Copyright © [2008] IEEE.
Reprinted from Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques, (TAIC PART 2008), pp. 13-22

This material is posted here with permission of the IEEE. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org
By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

Exploring the relationship of a file's history and its fault-proneness: An empirical study

Timea Illes-Seifert, Barbara Paech
Institute for Computer Science, University of Heidelberg
Im Neuenheimer Feld 326, D-69120 Heidelberg
{illes, peach}@informatik.uni-heidelberg.de

Abstract

Knowing which particular characteristics of software are indicators for defects is very valuable for testers in order to allocate testing resources appropriately. In this paper, we present the results of an empirical study exploring the relationship between history characteristics of files and their defect count. We analyzed nine open source java projects across different versions in order to answer the following questions: 1) Do past defects correlate with a file's current defect count? 2) Do late changes correlate with a file's defect count? 3) Is the file's age a good indicator for its defect count? The results are partly surprising. Only 4 of 9 programs show moderate correlation between a file's defects in previous and in current releases in more than the half of analysed releases. In contrast to our expectations, the oldest files represent the most fault-prone files. Additionally, late changes influence file's defect count only partly.

1. Introduction

The knowledge about particular characteristics of software that are indicators for defects is very valuable for testers because it helps them to focus the testing effort and to allocate their limited resources appropriately. Information about the software project can be collected from versioning control and bug tracking systems. These systems contain a large amount of information documenting the evolution of a software project.

In practice, this information is often not deeply analysed in order to gain information which facilitates decisions in the present and permits reliable predictions for the future. Based on history characteristics extracted from versioning control systems, e.g. the number of defects in previous versions of a file, estimates for the future evolution can be made. Thus, for example, the expected defects can be predicted which allows to perform accurate testing effort estimates. Similarly, knowing defect detection rates over time of former releases, one can make predictions on remaining defects at the current point of time. This facilitates the decision whether the software can be released or not. Information contained in versioning control and defect tracking systems can also be *combined*. For example, the relationship between history characteristics (e.g. a file's age) and software quality (e.g. measured by the defect count) can be explored. It is very valuable to know particular history characteristics of a file indicating its fault proneness because it helps testers to focus their testing effort on these specific files [5], [6], [7], [8], [9], [10], [11].

In this paper, we present the results of an empirical study exploring the relationship between history characteristics and quality in open source programs. For this purpose, we analysed 9 open source java products during their whole lifetime. We use the defect count of a file as an indicator for its software quality and relate this measure to history characteristics of that file. Particularly, we analyse the following questions: (1) Do past defects influence a file's current defect count? (2) Do late changes influence a file's defect count? (3) Does the file's age influence its defect count?

The remainder of this paper is organized as follows. Section 2 introduces basic definitions and concepts. Section 3 presents history characteristics analysed in this study. The design of our study is described in Section 4. In Section 5, the data collection and analysis procedures are reported, whereas the Sections 6 to 8 contain the results of our empirical study. In Section 9, we discuss the threats to validity and in Section 10 an overview of related work is given. Finally, Section 11 concludes the paper and describes our future work.

978-0-7695-3383-4/08 $25.00 © 2008 IEEE
DOI 10.1109/TAIC-PART.2008.16

IEEE computer society

2. Basic terms and definitions

In this section, basic concepts and terms used in this paper are introduced.

Versioning Control Systems (VCS) are useful for recording the history of documents edited by several developers. In order to edit a file, a developer has to checkout this file, edit it and commit this file back into the repository. Each time a developer commits a file, a message, describing what has been changed, can be optionally added. CVS[1], ClearCase[2], SourceSafe[3] and SVN[4] are examples for such systems.

History Touch (HT). We define a history touch (HT) to be one of the commit actions where changes made by developers are submitted and include modifying, adding or removing files.

Birth of a file denotes the point of time of its first occurrence in the VCS, i.e. the date, the file has been added to the VCS.

Death of a file denotes the point of time of its removal from the VCS.

Present denotes the point in time where our empirical study started.

The **system age** is computed as `Present – Birth` of the "oldest" file.

History. The history of a file subsumes all HTs that occurred to that file from its birth until present or until its death.

Release denotes a point in time in the history of a project which denotes that a new or upgraded version is available. In this study, we considered only final releases of the open source projects.

In this paper, we use the definition of defects and failures provided in [1]: A defect is "a flaw in a component or system that can cause the component or system to fail to perform its required function. A defect, if encountered during execution, may cause a failure of the component or system". Thus, a failure is the observable "deviation of the component or system from its expected delivery, service or result".

Defect count is the number of defects identified in a software entity. In this paper, we count the number of defects of a file. The file *a* is more **fault-prone** than the file *b* if the defect count of the file *a* is higher than the defect count of the file *b*.

3. History Characteristics

In this paper, we distinguish three categories of history characteristics: defect history characteristics, release history characteristics as well as file age characteristics and analyse to what extent these characteristics influence the defect count of a file.

Defect history characteristics subsume all characteristics of a file concerning previously found defects.

Release history characteristics subsume all characteristics of a file concerning the point of time between two releases when a HT occurs. For a detailed analysis, we divide the period between two releases in 5 phases.

`hotFix`: denotes the first 5% of time of the total period between two releases.

`postRelease`: this phase follows the **`hotFix`** phase and denotes the next 10% of the total period between two releases.

`preRelease`: this phase is followed by the **`lastMinuteFix`** phase and denotes 10% of the total period before the **`lastMinuteFix`** phase.

`lastMinuteFix`: this phase denotes the last 5% of time before release.

`moderation`: this phase denotes the period between the **`postRelease`** and **`preRelease`** phase and makes up 70% of the total period between two releases.

Figure 1 illustrates the release history characteristics.

Figure 1. Release history characteristics

File age characteristics subsume all file characteristics related to its age. According to its age, we classify files in one of the following categories[5]:

`Newborn:` A file is newborn at its birthday.

`Young:` `< 0.5 * SystemAge AND not Newborn` (all files that are not older than the half of a system's age and that are not classified as **`Newborn`**)

---

[1] http://www.nongnu.org/cvs/

[2] http://www-306.ibm.com/software/awdtools/clearcase/

[3] http://www.microsoft.com/ssafe/

[4] http://subversion.tigris.org/

[5] We adopted the classification of class hierarchy histories presented in [17]

`Old: >= 0.5 * SystemAge` (all files that are older than or equal to the half of a system's age).

4. Study design

In this Section details on the experiment are described.

4.1. Goal and research questions

The main goal of this empirical study is to analyse the influence of a file's history on its defect count. These are our research hypotheses and their rationale:

H1: The number of defects found in the previous release of a file correlates with its current defect count. The rationale behind this hypothesis is that files that tend to be complex, not well understood and fault-prone remain not well understood and fault-prone.

H2: Release characteristics of a file correlate with its defect count. Particularly, the following sub-hypotheses can be formulated:

H2.1: The defect count of a file increases with the number of HTs in the **`hotFix`** and in the **`postRelease`** phase. The rationale behind this hypothesis is that changes that occur shortly after a software release are quickly implemented and represent not well tested patches which lead to further defects in the corresponding file.

H2.2: The defect count of a file increases with the number of HTs in the **`preRelease`** and in the **`lastMinuteFix`** phase. The rationale behind this hypothesis is that last minute changes and features are not well tested and also increase a file's defect count.

H3: A file's age is an indicator for its defect count. Particularly, the following sub-hypotheses can be formulated:

H3.1: **`Newborn`** and **`young`** files are the most fault-prone files. The rationale behind this hypothesis is that **`Newborn`** and **`Young`** files represent new features that might be not well understood and consequently more fault-prone than old files.

H3.2: **`Old`** files have the lowest defect count. The rationale behind this hypothesis is that old files represent stable functionality which matured over years so that most of the defects have already been removed.

4.2. Independent Variables

The independent variables' definitions are based on the history characteristics described in Section 3 and are summarized in Table 1.

Table 1. Independent variables

ID	Description
D_{PREi}	Number of defects reported for a file between release i-1 and release i.
`HF`	Number of HTs performed on a file in the phase **`hotFix.`**
`PreR`	Number of HTs performed on a file in the phase **`preRelease.`**
`PostR`	Number of HTs performed on a file in the phase **`postRelease.`**
`LM`	Number of HTs performed on a file in the phase **`lastMinuteFix.`**
`Mod`	Number of HTs performed on a file in the phase **`moderation.`**
F-N	**`NewBorn`** file
F-Y	**`Young`** file
F-O	**`Old`** file

4.3. Dependent Variable

The dependent variable of our study is the defect count of a file that occurred between two consecutive releases during its history. Thus, D_{CURRi} denotes the number of defects reported for a file *after* release i and *before* release i+1.

We relate a characteristic j in release n of a file to the defect count reported to that file between release n and release n+1. Figure 3 illustrates how file characteristics are related to corresponding defect densities for particular releases.

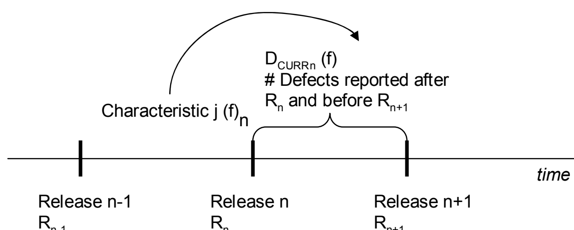


Figure 2. Defect count and characteristics of a file

4.4. Subject projects

In this study, we analysed 9 open source projects. We applied the following criteria when selecting the projects: (1) A bug tracking system is available. (2) Number of HTs > 50.000. (3) The project is written in Java. We included OSCache, a project that does not fulfil the criteria defined above, in order to compare the results obtained for all other projects with a smaller but mature project.

Apache Ant (Ant)[6] is a Java application for automating the build process. **Apache Formatting Objects Processor** (Apache FOP)[7] reads a formatting object (FO) tree and renders the resulting pages to a specified output, e.g. PDF. **Chemistry Development Kit** (CDK)[8] is a Java library for bio- and chemo-informatics and computational chemistry. **Freenet**[9] is a distributed anonymous information storage and retrieval system. **Jetspeed2**[10] is an open portal platform and enterprise information portal. **Jmol**[11] is a „Java molecular viewer for three-dimensional chemical structures. **OSCache**[12] is a Java application which allows performing fine grained dynamic caching of JSP content, servlet responses or arbitrary objects. **Pentaho**[13] is a Java based business intelligence platform. **TV-Browser** [14] is a Java based TV guide.

Table 2 summarizes the attributes of the analyzed projects. A * behind the data in the column "Project since" denotes the date of the registration of the project in SourceForge15. For the rest, the year of the first commit in the versioning system is indicated. The column "OS-Project" contains the name of the project followed by the project's latest version for which the metrics "LOC" (Lines of Code) and the number of files have been computed. The 3rd and the 4th columns contain the number of defects registered in the defect database and the number of HTs extracted from the VCS.

Table 2. Subject Programs

OS-Project	Project since	# Defects	# HTs	LOC	# Files
1. Ant (*1.7.0*)	2000	4804	62763	234253	1550
2. FOP (*0.94*)	2002*	1478	30772	192792	1020
3. CDK (*1.0.1*)	2001*	602	55757	227037	1038
4. Freenet (*0.7*)	1999*	1598	53887	68238	464
5. Jetspeed2 (*2.1.2*)	2005	630	36235	236254	1410
6. Jmol (*11.2*)	2001*	421	39981	117732	332
7. Oscache (*2.4.1*)	2000	2365	1433	19702	113
8. Pentaho (*1.6.0*)	2005*	856	58673	209540	570
9. TV-Browser (*2.6*)	2003	190	38431	170981	1868

5. Data collection and analysis

In order to analyse the relationship between defect count and history characteristics of files, the defect count per file has to be computed. Defect tracking systems contain information on the defects recorded during the lifetime of a project, amongst others the defect ID and additional, detailed information on the defect. But the defect tracking systems usually do not give any information on which files are affected by the defect. Therefore, information contained in VCS has to be analysed. For this purpose, we extract the information contained in the VCS into a history table in a data base. Additionally, we extract the defects of the corresponding project into a defect table in the same data base. Then, we use a 3-level algorithm to determine the defect count per file.

Direct search: First, we search for messages in the history table containing the defect-IDs contained in the defect table. Messages containing the defect-ID and a text pattern, e.g. "fixed" or "removed", are indicators for defects that have been removed. In this case, the number of defects of the corresponding file has to be increased. **Keyword search:** In the second step, we search for keywords, e.g. "defect fixed", "problem fixed", within the messages which have not been investigated in the step before. We use about 50 keywords. **Multi-defects keyword search:** In the last step, we search for keywords which give some hints that more than one defect has been removed (e.g. „two defects fixed"). In this case, we increase the number of defects accordingly. We used SPSS[16], version 11.5, for all statistical analyses.

6. Do past defects influence a file's current defect count?

In order to analyse H1, we first computed the correlation between the defect count of each two consecutive releases, D_{PREi} and D_{CURRi}. The results are listed in Table 3. For each open source program, we computed the Spearman rank-order correlation coefficient. This coefficient [2] is a measure for the dependency between two variables, in this case the dependency between D_{PREi} and D_{CURRi}. The coefficient can take values between -1 and 1, whereas 0 represents no linear correlation. The first and second columns indicate the releases for which the correlation coefficient between D_{PREi} and D_{CURRi} have been computed. The third column indicates the Spearman rank correlation coefficient. For instance, in the open source project Ant a moderate correlation (0.353, 0.338 respectively 0.334) between D_{PREi} and D_{CURRi} can be determined for all analysed releases. These correlations are significant at 0.01 level (**). For the

[6] http://ant.apache.org/
[7] http://xmlgraphics.apache.org/fop/index.html
[8] http://sourceforge.net/projects/cdk/
[9] http://freenetproject.org/whatis.html
[10] http://portals.apache.org/jetspeed-2/
[11] http://jmol.sourceforge.net/
[12] http://www.opensymphony.com/oscache/
[13] http://sourceforge.net/projects/pentaho/
[14] http://www.tvbrowser.org/
[15] http://sourceforge.net/

[16] SPSS, http://www.spss.com/

sake of completeness, the last column contains the Pearson correlation coefficient. The Pearson correlation coefficient is also a measure of the association between two variables but it is not as robust as the Spearman rank correlation coefficient beacause it assumes a normal distribution and is not robust in case of atypical values (e.g. outliers) [3].

Only for the project Ant, a significant correlation with a Spearman coefficient above 0.3 between D_{PREi} and D_{CURRi} can be determined in *all releases*. In 3 of the projects (CDK, Jmol and OSCache), at least the half of the analysed releases show a significant correlation with a Spearman coefficient above 0.3 between past and current defects in files. 3 of the projects, Freenet, Pentaho and TVBrowser show significant correlations in 25% - 33% of the analysed releases. For two projects (ApacheFOP and Jetspeed2), none of the analysed releases show significant correlations with a Spearman coefficient above 0.3 between D_{PREi} and $D_{CURRi.}$ These results are summarized in Figure 2.

Based on the results of the correlation analysis, our research hypothesis H1 cannot be confirmed. *The number of defects found in the previous release of a file does not influence its current defect count.*

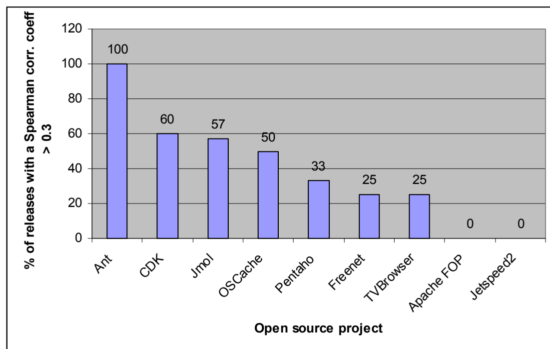


Figure 2. Correlation results for defect characteristics

7. Does the release history of a file influence its defect count?

In order to explore the relationship between the defect count and release history characteristics of a file, the Spearman rank-order correlation coefficient measuring the relationship between the dependent variable (D_{CURRi}) and the independent variables (HF, PreR, PostR, LM) was computed. The Spearman rank-order correlation coefficient measures the extent to which the number of changes performed on a file during a phase (e.g. hotFix) correlates with the later defect count of a file. In this case, it is a measure for the correlation between D_{CURRi} and HF.

Table 3. Correlation analysis for the influence of past defect characteristics on the current defect count. Correlations significant at 0.01 level (**), and at 0.05 level (*)

Ant					
Release i-1	**Release i**	**Spearman**		**Pearson**	
1.5.3.1	*1.6.0*	0.353	**	0.454	**
1.6.0	*1.6.1*	0.338	**	0.461	**
1.6.1	*1.7.0*	0.334	**	0.476	**
Apache FOP					
Release i-1	**Release i**	**Spearman**		**Pearson**	
pre	*0.2*	0.103	**	0.12	**
0.2	*0.93*	0.148	**	0.25	**
0.93	*0.91*	0.111		0.012	
CDK					
Release i-1	**Release i**	**Spearman**		**Pearson**	
CDK-2001	*CDK-2002*	0.473	**	0.429	**
CDK-2002	*CDK-2004*	0.349	**	0.389	**
CDK-2004	*CDK-2005*	0.3	**	0.328	**
CDK-2005	*CDK-2006*	0.063	*	0.216	**
CDK-2006	*1.0*	0.123	**	0.179	**
Freenet					
Release i-1	**Release i**	**Spearman**		**Pearson**	
0.4	*0.5.0*	0.176	**	0.708	**
0.5.0	*0.5.1*	-0.017		0.527	**
0.5.1	*0.5.2*	0.112		0.213	*
0.5.2	*0.7*	0.605	**	0.956	**
Jetspeed2					
Release i-1	**Release i**	**Spearman**		**Pearson**	
pre	*2.0*	0.201	**	0.187	**
2.0	*2.1*	0.1	**	0.115	**
Jmol					
Release i-1	**Release i**	**Spearman**		**Pearson**	
1	*2*	0.42	**	0.69	**
2	*6*	0.178	*	0.068	
6	*9*	0.025		-0.032	
9	*10.0*	0.053		-0.014	
10.0	*10.2*	0.485	**	0.71	**
10.2	*11*	0.481	**	0.837	**
11	*11.2*	0.512	**	0.905	**
OSCache					
Release i-1	**Release i**	**Spearman**		**Pearson**	
pre	*2.1*	0.429	**	0.214	
2.1	*2.4*	0.202		0.326	*
Pentaho					
Release i-1	**Release i**	**Spearman**		**Pearson**	
pre	*1.2.0*	0.068	**	0.203	**
1.2.0	*1.2.1*	0.089		0.092	*
1.2.1	*1.2.6*	0.218	**	0.307	**
TVBrowser					
Release i-1	**Release i**	**Spearman**		**Pearson**	
0.9	*1.0*	0.225	**	0.281	**
1.0	*2.0*	0.184	**	0.091	
2.0	*2.2*	0.265	**	0.217	**
2.2	*2.6*	0.399	**	0.38	**

Table 4 shows the results of the correlation analysis. For each phase, the table shows the ID and name of the analysed program. We computed the Spearman rank correlation coefficient for *each release*

of the analysed programs. In the columns "MAX (Spearman)" and "MIN (Spearman)" the maximum respectively the minimum computed Spearman coefficient is indicated. The next two columns indicate the percentage of releases with a significant correlation coefficient above 0.3 and the percentage of releases with a significant correlation (that can be below 0.3). The last column indicates the percentage of the analysed projects that do not show any significant correlation.

Do hotfixes that occur shortly after a program's release induce more defects? Looking at the correlation coefficients for the phases "**`hotFix`**" and "**`postRelease`**" we can derive the following conclusions:

(1) Most of the projects show high correlation coefficients between the number of changes performed in the **`hotFix`**, respectively in the **`postRelease`** phase and the defect count in at least one release. In the case of the **`hotFix`** phase, 6 of 9 and in the case of the **`postRelease`** phase 8 of 9 programs show a correlation coefficient above 0.3 at least in one of the analysed releases. (2) But there is only one single project that shows a correlations coefficient above 0.3 in all analysed versions (Apache FOP, in the **`hotFix`** phase). 4 of the 9 projects show signigicant correlations in fewer than half of the analysed releases. This is true for both, the **`hotFix`** and the **`postRelease`** phase. Thus, we have to reject H2.1.

The defect count of a file does not increase with the number of HTs performed in the **`hotFix`** *and in the* **`preRelease`** *phase.*

Do late changes that occur shortly before a program's release induce more defects? When we analyse the correlations coefficients for the phases "**`preRelease`**" and "**`lastMinuteFix`**" we can derive the following conclusions:

(3) Most of the projects (8 of 9) show high correlation coefficients between the number of changes performed in the **`preRelease`** phase in at least one release. (4) In the case of the **`lastMinuteFix`** phase, only 5 of 9 projects show high correlation coefficients in at least one release. In case of the **`pre-release`** phase, 2 projects (Freenet and OSCache) show high correlation coefficients in all analysed releases. 7 of the 9 projects show signigicant correlations in at least the half of the analysed releases. (5) In case of the **`lastMinuteFix`** phase, only 2 projects show significant correlations above 0.3 in more than the half of the analysed releases.

Based on the conclusions stated before, we can partly reject the research hypothesis H2.2. *The defect count of a file increases with the number of HTs in the* **`preRelease`** *phase. This is not true for the* **`lastMinuteFix`** *phase.* Finally, we can conclude, that *release history characteristics have only little influence on a file's defect count.*

Table 4. Correlation analysis for release characteristics and defect count

ID	OS-Program	MAX (Spearman)		MIN (Spearman)		% releases with significant corr. Above 0.3	% releases with significant corr.	% releases without significant corr.
		Hotfix						
1	Ant	0.572	*	0.021		25	25	75
2	Apache-FOP	0.458	**	0.335	*	100	100	0
3	CDK	0.284	**	0.098		0	40	60
4	Freenet	0.457	**	0.248		60	60	40
5	Jetspeed2	0.181	**	0.106	**	0	67	33
6	Jmol	0.707	**	0.016		50	50	50
7	Oscache	0.091		0.091		0	0	100
8	Pentaho	0.696	**	0.001		50	50	50
9	TV-Browser	0.584	**	-0.087		80	60	40
		Postrelease						
1	Ant	0.259	*	0.051		0	25	75
2	Apache-FOP	0.501	**	0.045		67	67	33
3	CDK	0.571	**	0.032		20	40	60
4	Freenet	0.588	**	0.396	**	60	60	40
5	Jetspeed2	0.366	**	0.065		33	33	67
6	Jmol	0.646	**	0.093		25	63	38
7	Oscache	0.494	**	0.033		67	33	67
8	Pentaho	0.781	**	0.188	**	50	100	0
9	TV-Browser	0.648	**	-0.026		40	40	60
		Prerelease						
1	Ant	0.455	**	0.277	*	75	100	0
2	Apache-FOP	0.405	**	0.097		67	67	33
3	CDK	0.629	**	0.004		20	40	60
4	Freenet	0.552	**	0.322	**	100	100	0
5	Jetspeed2	0.249	*	0.153	*	0	67	33
6	Jmol	0.659	**	0.087		50	75	25
7	Oscache	0.943		0.378		100	0	100
8	Pentaho	0.531	**	-0.121		50	50	50
9	TV-Browser	0.384	**	0.292	**	80	100	0
		LastMinuteFix						
1	Ant	0.293	**	-0.017		0	50	50
2	Apache-FOP	0.132		0.074		0	0	100
3	CDK	0.425	**	0.069		20	40	60
4	Freenet	0.679	**	-0.003		60	60	40
5	Jetspeed2	0.27	**	0.118	*	0	67	33
6	Jmol	0.596	**	-0.074		25	38	63
7	Oscache	0.559		0.559		33	0	100
8	Pentaho	0.361	**	-0.274	**	50	75	25
9	TV-Browser	0.618	**	-0.026		60	40	60

8. Does the file's age influence its defect count?

In order to analyse the relationship between a file's age and its defect count, we grouped the data into three categories: **`Newborn`**, **`Young`** and **`Old`** files and analysed visually the defect densities in each of these

categories: Have **Newborn** and **Young** files on average a higher defect count than **Old** files? Figure 3 shows for the program "Ant" the mean defect count in each category: **Newborn** (F-N), **Young** (F-Y), **Old** (F-O). The mean defect count is the arithmetic mean, computed as the sum of the defect counts of the files in each group (**Newborn**, **Young** and **Old**) divided by the number of files in each group.

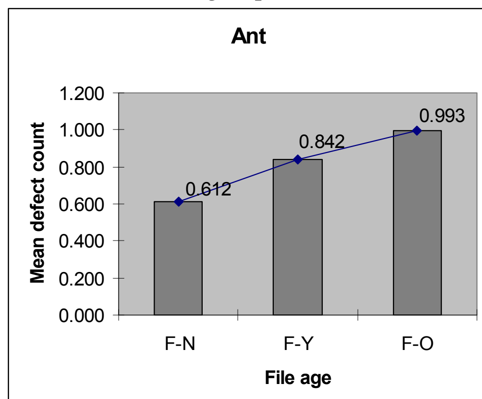


Figure 3. Mean defect count vs. file age for ANT

The mean defect count for **Newborn** files is 0.612, for **Young** files 0.842 and for **Old** files 0.993. The difference of the mean defect count in the categories **Newborn**, **Young** and **Old** is in all analysed programs statistically significant at any chosen significance level.[17] In nearly all projects (7 of 9) the mean defect count for **Old** files is the highest and that for **Newborn** files the lowest. In 2 of 9 projects, the **Young** files have the highes defect count.

Because these results were surprising, we performed a more detailed analysis. For this purpose, we refined our categories and analysed to what extent the defect count of a file depends on its age AND on its stability. Stable files subsume all files that have been changed below average; unstable files have been changed above average. Thus, we analyse, for example, to what extent **Old** files, that have been frequently changed (these are **Old** + unstable files) are more fault prone than **Old** files that have not been frequently changed (**Old** + stable files). The refined categories are the following ones: **N-unst** (all **Newborn** and unstable files), **Y-unst** (all **Young** and unstable files), **O-unst** (all **Old** and unstable files), **N-stab** (all **Newborn** and stable files), Y-stab (all **Young** and stable files), **O-stab** (all **Old** and stable files).

We performed again a visual analysis which related the mean defect count to each of the refined categories. The x-axis contains the refined category: On the y-axis, the mean defect count in each of these categories is indicated. For example, for the project Ant (Figure 4), the mean defect count of **Young** and unstable files (**Y-unst**) is 1.745. The highest defect count have **Newborn** and unstable files (defect count is 1.808). Stable files have on average lower defect counts than unstable files.

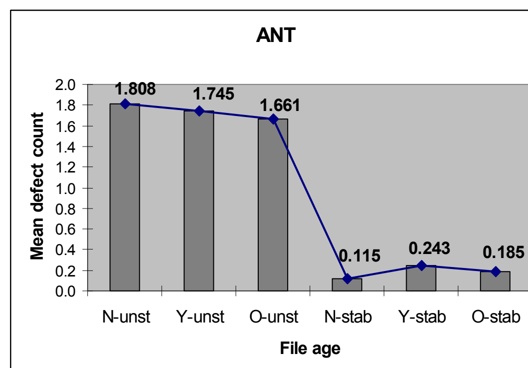
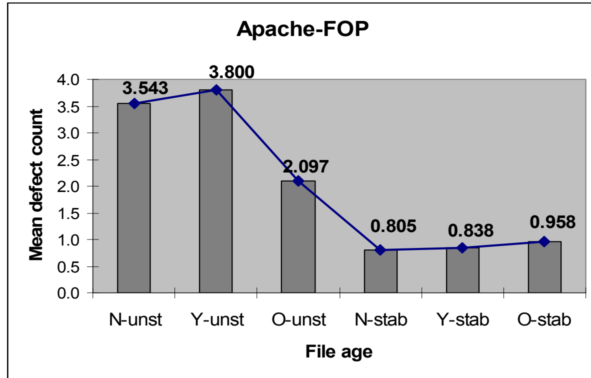


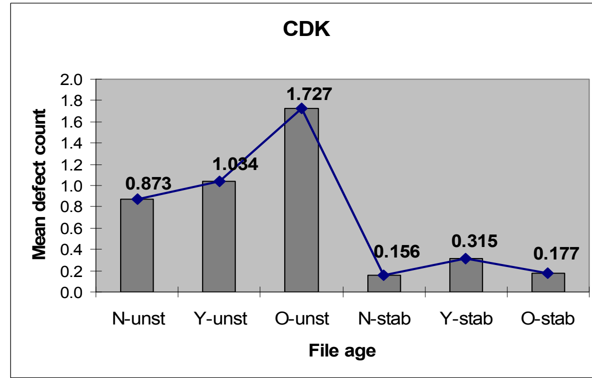
Figure 4. Mean defect count vs. file age and stability for ANT

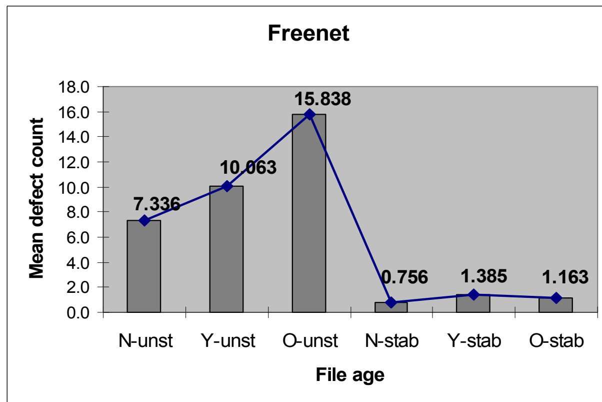
Figure 5 shows the visual analysis for all projects. For 6 of 9 projects (CDK, Freenet, Jmol, Oscache, Pentaho, TVBrowser), the highest defect count is found in files that are **Old** and frequently changed. In three projects (Ant, Jetspeed2 and Pentaho), the defect count for unstable files does not differ very much in any of the **Newborn**, **Young** and **Old** files. In only one single case (Apache-FOP), **Newborn** and **Young** files that are unstable show a significantly higher defect count than **Old** unstable files. In nearly all projects (except Pentaho), **Newborn** stable files have the lowest defect count. In 6 of 9 projects, **Newborn** unstable files are less error-prone than unstable **Young** and **Old** files. Independent of the file age, stable files are less error-prone than unstable files.

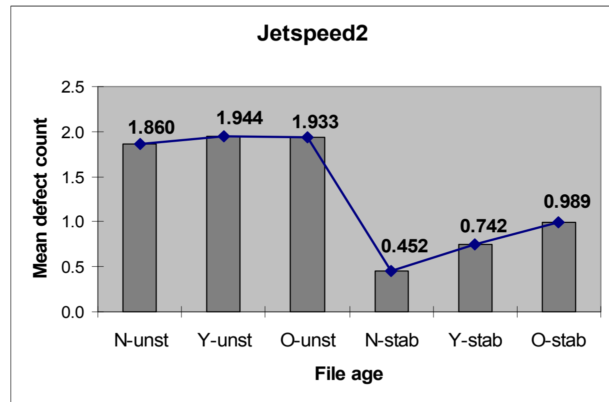
We can conclude that in 6 of the 9 projects the *file's age influences its defect count*. In the other cases, the stability of a file is a better indicator for a file's defect count. In this case the following holds: the more changes have been performed on a file, the higher is its defect count.

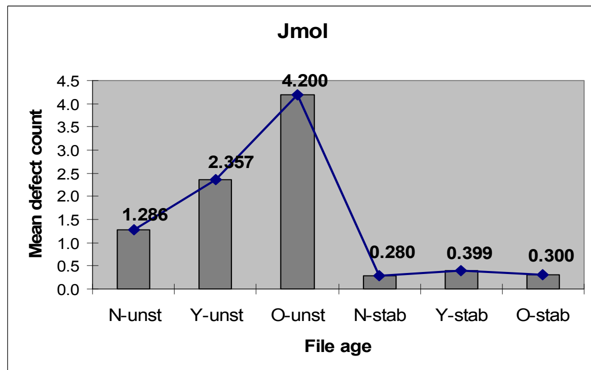
[17] To obtain statistical evidence, we performed the Kruskal-Wallis [2] non-parametric test. A non-parametric test does not make any assumptions concerning the distribution of parameters (in contrast to parametric tests). Differences between several populations can be analyzed with the help of the Kruskal-Wallis test (in our case, differences between Newborn, Young and Old Files). The null hypothesis is that the defect count is the same in both groups; the alternative hypothesis is that it is not. Based on this test, it can be concluded that there is strong evidence from the data that Newborn files have fewer defects than Young files that have fewer defects than Old files.

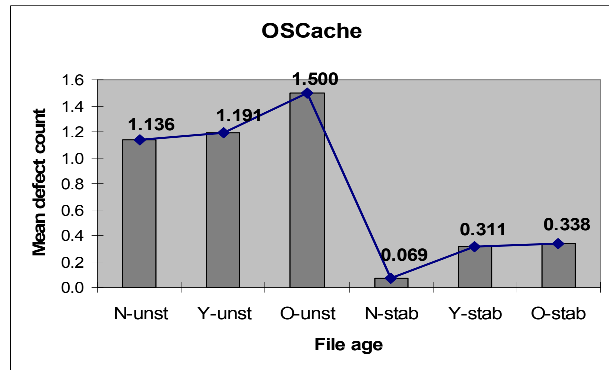


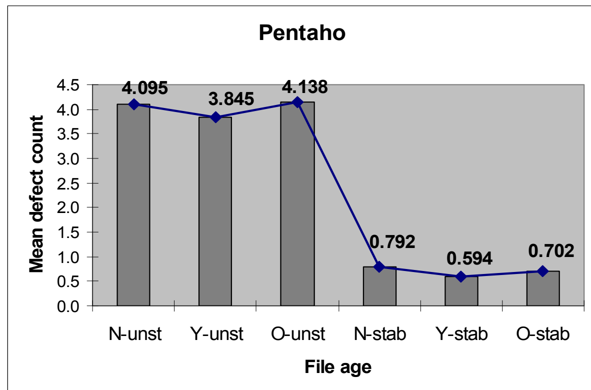












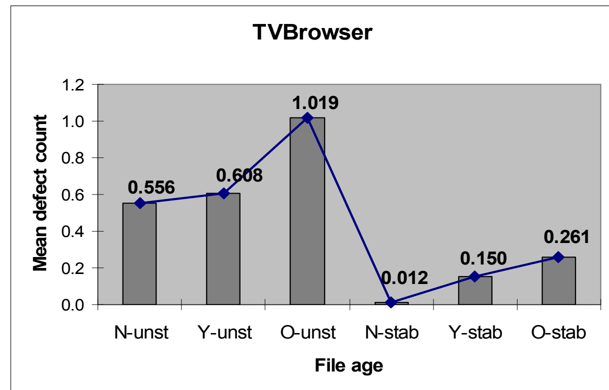


Figure 5. Mean defect count vs. file age and stability

Our research hypothesis H3 can be largely confirmed. In addition, we must reject the research hypotheses H3.1 and H3.2. **Newborn** and **Young** files are not the most fault-prone files. Based on our analyses, **Old** *and unstable files are the most error-prone files.*

9. Threats to validity

Internal validity is concerned with the degree to which conclusions about the causal effect of the independent variables on the dependent variable can be drawn [2]. One threat to validity is that not all developers deliver meaningful messages when they check-in files. Developers, for example, can also check in files without specifying any reason, even though they had corrected a defect. Thus, the defect count of a file can be higher than the defect count computed by our algorithm. This concern is alleviated by the size of the analysed OSPs.

External validity is concerned with the degree to which results can be generalized [2]. This issue is alleviated by the number and diversity of the analysed OSPs. The more OSP programs show the same characteristics, the higher the probability that other OSP programs would also show these characteristics. Additionally, we choose programs from different application domains in order to increase the representativeness of the study results. However, history characteristics of OSP programs and of commercially produced software may differ from each other. Furthermore, analyses of additional programs that are intended in our future work would increase the external validity.

10. Related work

To our knowledge, this is the first study that analyses the influence of a file's history on its defect count deeply.

There are several other studies that focus on predicting the defect count of a software entity by combining product metrics and history metrics ([4], [5], [6], [7], [8], [9], [10], [11]). One of the main features that distinguishes our study from these studies is its magnitude. While most of the studies considered only one program, we have analysed 9 open source projects. Additionally, in contrast to our study, the aim of these studies is defect prediction. Our main goal is to analyse to what extend history characteristics influence software's defect count without selecting the best prediction model. Another difference to these studies, except of the study reported in [9], is that all other studies analyse commercial software.

In [4], [6], [7], [8], [10], and [11] age is used as independent variable but the definitions used in these studies differ from our classification. For example in [10] and [11] only two file categories are defined: "new" and "pre-existing in a previous release". In [7], the age of a file is measured by the number of previous releases in which that file appeared, whereas in [8]the age is measured in months. All these studies confirm our hypothesis that age is an indicator for a file's defect count. But in contrast to our study, they report contrary results. Independent of the measures used for a software entity's age, the studies report that the younger a file the higher its defect count. One cause for such different results can be that that the architecture in open source projects is not as stable as in commercial development. Old files are and must be (as a result of bad design) frequently changed and these changes induce more defects.

Previous defects are considered in the studies [4], [5], [6], [7], [8], [9], [10], and [11]. In [4], [6] and [11] all defects (that occurred in all previous releases) are considered. In [3], [9] and [10], pre-release defects are analysed. In [7] and [8], the number of defects identified in the prior release are considered. The results are contradictory. The results in [4], [8], [9] and [10] confirm our results that previous defects influence the current defect count only partly. The other studies lead to contrary results. We can conclude that the number of past defects may be an indicator for the number of current defects but there are other more reliable indicators.

To our knowledge, the relationship between release history characteristics and defect count has not been analysed empirically yet.

A huge amount of research papers analyse the influence of other metrics of a software entity and its defect count, amongst others in [12], [13], [14], [15] and in [16].

11. Conclusion and future work

In this paper, we investigated the correlation between a file's history and its defect count. Contrary to our expectation, the defect count of a previous release of a file does not influence its current defect count in most of the analysed projects. Additionally, the defect count does not increase with the number of changes (HTs) performed shortly after release. Stronger statistical evidence can be derived for the relationship between the number of changes performed shortly before a file's release and its defect count. The

defect count of a file increases with the number of HTs performed in the period between 85 – 95 % of the time before release. Very late changes (in the last 5% of the time before release) do not correlate with a file's defect count.

A file's age is a good indicator for its defect count. In almost all cases, the mean defect count differs significantly depending on a file's category (newborn, young and old). In other cases, a file's stability is a better indicator for the defect count. The stability of a file classifies a file according to the number of changes (HTs) performed on that file. Files that have been changed below average are less fault-prone than files that have been changed above average.

The most fault-prone files are old files that have been changed above average. One reason is that unstable old files are indicators for bad design. Every time a change occurs, old files are also affected, which causes defects in each release. Additionally, in nearly all projects, the youngest files – the newborn files – have the lowest defect count.

This knowledge is useful for different roles in the development process. Testers can focus their testing activities on particularly fault-prone files, e.g. on old unstable files. Quality engineers can monitor development activities and initiate reviews for often changed old files in order to prevent a high defect count. Additionally, old files changed too often and causing high defect densities can be indicators for bad design. Thus, maintainers can identify candidates for refactorings.

Our future work will focus on analysing other measures for a file's age and its previous defects, as reported in related work, in order to get more precise comparison between our results and the results reported in literature. Additionally, we will focus on analysing to what extent history characteristics combined with code characteristics, e.g. code complexity metrics, can be considered as good indicators for a file's defect count. We expect that history characteristics improve the quality of the indicators that are based on code characteristics only. For example, we expect that old, often changed and complex files are more fault-prone than old and complex files that have not been changed frequently.

12. References

[1] International Software Testing Qualifications Board. *ISTQB Standard Glossary of Terms used in Software Testing* V1.1, 2005.

[2] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: an Introduction*. Kluwer Academic Publishers, 2000.

[3] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*: Brooks/Cole, 1998.

[4] Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. 2000. Predicting fault incidence using software change history. IEEE Transactions on Software Engineering, vol. 26.

[5] Arisholm, E. and Briand, L. C. 2006. Predicting fault-prone components in a java legacy system. In Proceedings of the 2006 ACM/IEEE international Symposium on Empirical Software Engineering (Rio de Janeiro, Brazil, September 21 - 22, 2006). ISESE '06. ACM, New York, NY, 8-17.

[6] Khoshgoftaar, T. M., Allen, E. B., Halstead, R., Trio, G. P., and Flass, R. M. 1998. Using Process History to Predict Software Quality. Computer 31, 4 (Apr. 1998), 66-72.

[7] Ostrand, T. J., Weyuker, E. J., and Bell, R. M. 2005. Predicting the location and number of faults in large software systems. IEEE Trans. Software Eng., vol. 31, pp. 340-355.

[8] Bell, R. M., Ostrand, T. J., and Weyuker, E. J. 2006. Looking for bugs in all the right places. 2006. In Proceedings of the 2006 international Symposium on Software Testing and Analysis (Portland, Maine, USA, July 17 - 20, 2006). ISSTA '06. ACM, New York, NY, 61-72

[9] Schröter, A., Zimmermann,T. Premraj, R., and , R.,Zeller, A. If Your Bug Database Could Talk. Proceedings of the 5th International Symposium on Empirical Software Engineering, Volume II: Short Papers and Posters, pp. 18-20, 2006.

[10] Ostrand, T. J. and Weyuker, E. J. 2002. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT international Symposium on Software Testing and Analysis* (Roma, Italy, July 22 - 24, 2002). ISSTA '02. ACM, New York, NY, 55-64.

[11] M. Pighin, A. Marzona, An Empirical Analysis of Fault Persistence Through Software Releases, International Symposium on Empirical Software Engineering, 2003.

[12] Denaro, G and Pezzè, M. 2002. An empirical evaluation of fault-proneness models. In Proceedings of the International Conference on Software Engineering (ICSE 2002), Orlando, Florida, USA, pp. 241-251.

[13] Basili, V. R., Briand, L. C., and Melo, W. L. 1996. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering vol. 22, pp. 751-761.

[14] Denaro, G., Morasca, S. and Pezzè, M. 2002. Deriving models of software fault-proneness. In Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering Ischia, Italy, pp. 361 - 368.

[15] Nagappan, N., Ball, T., and Zeller, A. 2006. Mining metrics to predict component failures. In Proceedings of the International Conference on Software Engineering (ICSE 2006), Shanghai, China.

[16] Gyimothy, T., Ferenc, R., and Siket, I. 2005. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. IEEE Trans. Softw. Eng. 31, 10 (Oct. 2005), 897-910.

[17] Girba, T. Lanza, M. Ducasse, S., Characterizing the Evolution of Class, Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on Hierarchies, 21-23 March 2005, pp 2- 11.