ELSEVIER

The Journal of Systems and Software 61 (2002) 173–187

The Journal of Systems and Software

www.elsevier.com/locate/jss

Classification and evaluation of defects in a project retrospective

Marek Leszak[a,*], Dewayne E. Perry[b,1], Dieter Stoll[a,2]

[a] Optical Networking Group, Lucent Technologies, Thurn-und-Taxis-Str. 10, 90411 Nürnberg, Germany
[b] Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712, USA

Received 1 March 2001; received in revised form 1 July 2001; accepted 1 September 2001

Abstract

There are three interdependent factors that drive our development processes: interval, quality and cost. As market pressures continue to demand new features ever more rapidly, the challenge is to meet those demands while increasing, or at least not sacrificing, quality. One advantage of defect prevention as an upstream quality improvement practice is the beneficial effect it can have on interval: higher quality early in the process results in fewer defects to be found and repaired in the later parts of the process, thus causing an indirect interval reduction.

We report a retrospective analysis of the defect modification requests (MRs) discovered while building, testing, and deploying a release of a network element as part of an optical transmission network. The study consists of three investigations: a root-cause defect analysis (RCA) study, a process metric study, and a code complexity investigation. Differing in the quantities that we anticipate to be related to found defects, they all have in common the goal of identifying early quality indicators.

The core of this threefold study is the root-cause analysis. We present the experimental design of this case study in some detail and its integration into the development process. We discuss the novel approach we have taken to defect and root cause classification and the mechanisms we have used for randomly selecting the MRs, to analyze and collecting the analyses via a web interface. We present the results of our analyses of the MRs and describe the defects and root causes that we found, and delineate the countermeasures created either to prevent those defects and their root causes or to detect them at the earliest possible point in the development process. We conclude the report on the root-cause analysis with lessons learned from the case study and from our experiences during subsequent usage of this analysis methodology for in-process measurement.

Beyond the root-cause analysis, we first present our findings on the correlation between defects detected and the adherence to our development process. Second, we report on our experience with analyzing static code properties and their relation to observed defect numbers and defect densities.

Keywords: Root-cause analysis; Defect prevention; Software metrics; Project retrospective; Process improvement; Change management

1. Introduction

1.1. Project context

The product in our study is a network element (NE) that is a flexibly configurable transmission system in an optical network, consisting of circuit packs, ASICs, software units, and a craft terminal. The total head count for this release was 180 people and the development project lasted for 19 months.

The NE software is developed in teams of 5–10 people. A typical (large) NE configuration can consist of many different hardware board types and up to 150 different software components. A software team is responsible for a collection of functionally related components, which altogether form an architectural unit, called 'software domain' within this paper. The overall size of the NE software product is around 900 K-NCSL (non-commentary source lines), 51% being newly developed software.

This release has been a very important and critical one especially for the European market. Management concern for process improvement enabled several

* Corresponding author. Tel.: +49-911-526-3382.
E-mail addresses: mleszak@lucent.com (M. Leszak), perry@ece.utexas.edu, URL: www.ece.utexas.edu/~perry (D.E. Perry), dieterstoll@lucent.com (D. Stoll).
[1] Tel.: +1-512-471-2050.
[2] Tel.: +49-911-526-2624.

0164-1212/02/$ - see front matter
PII: S0164-1212(01)00146-7

project retrospective activities, one of them being the root-cause defect analysis (RCA) project, two others have been a process compliance and a code complexity study. Several improvement projects towards, for example, better effort estimation, more efficient development, and predictable and higher quality (measured in number of defect MRs) have been performed afterwards.

1.2. Retrospective analysis

The studies we are presenting have all been done retrospectively, but were based on different data sets from the same project context. The input data we used were defect numbers, static code analysis data, and defect classification data from a sample from all defects. The last set of data constitutes the basis for the central part of this paper, the RCA, which is an extended version of Leszak et al. (2000). The other two data sets were input to smaller investigations, one on process compliance, the other on code complexity and its relation to defect insertion.

For the RCA a team has been constituted as cross-functional team: members represented the NE software and hardware domains, as well as the independent integration and certification department and quality support group. We also have been supported by members of the Bell Laboratories Software Productions Research Department who brought extensive experience from other similar studies (e.g., Perry and Stieg, 1993) into our team. The mission of the RCA project was to

- analyze sample defect MRs; find systematic root causes of defects;
- analyze major customer-reported MRs during the maintenance release (so-called Post-GA MRs, GA = general availability of the product);
- propose improvement actions, as input for current development projects, in order to reduce number of critical defects (severity 1 and 2 MRs) and to reduce rework cost, e.g., MR fix effort.

The other two investigations on process compliance and on static code analysis measures have been performed by quality assurance representatives in cooperation with the software support team. Data were readily available from quality gate reviews and from measurement tools evaluating static code characteristics like code size and complexity.

1.3. Limitations

The root-cause study has focused on defect analysis and determining the underlying root causes of those defects. There are more general perspectives that one might take (for example, what went well and what went wrong) but those have been out of scope for this study. Moreover, correlations with other product metrics have not been considered either, though it would be very interesting to analyze, for example, defect distributions in relation to test effort. We have focused our effort analysis on the reproduction, investigation and repair of the defect MRs, not on the retest effort that makes up a significant part of the rework effort.

Due to time pressures resulting from limitations on team member availability for RCA, we were not able to implement the formal analysis training and testing to establish a defined level of inter-rater reliability i.e., different analysts often came up with different RCA results for the same MRs. However, there are two mitigating factors. First, the analysts looking at the MRs were members of the team putting together and reviewing the web-based analysis tool. This participation resulted in project relevant aspects being included in the questionnaire. We argue that this participation also resulted in a shared understanding of the components of the analysis. Second, we did implement informal consistency checks during the analysis process. Where inconsistencies were found, the analysts made subsequent corrections to the defect analyses. Third, after all data have been analyzed a comparison of the team specific results took place which revealed that the differences between teams where consistent with the subjective expectations of the analysts. Thus, we are relatively confident in the consistent ratings of the resulting data.

The investigation of the relation between defects and process non-compliance should be regarded as an accidental finding like there are uncountably many in the history of science. As such it needs further study and investigation before it could be used as a standard tool for early defect indication, work that still remains to be done.

1.4. Relation to other work

Prior work on software faults has generally been reported on initial developments and focused on the software faults themselves rather than their underlying causes. The work of Endress (1975), one of the earliest papers to analyze software faults, based his error classification on the primary activities of designing and implementing algorithms in an operating system. Thayer et al. (1978) provided an extensive categorization of faults based on several large projects. Schneidewind and Hoffmann (1979) categorized faults according to their occurrence in the development life-cycle. Ostrand and Weyuker (1984) introduced a novel attributed categorization scheme delineating fault category, type, presence and use. Finally, Basili and Perricone (1984) provided an analysis of a medium scale system.

Our current study is based in part on earlier work by Perry and Evangelist (1985, 1987) on interface faults which, while cognizant of the earlier fault categorization

work listed above, derived its list of interface faults from the fault data rather than using a pre-existing categorization. It is based also on the work by Perry and Stieg (1993) – a study of one of the releases of one of Lucent's very large switching systems. The fault categorization used in this study was based in part on the published categorizations and part on the experience of the developers in the reported project.

The work here is similar in intent but differs in implementation details. First, the defect categories are an improvement on the original categories in that they are separated into three classes for better human factors reasons, breaking up a large set of defects into three reasonable sized sets of defects. Second, the effort estimation scales are uniform here where they were different there; further, we added investigation effort here. Third, we expanded the root causes over what we had in the original study. This expansion was done in conjunction with the knowledgeable developers from the project and reflects both the current state of their project and their processes. Fourth, and our most novel aspect from a research point of view, we allowed for multiple root causes to be defined as well as for no root causes (i.e., a simple mistake with no underlying, lurking cause). And finally, rather than surveying the entire set of defect MRs, we have randomly selected a statistically significant sample from each of the domains for detailed analysis.

Card's "Learning from out Mistakes with Defect Causal Analysis" (Card, 1998) provides a generic process which is congruent with the process followed here.

There are several strands of related work that are similar to ours but which did not have a direct influence on our approach. Chillarege et al. in their paper "Orthogonal Defect Classification" (Chillarege et al., 1992) focuses on defect types and defect triggers as a means of feedback to the development process. In spirit, this is much the same as our approach except that their method is used throughout the entire development process for immediate feedback where ours is essentially a retrospective and 'end of development' feedback process. Also the subsequent elaboration of their approach in Orthogonal Defect Classification at IBM,[3] appears to be too detailed and complex to be applied to the kind of software application we studied.

More recent work on defect and root-cause analysis by Yu et al. (1997, 1998) has followed a process similar to ours, but using different defect and root-cause classification schemes. They have not focused explicitly on the effort related to the defects. However, the general shape of their results are similar to ours.

In a larger context, our methodology applies key aspects of the defect prevention process area of the CMM (Paulk et al., 1993) and is similar to the one described in Endress (1975).

For the process compliance part of our work, several case studies on error-prone modules e.g., by Khoshgoftaar and Allen (1999) are related to our work. However, their classification does not relate defects per module to the degree to which a defined process has been followed.

The definition of a process consistency metric in Krishnan and Keller (1999) is based on following certain CMM key process areas and relating those to field defects. Our study bases process compliance on more detailed, quality related activities like unit test and inspections and relates the execution or omission of those activities to defects found in-process.

1.5. Organization of the paper

We first discuss the root-cause analysis study in Section 2 which constitutes the core and the most multifaceted part of the paper. We introduce our RCA methodology in Section 2.1. We then present our data analysis in Section 2.2 focusing first on how we prepared the data, then on the results of our general analyses of the defects, effort and root causes, and finally on the selection of critical root causes to be either prevented or found earlier. In Section 2.3, countermeasures and follow-up improvement actions are discussed. Section 2.4 summarizes lessons learned from the retrospective analysis, as well as from the subsequent implementation as in-process measurement (ipRCA).

In the following two sections we report on two smaller sized investigations.

Section 3 presents the stunningly clear relation between defects found and process non-compliance during development by introducing a process metric derived from our development process and comparing defect numbers and densities found for SW components falling into different process non-compliance categories.

Section 4 then gives a brief summary of our unsatisfying experiences with static code analysis and its relation to observed defects and defect densities.

Finally in Section 5 we summarize the paper and give an outlook into future activities.

2. Root-cause defect analysis (RCA)

We elaborate first on our methodology for RCA.

2.1. RCA methodology

2.1.1. Embedding of RCA into the development process

Fig. 1 depicts the process workflow in which RCA plays a key role for analyzing defects and learning from systematic root causes: from the project's MR

[3] http://www.research.ibm.com/softeng/ODC/ODC.HTM

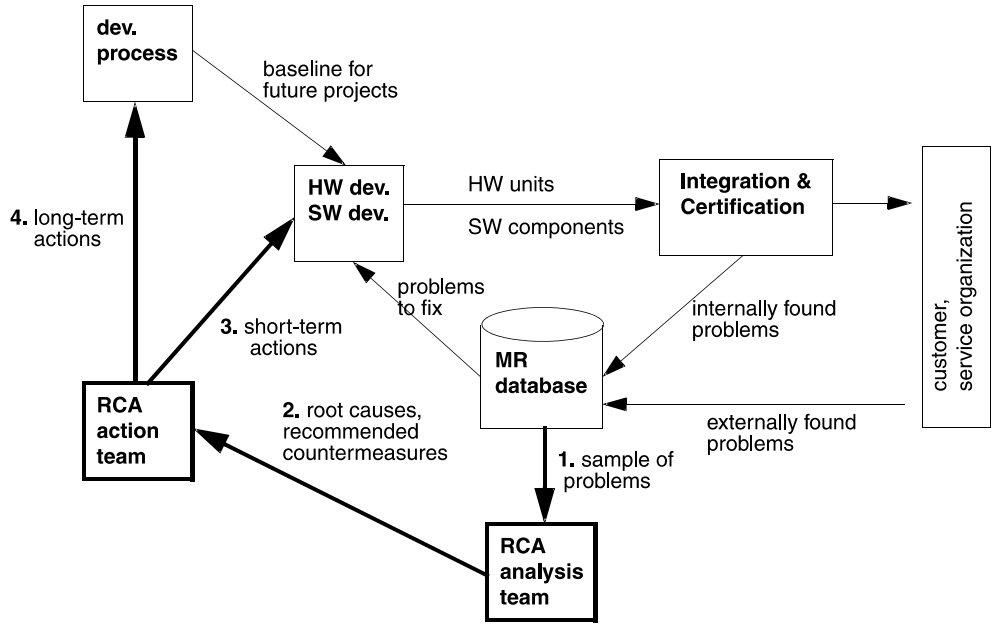

Fig. 1. Overall development workflow, enhanced by RCA procedure.

databases, samples of both internally detected and customer delivered defects are selected (1) and enhanced by RCA information by the RCA team. Upon analysis of the systematic root causes, countermeasures are defined (2) and proposed as either short-term or long-term improvement actions. Upon management alignment and approval, the former will impact the development project directly, whereas the latter will lead to improvements of the organizational development process.

2.1.2. MR classification scheme

Each MR in the selected sample has been analyzed along the categories 'phase detected', 'defect type', 'real defect location', 'defect trigger', and 'barrier analysis info', described in detail below.

Phase detected information. An important fact needed if we are to find defects earlier is when the defect was in fact found. The analyst may choose any one of 10 process phases (see Table 1) as the phase in which the defect is found. In addition, the analyst may indicate why the defect was not found earlier.

Defect types. We divided the defects into three classes of defects: implementation, interface and external. Within each of these classes there are a set of appropriate defects types, depicted in Table 2.

Each type can then be further refined by indicating its defect nature: *incorrect*, *incomplete*, *other*. These were to be applied wherever they were appropriate. Their use was to reduce the number of explicit defect types. The other part of the defect classification focused on reclassifying the severity if that was necessary and reporting the amount of effort to reproduce the defect, to investigate it, and to repair it. We used a uniform scale for these effort reports: zero (meaning negligible effort), less than one day, one to five days (i.e., up to a week),

Table 1
Lifecycle phases for network element development (associated RCA analysis results are in Fig. 3)

Lifecycle phase	Included process steps
1. System definition	Specification of requirements and of systems architecture
2. HW design	Design of HW entities e.g., circuit packs and ASICs
3. IMS	'Integration of minimal system', i.e., integration of all HW pieces for a network element
4. SW design	Software architecture definition and design, coding and unit testing of SW components
5. SW integration	Integration of SW components into a complete SW image and grey-box testing of the interworking of SW components, on host-level without the target hardware
6. System integration	Integration of all HW and SW entities for a network element, as well as subsequent functional test of their interworking
7. System test	Black-box acceptance testing of requirements on network element level
8. Reproducibility	Phase after successful system test, at delivery of the final HW configuration and SW load, but prior to 'General Availability'
9. Deliveries	Phase after General Availability i.e., operation at customer networks
10. Prior to project start	Phases related to previous product release, prior to 'System definition' of current release

Table 2
Classification of defect types (associated RCA analysis results are in Fig. 2)

Implementation	Interface	External
1. Data design/usage	9. Data design/usage	16. Development environment
2. Resource allocation/usage	10. Functionality design/usage	17. Test environment (tools/infrastructure)
3. Exception handling	11. Communication protocol	18. Test environment (test cases/suites)
4. Algorithm	12. Process coordination	19. Concurrent work (other releases)
5. Functionality	13. Unexpected interactions	20. Previous (inherited from previous release)
6. Performance	14. Change coordination	21. Other
7. Language pitfalls	15. Other	
8. Other		

five to 20 days (one to four weeks, i.e., up to a month), and more than 20 days (more than a month).

Real defect location. The real defect location specified either a document identifier, or whether it was software or hardware. 'Real' location characterizes the fact that in real projects some defects are not fixed by correcting the 'real' error-causing component, but rather by a so-called 'work-around' somewhere else.

Defect triggers. Our approach to defect triggers (root causes) is rather novel. There are a number of dimensions that may in fact be at the root of each of the defects – that is, there may be several underlying causes rather than just one. We therefore provided a set of four *inherently non-orthogonal* classes of root causes: phase-related, human-related, project-related and review-related.

- *The phase triggers* are the standard development phases or documents: requirements, architecture, high level design, component spec/design, component implementation and load building. The phase related root causes could be qualified by the nature of the trigger: incorrect, incomplete, ambiguous, changed/revised/evolved, not aligned with customer needs, and not applicable (the default).
  Associated analysis results are in Fig. 4.
- *The human related triggers* are: change coordination, lack of domain knowledge, lack of system knowledge, lack of tools knowledge, lack of process knowledge, individual mistake, introduced with other repair, communications problem, missing awareness of need for documentation, and the default, not applicable. The "individual mistake" trigger is similar to the "Execution/oversight" category of Yu (1998). It reflects the fact that sometimes you just make mistakes.
  Associated analysis results are in Fig. 5.
- *The project triggers* are time pressure, management mistake, caused by other product, and not applicable (again the default).
- *The review triggers* are no or incomplete review, not enough preparation, inadequate participation, and not applicable. (Note that a review is a formal moderator-controlled inspection of a document or code artifact. Our review process is described in Laitenberger et al. (1999).)
- *other triggers* and, of course, there is the escape "other" allowing a different trigger to be specified.

Barrier analysis information. Finally, the analyst may suggest measures for ensuring earlier defect detection and/or for preventing or avoiding the defect altogether.

2.1.3. MR selection procedure

As is usual in these kinds of studies, there is a problem with the magnitude of the amount of work that would have to be done to analyze all the relevant MRs to get a complete picture of the defects and their causes. One way of reducing the amount of work is to randomly select a significant subset of the MRs to represent the whole set and carefully analyze that subset. Thus our selection procedure was as follows:

- Define a set of MRs per domain (not per team), such that a subteam analyses selected MRs per domain.
- For each set of MRs per domain:
  - filter out inappropriate MRs;
  - split into n MR subsets, such that each MR in a certain subset "S_i" belong to exactly one domain;
  - from the MR subset S_i , select further a (typically much smaller) subset S_i' such that
    - one part (say 5–10) is selected manually by the subteam analyzing S_i', based on own selection criteria like "this MR hurt us a lot", long lifetime, overly complex problem solution, etc.
    - The second part is a random sample of S_i of order 40 MRs. In case S_i is not 'significantly larger' than 40, all MRs in S_i are selected.
    - Explicitly not excluded are severity 3 or 4 MRs (being not customer-visible), no-change MRs (false positives), and documentation MRs.

2.1.4. Methodology to derive countermeasures and improvement actions

Our methodology entails four steps.

(1) *Selection of most significant MR subset*. Critical for proposing countermeasures is the necessity of focussing on a reasonable subset of all defects.

To arrive at such a set we apply a filtering mechanism. The filter cannot be defined beforehand, but is the outcome of a first analysis step. In this first step, the goal is to identify selection criteria which filter those MRs that have together a significant part of rework effort and which to a large extent are found late in the development process. This way we arrive at a first subset of MRs that will be analyzed in more detail.

As second criterion for finding important defects, we look into Post-GA MRs and search there for dominant contributions. If the defects that are found to be important differ in their characteristics from the first set, we get a second set of MRs for detailed study. These results provide the statistical input to the team for the selection of countermeasures that were suggested for each MR during its analysis.

(2) *Prioritization of countermeasures*. The RCA action team brainstormed proposals and weighted each proposal with overall consensus, according to three factors: *statistical weight as percentage of total effort*, *effectiveness of the suggested countermeasure* and *estimated cost of its implementation*, on a scale of 0–1. The product of the values is taken to get a first ranking of countermeasures. Finally this ranking is taken as basic, but not fully binding, input to select the countermeasures to tackle. Typically

- countermeasures with weight > 0.5 should be selected
- the number of countermeasures should be 'small' e.g., < 20, to remain manageable w.r.t. organizational changes

(3) *Definition of improvement actions*. We conducted a two-day workshop with the analysts in which we focussed on the selected subset of defects and root causes to determine the appropriate actions to the defined and prioritized countermeasures. The results are summarized in Section 2.3.

4. *Deployment of improvement actions*. Results were presented to our R&D Management Leadership Team and the development teams. Key improvements proposed have been approved and their implementation initiated, see details in Section 2.3.

2.2. Data measurements and analysis

We first discuss the issue of preparing the data for analysis, then present our general analysis results and conclude with a discussion of how we selected the critical root causes as a focus for more detailed analysis and group discussion.

Before continuing, we briefly clarify the terminology we use in our MR handling process. The following types of MRs are distinguished:

- *initialization MR:* used to add an artifact for the first time to the configuration management repository. Once an MR of this kind is closed, other MRs, of type enhancement or defect, may be created on the associated artifact;
- *enhancement MR:* used to add new functionality as part of a new release, i.e., as planned evolution of an existing system;
- *defect MR:* used to correct any fault in specification, design, or implementation. For each problem detected, a new MR is issued. If several artifacts are affected by the correction, this is handled by MR spawns. In this paper, we consider always the problem-related MRs, not their spawned sub-MRs.

2.2.1. Data preparation

Data screening for the analysis proceeded in two steps.

1. *Consistency checking*. A comparison of results between different domains was conducted. The rationale being the elimination of misunderstandings of terms and verification of results. In particular the analysts have been asked to provide reasons for a typical behavior or to correct the classification if it was due to a misinterpretation of terms.
2. *Preparation of a representative sample*. We separated from those settled analysis data the MRs which have not been selected randomly. From the rest of the MRs we split off so-called no-analysis MRs which were erroneously classified as defects but in fact were initial MRs or enhancement MRs. The remaining MRs constitute a basis of 427 MRs belonging to 13 domains (Post-GA MRs counted as separate domain).

Each domain was represented with at least 8% of its MRs. Typically 20–40% of the MRs were analyzed, these percentages being a consequence of the chosen minimal sample size of 40 MRs, see Section 2.1.3. Taking also the total number of MRs per domain we had a multiplicity factor depending on the domain that each MR was weighted with during the analysis. E.g. each MR belonging to a sample that was represented with 20% of its MRs was counted as 5 MRs with all the characteristics the particular sample MR had like severity, defect type, etc. These weighted MRs were used throughout the following analysis, to extrapolate from the random sample to the total set of MRs.

Extra MRs that have been analyzed in addition to the random sample MRs were negligible in number except for one domain. The MRs of this domain have been included in the comparison between domains as separate group of MRs. The remaining extra MRs have not been

considered in the statistical analysis but only in the manual evaluation of countermeasures.

2.2.2. General analysis results

Our statistical analysis is mainly descriptive in nature. Thus the bulk of evaluation consists in graphical or tabular aggregation of the results of our investigations and is done using the "S" software tool (Becker et al., 1988) in an exploratory way. Most results, depicted in Figs. 2–6, are presented using Pareto charts.

From the distribution shown in Fig. 2, we can derive several general results:

1. External defects are negligible, except for type "inherited from previous release".
2. Interface defects consume about 25% of effort, the largest amount being caused by unexpected interactions, followed by functionality and data design.
3. Implementation defects consume 75% of all effort and are dominated by defects of type algorithm and of type functionality. These defects will be studied in more detail below.

Interesting is the mismatch between number and effort which is most significant for defects of type previous, unexpected interaction, performance, and data design. This is reflected in the estimated effort (in person days) per MR of those defects. On the average, we find 4.6 days for external, 6.2 for interface, 4.7 for implementation defects. Outliers are data design (at the low end) with 1.9 days and (at the high end) inherited defects 32.8, unexpected interactions 11.1 and performance defects 9.3.

As depicted in Fig. 3, system integration represents 50% of the distribution for defect detection, while the elimination of those defects taking up almost 60% of effort. As expected, the data show a significant variation in effort per MR. Defects found in SW design and SW integration require less effort than those found in system

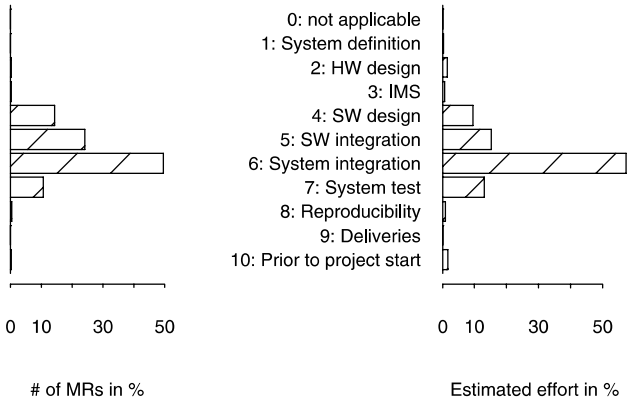

Fig. 3. Distribution of phase where defect detected.

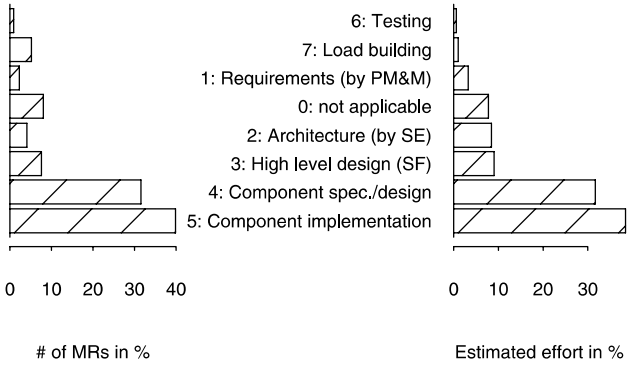

Fig. 4. Distribution of phase where defect originated.

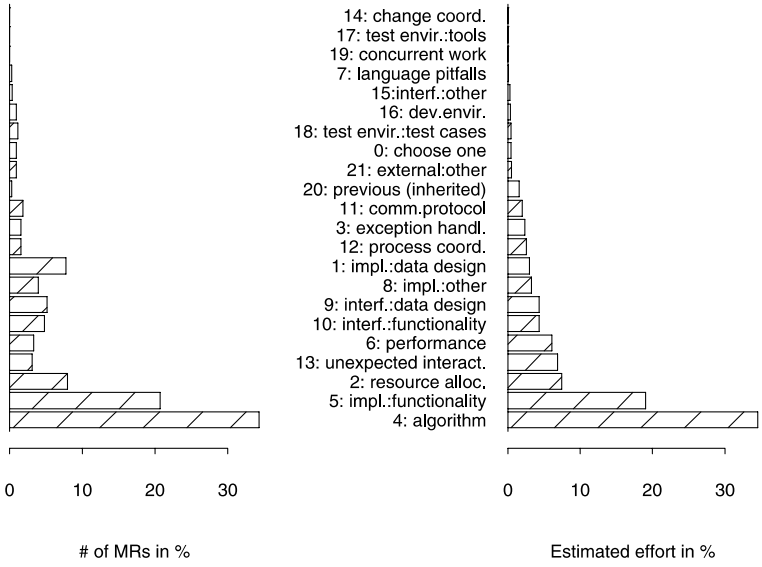

Fig. 2. Distribution of defect types.

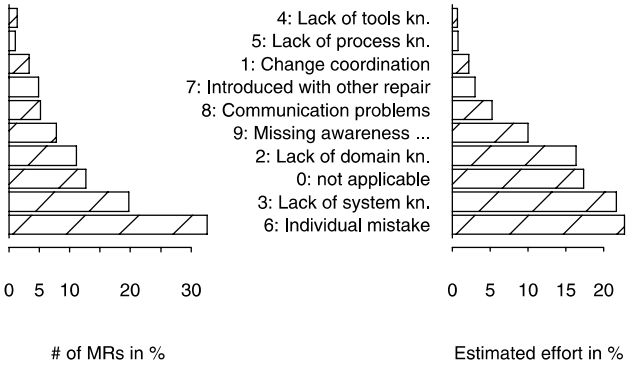

Fig. 5. Distribution of human root causes.

integration and test. The estimated effort per MR (in person-days) as extracted from those figures are 3 days during SW design and integration, 6 days during system integration and system test and 9 days after delivery.

As shown in Fig. 4, defects are injected into the system predominantly (71%) within the component oriented phases of component specification, design and implementation. As an interesting outcome of the analysis, we observe that defects from the requirements phase do not consume on the average tremendously more fix effort to be eliminated than others. Rather architectural mistakes turn out to require much more effort. It turns out that the required effort is for defects originating from requirements 6.5 days, from architecture 10 days, from high level design 5.8 days and from component spec./design 5 days.

An important study decision was to allow for several root causes to be specified during analysis of each MR. The intuition is that there may well be several factors contributing to the occurrence of a defect. Thus, in addition to phase, we have allowed human, project, and review root causes to be specified. These non-orthogonal classifications give indications as to what played a role in a defects occurrence. A useful way of looking at the data is to take the inverse percentages as an indication how many defects remain unaffected if the particular root cause were eliminated.

Viewing the data this way, we can see from Fig. 5 that eliminating individual mistakes would have no effect on 67% of all defects, eliminating lack of system and domain knowledge would have no effect on 69%. If all communication-related problems were to be solved, 87% of all defects remained unaffected.

For the selection of project root causes, while time pressure was chosen to be one affecting factor in 40% of all defects, mostly project root causes were not considered relevant.

Review-related root causes have been considered in 73% of all MRs and inadequate reviews have been specified as important in 48% of all MRs. Thus in 66% of all defects where review root causes have been considered at all, review deficiencies have been diagnosed.

2.2.3. Selection of critical root-causes, to be improved or eliminated

As first step in figuring out dominant contributions, the distributions of MRs according to their defect type were studied with the result that defects of type algorithm and of type functionality (defect class 'implementation') dominated by far all other defect types. 'Functionality defect' refers to missing or wrong functionality (w.r.t. requirements) in a design or code artifact whereas 'algorithm defect' refers to an inadequate (effi-

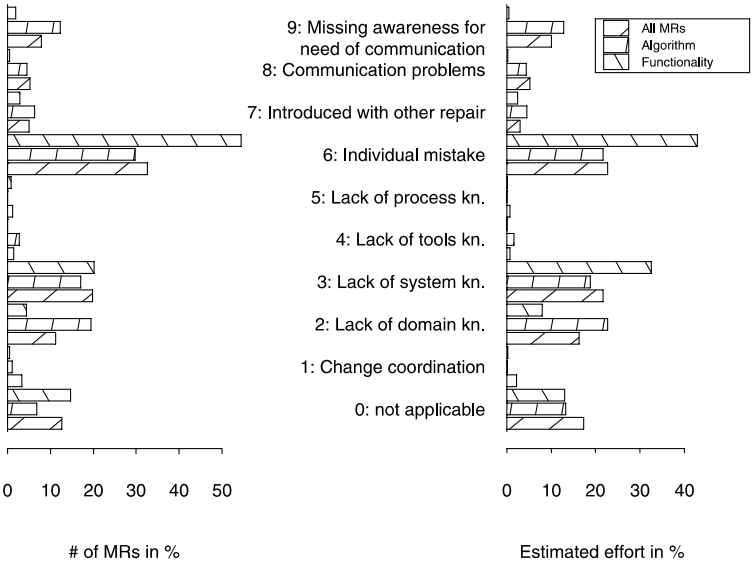

Fig. 6. Human root causes found for defects of type algorithm and functionality, related to weighted mean number of MRs. [Three bars are displayed for each root cause. They show the respective results for all analyzed MRs and for the subsets of MRs classified as "algorithm" or "functionality" defects. The human root cause description starts off at the center-bar of the group of three bars.]

ciency) or wrong (correctness) algorithmic realization. In terms of numbers those defects represent 34% and 21%, respectively, of the defect population and 35% and 19% of the fix effort. The remaining defects are distributed over 16 other defect types.

Of particular interest are the Post-GA defects because they are typically detected by a customer. Of all Post-GA MRs 14% are classified as type algorithm and 68% of type functionality. This re-enforces our interest in those two defect types as deserving further detailed studies.

Specific analysis of MRs with defect type 'algorithm' or 'functionality'. MRs of defect class "implementation" and defect type "algorithm" or "functionality" have been correlated with the phase when they have been detected:

Defect type	SW design (%)	SW integration (%)	System integration (%)	System test (%)
All MRs	14	24	49	11
Algorithm	3	14	71	12
Functionality	13	23	47	14
All other MRs	23	32	33	9

The table shows that for all defect types, more than 60% of the defects are detected late in the process, namely, in system integration and system test. Since the average is 60% this also shows that the remaining 40% of all MRs is typically detected earlier. In particular, algorithm defects exceed the average finding in system integration by almost 50% and the finding of other MRs by 100%. Besides the pure numbers, late detection is a second strong argument for looking into the root causes of these particular defect types. To this end the correlations with various root causes were investigated. The correlation with the phase when the defect was introduced

Defect type	Architecture (%)	High level design (%)	Component specification & design (%)	Component implementation (%)
All MRs	4	8	31	40
Algorithm	0	0	48	46
Functionality	4	15	25	44

shows no unexpected behavior. As one would assume, algorithm defects are introduced during design, specification and implementation, deviating significantly from the average distribution of defect detection. Functionality defects occur in rates close to the average behavior.

The correlation with human root causes (Fig. 6) shows significant contribution from lack of domain and system knowledge, and of individual mistakes. Not unexpectedly the contribution from lack of domain knowledge is smaller in case of functionality defects.

With respect to the correlation with review root causes, we observe that of all MRs of defect type algorithm 75% are afflicted with inadequate reviews with a total of 84% reporting review root causes. For functionality the amounts are 30% inadequate reviews with a total of 67% reporting review root-causes.

Contrasted with an average of 48% inadequate reviews on the basis of a reporting rate of 73%, we may infer that in particular algorithm defects escape earlier detection due to review deficiencies.

From all project root causes that have been available for selection, only time pressure constitutes a major part. On the average 40% of all defects are related to time pressure whereas this amount is 70% for defects of type algorithm and 17% for type functionality.

The analysis thus far indicates that important areas to look for improvements are reviews, domain and system knowledge, and test strategies (e.g., defined vs. achieved test coverage) because of late detection of defects. Means of prevention and earlier detection – provided by the MR analysts – are further evaluated manually to arrive at concrete countermeasures.

Specific analysis of Post-GA MRs. Although we intended to get a subset of MRs from a detailed analysis of Post-GA MRs, the outcome may be summarized very briefly. In fact, we observe that defects of type algorithm and functionality again dominate by far all other defects. Thus having arrived at this subset already nothing must be added to cover defect causes that become visible to the customer. With respect to finding countermeasures this sample adds, however, the question why so many defects have been classified as "introduced by another repair".

2.3. Countermeasures and improvement actions

Our strategy was to find and deploy an effective set of improvement actions, so that for future development projects

- the overall number of defect MRs is significantly reduced;
- the defects are detected earlier in the lifecycle;
- the mean effort to fix a defect is reduced;
- the actions are really effective, i.e., focusing on systematic errors which result in a small number of process changes, promising at the same time to affect multiple defect root causes.

In the countermeasure definition meeting, the team decided to select 10 focus areas on the basis of the statistical data evaluation. The areas chosen from the effort distribution over the phase defects have been introduced, are component specification and design, component implementation and architecture. Note that this selection covers all algorithm defects and 73%

functionality defects which were found to dominate all other defect types. Review root causes were selected as one single category. From the human root cause data the team selected 'individual mistake', 'lack of system knowledge' combined with 'lack of domain knowledge' as one area, and selected 'introduced with other repair' and 'communication problems' together with 'missing awareness for need of communication' as another area. The reason for selecting categories with small contributions was the insight that only human and review root causes can be addressed by countermeasures directly. Finally the team added the categories 'subcontracted software components' and 'project management', due to specific proposals for means of prevention found in the analysis data. Within these categories a set of countermeasures was distilled from the suggestions provided by the analysts during the analysis. The same procedure was followed to arrive at a set of measures for earlier detection of defects found in phases system integration, system test, and maintenance. All countermeasures thus assembled have been weighted to arrive at a ranking. As ranking criteria the team decided to use three inputs:

- savings potential per RC area, represented by portion of total bugfix effort (this the maximum effort that can be saved by avoiding defects of this particular category),
- effectiveness per countermeasure, i.e., estimated percentage of MRs of this RC area which can maximally be influenced by a countermeasure,
- cost per countermeasure, i.e., estimated additional cost to implement the countermeasure. ('Additional' costs refer e.g., to enhanced tools and/or processes that need to be newly introduced.) In order to combine the cost measured in 1000 US$ with the other ranking criteria we mapped cost ranges onto factors in the interval $[0, 1]$ as depicted in Table 3.

The following main countermeasures (CM) and associated improvement projects (IPs)/activities have been defined, with increasing potential benefit in the ordered list below. All proposed IPs have been started and are ongoing. Note also that countermeasures are defined even in areas where SW development is already comparatively mature. Since our organization satisfies CMM level 3 criteria in several key process areas, we understand the improvement activities as one of the means that allow us to reach level 3 fully. Potential savings, effectiveness and cost (S/E/C) are shown in brackets.

Table 3
Mapping of cost ranges per countermeasure into weighting factors

Cost range in K-US$	Factor
0–6	1.0
6–30	0.8
30–130	0.6
130–600	0.4
> 600	0.2

CM1: component specification and design documentation

- ensure required contents, especially include compliance to non-functional and performance requirements (32%/20%/0.8);
- IP: improve requirements management and systems engineering process w.r.t. traceability process and capturing non-functional SW requirements;
- IP: introduce performance engineering (i.e., performance modelling, budgeting, and measurements).

CM2: component implementation

- increase usage of static and dynamic code analysis tools (coding standards checking, memory leak detection, code coverage analysis) (40%/15%/0.8);
- better unit tests (higher test coverage, complete test specification, systematic case selection, better host test environment, test bed, test automation) (40%/35%/0.4).

IP: code analysis tools and unit test tools usage as standard procedure in development process, fully integrated with load build environment. (Note: other product improvements on implementation level, e.g., cleanroom software engineering, sophisticated coding standard based on pre-/post-conditions, etc., have not been tackled: this would mean a major paradigm shift – considered too risky for ongoing development of releases within the same product line in a highly competitive market.)

CM3: system and domain knowledge

- extend training offers and attendance on architecture and application domain, improve systems design skills (38%/35%/0.6).

IP: enhanced training program, assigned training coordinator.

CM4: document & code reviews

- analyze review culture and performance, improve review process (66%/30%/0.8).

IP: include total effort for reviews in realistic planning, ensure sufficient review participation, increase awareness for review importance by e.g., better training, establish review process control: strict entry conditions, scheduling, timing. An review improvement project has been started in cooperation with Fraunhofer Institute for Experimental Software Engineering (IESE). First analysis results appeared in Laitenberger et al. (1999).

CM5: project management

- increase process compliance, i.e., completeness of exit conditions of systems and software development process (100%/30%/0.5).

IP: study correlation of component measurements (size, defects, complexity) and process compliance (see significant results in Section 3).

IP: Implement database system for all project-related data, supporting project tracking and reporting early warnings on process issues.

2.4. Lessons learned

For the insight we gained, we describe first insights related to RCA study after GA. We focus then on insights based on RCA study conducted in-process, before GA.

2.4.1. Retrospective RCA

(1) Bugfix costs *do not grow* exponentially by phase, but rather linearly. (Note, however, that we do not consider re-testing effort which would have added a significant amount to total rework costs.)

(2) The majority of defects *do not originate* in early phases.

(3) Within the same project, the defect attribute distribution per SW domain revealed large differences. (To our knowledge this has not been reported in other studies.)

The number of defects per domain found in system test ranged from 0% to 55%, the respective range is 5–95% for defects found in system test and system integration together. Although intriguing, we learned that these numbers cannot simply be attributed to differences in the quality of SW artifacts. To a significant extent they are due to architecture caused differences of domains. Thus, some have been targets of requirement changes, others could reuse existing functionality, others have higher operation profiles due to belonging to a lower architectural layer, etc. Although interesting, the data available did not permit a detailed comparison along these lines which therefore is left for future studies. In particular, it would be interesting to disentangle the architectural aspect from the team cultural one, e.g., how unit testing or reviews are done, because it would permit identification and promotion of best practices. An interesting side-result of the comparison is the fact that Post-GA defects are to a much larger extent (30%) than on average (5%) caused by another repair which may be traced back to a project's 'end-game' pressure.

(4) There is a significant influence of human factors on defect injection. Our study extended similar ones with regard to human factors for defect infection, and it made them more explicit. We recognized that this was in fact a particularly important attribute. It allowed us to separate randomly inserted defects due to unavoidable (human) mistakes and systematically introduced defects due to mismatches in required and available technical and/or soft skills. In software engineering work, the "human factor" should receive higher focus.

(5) RCA has a low and tolerable effort, relative to its apparent benefits. Two technical insights that we think are worthwhile mentioning, as well. In spite of starting the activity several months after project completion, and that the defects to be analyzed were on the average about a year old, the mean time for analysis was just 19 min. Thus such activities are even cheaper if they are performed during the project when the detailed knowledge about defects can be recalled easily. In-process RCA is a cost effective mean to identify deficiencies and improvement areas. When combined with statistical analysis, which of course is only possible in rather large development projects, conclusions about countermeasure selection can be made sound and put on a solid basis with regard to costs and potential benefits.

2.4.2. In-process RCA

In-process RCA has been carried out for a successor release of the one we undertook the RCA study. The goals of this project were to

- allow for early feedback, within the same project, for systematic root-causes of defects;
- improve the quality focus, especially to create a culture for defect prevention, of the software organization.

This project was a rather medium-size feature enhancement release. Due to the size of the release and the very high quality focus of the project team, only 250 defect MRs have been produced before GA (customer delivery).

We experimented with an enhanced and much more detailed version of our RCA scheme, by incorporating details from IBM's ODC model.[3] All project team members have been trained to classify each defect MR after resolution.

We failed to have significant analysis results, due to several reasons:

- we fully relied on correct data input from our engineers, but around 30% of the entries have been inconsistent. Probable cause was the RCA scheme that was realized to be overly complex and was not fully mastered by casual users who had to provide the required data. (This was despite the fact that a database web application was implemented to capture all RCA information. This application did not check for inter-attribute dependencies and constraints, though.) The respective data records could not be rescued without extensive effort of the providers.

- the surprisingly low number of defects classified and the very detailed scheme led to too small sample sizes per attribute value (significant portion of the MRs were classified into groups of < 5 items).
- although we followed a principle of a mature measurement procedure – provide immediate feedback from data collected to all stakeholders of the data – the project team felt that the complex scheme caused a high amount of measurement reports to be studied, and had lost track of it due to project pressure.

We have learned several lessons from this experience and would like to re-introduce ipRCA into our next product release in the following way:

- the MR attribution for RCA will be done by members of a small dedicated RCA analysis team, as in the retrospective analysis described in this paper;
- the analysis will be performed at major intermediate deliveries, i.e., after each delivery to system test (Card, 1998)
- random sampling will be applied to select MRs to be analyzed;
- analysis result will undergo some peer-to-peer validation, to maximize inter-rater reliability.

3. Process compliance study

We introduce a novel process metric which we have measured and correlated with two other essential metrics on the SW product. (The complete study is captured in Stoll et al., 1999.) Process compliance is defined as the degree to which a documented process is followed in a development project. A process metric clusters SW components in three groups, characterizing the degree of process compliance during component development of the studied project: red, yellow, green. In our case study, defect density differs significantly between the groups.

This simple, efficient, and early applicable discriminator between SW components leads to higher final product quality, if used as milestone exit criterion in the component development process.

Relating various component-oriented metrics, i.e., process compliance and defect density leads to constructive recommendations with regard to

- identifying critical, error-prone components for intensified code review and unit test
- indicating the risk of 'reusing' a component from a previous project, if the amount of changes can lead to decayed design and thus to excessive defect density.

Typically for an enhancement release, more than 80% of all software defects are introduced during component development. Therefore, analyzing product quality and steps towards preventing defects should focus on the software component development process. To improve product quality, these recommendations are currently being discussed with the successor project of the one under study.

Our concept for defining an adequate process metric, to be related with defects, is based on

- typical quality-related activities demanded by a mature component-based development process, e.g., design and code reviews, unit testing, delivery to next stage only after fixing of major defects accomplished;
- for each software component, process compliance is represented by assessing these activities as red, yellow, or green, depending on the degree to which those have been done or not. E.g. the mapping onto "red" means that minimal quality requirements have not been met by the respective component.

The metrics definition is depicted in Fig. 7. The data source for this metric is based on evaluations during so-called *milestone reviews* for all the software components, recorded by a quality engineer at the end of component development. Some components that were planned to be reused without changes or with only marginal changes were not under quality assessment, and thus a fourth metric value "no classification" was added because the metric could not be applied. The meaning of the classification criteria for each component, listed from top to bottom of Fig. 7, is as follows:

- test coverage – refers to the completeness of unit testing done, based on defined unit test specification;
- completeness of test specification and test execution – here both the underlying testcases and the test results (testcases executed and passed) is assessed by the internal customer of the next development phase and by the independent quality manager;
- completeness of code reviews – here the amount of reviews accomplished on new/changed parts of the component is measured;
- completeness of design reviews – here the amount of reviews accomplished is measured, again for the new and changed parts;
- amount of unresolved problems – here the open defect MRs are claimed to have an impact on the delivered quality of the component. The impact depends especially on the severity of the MRs, i.e., the higher the severity, the higher the impact on the component's operation is observable.

The final classification result for an SW component is obtained by following the decision trees from left to right. The overall result is red if at least one decision is red, it is green, if all decisions are green otherwise it is yellow. Twenty two components have been classified "green", 21 "yellow", 27 "red". There were in addition 66 unclassified components.

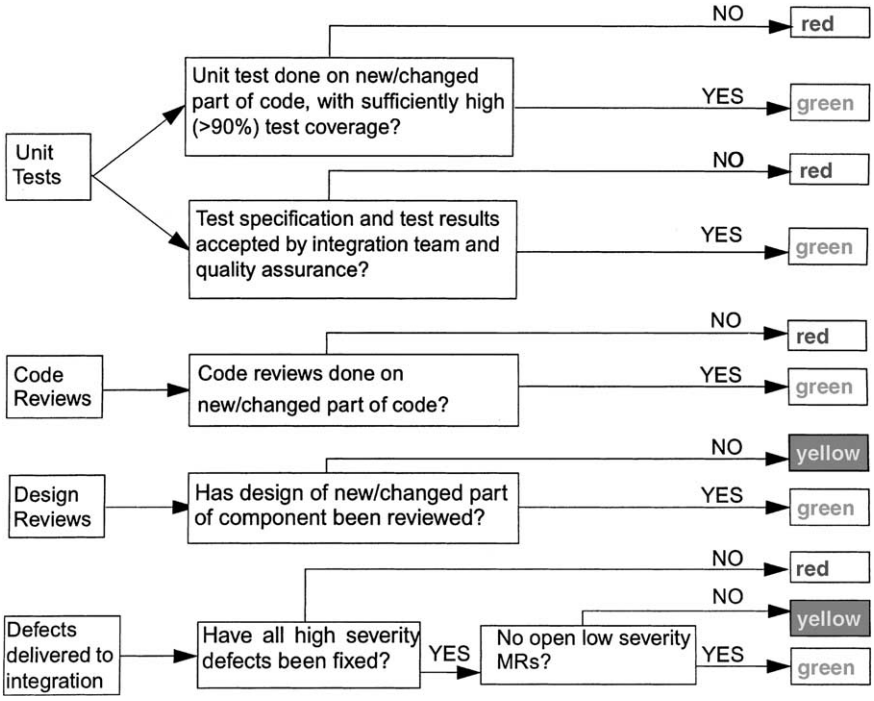

Fig. 7. Process metric definition, on level of software component, measured prior to SW delivery to system integration.

Fig. 8 shows the percentage of components having less than N defects or less than x defects per kNCSL, respectively (all severities). It is clearly demonstrated that the group with worst behavior (40% of components with more than 50 defects and more than 7 defects per kNCSL) is the one with "red" classified components. The best behavior is observed for unclassified components. This justifies a posteriori that the components were not put under quality control (thus no additional effort was wasted). Interestingly, "yellow" and "green" group do not differ significantly. Identical conclusions apply, if only high severity MRs are considered.

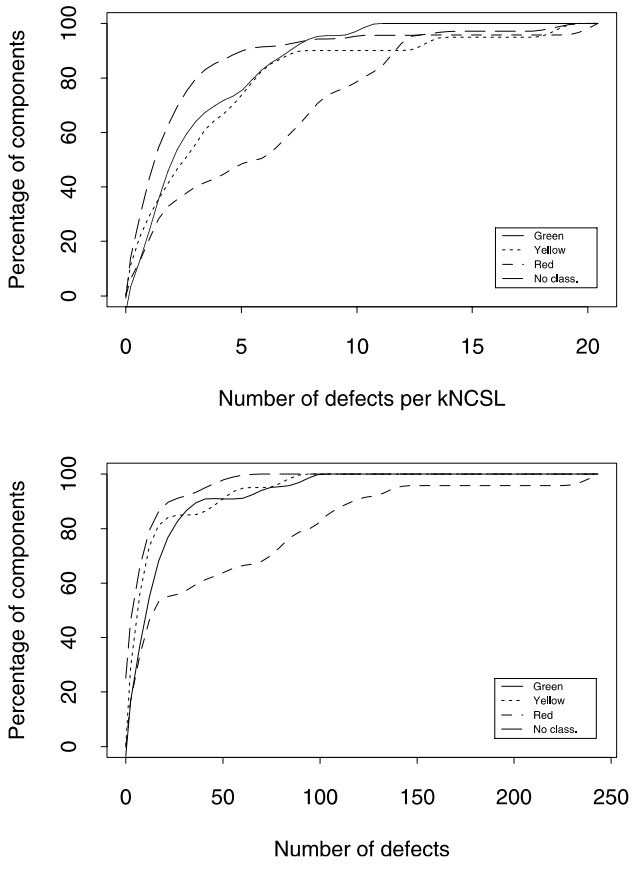

Fig. 8. Software process compliance metric.

From the results presented, it is clear that a process metric has been identified that allows an *early classification of error-prone software components*. The statistically significant difference in the defect distribution between differently classified groups allows us to use this process metric as exit criterion to SW component development. For our development process we conclude that there should be a mandatory milestone review, before components are passed on to system integration. During this review, all components should be classified according to the process metric. Those classified as "red" should not automatically be allowed to be handed over to system integration immediately, but rather should be improved first.

4. Code size and complexity study

On the same basis of defect data the third investigation aimed at a verification and justification of the predictive power of static code analysis. Various size, complexity and dependency measures have been

introduced in the past and it is believed that such static code analysis helps in identifying components with high defect density or prone to errors. Since we could not find convincing evidence for the capability of static code analysis throughout the literature that was available to us, we started an own investigation.

We applied the tool QAC (Programming Research Ltd, 1998) to the available code base and extracted size information and complexity and cohesion data: NCSL, CNCSL, cyclomatic complexity, static path count, nesting level. The aim we had was to identify a correlation between a combination of these metrics and the number of defects or defect density of SW components. To this end we not only performed regression tests on the entire set of components but furthermore subdivided the set into two groups with different ratio of changed/new to total code size.

Within the group of components that had more than 50% changed or new code, 70% of components have defect density larger than 1.5 per kNCSL. The risk that an individual from this group has large defect density was clearly tied to its static code properties.

In the second group of components with less than 50% changed or new code it was not possible to relate static code properties to defect numbers. Unless additional information is made available there is no value seen in measuring static code properties of such components. Our assumption is that, by adding information like the character of the changes (e.g., interface change, functional enhancement, new algorithm...) and in particular adding the information which piece of the code of a component was changed with an MR, code properties can become sensible metrics data.

Throughout our investigation, we found that size and complexity measures give essentially the same information on the expected defect numbers. Either no correlation between metric and defect data was found, or all used metrics showed similarly good correlation that could not be improved by adding more metrics to the regression analysis.

A side result of this study was the observation that the defect density of reused components with CNCSL/NCSL > 0.4 was in all cases above average over all components. From this fact, we conclude that a limit on the degree of change should be set. We concluded that *the limit of* 25% *in the ratio of changed to total code size CNCSL/NCSL* should not be exceeded and that components that have an estimated ratio of 25% or larger should be recoded entirely.

5. Summary and outlook

Measurement and evaluation is a critical activity throughout the entire software development and evolution lifecycle. It is fundamental to determining whether the software products we develop, have the desired functional and non-functional properties; it is fundamental to determining whether we have the desired cost, interval and quality attributes of our software development and evolution processes.

We have presented three different measurement and evaluation activities about an evolution point in the same project: a retrospective analysis of defects to determine the underlying root causes, an analysis of the relationship between process deviations and software defects, and an analysis of the relationship between static code properties and software defects.

5.1. RCA study

We have described the origins of our study, delineated the process of our retrospective root-cause analysis, and provided some of the analyses we performed on the data as illustrations and support for our subsequent improvement decisions.

We introduced a novel approach to root-cause analysis in this study: the possibility of more than one underlying root-cause. We replaced the previously known one-dimensional root-cause classification (that allows only for a single unique root-cause to be selected) by a three-dimensional root-cause space. There are several dimensions of underlying causes (for example, phase related causes such as ambiguous design documentation, human knowledge related causes such as lack of domain knowledge) which may interact with each other. The three dimensions are spanned by human, project, and life-cycle phase root cause. Unique root-cause selection thus requires specification of a root-cause value in each of the three directions. This choice reflects the general richness that underlies most of the faults that occur in the building and evolution of large complex software systems. We feel that this provides a more realistic view of what is going on and removes the need to force fit a problem into one specific category. We also allowed for the fact that sometimes people simply make mistakes and that there is nothing lurking beneath the surface that is the cause of that particular defect. The significant number of defects that fell into this category confirms our intuition about this problem.

The rest of our contributions to RCA are incremental to the approach in Perry and Stieg (1993).

Our RCA study yielded a number of important insights, some of them new, some of them confirming previously revealed insights.

- the cost of fixing bugs increases linearly, not exponentially, across the earlier phases (Perry and Stieg, 1993);
- the majority of the defects are in the design and coding phase (Perry and Stieg, 1993);
- different subsystems in the same project have different defect profiles;

- domain and system knowledge continue to be one of the largest underlying problems in software development (Perry and Stieg, 1993).

5.2. Process compliance study

Our process conformance metric (measuring the conformance to or deviation from the defined process) enabled us to provide an early classification of error prone software components. Moreover, the number of defects per changed NCSL varies with the degree of reuse and the total defect density.

5.3. Code size and complexity study

Complexity is an essential characteristic of the software systems that we build. We found that there is a significant correlation between the percentage of change in reused code and the number of defects found in those changed components. Where CNCSL/NCSL was greater than 0.4, the observed defect density was above average for all components. To limit the amount of fault injection in reused components, we recommend that a component be rewritten if more than 25% of the code is changed. As changes are made both to products and to processes, we need to measure and evaluate both to ensure that they conform to our desired set of characteristics.

Acknowledgements

The qualified contributions of the many RCA team members is largely appreciated.

References

Basili, V.R., Perricone, B.T., 1984. Software errors and complexity: An empirical investigation. Communications of the ACM 27 (1), 42–52.

Becker, R.A., Chambers, J.M., Wilks, A.R., 1988. The new S Language. Chapman and Hall, London.

Card, D.N., 1998. Learning from our Mistakes with Defect Causal Analysis. IEEE Software, New York, pp. 56–63.

Chillarege, R. et al., 1992. Orthogonal defect classification – a concept for in-process measurements. IEEE Transactions on Software Engineering 18 (11), 943–956.

Endress, A., 1975. An analysis of errors and their causes in systems programs. IEEE Transactions on Software Engineering SE-1 (2), 140–149.

Khoshgoftaar, T.M., Allen, E.B., 1999. A comparative study of ordering and classification of fault-prone software modules. Empirical Software Engineering Journal 4 (2), 159–186.

Krishnan, M.S., Keller, M.I., 1999. Measuring process consistency: Implications for reducing software defects. IEEE Transactions on Software Engineering 25 (6), 800–815.

Laitenberger, O., Leszak, M., Stoll, D., El-Amam, K., 1999. Causal analysis of review success factors in an industrial setting. In: Proceedings of the 6th IEEE International Symposium on Software Metrics, West Palm Beach, FL.

Leszak, M., Perry, D.E., Stoll, D., 2000. A case study in root cause defect analysis. In: Proceedings of IEEE International Conference on Software Engineering (ICSE-22), Limerick/Ireland, 7–9 June.

Ostrand, T.J., Weyuker, E.J., 1984. Collecting and categorizing software error data in an industrial environment. The Journal of Systems and Software 4, 289–300.

Paulk, M.C., Curtis, B., Chrisis, M.B., 1993. Capability Maturity Model for Software (CMM) Version 1.1. SEI Report, CMU/SEI-93-TR, 1993 (RCA requirements in key process area "defect prevention", CMM level5).

Perry, D.E., Evangelist, W.M., 1985. An empirical study of software interface faults. In: Proceedings of the International Symposium on New Directions in Computing, IEEE CS, Trondheim Norway, August, 32–38.

Perry, D.E., Evangelist, M., 1987. An empirical study of software interface faults – an update. In: Proceedings of the 20th Hawaii International Conference on System Sciences, pp. 113–126.

Perry, D.E., Stieg, C.S., 1993. Software faults in a large real-time system: a case study. In: 4th European SW Engineering Conference, Garmisch-Partenkirchen, p. 10.

Programming Research Ltd, 1998. QAC V4.1 – Deep Flow Static Analyzer User's Guide, UK.

Schneidewind, N.F., Hoffmann, H.M., 1979. An experiment in software error data collection and analysis. IEEE Transactions on Software Engineering SE-5 (3), 276–286.

Stoll, D., Leszak, M., Heck, T., 1999. Measuring process and product characteristics of software components – a case study. In: Proceedings of the 3rd Conference on Quality Engineering in Software Technology (CONQUEST-'99), Nuremberg, 27–29 September, ISBN3-00-004774-3.

Thayer, T.A., Pipow, M., Nelson, E.C., 1978. Software reliability – a study of large project reality. In: TRW Series of Software Technology, vol. 2. North Holland, Amsterdam.

Yu, W.D., Barshefsky, A., Huang, S.T., 1997. An empirical study of software faults preventable at a personal level in a very large software development environment. Bell Labs Technical Journal 2 (3), 221–232.

Yu, W.D., 1998. A software prevention approach in coding and root cause analysis. Bell Labs Technical Journal 3 (2), 3–21.

Dr. Marek Leszak manages the software process, quality, and measurement and improvement activities for Lucent Technologies, R&D Optical Networking at Nürnberg. He graduated (as Dipl.-Inf.) at University of Karlsruhe in 1979, and as Ph.D. (Dr.-Ing.) at Technical University of Berlin in 1986, both in Computer Science. He spent one year as a post-graduate at Ohio State University at Columbus, OH and seven years at the computer research center in Karlsruhe. In 1988, he joined the predecessor of the Lucent Nürnberg organization. Current interests include software process efficiency using metrics-driven quality improvements and database-oriented workflow management supporting process enactment.

Prof. Dewayne E. Perry is Professor and Motorola Regents Chair of Software Engineering in the Department of Electrical and Computer Engineering at The University of Texas at Austin. He is also the Director of the Executive Software Engineering program. His responsibilities at UT Austin include creating a software engineering program both at the graduate and undergraduate levels. His research interests are software architecture experimental software engineering, software process, and measurement and evaluation. He is a Co-editor in Chief of Wiley's Software Process: Improvement and Practice; an former associate editor of IEEE Transactions on Software Engineering; a member of ACM SIGSOFT and IEEE Computer Society; and has served as organizing chair, program chair and program committee member on various software engineering conferences.

Dr. Dieter Stoll is working on performance engineering and metrics for Lucent Technologies, R&D Optical Networking at Nürnberg. He graduated 1988 with a degree in physics from the University of Erlangen-Nürnberg where he also obtained a Ph.D. in 1992 in the same discipline. He spent 1/2 year at the University Paris-Sud in Orsay and two years at Tokyo University as researcher. In 1996, he joined Lucent Technologies. Main interests are in the areas of performance engineering, architectures, software reliability and metrics.