A Case Study of Defect Introduction Mechanisms

Arbi Ghazarian

Department of Computer Science, University of Toronto, Canada
arbi@cs.toronto.edu

Abstract. It is well known that software production organizations spend a sizeable amount of their project budget to rectify the defects introduced into the software systems during the development process. An in depth understanding of the mechanisms that give rise to defects is an essential step towards the reduction of defects in software systems. In line with this objective, we conducted a case study of defect introduction mechanisms on three major components of an industrial enterprise resource planning software system, and observed that external factors including incomplete requirements specifications, adopting new, unfamiliar technologies, lack of requirements traceability, and the lack of proactive and explicit definition and enforcement of user interface consistency rules account for 59% of the defects. These findings suggest areas where effort should be directed.

Keywords: Defects Sources, Defect Root Cause Analysis, Case Study.

1 Introduction

It has been frequently mentioned in the software engineering literature that maintenance activities are the dominant costs of developing software systems. Studies have shown that changes made to software systems account for 40 to 90 percent of the total development costs [4] [10] [3] [7] [6]. These changes can be corrective, adaptive, perfective, or preventive. Corrective changes deal with fixing the defects introduced into the software systems during the development process, and account for a significant portion of the maintenance costs; in their study of 487 data processing organizations, Lientz and Swanson [15] reported that, on the average, about 21% of the maintenance effort is allocated to the corrective maintenance. More recent studies have concluded that, in spite of the advances in software engineering in the past few decades, the maintenance problems have remained the same [18] [22]. These figures clearly indicate the potential economic value that can be gained from leveraging defect prevention techniques. A reduced rate of defects in the delivered software results in a reduction in the corrective maintenance activities, which in turn translates into a lower total development cost.

The importance of defect prevention has been emphasized by quality standards such as the Software Engineering Institute's Capability Maturity Model (SEI-CMM) [26], where defect prevention is a key process area for the optimizing maturity level (i.e., CMM Level 5). According to CMM [25]:

"Defect prevention involves analyzing defects that were encountered in the past and taking specific actions to prevent the occurrence of those types of defects in the future. The defects may have been identified on other projects as well as in earlier stages or

P. van Eck, J. Gordijn, and R. Wieringa (Eds.): CAiSE 2009, LNCS 5565, pp. 156–170, 2009.

tasks of the current project. Defect prevention activities are also one mechanism for spreading lessons learned between projects."

It goes without saying that an important first step to devising tools, techniques, and processes to counteract the mechanisms that give rise to software defects is to gain a profound understanding of these mechanisms. The work reported in this paper is an effort in this direction. However, it should be noted that due to the high variations among software projects in terms of the programming language, development process, design complexity, team size, organizational structure, application domain, individual team member characteristics, and many other factors, it is not possible to identify all possible defect introduction mechanisms in a single case study. Many studies of real-world systems are required to understand the full spectrum of defect introduction mechanisms.

We noticed that there are few published studies of defects on business software systems. Instead, most previous studies have been conducted on system-level software products in other software domains such as telecommunication, networking, real-time, and control systems. As a result, it is not clear to what extent results from these studies are applicable to business software systems. We, therefore, believe that there is a need for more studies of defects, similar to those performed in other domains, in the domain of business software systems. The importance of repeated studies by different researchers in establishing confidence in the results is well recognized by software engineering researchers. Hofer and Tichy [12] analyzed all the refereed papers that appeared in the Journal of Empirical Software Engineering from its first issue in January 1996 through June 2006, and observed that 26% of the papers describing an experiment were replications. Replication is needed to obtain solid evidence [23]. Moreover, it is only through conducting a multitude of similar studies and comparing the findings of these studies (i.e., looking for commonalities, differences, and patterns in defects in various projects) that we can gain deeper insights into questions such as:

- Which defect introduction mechanisms are common in all types of systems?
- Which defect introduction mechanisms are specific to, or occur more frequently in certain application types or domains? For instance, are the types of defects in business software applications different from those of scientific software systems?
- On average, what percentage of defects in software systems are pure logic mistakes on the programmers'part, and what percentage of defects are rooted outside the code and in other external sources such as incorrect or incomplete specifications?

Eldh et al. [8] emphasize that it is important to regularly collect and report findings about software defects from real industrial and commercially used systems to keep information in tune with development approaches, software and faults. They identify the lack of recent industry data for research purposes as the key problem. This view is supported by Mohagheghi et al. [17] who argue that there is a lack of published empirical studies on industrial systems and that many organizations gather large volumes of data on their software processes and products, but either the data are not analyzed properly, or the results are kept inside the organization. This situation hinders the spreading of lessons learned between projects in various organizations. The case study reported in this paper is a response to this need for more empirical studies on industrial systems. The main purpose of the study is to collect empirical evidence to answer the following two research questions:

1. What are the mechanisms that gave rise to defects in the case under study?
2. How large a role each identified mechanism has played in introducing defects in the case under study?

To answer our research questions, we performed root cause analysis (RCA) on 449 defects from a commercial Enterprise Resource Planning (ERP) software system. Our analysis, backed up by evidence drawn from project data including the defect reports in the defect tracking system, source code, requirements specifications, and test cases, as well as group sessions and individual interviews with the project team members identified a number of defect categories and their root causes, along with their frequencies.

Overall, we found that in our case, the causes for 59% of the reported defects were rooted in external factors including incomplete requirements specifications, adopting new, unfamiliar technologies, lack of requirements traceability, and the lack of proactive and explicit definition and enforcement of user interface consistency rules.

The rest of this paper is organized as follows: in Section 2, we report on our empirical study of defect introduction mechanisms. In Section 3, we discuss the related work. The conclusion and directions for future work follow in Section 4.

2 Case Study

2.1 Context of the Case Study

The organization where the case study was conducted is a manufacturer of telecommunication devices and an information technology company. For confidentiality reasons, we keep the organization anonymous. For the past five years, the IT department has been actively involved in developing a web-based ERP system comprised of various subsystems[1] including Bookkeeping, Inventory Management, Human Resources, Administration, Manufacturing, Procurement, and Workflow Management. The implementation of the ERP system is carried out in Java programming language, and the software developers involved in the project have an average of 7 years of industry experience.

The IT department follows a customized development process, which borrows concepts from both Rational Unified Process (RUP) and Extreme Programming (XP) methodologies. For instance, most programming is performed in pairs, which is an XP practice, whereas the requirements and analysis phases are conducted through a more traditional RUP-like process using use cases. At the time of this study, the project team was comprised of 28 individuals in various roles including a development manager, 5 system analysts, 13 developers, 4 testers, 2 graphics designers, and 3 marketing representatives. As with most long-term software projects, a number of individuals have left the team during the project, while a few others have joined the team. The team has had 35 members in its largest size. The development of the Bookkeeping, Human Resources, Administration, and Inventory Management subsystems has been completed, while the remaining subsystems are still under development. The final product is estimated to contain 200,000 lines of code.

[1] Throughout this paper, we use the terms system, subsystem, component, and module (source-code file) as units of system decomposition from largest to smallest, respectively.

The company uses the following process for handling defects. When a defect is detected, a defect report is registered in a defect tracking software tool. Each defect report captures a set of information including a unique defect identifier, a summary of the defect, a detailed description, the date the defect report is created, the subsystem to which the defect is attributed, the reporter of the defect, the assignee of the defect, priority, status, resolution, any number of additional comments by team members, and the date the defect report is last updated.

During system testing, all detected defects are assigned to a single team member (i.e., a point of contact between the testing and development teams), who in turn reviews the reported defects and further assigns them to the appropriate developers (i.e., the developer who has introduced the defect into the system) to be fixed. The idea behind this practice is that the developer who implements a feature is also the most qualified team member to rectify the defects reported on that feature since he/she is considered to be the most knowledgeable team member about the implementation details of that feature and therefore should be able to rectify the defect more reliably and in less time.

All team members, periodically or upon request by a team member, review all or some of the reported defects, and if they have any specific information or comments that can facilitate the correction of the defects, add them to the defect reports in the defect tracking system. Typical information in the added comments includes the cause of the defects, the location of the defects in the source code, how to fix the defects, and comments to clarify the descriptions of the defects. Developers assigned to the defects then use this information to correct the reported defects.

2.2 Description of the Case

All ERP subsystems in the studied organization are built on top of a shared infrastructure layer, which is composed of a set of reusable libraries and frameworks. The components in the infrastructure layer are either developed in-house or acquired as open-source software. All subsystems follow a three-tier layered architectural style, which is comprised of user interface, application logic, and data access layers. Each layer is considered a distinct component and as such each subsystem is divided into three major components. Our case study concerns the three components of the Bookkeeping subsystem. The first five columns in Table 1 present the characteristics of the target components in terms of the types of the modules (i.e., source-code files) in each

Table 1. Characteristics and Metrics of the Target Components

Component	Module Type	Size (LOC)	Module Count	Avg. Module Size (LOC)	Incorrect Impl.	Missing Impl.	Defect Count	Defect Density
User Interface	JSP	7802	41	191	180	150	330	38.606
	Javascript	678	1					
	XML	126	3					
Application Logic	Java	9434	24	393	22	41	63	6.677
Data Access	Java	4602	59	78	56	0	56	12.168
Total		**22642**	**128**		**258**	**191**	**449**	

component, the size of each component, the number of modules in each component, and average module size. The sizes of the components and modules are reported in non-commented and non-blank lines of code (LOC).

2.3 Case Study Process and Data Collection

At the time of this study, there were 482 defect reports in the defect tracking system, assigned to the target subsystem that we used in our case study. The time span between the first and the last defect was 17 months. Of these, 4 defects were labeled as "Duplicate", and 29 defects were labeled as "Not a Bug". The defects labeled as "Duplicate" were reported twice in the defect tracking system, whereas the ones labeled as "Not a Bug" were not actually defects and were initially reported as defects as a result of the incorrect usage of the subsystem or the unfamiliarity of the reporters of these defects with the correct behavior of the subsystem. We excluded these 33 defect reports, which left us 449 unique defects to study.

To analyze the distribution of the defects over the target components, the author of the paper and a member of the studied organization, who had a thorough understanding of the system, independently followed the analysis process described below to analyze each of the defect reports and attribute them to their corresponding components. This analysis of the distribution of the defects over the three target components was required since this information was not readily available; records in the defect tracking system included a data field that captured the attribution of the defects to the subsystems, but no data fields were available to capture the attribution of the defects to the lower-level units such as components and modules. The results from this analysis were required to compute the defect count and defect density measures for each of the components.

We used the following process to analyze the distribution of the reported defects over the three target components. We started our analysis by checking the information recorded in the "Defect Summary", "Detailed Description", and all of the available "Additional Comments" fields for each of the defect reports. Based on the information recorded in these fields, a group of the defects could be directly attributed to their corresponding components in the source code. For the remaining group of the defects, where the attribution of the defects to the components could not be derived from the information available in the defect reports, we recovered this information through conducting a series of defect review sessions. Each session was attended by two people: a research investigator who facilitated the session and recorded the recovered information, and a developer who had been involved in fixing the defects. During each defect review session, all defects fixed by the participant developer were discussed and attributed to their corresponding component. Where necessary, the system's source code was consulted to locate the components related to the defects. In addition to the attribution of the defects to their corresponding components, where possible, for each defect, we identified the internal cause of the defect in the source code (e.g. an incorrect database query statement or missing source code statements). We also determined whether the defect was caused as a result of a missing or incorrect implementation.

To ensure the quality of the collected information, the abovementioned analysis process was conducted twice and independently. The results of the two analyses were in close agreement (Cohen's Kappa inter-rater agreement of 0.79), which is an indication

of the objectivity of the analyses performed. The cases where there was a difference between the two analyses were jointly reexamined to reach a consensus.

We then measured the sizes of the target components in non-blank and non-commented lines of code using a software tool, and computed the defect density for each of the target components. The results are summarized in the last two columns in Table 1, which present the Defect Count and Defect Density (in defects per KLOC) for each of the target components, respectively.

We followed our data collection and analysis process by performing root cause analysis of the reported defects through conducting a series of group sessions and interviews with the team members. During these sessions, we used input from project team members and, for each defect, identified the external factor that underlay the internal cause of the defect (e.g., the unfamiliarity of the developers with the new query language underlying the incorrect query statements, or incomplete requirements specification documents underlying the missing source code statements). To ensure the correctness of the identified root causes, the team members frequently consulted the project data including defect information in the defect tracking system, requirements documents, test cases, and the source code, as well as the results of the analysis of the attribution of the defects to the components and the collected project metrics (size, defect count, and defect density). As a result of these sessions, we traced each defect back to its origin, which led to the classification of the defects based on their root causes. Unfortunately, the available data was not detailed enough to allow us to calculate the cost and effort spent on rectifying the defects.

2.4 Results

Based on a detailed analysis of the data collected during the study, we make several observations about the mechanisms that gave rise to defects in the studied subsystem. We discuss the impact of each identified mechanism on defect rate of the subsystem.

The Impact of Adopting New, Unfamiliar Technologies on Defect Rate. The data from our study show that of the 56 defects attributed to the data access component (see Table 1), 47 (roughly 84%) were caused by incorrect database query statements. We discussed this finding with the development team and found that the unexpected number of query-related defects in the data access component was due to the adoption of a new database query technology. The development team had adopted an unfamiliar query language to implement the data access component. Since there had been no previous experience and expertise on the newly adopted query language, the team had encountered many problems with this new technology. Only 16% of the defects attributed to the data access component were caused by the code written in the Java language, which constitutes the bulk of the code in the data access component and serves as the host language for the embedded queries.

We excluded the defects directly caused by the introduction of the new technology and recalculated a defect density of 1.955 for the data access component. Comparing this new calculated value for the defect density in the data access component with its current value from Table 1 (12.168) clearly demonstrates the negative impact of adopting the new, unfamiliar technology in increasing the defect rate in this component. As

a result of adopting the unfamiliar technology, the defect density of the data access component has increased by a factor of 6.22. The data from our study suggest that:

The adoption of new, unfamiliar technologies into a software component is a risk factor that has adverse effects on the component's quality in terms of its defect rate.

This observation is not surprising. There is an intuitive consensus in the software engineering literature on the correctness of this proposition. However, empirical evidence taken from industrial software systems, like ours, to support it can strengthen our beliefs in this proposition.

The Impact of Incomplete Requirements Specifications on Defect Rate. As part of our data analysis, we classified the reported defects under two broad categories of incorrect implementation and missing implementation, and observed that about 57.5% of the defects (258 cases) were caused as a result of incorrect implementations of the requirements in the code, whereas the remaining 42.5% of the defects (191 cases) were a result of missing implementations from the code. Columns 6 and 7 in Table 1 present the distribution of defects classified as incorrect versus missing implementation over the three components. A chi square test of independence, at the significance level of 0.05, revealed that the type of defect (incorrect or missing implementation) is dependent on component type (user interface, application logic, or data interface). We further observed that the majority of the defects classified under the missing implementation category were related to missing business rules and data validations. Defects caused by missing implementations (i.e., the code that is necessary is missing) have also been referred to as *"faults of omission"* in the literature [8].

Our inspection of the requirements specification documents revealed that in 156 cases (roughly 82% of the defects in the missing implementation category), the system analysis team had not explicitly included the requirements in the requirements specification documents, and were consequently missing from the system's implementation as well. In a series of defect root cause analysis group session with the team members, we reviewed all of the 191 defects in the missing implementation category, and confirmed that the reason for the 156 missing implementation cases which didn't have corresponding requirements was actually the missing requirements.

We know for a fact that the requirements specification documents were the main means through which the requirements of the system were communicated to developers. The introduction of this group of defects into the system's source code can be directly traced back to the incomplete requirements specifications. This conclusion is consistent with the results of the interviews conducted with the development team members. When asked about the relatively large number of defects related to missing implementations, the team members expressed their opinions in statements such as "what is obvious for the business analysis team is not clear to anyone else in the development team. Consequently, if they fail to communicate some of the business requirements in an explicit manner, we are highly likely to miss these requirements in our implementations" and "when we start the development of a new use case of the system, we are not provided with all the details. What we initially receive from the business side includes a description of the use case including the main and alternative flows, and some of the major related business requirements, but the documents are not comprehensive enough

to cover all aspects of the use cases including some of the data validations and less obvious business rules. Therefore, a number of missing requirements are detected during the system testing".

An interesting aspect of this observation is that it puts into question the comprehensiveness of the traditional view of a defect as any characteristic of the system that does not comply to its predetermined (proactively and explicitly documented) specification. In our case, we observed that there were no predetermined specifications for a significant number of functionalities in the system and yet the absence of these functionalities from the system were reported as defects, whose rectification were required for the correct operation of the system. An interview with the testing team revealed that they had partly relied on their implicit knowledge of the system domain to test the software system. For instance, while the requirements documents lacked some of the data validation rules, the testing team, relying on their knowledge of the system, had identified and incorporated some of these missing rules into their test cases. A consequence of this phenomenon is that a part of software requirements are documented outside the requirements specification documents, and inside the test cases and defect records. This practice, over time, can lead to the loss of parts of the system knowledge as a part of requirements are buried within test cases and defect reports. Our data suggest that:

Incomplete requirements specifications (i.e., some requirements are not explicitly stated in the specifications) are a mechanism for introducing defects into software systems in the form of missing implementations.

As mentioned earlier, in our case, the development team was following a traditional document-centric requirements process. This means that the majority of communication between the business analysis and development teams was taking place through requirements documents. A direct consequence of this reliance on documentation as the main form of communicating system requirements is that the performance of the system developers (e.g., in terms of the number of defects related to missing implementations introduced into the system) becomes partly dependent on the quality of the documents produced by the requirements team (e.g., in terms of the completeness of the requirements). The results could be different in projects with an agile requirements process, where the project team mostly relies on verbal communication of requirements.

The Impact of the Lack of Requirements Traceability on Defect Rate. An interesting observation was that in the remaining 35 defects classified under the missing implementation category (roughly 18% of the defects in this category), the corresponding requirements were existing somewhere in the requirements documents and were somehow overlooked by developers.

Our inspection of the requirements documents in conjunction with group sessions with the team members revealed that these cases were related to the business concepts or data items and their associated business rules and data validations that were defined in one requirements document and implicitly referred to in other requirements documents. For instance, consider the case where a step in one of the system use cases states that the user shall enter a data item into the system as part of the data entry for that use case, without explicitly making references to the other use cases of the system where the data validation rules for this data item have been specified. Since there were no explicit

links between the parts of a document that referred to external business concepts or data items and the parts of the other documents that actually specified the requirements for that concept (requirement-requirement traceability link), developers either did not realize that some of the requirements pertaining to the feature are defined somewhere else, or had to manually navigate between the various requirements documents to capture a complete view of the requirements pertaining to the feature under development. This process of moving from a requirements document to another in order to collect all the relevant requirements can be problematic when a document has many points of jump (i.e., items mentioned in a document are specified in other documents, but not explicitly linked to those external documents). In other words, the requirements related to a feature under development were scattered across different documents without an explicit mechanism for cross-referencing, and the human errors involved in the process of navigating between the various documents and collecting the complete set of requirements had led to the introduction of a group of defects into the system in the form of missing implementations. Since, other than system testing, there was no other mechanism in place to verify the completeness of the implementation of a feature under development with regards to its specified requirements, these missing implementation defects were remained hidden until system testing. Our data suggest that:

The lack of traceability between requirements specification documents, when the requirements pertaining to a feature of the system are scattered across multiple documents, plays a role in the occurrence of the cases where the requirements are existing in the requirements documents but missing from the implementation.

The Impact of Not Proactively Defining and Enforcing the User Interface Consistency Rules on Defect Rate. Another observation is that the cause of 8% of the defects in the user interface component (or 6% of the total number of defects in the subsystem under study) can be directly traced back to inconsistencies in the user interface. A close examination of this group of defects in conjunction with a group session with developers revealed that in the absence of explicit consistency rules for the unification of the system's user interface behavior, developers had made individualistic and ad hoc decisions in their implementations, which led to inconsistencies in the user interface of the system. The descriptions given for these defects in the defect tracking system refer to various types of inconsistencies in the user interface of the system including inconsistencies in the screen layouts, user interface navigation methods, fonts, and data formats in various screens and reports displayed by the system. In the subsystem under study, these forms of inconsistencies were considered to be detrimental to the usability of the application and as such any occurrences of such inconsistencies in the user interface were reported as defects. The data from our study suggest that:

The lack of explicit and proactive definition and enforcement of implementation consistency rules leads to defects in cases, such as the user interface, where inconsistent implementations are considered defects.

The Impact of Software Size on Defect Rate. The relationship between defect measures such as the defect count and defect density, and software size has been the subject of many studies in the literature. For example, [9], [19], [2], [21], [24], [1], [11], [20],

and [16] are some of the studies that have been conducted in this area. Table 4 in the related work section of this paper summarizes the key results from these nine studies. Some of the previous studies report a relationship between these parameters, while others do not. What makes the situation complicated is that the results from the studies where a relationship between these two parameters have been observed are conflicting. Some of these studies report a rising trend of defects as the size increases, while other studies report a declining trend of defects as the size grows. Others have tried to explain these rising and declining trends.

The data from our study does not suggest any noticeable dependence between defect rate and component size. 73.5% of all reported defects are attributed to the user interface component. In contrast to this high defect concentration, the business logic, and data access components have a share of only 14% and 12.5% of the total reported defects, respectively. Obviously, this noticeable difference between the defect distributions over the user interface component and the other two components cannot be attributed to the sizes of the components. Although the user interface component is almost the same size as the business logic component, and only twice the size of the data access component, it has at least five times more defects compared to each of the other two components. Since our data come from only three components, we cannot draw any conclusions about the dependence between the defect rate and component size.

We would like to study the relationship between defect measures and module size (as opposed to component size). Unfortunately, no data were available on the distribution of defects over the modules. An attempt to collect these data after the fact would be extremely time consuming and error prone.

2.5 Summary of the Results

Based on the analysis of the collected data, we can now answer the two research questions posed in the introduction.

1. What are the mechanisms that gave rise to defects in the case under study?
2. How large a role each identified mechanism has played in introducing defects in the case under study?

Table 2 summarizes the answers to these questions. The major conclusions and contributions of our study are presented below. These findings suggest areas where effort should be directed.

Table 2. Defect Introduction Mechanisms Identified in the Subsystem Under Study

Defect Introduction Mechanism	**Defect Count**	**% of Defects**
Incorrect implementations not linked to external causes	184	41
Incomplete requirements specifications	156	34.7
Adopting new, unfamiliar technology	47	10.5
Lack of traceability between requirements specifications	35	7.8
Lack of consistency in the user interface	27	6
Total	449	100

- A significant portion of the defects originate outside the source code. This finding is supported by the evidence that 59% of the reported defects were propagated into the considered subsystem from external sources including incomplete requirements specifications, adopting new, unfamiliar technologies, lack of requirements traceability, and the lack of proactive and explicit definition and enforcement of user interface consistency rules.
- Specification-related defects represent the largest category of defects (42.5%, of which 34.7% were caused by incomplete requirements specifications, and 7.8% were a result of a lack of traceability between various requirements specifications).

2.6 Implications of the Findings

Given the profile of defect sources identified in our case study, in Table 3, we propose a set of mitigation strategies that software development organizations can employ to counteract these defect introduction mechanisms.

Table 3. Defect Mitigation Strategies

Defect Source	Defect Mitigation Strategy
Incomplete requirements specifications	• Improving the requirements process in terms of the completeness of the requirements specifications • Explicit documentation of all business rules and data validations
Adopting new, unfamiliar technology	• Thorough evaluation of new technologies before adopting them • Provision of sufficient training when a decision is made to adopt a new technology
Lack of traceability between requirements	• Practicing requirements traceability • Automated support for requirements process including tools to help the project team to keep track of the relationships between requirements and requirements status
Lack of consistency in the user interface	• Proactive definition and enforcement of user interface design rules to unify the implementation of user interface look and behavior

2.7 Threats to Validity

Several factors potentially affect the validity of our findings. We discuss these factors under standard types of validity threats in empirical studies.

Construct Validity. The construct validity criterion questions whether the variables measured and studied truly reflect the phenomenon under study. We use defect count and defect density as surrogate measures for the quality of the software components. These measures are widely used in software engineering studies. All component and module sizes were measured in lines of non-commented and non-blank code. The purpose of the study was to identify some of the mechanisms that give rise to defects in software systems. Our observations involve the calculated measures of size, defect count (for various categories of defects), and defect density. Therefore, the variables measured and studied, truly reflect the purpose of the study.

Internal Validity. The internal validity of a study is concerned with distinguishing true causal relationships from those effected by confounding variables. In our study, the incorrect attribution of the defects to the components could be a threat to the internal validity of our findings. To minimize the potential effect of this confounding factor, the author of the paper and a member of the development team, who had a detailed knowledge of the system and had been actively involved in fixing the reported defects, independently analyzed the defects and attributed them to their corresponding components. For the great majority of the defects, we were able to accurately assign the defects to their corresponding components. The accuracy of the assignment of defects to the components was evident from the closely matching results of the two analyses. The cases where there was a difference in the results of the two analyses were jointly reexamined and resolved. We are highly confident that the attribution of defects to their corresponding components was accomplished accurately.

We studied the entire population of the reported defects in the subsystem under study. This effectively eliminated any potential sampling bias, which can be a problem for studies where a selected sample of the population is included in the study. Furthermore, the inclusion of the complete set of defects in our study not only helped us to obtain a complete picture of the defects and their root causes, but also increased the reliability of the conclusions drawn from the analysis of the data.

In studies where components developed in multiple languages are involved, an *equivalent code* must be calculated for the components, to make the comparison of the component sizes meaningful. The software tool used in our study to measure the component sizes provides an equivalent code size in a hypothetical third generation programming language. To calculate the equivalent sizes, the software tool multiplies the component size in Java with 1.36, Javascript with 1.48, JSP with 1.48, and XML with 1.90. We recalculated all the metric data collected in our study using the equivalent sizes. The results did not change any of our findings or conclusions.

To further validate our findings, we discussed them with the team members during our interviews and group sessions. They believe that the five findings of the study, presented in Section 2.4 of this paper, capture an accurate portrayal of their situation.

External Validity. The external validity of a study is concerned with the extent to which the findings of the study can be generalized. Our dataset was taken from one product of a development organization, which can be a limit to the generalizability of our findings. The results of our study can be considered as early evidence for some of the mechanisms that give rise to defects in software systems. Gaining more confidence in the results of the present study requires further replication of the study with other systems within and outside the considered organization. This is planned as future work.

3 Related Work

The work of Leszak et al. [13] is similar to ours in intent. They conducted a root cause analysis study of defects in a large transmission network element software, and based on the findings of the study devised countermeasures to either prevent the defects or detect them earlier in the development process. Results from our study contradict their

findings. They concluded that the majority of defects do not originate in early phases. They report that in the system they studied defects were introduced into the system predominantly (71%) within the component-oriented phases of component specification, design, and implementation. In contrast to our study results, in their case, requirements-related defects did not have a significant contribution to the total number of defects.

Eldh et al. [8] studied and classified the failures in a large complex telecommunication middleware system. They concluded that faults related to unclear specifications (46.6%) dominates among the software faults. In their case, faults of omission and spurious faults accounted for 38.3% and 8.3% of the total faults, respectively. Our study agrees with their finding. In our case, we also observed that specification-related defects represent the largest category of defects (42.5%). Our work is different from theirs both in its motivation and the focus of the study. Our goal is to identify the causes of defects so that appropriate actions can be initiated to prevent the sources of defects. To fulfill our goal, we performed root cause analysis of the defects and traced the defects to their sources outside the code and into the external factors (e.g., specifications, processes, and decisions). In contrast, the main motivation behind the classification of faults in Eldh et al.'s work is to investigate software test techniques through injecting the identified classes of faults into code. As a result, in contrast to our focus on the external root causes of the defects, their focus is on the static origin of the faults within the code.

From a research methodology point of view, our data selection approach is different from both Eldh et al. [8] and Leszak et al.'s [13] study. In our study, the entire population of defects in the considered subsystem was included for analysis. In comparison, the data used in Eldh et al.'s [8] study were selected by convenience sampling. they selected their sample data set from the defects whose labels in the configuration management system allowed them to trace the failures back to their origins in the system's source code. Leszak et al.'s [13] used a combination of manual and random sampling. Another distinction between our study and the two previous studies is that our data is taken from an ERP system, which is a business software system, whereas both previous studies collected data from system-level software products namely, telecommunication middleware software and transition network element software.

Chillarege et al. [5] propose a semantic classification of defects called Orthogonal Defect Classification (ODC), which is comprised of eight distinct defect types, each capturing the meaning of a type of defect fix. The distribution of defects over the ODC

Table 4. Summary of Observations from the Previous Studies on Defect Measures and Size

Study	Results
Fenton and Ohlsson [9]	(a) No significant relation between fault density and module size. (b) A weak correlation between module size and the number of pre-release faults. (c) No correlation between module size and the number of post-release faults.
Ostrand and Weyuker [19]	(a) Fault density slowly decreases with size. (b) Files including a high number of faults in one release, remain high-fault in later releases. (c) Newer files have higher fault density than older files.
Basili and Perricone [2]	Larger modules are less error prone, even when they are more complex in terms of cyclomatic complexity.
Shen et al. [21]	(a) Of 108 modules studied, for 24 modules with sizes exceeding 500 LOC, the size does not influence the defect density. (b) For the remaining 84 modules, defect density declines as the size grows.
Withrow [24]	A minimum defect density for modules with sizes between 161 and 250 LOC, after which the defect density starts increasing with module size.
Banker and Kemerer [1]	Proposed a hypothesis that for any given environment, there is an optimal module size. For lesser sizes, there is rising economy, and for greater sizes, the economy declines due to rising number of communication paths.
Hatton [11]	(a) For sizes up to 200 LOC, the total number of defects grows logarithmically with module size, giving a declining defect density. (b) For larger modules, a quadratic model is suggested.
Rosenberg [20]	Argued that the observed phenomenon of a declining defect density with rising module sizes is misleading.
Malaiya and Denton [16]	(a) Proposed that there are two types of defects: module-related defects, and instruction-related defects. (b) Module-related defects decline with growing module size. (c) The number of instruction-related defects rises with growing module size. (d) An optimal module size for minimum defect density is identified.

classes changes with time, which provides a measure of the progress of the product through the process. The main motivation for ODC is to provide feedback to developers during the development process. In contrasts to ODC's focus on providing in-process feedback, the classes of defects identified by our study, along with the observed distribution of defects over these classes can provide after-the-fact feedback to developers, which can be used to improve the development of the next subsystems within the organization. In this sense, a study like ours can serve as a means for spreading lessons learned between projects. Given the qualitative nature of performing root cause analysis of defects, the resources required to perform the analysis are significant, which might make it impractical for providing in-process feedback.

There have been several studies about the relationship between defect-based measures such as defect count and defect density, and software size. Table 4 summarizes the findings of these studies.

4 Conclusion and Future Work

Defect prevention is a possible way to reduce software maintenance costs. However, to devise tools, techniques, and processes to support defect prevention requires an understanding of the mechanisms that give rise to defects during the development process. Case studies of real-world industrial systems are a systematic approach towards gaining such an insight. The study reported in this paper is meant to serve such a purpose.

Our case study identified four possible defect introduction mechanisms including incomplete requirements specifications, adopting new, unfamiliar technologies, lack of traceability of requirements, and the lack of explicit definition of user interface consistency rules that collectively account for 59% of the defects in the subsystem under study. These four defect introduction mechanisms are all well understood, which suggests that the cause of a significant portion of defects in industrial software projects is not a lack of knowledge, but rather a lack of application of existing knowledge.

In our future work, we intend to replicate this study with the other subsystems in the considered organization, as well as with ERP systems in other organizations in order to determine if patterns of defect introduction mechanisms exist among ERP systems. This should also give us insights into the factors affecting the magnitude of the defects introduced by each identified mechanism.

For the remaining 41% of the reported defects, no external contributing factors could be found. We intend to investigate this group of defects in future work. We plan to analyze historical data from the revision control system to track and analyze the changes made to the modules to fix the defects in order to understand the nature of these defects.

References

1. Banker, R.D., Kemerer, C.F.: Scale Economics in New Software Development. IEEE Transactions on Software Engineering, 1199–1205 (October 1989)
2. Basili, V.R., Perricone, B.R.: Software Errors and Complexity. Communications of ACM 27, 42–45 (1984)
3. Bersoff, E., Henderson, V., Siegel, S.: Software Configuration Management. Prentice-Hall, Englewood Cliffs (1980)

4. Boehm, B.W.: Software and its Impacts: A Quantitative Assessment. Datamation 9, 48–59 (1973)
5. Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.: Orthogonal Defect Classification - A Concept for In-Process Measurements. IEEE TSE 18(11), 943–956 (1992)
6. Cleland-Huand, J., Chang, C.K., Christensen, M.: Event-Based Traceability for Managing Evolutionary Change. IEEE TSE 29(9), 1226–1242 (2003)
7. Devanbu, P., Brachman, R.J., Selfridge, P.G., Ballard, B.W.: LaSSIE: A Knowledge-Based Software Information System. Com. of ACM 34(5), 34–49 (1991)
8. Eldh, S., Punnekkat, S., Hansson, H., Jonsson, P.: Component Testing Is Not Enough - A Study of Software Faults in Telecom Middleware. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 74–89. Springer, Heidelberg (2007)
9. Fenton, N.E., Ohlsson, N.: Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Transactions on Software Engineering 26(8), 797–814 (2000)
10. Fjelstad, R.K., Hamlen, W.T.: Application Program Maintenance Study - Report to Our Respondents. Technical Report, IBM Corporation, DP Marketing Group (1986)
11. Hatton, L.: Reexamining the Fault Density-Component Size Connection. IEEE Software, 89–97 (March 1997)
12. Hofer, A., Tichy, W.F.: Status of Empirical Research in Software Engineering. In: Empirical Software Engineering Issues, pp. 10–19. Springer, Heidelberg (2007)
13. Leszak, M., Perry, D.E., Stoll, D.: A Case Study in Root Cause Defect Analysis. In: Proceedings of International Conference on Software Engineering, ICSE 2000, pp. 428–437 (2000)
14. Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of Application Software Maintenance. Communications of the ACM 21(6), 466–471 (1978)
15. Lientz, B.P., Swanson, E.B.: Software Maintenance Management. Addison-Wesley, Reading (1980)
16. Malaiya, K.Y., Denton, J.: Module Size Distribution and Defect Density. In: Proceedings of ISSRE 2000, pp. 62–71 (2000)
17. Mohagheghi, P., Conradi, R., Killi, O.M., Schwarz, H.: An Empirical Study of Software Reuse vs. In: Defect-Density and Stability. In: Proceedings of ICSE 2004, pp. 282–292 (2004)
18. Nozek, J.T., Palvia, P.: Software Maintenance Management: Changes in the Last Decade. Journal of Software Maintenance: Research and Practice 2(3), 157–174 (1990)
19. Ostrand, T.J., Weyuker, E.J.: The Distribution of Faults in a Large Industrial Software System. In: Proceedings of ISSTA 2002, pp. 55–64 (2002)
20. Rosenberg, J.: Some Misconceptions about Lines of Code. In: Proceedings of the International Software Metrics Symposium, November 1997, pp. 137–142 (1997)
21. Shen, V.Y., Yu, T., Thebut, S.M.: Identifying Error-Prone Software- An Empirical Study. IEEE Transactions on Software Engineering 11, 317–324 (1985)
22. Vliet, H.V.: Software Engineering: Principles and Practices. John Wiley & Sons, Chichester (2000)
23. Tichy, W.F.: Should Computer Scientists Experiment More? IEEE Computer 31(5), 32–40 (1998)
24. Withrow, C.: Error Density and Size in Ada Software. IEEE Software, 26–30 (1990)
25. Whitney, R., Nawrocki, E., Hayes, W., Siegel, J.: Interim Profile: Development and Trial of a Method to Rapidly Measure Software Engineering Maturity Status. Technical Report,CMU/SEI-94-TR-4, ESC-TR-94-004, March 26-30 (1994)
26. `http://www.sei.cmu.edu/index.html`