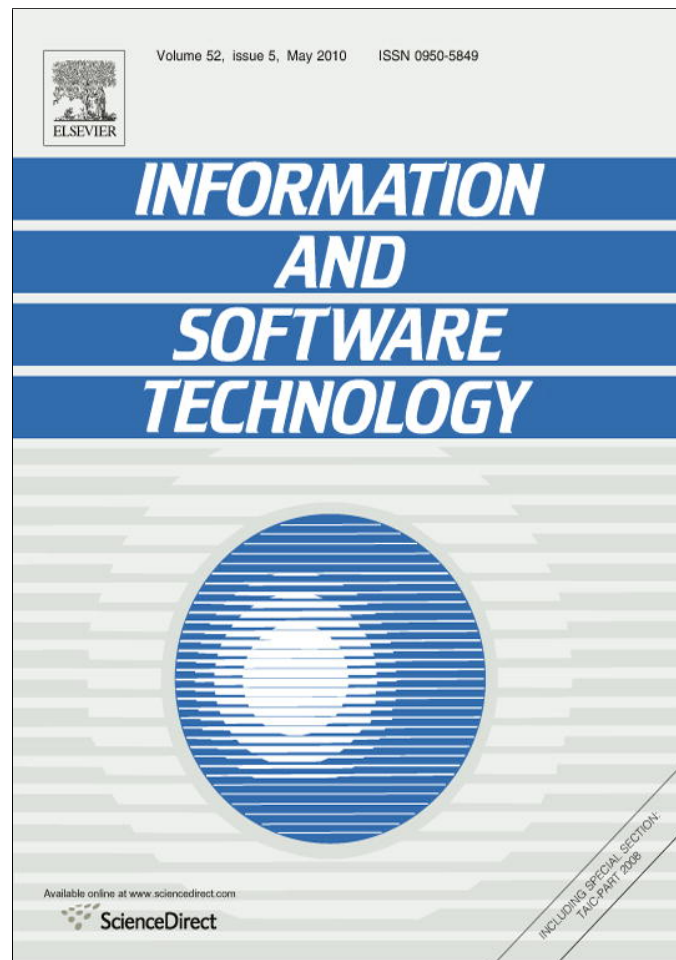


Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

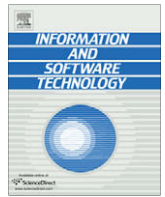
In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

<http://www.elsevier.com/copyright>



Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs

Timea Illes-Seifert *, Barbara Paech

Institute for Computer Science, University of Heidelberg, Im Neuenheimer Feld 326, D-69120 Heidelberg, Germany

ARTICLE INFO

Article history:
Available online 16 December 2009

Keywords:
Empirical study
Software testing
Software history/evolution

ABSTRACT

Context: 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.

Objective: In this paper, we explore the relationship between several historical characteristics of files and their defect count.

Method: For this purpose, we propose an empirical approach that uses statistical procedures and visual representations of the data in order to determine indicators for a file's defect count. We apply this approach to nine open source Java projects across different versions.

Results: Only 4 of 9 programs show moderate correlations between a file's defects in previous and in current releases in more than half of the analysed releases. In contrast to our expectations, the oldest files represent the most fault-prone files. Additionally, late changes correlate with a file's defect count only partly. The number of changes, the number of distinct authors performing changes to a file as well as the file's age are good indicators for a file's defect count in all projects.

Conclusion: Our results show that a software's history is a good indicator for ist quality. We did not find one indicator that persists across all projects in an equal manner. Nevertheless, there are several indicators that show significant strong correlations in nearly all projects: DA (number of distinct authors) and FC (frequency of change). In practice, for each software, statistical analyses have to be performed in order to evaluate the best indicator(s) for a file's defect count.

© 2009 Elsevier B.V. All rights reserved.

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. Based on historical 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. For example, the expected defects can be predicted allowing accurate testing. Similarly, knowing defect detection rates of former releases over time, one can make predictions on remaining defects at the current point of time. This helps decide whether the software should be re-

leased. Information contained in versioning control and defect tracking systems can also be *combined*. For example, the relationship between historical characteristics (e.g. a file's age) and software quality (e.g. measured by the defect count) can be explored. It is very useful to know particular historical characteristics of a file indicating its fault proneness because it helps testers to focus their testing effort on these specific files [19,9,7].

In this paper, we present the results of an empirical study exploring the relationship between historical characteristics and quality in open source programs. For this purpose, we analysed nine 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 historical characteristics of that file. Particularly, we analyse the following questions:

- (1) Do past defects correlate with a file's current defect count?
- (2) Do release characteristics, e.g. late changes or hot fixes, correlate with a file's defect count?
- (3) Is a file's age an indicator for its defect count?
- (4) Is there a history indicator that correlates with a file's defect count in all projects?

* Corresponding author.

E-mail addresses: illes@informatik.uni-heidelberg.de (T. Illes-Seifert), paech@informatik.uni-heidelberg.de (B. Paech).

To answer these questions, we propose an empirical approach that uses several statistical procedures and visual representations of the data in order to determine those files that are responsible for poor quality (expressed by a high defect count). For interval and ratio scaled variables (e.g. the number of past defects), we use correlation analyses to determine the (linear) association between historical characteristics of files and their defect count. For nominal or ordinal scaled variables, we analyse visual representations of the data; then we apply statistical procedures to support statistically the significance of the results obtained by visual analyses. We call this analysis simple analysis of defect variance. In a further step of our approach, detailed analyses are performed in order to get more precise results. By combined analyses of defect variance, the relationship between more independent variables and a file's defect count is analysed. For instance, we analyse the extent to which the defect count of a file depends on its age and on the number of changes performed to this file. Similarly to the simple analyses, combined analyses of defect variance consist of both visual and statistical procedures.

The advantage of this approach is its applicability in practice. Due to the proposed visual representations, an easy interpretation of the data is allowed, thus making this approach an intuitive one. On the other hand, the approach aims at deriving reliable conclusions from data by requiring statistical tests that support the results derived visually.

The remainder of this paper is organized as follows. Section 2 introduces basic definitions and concepts. Section 3 presents the historical characteristics analysed in this study. The design of our study is described in Section 4. In Section 5, data collection and in Section 6 analysis procedures are reported. Section 7 details the empirical approach applied to analyse the relationship of a file's history and its defect count. Sections 8–11 contain the results of our empirical study and Section 12 the discussion of these results. In Section 13, the threats to validity are presented and in Section 14 an overview of related work is given. Finally, Section 15 concludes the paper and describes our future work.

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 SourceSafe³ and SVN⁴ are examples for such systems.

History Touch (HT). We define a history touch to be one of the commit actions where changes made by developers are submitted. Amongst others, each HT has the following attributes: author, affected files(s), date of change, and message.

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 $\text{Present} - \text{Birth}$ of the “oldest” file.



Fig. 1. Release history.

History. The history of a file comprises 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 [14]: 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. Historical characteristics

In this paper, we distinguish four categories of historical characteristic: defect history, release history, change history, as well as file age and analyse their relationship to the defect count of a file.

Defect history of a file concerns previously found defects.

Release history of a file concerns the time between two releases at which a HT occurs. For a detailed analysis, we divide the period between two releases into five phases.

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

post-Release: This phase follows the **hotFix** phase and denotes the next 10% of the total period between two releases.

pre-Release: This phase denotes 10% of the total period immediately before the **lastMinuteFix** phase.

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

moderation: This phase denotes the period between the **post-Release** and **pre-Release** phase and makes up 70% of the total period between two releases.

Fig. 1 illustrates the release history phases.

The **change history** of a file comprises the number, size and author(s) of the HTs performed to that file. As presented in [13], we define the following three change historical characteristics:

- *FC (Frequency of Change):* Number of HTs per file performed between two consecutive releases.
- *DA (Distinct Authors):* Number of distinct authors that performed HTs to a file between two consecutive releases.
- *CF-SUM/AVG (Co-Changed Files):* Total/average number of files that have been conjointly checked in with a file between two consecutive releases.

Stable files are files have been changed (according to their FC metric) less than average; **unstable** files have been changed more than average.

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/>.

Fluctuating files have been changed by an above average number of distinct authors (DA metric) and **non-fluctuating** files have a below average DA metric.

File age; according to age, we classify files in one of the following categories⁵:

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**)

Old: $\geq 0.5 * SystemAge$ (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 hypotheses

The main goal of this empirical study is to analyse the relationship of different historical characteristics of a file and 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 not well understood and fault-prone remain not well understood and fault-prone.

H2: Release characteristics of a file correlate with its defect count. Specifically, 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 **post-Release** 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 **pre-Release** 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.

H4: There is a historical characteristic that is a good indicator for a file's defect count in all projects. The rationale behind this hypothesis is that there are historical characteristics that influence the defect proneness of files independent of an application's domain or other program characteristics.

4.2. Independent variables

The independent variables' definitions are based on the historical 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 pre-Release
PostR	Number of HTs performed on a file in the phase post-Release
LM	Number of HTs performed on a file in the phase lastMinuteFix
Mod	Number of HTs performed on a file in the phase moderation
FC	Frequency of change
DA	Distinct authors
CF	Co-Changed files
F-N	NewBorn file
F-Y	Young file
F-O	Old file

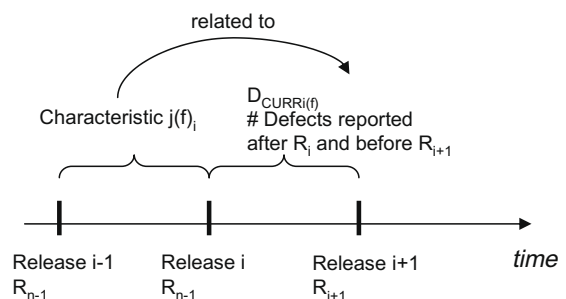


Fig. 2. Defect count and characteristics of a file.

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 change in release i of a file to the defect count reported to that file between release i and release $i + 1$. Fig. 2 illustrates how file characteristics are related to corresponding defect densities for particular releases.

4.4. Subject projects

In this study, we analysed nine open source projects that we searched in large repositories for open source projects, mainly SourceForge⁶ and Java-Source.⁷ As required in [12], the following criteria have been applied when selecting the projects:

- (1) The project is of a large size in order to permit significant results. Thus, the size of the selected projects ranges from about 70,000 LOC to about 240,000 LOC. This criterion guarantees that the empirical results are statistically significant.
- (2) A well documented defect history is available. We searched for projects for which a bug tracking system is available. The availability of a bug tracking is a prerequisite for a project to be considered in the empirical study.
- (3) A well documented history is available. In order to extract historical characteristics automatically, we searched for projects for which a VCS is available. For each HT, at least the following information has to be available: author, date, and message.
- (4) The project is mature so that effects will have appeared if present. According to this criterion, we selected projects with a number of check-ins (we call them history touches – HT) in a versioning control system (VCS) greater than 50,000.

⁵ We adopted the classification of class hierarchy histories presented in [9].

⁶ <http://sourceforge.net/>.

⁷ <http://java-source.net/>.

Table 2
Subject programs.

OS-Project	Project since	# Defects	# HTs	LOC	# Files
1. Ant (1.7.0)	2000	4804	62,763	234,253	1550
2. FOP (0.94)	2002*	1478	30,772	192,792	1020
3. CDK (1.0.1)	2001*	602	55,757	227,037	1038
4. Freenet (0.7)	1999*	1598	53,887	68,238	464
5. Jetspeed2 (2.1.2)	2005	630	36,235	236,254	1410
6. Jmol (11.2)	2001*	421	39,981	117,732	332
7. OSCache (2.4.1)	2000	2365	1433	19,702	113
8. Pentaho (1.6.0)	2005*	856	58,673	209,540	570
9. TV-Browser (2.6)	2003	190	38,431	170,981	1868

For comparability and generalisation of the results a single programming language, Java, has been chosen. We included OSCache, a project that does not fulfil all criteria defined above, in order to compare the results obtained for all other projects with a smaller but mature project (this project exists since 2000). As a result of the search, the following PROJECTS have been identified.

Apache Ant (Apache Ant)⁸ is a Java application for automating the build process using an XML file where the build process as well as its dependencies can be described.

Apache Formatting Objects Processor (Apache FOP)⁹ is a Java application that reads a formatting object (FO) tree and renders the resulting pages to a specified output. Output formats are e.g. PDF, PS, XML or PNG.

Chemistry Development Kit (CDK)¹⁰ is a Java library for bio- and chemo-informatics and computational chemistry.

Freenet¹¹ is a distributed anonymous information storage and retrieval system. Users can use Freenet, e.g. for publishing websites, communicating via message boards or sending emails.

Jetspeed2¹² is an open portal platform and enterprise information portal.

Jmol¹³ is a Java molecular viewer for three-dimensional chemical structures. Features include reading a variety of file types and output from quantum chemistry programs as well as animation of multi-frame files and computed normal modes from quantum programs.

OSCache¹⁴ is a Java application which performs fine grained dynamic caching of JSP content, servlet responses or arbitrary objects.

Pentaho¹⁵ is a Java based business intelligence platform that includes reporting, analysis (OLAP), dashboards, data mining and data integration.

TV-Browser¹⁶ is a Java based TV guide.

Table 2 summarizes the attributes of the analysed projects. A* behind the data in the column "Project since" denotes the date of the registration of the project in SourceForge.¹⁷ 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.

5. Data collection

In this section, data collection procedures used in this study will be described.

5.1. Computing the defect count per file

In order to analyse the relationship between defect count and historical characteristics of files, the defect count per file has to be computed. Usually, 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. However, 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. Similar to the approach used in [8,4,26], we then use a 3-level algorithm to determine the defect count per file. At each level, a particular search strategy is applied.

5.1.1. 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.

5.1.2. 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.

5.1.3. Multi-defects keyword search

In the last step, we search for keywords which give some hints that two or more defects have been removed (e.g. "two defects fixed"). In this case, we increase the number of defects accordingly.

5.2. Keyword definition and validation

The definition and validation of keywords is an iterative process consisting of the following procedures.

5.2.1. Validation of the direct search

In a first step, we analysed whether the HTs found by direct search actually contain an indication that a defect has been corrected. For this purpose, 20% of all HTs found by the first algorithm step have been validated manually. Almost all messages found in this step (99% or more in all projects) have been classified correctly by the algorithm. One reason for this is that the messages are simple, using standard phrases like:

"Bugfix #<BUG-ID>: <What has been done>"
 "Fixed bug related to PR: <Problem Report-ID> submitted by <Submitter>"
 "A fix to ... PR: <Problem Report-ID> submitted by <Submitter>"
 "A bug in ... Bugzilla report <BUGID> submitted by <Submitter>"
 "Correction of <What has been corrected>, #<BUGID> ... "
 "Fix problem ... #<BUGID> submitted by <Submitter>"
 "Fix for #<BUGID>"
 "Fix problem with ... #<BUGID>"

5.2.2. Validation of existing keywords

The main goal of this step is to determine whether the HTs identified by the second and third level of the algorithm actually contain an indication that a defect has been corrected. If this is not the case, the corresponding keyword may be too general, ambiguous or incorrect and must be either refined or removed. A total of 10% of the HTs found by the algorithm have been selected ran-

⁸ <http://ant.apache.org/>.

⁹ <http://xmlgraphics.apache.org/fop/index.html>.

¹⁰ <http://sourceforge.net/projects/cdk/>.

¹¹ <http://freenetproject.org/whatis.html>.

¹² <http://portals.apache.org/jetspeed-2/>.

¹³ <http://jmol.sourceforge.net/>.

¹⁴ <http://www.opensymphony.com/oscache/>.

¹⁵ <http://sourceforge.net/projects/pentaho/>.

¹⁶ <http://www.tvbrowser.org/>.

¹⁷ <http://sourceforge.net/>.

domly and validated in such a way. Thus, incorrect patterns have been removed and ambiguous ones refined.

5.2.3. Searching for missing keyword patterns

The main goal of this step is to identify keyword patterns not included in the search so far. For this purpose, HTs containing weak keywords like “fix” or “problem” have been analysed in order to determine missing complex patterns like “error fixed” or “problem corrected”.

Finally, HTs that have not been selected by any of the levels of the algorithm have been analysed in order to determine if some keywords are missing. For each project 100–200 HTs have been selected randomly and investigated for additional keywords. Only in the case of the OSCache project was one additional keyword found.

5.3. Algorithm performance

Formally, determining whether a HT is defect correcting (dc-HT) or not (ndc-HT) is a classification problem. Accordingly, each HT is mapped to one of the element of the set {positive = (dc-HT), negative = (ndc-HT)}. The algorithm represents a classification model that predicts whether an instance is positive or negative. Given a HT, there are four possibilities:

- **True positive (TP):** This is the case if the HT is positive (= dc-HT) and it is classified as positive by the algorithm.
- **False negative (FN):** The HT is positive but classified as negative (= ndc-HT).

Table 3

Algorithm performance: percentage of correctly classified HTs out of 10% of all HTs found by the algorithm.

Project		% of correctly classified dc-HTs
1	FOP	0.993
2	Ant	0.974
3	CDK	0.987
4	Freenet	0.997
5	Jetspeed2	0.994
6	Jmol	0.998
7	OSCache	0.999
8	Pentaho	0.996
9	TVBrowser	0.995

Table 4

Algorithm performance: anti-pattern analysis results.

Project		% of the number of HTs in the intersection relative to the number of dc-HTs found by the algorithm	Classification accuracy
1	Ant	0.03	0.947
2	FOP	0.06	0.941
3	CDK	0.03	0.947
4	Freenet	0.29	0.994
5	Jetspeed2	0.27	0.990
6	Jmol	0.01	1.000
7	OSCache	0.21	1.000
8	Pentaho	0.04	0.955
9	TVBrowser	0.03	0.900

Table 5

Algorithm performance: overall performance.

	Ant	FOP	CDK	Freenet	Jetspeed2	Jmol	OSCache	Pentaho	TV-Browser	MIN	MAX
Precision	0.945	0.985	0.917	0.977	0.981	0.968	0.964	0.977	0.969	0.985	0.917
Accuracy	0.979	0.983	0.968	0.958	0.989	0.981	0.972	0.975	0.970	0.989	0.958

- **True negative (TN):** The HT is classified as negative and it is actually negative.
- **False positive (FP):** The HT is actually negative but classified as positive.

In order to determine the overall performance of our search, three analyses have been performed: true-positives analysis, anti-pattern analysis, and the overall performance analysis.

For the **true-positives analysis** we randomly selected 10% of all HTs found by the algorithm and analysed whether the HTs have been correctly classified as dc-HTs. Table 3 summarizes the results of our analysis. For each project, the percentage of correctly classified dc-HTs (true positives) is indicated. The results show high classification accuracy with respect to the correctly classified dc-HTs that ranges from 97% to 99%.

For the **anti-pattern analysis**, we defined a set of keyword “anti-patterns” that indicate a non-defect-correcting HT, e.g. “initial revision”, “refactoring” or “removed warnings”. Then we computed the intersection of both: the set of non-defect-correcting HTs and the set of defect-correcting HTs. All HTs that lie in the intersection can be a sign for an erroneous classification. Table 4 shows the results of this analysis. For each project, the percentage of HTs lying in the intersection set is indicated (relative to the total number of dc-HTs identified by the algorithm). The last column indicates the percentage of correctly classified dc-HTs in the intersection set.

This analysis underlines the results obtained by the true-positives analysis. The classification accuracy with respect to the correctly classified dc-HTs in the intersection set ranges from 94% to 100%.

In order to evaluate the **overall performance** of the algorithm we randomly selected 1000 HTs in each project and analysed whether it was a TP, TN, FP, or FN. Then, we compared precision and accuracy of the classification by the algorithm.

The *precision* can be calculated as:

$$\text{precision} = \frac{TP}{TP + FP}$$

The precision indicates the probability that the HT is actually “positive” when the algorithm computes this.

The overall accuracy of the algorithm can be calculated by:

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Table 5 summarizes precision and accuracy for all projects. The precision of the algorithm is high across all projects. It ranges from 0.917 to 0.985. Thus, it is very probable that a HT is actually positive if this is determined by the algorithm.

The overall accuracy is also high and ranges from 0.958 to 0.989. In six cases, the overall accuracy is higher than the precision. In two other cases, both values (precision and accuracy) are quite similar.

5.4. Defect correction density

On average, 14% of all HTs contain messages reporting a defect (defect-correcting message). The maximum is 31.9% in case of Jetspeed2 and the minimum is 2.9% in case of TVBrowser. Fig. 3 illus-

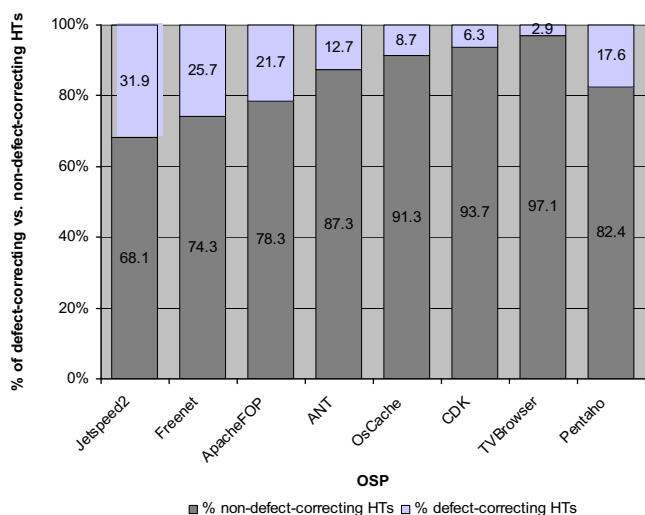


Fig. 3. Defect correction density in HTs.

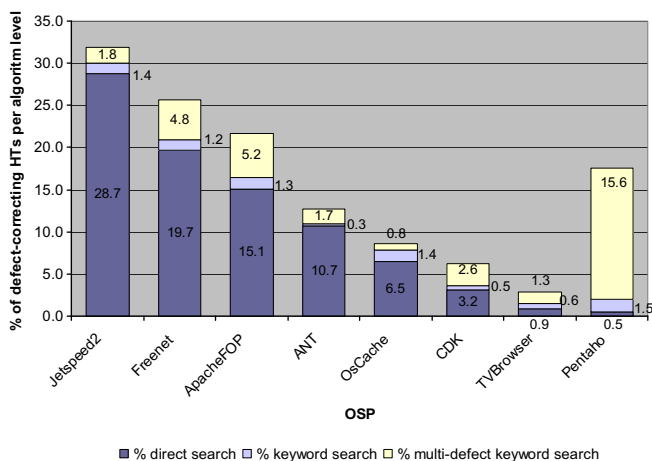


Fig. 4. Defect detection per algorithm level.

trates the percentage of defect-correcting messages (these are messages which have been found in one of the steps of the algorithm presented in Section 5.1) for each project. Consequently, most of the HTs are non-defect correcting (e.g. initial check-in, extension of the functionality, etc.) than defect correction.

In almost all programs, the percentage of HTs found by direct search makes up the biggest part of all HTs found by any level of the algorithm. For example, in the case of Jetspeed2, 28.7% of all HTs (in the history table) contain a reference to a bugID in the defect table, 1.4% of all messages contain one of the keywords found by direct search and 1.8% of all messages contain keywords that indicate that more than one defect has been corrected (found by the multi-defect keyword search). Pentaho is the only program for which the most of the defects have been found by the multi-keyword search. Fig. 4 illustrates the percentage of defect-correcting HTs per each level of the algorithm (direct search, keyword search, multi-defect keyword search) for each project.

6. Data analysis

In order to analyse the relationship between the defect count and historical characteristics of files, we use several analyses depending on the measurement scale of the variables. For ratio and interval scaled variables, we perform correlation analyses.

For nominal and ordinal scaled variables, simple respectively combined analyses of defect variance are performed. For both analyses of defect variance we use visual means and statistical procedures. The statistical procedures serve to statistically support the significance of the results obtained by visual analyses.

We used SPSS,¹⁸ version 11.5, for all statistical analyses.

6.1. Correlation analysis

The dependency between two interval or ratio variables, such as the number of HTs performed to a file (measured by the FC metric) and a file's defect count, can be expressed by the Spearman rank-order correlation coefficient. This coefficient is a measure for the (linear) dependency between two variables [29]. The coefficient can take values between -1 and 1, whereas 0 represents no linear correlation. Another statistical measure of the association between two variables is the Pearson correlation coefficient. This coefficient is not as robust as the Spearman rank correlation coefficient because it assumes a normal distribution and is not robust in case of atypical values (e.g. outliers) [7]. For the sake of completeness, we mostly indicate both coefficients.

6.2. Simple analysis of defect variance

We use this analysis in order to determine the relationship between a nominal or ordinal independent variable and a file's defect count, i.e. we analyse the variance of the defect count in different categories of the independent variable. A nominal or ordinal variable classifies the entities according to an attribute. For example, the age metric of a file classifies the entities with respect to their age into one of the categories: **Newborn**, **Young** or **Old**.

For analysis, we display the data in a diagram that we call *defect variance analysis diagram (DVA)*. This diagram relates the mean defect count to each of the defined categories as follows: the x-axis contains the category. On the y-axis, the mean defect count in each of these categories is indicated. The mean defect count is the arithmetic mean, computed as the sum of the defect counts of the files in each category divided by the number of files in each group. For example, Fig. 5 shows the DVA diagram for the program "Ant" with respect to the independent variable age. The mean defect count for **Newborn** files (F-N) is 0.612, for **Young** (F-Y) files 0.842 and for **Old** files (F-O) 0.993. Based on the DVA, we can see that **old** files are the most fault-prone files.

In order to obtain statistical evidence for the results we derived visually, we perform the Kruskal-Wallis [29] 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 analysed with the help of the Kruskal-Wallis test (in our example, 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.

6.3. Combined analysis of defect variance

In order to analyse the relationship between two categorical independent variables and a file's defect count, we again use the DVA diagram for the visual analysis. We obtain the categories by combining the original ones and perform the analysis as described for the simple categorical analysis. For example, we can analyse to what extent the defect count of a file depends on its age AND on its stability. Thus, we analyse the extent to which **old** files, that have been frequently changed (these are **old** and **unstable** files)

¹⁸ SPSS, <http://www.spss.com/>.

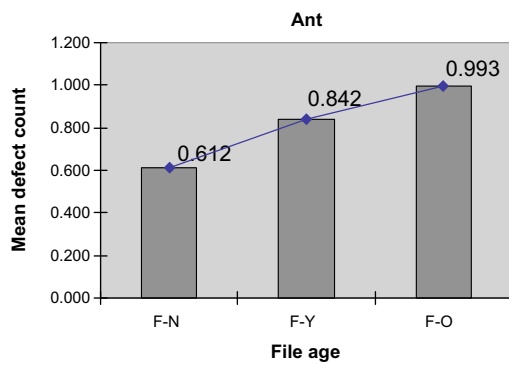


Fig. 5. Simple DVA for Ant: mean defect count vs. file age. According to the Kruskal–Wallis non-parametric test, the results are statistically significant at the 0.01 significance level.

Table 6
Category definition matrix for age × stability.

		Stability	
		Stable	Unstable
Age	Newborn	N-stab	N-unst
	Young	Y-stab	Y-unst
	Old	O-stab	O-unst

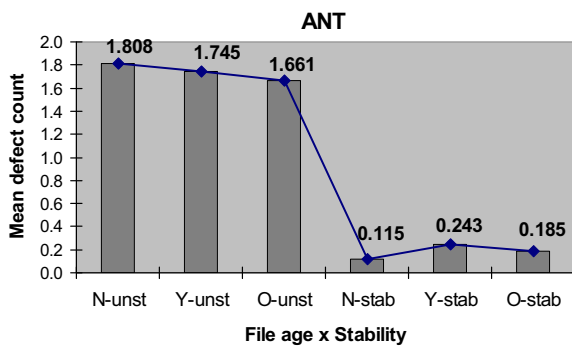


Fig. 6. Combined DVA for Ant: mean defect count vs. file age and stability. According to the Kruskal–Wallis non-parametric test, the results are statistically significant at the 0.01 significance level.

are more fault-prone than **old** files that have not been frequently changed (**old** and **stable** files). In this example, the refined categories can be defined as shown in Table 6.

The DVA relates the mean defect count to each of the refined categories. For instance, for the project Ant (see Fig. 6), 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. To confirm the results obtained by the visual analysis statistically, the Kruskal–Wallis test has to be applied. Similarly to the simple categorical analysis, the null hypothesis is that there are no differences in the mean defect counts among the *refined* categories, the alternative hypothesis is that it is not.

7. Empirical approach

In this section, we present an empirical approach that can be used to determine indicators for quality lacks in software.

7.1. Identification of software and software releases

In the first step, the objects of investigation have to be determined. Thus the software or software components and the corre-

sponding releases are to be analysed. In this study, we identified nine open source projects and for each of them, we determined major releases.

The following criteria increase the success and significance of the analyses.

- (a) *Size*: As described in Section 4.4, the size of the software or of the analysed components guarantees that the results are statistically significant.
- (b) *Maturity*: The maturity of software guarantees that effects will have appeared if present.
- (c) *Version controlled source code*: In order to be able to identify different releases of software the availability of a VCS controlled source code is a prerequisite.
- (d) *Documented history*: The availability of a documented history, mostly in terms of a VCS is a prerequisite for all analyses concerning the relationship between historical characteristics and software quality.
- (e) *Documented defect history*: In the case that the quality of the software is measured in terms of defects, the availability of a documented defect history is also indispensable. Usually, the defect history is documented within a defect tracking system.
- (f) *Source code*: The availability of source code is a prerequisite for all analyses concerning the relationship between code characteristics and software quality. In the case that COTS components for which the source code is not available are part of the software to be analysed, structural analyses are difficult.

In this study, the size (a), the maturity (b), version controlled source code (c) and the availability of a documented history within a VCS (d) were criteria that have been applied when selecting the open source projects. We expressed quality in terms of the number of defects that occurred in a file. Consequently, the availability of a documented defect history (e) in terms of a defect tracking system was also a prerequisite for a project to be included in the study. Since the correlation of structural code characteristics with software quality metrics was not in focus of this study, criterion (f) was not a prerequisite. However, since we plan to analyse the relationship between history and source code characteristics *in combination* in our future work, we searched for open source projects for which the source code is available (criterion f).

7.2. Definition of granularity

The granularity of the analyses to be performed has to be defined. In this study, we performed all analyses at the file level. Nevertheless, more detailed analyses at the class level or more synthesised analyses at the package level are also possible.

7.3. Definition of quality

In this step, the dependent variable has to be defined. How should quality be measured? In this study, we measured quality in terms of the number of defects that occurred in a file. More detailed analyses that differentiates, for example, between pre-release (defects that occurred before release) and post-release (defects that occurred after release) defects as quality measures are also possible.

7.4. Definition of quality indicators

In this step, the dependent variable(s) have to be defined. In the first step, hypotheses on possible quality indicators for a software's quality should be formulated. Based on these hypotheses, the

dependent variables and corresponding metrics can be derived. Usually, a lot of metrics can be calculated automatically. Thus, the selection of the “right” set of variables is not easy. Testers’ experience and results from previous analyses can be used as input to define a manageable set of independent variables. This step also includes the definition of the measurement scale for the variables. For example, we divided the data in three groups according to a file’s age: **Newborn**, **Young** and **Old**. Thus, the variable age is on ordinal scale.

7.5. Measurement

In this step, data defined in steps 3 and 4 for all identified software entities at the level defined in step 2 has to be collected. In addition, collection procedures have to be validated for a randomly selected part of the data. In this study, a file’s defect count has to be determined retrospectively. Since the number of HTs is too high for a manual classification in defect-correcting and non-defect-correcting HTs, we analysed the HTs automatically (The algorithm is presented in Section 5.1).

7.6. Simple analyses

In this step, first analyses that show the first tendency with respect to the hypotheses formulated in step 4 have to be performed. Correlation analyses and simple analyses of variance of the dependent variable can be performed for ratio scaled respectively for nominal or ordinal scaled variables. In this study, we conducted correlation analyses for all ratio scaled variables. For instance, we analysed the correlation between the number of distinct authors who changed a file (metric DA) and a file’s defect count. In addition, we performed simple analyses of defect variance for nominal and ordinal scaled variables. For example, we analysed the variance of the defect count among the categories **Newborn**, **Young** and **Old** for the variable age.

7.7. Detailed analyses

In order to refine the results obtained by simple analyses, the relationship between more than one independent variable and the dependent variable has to be analysed. Detailed analyses can be performed to get more precise results. For example, based on simple correlation analyses, a strong correlation between the defect count and the metrics DA (distinct authors) and FC (frequency of change) could be determined. In the second step, we combined these metrics in order to get more precise results on fault-prone files. Proceeding this way, we found out that **fluctuating unstable** files are the most fault-prone files. For the independent variable “age”, we performed a simple analysis of defect variance in the first step. As the results were surprising – **Old** files proved to be the most fault-prone ones, we performed a detailed analysis and examined the influence of age and stability, as well as age and fluctuation in combination. These detailed analyses revealed a more precise view. Based on the results, it could be concluded that, for instance, **Old unstable** or **Old fluctuating** files are the most fault-prone ones. In addition, **Old non-fluctuating or stable** files are much less fault-prone than **Old fluctuating** or **unstable** files.

7.8. Synthesis of results

In this step, conclusions on the results have to be drawn. Which are good indicators for software quality? Which indicators persist across several releases? Based on the results of the synthesis, it can be decided which measures can be taken to improve the qual-

ity and who (i.e. which roles) should be involved in improving the quality.

8. Do past defects correlate with a file’s current defect count?

In order to analyse H1, we performed a correlation analysis, i.e. we computed the correlation between the defect count of each two consecutive releases, D_{PREi} and D_{CURRI} . The results are listed in Table 7. For each project, we computed the Spearman rank-order correlation coefficient between D_{PREi} and D_{CURRI} . The first and second columns indicate the releases for which the correlation coefficients 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 sake of completeness, the last column contains the Pearson correlation coefficient.

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 three of the projects (CDK, Jmol and

Table 7

Correlation analysis for H1. Correlations significant at 0.01 level (**), and at 0.05 level (*).

Release $i - 1$	Release i	Spearman		Pearson	
Ant					
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					
pre	0.2	0.103	**	0.12	**
0.2	0.93	0.148	**	0.25	**
0.93	0.91	0.111		0.012	
CDK					
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	
Free net					
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					
pre	2.0	0.201	**	0.187	**
2.0	2.1	0.1	**	0.115	**
Jmol					
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					
pre	2.1	0.429	**	0.214	
2.1	2.4	0.202		0.326	*
Pentaho					
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					
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	**

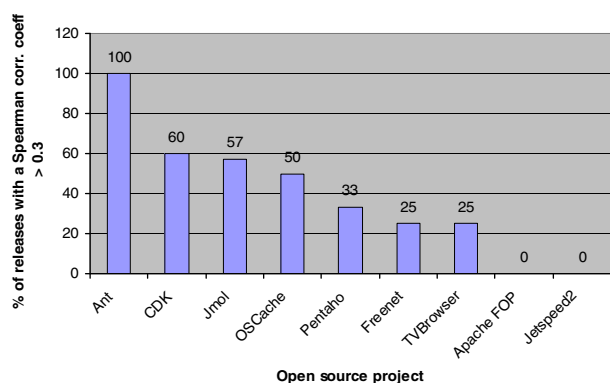


Fig. 7. Correlation results for defect characteristics.

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. Three 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 Fig. 7.

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 correlate with its current defect count.*

9. Does a file's release history correlate with its defect count?

In order to explore the relationship between the defect count and the release history of a file, we performed a correlation

Table 9
Maximum/minimum percentage of files in each of the categories **Newborn**, **Young** and **Old**.

Project	Newborn		Young		Old	
	MAX (%)	Min (%)	MAX (%)	Min (%)	MAX (%)	Min (%)
1 Ant	43	18	74	410	44	8
2 FOP	40	40	60	48	41	24
3 CDK	54	21	70	44	9	4
4 Freenet	51	19	68	29	4	2
5 Jetspeed2	48	13	57	56	30	9
6 Jmol	64	27	63	35	6	2
7 OSCache	39	9	45	8	72	16
8 Pentaho	63	63	71	37	10	0
9 TVBrowser	86	22	59	12	96	24

Table 8
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	*	25	25	75
2	Apache-FOP	0.458	**	100	100	0
3	CDK	0.284	**	0	40	60
4	Freenet	0.457	**	60	60	40
5	Jetspeed2	0.181	**	0	67	33
6	Jmol	0.707	**	50	50	50
7	OSCache	0.091	**	0	0	100
8	Pentaho	0.696	**	50	50	50
9	TV-Browser	0.584	**	80	60	40
Post-release						
1	Ant	0.259	*	0	25	75
2	Apache-FOP	0.501	**	67	67	33
3	CDK	0.571	**	20	40	60
4	Freenet	0.588	**	60	60	40
5	Jetspeed2	0.366	**	33	33	67
6	Jmol	0.646	**	25	63	38
7	OSCache	0.494	**	67	33	67
8	Pentaho	0.781	**	50	100	0
9	TV-Browser	0.648	**	40	40	60
Pre-release						
1	Ant	0.455	*	75	100	0
2	Apache-FOP	0.405	*	67	67	33
3	CDK	0.629	*	20	40	60
4	Freenet	0.552	*	100	100	0
5	Jetspeed2	0.249	*	0	67	33
6	Jmol	0.659	*	50	75	25
7	OSCache	0.943	**	100	0	100
8	Pentaho	0.531	**	50	50	50
9	TV-Browser	0.384	**	80	100	0
LastMinuteFix						
1	Ant	0.293	**	0	50	50
2	Apache-FOP	0.132	**	0	0	100
3	CDK	0.425	**	20	40	60
4	Freenet	0.679	**	60	60	40
5	Jetspeed2	0.27	**	0	67	33
6	Jmol	0.596	**	25	38	63
7	OSCache	0.559	**	33	0	100
8	Pentaho	0.361	**	50	75	25
9	TV-Browser	0.618	**	60	40	60

analysis, i.e. we computed the correlation between the dependent variable (D_{CURRi}) and the independent variables (HF, PreR, PostR, LM). In this case, 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. Thus, the Spearman coefficient it is a measure for the correlation between D_{CURRi} and HF.

Table 8 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.

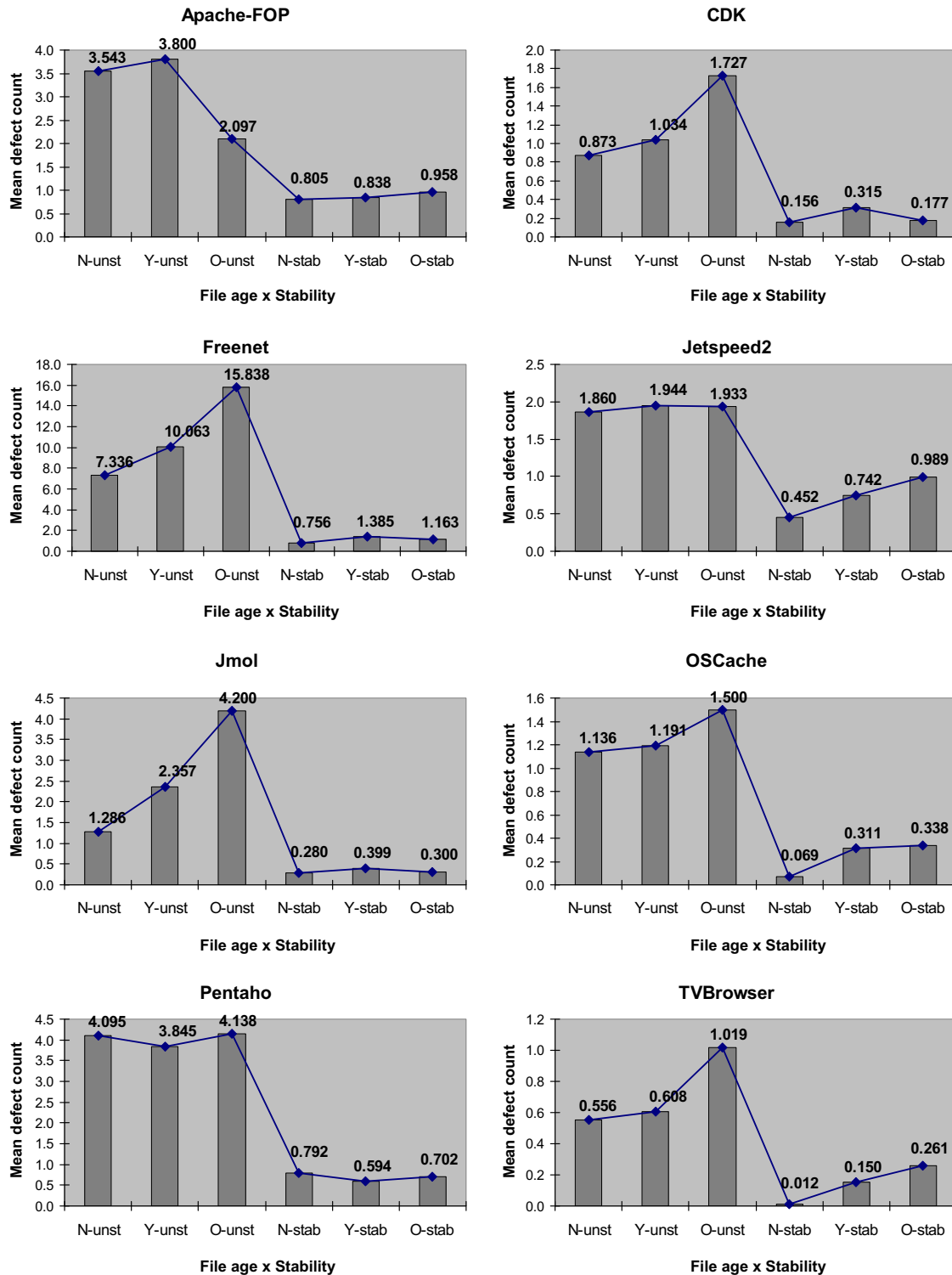


Fig. 8. DVA: mean defect count vs. file age and stability. According to the Kruskal–Wallis non-parametric test, all results are statistically significant at the 0.01 significance level.

Do hotfixes that occur shortly after a program's release induce more defects? Looking at the correlation coefficients for the phases "hotFix" and "post-Release" 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 post-Release 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 post-Release phase 8 of 9 programs show a correlation coefficient above 0.3 at least in one of the analysed releases.
- (2) However, there is only one single project that shows a correlations coefficient above 0.3 in all analysed versions (Apache FOP, in the hotFix phase). Four of the nine projects show significant correlations in fewer than half of the analysed releases. This is true for both hotFix and the post-Release 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 post-Release phase.

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

- (1) Most of the projects (8 of 9) show high correlation coefficients between the number of changes performed in the pre-Release phase in at least one release.
- (2) 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, two projects (Freenet and OSCache) show high correlation coefficients in all analysed releases. Seven of the nine projects show significant correlations in at least the half of the analysed releases.
- (3) In case of the lastMinuteFix phase, only two 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 pre-Release phase. This is not true for the lastMinuteFix phase. Finally, we can conclude, that there is little statistical evidence for the correlation of a file's release history with its defect count.

10. Is a file's age and indicator for 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 performed a simple analysis of defect variance: have **Newborn** and **Young** files on average a higher defect count than **Old** files? Fig. 14 shows the percentage of files in each of the categories **Newborn**, **Young** and **Old** for each release of the Ant program. At the beginning of the project, the **Newborn** files make up the most of all files in the project. Later in the development cycle, the percentage of **Young** and **Old** files increases. Such a diagram is characteristic for most of the projects. Detailed information on the maximum/minimum percentage of files in each of the categories **Newborn**, **Young** and **Old** for each project is indicated in Table 9.

Fig. 5 shows the mean defect count for the program "Ant" in each category: **Newborn** (F-N), **Young** (F-Y), **Old** (F-O).

In nearly all projects (7 of 9) the mean defect count for **Old** files is the highest and that for **Newborn** files is the lowest one. In 2 of 9 projects, the **Young** files have the highest defect count. According to the Kruskal–Wallis non-parametric test, these results are significant at the 0.01 significance level.

Due to the surprising results, we performed two more detailed analyses. For this purpose, we refined our categories and analysed the following questions, applying a combined analysis of defect variance:

1. To what extent does the defect count of a file depend on its age AND on its stability?
2. To what extent does the defect count of a file depend on its age AND on its fluctuation?

In order to answer the first question, we relate the mean defect count to each of refined categories presented in Table 6. Then, we built the DVA diagram. Fig. 8 shows the DVA diagrams 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.

Unstable **old** files are three times (ApacheFOP) to 22 (OSCache) times more fault-prone than **stable** **Newborn** files. In the case of TV-Browser, the factor is as high as 85.

We can conclude that in 6 of 9 projects the file's age is a good indicator for its defect count. In other cases, the stability of a file is a

Table 10
Category definition matrix for age × fluctuation.

		Fluctuation	
		fluctuating	Non-fluctuating
Age	Newborn	N-F	N-nF
	Young	Y-F	Y-nF
	Old	O-F	O-nF

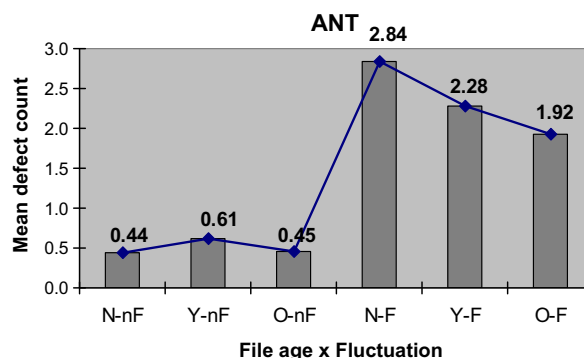


Fig. 9. DVA: mean defect count vs. file age and fluctuation for Ant. According to the Kruskal–Wallis non-parametric test, the results are statistically significant at the 0.01 significance level.

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.

To answer the second question, we relate the mean defect count to each of refined categories, presented in the matrix in Table 10.

Then, we build the DVA diagram. For instance, for the project Ant (see Fig. 9), the mean defect count of **Young** and **fluctuating** files (**Y-F**) is 2.28. The highest defect counts have **Newborn** and

fluctuating files (defect count is 2.84). **Non-fluctuating** files have on average lower defect counts than **fluctuating** files.

Fig. 10 shows the DVA for all projects. For 6 of 9 projects (CDK, Freenet, Jetspeed2, Jmol, OsCache, Pentaho), the highest defect count is found in files that are **Old** and **fluctuating**. In the case of Ant and TVBrowser, **fluctuating** and **Newborn** files are the most fault-prone ones. Only in case of the Apache-FOP project, **non-fluctuating Young** files have the highest defect count. The lowest defect count show **non-fluctuating Newborn** files. This is the case for all projects.

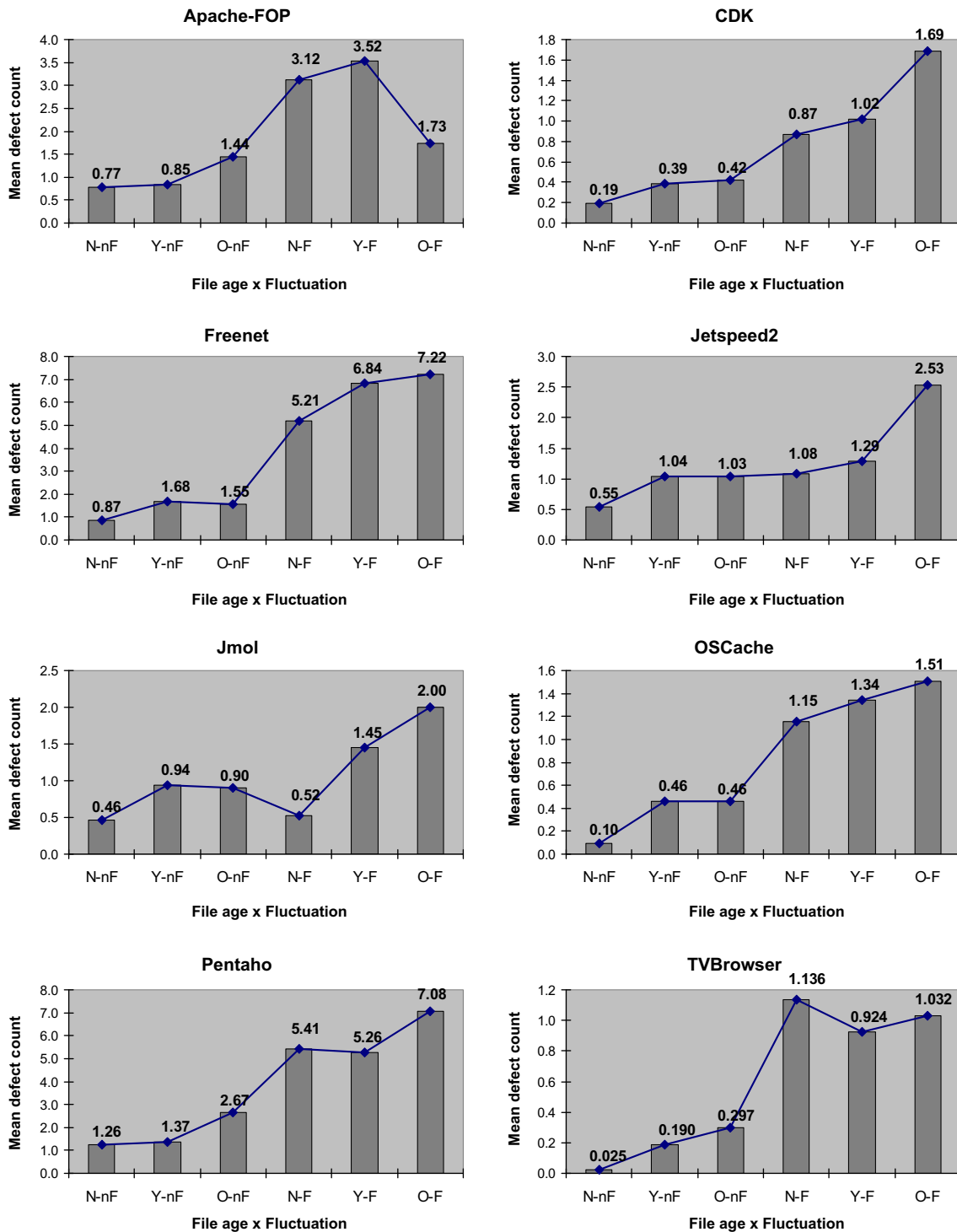


Fig. 10. DVA: mean defect count vs. file age and fluctuation. According to the Kruskal–Wallis non-parametric test, all results are statistically significant at the 0.01 significance level.

Table 11

Comparison of correlation coefficients for several historical characteristics.

ID	OSP	DA	FC	CF-SUM	CF-MAX	Past defects	Release history			
							HF	Post R	Pre R	LM
1	Ant	0.684 **	0.597 **	0.504 **	0.399 **	0.353 **	–	–	–	–
2	Apache-FOP	0.38 **	0.431 **	0.285 **	0.203 **	–	0.458 **	0.501 **	0.455 **	–
3	CDK	0.415 **	0.437 **	0.211 **	0.142 **	0.473 **	–	–	–	–
4	Freenet	0.741 **	0.641 **	0.22 **	–0.02 **	–	0.457 **	0.588 **	0.552 **	0.679 **
5	Jetspeed2	0.741 **	0.641 **	0.22 **	–0.02 **	0.201 **	–	–	–	–
6	Jmol	0.16 **	0.408 **	0.141 **	–0.042 **	0.512 **	–	–	–	–
7	OSCache	0.48 **	0.626 **	0.517 **	0.036 **	0.429 **	–	0.494 **	0.943 **	–
8	Pentaho	0.352 **	0.503 **	0.416 **	0.272 **	–	–	–	–	–
9	TV-Browser	0.471 **	0.442 **	–0.189 **	–0.248 **	–	0.584 **	–	0.384 **	0.618 **

Table 12

Category definition matrix for stability × fluctuation.

		Stability	
		Stable	Unstable
Fluctuation	Fluctuating	F-stab	F-unstab
	Non-fluctuating	nF-stab	nF-unstab

Independent of the file age, **non-fluctuating** files are less error-prone than **fluctuating** files.

Fluctuating Old files are two times (ApacheFOP) to 15 (OSCache) times more fault-prone than **non-fluctuating** and **Newborn** files. In the case of TV-Browser, the factor is as high as 41.

Only in three projects (CDK, Freenet, Jetspeed2), the defect count for **non-fluctuating** files does not differ very much in any of the **Young** and **Old** files. In these cases, **non-fluctuating Young** and **Old** files are about two times more fault-prone than **non-fluctuating Newborn** files. Thus, the file's age is a good indicator for its defect count.

We can conclude that by combining the two metrics (a file's age and its fluctuation or a file's age and its stability), more detailed conclusions on the relationship between these metrics and a file's defect count can be drawn than this is the case when analysing the metrics each taken separately.

Our research hypothesis H3 can be largely confirmed, a file's age is a good indicator for its defect count. 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, as well as Old and fluctuating files are the most error-prone files.

11. Is there a history indicator that persists across all projects?

In order to answer this question, we compare the Spearman rank-order correlation coefficients representing the dependency between historical characteristics and a file's defect count. A detailed analysis of the relationship between the DA, FC and CF metrics and a file's defect count along with descriptive statistics is presented in our previous work [13].¹⁹

Table 11 shows the results of this comparison. The columns contain the Spearman rank-order correlation coefficients for the

¹⁹ On average, 1.14 – 2.91 distinct authors performed HTs to a file. The minimum count of distinct authors is 1 in all projects, whereas the maximum count is 40 authors in case of the Freenet project. The ratio between stable and unstable files ranges from 1.8 to 3.3. The ratio between non-fluctuating and fluctuating files ranges from 0.4 to 7.0. Table 16 summarises basic characteristics of OSPs with respect to the DA metric and the ration between stable/unstable respectively no-fluctuating/fluctuating files.

corresponding metrics. In the column “Past defects”, we included the maximum correlation coefficient for a particular project obtained by the correlation analysis presented in Section 8. We included the corresponding value only in the case when at least half of the analysed releases contain significant correlations above 0.2. The same way we proceeded with the information on the Release history when including correlation coefficients obtained by the analysis in Section 9.

Based on this first analysis, it can be concluded that there is no indicator that shows the strongest correlation in all projects. But the metrics DA and FC prove to be reliable indicators across all projects, i.e. even if in some projects there are stronger correlations than those for DA and FC, these metrics show significant correlations above 0.4 across most analysed projects.

For a detailed analysis, we performed a combined analysis of defect variance. For this purpose, we refined our categories and analysed to what extent the defect count of a file depends on its stability AND on its fluctuation. Thus, we analyse, for example, to what extent **non-fluctuating** files, that have been frequently changed (these are **non-fluctuating** and **unstable** files) are more fault-prone than **non-fluctuating** files that have not been frequently changed (**Old** and **stable** files). Consequently, we relate the mean defect count to each of refined categories, presented in Table 12.

Fig. 11 shows the DVA for all projects. The highest defect counts have **unstable fluctuating** files (**F-unstab**). This is true for 8 of 9 projects. One exception is the Jmol project. In this case, the highest defect counts have **unstable** files that have been changed by distinct authors below average. The lowest defect counts show **stable** and **non-fluctuating** files (**nF-stab**). This is the case for 7 of 9 projects (Ant, Apache-FOP, CDK, Freenet, OSCache, Pentaho and TVBrowser). In other two cases, the defect count for **unstable** files does not differ very much in any of the **fluctuating** and **non-fluctuating** files.

Unstable fluctuating files are three times (Jetspeed2) to 15 (OS-Cache) times more fault-prone than **stable** and **non-fluctuating** files. To obtain statistical evidence, we performed the Kruskal–Wallis non-parametric test. According to this test, all results are statistically significant at the 0.01 significance level.

Based on the analysis results presented in this section and in Section 10, H4 can be confirmed to a certain extent. The number of distinct authors that performed changes to a file (DA metric) as well as the number of HTs (FC metric) show strong correlations with a file's defect count. In combination, these indicators can be used to determine the files that are most fault-prone: In nearly all cases, these are unstable fluctuating files.

As analysed in Section 10, a file's age in combination with its fluctuation and stability is a good indicator for its fault-proneness. In all cases, old fluctuating and old unstable files are the most fault-prone files.

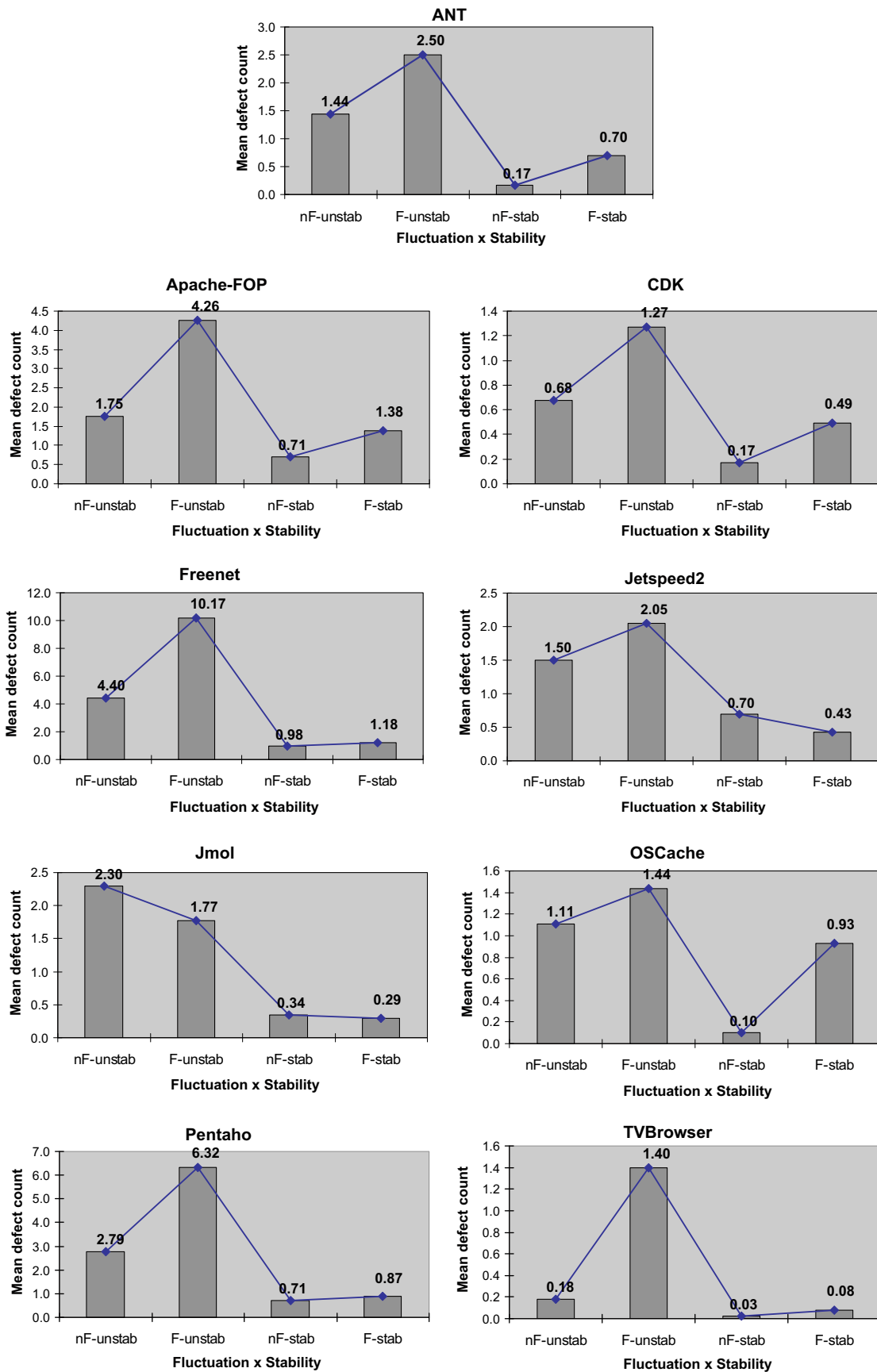


Fig. 11. DVA: mean defect count vs. fluctuation and stability. According to the Kruskal–Wallis non-parametric test, all results are statistically significant at the 0.01 significance level.

12. Discussion

In this Section, we discuss an alternative, coarser grained categorisation for release history and show that this categorisation does not differ significantly from the results obtained in Section 9. In addition, we discuss advantages and disadvantages of analyses on different granularity levels e.g. on file vs. on package level. The more detailed a categorization is the more precise are the results. But increasing the analysis granularity means on the other hand that the effort to evaluate the results increases, too. Thus, a trade-off between a coarse grained (=easy to apply and analyse in practice) and fine grained (=precise results but costly to analyse) has to be performed.

Definition of time bands. To investigate hypothesis H2, we analysed whether a file's release history correlates with its defect count. For this purpose, we subdivided the time between two releases into five phases. These phases represent percentages of time between the considered releases (5%, 10%, etc.). Our results show that there is little statistical evidence that release characteristics as defined in Section 3 correlate with a file's defect count. For the analysed projects, the time between two releases ranges from about 5 months to 2 years. The average time interval between two releases in all projects is about 1 year (1.1 years). Accordingly, the time interval for **HF** and **LM** phases (both denoting 5% of the whole time period between two releases) ranges from 7 to 37 days. The average time interval for **HF** and **LM** is 20 days. The time interval for **PostR** and **PreR** (10% of the whole time period between two releases) ranges from 14 to 74 days. The average time intervals for **PostR** and **PreR** are 40 days. Fig. 13 summarises the different ranges of time intervals in terms of box plots. Fig. 13a shows the box plot for time intervals between two releases for the analysed projects (in years). Fig. 13b shows the box plot for the time intervals for the **HF** and **LM** phases and Fig. 13c shows the box plot for time intervals for the **PostR** and **PreR** phases (in days).

Nevertheless, the results may differ in case of agile projects or projects having short release cycles. In these cases, e.g. 5% of the time may represent only few days. In addition, the definition of release historical characteristics may be adapted for such projects by e.g. comprising several release cycles to one or by subdividing the time between two release cycles into fewer phases.

We performed an additional, coarser grained analysis in which we merged the phases defined before. The resulting phases are **HP** (**hotFix** and **post-Release**) and **PL** (**pre-Release** and **last-**

MinuteFix). The resulting phases denote the first respectively the last 15% of time between two releases. The analysis results are similar to the results obtained by the detailed analysis in Section 9. Only two projects show significant correlations in more than half of the analysed releases for **PL** as well as for the **HP** phases. Only one project shows significant correlations above 0.3 in *all* analysed releases. Thus, this analysis underlines the results obtained in Section 9. Based on the data, there is little statistical evidence for the correlation of a file's release history with its defect count.

Detailed results are shown in Table 13. For each of the aggregated phases (**HP** = **hotFix** and **post-Release**, **PL** = **pre-Release** and **LastMinuteFix**) the table shows the ID and the name of the analysed project. 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.

12.1. Aggregating information

Exploring the history of software projects requires the cleaning up, processing, transformation, analysis and interpretation of a high amount of data. Thus, measurements that synthesize the evolution of a software entity have to be defined [9]. For analysing correlations between a file's defect count and its age, we classified the data into three groups (**Newborn**, **Young** and **Old**). A more detailed classification would lead to more precise results. However, we choose a simple categorisation for the following two main reasons:

- Applicability in practice.* The main advantage of the empirical approach presented in Section 7 is its applicability in practice. The definition of simple categories supports this approach because the more detailed a categorisation is, the more time-consuming to analyse and interpret in practice.
- Metaphorisation:* By having less but meaningful categories, it simplifies the communication and interpretation of the results. It is more difficult to find meaningful names for e.g. 10 categories.

Table 13

Correlation analysis for aggregated release characteristics and defect count.

ID	Project	MAX (Spearman)	MIN (Spearman)	% Releases with sign. corr. above 0.3	% Releases with sign. corr.	% Releases without significant corr.
<i>HP: Hotfix and Post-release</i>						
1	Ant	0.208	* -0.111	** 0	33	67
2	Apache-FOP	0.472	** 0.204	** 25	50	50
3	CDK	0.499	** 0.151	** 17	50	50
4	Freenet	0.451	** 0.359	** 100	100	0
5	Jetspeed2	0.32	** 0.104	** 33	100	0
6	Jmol	0.477	** 0.182	* 33	67	33
7	OSCache	0.494	** 0.494	** 33	33	67
8	Pentaho	0.763	** 0.25	** 50	75	25
9	TV-Browser	0.547	** 0.167	* 40	60	40
<i>PL: Post-release and LastMinuteFix</i>						
1	Ant	0.405	** 0.268	** 33	83	17
2	Apache-FOP	0.366	** 0.228	** 25	75	25
3	CDK	0.544	** 0.192	** 17	50	50
4	Freenet	0.632	** 0.293	** 67	83	17
5	Jetspeed2	0.229	** 0.117	** 0	67	33
6	Jmol	0.561	** 0.2	* 44	56	44
7	OSCache	-	-	-	-	-
8	Pentaho	0.406	** -0.229	** 25	75	25
9	TV-Browser	0.421	** 0.18	* 60	100	0

A trade-off between a coarse grained (=easy to apply and analyse in practice) and fine grained (=precise results but costly to analyse) has to be performed. In addition, an existing categorisation can be refined. This can be necessary when e.g. current results differ significantly from results obtained in past analyses. In addition, the testers' experience may play an important role when deciding to perform detailed analyses. In cases that the results do not reflect the testers' expectations/hypotheses, a detailed categorisation would help to get more precise results.

For analysing correlations between a file's defect count and its change history, we performed correlation analyses. For *detailed* analyses (e.g. combined analyses of defect variance, see Section 6), we classified the data into two groups. The reasons for choosing only two groups are the same as for the independent variable "age". In addition, for combined analyses, the analysis and interpretation complexity increases with the number of categories. Thus, for analysing two independent variables with each three categories, nine (3 × 3) results have to be analysed.

All analyses have been performed at file level. The main reason for not performing analyses on a higher level e.g. on package level is that a package consists of several very heterogeneous files with respect to their age, number of authors performing them, etc. Thus an aggregation (by computing the sum, maximum, average, or median) is difficult and loses too much information. An aggrega-

tion is best suited for e.g. the lines of code metric (LOC). The total LOC of a package has a "meaning" and can be computed by summing up the LOC-metrics of the files/classes contained in it. The loss of information is also the reason for not performing the analyses across all project, i.e. by computing the correlation between one independent variable and the defect count in files across *all* projects. For example, the informative value that in 7 of 9 projects a particular independent variable correlates with a file's defect count is much higher than the indication of a correlation coefficient of e.g. 0.4 computed by merging files of *all* projects together. An aggregated correlation coefficient of 0.4 would not give any information whether all projects show this moderate correlation or whether some projects correlate very strong while others correlate very weak so that the overall result is a moderate correlation.

13. 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 [29]. One threat to validity is the problem of collective check-ins. Collective check-ins denote HTs where a set of files are checked in after a developer has removed two or more defects. Suppose that a developer has removed *defect1* in files A and B and *defect2* in the files B and C. Then, the developer checks in the files A, B, and C with the HT message "... two defects removed ...". The algorithm presented in Section 5.1 would increase the defect count of each of the files A, B and C by two, instead of increasing the defect count in file A and C by 1 and only in file B by two. The following example shows a check-in in the program Ant that contains references to seven bugIDs:

"Fix label length issues Other fixes unearthed after major refactoring of VSS tasks PR: #11562 #8451 #4387 #12793 #14174 #13532 #14463 Submitted by..."

It is not clear if the correction of a defect affected all files that have been checked in conjointly. Thus, collective check-ins represent a threat to validity and can lead to imprecision in the

Table 14
Average, maximum and minimum defect count per HT.

ID	Project	Average defect count per HT	Maximum defect count per HT	Minimum defect count per HT
1	Ant	1.06	7	1
2	FOP	1.57	11	1
3	CDK	1.04	3	1
4	Freetnet	1.34	19	1
5	Jetspeed2	1.04	5	1
6	Jmol	1.02	2	1
7	OsCache	1.16	4	1
8	Pentaho	1.11	2	1
9	TV-Browser	1.04	2	1



Fig. 12. Percentage of HTs for different defect counts per HT. For the sake of clarity, values below 0.7 are not displayed in the chart. Nevertheless, the values are displayed in the data table below the chart.

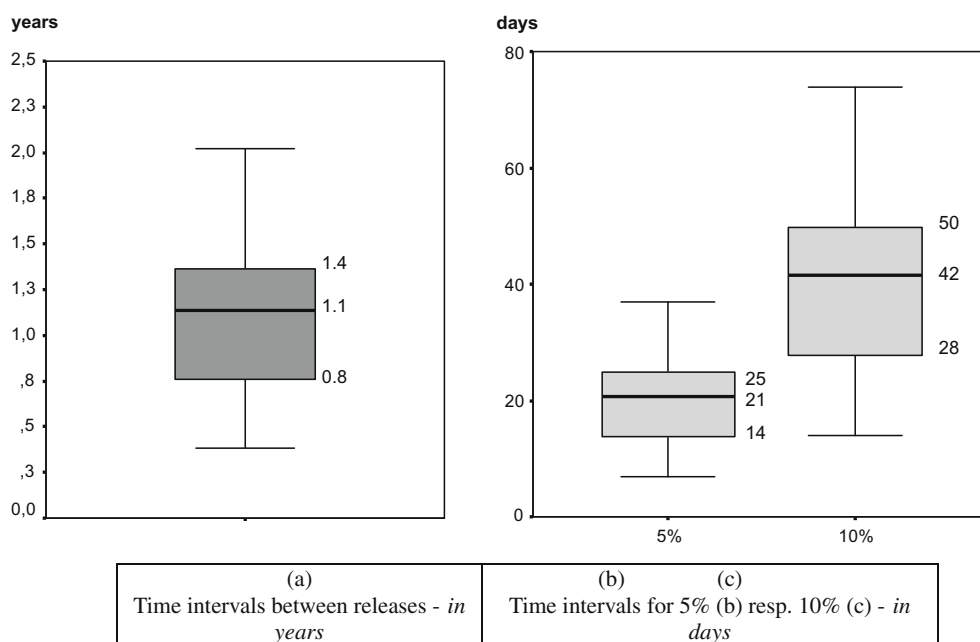


Fig. 13. Time interval between releases and for the phases HF and LM resp. PostR and PreR.

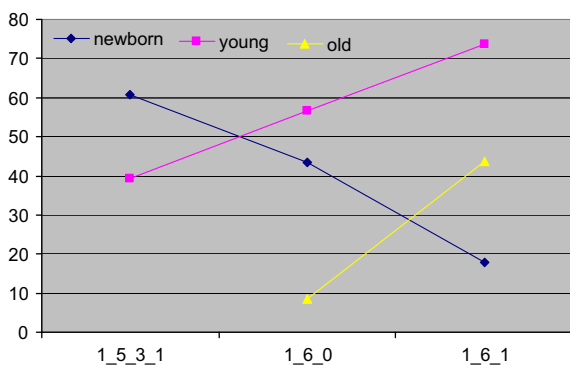


Fig. 14. Percentage of Newborn, Young and Old files per release.

computation of the defect count. The assumption is that such messages are uniformly distributed among all developers and files. Additionally, the average defect count per HT is low in all projects. This fact diminishes the threat to validity.

The average defect count per HT ranges from 1.02 (in case of the Pentaho program) to 1.57 (in case of the ApacheFOP program). In case of three programs (Jmol, Pentaho and TVBrowser), the maximum defect count per HT is only two. Only in case of two programs, Freenet and ApacheFOP, the maximum defect count per HT is above 10. Table 14 summarizes the average, maximum and the minimum defect count per HT for all programs.

In almost all programs, above 90% of the HTs contain references to only a single defect. Only in case of the Pentaho and the Freenet programs, 88.8% respectively 79.9% of the HTs contain a single defect. A very low percentage of the HTs contain two defects. Apart from ApacheFOP and OSCache, nearly zero percent of the HTs contain more than three defects. Fig. 12 shows the percentage of HTs for each project for different defect counts per HT.

Another 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 algo-

rithm. This concern is alleviated by the size of the analysed projects.

External validity is concerned with the degree to which results can be generalized [29]. We choose programs from different application domains in order to increase the representativeness of the study results. However, historical characteristics of open source projects and of commercially produced software may differ from each other. In addition, our results may differ to results which one would obtain when analysing toy projects or projects in a particular domain, e.g. automotive. Furthermore, analyses of additional programs that are intended in our future work would increase the external validity.

14. Related work

In this Section, related work is presented. First, we give an overview on related work concerning the fact extraction, i.e. the problem of determining the defect count of a software entity. Then, we present related work focusing on the analysis of the relationship between software characteristics and software quality.

14.1. Fact extraction to determine the defect count for a software entity

A key problem when analysing defects in software is to compute the defect count of the analysed software entity. Usually, in open source, as well as in commercial versioning control systems, it is not possible to distinguish between a HT that reports a defect (so called defect-correcting HTs) and a HT that reports any other change, e.g. the introduction of new functionality, performed to the software. The authors in [27] report on several empirical studies in which different procedures to determine the defect count of an entity have been applied. Depending on the analysis context, e.g. the kind of software (open source or commercial) and on the automation degree there are several approaches for assigning the type (defect-correcting, non-defect-correcting) to a HT. Table 15 compares the different approaches and gives references to literature where the corresponding approach has been applied. Approach 1 is the only one which is applied at development stage thus being

Table 15
Approaches for classification of HTs as defect-correcting and non-defect-correcting HTs.

Approach	Assumption, description	Automation degree	Limitations	References
Approach 1 Classification by explicit link in the VCS	The VCS system provides the possibility to track the type of HT performed (defect-correcting and non-defect-correcting HT)	Manual assignment of a HT's type when files are checked-in (at development time)	The quality of the classification depends on the discipline to indicate manually the type of the HT. Nevertheless, the most reliable classification approach	[27]
Approach 2 Classification by retrospective manual assignment	Manual assignment is the most reliable	Retrospective manual assignment of a HT's type	Applicable only for small-sized projects	[27]
Approach 3 Classification by keyword analysis	HTs contain keywords within their messages that indicate whether the HT is a defect-correcting or a non-defect-correcting HT	Automated keyword analysis of the HT messages	Misclassification possible: false positives and false negatives Usually, the only possibility to classify HTs in open source projects	[31,26,8,4,13]
Approach 4 Classification by the number of co-changed files	Small changes (affecting 1–2 files) are indicators for defect-correction. Larger changes (affecting more than two files) are indicators for “real” changes (e.g. new functionality)	Automated analysis of the number of co-changed files. Empirical evidence required for the analysed program in order to validate the assumption	Assumption that faults are local does not always apply. It may depend on the program/project if such classification is appropriate	[27]
Approach 5 Classification by the development stage	HTs performed after “official” unit testing are considered defect-correcting HTs	Automated analysis of HTs depending on the phase of their check-in	Overlapping of development and testing phases may lead to misclassification	[27]

Table 16
Statistics for different historical characteristics.

Project	DA	DA		Avg ratio stable/ unstable files	Avg ratio non- fluctuating / fluctuating files
		Max	Min		
1	Ant	17	1	1.9	2.2
2	FOP	14	1	2.1	2.1
3	CDK	10	1	1.8	2.5
4	Freetnet	40	1	3.3	1.8
5	Jetspeed2	7	1	2.4	1.3
6	Jmol	11	1	2.5	2.3
7	OSCache	4	1	2.7	0.4
8	Pentaho	3	1	1.8	5.6
9	TVBrowser	7	1	2.4	7.0

the most reliable approach. Approach 2 is a manual approach, too. But since it is a retrospective assignment, it is only suited for small-sized projects. Approach 3 has been applied in this study. As described in Section 5.1, the HT messages are automatically analysed for containing keywords (e.g. Bug IDs contained in the bug tracking system or keywords like “bug fixed”) indicating that a defect has been removed. Main drawback of this approach is the possibility of misclassification: non-defect-correcting HTs that have been categorized as defect-correcting HTs and accordingly, defect-correcting HTs that have been categorized as non-defect-correcting HTs. In [8] an approach for combining data of VCSs and defect tracking systems is presented. The messages recorded in the VCS are searched for Bug IDs contained in the defect tracking system using regular expressions. In [4], HTs are searched for keyword patterns like “Fixes bug ID” or “ID:”. In [26,30], the authors combined the Bug ID search with the keyword search. In the first step they look for Bug IDs contained in the defect tracking system that are referenced in the text of a HT's message. In order to increase the trust level of the results obtained in the first step, the messages obtained in the first step are search for keywords such as “fixed” or “bug”. The first step of the algorithm presented in Section 5.1 basically corresponds to the approach presented in [26,30]. The second step of the algorithm corresponds to the approach described in [4]. The multi-defects keyword search has not been considered in literature yet. Approach 4 classifies HTs as defect-correcting and non-defect-

correcting depending on the number of Co-Changed files. Approach 5 classifies files with respect to the development stage at which the HT has been performed.

Since each of the automatically computed classification of HTs as defect-correcting or non-defect-correcting rely on assumptions (Approaches 3–5), which when violated lead to misclassifications, the manual classifications proposed in Approach 1 and Approach 2 are more reliable. Due to the drawbacks presented for Approach 3, this approach has been mainly used for analysing open source projects because this is the only possibility to categorize HTs in such projects. Usually, neither in case of commercial systems, nor in case of open source projects the type of HT is tracked. In addition, Approach 2 can only be applied for very small projects. The projects analysed in this study have too many HTs so that a retrospective analysis of all HTs is impossible. The assumption about the locality of the defects (Approach 4) can vary from project to project. In addition, in open source projects, usually the division of the development process into several sub-phases (e.g. integration testing, system testing) is missing respectively is not transparent. Thus, it is not possible to apply Approach 4 and 5 in an open source context.

14.2. Indicators of software quality

To our knowledge, this is one of few studies that analyses the relationship of a file's history and 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 [10,1,15,20,3,25,21,23,31]. One of the main features that distinguish our study from these studies is its magnitude. While most of the studies considered only one program, we have analysed nine projects. Additionally, in contrast to our study, the aim of these studies is defect prediction.

Our main goal is to analyse the extent to which historical characteristics are good indicators for the software's defect count without selecting the best prediction model. Another difference to these studies, except of the study reported in [25], is that all other studies analyse commercial software. In [30], open source and commercial software has been analysed.

In [10,15,20,3,21,23] age is used as an independent variable but the definitions used in these studies differ from our classification.

For instance, in [10,11] 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 the architecture in open source programs 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 [16,10,1,15,20,3,25,21,23]. In [10,15,23], all defects (that occurred in all previous releases) are considered. In [7,25,21], pre-release defects are analysed. In [20,3], the number of defects identified in the prior release is considered. The results are contradictory. The results in [10,3,25,21] confirm our results that previous defects correlate with 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 and defect count has not been analysed empirically previously.

Except the study reported in [25], all other studies [1,10,15,22,20,3,28,25] support our finding with respect to the relationship between the number of changes performed to a software entity and its quality. In [25], only pre-release defects correlate with the number of changes performed to software entities. The authors define pre-release defects to be all defects found 6 months before release.

The studies presented in [3,28,25], and in [10] analyse the relationship between the number of authors performing changes to files and the software's quality. The study reported in [28] confirms our results. In [25], only pre-release defect correlate with the number of authors performing changes. The results reported in [3] and in [10] differ from our results. A possible explanation for this difference is that there is no common understanding of the problem domain in open source development so that changes to a software entity, performed by different developers induce more defects than it is in the case of commercial development as reported in the studies [3,10].

The number of co-changed files is not reported in any study. Most studies analysing this relationship are more fine-grained, i.e. they analyse the extent to which the number of changed lines of code impacts on the defect count, for example in [18,17] and in [24].

A huge amount of research papers analyse the relationship between other metrics of a software entity and its defect count, amongst others in [5,2,6,19] and in [11].

15. Conclusion and future work

In this paper, we investigated the relationship between a file's historical characteristics and its defect count. Contrary to our expectation, the defect count of a previous release of a file does not correlate with 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% and 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.

Our results show that a software's history is a good indicator for its quality. In this study, we express quality by the number of defects. We did not find one indicator that persists across all projects in an equal manner. Nevertheless, there are several indicators that show significant strong correlations in nearly all projects: DA (number of distinct authors) and FC (frequency of change).

One explanation of the strong correlations between a file's DA metric and its defect count is the lack of responsibility for that particular file that leads to uncoordinated and fault-prone changes so that “too many cooks spoil the broth”. A second possible explanation is that files that are changed by many authors capture too much functionality that is used and changed by a lot of authors. Thus, these files are indicators for bad design leading to a high defect count.

Strong significant correlations of the FC metric with a file's defect count indicates that particular parts of the application are not well understood and often need rework. Consequently, these files are fault prone.

The empirical results did not support all hypotheses concerning the influence of a file's age on its defect count. In fact, 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**). But in contrast to our expectation, **old** files proved to be the most fault-prone files.

Detailed analyses can be performed in order to get more precise results and to restrict the set of defect prone files. For this purpose, the relationship between more independent variables and a file's defect count can be determined. We analysed whether a file's stability and its age in combination are indicators for its fault-proneness. The most fault-prone files are **old** files that have been changed above average. In addition, in nearly all projects, the youngest **stable** files – the **Newborn** files – have the lowest defect count. Similarly, we also analysed whether a file's fluctuation and its age in combination are good indicators for its defect count. Fluctuation is an indicator for the number of distinct authors that performed changes to that file. **old fluctuating** files are the most fault-prone files whereas **Newborn non-fluctuating** files have in average the lowest defect count. One reason for these results might be that **unstable fluctuating Old** files are indicators for bad design. Every time a change occurs, **old** files are also affected, which causes defects in each release. In addition, these files lack of responsibility so that a lot of authors perform changes that lead to defects, too.

By analysing different indicators in combination, more detailed results can be derived. Such a detailed analysis should be done in order to specify the results obtained by a simple analysis. Thus, the set of fault-prone files can be constrained. In this study, we analysed the relationship between a file's stability, its fluctuation and the defect count. The results show that **unstable fluctuating** files are 3–15 times more fault-prone than **stable** and **non-fluctuating** files. In one particular case, the factor is as high as 40. In addition, we analysed the influence of a file's stability and age as well as of a file's fluctuation and age on its fault-proneness. **Unstable old** files are 3–15 times more fault-prone than **stable Newborn** files. In one particular case, the factor is as high as 47. Similarly **fluctuating Old** files are 3–22 times more fault-prone than **non-fluctuating Newborn** files. In one particular case, the factor is as high as 85.

Knowing which particular historical characteristics are indicators for a file's quality (in this study expressed by its defect count) 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 and fluctuating** files. Quality engineers can monitor development activities and initiate reviews, e.g. for often changed **old** files in order to prevent a high defect count. Additionally, **old** files changed too often by a number of authors above

average and causing high defect counts can be indicators for bad design. Thus, maintainers can identify candidates for refactorings.

All analyses presented in this paper have been performed by applying an empirical approach that gives guidance in finding indicators for (poor) software quality. This approach consists of several steps: after the identification of the software and software releases, the granularity of the analyses (e.g. analyses on file level) has to be decided. Next, the definition of the dependent variable (the definition of how quality will be expressed) and of the independent variables (e.g. the definition of all historical characteristics) has to be performed. During the measurement step, all data has to be collected. The next two steps consist of analyses that have to be performed depending on the scale of the variables. Finally, the results have to be synthesized and conclusions have to be derived. The approach has several strengths. First, it is easy to understand. In order to be applicable in practice, the approach has to be easily comprehensible and intuitive. For this purpose, the results should not be encrypted within complex formula. In addition, visual representations for the analyses are used in order to enable a standardized intuitive interpretation of the results. Second, the approach follows statistical procedures. All results obtained by visual means are statistically validated. Thus, more reliable decisions can be taken because the probability of accidental effects is minimized.

Our future work will focus on analysing further 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 historical 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 historical characteristics improve the quality of the indicators that are based on code characteristics only. For instance, we expect that **o1a**, often changed and complex files are more fault-prone than **o1d** and complex files that have not been changed frequently.

References

- [1] E. Arisholm, L.C. Briand, 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, 2006, pp. 8–17.
- [2] V.R. Basili, L.C. Briand, W.L. Melo, A validation of object-oriented design metrics as quality indicators, IEEE Transactions on Software Engineering 90 (1996) 751–761.
- [3] R.M. Bell, T.J. Ostrand, E.J. Weyuker, Looking for bugs in all the right places, 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, 2006, pp. 61–72.
- [4] D. Cubranic, G.C. Murphy, Hipikat: recommending pertinent software development artefacts, in: 25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, 2003, pp. 408–418.
- [5] G. Denaro, M. Pezzè, An empirical evaluation of fault-proneness models, in: Proceedings of the International Conference on Software Engineering (ICSE 2002), Orlando, Florida, USA, 2002, pp. 241–251.
- [6] G. Denaro, S. Morasca, M. Pezzè, Deriving models of software fault-proneness, in: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering Ischia, Italy, 2002, pp. 361–368.
- [7] N.E. Fenton, S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach: Brooks/Cole, 1998.
- [8] M. Fischer, M. Pinzger, H. Gall, H. Populating a release history database from version control and bug tracking systems, in: Proceedings of the International Conference on Software Engineering (ICSE 2002), Orlando, Florida, USA, 2002, pp. 241–251.
- [9] T. Girba, M. Lanza, S. Ducasse, Characterizing the evolution of class, in: Software Maintenance and Reengineering, CSMR 2005, Ninth European Conference on Hierarchies, 2005, pp. 2–11.
- [10] T.L. Graves, A.F. Karr, J.S. Marron, H. Siy, Predicting fault incidence using software change history, IEEE Transactions on Software Engineering 26 (2000) 653–661.
- [11] T. Gyimothy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Transactions on Software Engineering 31 (10) (2005) 897–910.
- [12] L. Hatton, The Role of Empiricism in Improving the Reliability of Future Software, Keynote Talk at TAIC PART 2008, <<http://www.leshatton.org/Documents/TAIC2008-29-08-2008.pdf>>, 2009.
- [13] T. Illes-Seifert, B. Paech, Exploring the relationship of history characteristics and defect count: an empirical study, in: Proceedings of the 2008 Workshop on Defects in Large Software Systems (Seattle, Washington, July 20–20, 2008), DEFECTS '08, ACM, New York, NY, 2008, pp. 11–15.
- [14] International Software Testing Qualifications Board, ISTQB Standard Glossary of Terms Used in Software Testing V1.1, 2005.
- [15] T.M. Khoshgoftaar, E.B. Allen, R. Halstead, G.P. Trio, R. M Flass, Using process history to predict software quality, Computer 31 (4) (1998) 66–72.
- [16] S. Kim, T. Zimmermann, E.J. Whitehead Jr., A. Zeller, Predicting faults from cached history, in: Proceedings of the 29th International Conference on Software Engineering (May 20–26, 2007), International Conference on Software Engineering, IEEE Computer Society, Washington, DC, 2007, pp. 489–498.
- [17] L. Layman, G. Kudrjavets, N. Nagappan, Iterative identification of fault-prone binaries using in-process metrics, in: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (Kaiserslautern, Germany, October 09–10, 2008), ESEM '08, ACM, New York, NY, 2008, pp. 206–212.
- [18] N. Nagappan, T. Ball, Use of relative code churn measures to predict system defect density, in: Proceedings of the 27th International Conference on Software Engineering (St. Louis, MO, USA, May 15–21, 2005), ICSE '05, ACM, New York, NY, 2005, pp. 284–292.
- [19] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in: Proceedings of the International Conference on Software Engineering (ICSE 2006), Shanghai, China, 2006, pp. 452–461.
- [20] T.J. Ostrand, E.J. Weyuker, R.M. Bell, Predicting the location and number of faults in large software systems, IEEE Transactions on Software Engineering 31 (2005) 340–355.
- [21] T.J. Ostrand, E.J. Weyuker, 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, 2002, pp. 55–64.
- [22] M. Ohlsson, A. von Mayrhauser, B. McGuire, C. Wohlin, Code decay analysis of legacy software through successive releases, in: Proceedings of the IEEE Aerospace Conference, 1999 (Section 7.401).
- [23] M. Pighin, A. Marzona, An empirical analysis of fault persistence through software releases, in: International Symposium on Empirical Software Engineering, 2003, pp. 206.
- [24] R. Purushothaman, Toward understanding the rhetoric of small source code changes, IEEE Transactions on Software Engineering 31 (6) (2005) 511–526.
- [25] A. Schröter, T. Zimmermann, R. Premraj, A. Zeller, If your bug database could talk, in: Proceedings of the 5th International Symposium on Empirical Software Engineering, Short Papers and Posters, vol. 2, 2006, pp. 18–20.
- [26] J. Sliwersky, T. Zimmermann, A. Zeller, When do changes induce fixes? On Fridays, in: Proceedings of the International Workshop on Mining Software Repositories (MSR 2005), St. Louis, Missouri, USA, 2005, pp. 1–5.
- [27] E.J. Weyuker, T.J. Ostrand, Comparing methods to identify defect reports in a change management database, in: Proceedings of the 2008 Workshop on Defects in Large Software Systems (Seattle, Washington, July 20–20, 2008), DEFECTS '08, ACM, New York, NY, 2008, pp. 27–31.
- [28] E.J. Weyuker, T.J. Ostrand, R.M. Bell, Using developer information as a factor for fault prediction, in: Proceedings of the Third international Workshop on Predictor Models in Software Engineering (May 20–26, 2007), International Conference on Software Engineering, IEEE Computer Society, Washington, DC, 2007, p. 8.
- [29] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, 2000.
- [30] T. Zimmermann, N. Nagappan, A. Zeller, Predicting bugs from history, in: T. Mens, S. Demeyer (Eds.), Software Evolution, Springer, 2008, pp. 69–88.
- [31] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE), Minneapolis, USA, 2007, pp. 9–9.