are easier to fix.

The results further suggest that some of the criteria that programmers apply (probably unknowingly) to determine whether a bug is easy to fix, is the number of future patches it would require, and the amount of work involved in these patches. This is despite the fact that this information is not supposed to be available to the programmers at the time the first fix is made.

It is anticipated that the method of analysis introduced in this work would have other applications not only in software engineering, but also outside of computer science.

Indeed, in the work presented in the third part, we apply this same method to reason about whether the inheritance relationship found between software classes follows a well-known phenomena.
Part I

Introduction
Chapter 1

Overview

The fact that the Merriam-Webster online dictionary recognizes neither the word “refix”\(^1\) nor the word “remend”\(^2\) may be the result of a very natural and simple logic: if an object is fixed, it ceases to be broken and cannot be refixed. Software engineers are, however, well familiar with tenacious bugs—those bugs which seem to recur again and again after they have been supposedly fixed, refixed and refixed again.

Recent studies [22, 25] indicate that multiple patches to software are found in a hefty portion of resolved bugs. (17%–45% in the projects studied by Nguyen et al. [22] and 22%–33% in the projects studied by Park et al. [25]).

It was also found [25] that these bugs take longer to resolve, that their severity tends to be higher than average and that they induce programmers to engage more in bug discussions.

The research community has devoted considerable research effort to the study of the prevalence of bug refixing [12, 22, 29, 41] and to the development of tools and methods for dealing with this predicament [1, 13, 15, 22, 31, 38, 39, 41] (see reference [25] for a survey).

The question that motivated this work is whether programmers are aware that certain bug fixes that they commit are incomplete and that the bug may require future additional fixes, perhaps even to source files which were not included in the fix.

Our hypotheses were that programmers will usually prefer to fix the easy bugs first while delaying the handling of the more problematic bugs, and that, based

\(^1\)http://www.merriam-webster.com/dictionary/refix
\(^2\)http://www.merriam-webster.com/dictionary/remend
on intuition or other factors, programmers will classify bugs that require multiple fixes as problematic and hence tend to postpone their fix.

### 1.1 Method and results

To test these hypotheses we develop a mathematical model for a retrospective analysis of bugs maintenance history. The model is able to compute the dependency of the “preference value” of bug fixing on other bug properties. (One can think of this value as the relative likelihood that a bug will be selected for fixing among all other present open bugs.)

We applied this model to a sizable portion of the Eclipse code and bug histories. The results largely confirm these hypotheses, showing that (i) programmers are indeed more likely to attend first to bugs whose first fix requires smaller modifications to fewer source files; and, that (ii) the relative likelihood that a bug will receive its first fix is associated negatively with its future history, i.e., the first fix of a bug is likely to be put off if its future will unfold multiple or sizable refixes.

The finding are not to be taken as an implication that programmers have supernatural means for predicting the future, and that they employ this means for putting off “trouble”, but rather to suggest that there are some characteristics of the bug, which perhaps taken together with intuition, generate “bug smells” that distract programmers.

Our work eliminates some obvious candidates for these characteristics such as first fix size and bug severity; there is exciting work ahead in identifying the predictors of bugs’ tenacity.

### 1.2 Other applications

One other application of the technique developed here is in the analysis of bug triaging policy as applied by the software developers. Despite the use of tools for bug triaging and tracking, developers, particularly in open-software projects but also in commercial projects, may deviate from the declared policy. The staggering popularity of the “release early, release often” slogan, may aggravate the problem since the short users feedback loops generate a stream of bugs which is never exhausted.

Indeed, we found that developers in the Eclipse project do not always prefer bugs of higher priority and severity.

The history analysis techniques introduced and exemplified in this work on the bugs domain, are expected to be useful in other contexts. In the third part of this research, we introduce another application of this technique that focuses on
analyzing the diverse phenomena which are thought to follow preferential attachment (see Newman’s survey [21]). Interestingly some of the phenomena that are likely to be explained by preferential attachment and the ensuing power-law are found in the software domain [7, 34, 37]. Other applications may include analysis of medical triage, hidden discrimination against minorities in governmental services, etc.

1.3 Related work

A line of work related to ours begun with the work of Kim, Whitehead and James [16] who argued in favor of the study of the time it takes to fix reported bugs, claiming that this metric plays a major role when measuring software quality of a system, just as the more traditional bugs count metric.

Studies followed suit: Panjer [23] used data mining techniques to predict the time needed to fix bugs with an accuracy up to 34.9%. Weiss et al. [36] used the average time it took to fix past, textually similar bug-reports, to predict how long it will take to fix a newly reported bug.

Later, Giger et al. [11] applied decision tree analysis to investigate the relationships between bug report attributes and the time required for fixing it; they found, for example, that using post-submission data, like the assignee, can improve the precision of the prediction. In a sense, we continue this work by studying one important factor that contributes to the total time it takes to fix a bug, namely, when do programmers attend to it first.

Bettenburg et al [4] who conducted a survey among developers to find what information available in a bug-report they find the most useful in fixing a bug quickly. The study finds a mismatch between what developers need and what users provide in the bug-report. We study whether this mismatch might lead the developers to rely on other factors. Our work complements their work in determining these factors not by asking the programmers, but by analyzing their behavior.
Chapter 2

Mathematical model for preference

Our analysis employs an abstract model of preference in which the behavior of a chooser (which can be an individual, an organization, or even a non-human agent) is described as a sequence of $n$ random selections, each made in its unique probability space.

To understand this abstract setting better, imagine an individual mingling in a cocktail party. This individual examines the room, searching for people who are free for a chat. After identifying the set of “chat candidates”, this individual chooses one, and only one, of these candidates, approaches him or her in order to start the chat. When this chat ends, our hero repeats the process for selecting the next chat target.

Our task is to learn the inner preferences that guide this cocktail party behavior: men vs. women, hair or eye color preference, and even the preferences of approaching better connected people. The difficulty in doing so is that this learning must be conducted based on observations only. Worse, in each of the “chat events”, the selection is made amongst a different set of people. And to complicate things even more, the sequence of sets of “chat candidates” may be very biased, depending on the property we are interested in. For example, it may very well be that the better connected candidates tend to occur less frequently in this sequence.

The essence of the mathematical technique we develop here is the definition of a set of mathematical unknowns, one for each “preference value” associated with any distinct value of the property under investigation. The observations yield a system of polynomial equations for these variables. The unknowns are then found by a numerical solution of this system of equations.

More formally, we require that at any point in this $n$-sequence, the chooser
is presented with a universe $u$ of entities each being characterized by a value of some property. The values of the property are not necessarily numerical; they may be ordered pairs, triples, tuples of any size, or encode the values of multiple properties in any other way. The chooser then selects precisely one of the entities in $u$. The crucial point is that the selection is made at random where the underlying distribution depends only on the values of this property.

In our case, the entities are open bugs, while the chooser may be thought of as a software manager presented with the current set of bugs and asked to decide which of these to deal with. While there are many factors that may influence this decision, our abstract model concentrates on the values of a certain measurable property (e.g., the effort required to fix a bug), or even a combination of such properties.

Assume that $V$, the set of values of the said property, is finite, i.e.,

$$V = \{v_1, \ldots, v_\ell\}.$$  

For $v \in V$, let $u_v$ be the number of entities in $u$ whose value is $v$. Also, let $x_v \geq 0$ be the preference value of $v$, i.e., the weight assigned to entities with value $v$ in the random selection process. Then, the probability of selecting an entity from $u$ whose property value is $v$ is simply $u_v x_v$, the weight of entities with value $v$, divided by $\sum_{v' \in V} x_{v'} u_{v'}$, the total weight of all entities, that is,

$$\frac{u_v x_v}{\sum_{v' \in V} x_{v'} u_{v'}}. \tag{2.1}$$

The challenge at hand is determining $x = (x_1, \ldots, x_\ell)$, the vector of unknown preference values. (Clearly, this vector can only determined up to an arbitrary scaling factor.)

The relative magnitude of the preference values is telling of the extent of dependency of the selection on the property. If all the values in $x$ are equal, then the selection process does not depend at all on the property under inspection. The greater the variety of these values, the greater the dependency of the selection in the values the property assumes.

Note that preferential attachment is a special case of our model, in which $V$ is a set of non-negative integers and we expect to be able to show that $x_v \propto (v + b)$ for some positive constant $b$.

The difficulty in determining $x$ is that a single random choice is useless for inferring about the underlying probability space in the same way that the result
of a single coin flip tells nothing of the coin’s bias. Moreover, unlike the coin flips example, our model does not permit repetitions of the experiment in equal conditions.

2.1 A system of equations for the unknown preference values

What we shall do instead is consider together the universes \( u_1, \ldots, u_n \) that occur in the sequence and use these to write and solve a system of equations for \( x \). We write a universe \( u_i \) as an \( \ell \)-vector \( u_i = (u_{i,1}, \ldots, u_{i,\ell}) \), or, for brevity, as \( u_i = (u_{i,1}, \ldots, u_{i,\ell}) \), and define an \( \ell \times n \)-matrix \( U \) in which the \( i^{th} \) row is the vector \( u_i \):

\[
U = \begin{pmatrix}
  u_1 \\
  \vdots \\
  u_n
\end{pmatrix}
\]  \hspace{1cm} (2.2)

Also, let the \( n \)-vectors \( w_j, j = 1, \ldots, \ell \) be the columns of \( U \), i.e.

\[
U = (w_1, \ldots, w_\ell).
\]

For \( i = 1, \ldots, n \) and \( j = 1, \ldots, \ell \), let \( \delta_{i,j} \) be the binary random variable assuming the value 1 if a chooser selects at step \( i \) an entity with a value \( v_j \), and 0 otherwise. Then, the expected value of \( \delta_{i,j} \) is simply

\[
u_{i,j}x_j/u_i \cdot x
\]

where “\( \cdot \)” denotes the dot vector product. Also, for \( j = 1, \ldots, \ell \), define the counting random variable

\[
t_j = \sum_{i=1}^n \delta_{i,j},
\]  \hspace{1cm} (2.3)

that is, \( t_j \) is the total number of times a value \( v_j \) was selected. By linearity of expectations

\[
E[t_j] = \sum_{i=1}^n E[\delta_{i,j}] = \sum_{i=1}^n \frac{u_{i,j}x_j}{u_i \cdot x}
\]  \hspace{1cm} (2.4)
We can even write

\[ E[t_j] = (w_j \cdot y)x_j \]  

(2.5)

where

\[ y = \left( \frac{1}{u_1 \cdot x}, \ldots, \frac{1}{u_n \cdot x} \right). \]  

(2.6)

or, by agreeing that division of vectors is taken point-wise, \( y = 1/U \cdot x \) (where \( 1 \) is an \( \ell \)-vector whose entries are all 1, and where “\( \cdot \)” denotes the usual multiplication of a matrix by a vector).

Now all \( \ell \) equations (2.5) can be written concisely as

\[ E[t] = (U^{tr} \cdot y) \circ x = \left( U^{tr} \cdot \frac{1}{U \cdot x} \right) \circ x. \]  

(2.7)

The above system has \( \ell \) equations for the \( \ell \) unknowns \( x \), where matrix \( U \) is given by the problem specification. Vector \( t \) is also known, since each entry in it represents the number of times an entity with a specific quantity value is selected. We have no practical means for computing the expectation of \( t \) though, but if the individual values in it are not too small, we shall use the maximum-likelihood approximation \( t \approx E[t] \).

### 2.2 A fixed point numerical solution

The high degree of the polynomials in the system (2.7) precludes an analytical solution. A numerical solution based on the fixed point method is feasible though. Rewrite (2.7) in the form

\[ x = E[t] \bigg/ \left( U^{tr} \cdot \frac{1}{U \cdot x} \right) \]

and define \( x_{k+1} \), the \((k + 1)\)th approximation of \( x \) based on \( x_k' \), the previous such approximation:

\[ x_{k+\frac{1}{2}} = E[t] \bigg/ \left( U^{tr} \cdot \frac{1}{U \cdot x_k} \right) \]

\[ x_{k+1} = \frac{x_{k+\frac{1}{2}}}{|x_{k+\frac{1}{2}}|} \]  

(2.8)
We applied (2.8) in this study with excellent results, finding a solution to (2.7) in our domain with error not exceeding $10^{-15}$ after a few dozens of iterations whose total CPU time is measured in seconds. In fact, Amnon J. Meir and Irad Yavneh (personal communication) proved that convergence to the unique solution is guaranteed under broad conditions. An inevitable error though is due to the substitution (common to all statistical analyses of this sort) of $E[t]$ by $t$. 
Chapter 3

Power Law

What is in common to the intensity of wars, size of moon craters, number of scientific citations, and a wide array of object-oriented software metrics? The distribution of all of these, and many other, seemingly unrelated, natural and human-generated phenomena, is governed by what is known as power law, which, roughly speaking, is a distribution function $P(x)$ of a random variable, in which, for at least part of the entire range of $x$,

$$P(X) \propto x^{-\alpha},$$

where $\alpha$ is some positive real constant (typically small).

The term power-law includes in it Pareto distribution [24], Zipf distribution [43], Zipf-Mandelbrot distribution [19]. Power-law was observed in a myriad of domains, of which the intensity of wars [30], size of moon craters [20] and the number of scientific citations [28] make only a small sample. (See references [6, 21] for a list of more such domains.)

The discovery of power-laws in software dates back to the end of previous millennium [7], and has been (re-)discovered at least by Valverde, Cancho, Ferrer and Sole [34], by Wheeldon and Counsell [37], Potanin, Noble, Frean and Biddle [27]. (See also the extensive treatise of Baxter et al. [3].)

The most popular model for explaining power-law distributions in software, just as in other domains, is, arguably, the Yule process, also known as preferential attachment. (We shall be using the two terms interchangeably.) G. U. Yule first described [40] the process bearing his name to explain the distribution of the number of species per genus in nature, arguing that the probability that a species is added to a new genus is proportional to the number of species the genus already
has. Under certain liberal conditions, evolution will converge to a power-law distribution of genera classified by the number of species they have.

The more modern term preferential attachment process refers to a process by which resource (such as wealth) tendency to attach to an existing accumulation of this quantity is proportional to the current size of accumulation.

A famous modern application of the preferential attachment model was that of Barabási and Albert [2] to explain the topology of the web. In software, Concas et al. [8] argued that the Yule process might be responsible for the power-law distribution of the famous Number of Children (NOC) object oriented metric of Chidamber and Kemerer [5].

Further, recently, Turnu et al. [33] used a simulation of a multi-staged Yule process to show that this modification can be used to explain earlier findings of the power-law distribution of the names of variables and methods, the number of calls to a method, and the number of children of classes.

Indeed, preferential attachment provides a very plausible explanation for these and other software metrics: Envision a programmer in search for the appropriate method to call. Then, we may argue that this programmer is more likely to encounter, and therefore use, a method which has been called more frequently than other methods. Also, when creating a new class, the programmer, who seeks a good starting point, will search the most appropriate existing class to extend. Classes which served more often as parents are more likely to parent again, either because they are more visible to the programmer, or because their usability has been demonstrated and improved by the number of children they already gathered.

In this work, we only study the distribution of the famous NOC (number of children) object oriented metric, as defined by Chidamber and Kemerer [5]. One reason for choosing this metric is that it was shown that it follows a power-law in several independent studies [3,9,37]. Further, Chidamber and Kemerer [5] explain that NOC is a measure of reuse, and of the potential influence that a class has on the overall system design. We believe that NOC is in many ways a measure of the use of the object-oriented paradigm. The main question that interests us here is what is it in a class that makes it more extensible.

Unlike previous work (e.g., references [9,33]) we do not argue for the existence of an underlying Yule process by fitting the empirical data with the theoretical findings. (We do show however that some of the datasets follow a power-law quite closely, while, imposing a power-law on other datasets necessitates the removal of many “outliers”.) Instead, we try to evaluate whether the probability of attachment is indeed the “preference value” hypothesized by the theoretical pref-
erential assignment process, and more generally, we ask what are the properties of a class that make it more extensible than others.

To this end, we develop the technique of *evolution monitoring*, which, by inspecting the dynamics of the evolution of a Yule-like process, can compute the *preference values*. Evolution monitoring captures snapshots of the versions’ history of a software artifact and computes from it the *preference value* of classes with a certain number of children, that is, the likelihood of such classes being selected as parents. The preference values become unknowns in a system of multivariate polynomial equations; are computed by an iterative numerical solution.

This solution, though numerically stable and fast converging, is noisy. The reason is that our raw data suffers from statistical fluctuations. Hence, the accuracy of the coefficients of the polynomial equations is limited. Rather than using the computed preference values, we manage noise by employing Kendall tau rank correlation coefficient to reason about the correlation between the predictor, e.g., the current number of children, and the preference values. In particular, this coefficient is telling of the correlation’s sign, positive or negative, and its strength.

Our study employs a corpus of 16 JAVA software artifacts of moderately large size. We verified that in these artifacts, the distribution of the number of children follows a power-law. We extracted the history of each of the artifacts from its respective source code version management repository. Evolution monitoring was then applied to assess the quality of prediction of *parenting probability* by the number of children, and other candidate predictors.

We were unable to find evidence that the likelihood that a class is chosen as a parent, depends *linearly* on the number of children it currently has, as per the hypothesis of classical preferential attachment. Instead, we engage in what we shall call *weak preferential attachment*, by which the likelihood that a class is chosen as a parent *tends to be monotonically increasing* with the number of children it currently has.

Thus, we demonstrate that the number of children is a good *predictor* of parenting probability. Concretely, we show that the more children a class has, the more likely it is to parent again.

---

1The term “predictor” as it is used here means: “positively (or negatively) correlated with the probability that a class is selected again as parent, as measured throughout the history; this is not the same as correlation with the total number of children a class has, say, at the end of this history. To understand the difference, suppose that the data shows that classes that have a large number of methods, also have many extenders. Then, it could be the case that the number of methods in a class, is unrelated to the probability that the class is chosen as a base class. For example, the actual situation could be that classes have more children, tend to evolve later to have more methods.
We further found that the strongest and unequivocal predictor of extensibility is the overall number of changes that the parent underwent previously, as well as the recency of these changes. This finding might be intriguing since a large number of changes may be indicative of a shaky implementation; however, we believe that this finding shows that extensibility is achieved by a process of refinement and perfection of the base class. Still, it might be argued that the quality of these predictors is due to the fact that in the open-source world (which defined our sampling space), developers are likely to modify classes so as that they meet their needs, before extending them.

We believe that this research represents the first attempt in the software domain of investigating the dynamics of evolution of software metrics. It should be said that our work is different from that of Turnu et al. [33] who studied a small number of major releases of three software artifacts (Eclipse, JDK and Ant), but fitted each release with its specifically parametrized Yule process. In contrast, evolution monitoring considers the entire history as a whole, trying to determine whether it reflects a preferential attachment process or a generalization thereof.
Chapter 4

Preferential Attachment

In the term *urn model* we shall refer to a multi-set of *urns*, each containing a number of indistinguishable *balls*. Urns in this model can be distinguished only by the number of balls in them—all urns with the same number of balls are considered identical. An urn model including urns with up to \( \ell \) balls can thus be written as an \((\ell + 1)\)-vector of integers \( u = (u_0, u_1, \ldots, u_{\ell}) \) where \( u_b \) is the number of urns with \( b \) balls in them.

The *classical preferential attachment process* (also known as the *Yule process* [40] or the Yule-Simon process [32]) is concerned with two kinds of operations that evolve an urn model: *urn addition* in which an urn with a specified number of balls is added, and *ball throwing* in which a single ball is added to an urn with a specified number of balls.

More specifically, the process is defined by two integers \( a > 0 \) and \( b_0 \geq 0 \) and a real number \( c > -b_0 \): The process starts with a model with no urns. This empty model then evolves by a series of \( n \) iterations, each iteration comprising of an *addition* of an urn with \( b_0 \) balls, followed by a *ball throws*. Throws occur at random, where an urn with \( b \) balls in it is selected with probability proportional to \( b + c \). Let \( P(b) = u_b / \sum_{b'} u_{b'} \) denote the fraction of urns with \( b \) balls. Then, it can be shown [21] that

\[
\lim_{n \to \infty} P(b) = \frac{B(b + c, \alpha)}{B(b_0 + c, \alpha - 1)}.
\] (4.1)

where \( \alpha > 2 \) is a constant defined by

\[
\alpha = 2 + \frac{b_0 + c}{a}
\] (4.2)
and $B(\cdot, \cdot)$ is Euler beta function. Now, since $B(x, y)$ is proportional to $x^{-y}$ for large $x$, irrespective of $y$, it follows that $P(b) \propto (b + c)^{-\alpha}$ (what we call a skewed power law), for sufficiently large $b$ and $n$. For large values of $b$ this proportional relation can be written in the non-skewed form $P(b) \propto b^{-\alpha}$. Thus, preferential attachment gives rise to the power-law behavior. Consequently, it became a popular explanation of this behavior.

4.1 Evolution Monitoring

The practice of applying (2.8) requires that we eliminate those values of $b$ for which the matrix $U$ provides no information, since the system of equations is singular in these. Specifically, we only concentrate on values of $b$ for which at least one model presented urns with $b$ balls, i.e.,

$$\left( \sum_{i=1}^{n} u_i \right)_b \neq 0. \quad (4.3)$$

If evolution monitoring is performed on the entire sequence of models obtained in a preferential attachment process starting from the empty model, the condition (4.3) never occurs for $0 \leq b \leq \ell + 1$.

In the general case that evolution monitoring is applied to an arbitrary sequence of intermediate urn models, let $m \leq \ell + 1$ denote the number of distinct $b$ values for which the condition (4.3) holds, and eliminate from the system all equations that pertain to other values of $b$. In retrospection, we see that evolution monitoring is applicable also to the more general setting in which one property of urns defines the likelihood of another property increasing. In our domain of interest, we can apply evolution monitoring, e.g., to compute how the number of methods in class defines the preference value of this class serving as a parent. In doing so, the matrix $U$ is defined by the counts of the number of classes with any number of methods in the intermediate models, while $t_b$ is the number of times a class with $b$ methods is selected as a parent.

Finally, we note that Leskovec, Backstrom, Kumar and Tomkins [18] method of approximating $x$ can be written with our notation as

$$x = \frac{t}{\left( \sum_{i=1}^{n} u_i \right)} , \quad (4.4)$$

i.e., the preference value of an urn with $b$ balls is the number of times such an urn was selected, divided by the total number of total occurrences of urns of this size during the entire evolution process.
Relying on this division is, of course, inaccurate: the probability of selecting an urn with \( b \) balls does not depend only on the number of urns with this many balls, but also on the full urns spectrum at the time of selection. The approximation is valid however when the intermediate models are selected when the underlying system is close to its equilibrium state, in which case, the spectrum of distributions is similar in all models.
Part II

On Incomplete Bug Fixes
Chapter 5

Bug-Fixes data set

We focused on changes to the Eclipse source code repository that had occurred in the time period between April 3rd 2005 and May 2nd 2012. This period represents about 60% of the Eclipse development history.

Changes to non-JAVA files, including e.g., XML information, documentation, icons, images, .jar files and other binary files were eliminated from our study. In total, the code history tracked involved some three hundred thousand changes conducted by over 150 developers to circa sixty thousand source files.

Recall that Eclipse version management is carried out by CVS which tracks changes to individual files and not at transaction level [42]. To make it possible to identify changes and in particular bug fixes which pertain to multiple files, we then employed the cvs2svn\(^1\) tool to convert the raw data into subversion format, in which co-occurring changes are consolidated into a single commit operation. In total, we identified 58,985 subversion revisions in this seven year period, which amount to about 23 revisions on average per day.

Each one of those revisions has a corresponding commit log that contains information regarding when it took place, by whom and a textual description for why it was needed. For example take a look at the following log, corresponding to three revisions numbered 51348, 61499 and 75108.

The bug (or bugs) fixed in each commit can then be identified (as standard in the literature [10, 16, 35]) by pattern matching. So, for example, from the log message of the revision 75108 above, it can be inferred that two bugs, namely bug with id 178333 and another with id 178347 were fixed. As usual, one should be careful when applying pattern matching methods, for example many commit

\(^1\)http://cvs2svn.tigris.org/
messages contain small numbers like 2006 that from context one can learn that it
corresponds to a year, and large numbers which in our case usually related to test
suite numbers. While manual inspection was helpful in identifying such extremes,
we also relied in cross-referencing those bug ids with the ids found in the bugs
repository.

Another bit of information, that was extracted and used from the commit log,
is the set of files constituting the change. Moreover, the details of the change
that happened in each file was also obtained, by comparing the file to its previous
version.

More information on the bug, most importantly, the time it was opened, was
then extracted from the Eclipse’s BugZilla defect management system. BugZilla’s
bug report also contain information on the bug such as its severity, from a bug
submitter point of view.

Overall, in the said time period we found 26,689 unique bugs which were fixed
at least once by a modification to a JAVA source file.

Looking at when these bugs were reported, reveals that the vast majority of
them where opened in the same period, while around 2000 of them predate it.

The distribution of number of bug births per month among those we focused
on is depicted in Figure 5.3.

The sheer size of Eclipse, along with the diversity of application domains in-
cluded in it, may make this single project a good representative of JAVA open

Figure 5.1: Subversion log messages
source development culture. Nevertheless, we do not argue that all of our finding are universal to any programmer in any programming culture and in any develop-
opment process and methodology. To the contrary, we believe that although the analysis technique is universally applicable, and although the fact programmers have their ways of predicting the tenacity of bugs is not unique to Eclipse, the details may be different for different projects.

For the purpose of demonstration, we applied the same analysis techniques to two other large scale projects, Mozilla and Webkit, which, just like Eclipse, employ Bugzilla for tracking bugs.

WebKit is a browser engine that powers browsers such as Google’s Chrome and Apple’s Safari, while Mozilla-core is the core library shared by most of Mozilla applications.

Table 5.1 summarizes and compares the principal characteristics of the three large projects we studied.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Revisions</th>
<th>Time frame</th>
<th>#Bug reports</th>
<th>#Prog. Lang.</th>
<th>Revision Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>58,987</td>
<td>2005–2012</td>
<td>26,690</td>
<td>JAVA</td>
<td>CVS</td>
</tr>
<tr>
<td>Mozilla</td>
<td>326,048</td>
<td>1998–2013</td>
<td>50,475</td>
<td>C++, some JAVA, some JAVASCRIPT</td>
<td>Mercurial</td>
</tr>
<tr>
<td>Webkit</td>
<td>127,998</td>
<td>2001–2013</td>
<td>36,190</td>
<td>C++, some JAVASCRIPT</td>
<td>Git</td>
</tr>
</tbody>
</table>

Table 5.1: The three large projects studied by this research
Chapter 6
Metrics of effort invested in bug fixing and refixing

Let us distinguish between the first fix (or fix for short) of the bug, i.e., the first revision checked into the source control system which is tagged as fixing the bug, and the refix which constitutes zero or more revisions checked in after the first fix, which are still tagged as addressing this bug.

We shall measure the amount of effort invested in bug fixing and refixing by the following metrics:

- **File count**, or $F$, which is the number of source files that were modified in the relevant revision or revisions (A file modified in $r$ different revisions of the refix, contributes $r$ to $F$); and,

- **Token change**, or $T$, defined as the number of JAVA tokens added to or removed from the affected source files (if $t_1$ tokens were added to a certain source file in one revision of a refix and $t_2$ tokens were removed from it in another such revision, then the contribution of this file to $T$ is $t_1 + t_2$; similarly, if $t_3$ tokens were removed from a certain file and $t_4$ tokens were added to another file in the same revision, then these two files contribute $t_3 + t_4$ to $T$).

Two additional metrics pertain to the refix of bugs (but not to their first fix):

- **Revision count**, or $R$, which is simply the number of revisions that constitute the refix; and,

23
- Spread, or $S$, defined as the number of distinct files modified in any of the refixing revisions, but intact during the first fix (the contribution to $S$ of a file modified multiple times during the refix is 1 if the file was not modified in the fix, and 0 otherwise).

We apply superscript notation to these, and will be discussing the quantities $\text{FIX}^F$, $\text{FIX}^T$, $\text{PREFIX}^F$, $\text{PREFIX}^T$, $\text{PREFIX}^R$, and, $\text{PREFIX}^S$ in the sequel. In extracting the data set we compute the values of all of these metrics for each of the bugs, and associate their values with each revision in which the bug was fixed.  

Figure 6.1 depicts on a doubly-logarithmic scale the complementary cumulative distribution function (CCDF) of the total number of revisions required to fix a bug, that is, $\text{PREFIX}^R + 1$.

![Figure 6.1: Complementary cumulative distribution of $\text{PREFIX}^R+1$ (the total number of revisions required to fix a bug)](image)

We see that at least a quarter of the bugs were not fixed at the first revision in

---

1. Note that the metrics we use are somewhat rough; a more accurate measure of the effort would be in computing the Levenshtein distance of the sources, possibly accounting for renaming. Also, one can argue that producing a revision that applies major modification to one file and a tiny modification to another requires less effort than that may be required for a revision in which the amount of work is balanced, etc. The weeks of processing time required to process our huge data set made such extensions infeasible.
which they were attended, and the fix of about one percent of the bugs required five revisions or more. The CCDF seems to follow a straight line in the logarithmic plane, which suggests that the metric $\text{REFERIX}^R + 1$ obeys a power-law.

![Figure 6.2: Complementary cumulative distribution of $\text{FIX}^F$ (number of files changed in the first revision fixing a bug)](image)

What’s the distribution of bug fixing effort in the first attempt? Figure 6.2 and Figure 6.3 show, respectively, the CCDF of $\text{FIX}^F$ and $\text{FIX}^T$.

We see that $\text{FIX}^F$ is heavy tailed and seems to follow a power-law. On the other hand, $\text{FIX}^T$ exhibits faster than polynomial decay. In fact, Figure 6.4 which redraws Figure 6.3 on a semi-logarithmic scale indicates that $\text{FIX}^T$ decays exponentially in the majority of the range it spans.

To conserve space we state, without attaching figures, that the distribution of the other file-based metrics ($\text{REFERIX}^F$ and $\text{REFERIX}^S$) is power-law like. Similarly, we state that the decay of $\text{REFERIX}^T$ resembles an exponential.

We expect that the metrics $\text{FIX}^F$ and $\text{FIX}^T$ will play a role in the decision to make the first fix to a bug. What’s more interesting is the impact on this decision that the metrics $\text{REFERIX}^F$, $\text{REFERIX}^T$, $\text{REFERIX}^R$, and, $\text{REFERIX}^S$ may have. The reason is that the values of these metrics can only be computed by examining the bug’s future. If such an impact is demonstrated, then we may venture saying that programmers have some means for estimating that some bugs would require more
refixes than others.

An immediate concern is that this estimation is a result of correlation between the effort required for the first fix of a bug, the number of refixes and the effort required for these. In judging this threat to validity, we computed the Pearson correlation of all pairs of our effort metrics. The results are tabulated in Table 6.1.

<table>
<thead>
<tr>
<th></th>
<th>(\text{FIX}^F)</th>
<th>(\text{FIX}^T)</th>
<th>(\text{REFIX}^F)</th>
<th>(\text{REFIX}^T)</th>
<th>(\text{REFIX}^R)</th>
<th>(\text{REFIX}^S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{FIX}^F)</td>
<td>1.00</td>
<td>0.15</td>
<td>0.15</td>
<td>0.02</td>
<td>0.05</td>
<td>0.06</td>
</tr>
<tr>
<td>(\text{FIX}^T)</td>
<td>0.15</td>
<td>1.00</td>
<td>0.06</td>
<td>0.01</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>(\text{REFIX}^F)</td>
<td>0.15</td>
<td>0.06</td>
<td>1.00</td>
<td>0.07</td>
<td>0.15</td>
<td>0.39</td>
</tr>
<tr>
<td>(\text{REFIX}^T)</td>
<td>0.02</td>
<td>0.01</td>
<td>0.07</td>
<td>1.00</td>
<td>0.35</td>
<td>0.41</td>
</tr>
<tr>
<td>(\text{REFIX}^R)</td>
<td>0.05</td>
<td>0.03</td>
<td>0.15</td>
<td>0.35</td>
<td>1.00</td>
<td>0.33</td>
</tr>
<tr>
<td>(\text{REFIX}^S)</td>
<td>0.06</td>
<td>0.01</td>
<td>0.39</td>
<td>0.41</td>
<td>0.33</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 6.1: Pearson correlation of metrics of bug fixing effort

Examining the correlation values of metrics of the first fix, and metrics of the refix, we see that they are all very low.

![Graph showing the complementary cumulative distribution of \(\text{FIX}^T\) (number of tokens changed in the first revision fixing a bug).](image)

Figure 6.3: Complementary cumulative distribution of \(\text{FIX}^T\) (number of tokens changed in the first revision fixing a bug)
Figure 6.4: A redraw of Figure 6.3 on a semi-logarithmic scale
Chapter 7

Results

In this chapter, we study what we shall call the preference value of bugs, defined as the relative likelihood of a specific open, never fixed before, bug to be fixed in the next revision. The study will employ the model described in the previous chapter to compute dependency of the preference value (denoted $x$) on the user-defined severity and priority labels and on the fixing effort metrics defined above in Table 6.

7.1 Impact of bug properties and the first fix effort on the selection of the next bug to fix

Having established the mathematical model for the analyses of histories, we are ready to present our empirical results. Our interest lies with the criteria that programmers apply in choosing which bug to fix. In the terminology of the previous chapter, a universe is the set of open bugs which have never been fixed. The selection points are check in operations (commits) of a revision which fixes a bug for the first time. And, we would like to determine whether the factors are involved in selecting from among all present open bugs, the next bug to fix.

To this end, we fix a property and classify the open bugs by this property, and then compute vector $x$ of what might be called bug preference values. Figure 7.1 depicts the preference values when the bug distinguishing property is bug severity.

Vector $x$ is portrayed here and henceforth by scaling it so that its maximal value is 100%. In this figure, this maximum is (unsurprisingly) obtained for bugs of severity blocker.

In addition, each column in the figure is adorned by the value of $t_j$ (see previous chapter) that was used in computing its preference value. In Figure 7.1, we see for example that blocker bugs were fixed 216 times. Recall that although the
computation of the preference value is numerically very accurate, it may introduce errors due to the substitution of $t_j$ for $E[t_j]$. The relative error of this substitution is in the order of $1/\sqrt{t_j}$.

Note the Kendall $\tau$ rank correlation coefficient$^1$ and its significance value$^2$ depicted in the figure. We see that overall, more severe bugs tend to be attended first, and that this tendency is statistically significant at the 95% significance level.

Studying the actual value of preference values we learn the somewhat surprising fact that the preference value of critical, major and normal and trivial are quite similar. The preference value of minor and enhancement type bugs is however significantly smaller than these.

Preference value values vs. bug priority are depicted in Figure 7.2.

Next we study the impact of bug-fix effort on the likelihood of the bug being selected to be fixed. Figure 7.3 compares the preference value values of bugs which required modification of different numbers of source files. We see that

---

$^1$Recall that this coefficient is a non-parametric statistics of a set of value pairs; the coefficient ranges between -1 in the case that the set represents a monotonically decreasing function and 1 in the case that this function is monotonically increasing.

$^2$computed by exhaustive counting of all permutations for small sets, and by the standard approximation for large sets.
Figure 7.2: Preference value vs. priority

Overall, programmers prefer to fix first bugs which require fewer files to fix (\(\tau = -0.64\)), however the dependency of the preference values in the number of files is not too strong: the lowest preference is about 60\% of the highest.

The same phenomena is found when we measure the effort by tokens rather than files, as can be seen in Figure 7.4. (Observe that since \(\text{FIX}^T\) spans quite a large range and a value of \(\text{FIX}^T\) is typically assumed by a very small number of bugs, preference values were computed with respect to a binned value of \(\text{FIX}^T\). Since the decay of \(\text{FIX}^T\) close to exponential (see Figure 6.4), binning of this metric was by its logarithm.)

The figure tells us that programmers tend to attend first to bugs which can be corrected by fewer additions: The highest preference value this time is about three times greater than the lowest; the absolute value of the Kendall \(\tau\) value is greater; and, it is even more statistically significant.
Figure 7.3: Preference value vs. $\text{FIX}^F$ (number of files changed in the first revision fixing the bug)

Figure 7.4: Preference value vs. $\text{FIX}^T$ (number of tokens added in the first revision fixing the bug)
7.2 Impact of future refixing effort on the selection of the next bug to fix

We now turn to the study of the impact of metrics of the refixing effort on the preference value. We continue to concentrate on the preference of the first fix of a bug.

Figure 7.5: Preference value vs. \textsc{refix}^R (number of future refixing revisions)

\textbf{Figure 7.5} depicts this value vs. \textsc{refix}^R.

The figure demonstrates that programmers generally prefer fixing those bugs which require fewer future refixing revisions, and the statistical significance of this trend is at least 95%. The relative difference of the preference levels is not meager: developers are about two times more likely to fix bugs with no further refixes than bugs with three future refixes.

\textbf{Figure 7.6} depicts the dependency of the preference value of a bug on \textsc{refix}^S—the number of files modified in refixes, but not in the original fix. Although the figure may visually suggest that programmer tend to prefer bugs whose \textsc{refix}^S value is smaller, we cannot ascribe statistical significance to this claim.

The plot of the preference value against \textsc{refix}^F (\textbf{Figure 7.7}) does not reveal that programmers tend to prefer fixing bugs which will require fewer future file modifications. In fact, the close to zero Kendall coefficient ($\tau = -0.05$) suggests that programmers are not able to distinguish between bugs based on this criteria.
Figure 7.6: Preference value vs. \textsc{refix}\textsuperscript{S} (number of files modified during refixes, but left intact during initial fix)

Figure 7.8 shows the impact of \textsc{refix}\textsuperscript{T}, the fourth metric of refixing effort, on the preference value for bug fixing.

Evidently, the impact of this metric is major: the \(-0.56\) value Kendall \(\tau\) coefficient is significant at the 99.9\% level, and, the values of the preference values span a large multiplicative range.

We conclude this section by drawing attention to a common phenomenon: All figures presented here, and to a limited extent, also Figure 7.3 and Figure 7.4 in the previous section, exhibit an increase of the preference value at the extreme right of the figure, i.e., it seems as if bugs whose fixing and refixing effort is the highest are more likely to be fixed than what we would expect by the preference value of bugs with high, but a bit more moderate effort. Admittedly, the preference value is never fully monotonic, and sporadically there are “spikes” which may require deeper research, e.g., the high preference value in \textsc{fix}\textsuperscript{T} = 2 (Figure 7.4). Still, the increase in the preference value at the right end of the spectrum seems to follow a consistent pattern.

We were unable to explain or even pinpoint this phenomenon. Further research is in place here to study of the nature of the class of bugs which require high refixing effort yet are quite attractive to programmer in the first fix. The small
Figure 7.7: Preference value vs. $\text{REFIX}^F$ (number of file modification operation during refixes)

number of bugs (examine the $t_j$ values) in the right-most column of the figures suggests that these research can make use of manual inspection.
Figure 7.8: Preference value vs. $\text{REFIX}^T$ (number of tokens added during refixes)
Chapter 8

Conclusions, discussion, and directions for further research

We now further the discussion of our findings on the preference value, draw conclusions, and point out directions for further research.

First, note that due to scaling, any single \( x \) that is computed with respect to some fixed property is meaningless when considered in isolation. What’s important is whether this value is large or small in comparison with the other values computed with respect to this property.

We have seen for example that programmers’ preference of first fixing blocker bugs is at least twice than that of all other severity levels (Figure 7.1). This finding does not mean that bugs of lower severity go unattended whenever blocker bugs are open. The reason is that in Eclipse, as in any other large project, there are many individuals engaged in maintenance and in further development of the project.

The large preference value of blocker bugs is to be interpreted with respect to the team of developers, rather than to any individual in it. The team, as a whole is two times more likely to fix blocker bugs than other bugs. The team’s preference does not however preclude individual members from investing their resources on other bugs and product enhancement.

It is interesting to see that the impact of other severity levels on the preference value is quite minimal. We leave for further research the verification of the obvious conjecture that in the eyes of software developers (at least those involved in the Eclipse project) the seven different severity levels defined by BugZilla degrade into three: blocker, ordinary and enhancement/minor.

A similar conjecture is raised by Figure 7.2. It seems as if there are two priority levels that matter: ordinary priority corresponding to levels P1, P2, and P3 and
Observe that Figure 7.1 exhibits an inversion in that bugs of minor severity receive about two thirds of the attention of bugs marked trivial, despite the “official” definition of severity levels in BugZilla in which minor bugs are more severe than trivial bugs.

This anomaly may suggest that bugs are marked trivial not because they are of trivial severity (as they should), but rather because they require a trivial amount of work (as the name suggests). We found that on average, $\text{FIX}^T$ of minor bugs was twice as high as that of trivial bugs, but clearly, more research is required to evaluate the quality of severity and priority judgments of bugs.

Second, recall that the computed $x$ values include a statistical error. This means that minute differences in $x$ values, e.g., the difference between $\text{REFIX}^R = 4$ and $\text{REFIX}^R = 5$ (Figure 7.5), are to be disregarded. Such small changes are likely to be the fruit of a statistical fluctuation.

Our extensive use of the Kendall $\tau$ rank correlation coefficient is designed to overcome statistical errors of this sort. In all of our findings $\tau$ was negative, but values closer to -1, represent a consistent decrease of the $x$ value with the increase of the property under inspection. The statistical significance level of the $\tau$ value tells us how likely it is that this trend is a result of a mere coincidence. What’s more important, is the trend that the $x$ values represent over the entire spectrum of the inspected property.

By examining these Kendall $\tau$ values in Figure 7.3 and Figure 7.4, we can say that with good certainty that the development team generally prefers dealing first with bugs that require less effort (i.e., lower $\text{FIX}^F$ and $\text{FIX}^T$), as in the famous shortest processing time first scheduling rule. Moreover, our findings also suggest that programmers are able to appreciate the programming effort (the flow of incoming bugs in our data set seems steady).

More importantly, the statistically significant $\tau$ value in Figure 7.8 tells us that the development team, as a whole, has a good appreciation of the amount of future programming effort for refixing the bugs, even at the time the bug is supposedly fixed, and that the team tends to delay fixing bugs which are more likely to require more refixing effort.

Further, the statistically significant $\tau$ value in Figure 7.5 tells us that the development team possesses an ability to detect recalcitrant bugs, i.e., bugs that require would require more refixing iterations (high $\text{REFIX}^R$) and that the team has the tendency to put off the initial fix of these bugs.

We know from Table 6.1 that effort spent in the first fix is a poor indicator of future fixing effort. Further, previous research tells us that bugs that require
refixes are more likely to be of higher severity. But, since bugs of higher severity are more likely to be treated first (Figure 7.1) we can conclude that severity cannot be a predictor of \( \text{refix}^R \). More research is therefore in place to identify the means by which programmers can estimate the amount of refixing effort and the number of refixing iterations.

This research direction may even lead to a translation of the programmer’s ability to detect incomplete bug fixes into automatic tools that would point out such fixes to software developers, and perhaps even propose ideas on how to make the fix more complete.

Third, we ask whether it is possible to extract useful information from Figure 7.6 and Figure 7.7 (representing, respectively, the dependency of \( x \) on \( \text{refix}^S \) and \( \text{refix}^F \)) despite the Kendall \( \tau \) values being statistically insignificant?

Recall that the error in the \( t_j \) value is proportional to \( 1/\sqrt{t_j} \). Re-examining the figures, we see that the \( t_j \) value in the first three columns of these is no less than 859, hence the error in \( E[t] \) is in the order of 3\%. It makes sense to assume that the error in the corresponding \( x \) value is similarily small. (In fact, the iterative method of finding the \( x \) values, i.e., equations (2.8) suggests that the error in \( x_j \) is linear in the error in \( t_j \).)

We see that the difference in the respective \( x \) values is not meager. The first three columns of the figures therefor suggest the development team has some rough estimate of \( \text{refix}^F \) and \( \text{refix}^S \).

The crude estimate that \( \text{refix}^F > 0 \) is not interesting since it can be explained by the estimate we demonstrated already of \( \text{refix}^R \). However, we believe that it is remarkable that programmers have the intuition that their bug fixes may be incomplete and require touching files not included in the first fix.

Fourth, we should mention the “spikes” seen occasionally in the figures, the most prominent of these are the high values of \( x \) in Figure 7.8 for \( \text{refix}^T = 1 \) and \( \text{refix}^T = 4 \). These may be attributed to two factors:

- individual statistical error which is more major for small values of \( t_j \), e.g., in the case \( \text{refix}^T = 1 \), we have \( t_j = 33 \) and the error is in the order of 17\%; and,

- the fact that together the figures represent a large number of computed \( x \) values, which increases the probability that one or two of these would exhibit a statistically significant deviation from its expectation.

It should be assumed therefore that re-doing the analysis on a number of major code base and accompanying bugs’ database is likely to remove these spikes.
One may also ask whether the general findings here with regard to Eclipse are applicable to other projects.

Table 8.1 compares Eclipse with two other projects, in comparing Kendall’s rank correlation coefficient of preference values with respect to all metrics for the three projects.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Eclipse Kendall τ</th>
<th>Sig.</th>
<th>Mozilla Kendall τ</th>
<th>Sig.</th>
<th>Webkit Kendall τ</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Severity</td>
<td>-0.81</td>
<td>95%</td>
<td>-0.81</td>
<td>99%</td>
<td>-0.52</td>
<td></td>
</tr>
<tr>
<td>Priority</td>
<td>-0.60</td>
<td></td>
<td>-1.00</td>
<td>99%</td>
<td>-0.40</td>
<td></td>
</tr>
<tr>
<td>$\text{FIX}^F$</td>
<td>-0.64</td>
<td>95%</td>
<td>-0.51</td>
<td>95%</td>
<td>-0.60</td>
<td>99%</td>
</tr>
<tr>
<td>$\text{FIX}^T$</td>
<td>-0.71</td>
<td>99.9%</td>
<td>-0.23</td>
<td></td>
<td>-0.62</td>
<td>99.9%</td>
</tr>
<tr>
<td>$\text{REFIX}^R$</td>
<td>-0.61</td>
<td>99.9%</td>
<td>-0.94</td>
<td>99%</td>
<td>0.05</td>
<td></td>
</tr>
<tr>
<td>$\text{REFIX}^S$</td>
<td>-0.38</td>
<td>95%</td>
<td>-0.53</td>
<td>95%</td>
<td>-0.29</td>
<td></td>
</tr>
<tr>
<td>$\text{REFIX}^F$</td>
<td>-0.05</td>
<td>99.9%</td>
<td>-0.56</td>
<td>95%</td>
<td>-0.38</td>
<td></td>
</tr>
<tr>
<td>$\text{REFIX}^T$</td>
<td>-0.56</td>
<td>99.9%</td>
<td>-0.37</td>
<td>95%</td>
<td>-0.41</td>
<td>95%</td>
</tr>
</tbody>
</table>

Table 8.1: Kendall’s rank correlation coefficient of preference values with respect to all metrics in the studied projects.

There are a number of interesting facts that the table tells us. First, we see that Kendall’s $\tau$ is negative for all projects and all metrics. Specifically this means that the fact that programmers have the ability to predict the tenacity of a bug, is far from being unique to any idiosyncrasies of the Eclipse development culture. (It may still be particular to open source projects.)

Second, we see that in Mozilla, the coefficient’s value with respect to bugs’ priority is -1 and that its significance value is very high. This means that the development culture in Mozilla is very respectful of bug priority values. In contrast, it appears that the influence of bug priority on triaging policy in WebKit is as meager as in Eclipse.

Third, we see that the impact of $\text{REFIX}^F$ on bug triaging policy which was small and even negligible in Eclipse is much greater in the two other projects, even significantly so in one of the two other projects.

Fourth, it is a bit interesting that Kendall’s $\tau$ is so small for $\text{REFIX}^R$ in WebKit. Further, project specific study, may be in place here. We leave this for future research.

We finally remark that it should be valuable to compare our results with the way programmers view their own triaging policy (a task which requires very dif-
ferent analysis techniques than those described here). Unfortunately, our attempt to recruit Eclipse programmers for this task had limited success.

The results, based on the responses of six such programmers are, as might be expected, not statistically significant, but still may be worthy of mention. Almost all programmers indicated that they handle five or more (sometimes much more) bugs concurrently, that they prefer fixing bugs which require fewer lines of code, residing in fewer files. Interestingly, programmers tended to disagree, sometimes strongly disagree with the statements that priority (severity) are prime factors in their bug triaging policy. We leave the continuation of this direction to further research.
Part III

How Do Classes Choose Their Parents?
Chapter 9

Data Corpus

The corpus of data used in our study consists of 16 open-source JAVA software artifacts drawn from a variety of application domains. All artifacts are being actively developed at the time of writing, and all offer to the public their managed source code repository.

All code repositories were available using sub-version [26], except Eclipse project which was available through CVS\(^1\) and converted to sub-version by us.

Concentrating on code evolution, we selected artifacts whose published development history features a substantial number of versions. The particular artifacts used were:

(A) *Eclipse*: the Eclipse platform project.
(B) *RSSOwl*: a graphical news aggregator application built on top of Eclipse;
(C) *SQuirrel SQL*: a graphical database administration tool;
(D) *Jython*: a JAVA implementation of the PYTHON programming language;
(E) *Mojarra*: the reference implementation of JavaServer Faces—a framework for assisting in writing JavaServer applications;
(F) *Castor*: a framework for binding JAVA objects, XML files and databases;
(G) *Vuze*: a BitTorrent client application;

\(^{1}\)http://cvs.nongnu.org/
Table 9.1: Software artifacts comprising the data corpus

(H) **Jitsi**: a voice over IP, videoconferencing, and instant messaging application;

(I) **iText**: a library for the creation and manipulation of PDF files;

(J) **Apache Derby**: part of Apache database project (also distributed as “Java DB” by Oracle);

(K) **Apache Ant**: a library and command-line tool for automating software build process (incidentally, Apache Ant was one of the three artifacts in the Turnu et. al. study);

(L) **Xalan**: an implementation of the XPATH and XSLT languages;

(M) **Xerces**: a collection of libraries for handling XML structured data and files;

(N) **DrJava**: a lightweight pedagogical IDE for Java;

(O) **aTunes**: a player and manager of audio files; and,

(P) **weka**: a popular suite of machine learning software written in Java.
These artifacts were retrieved from the following public repositories: source-forge.net from which RSSOwl, SQuirreL SQL, Jython, iText, DrJava, aTunes, jEdit and weka were obtained; apache.org including Apache Derby, Apache Ant, Xalan, Xerces, and Tomcat; java.net—source to Mojarra and Jitsi; while Castor and Vuze were fetched from, respectively, codehaus.org and vuze.com:

Table 9.1 summarizes the artifacts constituting the corpus. For reproducibility sake, the table makes an account of the range of versions (as numbered by the sub-version system) that were investigated.

The next pair of columns provides some size characteristics of the artifacts and the history we explored: The number of classes\(^2\) reported in the table is that found in the artifacts’ final version. Our analysis ignored inner, local and anonymous classes. Non-public classes, i.e., classes which may occur in the same file as other public classes were ignored as well.

Observe that the total number of versions investigated may be smaller than the versions’ range since some intermediate versions may have not been stored in the repository.

We can see that the artifacts were not meager in size, and that each yielded at least a thousand versions for the study. The median number of classes was about a thousand, while variety of the number of classes, as measured by the Median Absolute Deviation (M.A.D) statistics is not small.

The overall size of the corpus was more than sixty thousand classes, and, overall, we studied over one hundred thousand development versions.

In our study, each of the versions was fetched by our sub-version client. Full semantical parsing of the source proved to be not only resource costly but also a challenge in face of the myriad of changing dependency requirements made by the artifacts. Instead, we wrote a lightweight and forgiving parser, which, under the assumption of a syntactically correct input, generated many essential statistics of all classes found in the repository, including number of methods, fields, constructors, etc. The parser also generated the inheritance hierarchy of each of the downloaded versions.

Using this information, a change log was then produced for each of the artifacts, including an entry for each change to a class in the artifact. The “Per Version” column in Table 9.1 presents the average version granularity of each of the artifacts, that is, the average number of classes that were modified in anyway in a version. We see that granularity is typically small, ranging between 2.4 and

\(^2\)Here the term “class” shall refer to JAVA classes, interfaces, enums, and even annotations. As usual, the bulk of classes in the repositories were classes.
9.9 classes modified by a version. This small granularity makes it possible for us to analyze each change to a class in the context of the previous version, i.e., ignoring other co-occurring modifications.

Class changes were further classified as either an insertion (creation of a new class), an update (in which the body of a class was changed), deletion (in which the class was removed), or renaming (in case the class name changed). The final four columns of Table 9.1 give the relative numbers of each of these change kind. As expected, most of the changes are updates, with a class being added to a repository once for about eight updates. Renaming is very infrequent and in some of the applications does not occur. Deletions are usually rare with the notable exceptions of Itext and Quirrel-sql, which probably underwent a massive rewrite.

Our evolution analyzer follows this change log, and updates the model as necessary for each of the respective changes. As should be clear, our method of analysis is not sensitive to major rewrites (as might have happened in Itext and Squirrel-sql). However, if such a rewrite also represents a change of the artifact “culture”, this change could not be detected by the analyzer.
Chapter 10

Power Law Distribution of the Number of Immediate Extenders

Recall that the JAVA programming language has three variants of inheritance: a class may extend another class; it may implement one or more interfaces; and, an interface may implement another interface.

This work analyzes only the first of these variants for the following reasons: first, since the semantics of these variants are quite different which gives reasons to believe that programmers employ different criteria for each of these. Second, the number of interfaces in our data corpus is so small, that statistically significant conclusions about inheritance from these are unlikely. Third, some of the properties we study are meaningless for interfaces.

We now turn to the study of the distribution of the number of immediate extenders that a class has, henceforth denoted #Extenders. This metric counts both class-class and interface-interface extends relationships. However, since it does not include class-interface pairs which are tied together in an implements relationship, it is not exactly the NOC metric.

Our findings largely confirm Concas et. al.’s [8] namely, that #Extenders follows a power-law. The affirmation that the distributions of this metric in the corpus are, or close to being, power-law, makes the preferential attachment hypothesis plausible for these.

10.1 Cautions

As usual, the fundamental technique for detecting a power law distribution is based on the fact that if \( P(b) \propto b^{-\alpha} \), then \( P(b) \) takes a straight line shape when drawn vs. \( b \) on a doubly logarithmic scale. Two comments apply before we begin.
First, recall that if a process is governed by a Yule process, then the resulting distribution is only approximated by a power law. Specifically, we obtain a skewed version, \( P(b) \propto (b + c)^{-\alpha} \) for some constant \( c \), and even this skewed version does not apply to small values of \( b \).

Secondly, when a distribution is surmised from observations, then \( P(b) \) suffers from statistical fluctuations, especially in the high end range of the random variable where the probabilities are small. To smooth out these, we use the standard technique (see e.g., [9]) of replacing \( P(b) \) by the complementary cumulative distribution function (CCDF), a process which reduces, but not does not eliminate entirely, the noise in large values of \( b \).

![Figure 10.1: Complementary cumulative distribution of the simulated preferential attachment process with \( b_0 = 0; a = 1; c = 1; n = 1,000 \).](image)

Recall that the CCDF of a random variable \( b \) is defined by the integral

\[
\text{CCDF}(b) = \int_b^{\infty} P(b')db'
\]

or by the sum

\[
\text{CCDF}(b) = \sum_{x}^{\infty} P(b')
\]
when \( b \) is discrete. Also recall that if a distribution is power law, then so is its CCDF. Therefore, the CCDF should also show as a straight line on the doubly logarithmic plane.

To better appreciate these two concerns, consider Figure 10.1 which shows the CCDF of a simulation of a preferential attachment process process with the following characteristics: \( b_0 = 0 \), i.e., a newly generated urn does not contain any balls in it), \( a = 1 \) i.e., there is one ball thrown for each urn creation, and \( c = 1 \), i.e., the “preference” of an urn with \( b \) balls is \( b + 1 \), and \( n = 1000 \), i.e., the basic preferential attachment step was applied 1,000 times.

The red points in the figure represent observed values. The straight line interpolation, which assumes a non-skewed power law, is drawn in blue, while the skewed power law interpolation is green.

We see that the skewed interpolation achieves a much closer fit than the non-skewed interpolation. Despite the data coming from a clean simulation which strictly adheres to the preferential attachment regime, statistical noise makes even the skewed interpolation imperfect.

### 10.2 Distribution of \#Extenders in the Corpus

Figure 10.2 depicts, using a doubly logarithmic scale, the CCDF of \#Extenders in the artifacts that comprise the corpus.

Even a superficial visual inspection of the figure suggests that most distributions obey a power law. More often than not, we see that the red observation points can be fitted closely with a straight line. In some of the sub-figures, the fitting appears to be tight, while in other, deviations do appear.

The green curves in the figure represent a skewed power law interpolation, while the blue interpolation curves are similar, except that the interpolation omits some outliers (see Appendix 14 for details). Again a visual inspection suggests a good fit of these interpolating curves.

This impression is confirmed by Table 10.1, which provides what is known as the confidence level of the fit.\(^1\)

The first five columns of the table give the vital data of the green curves: the “\( k \)” column is the number of data points in the interpolated CCDF, “Conf.” gives the confidence level, while \( \alpha \) and \( c \) define the skewed power law. We see that for four applications (K), (L), (M) and (N) fitting was at the 100\% level (this finding on (K) confirms Turnu et. al.’s findings) while two other artifacts, (I), and (O) came close at the 98\% or higher level.

Fitting of the remaining 10 artifacts was not as good. Recalling that the data is the fruit of a concentrated human design activity, rather than a natural physi-
Full Set | Outliers Removed
---|---
Id | k | Conf. | $\alpha$ | c | Conf. | o | $\alpha$ | c
A | 98 | 0% | 2.2 | 0.28 | 100% | 1 | 2.3 | 0.38
B | 38 | 0% | 1.4 | 0.00 | 94% | 10 | 1.5 | 0.01
C | 32 | 12% | 1.8 | 0.04 | 98% | 7 | 1.8 | 0.03
D | 18 | 0% | 2.2 | 0.30 | 99% | 5 | 2.2 | 0.21
E | 18 | 15% | 2.3 | 0.24 | 96% | 9 | 2.2 | 0.19
F | 21 | 1% | 2.3 | 0.24 | 95% | 6 | 2.4 | 0.34
G | 19 | 67% | 2.4 | 0.20 | 99% | 2 | 2.4 | 0.21
H | 20 | 54% | 2.2 | 0.14 | 94% | 6 | 2.2 | 0.15
I | 12 | 98% | 2.5 | 0.35 | 98% | 0 | 2.5 | 0.35
J | 20 | 79% | 2.6 | 0.37 | 97% | 1 | 2.6 | 0.38
K | 18 | 100% | 2.3 | 0.30 | 100% | 0 | 2.3 | 0.30
L | 16 | 100% | 2.4 | 0.34 | 100% | 0 | 2.4 | 0.34
M | 16 | 100% | 2.4 | 0.35 | 100% | 0 | 2.4 | 0.35
N | 17 | 100% | 2.2 | 0.23 | 100% | 0 | 2.2 | 0.23
O | 19 | 99% | 1.9 | 0.05 | 99% | 0 | 1.9 | 0.05
P | 17 | 40% | 2.4 | 0.29 | 92% | 10 | 2.4 | 0.25

Table 10.1: Skewed power-law interpolation (of all data, and outliers removal) of the CCDF $\#\text{Extenders}$ in the artifacts of the corpus.

cal process, we anticipate more outliers than the usual [8]. The remaining four columns in Table 10.1 summarize the results of an interpolation while successively removing outliers to obtain higher confidence levels. The “o” column is the number of outliers that had to be removed to reach the 95% confidence level.

Observe that in Eclipse (A), a removal of just one outlier lead to a dramatic improvement of the confidence level. However, in other artifacts, a substantial number of outliers had to be removed to approach the desired confidence level possible. The most extreme case was weka (P), in which even more than half the observations points were marked as outliers, the confidence level did not exceed 92%.

1Take note that in saying that an interpolation is in the 95% confidence interval, we do not mean that the fit is correct with 95% probability; should this be the case, all that is known is that the null hypothesis cannot be rejected assuming the usual 0.05 confidence level, where the null hypothesis being the height of each observation point (which must be an integer) is distributed binomially around the value determined by the interpolation.
Figure 10.2: Complementary cumulative distribution of #Extenders in the corpus (doubly logarithmic planes).
Figure 10.3: Complementary cumulative distribution of \#Extenders for the rest of the corpus.
Chapter 11

Examining the Preferential Attachment Hypothesis

Having confirmed that the distributions of #Extenders in the corpus follow, more or less, a power law distribution, we ask now whether these distributions are the product of a preferential attachment process. More generally, we may ask whether a newly created class, or a class being re-parented, is more likely to choose as a parent a class that has a greater number of children.

Figure 11.1 depicts, on the logarithmic plane, the preference values, as computed by evolution monitoring, vs. #Extenders.

Observe that the y-axis in the figure has no units. The reason is that the preference values are determined only up to a multiplicative scaling factor. As usual, the algorithm is 10-based, so each successive horizontal lines in the grid represent a ten-fold increase in preference values.

Even a quick look at Figure 11.1 reveals that in all software artifacts, classes that have more extenders, are more likely to parent again. We need however to make a small diversion to discuss binning and branches, before deriving any further conclusions.

11.1 Binning

As mentioned above in Chapter 4.1, the numerical process requires the expected values $E[t_0], \ldots, E[t_\ell]$; an inherent error arises from the fact that we must use the observed values $t_0, \ldots, t_\ell$. For those values of $b$ for which $|t_j|$ is large, the approximation $t_0 \approx E[t_0]$ should be accurate. This approximation is however clearly invalid when $t_0 = 1$, which is the typical case for the large values of $b$. 
Another issue is that the distributions of adjacent entries in $t_b = 1$, are not independent: consider for example the case in which at the end of the software evolution there is a class with 100 extenders, while all other classes have much fewer extenders. In this case, then necessarily $t_{99} = 1$.

To deal with this adverse effect we applied a binning process prior to computing the preference values presented in Figure 11.1. In this process, adjacent values in the $t$ are added together until their sum reaches a predefined threshold. The underlying assumption that justifies this process is that we expect $E[t_b]$ to be close both to $E[t_{b+1}]$ and $E[t_{b-1}]$. Thus, we can make the approximation

$$E[t_{b-1} + t_b + t_{b+1}] \approx t_{b-1} + t_b + t_{b+1}.$$ 

In all applications of evolution monitoring we applied a threshold of 10, except that if the vector had more than 200 entries, this threshold was allowed to increase to 50.

### 11.2 Branches

A branching phenomenon can be observed in some of the plots of Figure 11.1, e.g., in artifacts (B), (D), (F) (G), (I) and (J). Branching means that a substantial number of the points appear to fall on a distinct cluster which lies above the main cluster of points. The distance between the main and the secondary cluster can even reach two or three orders of magnitude.

This phenomenon is a product of the analysis method; it does not indicate any tendency of classes with a certain number of extenders to be more attractive parents than classes with a similar number of extenders. In fact, the same branching phenomenon can be observed while monitoring a simulated preferential attachment process.

Roughly speaking, branching occurs if the set of indices $0, \ldots, \ell$ can be divided into two or more sets such that there is little “interaction” between buckets whose size belongs to different sets. In this cases, the numerical solution tends to generate an independent solution for each of these sets, and combine them together after applying different normalization to these.

Binning reduces branching, but, evidently, it does not eliminate it entirely.

### 11.3 Generalized Preferential Attachment

Some of the curves, e.g., Figure 11.1(A), seem to depict a straight line, thus supposedly confirming the preferential attachment hypothesis. However, a
straight line in the doubly logarithmic plane means only that the preference value is proportional to \( b^\gamma \), while vanilla preferential attachment requires that \( \gamma = 1 \).

Examine Figure 11.2, depicting the computed preference values of artifacts Eclipse (A) and Mojarra (E) in the normal-, non-logarithmic-, scale.

Although the preference values do appear increasing in both cases, it is hard to imagine a very convincing linear fit.

The situation is even worse if we examine Figure 11.3 depicting the entire range, i.e., classes with any number of children.

In view of Figure 11.2 and Figure 11.3, and in view of the errors due to branching and binning, we are inclined to abandon the hope of showing that

\[ x_b \propto (b + c) \]

for some constant \( c \), as required for confirming the vanilla preferential attachment process. Instead we will try to confirm a weaker characterization, namely that

\[ x_b \text{ tends to be increasing with } b \]

Three arguments may justify this weakening of the model.

1. Our hand waving argument in support of preferential attachment process in the selection of which class to extend, does not necessarily imply a linear dependency of the preference value in the current number of extenders.

2. The literature discusses several variants of preferential attachment which also lead to fat tailed distribution. For example, if \( x(u) = 1 \), that is, the probability of selecting an urn during a throw does not depend on the number of balls in it then the resulting distribution is exponential [2]. And, in the case that

\[ x(u) = (b + c)^\gamma \]

for a constant \( \gamma < 1 \), then the resulting distribution is a stretched exponential [17], specifically,

\[ P(b) \propto b^{-\alpha}(e^{-R(b)}). \] \hspace{1cm} (11.1)

where \( R(b) \) is an expression which asymptotically behaves like \( b^{1-\alpha} \). Note that a generalized preferential attachment with \( \gamma \neq 1 \) also shows as a straight line with slope \( \gamma \) in the logarithmic plane.

3. The difficulties, which were also demonstrated above, of fitting a pure power-law curve against actual data lead some investigators to the conclusion that although the distribution is fat-tailed, it is not characterized best by power-law [3].
Table 11.1: Kendall $\tau$ and Pearson-$\rho$ coefficients computed from the preference values of the evolution of $\#\text{Extenders}$ in the artifacts in the corpus.

<table>
<thead>
<tr>
<th>Artifact</th>
<th>$n$</th>
<th>$m$</th>
<th>$\tau$</th>
<th>$\rho$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>11,696</td>
<td>197</td>
<td>0.69</td>
<td>0.50</td>
</tr>
<tr>
<td>Rssowl</td>
<td>5,330</td>
<td>198</td>
<td>0.56</td>
<td>0.38</td>
</tr>
<tr>
<td>Squirrel-sql</td>
<td>1,200</td>
<td>71</td>
<td>0.60</td>
<td>0.27</td>
</tr>
<tr>
<td>Jython</td>
<td>1,043</td>
<td>56</td>
<td>0.66</td>
<td>0.63</td>
</tr>
<tr>
<td>Mojarra</td>
<td>950</td>
<td>44</td>
<td>0.78</td>
<td>0.02</td>
</tr>
<tr>
<td>Castor</td>
<td>885</td>
<td>53</td>
<td>0.57</td>
<td>0.38</td>
</tr>
<tr>
<td>Vuze</td>
<td>659</td>
<td>40</td>
<td>0.69</td>
<td>0.11</td>
</tr>
<tr>
<td>Jitsi</td>
<td>560</td>
<td>30</td>
<td>0.77</td>
<td>0.86</td>
</tr>
<tr>
<td>Itext</td>
<td>542</td>
<td>30</td>
<td>0.60</td>
<td>-0.03</td>
</tr>
<tr>
<td>Derby</td>
<td>466</td>
<td>40</td>
<td>0.42</td>
<td>0.04</td>
</tr>
<tr>
<td>Ant</td>
<td>388</td>
<td>27</td>
<td>0.79</td>
<td>0.77</td>
</tr>
<tr>
<td>Xalan</td>
<td>245</td>
<td>9</td>
<td>0.61</td>
<td>0.38</td>
</tr>
<tr>
<td>Xerces</td>
<td>238</td>
<td>8</td>
<td>0.64</td>
<td>0.28</td>
</tr>
<tr>
<td>Drjava</td>
<td>234</td>
<td>12</td>
<td>0.73</td>
<td>0.82</td>
</tr>
<tr>
<td>Atunes</td>
<td>212</td>
<td>18</td>
<td>0.76</td>
<td>0.16</td>
</tr>
<tr>
<td>Weka</td>
<td>107</td>
<td>9</td>
<td>0.83</td>
<td>0.76</td>
</tr>
</tbody>
</table>

| $Mean$     | 1547.2| 52.6  | 0.67   | 0.40   |
| $Median$   | 551.0 | 35.0  | 0.68   | 0.38   |
| $M.A.D$    | 325.5 | 19.5  | 0.08   | 0.26   |

11.4 Confirming the Weak Preferential Attachment Hypothesis

To test whether the preference value of a class increases with the number of extenders it has, we computed the $\tau$ coefficient for each of the artifacts in the corpus, i.e., in the point sets depicted in Figure 11.1. The results, along with the Pearson $\rho$ correlation values for the empirical (non-logarithmic) data, are tabulated in Table 11.1.

Column “$n$” in the table is the number of models produced in the course of evolution monitoring, i.e., the total number of $\text{extends}$ relationships generated in the artifact’s history. The $\ell + 1$ column is the number of distinct values after the above mentioned binning. The table employs the convention of **boldfacing** Kendall-tau values which are statistically significant at the 0.99 level.
We see that all but two of the Kendal-tau values are statistically significant; they are all positive, and all are greater than 0.4. Their mean is 0.67 while the median is 0.68. Most values are close to the median, with the medianic absolute deviation (MAD) being 0.08.

We have thus confirmed the weak preferential attachment hypothesis.

The Pearson-$\rho$ coefficient can be used to test whether the dependency is linear (i.e., the standard version of the preferential attachment hypothesis). Many of the artifacts in the corpus feature $\rho$ values which are very close to zero. Recalling also that Pearson $\rho$ values up to $\pm 0.5$ correspond to scatters which do no resemble straight lines, leads to the conclusion that empirical support in our corpus of linear dependency is quite limited.
Figure 11.1: Preference value, as computed by evolution monitoring, vs. \#Extenders in the corpus (log-log scale)
Figure 11.2: Preference values of artifacts Eclipse (A) and Mojarra (E) in a linear scale (classes with at most 40 children).

Figure 11.3: Preference values of artifacts Eclipse (A) and Mojarra (E) in a linear scale (all classes).
Chapter 12

Other Candidate Predictors of Parenting Probability

In this chapter we search for other class properties that predict the likelihood of it being a parent. Observe that the abstract framework evolution monitoring is applicable also when the values present in matrix $U$, are not the number of balls in an urn, but rather any other urn property. Thus, evolution monitoring can be used to assess the quality of any urn property as a predictor of the urn being selected.

In our domain of interest, we can apply evolution monitoring not only to compute the dependency of the preference value on the number of extenders, but also on any other class property. When considering, e.g., the number of methods in a class property, any model encountered during the evolution is defined by a vector, in which location $j$ is the number of classes, at this point in the evolution which has $j$ methods.

In doing so, $n$, the number of models observed in the monitoring process is always the same. The number of different predictor values, $\ell + 1$, depends however on the number of distinct values of the predictor encountered during the monitoring.

We first examine the prediction quality of four other natural class properties: the number of constructors ($\#\text{Ctors}$), fields ($\#\text{Fields}$), methods ($\#\text{Methods}$) and tokens ($\#\text{Tokens}$), that the candidate parent class has. The first four numerical column in Table 12.1 provide the Kendall-$\tau$ values of these potential predictors in the data corpus.

A quick comparison of these four columns with Table 11.1 shows that each of $\#\text{Ctors}$, $\#\text{Fields}$, $\#\text{Methods}$, $\#\text{Tokens}$ is inferior to $\#\text{Extenders}$ in predicting the preference value: there are fewer statistically significant $\tau$ values, fewer positive values, and
Table 12.1: Kendall $\tau$ coefficients of the correlation of each of #Ctors, #Fields, #Methods, #Tokens, Age, Dust and #Changes class properties with the respective preference values, as computed by monitoring the evolution of #Extenders.

<table>
<thead>
<tr>
<th>Id</th>
<th>Artifact</th>
<th>#Ctors</th>
<th>#Fields</th>
<th>#Methods</th>
<th>#Tokens</th>
<th>Age</th>
<th>Dust</th>
<th>#Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Eclipse</td>
<td>0.20</td>
<td>-0.13</td>
<td>0.46</td>
<td>0.35</td>
<td>-0.21</td>
<td>-0.75</td>
<td>0.77</td>
</tr>
<tr>
<td>B</td>
<td>Rssowl</td>
<td>0.80</td>
<td>-0.42</td>
<td>-0.13</td>
<td>-0.02</td>
<td>0.04</td>
<td>-0.38</td>
<td>0.57</td>
</tr>
<tr>
<td>C</td>
<td>Squirrel-sql</td>
<td>1.00</td>
<td>0.38</td>
<td>0.13</td>
<td>-0.03</td>
<td>-0.15</td>
<td>-0.75</td>
<td>0.79</td>
</tr>
<tr>
<td>D</td>
<td>Jython</td>
<td>0.33</td>
<td>-0.06</td>
<td>0.61</td>
<td>0.39</td>
<td>0.13</td>
<td>-0.46</td>
<td>0.64</td>
</tr>
<tr>
<td>E</td>
<td>Mojarra</td>
<td>-0.67</td>
<td>0.32</td>
<td>0.52</td>
<td>0.26</td>
<td>0.34</td>
<td>-0.75</td>
<td>0.88</td>
</tr>
<tr>
<td>F</td>
<td>Castor</td>
<td>0.20</td>
<td>0.44</td>
<td>0.53</td>
<td>0.29</td>
<td>0.01</td>
<td>-0.51</td>
<td>0.75</td>
</tr>
<tr>
<td>G</td>
<td>Vuze</td>
<td>0.60</td>
<td>0.05</td>
<td>0.11</td>
<td>0.07</td>
<td>-0.56</td>
<td>-0.77</td>
<td>0.74</td>
</tr>
<tr>
<td>H</td>
<td>Jitsi</td>
<td>0.87</td>
<td>-0.36</td>
<td>0.27</td>
<td>0.20</td>
<td>-0.36</td>
<td>-0.57</td>
<td>0.75</td>
</tr>
<tr>
<td>I</td>
<td>Itext</td>
<td>0.14</td>
<td>0.46</td>
<td>0.60</td>
<td>-0.30</td>
<td>0.34</td>
<td>-0.50</td>
<td>0.67</td>
</tr>
<tr>
<td>J</td>
<td>Derby</td>
<td>0.67</td>
<td>0.02</td>
<td>0.70</td>
<td>0.57</td>
<td>0.26</td>
<td>-0.88</td>
<td>0.82</td>
</tr>
<tr>
<td>K</td>
<td>Ant</td>
<td>0.00</td>
<td>0.13</td>
<td>0.54</td>
<td>0.13</td>
<td>0.76</td>
<td>-0.83</td>
<td>0.94</td>
</tr>
<tr>
<td>L</td>
<td>Xalan</td>
<td>0.40</td>
<td>0.77</td>
<td>0.57</td>
<td>0.52</td>
<td>-0.30</td>
<td>-0.78</td>
<td>0.45</td>
</tr>
<tr>
<td>M</td>
<td>Xerces</td>
<td>0.67</td>
<td>0.51</td>
<td>0.50</td>
<td>0.30</td>
<td>-0.35</td>
<td>-0.58</td>
<td>0.85</td>
</tr>
<tr>
<td>N</td>
<td>Drjava</td>
<td>1.00</td>
<td>0.11</td>
<td>0.02</td>
<td>-0.08</td>
<td>-0.66</td>
<td>-0.63</td>
<td>0.25</td>
</tr>
<tr>
<td>O</td>
<td>Atunes</td>
<td>0.33</td>
<td>-0.07</td>
<td>0.29</td>
<td>-0.26</td>
<td>-0.56</td>
<td>-0.64</td>
<td>0.52</td>
</tr>
<tr>
<td>P</td>
<td>Weka</td>
<td>-1.00</td>
<td>0.60</td>
<td>0.44</td>
<td>0.28</td>
<td>-0.35</td>
<td>-0.82</td>
<td>0.86</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td>0.35</td>
<td>0.17</td>
<td>0.39</td>
<td>0.20</td>
<td>-0.10</td>
<td>-0.66</td>
<td>0.70</td>
</tr>
<tr>
<td>Median</td>
<td></td>
<td>0.37</td>
<td>0.12</td>
<td>0.48</td>
<td>0.27</td>
<td>-0.18</td>
<td>-0.69</td>
<td>0.75</td>
</tr>
<tr>
<td>M.A.D</td>
<td></td>
<td>0.30</td>
<td>0.26</td>
<td>0.13</td>
<td>0.13</td>
<td>0.26</td>
<td>0.12</td>
<td>0.10</td>
</tr>
</tbody>
</table>

both the mean and the median are smaller. Also, the variety of the $\tau$ values in the artifacts in the corpus, as measured by the MAD statistics, is greater than in #Extenders. The best predictor is #Methods while the worst is #Fields. We can still surmise that larger classes (where size can be measured in #Ctors, #Fields, #Methods or #Tokens) are slightly more likely to be extended than smaller classes.

The next three numerical columns in Table 12.1 provide the $\tau$ values of three other class properties. These properties pertain to the class’s history rather than to its current state:

1. **Age.** the number of versions since the class was first introduced into the artifact;
2. **Dust.** the number of versions since the class was last modified; and,
3. \textit{Changes.} the number of previous versions in which the class was changed. All three metrics are defined with respect to the current version.

Assessing the quality of history metrics as predictors of parenting probability, we see that \textit{Changes} is a better predictor than \textit{Extenders}—classes which collected more changes than others are more likely to be extended. The Dust property comes second, but it is also a better predictor than \textit{Extenders} (although not as good as \textit{Changes}). We have that classes which were recently changed are more likely to be extended.

\textit{Age} is far from being a good predictor of a class being extended. If anything, we may conjecture that newer classes are (very) slightly more likely to be extended.
Chapter 13

Conclusions

In this part we concentrated in one important software metric: \#Extenders, which is almost identical to the famous NOC metric. Our first objective was to confirm the preferential attachment hypothesis with regard to the evolution of this metric, that is, that the probability that a class is selected as a base class, has a *linear* dependency on the number of extenders the class currently has. To do so, we developed *evolution monitoring*, a new analysis technique, for computing exact dependency.

We were unable to demonstrate linear dependency, the main obstacle probably being statistical error. Instead we resorted to a weaker version of the hypothesis, namely the probability that a class is selected as a base class, *increases* with the number of extenders the class currently has. This weaker version was confirmed by examining the Kendall-tau coefficient and its statistical significance.

We saw that in all 16 software artifacts in the corpus, the Kendall tau coefficient was positive; it was statistically at the 99% level in all but two artifacts, the mean was 0.67 while the median is 0.68. In contrast, linear dependency (Pearson-rho coefficient greater than 0.5), was demonstrated in only four artifacts.

The size of a class (as measured by either \#Ctors, \#Fields, \#Methods, and \#Tokens) proved to be a much weaker predictor of extendibility.

We found that the age of a class, is not very telling of its extendibility. In contrast, the number of changes applied to a class, and the recency of these changes, are strong predictors of a class extendibility, even more the current number of classes that extend it.

In view of these results, we believe that a simple preferential attachment process is not likely to explain extensibility. The results seem to be pointing at a more elaborate model, by which extendibility is not an issue of age or popularity, but
rather of adaptability. In this model, classes are not necessarily created extensible, but rather evolve into serving this objective with enhancement made during the evolution process.

What would be the impact of such a model? Gil and Cohen suggested using the $\alpha$ value of power law distribution as means for calibrating metrics against the “common programming practice”. Later on, Baxter et. al. [3] suggested that the understanding of distribution could be used to evaluate the absolute quality of design:

*If we are seeing different distributions due to different designs, if we could understand how aspects of the design related to the kind of distribution exhibited, there is the potential for developing a quantitative measure for design quality. Having such a measure could have tremendous impact on how software is developed in the future.*

An identification of such a better model, might be able to help us assess not just the quality of overall design, which is of static nature, but also understand the dynamics of the development process, and maybe even better understand the quality of the manner in which software developers maintain code and further evolve it.

Another orthogonal, and less ambitious direction for further research, is application of evolution monitoring to phenomena drawn from domains other than software.
Part IV

appendices
Chapter 14

Method of Interpolation and Removal of Outliers

This appendix explains the interpolation method we used and may be skipped in first read of this manuscript.

Baxter et. al. [3] suggested the following procedure for dealing with these difficulties. Let \( f(b, \alpha, \beta, c) = \beta(b + c)^{-\alpha} \) be the skewed power-law probability density function of \( b \) parametrized by \( \alpha \), \( \beta \) and \( c \). Also, let \( Q \) be the weighted sum of squares of differences between function \( f \) and the data points, i.e.,

\[
Q = \sum_{b} \frac{1}{u_b} (u_b - f(b, \alpha, \beta, c))^2
\] (14.1)

where the summation is only taken over the \( k \) values of \( b \) for which \( u_b \) is not zero. The values of \( \alpha \), \( \beta \) and \( c \) which minimize \( Q \) can be estimated numerically by employing a generalized weighted least squares algorithm, e.g., the Levenberg-Marquardt’s algorithm.

According to Baxter et. al., the quality of the fit can be estimated by assuming that for a fixed values of the parameters \( Q \) follows a normal distribution. As the authors show, the mean of \( Q \) is \( k - 1 \) while its variance is \( k - 2 \). If \( Q \) is substantially larger than the mean, then the skewed power-law null hypothesis is rejected. Conversely, for a given confidence level (say 95\%), the mean, the variance and the normality assumption can be used to compute a confidence interval around the mean. If \( Q \) falls within this confidence interval then this is used as a supporting evidence that the data is skewed-power-law distributed. To account for the cautions mentioned above of the small and large values of \( b \), Baxter et. al. recommend
repeating the procedure while trimming a fraction $\rho$ of the data points from both ends, for $\rho = 5\%, 10\%, 20\%$.

We propose to further improve Baxter et. al.’s using the observation that the quality of the fit is better approximated with Pearson’s $\chi^2$ test. As Baxter et. al., we assume that the expected value of $b$ is indeed $f(b, \alpha, \beta, c)$, and further that $b_i$ is binomially distributed around this expectation.

Approximating the binomial distribution by the normal distribution, we obtain that $z_b = u_b - f(b, \alpha, \beta, c)/\sqrt{f(b, \alpha, \beta, c)}$ follows a standardized normal distribution (with expectation 0 and standard deviation 1). Now, the distribution of the random variable $Q'$ defined by

$$ Q' = \sum_b z_b^2 = \sum_b \left( \frac{u_b - f(b, \alpha, \beta, c)}{f(b, \alpha, \beta, c)} \right)^2 $$

is close to that of the $k^{th}$ $\chi^2$ distribution. We further observe that the weighted sum of squares of differences

$$ \sum_b w_b (u_b - f(b, \alpha, \beta, c))^2 $$

as used in the generalized weighted least squares algorithm, is precisely $Q'$ if the weights are selected to be the reciprocals of the interpolating curve, $w_b = 1/f(b, \alpha, \beta, c)$.

Since the values of the weights are not known prior to the parameter fitting, we recommend applying the fitting process twice: first to determine the approximate value of the weights, and then to fit against these approximations. (In our experience executing this process more than once does not reduce $Q'$ any further.)

As Baxter et al., we recommend removing outliers, but we do so in a slightly different manner: we apply the $\chi^2$ interpolation method iteratively, where each iteration identifies the worst fitting data point, that is the particular $b$ whose contribution to the $\chi^2$ summation, i.e., $(u_b - f(b, \alpha, \beta, c))^2/f(b, \alpha, \beta, c)$, was the largest in the preceding iteration, marks it as an outlier, removes it from the data set, and then proceeds to interpolating the remaining data points. Iteration stopped as soon as the confidence level reaches 95%.
For making a judgment on whether \( x(b) \) increases or decreases with \( b \), we use \textit{Kendall-tau rank correlation coefficient} \cite{14}, which is similar to Pearson correlation, except that it is non-parametric, i.e., it does not rely on assumptions about a specific data distribution and it depends only on the relative order of the values, rather than their actual magnitude.

A quick reminder of the definition of the coefficient follows. Two value pairs \( \langle b_1, x_{b_1} \rangle \) and \( \langle b_2, x_{b_2} \rangle \) are \textit{concordant} if both \( b_1 > b_2 \) and \( x_{b_1} > x_{b_2} \) or if both \( b_1 < b_2 \) and \( x_{b_1} < x_{b_2} \). Otherwise, the two pairs are \textit{discordant}. Given a set of \( m \) pairs of values, consider all \( m(m - 1)/2 \) possible selections of two elements from this set, and let \( m_c \) be the number of concordant selections, and \( m_d \) be the number of discordant pairs. Then, the coefficient, which ranges between \(-1\) and \(1\) is defined by

\[
\tau = \frac{m_c - m_d}{m(m - 1)/2} \tag{15.1}
\]

The theory also defines a value \( z \),

\[
z = \tau \frac{3\sqrt{m(m - 1)}}{\sqrt{4m + 10}} \tag{15.2}
\]

whose distribution is very close to that of a standardized normally distributed random variable when \( m > 10 \). Examining the value of \( z \) makes it possible to estimate the statistical significance of \( \tau \)'s actual value. (For \( m \leq 10 \), one must consult statistical tables to determine the statistical significance of the coefficient.)
We shall compute the value of \( \tau \) for the set of \( \ell + 1 \) data points, i.e., the pairs \((b, x(b))\). If the function is \( x(b) \) is monotonically increasing then \( \tau = 1 \); if it is monotonically decreasing then \( \tau = -1 \); and, if \( x(b) \) does not show an increasing or decreasing properties, then \( \tau = 0 \). If \( x(b) \) are random values selected independent of \( b \), then \( \tau \) would be close to zero.

More generally, greater positive values of \( \tau \) indicate a greater positive “correlation” between \( x(b) \) and \( b \), while smaller negative values indicate greater negative “correlation” between \( b \) and \( x(b) \).
Bibliography


In the case of partial analysis, the world is a collection of open bugs and changes, and the development team chooses which bugs to fix. The influence of the properties of the bugs is issued in the following choice:

- Preference of the bug type mentioned in the report.
- The number of files that were damaged in the first attempt to fix the bug.
- The number of files that were added in the first attempt to fix the bug.
- The number of comparisons that were made in the first attempt to fix the bug.
- The number of attempts that were made in the first attempt to fix the bug.
- The number of fixes that were made in the first attempt to fix the bug.
- The number of files that were added in the first attempt to fix the bug.
- The number of attempts that were made in the first attempt to fix the bug.

In the second part, we introduce an additional implementation to the method, and by means of it, we check in a more realistic way, the decrease of the network Preferential Attachment empiric trend of the phenomenon, regarding common metrics of data: number of children (NOC: Number of children).

The phenomenon is a well-known phenomenon in many scientific fields, and it is common to see that phenomena, whose values are higher, grow faster when compared to phenomena with lower values. This phenomenon is one of the explanations for the distribution in various fields. Recently, research in the field of software has proposed this phenomenon as an explanation for the distribution of several metrics of the data, and between them: the number of children in a module. That is, the hypothesis is that when a module is added to an existing module, it is added to the module that already has the most children.

We found evidence of the phenomenon, although the confidence was not high. In our empirical studies, we noted that the properties of the changes that occurred in the module were also good at predicting which modules were chosen.
Committing new features and fixing bugs is a continuous process. For every project, we use version control: a bug database and a code repository. With the help of commit expressions, we can track new features that the developer incorporates with every operation. This way, we are aware of bug IDs for the code repository, and we can analyze features, such as bug counts, which indicate the number of bugs fixed.

Bug tracking is a critical tool to analyze the behavior of users. Developers can use this tool to identify bugs that are not fixed before new features are added. This information is crucial for developing new features.

For example, developers may notice that a certain feature is not working as expected. They can use the bug tracking system to identify the bug and fix it before adding new features. This way, developers can ensure that the new features are working correctly.

In conclusion, bug tracking is an essential tool for developers to ensure that their code is functioning correctly.

תקציר

באגים (וקיוי התוכנה) הם תקלות בחלקי השרתים של תוכנה במachineryי היום יום של פרויקטים ותוכנות גדולים, מ💚طائرת נייק וגיוניקו.تحرן בالجزי במדבב פסק על תקלה של תוכנה בפרוייקט והם בהיותו גדולות, מסחריים או קהליים. הזומצם של תקלה של פסק ורשות רוסו artificialית של באגי בגרסאות חדשות בין תוכנה جديدة וחדשים.

יין, בoultry מתכון חתך העקרותית לתחדשות ותחדשות מזרחי חתיכה.

(45% מחקרים קודמים של המחקר מתאימים ליצירת באגים) ב AVG מחקרים מדענים עדכ.

מצריכים מספר תיקונים. זה ממעלה גם בזורה, אך פורח את הביצה. המטרה מככ, התכונות נצלות בשני בגרירות באזרחי ומענה על לולא. זה בלני, וכל מבית החיתות והקוד של התיקון

ותרו וה частור קואליציה בכיר באנטוך ב티וקיקה.

(45% מחקרים קודמים של המחקר מתאימים ליצירת באגים) ב AVG מחקרים מדענים עדכ.

템돔, עבור, יהיה בשאלתטים הראמה המתחיתנות גוני ותחדשות של חלקיים בחרים. פיצונים, בנו מחקרים בبرامج ינות בגרירות בחרים, שתחדשות של תקנון של באגים בזכות

תרי, מרוכז פרסי הזדמנויות של פיתוחו כארה ذوญ של חוגים במגישות בחרים. פיתוחים, המאירים בין הביאנקים בכיי תשלומי במסלולים ובאראטים

לאש כל פיתוח שיחה ומיתוסי תקנון של תקנון התוכנה מתו בחרים, תקנון firefighter או תקנון הרוסות. בפגישות הזנויות של תוכנה, רכתי החלל שחרר בחרים, בנו מחקרים בبرامج ינות בגרירות בחרים. פיתוחים, המאירים בין הביאנקים בכיי תשלומי במסלולים ובאראטים

(source עלמנת להשיב על שאלות המחקר, ישמע, את השיעור של עות Ecology תחת kost, ומ.raise) תוצאות החיתות kost kod, מאמיצים להבגיא של בכריית kod התחזוקה code repository) - אכיפפים

בנוסף, על לשון, ישמע את איזורית הכה kod, ומ_raise ההתרות kod code repository) - אכיפאים

Webkit ר Mozilla Firefox

Technion - Computer Science Department - M.Sc. Thesis MSC-2013-20 - 2013
המוץ נועשו בהנחיית פרופ’ יוסי גל בפקולטה למדעי המחשב

אני מודה לливสวยי על התמימנותו鳄 Fighters בriting הממחשי

Technion - Computer Science Department - M.Sc. Thesis MSC-2013-20 - 2013
על תיקונים חלקיים של באניס
ורידוטואציה של מפתחים
לגביהם

חרבר על מחקר

לשם مليוח חלקי של הדרישות לכבלי המתור
מגייסטר לאдесяים במדעי המחשב

סביה אגבאריה

מונח לסטט הטכניון - מכון טכנולוגי לישראל

8 באוגוסט 2013

חרפת Av 5773
על תיקונים חלקיים של באגים והאינטואיציה של מפתחים
לגביהם

סבייה אגבאריה