SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. (2010)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.426

Improved code defect detection with fault links

Jane Huffman Hayes[1,*,†], Inies R. Chemannoor[2] and E. Ashlee Holbrook[3]

[1]*Department of Computer Science, University of Kentucky, Lexington, KY 40506, U.S.A.*
[2]*Intel, Santa Clara, CA 95051, U.S.A.*
[3]*Lexmark, Lexington, KY 40506, U.S.A.*

SUMMARY

Fault links represent relationships between the types of code faults, or defects, and the types of components in which faults are detected. For example, our prior work validated that a fault link exists between Controller components and Control/Logic faults (such as unreachable code). Fault link information can guide code reviews, walkthroughs, testing, maintenance, and can advise fault seeding. In this paper, we use fault links to augment code reviews. Two experiments were undertaken to evaluate the usefulness of fault links, one with 26 Computer Science students and another with 24 software engineering professionals. The first experiment showed that fault link information assisted in finding more total defects and more 'hard to detect' defects, in the same amount of time, in a Java component of an online course management application. The experiment was repeated with professionals, adding a second Java component from the same application. For the second experiment, more total defects were found by the participants using fault link information for one of the two components and more hard to detect defects were found, in the same amount of time, in both Java components. The group using fault link information for code walkthroughs found, on average, 1.7–2 times more faults and 2–3 times more hard faults than the control group. Copyright © 2010 John Wiley & Sons, Ltd.

Received 25 September 2008; Revised 16 September 2009; Accepted 12 November 2009

KEY WORDS: fault taxonomy; code inspections; defect-based analysis; fault-based analysis; fault link

1. INTRODUCTION

Widespread virus attacks, growing daily reliance, software incompatibilities, system and software hacking, and competition between software firms all serve to increase the demand for high quality, reliable software. Some reasons that software companies fail to produce quality products are the lack of resources to ensure the software quality (time, money, tools, etc.) and the lack of knowledge regarding the timely use of resources. Fault-based analysis (FBA) [1] seeks to address these problems by applying techniques early in the life cycle, using information about the historical occurrence of pre-specified fault types to drive the selection of the verification and validation (V&V) techniques. A fault taxonomy is an important requirement for applying FBA.

*Correspondence to: Jane Huffman Hayes, Department of Computer Science, University of Kentucky, Mail Stop 0495, Lexington, KY 40506-0495, U.S.A.
†E-mail: hayes@cs.uky.edu

Contract/grant sponsor: NASA Johnson Space Center
Contract/grant sponsor: International Space Station program; contract/grant number: NNG04GA38G

Our semantic fault model work [2], our work on input validation testing [3], Offutt's work on testing coupling effect [4], our work on traceability [5, 6], and our work on requirement faults [1] led us to a conjecture about faults: the types of mistakes made by programmers are largely dependent on the type of module that is being developed or modified. We refer to this relationship as a *fault link* that is part of a larger relationship or sequence called a fault chain. The term *fault chain* refers to a relationship that exists between faults that have occurred during different phases of the software development life cycle. For example, a fault that results in unnecessary processing that is caught during software testing may be traced back to an ambiguous requirement that existed in the requirements specification. A fault chain also exists when faults are repaired and new faults are introduced as a result of the repair actions.

Our prior work established the existence of fault links for a specific domain [7]. Here, we focus on using the knowledge of fault links to improve the Verification and Validation (V&V) activities, such as code inspections. It may seem intuitive that fault link knowledge will assist with quality assurance activities, but empirical validation is required. The need for experiments in software engineering is evident [8]. New technologies and changes in the software process require testing before adoption, in order to determine their impact in the specific context and domain [9].

We posit that current software development and V&V practices improve with the knowledge of fault links by allocating quality assurance resources to the most probable component type/fault type pairs for the domain. We provide a customized design and a code walkthrough checklist for new developments (based on fault links) and recommend that customized exit criteria be added to walkthrough checklists for maintained design or code. In addition, we can offer guidance to testing and reliability researchers who rely on fault seeding as a technique evaluation mechanism. As pointed out by Offutt and Hayes [2], researchers tend to seed syntactically small faults, faults that only affect the program execution negatively for a small number of variable values. Through our fault link research, we gain deeper insight into true distributions of fault types and understand which types of faults researchers should be seeding based on the component type. The contribution of this paper is several-fold: (a) we use fault links to augment the code review process and (b) we empirically evaluate the application of fault links in two code reviews, one with students and the other with professionals.

In this paper, we examine fault links for the online course management software domain. We then undertake two experiments to examine the effectiveness of fault link information-enhanced inspections. The remainder of the paper is organized as follows. Fault links are presented in Section 2. Section 3 describes the experiments. The results are presented in Section 4. The related work is presented in Section 5. Section 6 is devoted to conclusions and future work.

2. FAULT LINKS AND APPLICATIONS

This section discusses fault links and how they are applied to assist with code walkthroughs.

2.1. Fault links

A number of definitions are in order. A component can be a single statement or a single function or procedure that contributes to the purpose of a software program [10, 11]. According to the Merriam Webster's dictionary, taxonomy is 'the study of the general principles of scientific classification [12]'. We define taxonomies for faults and for components as follows:

- *Generic code-fault taxonomy*: A fault taxonomy that can be used to classify faults that occur in any domain or project.
- *Generic component taxonomy*: A component taxonomy that can be used to classify components that occur in any domain or project.

We work with both of the above. The generic taxonomies have been obtained through an extensive literature survey and also by categorizing defects on two projects. Defect reports for

DOI: 10.1002/stvr

Code fault taxonomy definitions:

1. Data:

1.1. Data definition: The process of assigning attributes and/or values to a data structure is called data definition. Based on what is assigned, we can further decompose data definition into two sub-categories:

1.1.a. Data declaration: The process of assigning data type and memory bytes to data structures is called data declaration. Errors during the declaration of data fall under this category.

Example: "int x;" instead of "float x;"

1.1.b. Data initialization: The process of assigning the start or initial value without any computation is called data initialization. Errors during initialization fall under this category.

Example: "x=1;" instead of "x=0;"

1.2. Data representation: Well-defined data can represent relevant or irrelevant aspects of the software. Incorrect representation of data falls under this category.

Example: Representing variable x as the area of a triangle instead of the area of a square.

1.3. Data accessing: This refers to accessing data from data structures that are defined accurately. The accessed data structures are presumed to be correctly defined. The following is a list of sub-categories of fault types that are grouped under data accessing:

Example: Incorrect data type for processing or incorrect storing and retrieving of data or incorrect data referenced.

Component type taxonomy definitions:

Data-centric: Modules that deal with data definition and handling fall under this category. Access to a database is classified as data-centric.

Computational-centric: Modules that calculate or compute results belong in this category. If one looks at a lower level of detail, any statement that changes any variable or state of the program falls under this category.

Controller: Modules that control the sequence of program execution falls under this category. These collaborative components (or statements, at a lower level of detail) form a backbone for the software because they determine the instructions to be executed and the number of times they are to be executed.

Figure 1. Subset of Code fault and component type taxonomy definitions.

the Apache Web Server (Version 1.23.x) and the Mozilla Web Browser (Version 3.23.x) were examined [7]. Subsets of the generic component taxonomy and the generic fault taxonomy, referred to hereafter as merely the component and fault taxonomy, are shown in Figure 1[‡].

We followed a subset of the steps presented by Huffman Hayes [1] and developed a domain-specific component taxonomy and a domain-specific fault taxonomy: select a fault taxonomy as a basis for the work, examine sample code faults, adopt or build a method for extending the fault taxonomy, and implement the method for tailoring the taxonomy. Each is discussed below.

2.1.1. Component taxonomy. Any program, be it simple or complex, can be viewed as a component or a combination of components. Each component serves a unique purpose. Note that we use the terms 'module' and 'component' interchangeably. For the online course management software domain, we identified eight component types: data-centric, computational-centric, controller, view, interaction, utility, error handling, and environmental setup/configuration. We endeavoured to make the component taxonomy language-independent and method-independent (i.e., procedural, object-oriented, component-based, etc.). This same goal applied when we developed the fault taxonomy, discussed next.

2.1.2. Code fault taxonomy. The code fault taxonomy is shown in Figure 2. The branches of the tree represent fault categories that are language-independent, but the leaves may be language-dependent. For example, the control/logic fault type applies to any language, but register reuse is only applicable to languages such as C. The taxonomy was built using only bug reports and source code. It does not require that specifications are available for review. The fault types in the taxonomy are significant and have been shown to be important fault types in the past [10, 14–20].

[‡]Complete taxonomies can be found in the work of Chemannoor [13].

DOI: 10.1002/stvr

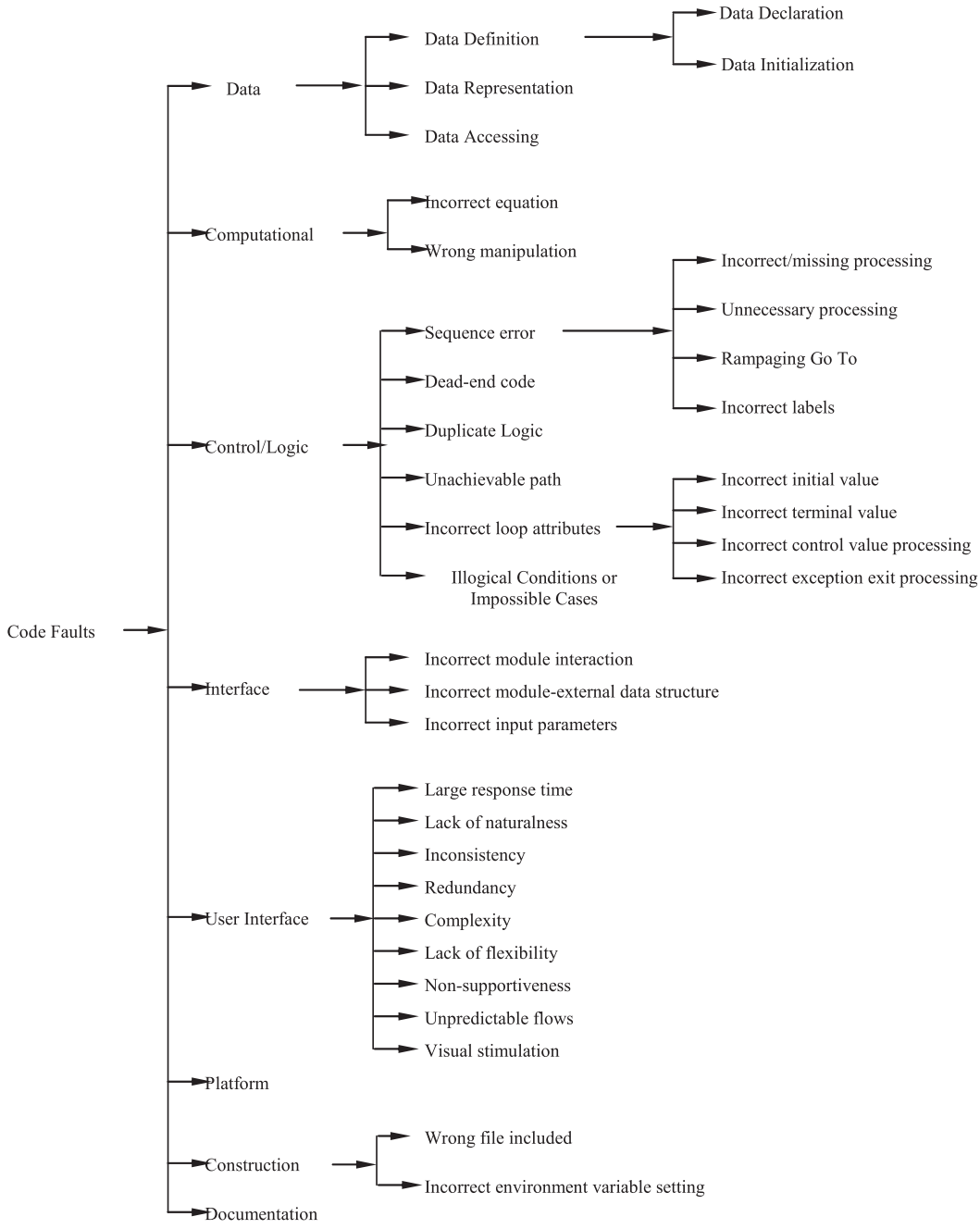

Figure 2. Generic Code Fault Taxonomy.

2.1.3. Fault links. We examined a number of hypothesized fault links for the online course management software domain [7]. We found evidence of a number of fault links, listed below. The first column provides the abbreviation of the conjectured fault link, for example, IFVC stands for Interface Faults occur in View Components. The second column describes the conjectured fault link, and the final column indicates if evidence was found to indicate that the fault link exists. Note that there was only weak support for two of the fault links (DCDF and VCUIF).

DOI: 10.1002/stvr

Conjecture	Conjectured Fault Link	Supported?
DCDF	Data components (DC) have Data faults (DF)	Weak
DFDC	Data faults occur in Data components	No
CCCLF	Controller components (CC) have C/L faults (CLF)	Yes
VCUIF	View components (VC) have User Interface faults (UIF)	Weak
UIFVC	User Interface faults occur in View components	Yes
UCCLF	Utility components (UC) have C/L faults	Yes
IFVC	Interface faults (IF) occur in View components	Yes
PFVC	Platform faults (PF) occur in View components	Yes

We also discovered these fault links that had not been hypothesized, shown below.

Discovered Fault Link
Data components have C/L faults (DCCLF)
Computational components have C/L faults (CCCLF)
Computational faults occur in Controller components (CFCC)
Data faults occur in View components (DFVC)
Utility components have C/L faults (UCCLF)

In total, 10 fault links were found to be strongly supported by the empirical study of Hayes *et al.* [7], and two were weakly supported. The full details of the determination of these fault links can be found in the work by Huffman Hayes *et al.* [7] and by Chemannoor [21]. These details have not been included here due to space considerations.

2.2. Applications

Although our method establishes component–fault relationships that can be used in different phases of the software development life cycle, we focus on applying fault links to detect defects during code inspections or walkthroughs. With information about the fault links, one need only know the type of component being reviewed in order to build a tailored, more effective checklist. For example, if one is reviewing a Utility component, one knows that these tend to have control/logic faults. Items can be added to the checklist such as: 'Are the loop exit conditions checked accurately?' or 'Are any control/logic statements missing?' Tailoring a checklist in this manner results in adding checks that may not have otherwise been performed, or that may have seemed 'generic' previously (but are now known to particularly apply based on the component type).

A typical checklist (control) is shown in Appendix B. Note that most of the items in the checklist are very general in nature. A checklist for a Controller component (experimental) is shown in Appendix A. In the online course management software domain, these components typically have many control/logic faults and data faults, with fewer computation faults, User Interface faults, and platform faults. Hence, the checklist has been so tailored (it includes a description of the typical fault links based on component type, a definition of the fault types, as well as tailored checklist questions). This approach has some similarity to searching for common programmer mistakes when performing reviews. However, the 'common mistakes' for which our method searches have been determined based on the component type and on the software domain.

Many of the items are common between the checklist in Appendixes A and B. The few items that are unique to Appendix A have been highlighted with a border (the legend in Appendix A indicates that these are unique items).

DOI: 10.1002/stvr

3. EXPERIMENTAL VALIDATION

In order to evaluate the usefulness of fault links, we conducted two experiments, one with graduate students and the other with professional software engineers. In both experiments, the subjects performed inspections of source code components using walkthrough checklists. In designing and performing the validation, we followed the guidelines for experimentation found in Wohlin *et al.* [22] augmented by guidelines on case studies for method evaluation posited by Kitchenham *et al.* [23]. The research questions investigated in both experiments were:

RQ1: Does the knowledge of fault links for a domain make the code inspection process more effective?
RQ2: Does the knowledge of fault links for a domain assist in detecting 'hard to find' defects (called 'hard faults')?

The planning and operation of the first experiment follows.

3.1. Experiment one (*students*)

The subjects, variables, hypotheses, artifacts, experimental design, and threats to validity are presented below.

3.1.1. Subjects. The subjects were upper division undergraduate Computer Science students enrolled in the capstone Senior Design course at the University of Kentucky (UK). During the semester, the students were instructed in a spectrum of code inspection techniques, ranging from informal reviews to walkthroughs, inspections, and formal technical reviews. They had sufficient knowledge and experience in order to carry out the experiment. There were two groups of code inspectors and two supervisors. Furthermore, within both groups, the members were divided into two-member teams. The groups were formed randomly by using a PERL tool developed at UK called the groupgenerator [24]. The groupgenerator program accepts as input a text file that contains the names of all the participants. Using the time as a seed, the groupgenerator randomly places the participants into different groups and, finally, writes the grouped names into an output file.

3.1.2. Variables. The independent and dependent variables are defined for the experiment.

- Independent Variable is the inspection technique used. The technique either utilized a checklist that was enhanced with the fault link information or a checklist that did not include the fault link information.
- As the time for the inspection was constant for all participants, the Dependent Variables measured are the faults detected. Specifically:
  - Number of faults found in the component.
  - Number of 'hard faults' found in the component.

3.1.3. Hypotheses. The null hypotheses for the first experiment are from the code inspector's perspective and address the effectiveness of fault link information-enhanced inspections in finding faults in a code component.

- H_{rate}: There is no difference in the rate of faults found between the inspectors utilizing the fault link information-enhanced checklists and the inspectors not utilizing such checklists.
- H_{ratehard}: There is no difference in the rate of 'hard faults' found between the inspectors utilizing the fault link information-enhanced checklists and the inspectors not utilizing such checklists.

3.1.4. Artifacts. The artifacts for the first experiment are briefly described below.

DOI: 10.1002/stvr

3.1.4.1. Code component. We chose a component from the Electronic Personal Organic Chemistry Homework (EPOCH) project for the student experiment. The system is an online homework management program and serves as a teaching aid in an organic chemistry course at UK [25]. EPOCH attempts to give students the feedback for wrong answers, thus enabling them to arrive at the correct answer. In addition to the homework program, EPOCH consists of an authoring tool to create problems and an instructor tool to assemble assignments. EPOCH is implemented using various programming languages, including Java, PERL, JSP, HTML, and PROLOG. The component (GradeStore) was chosen because it could be reasonably inspected within the available time. GradeStore is 177 lines of code (LOC), highly representative of the size of components found in large industrial applications: Zhang and Tan found that 'the average (mean) size of Java class regardless of the nature of the systems is about 114 LOC [26].' Using our component taxonomy (Section 2), the component was categorized as a data-centric component. The fault links data derived from the EPOCH project indicate that a data-centric component historically has 60% control/logic faults and 40% data faults. Using this information, analyses of EPOCH bug reports (for re-seeding previously repaired faults), and help from the developer of the EPOCH project, we seeded faults into the chosen component: 10 control/logic faults (60%) and six data faults (40%). We re-seeded as many 'naturally occurring' faults as possible (those that had previously been found and repaired), and all other faults seeded were from the collection of faults analysed as part of the fault link work [7].

3.1.4.2. Documents. The documents that were common between the control and experimental groups were: component source code (line numbered, including blank lines and comments), component type definition, fault taxonomy definitions and criteria, generic checklist, fault report sheet, and questionnaire[§]. The source code component was seeded with faults prior to the experiment, as discussed above. The fault taxonomy definitions and criteria aid the inspectors in categorizing the faults detected during inspection. The generic checklist contains a list of questions related to issues commonly present in code components. The inspectors used the fault report sheet to provide a description of the discovered faults. The questionnaire was used to obtain feedback from the participants after the inspection process, such as on the usefulness and correctness of the distributed documents.

Besides the above-mentioned documents, the experimental group received some additional documents, including a tailored checklist and the component taxonomy definition. The tailored checklist was constructed specifically for data-centric components, utilizing knowledge of the fault links between data-centric components and control/logic and data faults [7]. For example, the tailored checklist item 'Are the variables used in the IF statements correct?' assists the inspectors to ensure the correctness of the IF constructs (i.e., a control/logic statement) in the code. The component taxonomy definitions aid the participants in understanding the main purpose of the component to be inspected.

3.1.4.3. Fault classification. We categorized the seeded faults as easy, medium, or hard to detect based on our experience with the Java language (used in EPOCH) and with the code inspection process, and using previous classifications developed in prior work [27]. A code fault taxonomy developed by the U.S. Nuclear Regulatory Commission and Electric Power Research Institute tagged certain code faults as 'hard to detect' (not obvious upon inspection, subtle, etc.). Faults noted as 'hard faults' in the work by Miller *et al.* [27] have thus been classified as hard to detect in this work. The code component inspected, GradeStore, contains seven hard faults, three medium faults, and six easy faults.

3.1.5. Design. There were 26 participants (i.e., students), randomly assigned to the control group (14) and to the experimental group (12). The groups were further divided into teams, so that the control and the experimental group had seven and six two-member teams, respectively. During the

[§]All artifacts may be found in Chemannoor [13].

DOI: 10.1002/stvr

inspection process, every member of a group was restricted to communicating only with his/her team partner. No other restrictions were placed on the two-person teams, they were permitted to work together as they saw fit. Two supervisors were available (one for each group) to aid the inspectors with timely clarification of questions or issues. The members of both groups were provided with materials as described above. The format and content of all the materials followed the current industrial standards.

The collected data were descriptively and statistically analysed. A normal distribution was present in the experiment data (per visual examination of a histogram of the residuals) as well as equal variances, hence the student's t-test was used to analyse the performance of the two groups.

3.1.6. Threats to validity. In this section, we address the threats to conclusion, internal, construct, and external validity [22]. To reduce the threats to conclusion validity, we ensured that all assumptions were met for the applied statistical tests. There were minimal threats to internal validity. There was limited social threat: there was no compensation for participation; and grades were assigned based on the participation, and not on the performance in the experiment. In other words, there was nothing that the participants had to gain from the outcome of the experiment. The experiment was a mandatory part of the course; so, this minimizes the selection threat. The use of random assignment to groups is a known method for reducing the selection bias. For example, Kunz *et al.* found that not using randomized selection resulted in an overestimation of the study effects [28]. Also, the Preliminary Guidelines for Empirical Research in Software Engineering state that 'Subject/objects should be allocated to treatments in an unbiased manner or the experiment will be compromised. It is customary to allocate subjects/objects to treatments by randomization [29].' To reduce the threats to construct validity, we used the standard measures of effectiveness. We administered a questionnaire after the experiment to help validate the results. Also, we used an open-source checklist to ensure that the industry standards were applied for the code inspection.

External validity was reduced due to the use of students. However, Tichy stated that using students is acceptable as long as they have been trained adequately and as long as the data are used to test initial hypotheses prior to the experiments with professionals [30]. Also, the students were taking the capstone course during their final year, and were close to starting their careers in industry. None of the students were familiar with the domain, so we did not block on domain. We cannot generalize the results to other application domains, systems, or languages. The experiment is based on a theory about a fault link taxonomy based on existing knowledge and empirical studies. New fault links may be found, and, of course, some applications may not have certain faults. The researchers seeded faults in proportion to the fault links for specific component types. 'Naturally occurring' faults were used, where possible, to minimize any bias. For example, faults found previously in EPOCH were re-introduced. Also, faults discovered by Apache and Mozilla developers and users were seeded (from the work done by Hayes *et al.* [7]). It is not uncommon for researchers to present a new method or technique that is based on a particular idea and then seed faults into real components to evaluate the effectiveness of the method. This approach has been used in many testing papers and reading technique papers. Other code inspection researchers have seeded their modules with one fault for every 20–27 LOC [31]. Our defect density is higher than that (one fault every 11 LOC), but our density was driven by the faults that had actually been found in the module previously by the developer, also a common practice in the literature. This experiment was used as a pilot experiment for a follow-up experiment with professionals.

3.1.7. Experiment operation. Days before the actual inspection, all the students were provided with materials to help them understand the piece of code that they would be inspecting. The materials distributed were: a component description and a tutorial of programming languages (esp., Java and SQL) briefly describing the concepts necessary for understanding the component. In addition to the above two items, the students were also provided with a questionnaire, which was completed by the students before the code inspection. The questionnaire was used to screen students and to remove any students without the basic skills necessary to complete the inspection.

DOI: 10.1002/stvr

The code inspectors were informed in advance and at the beginning of the experiment about the time allotted for the inspection process. The total time allotted for the inspection was 75 minutes. Note that this is well within the industry practice where 100–200 LOC are typically reviewed in one hour [32, 33]. On the day of the experiment, the supervisors distributed the necessary documents to help the inspectors carry out the inspection process. The experiment began on time and went smoothly during the allotted time frame. At the end of the process, the supervisors promptly collected the fault report sheet and questionnaire.

3.2. *Experiment two* (*professionals*)

The subjects, variables, hypotheses, artifacts, experimental design, and threats to validity for the second experiment are presented below.

3.2.1. Subjects. The subjects were professional software engineers at numerous companies, recruited using convenience sampling. A questionnaire was completed by each participant prior to the experiment. This questionnaire helped determine the participant's experience with code inspection, with the domain, and assisted us in evaluating their understanding of the required programming language concepts. In order to ensure that any bias that existed was in favour of the control method, those most experienced with code inspections were placed in the control group.

Based on the information in the questionnaires, a profile of the subjects is shown in Table I. Column one provides the participant identifier and group (experimental group member identifiers are lettered A1 through A12 and control group member identifiers are lettered B1 through B12). Those who completed the project had, on average, 7.03 years of software engineering experience, 2.7 years of code walkthrough experience, and 2.8 years of Java experience. The control group had, on average, 8.18 years of software engineering experience, 4.2 years of code inspection experience, and 3.5 years of Java experience. The experimental group had 5.79 years of software engineering experience, 1.25 years of code inspection experience, and 2.13 years of Java experience.

3.2.2. Variables. The independent and dependent variables for the second experiment are identical to those of the first experiment.

3.2.3. Hypotheses. The hypotheses for the second experiment are from the inspector's perspective and address the effectiveness of fault link information-enhanced inspections in finding faults in a code component. They are identical to those of the first experiment.

3.2.4. Artifacts. The artifacts from the first experiment were used for the second experiment also. The artifacts differed only in that some minor changes to the formatting were made based on the suggestions collected during the pilot experiment (such as providing larger spaces for writing comments on the fault collection sheet). In addition, a second code component was introduced for review (to ensure that there was not a component effect). A description of this component was provided to the participants. The second component, HWCreateSession, is a controller component, also from EPOCH. HWCreateSession is 188 LOC, representative of the size of components found in industry [26]. Historically, these components have 45% control/logic faults, 27% data faults, 9% computation faults, 9% User Interface faults, and 9% platform faults in the online course management software domain. Again, using the previously repaired faults and faults from the fault link work [7], we seeded 13 defects: 6 control/logic, 4 data, 1 computation, 1 User Interface, and 1 platform fault. Of these, 10 were classified as hard to find [27]. Appendix D provides the details on these faults: the fault type, which are hard to find, and which participants found the faults.

3.2.5. Design. There were 29 participants, of which 24 completed the experiment. There were 12 participants assigned to the control group and 12 assigned to the experimental group, as explained above. Two supervisors were available (one for each group) to aid the participants with timely answers to the questions. During the inspection process, every participant was restricted to communicating only with his/her supervisor. The members of both groups were provided with

DOI: 10.1002/stvr

materials as described above. The participants had 75 minutes to review both components, and were not given any specific order in which to review the components. We asked them to spend half the time on each component. The instructions to the Control group are shown in Appendix C. The Experimental group had identical instructions, but these two sentences replaced the sentence beginning 'use the Checklist.doc': 'first read the second page of the file Checklist.doc; then use that information as well as the first page of Checklist.doc to guide your walkthrough of each module.' The format and content of materials followed the current industrial standards. The collected data

Table I. Profile of Professional Participants.

Participant ID-group	Company	S/W Eng. experience	Code walkthrough experience	Java experience	Status
A12—exp.	Rockwell	30 years	Moderate amount	Code reading only, self taught	Completed
A8—exp.	SAIC	2 years	None	Developed Java and had 3 courses	Completed
B11—control	Intsolutions	5 years	2–3 years	None	Completed
B9—control	Perot Systems	18 years	Significant experience in code walkthroughs	5 years	Completed
A9—exp.	SAIC	3 years	College work only	2 years	Completed
A10—exp.	SAIC	5 years	Never performed a formal code walkthrough	3 years	Completed
A6—exp.	Avnet	5 years	Maybe 0.5 year experience	4 years	Completed
B8—control	Lexmark	6 years	6 years, 4 years in courses and 2 years in work	4 years, 2 in courses and 2 in work	Completed
B6—control	Perot Systems	6 years	Work experience, projects in latter years strongly emphasize code walkthroughs	6 years	Completed
A7—exp	IBM	none	Not much	Not much	Completed
B12—control	Lexmark	4 years	Experience through work	Through courses	Completed
A5—exp	Lexington, KY corporation	6 months	None	Very little	Completed
B2—control	Rockwell	3 years	Use it unofficially on a regular basis on all projects	Several programming projects over 3 years	Completed
B5—control	SWIFT	9 years	7 years	Through courses	Completed
A4—exp	IBM	5 years	Only informally	Only course work, a while ago	Completed
B7—control	Lexington, KY corporation	5 years	One year experience on each of two jobs	3.5 years	Completed
A1—exp	Rockwell	3 years	Only a few code walkthroughs	About 2 years through courses only	Completed
Participant 25 control	Exstream	3 years	Experience from work and courses, 3 years	2 years	Dropped out
B10—control	Lexmark	12 years	8 years	8 years of work experience	Completed
A3—exp	Perot Systems	5 years	Some	One class over 6 years ago	Completed
B4—control	Lexington, KY corporation	2 years 8 months	1 year of many inspections	Work with it for 6 years (includes courses)	Completed

DOI: 10.1002/stvr

Table I. *Continued*.

Participant ID-group	Company	S/W Eng. experience	Code walkthrough experience	Java experience	Status
A2—exp	U.K.	5 years as part time programmer	1 year from college classes	4 years as a research asst	Completed
Participant 26 exp	Winchester, KY corporation	1.5 year	Some	1 year from a class	Dropped out
Participant 27 exp	Rockwell	10 years on programmable logic controllers	Once or twice per year	None	Dropped out
B3—control	Rockwell	20 years	3–4 times per month	Took courses	Completed
Participant 28 exp	Stroud Engineering Services, Inc.	18 years	Walkthroughs only for a few more complex projects	Very limited	Dropped out
B1—control	IBM	7.5 years	Experience through work and courses	5 years	Completed
Participant 29 control	Lexington, KY corporation	8 years	8 years	8 years	Dropped out
A11—exp	ACS	6 years	3 years	3 years	Completed

were descriptively and statistically analysed. A normal distribution was present in the experiment data (per visual examination of a histogram of the residuals). The Bartlett test for the homogeneity of variances had a resulting probability of $p>0.05$ indicating that the variances are homogeneous. Also, Levene's test for homogeneity of variances showed that the p statistic was not significant at the 0.05 level, hence the variances are homogeneous. With the normality and homogeneity of variances assumptions met, the t-test was used to analyse the performance of the two groups.

3.2.6. Threats to validity. In this section, we address the threats to conclusion, internal, construct, and external validity [22]. The threats were the same as for experiment one except as noted here. Concerning internal validity, there was limited social threat as there was no compensation for participation. There is a selection threat and a history threat (unavoidable differences in the subjects' experience). To reduce this, we placed the most experienced professionals in the control group to bias against the fault link method. As a result of our quasi-experimental design, it is possible that our subjects may not be representative and that there could be other confounding variables that influenced the two groups [34]. There were two subjects from the same company (A12 and B3) with 20 plus years of experience. The subject with the greatest amount of code reading and Java experience was placed on the control team to bias against the fault link method (B3). The subjects were instructed not to speak with anyone else during the experiment, but it is possible that communications might have occurred. Also, there is a possible internal threat to validity regarding the amount of time spent by the participants on the inspection. The professionals were asked to monitor their own time and to work for 75 minutes. It is possible that some participants worked for more or less than 75 minutes, though all reported that they had worked for 75 minutes. Based on the data from Jacob and Pillai [32] and a study by a Software Process Improvement Network (SPIN) [33], professionals require roughly one hour to review up to 200 LOC. With the size of the two components (subtracting 20% of the size which are content free lines), the time given for the review should have been 87 minutes versus 75 minutes. However, the control and experimental group were given the same amount of time. If anything, this biased the experiment in terms of making it less likely that the professionals (in either group) had enough time to find as many defects as possible.

We also suffered some attrition; five of the 29 participants did not complete the experiment. There did not seem to be a pattern for the attrition except that all participants who dropped out claimed that it was due to pressing work deadlines. Some dropouts work in Lexington, some work

DOI: 10.1002/stvr

in Ohio, and some in Indiana. Some dropouts were in the experimental group, whereas some were in the control group.

External validity threat was minimal as we used skilled professionals as the participants. None of the professionals were familiar with the domain, so we did not block on domain. The majority of the participants were from the Midwestern United States, but there were a few participants from Canada and the west coast of the United States. We reduced the threat that there is a component bias by using a second component from the system. There is still the possibility of a system bias, though. We cannot generalize the results to other application domains, systems, or languages. The discussion regarding the seeding of faults by the researchers in Section 3.1.6 also applies here. The defect density was one fault for every 14.5 LOC. This experiment took advantage of the lessons learned from the pilot experiment, and hence had reduced the threats to validity. In addition, experimental reliability [23] was evident—the study was repeated with professionals and had similar results.

3.2.7. Experiment operation. Days before the actual inspection, all professionals were provided with materials, via e-mail, to help them understand the code components that they would be inspecting. The same materials distributed for experiment one were distributed for this experiment, along with a second code component and its description. The participants were informed in advance and at the beginning of the experiment about the time allotted for the inspection process. The total time allotted for the inspection was 75 minutes. On the day of the experiment, the supervisors emailed the necessary documents to the participants. The participants were permitted to complete the experiment in their own work space and were on the honour system for the amount of time used. All the participants reported back the actual time used, and all reported that they had worked for 75 minutes. Each participant returned (via fax or by emailing scanned artifacts) the fault report sheet and the questionnaire to the supervisors.

3.3. Comparison of the experiments

To the extent possible, the same artifacts were used for both experiments. Minor corrections discovered during the student experiment were made to the artifacts before the professional experiment. Also, the professionals reviewed a second component. The second component was added to minimize the possibility that the effects were component-specific. The student experiment had pairs of students examining the code components versus individuals. This design was in keeping with our project work in the class (projects undertaken by pairs and groups, not individuals).

4. EXPERIMENTAL RESULTS

This section presents the results of both experiments.

4.1. Experiment one (*students*)

In this section, the analysis of experiment one is presented. The analysis addresses the effectiveness of fault detection for the GradeStore code component from EPOCH.

4.1.1. Results. The results from the code inspection are shown in Table II. The experimental group is labeled 'A' and the teams within the group are labeled 'A1,' 'A2,' 'A3,' etc. Similarly, the control group is labeled 'B.' As mentioned earlier, 16 faults were seeded into the GradeStore component. Table II presents the number of faults found by every team within a group. For example, in Group A, Team A3 found 12 of the 16 seeded faults. In Group B, Team B6 found 3 of the 16 seeded faults, etc.

The students' t-test was used for statistical analysis of the results. Our null hypothesis for the rate of faults found (H_{rate}) is that the number of faults found will not vary between the experimental and control group. Our alternative hypothesis is that a statistically significant difference in the

DOI: 10.1002/stvr

Table II. Number of Faults Found.

Groups	Experimental Group (A)						Control Group (B)						
Teams	A1	A2	A3	A4	A5	A6	B1	B2	B3	B4	B5	B6	B7
Number of faults found	10	6	12	6	9	5	9	8	1	1	8	3	1
Number of faults seeded	16												
Mean	8.00						4.43						
95% confidence interval of mean	Lower bound 5.79 Upper bound 10.2						Lower bound 1.65 Upper bound 7.2						
Std. deviation	2.76						3.74						
Std. error of mean	1.13						1.41						

Table III. Number of Hard Faults Found.

Groups	Experimental Group (A)						Control Group (B)						
Teams	A1	A2	A3	A4	A5	A6	B1	B2	B3	B4	B5	B6	B7
Number of hard faults found	6	4	8	4	5	4	5	5	0	1	5	2	0
Number of hard faults seeded	7												
Mean	5.17						2.57						
95% confidence interval of mean	Lower bound 3.9 Upper bound 6.45						Lower bound 0.81 Upper bound 4.32						
Std. deviation	1.60						2.37						
Std. error of mean	0.65						0.93						

number of faults found will exist between the experimental and control group. We will reject the null in favour of the alternative when the probability that the observed results are due to chance is 0.05 or less. The mean, confidence interval, standard deviation, and the standard error of means were calculated for both groups and are shown in the table above. The p-value was 0.04, indicating that the results obtained were statistically significant (i.e., there is a significant difference in the number of faults found between the groups). When fault links are used to assist with code inspection, the mean number of faults found is 8.0 and its 95% confidence interval is 5.79–10.2.

The results for the hypothesis on the rate of hard faults found (H_{ratehard}) are shown in Table III. The experimental group found an average of 5.17 hard faults (standard deviation of 1.6) and the control group found 2.57 (standard deviation of 2.37). The p-value for the rate of hard faults found was 0.02, indicating that the results obtained were statistically significant and we can reject the null hypothesis in favour of the alternative (i.e., there is a significant difference in the rate of hard fault detection between the groups). When fault links are used to assist with code inspection, the mean number of hard faults found is 5.17 and its 95% confidence interval is 3.9–6.45.

4.1.2. Discussion. No restrictions were placed on the teams as to how they worked together. The supervisors did observe that all the teams used a similar format: both members looked for faults and then discussed them with each other and turned in a record of joint work.

The results of the hypotheses are summarized below.

- H_{rate}: The fault link method found almost two times more faults than the generic inspection checklists. The results were statistically significant.
- H_{ratehard}: The fault link method found two times more hard faults than the generic checklists. The results were statistically significant.

4.2. Experiment two (professionals)

In this section, the analysis of experiment two is presented. The analysis addresses the effectiveness of fault detection for two code components from EPOCH.

DOI: 10.1002/stvr

Table IV. Experiment Two—Number of Faults Found—Gradestore.

Groups	Participants	Number of faults found	Number of faults seeded	Mean	95% confidence interval of mean	Std. deviation	Std. error of mean
Experimental Group (A)	A1	6	16	8.08	Lower bound 6.19 Upper bound 9.97	3.34	0.96
	A2	14					
	A3	1					
	A4	6					
	A5	7					
	A6	8					
	A7	11					
	A8	7					
	A9	11					
	A10	8					
	A11	7					
	A12	11					
Control Group (B)	B1	4		4.00	Lower bound 2.43 Upper bound 5.56	2.76	0.80
	B2	3					
	B3	8					
	B4	6					
	B5	3					
	B6	0					
	B7	4					
	B8	8					
	B9	6					
	B10	5					
	B11	1					
	B12	0					

4.2.1. Results. The results from the code inspection are shown in Table IV. For example, in Group A, Participant A7 found 11 of the 16 seeded faults in GradeStore. In Group B, Participant B3 found 8 of the 16 seeded faults, etc.

The students' t-test was used for statistical analysis of the results. Our null hypothesis for rate of faults found (H_{rate}) is that the number of faults found will not vary between the experimental and control group. Our alternative hypothesis is that a statistically significant difference in the number of faults found will exist between the experimental and control group. We will reject the null in favour of the alternative when the probability that the observed results are due to chance is 0.05 or less. The mean, confidence interval, standard deviation, and the standard error of means were calculated for both groups for the GradeStore component and are shown in Table IV. The p-value was 0.0018, indicating that the results obtained were statistically significant. When fault links are used to assist with code inspection, the mean number of faults found is 8.08 and its 95% confidence interval is 6.19–9.97.

The mean, confidence level, standard deviation, and the standard error of means were calculated for both groups for the HWCreateSession component and are shown in Table V. The p-value was 0.06, indicating that the results obtained were not quite significant.

The results for the hypothesis on the rate of hard faults found (H_{ratehard}) for component GradeStore are shown in Table VI. The experimental group found an average of 5.17 hard faults (standard deviation of 2.52) and the control group found 1.58 (standard deviation of 1.62). The p-value for the rate of hard faults found was 0.0002, indicating that the results obtained were statistically significant and we can reject the null hypothesis in favour of the alternative. When fault links are used

DOI: 10.1002/stvr

Table V. Experiment Two—Number of Faults Found—HWCreateSession.

Groups	Participants	# of faults found	# of faults seeded	Mean	95% confidence interval of mean	Std. deviation	Std. error of mean
Experimental Group (A)	A1	3	13	3.67	Lower bound 2.45 Upper bound 4.88	2.15	0.62
	A2	6					
	A3	0					
	A4	2					
	A5	5					
	A6	7					
	A7	3					
	A8	2					
	A9	5					
	A10	1					
	A11	5					
	A12	5					
Control Group (B)	B1	1		2.17	Lower bound 0.77 Upper bound 3.57	2.48	0.72
	B2	2					
	B3	5					
	B4	8					
	B5	0					
	B6	0					
	B7	2					
	B8	3					
	B9	4					
	B10	0					
	B11	1					
	B12	0					

to assist with code inspection, the mean number of hard faults found is 5.17 and its 95% confidence interval is 3.74–6.6.

The results for the hypothesis on the rate of hard faults found (H_{ratehard}) for component HWCreate-Session are shown in Table VII. The experimental group found an average of 2.33 hard faults (standard deviation of 1.5) and the control group found 1.08 (standard deviation of 1.44). The p-value for the rate of hard faults found was 0.02, indicating statistical significance and allowing us to reject the null hypothesis in favour of the alternative. When fault links are used to assist with code inspection, the mean number of hard faults found is 2.33 and its 95% confidence interval is 1.48–3.18.

4.2.2. Discussion. The results of the hypotheses are summarized below.

- H_{rate}: The fault link method was significantly more effective (found almost two times more faults for GradeStore and 1.7 times more faults for HWCreateSession) at detecting faults than generic inspection checklists. The GradeStore results were statistically significant, the HWCreateSession results were not quite significant.
- H_{ratehard}: The fault link method was significantly more effective (found two times more hard faults for HWCreateSession and three times more for GradeStore) at detecting hard faults than generic checklists. The results were statistically significant.

Let us examine the results in terms of the types of faults detected. We focus on the module that was most difficult for participants (regardless of treatments), HWCreateSession. As can be seen in Appendix D, there were 13 faults in the module: six were control/logic faults, four data faults, one computation fault, one interface fault, and one platform fault. Looking at all participants, regardless of the treatment, it can be seen that of the opportunities to find the control/logic faults

DOI: 10.1002/stvr

Table VI. Experiment Two—Number of Hard Faults Found—GradeStore.

Groups	Participants	# of faults found	# of faults seeded	Mean	95% confidence interval of mean	Std. deviation	Std. error of mean
Experimental Group (A)	A1	4	7	5.17	Lower bound 3.74 Upper bound 6.6	2.52	0.73
	A2	11					
	A3	1					
	A4	5					
	A5	5					
	A6	4					
	A7	5					
	A8	4					
	A9	7					
	A10	4					
	A11	4					
	A12	8					
Control Group (B)	B1	2		1.58	Lower bound 0.77 Upper bound 3.57	1.62	0.47
	B2	1					
	B3	5					
	B4	2					
	B5	0					
	B6	0					
	B7	1					
	B8	4					
	B9	2					
	B10	2					
	B11	0					
	B12	0					

(each participant could have found all six), 13% were found; 27% of the four data faults; 40% interface fault; 23% computation fault; and 18% platform fault.

As discussed in Section 2.2, the checklist used by the experimental participants was augmented with items related to the fault links typical for the component type. There were seven items added to the checklist (as can be seen in Appendix A). Of these seven items, two related to control/logic faults (for example, one item says 'Missing control/logic statements may cause improper functioning of the component'), six related to data, and one related to computation (note that one item related to data and computation and one related to control/logic and data). The experimental participants found 18% of the control/logic faults, 34% of data faults, 54.4% interface fault, 27.2% computation fault, and 27.2% platform fault. This can be compared with the control participants who found half as many control/logic faults as the experimental team at 9%, found 20% of the data faults, found half as many interface faults as the experimental team at 27% (compared to 54.4%), were at 18% for the computation fault, and 9% for the platform fault.

It can also be observed from Appendix D that the faults marked as 'hard' by the experimenters were indeed more rarely found than other faults. Of the 10 faults marked as hard, three could be considered 'very hard' to find as less than 5% were found (three faults that could each be found by 22 participants). Five of the 10 could be considered 'hard' to find as less than 23% were found. This means that only 5–23% of the eight faults marked as hard were found. Only 27% of the two remaining faults marked as hard were found. One fault was not marked as hard, yet only 18% of the participants found it (a data fault). Having examined the faults and the checklists, let us now examine the participants in detail.

We noted that subjects A12 and B3 were from the same company. They both found a significant portion of the seeded faults (A12 found 11 faults and B3 found 8 faults in Gradestore; A12 found 5 faults and B3 found 5 faults in HWCreateSession). It is possible that communications

DOI: 10.1002/stvr

Table VII. Experiment Two—Number of Hard Faults Found—HWCreateSession.

Groups	Participants	# of faults found	# of faults seeded	Mean	95% confidence interval of mean	Std. deviation	Std. error of mean
Experimental Group (A)	A1	2	10	2.33	Lower bound 1.48 Upper bound 3.18	1.50	0.43
	A2	4					
	A3	0					
	A4	1					
	A5	3					
	A6	4					
	A7	1					
	A8	1					
	A9	4					
	A10	1					
	A11	3					
	A12	4					
Control Group (B)	B1	0		1.08	Lower bound 0.26 Upper bound 1.89	1.44	0.42
	B2	1					
	B3	3					
	B4	3					
	B5	0					
	B6	0					
	B7	1					
	B8	1					
	B9	4					
	B10	0					
	B11	0					
	B12	0					

might have occurred or that the subjects found so many faults due to similar skills and background.

There were a few of the participants (A3, A1, A10) who found far fewer faults in both components than did their counterparts. An examination of the backgrounds of these participants indicates that the software engineering experience and the experience with Java are similar to that of the overall experimental group. The amount of experience with code walkthroughs, however, is much lower for this subset of the experimental group and may explain their lower than average fault detection.

We examined the post-experiment surveys returned by the subjects (we received five from experimental and two from control subjects). Two of the experimental subjects said that it was a good session, that the session and the documents could not be improved, and that they found the session to be a useful experience. One experimental subject commented that they got tired near the end of the time limit and were losing concentration. One experimental subject found the session fun and said he/she liked the challenge of a time limit in which to review both modules. He/she found that the documents 'sacrificed clarity for brevity' but found the session useful and had no suggestions on how to improve it. Another experimental subject agreed that the document for HWCreateSession could have been clearer, but also said the session was 'pretty interesting' and useful. One experimental subject noted that GradeStore was very condensed and it would have been nice to have some spacing added for readability, and also commented that they were very out of practice on code reviews. One control subject said that they found Gradestore frustrating and wondered if it would compile, but had no suggestions on how to improve the session or the documents. One control subject stated that the session and documents could not be improved, but noted that the two components were very dissimilar (one on a database query and one on setting up a customer-defined data structure). The control subject also found the session to be a useful experience.

DOI: 10.1002/stvr

5. RELATED WORK

We present an abbreviated survey of the related work in four categories[¶]: fault surveys (involving research that only uses defects, faults, or errors), component/module surveys (involving research that only uses software components), both component and fault surveys (involving research that use both), and code inspection methods.

5.1. Fault surveys

Faults have traditionally been characterized syntactically [35–37], including: the position in the program where faults occur [38], the software development phase that generated the faults [39, 40], the testing technique that detected the faults [2], and the type of statement in which the faults occur [41]. As part of a National Aeronautics and Space Administration (NASA) funded project, Hayes [1] presents a requirements fault taxonomy and a methodology for requirement FBA. The FBA technique provides guidelines to detect and/or prevent different classes of requirement faults. Requirement faults from six NASA systems were examined to build a requirement fault taxonomy specific to NASA. Processes to tailor the taxonomy to a specific class of projects (or domain) or to a specific project were also presented and applied. The work differs from the current work in that it concentrates on requirement faults and does not examine the relationships between the component and fault types.

IBM's Orthogonal Defect Classification [42] classifies faults based on programmer mental mistakes. Defect type distribution is used to measure the progress of a product and demonstrates the use of the defect trigger distribution to evaluate the effectiveness and completeness of the verification process. However, ODC uses a high-level classification of the software defects and does not address fault links.

Shooman and Bolsky [43] perform an experiment to collect basic information about software errors. They focus on determining the nature and frequency of errors, whereas our work addresses the component and fault types. They set out to perform a pilot study to investigate the error density of modules (error density of a module is the percentage of the module's total number of LOC that contain errors) and to develop data on how to use the available debugging resources. Their experimental results show that a large percentage of the errors were found by hand processing (without the aid of computer testing techniques).

Lutz [44] analyses the root causes of requirement faults that lead to safety-related software errors in a safety critical, embedded system by adopting the classification scheme proposed by Nakajo and Kumis [45]. This classification scheme moves backwards in time from the apparent software error to an analysis of the root cause. Lutz presents only a high-level classification for both program and requirement faults, focusing only on embedded systems.

Ostrand and coworkers [46] quantitatively analyse the faults and failures of a major commercial system and found observations identical to those made previously by Fenton and Ohlsson [47]. Fenton and Ohlsson provide evidence to substantiate that software systems developed under the same environment result in similar fault densities, when tested in similar testing phases. Hamdioui *et al.* [48], in an effort to aid test engineers in dealing with new dynamic fault classes, mathematically analyse the dynamic fault classes based on primitive fault concepts. Their study emphasizes the dynamic memory-related faults whereas our work deals with the static run-time faults.

Dehlinger and Lutz [49] introduce 'product line software fault tree analysis' to improve the software quality, where a product line is defined as a set of systems that are developed from a common set of core requirements and share a suite of common traits. Software Fault Tree Analysis (SFTA) [49] is a technique for investigating causes that contribute to the potential hazards in safety-critical applications. The SFTA technique has been adapted to product lines in order to

[¶]A comprehensive survey can be found in [13].

DOI: 10.1002/stvr

derive reusable analysis assets for future systems within the existing product line. However, the application of the method to an implemented software system is not evident, as it is for our work.

Offutt and Alexander [50] study the characteristics of the program faults that occur in object-oriented software to improve the testing techniques. They posit that a full understanding of these characteristics is crucial to several research areas. The paper presents a model of the appearance and realization of object-oriented faults and defines the specific categories of inheritance and polymorphic faults. As opposed to our work, their fault categories relate only to object-oriented software and concentrate only on inheritance and polymorphic faults. Offutt and Hayes [2], in order to analyse the characteristics of program faults, propose a semantic model for fault categorization based on the syntactic and semantic size of a fault. They believe that viewing faults through this characterization can solve many problems faced by fault-based testing techniques. This work may assist with future improvements to fault link checklists.

Xie and Engler [51] illustrate the importance and usefulness of redundant errors. They believe that redundant errors are just as serious as other errors (termed hard errors). In order to experimentally verify this idea, they develop and apply five redundant checkers on large open-source projects. They show that redundant errors can assist in finding mistakes and omissions in specifications. Although the study discovers new fault types, it is not as exhaustive and generic as the one presented here.

5.2. Component/module surveys

Khoshgoftaar and Allen [11, 52, 53] classify a software module as either fault-prone or non-fault-prone. They demonstrate how module-order models can be used for classification [52] and compare them with statistical classification models [11]. Khoshgoshtaar and Allen [53] attempt to control the overfitting problem that causes classification models [11] to miscalculate the fault-proneness of a component. As compared with our work, the authors do not classify faults and the module classification that they present is not as detailed as the work presented here. Ohlsson *et al.* [54] model fault-proneness statistically over a series of four releases. The model includes change measures at various levels of analysis, such as the number of defect fix reports attributed to a module, an interaction measure of defect repairs that involve more than one module, etc. Their analysis of case study data shows that fault-prone modules exhibit a higher system impact (total number of changes to .c and .h files in a release per module) across releases. In addition to examining the number of faults per module, our work examines the types of faults attributed to a module that has also been categorized by type.

In a similar vein, Zhao and Hayes describe a method for categorizing components as easy to change or not easy to change [55]. Bieman *et al.* [56] identify the change-proneness of C++ code based on the intentional use or lack of use of patterns. They demonstrate that some patterns are more change-prone based on different categories of maintenance (i.e., corrective versus enhancement). However, they make no attempt to classify faults. Bieman *et al.* [57] also present findings that suggest a strong relationship between the class size and the number of changes, with larger classes changing more frequently. The investigators did not identify the type of change or fault in these studies, however.

Damiani *et al.* [58] present a hierarchy-aware classification schema for object-oriented code. The behavioural characteristics categorize the components, such as service-provided, algorithm-employed, and data-needed. These characteristics can be constructed from the application models or extracted semi-automatically from the class interfaces. This classification method is for object-oriented software projects, whereas the research we present here is generic and can be applied to both procedural and object-oriented software projects.

5.3. Component and fault surveys

Basili and Perricone [10] analyse the interaction between the frequency and distribution of errors during software development, the maintenance of the developed software, and a number of environmental factors such as the complexity of software. The paper defines a module as a named

DOI: 10.1002/stvr

subfunction, subroutine, or the main program. The authors classify a module either as modified or new (developed specifically for the software project under analysis). The module classification is high level as compared with the one presented here. The authors also classify software errors into five categories, some of which we use in our work. The research of Ohlsson *et al.* [54], described above and compared with our work above, is the motivation for the construction of a fault architecture [47], determining fault coupling and cohesion measures at the module and subsystem levels, within a release and across releases.

Ostrand and Weyuker [46], with the aim of aiding organizations to determine the optimal use of their testing resources, identify various file characteristics to serve as the predictors of fault-proneness. Based on a series of 13 releases of a large evolving industrial software system, they observe that: (i) faults are concentrated in a small number of files and in a small percentage of the code, (ii) decrease of testing efforts for previously high-fault files is a mistake, and (iii) 'all late-pre-release faults always appeared in under 5% of the files [46].' The researchers do not classify modules and faults, however.

5.4. Code inspection

Fagan introduced the idea of inspections (now called Fagan Inspections) [52]. He presented well-defined steps for the inspection as well as roles for the participants [59]. He updated the work, finding that the inspections had definitely improved the defect detection over a 10-year period, with organizations such as AETNA and IBM reporting that the inspections were helping them to discover 82–93% of all the defects detected [60]. Fagan noted the importance of exit criteria repeatedly, listing it as one of the three critical requirements for performing inspections and also noting that exit criteria need to be objective and repeatable [60]. In addition, he provided rates that can be used for planning code reviews, such as that 500 non-commentary source statements per hour can be examined for an overview [60]. These rates have been updated by Laitenberger and DeBaud [61], they found that two hours would be a reasonable time period for inspecting 200–300 lines of C code. These guidelines helped us allocate the time for the inspection.

Ackerman *et al.* present their views on inspections based on their experience [62]. They note that two steps of the inspection process are often overlooked: overview and preparation. Although the authors do not discuss the taxonomies of defects, they do provide a sample checklist for inspecting requirements specifications, and they provide some guidance on the defects for which to search. They feel that having a definition of the types of defects to be found focuses the search, but they do not provide such definitions [62]. Our work specifically addresses this notion by providing fault links to guide the search for commonly occurring fault types based on the component type.

Aurum *et al*. [63] looked at the state-of-the-art in code inspection techniques 25 years after Fagan's seminal paper [59] and found that there are differences in the structural models of inspection used now (such as the number of inspectors, inspection without a meeting, etc.) as well as differences in the support of the model (such as reading techniques, automated support for the inspection meeting). Our work provides a technique to support code inspection. They also point out and summarize the empirical validation of many of these suggested improvements to software inspection [63]. Some of the improvements from their survey are discussed below.

A number of researchers have looked at the effect of the number of inspectors on defect detection effectiveness. Porter *et al.* [64] undertook a controlled study and found that individual inspectors have much lower performance than two-person teams or four-person teams. They also found that there is not a significant difference between the effectiveness of two-person teams and four-person teams. Shull *et al*. examined the effectiveness of perspective-based reading (PBR) [65]. They found, through many studies with professionals and students, that PBR helps individuals and teams of reviewers to find more defects in artifacts when dealing with unfamiliar domains. Vermunt *et al*. [66] applied Group Support Systems (GSS) to the inspection problem. They had students and professionals review a four-page document either with or without GSS aid, they found that only 5% of the defects were not found in

DOI: 10.1002/stvr

the preparation phase. There were no differences in the performance between the GSS and non-GSS subjects, but the non-GSS subjects evaluated their session much higher than GSS subjects [66]. Our work differs from the above in that we apply a new code inspection technique.

Brykczynski [67] surveyed inspection checklists and found many checklists related to code. None of these checklists were based on component type or fault links, however. In fact, the discussion of project-specific checklists did not mention the idea of using historical fault occurrence information to tailor a checklist, as is done in our work.

6. CONCLUSIONS AND FUTURE WORK

The analysis lends support that fault links are useful in the process of code inspection or walk-through. We found that participants using our fault-link enhanced checklist found 1.7–2 times more faults and 2–3 times more hard faults in the same amount of time as participants who used a generic checklist. For experiment one (students), we were able to reject the null hypothesis that no difference existed between the group using the fault link method and the control group in all cases: H_{rate} was rejected in favour of the alternative, and the fault link group outperformed the control group in terms of the number of faults detected; and H_{hardrate} was rejected in favour of the alternative, the experimental group outperformed the control group in terms of the number of hard faults discovered. For experiment two (professionals), we were able to reject the null hypothesis that no difference existed between the group using the fault link method and the control group in all cases except one: H_{rate} was rejected in favour of the alternative for GradeStore, and the fault link group outperformed the control group in terms of the number of faults detected; H_{hardrate} was rejected in favour of the alternative for GradeStore and HWCreateSession, the experimental group outperformed the control group in terms of the number of hard faults discovered; however, H_{rate} was not rejected in favour of the alternative for HWCreateSession as the p-value was 0.06 and thus not quite significant.

In terms of the research questions, we found support for a 'yes' answer to RQ1, *Does the knowledge of fault links for a domain make the code inspection process more effective?*, in the student experiment and for the GradeStore component of the professional experiment. We found support for a 'yes' answer to RQ2, *Does the knowledge of fault links for a domain assist in detecting 'hard to find' defects* (*hard faults*)*?*, in both experiments and with both components. We cannot generalize the results to other application domains, systems, or languages, however. We found support for our fault link method in the online course management domain for the Java language.

The work on the fault taxonomy and the component taxonomy is ongoing with the hope that others will assist in their validation and improvement. We plan to examine languages such as Lisp that provide control abstraction. The taxonomies are not completely orthogonal. Evaluating this aspect of the taxonomy is an area of the future work.

Our experiments assessed the usefulness of fault links to aid software engineers or code inspectors. We need to conduct more experiments to verify that fault links will assist other stakeholders such as requirement engineers. A larger-scale study with a variety of industry projects (including embedded and heterogeneous applications) across diverse domains and languages is needed before any broad conclusions can be reached. A specific question of interest is whether fault links are useful for safety critical development efforts. Also, we plan to examine the usefulness of each individual fault link.

The main direction for the future work is the expansion of the fault link idea into a study of fault chains. Faults rarely occur in isolation. They may be related longitudinally within a release (e.g., a design fault leads to a code fault) or across releases (e.g., incomplete fault repair). Several types of fault chains have been identified, and more will inevitably be discovered as research progresses. The ultimate goal of this work is to identify the V&V or quality assurance techniques to take advantage of the knowledge of fault chains to prevent or detect faults as early as possible, as part of FBA, to assist with developing reliable software systems.

DOI: 10.1002/stvr

APPENDIX A–CHECKLIST USED IN THE EXPERIMENT - EXPERIMENTAL

Code Inspection Checklist

Inspector Name:_____

Component Name:_____

The piece of code or component given to you is classified as a data-centric component. The results obtained from our research indicate that a data-centric component historically has 60% control/logic and 40% data faults (definitions next page). Thus, when performing a code walkthrough on such a component, one should make sure that the following issues have been addressed.

__IF statements:
- __Are attributes of the input parameters compared to correct values?
- __ Are variables used in the IF statements correct?
- __ Are correct values compared in the IF statements?
- __ Are strings compared using the equals () function (strings have to use equals ())?

__ Loop attributes:
- __ Correct initial values for the loop control variables
- __ Correct terminal values for the loop control variables
- __ Correct processing of the loop control variables
- __ Loops with exits (i.e., no infinite loops)
- __ Are the loop exit conditions checked accurately?

__ Missing control/logic statements may cause improper functioning of the component

__ Variables declared and initialized to correct values
__ DB accessing statements refer to correct fields in the table

__ Array attributes:
- __Correct array declarations
- __Array subscript or index always begins from 0 (zero) in Java
- __ Initial value of the array reflects its default value
- __ Sufficient array space to store values for varying inputs

__ Meaningful component name
__ Source file introductory comments are properly formatted and completely filled out
__ Descriptions for header and source file properly describe module functions
__ Method separators and headers exist for every method
__ Line counts are within acceptable limits (try to keep each module less than 500LOC)
__ All variables are described in appropriate locations
__ Variable descriptions are accurate and in sufficient detail
__ All declared local variables are used in the code
__ Variable names are meaningful and unambiguous
__ All variables are initialized before use
__ Methods/Functions only perform one task
__ Methods/Functions are properly commented for easy understanding
__ External specifications of the method are easy to understand
__ Spaces, parentheses, and continuation lines are appropriately used to make the code readable
__ Error handling (try – catch blocks are employed and used correctly)
__ Values computed and stored in variables are correct

Inspector signature: _____Date: _____

DOI: 10.1002/stvr

Results from our Research

Component #1 (GradeStore.java): This component is classified as a data-centric component. The results obtained from our research indicate that a data-centric component, historically, has this fault distribution: 60% control/logic and 40% data faults (definitions given below).

Component #2 (HWCreateSession.java): This component is classified as a controller component. The results obtained from our research indicate that a controller component, historically, has this fault distribution: 45% control/logic, 27% data, and 9% computational, UI, and Platform faults. We also found that all the computational faults occurred in controller modules.

Fault Definitions:

Data:

Data, which form basic building blocks of any software, are stored in data structures such as constants, variables, arrays, etc. within the software. These data structures go through several stages before they are actually put into use. In most languages, the data structures are declared, defined, and represented before being used. Faults occurring due to errors in any of these stages fall under this category. However, these faults are not due to incorrect computation.

Control / Logic:

The control and logic statements form the backbone of any software application being developed. These statements are decision-making statements that cause the software to take a particular path or to remain in a specific state. Errors occurring in these statements can occasionally result in very expensive faults that can compromise software performance.

Computational:

Computation is one of the several ways in which data is processed to obtain the required results either to conduct further computation or to provide necessary information to the user or to other modules.

UI:

The user interface is the main point of contact between the user and the system. The user interacts with the system in order to carry out a specific and important task. Depending on the user's experience with the interface, the system may succeed or fail in helping the user to carry out the task. Errors during the user interface design may lead to faults that may frustrate the user.

Platform:

An example of this fault type: the software product works correctly under one operating environment but does not in another. The fault type is not due to varying environment settings, but due to lack of options to set the environment. For example, the software works correctly with Internet Explorer 5.0 but does not work well with Internet Explorer 4.0. The problem is that there are no options in Internet Explorer 4.0 to get the software to work properly.

Legend:

Item is unique to this checklist (does not exist in the other checklist)

Item appears on both checklists

DOI: 10.1002/stvr

APPENDIX B–CHECKLIST USED IN THE EXPERIMENT - CONTROL

Code Inspection Checklist

Inspector Name:_____

Component Name:_____

__ Correct variable and array declarations
__ Meaningful component name
__ Source file introductory comments are properly formatted and completely filled out
__ Descriptions for header and source file properly describe module functions
__ Method separators and headers exist for every method
__ Line counts are within acceptable limits (try to keep each module less than 500LOC)
__ All variables are described in appropriate locations
__ Variable descriptions are accurate and in sufficient detail
__ All declared local variables are used in the code
__ Variable names are meaningful and unambiguous
__ All variables are initialized before use
__ Methods/Functions only perform one task
__ Methods/Functions are properly commented for easy understanding
__ External specifications of the method are easy to understand
__ Spaces, parentheses, and continuation lines are appropriately used to make the code readable
__ Correct indentation is used
__ Error handling (try – catch blocks are employed and used correctly)

Inspector signature: _____Date: _____

APPENDIX C–INSTRUCTIONS SENT TO SUBJECTS - CONTROL

Thanks for agreeing to participate in our experiment. Here are the instructions:

Please spend 75 minutes on the code review - please ensure that you spend half the time on each module (even if you do not get finished).
Please do not speak with anyone else about this experiment to ensure no bias is introduced.

Please proceed in this manner:

- use the Checklist.doc list to guide your walkthrough of each module
- as you find problems with the code, please note it on the fault_report_sheet.doc form: entering the module name and line number in the first column, a description in the second column, and your judgment of how hard or easy it was to find in the final column.

REPEAT the above process for the second module

- then complete the Survey_Sheet.doc form

APPENDIX D–HWCREATESESSION[‖]

HW Create Session	Fault type:	Control/ Logic (C/L)	C/L	Data	C/L	Data	Data	C/L	Platform	C/L	C/L	Inter-face	Data	Computation
	Hard?:	Hard	Hard		Hard	Hard	Hard	Hard	Hard	Hard	Hard			Hard
	Fault number:	1	2	3	4	5	6	7	8	9	10	11	12	13

[‖]Detailed data for reachable/agreeable participants.

DOI: 10.1002/stvr

HW Create Session	Fault type:	Control/ Logic (C/L)	C/L	Data	C/L	Data	Data	C/L	Platform	C/L	C/L	Inter-face	Data	Computation
Experimental Participants	A1	x									x		x	
	A2				x					x	x	x	x	x
	A3													
	A4											x	x	
	A5			x		x	x						x	x
	A6			x	x				x	x		x	x	x
	A7										x	x	x	
	A8								x			x		
	A9				x		x		x		x		x	
	A10									x				
	A12					x	x			x		x	x	
Control Participants	B1											x		
	B2			x		x								
	B3				x	x			x					
	B4			x		x					x	x	x	x
	B5													
	B6													
	B8											x	x	x
	B9				x	x	x	x						
	B10													
	B11				x					x				
	B12													

ACKNOWLEDGEMENTS

We thank Stephanie Ferguson, Pete Cerna, Kenny Todd, Mike Norris, Bill Gerstenmaier, Bill Panter, Marcus Fisher, and Lisa Montgomery. We thank Prashant Ramachandran and Dr Raphael Finkel. We thank the participants in the two experiments. We thank Arnold Stromberg and Olga Dekhtyar for their expert advice on the statistical analyses.

This work was partially funded by NASA Johnson Space Center and the International Space Station program under grant NNG04GA38G.

REFERENCES

1. Huffman Hayes J. Building a requirement fault taxonomy: Experiences from a NASA verification and validation research project. *IEEE International Symposium on Software Reliability Engineering* (*ISSRE*) *2003*, Denver, CO, 2003.
2. Offutt J, Huffman Hayes J. A semantic model of program faults. *International Symposium on Software Testing and Analysis* (*ISSTA 96*), San Diego, CA, January 1996.
3. Huffman Hayes J, Input Validation Testing: A System Level, Early Lifecycle Technique. *Thesis*, George Mason University, Fairfax, VA, January 1999. Advisor: A. Jefferson Offutt.
4. Offutt J. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology* 1992; **1**(1): 3–18.
5. Huffman Hayes J, Dekhtyar A, Osbourne J. Improving requirements tracing via information retrieval. *Proceedings of the International Conference on Requirements Engineering*, Monterey, CA, September 2003.
6. Huffman Hayes J, Dekhtyar A, Sundaram S. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering* 2006; **32**(1):4–19.
7. Huffman Hayes J, Chemannoor I, Surisetty V, Andrews A. Fault links: Exploring the relationship between module and fault types. *Proceedings of the European Dependable Computing Conference* (*EDCC*), Budapest, Hungary, April 2005.
8. Rombach HD, Basili V, Selby R. *Experimental Software Engineering Issues*: *Critical Assessment and Future Directions* (*Lecture Notes in Computer Science*). Springer: Berlin, 1993.
9. Munch J, Rombach HD, Rus I. Creating an advanced software engineering laboratory by combining empirical studies with process simulation. *Proceedings of the International Workshop on Software Process Simulation and Modeling* (*ProSim 2003*), Portland, OR, U.S.A., 3–4 May 2003.

DOI: 10.1002/stvr

10. Basili VR, Perricone BT. Software errors and complexity: An empirical investigation. *Communications of the ACM* 1984; **27**(1): 42–51.
11. Khoshgoftaar TM, Allen EB. Classification of fault-prone software modules: Prior probabilities, costs, and model evaluation. *Empirical Software Engineering* 1998; **3**:275–298.
12. Merriam-Webster's Online Dictionary. Available at: http://www.merriam-webster.com/dictionary/taxonomy [7 December 2009].
13. Chemannoor IR. *Fault Links* [*Electronic Resource*]: *Identifying Module and Fault Types and their Relationship*, University of Kentucky Libraries, Lexington, KY, 2004. Available at: http://lib.uky.edu/ETD/ukycosc2004t00211/MasterDegreeThesisReportIniesRCM.pdf [7 December 2009].
14. Duncan IMM, Robson DJ. An exploratory study of common coding faults in C programs. *A Technical Report*, Centre for Software Maintenance, University of Durham, England, May 1991.
15. Endres A. An analysis of errors and their causes in system programs. *Proceedings of the 1975 International Conference on Reliable Software in SIGPLAN Notices*, vol. 10(6), 1975; 327–336.
16. Gram C. A software engineering view of user interface design. Engineering for human–computer interaction. *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human–Computer Interaction*, Yellowstone Park, U.S.A., August 1995. Chapman & Hall: London, 1996; 293–304.
17. Macaulay L. *Human–Computer Interaction for Software Designers*. International Thomson Computer Press: London, 1995.
18. Mayhew DJ. *Principles and Guidelines in Software User Interface Design*. Prentice-Hall: Englewood Cliffs, NJ, 1992.
19. Shneiderman B. *Designing the User Interface*: *Strategies for Effective Human–Computer Interaction*. Addison-Wesley: Reading, MA, 1992.
20. Sullivan M, Chillarege R. Software defects and their impact on system availability—A study of field failures in operating systems. *Digest 21st International Symposium on Fault-tolerant Computing*, Montreal, Canada, June 1991.
21. Chemannoor I. Fault links: Identifying module and fault types and their relationship. *Master's Thesis Report*, University of Kentucky, September 2004.
22. Wohlin C, Runeson P, Höst M, Ohlsson MC. Regnell B, Wesslén A. *Experimentation in Software Engineering*: *An Introduction*. Kluwer Academic Publishers: Norwell, MA, 2000.
23. Kitchenham BA, Pickard L, Pfleeger SL. Case studies for method and tool evaluation. *IEEE Software* 1995; **12**(4):52–62.
24. Software Verification and Validation Laboratory, University of Kentucky. Available at: http://selab.netlab.uky.edu/ [7 December 2009].
25. Project #1: Electronic personal organic chemistry homework (EPOCH). Available at: http://www.prenhall.com/aceorganic/ [7 December 2009].
26. Hongyu Z, Hee Beng Kuan T. An empirical study of class sizes for large Java systems. *Fourteenth Asia–Pacific Software Engineering Conference* (*APSEC*), Nagoya, Japan, 4–7 December 2007; 230–237.
27. Miller L, Mirsky S, Huffman Hayes J. Guidelines for the Verification and Validation of Expert System Software and Conventional Software. *NUREG/CR-6316*, U.S. Nuclear Regulatory Commission and Electric Power Research Institute, March 1995.
28. Kunz R, Vist GE, Oxman AD. *Randomisation to Protect Against Selection Bias in Healthcare Trials*. Available at: http://www.cochrane.org/reviews/en/mr000012.html [7 December 2009].
29. Kitchenham BA, Pfleeger SL, Pickard LM, Jones PW, Hoaglin DC, Emam KE, Rosenberg J. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 2002; **28**(8):721–734.
30. Tichy WF. Hints for reviewing empirical work in software engineering. *Empirical Software Engineering* 2000; **5**(4):309–312.
31. Laitenberger O. Studying the effects of code inspection and structural testing on software quality. *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, *1998*, Paderborn, Germany, 4–7 November 1998; 237–246.
32. Jacob A, Pillai SK. Statistical process control to improve coding and code review. *IEEE Software* 2003; **20**(3):50–55.
33. Egypt—SPIN Newsletter, Issue 2, April–June 2003. Available at: http://www.secc.org.eg/SPIN%20Newsletter/EGYPT_SPIN_Newsletter%20(Issue2).pdf [7 December 2009].
34. Quasi-experimental design. Available at: http://www.psynt.iupui.edu/Users/cgoodlet/B311/Quasi.htm [7 December 2009].
35. Beizer B. *Software Testing Techniques* (2nd edn). Van Nostrand Reinhold, Inc: New York, NY, 1990. ISBN: 0-442-20672-0.
36. Marick B. A survey of software fault surveys. *A Technical Report UIUCDCS-R-90-1651*, University of Illinois, 1990; 2–23.
37. IEEE Standard Classification for Software Anomalies. *IEEE Std 1044.1-1995*, 12 December 1995.
38. Huffman Hayes J, Mohamed N, Gao T. The observe-mine-adopt model: An agile way to enhance software maintainability. *Journal of Software Maintenance and Evolution*: *Research and Practice* 2003; **15**(5):297–323.
39. Miller LA, Groundwater EH, Hayes JH, Mirsky SM. Guidelines for the verification and validation of expert system software and conventional software. *SAIC*, vol. 2, 1995; 100.

DOI: 10.1002/stvr

40. Lanubile F, Shull F, Basili VR. Experimenting with error abstraction in requirements documents. *Proceedings of the 5th International. Symposium on Software Metrics*, Bethesda, MD, 1998.
41. Freimut B, Developing and Using Defect Classification Schemes. *IESE-Report No. 072.01/E, Version 1.0*, Fraunhofer IESE, September 2001.
42. IBM Research. *Details of ODC v5.11*, Center for Software Engineering. Available at: http://www.research.ibm.com/softeng/ODC/DETODC.HTM [7 December 2009].
43. Shooman ML, Bolsky MI. Types, distribution, and test and correction times for programming errors. *Proceedings of the International Conference on Reliable Software*, Los Angles, CA, 21–23 April 1975; 347–357.
44. Lutz RR. Analyzing software requirements errors in safety-critical, embedded systems. *The Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, CA, 4–6 January 1993; 4–6.
45. Nakajo T, Kumis H. A case history analysis of software error causes–effect relationships. *IEEE Transactions on Software Engineering* 1991; **17**(8):830–838.
46. Ostrand T, Weyuker EJ. The distribution of faults in a large industrial software system. *Proceedings of ISSTA 2002 and ACM SIGSOFT*, Rome, Italy, vol. 27(4), 2002; 55–64.
47. Fenton NE, Ohlsson N. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering* 2000; **26**(8):797–814.
48. Hamdioui S, Gaydadjiev GN, van de Goor AdJ. A fault primitive based analysis of dynamic memory faults. *IEEE 14th Annual Workshop On Circuits*, *Systems and Signal Processing*, Veldhoven, The Netherlands, 2003; 84–89.
49. Dehlinger J, Lutz RR. Software fault tree analysis for product lines. *Eighth IEEE International Symposium on High Assurance Systems Engineering* (*HASE'04*), Tampa, FL, 25–26 March 2004.
50. Offutt J, Alexander R. A fault model for subtype inheritance and polymorphism. *IEEE International Symposium on Software Reliability Engineering* (*ISSRE '01*), Hong Kong, People's Republic of China, November 2001; 84–95.
51. Xie Y, Engler D. Using redundancies to find errors. *IEEE Transactions on Software Engineering* 2003; **29**(10): 915–928.
52. Khoshgoftaar TM, Allen EB. A comparative study of ordering and classification of fault-prone software modules. *Empirical Software Engineering* 1999; **4**:159–186.
53. Khoshgoshtaar TM, Allen EB. Controlling overfitting in classification-tree models of software quality. *Empirical Software Engineering* 2001; **6**:59–79.
54. Ohlsson M, Andrews A, Wohlin C. Modelling fault-proneness statistically over a sequence of releases: A case study. *Journal of Software Maintenance and Evolution*: *Research and Practice* 2001; **13**:167–199.
55. Huffman Hayes J, Patel SC, Zhao L. A metrics-based software maintenance effort model. *Conference on Software Maintenance and Reuse 2004*: *CSMR 05/04*, Tampere, Finland, 2004; 254–260.
56. Bieman J, Andrews A, Yang H. Analysis of change-proneness in software using patterns: a case study. *Seventh European Conference on Software Maintenance and Reengineering*, Benevento, Italy, March 2003.
57. Bieman J, Straw G, Wang H, Mungar PW, Alexander RT. Design patterns and change proneness: An examination of five evolving systems. *IEEE METRICS*, 2003; 40–49.
58. Damiani E, Fugini MG, Bellettini C. A hierarchy-aware approach to faceted classification of object-oriented components. *ACM Transactions on Software Engineering and Methodology* 1999; **8**(3):215–262.
59. Fagan ME. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 1976; **15**(3):182–211.
60. Fagan ME. Advances in software inspections. *IEEE Transactions on Software Engineering* 1986; **SE-12**(7): 744–751.
61. Laitenberger O, DeBaud JM. Perspective-based reading of code documents at Robert Bosch GmbH. *Information and Software Technology* 1997; **39**(11):781–791.
62. Ackerman AF, Buchwald LS, Lewski FH. Software inspections: An effective verification process. *IEEE Software* 1989; **6**(3):31–36.
63. Aurum A, Petersson H, Wohlin C. State-of-the-art: Software inspections after 25 years. *Software Testing*, *Verification and Reliability* 2002; **12**(3):133–154.
64. Porter AA, Siy HP, Toman CA, Votta LG. An experiment to assess the cost-benefit of code inspections in large scale software development. *IEEE Transactions on Software Engineering* 1997; **23**(6):329–346.
65. Shull F, Rus I, Basili V. How perspective-based reading can improve requirements inspections. *IEEE Computer* 2000; **33**(7):73–79.
66. Vermunt A, Smits M, Van der Pijl G. Using GSS to support error detection in software specifications. *Proceedings of the 31st Annual Hawaii International Conference on System Sciences*. IEEE Computer Society Press: Los Alamitos, CA, 1998; 566–574.
67. Brykczynski B. A survey of software inspection checklists. *ACM SIGSOFT Software Engineering Notes* 1999; **24**(1):82–89.

DOI: 10.1002/stvr