Why Software Repositories Are Not Used For Defect-Insertion Circumstance Analysis More Often: A Case Study

Lutz Prechelt[a], Alexander Pepper[b]

[a]*Freie Universität Berlin, Berlin, Germany*
[b]*Infopark AG, Berlin, Germany*

Abstract

Context: Root-cause analysis is a data-driven technique for developing software process improvements in mature software organizations. The search for individual process correlates of high defect densities, which we call *defect insertion circumstance analysis* (DICA), is potentially both effective and cost-efficient as one approach to be used when attempting a general defect root cause analysis. In DICA, data from existing repositories (version archive, bug tracker) is evaluated largely automatically in order to determine conditions (such as the people, roles, components, or time-periods involved) that correlate with higher-than-normal defect insertion frequencies. Nevertheless, no reports of industrial use of DICA have been published. *Objective:* Determine the reasons why DICA is not used more often by practitioners. *Method:* We use a single-case, typical-case, revelatory-type case study to evaluate in parallel the importance of six plausible reasons (R1 to R6). The case is based on 11 years of repository data from a small but mature software company building a product in the high-end content management system domain and describes a four person-months effort to make use of these data. *Results:* While DICA required non-negligible effort (R3) and some degree of inventiveness (R2), the most relevant roadblock was insufficient reliability of the results (R6) combined with the difficulty of assessing this reliability (R5). We identify three difficulties that led to this outcome. *Conclusion:* Current repository mining methods are too immature for successful DICA. Gradual improvements are unlikely to help; different principles of operation will be required. Even with such different techniques, issues with input data quality may continue to make good results difficult-to-have.

Keywords: mining software repositories, version archive, bug tracker, defect, root cause analysis, bug, bugfix

1. Introduction

Mining software repositories (MSR) is a set of techniques that exploit the data stored in existing databases such as source code version archives or issue tracking databases in order to obtain relevant insights (general or specific) about software products or software development processes. One of the potentially useful application areas when mining software repository data is understanding defect insertion and defect removal processes. More specifically, a process that we will call *defect insertion circumstance analysis* (DICA) attempts to identify correlates of defect insertions: when, where (contexts), and how they happen and who makes them happen. Such insights can potentially be used to make valuable process improvements.

The Case Study in the sense of Yin (2003) is a research method most suitable when "a 'how' or 'why' question is being asked about a contemporary set of events over which the investigator has little or no control".

1.1. Research question

The present article is a case study aimed at the research question formulated in the title: Why aren't MSR techniques used for DICA more often than they are? The question is based on the observation that, although the MSR community strives to address and include practitioners, there are not many reports on MSR usage from practitioners – which may be understandable for some types of MSR but appears surprising for DICA.

We answer it by investigating a single, arguably common (see below) case of industrial DICA as it occured in its real context.

1.2. Research contribution

The answer we found for the research question is twofold: First, there appears to be no affordable method for assessing the reliability of the results obtained from a DICA. As a consequence, practitioners find DICA to be too risky to be worth its substantial effort. Second, the reliability of these results appears to be low. As a consequence, practitioners cannot expect sound answers from a DICA and will hence not find it sufficiently valuable.

Our article makes the following research contributions:

- It presents results of a 4-person-month DICA attempt for a large 11-year industrial software repository performed by practioners at software company Infopark. We are not aware of any other such report from an explicitly described industrial setting.

Email addresses: prechelt@inf.fu-berlin.de (Lutz Prechelt), alexander.pepper@infopark.de (Alexander Pepper)

- It sorts out, at least roughly, the relative contributions of five different potential reasons (R2 to R6) for not using DICA more often.
- It demonstrates the dominant weight of the two reasons mentioned above. The second of these is known in the MSR community and is being worked on, but the first, although rather fundamental, does not currently receive much attention at all.

1.3. Structure of this article

Section 2 introduces the domain of study. It introduces terminology in 2.1, explains the basics of MSR in 2.2, describes the procedure and potential benefits of DICA in 2.3, and discusses the research question's foundation: whether DICA is indeed used rarely (2.4).

Section 3 explains the design of the case study. It introduces the Infopark DICA case (3.1, 3.2), lists possible reasons why DICA is performed rarely (3.3) which we will use as the propositions to be investigated by the case study, describes the sources of evidence and the forms of triangulation used during the case study (3.4), and explains why a single-case study is satisfactory in this particular situation (3.5).

Sections 4 to 7 describe the four major phases of Infopark's DICA attempt:

- establishing bugfix links,
- discriminating true defects from other "bugs",
- mapping defect corrections to defect insertions, and finally
- performing the actual DICA.

Each of these sections describes what Infopark did, what problems it encountered, how it handled those problems, what results it obtained, and then inteprets these facts for the purposes of the case study. Section 8 discusses threats to validity.

Section 9 discusses inhowfar the results should be viewed as new when considering related work by other researchers and Section 10 presents conclusions.

2. The domain of study: MSR and DICA

2.1. Terminology

We choose our terms such as to make the discussion in the present article simpler; these definitions are not intended to be fit for general purposes. In particular, we constrain our discussion to phenomena observable on the level of program source code, because that is the sort of information our data sources provide.

Many of these terms talk about fuzzy phenomena so some of the definitions are unavoidably vague. This fuzzyness is in fact an important phenomenon in our case study, but resolving it is not our goal, so we do not aim at making the definitions maximally precise.

- A *change* is a set of additions, deletions, and modifications to existing software that are being checked into the source code version archive together. It is represented by the check-in transaction's source code delta and its commit message.
- A *defect* is a property of source code that triggers avoidable rework (possibly outside the observed timeframe).
- *Rework* is any modification performed on a section of program source code that was written and checked in earlier. In our analysis, the unit of rework is a single change (check-in transaction). We call rework *avoidable* if the need to make that change was in principle known at the time of the original work. We call rework *unavoidable* if that need arose only later or could (from the point of view of the software developers) be known only later.
- An *issue* is a property of software that is addressed in unavoidable rework. In practice, it is often difficult to discriminate defect and issue, which turns out to be important in our study.
- *Bug* is a synonym for either a defect or an issue in everyday software developer language and the discrimination is often not made. A *bugtracker entry* is an entry in a change request database and addresses either a defect or an issue.
- A *bugfix* is rework (specifically: a change) that is intended to partially or fully resolve the defect or issue described by a bugtracker entry (even if that intention is not achieved). A *bugfix link* is a pair of a bugtracker entry and its corresponding bugfix.
- A *defect correction* is a bugfix whose bugtracker entry describes a defect (rather than an issue).
- A *defect insertion* is a change that introduces one or more defects into the software. Note that a defect correction may introduce new defects as well.

2.2. Mining software repositories (MSR)

In mature software software processes, it will often be useful to employ quantitative data obtained during process execution to optimize day-to-day project control and to spot general improvement opportunities for the process as a whole (Haley, 1996). Such behavior is for instance suggested by the process areas of CMMI levels 4 and 5 (CMMI Product Team, 2010), in particular the Causal Analysis and Resolution level-5 process area and its Determine Causes of Selected Outcomes goal. Unfortunately, obtaining and analyzing suitable data can be costly. One attractive approach for overcoming this cost problem would be using data that is collected anyway and automating its analysis (Cook et al., 1998). This approach has been followed for a number of years by a community working on "mining software repositories" (MSR) that formed after some initial works in

the 1990s (Graves and Mockus, 1998; Graves et al., 2000) and holds a yearly workshop/conference called MSR since 2004[1].

MSR is defined as "analyz[ing] the rich data available in software repositories to uncover interesting and actionable information about software systems and projects" (MSR 2014 call for papers) and the repositories in question are for instance "source control systems, archived communications between project personnel, and defect tracking systems" (MSR 2014 call for papers), requirements management databases, or project planning databases.

Topics in MSR include general infrastructure tasks such as data extraction and cleansing as well as many types of applications, for instance "characterization, classification, and prediction of software defects", "analysis of change patterns and trends", "prediction of future software qualities", building "models for social and development processes", "models of software project evolution", and reliability models, or supporting "search-driven software development" (all from MSR 2014 call for papers).

2.3. Defect insertion circumstance analysis (DICA)

Due to the huge fraction of work that software processes tend to spend on avoidable rework (Haley, 1996; Shull et al., 2002), defects are a natural area of huge interest for MSR research. The specific goals may be formulated differently (such as defect characterization versus defect prediction), but, at least from a practitioner's point of view, the goal is always similar: Understand past problems in order to take efficient precautions against future problems.

The CMMI Level 5 process area CAR (Causal Analysis and Resolution) suggests that for mature software organizations, performing defect root cause analysis is a valuable step for feeding process improvements. Such improvements are attractive because of their constructive nature: Rather than constantly spending large amounts of work on provoking failures (testing) and then removing the corresponding defects, a much smaller amount of work (sometimes only a one-time effort) is spent on avoiding those defects in the first place.

If successful, MSR-based DICA semi-automates a sizable fraction of the root cause analysis work by answering questions such as the following (among others):

- Are certain developers a strong source of defects (no matter whether this is due to lack of skill, lack of care, lack of knowledge about the code they need to work on, frequency of interruptions, difficulty of their tasks, or some other reason)?
- Are particular modules of the product involved in defective changes overly often?
- Are particular kinds of design constructs conspicuously often involved in defective changes?
- Do certain times of day or days of the week or weeks of the year appear particularly defect-prone?
- Are certain phases during the release cycle particularly defect-prone?

As a result, the organization performing the analysis for instance

- might find that the defect insertion density rises pronouncedly in a particular time stretch during each release process (and then react by changing that process);
- might conclude that the high defect density produced by certain developers is due to a clumsy design of these developers' roles in the process (and then react by modifying those roles);
- might determine the density of certain types of defect in a particular module to be much higher than warranted (and then react by reengineering that module).

Such results are only correlational in nature, not causational, but may help narrowing down possible root causes or help to improve the process even without knowing the actual root cause of some effect.
The prodecure for performing DICA basically consists of the following steps:

1. Identify all bugfix links.
2. Remove bugfix links that pertain to issues rather than defects.
3. For each defect-correcting change, identify the corresponding defect-inserting change (defect correction to defect insertion mapping, DCDIM).
4. Tabulate the defect insertions appropriately in order to answer the above questions (DICA proper).

By some initial investment, all four steps can in principle be fully automated, only the interpretation of the step 4 results should be manual.

Do not confuse DICA with defect prediction. Defect prediction is a practice that belongs to the CMMI Level 4 process area QPM (Quantitative Project Management). Defect prediction is (mostly) about allocating resources for product cleanup and analytical quality assurance (testing and reviews); defect prediction models can make use of predictor factors not accessible for intentional change. In contrast, DICA is about improving the process itself and so has to focus on factors allowing direct intervention. There is a lot of work (some quite successful) on defect prediction (Hall et al., 2012), but little on DICA.

2.4. Is DICA really rare?

The MSR community has always considered itself as oriented towards practical application of the techniques and wanted to involve practitioners, not only researchers, expressed by the explicit goal of the MSR conference to form and support a "community of researchers and practitioners" (MSR 2005, 2006, 2007, and 2008 calls for papers). Alas, few practitioners ever

[1] http://www.msrconf.org, all MSR calls for papers can be found here. The references and quotations from this website are as of 2012-04-26.

spoke up at MSR which may explain why the above formulation turned into "the science and practice" since the 2009 CfP. However, the ambition of practical applicability is still there as is shown by the call for "actionable information" mentioned above as well as the fact that the CfP topic "Case studies on extracting data from repositories of large long lived projects" (MSR 2005 to 2009 calls for papers) defiantly turned into "[...] of large long-lived and/or industrial projects" since 2010.

More concretely, we will consider the proceedings of MSR 2011, 2012, and 2013. The proceedings of MSR 2011 contain 20 full-length papers and 6 short papers, plus a set of another 6 short papers all reporting mining results for the same set of repositories ("the mining challenge"), but on various topics and questions.

Of these 32 contributions, 9 are concerned largely or mostly with defect data: three employ defect-related data in order to study one particular pre-specified aspect, such as operating system compatibility issues (Wang et al., 2011) characteristics of the social networks involved in handling inter-project defects (Canfora et al., 2011), or differences in defect-correction processes between security defects and performance issues (Zaman et al., 2011) One compares five text retrieval methods for the task of mapping a bug description to a set of files it may pertain to (Rao and Kak, 2011). One contribution evaluates the quality of existing statistical models that explain bug-fixing time (Bhattacharya and Neamtiu, 2011), and two evaluate methods for predicting defect-proneness (Sadowski et al., 2011; Giger et al., 2011). One article performs DICA (Eyolfson et al., 2011).

All of these come from academic researchers and almost all of the data is from open source projects. Only two defect-related contributions come from industrial participants. The first comes from IBM research (researchers, not practitioners) and evaluates which statistic is best tracked for understanding trends of time to close bug reports (as a maintenance efficiency measure): the mean or percentiles (Zeltyn et al., 2011). The remaining one is the only one from practioners and comes from Cisco. Its topic are the practical issues of establishing a set of metrics (that aim at measuring overall software and software process quality) throughout the company (Rotella and Chulani, 2011).

Similar numbers hold for MSR 2012; its number of DICA studies is one: an academic short paper on open source data that performs no validation of its results whatsoever (Asaduzzaman et al., 2012). For MSR 2013, the number of DICA studies is zero. As we see, (a) MSR contributions from practitioners appear quite rare and (b) there is more breadth in the research questions than in the number of reported cases reported for the same question – and this is both similar when looking at previous MSR years or other venues. We know of not one single article reporting on DICA by practitioners.

This scarceness is in sharp contrast to the conclusions of the first DICA use reported in the literature (Śliwerski et al., 2005). The article studies Mozilla and Eclipse, finds that larger changes are more defect-prone, finds that defect-proneness is highest on Fridays, and proceeds to promise "a wide range of applications" and also that the "findings can be generated automatically for arbitrary projects.". Which sounds both great and easy.

So the question "Why does DICA not appear to be used in practice?" is wide open. Our current article answers it.

3. Design and method of the case study

3.1. The case: DICA for Infopark CMS Fiona

Infopark was founded in 1994 and built the first version of its main product CMS Fiona in 1997. CMS Fiona is a content management system for large-scale, high-traffic web sites with both static and dynamic parts. It provides strong consistency-keeping mechanisms for static content.

The following properties of Infopark, CMS Fiona, and the development process used are relevant for the current study:

a). In the CMS domain, feature requests are frequent and there is no clear line between defects and issues. Consequently, Infopark often treats the implementation of small improvements to the functionality just like defects and such improvements represent a substantial fraction of the "bugfix" data. Because feature requests typically involve more code than defect corrections, the data contains many non-small bugfixes.

b). Infopark has always had low turnover of staff and is therefore able to follow intended processes and good practices stably (as described by the subsequent items).

c). The commit message of a bugfix commit typically mentions the number of the corresponding bugtracker entry and often vice versa, so bugfix links could be established in sufficient density (high-enough recall) and with high precision.

d). Bugfixes are committed separately, that is, any one bugfix commit typically addresses only a single bugtracker entry, not several.

e). The first bugfix would sometimes not eliminate the problem in a satisfactory way but rather provide a quick correction of the failure only (preliminary fix); a second bugfix may then provide a cleaner and more complete solution later.

f). To keep the code in good shape, refactorings (in particular renames of member variables or methods) and other reorganization and cleanup work are performed routinely, also as part of bugfixes.

In spring 2011, Infopark assigned one of its software engineers[2] the task to find out whether and how the repository data could be used to obtain useful insights about the Fiona product and/or the process by which it is developed. The idea was to pick low-hanging fruit and gain experience with mining (specifically the Infopark repositories) underway. No specific questions were imposed and any kind of insight was considered potentially valuable.

[2]This person is the second author of this article, Alexander Pepper. He had been working at Infopark for several years.

Until the fall of 2011, Pepper spent more than 4 person months on designing, preparing, debugging, executing, and interpreting an MSR analysis. Infopark decided it wanted to understand how practical and useful DICA might be and so DICA was set as the analysis goal[3].

3.2. Infopark's DICA quality criteria

Infopark considered *incorrect conclusions* to be a major risk for DICA, as those might lead to fruitless process change effort or even counterproductive process changes. Infopark decided that the analysis methods must be such that, when searching for phenomena of type X (such as "a time of day (hour) during which the likelihood that a commit is defective is at least 33% higher than on average"), at least half of all actual X phenomena must be found (i.e. 50% DICA recall) and that at least four out of any five candidate X phenomena thus found must be true X rather than false positives (80% DICA precision).

High recall is needed to avoid acting on a view of reality that is very incomplete and hence potentially not representative. High precision is needed to avoid acting on something that is just misleading analysis noise and not real at all.

3.3. Propositions: Possible reasons why DICA is rare

If DICA is rarely used by practitioners, any case of non-use will be either because the use is not feasible for some reason (including not knowing about DICA techniques at all) or because the cost/benefit ratio appears unattractive. Note that the crucial factor is not the true cost/benefit ratio but rather the expected ratio as it is estimated when making the decision whether to apply DICA or not.

Splitting these possibilities up a bit more, we come up with the following presumably comprehensive list of possible reasons why DICA is not often used:

- **R1**. The practitioners do not know that MSR DICA techniques exist.
- **R2**. The practitioners find the application of MSR DICA techniques too difficult to be feasible ("too difficult").
- **R3**. The practitioners find the absolute amount of work involved in applying MSR DICA techniques too high to make the step at all ("too laborious").
- **R4**. The insights suggested by the DICA results are perceived as uninteresting or as unusable for driving process improvement steps ("not useful").
- **R5**. The result data quality cannot be assessed well enough to judge the suitability of the techniques ("unknown reliability").
- **R6**. The result data quality cannot be made high enough so that practitioners will invest into the resulting process improvement steps with confidence ("insufficient reliability").

We will use these reasons as the propositions to be evaluated by our case study; the goal being to shed light on the relative importance of each. As R1 was not present in the Infopark case, we have nothing to say about it. Our main contribution will be analyzing the problem structure underlying R5 and R6.

The strength of the case study method lies in its ability to help untangle multiple competing explanations of a phenomenon (Yin, 2003, pp. ix-x). In this sense, R1 through R6 are competing explanations of the fact that so few uses of industrial DICA appear to occur and our study describes how R5 and R6 are the strongest ones of these.

3.4. Sources of evidence and use of triangulation

Two characteristics are key to provide case studies with this untangling capability: First, using multiple sources of evidence (rather than just one) in order to provide a broad evidence base and hence reduce the influence of the limitations that any single source of evidence is bound to have. Second, using triangulation in order to make the analysis more robust. Triangulation means using multiple perspectives of the same phenomenon together so that they either corroborate each other or ill-chosen perspectives can be recognized as such and sorted out.

In the present case study, we directly or indirectly use many sources of evidence, in particular

1. the context information as described in Section 3.1 and beyond,
2. the quality postulates as described in Section 3.2,
3. the raw data of the version archive,
4. the raw data of the bugtracker,
5. the analysis steps taken during the DICA,
6. the issues encountered in taking those steps,
7. the results obtained from those steps,
8. some potential steps that were not actually taken,
9. the arguments made for deciding for a step or against a potential step,
10. and finally Infopark's interpretation of the steps' results.

The decisions and the interpretations sometimes involve Pepper only and sometimes other Infopark employees as well.

As for triangulation, Yin (2003) discusses four types, two of which we apply in the study: We view some aspects of the phenomenon under study (low use of DICA by practitioners) in light of several of our sources of evidence; this constitutes *data triangulation*, the most common form of triangulation in case studies. Furthermore, as each of the reasons R1 through R6 is an alternative and potentially comprehensive explanation of the low DICA use, they can be interpreted as competing theories of low DICA use. Therefore, whenever we consider more than one of the propositions at once in the study, this constitutes a straightforward form of *theory triangulation*.

[3] Infopark considers itself an agile organization and so does not apply the CMMI. If it did, it would be roughly at CMMI Level 2, but, being a small organization, it would apply CMMI's continuous representation rather than going for the staged representation's Level 3. Therefore, attempting a Level 5 practice such as DICA is ambitious but is not an outrageous idea.

3.5. Why just a single case?

The present case study investigates just one single case. Obviously, a case study's convincingness can be higher (in particular with respect to the generalizability of its results) if it considers multiple cases. So why should the reader be satisfied with only the Infopark case?

Yin (2003) lists five types of situation in which a single case can be sufficient for a valuable and convincing result: (1) a critical case, (2) an extreme or unique case, (3) a typical or representative case, (4) a revelatory case, (5) a longitudinal case.

Two of these apply here. First, we consider the Infopark case not unusual and a suitable stand-in for a relevant class of software organizations. In this constrained sense, our case is a typical case. Too few industrial MSR studies have been published to provide strong evidence for that claim – or against it. Second, and more importantly, we know of no MSR study to make the point that ours does (namely that assuring the reliability of the DICA results is so hard in practical settings as to make the approach worthless), so the Infopark case can be considered a revelatory one as well.

3.6. Structure of the case presentation

We will now proceed to present the case itself as it unfolded in four major steps or phases: (1) establishing bugfix links, (2) discriminating defects from issues, (3) mapping defect corrections to defect insertions (DCDIM), and finally (4) performing the actual DICA.

Each of these sections has two subsections. The first subsection[4] is on the data level (case level) and describes what Infopark did, what problems they encountered in doing it, and what results they obtained. The second is on the case study level and interprets these facts in terms of the propositions and case study goals.

4. Step 1: Establishing the bugfix links

4.1. Case: Steps, issues, results

A complete and accurate set of bugfix links is a major requirement for the trustworthyness of subsequent DICA results, as any error in the bugfix links will directly deteriorate the quality of those results. Consequently, Infopark spent more than 20% of the work time allocated for the DICA on preparing a high-quality set of bugfix links.

That work, which is described in detail in a separate article (Prechelt and Pepper, 2014), started at a version archive (CVS, SVN, and Git) of 11 years of development (more than 45,000 commits) and a Bugzilla database from 8 years of development (9,444 entries from 2003 to 2011, so the data reflects a mature product stage only) in which 5,005 bugfix link candidates had been identified, which refered to 4,499 different commits and 3,203 different Bugzilla entries. These candidates were found by simply collecting all potential ID reference numbers found anywhere in commit messages, bug entry descriptions, and bug entry comments and so are likely to be fairly complete but to contain many false positives. As for the completeness, we surveyed Infopark developers for the fraction of commits that are bugfix commits and the fraction of those that have a corresponding Bugzilla entry. The means of the answers suggested there ought to be 5,072 bugfix links overall, so that our candidate set might be up to 98.7% complete. The individual estimates varied a lot, however, so the overall estimate should be considered rough. Also, theoretically any pair of commit and Bugzilla entry is a potential bugfix link, so there are 425 million theoretical candidates, which makes an accurate determination of recall impractical. Instead, our recall calculations are based on our set of 5005 candidates and should thus also be considered rough.

Pepper devised a set of five filters to cut down the set of candidates to the best possible approximation of the actual bugfix links:

- FB: reject overly frequent Bugzilla IDs (because no markup was required for them: any integer was considered a potential Bugzilla ID)
- FC: reject overly frequent commit IDs
- SB: reject small Bugzilla IDs
- TT: reject major timetravel (i.e., candidates where an ID was mentioned before it was created, except allowing for inter-server clock differences)
- LU: reject late updates of Bugzilla entries (that appear to have happened unrealistically later after the corresponding bufix commit)

Each of these has a single tuning parameter. Pepper used simple diagnostic x/y plots of filter strength versus tuning parameter to select a parameter value for each that appeared sufficiently strong but not too strong.

A sixth criterion was used to immediately accept a candidate, rather than reject it:

- UD ("not unidirectional"): accept all bi-directional bugfix links (where the commit comment mentions the Bugzilla ID and also the Bugzilla entry mentions the commit ID.)

He validated the individual filters by manually checking 2,500 of the candidate bugfix links that are particularly likely to be wrong and ended up with 4,047 validated bugfix links as the output of the tuned and validated filtering chain. The individual filters exhibited correct-filtering precision between 43% and 99%, but that applied only to those between 0.3% and 11% of the candidates actually filtered out, resulting in individual results precision between 73% and 81% (because the raw candidate links already have 73% precision). The UD criterion has 99% results precision.

For the overall filtering chain, the bugfix link retrieval results precision was 93%, a very good result. The rough estimate of retrieval recall came out at 65%, a sufficiently good result.

[4] Section 7 actually has seven such subsections.

4.2. Case study: Interpretation (R2, R3)

In terms of results step 1 worked out well, but the manual validation of so many candidates made it a major effort, lending some credibility to proposition **R3** ("too laborious"). On the other hand, this is a one-time effort only, so the weight of this observation is modest.

However, designing the filtering chain was research-level work (Prechelt and Pepper, 2014) and without at least the larger part of it, even the bugfix link retrieval precision would not have achieved the precision of 80% required for the overall DICA. Other organizations might have shied away at this point, which lends some credibility to proposition **R2** ("too difficult") as well.

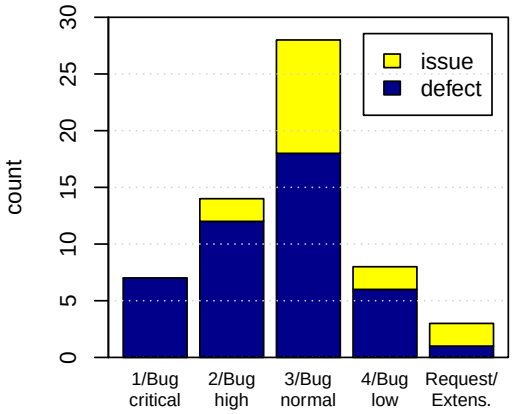

Figure 1: Sample of Bugzilla entries by "severity".

5. Step 2: Discriminating defects from issues

The common software developer term "bug" (refering to something described by a bugtracker entry) is misleading and it would be a grave mistake to assume that what software developers call a "bug" is always a defect. If a DICA was applied to "bugs" that in fact were issues, the results and subsequent response actions would be random: By definition the rework performed due to an issue was unavoidable and so neither people nor process can be held responsible for not avoiding it. We must apply DICA only to proper defects.

Unfortunately, at least in the domain of interactive systems whose development is not contracted out, it is common *not* to have a precise specification of the system's intended behavior. Instead, the role of a specification is filled by some combination of general domain understanding, the behavior of previous versions of the product, general UI principles (consistency etc.), general requests from customers, users, and product managers, and customer-specific requests coming from particularly important customers.

The latter would clearly be non-defect issues and some other behaviors are so obviously unintended that they are clearly defects. But there is a large fraction (perhaps a majority) of change requests that lies in between and requires careful consideration before labeling them as defect or non-defect. In such a setting it is not surprising that the development team as a whole does not possess a uniform concept of "defect". As a result, the bugtracker data may not allow to discriminate defects from issues reliably in order to analyse defects only.

Therefore, the present section investigates how the discrimination of defects and issues would be done for the Infopark data and how well it works. We find that it does not work well enough.

5.1. Case: Steps, issues, results (difficulty A)

Pepper was aware of the "bug" bug and decided he needed to assess its size. Each entry in Infopark's Bugzilla bugtracker contains a field *severity* with the following possible values: "1/Bug: Critical", "2/Bug: High", "3/Bug: Normal", "4/Bug: Low", "Request/Extension". The latter value is intended to identify issues, all others should only be used for defects. "Normal" is the default value if the developer does not choose severity explicitly.

Pepper drew a random sample of 60 bugfix links and inspected their respective Bugzilla entries manually: Given insider knowledge about CMS Fiona's functionality, application domain, and bug reporting processes, which of the entries should be considered a defect description rather than an issue description?

Except for the "critical" category, all severity categories were polluted with incorrect entries as shown in Figure 1. Theoretically, it should have been sufficient to ignore bugfix links in the "Request/Extension" category. Practically, however, also the "low" and "normal" categories contain more issues than acceptable: Their precision is only 75% and 64%, respectively, so that the required 80% precision would be ruined before the actual circumstance analysis has even started. Of course the analysis results could in principle have better precision than the analysis inputs. However, relying on such a healing effect would be haphazard and Infopark was not keen on trying it.

By using only data of bugfix links marked "critical" and "high", the precision of selecting only defects would be increased to 90%. However, the recall then is only 43%.

Neither of these is good enough: First, the precision of 90% will subsequently be reduced by the fact that bugfixes for issues are much larger on average than those for defects. When tracing these fixes back to their "fix-inducing changes", they will point to a lot more different places, which will all be assumed to be defect insertions, which will distort the results of the analysis strongly.

Second, the recall of 43%, combined with the 65% recall of the bugfix link search mean we get to see only 28% of all defect fixes and run the danger of drawing a rather irrepresentative picture: Infopark was aware that many important defects were in fact flagged with the default severity, "normal".

We call the difficulty of properly discriminating defects from issues *difficulty A*. The resulting incomplete recall could be taken as a reason for stopping the whole analysis and Infopark's analysis attempt could have ended at this point. However, Infopark decided to assume this problem could be solved in the long run (e.g. by improving the definition of "bug" used by the developers and increasing their attention to setting a proper severity

value) and to investigate what would then happen in the next step, for which they now used only the fixes for bugtracker entries marked "critical" or "high".

5.2. Case study: Interpretation (R5, R6)

Difficulty A appears to be widespread: Using a refined bug/issue differentiation, Herzig et al. (2013) report that between 38% and 48% of the bugtracker entries were tagged wrongly across five open source projects.

Linguistic techniques such as that reported by Antoniol et al. (2008) (that rely on the text of the bugtracker entry rather than its tag) can potentially help, but even if Pepper had been aware of them, they are not readily applicable: They require a large manually classified training set, have parameters that need tuning, may produce high misclassification in a particular domain and organization, and are not directly applicable for languages where stemming is more complicated than in English (such as German, which is used in most of the Infopark bugtracker entries). What does this tell us about our propositions?

Propositions R5 ("unknown reliability") and R6 ("insufficient reliability") both talk about the quality of the DICA *results*. In contrast, the above problem is concerned with one aspect of the *input* data quality. Had Infopark indeed stopped the analysis at this point, arguing that the low recall of proper defects *might* distort the results too much, it would have supported the claim that R5 is the main force underlying low DICA usage.

But Infopark acted differently: They postponed the decision and tried to understand the DICA data quality issues more fully, so that neither R5 nor R6 gain much credibility in this stage.

6. Step 3: Mapping defect corrections to defect insertions (DCDIM)

Once the defect-related bugfix links have been isolated, the next step is the DCDIM: mapping the defect correction ("bug fix") to the corresponding defect insertion ("bug commit").

The basic approach of DCDIM is simple and consists of four steps:

- S1: Consider which lines have been changed in a bugfix,
- S2: assume all of these lines (but only these lines) to be defective,
- S3: trace backwards through the version history to identify for each of these lines the last commit that has changed the line, and
- S4: treat each such commit to be a defect insertion.

Unfortunately, every single one of these steps has problems and can go wrong:

- P1: The bugfix change may have touched other lines than only the lines of the actual defect correction,
- P2: the defect correction itself may also have changed more lines than just the defective ones, e.g. in order to provide a cleaner fix[5],
- P3: a line change in the history may leave the program behavior unchanged,
- P4: the actual defect insertion may have been earlier and the last change is in fact unrelated to the defect.

Furthermore, the method is inherently unable to pinpoint the bugfix commit for many defects resulting from omission (as opposed to commission), because newly inserted lines have no version history that could be traced (P5).

6.1. Case: Steps, issues, results (naive set, improved set, difficulty B)

The procedure implementing DCDIM is usually called SZZ algorithm, after the authors of the article that introduced it (Śliwerski et al., 2005). Infopark used it with several improvements, including some of the improvements described by the successor article of Kim et al. (2006). Specifically, they ignored changes that modify only whitespace, they ignored changes that modify only comments, they ignored changes that replace complete (external) components with a new version (which typically involves hundreds of changes). Annotation graphs were not applied because Infopark was not interested in a finer granularity than one file. The implementation of the procedure used the MininGit[6] software (a fork of CVSAnalY (Robles et al., 2004)) with the HunkBlame extension.

Some of the above extensions were not yet present in that software and also Pepper identified a number of defects in it, so he extended and debugged the software and submitted those changes back to the MininGit open source project – again a substantial (and largely unexpected) effort.

The next issue was the quality of the DCDIM results: If many defect corrections are non-minimal (P1), the results may be full of false positives, i.e., have very low precision. Furthermore, if there are many defects of omission (P5), the results might be very incomplete, i.e., have low recall.

A near-complete manual validation like for the bugfix links was out of the question, because validating a candidate defect correction/defect insertion pair involves program understanding on a sometimes non-small scale and every time in a different program context, which does not scale well.

Nevertheless Pepper decided that further progress was impossible with at least a small sample of manually validated pairs. He developed the following approach for drawing such a sample: Start from the set of 1,250 bugfix commits that relate to a Bugzilla entry with severity "critical" or "high" (cf. Section 5.1). These comprise 4,814 files changed. The medium

[5] With respect to P1 and P2, Herzig and Zeller (2013) report for five open source projects that between 7% and 20% of the defect correction changes address multiple concerns at once ("tangled changes") rather than just the defect correction concern. The additional concerns can be much larger in terms of the number of lines changed, so P1 and P2 tend to be major issues.

[6] https://github.com/SoftwareIntrospectionLab/MininGit

number of lines changed per file is 3; the medium number of lines changed in a whole commit is 8. From the 4,814 file changes, draw a stratified random sample of 100 file changes as follows:
(a) 25 file changes from files with at most 3 lines changed;
(b) 25 file changes from files with more than 3 lines changed;
(c) 25 file changes from one file from a commit with at most 8 lines changed;
(d) 25 file changes from one file from a commit with more than 8 lines changed.

For these 100 file changes, DCDIM pointed out a correct defect insertion commit only 14 times. 5 times it pointed to the wrong commit, 81 times there was no defect in the modified lines at this point (hunk) at all. This is a depressingly bad result from the standard SZZ-based DCDIM procedure. It means that from a straightforward application of DCDIM we should expect a precision of only 14% – even *after* we have confined ourselves to only the bug reports with severity "critical" and "high" in order to improve precision! We will call the DCDIM results obtained in such manner the *naive set*.

How does this low precision arise? The reasons are found in the elements a), e), and f) of the development process mentioned in Section 3.1:

(1) False positive bugfix links that pertain to issues rather than defects will tend to involve larger changes according to a) and will hence usually result in several false positive DCDIM results, not just one.

(2) A quick preliminary fix and its subsequent clean final fix, which are common according to e), will refer to the same Bugzilla ID and will thus produce two bugfix links. For the later one, SZZ will incorrectly identify the preliminary fix to be the defect introduction; another false positive. Also, the final fix often involves structural modifications and hence touches other parts of the same file or other files, leading to still more (possibly many more) false positives.

(3) Local refactorings and cleanups done while fixing bugs according to f) enlarge their bugfix commits and blur the identification of defect-inserting changes, leading to many false positives.

(4) Refactorings and cleanups done independently from bugfixes according to f) will sometimes modify lines containing a defect and will make SZZ incorrectly report the refactoring commit to be the defect-introducing commit, leading to further false positives.

Of the 14 *correct* answers, however, only one came from subset (b) and one other from subset (d) of the stratified sample; the remaining 12 were all from the small bugfixes in (a) and (c). When restricting the sample to bugfixes that modified at most 3 lines overall, it contains only 18 file changes and DCDIM finds a correct defect insertion commit 9 times (50% precision).

So in order to obtain at least somewhat reliable DCDIM results, Pepper applied the following heuristic: *Apply DCDIM only to bugfixes that modify at most 3 lines overall.* We will call the DCDIM results obtained in such manner the *improved set*.

When restricting the input for computing the naive set to the input for computing the improved set, the input shrinks from 1,250 to 395 commits (32%) and from 4,814 to 441 files changed (9%). As a result of this drastic reduction of the candidate set, a reduction needed to improve the precision to at least medium values, the local recall should not be expected to be higher than those 32%.

At estimated 50% precision and 32% (or less) recall of the DCDIM results, there is little ground for being optimistic about the quality of subsequent DICA results. We call this inability of the SZZ algorithm to perform sufficiently-correct DCDIM *difficulty B*.

Normally, the resulting irrepresentativeness to be expected would likely be taken as another reason for stopping the whole analysis. However, since it did not involve a large amount of additional work, Infopark decided to make the fourth step nevertheless in order to understand the DICA problem space more completely.

6.2. Case study: Interpretation (R2, R3, R5)

Many of our propositions get some exposure in this step.

First, as in step 1, substantial effort is involved: Some for getting the software to work (and work properly), some more for manual validation of a sample of results; several weeks overall. And while the software part is, again as in step 1, a one-time effort, the validation may well need to be repeated for subsequent analysis of newer data, because the commit practices are likely to change as a side-effect of the previous analysis. So there is some support for proposition **R3** ("too laborious"), but at least Infopark did not find the effort prohibitive.

Second, coming up with the "improved set" filtering heuristic was, again as in step 1, a non-trivial creative feat and without it the results are clearly unacceptably bad, lending some credibility to proposition **R2** ("too difficult") as well. However, both of these arguments are vastly outweighed by the third one:

The bleak outlook resulting from the low precision and recall estimates for even the improved set strongly points to proposition **R5** ("unknown reliability"): It is totally unclear whether the later DICA results can be trusted, so why should anybody make such an MSR effort at all?

Note that not much effort is actually needed for a would-be DICA user to come to this conclusion: DCDIM-checking any small sample of bugfix links would suffice, because unless the respective repository has *much* nicer characteristics than Infopark's, the results will be so bad that little optimism will remain no matter how that sample is chosen. This means the effort (R3) and ingenuity (R2) involved in steps 1, 2, and half of 3 are not needed to decide that DICA is most likely not worth pursuing (R5).

Pepper and Infopark, however, were persistent and stubbornly pursued a further evaluation nevertheless.

7. Step 4: Performing the actual DICA

Step 4 is the harvesting step: Formulate a defect insertion circumstance question, aggregate the DCDIM data accordingly to answer it (DICA), then decide on process change steps to trigger as a consequence.

For instance Infopark might ask whether there were weekdays on which there was "much higher" defect insertion proneness than on others. If so, further investigation would try to uncover why this was so and, once known, the process would be changed to remove or control that cause.

7.1. Case: Decision how to proceed

Based on the discouraging DCDIM data quality results from step 3, Infopark decided they must not trust any subsequent DICA result blindly but rather needed some way of validating its quality once more.

After some discussion[7], Infopark decided to proceed as follows:

- They would pose a quantitative model of DICA and the subsequent process change steps. The sole purpose of that model would be to estimate the quality of the DICA results with respect to suggesting valid (rather than arbitrary) target areas for possible process changes. The model will be described in Subsection 7.2.
- They understood that this model would be dubious, there was no practical way of validating it, and it could hence itself not be trusted.
- However, as no better source of information would be available, they decided that, should the model indicate the suggestion quality would be *low* (which is indeed what happened), Infopark would consider the whole DICA effort to be unhelpful and would stop it.
- Should the model indicate suggestion quality might be *high*, decisions as to how to proceed would be made subsequently.

The evaluation of suggestion quality based on the model will be described in Subsections 7.3 to 7.7.

This approach means the initial DICA questions asked need not be valuable or sophisticated – they only serve to decide whether DICA results can be trustworthy at all. The first questions asked will thus be simple ones.

7.2. Case: The Good Highlights DICA quality model

Ideally, a DICA quality model should evaluate the cost/benefit ratio of the process change steps eventually taken. As the universe of such steps is essentially unlimited and their benefits are impossible to estimate, this goal was immediately found too ambitious.

Infopark decided to settle for precision and recall once again: A DICA was of good quality if and only if it would point out ("highlight") most of the right objects of interest (be it weekdays, developers, developer roles, or whatever was currently being analyzed) and almost no wrong ones. Most of the right ones meant at least 50% recall. Almost no wrong ones meant not too much less than 90% precision. Then the subsequent process improvement steps would presumably be worth their while, otherwise they would not be or at least be considered too risky in this regard.

There is still a problem though: In the DCDIM step, we have defined the naive set and the improved set of results and we have derived rough estimates of their quality. However, for evaluating DICA results this is insufficient. Rather than global quality estimates we need a specific, case-by-case ground truth (exact or approximate).

Infopark decided to derive an approximate ground truth by means of an assumption as follows. Define the *true set* to be the correct result the DCDIM *should* have produced. This would be the exact ground truth. Its precision would be 100% and it should be used for valiging the quality of a practical DCDIM. Alas, the true set is not known. Now make the following assumption: *The highlighting validity of the naive set as measured by the improved set is about the same as the highlighting validity of the improved set would be if we could measure it by the true set.* The assumption is certainly not true exactly, but plausibly true approximately. It was the best approach anyone could think of and Infopark was hence willing to accept it.

Based on this assumption, the quality evaluation procedure works as follows: Postulate a cutoff parameter for the allowed defect insertion density and measure precision and recall of DCDIM when applied to highlighting the items that lie above the cutoff. Infopark expected that "50% higher than the median" would usually be a sensible cutoff criterion but was willing to accept a different cutoff if the data suggested it.

To summarize the idea:

- For applying DICA one would normally use the improved set, not the naive set.
- However, there is insufficient trust in the DICA result quality from even the improved set. Infopark needs to build trust first or tear it down completely.
- Therefore, they apply DICA to the naive set and use the improved set as stand-in for the ground truth for evaluating the result's quality, assuming that this measurement will come out at least roughly similar to the result of an actual DICA applied to the improved set.
- If the indicated quality is low, Infopark will assume the real quality is low, too, and will stop the DICA effort.
- If the indicated quality is high, the real quality is still uncertain, so further investigation will be necessary.

7.3. Case: Results for components

In their first analysis, Pepper wanted to identify the subsystems (components) of CMS Fiona that are most prone to defect insertion[8] as measured by the percentage of commits that insert a defect.

[7] Now no longer within Infopark alone but rather including the researcher, Prechelt.

[8] Note this can *not* simply be computed from the number of fixes, primarily because one fix can correspond to several defective commits.

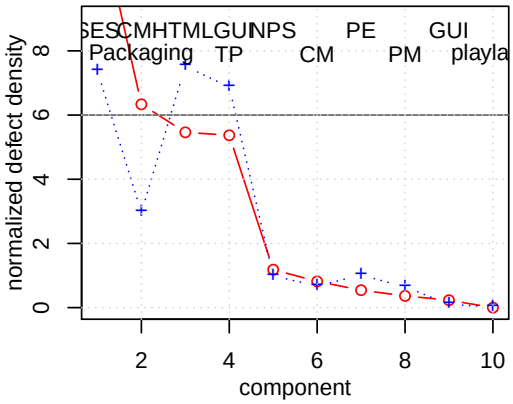

Figure 2: Normalized defect insertion densities for the improved set (circles) and the naive set (plus signs) for each component; ordered decreasingly. 1 represents the median of the respective set. Cutoff criterion: Identify all components whose defect insertion density is higher than 6 medians.

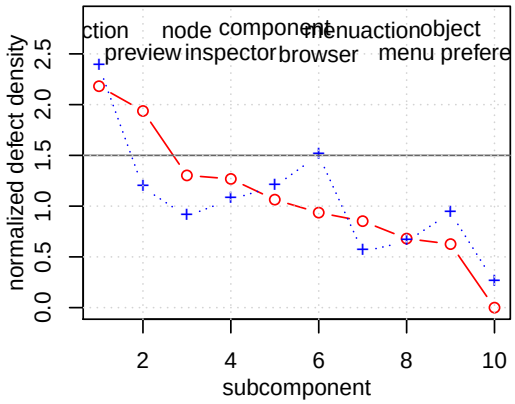

Figure 3: Normalized defect insertion densities for the improved set (circles) and the naive set (plus signs) for a set of subcomponents; ordered decreasingly. The identity and purpose of the subcomponents is irrelevant here. 1 represents the median of the respective set. Cutoff at 1.5 medians.

Figure 2 shows the outcome. Looking at the plus-labeled data (which is what one would see in practice), Pepper decided that the expected cutoff at 1.5 times the median was not sensible. Sensible ones appeared to be 5, 6, or 7. He picked 6. The ideal result (based on the circles) now identified components 1 and 2, whereas the practical result (based on the plusses) identified components 1, 3, and 4. This is a recall of 50% and a precision of 33%; clearly insufficient.

Further analysis provides some insights how this low quality comes to be: Qualitatively, any line segment pointing up rather than down signals a confusion in the rank order (in our case: at 3 and 7). Such confusion can reach farther than to the next entry (in our case: 4 is still wrong compared to 2). Quantitatively, one can summarize the difference between the two curves that threatens precision and recall by a *coefficient of variation (CV)*: Compute the ratio of each circle-value to the corresponding plus-value. For the set of ratios, compute mean m and standard deviation s. Then $CV = s/m$. Ideally, the CV should be zero. Practically, we obtain a CV of 0.66, which means the average(!) error is two thirds as large as the value itself; not a high similarity. CVs of 0.2 or less may not have much impact on precision and recall, but variation beyond 0.3 or so will often cause mistakes.

7.4. Case: Results for subcomponents

Pepper repeated the same analysis for modules (subcomponents) of one area of what appears (according to the plusses!) to be the top-defect-prone component: CMHTMLGUI.

Figure 3 shows the outcome. Now using the standard "50% above the median" cutoff criterion, the ideal result identifies components 1 and 2, the practical result finds 1 and 6 instead; a recall of 50% and a precision of 50%; also clearly insufficient.

Given the many huge estimation errors, the result could have been worse. The CV is 0.50.

7.5. Case: Results for developers

Now Pepper asked, again in the same manner, which developers' commits are most often defect-inserting.

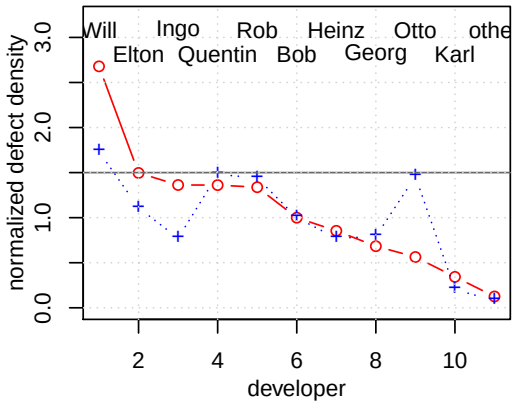

Figure 4: Normalized defect insertion densities for the improved set (circles) and the naive set (plus signs) for those ten developers with the highest values plus the rest combined as "other"; ordered decreasingly. Names are replaced by pseudonyms. 1 represents the median of the respective set. Cutoff at 1.5 medians.

Figure 4 shows the outcome. The standard "50% above the median" cutoff criterion (cutoff 1.5), finds only developer 1 as both the ideal and the practical[9] result (recall 100%, precision 100%); in principle a perfect and hence encouraging outcome. However, this outcome is a lucky strike: At cutoff 1.4 we would have found 1, 4, 5, 9 instead of the correct 1, 2 (recall 50%, precision 33%); at cutoff 1.3 we would have found 1, 4, 5, 9 instead of the correct 1, 2, 3, 4, 5 (recall 60%, precision 75%). This variation stems from the many rank confusions in this pair of lines. Overall, the developer analysis did also not increase Infopark's trust in DICA result quality. The CV is 0.36.

7.6. Case: Results for weekdays

Next is a similar analysis for the density of defect insertions per day of the week. The result is shown in Figure 5, this time in natural order, not sorted order.

Here, the standard cutoff at 1.5 does not point out any day at all as the distribution is rather smooth, so the DICA result is not

[9] The value of developer 4 is 1.496. Its larger-looking position is an artifact of the plotting software.

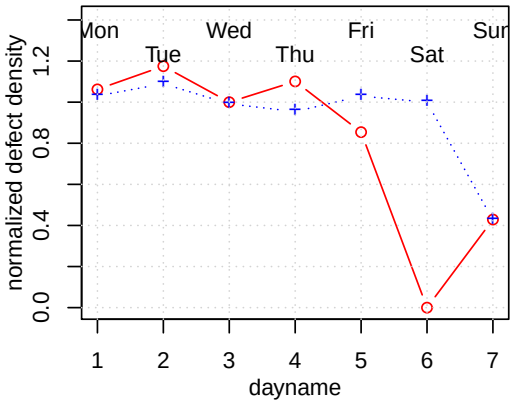

Figure 5: Normalized defect insertion densities for the improved set (circles) and the naive set (plus signs) for the seven days of the week; in natural order. 1 represents the median of the respective set. The cutoff at 1.5 medians is too high to even be visible.

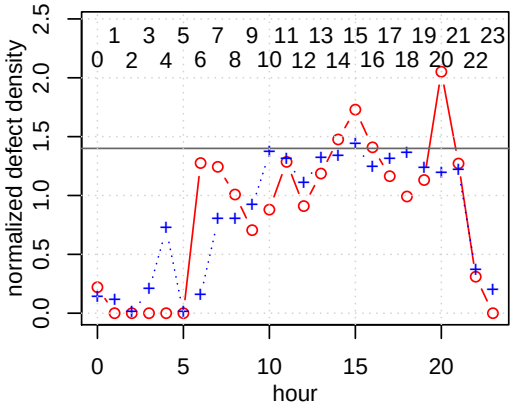

Figure 6: Normalized defect insertion densities for the improved set (circles) and the naive set (plus signs) for the 24 hours of the day; in natural order. 1 represents the median of the respective set. The cutoff is at 1.4 medians.

wrong, but it isn't helpful either. Therefore, Pepper just went for the top two days instead, the practical result finds the highest day correctly (Tuesday), but the second-highest wrongly (Wednesday instead of Thursday, yielding a precision and recall of 50%); a semi-satisfactory result at best. The CV is 0.46.

7.7. Case: Results for times of day

Finally[10], Pepper also repeated the previous analysis for the different hours of the day as shown in Figure 6.

The default cutoff at 1.5 would again flag out nothing, but just barely so. We therefore lower it to 1.4. Now the practical result suggests hour 15 to be dangerous, which is correct, but misses the also correct hours 14, 16, and the most important one of all, 20 (recall 25%, precision 100%). At cutoff 1.3 we would have found 10, 13, 14, 15, 17, 18 instead of the correct 14, 15, 16, 20 (recall 75%, precision 33%). Both of these results are insufficient. The *CV* is rather high at 1.45.

[10] Additional analyses of interest would have investigated developer roles and developer/component relationships. As the previous results were not encouraging and these new ones would have involved additional manual data preparation work, Infopark dropped them.

7.8. Case study: Interpretation (R2, R5, R6, difficulty C)

The invention of the Good Highlights model may seem to suggest support for **R2** ("too difficult"), as it required a good understanding of evaluation methodology. However, we must consider that this whole validation business should not have been needed in the first place: Had all gone well, a high quality of the hand-validated DCDIM results would have indicated (already in step 3!) that the DICA results should be trusted. So the invention of the Good Highlights model does in fact support **R5** ("unknown reliability") even more than R2.

The actual DICA results point in a similar direction: For none of the five subanalyses the results were convincingly good; quite on the contrary – most of them were clearly bad. This gives support to **R6** ("insufficient reliability"): Current DICA techniques do not appear to be up to the task, at least for the Infopark repository[11]. Had the results been better, this would not have helped much either, because then Infopark would not have known whether they should trust the Good Highlights model; the results could just as well have been invalid and hence misleading. We call this inability to validate the quality of the DICA results *difficulty C*. It provides additional support for **R5** ("unknown reliability").

8. Threats to validity

8.1. External Validity

The results of our analysis are strongly dependent on the development process underlying the repository. The problem from mixing defects and issues (difficulty A, step 2) will be smaller if the organization applies a suitable definition of defect more consistently. It could also be larger, though, if a category for feature requests is not used at all in the bugtracker.

We think that Infopark is reasonably typical of most of the software world in this respect, with the possible exception of large software organizations where the bugtracker is maintained by dedicated staff, in which case the problem should be smaller.

The low quality of the DCDIM (difficulty B, step 3) will be smaller if bugfixes are smaller and bigger if they are larger.

This depends a lot on the nature of the software product, its development stage, and on development style. We think that at least for long-running web application developments the Infopark repository is again reasonably typical. There appears to be a trend towards better isolation of bugfixes and smaller commits in general in organizations as they start using Git as their versioning systems, because Git provides strong support for taking apart a mixed set of changes that have been performed to a set of files into multiple separate commits. This should help making the problem smaller over time, but will take a long time to ripple first through the software organizations and then through the repositories within those organizations.

Beyond these considerations, the case for R5 and R6 as the dominating reasons for not using DICA is even stronger than it

[11] Repositories more amenable to successful DICA are likely to exist, but we do not expect them to be common.

appears above, for the following reason: The bugfix link determination (step 1) created 7% false positives. In normal practice, the 7% false positives would make it harder to obtain valid DICA results. In the Infopark case, due to the extensive manual validation performed, the identity of most of the false positives is known. Infopark of course eliminated those from the dataset to make the subsequent analysis more reliable, which even strengthens the relevance of the negative overall outcome.

8.2. *Construct Validity*

Our research question concerns decison-making about using or not using DICA. The study directly reports on the actual decision-making as it has occured at Infopark. No proxy measurements are needed and hence threats to construct validity do not exist.

8.3. *Internal Validity*

The accuracy and completeness of our report relies mostly on the computerized record of the MSR activities and results themselves and only at a few points on Pepper's memory and manual work notes. These few points were all salient and important to him at the time so a relevant distortion appears unlikely. The appropriateness of our conclusions given our report can fully be judged by the reader.

As for the validity of the *difficulty A* finding: The sample used for checking the defect/issue discrimination (Section 5.1) was rather small, so that the results are coarse-grained and embody substantial sampling error. However, this sampling error is too small to change the difficulty A conclusion; in particular, the 32% recall figure has a 98-percent confidence interval ranging from 19% to only 46%.

9. Related Work

Successful DICA requires overcoming difficulties **A** (discrimination of defects from issues in the bug tracker in step 2), **B** (reliable DCDIM in step 3), and **C** (establishing the trustworthiness of the DCDIM in step 4). Reading the DCDIM/DICA literature, it quickly becomes clear that A receives not much attention[12] and C receives almost none at all– which also threatens to invalidate any intended contributions to B.

We will therefore review the literature and report weaker aspects such as the definition of "bug" that is used or the efforts made to validate DCDIM results. We mark up the discussion with letters A, B, C to indicate which of the difficulties it pertains to.

B: The original authors of the SZZ algorithm, Śliwerski et al. (2005), were well aware that not all of the changes the algorithm points out are defect insertions; they therefore called the outputs "fix-inducing changes" instead. Unfortunately, this term is still misleading: If the output of the algorithm is incorrect (not a rare case as we have seen), the particular changes found did not "induce" the bugfix. **A**: If the bugfix addresses an issue (rather than a defect), which by definition involves *unavoidable* rework, it is inappropriate even a-priori to say that *any* change has induced it. The SZZ article does not address this problem.

A: Kim et al. (2007) (which is about defect prediction, not DICA) features the term "fault" in the title, uses "bug" and "fault" interchangeably, but defines neither of them explicitly. Later on, the article says that faults are considered to be the result of cognitive breakdowns, which means the term is intended to mean defect only, not issue. Nevertheless, no filtering is performed to eliminate issues from the data. **B**: The original article Śliwerski et al. (2005) performs DICA using a well-devised array of criteria to accumulate evidence that a certain change *might* be fix-inducing – and then it just hopes for the best; no manual validation of the output is performed at all and no discussion of threats to validity is given.

A: The successor article Kim et al. (2006) also provides no definition of defect. **B**: It introduces a number of improvements and uses two example repsitories (from the Columba and Eclipse projects) for assessing their impact. The authors boldly assume that any difference observed in the algorithm's output when introducing an improvement will only reflect *improved* precision (i.e. reducing the number of false positives) and *improved* recall (i.e. reducing the number of false negatives) but will never worsen them, so they do not manually validate these outputs. They introduce the improvements one by one and add a manual assessment as the final step. However, the manual assessment only addresses whether the fix really addresses a defect, but not whether the algorithm output is indeed a corresponding defect insertion. They report about 95% precision of the defect correction data. The "threats to validity" section does not mention the correctness of identified bug-introducing changes.

A: Eyolfson et al. (2011) do not provide a definition of defect either but is more explicit about it: The article uses the term "bug-introducing commit" and says in Section 2 "Despite our terminology, a bug-introducing commit is not necessarily bad code; it is possible that the later fix is adaptive or perfective, updating the code to work with changes in third-party code, or reflecting a change in requirements". (The same wording remains in Eyolfson et al. (2013).) **B**: It also uses SZZ for DICA, but performs manual verification only to make sure fixes are really fixes, not to verify defect insertions[13]. The "threats to validity" section mentions the importance of validating bug-introductions, but does not address it.

A: Williams and Spacco (2008) use the pair of terms "fix-inducing change" and "bug-fixing commit", but do not provide definitions or discuss the defect/issue discrimination explicitly. **B**: The article applies SZZ and *does* perform a proper manual validation of the identified defect insertions for a sample of 25 defect corrections from the Eclipse project. It finds that the 25 corrections consisted of only 50 lines overall, that 43 of those actually were defect corrections (86%) and that 33 of the cor-

[12] There is more attention to A in the defect prediction literature recently but little in the DICA literature.

[13] There is a statement "A brief manual inspection of bug-introducing commits did not reveal any anomalies." (and the same wording remains in Eyolfson et al. (2013).) but no elaboration whatsoever regarding that inspection itself.

responding fix-inducing lines were defect insertions. We read this to mean that precision was approximately 66% (the article is not explicit about this). This is a much better DCDIM precision than that found by Infopark. The difference rests on three pillars: An absent (and hence flawless) step 1, an absent (and hence flawless) step 2, and a project with mostly very small bugfixes. Despite these advantages, precision is insufficient by Infopark's standards and without steps 1 and 2 there is not even a procedure for achieving whatever level of recall.

All of the DICA articles stop at pure DCDIM or shortly thereafter; none of them discusses the validity of conclusions from a DICA as in step 4.

10. Conclusions

We will first summarize the findings from the four steps of our DICA case in order to answer the research question (Section 10.1). We then reframe the contents of the case as a "normal" MSR application study (Section 10.2), comment on the state of affairs in previous MSR research (Section 10.3), and summarize what future work appears to be needed (Section 10.4).

10.1. Regarding the Research Question

- Step 1 of establishing the bugfix links (Section 4) produced good results, yet involved a lot of work, so it might be perceived to support proposition **R3** ("too laborious"). However, most of that work is a one-time effort. It is fully reusable not just within Infopark but even beyond, so it does not provide a strong argument and overall step 1 does not supply much insight into our research question at all.

- Step 2 of discriminating defects from issues (Section 5) highlighted difficulty A: Defects and issues are very often not held apart sufficiently well in the bug tracker, so that DCDIM results will be polluted with mappings from non-defect changes and DICA results may hence become misleading. This lends a lot of credibility to **R6** ("insufficient reliability").

- Step 3 of mapping defect corrections to defect insertions (DCDIM, Section 6) highlighted difficulty B: Even when avoiding difficulty A, the mapping is far too unreliable to make correct DICA conclusions likely; again a strong argument in favor of proposition **R6**. One part of the reason is non-minimal defect corrections, so that organizations that strictly keep their corrections minimal might obtain better results[14], but even when avoiding this issue also (by using only very small corrections: the improved set), the precision was only 50% due to additional changes that often happen between defect insertion and defect correction and that mislead the SZZ algorithm. This performance is clearly too low and gives still more weight to **R6** ("insufficient reliability").

- Step 4 of performing the actual DICA (Section 7) adds an additional dimension: Even if the DCDIM results would have been better, Infopark would not have known whether they should trust the DICA results as there is no practical way of validating them (as opposed to the small-sample approach to validating DCDIM results). This is a fundamental problem that would disappear only with near-perfect DCDIM results and introduces **R5** ("unknown reliability") as the second major and valid reason why DICA is rarely used.

Overall, the low reliability (**R6**) achieved for the DICA results, reinforced by the lack of a practical assessment method for that reliability (**R5**), are clearly the dominant reasons why Infopark dropped its DICA initiative and we see little reason why many other organizations should come to a different conclusion.

10.2. Regarding the (Quality of) DICA Results

If (and only if!) one is willing to take the Good Highlights model (Section 7.2) seriously, one can reinterpret the improved-set results of step 4 as the presumably valid results of a DICA and their comparison to the naive-set results as a DICA validity assessment. These results can then be summarized as follows:

- In our simulated decision-making for responsive action after a defect insertion circumstance analysis (Section 7), the quality requirements formulated by Infopark ($\geq$ 50% recall, $\geq$ 80% precision, Section 3.2) could not be achieved. The results suggest Infopark may have been successful for identifying problematic *developers* – but only by sheer luck (Section 7.5). They were unsuccessful for identifying problematic *components* (33% precision, Section 7.3), *subcomponents* (50% precision, Section 7.4), *weekdays* (50% precision, Section 7.6), and *times of day* (either only 25% recall or only 33% precision, Section 7.7).

- For instance, had Infopark taken the naive analysis results seriously, it would have missed the valuable insight that work should rather end *before* 8 pm (Figure 6) and would instead have searched for sources of problems in developer Otto's work, although those are just analysis artifacts and in fact Otto's commits are just fine (Figure 4).

- On the side, the analysis appears to corroborate the (equally uncertain) result of Eyolfson et al. (2011) that there is no generally problematic day of the week; rather, this is likely project-specific (Section 7.6).

- Also on the side (but interestingly), the analysis contradicts another result of Eyolfson et al. (2011), which had found late-night commits to be the most defect-prone in

[14] Note that the high quality of the Infopark data with respect to bugfix links (Prechelt and Pepper, 2014) suggests that the non-minimal corrections are *not* simply a sign of low development discipline. Rather, we conjecture the effect emerges from application domain and business context (e.g. in the form of time pressure when fixing critical defects for the high-end enterprise customers) and organizational context (e.g. because co-located development is easier to coordinate than open-source development so that larger commits create problems only rarely).

Linux and PostgresQL data. In contrast, Infopark developers appear particularly careful in those few cases when they work so late at all and as a result the nightly commits have a *lower* probability of having defects than daytime commits (Section 7.7).

Before you quote these results, please make sure you are aware of the leap of faith involved in adopting the Good Highlights model on which they are based.

One main outcome of our work is the insight that we have no practical procedure for assessing the correctness of the results of realistic DICA uses: Defect prediction research has reasonable-quality ground truth available simply by waiting for the defects to surface. Verification of DICA results, in contrast, would require the (manual) validation of all DCDIM results, which, as discussed in Section 6, is impractical.

10.3. Regarding the Related Work

The MSR community has been very inventive to find ways in which repository data could be used. Much less work, at least of the DCDIM type, has yet carefully assessed whether practical application is realistic in terms of the quality of available input data or whether the analysis results will be sufficiently reliable even then.

In particular, our discussion in Section 9 showed that MSR works in the DCDIM realm often

- do not introduce (let alone operationalize) a suitable notion of defect,
- perform rather little validation of the DCDIM results found, and
- hardly discuss the difficulties involved in validating DICA (or even only DCDIM) results.

10.4. Regarding Further Work

Three obstacles stand in the way of reliable DICA results:
(1) Imperfect bug tracker entry type metadata (which leads to confusion of defects with issues),
(2) non-minimal defect corrections, and
(3) additional changes between defect insertion time and defect correction time that happen to happen at subsequently defect-corrected locations. The latter two both lead to low DCDIM reliability.

Further research should investigate how processes and tools need to be designed in order to minimize (1) and (2). Minimizing (2) requires a lot of discipline on the process level, but the hunk-picking functionality of "git add" interactive mode in the Git versioning system is a good start for the tool support. A better DCDIM technique than the simple line-change tracking performed by the SZZ method will be needed to overcome (3). The (yet immature) dependency-based analysis approach proposed by Sinha et al. (2010) is probably a good start here.

Acknowledgment

We thank Thomas Witt and the development team at Infopark for their help towards understanding the repository data. We thank Stephan Salinger for helpful comments on a draft of the article.

References

Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., Guéhéneuc, Y.G., 2008. Is it a bug or an enhancement?: A text-based approach to classify change requests, in: Proc. 2008 Conf. of the Center for Advanced Studies on Collaborative Research: meeting of minds (CASCON '08), ACM. Article 23.

Asaduzzaman, M., Bullock, M.C., Roy, C.K., Schneider, K.A., 2012. Bug introducing changes: A case study with android, in: Proc. 9th Working Conf. on Mining Software Repositories, IEEE. pp. 116–119.

Bhattacharya, P., Neamtiu, I., 2011. Bug-fix time prediction models: Can we do better?, in: 8th Working Conf. on Mining Software Repositories, ACM.

Canfora, G., Cerulo, L., Cimitile, M., Di Penta, M., 2011. Social interactions around cross-system bug fixings: The case of FreeBSD and OpenBSD, in: 8th Working Conf. on Mining Software Repositories, ACM.

CMMI Product Team, 2010. CMMI for Development, Version 1.3. Technical Report CMU/SEI-2010-TR-033. Software Engineering Institute, Carnegie Mellon University. Pittsburgh, PA.

Cook, J.E., Votta, L.G., Wolf, A.L., 1998. Cost-effective analysis of in-place software processes. IEEE Trans. on Software Engineering 24, 650–663. doi:http://doi.ieeecomputersociety.org/10.1109/32.707700.

Eyolfson, J., Tan, L., Lam, P., 2011. Do time of day and developer experience affect commit bugginess?, in: Proceeding of the 8th Working Conference on Mining Software Repositories, ACM, New York, NY, USA. pp. S. 153–162. URL: http://doi.acm.org/10.1145/1985441.1985464, doi:http://doi.acm.org/10.1145/1985441.1985464.

Eyolfson, J., Tan, L., Lam, P., 2013. Correlations between bugginess and time-based commit characteristics. Empirical Software Engineering 19, online–first. doi:10.1007/s10664-013-9245-0.

Giger, E., Pinzger, M., Gall, H.C., 2011. Comparing fine-grained source code changes and code churn for bug prediction, in: Proceeding of the 8th working conference on Mining software repositories, ACM, New York, NY, USA. pp. 83–92. URL: http://doi.acm.org/10.1145/1985441.1985456, doi:http://doi.acm.org/10.1145/1985441.1985456.

Graves, T.L., Karr, A.F., Marron, J., Siy, H., 2000. Predicting fault incidence using software change history. IEEE Transactions on Software Engineering 26, 653–661. doi:http://doi.ieeecomputersociety.org/10.1109/32.859533.

Graves, T.L., Mockus, A., 1998. Inferring change effort from configuration management databases, in: Proc. 5th Int'l. Symposium on Software Metrics (METRICS), IEEE CS Press. pp. 267–273.

Haley, T.J., 1996. Software process improvement at Raytheon. IEEE Software 13, 33–41. URL: http://doi.ieeecomputersociety.org/10.1109/52.542292.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2012. A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering 38, 1276–1304. doi:10.1109/TSE.2011.103.

Herzig, K., Just, S., Zeller, A., 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction, in: Proc. 2013 Int'l. Conf. on Software Engineering (ICSE '13), IEEE Press. pp. 392–401.

Herzig, K., Zeller, A., 2013. The impact of tangled code changes, in: Proc. 10th Working Conf. on Mining Software Repositories, IEEE. pp. 121–130.

Kim, S., Zimmermann, T., Pan, K., Whitehead, Jr., E., 2006. Automatic identification of bug-introducing changes, in: Proceedings of the 21st IEEE International Conference on Automated Software Engineering, IEEE Computer Society, Washington, DC, USA. pp. S. 81–90. URL: http://portal.acm.org/citation.cfm?id=1169218.1169308, doi:10.1109/ASE.2006.23.

Kim, S., Zimmermann, T., Whitehead, Jr., E., Zeller, A., 2007. Predicting faults from cached history, in: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA. pp. S. 489–498. URL: http://dx.doi.org/10.1109/ICSE.2007.66, doi:10.1109/ICSE.2007.66.

Prechelt, L., Pepper, A., 2014. Reliable bugfix links via bidirectional references and tuned heuristics. Automated Software Engineering (submitted for review) URL: ftp://ftp.mi.fu-berlin.de/pub/reports/tr-b-14-01.pdf.

Rao, S., Kak, A., 2011. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models, in: 8th Working Conf. on Mining Software Repositories, ACM.

Robles, G., Koch, S., González-Barahona, J.M., 2004. Remote analysis and measurement of libre software systems by means of the CVSAnalY tool, in: Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems, IEEE Computer Society, Washington, DC, USA. pp. 51–55. doi:10.1.1.58.6959.

Rotella, P., Chulani, S., 2011. Implementing quality metrics and goals at the corporate level, in: 8th Working Conf. on Mining Software Repositories, ACM.

Sadowski, C., Lewis, C., Lin, Z., Zhu, X., Whitehead, Jr., E., 2011. An empirical analysis of the FixCache algorithm, in: Proceeding of the 8th Working Conference on Mining Software Repositories, ACM, New York, NY, USA. pp. 219–222.

Shull, F., Basili, V., Boehm, B., Brown, A.W., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., Zelkowitz, M., 2002. What we have learned about fighting defects, in: Proc. of the 8th IEEE Symposium on Software Metrics, pp. 249–258. doi:10.1109/METRIC.2002.1011343.

Sinha, V.S., Sinha, S., Rao, S., 2010. Buginnings: Identifying the origins of a bug, in: Proc. of the 3rd India Software Engineering Conf. (ISEC '10), ACM press. pp. 3–12.

Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? ACM SIGSOFT Software Engineering Notes 30. doi:http://doi.acm.org/10.1145/1082983.1083147.

Wang, X.O., Baik, E., , Devanbu, P., 2011. Operating system compatibility analysis of Eclipse and Netbeans based on bug data, in: 8th Working Conf. on Mining Software Repositories, ACM.

Williams, C., Spacco, J., 2008. SZZ revisited: Verifying when changes induce fixes, in: Int'l. Workshop on Defects in Large Software Systems, ACM press. pp. 32–36.

Yin, R.K., 2003. Case Study Research: Design and Methods. Sage. URL: http://www.amazon.de/Case-Study-Research-Design-Methods/dp/0761925538/.

Zaman, S., Adams, B., Hassan, A.E., 2011. Security versus performance bugs: A case study on Firefox, in: 8th Working Conf. on Mining Software Repositories, ACM.

Zeltyn, S., Tarr, P., Cantor, M., Delmonico, R., Kannegala, S., Keren, M., Kumar, A.P., Wasserkrug, S., 2011. Improving efficiency in software maintenance, in: 8th Working Conf. on Mining Software Repositories, ACM.